

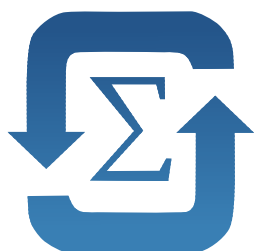
Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»



Кафедра прикладной математики

Лабораторная работа №3, 4
по дисциплине «Уравнения математической физики»

Решение гармонических задач Решение несимметричных СЛАУ



Факультет:	ПМИ
Группа:	ПМ-63
Студент:	Шепрут И.И.
Вариант:	1, 9
Преподаватель:	Патрушев И.И.

Новосибирск
2019

1 Цель работы

Разработать программу решения гармонической задачи методом конечных элементов. Провести сравнение прямого и итерационного методов решения получаемой в результате конечноэлементной аппроксимации СЛАУ.

Изучить особенности реализации итерационных методов BCG, BCGStab, GMRES для СЛАУ с несимметричными разреженными матрицами. Исследовать влияние предобуславливания на сходимость этих методов.

2 Задание

1. Выполнить конечноэлементную аппроксимацию исходного уравнения в соответствии с заданием. Получить формулы для вычисления компонент матрицы A и вектора правой части b .
2. Реализовать программу решения гармонической задачи с учетом следующих требований:
 - язык программирования C++ или Фортран;
 - предусмотреть возможность задания неравномерной сетки по пространству, разрывность параметров уравнения по подобластям, учет краевых условий;
 - матрицу хранить в разреженном строчном формате с возможностью регенерации ее в профильный формат;
 - реализовать (или воспользоваться реализованными в курсе «Численные методы») методы решения СЛАУ: итерационный — локально-оптимальную схему или метод сопряженных градиентов для несимметричных матриц с предобуславливанием и прямой — LU-разложение или его модификации [2, стр. 871], [3].
3. Протестировать разработанную программу на полиномах первой степени.
4. Провести исследования реализованных методов для сеток с небольшим количеством узлов 500-1000 и большим количеством узлов — порядка 20000-50000 для различных значений параметров: $10^{-4} \leq \omega \leq 10^9$, $10^2 \leq \lambda \leq 8 \cdot 10^5$, $0 \leq \sigma \leq 10^8$, $8.81 \cdot 10^{-12} \leq \chi \leq 10^{-10}$. Для всех решенных задач сравнить вычислительные затраты, требуемые для решения СЛАУ итерационным и прямым методом.

Лабораторная работа №3. Вариант 1: Решить одномерную гармоническую задачу в декартовых координатах, базисные функции — линейные.

Лабораторная работа №4. Вариант 9: Реализовать решение СЛАУ методом BSGSTAB с LU-предобуславливанием.

3 Исследования

Далее под нормой решения будет подразумеваться следующее значение: $\int_{-\infty}^{\infty} (u^s(x) - u^{s*}(x))dx + \int_{-\infty}^{\infty} (u^c(x) - u^{c*}(x))dx$, где u^s , u^c — истинные функции, а u^{s*} , u^{c*} — их конечноэлементные аппроксимации.

3.1 Изменение констант

Исследования проводились для функций: $u^s(x) = 3x$, $u^c(x) = -10x$, на отрезке: $x \in [1, 2]$, с начальными значениями констант: $\omega = 1$, $\lambda = 1$, $\sigma = 1$, $\chi = 10^{-11}$. Требуемая невязка при решении СЛАУ: $\varepsilon = 10^{-16}$.

Методы для решения СЛАУ: ЛОС с LU предобуславливанием; BSGSTAB с LU предобуславливанием.

Время указано в микросекундах.

3.1.1 Размер сетки 100

Параметр	LOS норма	BSG норма	LOS время	BSG время	LOS итераций	BSG итераций
$\omega = 0.1 \cdot 10^{-3}$	$2.94 \cdot 10^{-11}$	$2.82 \cdot 10^{-11}$	129	52	3	2
$\omega = 0.1 \cdot 10^{-2}$	$2.95 \cdot 10^{-11}$	$3.03 \cdot 10^{-11}$	43	51	3	2
$\omega = 0.01$	$2.93 \cdot 10^{-11}$	$3.15 \cdot 10^{-11}$	53	49	3	2
$\omega = 0.1$	$2.94 \cdot 10^{-11}$	$2.82 \cdot 10^{-11}$	56	49	3	2
$\omega = 0$	$2.94 \cdot 10^{-11}$	$2.83 \cdot 10^{-11}$	41	48	3	2
$\omega = 1$	$2.96 \cdot 10^{-11}$	$3.16 \cdot 10^{-11}$	42	49	3	2
$\omega = 10$	$2.1 \cdot 10^{-11}$	$2.1 \cdot 10^{-11}$	45	49	3	2
$\omega = 1 \cdot 10^2$	$8.12 \cdot 10^{-12}$	$8.18 \cdot 10^{-12}$	43	49	3	2
$\omega = 10 \cdot 10^2$	$4.16 \cdot 10^{-12}$	$4.16 \cdot 10^{-12}$	36	48	2	2
$\omega = 1 \cdot 10^4$	$2.39 \cdot 10^{-12}$	$2.39 \cdot 10^{-12}$	30	33	1	1
$\omega = 1 \cdot 10^5$	$4.35 \cdot 10^{-13}$	$4.36 \cdot 10^{-13}$	28	33	1	1
$\omega = 10 \cdot 10^5$	$7.14 \cdot 10^{-14}$	$2.15 \cdot 10^{-13}$	28	49	1	2
$\omega = 1 \cdot 10^7$	$5.61 \cdot 10^{-13}$	$7.36 \cdot 10^{-12}$	29	49	1	2
$\omega = 1 \cdot 10^8$	NaN	NaN	89	163	11	11
$\omega = 10 \cdot 10^8$	NaN	NaN	92	166	11	11

Параметр	LOS норма	BSG норма	LOS время	BSG время	LOS итераций	BSG итераций
$\lambda = 1 \cdot 10^2$	$2.96 \cdot 10^{-11}$	$3.29 \cdot 10^{-11}$	46	109	3	2
$\lambda = 10 \cdot 10^2$	$2.95 \cdot 10^{-11}$	$2.83 \cdot 10^{-11}$	44	50	3	2
$\lambda = 1 \cdot 10^4$	$2.95 \cdot 10^{-11}$	$4.63 \cdot 10^{-11}$	45	49	3	2
$\lambda = 1 \cdot 10^5$	$2.94 \cdot 10^{-11}$	$1.33 \cdot 10^{-10}$	45	49	3	2
$\lambda = 10 \cdot 10^5$	$2.93 \cdot 10^{-11}$	$3.17 \cdot 10^{-10}$	42	49	3	2
$\lambda = 8 \cdot 10^6$	$2.92 \cdot 10^{-11}$	$1.91 \cdot 10^{-9}$	43	49	3	2

Параметр	LOS норма	BSG норма	LOS время	BSG время	LOS итераций	BSG итераций
$\sigma = 0$	$2.92 \cdot 10^{-11}$	$2.99 \cdot 10^{-11}$	42	49	3	2
$\sigma = 1$	$2.96 \cdot 10^{-11}$	$3.16 \cdot 10^{-11}$	43	49	3	2
$\sigma = 10$	$2.17 \cdot 10^{-11}$	$2.23 \cdot 10^{-11}$	44	49	3	2
$\sigma = 1 \cdot 10^2$	$7.76 \cdot 10^{-12}$	$7.76 \cdot 10^{-12}$	44	49	3	2
$\sigma = 10 \cdot 10^2$	$4.13 \cdot 10^{-12}$	$4.14 \cdot 10^{-12}$	37	50	2	2
$\sigma = 1 \cdot 10^4$	$2.39 \cdot 10^{-12}$	$2.4 \cdot 10^{-12}$	29	33	1	1
$\sigma = 1 \cdot 10^5$	$4.35 \cdot 10^{-13}$	$4.33 \cdot 10^{-13}$	28	32	1	1
$\sigma = 10 \cdot 10^5$	$7.22 \cdot 10^{-14}$	$1.68 \cdot 10^{-13}$	30	49	1	2
$\sigma = 1 \cdot 10^7$	$4.12 \cdot 10^{-13}$	$3.24 \cdot 10^{-11}$	26	49	1	2
$\sigma = 1 \cdot 10^8$	$4.32 \cdot 10^{-12}$	$1.32 \cdot 10^{-9}$	27	49	1	2

Параметр	LOS норма	BSG норма	LOS время	BSG время	LOS итераций	BSG итераций
$\xi = 0.88 \cdot 10^{-11}$	$2.87 \cdot 10^{-11}$	$3.05 \cdot 10^{-11}$	42	49	3	2
$\xi = 0.1 \cdot 10^{-11}$	$3.05 \cdot 10^{-11}$	$3.03 \cdot 10^{-11}$	42	71	3	2
$\xi = 1 \cdot 10^{-11}$	$2.96 \cdot 10^{-11}$	$3.16 \cdot 10^{-11}$	42	49	3	2
$\xi = 1 \cdot 10^{-10}$	$3.77 \cdot 10^{-11}$	$3.81 \cdot 10^{-11}$	41	49	3	2

Вывод: LOS работает быстрее, чем BSG.

Вывод: в среднем у BSG меньше итераций, чем у LOS.

Вывод: норма решения лучше у ЛОС, чем у BSG.

Вывод: при увеличении ω норма улучшается, однако до некоторого предела $\omega = 10^8$, после которого метод расходится.

Вывод: при увеличении λ норма для ЛОС никак не меняется, когда для BSG она ухудшается.

Вывод: при увеличении σ норма повышается, и достигает экстремума в $\sigma = 10^6$.

Вывод: изменение χ в допустимых пределах ни на что особо не влияет.

3.1.2 Размер сетки 50000

Параметр	LOS норма	BSG норма	LOS время	BSG время	LOS итераций	BSG итераций
$\omega = 0.1 \cdot 10^{-3}$	$3.66 \cdot 10^{-7}$	$1.02 \cdot 10^{-4}$	24,400	37,100	3	2
$\omega = 0.1 \cdot 10^{-2}$	$3.63 \cdot 10^{-7}$	$4.5 \cdot 10^{-5}$	24,800	52,200	3	2
$\omega = 0.01$	$3.61 \cdot 10^{-7}$	$1.14 \cdot 10^{-5}$	35,800	32,400	3	2
$\omega = 0.1$	$3.62 \cdot 10^{-7}$	$1.1 \cdot 10^{-5}$	25,300	47,800	3	2
$\omega = 0$	$3.67 \cdot 10^{-7}$	$4.12 \cdot 10^{-6}$	31,800	32,300	3	2
$\omega = 1$	$3.42 \cdot 10^{-7}$	$1.8 \cdot 10^{-5}$	27,200	32,300	3	2
$\omega = 10$	$2.79 \cdot 10^{-7}$	$4.86 \cdot 10^{-6}$	30,700	48,500	3	2
$\omega = 1 \cdot 10^2$	$2.31 \cdot 10^{-8}$	$1.46 \cdot 10^{-7}$	50,600	34,300	3	2
$\omega = 10 \cdot 10^2$	$9.33 \cdot 10^{-8}$	$9.3 \cdot 10^{-8}$	32,200	29,300	3	2
$\omega = 1 \cdot 10^4$	$2.84 \cdot 10^{-9}$	$2.82 \cdot 10^{-9}$	36,600	33,900	3	2
$\omega = 1 \cdot 10^5$	$2.07 \cdot 10^{-10}$	$2.07 \cdot 10^{-10}$	24,700	28,700	3	2
$\omega = 10 \cdot 10^5$	$2.36 \cdot 10^{-11}$	$2.36 \cdot 10^{-11}$	26,400	29,400	3	2
$\omega = 1 \cdot 10^7$	$1.2 \cdot 10^{-11}$	$1.2 \cdot 10^{-11}$	21,400	30,000	2	2
$\omega = 1 \cdot 10^8$	$1.26 \cdot 10^{-11}$	$1.27 \cdot 10^{-11}$	16,400	19,000	1	1
$\omega = 10 \cdot 10^8$	$1.32 \cdot 10^{-11}$	$1.32 \cdot 10^{-11}$	16,700	18,600	1	1

Параметр	LOS норма	BSG норма	LOS время	BSG время	LOS итераций	BSG итераций
$\lambda = 1 \cdot 10^2$	$3.85 \cdot 10^{-8}$	$2.5 \cdot 10^{-5}$	24,700	27,500	3	2
$\lambda = 10 \cdot 10^2$	$4.05 \cdot 10^{-7}$	$5.52 \cdot 10^{-5}$	25,500	27,600	3	2
$\lambda = 1 \cdot 10^4$	$6.36 \cdot 10^{-8}$	$6.9 \cdot 10^{-5}$	24,200	27,900	3	2
$\lambda = 1 \cdot 10^5$	$5.36 \cdot 10^{-8}$	$5.13 \cdot 10^{-4}$	24,500	27,400	3	2
$\lambda = 10 \cdot 10^5$	$2.46 \cdot 10^{-7}$	$6.01 \cdot 10^{-4}$	24,100	36,300	3	3
$\lambda = 8 \cdot 10^6$	$4.15 \cdot 10^{-7}$	$2.49 \cdot 10^{-4}$	24,100	38,100	3	3

Параметр	LOS норма	BSG норма	LOS время	BSG время	LOS итераций	BSG итераций
$\sigma = 0$	$3.66 \cdot 10^{-7}$	$3.67 \cdot 10^{-6}$	27,200	27,500	3	2
$\sigma = 1$	$3.42 \cdot 10^{-7}$	$1.8 \cdot 10^{-5}$	23,600	27,900	3	2
$\sigma = 10$	$2.8 \cdot 10^{-7}$	$3.44 \cdot 10^{-6}$	25,300	27,200	3	2
$\sigma = 1 \cdot 10^2$	$4.09 \cdot 10^{-8}$	$1.38 \cdot 10^{-7}$	23,800	27,300	3	2
$\sigma = 10 \cdot 10^2$	$4.29 \cdot 10^{-9}$	$6.61 \cdot 10^{-9}$	25,900	26,700	3	2
$\sigma = 1 \cdot 10^4$	$4.31 \cdot 10^{-10}$	$4.53 \cdot 10^{-10}$	24,200	27,100	3	2
$\sigma = 1 \cdot 10^5$	$3.78 \cdot 10^{-11}$	$4.02 \cdot 10^{-11}$	23,500	27,500	3	2
$\sigma = 10 \cdot 10^5$	$1.54 \cdot 10^{-11}$	$1.53 \cdot 10^{-11}$	24,400	27,200	3	2
$\sigma = 1 \cdot 10^7$	$1.32 \cdot 10^{-11}$	$1.32 \cdot 10^{-11}$	19,900	27,100	2	2
$\sigma = 1 \cdot 10^8$	$1.3 \cdot 10^{-11}$	$1.3 \cdot 10^{-11}$	17,800	18,500	1	1

Параметр	LOS норма	BSG норма	LOS время	BSG время	LOS итераций	BSG итераций
$\xi = 0.88 \cdot 10^{-11}$	$3.4 \cdot 10^{-7}$	$1.37 \cdot 10^{-5}$	24,000	28,300	3	2
$\xi = 0.1 \cdot 10^{-11}$	$3.42 \cdot 10^{-7}$	$9.45 \cdot 10^{-6}$	24,200	26,900	3	2
$\xi = 1 \cdot 10^{-11}$	$3.42 \cdot 10^{-7}$	$1.8 \cdot 10^{-5}$	24,700	28,100	3	2
$\xi = 1 \cdot 10^{-10}$	$3.38 \cdot 10^{-7}$	$1.55 \cdot 10^{-5}$	23,700	27,900	3	2

Вывод: при увеличении ω норма улучшается.

Вывод: норма при изменении λ колеблется сложным образом без явных экстремумов или монотонностей.

Вывод: при увеличении σ норма повышается монотонно.

Вывод: время решения увеличилось примерно в 1000 раз, когда количество элементов только в 500.

Остальные выводы подтвердились.

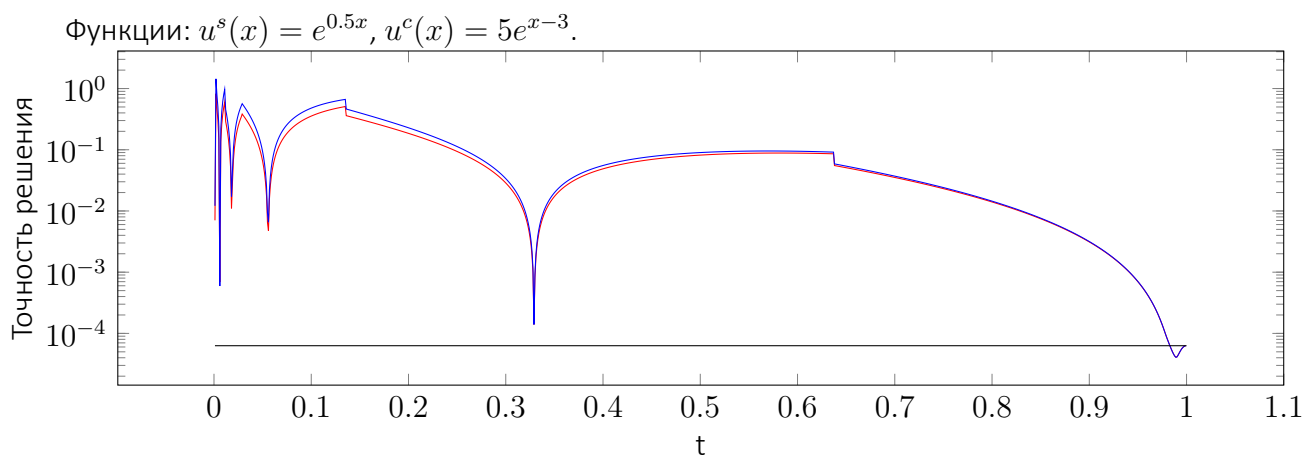
3.2 Зависимость точности от нелинейной сетки

Неравномерная функция рассчитывается аналогично предыдущей работе, используются функции $m_{3,t}(x)$, $m_{4,t}(x)$. Красным обозначается сетка, сгущающаяся к началу, синим к концу.

Исследования проводились на отрезке: $x \in [1, 2]$, с значениями констант: $\omega = 1$, $\lambda = 1$, $\sigma = 1$, $\chi = 1$. Требуемая невязка при решении СЛАУ: $\varepsilon = 10^{-16}$. Число элементов сетки: 50.

Замечание: так как точность решения или «норма» рассчитывается при помощи интеграла, то неравномерная сетка может более точно аппроксимировать значения функции в точках узлов, но менее точно делать это между узлами, и так как при неравномерной сетке пространство между узлами сильно увеличивается для некоторых значений t .

3.2.1 exp



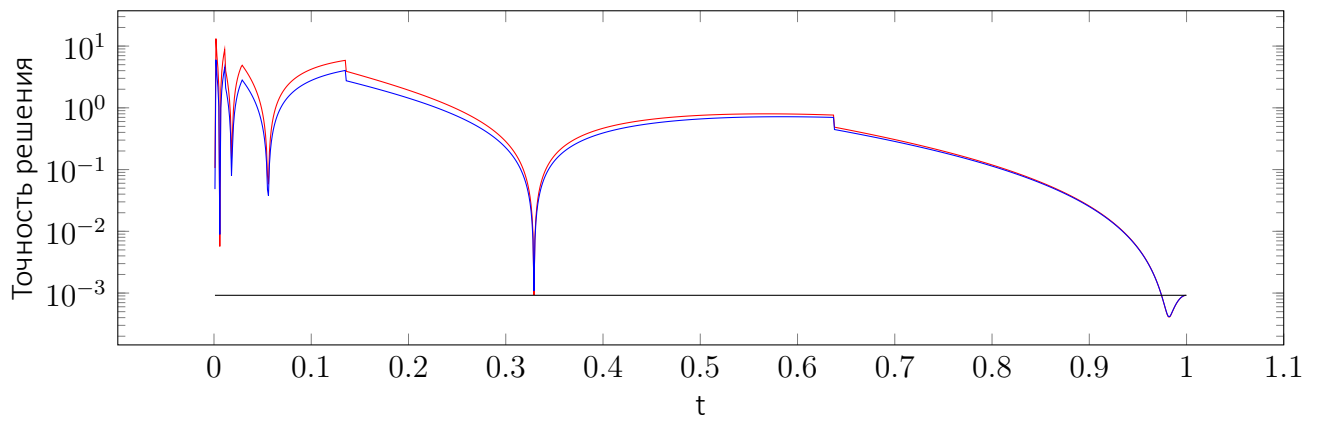
Вывод: аналогично предыдущей работе, наблюдается экстремум около точки $t = 0.3$, но в данном случае этот экстремум не дает прироста точности больше, чем равномерная сетка.

Вывод: аналогично предыдущей работе, наблюдается экстремум возле $t = 1$, при котором точность чу лучше чем при равномерной сетке.

Вывод: нет особого отличия в точности от сгущения к одному или к другому концу сетки.

3.2.2 -exp

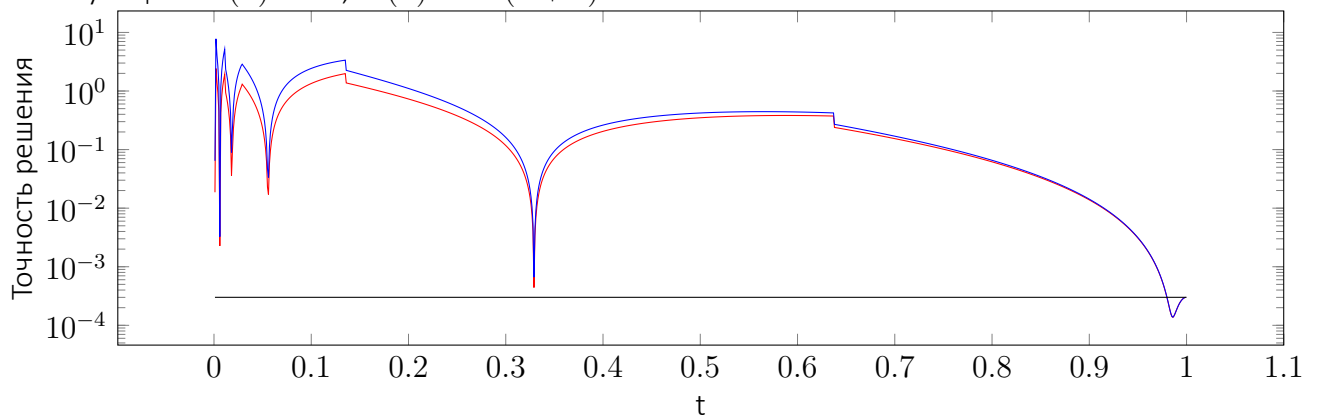
Функции: $u^s(x) = e^{-0.5x}$, $u^c(x) = 5e^{-x+3}$.



Вывод: обращение предыдущей функции в обратную сторону никак не повлияло на точность.

3.2.3 x^2

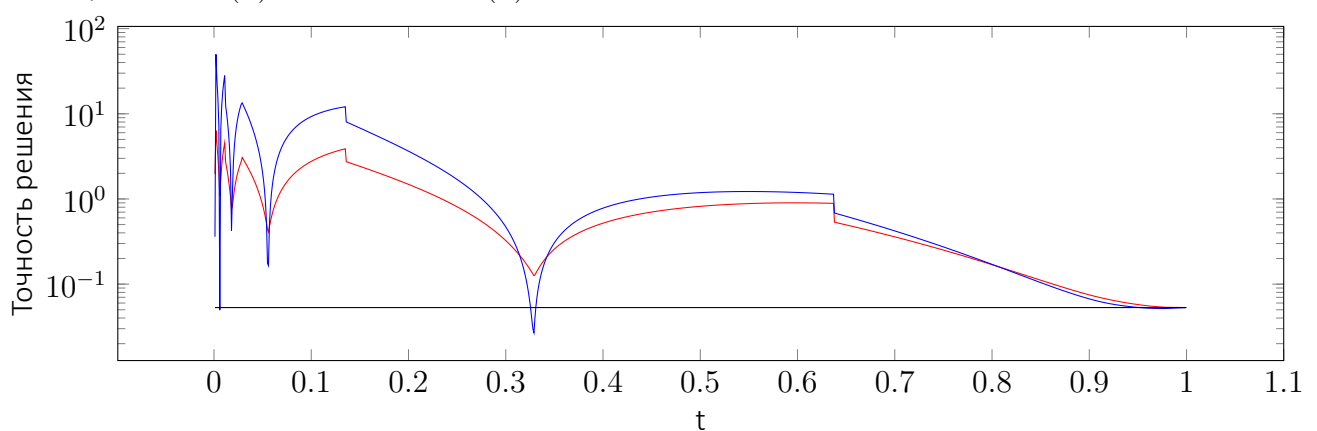
Функции: $u^s(x) = x^2$, $u^c(x) = 3x(x + 2) - 5$.



Вывод: предыдущие выводы не изменились.

3.2.4 Сложная функция

Функции: $u^s(x) = 3x^4 + 2e^x$, $u^c(x) = 6x - x^{e^x}$.



Вывод: на этот раз в точке возле $t = 0.3$ точность получилась больше, чем в равномерной сетке, а вот экстремум возле $t = 1$ отсутствовал вовсе.

4 Код программы

FILE main.cpp

```
1 #include <iomanip>
2 #include <chrono>
3 #include <fstream>
4 #include <thread>
5
6 #include "../2/lib.h"
7 #include <diagonal.h>
8
9 using namespace std;
10
11 class time_counter
12 {
13 public:
14     void start(void) {
15         _start = std::chrono::high_resolution_clock::now();
16     }
17     void end(void) {
18         _end = std::chrono::high_resolution_clock::now();
19     }
20     double get_microseconds(void) const {
21         return std::chrono::duration_cast<std::chrono::microseconds>(_end - _start).count();
22     }
23 private:
24     std::chrono::high_resolution_clock::time_point _start, _end;
25 };
26
27 //-----
28 double operator*(const vector<double>& a, const vector<double>& b) {
29     double sum = 0;
30     for (int i = 0; i < a.size(); ++i)
31         sum += a[i] * b[i];
32     return sum;
33 }
34
35 //-----
36 double length(const vector<double>& mas) {
37     return sqrt(mas*mas);
38 }
39
40 //-----
41 vector<double> to(const Vector& a) {
42     vector<double> result(a.size());
43     for (int i = 0; i < a.size(); ++i)
44         result[i] = a(i);
45     return result;
46 }
47
48 //-----
49 Vector to(const vector<double>& a) {
50     Vector result(a.size());
51     for (int i = 0; i < a.size(); ++i)
52         result(i) = a[i];
53     return result;
54 }
55
56 //-----
57 //-----
58 //-----
59
60 //-----
61 struct matrix
62 {
63     int n;
64     vector<double> d, l, u;
65     vector<int> i, j;
66
67     void init(int n1) {
68         n = n1;
69         d.clear();
70         l.clear();
71         u.clear();
72         i.clear();
73         j.clear();
74         d.resize(n);
75         i.resize(n+1, 0);
76     }
77
78     void toDense(Matrix& m) const {
79         m.resize(n, n, 0);
80         for (int i = 0; i < n; ++i) {
81             m(i, i) = d[i];
```

```

82     for (int j = 0; j < lineElemCount(i); ++j) {
83         m(i, lineElemRow(i, j)) = l[lineElemStart(i) + j];
84         m(lineElemRow(i, j), i) = u[lineElemStart(i) + j];
85     }
86 }
87 }
88
89 int lineElemStart(int line) const {
90     return i[line];
91 }
92 int lineStart(int line) const {
93     return j[lineElemStart(line)];
94 }
95 int lineSize(int line) const {
96     return line - lineStart(line);
97 }
98 int lineElemRow(int line, int elem) const {
99     return j[lineElemStart(line) + elem];
100 }
101 int lineElemCount(int line) const {
102     return i[line+1]-i[line];
103 }
104 };
105
106 //-----
107 //-----
108 //-----
109
110 void lu_decompose(const matrix& a, matrix& lu) {
111     lu = a;
112     for (int i = 0; i < lu.n; ++i) {
113         // Заполняем нижний треугольник
114         int line_start = lu.lineElemStart(i);
115         int line_end = lu.lineElemStart(i+1);
116         for (int j = line_start; j < line_end; ++j) {
117             double sum = 0;
118
119             int row = lu.j[j];
120             int row_start = lu.lineElemStart(row);
121             int row_end = lu.lineElemStart(row+1);
122
123             int kl = line_start;
124             int ku = row_start;
125
126             while (kl < j && ku < row_end) {
127                 if (lu.j[kl] == lu.j[ku]) { // Совпадают столбцы
128                     sum += lu.l[kl] * lu.u[ku];
129                     ku++;
130                     kl++;
131                 } else if (lu.j[kl] < lu.j[ku]) {
132                     kl++;
133                 } else {
134                     ku++;
135                 }
136             }
137
138             lu.l[j] = (a.l[j] - sum) / lu.d[row];
139         }
140
141         // Заполняем верхний треугольник
142         int row_start = lu.lineElemStart(i);
143         int row_end = lu.lineElemStart(i+1);
144         for (int j = line_start; j < line_end; ++j) {
145             double sum = 0;
146
147             int line = lu.j[j];
148             int line_start = lu.lineElemStart(line);
149             int line_end = lu.lineElemStart(line+1);
150
151             int kl = line_start;
152             int ku = row_start;
153
154             while (kl < line_end && ku < j) {
155                 if (lu.j[kl] == lu.j[ku]) { // Совпадают столбцы
156                     sum += lu.l[kl] * lu.u[ku];
157                     ku++;
158                     kl++;
159                 } else if (lu.j[kl] < lu.j[ku]) {
160                     kl++;
161                 } else {
162                     ku++;
163                 }
164             }
165
166             lu.u[j] = (a.u[j] - sum) / lu.d[line];
167         }
168
169         // Расчитываем диагональный элемент
170         double sum = 0;

```



```

171     int line_row_start = lu.lineElemStart(i);
172     int line_row_end = lu.lineElemStart(i+1);
173     for (int j = line_row_start; j < line_row_end; ++j)
174         sum += lu.l[j] * lu.u[j];
175
176     lu.d[i] = sqrt(a.d[i] - sum);
177 }
178 }
179
180 //-----
181 //-----
182 //-----
183
184 //-----
185 void mul(const matrix& a, vector<double>& x_y) {
186     vector<double> result(a.n, 0);
187
188     for (int i = 0; i < a.n; ++i) {
189         int start = a.lineElemStart(i);
190         int size = a.lineElemCount(i);
191         for (int j = 0; j < size; ++j) {
192             result[i] += a.l[start + j] * x_y[a.lineElemRow(i, j)];
193             result[a.lineElemRow(i, j)] += a.u[start + j] * x_y[i];
194         }
195     }
196
197     // Умножение диагональных элементов на вектор
198     for (int i = 0; i < a.n; ++i)
199         result[i] += a.d[i] * x_y[i];
200
201     x_y = result;
202 }
203
204 //-----
205 void mul_t(const matrix& a, vector<double>& x_y) {
206     vector<double> result(a.n, 0);
207
208     for (int i = 0; i < a.n; ++i) {
209         int start = a.lineElemStart(i);
210         int size = a.lineElemCount(i);
211         for (int j = 0; j < size; ++j) {
212             result[i] += a.u[start + j] * x_y[a.lineElemRow(i, j)];
213             result[a.lineElemRow(i, j)] += a.l[start + j] * x_y[i];
214         }
215     }
216
217     // Умножение диагональных элементов на вектор
218     for (int i = 0; i < a.n; ++i)
219         result[i] += a.d[i] * x_y[i];
220
221     x_y = result;
222 }
223
224 //-----
225 void mul_l_invert_t(const matrix& l, vector<double>& y_x) {
226     for (int i = l.n - 1; i >= 0; i--) {
227         int start = l.lineElemStart(i);
228         int size = l.lineElemCount(i);
229
230         y_x[i] /= l.d[i];
231         for (int j = 0; j < size; ++j)
232             y_x[l.lineElemRow(i, j)] -= y_x[i] * l.l[start + j];
233     }
234 }
235
236 //-----
237 void mul_u_invert_t(const matrix& u, vector<double>& y_x) {
238     for (int i = 0; i < u.n; ++i) {
239         int start = u.lineElemStart(i);
240         int size = u.lineElemCount(i);
241
242         sumreal sum = 0;
243         for (int j = 0; j < size; ++j)
244             sum += u.u[start + j] * y_x[u.lineElemRow(i, j)];
245         y_x[i] = (y_x[i] - sum) / u.d[i];
246     }
247 }
248
249 //-----
250 void mul_l_invert(const matrix& l, vector<double>& y_x) {
251     for (int i = 0; i < l.n; ++i) {
252         int start = l.lineElemStart(i);
253         int size = l.lineElemCount(i);
254
255         sumreal sum = 0;
256         for (int j = 0; j < size; ++j)
257             sum += l.l[start + j] * y_x[l.lineElemRow(i, j)];
258         y_x[i] = (y_x[i] - sum) / l.d[i];

```

```

259 }
260 }
261
262 //-----
263 void mul_u_invert(const matrix& u, vector<double>& y_x) {
264     for (int i = u.n-1; i >= 0; i--) {
265         int start = u.lineElemStart(i);
266         int size = u.lineElemCount(i);
267
268         y_x[i] /= u.d[i];
269         for (int j = 0; j < size; ++j)
270             y_x[u.lineElemRow(i, j)] -= y_x[i] * u.u[start + j];
271     }
272 }
273
274 //-----
275 void mul_u(const matrix& u, vector<double>& x_y) {
276     vector<double> result(u.n, 0);
277
278     for (int i = 0; i < u.n; ++i) {
279         int start = u.lineElemStart(i);
280         int size = u.lineElemCount(i);
281         for (int j = 0; j < size; ++j) {
282             result[u.lineElemRow(i, j)] += u.u[start + j] * x_y[i];
283         }
284     }
285
286     // Умножение диагональных элементов на вектор
287     for (int i = 0; i < u.n; ++i)
288         result[i] += u.d[i] * x_y[i];
289
290     x_y = result;
291 }
292
293 //-----
294 void mul(const vector<double>& d, vector<double>& x_y) {
295     for (int i = 0; i < d.size(); i++)
296         x_y[i] *= d[i];
297 }
298
299 //-----
300 void mul_invert(const vector<double>& d, vector<double>& x_y) {
301     for (int i = 0; i < d.size(); i++)
302         x_y[i] /= d[i];
303 }
304
305 //-----
306 //-----
307 //-----
308
309 //-----
310 class SLAU
311 {
312 public:
313
314     //-----
315     pair<int, double> los2() {
316         lu_decompose(a, lu);
317         x.clear();
318         x.resize(n, 0);
319
320         r = x;
321         mul(a, r);
322         for (int i = 0; i < n; i++)
323             r[i] = f[i] - r[i];
324         mul_l_invert(lu, r);
325
326         z = r;
327         mul_u_invert(lu, z);
328
329         p = z;
330         mul(a, p);
331         mul_l_invert(lu, p);
332
333         double flen = sqrt(f*f);
334         double residual;
335
336         int i = 0;
337         while (true) {
338             double pp = p*p;
339             double alpha = (p*r) / pp;
340             for (int i = 0; i < n; ++i) {
341                 x[i] += alpha * z[i];
342                 r[i] -= alpha * p[i];
343             }
344             t1 = r;
345             mul_u_invert(lu, t1);
346             t2 = t1;
347             mul(a, t2);

```

```

348     mul_l_invert(lu, t2);
349     double beta = -(p*t2) / pp;
350     for (int i = 0; i < n; ++i) {
351         z[i] = t1[i] + beta * z[i];
352         p[i] = t2[i] + beta * p[i];
353     }
354     residual = length(r) / flen;
355     i++;
356
357     if (is_log) cout << "Iteration: " << setw(4) << i << ", Residual: " << setw(20) <<
    ↪ setprecision(16) << residual << endl;
358     if (fabs(residual) < eps || i > maxiter)
359         break;
360 }
361
362 return {i, residual};
363 }
364
365 //-----
366 pair<int, double> bsg_stab_lu() {
367     lu_decompose(a, lu);
368     x.clear();
369     x.resize(n, 0);
370     vector<double> r0(n, 0);
371     vector<double> y = x;
372     mul(a, y);
373
374     r = x;
375     mul(a, r);
376     for (int i = 0; i < n; i++)
377         r[i] = f[i] - r[i];
378     mul_l_invert(lu, r);
379
380     z = r;
381     mul_u_invert(lu, z);
382
383     r0 = r; // r0 - это r0
384     p = r;
385
386     double flen = sqrt(f*f);
387     double residual;
388
389     int i = 0;
390     while (true) {
391         t1 = z;
392         mul_u_invert(lu, t1);
393         mul(a, t1);
394         mul_l_invert(lu, t1);
395         double rr0 = r*r0;
396         double alpha = (rr0) / (t1*r0);
397         for (int i = 0; i < n; ++i)
398             p[i] = r[i] - alpha * t1[i];
399
400         t2 = p;
401         mul_u_invert(lu, t2);
402         mul(a, t2);
403         mul_l_invert(lu, t2);
404         double gamma = (p*t2) / (t2*t2);
405         for (int i = 0; i < n; ++i) {
406             y[i] = y[i] + alpha * z[i] + gamma * p[i];
407             r[i] = p[i] - gamma * t2[i];
408         }
409
410         double beta = alpha*(r*r0)/(gamma * rr0);
411         for (int i = 0; i < n; ++i)
412             z[i] = r[i] + beta * z[i] - beta * gamma * t1[i];
413         x = y;
414         mul_u_invert(lu, x);
415
416         residual = length(r) / flen;
417         i++;
418
419         if (is_log) cout << "Iteration: " << setw(4) << i << ", Residual: " << setw(20) <<
    ↪ setprecision(16) << residual << endl;
420         if (fabs(residual) < eps || i > maxiter)
421             break;
422     }
423
424     return {i, residual};
425 }
426
427 int n, maxiter;
428 double eps;
429 matrix a, lu;
430 vector<double> f;
431 vector<double> r, z, p;
432 vector<double> x, t1, t2;
433 bool is_log;

```

```

434 };
435
436
437 //-----
438 //-----
439 //-----
440 //-----
441 //-----
442 //-----
443 //-----
444 //-----
445 //-----
446 //-----
447 //-----
448 //-----
449 //-----
450 //-----
451
452 struct Constants
453 {
454     double sigma, lambda, xi, omega;
455 };
456
457 //-----
458 Function1D calcRightPartS(
459     const Function1D& us,
460     const Function1D& uc,
461     const Constants& c
462 ) {
463     // fs = -lambda * div(grad us) - omega * sigma * uc - omega^2 * xi * uc
464     return [=](double x) -> double {
465         using namespace placeholders;
466         auto divgrad = calcSecondDerivative(us);
467         return -c.lambda * divgrad(x) - c.omega * c.sigma * uc(x) - c.omega * c.omega * c.xi * us(x);
468     };
469 }
470
471 //-----
472 Function1D calcRightPartC(
473     const Function1D& us,
474     const Function1D& uc,
475     const Constants& c
476 ) {
477     // fs = -lambda * div(grad uc) + omega * sigma * us - omega^2 * xi * uc
478     return [=](double x) -> double {
479         using namespace placeholders;
480         auto divgrad = calcSecondDerivative(uc);
481         return -c.lambda * divgrad(x) + c.omega * c.sigma * us(x) - c.omega * c.omega * c.xi * uc(x);
482     };
483 }
484
485 const int count_integral = 50;
486
487 //-----
488 double calcPIntegral(int i, int j, const lin_approx_t& u, const Constants& c) {
489     auto f = [=](double x) -> double {
490         return c.lambda * u.basic_grad(x, i) * u.basic_grad(x, j) - c.omega * c.omega * c.xi *
491             ↪ u.basic(x, i) * u.basic(x, j);
492     };
493     vector<double> X;
494     if (i == j)
495         make_grid(X, u.left(i), u.right(i), count_integral);
496     else
497         make_grid(X, min(u.middle(i), u.middle(j)), min(u.right(i), u.right(j)), count_integral);
498     return integral_gauss3(X, f);
499 }
500
501 //-----
502 double calcCIntegral(int i, int j, const lin_approx_t& u, const Constants& c) {
503     auto f = [=](double x) -> double {
504         return u.basic(x, i) * u.basic(x, j);
505     };
506     vector<double> X;
507     if (i == j)
508         make_grid(X, u.left(i), u.right(i), count_integral);
509     else
510         make_grid(X, min(u.middle(i), u.middle(j)), min(u.right(i), u.right(j)), count_integral);
511     return c.omega * c.sigma * integral_gauss3(X, f);
512 }
513
514 //-----
515 EMatrix calcLocalMatrix(int i, int j, const lin_approx_t& u, const Constants& c, bool isCalcLeftDown) {
516     EMatrix result(4, 4);
517     double p11 = calcPIntegral(i, i, u, c);
518     double c11 = calcCIntegral(i, i, u, c);
519     double p12 = calcPIntegral(i, j, u, c);
520     double c12 = calcCIntegral(i, j, u, c);
521     double p21 = p12;
522     double c21 = c12;

```

```

522 double p22 = (isCalcLeftDown) ? calcPIntegral(j, j, u, c) : 1;
523 double c22 = (isCalcLeftDown) ? calcCIntegral(j, j, u, c) : 1;
524
525 result <<
526     p11, -c11, p12, -c12,
527     c11, p11,  c12, p12,
528     p21, -c21, p22, -c22,
529     c21, p21,  c22, p22;
530 return result;
531 }
532
533 //-----
534 EMatrix calcGlobalMatrix(const lin_approx_t& u, const Constants& c) {
535     EMatrix result(u.size() * 2, u.size() * 2);
536     result.fill(0);
537
538     for (int i = 0; i < u.size()-1; ++i) {
539         auto l = calcLocalMatrix(i, i+1, u, c, i == u.size()-2);
540         for (int x = 0; x < l.rows(); ++x) {
541             for (int y = 0; y < l.cols(); ++y) {
542                 result(i*2 + x, i*2 + y) = l(x, y);
543             }
544         }
545     }
546
547     return result;
548 }
549
550 //-----
551 matrix calcGlobalMatrixProfile(const lin_approx_t& u, const Constants& c) {
552     matrix result;
553     result.n = u.size() * 2;
554     result.d.resize(result.n, 0);
555     result.i.resize(result.n+1, -1);
556
557     int counter = 0;
558     for (int i = 0; i < u.size()-1; ++i) {
559         auto l = calcLocalMatrix(i, i+1, u, c, true);
560         for (int y = 0; y < l.cols(); ++y) {
561             if (result.i[i*2 + y] == -1) {
562                 result.i[i*2 + y] = counter;
563             }
564             if ((i != 0 && y > 1) || (i == 0)) {
565                 for (int x = 0; x <= y; ++x) {
566                     if (x == y) {
567                         result.d[i*2 + x] = l(x, y);
568                     } else {
569                         result.j.push_back(i*2 + x);
570                         result.u.push_back(l(x, y));
571                         result.l.push_back(l(y, x));
572                         counter++;
573                     }
574                 }
575             }
576         }
577     }
578     result.i.back() = counter;
579
580     return result;
581 }
582
583 //-----
584 template<class V>
585 V calcB(
586     const lin_approx_t& u,
587     const Constants& c,
588     const Function1D& fs,
589     const Function1D& fc
590 ) {
591     V result(u.size() * 2);
592     for (int i = 0; i < u.size(); ++i) {
593         auto funs = [&] (double x) -> double { return fs(x) * u.basic(x, i); };
594         auto func = [&] (double x) -> double { return fc(x) * u.basic(x, i); };
595         vector<double> X;
596         make_grid(X, u.left(i), u.right(i), count integral);
597         result(i*2 + 0) = integral_gauss3(X, funs);
598         result(i*2 + 1) = integral_gauss3(X, func);
599     }
600     return result;
601 }
602
603 //-----
604 template<class V>
605 void setAnswer(
606     lin_approx_t& us,
607     lin_approx_t& uc,
608     const V& result
609 ) {

```

```

610     us.q.resize(us.size());
611     uc.q.resize(uc.size());
612     for (int i = 0; i < us.size(); ++i) {
613         us.q[i] = result(i*2 + 0);
614         uc.q[i] = result(i*2 + 1);
615     }
616 }
617
618 //-----
619 MatrixDiagonal calcGlobalMatrixDiag(const lin_approx_t& u, const Constants& c) {
620     Diagonal d(u.size() * 2);
621     vector<int> format = {0, -3, -2, -1, 1, 2, 3};
622     MatrixDiagonal result(u.size() * 2, format);
623     for (int i = 0; i < u.size()-1; ++i) {
624         auto l = calcLocalMatrix(i, i+1, u, c, i == u.size()-2);
625         for (int x = 0; x < l.rows(); ++x) {
626             for (int y = 0; y < l.cols(); ++y) {
627                 int line = i*2 + x;
628                 int row = i*2 + y;
629                 int diag = d.calcDiag_byLR(line, row);
630                 int pos = d.calcPos_byLR(line, row);
631                 diag = distance(format.begin(), find(format.begin(), format.end(), diag));
632                 result.begin(diag)[pos] = l(x, y);
633             }
634         }
635     }
636     return result;
637 }
638
639 //-----
640 void setFirstBoundaryConditions(
641     EMatrix& A,
642     EVector& b,
643     const Function1D& us_true,
644     const Function1D& uc_true,
645     const lin_approx_t& u
646 ) {
647     auto clear_line = [&] (int line) {
648         for (int i = 0; i < A.cols(); ++i) A(line, i) = 0;
649     };
650
651     clear_line(0); A(0, 0) = 1; b(0) = us_true(u.middle(0));
652     clear_line(1); A(1, 1) = 1; b(1) = uc_true(u.middle(0));
653
654     int end = A.cols()-1;
655     clear_line(end-1); A(end-1, end-1) = 1; b(end-1) = us_true(u.middle(u.size()-1));
656     clear_line(end); A(end, end) = 1; b(end) = uc_true(u.middle(u.size()-1));
657 }
658
659 //-----
660 void setFirstBoundaryConditions(
661     MatrixDiagonal& A,
662     Vector& b,
663     const Function1D& us_true,
664     const Function1D& uc_true,
665     const lin_approx_t& u
666 ) {
667     auto clear_line = [&] (int line) {
668         matrix_diagonal_line_iterator it(A.dimension(), A.getFormat(), false);
669         for (; !it.isEnd(); ++it)
670             for (; !it.isLineEnd(); ++it)
671                 if (it.i == line)
672                     A.begin(it.dn)[it.di] = 0;
673     };
674
675     clear_line(0); A.begin(0)[0] = 1; b(0) = us_true(u.middle(0));
676     clear_line(1); A.begin(0)[1] = 1; b(1) = uc_true(u.middle(0));
677
678     int end = A.dimension()-1;
679     clear_line(end-1); A.begin(0)[end-1] = 1; b(end-1) = us_true(u.middle(u.size()-1));
680     clear_line(end); A.begin(0)[end] = 1; b(end) = uc_true(u.middle(u.size()-1));
681 }
682
683 //-----
684 void setFirstBoundaryConditions(
685     matrix& A,
686     vector<double>& b,
687     const Function1D& us_true,
688     const Function1D& uc_true,
689     const lin_approx_t& u
690 ) {
691     auto clear_lines = [&] (vector<int> lines) {
692         for (auto& i : lines) A.d[i] = 0;
693
694         for (int i = 0; i < A.i.size()-1; i++) {
695             for (int pj = A.i[i]; pj < A.i[i+1]; pj++) {
696                 int j = A.j[pj];

```

```

698         if (find(lines.begin(), lines.end(), j) != lines.end()) A.u[pj] = 0;
699         if (find(lines.begin(), lines.end(), i) != lines.end()) A.l[pj] = 0;
700     }
701 };
702
703
704 int end = A.n-1;
705 clear_lines({0, 1, end-1, end});
706
707 A.d[0] = 1; b[0] = us_true(u.middle(0));
708 A.d[1] = 1; b[1] = uc_true(u.middle(0));
709
710 A.d[end-1] = 1; b[end-1] = us_true(u.middle(u.size()-1));
711 A.d[end] = 1; b[end] = uc_true(u.middle(u.size()-1));
712 }
713
714 //-----
715 //-----
716 //-----
717
718 double spacea = 1, spaceb = 2;
719
720 //-----
721 struct Result1
722 {
723     double los_integral_norm, bsg_integral_norm;
724     double los_time, bsg_time;
725     int los_iter, bsg_iter;
726 };
727
728 ostream& operator<<(ostream& out, const Result1& res) {
729     out
730         << res.los_integral_norm << "\t"
731         << res.bsg_integral_norm << "\t"
732         << res.los_time << "\t"
733         << res.bsg_time << "\t"
734         << res.los_iter << "\t"
735         << res.bsg_iter;
736     return out;
737 }
738
739 Result1 calcMethod(
740     const Constants& c,
741     Function1D us_true,
742     Function1D uc_true,
743     const vector<double>& grid
744 ) {
745     auto fs = calcRightPartS(us_true, uc_true, c);
746     auto fc = calcRightPartC(us_true, uc_true, c);
747
748     lin_approx_t us, uc;
749     us.x = grid;
750     uc.x = grid;
751
752     auto A2 = calcGlobalMatrixProfile(us, c);
753     auto b2 = to(calcB<Vector>(us, c, fs, fc));
754     setFirstBoundaryConditions(A2, b2, us_true, uc_true, us);
755
756     SLAU slau;
757     slau.maxiter = 10;
758     slau.eps = 1e-16;
759     slau.is_log = false;
760     slau.n = A2.n;
761     slau.a = A2;
762     slau.f = b2;
763     slau.x.resize(slau.n);
764     slau.t1.resize(slau.n);
765     slau.t2.resize(slau.n);
766
767     time_counter t1;
768     t1.start();
769     auto res_los = slau.los2();
770     t1.end();
771     setAnswer(us, uc, to(slau.x));
772     double los_residual = norm(us_true, us) + norm(uc_true, uc);
773
774     time_counter t2;
775     t2.start();
776     auto res_bsg = slau.bsg_stab_lu();
777     t2.end();
778     setAnswer(us, uc, to(slau.x));
779     double bsg_residual = norm(us_true, us) + norm(uc_true, uc);
780
781     return {los_residual, bsg_residual, t1.get_microseconds(), t2.get_microseconds(), res_los.first,
782         ↵ res_bsg.first};
783 }
784
785 //-----
786 struct grid_point { double t, los_norm, bsg_norm; };

```

```

786 vector<grid_point> calcGrid(
787     const Constants& c,
788     int elemCount,
789     Function1D us_true,
790     Function1D uc_true,
791     function<Function1D(double)> moveMaker
792 ) {
793     int n = 1000;
794     vector<grid_point> result;
795     for (double t = 1.0/n; t < 1.0 + 1.0/n; t += 1.0/n) {
796         vector<double> grid;
797         make_grid(grid, spacea, spaceb, elemCount, moveMaker(t));
798
799         auto res = calcMethod(c, us_true, uc_true, grid);
800
801         result.push_back({t, res.los_integral_norm, res.bsg_integral_norm});
802     }
803     return result;
804 }
805
806 //-----
807 void writeParametersInvestigation(
808     Function1D us_true,
809     Function1D uc_true,
810     int elemCount,
811     const string& filename
812 ) {
813     vector<double> grid;
814     make_grid(grid, spacea, spaceb, elemCount);
815
816     // omega: 10^-4, 10^9
817     // lambda: 10^2, 8*10^6
818     // sigma: 0, 10^8
819     // xi: 8.81*10^-12, 10^-10
820
821     vector<double> omega = {1e-4, 1e-3, 1e-2, 1e-1, 0, 1, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9};
822     vector<double> lambda = {1e2, 1e3, 1e4, 1e5, 1e6, 8e6};
823     vector<double> sigma = {0, 1, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8};
824     vector<double> xi = {8.81e-12, 1e-12, 1e-11, 1e-10};
825
826     Constants c;
827     c.omega = 1;
828     c.lambda = 1;
829     c.sigma = 1;
830     c.xi = 1e-11;
831
832     #define write_parameter(par) {\
833         ofstream fout(filename + " " + #par + ".txt"); \
834         fout << "param\tlos_norm\tbsg_norm\tlos_time\tbsg_time\tlos_iter\tbsg_iter" << endl; \
835         fout << scientific << setprecision(2); \
836         for (auto& i : par) { \
837             c.par = i; \
838             auto res = calcMethod(c, us_true, uc_true, grid); \
839             fout << "
840 " << #par << " = " << write_for_latex_double(i, 2) << " " << "\t" << res << endl; \
841             } c.par = 1; \
842             fout.close(); }\
843
844     write_parameter(omega);
845     write_parameter(lambda);
846     write_parameter(sigma);
847     write_parameter(xi);
848 }
849
850 //-----
851 void writeGridInvestigation(
852     Function1D us_true,
853     Function1D uc_true,
854     int elemCount,
855     const string& filename) {
856     ofstream fout(filename);
857
858     fout << "t\tnorma\tnormb\tnorm" << endl;
859
860     Constants c;
861     c.lambda = 1;
862     c.omega = 1;
863     c.xi = 1;
864     c.sigma = 1;
865
866     using namespace placeholders;
867     auto grid0 = calcGrid(c, elemCount, us_true, uc_true, bind(getMove0, _1, 1));
868     auto grid1 = calcGrid(c, elemCount, us_true, uc_true, bind(getMove1, _1, 1));
869
870     for (int i = 0; i < grid0.size(); ++i) {
871         fout
872     }
873 }

```



```

874         << grid0[i].t << "\t"
875         << grid0[i].bsg_norm << "\t"
876         << grid1[i].bsg_norm << "\t"
877         << grid0[grid0.size()-2].bsg_norm << endl;
878     }
879
880     fout.close();
881 }
882
883 //-----
884 //-----
885 //-----
886
887 int main() {
888     // cout.precision(3);
889
890     // Constants c;
891     // c.lambda = 32;
892     // c.omega = 100;
893     // c.xi = 10;
894     // c.sigma = 24;
895
896     // auto us_true = [] (double x) -> double { return 3*x*x*x*x + 2*exp(x); };
897     // auto uc_true = [] (double x) -> double { return 6*x - pow(x, exp(x)); };
898
899     // auto fs = calcRightPartS(us_true, uc_true, c);
900     // auto fc = calcRightPartC(us_true, uc_true, c);
901
902     // lin_approx_t us, uc;
903     // make_grid(us.x, 1, 2, 45000);
904     // uc.x = us.x;
905
906     // /*auto A = calcGlobalMatrix(us, c);
907     // auto b = calcB<EVector>(us, c, fs, fc);
908     // setFirstBoundaryConditions(A, b, us_true, uc_true, us);
909
910     // Eigen::JacobiSVD<EMatrix> svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV);
911     // EVector x = svd.solve(b);
912
913     // setAnswer(us, uc, x);*/
914
915     // double residual;
916
917     // auto A2 = calcGlobalMatrixProfile(us, c);
918     // auto b2 = to(calcB<Vector>(us, c, fs, fc));
919     // setFirstBoundaryConditions(A2, b2, us_true, uc_true, us);
920
921     // SLAU slau;
922     // slau.maxiter = 10;
923     // slau.eps = 1e-16;
924     // slau.is_log = true;
925     // slau.n = A2.n;
926     // slau.a = A2;
927     // slau.f = b2;
928     // slau.x.resize(slau.n);
929     // slau.t1.resize(slau.n);
930     // slau.t2.resize(slau.n);
931
932     // time_counter t1;
933     // t1.start();
934     // slau.los2();
935     // t1.end();
936     // setAnswer(us, uc, to(slau.x));
937     // residual = norm(us_true, us) + norm(uc_true, uc);
938     // cout << "los 2 residual: " << residual << endl << endl;
939     // cout << "los 2 time: " << t1.get_milliseconds() << endl;
940
941     // time_counter t2;
942     // t2.start();
943     // slau.bsg_stab_lu();
944     // t2.end();
945     // setAnswer(us, uc, to(slau.x));
946     // residual = norm(us_true, us) + norm(uc_true, uc);
947     // cout << "bsg lu residual: " << residual << endl << endl;
948     // cout << "bsg lu time: " << t2.get_milliseconds() << endl;
949
950     // /*auto A1 = calcGlobalMatrixDiag(us, c);
951     // auto b1 = calcB<Vector>(us, c, fs, fc);
952     // setFirstBoundaryConditions(A1, b1, us_true, uc_true, us);*/
953     // /*Matrix denseA(A.cols(), A.rows());
954     // for (int i = 0; i < A.cols(); i++) {
955     //     for (int j = 0; j < A.rows(); j++) {
956     //         denseA(i, j) = A(i, j);
957     //     }
958     // }
959     // MatrixDiagonal A2(denseA);*/
960
961     // /*Vector x1;

```

```

962 // SolverSLAE Iterative solver;
963 // solver.epsilon = 1e-7;
964 // solver.isLog = false;
965 // solver.maxIterations = 5000;
966 // solver.start = Vector(us.size() * 2, 0);
967 // solver.w = 0.8;
968 // solver.seidel(A1, b1, x1);
969
970 // setAnswer(us, uc, x1);
971
972 // Vector b2;
973 // mul(A1, x1, b2);
974 // b2.negate();
975 // sum(b1, b2, b2);
976 // cout << "residual slae: " << calcNorm(b2) << endl;*/
977
978 // /*Matrix dense; A1.toDenseMatrix(dense);
979 // cout << A << endl << b << endl;
980 // dense.save(cout); cout << endl;
981 // b1.save(cout); cout << endl;
982 // cout << x << endl;
983 // x1.save(cout); cout << endl;*/
984
985 // /*Matrix dense; A2.toDense(dense);
986 // cout << A << endl;
987 // dense.save(cout); cout << endl;*/
988
989 // //double residual = norm(us_true, us) + norm(uc_true, uc);
990
991 // /*cout << "answer s: " << us.q << endl;
992 // cout << "should be s: " << calcTrulyApprox(us.x, us_true).q << endl;
993
994 // cout << "answer s: " << uc.q << endl;
995 // cout << "should be s: " << calcTrulyApprox(uc.x, uc_true).q << endl;*/
996
997 // //cout << "residual: " << residual << endl;
998
999 // system("pause");
1000
1001 vector<thread> t;
1002
1003 t.emplace_back([](){
1004     writeParametersInvestigation(
1005         [] (double x) -> double { return 3.0*x; },
1006         [] (double x) -> double { return -10.0*x; },
1007         100,
1008         "3_parameters_100"
1009     );
1010 });
1011
1012 t.emplace_back([](){
1013     writeParametersInvestigation(
1014         [] (double x) -> double { return 3.0*x; },
1015         [] (double x) -> double { return -10.0*x; },
1016         50000,
1017         "3_parameters_50000"
1018     );
1019 });
1020
1021 t.emplace_back([](){
1022     writeGridInvestigation(
1023         [] (double x) -> double { return exp(x/2); },
1024         [] (double x) -> double { return 5*exp(x-3); },
1025         50,
1026         "3_grid_exp.txt"
1027     );
1028 });
1029
1030 t.emplace_back([](){
1031     writeGridInvestigation(
1032         [] (double x) -> double { return exp(-x/2); },
1033         [] (double x) -> double { return 5*exp(-x+3); },
1034         50,
1035         "3_grid_exp_minus.txt"
1036     );
1037 });
1038
1039 t.emplace_back([](){
1040     writeGridInvestigation(
1041         [] (double x) -> double { return x*x; },
1042         [] (double x) -> double { return 3.0*x*(x+2)-5; },
1043         50,
1044         "3_grid_x2.txt"
1045     );
1046 });
1047
1048 t.emplace_back([](){
1049     writeGridInvestigation(

```

```

1050     [] (double x) -> double { return 3*x*x*x*x + 2*exp(x); },
1051     [] (double x) -> double { return 6*x - pow(x, exp(x)); },
1052     50,
1053     "3_grid_uuu.txt"
1054 );
1055 });
1056
1057 for (auto& i : t) i.join();
1058 }

```