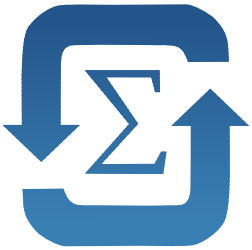
Министерство образования и науки Российской Федерации

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 предобусловливания 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 перегенерации ее в профильный формат;
   - реализовать (или воспользоваться реализованными в курсе «Численные метды») 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: $10^{-4} \leqslant \omega \leqslant 10^{9}$, $10^{2} \leqslant \lambda \leqslant 8 \cdot 10^{5}$, $0 \leqslant \sigma \leqslant 10^{8}$, $8.81 \cdot 10^{-12} \leqslant \chi \leqslant 10^{-10}$. 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

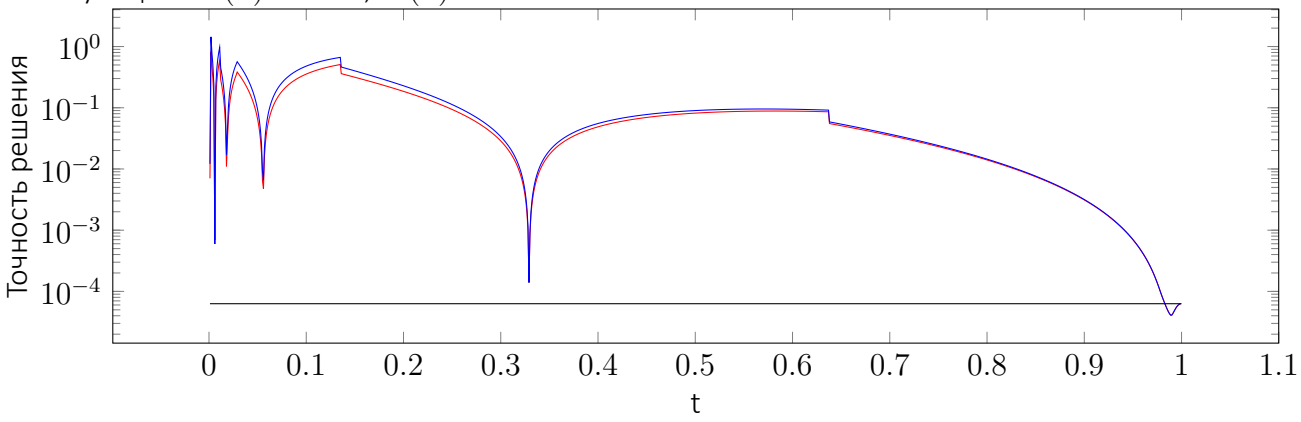
Неравномерная функция рассчитывается аналогично предыдущей работе, используются функции $m_{3,t}(x)$, $m_{4,t}(x)$. Красным обозначается сетка, сгущающаяся к началу, синим к концу.

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

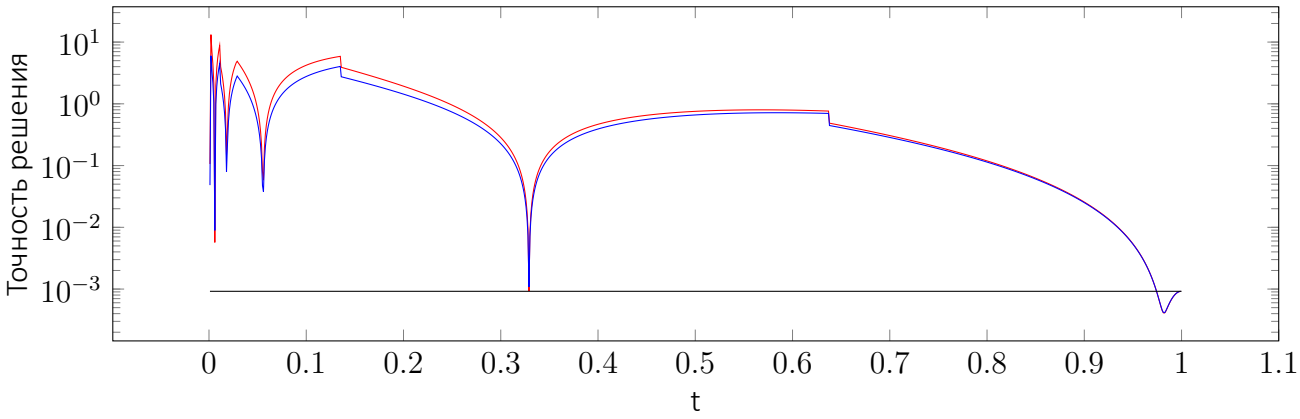
Функции: $u^s(x) = e^{0.5x}$, $u^c(x) = 5e^{x-3}$.



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

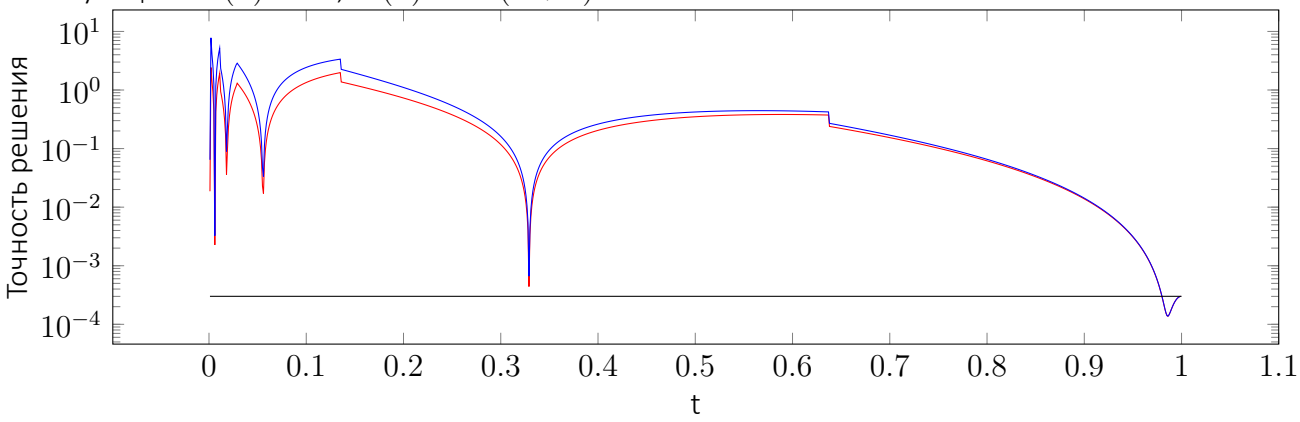
Функции: $u^s(x) = e^{-0.5x}$, $u^c(x) = 5e^{-x+3}$.



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

3.2.3 x2

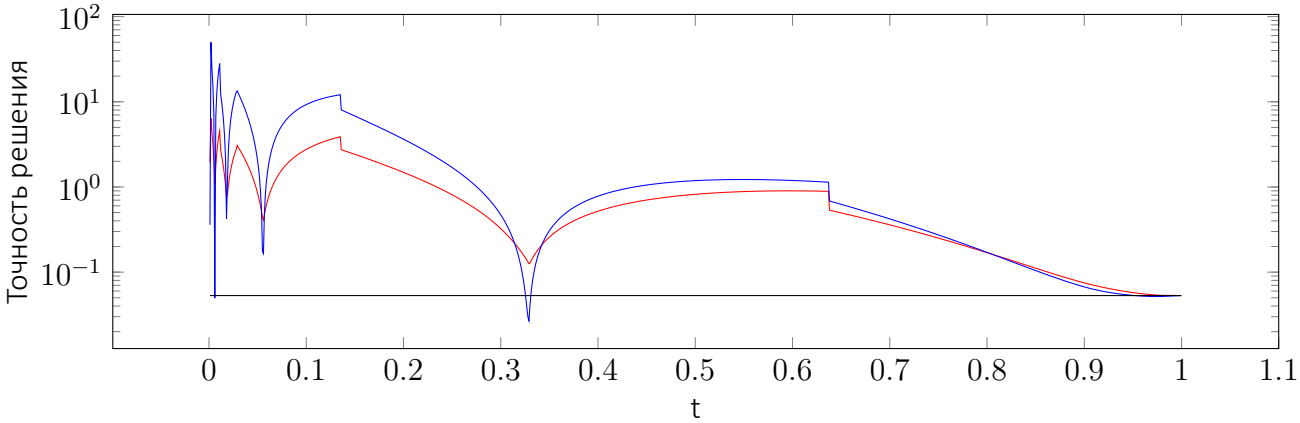
Функции: $u^s(x) = x^2$, $u^c(x) = 3x(x+2) - 5$.



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**

```cpp
#include <iomanip>
#include <chrono>
#include <fstream>
#include <thread>

#include "../2/lib.h"
#include <diagonal.h>

using namespace std;

class time_counter
{
public:
    void start(void) {
        _start = std::chrono::high_resolution_clock::now();
    }
    void end(void) {
        _end = std::chrono::high_resolution_clock::now();
    }
    double get_microseconds(void) const {
        return std::chrono::duration_cast<std::chrono::microseconds>(_end - _start).count();
    }
private:
    std::chrono::high_resolution_clock::time_point _start, _end;
};

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

//---------------------------------------------------------------------------
double length(const vector<double>& mas) {
    return sqrt(mas*mas);
}

//---------------------------------------------------------------------------
vector<double> to(const Vector& a) {
    vector<double> result(a.size());
    for (int i = 0; i < a.size(); ++i)
        result[i] = a(i);
    return result;
}

//---------------------------------------------------------------------------
Vector to(const vector<double>& a) {
    Vector result(a.size());
    for (int i = 0; i < a.size(); ++i)
        result(i) = a[i];
    return result;
}

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

//---------------------------------------------------------------------------
struct matrix
{
    int n;
    vector<double> d, l, u;
    vector<int> i, j;

    void init(int n1) {
        n = n1;
        d.clear();
        l.clear();
        u.clear();
        i.clear();
        j.clear();
        d.resize(n);
        i.resize(n+1, 0);
    }

    void toDense(Matrix& m) const {
        m.resize(n, n, 0);
        for (int i = 0; i < n; ++i) {
            m(i, i) = d[i];
```

```
            for (int j = 0; j < lineElemCount(i); ++j) {
                m(i, lineElemRow(i, j)) = l[lineElemStart(i) + j];
                m(lineElemRow(i, j), i) = u[lineElemStart(i) + j];
            }
        }
    }

    int lineElemStart(int line) const {
        return i[line];
    }
    int lineStart(int line) const {
        return j[lineElemStart(line)];
    }
    int lineSize(int line) const {
        return line - lineStart(line);
    }
    int lineElemRow(int line, int elem) const {
        return j[lineElemStart(line) + elem];
    }
    int lineElemCount(int line) const {
        return i[line+1]-i[line];
    }
};

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

void lu_decompose(const matrix& a, matrix& lu) {
    lu = a;
    for (int i = 0; i < lu.n; ++i) {
        // Заполняем нижний треугольник
        int line_start = lu.lineElemStart(i);
        int line_end = lu.lineElemStart(i+1);
        for (int j = line_start; j < line_end; ++j) {
            double sum = 0;

            int row = lu.j[j];
            int row_start = lu.lineElemStart(row);
            int row_end = lu.lineElemStart(row+1);

            int kl = line_start;
            int ku = row_start;

            while (kl < j && ku < row_end) {
                if (lu.j[kl] == lu.j[ku]) { // Совпадают столбцы
                    sum += lu.l[kl] * lu.u[ku];
                    ku++;
                    kl++;
                } else if (lu.j[kl] < lu.j[ku]) {
                    kl++;
                } else {
                    ku++;
                }
            }

            lu.l[j] = (a.l[j] - sum) / lu.d[row];
        }

        // Заполняем верхний треугольник
        int row_start = lu.lineElemStart(i);
        int row_end = lu.lineElemStart(i+1);
        for (int j = line_start; j < line_end; ++j) {
            double sum = 0;

            int line = lu.j[j];
            int line_start = lu.lineElemStart(line);
            int line_end = lu.lineElemStart(line+1);

            int kl = line_start;
            int ku = row_start;

            while (kl < line_end && ku < j) {
                if (lu.j[kl] == lu.j[ku]) { // Совпадают столбцы
                    sum += lu.l[kl] * lu.u[ku];
                    ku++;
                    kl++;
                } else if (lu.j[kl] < lu.j[ku]) {
                    kl++;
                } else {
                    ku++;
                }
            }

            lu.u[j] = (a.u[j] - sum) / lu.d[line];
        }

        // Расчитываем диагональный элемент
        double sum = 0;
```

```
        int line_row_start = lu.lineElemStart(i);
        int line_row_end = lu.lineElemStart(i+1);
        for (int j = line_row_start; j < line_row_end; ++j)
            sum += lu.l[j] * lu.u[j];

        lu.d[i] = sqrt(a.d[i] - sum);
    }
}

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

//---------------------------------------------------------------------------
void mul(const matrix& a, vector<double>& x_y) {
    vector<double> result(a.n, 0);

    for (int i = 0; i < a.n; ++i) {
        int start = a.lineElemStart(i);
        int size = a.lineElemCount(i);
        for (int j = 0; j < size; j++) {
            result[i] += a.l[start + j] * x_y[a.lineElemRow(i, j)];
            result[a.lineElemRow(i, j)] += a.u[start + j] * x_y[i];
        }
    }

    // Умножение диагональных элементов на вектор
    for (int i = 0; i < a.n; ++i)
        result[i] += a.d[i] * x_y[i];

    x_y = result;
}

//---------------------------------------------------------------------------
void mul_t(const matrix& a, vector<double>& x_y) {
    vector<double> result(a.n, 0);

    for (int i = 0; i < a.n; ++i) {
        int start = a.lineElemStart(i);
        int size = a.lineElemCount(i);
        for (int j = 0; j < size; j++) {
            result[i] += a.u[start + j] * x_y[a.lineElemRow(i, j)];
            result[a.lineElemRow(i, j)] += a.l[start + j] * x_y[i];
        }
    }

    // Умножение диагональных элементов на вектор
    for (int i = 0; i < a.n; ++i)
        result[i] += a.d[i] * x_y[i];

    x_y = result;
}

//---------------------------------------------------------------------------
void mul_l_invert_t(const matrix& l, vector<double>& y_x) {
    for (int i = l.n - 1; i >= 0; i--) {
        int start = l.lineElemStart(i);
        int size = l.lineElemCount(i);

        y_x[i] /= l.d[i];
        for (int j = 0; j < size; ++j)
            y_x[l.lineElemRow(i, j)] -= y_x[i] * l.l[start + j];
    }
}

//---------------------------------------------------------------------------
void mul_u_invert_t(const matrix& u, vector<double>& y_x) {
    for (int i = 0; i < u.n; ++i) {
        int start = u.lineElemStart(i);
        int size = u.lineElemCount(i);

        sumreal sum = 0;
        for (int j = 0; j < size; ++j)
            sum += u.u[start + j] * y_x[u.lineElemRow(i, j)];
        y_x[i] = (y_x[i] - sum) / u.d[i];
    }
}

//---------------------------------------------------------------------------
void mul_l_invert(const matrix& l, vector<double>& y_x) {
    for (int i = 0; i < l.n; ++i) {
        int start = l.lineElemStart(i);
        int size = l.lineElemCount(i);

        sumreal sum = 0;
        for (int j = 0; j < size; ++j)
            sum += l.l[start + j] * y_x[l.lineElemRow(i, j)];
        y_x[i] = (y_x[i] - sum) / l.d[i];
```

```
    }
}

//---------------------------------------------------------------------------
void mul_u_invert(const matrix& u, vector<double>& y_x) {
    for (int i = u.n-1; i >= 0; i--) {
        int start = u.lineElemStart(i);
        int size = u.lineElemCount(i);

        y_x[i] /= u.d[i];
        for (int j = 0; j < size; ++j)
            y_x[u.lineElemRow(i, j)] -= y_x[i] * u.u[start + j];
    }
}

//---------------------------------------------------------------------------
void mul_u(const matrix& u, vector<double>& x_y) {
    vector<double> result(u.n, 0);

    for (int i = 0; i < u.n; ++i) {
        int start = u.lineElemStart(i);
        int size = u.lineElemCount(i);
        for (int j = 0; j < size; j++) {
            result[u.lineElemRow(i, j)] += u.u[start + j] * x_y[i];
        }
    }

    // Умножение диагональных элементов на вектор
    for (int i = 0; i < u.n; ++i)
        result[i] += u.d[i] * x_y[i];

    x_y = result;
}

//---------------------------------------------------------------------------
void mul(const vector<double>& d, vector<double>& x_y) {
    for (int i = 0; i < d.size(); i++)
        x_y[i] *= d[i];
}

//---------------------------------------------------------------------------
void mul_invert(const vector<double>& d, vector<double>& x_y) {
    for (int i = 0; i < d.size(); i++)
        x_y[i] /= d[i];
}

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

//---------------------------------------------------------------------------
class SLAU
{
public:

//---------------------------------------------------------------------------
pair<int, double> los2() {
    lu_decompose(a, lu);
    x.clear();
    x.resize(n, 0);

    r = x;
    mul(a, r);
    for (int i = 0; i < n; i++)
        r[i] = f[i] - r[i];
    mul_l_invert(lu, r);

    z = r;
    mul_u_invert(lu, z);

    p = z;
    mul(a, p);
    mul_l_invert(lu, p);

    double flen = sqrt(f*f);
    double residual;

    int i = 0;
    while (true) {
        double pp = p*p;
        double alpha = (p*r) / pp;
        for (int i = 0; i < n; ++i) {
            x[i] += alpha * z[i];
            r[i] -= alpha * p[i];
        }
        t1 = r;
        mul_u_invert(lu, t1);
        t2 = t1;
        mul(a, t2);
```

```
        mul_l_invert(lu, t2);
        double beta = -(p*t2) / pp;
        for (int i = 0; i < n; ++i) {
            z[i] = t1[i] + beta * z[i];
            p[i] = t2[i] + beta * p[i];
        }
        residual = length(r) / flen;
        i++;

        if (is_log) cout << "Iteration: " << setw(4) << i << ", Residual: " << setw(20) <<
        ↪  setprecision(16) << residual << endl;
        if (fabs(residual) < eps || i > maxiter)
            break;
    }

    return {i, residual};
}

//-------------------------------------------------------------------------
pair<int, double> bsg_stab_lu() {
    lu_decompose(a, lu);
    x.clear();
    x.resize(n, 0);
    vector<double> r0(n, 0);
    vector<double> y = x;
    mul(a, y);

    r = x;
    mul(a, r);
    for (int i = 0; i < n; i++)
        r[i] = f[i] - r[i];
    mul_l_invert(lu, r);

    z = r;
    mul_u_invert(lu, z);

    r0 = r; // r0 - это r0
    p = r;

    double flen = sqrt(f*f);
    double residual;

    int i = 0;
    while (true) {
        t1 = z;
        mul_u_invert(lu, t1);
        mul(a, t1);
        mul_l_invert(lu, t1);
         double rr0 = r*r0;
        double alpha = (rr0) / (t1*r0);
        for (int i = 0; i < n; ++i)
            p[i] = r[i] - alpha * t1[i];

        t2 = p;
        mul_u_invert(lu, t2);
        mul(a, t2);
        mul_l_invert(lu, t2);
        double gamma = (p*t2) / (t2*t2);
        for (int i = 0; i < n; ++i) {
            y[i] = y[i] + alpha * z[i] + gamma * p[i];
            r[i] = p[i] - gamma * t2[i];
        }

        double beta = alpha*(r*r0)/(gamma * rr0);
        for (int i = 0; i < n; ++i)
            z[i] = r[i] + beta * z[i] - beta * gamma * t1[i];
        x = y;
        mul_u_invert(lu, x);

        residual = length(r) / flen;
        i++;

        if (is_log) cout << "Iteration: " << setw(4) << i << ", Residual: " << setw(20) <<
        ↪  setprecision(16) << residual << endl;
        if (fabs(residual) < eps || i > maxiter)
            break;
    }

    return {i, residual};
}

int n, maxiter;
double eps;
matrix a, lu;
vector<double> f;
vector<double> r, z, p;
vector<double> x, t1, t2;
bool is_log;
```

```
};

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

struct Constants
{
    double sigma, lambda, xi, omega;
};

//---------------------------------------------------------------------------
Function1D calcRightPartS(
    const Function1D& us,
    const Function1D& uc,
    const Constants& c
) {
    // fs = -lambda * div(grad us) - omega * sigma * uc - omega^2 * xi * uc
    return [=](double x) -> double {
        using namespace placeholders;
        auto divgrad = calcSecondDerivative(us);
        return -c.lambda * divgrad(x) - c.omega * c.sigma * uc(x) - c.omega * c.omega * c.xi * us(x);
    };
}

//---------------------------------------------------------------------------
Function1D calcRightPartC(
    const Function1D& us,
    const Function1D& uc,
    const Constants& c
) {
    // fs = -lambda * div(grad uc) + omega * sigma * us - omega^2 * xi * uc
    return [=](double x) -> double {
        using namespace placeholders;
        auto divgrad = calcSecondDerivative(uc);
        return -c.lambda * divgrad(x) + c.omega * c.sigma * us(x) - c.omega * c.omega * c.xi * uc(x);
    };
}

const int count_integral = 50;

//---------------------------------------------------------------------------
double calcPIntegral(int i, int j, const lin_approx_t& u, const Constants& c) {
    auto f = [&] (double x) -> double {
        return c.lambda * u.basic_grad(x, i) * u.basic_grad(x, j) - c.omega * c.omega * c.xi *
        ↪  u.basic(x, i) * u.basic(x, j);
    };
    vector<double> X;
    if (i == j)
        make_grid(X, u.left(i), u.right(i), count_integral);
    else
        make_grid(X, min(u.middle(i), u.middle(j)), min(u.right(i), u.right(j)), count_integral);
    return integral_gauss3(X, f);
}

//---------------------------------------------------------------------------
double calcCIntegral(int i, int j, const lin_approx_t& u, const Constants& c) {
    auto f = [&] (double x) -> double {
        return u.basic(x, i) * u.basic(x, j);
    };
    vector<double> X;
    if (i == j)
        make_grid(X, u.left(i), u.right(i), count_integral);
    else
        make_grid(X, min(u.middle(i), u.middle(j)), min(u.right(i), u.right(j)), count_integral);
    return c.omega * c.sigma * integral_gauss3(X, f);
}

//---------------------------------------------------------------------------
EMatrix calcLocalMatrix(int i, int j, const lin_approx_t& u, const Constants& c, bool isCalcLeftDown) {
    EMatrix result(4, 4);
    double p11 = calcPIntegral(i, i, u, c);
    double c11 = calcCIntegral(i, i, u, c);
    double p12 = calcPIntegral(i, j, u, c);
    double c12 = calcCIntegral(i, j, u, c);
    double p21 = p12;
    double c21 = c12;
```

```
    double p22 = (isCalcLeftDown) ? calcPIntegral(j, j, u, c) : 1;
    double c22 = (isCalcLeftDown) ? calcCIntegral(j, j, u, c) : 1;

    result <<
        p11, -c11, p12, -c12,
        c11, p11,  c12, p12,
        p21, -c21, p22, -c22,
        c21, p21,  c22, p22;
    return result;
}

//---------------------------------------------------------------------------
EMatrix calcGlobalMatrix(const lin_approx_t& u, const Constants& c) {
    EMatrix result(u.size() * 2, u.size() * 2);
    result.fill(0);

    for (int i = 0; i < u.size()-1; ++i) {
        auto l = calcLocalMatrix(i, i+1, u, c, i == u.size()-2);
        for (int x = 0; x < l.rows(); ++x) {
            for (int y = 0; y < l.cols(); ++y) {
                result(i*2 + x, i*2 + y) = l(x, y);
            }
        }
    }

    return result;
}

//---------------------------------------------------------------------------
matrix calcGlobalMatrixProfile(const lin_approx_t& u, const Constants& c) {
    matrix result;
    result.n = u.size() * 2;
    result.d.resize(result.n, 0);
    result.i.resize(result.n+1, -1);

    int counter = 0;
    for (int i = 0; i < u.size()-1; ++i) {
        auto l = calcLocalMatrix(i, i+1, u, c, true);
        for (int y = 0; y < l.cols(); ++y) {
            if (result.i[i*2 + y] == -1) {
                result.i[i*2 + y] = counter;
            }
            if ((i != 0 && y > 1) || (i == 0)) {
                for (int x = 0; x <= y; ++x) {
                    if (x == y) {
                        result.d[i*2 + x] = l(x, y);
                    } else {
                        result.j.push_back(i*2 + x);
                        result.u.push_back(l(x, y));
                        result.l.push_back(l(y, x));
                        counter++;
                    }
                }
            }
        }
    }
    result.i.back() = counter;

    return result;
}

//---------------------------------------------------------------------------
template<class V>
V calcB(
    const lin_approx_t& u,
    const Constants& c,
    const Function1D& fs,
    const Function1D& fc
) {
    V result(u.size() * 2);
    for (int i = 0; i < u.size(); ++i) {
        auto funs = [&] (double x) -> double { return fs(x) * u.basic(x, i); };
        auto func = [&] (double x) -> double { return fc(x) * u.basic(x, i); };
        vector<double> X;
        make_grid(X, u.left(i), u.right(i), count_integral);
        result(i*2 + 0) = integral_gauss3(X, funs);
        result(i*2 + 1) = integral_gauss3(X, func);
    }
    return result;
}

//---------------------------------------------------------------------------
template<class V>
void setAnswer(
    lin_approx_t& us,
    lin_approx_t& uc,
    const V& result
) {
```

```
    us.q.resize(us.size());
    uc.q.resize(uc.size());
    for (int i = 0; i < us.size(); ++i) {
        us.q[i] = result(i*2 + 0);
        uc.q[i] = result(i*2 + 1);
    }
}

//---------------------------------------------------------------------------
MatrixDiagonal calcGlobalMatrixDiag(const lin_approx_t& u, const Constants& c) {
    Diagonal d(u.size() * 2);
    vector<int> format = {0, -3, -2, -1, 1, 2, 3};
    MatrixDiagonal result(u.size() * 2, format);
    for (int i = 0; i < u.size()-1; ++i) {
        auto l = calcLocalMatrix(i, i+1, u, c, i == u.size()-2);
        for (int x = 0; x < l.rows(); ++x) {
            for (int y = 0; y < l.cols(); ++y) {
                int line = i*2 + x;
                int row = i*2 + y;
                int diag = d.calcDiag_byLR(line, row);
                int pos = d.calcPos_byLR(line, row);
                diag = distance(format.begin(), find(format.begin(), format.end(), diag));
                result.begin(diag)[pos] = l(x, y);
            }
        }
    }

    return result;
}

//---------------------------------------------------------------------------
void setFirstBoundaryConditions(
    EMatrix& A,
    EVector& b,
    const Function1D& us_true,
    const Function1D& uc_true,
    const lin_approx_t& u
) {
    auto clear_line = [&] (int line) {
        for (int i = 0; i < A.cols(); ++i) A(line, i) = 0;
    };

    clear_line(0); A(0, 0) = 1; b(0) = us_true(u.middle(0));
    clear_line(1); A(1, 1) = 1; b(1) = uc_true(u.middle(0));

    int end = A.cols()-1;
    clear_line(end-1); A(end-1, end-1) = 1; b(end-1) = us_true(u.middle(u.size()-1));
    clear_line(end);   A(end, end) = 1;     b(end) = uc_true(u.middle(u.size()-1));
}

//---------------------------------------------------------------------------
void setFirstBoundaryConditions(
    MatrixDiagonal& A,
    Vector& b,
    const Function1D& us_true,
    const Function1D& uc_true,
    const lin_approx_t& u
) {
    auto clear_line = [&] (int line) {
        matrix_diagonal_line_iterator it(A.dimension(), A.getFormat(), false);
        for (; !it.isEnd(); ++it)
            for (; !it.isLineEnd(); ++it)
                if (it.i == line)
                    A.begin(it.dn)[it.di] = 0;
    };

    clear_line(0); A.begin(0)[0] = 1; b(0) = us_true(u.middle(0));
    clear_line(1); A.begin(0)[1] = 1; b(1) = uc_true(u.middle(0));

    int end = A.dimension()-1;
    clear_line(end-1); A.begin(0)[end-1] = 1; b(end-1) = us_true(u.middle(u.size()-1));
    clear_line(end);   A.begin(0)[end] = 1;   b(end) = uc_true(u.middle(u.size()-1));
}

//---------------------------------------------------------------------------
void setFirstBoundaryConditions(
    matrix& A,
    vector<double>& b,
    const Function1D& us_true,
    const Function1D& uc_true,
    const lin_approx_t& u
) {
    auto clear_lines = [&] (vector<int> lines) {
        for (auto& i : lines) A.d[i] = 0;

        for (int i = 0; i < A.i.size()-1; i++) {
            for (int pj = A.i[i]; pj < A.i[i+1]; pj++) {
                int j = A.j[pj];
```

```
                if (find(lines.begin(), lines.end(), j) != lines.end()) A.u[pj] = 0;
                if (find(lines.begin(), lines.end(), i) != lines.end()) A.l[pj] = 0;
            }
        }
    };

    int end = A.n-1;
    clear_lines({0, 1, end-1, end});

    A.d[0] = 1; b[0] = us_true(u.middle(0));
    A.d[1] = 1; b[1] = uc_true(u.middle(0));

    A.d[end-1] = 1; b[end-1] = us_true(u.middle(u.size()-1));
    A.d[end] = 1;   b[end] = uc_true(u.middle(u.size()-1));
}

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

double spacea = 1, spaceb = 2;

//-----------------------------------------------------------------------------
struct Result1
{
    double los_integral_norm, bsg_integral_norm;
    double los_time, bsg_time;
    int los_iter, bsg_iter;
};

ostream& operator<<(ostream& out, const Result1& res) {
    out
        << res.los_integral_norm << "\t"
        << res.bsg_integral_norm << "\t"
        << res.los_time << "\t"
        << res.bsg_time << "\t"
        << res.los_iter << "\t"
        << res.bsg_iter;
    return out;
}

Result1 calcMethod(
    const Constants& c,
    Function1D us_true,
    Function1D uc_true,
    const vector<double>& grid
) {
    auto fs = calcRightPartS(us_true, uc_true, c);
    auto fc = calcRightPartC(us_true, uc_true, c);

    lin_approx_t us, uc;
    us.x = grid;
    uc.x = grid;

    auto A2 = calcGlobalMatrixProfile(us, c);
    auto b2 = to(calcB<Vector>(us, c, fs, fc));
    setFirstBoundaryConditions(A2, b2, us_true, uc_true, us);

    SLAU slau;
    slau.maxiter = 10;
    slau.eps = 1e-16;
    slau.is_log = false;
    slau.n = A2.n;
    slau.a = A2;
    slau.f = b2;
    slau.x.resize(slau.n);
    slau.t1.resize(slau.n);
    slau.t2.resize(slau.n);

    time_counter t1;
    t1.start();
    auto res_los = slau.los2();
    t1.end();
    setAnswer(us, uc, to(slau.x));
    double los_residual = norm(us_true, us) + norm(uc_true, uc);

    time_counter t2;
    t2.start();
    auto res_bsg = slau.bsg_stab_lu();
    t2.end();
    setAnswer(us, uc, to(slau.x));
    double bsg_residual = norm(us_true, us) + norm(uc_true, uc);

    return {los_residual, bsg_residual, t1.get_microseconds(), t2.get_microseconds(), res_los.first,
    ↪  res_bsg.first};
}

//-----------------------------------------------------------------------------
struct grid_point { double t, los_norm, bsg_norm; };
```

```
vector<grid_point> calcGrid(
    const Constants& c,
    int elemCount,
    Function1D us_true,
    Function1D uc_true,
    function<Function1D(double)> moveMaker
) {
    int n = 1000;
    vector<grid_point> result;
    for (double t = 1.0/n; t < 1.0 + 1.0/n; t += 1.0/n) {
        vector<double> grid;
        make_grid(grid, spacea, spaceb, elemCount, moveMaker(t));

        auto res = calcMethod(c, us_true, uc_true, grid);

        result.push_back({t, res.los_integral_norm, res.bsg_integral_norm});
    }

    return result;
}

//-----------------------------------------------------------------------------
void writeParametersInvestigation(
    Function1D us_true,
    Function1D uc_true,
    int elemCount,
    const string& filename
) {

    vector<double> grid;
    make_grid(grid, spacea, spaceb, elemCount);

    // omega: 10^-4, 10^9
    // lambda: 10^2, 8*10^6
    // sigma: 0, 10^8
    // xi: 8.81*10^-12, 10^-10

    vector<double> omega = {1e-4, 1e-3, 1e-2, 1e-1, 0, 1, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9};
    vector<double> lambda = {1e2, 1e3, 1e4, 1e5, 1e6, 8e6};
    vector<double> sigma = {0, 1, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8};
    vector<double> xi = {8.81e-12, 1e-12, 1e-11, 1e-10};

    Constants c;
    c.omega = 1;
    c.lambda = 1;
    c.sigma = 1;
    c.xi = 1e-11;

    #define write_parameter(par) {\
        ofstream fout(filename + "_" + #par + ".txt"); \
        fout << "param\tlos_norm\tbsg_norm\tlos_time\tbsg_time\tlos_iter\tbsg_iter" << endl; \
        fout << scientific << setprecision(2); \
        for (auto& i : par) { \
            c.par = i; \
            auto res = calcMethod(c, us_true, uc_true, grid); \
            fout  << "
" << #par << " = " << writeforlatexdouble(i,2) << "" << "\t"  << res << endl; \
        } c.par = 1; \
        fout.close(); }\

    write_parameter(omega);
    write_parameter(lambda);
    write_parameter(sigma);
    write_parameter(xi);
}

//-----------------------------------------------------------------------------
void writeGridInvestigation(
    Function1D us_true,
    Function1D uc_true,
    int elemCount,
    const string& filename) {
    ofstream fout(filename);

    fout << "t\tnorma\tnormb\tnorm" << endl;

    Constants c;
    c.lambda = 1;
    c.omega = 1;
    c.xi = 1;
    c.sigma = 1;

    using namespace placeholders;
    auto grid0 = calcGrid(c, elemCount, us_true, uc_true, bind(getMove0, _1, 1));
    auto grid1 = calcGrid(c, elemCount, us_true, uc_true, bind(getMove1, _1, 1));

    for (int i = 0; i < grid0.size(); ++i) {
        fout
```

```
            << grid0[i].t << "\t"
            << grid0[i].bsg_norm << "\t"
            << grid1[i].bsg_norm << "\t"
            << grid0[grid0.size()-2].bsg_norm << endl;
    }

    fout.close();
}

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

int main() {
    // cout.precision(3);

    // Constants c;
    // c.lambda = 32;
    // c.omega = 100;
    // c.xi = 10;
    // c.sigma = 24;

    // auto us_true = [] (double x) -> double { return 3*x*x*x*x + 2*exp(x); };
    // auto uc_true = [] (double x) -> double { return 6*x - pow(x, exp(x)); };

    // auto fs = calcRightPartS(us_true, uc_true, c);
    // auto fc = calcRightPartC(us_true, uc_true, c);

    // lin_approx_t us, uc;
    // make_grid(us.x, 1, 2, 45000);
    // uc.x = us.x;

    // /*auto A = calcGlobalMatrix(us, c);
    // auto b = calcB<EVector>(us, c, fs, fc);
    // setFirstBoundaryConditions(A, b, us_true, uc_true, us);

    // Eigen::JacobiSVD<EMatrix> svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV);
    // EVector x = svd.solve(b);

    // setAnswer(us, uc, x);*/

    // double residual;

    // auto A2 = calcGlobalMatrixProfile(us, c);
    // auto b2 = to(calcB<Vector>(us, c, fs, fc));
    // setFirstBoundaryConditions(A2, b2, us_true, uc_true, us);

    // SLAU slau;
    // slau.maxiter = 10;
    // slau.eps = 1e-16;
    // slau.is_log = true;
    // slau.n = A2.n;
    // slau.a = A2;
    // slau.f = b2;
    // slau.x.resize(slau.n);
    // slau.t1.resize(slau.n);
    // slau.t2.resize(slau.n);

    // time_counter t1;
    // t1.start();
    // slau.los2();
    // t1.end();
    // setAnswer(us, uc, to(slau.x));
    // residual = norm(us_true, us) + norm(uc_true, uc);
    // cout << "los  2 residual: " << residual << endl << endl;
    // cout << "los  2 time:     " << t1.get_milliseconds() << endl;

    // time_counter t2;
    // t2.start();
    // slau.bsg_stab_lu();
    // t2.end();
    // setAnswer(us, uc, to(slau.x));
    // residual = norm(us_true, us) + norm(uc_true, uc);
    // cout << "bsg lu residual: " << residual << endl << endl;
    // cout << "bsg lu time:     " << t2.get_milliseconds() << endl;

    // /*auto A1 = calcGlobalMatrixDiag(us, c);
    // auto b1 = calcB<Vector>(us, c, fs, fc);
    // setFirstBoundaryConditions(A1, b1, us_true, uc_true, us);*/
    // /*Matrix denseA(A.cols(), A.rows());
    // for (int i = 0; i < A.cols(); i++) {
    //     for (int j = 0; j < A.rows(); j++) {
    //         denseA(i, j) = A(i, j);
    //     }
    // }
    // MatrixDiagonal A2(denseA);*/

    // /*Vector x1;
```

```
    // SolverSLAE_Iterative solver;
    // solver.epsilon = 1e-7;
    // solver.isLog = false;
    // solver.maxIterations = 5000;
    // solver.start = Vector(us.size() * 2, 0);
    // solver.w = 0.8;
    // solver.seidel(A1, b1, x1);

    // setAnswer(us, uc, x1);

    // Vector b2;
    // mul(A1, x1, b2);
    // b2.negate();
    // sum(b1, b2, b2);
    // cout << "residual slae: " << calcNorm(b2) << endl;*/

    // /*Matrix dense; A1.toDenseMatrix(dense);
    // cout << A << endl << b << endl;
    // dense.save(cout); cout << endl;
    // b1.save(cout); cout << endl;
    // cout << x << endl;
    // x1.save(cout); cout << endl;*/

    // /*Matrix dense; A2.toDense(dense);
    // cout << A << endl;
    // dense.save(cout); cout << endl;*/

    // //double residual = norm(us_true, us) + norm(uc_true, uc);

    // /*cout << "answer s:    " << us.q << endl;
    // cout << "should be s: " << calcTrulyApprox(us.x, us_true).q << endl;

    // cout << "answer s:    " << uc.q << endl;
    // cout << "should be s: " << calcTrulyApprox(uc.x, uc_true).q << endl;*/

    // //cout << "residual: " << residual << endl;

    // system("pause");

    vector<thread> t;

    t.emplace_back([](){
    writeParametersInvestigation(
        [] (double x) -> double { return 3.0*x; },
        [] (double x) -> double { return -10.0*x; },
        100,
        "3_parameters_100"
    );
    });

    t.emplace_back([](){
    writeParametersInvestigation(
        [] (double x) -> double { return 3.0*x; },
        [] (double x) -> double { return -10.0*x; },
        50000,
        "3_parameters_50000"
    );
    });

    t.emplace_back([](){
    writeGridInvestigation(
        [] (double x) -> double { return exp(x/2); },
        [] (double x) -> double { return 5*exp(x-3); },
        50,
        "3_grid_exp.txt"
    );
    });

    t.emplace_back([](){
    writeGridInvestigation(
        [] (double x) -> double { return exp(-x/2); },
        [] (double x) -> double { return 5*exp(-x+3); },
        50,
        "3_grid_exp_minus.txt"
    );
    });

    t.emplace_back([](){
    writeGridInvestigation(
        [] (double x) -> double { return x*x; },
        [] (double x) -> double { return 3.0*x*(x+2)-5; },
        50,
        "3_grid_x2.txt"
    );
    });

    t.emplace_back([](){
    writeGridInvestigation(
```

```
        [] (double x) -> double { return 3*x*x*x*x + 2*exp(x); },
        [] (double x) -> double { return 6*x - pow(x, exp(x)); },
        50,
        "3_grid_uuu.txt"
    );
    });

    for (auto& i : t) i.join();
}
```