Московский авиационный институт (национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторные работы по курсу «Численные методы»

Студент: М. А. Инютин
Преподаватель: Д. Л. Ревизников
Группа: М8О-407Б-19

Дата:
Оценка:
Подпись:

Москва, 2022

1 Численные методы решения дифференциальных уравнений с частными производными

1 Численные методы решения ДУЧП параболического типа

1.1 Постановка задачи

Используя явную и неявную конечно-разностные схемы, а также схему Кранка-Николсона, решить начально-краевую задачу для дифференциального уравнения параболического типа. Осуществить реализацию трёх вариантов аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трёхточечная аппроксимация со вторым порядком, двухточечная аппроксимация со вторым порядком. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением $U(x,t)$. Исследовать зависимость погрешности от сеточных параметров τ, h.

1.2 Вариант 10

$$\frac{\partial u}{\partial t} = a \cdot \frac{\partial^2 u}{\partial x^2} + b \cdot \frac{\partial u}{\partial y} + c \cdot u$$

$$a > 0, b > 0, c < 0$$

$$u'_x(0,t) + u(0,t) = e^{(c-a)t} \cdot (\cos(bt) + \sin(bt))$$

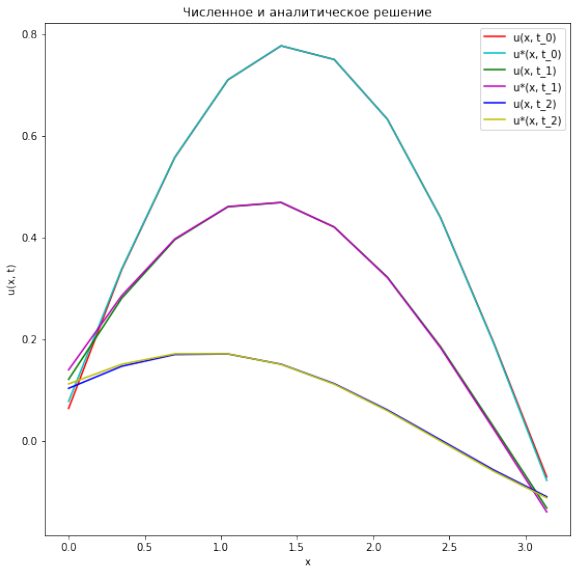
$$u'_x(\pi,t) + u(\pi,t) = -e^{(c-a)t} \cdot (\cos(bt) + \sin(bt))$$

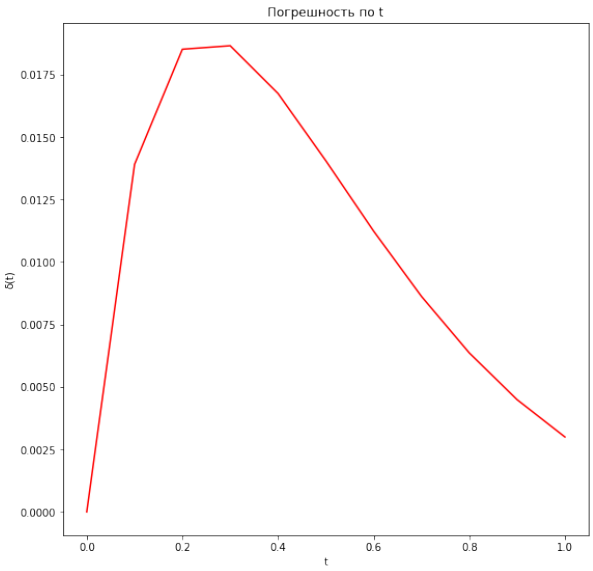
$$u(x,0) = \sin x$$

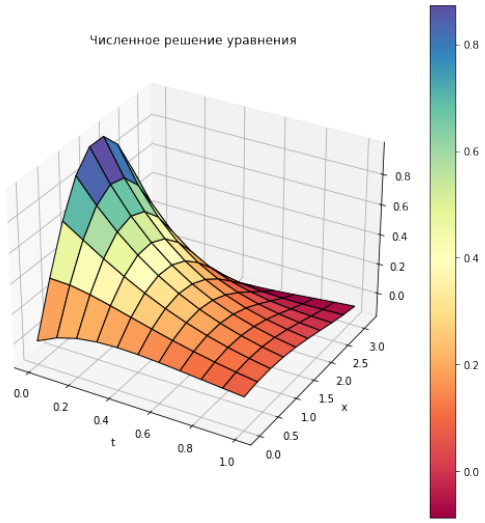
Аналитическое решение:

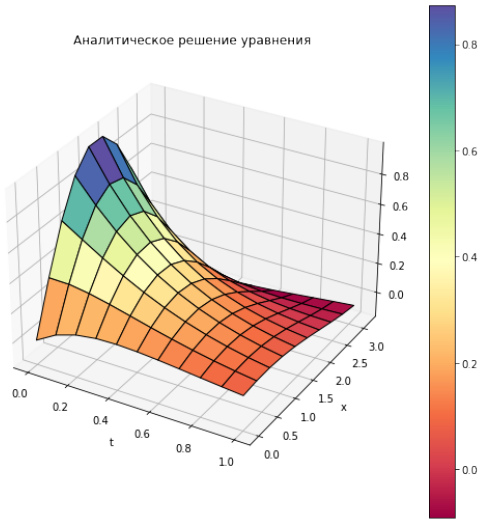
$$U(x,t) = e^{(c-a)t} \cdot \sin(x+bt)$$

1.3 Результат









1.4 Исходный код

```
#ifndef PPDE_HPP
#define PPDE_HPP

#include "../lab1_2/tridiag.hpp"

/* Parabolic Partial Differential Equation Solver */
class ppde_t {
    const double EPS = 1e-9;

    int n, K, boundary;
    double l, T, h, tau, theta;
    double a, b, c;
    double alpha_0, beta_0, alpha_l, beta_l;

    double h2;

    using vec = std::vector<double>;
    vec gamma_0, gamma_l;
    vec uk, uk1;

    using tridiag = tridiag_t<double>;
    vec trd_a, trd_b, trd_c, trd_d;

    void gen_explicit(int i, double& uik, double& dudx, double& d2udx2) {
        uik = uk[i];
        dudx = (uk[i + 1] - uk[i - 1]) / (2 * h);
        d2udx2 = (uk[i - 1] - 2 * uik + uk[i + 1]) / h2;
    }

    vec c_3t2p_0, c_3t2p_l;

    void prepare_3t2p() {
        c_3t2p_0.resize(4, 0);
        c_3t2p_0[0] = 2 * h * beta_0 - 3 * alpha_0;
        c_3t2p_0[1] = 4 * alpha_0;
        c_3t2p_0[2] = -alpha_0;
        c_3t2p_0[3] = 2 * h;

        c_3t2p_l.resize(4, 0);
        c_3t2p_l[0] = alpha_l;
        c_3t2p_l[1] = -4 * alpha_l;
        c_3t2p_l[2] = 2 * h * beta_l + 3 * alpha_l;
        c_3t2p_l[3] = 2 * h;
    }

    vec c_2t2p_0, c_2t2p_l;

```

```
48      void prepare_2t2p() {
49          c_2t2p_0.resize(4, 0);
50          c_2t2p_0[0] =
51              alpha_0 * (1 / tau + 2 * a / h2 - c) + beta_0 * (b - 2 * a / h);
52          c_2t2p_0[1] = -2 * alpha_0 * a / h2;
53          c_2t2p_0[2] = b - 2 * a / h;
54          c_2t2p_0[3] = alpha_0 / tau;
55
56          c_2t2p_l.resize(4, 0);
57          c_2t2p_l[0] = -2 * alpha_l * a / h2;
58          c_2t2p_l[1] =
59              alpha_l * (1 / tau + 2 * a / h2 - c) + beta_l * (b + 2 * a / h);
60          c_2t2p_l[2] = b + 2 * a / h;
61          c_2t2p_l[3] = alpha_l / tau;
62      }
63
64      void prepare_trd() {
65          trd_a.resize(n, 0);
66          trd_b.resize(n, 0);
67          trd_c.resize(n, 0);
68          trd_d.resize(n, 0);
69      }
70
71      void gen_boundary_0_2t1p(int k) {
72          trd_b[0] = beta_0 - alpha_0 / h;
73          trd_c[0] = alpha_0 / h;
74          trd_d[0] = gamma_0[k + 1];
75      }
76
77      void gen_boundary_0_3t2p(int k) {
78          double coef_row = c_3t2p_0[2] / trd_c[1];
79          trd_b[0] = c_3t2p_0[0] - trd_a[1] * coef_row;
80          trd_c[0] = c_3t2p_0[1] - trd_b[1] * coef_row;
81          trd_d[0] = c_3t2p_0[3] * gamma_0[k + 1] - trd_d[1] * coef_row;
82      }
83
84      void gen_boundary_0_2t2p(int k) {
85          trd_b[0] = c_2t2p_0[0];
86          trd_c[0] = c_2t2p_0[1];
87          trd_d[0] = c_2t2p_0[2] * gamma_0[k + 1] + c_2t2p_0[3] * uk[0];
88      }
89
90      void gen_boundary_0(int k) {
91          if (boundary == 1) {
92              gen_boundary_0_2t1p(k);
93          } else if (boundary == 2) {
94              gen_boundary_0_3t2p(k);
95          } else {
96              gen_boundary_0_2t2p(k);
```

```
97          }
98      }
99
100     void gen_boundary_l_2t1p(int k) {
101         trd_a.back() = -alpha_l / h;
102         trd_b.back() = alpha_l / h + beta_l;
103         trd_d.back() = gamma_l[k + 1];
104     }
105
106     void gen_boundary_l_3t2p(int k) {
107         double coef_row = c_3t2p_l[0] / trd_a[n - 2];
108         trd_a.back() = c_3t2p_l[1] - trd_b[n - 2] * coef_row;
109         trd_b.back() = c_3t2p_l[2] - trd_c[n - 2] * coef_row;
110         trd_d.back() = c_3t2p_l[3] * gamma_l[k + 1] - trd_d[n - 2] * coef_row;
111     }
112
113     void gen_boundary_l_2t2p(int k) {
114         trd_a.back() = c_2t2p_l[0];
115         trd_b.back() = c_2t2p_l[1];
116         trd_d.back() = c_2t2p_l[2] * gamma_l[k + 1] + c_2t2p_l[3] * uk[n - 1];
117     }
118
119     void gen_boundary_l(int k) {
120         if (boundary == 1) {
121             gen_boundary_l_2t1p(k);
122         } else if (boundary == 2) {
123             gen_boundary_l_3t2p(k);
124         } else {
125             gen_boundary_l_2t2p(k);
126         }
127     }
128
129     void gen_implicit(int k, bool mode_combo) {
130         prepare_trd();
131         for (int i = 1; i < n - 1; ++i) {
132             trd_a[i] = (theta * tau * (b / (2 * h) - a / h2));
133             trd_b[i] = (theta * tau * ((2 * a) / h2 - c)) + 1;
134             trd_c[i] = (theta * tau * -(a / h2 + b / (2 * h)));
135             if (mode_combo) {
136                 double uik, dudx, d2udx2;
137                 gen_explicit(i, uik, dudx, d2udx2);
138                 double rhs = a * d2udx2 + b * dudx + c * uik;
139                 trd_d[i] = uik + (1 - theta) * tau * rhs;
140             } else {
141                 trd_d[i] = uk[i];
142             }
143         }
144         gen_boundary_0(k);
145         gen_boundary_l(k);
```

```
146     }
147
148     void boundary_explicit_2t1p(int k) {
149         uk1[0] =
150             (gamma_0[k + 1] - alpha_0 * uk1[1] / h) / (beta_0 - alpha_0 / h);
151         uk1[n - 1] = (gamma_l[k + 1] + alpha_l * uk1[n - 2] / h) /
152                      (alpha_l / h + beta_l);
153     }
154
155     void boundary_explicit_3t2p(int k) {
156         double rhs0 = c_3t2p_0[3] * gamma_0[k + 1] - c_3t2p_0[2] * uk1[2] -
157                       c_3t2p_0[1] * uk1[1];
158         double lhs0 = c_3t2p_0[0];
159         uk1[0] = rhs0 / lhs0;
160         double rhsl = c_3t2p_l[3] * gamma_l[k + 1] - c_3t2p_l[0] * uk1[n - 3] -
161                       c_3t2p_l[1] * uk1[n - 2];
162         double lhsl = c_3t2p_l[2];
163         uk1[n - 1] = rhsl / lhsl;
164     }
165
166     void boundary_explicit_2t2p(int k) {
167         double rhs0 = c_2t2p_0[2] * gamma_0[k + 1] + c_2t2p_0[3] * uk[0] -
168                       c_2t2p_0[1] * uk1[1];
169         double lhs0 = c_2t2p_0[0];
170         uk1[0] = rhs0 / lhs0;
171         double rhsl = c_2t2p_l[2] * gamma_l[k + 1] + c_2t2p_l[3] * uk[n - 1] -
172                       c_2t2p_l[0] * uk1[n - 2];
173         double lhsl = c_2t2p_l[1];
174         uk1[n - 1] = rhsl / lhsl;
175     }
176
177     void boundary_explicit(int k) {
178         if (boundary == 1) {
179             boundary_explicit_2t1p(k);
180         } else if (boundary == 2) {
181             boundary_explicit_3t2p(k);
182         } else {
183             boundary_explicit_2t2p(k);
184         }
185     }
186
187     static void print_vec(std::ostream& out, const vec& v) {
188         size_t n = v.size();
189         for (size_t i = 0; i < n; ++i) {
190             if (i) {
191                 out << ' ';
192             }
193             out << v[i];
194         }
```

```
195         out << '\n';
196     }
197 
198     public:
199     friend std::istream& operator>>(std::istream& in, ppde_t& item) {
200         in >> item.n >> item.K >> item.boundary;
201         in >> item.l >> item.T >> item.h >> item.tau >> item.theta;
202         if (item.theta < 0 or item.theta > 1) {
203             throw std::invalid_argument("Theta is invalid!");
204         }
205         item.h2 = item.h * item.h;
206         in >> item.a >> item.b >> item.c;
207         in >> item.alpha_0 >> item.beta_0 >> item.alpha_l >> item.beta_l;
208         item.uk.resize(item.n);
209         item.gamma_0.resize(item.K);
210         item.gamma_l.resize(item.K);
211         for (int i = 0; i < item.n; ++i) {
212             in >> item.uk[i];
213         }
214         for (int k = 0; k < item.K; ++k) {
215             in >> item.gamma_0[k];
216         }
217         for (int k = 0; k < item.K; ++k) {
218             in >> item.gamma_l[k];
219         }
220         return in;
221     }
222 
223     void solve(std::ostream& out) {
224         prepare_3t2p();
225         prepare_2t2p();
226         if (theta < EPS) {
227             solve_explicit(out);
228         } else if (theta < 1) {
229             solve_combo(out);
230         } else {
231             solve_implicit(out);
232         }
233     }
234 
235     void solve_implicit(std::ostream& out) {
236         print_vec(out, uk);
237         for (int k = 0; k < K - 1; ++k) {
238             gen_implicit(k, false);
239             tridiag trd(trd_a, trd_b, trd_c);
240             uk1 = trd.solve(trd_d);
241             print_vec(out, uk1);
242             swap(uk1, uk);
243         }
```

```
244     }
245
246     void solve_combo(std::ostream& out) {
247         print_vec(out, uk);
248         for (int k = 0; k < K - 1; ++k) {
249             gen_implicit(k, true);
250             tridiag trd(trd_a, trd_b, trd_c);
251             uk1 = trd.solve(trd_d);
252             print_vec(out, uk1);
253             swap(uk1, uk);
254         }
255     }
256
257     void solve_explicit(std::ostream& out) {
258         print_vec(out, uk);
259         for (int k = 0; k < K - 1; ++k) {
260             uk1.assign(n, 0);
261             for (int i = 1; i < n - 1; ++i) {
262                 double uik, dudx, d2udx2;
263                 gen_explicit(i, uik, dudx, d2udx2);
264                 double rhs = a * d2udx2 + b * dudx + c * uik;
265                 uk1[i] = uik + tau * rhs;
266             }
267             boundary_explicit(k);
268             print_vec(out, uk1);
269             swap(uk1, uk);
270         }
271     }
272 };
273
274 #endif /* PPDE_HPP */
```

2 Численные методы решения ДУЧП гиперболического типа

2.1 Постановка задачи

Используя явную схему крест и неявную схему, решить начально-краевую задачу для дифференциального уравнения гиперболического типа. Аппроксимацию второго начального условия произвести с первым и со вторым порядком. Осуществить реализацию трех вариантов аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком, двухточечная аппроксимация со вторым порядком. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением $U(x,t)$. Исследовать зависимость погрешности от сеточных параметров τ, h.

2.2 Вариант 9

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + 2 \cdot \frac{\partial u}{\partial y} - 3 \cdot u - 2 \cdot \frac{\partial u}{\partial t}$$

$$u(0,t) = e^{-t} \cdot \cos(2t)$$

$$u(\frac{\pi}{2}, t) = 0$$

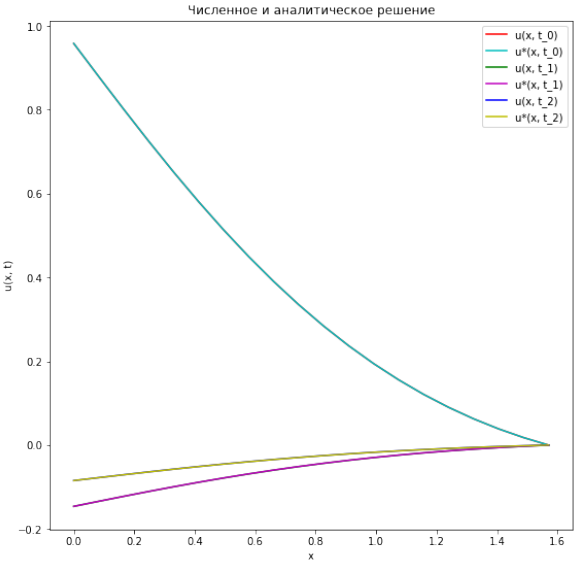
$$u(x,0) = e^{-x} \cdot \cos x$$

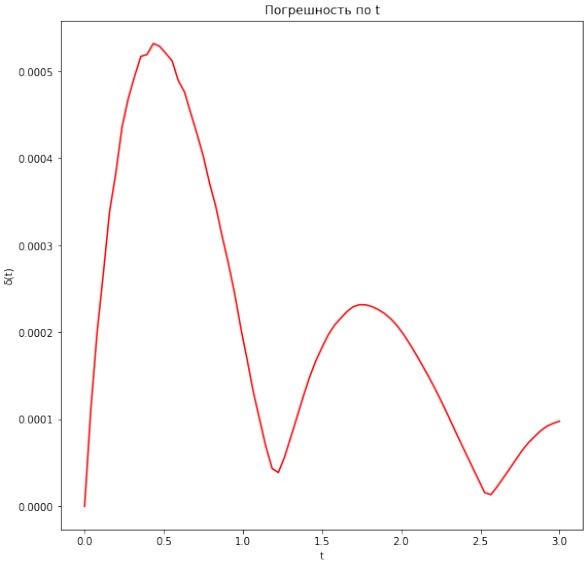
$$u'_t(x,0) = -e^{-x} \cdot \cos x$$

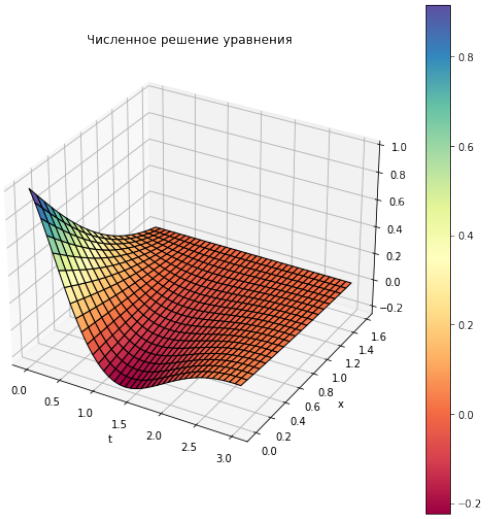
Аналитическое решение:

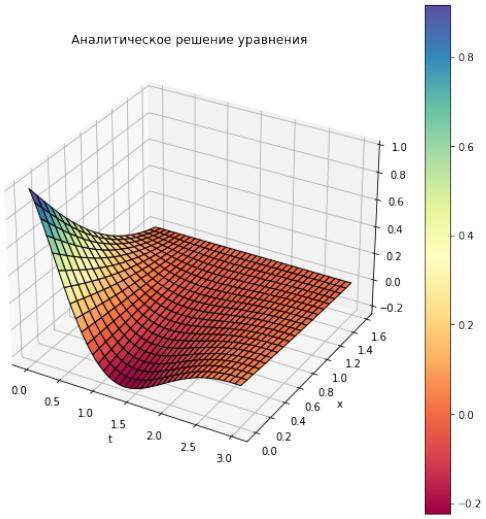
$$U(x,t) = e^{-t-x} \cdot \cos x \cdot \cos(2t)$$

2.3 Результат









2.4 Исходный код

```
#ifndef HPDE_HPP
#define HPDE_HPP

#include "../lab1_2/tridiag.hpp"

/* Hyperbolic Partial Differential Equation Solver */
class hpde_t {
    const double EPS = 1e-9;

    using vec = std::vector<double>;
    using vecvec = std::vector<vec>;

    int n, K, u1_degree, boundary;
    double l, T, h, tau;
    int theta;

    double a, b, c, d;
    vecvec f;

    /*
     * u_prev = u_{k-1}
     * u_k = u_k
     * u_next = u_{k + 1}
     */
    vec u_prev, u_k, u_next;

    double alpha_0, beta_0, alpha_l, beta_l;
    vec gamma_0, gamma_l;

    vec ddx_psi1, d2dx2_psi1;

    double h2, tau2;

    void prepare_u1_1() {
        for (int i = 0; i < n; ++i) {
            u_k[i] = u_prev[i] + u_k[i] * tau;
        }
    }

    void prepare_u1_2() {
        for (int i = 0; i < n; ++i) {
            u_k[i] = u_prev[i] +
                     tau2 / 2 *
                         (a * d2dx2_psi1[i] + b * ddx_psi1[i] + c * u_prev[i] +
                          f[i][0]) +
                     u_k[i] * (tau + d * tau2 / 2);
        }
```

```
48      }
49
50      void prepare_u1() {
51          if (u1_degree == 1) {
52              prepare_u1_1();
53          } else {
54              prepare_u1_2();
55          }
56      }
57
58      vec c_expl;
59      const int EXPLICIT_COEFS = 6;
60
61      void precalc_explicit() {
62          c_expl.resize(EXPLICIT_COEFS);
63          c_expl[0] = h2 * (2 - d * tau);
64          c_expl[1] = tau2 * (2 * a - h * b);
65          c_expl[2] = 4 * h2 + tau2 * (-4 * a + 2 * h2 * c);
66          c_expl[3] = tau2 * (2 * a + b * h);
67          c_expl[4] = h2 * (-2 - d * tau);
68          c_expl[5] = 2 * h2 * tau2;
69      }
70
71      vec c_impl;
72      const int IMPLICIT_COEFS = 6;
73
74      void precalc_implicit() {
75          c_impl.resize(IMPLICIT_COEFS);
76          c_impl[0] = tau2 * (-2 * a + b * h);
77          c_impl[1] = 2 * h2 + tau2 * (4 * a - 2 * h2 * c) - d * h2 * tau;
78          c_impl[2] = tau2 * (-2 * a - b * h);
79          c_impl[3] = 4 * h2;
80          c_impl[4] = h2 * (-2 - d * tau);
81          c_impl[5] = 2 * h2 * tau2;
82      }
83
84      vec c_3t2p_0, c_3t2p_l;
85      const int COEFS_3T2P = 4;
86
87      void prepare_3t2p() {
88          c_3t2p_0.resize(COEFS_3T2P);
89          c_3t2p_0[0] = 2 * h * beta_0 - 3 * alpha_0;
90          c_3t2p_0[1] = 4 * alpha_0;
91          c_3t2p_0[2] = -alpha_0;
92          c_3t2p_0[3] = 2 * h;
93
94          c_3t2p_l.resize(COEFS_3T2P);
95          c_3t2p_l[0] = alpha_l;
96          c_3t2p_l[1] = -4 * alpha_l;
```

```
97          c_3t2p_l[2] = 2 * h * beta_l + 3 * alpha_l;
98          c_3t2p_l[3] = 2 * h;
99      }
100
101     vec c_2t2p_0, c_2t2p_l;
102     const int COEFS_2T2P = 6;
103
104     void prepare_2t2p() {
105         c_2t2p_0.resize(COEFS_2T2P);
106         c_2t2p_0[0] = -alpha_0 * a -
107                       alpha_0 * h2 / 2 * (1 / tau2 - c - d / (2 * tau)) +
108                       beta_0 * (a * h - b * h2 / 2);
109         c_2t2p_0[1] = alpha_0 * a;
110         c_2t2p_0[2] = alpha_0 * h2 / tau2;
111         c_2t2p_0[3] = -alpha_0 * h2 / (2 * tau2) + alpha_0 * h2 * d / (4 * tau);
112         c_2t2p_0[4] = a * h - b * h2 / 2;
113         c_2t2p_0[5] = alpha_0 * h2 / 2;
114
115         c_2t2p_l.resize(COEFS_2T2P);
116         c_2t2p_l[0] = -alpha_l * a;
117         c_2t2p_l[1] = alpha_l * a +
118                       alpha_l * h2 / 2 * (1 / tau2 - c - d / (2 * tau)) +
119                       beta_l * (a * h + b * h2 / 2);
120         c_2t2p_l[2] = -alpha_l * h2 / tau2;
121         c_2t2p_l[3] = alpha_l * h2 / (2 * tau2) + alpha_l * h2 * d / (4 * tau);
122         c_2t2p_l[4] = a * h + b * h2 / 2;
123         c_2t2p_l[5] = -alpha_l * h2 / 2;
124     }
125
126     using tridiag = tridiag_t<double>;
127     vec trd_a, trd_b, trd_c, trd_d;
128
129     void prepare_trd() {
130         trd_a.resize(n, 0);
131         trd_b.resize(n, 0);
132         trd_c.resize(n, 0);
133         trd_d.resize(n, 0);
134     }
135
136     void gen_boundary_0_2t1p(int k) {
137         trd_b[0] = beta_0 - alpha_0 / h;
138         trd_c[0] = alpha_0 / h;
139         trd_d[0] = gamma_0[k + 1];
140     }
141
142     void gen_boundary_0_3t2p(int k) {
143         double coef_row = c_3t2p_0[2] / trd_c[1];
144         trd_b[0] = c_3t2p_0[0] - trd_a[1] * coef_row;
145         trd_c[0] = c_3t2p_0[1] - trd_b[1] * coef_row;
```

```
146         trd_d[0] = c_3t2p_0[3] * gamma_0[k + 1] - trd_d[1] * coef_row;
147     }
148
149     void gen_boundary_0_2t2p(int k) {
150         trd_b[0] = c_2t2p_0[0];
151         trd_c[0] = c_2t2p_0[1];
152         trd_d[0] = c_2t2p_0[4] * gamma_0[k + 1] - c_2t2p_0[5] * f[0][k + 1] -
153                    c_2t2p_0[2] * u_k[0] - c_2t2p_0[3] * u_prev[0];
154     }
155
156     void gen_boundary_0(int k) {
157         if (boundary == 1) {
158             gen_boundary_0_2t1p(k);
159         } else if (boundary == 2) {
160             gen_boundary_0_3t2p(k);
161         } else {
162             gen_boundary_0_2t2p(k);
163         }
164     }
165
166     void gen_boundary_l_2t1p(int k) {
167         trd_a.back() = -alpha_l / h;
168         trd_b.back() = alpha_l / h + beta_l;
169         trd_d.back() = gamma_l[k + 1];
170     }
171
172     void gen_boundary_l_3t2p(int k) {
173         double coef_row = c_3t2p_l[0] / trd_a[n - 2];
174         trd_a.back() = c_3t2p_l[1] - trd_b[n - 2] * coef_row;
175         trd_b.back() = c_3t2p_l[2] - trd_c[n - 2] * coef_row;
176         trd_d.back() = c_3t2p_l[3] * gamma_l[k + 1] - trd_d[n - 2] * coef_row;
177     }
178
179     void gen_boundary_l_2t2p(int k) {
180         trd_a.back() = c_2t2p_l[0];
181         trd_b.back() = c_2t2p_l[1];
182         trd_d.back() = c_2t2p_0[4] * gamma_l[k + 1] -
183                        c_2t2p_0[5] * f[n - 1][k + 1] -
184                        c_2t2p_0[2] * u_k[n - 1] + c_2t2p_0[3] * u_prev[n - 1];
185     }
186
187     void gen_boundary_l(int k) {
188         if (boundary == 1) {
189             gen_boundary_l_2t1p(k);
190         } else if (boundary == 2) {
191             gen_boundary_l_3t2p(k);
192         } else {
193             gen_boundary_l_2t2p(k);
194         }
```

```
195     }
196
197     void gen_implicit(int k) {
198         prepare_trd();
199         for (int i = 1; i < n - 1; ++i) {
200             trd_a[i] = c_impl[0];
201             trd_b[i] = c_impl[1];
202             trd_c[i] = c_impl[2];
203             trd_d[i] = c_impl[3] * u_k[i] + c_impl[4] * u_prev[i] +
204                        c_impl[5] * f[i][k + 1];
205         }
206         gen_boundary_0(k);
207         gen_boundary_l(k);
208     }
209
210     void boundary_explicit_2t1p(int k) {
211         u_next[0] =
212             (gamma_0[k + 1] - alpha_0 * u_next[1] / h) / (beta_0 - alpha_0 / h);
213         u_next[n - 1] = (gamma_l[k + 1] + alpha_l * u_next[n - 2] / h) /
214                         (alpha_l / h + beta_l);
215     }
216
217     void boundary_explicit_3t2p(int k) {
218         double rhs0 = c_3t2p_0[3] * gamma_0[k + 1] - c_3t2p_0[2] * u_next[2] -
219                       c_3t2p_0[1] * u_next[1];
220         double lhs0 = c_3t2p_0[0];
221         u_next[0] = rhs0 / lhs0;
222         double rhsl = c_3t2p_l[3] * gamma_l[k + 1] -
223                       c_3t2p_l[0] * u_next[n - 3] - c_3t2p_l[1] * u_next[n - 2];
224         double lhsl = c_3t2p_l[2];
225         u_next[n - 1] = rhsl / lhsl;
226     }
227
228     void boundary_explicit_2t2p(int k) {
229         double rhs0 = c_2t2p_0[4] * gamma_0[k + 1] - c_2t2p_0[5] * f[0][k + 1] -
230                       c_2t2p_0[2] * u_k[0] - c_2t2p_0[3] * u_prev[0] -
231                       c_2t2p_0[1] * u_next[1];
232         double lhs0 = c_2t2p_0[0];
233         u_next[0] = rhs0 / lhs0;
234         double rhsl = c_2t2p_l[4] * gamma_l[k + 1] -
235                       c_2t2p_l[5] * f[n - 1][k + 1] - c_2t2p_l[2] * u_k[n - 1] -
236                       c_2t2p_l[3] * u_prev[n - 1] - c_2t2p_l[0] * u_next[n - 2];
237         double lhsl = c_2t2p_l[1];
238         u_next[n - 1] = rhsl / lhsl;
239     }
240
241     void boundary_explicit(int k) {
242         if (boundary == 1) {
243             boundary_explicit_2t1p(k);
```

```
244         } else if (boundary == 2) {
245             boundary_explicit_3t2p(k);
246         } else {
247             boundary_explicit_2t2p(k);
248         }
249     }
250 
251     static void print_vec(std::ostream &out, const vec &v) {
252         size_t n = v.size();
253         for (size_t i = 0; i < n; ++i) {
254             if (i) {
255                 out << ' ';
256             }
257             out << v[i];
258         }
259         out << '\n';
260     }
261 
262     public:
263     friend std::istream &operator>>(std::istream &in, hpde_t &item) {
264         in >> item.n >> item.K >> item.u1_degree >> item.boundary;
265         in >> item.l >> item.T >> item.h >> item.tau >> item.theta;
266         item.h2 = item.h * item.h;
267         item.tau2 = item.tau * item.tau;
268         in >> item.a >> item.b >> item.c >> item.d;
269         if (item.a < 0) {
270             throw std::invalid_argument("a is invalid!");
271         }
272         item.f.resize(item.n, vec(item.K));
273         for (int i = 0; i < item.n; ++i) {
274             for (int j = 0; j < item.K; ++j) {
275                 in >> item.f[i][j];
276             }
277         }
278         item.u_prev.resize(item.n);
279         for (int i = 0; i < item.n; ++i) {
280             in >> item.u_prev[i];
281         }
282         item.u_k.resize(item.n);
283         for (int i = 0; i < item.n; ++i) {
284             in >> item.u_k[i];
285         }
286 
287         in >> item.alpha_0 >> item.beta_0 >> item.alpha_l >> item.beta_l;
288         item.gamma_0.resize(item.K);
289         item.gamma_l.resize(item.K);
290         for (int k = 0; k < item.K; ++k) {
291             in >> item.gamma_0[k];
292         }
```

```
293         for (int k = 0; k < item.K; ++k) {
294             in >> item.gamma_l[k];
295         }
296         if (item.u1_degree == 2) {
297             item.ddx_psi1.resize(item.n);
298             item.d2dx2_psi1.resize(item.n);
299             for (int i = 0; i < item.n; ++i) {
300                 in >> item.ddx_psi1[i];
301             }
302             for (int i = 0; i < item.n; ++i) {
303                 in >> item.d2dx2_psi1[i];
304             }
305         }
306         return in;
307     }
308 
309     void solve(std::ostream &out) {
310         print_vec(out, u_prev);
311         prepare_u1();
312         prepare_3t2p();
313         prepare_2t2p();
314         if (theta == 0) {
315             solve_explicit(out);
316         } else {
317             solve_implicit(out);
318         }
319     }
320 
321     void solve_implicit(std::ostream &out) {
322         precalc_implicit();
323         print_vec(out, u_k);
324         for (int k = 1; k < K - 1; ++k) {
325             gen_implicit(k);
326             tridiag trd(trd_a, trd_b, trd_c);
327             u_next = trd.solve(trd_d);
328             print_vec(out, u_next);
329             u_prev = u_k;
330             u_k = u_next;
331         }
332     }
333 
334     void solve_explicit(std::ostream &out) {
335         precalc_explicit();
336         print_vec(out, u_k);
337         for (int k = 1; k < K - 1; ++k) {
338             u_next.assign(n, 0);
339             for (int i = 1; i < n - 1; ++i) {
340                 double rhs = c_expl[1] * u_k[i - 1] + c_expl[2] * u_k[i] +
341                              c_expl[3] * u_k[i + 1] + c_expl[4] * u_prev[i] +
```

```
342                             c_expl[5] * f[i][k];
343                 double lhs = c_expl[0];
344                 u_next[i] = rhs / lhs;
345             }
346             boundary_explicit(k);
347             print_vec(out, u_next);
348             u_prev = u_k;
349             u_k = u_next;
350         }
351     }
352 };
353 
354 #endif /* HPDE_HPP */
```

3 Численные методы решения ДУЧП эллиптического типа

3.1 Постановка задачи

Решить краевую задачу для дифференциального уравнения эллиптического типа. Аппроксимацию уравнения произвести с использованием центрально-разностной схемы. Для решения дискретного аналога применить следующие методы: метод простых итераций (метод Либмана), метод Зейделя, метод простых итераций с верхней релаксацией. Вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением $U(x, y)$. Исследовать зависимость погрешности от сеточных параметров h_x, h_y.

3.2 Вариант 8

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2 \cdot \frac{\partial u}{\partial x} - 3 \cdot u$$

$$u(0, y) = \cos y$$

$$u(\frac{\pi}{2}, y) = 0$$

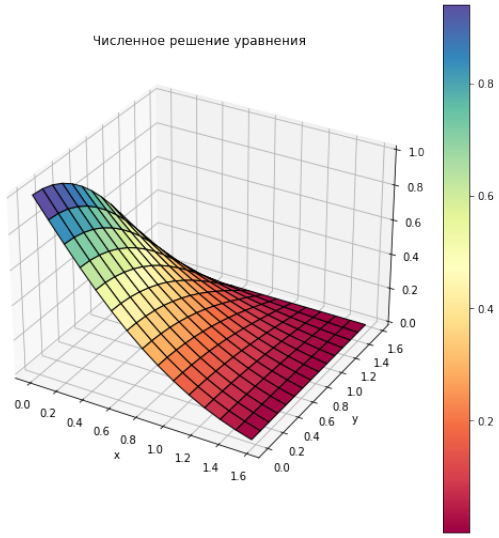
$$u(x, 0) = e^{-x} \cdot \cos x$$

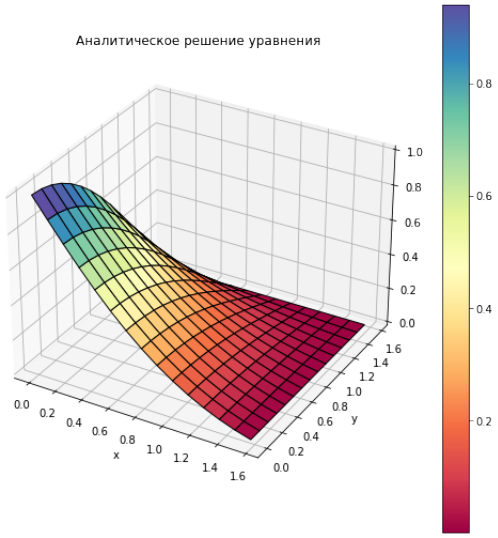
$$u(x, \frac{\pi}{2}) = 0$$

Аналитическое решение:

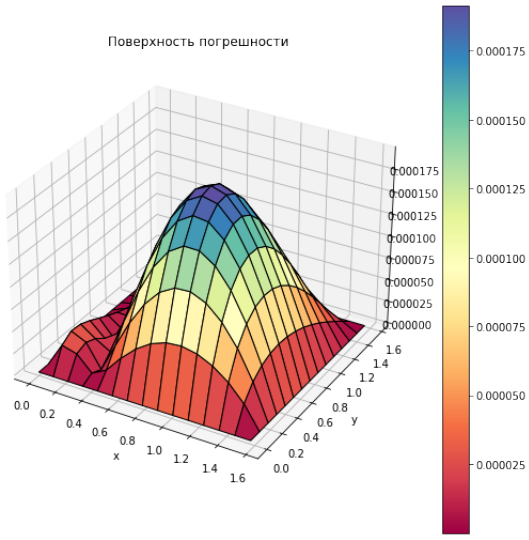
$$U(x, y) = e^{-x} \cdot \cos x \cdot \cos y$$

3.3 Результат





Численное решение уравнения
получено за 222 итераций



3.4 Исходный код

```
#ifndef EPDE_HPP
#define EPDE_HPP

#include "../matrix.hpp"

/* Elliptic Partial Differential Equation Solver */
class epde_t {
    using vec = std::vector<double>;
    using matr = matrix_t<double>;

    static constexpr double EPS = 1e-6;
    static const int N_COEFS = 7;

    int n, m, mu, iters;
    double omega, hx, hy, hx2, hy2;
    matr u;

    double coef[N_COEFS];
    double a, b, c, d, e, f, g;

    double alpha_0y, alpha_Ly, alpha_x0, alpha_xL;
    double beta_0y, beta_Ly, beta_x0, beta_xL;
    vec gamma_0y, gamma_Ly, gamma_x0, gamma_xL;

    void init() {
        a = coef[0];
        b = coef[1];
        c = coef[2];
        d = coef[3];
        e = coef[4];
        f = coef[5];
        g = coef[6];
        hx2 = hx * hx;
        hy2 = hy * hy;
        u = matr::uniform(n, m);
        gamma_x0.resize(n);
        gamma_xL.resize(n);
        gamma_0y.resize(m);
        gamma_Ly.resize(m);
        prepare_scheme();
    }

    static const int N_SCHEME = 5;
    double coef_sch[N_SCHEME];

    void prepare_scheme() {
        coef_sch[0] = 2 * (a / hx2 + c / hy2) - f;
```

```
48          coef_sch[1] = c / hy2 - e / (2 * hy);
49          coef_sch[2] = c / hy2 + e / (2 * hy);
50          coef_sch[3] = a / hx2 - d / (2 * hx);
51          coef_sch[4] = a / hx2 + d / (2 * hx);
52      }
53
54      void iter(const matr& u_prev, matr& u_next) {
55          for (int i = 1; i < n - 1; ++i) {
56              for (int j = 1; j < m - 1; ++j) {
57                  double rhs = coef_sch[1] * u_prev[i][j - 1] +
58                               coef_sch[2] * u_prev[i][j + 1] +
59                               coef_sch[3] * u_prev[i - 1][j] +
60                               coef_sch[4] * u_prev[i + 1][j] + g;
61                  double lhs = coef_sch[0];
62                  u_next[i][j] = rhs / lhs;
63              }
64          }
65          calc_boundary(u_next);
66      }
67
68      void calc_boundary(matr& u_) {
69          for (int i = 0; i < n; ++i) {
70              u_[i][0] = 0;
71              u_[i][m - 1] = 0;
72          }
73          for (int j = 0; j < m; ++j) {
74              u_[0][j] = 0;
75              u_[n - 1][j] = 0;
76          }
77
78          for (int i = 0; i < n; ++i) {
79              double rhs = gamma_x0[i] - alpha_x0 / hy * u_[i][1];
80              double lhs = beta_x0 - alpha_x0 / hy;
81              u_[i][0] += rhs / lhs;
82          }
83          for (int i = 0; i < n; ++i) {
84              double rhs = gamma_xL[i] - alpha_xL / hy * u_[i][m - 2];
85              double lhs = beta_xL - alpha_xL / hy;
86              u_[i][m - 1] += rhs / lhs;
87          }
88
89          for (int j = 0; j < m; ++j) {
90              double rhs = gamma_0y[j] - alpha_0y / hx * u_[1][j];
91              double lhs = beta_0y - alpha_0y / hx;
92              u_[0][j] += rhs / lhs;
93          }
94          for (int j = 0; j < m; ++j) {
95              double rhs = gamma_Ly[j] + alpha_Ly / hx * u_[n - 2][j];
96              double lhs = beta_Ly + alpha_Ly / hx;
```

```
            u_[n - 1][j] += rhs / lhs;
        }

        u_[n - 1][m - 1] /= 2;
        u_[n - 1][0] /= 2;
        u_[0][m - 1] /= 2;
        u_[0][0] /= 2;
    }

    double calc_delta(const matr& u_prev, const matr& u_next) {
        double delta = 0;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < m; ++j) {
                delta = std::max(delta, std::abs(u_prev[i][j] - u_next[i][j]));
            }
        }
        return delta;
    }

public:
    friend std::istream& operator>>(std::istream& in, epde_t& item) {
        in >> item.n >> item.m >> item.mu >> item.omega;
        in >> item.hx >> item.hy;
        for (int i = 0; i < N_COEFS; ++i) {
            in >> item.coef[i];
        }
        item.init();
        in >> item.alpha_0y >> item.alpha_Ly >> item.alpha_x0 >> item.alpha_xL;
        in >> item.beta_0y >> item.beta_Ly >> item.beta_x0 >> item.beta_xL;

        for (int i = 0; i < item.n; ++i) {
            in >> item.gamma_x0[i];
        }
        for (int i = 0; i < item.n; ++i) {
            in >> item.gamma_xL[i];
        }

        for (int i = 0; i < item.m; ++i) {
            in >> item.gamma_0y[i];
        }
        for (int i = 0; i < item.m; ++i) {
            in >> item.gamma_Ly[i];
        }

        return in;
    }

    void solve_simple() {
        double delta = 1;
```

```
146         while (delta > EPS) {
147             matr u_next(n, m);
148             iter(u, u_next);
149             delta = calc_delta(u, u_next);
150             u = u_next;
151             ++iters;
152         }
153     }
154 
155     void solve_zeidel() {
156         double delta = 1;
157         while (delta > EPS) {
158             matr u_next(u);
159             iter(u_next, u_next);
160             delta = calc_delta(u, u_next);
161             u = u_next;
162             ++iters;
163         }
164     }
165 
166     void solve_relax() {
167         double delta = 1;
168         while (delta > EPS) {
169             matr u_next(n, m);
170             iter(u, u_next);
171             u_next = omega * u_next + (1 - omega) * u;
172             delta = calc_delta(u, u_next);
173             u = u_next;
174             ++iters;
175         }
176     }
177 
178     void solve(std::ostream& out) {
179         iters = 0;
180         if (mu == 1) {
181             solve_simple();
182         } else if (mu == 2) {
183             solve_zeidel();
184         } else if (mu == 3) {
185             solve_relax();
186         } else {
187             throw std::runtime_error("Invalid mu");
188         }
189         u.comma_out(false);
190         out << iters << '\n';
191         out << u;
192     }
193 };
194 
```

```
195 #endif /* PPDE_HPP */
```

4 Численные методы решения многомерных задач математической физики

4.1 Постановка задачи

Используя схемы переменных направлений и дробных шагов, решить двумерную начально-краевую задачу для дифференциального уравнения параболического типа. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением $U(x,t)$. Исследовать зависимость погрешности от сеточных параметров τ, h_x, h_y.

4.2 Вариант 9

$$\frac{\partial u}{\partial t} = a \cdot \frac{\partial^2 u}{\partial x^2} + b \cdot \frac{\partial^2 u}{\partial y^2} + \sin x \cdot \sin y \cdot (\mu \cdot \cos(\mu t) + (a+b) \cdot \sin(\mu t))$$

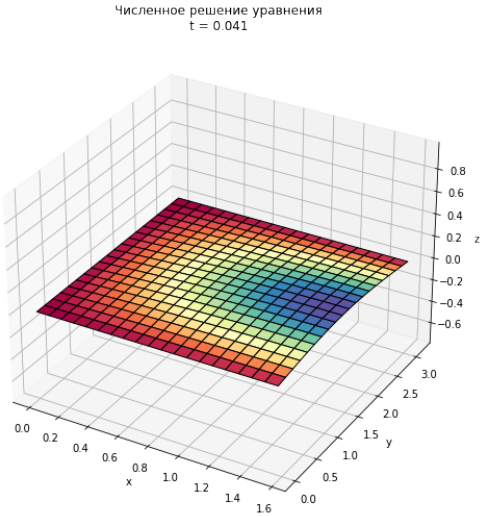
- $a = 1$, $b = 1$, $\mu = 1$;
- $a = 2$, $b = 1$, $\mu = 1$;
- $a = 1$, $b = 2$, $\mu = 1$;
- $a = 1$, $b = 1$, $\mu = 2$.

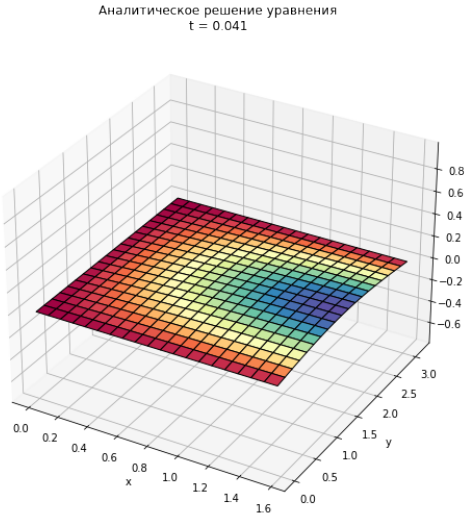
$$u(0, y, t) = 0$$
$$u(\frac{\pi}{2}, y, t) = \sin y \cdot \sin(\mu t)$$
$$u(x, 0, t) = 0$$
$$u(x, \pi, t) = -\sin x \cdot \sin(\mu t)$$
$$u(x, y, 0) = 0$$

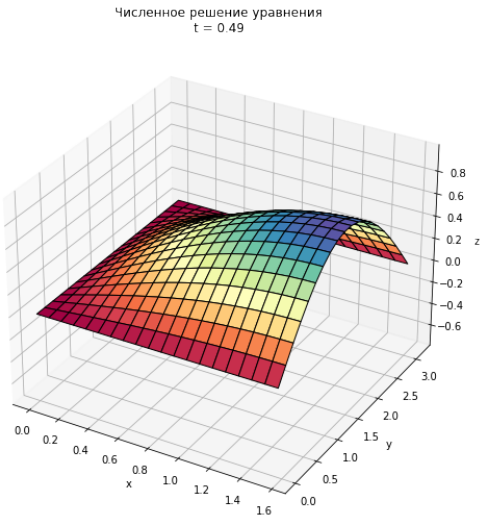
Аналитическое решение:

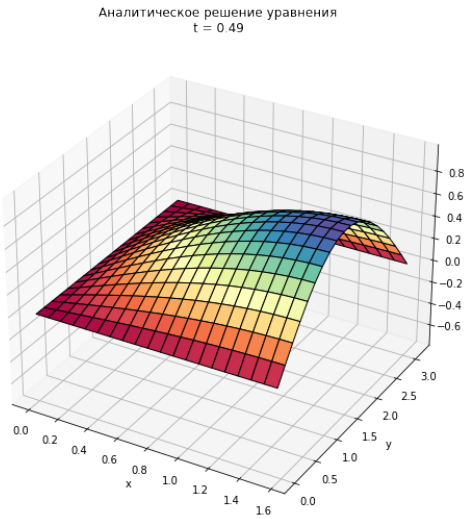
$$U(x, y, t) = \sin x \cdot \sin y \cdot \sin(\mu t)$$

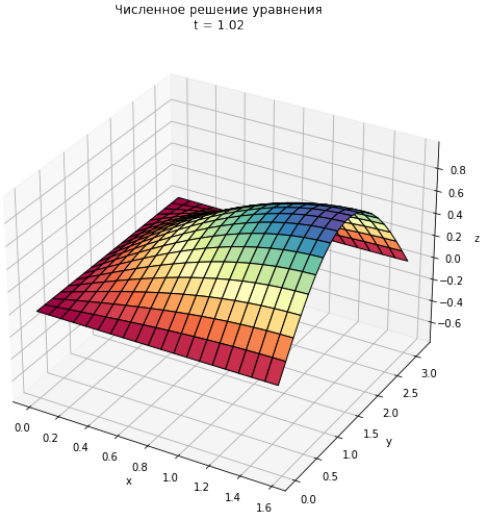
4.3 Результат

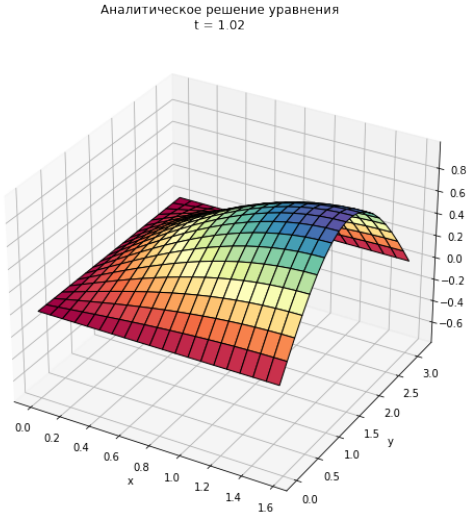


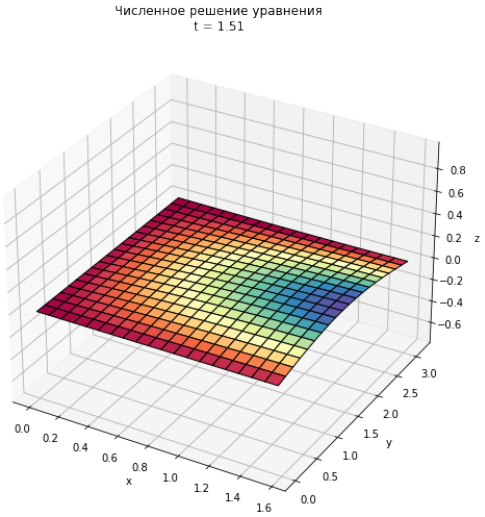


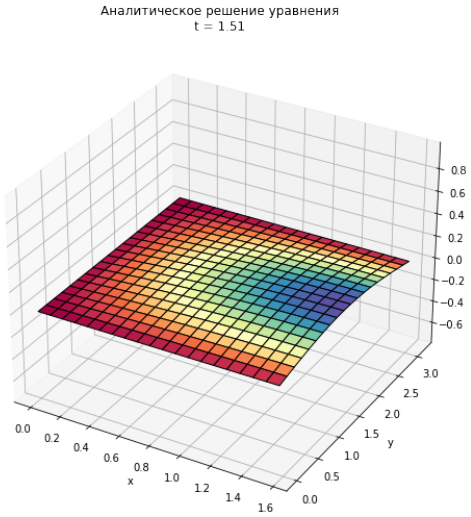


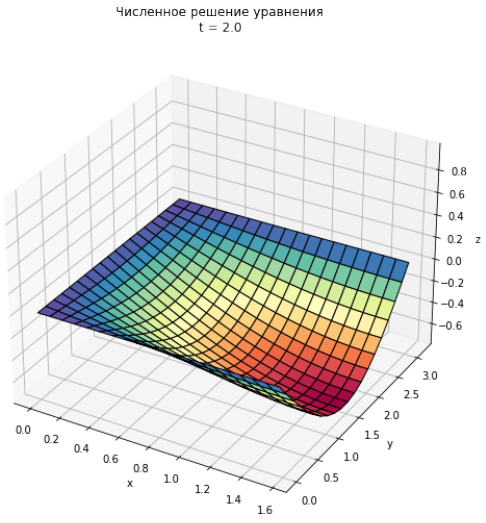


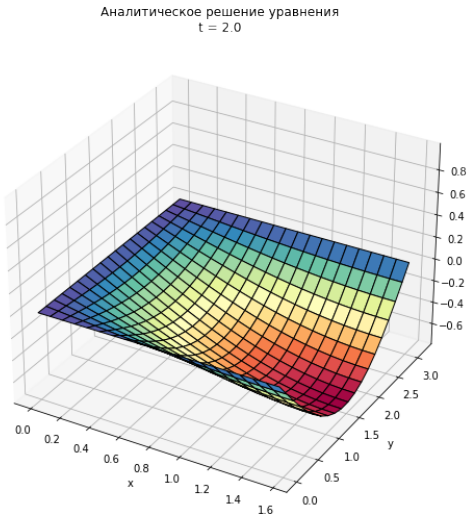

Численное решение уравнения
t = 1.02
x
y
z

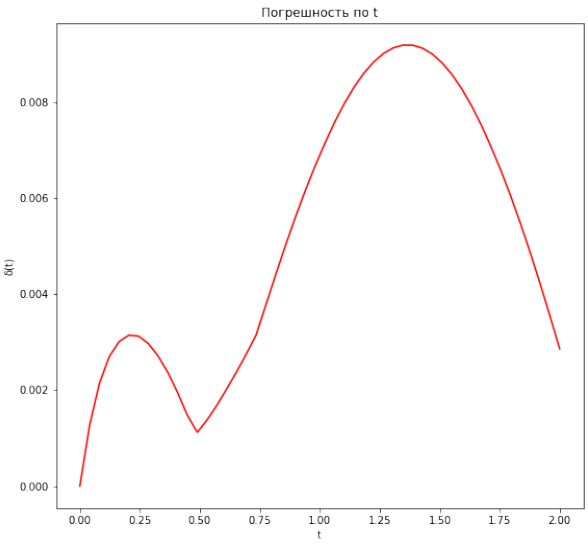

Аналитическое решение уравнения
t = 1.02
x
y
z


Численное решение уравнения
t = 1.51
x
y
z


Аналитическое решение уравнения
t = 1.51
x
y
z


Численное решение уравнения
t = 2.0
x
y
z


Аналитическое решение уравнения
t = 2.0
x
y


Погрешность по t
δ(t)
t

4.4 Исходный код

```
#ifndef PPDE2D_HPP
#define PPDE2D_HPP

#include "../lab1_2/tridiag.hpp"
#include "tensor3.hpp"

/* 2-D Parabolic Partial Differential Equation Solver */
class ppde2d_t {
    using vec = std::vector<double>;
    using tensor3d = tensor3_t<double>;
    using tridiag = tridiag_t<double>;

    int nx, ny, K, psi;
    double lx, ly, T, hx, hy, tau;
    double ax, ay, bx, by, c;
    double alpha_0_y, beta_0_y, alpha_x_0, beta_x_0, alpha_lx_y, beta_lx_y,
        alpha_x_ly, beta_x_ly;
    double hx2, hy2;

    tensor3d f, u, u_abab, u0, gamma_0_y, gamma_x_0, gamma_lx_y, gamma_x_ly;
    vec trd_a_x, trd_b_x, trd_c_x, trd_d_x;
    vec trd_a_y, trd_b_y, trd_c_y, trd_d_y;

    void init() {
        hx2 = hx * hx;
        hy2 = hy * hy;
        u0 = tensor3d(nx, ny, 1, 0.5);
        f = tensor3d(nx, ny, K);
        gamma_0_y = tensor3d(1, ny, K);
        gamma_x_0 = tensor3d(nx, 1, K);
        gamma_lx_y = tensor3d(1, ny, K);
        gamma_x_ly = tensor3d(nx, 1, K);
        trd_a_x.resize(nx);
        trd_b_x.resize(nx);
        trd_c_x.resize(nx);
        trd_d_x.resize(nx);
        trd_a_y.resize(ny);
        trd_b_y.resize(ny);
        trd_c_y.resize(ny);
        trd_d_y.resize(ny);
    }

    vec coefs_x_1, coefs_y_1;
    void prepare_coefs_1() {
        coefs_x_1.resize(6, 0);
        coefs_x_1[0] = ax / hx2 - bx / (2 * hx);
        coefs_x_1[1] = -2 * ax / hx2 - 2 / tau + c;
```

```
48          coefs_x_1[2] = ax / hx2 + bx / (2 * hx);
49          coefs_x_1[3] = -ay / hy2 + by / (2 * hy);
50          coefs_x_1[4] = -2 / tau + 2 * ay / hy2;
51          coefs_x_1[5] = -ay / hy2 - by / (2 * hy);
52
53          coefs_y_1.resize(6, 0);
54          coefs_y_1[0] = ay / hy2 - by / (2 * hy);
55          coefs_y_1[1] = -2 * ay / hy2 - 2 / tau + c;
56          coefs_y_1[2] = ay / hy2 + by / (2 * hy);
57          coefs_y_1[3] = -ax / hx2 + bx / (2 * hx);
58          coefs_y_1[4] = 2 * ax / hx2 - 2 / tau;
59          coefs_y_1[5] = -ax / hx2 - bx / (2 * hx);
60      }
61
62      vec coefs_x_2, coefs_y_2;
63      void prepare_coefs_2() {
64          coefs_x_2.resize(4, 0);
65          coefs_x_2[0] = ax / hx2 - bx / (2 * hx);
66          coefs_x_2[1] = -2 * ax / hx2 - 1 / tau + c;
67          coefs_x_2[2] = ax / hx2 + bx / (2 * hx);
68          coefs_x_2[3] = -1 / tau;
69
70          coefs_y_2.resize(4, 0);
71          coefs_y_2[0] = ay / hy2 - by / (2 * hy);
72          coefs_y_2[1] = -2 * ay / hy2 - 1 / tau + c;
73          coefs_y_2[2] = ay / hy2 + by / (2 * hy);
74          coefs_y_2[3] = -1 / tau;
75      }
76
77      vec bound_coefs_x_0;
78      void prepare_bound_x_0() {
79          bound_coefs_x_0.resize(2);
80          bound_coefs_x_0[0] = beta_x_0 - alpha_x_0 / hy;
81          bound_coefs_x_0[1] = alpha_x_0 / hy;
82      }
83
84      vec bound_coefs_x_ly;
85      void prepare_bound_x_ly() {
86          bound_coefs_x_ly.resize(2);
87          bound_coefs_x_ly[0] = -alpha_x_ly / hy;
88          bound_coefs_x_ly[1] = beta_x_ly + alpha_x_ly / hy;
89      }
90
91      vec bound_coefs_0_y;
92      void prepare_bound_0_y() {
93          bound_coefs_0_y.resize(2);
94          bound_coefs_0_y[0] = beta_0_y - alpha_0_y / hx;
95          bound_coefs_0_y[1] = alpha_0_y / hx;
96      }
```

```
97
98      vec bound_coefs_lx_y;
99      void prepare_bound_lx_y() {
100         bound_coefs_lx_y.resize(2);
101         bound_coefs_lx_y[0] = -alpha_lx_y / hx;
102         bound_coefs_lx_y[1] = beta_lx_y + alpha_lx_y / hx;
103     }
104
105     void prepare_bound() {
106         prepare_bound_0_y();
107         prepare_bound_lx_y();
108         prepare_bound_x_0();
109         prepare_bound_x_ly();
110     }
111
112   public:
113     friend std::istream& operator>>(std::istream& in, ppde2d_t& item) {
114         in >> item.nx >> item.ny >> item.K >> item.psi;
115         in >> item.lx >> item.ly >> item.T >> item.hx >> item.hy >> item.tau;
116         in >> item.ax >> item.ay >> item.bx >> item.by >> item.c;
117         item.init();
118         in >> item.f >> item.u0;
119         in >> item.alpha_0_y >> item.alpha_x_0 >> item.beta_0_y >>
120             item.beta_x_0;
121         in >> item.alpha_lx_y >> item.alpha_x_ly >> item.beta_lx_y >>
122             item.beta_x_ly;
123         in >> item.gamma_0_y >> item.gamma_x_0;
124         in >> item.gamma_lx_y >> item.gamma_x_ly;
125         return in;
126     }
127
128     void trd_x_set_bounds(const int j, const int k) {
129         trd_b_x[0] = bound_coefs_0_y[0];
130         trd_c_x[0] = bound_coefs_0_y[1];
131         trd_d_x[0] = (gamma_0_y(0, j, k) + gamma_0_y(0, j, k + 1)) / 2;
132         trd_a_x[nx - 1] = bound_coefs_lx_y[0];
133         trd_b_x[nx - 1] = bound_coefs_lx_y[1];
134         trd_d_x[nx - 1] = (gamma_lx_y(0, j, k) + gamma_lx_y(0, j, k + 1)) / 2;
135     }
136
137     void trd_y_set_bounds(const int i, const int k) {
138         trd_b_y[0] = bound_coefs_x_0[0];
139         trd_c_y[0] = bound_coefs_x_0[1];
140         trd_d_y[0] = gamma_x_0(i, 0, k + 1);
141         trd_a_y[ny - 1] = bound_coefs_x_ly[0];
142         trd_b_y[ny - 1] = bound_coefs_x_ly[1];
143         trd_d_y[ny - 1] = gamma_x_ly(i, 0, k + 1);
144     }
145
```

```
146     void calc_bounds_x(tensor3d& u_next, const int k) {
147         for (int i = 0; i < nx; ++i) {
148             double rhs_x_0 = (gamma_x_0(i, 0, k) + gamma_x_0(i, 0, k + 1)) / 2 -
149                              bound_coefs_x_0[1] * u_next(i, 1, 0);
150             double lhs_x_0 = bound_coefs_x_0[0];
151             u_next(i, 0, 0) = rhs_x_0 / lhs_x_0;
152             double rhs_x_ly =
153                 (gamma_x_ly(i, 0, k) + gamma_x_ly(i, 0, k + 1)) / 2 -
154                 bound_coefs_x_ly[0] * u_next(i, ny - 2, 0);
155             double lhs_x_ly = bound_coefs_x_ly[1];
156             u_next(i, ny - 1, 0) = rhs_x_ly / lhs_x_ly;
157         }
158     }
159
160     void calc_bounds_y(tensor3d& u_next, const int k) {
161         for (int j = 0; j < ny; ++j) {
162             double rhs_0_y =
163                 gamma_0_y(0, j, k + 1) - bound_coefs_0_y[1] * u_next(1, j, 0);
164             double lhs_0_y = bound_coefs_0_y[0];
165             u_next(0, j, 0) = rhs_0_y / lhs_0_y;
166
167             double rhs_lx_y = gamma_lx_y(0, j, k + 1) -
168                               bound_coefs_lx_y[0] * u_next(nx - 2, j, 0);
169             double lhs_lx_y = bound_coefs_lx_y[1];
170             u_next(nx - 1, j, 0) = rhs_lx_y / lhs_lx_y;
171         }
172     }
173
174     void solve(std::ostream& out) {
175         prepare_bound();
176         prepare_coefs_1();
177         prepare_coefs_2();
178         u = u0;
179         u_abab = u0;
180         out << u << '\n';
181         for (int k = 0; k < K - 1; ++k) {
182             if (psi == 1) {
183                 solve_1_x(u, u_abab, k);
184                 solve_1_y(u_abab, u, k);
185             } else {
186                 solve_2_x(u, u_abab, k);
187                 solve_2_y(u_abab, u, k);
188             }
189             out << u << '\n';
190         }
191     }
192
193     /* k -> k + 1/2 */
194     void solve_1_x(const tensor3d& u_prev, tensor3d& u_next, const int k) {
```

```
195         for (int j = 1; j < ny - 1; ++j) {
196             for (int i = 1; i < nx - 1; ++i) {
197                 trd_a_x[i] = coefs_x_1[0];
198                 trd_b_x[i] = coefs_x_1[1];
199                 trd_c_x[i] = coefs_x_1[2];
200                 trd_d_x[i] = coefs_x_1[3] * u_prev(i, j - 1, 0) +
201                              coefs_x_1[4] * u_prev(i, j, 0) +
202                              coefs_x_1[5] * u_prev(i, j + 1, 0) -
203                              (f(i, j, k) + f(i, j, k + 1)) / 2;
204             }
205             trd_x_set_bounds(j, k);
206             tridiag trd(trd_a_x, trd_b_x, trd_c_x);
207             vec res = trd.solve(trd_d_x);
208             for (int i = 0; i < nx; ++i) {
209                 u_next(i, j, 0) = res[i];
210             }
211         }
212         calc_bounds_x(u_next, k);
213     }
214
215     /* k + 1/2 -> k + 1 */
216     void solve_1_y(const tensor3d& u_prev, tensor3d& u_next, const int k) {
217         for (int i = 1; i < nx - 1; ++i) {
218             for (int j = 1; j < ny - 1; ++j) {
219                 trd_a_y[j] = coefs_y_1[0];
220                 trd_b_y[j] = coefs_y_1[1];
221                 trd_c_y[j] = coefs_y_1[2];
222                 trd_d_y[j] = coefs_y_1[3] * u_prev(i - 1, j, 0) +
223                              coefs_y_1[4] * u_prev(i, j, 0) +
224                              coefs_y_1[5] * u_prev(i + 1, j, 0) -
225                              f(i, j, k + 1);
226             }
227             trd_y_set_bounds(i, k);
228             tridiag trd(trd_a_y, trd_b_y, trd_c_y);
229             vec res = trd.solve(trd_d_y);
230             for (int j = 0; j < ny; ++j) {
231                 u_next(i, j, 0) = res[j];
232             }
233         }
234         calc_bounds_y(u_next, k);
235     }
236
237     /* k -> k + 1/2 */
238     void solve_2_x(const tensor3d& u_prev, tensor3d& u_next, const int k) {
239         for (int j = 1; j < ny - 1; ++j) {
240             for (int i = 1; i < nx - 1; ++i) {
241                 trd_a_x[i] = coefs_x_2[0];
242                 trd_b_x[i] = coefs_x_2[1];
243                 trd_c_x[i] = coefs_x_2[2];
```

```
244                 trd_d_x[i] = coefs_x_2[3] * u_prev(i, j, 0) -
245                              0.5 * (f(i, j, k) + f(i, j, k + 1)) / 2;
246             }
247             trd_x_set_bounds(j, k);
248             tridiag trd(trd_a_x, trd_b_x, trd_c_x);
249             vec res = trd.solve(trd_d_x);
250             for (int i = 0; i < nx; ++i) {
251                 u_next(i, j, 0) = res[i];
252             }
253         }
254         calc_bounds_x(u_next, k);
255     }
256 
257     /* k + 1/2 -> k + 1 */
258     void solve_2_y(const tensor3d& u_prev, tensor3d& u_next, const int k) {
259         for (int i = 1; i < nx - 1; ++i) {
260             for (int j = 1; j < ny - 1; ++j) {
261                 trd_a_y[j] = coefs_y_2[0];
262                 trd_b_y[j] = coefs_y_2[1];
263                 trd_c_y[j] = coefs_y_2[2];
264                 trd_d_y[j] =
265                     coefs_y_2[3] * u_prev(i, j, 0) - 0.5 * f(i, j, k + 1);
266             }
267             trd_y_set_bounds(i, k);
268             tridiag trd(trd_a_y, trd_b_y, trd_c_y);
269             vec res = trd.solve(trd_d_y);
270             for (int j = 0; j < ny; ++j) {
271                 u_next(i, j, 0) = res[j];
272             }
273         }
274         calc_bounds_y(u_next, k);
275     }
276 };
277 
278 #endif /* PPDE2D_HPP */
```