Московский авиационный институт (национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторные работы по курсу «Численные методы»

Студент: В. В. Бирюков
Преподаватель: Д. Л. Ревизников
Группа: М8О-407Б-19

Дата:
Оценка:
Подпись:

Москва, 2022

1 Решение начально-краевой задачи для дифференциальных уравнений в частных производных параболического типа

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 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.

Вариант: 10

$$\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial u}{\partial x} + cu,\ a > 0,\ b > 0,\ c < 0$$
$$u_x(0,t) + u(0,t) = \exp((c-a)t)(\cos(bt) + \sin(bt))$$
$$u_x(\pi,t) + u(\pi,t) = -\exp((c-a)t)(\cos(bt) + \sin(bt))$$
$$u(x,0) = \sin x$$
$$U(x,t) = \exp((c-a)t)\sin(x+bt)$$

2 Результаты работы

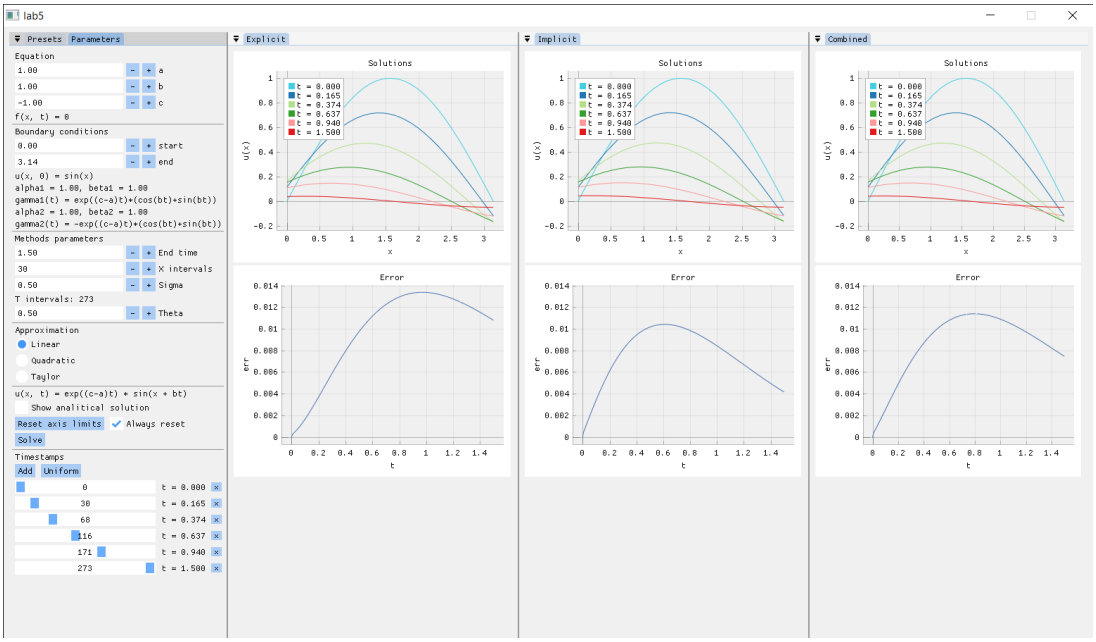


Рис. 1: Решение с аппроксимацией граничных условий с первым порядком

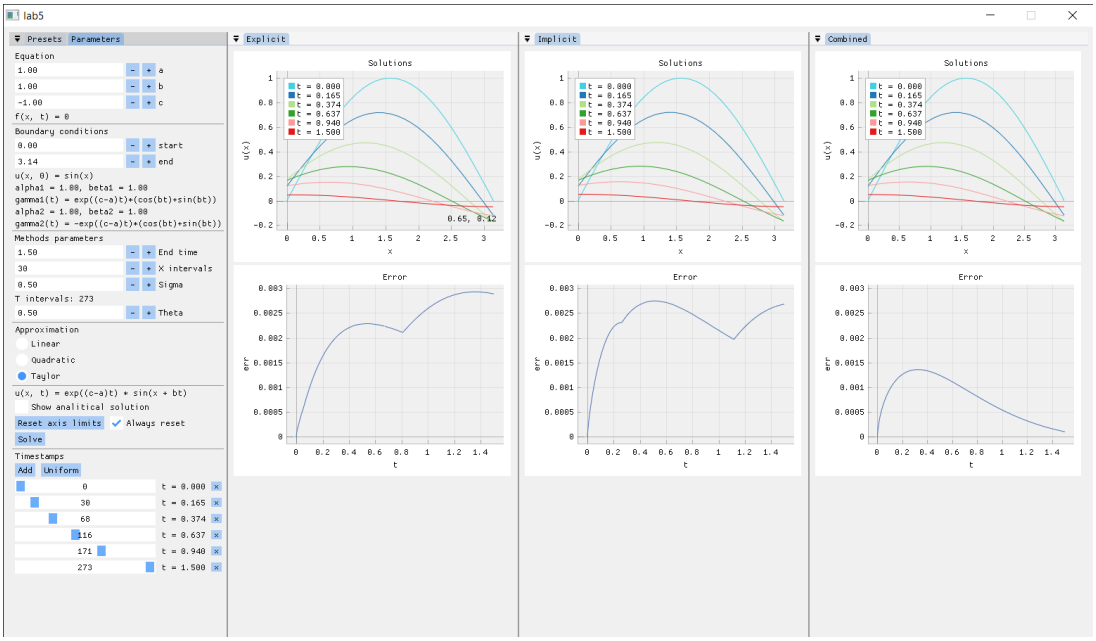


Рис. 2: Решение с аппроксимацией граничных условий со вторым порядком

3 Исходный код

common.hpp

```cpp
#pragma once

#include <tuple>
#include <vector>
#include <functional>

enum class ApproxType : int {
  Linear,
  Quadratic,
  Taylor
};

template <class T, template<class> class PDE>
std::tuple<std::vector<T>, std::vector<T>>
GenerateGrid(const PDE<T>& pde, T t_end, int h_count, double sigma, const std::
    function<T(int, double, T, T, T)>& CourantCondition) {
  int tau_count = CourantCondition(h_count, sigma, t_end, pde.end - pde.start, pde.a);
  std::vector<T> x(h_count + 1), t(tau_count + 1);
  T h = (pde.end - pde.start) / h_count;
  T tau = t_end / tau_count;

  for (int i = 0; i <= h_count; ++i) {
    x[i] = pde.start + h * i;
  }
  for (int k = 0; k <= tau_count; ++k) {
    t[k] = tau * k;
  }

  return {x, t};
}

template <class T>
struct Boundaries {
  struct Coeffs {
    T alpha, beta;
  };

  Coeffs left, right;
};
```

parabolic_pde.hpp

```cpp
#pragma once

#include <functional>
#include <vector>
#include <tuple>

#include "../linear/tridiagonal_matrix.hpp"
#include "../linear/vector.hpp"
#include "common.hpp"

namespace ParabolicPDE {
  template <class T>
  using grid_t = std::vector<std::vector<T>>;

  template <class T>
  struct PDE {
    T a, b, c;
    std::function<T(T, T)> f;
    std::function<T(T)> psi;
    T start, end;
    T alpha1, beta1;
    std::function<T(T)> gamma1;
    T alpha2, beta2;
    std::function<T(T)> gamma2;
    std::function<T(T, T)> solution;

    PDE() = default;

    PDE(T a, T b, T c, std::function<T(T, T)> f, std::function<T(T)> psi, T start, T
        end,
        T alpha1, T beta1, std::function<T(T)> gamma1, T alpha2, T beta2, std::function
            <T(T)> gamma2) :
        a(a), b(b), c(c), f(f), psi(psi), start(start), end(end), alpha1(alpha1), beta1
            (beta1), gamma1(gamma1),
        alpha2(alpha2), beta2(beta2), gamma2(gamma2) {}

    PDE(T a, T b, T c, std::function<T(T, T)> f) :
        a(a), b(b), c(c), f(f) {}

    void SetEquation(T a_, T b_, T c_, std::function<T(T, T)> f_) {
      a = a_;
      b = b_;
      c = c_;
      f = f_;
    }

    void SetBoundaries(std::function<T(T)> psi_, T start_, T end_, T alpha1_, T beta1_,
```

```
        std::function<T(T)> gamma1_,
45              T alpha2_, T beta2_, std::function<T(T)> gamma2_) {
46    psi = psi_;
47    start = start_;
48    end = end_;
49    alpha1 = alpha1_;
50    beta1 = beta1_;
51    gamma1 = gamma1_;
52    alpha2 = alpha2_;
53    beta2 = beta2_;
54    gamma2 = gamma2_;
55  }
56
57  void SetSolution(std::function<T(T, T)> solution_) {
58    solution = solution_;
59  }
60 };
61
62 template <class T>
63 int CourantCondition(int h_count, double sigma, T t_end, T end, T a) {
64   return t_end * a * h_count * h_count / (end * end * sigma);
65 }
66
67 template <class T>
68 std::tuple<std::vector<T>, std::vector<T>, grid_t<T>>
69 ExplicitSolver(const PDE<T>& pde, T t_end, int h_count, double sigma, ApproxType
      type) {
70   auto [x, t] = GenerateGrid<T, PDE>(pde, t_end, h_count, sigma, CourantCondition<T>)
        ;
71   return {x, t, ExplicitSolver(pde, x, t, t_end, type)};
72 }
73
74 template <class T>
75 grid_t<T> ExplicitSolver(const PDE<T>& pde, const std::vector<T>& x, const std::
      vector<T>& t, T t_end, ApproxType type) {
76   int h_count = x.size() - 1, tau_count = t.size() - 1;
77   grid_t<T> u(tau_count + 1, std::vector<T>(h_count + 1));
78   T h = (pde.end - pde.start) / h_count;
79   T tau = t_end / tau_count;
80
81   for (int i = 0; i <= h_count; ++i) {
82     u[0][i] = pde.psi(x[i]);
83   }
84
85   for (int k = 0; k < tau_count; ++k) {
86     for (int i = 1; i < h_count; ++i) {
87       T ddu = (u[k][i-1] - 2 * u[k][i] + u[k][i+1]) / (h * h);
88       T du = (u[k][i+1] - u[k][i-1]) / (2 * h);
89       u[k+1][i] = (pde.a * ddu + pde.b * du + pde.c * u[k][i] + pde.f(x[i], t[k])) *
```

```
            tau + u[k][i];
90      }
91
92      if (type == ApproxType::Linear) {
93        u[k+1][0] = (pde.gamma1(t[k+1]) - pde.alpha1 / h * u[k+1][1]) / (-pde.alpha1 /
              h + pde.beta1);
94        u[k+1][h_count] = (pde.gamma2(t[k+1]) + pde.alpha2 / h * u[k+1][h_count-1]) / (
              pde.alpha2 / h + pde.beta2);
95
96      } else if (type == ApproxType::Quadratic) {
97        u[k+1][0] = (pde.gamma1(t[k+1]) - pde.alpha1 / (2 * h) * (4 * u[k+1][1] - u[k
              +1][2])) / (-3 * pde.alpha1 / (2 * h) + pde.beta1);
98        u[k+1][h_count] = (pde.gamma2(t[k+1]) - pde.alpha2 / (2 * h) * (u[k+1][h_count
              -2] - 4 * u[k+1][h_count-1])) / (3 * pde.alpha2 / (2 * h) + pde.beta2);
99
100     } else if (type == ApproxType::Taylor) {
101       T div = h - h * h * pde.b / (2 * pde.a),
102         mult1 = (pde.c * h * h / (2 * pde.a) - 1 - h * h / (2 * pde.a * tau)),
103         mult2 = h * h / (2 * pde.a);
104
105       u[k+1][0] = (pde.gamma1(t[k+1]) - pde.alpha1 * mult2 / div * (u[k][0] / tau +
              pde.f(x[0], t[k+1])) - pde.alpha1 / div * u[k+1][1]) /
106               (pde.alpha1 * mult1 / div + pde.beta1);
107
108       div = -h - h * h * pde.b / (2 * pde.a);
109       u[k+1][h_count] = (pde.gamma2(t[k+1]) - pde.alpha2 * mult2 / div * (u[k][
              h_count] / tau + pde.f(x[h_count], t[k+1])) - pde.alpha2 / div * u[k+1][
              h_count-1]) /
110               (pde.alpha2 * mult1 / div + pde.beta2);
111     }
112   }
113   return u;
114 }
115
116 template <class T>
117 std::tuple<std::vector<T>, std::vector<T>, grid_t<T>>
118 ImplicitSolver(const PDE<T>& pde, T t_end, int h_count, double sigma, ApproxType
        type) {
119   auto [x, t] = GenerateGrid<T, PDE>(pde, t_end, h_count, sigma, CourantCondition<T>)
          ;
120   return {x, t, ImplicitSolver(pde, x, t, t_end, type)};
121 }
122
123 template <class T>
124 grid_t<T> ImplicitSolver(const PDE<T>& pde, const std::vector<T>& x, const std::
        vector<T>& t, T t_end, ApproxType type) {
125   int h_count = x.size() - 1, tau_count = t.size() - 1;
126   grid_t<T> u(tau_count + 1, std::vector<T>(h_count + 1));
127   T h = (pde.end - pde.start) / h_count;
```

```
128     T tau = t_end / tau_count;
129
130     for (int i = 0; i <= h_count; ++i) {
131       u[0][i] = pde.psi(x[i]);
132     }
133
134     T alpha = (pde.a / h - pde.b / 2) * tau / h,
135       beta = -1 - 2 * pde.a * tau / (h * h) + pde.c * tau,
136       gamma = (pde.a / h + pde.b / 2) * tau / h;
137
138     TDMatrix<T> matrix(h_count+1);
139     for (int i = 1; i < h_count; ++i) {
140       matrix.a[i] = alpha;
141       matrix.b[i] = beta;
142       matrix.c[i] = gamma;
143     }
144
145     Vector<T> v(h_count+1);
146     for (int k = 0; k < tau_count; ++k) {
147       for (int i = 1; i < h_count; ++i) {
148         v[i] = -u[k][i] - tau * pde.f(x[i], t[k+1]);
149       }
150       v[0] = pde.gamma1(t[k+1]);
151       v[h_count] = pde.gamma2(t[k+1]);
152
153       if (type == ApproxType::Linear) {
154         matrix.b[0] = -pde.alpha1 / h + pde.beta1;
155         matrix.c[0] = pde.alpha1 / h;
156
157         matrix.a[h_count] = -pde.alpha2 / h;
158         matrix.b[h_count] = pde.alpha2 / h + pde.beta2;
159
160       } else if (type == ApproxType::Quadratic) {
161         T coeff = -pde.alpha1 / (2 * h) / gamma;
162         matrix.b[0] = -3 * pde.alpha1 / (2 * h) + pde.beta1 - coeff * alpha;
163         matrix.c[0] = 2 * pde.alpha1 / h - coeff * beta;
164         v[0] -= coeff * v[1];
165
166         coeff = pde.alpha2 / (2 * h) / alpha;
167         matrix.a[h_count] = -2 * pde.alpha2 / h - coeff * beta;
168         matrix.b[h_count] = 3 * pde.alpha2 / (2 * h) + pde.beta2 - coeff * gamma;
169         v[h_count] -= coeff * v[h_count-1];
170
171       } else if (type == ApproxType::Taylor) {
172         T div = h - h * h * pde.b / (2 * pde.a),
173           mult1 = (pde.c * h * h / (2 * pde.a) - 1 - h * h / (2 * pde.a * tau)),
174           mult2 = h * h / (2 * pde.a);
175
176         matrix.b[0] = pde.alpha1 * mult1 / div + pde.beta1;
```

```
177         matrix.c[0] = pde.alpha1 / div;
178         v[0] -= pde.alpha1 * mult2 / div * (u[k][0] / tau + pde.f(x[0], t[k+1]));
179 
180         div = -h - h * h * pde.b / (2 * pde.a);
181         matrix.a[h_count] = pde.alpha2 / div;
182         matrix.b[h_count] = pde.alpha2 * mult1 / div + pde.beta2;
183         v[h_count] -= pde.alpha2 * mult2 / div * (u[k][h_count] / tau + pde.f(x[h_count
                ], t[k+1]));
184       }
185 
186       u[k+1] = matrix.Solve(v);
187     }
188     return u;
189   }
190 
191   template <class T>
192   std::tuple<std::vector<T>, std::vector<T>, grid_t<T>>
193   CombinedSolver(const PDE<T>& pde, T t_end, int h_count, double sigma, ApproxType
        type, double theta) {
194     auto [x, t] = GenerateGrid<T, PDE>(pde, t_end, h_count, sigma, CourantCondition<T>)
          ;
195     return {x, t, CombinedSolver(pde, x, t, t_end, type, theta)};
196   }
197 
198   template <class T>
199   grid_t<T> CombinedSolver(const PDE<T>& pde, const std::vector<T>& x, const std::
        vector<T>& t, T t_end, ApproxType type, double theta) {
200     int h_count = x.size() - 1, tau_count = t.size() - 1;
201     grid_t<T> u(tau_count + 1, std::vector<T>(h_count + 1));
202     T h = (pde.end - pde.start) / h_count;
203     T tau = t_end / tau_count;
204 
205     for (int i = 0; i <= h_count; ++i) {
206       u[0][i] = pde.psi(x[i]);
207     }
208 
209     T alpha = theta * (pde.a / h - pde.b / 2) * tau / h,
210       beta = -1 + theta * (-2 * pde.a * tau / (h * h) + pde.c * tau),
211       gamma = theta * (pde.a / h + pde.b / 2) * tau / h;
212 
213     TDMatrix<T> matrix(h_count+1);
214     for (int i = 1; i < h_count; ++i) {
215       matrix.a[i] = alpha;
216       matrix.b[i] = beta;
217       matrix.c[i] = gamma;
218     }
219 
220     Vector<T> v(h_count+1);
221     for (int k = 0; k < tau_count; ++k) {
```

```
222       for (int i = 1; i < h_count; ++i) {
223         T ddu = (u[k][i-1] - 2 * u[k][i] + u[k][i+1]) / (h * h);
224         T du = (u[k][i+1] - u[k][i-1]) / (2 * h);
225         T uik = pde.a * ddu + pde.b * du + pde.c * u[k][i] + pde.f(x[i], t[k]);
226         v[i] = -u[k][i] - tau * (theta * pde.f(x[i], t[k]) + (1 - theta) * uik);
227       }
228       v[0] = pde.gamma1(t[k+1]);
229       v[h_count] = pde.gamma2(t[k+1]);
230
231       if (type == ApproxType::Linear) {
232         matrix.b[0] = -pde.alpha1 / h + pde.beta1;
233         matrix.c[0] = pde.alpha1 / h;
234
235         matrix.a[h_count] = -pde.alpha2 / h;
236         matrix.b[h_count] = pde.alpha2 / h + pde.beta2;
237
238       } else if (type == ApproxType::Quadratic) {
239         T coeff = -pde.alpha1 / (2 * h) / gamma;
240         matrix.b[0] = -3 * pde.alpha1 / (2 * h) + pde.beta1 - coeff * alpha;
241         matrix.c[0] = 2 * pde.alpha1 / h - coeff * beta;
242         v[0] -= coeff * v[1];
243
244         coeff = pde.alpha2 / (2 * h) / alpha;
245         matrix.a[h_count] = -2 * pde.alpha2 / h - coeff * beta;
246         matrix.b[h_count] = 3 * pde.alpha2 / (2 * h) + pde.beta2 - coeff * gamma;
247         v[h_count] -= coeff * v[h_count-1];
248
249       } else if (type == ApproxType::Taylor) {
250         T div = h - h * h * pde.b / (2 * pde.a),
251           mult1 = (pde.c * h * h / (2 * pde.a) - 1 - h * h / (2 * pde.a * tau)),
252           mult2 = h * h / (2 * pde.a);
253
254         matrix.b[0] = pde.alpha1 * mult1 / div + pde.beta1;
255         matrix.c[0] = pde.alpha1 / div;
256         v[0] -= pde.alpha1 * mult2 / div * (u[k][0] / tau + pde.f(x[0], t[k+1]));
257
258         div = -h - h * h * pde.b / (2 * pde.a);
259         matrix.a[h_count] = pde.alpha2 / div;
260         matrix.b[h_count] = pde.alpha2 * mult1 / div + pde.beta2;
261         v[h_count] -= pde.alpha2 * mult2 / div * (u[k][h_count] / tau + pde.f(x[h_count
              ], t[k+1]));
262       }
263
264       u[k+1] = matrix.Solve(v);
265     }
266     return u;
267   }
268 }
```

2 Решение начально-краевой задачи для дифференциальных уравнений в частных производных гиперболического типа

1 Постановка задачи

Используя явную и неявную конечно-разностные схемы, 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.

Вариант: 2

$$\frac{\partial^2 u}{\partial t^2} = a^2 \frac{\partial^2 u}{\partial x^2},\ a^2 > 0$$
$$u_x(0,t) - u(0,t) = 0$$
$$u_x(\pi,t) - u(\pi,t) = 0$$
$$u(x,0) = \sin x + \cos x$$
$$u_t(x,0) = -a(\sin x + \cos x)$$
$$U(x,t) = \sin(x - at) + \cos(x + at)$$

2 Результаты работы

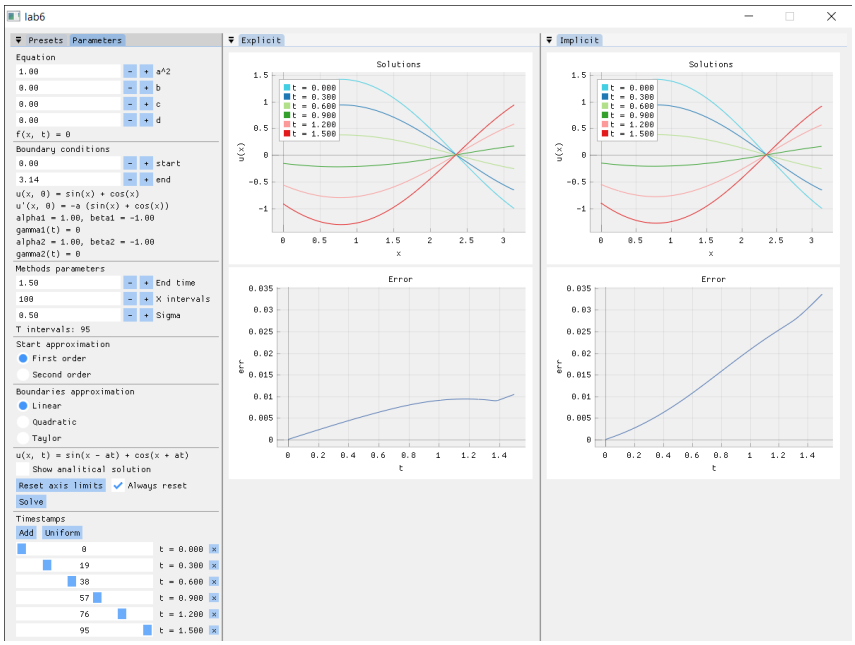


Рис. 3: Решение с аппроксимацией граничных и начальных условий с первым порядком

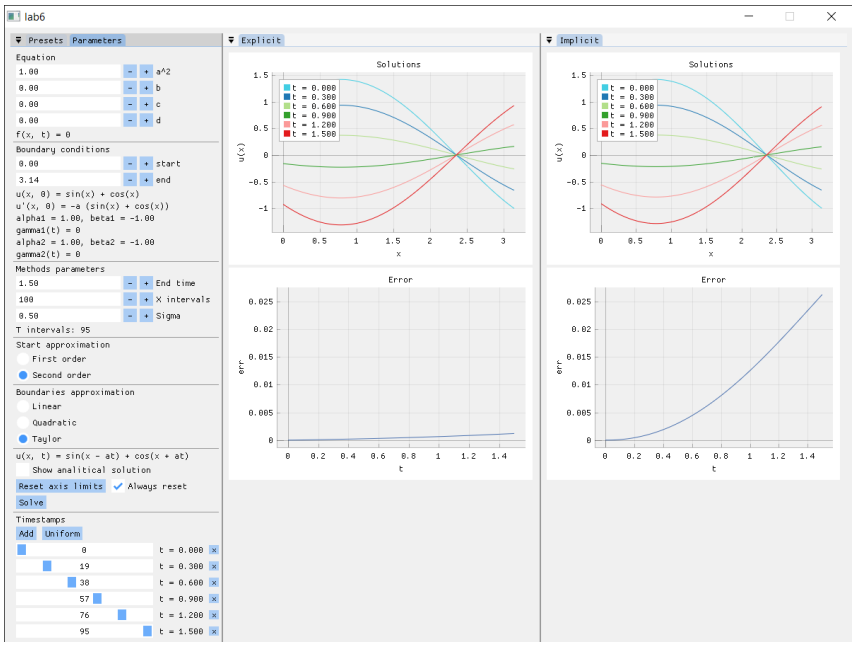


Рис. 4: Решение с аппроксимацией граничных и начальных условий со вторым порядком

3 Исходный код

```cpp
#pragma once

#include <functional>
#include <vector>
#include <tuple>

#include "../linear/tridiagonal_matrix.hpp"
#include "../linear/vector.hpp"
#include "common.hpp"

namespace HyperbolicPDE {
  template <class T>
  using grid_t = std::vector<std::vector<T>>;

  template <class T>
  struct PDE {
    using f_t = std::function<T(T)>;
    using f_x = f_t;
    using f_x_t = std::function<T(T, T)>;

    T a, b, c, d;
    f_x_t f;
    f_x psi1, dpsi1, d2psi1, psi2;
    T start, end;
    T alpha1, beta1;
    f_t gamma1;
    T alpha2, beta2;
    f_t gamma2;
    f_x_t solution;

    PDE() = default;

    PDE(T a, T b, T c, T d, f_x_t f, f_x psi1, f_x dpsi1, f_x d2psi1, f_x psi2, T start
        , T end,
        T alpha1, T beta1, f_t gamma1, T alpha2, T beta2, f_t gamma2, f_x_t solution) :
        a(a), b(b), c(c), d(d), f(f), psi1(psi1), dpsi1(dpsi1), d2psi1(d2psi1), psi2(
            psi2), start(start), end(end), alpha1(alpha1), beta1(beta1), gamma1(gamma1)
            ,
        alpha2(alpha2), beta2(beta2), gamma2(gamma2), solution(solution) {}

    PDE(T a, T b, T c, T d, f_x_t f, f_x psi1, f_x dpsi1, f_x d2psi1, f_x psi2, T start
        , T end,
        T alpha1, T beta1, f_t gamma1, T alpha2, T beta2, f_t gamma2) :
        a(a), b(b), c(c), d(d), f(f), psi1(psi1), dpsi1(dpsi1), d2psi1(d2psi1), psi2(
            psi2), start(start), end(end), alpha1(alpha1), beta1(beta1), gamma1(gamma1)
            ,
        alpha2(alpha2), beta2(beta2), gamma2(gamma2) {}
```

```
42
43      PDE(T a, T b, T c, T d, f_x_t f) :
44          a(a), b(b), c(c), d(d), f(f) {}
45
46      void SetEquation(T a_, T b_, T c_, T d_, f_x_t f_) {
47        a = a_;
48        b = b_;
49        c = c_;
50        d = d_;
51        f = f_;
52      }
53
54      void SetBoundaries(f_x psi1_, f_x dpsi1_, f_x d2psi1_, f_x psi2_, T start_, T end_,
            T alpha1_, T beta1_, std::function<T(T)> gamma1_,
55                  T alpha2_, T beta2_, std::function<T(T)> gamma2_) {
56        psi1 = psi1_;
57        dpsi1 = dpsi1_;
58        d2psi1 = d2psi1_;
59        psi2 = psi2_;
60        start = start_;
61        end = end_;
62        alpha1 = alpha1_;
63        beta1 = beta1_;
64        gamma1 = gamma1_;
65        alpha2 = alpha2_;
66        beta2 = beta2_;
67        gamma2 = gamma2_;
68      }
69
70      void SetSolution(f_x_t solution_) {
71        solution = solution_;
72      }
73    };
74
75    template <class T>
76    int CourantCondition(int h_count, double sigma, T t_end, T end, T a) {
77      return t_end * a * h_count / (end * sigma);
78    }
79
80    template <class T>
81    void StartConditions(const PDE<T>& pde, const std::vector<T>& x, const std::vector<T
          >& t, grid_t<T>& u, T tau, ApproxType type) {
82      for (size_t i = 0; i < x.size(); ++i) {
83        u[0][i] = pde.psi1(x[i]);
84        if (type == ApproxType::Linear) {
85          u[1][i] = u[0][i] + tau * pde.psi2(x[i]);
86        } else { // Taylor
87          u[1][i] = tau * (1 + tau * pde.d / 2) * pde.psi2(x[i]) + u[0][i] + tau * tau /
                2 * (pde.a * pde.d2psi1(x[i]) +
```

```
88                  pde.b * pde.dpsi1(x[i]) + pde.c * pde.psi1(x[i]) + pde.f(x[i], t[0]));
89      }
90    }
91  }
92
93  template <class T>
94  Boundaries<T> BoundariesConditions(const PDE<T>& pde, const std::vector<T>& x, const
        std::vector<T>& t, grid_t<T>& u, T h, T tau, ApproxType type) {
95    Boundaries<T> bound;
96
97    if (type == ApproxType::Linear) {
98      bound.left.alpha = -pde.alpha1 / h + pde.beta1;
99      bound.left.beta = pde.alpha1 / h;
100
101     bound.right.alpha = pde.alpha2 / h + pde.beta2;
102     bound.right.beta = -pde.alpha2 / h;
103
104   } else if (type == ApproxType::Quadratic) {
105     bound.left.alpha = -3 * pde.alpha1 / (2 * h) + pde.beta1;
106     bound.left.beta = 2 * pde.alpha1 / h;
107
108     bound.right.alpha = 3 * pde.alpha2 / (2 * h) + pde.beta2;
109     bound.right.beta = -2 * pde.alpha2 / h;
110
111   } else if (type == ApproxType::Taylor) {
112     bound.left.alpha = pde.alpha1 * (-1 - h * h / (2 * pde.a) * (1 / (tau * tau) -
          pde.c - pde.d / tau)) + pde.beta1 * h * (1 - pde.b * h / (2 * pde.a));
113     bound.left.beta = pde.alpha1;
114
115     bound.right.alpha = pde.alpha2 * (-1 - h * h / (2 * pde.a) * (1 / (tau * tau) -
          pde.c - pde.d / tau)) + pde.beta2* h * (-1 - pde.b * h / (2 * pde.a));
116     bound.right.beta = pde.alpha2;
117   }
118
119   return bound;
120 }
121
122 template <class T>
123 std::tuple<std::vector<T>, std::vector<T>, grid_t<T>>
124 ExplicitSolver(const PDE<T>& pde, T t_end, int h_count, double sigma, ApproxType
      start_type, ApproxType bound_type) {
125   auto [x, t] = GenerateGrid<T, PDE>(pde, t_end, h_count, sigma, CourantCondition<T>)
        ;
126   return {x, t, ExplicitSolver(pde, x, t, t_end, start_type, bound_type)};
127 }
128
129 template <class T>
130 grid_t<T> ExplicitSolver(const PDE<T>& pde, const std::vector<T>& x, const std::
      vector<T>& t, T t_end, ApproxType start_type, ApproxType bound_type) {
```

```
131     int h_count = x.size() - 1, tau_count = t.size() - 1;
132     grid_t<T> u(tau_count + 1, std::vector<T>(h_count + 1));
133     T h = (pde.end - pde.start) / h_count;
134     T tau = t_end / tau_count;
135 
136     StartConditions(pde, x, t, u, tau, start_type);
137     Boundaries bound = BoundariesConditions(pde, x, t, u, h, tau, bound_type);
138 
139     T gamma1, gamma2, coeff = (1 / tau - pde.d / 2) / tau;
140     for (int k = 1; k < tau_count; ++k) {
141       for (int i = 1; i < h_count; ++i) {
142         T ddu = (u[k][i-1] - 2 * u[k][i] + u[k][i+1]) / (h * h);
143         T du = (u[k][i+1] - u[k][i-1]) / (2 * h);
144         u[k+1][i] = (pde.a * ddu + pde.b * du + pde.c * u[k][i] + pde.f(x[i], t[k]) -
                pde.d / (2 * tau) * u[k-1][i] + (2 * u[k][i] - u[k-1][i]) / (tau * tau)) /
                coeff;
145       }
146 
147       gamma1 = pde.gamma1(t[k+1]);
148       gamma2 = pde.gamma2(t[k+1]);
149       if (bound_type == ApproxType::Taylor) {
150         gamma1 = gamma1 * h * (1 - pde.b * h / (2 * pde.a)) + pde.alpha1 * h * h / (2 *
                pde.a) * (-pde.f(x[0], t[k+1]) + pde.d / tau * u[k][0] + (u[k-1][0] - 2 *
                u[k][0]) / (tau * tau));
151         gamma2 = gamma2 * h * (-1 - pde.b * h / (2 * pde.a)) + pde.alpha2 * h * h / (2
                * pde.a) * (-pde.f(x[h_count], t[k+1]) + pde.d / tau * u[k][h_count] + (u[k
                -1][h_count] - 2 * u[k][h_count]) / (tau * tau));
152       }
153 
154       u[k+1][0] = (gamma1 - u[k+1][1] * bound.left.beta) / bound.left.alpha;
155       u[k+1][h_count] = (gamma2 - u[k+1][h_count-1] * bound.right.beta) / bound.right.
              alpha;
156 
157       if (bound_type == ApproxType::Quadratic) {
158         u[k+1][0] -= -pde.alpha1 / (2 * h) * u[k+1][2] / bound.left.alpha;
159         u[k+1][h_count] -= pde.alpha2 / (2 * h) * u[k+1][h_count-2] / bound.right.alpha
                ;
160       }
161     }
162     return u;
163   }
164 
165   template <class T>
166   std::tuple<std::vector<T>, std::vector<T>, grid_t<T>>
167   ImplicitSolver(const PDE<T>& pde, T t_end, int h_count, double sigma, ApproxType
          start_type, ApproxType bound_type) {
168     auto [x, t] = GenerateGrid<T, PDE>(pde, t_end, h_count, sigma, CourantCondition<T>)
            ;
169     return {x, t, ImplicitSolver(pde, x, t, t_end, start_type, bound_type)};
```

```
170 }
171
172 template <class T>
173 grid_t<T> ImplicitSolver(const PDE<T>& pde, const std::vector<T>& x, const std::
        vector<T>& t, T t_end, ApproxType start_type, ApproxType bound_type) {
174   int h_count = x.size() - 1, tau_count = t.size() - 1;
175   grid_t<T> u(tau_count + 1, std::vector<T>(h_count + 1));
176   T h = (pde.end - pde.start) / h_count;
177   T tau = t_end / tau_count;
178
179   StartConditions(pde, x, t, u, tau, start_type);
180   Boundaries bound = BoundariesConditions(pde, x, t, u, h, tau, bound_type);
181
182   T alpha = (pde.a / h - pde.b / 2) / h,
183     beta = - 2 * pde.a / (h * h) + pde.c + (pde.d / 2 - 1 / tau) / tau,
184     gamma = (pde.a / h + pde.b / 2) / h;
185
186   TDMatrix<T> matrix(h_count+1);
187   for (int i = 1; i < h_count; ++i) {
188     matrix.a[i] = alpha;
189     matrix.b[i] = beta;
190     matrix.c[i] = gamma;
191   }
192
193   Vector<T> v(h_count+1);
194   for (int k = 1; k < tau_count; ++k) {
195     for (int i = 1; i < h_count; ++i) {
196       v[i] = (- 2 * u[k][i] + u[k-1][i]) / (tau * tau) - pde.f(x[i], t[k+1]) + pde.d
              * u[k-1][i] / (2 * tau);
197     }
198     v[0] = pde.gamma1(t[k+1]);
199     v[h_count] = pde.gamma2(t[k+1]);
200
201     matrix.b[0] = bound.left.alpha;
202     matrix.c[0] = bound.left.beta;
203     matrix.a[h_count] = bound.right.beta;
204     matrix.b[h_count] = bound.right.alpha;
205
206     if (bound_type == ApproxType::Quadratic) {
207       T coeff = -pde.alpha1 / (2 * h) / gamma;
208       matrix.b[0] -= coeff * alpha;
209       matrix.c[0] -= coeff * beta;
210       v[0] -= coeff * v[1];
211
212       coeff = pde.alpha2 / (2 * h) / alpha;
213       matrix.a[h_count] -= coeff * beta;
214       matrix.b[h_count] -= coeff * gamma;
215       v[h_count] -= coeff * v[h_count-1];
216
```

```
217       } else if (bound_type == ApproxType::Taylor) {
218         v[0] = v[0] * h * (1 - pde.b * h / (2 * pde.a)) + pde.alpha1 * h * h / (2 * pde
                .a) * (-pde.f(x[0], t[k+1]) + pde.d / tau * u[k][0] + (u[k-1][0] - 2 * u[k
                ][0]) / (tau * tau));
219         v[h_count] = v[h_count] * h * (-1 - pde.b * h / (2 * pde.a)) + pde.alpha2 * h *
                 h / (2 * pde.a) * (-pde.f(x[h_count], t[k+1]) + pde.d / tau * u[k][h_count
                ] + (u[k-1][h_count] - 2 * u[k][h_count]) / (tau * tau));
220       }
221
222       u[k+1] = matrix.Solve(v);
223     }
224     return u;
225   }
226 }
```

3 Решение краевой задачи для дифференциальных уравнений в частных производных эллиптического типа

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 результатов с приведенным в задании аналитическим решением $U(x, y)$ Исследовать зависимость погрешности от сеточных параметров h_x, h_y.

Вариант: 8

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2\frac{\partial u}{\partial x} - 3u$$
$$u(0, y) = \cos y$$
$$u(\frac{\pi}{2}, y) = 0$$
$$u(x, 0) = \exp(-x)\cos x$$
$$u(x, \frac{\pi}{2}) = 0$$
$$U(x, y) = \exp(-x)\cos x \cos y$$

2 Результаты работы

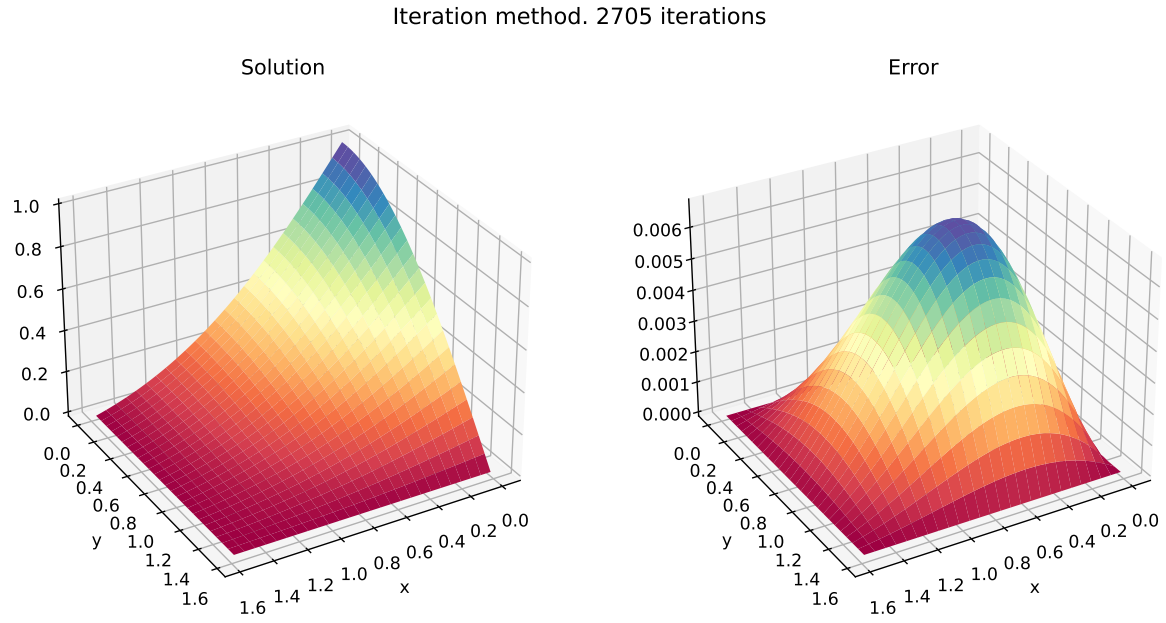


Рис. 5: Метод простых итераций

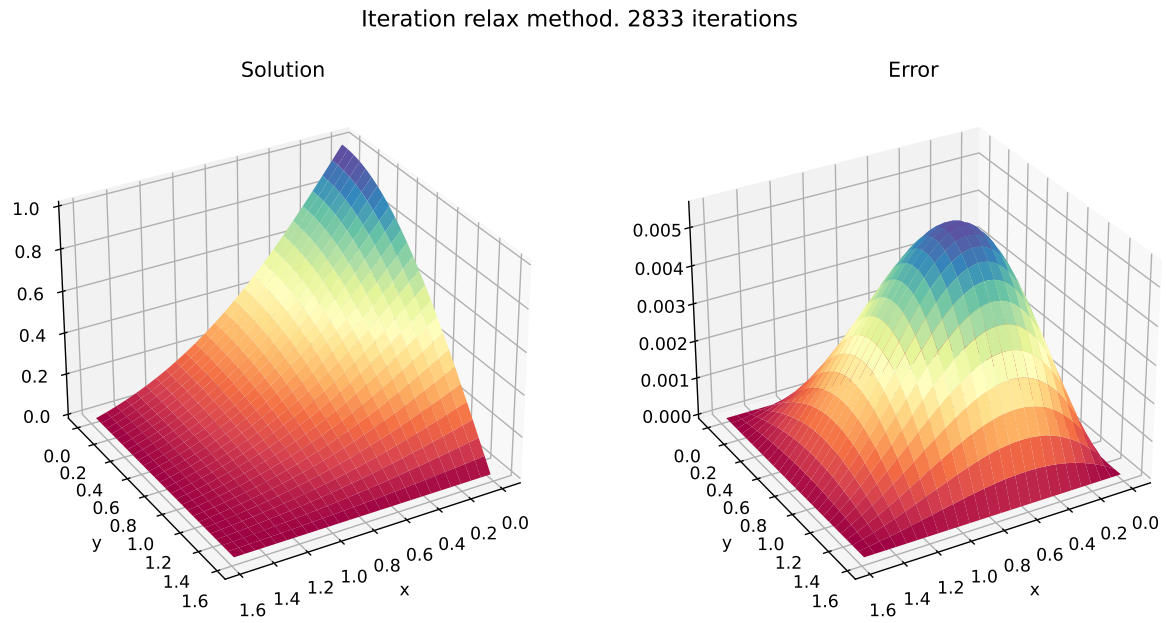


Рис. 6: Метод простых итераций с верхней релаксацией

Рис. 7: Метод Зейделя

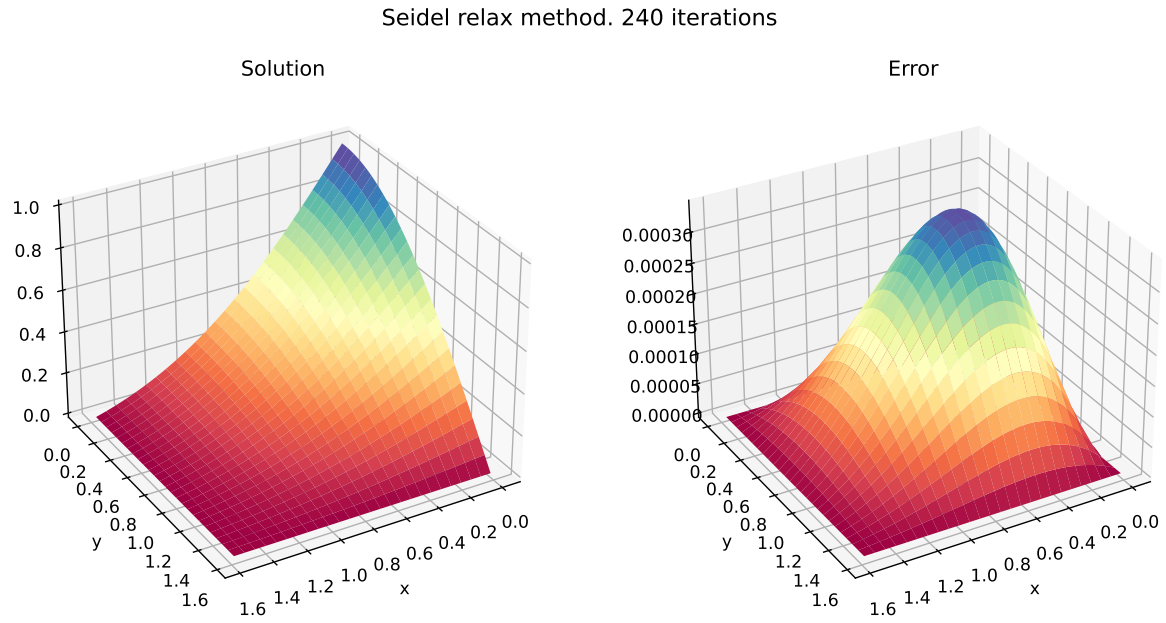


Рис. 8: Метод Зейделя с верхней релаксацией

3 Исходный код

```
#pragma once

#include <functional>
#include <vector>
#include <tuple>
#include <cmath>

#include "common.hpp"

namespace EllipticPDE {
  template <class T>
  using grid_t = std::vector<std::vector<T>>;

  template <class T>
  struct PDE {
    using f_x = std::function<T(T)>;
    using f_y = f_x;
    using f_x_y = std::function<T(T, T)>;

    T a, bx, by, c;
    f_x_y f;
    T x0, x1, y0, y1;
    T alpha_x0, beta_x0;
    f_y gamma_x0;
    T alpha_x1, beta_x1;
    f_y gamma_x1;
    T alpha_y0, beta_y0;
    f_x gamma_y0;
    T alpha_y1, beta_y1;
    f_x gamma_y1;
    f_x_y solution;

    PDE() = default;

    PDE(T a, T bx, T by, T c, f_x_y f, T x0, T x1, T y0, T y1,
        T alpha_x0, T beta_x0, f_y gamma_x0, T alpha_x1, T beta_x1, f_y gamma_x1,
        T alpha_y0, T beta_y0, f_x gamma_y0, T alpha_y1, T beta_y1, f_x gamma_y1, f_x_y
             solution) :
        a(a), bx(bx), by(by), c(c), f(f), x0(x0), x1(x1), y0(y0), y1(y1),
        alpha_x0(alpha_x0), beta_x0(beta_x0), gamma_x0(gamma_x0), alpha_x1(alpha_x1),
            beta_x1(beta_x1), gamma_x1(gamma_x1),
        alpha_y0(alpha_y0), beta_y0(beta_y0), gamma_y0(gamma_y0), alpha_y1(alpha_y1),
            beta_y1(beta_y1), gamma_y1(gamma_y1), solution(solution) {}

    PDE(T a, T bx, T by, T c, f_x_y f, T x0, T x1, T y0, T y1,
        T alpha_x0, T beta_x0, f_y gamma_x0, T alpha_x1, T beta_x1, f_y gamma_x1,
        T alpha_y0, T beta_y0, f_x gamma_y0, T alpha_y1, T beta_y1, f_x gamma_y1) :
```

```
45        a(a), bx(bx), by(by), c(c), f(f), x0(x0), x1(x1), y0(y0), y1(y1),
46        alpha_x0(alpha_x0), beta_x0(beta_x0), gamma_x0(gamma_x0), alpha_x1(alpha_x1),
              beta_x1(beta_x1), gamma_x1(gamma_x1),
47        alpha_y0(alpha_y0), beta_y0(beta_y0), gamma_y0(gamma_y0), alpha_y1(alpha_y1),
              beta_y1(beta_y1), gamma_y1(gamma_y1) {}
48
49    void SetEquation(T a_, T bx_, T by_, T c_, f_x_y f_) {
50      a = a_;
51      bx = bx_;
52      by = by_;
53      c = c_;
54      f = f_;
55    }
56
57    void SetBoundaries(T x0_, T x1_, T y0_, T y1_, T alpha_x0_, T beta_x0_, f_y
          gamma_x0_,
58                       T alpha_x1_, T beta_x1_, f_y gamma_x1_, T alpha_y0_, T beta_y0_,
                             f_x gamma_y0_,
59                       T alpha_y1_, T beta_y1_, f_x gamma_y1_) {
60      x0 = x0_; x1 = x1_; y0 = y0_; y1 = y1_;
61      alpha_x0 = alpha_x0_; beta_x0 = beta_x0_; gamma_x0 = gamma_x0_;
62      alpha_x1 = alpha_x1_; beta_x1 = beta_x1_; gamma_x1 = gamma_x1_;
63      alpha_y0 = alpha_y0_; beta_y0 = beta_y0_; gamma_y0 = gamma_y0_;
64      alpha_y1 = alpha_y1_; beta_y1 = beta_y1_; gamma_y1 = gamma_y1_;
65    }
66
67    void SetSolution(f_x_y solution_) {
68      solution = solution_;
69    }
70  };
71
72  template <class T>
73  inline T Relax(T old_value, T new_value, T relax) {
74    return old_value + relax * (new_value - old_value);
75  }
76
77  template <class T>
78  std::tuple<std::vector<T>, std::vector<T>, grid_t<T>, int>
79  IterationSolver(const PDE<T>& pde, int nx, int ny, T eps, double relax = 1) {
80    std::vector<T> x(nx + 1), y(ny + 1);
81    grid_t<T> u(nx + 1, std::vector<T>(ny + 1, 0)), next_u(nx + 1, std::vector<T>(ny +
          1, 0));
82
83    T hx = (pde.x1 - pde.x0) / nx;
84    T hy = (pde.y1 - pde.y0) / ny;
85
86    for (int i = 0; i <= ny; ++i) {
87      y[i] = pde.y0 + i * hy;
88    }
```

```
89      for (int i = 0; i <= nx; ++i) {
90        x[i] = pde.x0 + i * hx;
91      }
92
93      for (int i = 0; i <= nx; ++i) {
94        u[i][0] = pde.gamma_y0(x[i]) / (pde.beta_y0 - pde.alpha_y0 / hy);
95        u[i][ny] = pde.gamma_y1(x[i]) / (pde.beta_y1 + pde.alpha_y1 / hy);
96      }
97      for (int i = 0; i <= ny; ++i) {
98        u[0][i] = pde.gamma_x0(y[i]) / (pde.beta_x0 - pde.alpha_x0 / hx);
99        u[nx][i] = pde.gamma_x1(y[i]) / (pde.beta_x1 + pde.alpha_x1 / hx);
100     }
101
102     T eps_k;
103     int iter_count = 0;
104     T coeff = (2 * pde.a * (1.0 / hx / hx + 1.0 / hy / hy) + pde.c);
105     do {
106       eps_k = 0;
107       for (int i = 1; i < nx; ++i) {
108         for (int j = 1; j < ny; ++j) {
109           next_u[i][j] = Relax(u[i][j],
110                                (pde.a * ((u[i+1][j] + u[i-1][j]) / hx / hx + (u[i][j+1] + u
                                     [i][j-1]) / hy / hy) -
111                                 (pde.bx * (u[i+1][j] - u[i-1][j]) / hx + pde.by * (u[i][j
                                     +1] - u[i][j-1]) / hy) / 2 - pde.f(x[i], y[j])) / coeff
                                     ,
112                                 relax);
113
114           eps_k = std::max(eps_k, std::abs(next_u[i][j] - u[i][j]));
115         }
116       }
117
118       for (int i = 1; i < nx; ++i) {
119         next_u[i][0] = Relax(u[i][0], (pde.gamma_y0(x[i]) - pde.alpha_y0 / hy * next_u[
              i][1]) / (pde.beta_y0 - pde.alpha_y0 / hy), relax);
120         next_u[i][ny] = Relax(u[i][ny], (pde.gamma_y1(x[i]) + pde.alpha_y1 / hy *
              next_u[i][ny-1]) / (pde.beta_y1 + pde.alpha_y1 / hy), relax);
121
122         eps_k = std::max(eps_k, std::abs(next_u[i][0] - u[i][0]));
123         eps_k = std::max(eps_k, std::abs(next_u[i][ny] - u[i][ny]));
124       }
125       for (int i = 1; i < ny; ++i) {
126         next_u[0][i] = Relax(u[0][i], (pde.gamma_x0(y[i]) - pde.alpha_x0 / hx * next_u
              [1][i]) / (pde.beta_x0 - pde.alpha_x0 / hx), relax);
127         next_u[nx][i] = Relax(u[nx][i], (pde.gamma_x1(y[i]) + pde.alpha_x1 / hx *
              next_u[nx-1][i]) / (pde.beta_x1 + pde.alpha_x1 / hx), relax);
128
129         eps_k = std::max(eps_k, std::abs(next_u[0][i] - u[0][i]));
130         eps_k = std::max(eps_k, std::abs(next_u[nx][i] - u[nx][i]));
```

```
131       }
132
133       std::swap(next_u, u);
134
135       ++iter_count;
136
137     } while (eps_k > eps);
138
139     u[0][0] = (pde.gamma_y0(x[0]) - pde.alpha_y0 / hx * u[0][1]) / (pde.beta_y0 - pde.
          alpha_y0 / hx);
140     u[nx][0] = (pde.gamma_y0(x[nx]) - pde.alpha_y0 / hx * u[nx][1]) / (pde.beta_y0 -
          pde.alpha_y0 / hx);
141     u[0][ny] = (pde.gamma_y1(x[0]) + pde.alpha_y1 / hx * u[0][ny-1]) / (pde.beta_y1 +
          pde.alpha_y1 / hx);
142     u[nx][ny] = (pde.gamma_y1(x[nx]) + pde.alpha_y1 / hx * u[nx][ny-1]) / (pde.beta_y1+
           pde.alpha_y1 / hx);
143
144     return {x, y, u, iter_count};
145   }
146
147   template <class T>
148   std::tuple<std::vector<T>, std::vector<T>, grid_t<T>, int>
149   SeidelSolver(const PDE<T>& pde, int nx, int ny, T eps, double relax = 1) {
150     std::vector<T> x(nx + 1), y(ny + 1);
151     grid_t<T> u(nx + 1, std::vector<T>(ny + 1, 0));
152
153     T hx = (pde.x1 - pde.x0) / nx;
154     T hy = (pde.y1 - pde.y0) / ny;
155
156     for (int i = 0; i <= ny; ++i) {
157       y[i] = pde.y0 + i * hy;
158     }
159     for (int i = 0; i <= nx; ++i) {
160       x[i] = pde.x0 + i * hx;
161     }
162
163     for (int i = 0; i <= nx; ++i) {
164       u[i][0] = pde.gamma_y0(x[i]) / (pde.beta_y0 - pde.alpha_y0 / hy);
165       u[i][ny] = pde.gamma_y1(x[i]) / (pde.beta_y1 + pde.alpha_y1 / hy);
166     }
167     for (int i = 0; i <= ny; ++i) {
168       u[0][i] = pde.gamma_x0(y[i]) / (pde.beta_x0 - pde.alpha_x0 / hx);
169       u[nx][i] = pde.gamma_x1(y[i]) / (pde.beta_x1 + pde.alpha_x1 / hx);
170     }
171
172     T eps_k, next_u;
173     int iter_count = 0;
174     T coeff = (2 * pde.a * (1.0 / hx / hx + 1.0 / hy / hy) + pde.c);
175     do {
```

```
176       eps_k = 0;
177       for (int i = 1; i < nx; ++i) {
178         for (int j = 1; j < ny; ++j) {
179           next_u = Relax(u[i][j],
180                          (pde.a * ((u[i+1][j] + u[i-1][j]) / hx / hx + (u[i][j+1] + u[i][j
                                 -1]) / hy / hy) -
181                           (pde.bx * (u[i+1][j] - u[i-1][j]) / hx + pde.by * (u[i][j+1] - u
                                 [i][j-1]) / hy) / 2 - pde.f(x[i], y[j])) / coeff,
182                           relax);
183
184           eps_k = std::max(eps_k, std::abs(next_u - u[i][j]));
185           u[i][j] = next_u;
186         }
187       }
188
189       for (int i = 1; i < nx; ++i) {
190         next_u = Relax(u[i][0], (pde.gamma_y0(x[i]) - pde.alpha_y0 / hy * u[i][1]) / (
                 pde.beta_y0 - pde.alpha_y0 / hy), relax);
191         eps_k = std::max(eps_k, std::abs(next_u - u[i][0]));
192         u[i][0] = next_u;
193
194         next_u = Relax(u[i][ny], (pde.gamma_y1(x[i]) + pde.alpha_y1 / hy * u[i][ny-1])
                 / (pde.beta_y1 + pde.alpha_y1 / hy), relax);
195         eps_k = std::max(eps_k, std::abs(next_u - u[i][ny]));
196         u[i][ny] = next_u;
197       }
198       for (int i = 1; i < ny; ++i) {
199         next_u = Relax(u[0][i], (pde.gamma_x0(y[i]) - pde.alpha_x0 / hx * u[1][i]) / (
                 pde.beta_x0 - pde.alpha_x0 / hx), relax);
200         eps_k = std::max(eps_k, std::abs(next_u - u[0][i]));
201         u[0][i] = next_u;
202
203         next_u = Relax(u[nx][i], (pde.gamma_x1(y[i]) + pde.alpha_x1 / hx * u[nx-1][i])
                 / (pde.beta_x1 + pde.alpha_x1 / hx), relax);
204         eps_k = std::max(eps_k, std::abs(next_u - u[nx][i]));
205         u[nx][i] = next_u;
206       }
207
208       ++iter_count;
209
210     } while (eps_k > eps);
211
212     u[0][0] = (pde.gamma_y0(x[0]) - pde.alpha_y0 / hx * u[0][1]) / (pde.beta_y0 - pde.
              alpha_y0 / hx);
213     u[nx][0] = (pde.gamma_y0(x[nx]) - pde.alpha_y0 / hx * u[nx][1]) / (pde.beta_y0 -
              pde.alpha_y0 / hx);
214     u[0][ny] = (pde.gamma_y1(x[0]) + pde.alpha_y1 / hx * u[0][ny-1]) / (pde.beta_y1 +
              pde.alpha_y1 / hx);
```

```
215     u[nx][ny] = (pde.gamma_y1(x[nx]) + pde.alpha_y1 / hx * u[nx][ny-1]) / (pde.beta_y1+
            pde.alpha_y1 / hx);
216
217     return {x, y, u, iter_count};
218   }
219 }
```

4 Решение двумерной начально-краевой задачи для дифференциальных уравнений в частных производных параболического типа

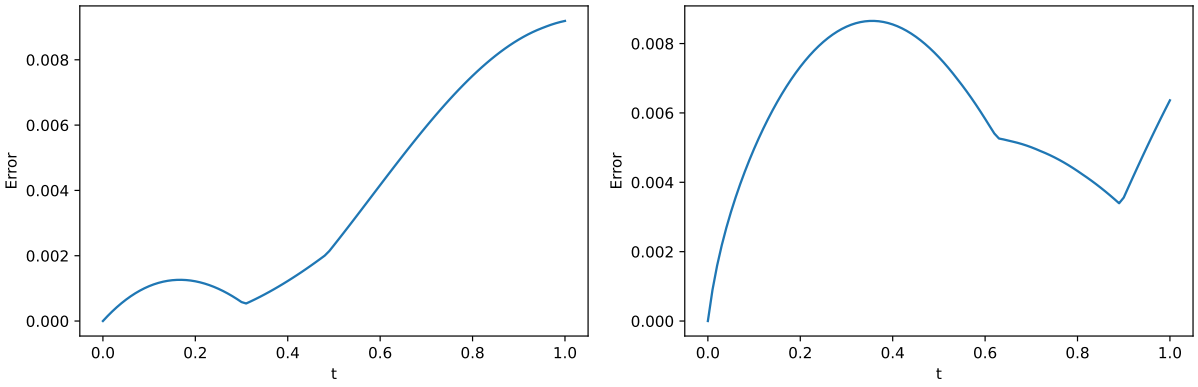
1 Постановка задачи

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.

Вариант: 8

$$\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial y^2} + \sin x \sin y(\mu\cos(\mu t) + (a+b)\sin(\mu t))$$
$$u(0,y,t) = 0$$
$$u(2\pi,y,t) = \sin y \sin(\mu t)$$
$$u(x,0,t) = 0$$
$$u(x,2\pi,t) = \sin x \sin(\mu t)$$
$$u(x,y,0) = 0$$
$$U(x,y) = \sin x \sin y \sin(\mu t)$$
$$a = 1,\ b = 1,\ \mu = 1$$

2 Результаты работы



(a) (b)

Рис. 9: Максимум погрешности решения в зависимости от времени для (a) метода переменных направлений и (b) метода дробных шагов

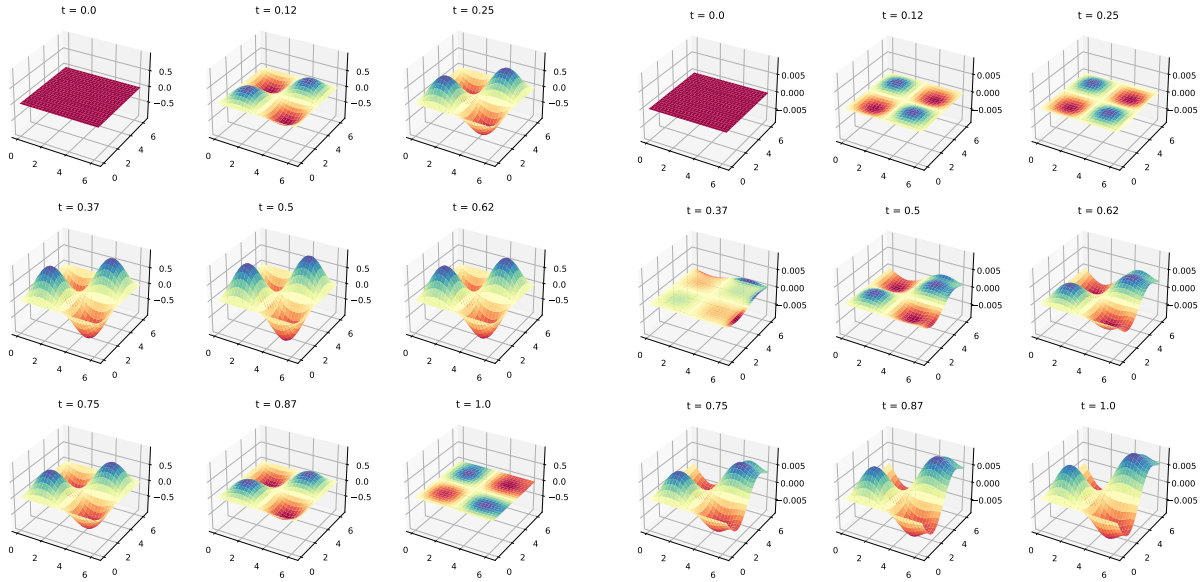


Рис. 10: Эволюция решения и погрешности для метода переменных направлений

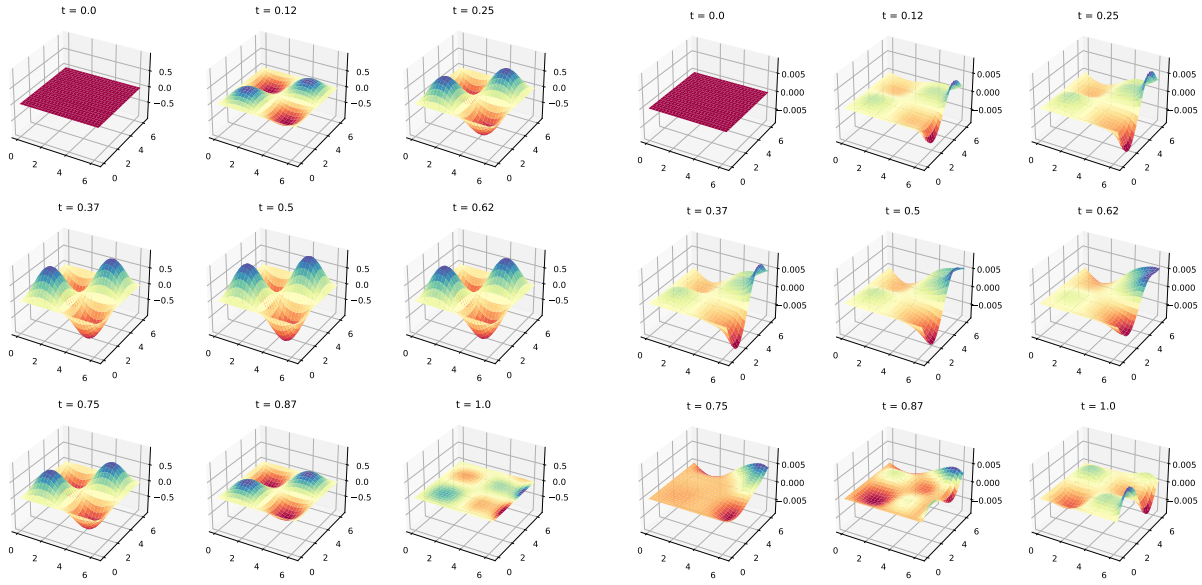


Рис. 11: Эволюция решения и погрешности для метода дробных шагов

3 Исходный код

```
#pragma once

#include <functional>
#include <vector>
#include <tuple>

#include "../linear/tridiagonal_matrix.hpp"
#include "../linear/vector.hpp"
#include "common.hpp"

template <int D, class T>
struct Tensor : public std::vector<Tensor<D - 1, T>> {
  static_assert(D >= 1, "Tensor dimension must be greater than zero");

  template <class... Args>
  Tensor(int size = 0, Args... args) : std::vector<Tensor<D - 1, T>>(size, Tensor<D -
      1, T>(args...)) {}
};

template <class T>
struct Tensor<1, T> : public std::vector<T> {
  Tensor(int size = 0, const T& value = T()): std::vector<T>(size, value) {}
};

```

```
24 namespace Parabolic2dPDE {
25   template <class T>
26   struct PDE {
27     using f_x_y = std::function<T(T, T)>;
28     using f_x_t = f_x_y;
29     using f_y_t = f_x_y;
30     using f_x_y_t = std::function<T(T, T, T)>;
31
32     T a, bx, by, c;
33     f_x_y_t f;
34     T x0, x1, y0, y1;
35     f_x_y psi;
36     T alpha_x0, beta_x0;
37     f_y_t gamma_x0;
38     T alpha_x1, beta_x1;
39     f_y_t gamma_x1;
40     T alpha_y0, beta_y0;
41     f_x_t gamma_y0;
42     T alpha_y1, beta_y1;
43     f_x_t gamma_y1;
44     f_x_y_t solution;
45
46     PDE() = default;
47
48     PDE(T a, T bx, T by, T c, f_x_y_t f, T x0, T x1, T y0, T y1, f_x_y psi) :
49         a(a), bx(bx), by(by), c(c), f(f), x0(x0), x1(x1), y0(y0), y1(y1), psi(psi) {}
50
51     PDE(T a, T bx, T by, T c, f_x_y_t f, T x0, T x1, T y0, T y1, f_x_y psi,
52         T alpha_x0, T beta_x0, f_y_t gamma_x0, T alpha_x1, T beta_x1, f_y_t gamma_x1,
53         T alpha_y0, T beta_y0, f_x_t gamma_y0, T alpha_y1, T beta_y1, f_x_t gamma_y1,
               f_x_y_t solution) :
54         a(a), bx(bx), by(by), c(c), f(f), x0(x0), x1(x1), y0(y0), y1(y1), psi(psi),
55         alpha_x0(alpha_x0), beta_x0(beta_x0), gamma_x0(gamma_x0), alpha_x1(alpha_x1),
               beta_x1(beta_x1), gamma_x1(gamma_x1),
56         alpha_y0(alpha_y0), beta_y0(beta_y0), gamma_y0(gamma_y0), alpha_y1(alpha_y1),
               beta_y1(beta_y1), gamma_y1(gamma_y1), solution(solution) {}
57
58     PDE(T a, T bx, T by, T c, f_x_y_t f, T x0, T x1, T y0, T y1, f_x_y psi,
59         T alpha_x0, T beta_x0, f_y_t gamma_x0, T alpha_x1, T beta_x1, f_y_t gamma_x1,
60         T alpha_y0, T beta_y0, f_x_t gamma_y0, T alpha_y1, T beta_y1, f_x_t gamma_y1) :
61         a(a), bx(bx), by(by), c(c), f(f), x0(x0), x1(x1), y0(y0), y1(y1), psi(psi),
62         alpha_x0(alpha_x0), beta_x0(beta_x0), gamma_x0(gamma_x0), alpha_x1(alpha_x1),
               beta_x1(beta_x1), gamma_x1(gamma_x1),
63         alpha_y0(alpha_y0), beta_y0(beta_y0), gamma_y0(gamma_y0), alpha_y1(alpha_y1),
               beta_y1(beta_y1), gamma_y1(gamma_y1) {}
64
65     void SetEquation(T a_, T bx_, T by_, T c_, f_x_y_t f_, T x0_, T x1_, T y0_, T y1_,
           f_x_y psi_) {
66       a = a_;
```

```
67      bx = bx_;
68      by = by_;
69      c = c_;
70      f = f_;
71      x0 = x0_; x1 = x1_; y0 = y0_; y1 = y1_;
72      psi = psi_;
73    }
74
75    void SetBoundaries(T alpha_x0_, T beta_x0_, f_y_t gamma_x0_,
76                       T alpha_x1_, T beta_x1_, f_y_t gamma_x1_,
77                       T alpha_y0_, T beta_y0_, f_x_t gamma_y0_,
78                       T alpha_y1_, T beta_y1_, f_x_t gamma_y1_) {
79      alpha_x0 = alpha_x0_; beta_x0 = beta_x0_; gamma_x0 = gamma_x0_;
80      alpha_x1 = alpha_x1_; beta_x1 = beta_x1_; gamma_x1 = gamma_x1_;
81      alpha_y0 = alpha_y0_; beta_y0 = beta_y0_; gamma_y0 = gamma_y0_;
82      alpha_y1 = alpha_y1_; beta_y1 = beta_y1_; gamma_y1 = gamma_y1_;
83    }
84
85    void SetSolution(f_x_y_t solution_) {
86      solution = solution_;
87    }
88  };
89
90  template <class T>
91  std::tuple<Tensor<3, T>, std::vector<T>, std::vector<T>, std::vector<T>>
92  AlternatingDirectionMethod(const PDE<T>& pde, int nx, int ny, int nt, T t_end) {
93    std::vector<T> x(nx + 1), y(ny + 1), t(nt + 1);
94    Tensor<3, T> u(nt + 1, nx + 1, ny + 1);
95    Tensor<2, T> p(nx + 1, ny + 1);
96
97    T hx = (pde.x1 - pde.x0) / nx;
98    T hy = (pde.y1 - pde.y0) / ny;
99    T tau = t_end / nt;
100
101   for (int i = 0; i <= nx; ++i) {
102     x[i] = pde.x0 + i * hx;
103   }
104   for (int i = 0; i <= ny; ++i) {
105     y[i] = pde.y0 + i * hy;
106   }
107   for (int i = 0; i <= nt; ++i) {
108     t[i] = i * tau;
109   }
110
111   for (size_t i = 0; i < x.size(); ++i) {
112     for (size_t j = 0; j < y.size(); ++j) {
113       u[0][i][j] = pde.psi(x[i], y[j]);
114     }
115   }
```

```
116
117     Vector<T> vx(nx+1), vy(ny+1);
118     TDMatrix<T> mx(nx+1), my(ny+1);
119
120     T alpha = (- pde.a / hx + pde.bx / 2) / hx,
121       beta = 2 * pde.a / hx / hx + 2 / tau - pde.c,
122       gamma = (- pde.a / hx - pde.bx / 2) / hx;
123     for (int i = 1; i < nx; ++i) {
124       mx.a[i] = alpha;
125       mx.b[i] = beta;
126       mx.c[i] = gamma;
127     }
128
129     alpha = (- pde.a / hy + pde.by / 2) / hy,
130     beta = 2 * pde.a / hy / hy + 2 / tau - pde.c,
131     gamma = (- pde.a / hy - pde.by / 2) / hy;
132     for (int j = 1; j < ny; ++j) {
133       my.a[j] = alpha;
134       my.b[j] = beta;
135       my.c[j] = gamma;
136     }
137
138     for (int k = 0; k < nt; ++k) {
139       for (int j = 1; j < ny; ++j) {
140         for (int i = 1; i < nx; ++i) {
141           vx[i] = u[k][i][j-1] * (pde.a / hy - pde.by / 2) / hy +
142                   2 * u[k][i][j] * (1 / tau - pde.a / hy / hy) +
143                   u[k][i][j+1] * (pde.a / hy + pde.by / 2) / hy +
144                   pde.f(x[i], y[j], t[k] + tau / 2);
145         }
146         vx[0] = pde.gamma_x0(y[j], t[k] + tau / 2);
147         vx[nx] = pde.gamma_x1(y[j], t[k] + tau / 2);
148
149         mx.b[0] = -pde.alpha_x0 / hx + pde.beta_x0;
150         mx.c[0] = pde.alpha_x0 / hx;
151
152         mx.a[nx] = -pde.alpha_x1 / hx;
153         mx.b[nx] = pde.alpha_x1 / hx + pde.beta_x1;
154
155         vx = mx.Solve(vx);
156
157         for (int i = 0; i <= nx; ++i) {
158           p[i][j] = vx[i];
159         }
160       }
161
162       for (int i = 0; i <= nx; ++i) {
163         p[i][0] = (pde.gamma_y0(x[i], t[k] + tau/2) - pde.alpha_y0 / hy * p[i][1]) / (-
                pde.alpha_y0 / hy + pde.beta_y0);
```

```
164       p[i][ny] = (pde.gamma_y1(x[i], t[k] + tau/2) + pde.alpha_y1 / hy * p[i][ny-1])
              / (pde.alpha_y1 / hy + pde.beta_y1);
165     }
166
167     for (int i = 1; i < nx; ++i) {
168       for (int j = 1; j < ny; ++j) {
169         vy[j] = p[i-1][j] * (pde.a / hx - pde.bx / 2) / hx +
170                 2 * p[i][j] * (1 / tau - pde.a / hx / hx) +
171                 p[i+1][j] * (pde.a / hx + pde.bx / 2) / hx +
172                 pde.f(x[i], y[j], t[k+1]);
173       }
174       vy[0] = pde.gamma_y0(x[i], t[k+1]);
175       vy[ny] = pde.gamma_y1(x[i], t[k+1]);
176
177       my.b[0] = -pde.alpha_y0 / hy + pde.beta_y0;
178       my.c[0] = pde.alpha_y0 / hy;
179
180       my.a[ny] = -pde.alpha_y1 / hy;
181       my.b[ny] = pde.alpha_y1 / hy + pde.beta_y1;
182
183       vy = my.Solve(vy);
184
185       for (int j = 0; j <= ny; ++j) {
186         u[k+1][i][j] = vy[j];
187       }
188     }
189
190     for (int j = 0; j <= ny; ++j) {
191       u[k+1][0][j] = (pde.gamma_x0(y[j], t[k+1]) - pde.alpha_x0 / hx * u[k+1][1][j])
              / (-pde.alpha_x0 / hx + pde.beta_x0);
192       u[k+1][nx][j] = (pde.gamma_x1(y[j], t[k+1]) + pde.alpha_x1 / hx * u[k+1][nx-1][
              j]) / (pde.alpha_x1 / hx + pde.beta_x1);
193     }
194   }
195
196   return {u, x, y, t};
197 }
198
199 template <class T>
200 std::tuple<Tensor<3, T>, std::vector<T>, std::vector<T>, std::vector<T>>
201 FractionalStepMethod(const PDE<T>& pde, int nx, int ny, int nt, T t_end) {
202   std::vector<T> x(nx + 1), y(ny + 1), t(nt + 1);
203   Tensor<3, T> u(nt + 1, nx + 1, ny + 1);
204   Tensor<2, T> p(nx + 1, ny + 1);
205
206   T hx = (pde.x1 - pde.x0) / nx;
207   T hy = (pde.y1 - pde.y0) / ny;
208   T tau = t_end / nt;
209
```

```
210     for (int i = 0; i <= nx; ++i) {
211       x[i] = pde.x0 + i * hx;
212     }
213     for (int i = 0; i <= ny; ++i) {
214       y[i] = pde.y0 + i * hy;
215     }
216     for (int i = 0; i <= nt; ++i) {
217       t[i] = i * tau;
218     }
219
220     for (size_t i = 0; i < x.size(); ++i) {
221       for (size_t j = 0; j < y.size(); ++j) {
222         u[0][i][j] = pde.psi(x[i], y[j]);
223       }
224     }
225
226     Vector<T> vx(nx+1), vy(ny+1);
227     TDMatrix<T> mx(nx+1), my(ny+1);
228
229     T alpha = (- pde.a / hx + pde.bx / 2) / hx,
230       beta = 2 * pde.a / hx / hx + 1 / tau - pde.c,
231       gamma = (- pde.a / hx - pde.bx / 2) / hx;
232     for (int i = 1; i < nx; ++i) {
233       mx.a[i] = alpha;
234       mx.b[i] = beta;
235       mx.c[i] = gamma;
236     }
237
238     alpha = (- pde.a / hy + pde.by / 2) / hy,
239     beta = 2 * pde.a / hy / hy + 1 / tau - pde.c,
240     gamma = (- pde.a / hy - pde.by / 2) / hy;
241     for (int j = 1; j < ny; ++j) {
242       my.a[j] = alpha;
243       my.b[j] = beta;
244       my.c[j] = gamma;
245     }
246
247     for (int k = 0; k < nt; ++k) {
248       for (int j = 1; j < ny; ++j) {
249         for (int i = 1; i < nx; ++i) {
250           vx[i] = u[k][i][j] / tau + pde.f(x[i], y[j], t[k]) / 2;
251         }
252         vx[0] = pde.gamma_x0(y[j], t[k+1]);
253         vx[nx] = pde.gamma_x1(y[j], t[k+1]);
254
255         mx.b[0] = -pde.alpha_x0 / hx + pde.beta_x0;
256         mx.c[0] = pde.alpha_x0 / hx;
257
258         mx.a[nx] = -pde.alpha_x1 / hx;
```

```
259         mx.b[nx] = pde.alpha_x1 / hx + pde.beta_x1;
260
261         vx = mx.Solve(vx);
262
263         for (int i = 0; i <= nx; ++i) {
264           p[i][j] = vx[i];
265         }
266       }
267
268       for (int i = 0; i <= nx; ++i) {
269         p[i][0] = (pde.gamma_y0(x[i], t[k+1]) - pde.alpha_y0 / hy * p[i][1]) / (-pde.
              alpha_y0 / hy + pde.beta_y0);
270         p[i][ny] = (pde.gamma_y1(x[i], t[k+1]) + pde.alpha_y1 / hy * p[i][ny-1]) / (pde
              .alpha_y1 / hy + pde.beta_y1);
271       }
272
273       for (int i = 1; i < nx; ++i) {
274         for (int j = 1; j < ny; ++j) {
275           vy[j] = p[i][j] / tau + pde.f(x[i], y[j], t[k+1]) / 2;
276         }
277         vy[0] = pde.gamma_y0(x[i], t[k+1]);
278         vy[ny] = pde.gamma_y1(x[i], t[k+1]);
279
280         my.b[0] = -pde.alpha_y0 / hy + pde.beta_y0;
281         my.c[0] = pde.alpha_y0 / hy;
282
283         my.a[ny] = -pde.alpha_y1 / hy;
284         my.b[ny] = pde.alpha_y1 / hy + pde.beta_y1;
285
286         vy = my.Solve(vy);
287
288         for (int j = 0; j <= ny; ++j) {
289           u[k+1][i][j] = vy[j];
290         }
291       }
292
293       for (int j = 0; j <= ny; ++j) {
294         u[k+1][0][j] = (pde.gamma_x0(y[j], t[k+1]) - pde.alpha_x0 / hx * u[k+1][1][j])
              / (-pde.alpha_x0 / hx + pde.beta_x0);
295         u[k+1][nx][j] = (pde.gamma_x1(y[j], t[k+1]) + pde.alpha_x1 / hx * u[k+1][nx-1][
              j]) / (pde.alpha_x1 / hx + pde.beta_x1);
296       }
297     }
298
299     return {u, x, y, t};
300   }
301 }
```