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:	А. Л. Ядров
Преподаватель:	Д. Е. Пивоваров
Группа:	М8О-408Б-20
Дата:	
Оценка:	
Подпись:	

Москва, 2024

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

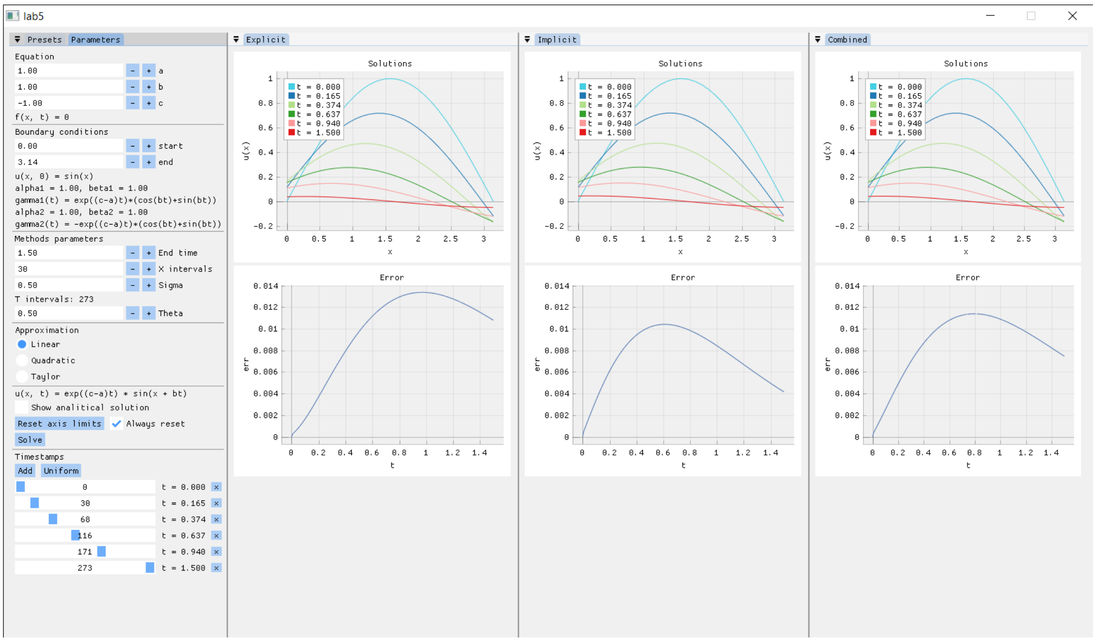

Рис. 1: Решение с аппроксимацией граничных условий с первым порядком

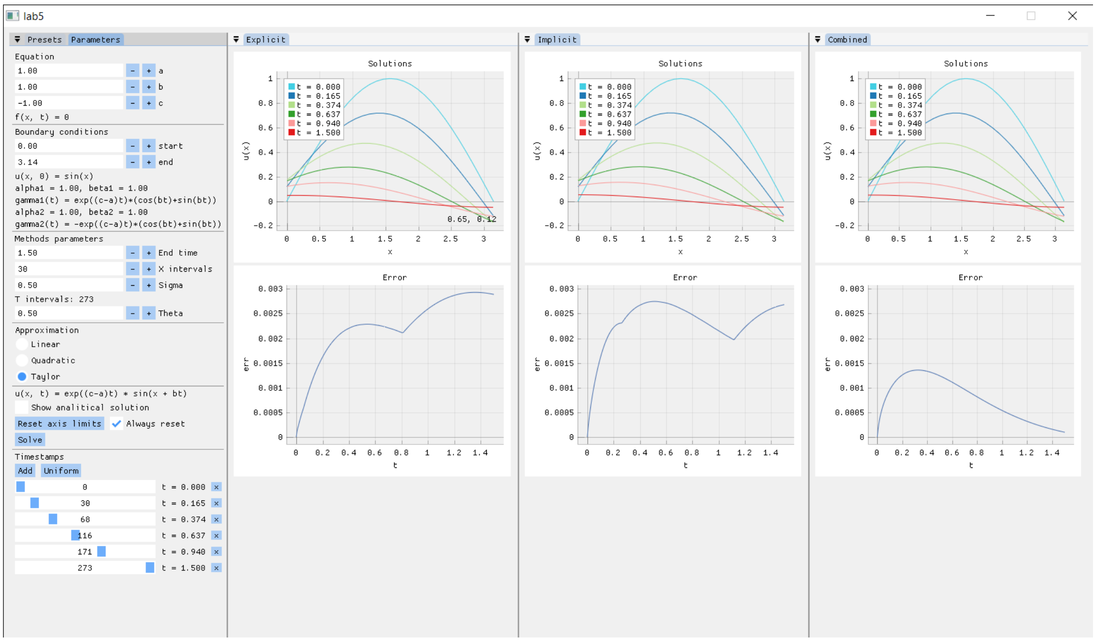

Рис. 2: Решение с аппроксимацией граничных условий со вторым порядком

3 Исходный код

common.hpp

```
#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_,
               T alpha2_, T beta2_, std::function<T(T)> gamma2_) {
    psi = psi_;
    start = start_;
    end = end_;
    alpha1 = alpha1_;
    beta1 = beta1_;
    gamma1 = gamma1_;
    alpha2 = alpha2_;
    beta2 = beta2_;
    gamma2 = gamma2_;
  }

  void SetSolution(std::function<T(T, T)> solution_) {
    solution = solution_;
  }
};

template <class T>
int CourantCondition(int h_count, double sigma, T t_end, T end, T a) {
  return t_end * a * h_count * h_count / (end * end * sigma);
}

template <class T>
std::tuple<std::vector<T>, std::vector<T>, grid_t<T>>
ExplicitSolver(const PDE<T>& pde, T t_end, int h_count, double sigma, ApproxType
    type) {
  auto [x, t] = GenerateGrid<T, PDE>(pde, t_end, h_count, sigma, CourantCondition<T>)
      ;
  return {x, t, ExplicitSolver(pde, x, t, t_end, type)};
}

template <class T>
grid_t<T> ExplicitSolver(const PDE<T>& pde, const std::vector<T>& x, const std::
    vector<T>& t, T t_end, ApproxType type) {
  int h_count = x.size() - 1, tau_count = t.size() - 1;
  grid_t<T> u(tau_count + 1, std::vector<T>(h_count + 1));
  T h = (pde.end - pde.start) / h_count;
  T tau = t_end / tau_count;

  for (int i = 0; i <= h_count; ++i) {
    u[0][i] = pde.psi(x[i]);
  }

  for (int k = 0; k < tau_count; ++k) {
    for (int i = 1; i < h_count; ++i) {
      T ddu = (u[k][i-1] - 2 * u[k][i] + u[k][i+1]) / (h * h);
      T du = (u[k][i+1] - u[k][i-1]) / (2 * h);
      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];
    }

    if (type == ApproxType::Linear) {
      u[k+1][0] = (pde.gamma1(t[k+1]) - pde.alpha1 / h * u[k+1][1]) / (-pde.alpha1 / h + pde.beta1);
      u[k+1][h_count] = (pde.gamma2(t[k+1]) + pde.alpha2 / h * u[k+1][h_count-1]) / (pde.alpha2 / h + pde.beta2);

    } else if (type == ApproxType::Quadratic) {
      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);
      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);

    } else if (type == ApproxType::Taylor) {
      T div = h - h * h * pde.b / (2 * pde.a),
        mult1 = (pde.c * h * h / (2 * pde.a) - 1 - h * h / (2 * pde.a * tau)),
        mult2 = h * h / (2 * pde.a);

      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]) /
                  (pde.alpha1 * mult1 / div + pde.beta1);

      div = -h - h * h * pde.b / (2 * pde.a);
      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]) /
                  (pde.alpha2 * mult1 / div + pde.beta2);
    }
  }
  return u;
}

template <class T>
std::tuple<std::vector<T>, std::vector<T>, grid_t<T>>
ImplicitSolver(const PDE<T>& pde, T t_end, int h_count, double sigma, ApproxType type) {
  auto [x, t] = GenerateGrid<T, PDE>(pde, t_end, h_count, sigma, CourantCondition<T>);
  return {x, t, ImplicitSolver(pde, x, t, t_end, type)};
}

template <class T>
grid_t<T> ImplicitSolver(const PDE<T>& pde, const std::vector<T>& x, const std::vector<T>& t, T t_end, ApproxType type) {
  int h_count = x.size() - 1, tau_count = t.size() - 1;
  grid_t<T> u(tau_count + 1, std::vector<T>(h_count + 1));
  T h = (pde.end - pde.start) / h_count;
```

```
  T tau = t_end / tau_count;

  for (int i = 0; i <= h_count; ++i) {
    u[0][i] = pde.psi(x[i]);
  }

  T alpha = (pde.a / h - pde.b / 2) * tau / h,
    beta = -1 - 2 * pde.a * tau / (h * h) + pde.c * tau,
    gamma = (pde.a / h + pde.b / 2) * tau / h;

  TDMatrix<T> matrix(h_count+1);
  for (int i = 1; i < h_count; ++i) {
    matrix.a[i] = alpha;
    matrix.b[i] = beta;
    matrix.c[i] = gamma;
  }

  Vector<T> v(h_count+1);
  for (int k = 0; k < tau_count; ++k) {
    for (int i = 1; i < h_count; ++i) {
      v[i] = -u[k][i] - tau * pde.f(x[i], t[k+1]);
    }
    v[0] = pde.gamma1(t[k+1]);
    v[h_count] = pde.gamma2(t[k+1]);

    if (type == ApproxType::Linear) {
      matrix.b[0] = -pde.alpha1 / h + pde.beta1;
      matrix.c[0] = pde.alpha1 / h;

      matrix.a[h_count] = -pde.alpha2 / h;
      matrix.b[h_count] = pde.alpha2 / h + pde.beta2;

    } else if (type == ApproxType::Quadratic) {
      T coeff = -pde.alpha1 / (2 * h) / gamma;
      matrix.b[0] = -3 * pde.alpha1 / (2 * h) + pde.beta1 - coeff * alpha;
      matrix.c[0] = 2 * pde.alpha1 / h - coeff * beta;
      v[0] -= coeff * v[1];

      coeff = pde.alpha2 / (2 * h) / alpha;
      matrix.a[h_count] = -2 * pde.alpha2 / h - coeff * beta;
      matrix.b[h_count] = 3 * pde.alpha2 / (2 * h) + pde.beta2 - coeff * gamma;
      v[h_count] -= coeff * v[h_count-1];

    } else if (type == ApproxType::Taylor) {
      T div = h - h * h * pde.b / (2 * pde.a),
        mult1 = (pde.c * h * h / (2 * pde.a) - 1 - h * h / (2 * pde.a * tau)),
        mult2 = h * h / (2 * pde.a);

      matrix.b[0] = pde.alpha1 * mult1 / div + pde.beta1;
```

```
      matrix.c[0] = pde.alpha1 / div;
      v[0] -= pde.alpha1 * mult2 / div * (u[k][0] / tau + pde.f(x[0], t[k+1]));

      div = -h - h * h * pde.b / (2 * pde.a);
      matrix.a[h_count] = pde.alpha2 / div;
      matrix.b[h_count] = pde.alpha2 * mult1 / div + pde.beta2;
      v[h_count] -= pde.alpha2 * mult2 / div * (u[k][h_count] / tau + pde.f(x[h_count], t[k+1]));
    }

    u[k+1] = matrix.Solve(v);
  }
  return u;
}

template <class T>
std::tuple<std::vector<T>, std::vector<T>, grid_t<T>>
CombinedSolver(const PDE<T>& pde, T t_end, int h_count, double sigma, ApproxType type, double theta) {
  auto [x, t] = GenerateGrid<T, PDE>(pde, t_end, h_count, sigma, CourantCondition<T>);
  return {x, t, CombinedSolver(pde, x, t, t_end, type, theta)};
}

template <class T>
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) {
  int h_count = x.size() - 1, tau_count = t.size() - 1;
  grid_t<T> u(tau_count + 1, std::vector<T>(h_count + 1));
  T h = (pde.end - pde.start) / h_count;
  T tau = t_end / tau_count;

  for (int i = 0; i <= h_count; ++i) {
    u[0][i] = pde.psi(x[i]);
  }

  T alpha = theta * (pde.a / h - pde.b / 2) * tau / h,
    beta = -1 + theta * (-2 * pde.a * tau / (h * h) + pde.c * tau),
    gamma = theta * (pde.a / h + pde.b / 2) * tau / h;

  TDMatrix<T> matrix(h_count+1);
  for (int i = 1; i < h_count; ++i) {
    matrix.a[i] = alpha;
    matrix.b[i] = beta;
    matrix.c[i] = gamma;
  }

  Vector<T> v(h_count+1);
  for (int k = 0; k < tau_count; ++k) {
```

```
      for (int i = 1; i < h_count; ++i) {
        T ddu = (u[k][i-1] - 2 * u[k][i] + u[k][i+1]) / (h * h);
        T du = (u[k][i+1] - u[k][i-1]) / (2 * h);
        T uik = pde.a * ddu + pde.b * du + pde.c * u[k][i] + pde.f(x[i], t[k]);
        v[i] = -u[k][i] - tau * (theta * pde.f(x[i], t[k]) + (1 - theta) * uik);
      }
      v[0] = pde.gamma1(t[k+1]);
      v[h_count] = pde.gamma2(t[k+1]);

      if (type == ApproxType::Linear) {
        matrix.b[0] = -pde.alpha1 / h + pde.beta1;
        matrix.c[0] = pde.alpha1 / h;

        matrix.a[h_count] = -pde.alpha2 / h;
        matrix.b[h_count] = pde.alpha2 / h + pde.beta2;

      } else if (type == ApproxType::Quadratic) {
        T coeff = -pde.alpha1 / (2 * h) / gamma;
        matrix.b[0] = -3 * pde.alpha1 / (2 * h) + pde.beta1 - coeff * alpha;
        matrix.c[0] = 2 * pde.alpha1 / h - coeff * beta;
        v[0] -= coeff * v[1];

        coeff = pde.alpha2 / (2 * h) / alpha;
        matrix.a[h_count] = -2 * pde.alpha2 / h - coeff * beta;
        matrix.b[h_count] = 3 * pde.alpha2 / (2 * h) + pde.beta2 - coeff * gamma;
        v[h_count] -= coeff * v[h_count-1];

      } else if (type == ApproxType::Taylor) {
        T div = h - h * h * pde.b / (2 * pde.a),
          mult1 = (pde.c * h * h / (2 * pde.a) - 1 - h * h / (2 * pde.a * tau)),
          mult2 = h * h / (2 * pde.a);

        matrix.b[0] = pde.alpha1 * mult1 / div + pde.beta1;
        matrix.c[0] = pde.alpha1 / div;
        v[0] -= pde.alpha1 * mult2 / div * (u[k][0] / tau + pde.f(x[0], t[k+1]));

        div = -h - h * h * pde.b / (2 * pde.a);
        matrix.a[h_count] = pde.alpha2 / div;
        matrix.b[h_count] = pde.alpha2 * mult1 / div + pde.beta2;
        v[h_count] -= pde.alpha2 * mult2 / div * (u[k][h_count] / tau + pde.f(x[h_count], t[k+1]));
      }

      u[k+1] = matrix.Solve(v);
    }
    return u;
  }
}
```