

**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 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  $\tau, h$ .

**Вариант: 2**

$$\frac{\partial^2 u}{\partial t^2} = a^2 \frac{\partial^2 u}{\partial x^2}, \quad a^2 > 0$$

$$u_x(0, t) - u(0, t) = 0$$

$$u_x(\pi, t) - u(\pi, t) = 0$$

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

$$u_t(x, 0) = -a(\sin x + \cos x)$$

$$U(x, t) = \sin(x - at) + \cos(x + at)$$

## 2 Результаты работы



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



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

### 3 Исходный код

```
1  #pragma once
2
3  #include <functional>
4  #include <vector>
5  #include <tuple>
6
7  #include "../linear/tridiagonal_matrix.hpp"
8  #include "../linear/vector.hpp"
9  #include "common.hpp"
10
11 namespace HyperbolicPDE {
12     template <class T>
13         using grid_t = std::vector<std::vector<T>>>;
14
15     template <class T>
16     struct PDE {
17         using f_t = std::function<T(T)>;
18         using f_x = f_t;
19         using f_x_t = std::function<T(T, T)>;
20
21         T a, b, c, d;
22         f_x_t f;
23         f_x psi1, dpsi1, d2psi1, psi2;
24         T start, end;
25         T alpha1, beta1;
26         f_t gamma1;
27         T alpha2, beta2;
28         f_t gamma2;
29         f_x_t solution;
30
31         PDE() = default;
32
33         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
34             , T end,
35             T alpha1, T beta1, f_t gamma1, T alpha2, T beta2, f_t gamma2, f_x_t solution) :
36             a(a), b(b), c(c), d(d), f(f), psi1(psi1), dpsi1(dpsi1), d2psi1(d2psi1), psi2(
37                 psi2), start(start), end(end), alpha1(alpha1), beta1(beta1), gamma1(gamma1)
38             ,
39             alpha2(alpha2), beta2(beta2), gamma2(gamma2), solution(solution) {}
40
41         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
42             , T end,
43             T alpha1, T beta1, f_t gamma1, T alpha2, T beta2, f_t gamma2) :
44             a(a), b(b), c(c), d(d), f(f), psi1(psi1), dpsi1(dpsi1), d2psi1(d2psi1), psi2(
45                 psi2), start(start), end(end), alpha1(alpha1), beta1(beta1), gamma1(gamma1)
46             ,
47             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_,
55     T alpha1_, T beta1_, std::function<T(T)> gamma1_,
56     T alpha2_, T beta2_, std::function<T(T)> gamma2_) {
57     psi1 = psi1_;
58     dpsi1 = dpsi1_;
59     d2psi1 = d2psi1_;
60     psi2 = psi2_;
61     start = start_;
62     end = end_;
63     alpha1 = alpha1_;
64     beta1 = beta1_;
65     gamma1 = gamma1_;
66     alpha2 = alpha2_;
67     beta2 = beta2_;
68     gamma2 = gamma2_;
69 }
70
71 void SetSolution(f_x_t solution_) {
72     solution = solution_;
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>
82     & t, grid_t<T>& u, T tau, ApproxType type) {
83     for (size_t i = 0; i < x.size(); ++i) {
84         u[0][i] = pde.psi1(x[i]);
85         if (type == ApproxType::Linear) {
86             u[1][i] = u[0][i] + tau * pde.psi2(x[i]);
87         } else { // Taylor
88             u[1][i] = tau * (1 + tau * pde.d / 2) * pde.psi2(x[i]) + u[0][i] + tau * tau /
89                 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 | }

```