

**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

$$\frac{\partial u}{\partial t} = a \frac{\partial^2 u}{\partial x^2} + b \frac{\partial u}{\partial x} + cu, \quad a > 0, \quad b > 0, \quad c < 0$$

$$u_x(0, t) + u(0, t) = \exp((c - a)t)(\cos(bt) + \sin(bt))$$

$$u_x(\pi, t) + u(\pi, t) = -\exp((c - a)t)(\cos(bt) + \sin(bt))$$

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

$$U(x, t) = \exp((c - a)t) \sin(x + bt)$$

2 Результаты работы

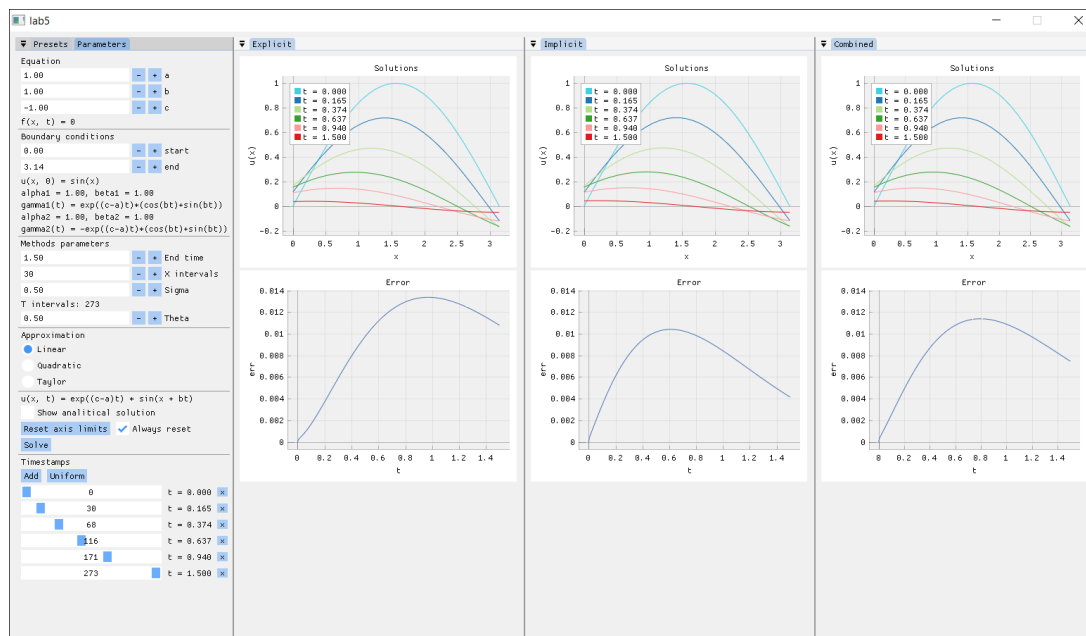


Рис. 1: Решение с аппроксимацией граничных условий с первым порядком

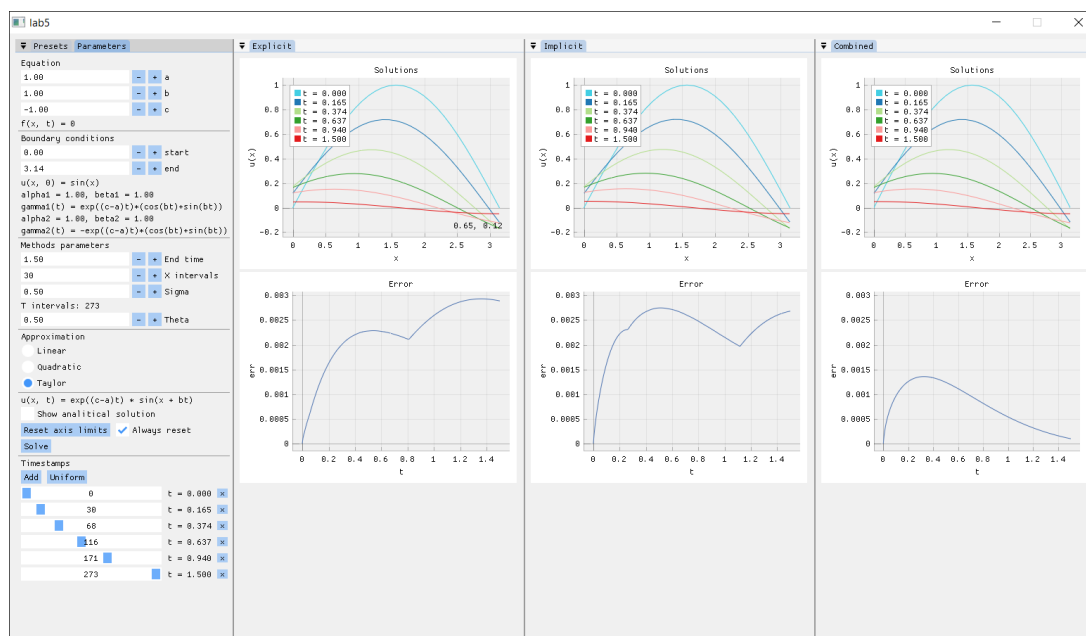


Рис. 2: Решение с аппроксимацией граничных условий со вторым порядком

3 Исходный код

common.hpp

```
1  #pragma once
2
3  #include <tuple>
4  #include <vector>
5  #include <functional>
6
7  enum class ApproxType : int {
8      Linear,
9      Quadratic,
10     Taylor
11 };
12
13 template <class T, template<class> class PDE>
14 std::tuple<std::vector<T>, std::vector<T>>
15 GenerateGrid(const PDE<T>& pde, T t_end, int h_count, double sigma, const std::
    function<T(int, double, T, T, T)>& CourantCondition) {
16     int tau_count = CourantCondition(h_count, sigma, t_end, pde.end - pde.start, pde.a);
17     std::vector<T> x(h_count + 1), t(tau_count + 1);
18     T h = (pde.end - pde.start) / h_count;
19     T tau = t_end / tau_count;
20
21     for (int i = 0; i <= h_count; ++i) {
22         x[i] = pde.start + h * i;
23     }
24     for (int k = 0; k <= tau_count; ++k) {
25         t[k] = tau * k;
26     }
27
28     return {x, t};
29 }
30
31 template <class T>
32 struct Boundaries {
33     struct Coeffs {
34         T alpha, beta;
35     };
36
37     Coeffs left, right;
38 };
```

parabolic_pde.hpp

```

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 ParabolicPDE {
12     template <class T>
13     using grid_t = std::vector<std::vector<T>>>;
14
15     template <class T>
16     struct PDE {
17         T a, b, c;
18         std::function<T(T, T)> f;
19         std::function<T(T)> psi;
20         T start, end;
21         T alpha1, beta1;
22         std::function<T(T)> gamma1;
23         T alpha2, beta2;
24         std::function<T(T)> gamma2;
25         std::function<T(T, T)> solution;
26
27         PDE() = default;
28
29         PDE(T a, T b, T c, std::function<T(T, T)> f, std::function<T(T)> psi, T start, T
            end,
30             T alpha1, T beta1, std::function<T(T)> gamma1, T alpha2, T beta2, std::function
                <T(T)> gamma2) :
31             a(a), b(b), c(c), f(f), psi(psi), start(start), end(end), alpha1(alpha1), beta1
                (beta1), gamma1(gamma1),
32             alpha2(alpha2), beta2(beta2), gamma2(gamma2) {}
33
34         PDE(T a, T b, T c, std::function<T(T, T)> f) :
35             a(a), b(b), c(c), f(f) {}
36
37         void SetEquation(T a_, T b_, T c_, std::function<T(T, T)> f_) {
38             a = a_;
39             b = b_;
40             c = c_;
41             f = f_;
42         }
43
44         void SetBoundaries(std::function<T(T)> psi_, T start_, T end_, T alpha1_, T beta1_,

```

```

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

```

```

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

```

```

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 }

```