Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа 3 по курсу «Численные методы»

Студент: В. С. Нелюбин
Группа: М8О-408Б

Москва, 2024

Лабораторная работа 3

Задача: 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 . Исследовать зависимость погрешности от сеточных параметров . .

9.

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

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

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

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

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

Аналитическое решение: $U(x, y) = \exp(-y)\cos x \cos y$.

1 Исходный код

Программа хранит двухмерную сетку с координатами X Y, и постепенно приближает начальное состояние к правильному решению. Цикл заканчиается, если расхождение решений достаточно мало, либо количество шагов превысило 100. Есть несколько функций расчёта, для каждого из методов. Условия задачи и параметры сетки вынесены как константы и отдельные функции по возможности.

```
1  #include <iostream>
2  #include <cmath>
3  #include <vector>
4
5  const double MAX_X = M_PI / 2;
6  const double MAX_Y = M_PI / 2;
7
8  double phi_X0(double y)
9  {
10     return exp(-y) * cos(y);
11 }
12
13 double phi_XL(double y)
14 {
15     return 0;
16 }
17
18 double phi_Y0(double x)
19 {
20     return cos(x);
21 }
22
23 double phi_YL(double t)
24 {
25     return 0;
26 }
27
28 double correct(double x, double y)
29 {
30     return exp(-y) * cos(x) * cos(y);
31 }
32
33 std::vector<double> progon(std::vector<double> ar, std::vector<double> br, std::vector
       <double> cr, std::vector<double> dr)
34 {
35     // printf("%d %d %d %d\n", ar.size(), br.size(), cr.size(), dr.size());
36     int len = br.size();
37     if (dr.size() != len)
38     {
```

```
39          throw std::invalid_argument("bad array dimensions");
40      }
41      if (ar.size() == len)
42      {
43          if (ar[0] != 0)
44          {
45              throw std::invalid_argument("a[0] must be 0");
46          }
47      }
48      else if (ar.size() + 1 == len)
49      {
50          ar.insert(ar.begin(), 0);
51      }
52      else
53      {
54          throw std::invalid_argument("bad array dimensions");
55      }
56      if (cr.size() == len)
57      {
58          if (cr[len - 1] != 0)
59          {
60              throw std::invalid_argument("c[LAST] must be 0");
61          }
62      }
63      else if (cr.size() + 1 == len)
64      {
65          cr.insert(ar.end(), 0);
66      }
67      else
68      {
69          throw std::invalid_argument("bad array dimensions");
70      }
71      std::vector<double> P;
72      std::vector<double> Q;
73      for (int i = 0; i < len; i++)
74      {
75          double help = br[i];
76          if (i > 0)
77          {
78              help += ar[i] * P[i - 1];
79          }
80          P.push_back(-cr[i] / help);
81          double help2 = dr[i];
82          if (i > 0)
83          {
84              help2 -= ar[i] * Q[i - 1];
85          }
86          Q.push_back(help2 / help);
87      }
```

```
88
89      // for (int i = 0; i < len; i++)
90      // {
91      // printf("P[%d] = %lf\tQ[%d] = %lf\n", i, P[i], i, Q[i]);
92      // }
93
94      std::vector<double> result(len, Q[len - 1]);
95      for (int i = len - 2; i >= 0; i--)
96      {
97          result[i] = P[i] * result[i + 1] + Q[i];
98      }
99      return result;
100 }
101
102 /// DECISION: i w IS X AND j h IS Y
103 class griddy
104 {
105     int w;
106     int h;
107     double hx;
108     double hy;
109     std::vector<double> vals;
110
111     void process_arguments(int i, int j)
112     {
113         if ((i < 0) || (i >= w))
114         {
115             throw std::invalid_argument("index i outside of range");
116         }
117         if ((j < 0) || (j >= h))
118         {
119             throw std::invalid_argument("index j outside of range");
120         }
121     }
122
123 public:
124     griddy(int width, int height)
125     {
126         w = width;
127         h = height;
128         hx = MAX_X / (w - 1);
129         hy = MAX_Y / (h - 1);
130         vals = std::vector<double>(w * h);
131     }
132     griddy(griddy &other)
133     {
134         w = other.w;
135         h = other.h;
136         hx = other.hx;
```

```
137         hy = other.hy;
138         vals = std::vector<double>(w * h);
139         for (int i = 0; i < w * h; i++)
140         {
141             vals[i] = other.vals[i];
142         }
143     }
144     void annul()
145     {
146         for (int i = 0; i < vals.size(); i++)
147         {
148             vals[i] = 0;
149         }
150     }
151     double krant()
152     {
153         return pow(1 / hx, 2) * hy;
154     }
155     double *U_mut(int i, int j)
156     {
157         process_arguments(i, j);
158         int idx = i * h + j;
159         return &vals[idx];
160     }
161     double U(int i, int j)
162     {
163         return *U_mut(i, j);
164     }
165     void print()
166     {
167         // for (int i = 0; i < vals.size(); i++)
168         // {
169         // printf("%3.0lf ", vals[i]);
170         // }
171         // printf("\n");
172 
173         // for (int i = 0; i < w; i++)
174         // {
175         for (int j = 0; j < h; j++)
176         {
177             for (int i = 0; i < w; i++)
178             {
179                 printf("%8.4lf ", U(i, j));
180             }
181             printf("\n");
182         }
183     }
184     void cheat_set_correct()
185     {
```

```
186         for (int i = 0; i < w; i++)
187         {
188             for (int j = 0; j < h; j++)
189             {
190                 *U_mut(i, j) = correct(i * hx, j * hy);
191             }
192         }
193     }
194     void set_start()
195     {
196 
197         for (int i = 0; i < w; i++)
198         {
199             *U_mut(i, 0) = phi_Y0(i * hx);
200             *U_mut(i, h - 1) = phi_YL(i * hx);
201         }
202         for (int j = 0; j < h; j++)
203         {
204             *U_mut(0, j) = phi_X0(j * hy);
205             *U_mut(w - 1, j) = phi_XL(j * hy);
206         }
207     }
208     double square_error()
209     {
210         double result = 0;
211         for (int i = 0; i < vals.size(); i++)
212         {
213             result += pow(vals[i], 2);
214         }
215         return sqrt(result);
216     }
217     double n_row_error(int j)
218     {
219         double res = 0;
220         for (int i = 0; i < w; i++)
221         {
222             res += pow(U(i, j), 2);
223         }
224         return sqrt(res);
225     }
226     griddy diff(griddy &other)
227     {
228         if ((other.w != w) || (other.h != h))
229         {
230             throw std::invalid_argument("dimensions do not allign");
231         }
232         griddy rez = griddy(w, h);
233         for (int i = 0; i < w; i++)
234         {
```

```
235             for (int j = 0; j < h; j++)
236             {
237                 *rez.U_mut(i, j) = U(i, j) - other.U(i, j);
238             }
239         }
240         return rez;
241     }
242 
243     void lerp()
244     {
245         for (int i = 1; i < w - 1; i++)
246         {
247             double low_x = U(0, 0) * (w - i) / (w + 1);
248             low_x += U(w - 1, 0) * (i + 1) / (w + 1);
249             double high_x = U(0, h - 1) * (w - i) / (w + 1);
250             high_x += U(w - 1, h - 1) * (i + 1) / (w + 1);
251             for (int j = 1; j < h - 1; j++)
252             {
253                 double val = low_x * (h - j) / (h + 1);
254                 val += high_x * (j + 1) / (h + 1);
255                 *U_mut(i, j) = val;
256             }
257         }
258     }
259 
260     void liebman()
261     {
262         std::vector<double> new_vals(w * h);
263         for (int i = 1; i < w - 1; i++)
264         {
265             for (int j = 1; j < h - 1; j++)
266             {
267                 double p1 = hy * hy * (U(i + 1, j) + U(i - 1, j));
268                 double p2 = hx * hx * (U(i, j + 1) + U(i, j - 1));
269                 double p3 = hx * hx * hy * (U(i, j + 1) - U(i, j - 1));
270                 double p4k = 3 * hx * hx * hy * hy;
271                 double k1 = 2 * hy * hy;
272                 double k2 = 2 * hx * hx;
273                 double all_k = k1 + k2;
274                 double d = p1 + p2 + p3 + p4k * U(i, j);
275                 new_vals[i * w + j] = d / all_k;
276             }
277         }
278         for (int i = 1; i < w - 1; i++)
279         {
280             for (int j = 1; j < h - 1; j++)
281             {
282                 *U_mut(i, j) = new_vals[i * w + j];
283             }
```

```
284         }
285     }
286 
287     void zeidel()
288     {
289         for (int i = 1; i < w - 1; i++)
290         {
291             for (int j = 1; j < h - 1; j++)
292             {
293                 double p1 = hy * hy * (U(i + 1, j) + U(i - 1, j));
294                 double p2 = hx * hx * (U(i, j + 1) + U(i, j - 1));
295                 double p3 = hx * hx * hy * (U(i, j + 1) - U(i, j - 1));
296                 double p4k = 3 * hx * hx * hy * hy;
297                 double k1 = 2 * hy * hy;
298                 double k2 = 2 * hx * hx;
299                 double all_k = k1 + k2;
300                 double d = p1 + p2 + p3 + p4k * U(i, j);
301                 *U_mut(i, j) = d / all_k;
302             }
303         }
304     }
305 };
306 
307 double const ERROR_MARGIN = 0.0001;
308 
309 double shmain(int w, int h)
310 {
311     griddy x = griddy(w, h);
312     // printf("kurant %lf\n", x.krant());
313     x.annul();
314     // *x.U_mut(0, 0) = -1;
315     // *x.U_mut(w - 1, h - 1) = 1;
316     // x.cheat_set_correct();
317     x.set_start();
318     x.lerp();
319     int j;
320     for (j = 1; j < 100; j++)
321     {
322         griddy past_x = griddy(x);
323         // x.liebman();
324         x.zeidel();
325         past_x = past_x.diff(x);
326         if (past_x.square_error() < ERROR_MARGIN)
327         {
328             break;
329         }
330     }
331     printf("took %d iterations\n", j);
332     griddy y = griddy(x);
```

```
    y.cheat_set_correct();
    griddy z = x.diff(y);
    // z.print();
    // for (int j = 0; j < h; j++)
    // {
    // printf("%lf\n", z.n_row_error(j));
    // }
    return z.square_error();
}

int main()
{
    int w = 10;
    int h = 10;
    double er = shmain(w, h);
    printf("error is %lf\n", er);
    return 0;
}
```

2 Результаты

Поскольку программа завершает работу только тогда, когда ошибка слете достаточно малой, то имеет смысл не ошибка в конце вычислений, а количество итераций, за которое удалось прийти к решению. Для параметроа <10,10> (первое - шаги x, второй число - y) - 66 итераций
<20,10> - 147 (ограничение было снято)
<50,10> - 588
<20,10> - 151
<20,20> - 230
<20,50> - 694
<50,50> - 1149