

Metody numeryczne

Wojciech Chrobak

20 listopada 2017

Zadanie 2 - obowiązkowe

Kod

```
1  #include <iostream>
2  #include <cmath>
3  #include <vector>
4  #include <iomanip>
5
6  using namespace std;
7
8
9  const int sizeMatrix = 128;
10 double D = 4;
11 double E = 1;
12 double F = 1;
13 double G = 1;
14 double H = 1;
15
16 vector<double> b;
17
18 void printX(vector<double> A) {
19     for (int i = 0; i < A.size(); i++) {
20         cout << A[i] << " \\\\n";
21     }
22     cout << endl;
23 }
24
25 double norm(vector<double> u) {
26     double a = 0;
27     double norm;
28     for (int i = 0; i < u.size(); ++i) {
29         a += u[i] * u[i];
30     }
31     norm = sqrt(a);
32     return norm;
33 }
34
35 vector<double> subtract(vector<double> A, vector<double> B) {
36     vector<double> res;
37     res.assign(A.size(), 0);
38
39     for (int i = 0; i < res.size(); i++) {
40         res[i] = A[i] - B[i];
41     }
42
43     return res;
44 }
45
```

```

46 double multiVV(vector<double> A, vector<double> B) {
47     double res = 0;
48     for (int i = 0; i < A.size(); i++) {
49         res = res + A[i] * B[i];
50     }
51     return res;
52 }
53
54 // metoda Gaussa-Seidela
55 vector<double> GS() {
56     vector<double> x;
57     x.assign(sizeMatrix, 0);
58
59     vector<double> temp_x;
60     temp_x.assign(sizeMatrix, 0);
61
62     //////////////////////////////////////
63
64     int counter = 0;
65     while(true) {
66
67         temp_x = x; // poprzednie wartosci przyblizen
68
69         for (int i = 0; i < sizeMatrix; ++i) {
70             double sumL = 0;
71             double sumR = 0;
72             // lewa strona sumy
73             if(i > 0) {
74                 sumL = F * x[i-1];
75                 if(i >= 4) {
76                     sumL = sumL + H * x[i-4];
77                 }
78             }
79
80             // prawa strona sumy
81             if(i < sizeMatrix-1) {
82                 sumR = E * temp_x[i+1];
83                 if(i < sizeMatrix-4) {
84                     sumR = sumR + G * temp_x[i+4];
85                 }
86             }
87
88             x[i] = (b[i] - sumL - sumR)/D;
89         }
90         if( norm(subtract(x, temp_x)) < 0.000001)
91             break;
92         counter++;
93     }
94     return x;
95 }
96
97 // metoda gradientow sprzezonych
98 vector<double> CG() {
99     vector<double> x;
100     x.assign(sizeMatrix, 0);
101
102     vector<double> temp_x;
103     temp_x.assign(sizeMatrix, 0);
104
105     vector<double> r = b; // r_k+1
106     vector<double> temp_r = r; // r_k
107
108     vector<double> p = b; // p_k+1
109     vector<double> temp_p = p; // p_k

```

```

110
111 double alfa = 0;
112 double beta = 0;
113
114 vector<double> Ap; // Ap = A*p
115 Ap.assign(sizeMatrix,0);
116
117
118 //
119 ///////////////////////////////////////////////////////////////////
120
121 int counter = 0;
122 while (true) {
123
124     temp_x = x; // poprzednie wartosci przyblizen
125     temp_r = r;
126     temp_p = p;
127
128     ///////////////////////////////////
129     // alfa //
130
131     // licze Ap
132     for (int i = 0; i < sizeMatrix; ++i) {
133         if (i == 0) {
134             Ap[i] = temp_p[i] * D + temp_p[i + 1] * F + temp_p[i + 4] * H;
135         }
136         if (i > 0 && i < 4) {
137             Ap[i] = temp_p[i] * D + temp_p[i - 1] * E + temp_p[i + 1] * F + temp_p[i +
138 4] * H;
139         }
140         if (i >= 4 && i < sizeMatrix - 4) {
141             Ap[i] = temp_p[i] * D + temp_p[i - 1] * E + temp_p[i - 4] * G + temp_p[i +
142 1] * F + temp_p[i + 4] * H;
143         }
144         if (i >= sizeMatrix - 4 && i < sizeMatrix - 1) {
145             Ap[i] = temp_p[i] * D + temp_p[i - 1] * E + temp_p[i - 4] * G + temp_p[i +
146 1] * F;
147         }
148         if (i == sizeMatrix - 1) {
149             Ap[i] = temp_p[i] * D + temp_p[i - 1] * E + temp_p[i - 4] * G;
150         }
151     }
152
153     alfa = multiVV(temp_r, temp_r) / multiVV(temp_p, Ap);
154
155     ///////////////////////////////////
156
157     for (int i = 0; i < sizeMatrix; ++i) {
158         x[i] = temp_x[i] + alfa * temp_p[i];
159     }
160
161     ///////////////////////////////////
162
163     for (int i = 0; i < sizeMatrix; ++i) {
164         r[i] = temp_r[i] - alfa * Ap[i];
165     }
166
167     ///////////////////////////////////
168
169     beta = multiVV(r, r) / multiVV(temp_r, temp_r);
170
171     ///////////////////////////////////

```

```

169
170     for (int i = 0; i < sizeMatrix; ++i) {
171         p[i] = r[i] + beta * temp_p[i];
172     }
173
174     if( norm(subtract(x, temp_x)) < 0.000001)
175         break;
176     counter++;
177 }
178 return x;
179 }
180
181 int main() {
182     cout.setf(ios::fixed, ios::floatfield);
183     cout.precision(30);
184
185     // wyrazy wolne
186     b.assign(sizeMatrix, 1);
187
188
189     vector<double> x1;
190     vector<double> x2;
191
192     x1 = GS();
193     x2 = CG();
194
195     return 0;
196 }

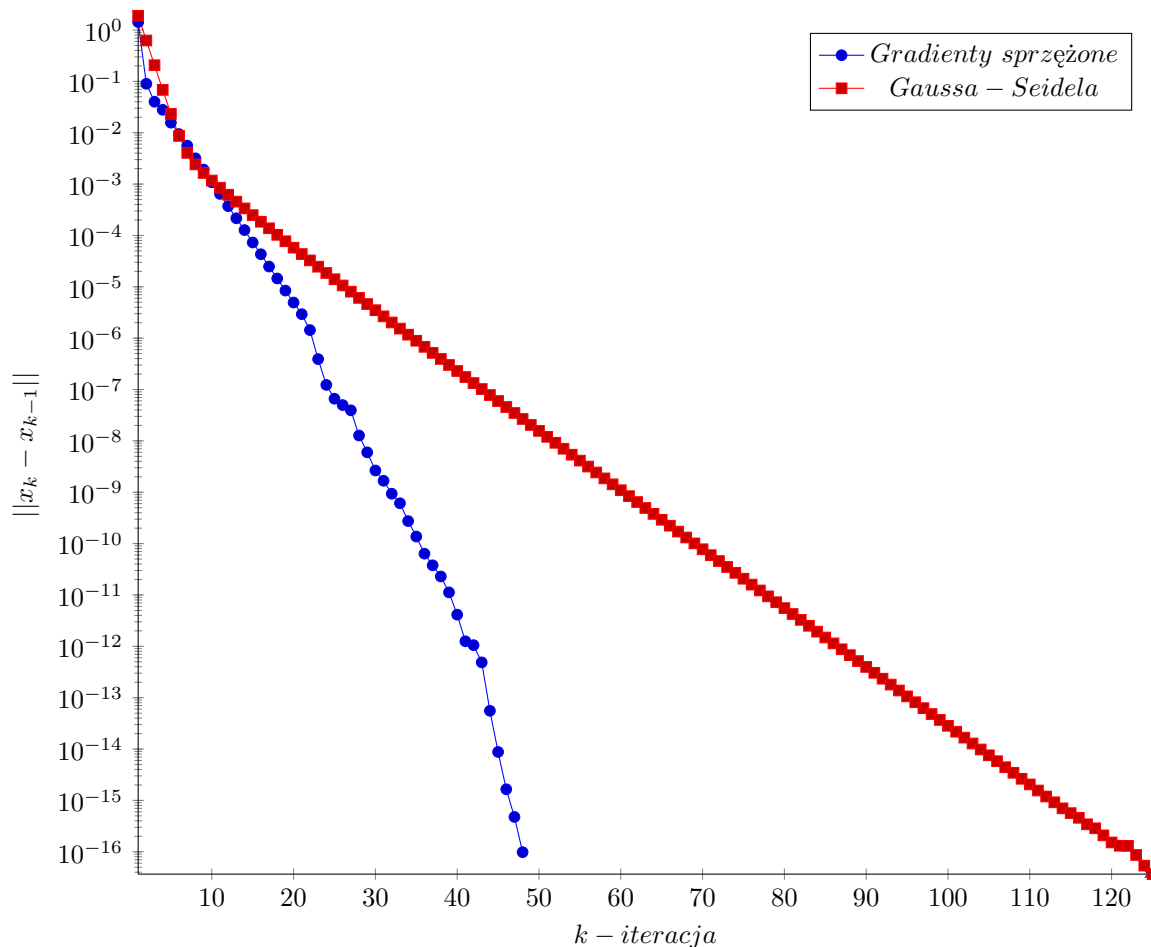
```

Wynik

Wypisuję tylko kilka dla sprawdzenia wyniku.

Gaussa-Seidela	Gradientów sprzężonych
$x = \begin{bmatrix} 0.194276795444196875850551009535 \\ 0.130930202478144686040195665555 \\ 0.146794908343523766713900613468 \\ \dots \\ 0.146794908343523738958324997839 \\ 0.130930202478144686040195665555 \\ 0.194276795444196875850551009535 \end{bmatrix}$	$x = \begin{bmatrix} 0.194276795444196792583824162648 \\ 0.130930202478144630529044434297 \\ 0.146794908343523738958324997839 \\ \dots \\ 0.146794908343523738958324997839 \\ 0.130930202478144658284620049926 \\ 0.194276795444196764828248547019 \end{bmatrix}$

Zależność normy



W metodzie Gaussa-Seidela $\|x_k - x_{k-1}\|$ potrzebujemy dużo więcej kroków iteracji niż w metodzie gradientów sprzężonych aby otrzymać przybliżone wyniki z oczekiwaną dokładnością.

Przykładowo, jeśli ustalimy $\epsilon = 10^{-6}$ dla naszej macierzy to musimy wykonać 35 iteracji algorytmu Gaussa-Seidela. Metoda gradientów sprzężonych jest dużo szybsza i oczekiwane przybliżenie otrzymamy już po 23 krokach iteracji.

Macierz A po rozkładzie Cholesky'ego nie zmienia swojej postaci i nadal jest to macierz z 4 dodatkowymi diagonalami. Zatem koszt takiego rozkładu wynosi $O((2P + 1)N) = O(9N)$