Program do zadania Nr.4:

1. Generowanie macierzy i main

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <numeric>

using namespace std;
using vec = vector<double>;
const double NEARZERO = 1.0e-10;

vector<vec> genMatrix(int n) {
    vector <vec> matrix(n, vec(n));

    matrix[0][0] = 4;
    matrix[0][4] = 1;
    matrix[1][0] = 1;
    matrix[4][0] = 1;
    matrix[n-2][n-1] = 1;
    matrix[n-1][n-1] = 4;
    for(int i = 1; i < n-1; i++){
        matrix[i][i] = 4;
        matrix[i][i+4] = 1;
        matrix[i+1][i] = 1;
        matrix[i-1][i] = 1;
    }
    for (int i = 1; i < n-4; i++) {
        matrix[i+4][i] = 1;
    }
    matrix[n-1][0] = 0;
    matrix[0][n-1] = 0;
    return matrix;
}

vec genResults(int n){
    vec res(n);
    for(int i = 0; i < n; i++){
        res[i] = 1;
    }
    return res;
}
```

```cpp
int main() {
    int i = 128;
        vector<vector<double>> array = genMatrix(i);
        vector<double> res0 = genResults(i);
        printMatrix(array, res0, i);
        vec X = conjugateGradientSolver( array, res0 );
        cout << "ConjugateGradient: \n";
        print(X);
        cout << "GaussSeidel: \n";
        gaussSeidelElimination(array,res0, i);
}
```

2. Metoda Gaussa-Seidla

```
40
41      /** METODA GUASSA-SEIDELA **/
42
43      void gaussSeidelElimination(vector<vec> array, vec arr, int n) {
44          int p = 0;
45          cout.precision( prec: 10);
46          vec result(n,  value: 0.0);
47          vec T(n,  value: 0.0); vec F(n,  value: 0.0);
48          double k = 1e10;
49          while (k >= 0.000001){
50              p = p + 1;
51              for (int i = 0; i < n; i++){
52                  double B = 0;
53                  T[i] = arr[i]/array[i][i];
54                  double C = array[i][i];
55                  for (int j = 0; j < n; j++){
56                      if (j!=i)
57                          B += array[i][j] * result[j];
58                  }
59                  result[i] = T[i]-B/C;
60              }
61              for (int i = 0; i < n; i++){
62                  F[i] = 0.0;
63                  for (int j = 0; j < n; j++){
64                      F[i] = F[i] + array[i][j]*result[j];
65                  }
66              }
67              for (int i = 0; i < n; i++){
68                  F[i] = F[i]-arr[i];
69              }
70              k = 0.0;
71              for (int i = 0;i < n;i++){
72                  k += F[i] * F[i];
73              }
74              k = sqrt(k);
75          }
76          cout << "Solution :\n";
77          for (int i = 0; i < n; i++){
78              cout << "X" << i+1 << "-->" << result[i] << "\n";
79          }
80      }
```

3. Metoda gradientu sprzężonego

```
82      /** METODA GRADIENTÓW SPRZĘŻONYCH **/
83      // Iloczyn wewnętrzny U i V.
84      double innerProduct(const vec &U, const vec &V){
85          return inner_product(U.begin(), U.end(), V.begin(), init: 0.0);
86      }
87      // Norma wektora
88      double vectorNorm(const vec &V){
89          return sqrt(innerProduct(V, V));
90      }
91      vec matrixTimesVector(const vector<vec>& array, const vec &V){
92          int n = array.size();
93          vec C(n);
94          for (int i = 0; i < n; i++)
95              C[i] = innerProduct(array[i], V);
96          return C;
97      }
98      // Liniowa kombinacja wektorów
99      vec vectorCombination(double a, const vec &U, double b, const vec &V){
100         int n = U.size();
101         vec W(n);
102         for (int j = 0; j < n; j++)
103             W[j] = a * U[j] + b * V[j];
104         return W;
105     }
106     void print(const vec &V){
107         cout.precision( prec: 10);
108         int n = V.size();
109         cout << "Solution: \n";
110         for (int i = 0; i < n; i++){
111             double x = V[i];
112             if (abs(x) < NEARZERO) x = 0.0;
113             cout << "x" << i+1 << "-->" << x << "\n";
114         }
115         cout << '\n';
116     }
117
118     vec conjugateGradientSolver(const vector<vec>& array, const vec& arr ){
119         double MARGIN = 1.0e-10;
120         int n = array.size();
121         vec X(n, value: 0.0);
122
123         vec R = arr;
124         vec P = R;
125         int k = 0;
126
127         while (k < n){
128             vec Rold = R;           // Przechowanie poprzednich pozostałości
129             vec AP = matrixTimesVector(array,P);
```

```
        double alpha = innerProduct(R,R) / max(innerProduct(P,AP),NEARZERO);
        X = vectorCombination( a: 1.0,X,alpha,P);            // Następne oszacowanie rozwiązania
        R = vectorCombination( a: 1.0,R,-alpha,AP);          // Pozostała
        if (vectorNorm(R) < MARGIN)
            break;
        double beta = innerProduct(R,R) / max(innerProduct(Rold,Rold),NEARZERO);
        P = vectorCombination( a: 1.0,R,beta,P);             // Następny gradient
        k++;
    }

    return X;
}

void printMatrix(const vector<vector<double>>& tab, const vector<double>& res, int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            printf("%.1f ", tab[i][j]);
        }
        printf("| %.1f", res[i]);
        printf("\n");
    }
}
```

Wyniki do zadania Nr.4:

Wyniki metod znajdują się w plikach tekstowych:

1. **WynikiGaussSeidel.txt**
2. **WynikiConjugateGradient.txt**

Komentarze do zadania Nr.4:

W tym zadaniu dla obliczania układu równań A*x=e, gdzie e_i = 1 macierzy A ∈ $R^{128x128}$ było skorzystano z dwóch metod: Metoda Gaussa-Seidela i metoda gradientu sprzężonego.

1. Metoda Gaussa-Seidela:

Kroki związane:
Krok 1: Obliczamy wartość dla wszystkich równań liniowych dla Xi. (Wstępna tablica musi być dostępna)
Krok 2: Obliczamy każdy Xi i powtórzamy powyższe kroki.
Krok 3: Skorzystamy z bezwzględnego względnego przybliżenia błędu po każdym kroku, aby sprawdzić, czy błąd występuje w ramach wcześniej określonej tolerancji.

Opis:

Metoda Gaussa – Seidla jest iteracyjną techniką rozwiązywania kwadratowego układu n równań liniowych o nieznanym x: Ax = b

Jest definiowany przez iterację:

$$L_*\mathbf{x}^{(k+1)} = \mathbf{b} \quad U\mathbf{x}^{(k)},$$

gdzie jest k p zbliżanie lub iteracji jest obok lub k + 1 iteracji i matryca rozkłada się na dolną trójkątnego elementu, a ściśle górnej trójkątnej składnik U:

$$\mathbf{x}^{(k)} \quad \mathbf{x},\ \mathbf{x}^{(k+1)}\mathbf{x}L_*A = L_* + U$$

Następnie rozkład A na jego dolny trójkątny składnik i jego ściśle górny trójkątny składnik daje:

$$A = L_* + U \qquad \text{where} \qquad L_* = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

Układ równań liniowych można przepisać jako:

$$L_*\mathbf{x} = \mathbf{b} \quad U\mathbf{x}$$

Metoda Gaussa – Seidla rozwiązuje teraz lewą stronę tego wyrażenia dla x, używając poprzedniej wartości dla x po prawej stronie. Analitycznie można to zapisać jako:

$$\mathbf{x}^{(k+1)} = L_*^{-1}(\mathbf{b} - U\mathbf{x}^{(k)}).$$

Jednak korzystając z trójkątnej postaci, elementy x (k + 1) mogą być obliczane sekwencyjnie przy użyciu podstawiania w przód:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i \quad \sum_{j=1}^{i\ 1} a_{ij}x_j^{(k+1)} \quad \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

Procedura jest generalnie kontynuowana, aż zmiany wprowadzone przez iterację będą poniżej pewnej tolerancji, takiej jak dostatecznie mała reszta.

2. Metoda gradientu sprzężonego

Opis metody:

Metoda gradientu sprzężonego jako metoda iteracyjna:

Jeśli właściwie dobierzemy sprzężone wektory pk, możemy nie potrzebować ich wszystkich do dobrej aproksymacji rozwiązania x*. Możemy więc spojrzeć na CG jak na metodę iteracyjną. Co więcej, pozwoli nam to rozwiązać układy równań, gdzie n jest tak duże, że bezpośrednia metoda zabrałaby zbyt dużo czasu. Oznaczmy punkt startowy przez x0. Bez starty ogólności możemy założyć, że x0 = 0 (w przeciwnym przypadku, rozważymy układ Az = b − Ax0). Zauważmy, że rozwiązanie x* minimalizuje formę kwadratową:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{b}, \quad \mathbf{x} \in \mathbf{R}^n.$$

Co sugeruje, by jako pierwszy wektor bazowy p1 wybrać gradient f w x = x0, który wynosi Ax0−b, a ponieważ wybraliśmy x0 = 0, otrzymujemy −b. Pozostałe wektory w bazie będą sprzężone do gradientu (stąd nazwa metoda gradientu sprzężonego).

Niech $\mathbf{r}_k$ oznacza rezyduum w *k*-tym kroku:

$$\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k.$$

Zauważmy, że rk jest przeciwny do gradientu f w x = xk, więc metoda gradientu prostego nakazywałaby ruch w kierunku rk. Tutaj jednak założyliśmy wzajemną sprzężoność kierunków pk, więc wybieramy kierunek najbliższy do rk pod warunkiem sprzężoności. Co wyraża się wzorem:

$$\mathbf{p}_{k+1} = \mathbf{r}_k - \frac{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}\mathbf{p}_k.$$

Upraszczając, otrzymujemy poniższy algorytm rozwiązujący Ax = b, gdzie macierz A jest rzeczywista, symetryczna i dodatnio określona. x_0 jest punktem startowym.

$r_0 := b - Ax_0$
$p_0 := r_0$
$k := 0$
repeat
  $\alpha_k := \frac{r_k^\top r_k}{p_k^\top A p_k}$
  $x_{k+1} := x_k + \alpha_k p_k$
  $r_{k+1} := r_k - \alpha_k A p_k$
  if r_{k+1} jest "wystarczająco mały" **then** exit loop **end if**
  $\beta_k := \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$
  $p_{k+1} := r_{k+1} + \beta_k p_k$
  $k := k + 1$
end repeat
Wynikiem jest x_{k+1}