Metody Obliczeniowe w Nauce i Technice

Mateusz Kocot

Laboratorium 2 – raport

Wszystkie zadania zaimplementowano z wykorzystaniem dołączonego do laboratorium kodu zawierającego klasę *AGHMatrix*.

Zad. 1.

1.1. Zaimplementowano funkcję dodawania macierzy:

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::operator+(const AGHMatrix<T>& rhs)
{
  if (this->rows != rhs.get_rows() || this->cols != rhs.get_cols())
  {
    throw std::invalid_argument("Wrong dimensions!");
  }
  std::vector< std::vector<T> > newMatrix(this->rows, std::vector<T>(this-
>cols));
  for (int i = 0; i < this->rows; i++)
  {
    for (int j = 0; j < this->cols; j++)
    {
      newMatrix[i][j] = this->matrix[i][j] + rhs(i, j);
    }
  }
  return AGHMatrix<T>(newMatrix);
}
```

1.2. Zaimplementowano funkcję mnożenia macierzy:

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::operator*(const AGHMatrix<T>& rhs)
{
  if (this->cols != rhs.get_rows())
  {
    throw std::invalid_argument("Wrong dimensions!");
  }
  std::vector< std::vector<T> > newMatrix(this-
>rows, std::vector<T>(rhs.get_cols(), 0));
  for (int i = 0; i < this->rows; i++)
  {
    for (int j = 0; j < rhs.get_cols(); j++)
    {
      for (int k = 0; k < this->cols; k++)
      {
```

```
        newMatrix[i][j] += this->matrix[i][k] * rhs(k, j);
      }
    }
  }
  return AGHMatrix<T>(newMatrix);
}
```

Zad. 2.

2.1. Zaimplementowano funkcję sprawdzającą symetryczność macierzy:

```
template<typename T>
bool AGHMatrix<T>::isSymmetric()
{
  if (this->rows != this->cols)
  {
    throw std::invalid_argument("Matrix should be square!");
  }
  for (int i = 1; i < this->rows; i++)
  {
    for (int j = 0; j < i; j++)
    {
      if (this->matrix[i][j] != this->matrix[i][j])
      {
        return false;
      }
    }
  }
  return true;
}
```

2.2. Zaimplementowano funkcję obliczającą wyznacznik macierzy kwadratowej. Wykorzystano dwie funkcje pomocnicze.

Funkcja *createMinor* tworzy nową macierz poprzez usunięcie i-tego wiersza i j-tej kolumny z macierzy wejściowej.

```
template<typename T>
AGHMatrix<T> AGHMatrix<T>::createMinor(int i, int j)
{
  std::vector< std::vector<T> > newMatrix;
  for (int k = 0; k < this->rows; k++)
  {
    if (k != i)
    {
      newMatrix.push_back({});
      for (int l = 0; l < this->cols; l++)
      {
```

```
        if (l != j)
        {
          newMatrix.back().push_back(this->matrix[k][l]);
        }
      }
    }
  }
  return AGHMatrix<T>(newMatrix);
}
```

Funkcja *minorDet* oblicza wyznacznik z macierzy wejściowej poprzez obliczenie wyznaczników minorów, czyli macierzy powstałych przez usunięcie i-tego wiersza i pierwszej (dowolnej) kolumny. Dodatkowo, zgodnie z odpowiednim wzorem, wyznacznik z minora mnożony jest przez odpowiedni element macierzy wejściowej oraz 1 albo -1 w zależności od tego, czy wiersz jest parzysty. Funkcja ta jest rekurencyjna.

```
template<typename T>
T AGHMatrix<T>::minorDet()
{
  if (this->rows == 1)
  {
    return this->matrix[0][0];
  }
  T det = 0;
  int sign = 1;
  for (int i = 0; i < this->rows; i++)
  {
    AGHMatrix<T> newMatrix = this->createMinor(i, 0);
    det += sign * this->matrix[i][0] * newMatrix.minorDet();
    sign = -sign;
  }
  return det;
}
```

Funkcja *det* jest już główną funkcją, która oprócz wywołania *minorDet* nie dopuszcza macierzy, które nie są kwadratowe.

```
template<typename T>
T AGHMatrix<T>::det()
{
  if (this->cols != this->rows)
  {
    throw std::invalid_argument("Matrix should be square!");
  }
  return this->minorDet();
}
```

2.3. Zaimplementowano funkcję zwracającą transpozycję danej jako argument macierzy:

```
template<typename T>
```

```
AGHMatrix<T> AGHMatrix<T>::transpose()
{
  std::vector< std::vector<T> > newMatrix(this->cols, std::vector<T>(this->rows));
  for (int i = 0; i < this->rows; i++)
  {
    for (int j = 0; j < this->cols; j++)
    {
      newMatrix[j][i] = this->matrix[i][j];
    }
  }
  return AGHMatrix<T>(newMatrix);
}
```

Zad. 3.

Zaimplementowano algorytm faktoryzacji LU macierzy wykorzystujący metodę Doolittle'a:

```
template<typename T>
std::vector< AGHMatrix<double> > AGHMatrix<T>::doolittle()
{
  if (this->rows != this->cols)
  {
    throw std::invalid_argument("Matrix should be square!");
  }
  int n = this->rows;

  std::vector< std::vector<double> > l(n, std::vector<T>(n, 0));
  for (int i = 0; i < n; i++)
  {
    l[i][i] = 1;
  }
  std::vector< std::vector<double> > u(n, std::vector<T>(n, 0));

  for (int i = 0; i < n; i++)
  {
    for (int j = i; j < n; j++)
    {
      u[i][j] = this->matrix[i][j];
      for (int k = 0; k < i; k++)
      {
        u[i][j] -= l[i][k] * u[k][j];
      }
    }
    for (int j = i + 1; j < n; j++)
    {
      double sum = 0;
```

```
      for (int k = 0; k < i; k++)
      {
        sum += l[j][k] * u[k][i];
      }
      l[j][i] = (1.0 / u[i][i]) * (this->matrix[j][i] - sum);
    }
  }

  return {AGHMatrix<double>(l), AGHMatrix<double>(u)};
}
```

Działanie algorytmu pomyślnie przetestowano na przykładzie z Wikipedii.

Zad. 4.

4.1. Zaimplementowano algorytm faktoryzacji Cholesky'ego macierzy:

```
template <typename T>
std::vector< AGHMatrix<double> > AGHMatrix<T>::cholesky()
{
  if (!this->isSymmetric())
  {
    throw std::invalid_argument("Matrix should be symmetric!");
  }
  int n = this->rows;

  std::vector< std::vector<double> > l(n, std::vector<T>(n, 0));
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j <= i; j++)
    {
      double sum = 0;
      if (j == i)
      {
        for (int k = 0; k < j; k++)
        {
          sum += l[j][k] * l[j][k];
        }
        l[j][j] = sqrt(this->matrix[j][j] - sum);
      }
      else
      {
        for (int k = 0; k < j; k++)
        {
          sum += l[i][k] * l[j][k];
        }
        l[i][j] = (this->matrix[i][j] - sum) / l[j][j];
      }
```

```
    }
  }

  AGHMatrix<double> L(l);
  return {L, L.transpose()};
}
```

Działanie algorytmu przetestowano pomyślnie na przykładzie z Wikipedii.

4.2. Celem obu algorytmów faktoryzacyjnych jest rozwiązanie układu równań liniowych.

W algorytmie faktoryzacji LU wyznaczane są dwie macierze trójkątne: dolna $\boldsymbol{L}$ i górna $\boldsymbol{U}$, w taki sposób, że macierz wejściowa $\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{U}$. Na głównej przekątnej jednej z nich znajdują się jedynki. Dzięki temu, rozkład macierzy jest jednoznaczny. W metodzie Doolitle'a na przemian obliczany jest element macierzy górnej i dolnej. Niestety, w niektórych przypadkach, w trakcie obliczania macierzy, może wystąpić dzielenie przez zero. Problem ten można wyeliminować wykorzystując metodę Doolittle-Crouta.

W algorytmie faktoryzacji Cholesky'ego wyznaczana jest tak naprawdę jedna macierz $\boldsymbol{L}$. Dla macierzy wejściowej $\boldsymbol{A}$ zachodzi równość $\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{L}^T$, gdzie $\boldsymbol{L}^T$. Nie każda macierz nadaje się do tej metody. By przeprowadzić faktoryzację Cholesky'ego, macierz musi być symetryczna oraz dodatnio określona.

Kiedy już można użyć algorytmu Cholesky'ego, jest on dwa razy wydajniejszy od faktoryzacji LU w rozwiązywaniu równań liniowych. Spowodowane jest to faktem, iż obliczana jest tylko jedna macierz trójkątna w porównaniu do dwóch w metodzie LU.

Zad. 5.

Zaimplementowano algorytm eliminacji Gaussa. Kod podzielono na dwie funkcje: *gaussSolve* oraz *gaussPrepareMatrix*.

Funkcja *gaussSolve* przyjmuje macierz współczynników równania $\boldsymbol{A}$ i macierz wyrazów wolnych $\boldsymbol{B}$:

```
AGHMatrix<double> gaussSolve(AGHMatrix<T> &A, AGHMatrix<T> &B)
```

Na początku wyznaczana jest połączona macierz $\boldsymbol{M}$ zawierająca współczynniki oraz wyrazy wolne:

```
int n = A.get_rows();
std::vector< std::vector<T> > newMatrix(n, std::vector<T>(n + 1));
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        newMatrix[i][j] = A(i, j);
    }
    newMatrix[i][n] = B(i, 0);
}
AGHMatrix<T> M(newMatrix);
```

Następnie wykonywana rekurencyjna jest funkcja *gaussPrepareMatrix*, której zadaniem jest przekształcenie macierzy $\boldsymbol{M}$ w macierz schodkową. Najpierw szukamy takiego wiersza, że jego pierwszy wyraz jest równy 0 oraz zamieniamy go z wierszem pierwszym:

```
for (int i = start; i < n; i++)
{
    if (M(i, start) != 0)
    {
        if (i != start)
        {
            for (int j = start; j <= n; j++)
            {
                std::swap(M(i, j), M(j, i));
            }
        }
        break;
    }
}
```

Wartość zmiennej *start* na początku wynosi 0, co oznacza, że rozpatrujemy pierwszy (o indeksie 0) wiersz. Innymi słowy, *start* wierszy jest już gotowe. Teraz od wiersza drugiego odejmujemy krotność wiersza pierwszego tak, by pierwszy element drugiego wiersza wynosił 0:

```
for (int i = start + 1; i < n; i++)
{
    double mult = M(i, start) / M(start, start);
    for (int j = start; j <= n; j++)
    {
        M(i, j) -= mult * M(start, j);
    }
}
```

Po tych przekształceniach pierwszy (*start*) wiersz jest gotowy, możemy przejść do drugiego (*start + 1*) wywołując rekurencyjnie funkcję:

```
gaussPrepareMatrix(M, start + 1);
```

Po przygotowaniu macierzy schodkowej wystarczy już tylko obliczyć rozwiązanie:

```
std::vector< std::vector<double> > x(n, std::vector<double>(1, 0));
for (int i = n - 1; i >= 0; i--)
{
    x[i][0] = M(i, n);
    for (int j = i + 1; j < n; j++)
    {
        x[i][0] -= M(i, j) * x[j][0];
    }
    x[i][0] /= M(i, i);
}
```

Licząc od dołu, zawsze będziemy mieli do czynienia z równaniem o jednej niewiadomej.

Zwracana jest odpowiednia macierz:

```
return AGHMatrix<double>(x);
```

Test algorytmu przeprowadzono z wykorzystaniem pakietu *NumPy* w języku *Python*:

```
import numpy as np

A = np.matrix([[0.0001, -5.0300, 5.8090, 7.8320],
               [2.2660, 1.9950,  1.2120, 8.0080],
               [8.8500, 5.6810,  4.5520, 1.3020],
               [6.7750, -2.253,  2.9080, 3.9700]])

b = np.matrix([9.5740, 7.2190, 5.7300, 6.2910]).transpose()

x = np.linalg.solve(A, b)
```

Znając już wartości zmiennych, wykonano (pomyślnie) test w języku *C++*:

```
std::vector< std::vector<double> > initX { {0.21602477},
                                           {-0.00791511},
                                           {0.63524333},
                                           {0.74617428} };
AGHMatrix<double> X(initX);
std::cout << "Gauss:" << std::endl;
AGHMatrix<double> XGauss = gaussSolve(A, B);
if (X == XGauss)
{
    std::cout << "ok" << std::endl;
}
else
{
    std::cout << "ERROR!" << std::endl;
}
```

Do porównania wykorzystano przeładowany operator porównania klasy *AGHMatrix*:

```
template<typename T>
bool AGHMatrix<T>::operator==(const AGHMatrix<T>& rhs)
{
  if (this->rows != rhs.get_rows() || this->cols != rhs.get_cols())
  {
    return false;
  }
  for (int i = 0; i < this->rows; i++)
  {
    for (int j = 0; j < this->cols; j++)
    {
      if (std::abs(this->matrix[i][j] - rhs(i, j)) > EPS)
      {
        return false;
```

```
      }
    }
  }
  return true;
}
```

Stałą $EPS = 10^{-8}$ wprowadzono ze względu na niedokładność typu zmiennopozycyjnego *double*.

Zad. 6.

6.1. Zaimplementowano metodę Jacobiego.

Najpierw sprawdzane jest, czy macierz $\boldsymbol{A}$ ma dominującą przekątną. Niestety, tylko wówczas metoda Jacobiego zwraca sensowne wyniki. Macierz $\boldsymbol{A}$ ma dominującą przekątną, gdy:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|.$$

```
int n = A.get_rows();
for (int i = 0; i < n; i++)
{
    T sum = -abs(A(i, i));
    for (int j = 0; j < n; j++)
    {
        sum += abs(A(i, j));
    }
    if (abs(A(i, i)) <= sum)
    {
        throw std::invalid_argument("Matrix should be diagonally dominant!");
    }
}
```

Następnie tworzone są odpowiednie macierze:

$\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U}$, gdzie $\boldsymbol{L}$ to macierz trójkątna dolna, $\boldsymbol{D}$ to macierz diagonalna, a $\boldsymbol{U}$ – macierz trójkątna górna.

$\boldsymbol{N} = \boldsymbol{D}^{-1}$

$\boldsymbol{M} = -\boldsymbol{N}(\boldsymbol{L} + \boldsymbol{U})$

```
std::vector< std::vector<double> > l(n, std::vector<double>(n, 0));
std::vector< std::vector<double> > nm(n, std::vector<double>(n, 0));
std::vector< std::vector<double> > u(n, std::vector<double>(n, 0));
for (int i = 0; i < n; i++)
{
    nm[i][i] = -1.0 / A(i, i);
    for (int j = 0; j < i; j++)
    {
```

```
        l[i][j] = A(i, j);
        u[j][i] = A(j, i);
    }
}
AGHMatrix<double> L(l);
AGHMatrix<double> N(nm);
AGHMatrix<double> U(u);

AGHMatrix<double> M = N * (L + U);

for (int i = 0; i < n; i++)
{
    N(i, i) = - N(i, i);
}
```

Należy jeszcze przyjąć początkowe wartości macierzy z wynikami $\boldsymbol{X}$. Ja wybrałem 0:

```
AGHMatrix<double> X(n, 1, 0);
```

Wówczas można już wykonywać kolejne iteracje algorytmu według wzoru:

$$\boldsymbol{X}^{n+1} = \boldsymbol{M}\boldsymbol{X}^{n} + \boldsymbol{N}\boldsymbol{B}$$

```
for (int i = 0; i < acc; i++)
{
    X = (M * X) + (N * B);
}
return X;
```

Niestety, macierz współczynników z poprzedniego zadania nie ma dominującej przekątnej. Algorytm sprawdzono na następującym przykładzie:

```
std::vector< std::vector<double> > initA2 { {12, 5, 2, 3},
                                            {-2, 10, -1, 5},
                                            {5, 1, 14, -7},
                                            {3, 2, -8, -16} };
AGHMatrix<double> A2(initA2);

std::vector< std::vector<double> > initB2 { {2},
                                            {8},
                                            {0},
                                            {-2} };
AGHMatrix<double> B2(initB2);
```

Wynikiem takiego układu jest:

```
std::vector< std::vector<double> > initX2 { {-0.17493889},
                                            {0.70358168},
                                            {0.08183654},
                                            {0.1392284} };
AGHMatrix<double> X2(initX2);
```

Sprawdzając wyniki metody Jacobiego dla różnych liczb iteracji okazało się, że dopiero po 41 iteracjach wynik metody Jacobiego jest równy prawdziwemu wynikowi równania z $EPS = 10^{-8}$. Warto jednak zaznaczyć, że już po 15 iteracjach wyniki były satysfakcjonujące:

```
iteration 15
-0.175169,
0.703336,
0.0819365,
0.139752
```

6.2. Metoda Jacobiego pozwala nam uzyskać wyniki z dowolną dokładnością, nie można jej jednak stosować w przypadku macierzy współczynników, które nie mają dominującej przekątnej. Zaletą eliminacji Gaussa jest brak ograniczeń związanych z dominującą przekątną. Metoda ta, w przeciwieństwie do metody Jacobiego, zwraca dokładne wyniki.

Obie metody operują na macierzach. Metoda Gaussa sprowadza połączoną macierz współczynników i wyrazów wolnych do macierzy schodkowej uzyskując łatwe do rozwiązania równania, natomiast metoda Jacobiego wykorzystuje odpowiednio obliczone macierze $\boldsymbol{M}$ i $\boldsymbol{N}$ do wyznaczania rozwiązań z coraz większą dokładnością.