	Politechnika Warszawska	***Programowanie równoległe i rozproszone***
Data: ***29.12.2017***	***Wykonawca:*** Królikowski Krzysztof (244739)	***Zadanie JB5.*** ***Rozwiązywanie układów równań liniowych metodą Jacobiego.***

1. Treść projektu.

Mamy dany układ równań liniowych $Ax = b$, A jest macierzą kwadratowa stopnia n. Dany jest rozkład $A = L + D + U$ gdzie L – macierz dolna trójkatna, D – macierz diagonalna, U – macierz górna trójkatna. Startując z pewnego x_0 kolejne przybliżenia obliczamy korzystając z równania:

$$Dx_{k+1} = -(L + U)x_k + b$$

Stąd wynika, że $(k + 1)$-sze przybliżenie i-tej składowej rozwiązania jest określone wzorem:

$$x_{i,k+1} = \frac{-\sum_{j=1,j\neq i}^{n} a_{ij}x_{j,k} + b_i}{a_{ii}}, \qquad i = 1, \dots, n$$

aż dojdziemy do momentu w którym $\|x_{k+1} - x_k\| < \varepsilon$, gdzie ε jest zadana dokładnością obliczeń. Napisać program, który umożliwi równoległe wyliczenie rozwiązania. Macierz A zostanie podzielona na poziome pasy, a każdym z pasów zajmie się osobny wątek/proces. Przyjąć, ze dane do zadania znajdują się w plikach tekstowych lub są generowane losowo (np. macierz A może być zdominowana diagonalnie). Rozwiązanie ma być zapisywane do pliku.

2. Analiza problemu

Program powinien spełniać dwie podstawowe funkcjonalności:

- losowe generowanie danych o zadanej wielkości,
- rozwiązanie układu równań metodą Jacobiego.

Aby algorytm Jacobiego był zbieżny macierz **A** musi mieć silnie dominującą diagonalę. Aby spełnić ten warunek, dla każdego elementu diagonali musi być spełnione tzw. silne kryterium sumy wierszowej[1]:

$$|a_{ii}| > \sum_{j=1, j\neq i}^{N\_COLS} |a_{ij}|$$

3. Struktura programu

Na podstawie specyfikacji problemu zbudowano program składający się z dwóch plików wykonywalnych wywoływanych z wiersza poleceń:

3.1 Generator danych (*randoms.exe*)

3.1.1 <u>Podstawowe informacje</u>

Jest to generator losowych macierzy A oraz wektorów b o zadanym przez użytkownika rozmiarze. Program zapisuje do plików tekstowych wygenerowaną przez użytkownika macierz oraz wektor.

rands.exe [size] [gain_multiplier] [A_matrix_filename] [b_vector_filename]

Przykład wywołania programu *rands.exe* dla rozmiaru problemu równego *100*, generującego macierz diagonalną o współczynniku wzmocnienia równym *1.02* oraz zapisującego macierz **A** do pliku *A100.txt* oraz wektor **b** do pliku *b100.txt*

rands.exe 100 1.02 A100.txt b100.txt.

3.1.2 <u>Opis wybranych funkcjonalności</u>

Każdy element macierzy A oraz wektora b jest generowany niezależnie i przy przyjęciu argumentu *range* = 1 oraz *offset* = 0, do macierzy wpisywana jest wartość zmiennoprzecinkowa z zakresu <0, 1>.

---

[1] T. Markiewicz, R. Szmurło, S. Wincenciak, *Metody numeryczne. Wykład na Wydziale Elektrycznym Politechniki Warszawskiej*, Warszawa: OWPW, 2014.

```
void generate_matrix(double **matrix,  int size, int range, float offset)
{
   int r,c;
   for(r=0; r<size; r++)
   {
      for(c=0; c<size; c++)
      {
         matrix[r][c] = ((float)rand()/(float)(RAND_MAX)+offset)*(float)range;
      }
   }
}
```

W programie uwzględniono możliwość losowego generowania macierzy zdominowanej diagonalnie. W tym celu modyfikuje się wartości diagonalne macierzy zgodnie z równaniem:

$a_{ii} = \sum_{j=1, j\neq i}^{N\_COLS} |a_{ij}| \cdot$ gain_muliplier, gdzie gain_muliplier jest współczynnikiem wzmocnienia diagonalizacji i może przyjąć dowolną wartość – do testów przyjęto współczynnik 1.2.

```
void makeDiagDominant(double** matrix, int rows, int cols, float gain_muliplier)
{
   int i,j;
   double sum_in_row;

   for (i=0; i<rows; i++)
   {
      sum_in_row = 0;
      for (j=0; j<cols; j++)
      {
         if(i!=j)
            sum_in_row = sum_in_row + fabs(matrix[i][j])* gain_muliplier;
      }
      matrix[i][i] = sum_in_row;
   }
}
```

3.1.3 Dodatkowe uwagi

- jeżeli gain_multiplier jest równy 0, to funkcja makeDiagDominant nie wywoła się – odpowiada za to logika wewnątrz funkcji głównej (*main*)
- wektor b wyznaczany jest pośrednio – tak naprawdę najpierw losowo generowany jest wektor rozwiązań x na podstawie którego wyznaczany jest wektor rozwiązań **b** (**b** = **Ax**), który następnie jest zapisywany na dysku.

3.2 Implementacja metody Jacobiego do rozwiązywania układu równań (*jacobi.exe*)

3.2.1 Podstawowe informacje

Zadaniem programu jest wyznaczenie wektora rozwiązań **x** na podstawie macierzy **A** oraz wektora b. Program dokonuje wczytania macierzy **A** oraz wektora **b** z plików tekstowych oraz umieszczeniu ich w tablicach typu double. Następnie wyznaczany jest metodą Jacobiego wektor x, który zostaje zapisany do pliku.

jacobi.exe [A_matrix_filename] [b_vector_filename]

Przykład wywołania programu *jacobi.exe* dla macierzy **A** zawartej w pliku *A100.txt* oraz wektora **b** z pliku *b100.txt*

jacobi.exe A100.txt b100.txt.

3.2.2 Opis wybranych funkcjonalności

Metoda Jacobiego jest typu iteracyjnego, stąd należy zdefiniować kryterium stopu. Logika programu przewiduje, że kolejne iteracje są prowadzone do momentu gdy $\|x_{k+1} - x_k\|_2 < \varepsilon$ lub też zostanie osiągnięta maksymalna liczba iteracji *max_it*. Osiągnięcie maksymalnej liczby iteracji określono na poziomie 10000, *zaś* ε = 1e-6.

```
int jacobi(double** A, double* b, double* x_,int size, int max_it, double epsilon)
{
            . . .
/* SOLVE LINEAR EQUATIONS */
   double ax, serr, rserr;
   k = 0;
   serr = 0;
   do //do_while
   {
      k = k + 1;
      for(i=0; i<rows; i++)
      {
         ax = 0;
         for(j=0; j<cols; j++)
            if(i!=j)
               ax = ax + (A[i][j]*x_0[j]);
         x_[i] = (-ax + b[i])/A[i][i];
      }
      for(i=0; i<rows; i++)
      {
         serr = pow(x_[i]-x_0[i],2);
         x_0[i]= x_[i];
      }
      rserr = sqrt(serr);
   } while(rserr>epsilon & k < max_it);
            . . .
}
```

Czas obliczeń mierzony jest tylko na funkcji *jacobi:*

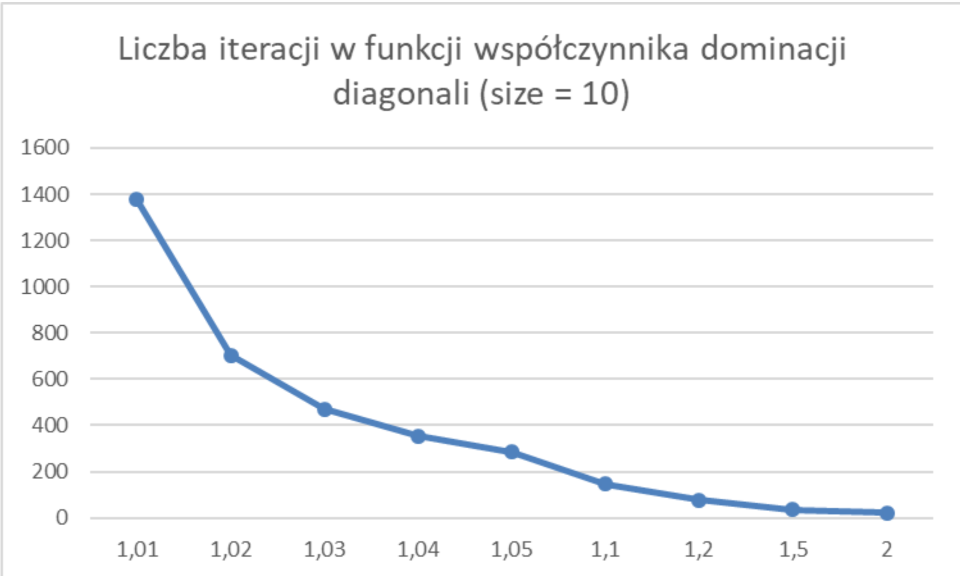
```
...
double cpu_time_used;
  start_time = clock();
  k = jacobi(A, b, x_, size, MAX_IT, EPSILON);
  end_time = clock();
  cpu_time_used = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
...
```

3.2.3 Dodatkowe uwagi

- zapis $\|x_{k+1} - x_k\|_2$ oznacza normę L2[2] (euklidesową) z wektora będącego różnicą między rozwiązaniem z iteracji k+1 oraz z iteracji k-tej

4. Testy wydajności

4.1.1 Wpływ współczynnika wzmocnienia na liczbę iteracji potrzebnej do uzyskania rozwiązania metodą Jacobiego.



Do dalszych testów przyjęto współczynnik wzmocnienia *gain_muliplier* = 1,02.

4.1.2 Wpływ rozmiaru problemu na czas obliczeń

Rozmiar problemu [liczba elementów macierzy]	*Czas obliczeń [s]*	*liczba iteracji [-]*	*rozmiar macierzy A [MB]*
100	0,044	699	0,09
1000	3,83	698	8,58
5000	76	698	214
10000	323,04	698	858

[2]Norma L2: $\|x\|_2 = \sqrt{x_1^2 + \cdots + x_n^2}$

Widoczna jest zależność, w której zwiększenie n krotne rozmiaru problemu powoduje w przybliżeniu n^2 krotny wzrost czasu rozwiązywania. Liczba iteracji natomiast jest silnie zależna od współczynnika diagonalizacji.

5. Walidacja wyników

Sprawdzenia dokonano dla problemu o rozmiarze 100. Polegało na zaczytaniu danych do programu MATLAB oraz wyznaczeniu rozwiązania. Norma euklidesowa z różnicy wektora będącego rozwiązaniem uzyskanym w omawianym programie oraz wektora wyznaczonego z użyciem programu MATLAB wyniosła 6.0997e-06, a więc jest na poziomie akceptowalnej dokładności.

Kod do walidacji wyników w programie MATLAB

```
A = load('A100.txt');
b = load('b100.txt');
x_ = load('x_result.txt');

x = A^-1*b;

diff = x_ - x
sqrt(diff' * diff)
```

Dodatkowo sprawdzono że proces kompilacji na systemach opartych na systemach linux przebiega poprawnie (gcc 4.4.7)

```
gcc version 4.4.7 20120313 (Red Hat 4.4.7-11) (GCC)
-bash-4.1$ gcc jacobi.c auxs.c -o jacobi.exe -lm
-bash-4.1$ gcc randoms.c auxs.c -o rands.exe -lm
-bash-4.1$ ./rands.exe 1000 1.02 "A.txt" "b.txt"
Input arguments: 3
 I will generate A matrix called 'A.txt', b vector called 'b.txt'.
 Size of the problem is 1000
 Gain is equal 1.020000
Matrix A has been saved successfully.
Vector b has been saved successfully.
-bash-4.1$ ./jacobi.exe A.txt b.txt
2 input arguments: I am using A matrix from 'A.txt' file and b vec
Size of the problem is equal to 1000
Loading A ...OK
Loading b...OK
Start solving equations
Solution achieved in 699 iteration. Elapsed time 4.820000 seconds
```