# MOwNiT Lab6 - sprawozdanie

Autor: Michał Flak

## Polecenie

```
    Korzystając z przykładu napisz program, który:

1.      Jako parametr pobiera rozmiar układu równań n

2.      Generuje macierz układu A(nxn) i wektor wyrazów wolnych b(n)

3.      Rozwiązuje układ równań

4.      Sprawdza poprawność rozwiązania (tj., czy Ax=b)

5.      Mierzy czas dekompozycji macierzy - do mierzenia czasu można
skorzystać z przykładowego programu dokonującego pomiaru czasu procesora
spędzonego w danym fragmencie programu.

6.      Mierzy czas rozwiązywania układu równań

·  Zadanie domowe: Narysuj wykres zależności czasu dekompozycji i czasu
rozwiązywania układu od rozmiaru układu równań. Wykonaj pomiary dla 10
wartości z przedziału od 10 do 1000.
```

## Rozwiązanie

Analogicznie jak w przykładzie tworzymy macierz, wektor i rozwiązujemy za pomocą GSL (tworzymy też kopię pierwotnej macierzy, bo dekompozycja modyfikuje nam pierwotne dane). Zapisujemy czas wykonania w zmiennych(metody generowania danych oraz mierzenia czasu do wglądu w listingu kodu na końcu sprawozdania):

```
    double* a_data = generate_matrix(n);
    double* a_copy = copyVector(a_data, n*n);

    double* b_data = generate_vector(n);

    gsl_matrix_view m
      = gsl_matrix_view_array (a_data, n, n);
    gsl_matrix_view m_copy
      = gsl_matrix_view_array (a_copy, n, n);

    gsl_vector_view b
      = gsl_vector_view_array (b_data, n);
```

```
  gsl_vector *x = gsl_vector_alloc (n);

  int s;

  gsl_permutation * p = gsl_permutation_alloc (n);

  double decomptime = decomp (&m.matrix, p, &s);

  double solvetime = solve (&m.matrix, p, &b.vector, x);
```

Poprawność rozwiązania sprawdzamy mnożąc pierwotną macierz przez wektor wynikowy za pomocą `gsl_blas_dgemv`. Porównujemy otrzymany wektor z pierwotnym wektorem `b` do epsilonu=0.005:

```
int compare_vectors(gsl_vector*a, gsl_vector*b) {
  if(a->size != b->size) return 0;

  const double eps = 0.005;

  for(int i = 0; i < a->size; i++){
    if(fabs(gsl_vector_get(a,i)-gsl_vector_get(b,i)) > eps) {
      return 0;
    }
  }
  return 1;
}

int verify_correctness(int n, gsl_matrix *m, gsl_vector*b, gsl_vector *x) {

  gsl_vector *y = gsl_vector_alloc(n);
  int err = gsl_blas_dgemv( CblasNoTrans, 1.0, m, x, 1.0, y );

  return compare_vectors(y, b);
}
```

Wypisujemy wynik na wyjście standardowe w formacie:

```
Size: 700, DTime: 0.10751800000000000, STime: 0.00103000000000000, Correct:
1
```

Następującym skryptem uruchamiamy program dla 10 wartości w przedziale [10,1000]:

```
rm wyniki.txt
for c in {10, 100, 200, 400, 500, 600, 700, 800, 900, 1000}
do
    ./lab6 $c | tee -a wyniki.txt
done
```

## Wyniki

Zawartość `wyniki.txt`:

```
Size: 100, DTime: 0.00036000000000000, STime: 0.00003400000000000, Correct: 1
Size: 200, DTime: 0.00272200000000000, STime: 0.00010400000000000, Correct: 1
Size: 400, DTime: 0.02155200000000000, STime: 0.00038700000000000, Correct: 1
Size: 500, DTime: 0.05089800000000000, STime: 0.00086500000000000, Correct: 1
Size: 600, DTime: 0.07247099999999999, STime: 0.00106600000000000, Correct: 1
Size: 700, DTime: 0.10751800000000000, STime: 0.00103000000000000, Correct: 1
Size: 800, DTime: 0.16430400000000001, STime: 0.00128000000000000, Correct: 1
Size: 900, DTime: 0.26297900000000002, STime: 0.00174800000000000, Correct: 1
Size: 1000, DTime: 0.34624500000000002, STime: 0.00233800000000000, Correct: 1
```

```
x <- seq(0,2.5,0.1)
plot(x, x^2-5,
main="x^2-5",
type="l",
col="blue")
grid()
```

# Zadanie 2

## Polecenie

```
Zmienić program tak, aby znajdował pierwiastek metodą siecznych oraz Brent-Dekker'a.

  *  Porownać metody.
  *  Zamienić program tak, aby spróbował znaleźć pierwiastek równania x^2-2*x+1=0.
  *  Narysować wykres tej funkcji za pomocą np. gnuplota.
  *  Wyjasnić działanie programu - dlaczego nie może znaleźć miejsc zerowych dla tego równania?
```

## Porównanie metod:

1. Metoda Brenta to połączenie metody siecznych, metody bisekcji oraz odwrotnej interpolacji kwadratowej. Łączy niezawodność bisekcji z szybkością pozostałych metod.
   x^2-5: 6 kroków x=2.2360634
   x^2-2*x+1: ERROR: endpoints do not straddle y=0

2. Metoda siecznych wymaga użycia solvera wykorzystującego pochodne funkcji. Jest to uproszczenie metody Newtona niewymagające liczenia pochodnej tak często.
   x^2-5: 5 kroków x=2.2360845
   x^2-2*x+1: 16 kroków x=1.0015480

### Wykres nowej funkcji x^2-2*x+1:

```
x <- seq(0,2.5,0.1)
plot(x, x^2-2*x+1,
main="x^2-2*x+1",
type="l",
col="blue")
grid()
```

### Wyjaśnienie zachowania programu:

Przy metodzie Brenta (jak również bisekcji) GSL oczekuje, że wartości funkcji w obu końcach przedziału będą miały różne znaki. W tym przypadku tak nie ma ponieważ delta=0, dlatego program wypisuje błąd.

## Zadanie 3

### Polecenie

```
Napisać program szukający miejsc zerowych za pomocą metod korzystających z
pochodnej funkcji.
Czym różnia się od poprzednich metod i dlaczego potrafią znaleźć
pierwiastek równania x^2-2*x+1=0?

    Porównać metodę Newtona, uproszczoną Newtona i Steffensona.
```

### Opis metod

Program napisano w zadaniu 2 w celu korzystania z metody siecznych (czyli uproszczonej Newtona).

Metody nie potrzebują żeby wartości funkcji na krańcach przeszukiwanego przedziału były różnych znaków, dlatego można ich używać do znajdowania pierwiastków w funkcjach o wartościach lokalnie wyłącznie nieujemnych / niedodatnich (styk wykresu z osią X w punkcie).

### Kod programu

```c
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_roots.h>

#include "demo_fn.h"

int
main (void)
{
  int status;
  int iter = 0, max_iter = 100;
  const gsl_root_fdfsolver_type *T;
  gsl_root_fdfsolver *s;
  double x0, x = 5.0, r_expected = sqrt (5.0);
  gsl_function_fdf FDF;
  //struct quadratic_params params = {1.0, 0.0, -5.0};
  struct quadratic_params params = {1.0, -2.0, 1};

  FDF.f = &quadratic;
  FDF.df = &quadratic_deriv;
  FDF.fdf = &quadratic_fdf;
  FDF.params = &params;

  T = gsl_root_fdfsolver_secant;
  s = gsl_root_fdfsolver_alloc (T);
  gsl_root_fdfsolver_set (s, &FDF, x);

  printf ("using %s method\n",
          gsl_root_fdfsolver_name (s));

  printf ("%-5s %10s %10s %10s\n",
          "iter", "root", "err", "err(est)");
  do
    {
      iter++;
      status = gsl_root_fdfsolver_iterate (s);
      x0 = x;
      x = gsl_root_fdfsolver_root (s);
      status = gsl_root_test_delta (x, x0, 0, 1e-3);

      if (status == GSL_SUCCESS)
        printf ("Converged:\n");

      printf ("%5d %10.7f %+10.7f %10.7f\n",
              iter, x, x - r_expected, x - x0);
    }
  while (status == GSL_CONTINUE && iter < max_iter);

  gsl_root_fdfsolver_free (s);
  return status;
}
```

## Porównanie funkcji:

- Newton:
  12 kroków, x=1.0009766
- Newton uproszczony (sieczne):
  16 kroków, x=1.0015480
- Steffenson:
  4 kroki, x=1.0000000