Metody Obliczeniowe w Nauce i Technice

Laboratorium 3 – raport

Mateusz Kocot

1. Funkcje testowe

Zaimplementowano trzy funkcje testowe: *f1*, *f2* oraz *f3*.

$$f_1(x) = \cos(x)\cosh(x) - 1$$

```
inline double f1(double x)
{
    return cos(x) * cosh(x) - 1;
}
```

Funkcja f_1 będzie testowana w przedziale: $\left[\frac{3}{2}\pi, 2\pi\right]$.

$$f_2(x) = \frac{1}{x} - \tan x$$

```
inline double f2(double x)
{
    return 1/x - tan(x);
}
```

Funkcja f_2 będzie testowana w przedziale: $\left[0, \frac{\pi}{2}\right]$.

$$f_3(x) = 2^{-x} + e^x + 2\cos(x) - 6$$

```
inline double f3(double x)
{
    return pow(2, -x) + exp(x) + 2 * cos(x) - 6;
}
```

Funkcja f_3 będzie testowana w przedziale: $[1,3]$.

2. Metoda bisekcji

Napisano funkcję realizującą metodę bisekcji dla danej funkcji f, przedziału $[a, b]$, dokładności *eps* oraz maksymalnej liczby iteracji *nmax*.

```
std::pair<double, int> solveBisection(double f(double), double a, double b,
                                      double eps, int nmax)
{
    double fa = f(a), fb = f(b);
    if (!oppositeSigns(fa, fb))
    {
        throw std::invalid_argument("The function values are not of
                                      opposite sign!");
    }
    double c, fc;
    int i = 0;
    while (i < nmax)
    {
        c = (a + b) / 2;
        fc = f(c);
        if (fc == 0 || fabs(c - a) <= eps)   // main loop condition
        {
            return {c, i};
        }
        i++;
        if (oppositeSigns(fa, fc))
        {
            b = c;
            fb = fc;
        }
        else
        {
            a = c;
            fa = fc;
        }
    }
    return {-1, i};
}
```

Wykorzystano także funkcję pomocniczą *oppositeSigns*, która zwraca *True* dla dwóch liczb o przeciwnych znakach (bez uwzględnienia 0).

```
inline bool oppositeSigns(double x, double y)
{
    return (x > 0 && y < 0) || (x < 0 && y > 0);
}
```

Przetestowano działanie metody dla funkcji z punktu 1. W przybliżeniu uzyskano następujące wyniki:

$$f_1: 4.73,$$

$$f_2: 0.86,$$

$$f_3: 1.83.$$

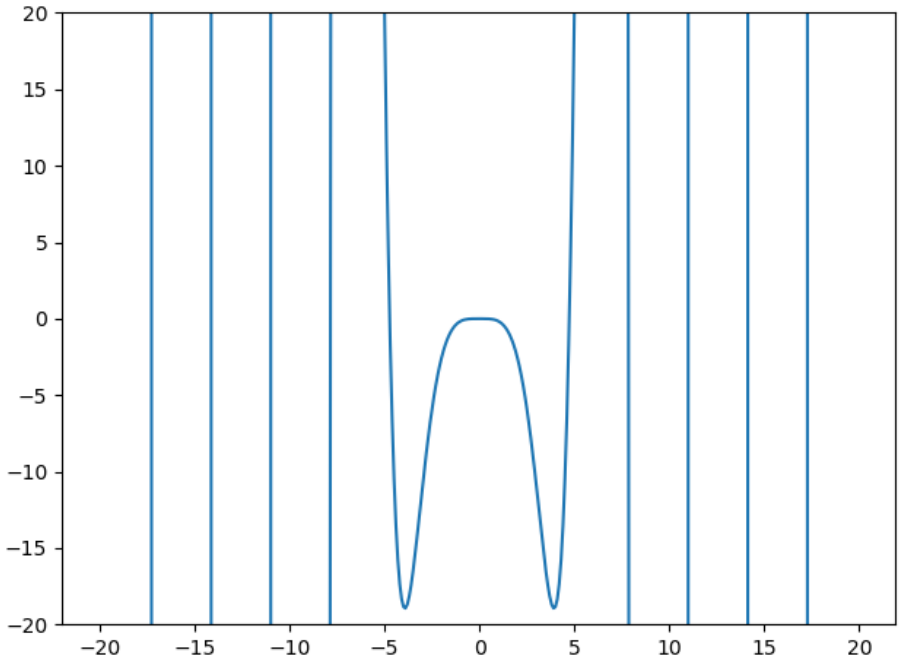
Obliczone miejsca zerowe są zgodne z prawdziwymi.

Oczywiście, dla różnych dokładności eps otrzymujemy różną dokładność wyniku. Tak, jak można się było spodziewać, wyniki dla $eps = 10^{-15}$ zaczynają się różnić od tych dla $eps = 10^{-7}$ mniej więcej od 8 cyfry znaczącej (w systemie dziesiętnym). Analogicznie, wyniki dla $eps = 10^{-33}$ zaczynają się różnić od tych dla $eps = 10^{-15}$ mniej więcej od 16 cyfry znaczącej.

Nie jesteśmy jednak w stanie otrzymać pożądanej dokładności $eps = 10^{-33}$ ze względu na użycie typu zmiennopozycyjnego *double*. Typ ten ma dokładność ok. 16 dziesiętnych cyfr znaczących i większej dokładności nie można uzyskać. W przypadku f_3 zastosowanie takiej dokładności wiąże się z błędem. Nawet dla 1000 iteracji nie uzyskano wyniku. W związku z tym, dalsze rozważania nie będą obejmować $eps = 10^{-33}$.

W przypadku $eps = 10^{-7}$ wymagana liczba iteracji jest równa ok. 23, natomiast dla $eps = 10^{-15}$ liczba ta jest równa ok. 50. Liczby te nie są przypadkowe, gdyż $\log_2 10^7 \cong 23$, natomiast $\log_2 10^{15} \cong 50$. Przedział jest dzielony na pół, stąd 2 u podstawy logarytmu.

Metodę bisekcji można wykorzystać do obliczenia pierwszych *k* pierwiastków funkcji f_1. Jej wygląd można zobaczyć na rys. 1.



Rys. 1. Wykres funkcji f_1

Będziemy operować na kolejnych przedziałach. Zaczynamy od $a = 0$ i $b = h$, gdzie h to odpowiednia wartość skoku. Dopóki nie będzie spełniona zależność $f(a)f(b) < 0$, należy zwiększać b, tj. $b = b + h$. Po odpowiedniej liczbie iteracji uzyskamy właściwy przedział i będziemy w stanie znaleźć pierwsze miejsce zerowe. W celu znalezienia kolejnych, ustawiamy $a = b$, $b = b + h$ i postępujemy tak samo, jak w przypadku pierwszego pierwiastka.

Należy oczywiście wybrać odpowiednią wartość skoku h. Nie może ona być zbyt mała, ponieważ czas działania będzie zbyt długi; nie może też być zbyt duża, gdyż wówczas miejsca zerowe mogą zostać pominięte.

3. Metoda Newtona

Napisano funkcję realizującą metodę Newtona dla danej funkcji f, przedziału $[a, b]$, dokładności *eps* oraz maksymalnej liczby iteracji *nmax*.

```
std::pair<double, int> solveNewton(double f(double), double a, double b,
                                   double eps, int nmax)
{
    if (!oppositeSigns(f(a), f(b)))
    {
        throw std::invalid_argument("The function values are not of
                                    opposite sign!");
    }
    double x = a, xold, fder;
    int i = 0;
    while(i < nmax) // First condition
    {
        fder = (f(x + h) - f(x - h)) / (2 * h);   // f'(x)
        xold = x;
        x = x - f(x) / fder;
        i++;
        if (fabs(x - xold) <= eps)   // Second condition
        {
            return {x, i};
        }
    }
    return {-1, i};
}
```

Pojawia się jednak problem, gdyż funkcja ta nie działa dla f_2 i przyjętego przedziału $\left[0, \frac{\pi}{2}\right]$. Problem ten wynika z faktu, iż nie istnieją wartości funkcji *tangens* w 0 i $\frac{\pi}{2}$. Należy więc przyjąć nowy przedział $\left[h, \frac{\pi}{2} - h\right]$ np. dla $h = 10^{-6}$. Założenie to nie wpływa na rozwiązanie.

Miejsca zerowe zwrócone przez metodę Newtona są takie same jak w poprzedniej metodzie.

Znacznie zmalała jednak liczba iteracji. Dla funkcji pierwszej są to (odpowiednio dla $eps = 10^{-7}$ i $eps = 10^{-15}$) 3 i 5 iteracji, dla funkcji drugiej: 23 i 24, a dla funkcji trzeciej: 8 i 9. W przypadku metody bisekcji, liczby iteracji dla określonej dokładności były takie same. Tutaj jednak jest inaczej. Wpływa na to nachylenie krzywej w punkcie a, tj. w punkcie, w którym rozpoczynamy algorytm. W przypadku f_2, funkcja w punkcie początkowym jest niemal pionowa, więc różnica między kolejnym wyznaczonym punktem będzie mała. Potrzeba zatem większej liczby iteracji.

Wadą metody Newtona jest konieczność obliczania pochodnej. Ja zdecydowałem się na numeryczne obliczenie pochodnych centralnych, natomiast dokładność tego rozwiązania nie jest optymalna. Metoda bisekcji, mimo iż znacznie wolniejsza, nie posiada takiej wady.

4. Metoda siecznych

Napisano funkcję realizującą metodę siecznych dla danej funkcji f, przedziału $[a, b]$, dokładności *eps* oraz maksymalnej liczby iteracji *nmax*.

```
std::pair<double, int> solveSecant(double f(double), double a, double b,
                                   double eps, int nmax)
{
    double fa = f(a), fb = f(b);
    if (!oppositeSigns(fa, fb))
    {
        throw std::invalid_argument("The function values are not of
                                     opposite sign!");
    }
    double x;
    int i = 0;
    while (i < nmax)    // First condition
    {
        x = (fb * a - fa * b) / (fb - fa);
        if (fabs(x - b) <= eps)     // Second condition
        {
            return {x, i};
        }
        i++;
        a = b;  fa = fb;
        b = x;  fb = f(x);
    }
    return {-1, i};
}
```

Analogicznie, jak w przypadku metody Newtona, dobre wyniki dla funkcji f_2 uzyskać można po nieznacznej zmianie rozpatrywanego przedziału.

Uzyskane wyniki są poprawne.

Liczby iteracji dla wszystkich trzech metod zebrano w tab. 1. dla $eps = 10^{-7}$ i w tab. 2. dla $eps = 10^{-15}$.

	f_1	f_2	f_3
metoda bisekcji	23	23	24
metoda Newtona	3	23	8
metoda siecznych	5	5	9

Tab. 1. Liczby iteracji dla $eps = 10^{-7}$

	f_1	f_2	f_3
metoda bisekcji	50	50	50
metoda Newtona	5	24	9
metoda siecznych	6	7	10

Tab. 1. Liczby iteracji dla $eps = 10^{-15}$

Jak widać, metody Newtona i siecznych są znacznie szybsze od metody bisekcji. Metoda siecznych w każdym przypadku potrzebuje podobną liczbę iteracji co metoda Newtona oprócz przypadku f_2, gdzie metoda siecznych jest sporo szybsza. Wynika z tego, że metoda ta w przeciwieństwie do metody Newtona nie jest narażona na stromą krzywą i nawet wówczas działa szybko.

Inną zaletą metody siecznych jest fakt, iż w przeciwieństwie do metody Newtona, nie potrzebuje ona liczyć pochodnych. Do przybliżania wykorzystuje dwa punkty.

Metoda siecznych okazała się być najlepszą metodą spośród rozpatrywanych. Nie potrzebuje ona dużej liczby iteracji do uzyskania określonej dokładności oraz nie musi liczyć pochodnych.