Laboratorium 3: Równania nieliniowe

Metody Obliczeniowe w Nauce i Technice

Wiktor Tarsa

Funkcje testowe
Utworzyłem w programie funkcje reprezentujące podane w instrukcji funkcje testowe:

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

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

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

Funkcje cos, cosh, tan, pow i exp to funkcje z biblioteki **cmath.**

Metoda bisekcji
Napisałem funkcję realizującą metodę bisekcji.

```
double findSoultion_bisection(double a, double b, double func(double),
double tolerance, int &iterationCount) {

   double c = (a + b) / 2;
   iterationCount = 0;
   while (!doubleCompare(func(c), 0, tolerance) && iterationCount < NMAX) {
       if (func(a) * func(c) < 0) b = c;
       else a = c;
       c = (a + b) / 2;
       iterationCount++;
   }
   return c;
}
```

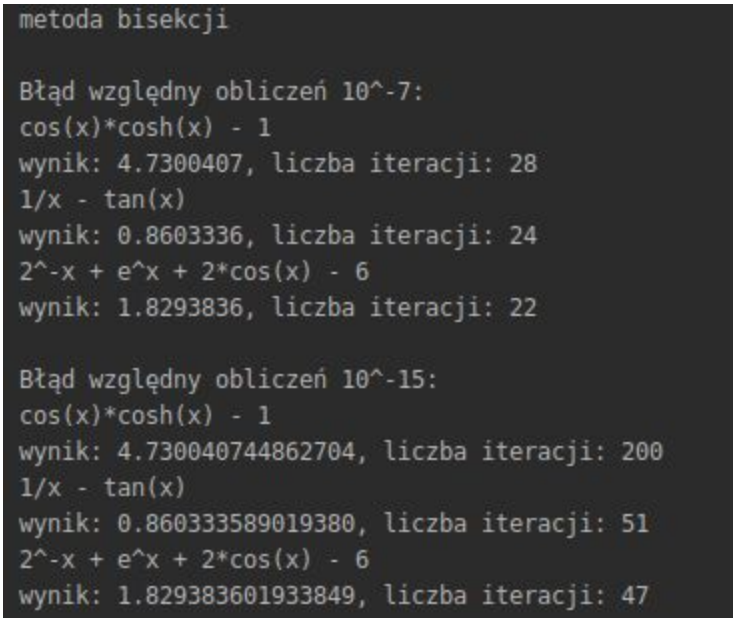
Funkcja oprócz krańców przedziału oraz błędu względnego przyjmuje jako argument także funkcję testową, w której zostanie znalezione rozwiązanie i referencję do licznika wykonanych operacji.

Istnieją przypadki w których funkcja nigdy nie znajdzie dokładnego rozwiązania, ponieważ rozwiązanie posiada nieskończone rozwinięcie dziesiętne(np f(x) = x - π). Aby zapobiec nieskończonej pętli wprowadziłem dodatkowy warunek w pętli while, który zatrzymuje pętlę po 200 iteracjach.

Do porównywania liczb typu double napisałem specjalną funkcję:

```
bool doubleCompare(double a, double b, double tolerance) {
   return fabs(a - b) < tolerance;
}
```

Poniżej zamieszczam wyznaczone miejsca zerowe funkcji testowych na podanych przedziałach wraz z błędem względnym obliczeń. Przy każdym wyniku wypisałem liczbę iteracji, jaka została wykonana by wyznaczyć wynik z określoną dokładnością.



Nie wyznaczyłem wyniku z dokładnością rzędu 10^-33, ponieważ dokładność powszechnie znanych typów danych w c++ jest znacznie mniejsza od 10^-33 (double ~ 10^-16, long double ~ 10^-22).

W celu wyznaczenia pierwszych k dodatnich pierwiastków funkcji należy:

1) ustalić odpowiedni ε oraz krańce przedziału: a = 0 i b = a + ε.
2) zwiększać ε tak długo, aż nierówność: f1(a)*f1(b) < 0 stanie się prawdziwa
3) wykonać bisekcję funkcji f1 na przedziale [a, b]
4) przypisać koniec przedziału do jego początku, tzn: a = b
5) wykonać kroki 1-4 k razy

Wydajność wyznaczania k dodatnich pierwiastków zależy od odpowiednio dobranego ε. Zbyt mała wartość ε spowolni działanie algorytmu, natomiast zbyt duża - może spowodować błąd w wyznaczaniu pierwiastków metodą bisekcji (gdy w przedziale będą znajdowały się dwa pierwiastki).

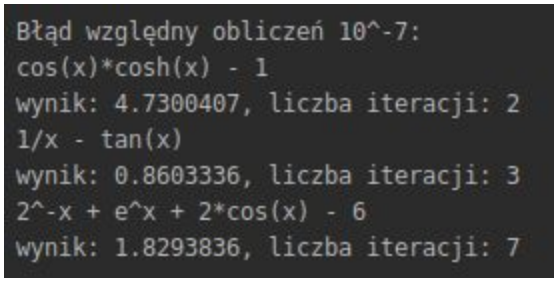
Metoda Newtona
Napisałem funkcję realizującą metodę Newtona.

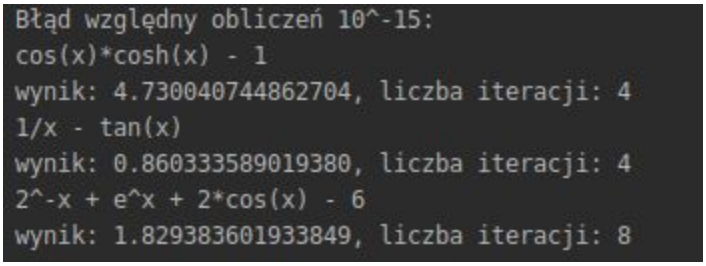
```
bool findSolution_Newton(double a, double func(double),double
deriv(double),
       int &iterations, double tolerance, double epsilon, double
&solution){

   for(int i = 0; i < iterations; i++){
       double y = func(a);
       double y_deriv = deriv(a);
       if(std::abs(y_deriv) < epsilon) break;
       solution = a - y/y_deriv;
       if(std::abs(solution - a) <= tolerance){
           iterations = i;
           return true;
       }
       a = solution;
   }
   return false;
}
```

Funkcja przyjmuje punkt startowy - równy wartości początku przedziału, funkcję oraz pochodną funkcji, referencję na maksymalną liczbę iteracji oraz referencję na rozwiązanie. Funkcja przyjmuje także jako argument wartość błędu bezwzględnego obliczeń. Funkcja zwraca true, jeśli rozwiązanie o satysfakcjonującej dokładności zostało znalezione lub false - w przeciwnym przypadku.

W tej metodzie potrzebujemy również pochodne funkcji, które zostały umieszczone na początku instrukcji, dlatego wyznaczyłem je za pomocą wolframalpha.com oraz napisałem odpowiednie funkcje reprezentujące pochodne.

Metodę testowałem na takich samych danych wejściowych, co metodę bisekcji. Wyniki wyglądają następująco:





Otrzymane wyniki są dokładnie takie same, jednak ilość iteracji, jaka jest potrzebna na wyznaczenie rozwiązania znacząco się zmniejszyła. Na podstawie tych danych można stwierdzić, że metoda Newtona jest znacznie bardziej wydajna od metody bisekcji.

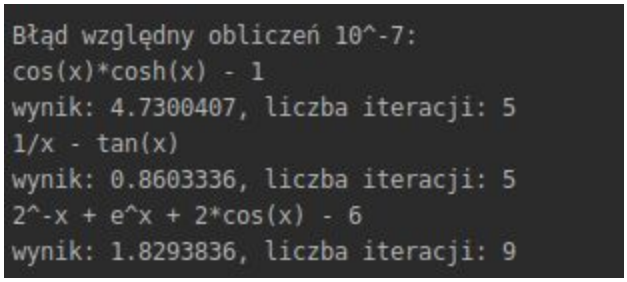
Metoda siecznych

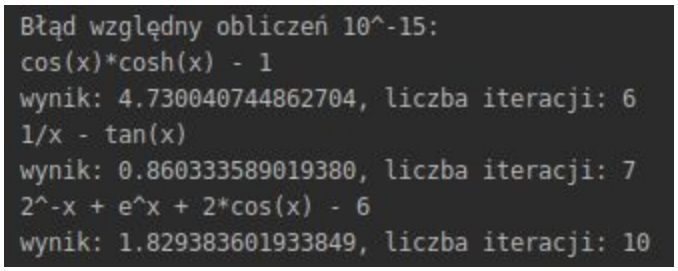
Napisałem funkcję realizującą metodę siecznych.

```
bool findSolution_secant(double a, double b, double func(double), int
&iterations, double &solution, double tolerance){

   for(int i = 0; i < iterations; i++){
       double c = b - func(b) * (b - a) / (func(b)-func(a));
       a = b;
       b = c;
       if(std::abs(b - a) <= tolerance){
           iterations = i;
           solution = c;
           return true;
       }
   }
   return false;
}
```

Funkcja przyjmuje krańce przedziału, funkcję matematyczną dla której szuka miejsc zerowych i wartość dopuszczalnego błędu względnego obliczeń. Funkcja jako argument przyjmuje również referencję na dwie zmienne: liczbę dopuszczalnych iteracji oraz wyznaczone miejsce zerowe. Funkcja zwraca true jeśli znalazła rozwiązanie w podanej liczbie iteracji lub false w przeciwnym przypadku.

Funkcję przetestowałem na tych samych danych, co funkcje realizujące dwie poprzednie metody.





Otrzymane wyniki są dokładnie takie same jak wyniki wyznaczone przy pomocy dwóch pozostałych metod. Metoda siecznych potrzebuje znacznie mniej iteracji, by wyznaczyć wynik, niż metoda bisekcji. Jednak w każdym przypadku ostatnia metoda była wolniejsza od metody Newtona.

Niekwestionowanym plusem metody siecznych jest to, że nie potrzebuje ona pochodnej funkcji dla której wyznacza miejsce zerowe.