AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Metody Obliczeniowe w Nauce i Technice:

Równania nieliniowe

Laboratorium 3

Przemysław Lechowicz

1. Funkcje testowe

Zaimplementowane zostały funkcje testowe wraz z ich przedziałami.

```
double f1_x0 = (3.0 / 2.0) * M_PI;
double f1_x1 = 2.0 * M_PI;

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

double f2_x0 = 0.0;
double f2_x1 = M_PI_2;

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

double f3_x0 = 1.0;
double f3_x1 = 3.0;

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

Miejsce zerowe funkcji $f_1(x)$ w przedziale $[\frac{3}{2}\pi, 2\pi]$ wynosi około 4.73004074486270.
Miejsce zerowe funkcji $f_2(x)$ w przedziale $[0, \frac{\pi}{2}]$ wynosi około 0.860333589019380
Miejsce zerowe funkcji $f_3(x)$ w przedziale $[1, 3]$ wynosi około 1.82938360193385

2. Metoda bisekcji

Zaimplementowana została metoda bisekcji, przyjmująca: krańce przedziału, funkcję, której miejsce zerowe obliczamy oraz wartość dopuszczalnego błędu bezwzględnego, a zwracająca strukturę zawierającą wynik i ilość iteracji.

```
solution bisection(double p, double q, double (fun)(double), double precision) {
    if (fun(p) * fun(q) >= 0) {
        throw std::invalid_argument("Wrong interval");
    }
    int operations=0;
    double m = p;
    while ((q - p) >= precision) {
        m = (p + q) / 2;

        if (fun(m) == 0.0) {
            break;
        }

        else if (fun(m) * fun(p) < 0) {
            q = m;
        }
        else {
            p = m;
        }
        operations++;
    }
    solution s;
    s.result = m;
    s.operations = operations;
    return s;
}
```

Dla każdej z funkcji i dla dokładności 10^{-7} funkcja obliczyła następujące miejsca zerowe:

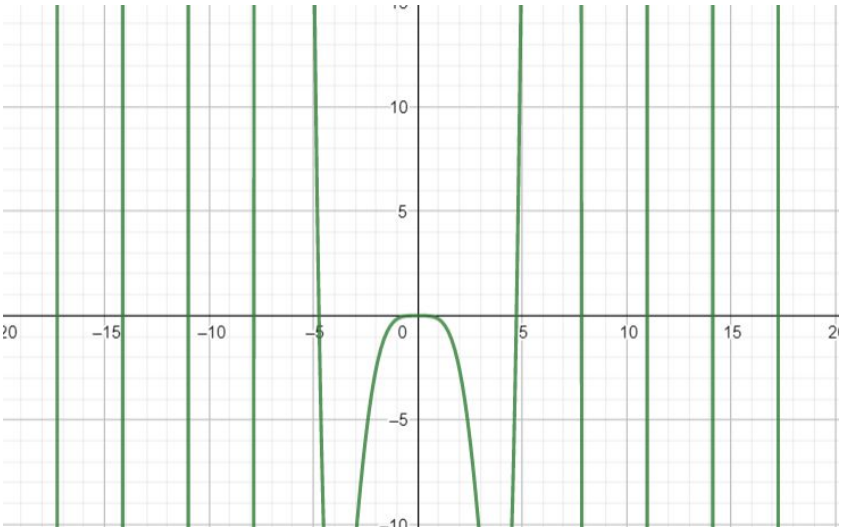
4.7300410 dla funkcji f_1
0.8603333 dla funkcji f_2
1.8293829 dla funkcji f_3

Są to wyniki poprawne, dlatego przyjmuję, że zaimplementowana funkcja działa poprawnie. Następnie dla każdej z funkcji została obliczona wymagana liczba operacji do uzyskania dokładności na poziomie 10^{-7}, 10^{-15} oraz 10^{-33}. wyniki zostały przedstawione w tabeli poniżej:

	Funkcja 1	Funkcja 2	Funkcja 3
10^{-7}	21	21	21
10^{-15}	48	48	48

Dla dokładności 10^{-33} nie udało się obliczyć wymaganej liczby iteracji, gdyż dokładność przekracza dokładność typu double oraz long double.

Wykres funkcji f_1 wygląda następująco:



W celu obliczenia k pierwszych pierwiastków tej funkcji należy zdefiniować odpowiednie przedziały $[a, b]$, w których występują pierwiastki (tj. takie a i b, że $f(a) * f(b) < 0$), i uruchomić w tych przedziałach algorytm bisekcji.
W tym celu można ustalić dowolny $\epsilon > 0$, oraz krańce przedziału, tj. $a = 0$ i $b = a + \epsilon$. Dopóki warunek $f(a) * f(b) < 0$ nie jest spełniony, należy do prawego krańca przedziału dodawać ϵ. Jeśli jest spełniony, należy uruchomić algorytm bisekcji, który wyznaczy dokładny pierwiastek. Następnie przypisujemy $a = b$ oraz $b = b + \epsilon$ i powtarzamy powyższe kroki aż do uzyskania k pierwiastków.

Należy jednak dobrać odpowiednie ϵ, ponieważ gdy ϵ będzie za duży mogą wystąpić błędy w wyznaczaniu pierwiastków np. gdy w przedziale $[a, a + \epsilon]$ będzie występował pierwiastek, ale $f(a) * f(a + \epsilon) > 0$.

3. Metoda Newtona

Zaimplementowana została metoda Newtona, przyjmująca: wartość, od której metoda zaczyna iterować, funkcję, której miejsce zerowe obliczamy, wartość dopuszczalnej różnicy między dwoma kolejnymi iteracjami oraz maksymalna liczba iteracji.

```
solution newtonMethod(double x, double fun(double), double epsilon, int max)
{
    double h = fun(x) / derivFunc(fun, x);
    double oldX = x;
    int operations = 0;
    do
    {
        oldX = x;
        x = x - fun(x) / derivFunc(fun, x);
        operations++;
    } while (abs(x - oldX) >= epsilon && operations <max);

    solution s;
    s.result = x;
    s.operations = operations;
    return s;
}
```

Dla każdej z funkcji, dla podanego lewego krańca przydziału i dla dokładności 10^{-7} funkcja obliczyła następujące miejsca zerowe:

4.7300407 dla f_1
-nan(ind) dla f_2
1.8293836 dla f_3

Natomiast dla prawego krańca:

4.7300407 dla f_1
1.5707963 dla f_2
1.8293836 dla f_3

Zauważamy, że funkcja podała złe odpowiedzi w obu przypadkach dla funkcji f_2. Dzieje się tak, ponieważ pierwszym krokiem algorytmu jest wyliczenie wartości funkcji w punkcie x_0 - w obu przypadku jest to niemożliwe, ponieważ w pierwszym przypadku dzielimy przez 0, w drugim obliczamy $tg(\frac{\pi}{2})$.

Zastosuję jednak drobną zmianę, dodając do lewego krańca przedziału wartość 10^{-6}, a odejmę tę wartość od prawego krańca.

Po tej modyfikacji metoda Newtona prawidłowo oblicza miejsce zerowe funkcji f_2.

Następnie porównuję ilość iteracji, które musi wykonać metoda Newtona, z ilością iteracji metody bisekcji.

Pomiary dla dokładności 10^{-7}:

	Funkcja 1	Funkcja 2	Funkcja 3
Bisekcja	21	21	21
Metoda Newtona	6	20	6

Pomiary dla dokładności 10^{-15}:

	Funkcja 1	Funkcja 2	Funkcja 3
Bisekcja	48	48	48
Metoda Newtona	8	21	7

Możemy zauważyć, że w każdym przypadku metoda Newtona potrzebowała mniej iteracji aby obliczyć miejsce zerowe danej funkcji.

Metoda Newtona ma ograniczenia związane z pochodną danej funkcji.
Metoda bisekcji jest wolniejsza, ale nie ma żadnych ograniczeń.

W naszym przypadku metoda Newtona działała błędnie, ponieważ krańce przedziału f_2 nie należały do dziedziny.

4. Metoda siecznych

Zaimplementowana została metoda siecznych, przyjmująca: krańce przedziału, funkcję, której miejsce zerowe obliczamy, wartość dopuszczalnej różnicy między dwoma kolejnymi iteracjami oraz maksymalna liczba iteracji.

```
solution secantMethod(double x1, double x2, double fun(double), double
precision, int maxiterations) {
    if (fun(x1) * fun(x2) > 0.0f)
    {
        throw std::invalid_argument("Wrong interval");
    }
    double x = x1;
    double xn1 = x1;
    double xn = x2;
    int iterations = 0;
    double oldX;
    do {
        oldX = x;
        x = xn - ((fun(xn) * (xn - xn1)) / (fun(xn) - fun(xn1)));
        xn1 = xn;
        xn = x;
        iterations++;
    } while (abs(x - oldX) > precision && iterations < maxiterations);
    solution s;
    s.iterations = iterations;
    s.result = x;
    return s;
}
```

Dla każdej z funkcji, dla podanych krańców przedziałów i dla dokładności 10^{-7} funkcja obliczyła następujące miejsca zerowe:

4.7300407 dla f_1
-nan(ind) dla f_2
1.8293836 dla f_3

Jak możemy zauważyć, dla funkcji f_2 metoda podobnie jak przy metodzie Newtona - podaje wynik - nan. Przyczyna jest identyczna jak w metodzie Newtona.

Ponownie zastosuję zmianę, dodając do lewego krańca przedziału wartość 10^{-6},, a odejmę tę wartość od prawego krańca.

Po tej modyfikacji metoda siecznych prawidłowo oblicza miejsce zerowe funkcji f_2

Następnie porównuję ilość iteracji, które musi wykonać metoda siecznych, z ilością iteracji bisekcji i metody Newtona.

Pomiary dla dokładności 10^{-7}:

	Funkcja 1	Funkcja 2	Funkcja 3
Bisekcja	21	21	21
Metoda Newtona	6	20	6
Metoda siecznych	5	6	9

Pomiary dla dokładności 10^{-15}:

	Funkcja 1	Funkcja 2	Funkcja 3
Bisekcja	48	48	48
Metoda Newtona	8	21	7
Metoda siecznych	7	7	11

Możemy zauważyć, że prawie dla każdego przypadku metoda siecznych potrzebowała najmniej iteracji. Jedynie dla f_3 lepsza okazywała się metoda Newtona, może to jednak mieć związek z faktem, że dla f_3 przedział był największy.

Różnica między metodą Newtona, a metodą siecznych polega na tym, że metoda Newtona wykorzystuje pochodną do obliczenia linii stycznej, podczas gdy metoda siecznych wykorzystuje przybliżenie liczbowe pochodnej w oparciu o dwa punkty.

W naszym przypadku metoda siecznych działała błędnie, ponieważ krańce przedziału f_2 nie należały do dziedziny.