# Laboratorium 3

Karol Hamielec

4/6/2020

## zad1 - Metoda Bisekcji

KOD:

```
double zad1(double a, double b, double eps, double (func)(double x), int &iter_num){

    iter_num = 0;

    while( my_abs(a-b) > eps){
        iter_num++;
        // std::cout << "iteration: " << iter_num << std::endl;
        double x1 = (a+b) / 2;

        if( my_abs(func(x1)) <= eps){
            break;
        }else if( func(x1) * func(a) < 0){
            b = x1;
        }else{
            a = x1;
        }
    }
    return ((a+b) / 2);
}
```

```
double my_abs(double x){
    if(x > 0){
        return x;
    }else{
        return -x;
    }
}
```

Aby znaleźć pierwsze pierwiastki możemy najpierw za $a$ przyjąć $a = \epsilon$, aby nie szukać w zerze. Następnie za b przyjąć sobie jakąś $\Delta$ np. = 1. I zaczynamy szukać pierwiasta w tym obszarze. Jeśli znajdziemy $x_1$ szukamy pierwiastka w $a$ równym dalej $\epsilon$, ale za b przyjmujemy $x_1 - \epsilon$ i szukamy tak w przediałach coraz bliższych zera. Jeśli znaleźliśmy za mało pierwiastków to szukamy w $a = 1 + \epsilon$, a $b = 2$, czyli $b = 2\Delta$ i tak powtarzamy algorytm do znalezienia zadanej liczby pierwiastków. Jeśli w pierwszym przedziale ($a = \epsilon$, $b = \Delta$) znalazło się za dużo pierwiastków to po prostu bierzemy pierwsze k pierwiastków najbliżej zera.

Wynik:

```
FUNCTON: 0

epsilon: 10^-7
result: 4.730040761 steps: 24

epsilon: 10^-15
result: 4.730040745 steps: 51

FUNCTON: 1

epsilon: 10^-7
result: 0.8603336024 steps: 24

epsilon: 10^-15
result: 0.860333589 steps: 51

FUNCTON: 2

epsilon: 10^-7
result: 1.829383612 steps: 23

epsilon: 10^-15
result: 1.829383602 steps: 51
```

# zad2 - Metoda Newtona

Kod:

```
double zad2(double a, double b, double eps, double(*func)(double x), int &iter_done,
int iter_todo){
    double x1;
    double x2;

    x1 = a;
    x2 = b;
    iter_done = 0;
    while(my_abs(func(x2)) > eps && my_abs(x2 - x1) >= eps && iter_done < iter_todo){
        x1 = x2;
        x2 = x1 - func(x1)/derivative(func, x1);
        iter_done++;
    }
    return x2;

}
```

```
double derivative(double (*func)(double x), double x0){
    double h = 1.0e-10;
    return (func(x0+h) - func(x0))/h;
}
```

Metoda Newtona ma zbieżność kwadratową, jest szybsza i wydajniejsza od metody bisekcji, która ma zbieżność liniową. Zbieżność niestety nie zawsze zachodzi np. kiedy punkt startowy jest zbyt daleko od szukanego pierwiastka równania.

Wynik:

```
FUNCTION: 0

epsilon: 10^-7
result: 4.730040745 steps: 6

epsilon: 10^-15
result: 4.730040745 steps: 8

FUNCTION: 1

epsilon: 10^-7
result: 1.570796327 steps: 1

epsilon: 10^-15
result: 3.425618459 steps: 38

FUNCTION: 2

epsilon: 10^-7
result: 1.829383602 steps: 6

epsilon: 10^-15
result: 1.829383602 steps: 7
```

# zad3 - Metoda siecznych

Kod:

```
double zad3(double a, double b, double eps, double(*func)(double x), int &iter_done,
int iter_todo){
    double x1;
    double x2;
    double x3;

    x1 = a;
    x2 = b;
    x3 = a;
    iter_done = 0;
    while(my_abs(func(x3)) >= eps && my_abs(x3 - x2) >= eps && iter_done < iter_tod
o){
        x1 = x2;
        x2 = x3;
        x3 = (func(x2)*x1 - func(x1)*x2) / (func(x2) - func(x1));
        iter_done++;
    }
    return x3;

}
```

Wynik:

```
FUNCTION: 0

epsilon: 10^-7
result: 4.730040744 steps: 4

epsilon: 10^-15
result: 4.730040745 steps: 6

FUNCTION: 1

epsilon: 10^-7
result: -nan steps: 1

epsilon: 10^-15
result: -nan steps: 1

FUNCTION: 2

epsilon: 10^-7
result: 1.829383593 steps: 9

epsilon: 10^-15
result: 1.829383602 steps: 11
```

Metoda również nie zawsze jest zbieżna, co widać na przykładzie funkcji 1. Liczby a i b muszą być wystarczająco blisko pierwiastka. Zbieżność jest szybsza od zbieżności liniowej, ale nie jest kwadratowa. Co również widać w liczbie potrzebnych iteracji - Metoda Newtona zbiega najszybciej, potem metoda siecznych i na końcu metoda bisekcji

normal