# *MOWNIT - Zestaw 2 Root Finding*

*Opracował: Mateusz Woś*

Wszystkie zadania zostały zaimplementowane w języku Python.

1. Napisać program do obliczenia pierwiastków równań x^2-5 = 0 oraz x^2-2*x+1=0 metodą bisekcji
    - Wyjaśnić działanie programu - dlaczego nie może znaleźć miejsc zerowych dla drugiego równania?

Na początek zdefiniowałem funkcje $x^2 - 5 = 0$ oraz $x^2 - 2x + 1 = 0$

```
def function1(x):
    return x ** 2 - 5


def function2(x):
    return x ** 2 - 2 * x + 1
```

Następnie zdefiniowałem metodę bisekcji.

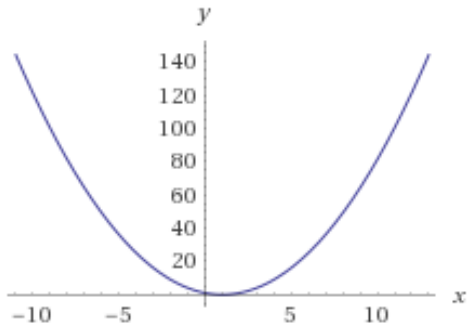
```
def check_sign(a, b):
    return a * b <= 0


def bisection(func, x_s, x_e, epsilon):
    assert check_sign(func(x_s), func(x_e))
    midpoint = x_s
    while abs(x_e - x_s) > epsilon:
        midpoint = (x_s + x_e) / 2.0
        if abs(func(midpoint)) <= epsilon:
            return midpoint
        elif not check_sign(func(x_s), func(midpoint)):
            x_s = midpoint
        else:
            x_e = midpoint
    return midpoint
```

```
Bisection method

function1
Root x = 2.2360801696777344 and its calculated value(should be close to 0):
5.4525226005353034e-05

function2
AssertionError
```

Funkcja *check_sign* sprawdza jedno z założeń metody bisekcji.
Funkcja, którą badamy musi przyjmować różne wartości na końcach przedziału $[a, b]$

$$f(a)f(b) < 0$$

Metoda bisekcji nie jest wystarczalna do obliczenia miejsca zerowego drugiej funkcji, ponieważ nie da się spełnić powyższego założenia funkcji *check_sign.* Wartości funkcji na całym swoim przedziale nie zmieniają znaku.




(*x* from −11 to 13)

Range:

$\{y \in \mathbb{R} : y \geq 0\}$ (all non-negative real numbers)

2. Zmienić program tak, aby znajdował pierwiastek metodą siecznych.

```
def secant(func, x_s, x_e, epsilon, max_iterations=100):
    fx0 = func(x_s)
    fx1 = func(x_e)
    for _ in range(max_iterations):
        if abs(fx1) < epsilon:
            return x_e
        try:
            denominator = (fx1 - fx0) / (x_e - x_s)
            x2 = x_e - fx1 / denominator
        except ZeroDivisionError:
            print("fx1 - fx0 error")
            raise

        x_s, x_e = x_e, x2
        fx0, fx1 = fx1, func(x_e)
    return x_e
```

```
Secant method

function1
Root x = 2.236067970034716 and its calculated value(should be close to 0):
-3.3384824682514136e-08

function2
Root x = 0.9955456570155872 and its calculated value(should be close to 0):
1.984117142284081e-05
```

W przypadku metody siecznych nie ma już problemu znalezienia miejsc zerowych obu funkcji.
Metoda ta polega na przyjęciu, że funkcja na dostatecznie małym odcinku $[a, b]$ w przybliżeniu zmienia się w sposób liniowy.
Wtedy na odcinku $[a, b]$ krzywą $y = f(x)$ możemy zastąpić sieczną. Za przybliżoną wartość pierwiastka przyjmujemy punkt przecięcia siecznej z osią OX.
Niestety metoda zawodzi jeżeli $f(x_n) = f(x_{n-1})$

3. Napisać program szukający miejsc zerowych za pomocą jednej z metod korzystających z pochodnej funkcji. Czym różni się od poprzednich metod i dlaczego potrafi znaleźć pierwiastek równania x^2-2*x+1=0?
    - Porównać efektywność użytych metod dla wybranych dokładności w postaci ilości iteracji oraz zużytego czasu CPU.
    - Przeanalizować jakie kryterium stosuje się do zakończenia obliczeń w każdym konkretnym przypadku

Na samym początku zdefiniowałem funkcję zwracającą przybliżoną wartość pochodnej w punkcie.

```
def derivative_fun(func, x, h=1e-6):
    return (func(x + h) - func(x - h)) / (2 * h)
```

Do obliczenia miejsc zerowych wybrałem metodę Newtona-Raphsona

```
def newton_raphson(func, x0, epsilon, max_iterations=100):
    for _ in range(max_iterations):
        x1 = x0 - (func(x0) / derivative_fun(func, x0))
        if abs(x1 - x0) < epsilon:
            return x1
        else:
            x0 = x1
    return x1
```

```
Newton-Raphson method

function1
Root x = 2.2360679774999785 and its calculated value(should be close to 0):
8.446576771348191e-13

function2
Root x = 1.0000152587939901 and its calculated value(should be close to 0):
2.3283086569847455e-10
```

Metoda różni się od poprzednich tym, iż wykorzystuję do obliczeń wartość pochodnej w zadanym punkcie.  Potrafi znaleźć pierwiastek drugiego równania, ponieważ nie wymaga, aby wartości na krańcach przedziałów miały przeciwny znak.

a)
Do mierzenia zużytego czasu CPU posłużyłem się prostym dekoratorem.
Aby otrzymać ilość iteracji, po której otrzymałem wynik zmodyfikowałem funkcję, aby zwracały dodatkowo numer iteracji.

```
from time import process_time

def time_decorator(fun):
    def wrapper(*args, **kwargs):
        start = process_time()
        res = fun(*args, **kwargs)
        print("Function name: {0}, CPU time: {1}"
              .format(fun.__name__, process_time() - start))
        return res

    return wrapper
```

Czas CPU zmierzyłem dla 10000 powtórzeń funkcji. Każda z funkcji dostała te same dane wejściowe.
Epsilon = 0.00003

Czas CPU dla pierwszej funkcji:

```
Function name: bisection_10k_iterations, CPU time: 0.234375
Function name: secant_10k_iterations, CPU time: 0.046875
Function name: newton_10k_iterations, CPU time: 0.0625
```

Czas CPU dla drugiej funkcji:

```
Function name: secant_10k_iterations, CPU time: 0.109375
Function name: newton_10k_iterations, CPU time: 0.21875
```

Ilość iteracji:
- Metoda bisekcji

```
Number of iterations for: function1 = 18
```

-Metoda siecznych

```
Number of iterations for: function1 = 7
Number of iterations for: function2 = 14
```

-Metoda Newtona-Raphsona

```
Number of iterations for: function1 = 4
Number of iterations for: function2 = 17
```

Widać tutaj, iż metoda siecznych jest najszybsza. Prawdopodobnie jest to spowodowane brakiem obliczania wartości pochodnej w punkcie, a także mniejszą liczbą iteracji. Brak konieczności liczenia pochodnej najlepiej zaobserwować przy pierwszej funkcji. Mimo większej liczby iteracji metoda siecznych nadal była szybsza.

b)
W pierwszej metodzie do zakończenia obliczeń posłużyłem się kryterium (x_koncowe – x_poczatkowe) < epsilon.
Gdy funkcja chce zacząć dzielić przedział na mniejsze obszary, niż wielkość epsilona iteracja zostanie zakończona. W tej metodzie mógłbym też odgórnie założyć ilość iteracji i na tej podstawie kończyć działanie algorytmu.
Dodatkowo sprawdzam podczas iteracji czy wartość w wyznaczonym punkcie nie jest już mniejsza niż odgórnie założona tolerancja.

W metodzie siecznych za kryterium końcowe przyjąłem ilość iteracji, która może być zdefiniowana przez użytkownika. Dodatkowo zabezpieczyłem się przed niepotrzebnymi iteracjami tolerancją. Sprawdzam czy w danej iteracji wartość funkcji w wyznaczonym punkcie (miejscu zerowym) nie jest już mniejsza niż zadany epsilon.

W metodzie Newtona-Raphsona tak jak w poprzedniej metodzie kryterium jest ilość iteracji. Sprawdzam też podczas iteracji czy różnica prawdopodobnego miejsca zerowego i poprzedniej jego wartości jest mniejsza od zadanego epsilonu. Jeśli dojdzie do takiej sytuacji, zwracam natychmiastowo wynik.