

# Metody Numeryczne - Zadanie XVII

Mateusz Kamiński

December 2025

## 1 Wstęp

W zadaniu należało znaleźć z dokładnością do  $10^{-6}$ , minimum funkcji

$$y = \frac{1}{4}x^4 - \frac{1}{2}x^2 - \frac{1}{16}x$$

Metodami złotego podziału oraz Brenta

## 2 Opis

### 2.1 Złoty podział

Do wyznaczenia minimum funkcji zastosowano algorytm oparty na metodzie złotego podziału. Metoda ta polega na iteracyjnym zawężaniu przedziału  $[a, c]$  zawierającego minimum funkcji, przy wykorzystaniu stałej

$$w = \frac{3 - \sqrt{5}}{2}.$$

W każdej iteracji wyznaczany jest nowy punkt:

$$d = \begin{cases} a + w|b - a|, & \text{gdy } |b - a| > |c - b|, \\ b + w|c - a|, & \text{w przeciwnym razie.} \end{cases}$$

Następnie porównywane są wartości funkcji  $f(b)$  oraz  $f(d)$ , a przedział poszukiwań jest odpowiednio zawężany. Iteracje są wykonywane do momentu spełnienia kryterium stopu

$$|c - a| < \varepsilon(|b| + |d|)$$

Metoda charakteryzuje się pewną, lecz liniową zbieżnością

## 2.2 Metoda Brenta

Dla trzech punktów  $a$ ,  $b$ ,  $c$  wykonywana jest interpolacja paraboliczna, a punkt minimum paraboli wyznaczany jest ze wzoru:

$$d = \frac{1}{2} \cdot \frac{a^2(f_c - f_b) + b^2(f_a - f_c) + c^2(f_b - f_a)}{a(f_c - f_b) + b(f_a - f_c) + c(f_b - f_a)}.$$

Krok interpolacyjny jest akceptowany tylko wtedy, gdy punkt  $d$  leży wewnątrz przedziału  $(a, c)$  oraz prowadzi do istotnego skrócenia przedziału. W przeciwnym przypadku wykonywany jest krok bisekcji:

$$d = \frac{a + c}{2}.$$

Algorytm kończy działanie po spełnieniu warunku:

$$|c - a| < \varepsilon(|b| + |d|).$$

Zastosowanie interpolacji parabolicznej pozwala znacząco zmniejszyć liczbę iteracji w porównaniu z algorytmem opartym wyłącznie na złotym podziale.

## 3 Implementacja

```
import numpy as np

num = np.float64

def horner(x: num) -> num:
    coeffs = np.array([0.25, 0.0, -0.5, -1/16, 0.0])
    p = coeffs[0]

    for i in range(1, len(coeffs)):
        p = p * x + coeffs[i]

    return p

def f(x: num) -> num:
    return horner(x)
```

```
# def f(x: num) -> num:
#     return 0.25 * x**4 - 0.5 * x**2 - (1/16) * x

def golden_ratio(a: num, b: num, c: num, tolerance: float = 1e-6) -> tuple[num, int]:
    w = (3 - np.sqrt(5)) / 2

    i = 0
    while True:
        i += 1
        if abs(b - a) > abs(c - b):
            d = a + w * abs(b - a)
        else:
            d = b + w * abs(c - b)

        if abs(c - a) < tolerance * (abs(b) + abs(d)):
            break

        fb, fd = f(b), f(d)

        if fd < fb:
            if d < b:
                c, b = b, d
            else:
                a, b = b, d

        else:
            if d < b:
                a = d
            else:
                c = d

    return b, i

def brent(a: num, b: num, c: num, tolerance: float = 1e-6) -> tuple[num, int]:
    prev_interval = abs(c - a)
    prev2_interval = prev_interval

    i = 0
    while True:
        i += 1

        fa, fb, fc = f(a), f(b), f(c)
        old_b = b

        # interpolacja paraboliczna
        denominator = a * (fc - fb) + b * (fa - fc) + c * (fb - fa)

        accept = False
        if abs(denominator) > 1e-14:
```

```

(fb - fa)    numerator = a ** 2 * (fc - fb) + b ** 2 * (fa - fc) + c ** 2 *
              d = numerator / (2*denominator)

              if a < d < c:
                  new_interval = max(abs(d - a), (c - d))
                  if new_interval <= 0.5 * prev2_interval:
                      accept = True

              if not accept:
                  d = (c + a) / 2.0

              fd = f(d)

              if fd < fb:
                  if d < b:
                      c, b = b, d
                  else:
                      a, b = b, d

              else:
                  if d < b:
                      a = d
                  else:
                      c = d

              prev2_interval = prev_interval
              prev_interval = abs(c - a)

              if abs(c - a) < tolerance * (abs(old_b) + abs(d)):
                  break

              return b, i

def main():
    a, b, c = 0.0, 1.0, 2.5
    tolerance = 1e-6

    gold_min, gold_iterations = golden_ratio(a, b, c, tolerance)
    brent_min, brent_iterations = brent(a, b, c, tolerance)

    print(f"Złoty podział: x = {gold_min:.8f}, iteracje = {gold_iterations}")
    print(f"Metoda Brenta: x = {brent_min:.8f}, iteracje = {brent_iterations}")

    a, b, c = -4.0, -1.5, -0.5
    gold_min, gold_iterations = golden_ratio(a, b, c, tolerance)
    brent_min, brent_iterations = brent(a, b, c, tolerance)

    print(f"Złoty podział: x = {gold_min:.8f}, iteracje =

```

```
{gold_iterations}")
    print(f"Metoda Brenta: x = {brent_min:.8f}, iteracje =
{brent_iterations}")

if __name__ == "__main__":
    main()
```

## 4 Wyniki

### 4.1 Rozwiązania

Dla punktów:  $a = 0.0$ ,  $b = 1.0$ ,  $c = 2.5$

złoty podział:  $x_{\min} = 1.02989591$ , iteracje = 40  
metoda Brenta:  $x_{\min} = 1.02989598$ , iteracje = 27

Dla punktów:  $a = -4.0$ ,  $b = -1.5$ ,  $c = -0.5$

złoty podział :  $x_{\min} = -0.96714900$ , iteracje = 39  
metoda Brenta :  $x_{\min} = -0.96714894$ , iteracje = 19

### 4.2 Podsumowanie

Jak widać algorytm złotego podziału potrzebował odpowiednio 40 oraz 39 iteracji, podczas gdy metoda Brenta kolejno 27 oraz 19 iteracji. Zatem zbieżność w metodzie Brenta jest prawie dwukrotnie szybsza od złotego podziału.