

Metody Numeryczne

Laboratorium 8: Rozwiązywanie równań nieliniowych

Wykonał: <imię i nazwisko>

Metoda zaliczenia:

Podczas zajęć należy wykonać poniższe polecenia oraz udzielić odpowiedzi na pytania zamieszczone w treści zadań.

Wszystkie funkcje wymagane w ramach ćwiczenia należy zaimplementować w pliku `main.py`. Poprawności ich działania należy zweryfikować za pomocą testów jednostkowych dostępnych w pliku `test_main.py`.

Cel zajęć:

Celem laboratorium jest poznanie numerycznych metod rozwiązywania układów równań nieliniowych. W ramach laboratorium przedstawione zostaną metody: *bisekcji*, *siecznych* oraz *Newtona*.

Tematem wiodącym podczas tych zajęć będzie poszukiwanie miejsc zerowych funkcji zdefiniowanej w następujący sposób:

$$f(x) = e^{-2x} + x^2 - 1$$

Funkcja $f(x)$ oraz jej pierwsza i druga pochodna ($f'(x)$, $f''(x)$) zostały zaimplementowane w pliku `main.py`, odpowiednio jako `func()`, `dfunc()` i `ddfunc()`.

Uwagi wstępne:

- Funkcje wymagające implementacji (lub zaimplementowane na poprzednich laboratoriach) oznaczone są pochylą czcionką maszynową (np. `my_func()`).
- W skrypcie stosuje się następujące skróty:
 - `np` - `numpy`,
 - `sp` - `scipy`.

Na wykresie przedstawiono funkcję $f(x)$ oraz jej pierwszą i drugą pochodną. Na podstawie analizy wykresu funkcji z osi OX widać dwa miejsca zerowe. Pierwsze znajduje się w pobliżu zera, a drugie po prawej stronie wykresu. Analizując zmiany znaku funkcji, a także kształt pochodnych, można zgrubnie określić przedziały, w których znajdują się pierwiastki funkcji.

Wyznaczone przedziały miejsc zerowych: pierwszy pierwiastek: (-0.5, 0.3) drugi pierwiastek: (0.5, 1.0)

Pierwsza pochodna pomaga określić monotoniczność funkcji, a druga pochodna informuje o wypukłości i może wskazać lepszy punkt startowy dla metod numerycznych.

Zadanie 2.

Najprostszą metodą do wyznaczenia miejsc zerowych funkcji nieliniowej jest *metoda bisekcji*.

Zaimplementuj funkcję `bisection()` pamiętając, że gwarancją zbieżności metody bisekcji dla poszukiwania miejsca zerowego funkcji $f(x)$ na odcinku $[a, b]$ są następujące założenia:

- Funkcja $f(x)$ jest ciągła w przedziale domkniętym $[a, b]$.
- Funkcja $f(x)$ przyjmuje różne znaki na końcach przedziału: $f(a)f(b) < 0$.

```
In [44]: def bisection(
a: int | float,
b: int | float,
f: Callable[[float], float],
epsilon: float,
max_iter: int,
) -> tuple[float, int] | None:
    if not isinstance(a, (float, int)) or not isinstance(b, (float, int)):
        return None
    if not isinstance(epsilon, float):
        return None
    if not isinstance(max_iter, int):
        return None
    if not callable(f):
        return None
    if a == b:
        return None
    if epsilon < 0:
        return None
    if max_iter <= 0:
        return None

    if f(a) * f(b) >= 0:
        return None

    counter = 0
    x0 = (a + b) / 2
    while abs(f(x0)) > epsilon and counter < max_iter:
        counter += 1
        if f(a) * f(x0) < 0:
            b = x0
        else:
            a = x0
        x0 = (a + b) / 2
    return x0, counter + 1
```

Na początku sprawdzane są wszystkie warunki poprawności danych wejściowych: typy argumentów, poprawność epsilon i `max_iter`, kolejność przedziału oraz to, czy funkcja zmienia znak na końcach przedziału. Następnie ustawiany jest licznik iteracji i wyznaczany pierwszy punkt jako środek przedziału. W pętli obliczana jest wartość funkcji w aktualnym punkcie i sprawdzane są warunki stopu: osiągnięcie wymaganej dokładności lub przekroczenie maksymalnej liczby iteracji. W każdej iteracji zawieszany jest przedział do tej części, w której funkcja zmienia znak. Nowy środek przedziału stale się kolejnym przybliżeniem rozwiązania. Po zakończeniu algorytm zwraca znaleziony pierwiastek i liczbę wykonanych iteracji.

Zadanie 3.

Bardziej zaawansowaną metodą wyznaczania miejsc zerowych jest *metoda siecznych*, która stanowi rozwinięcie metody Regula Falsi.

Zaimplementuj funkcję `secant()` pamiętając, że gwarancją zbieżności metody siecznych dla poszukiwania miejsca zerowego funkcji $f(x)$ na odcinku $[a, b]$ są następujące założenia:

- Funkcja $f(x)$ jest ciągła w przedziale domkniętym $[a, b]$.
- Funkcja $f(x)$ przyjmuje różne znaki na końcach przedziału: $f(a)f(b) < 0$.
- Pierwsza i druga pochodna funkcji $f(x)$ są ciągłe w przedziale domkniętym $[a, b]$.
- Pierwsza i druga pochodna funkcji $f(x)$ w przedziale domkniętym $[a, b]$ mają stały znak i są różne od zera.

```
In [45]: def secant(
a: int | float,
b: int | float,
f: Callable[[float], float],
epsilon: float,
max_iters: int,
) -> tuple[float, int] | None:
    if not isinstance(a, (float, int)) or not isinstance(b, (float, int)):
        return None
    if not callable(f):
        return None
    if a == b:
        return None
    if epsilon < 0:
        return None
    if max_iters <= 0:
        return None

    fa = f(a)
    fb = f(b)
    if fa * fb >= 0:
        return None

    x = (a * fb - b * fa) / (fb - fa)
    fx = f(x)
    counter = 1
    while abs(fx) > epsilon and counter < max_iters:
        if fa * fx < 0:
            b = x
            fb = fx
        else:
            a = x
            fa = fx
        x = (a * fb - b * fa) / (fb - fa)
        fx = f(x)
        counter += 1
    return x, counter
```

W metodzie siecznych na początku sprawdzane są warunki poprawności danych: typy argumentów, poprawność epsilon i `max_iters`, kolejność przedziału oraz to, czy funkcja zmienia znak na końcach przedziału. Następnie obliczane są wartości funkcji w punktach a i b . Na tej podstawie wyznaczane jest pierwsze przybliżenie miejsca zerowego za pomocą wzoru opartego na siecznej przechodzącej przez punkty $(a, f(a))$ i $(b, f(b))$. W pętli w kolejnych iteracjach aktualizujemy przedział, zastępując jeden z punktów na taki sam znak jak w bieżącym przybliżeniu. Ponownie obliczamy nowe przybliżenie według wzoru siecznych. Proces trwa aż do osiągnięcia wymaganej dokładności lub przekroczenia maksymalnej liczby iteracji. Na końcu zwracamy znaleziony pierwiastek i liczbę wykonanych kroków.

Zadanie 4.

Inną metodą, wykorzystywaną do poszukiwania miejsc zerowych funkcji, jest *metoda Newtona*, nazywana również metodą stycznych.

Podpunkt 1.

Algorytm metody Newtona wykorzystuje wartości pierwszej pochodnej, dlatego przed przystąpieniem do jej implementacji przygotuj pomocniczą funkcję `difference_quotient()`, służącą do wyznaczenia wartości ilorazu różnicowego.

Podpunkt 2.

Zaimplementuj funkcję `newton()` pamiętając, że gwarancją zbieżności metody Newtona dla poszukiwania miejsca zerowego funkcji $f(x)$ na odcinku $[a, b]$ są następujące założenia:

- Funkcja $f(x)$ jest ciągła w przedziale domkniętym $[a, b]$.
- Funkcja $f(x)$ przyjmuje różne znaki na końcach przedziału: $f(a)f(b) < 0$.
- Pierwsza i druga pochodna funkcji $f(x)$ są ciągłe w przedziale domkniętym $[a, b]$.
- Pierwsza i druga pochodna funkcji $f(x)$ w przedziale domkniętym $[a, b]$ mają stały znak i są różne od zera.

```
In [46]: def difference_quotient(
f: Callable[[float], float], x: int | float, h: int | float
) -> float | None:
    if not isinstance(x, (float, int)) or not isinstance(h, (float, int)):
        return None
    if not callable(f):
        return None
    if h == 0:
        return None
    return (f(x + h) - f(x)) / h

def newton(
f: Callable[[float], float],
df: Callable[[float], float],
ddf: Callable[[float], float],
a: int | float,
b: int | float,
epsilon: float,
max_iter: int,
) -> tuple[float, int] | None:
    """Funkcja aproksymująca rozwiązanie równania f(x) = 0 metodą Newtona.

    Args:
        f (Callable[[float], float]): Funkcja, dla której poszukiwane jest rozwiązanie.
        df (Callable[[float], float]): Pierwsza pochodna funkcji, dla której poszukiwane jest rozwiązanie.
        ddf (Callable[[float], float]): Druga pochodna funkcji, dla której poszukiwane jest rozwiązanie.
        a (int | float): Początek przedziału.
        b (int | float): Koniec przedziału.
        epsilon (float): Tolerancja liczba maszynowego (warunek stopu).
        max_iter (int): Maksymalna liczba iteracji.

    Returns:
        (tuple[float, int]):
            - Aproksymowane rozwiązanie,
            - Liczba wykonanych iteracji.
        Jeżeli dane wejściowe są niepoprawne funkcja zwraca 'None'.

    """
    if not callable(f):
        return None
    if not isinstance(a, (float, int)) or not isinstance(b, (float, int)):
        return None
    if f(a) * f(b) >= 0:
        return None
    if a == b:
        return None
    if not isinstance(epsilon, float):
        return None
    if not isinstance(max_iter, int):
        return None
    if epsilon < 0:
        return None
    if max_iter <= 0:
        return None

    fa = f(a)
    fb = f(b)
    dda = ddf(a)
    ddb = ddf(b)

    if fa * dda > 0:
        x = float(a)
    elif fb * ddb > 0:
        x = float(b)
    else:
        x = (a + b) / 2

    for it in range(1, max_iter + 1):
        fx = f(x)
        dfx = df(x)

        if dfx == 0:
            return None
        x_next = x - fx / dfx
        if abs(f(x_next)) <= epsilon:
            return (x_next, it)
        x = x_next

    return (x, max_iter)
```

Najpierw przygotowana jest funkcja `difference_quotient`, która oblicza iloraz różnicowy dla zadanej funkcji $f(x)$. Funkcja sprawdza poprawność argumentów, a następnie zwraca wartość $(f(x + h) - f(x)) / h$. Dzięki temu można numerycznie przybliżyć pierwszą pochodną funkcji w wybranym punkcie, co później wykorzystuje się w metodzie Newtona.

W funkcji `newton` najpierw weryfikowane są dane wejściowe: typy argumentów, poprawność epsilon i maksymalnej liczby iteracji oraz to, czy funkcja zmienia znak na końcach przedziału. Następnie wybierany jest punkt startowy na podstawie wartości funkcji i jej drugiej pochodnej, tak aby spełnione były założenia zbieżności. W pętli obliczana jest wartość funkcji i jej pochodnej w bieżącym punkcie, a następnie wyznaczane jest nowe przybliżenie pierwiastka ze wzoru $x_{(n+1)} = x_n - f(x_n) / f'(x_n)$. Iteracje trwają tak długo, aż wartość funkcji w kolejnym przybliżeniu będzie mniejsza od zadanej tolerancji lub zostanie osiągnięta maksymalna liczba kroków. Na końcu funkcja zwraca przybliżone miejsce zerowe oraz liczbę wykonanych iteracji.

Zadanie 5.

Dla funkcji $f(x)$ zdefiniowanej w sekcji Cel zajęć znajdź miejsca zerowe przy użyciu:

- funkcji `sp.optimize.root()`,
- funkcji `sp.optimize.fsolve()`.

```
In [39]: # ===== Twoja implementacja tutaj =====
solution_root = sp.optimize.root(
    fun=main.func, # funkcja f(x)
    x0=-1, # pierwszy punkt startowy
    tol=0.001, # tolerancja
)

print(solution_root.x, solution_root.fvec)

solution_fsolve, info, ier, msg = sp.optimize.fsolve(
    func=main.func,
    x0=-1,
    xtol=0.001,
    maxfev=1000,
    full_output=True
)

print(solution_fsolve, info['fvec'])

[-2.1736944e-17] 15
[-2.1736944e-17] 15
```

Zadanie 6.

Korzystając z przedziałów wyznaczonych w Zadaniu 1. znajdź miejsca zerowe funkcji $f(x)$ zdefiniowanej w sekcji Cel zajęć, przy użyciu:

- metody bisekcji,
- metody siecznych,
- metody Newtona,

z tolerancją równą $1e-10$.

Zbadaj dokładność (względem rozwiązania z Zadania 5) i czas obliczeń metod w zależności od liczby iteracji. Wyniki przedstaw na wykresach.

```
In [1]: fun = main.func
x1_range = [-0.5, 0.3]
x2_range = [0.5, 1.0]
epsilon = 1e-10

x1_true = sp.optimize.root(fun, x1_range[0]).x[0]
x2_true = sp.optimize.root(fun, x2_range[0]).x[0]

iteration_values = np.linspace(2, 100, 20, dtype=int)

time_bis_x1, time_sec_x1, time_new_x1 = [], [], []
time_bis_x2, time_sec_x2, time_new_x2 = [], [], []

error_bis_x1, error_sec_x1, error_new_x1 = [], [], []
error_bis_x2, error_sec_x2, error_new_x2 = [], [], []

for it in iteration_values:
    t = %timeit -r 3 -n 5 -q -o main.bisection(x1_range[0], x1_range[1], fun, epsilon, it)
    time_bis_x1.append(t.average)
    res = main.bisection(x1_range[0], x1_range[1], fun, epsilon, it)
    if res is None:
        error_bis_x1.append(np.nan)
    else:
        root, _ = res
        error_bis_x1.append(abs(root - x1_true))

    t = %timeit -r 3 -n 5 -q -o main.bisection(x2_range[0], x2_range[1], fun, epsilon, it)
    time_bis_x2.append(t.average)
    res = main.bisection(x2_range[0], x2_range[1], fun, epsilon, it)
    if res is None:
        error_bis_x2.append(np.nan)
    else:
        root, _ = res
        error_bis_x2.append(abs(root - x2_true))

    t = %timeit -r 3 -n 5 -q -o main.secant(x1_range[0], x1_range[1], fun, epsilon, it)
    time_sec_x1.append(t.average)
    res = main.secant(x1_range[0], x1_range[1], fun, epsilon, it)
    if res is None:
        error_sec_x1.append(np.nan)
    else:
        root, _ = res
        error_sec_x1.append(abs(root - x1_true))

    t = %timeit -r 3 -n 5 -q -o main.secant(x2_range[0], x2_range[1], fun, epsilon, it)
    time_sec_x2.append(t.average)
    res = main.secant(x2_range[0], x2_range[1], fun, epsilon, it)
    if res is None:
        error_sec_x2.append(np.nan)
    else:
        root, _ = res
        error_sec_x2.append(abs(root - x2_true))

    t = %timeit -r 3 -n 5 -q -o main.newton(fun, main.dfunc, main.ddfunc, x1_range[0], x1_range[1], epsilon, it)
    time_new_x1.append(t.average)
    res = main.newton(fun, main.dfunc, main.ddfunc, x1_range[0], x1_range[1], epsilon, it)
    if res is None:
        error_new_x1.append(np.nan)
    else:
        root, _ = res
        error_new_x1.append(abs(root - x1_true))

    t = %timeit -r 3 -n 5 -q -o main.newton(fun, main.dfunc, main.ddfunc, x2_range[0], x2_range[1], epsilon, it)
    time_new_x2.append(t.average)
    res = main.newton(fun, main.dfunc, main.ddfunc, x2_range[0], x2_range[1], epsilon, it)
    if res is None:
        error_new_x2.append(np.nan)
    else:
        root, _ = res
        error_new_x2.append(abs(root - x2_true))

fig, ax = plt.subplots(2, 1, figsize=(10, 10))

ax[0].plot(iteration_values, error_bis_x1, 'o-', label='Bisekcja')
ax[0].plot(iteration_values, error_sec_x1, 'o-', label='Secant')
ax[0].plot(iteration_values, error_new_x1, 'o-', label='Newton')
ax[0].set_title='Błąd metody vs liczba iteracji (pierwszy pierwiastek)',
        xlabel='Liczba iteracji', ylabel='Błąd bezwzględny')
ax[0].legend()
ax[0].grid()

ax[1].plot(iteration_values, error_bis_x2, 'o-', label='Bisekcja')
ax[1].plot(iteration_values, error_sec_x2, 'o-', label='Secant')
ax[1].plot(iteration_values, error_new_x2, 'o-', label='Newton')
ax[1].set_title='Błąd metody vs liczba iteracji (drugi pierwiastek)',
        xlabel='Liczba iteracji', ylabel='Błąd bezwzględny')
ax[1].legend()
ax[1].grid()

fig.show()
```

fig, ax = plt.subplots(2, 1, figsize=(10, 10))

ax[0].plot(iteration_values, time_bis_x1, 'o-', label='Bisekcja')

ax[0].plot(iteration_values, time_sec_x1, 'o-', label='Secant')

ax[0].plot(iteration_values, time_new_x1, 'o-', label='Newton')

ax[0].set_title='Czas vs liczba iteracji (pierwszy pierwiastek)',

xlabel='Liczba iteracji', ylabel='Średni czas [s]')

ax[0].legend()

ax[0].grid()

ax[1].plot(iteration_values, time_bis_x2, 'o-', label='Bisekcja')

ax[1].plot(iteration_values, time_sec_x2, 'o-', label='Secant')

ax[1].plot(iteration_values, time_new_x2, 'o-', label='Newton')

ax[1].set_title='Czas vs liczba iteracji (drugi pierwiastek)',

xlabel='Liczba iteracji', ylabel='Średni czas [s]')

ax[1].legend()

ax[1].grid()

fig.suptitle('Zestawienie czasu obliczeń metod w zależności od liczby iteracji')

plt.show()

W zadaniu porównano działanie trzech metod numerycznych: bisekcji, siecznych (secant) oraz Newtona. Na podstawie wykresów błęd oraz czasu obliczeń w zależności od liczby iteracji można wyciągnąć kilka prostych wniosków.

Na wykresach błęd metoda siecznych jako jedyna jest widoczna w początkowych iteracjach. Wynika to z tego, że metody Newtona oraz bisekcji bardzo szybko osiągają bardzo mały błąd, często bliski dokładności maszynowej (rzędu $1e-12$ i mniej), dlatego na wykresie wydają się jakby były na osi X, czyli niewidoczne. Oznacza to, że obie metody są bardzo dokładne już przy małej liczbie iteracji. Metoda siecznych także zbiega do rozwiązania, ale potrzebuje więcej kroków, więc jej błędy widać wyraźniej na wykresie.

Na wykresach czasu obliczeń również widać, że metoda Newtona działa najszybciej i jej czas prawie nie zależy od liczby iteracji. Wynika to z jej bardzo szybkiej zbieżności. Metoda bisekcji jest wolniejsza, ale stabilna i zawsze zbliża się do rozwiązania. Metoda siecznych w tej implementacji okazała się najwolniejsza i dodatkowo jej czas jest bardzo zmienny. Powodem jest to, że metoda siecznych wymaga więcej obliczeń, a jej zbieżność nie zawsze jest tak szybka jak w pozostałych metodach.

Podsumowując, metoda Newtona daje najlepsze wyniki: bardzo szybkie obliczenia i bardzo mały błąd. Metoda bisekcji jest wolniejsza, ale stabilna i dokładna. Metoda siecznych zbiega do rozwiązania, ale potrzebuje więcej iteracji i czasu, zwłaszcza w początkowej fazie obliczeń.

Materiały uzupełniające:

- Scipy Lecture Notes
- Numpy for Matlab users
- Python Tutorial - W3Schools
- Numpy
- Matplotlib
- Anaconda
- Learn Python for Data Science
- Learn Python
- Wujek Google i Ciocia Wikipedia