Sprawozdanie z laboratorium 6 - Pierwiastki równań nieliniowych

Hubert Miklas

06-05-2025

1 Wstęp

Tematem laboratorium było rozwiązywanie równań i układów równań nieliniowych, korzystając z metod aproksymacji miejsc zerowych funkcji jednej lub wielu zmiennych.

2 Treści zadań

1. Napisz iterację wg metody Newtona do rozwiązywania każdego z następujących równań nieliniowych:
   (a) $x\cos(x) = 1$;
   (b) $x^3 - 5x - 6 = 0$;
   (c) $e^x = x^2 - 1$.
2. (a) Pokaż, że iteracyjna metoda matematycznie jest równoważna z metodą siecznych przy rozwiązywaniu skalarnego nieliniowego równania $f(x) = 0$.

$$x_{k+1} = \frac{x_{k-1}f(x_k) - x_k f(x_{k-1})}{f(x_k) - f(x_{k-1})}$$

   (b) Jeśli zrealizujemy obliczenia w arytmetyce zmiennoprzecinkowej o skończonej precyzji, jakie zalety i wady ma wzór podany w podpunkcie (a), w porównaniu ze wzorem dla metody siecznych podanym poniżej?

$$x_{k+1} = x_k - f(x_k) \cdot \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

3. Zapisz iterację Newtona do rozwiązywania następującego układu równań nieliniowych:

$$\begin{cases} x_1^2 + x_1 x_2^3 = 9 \\ 3x_1^2 x_2 - x_2^2 = 4 \end{cases}$$

3 Metodyka

Skorzystano z następujących metod:

- Metoda Newtona - Metoda Newtona, inaczej metoda stycznych, jest iteracyjnym algorytmem wyznaczania przybliżonej wartości miejsca zerowego funkcji f, wykorzystującym lokalną liniową aproksymację za pomocą stycznej [3]. Kolejne przybliżenia oblicza się ze wzoru
$$x_{k+1} \;=\; x_k \;-\; \frac{f(x_k)}{f'(x_k)}\,,$$
który otrzymuje się, rozwiązując układ stycznej do wykresu f w punkcie $(x_k, f(x_k))$ z osią odciętych.

- Metoda Netwona dla funkcji wielu zmiennych - Jest to uogólniona metoda Newtona dla rozwiązywania układów równań. Przyjmujemy: Dla układu $F : \mathbb{R}^n \to \mathbb{R}^n$ zapis iteracji przyjmuje postać:
$$x_{k+1} \;=\; x_k \;-\; J_F(x_k)^{-1}\, F(x_k)\,,$$
gdzie J_F to macierz Jacobiego funkcji wektorowej F [3]

4 Rozwiązania

Zadanie 1: Obliczanie miejsc zerowych metodą Newtona dla równań nieliniowych

Do rozwiązania skorzystamy z metody Newtona opisanej powyżej [3].

Założenia Aby metoda zbiegała, powinny być spełnione założenia:

1. f jest ciągła i ma ciągłą pochodną w otoczeniu pierwiastka,
2. istnieje dokładnie jedno x^* takie, że $f(x^*) = 0$,
3. $f'(x^*) \neq 0$,
4. punkt startowy x_0 jest dostatecznie blisko x^*.

Zbieżność Metoda Newtona ma zbieżność kwadratową (rząd 2), co oznacza, że dla wystarczająco bliskiego startu liczba poprawnych cyfr szacunku w przybliżeniu podwaja się każdą iterację [1].

4.1 Rozwiązania

4.1.1 Równanie (a): $x\cos(x) = 1$

Funkcja ta będzie miała nieskończenie wiele rozwiązań. Niemniej jednak, jesteśmy w stanie wyznaczyć miejsca zerowe przy zadanych punktach różnych od zera.

Dla równania $x\cos(x) = 1$ przekształcamy je do postaci: $f(x) = x\cos(x) - 1 = 0$

Pochodna funkcji to: $f'(x) = \cos(x) - x\sin(x)$

Wzór iteracyjny metody Newtona: $x_{n+1} = x_n - \frac{x_n\cos(x_n)-1}{\cos(x_n)-x_n\sin(x_n)}$

4.1.2 Równanie (b): $x^3 - 5x - 6 = 0$

Dla równania wielomianowego $x^3 - 5x - 6 = 0$, definiujemy: $f(x) = x^3 - 5x - 6$

Pochodna: $f'(x) = 3x^2 - 5$

Wzór Newtona: $x_{n+1} = x_n - \frac{x_n^3-5x_n-6}{3x_n^2-5}$

4.1.3 Równanie (c): $e^{-x} = x^2 - 1$

Równanie przekształcamy do postaci: $f(x) = e^{-x} - x^2 + 1 = 0$
Pochodna funkcji: $f'(x) = -e^{-x} - 2x$
Wzór iteracyjny: $x_{n+1} = x_n - \frac{e^{-x_n} - x_n^2 + 1}{-e^{-x_n} - 2x_n}$

4.1.4 Analiza zbieżności

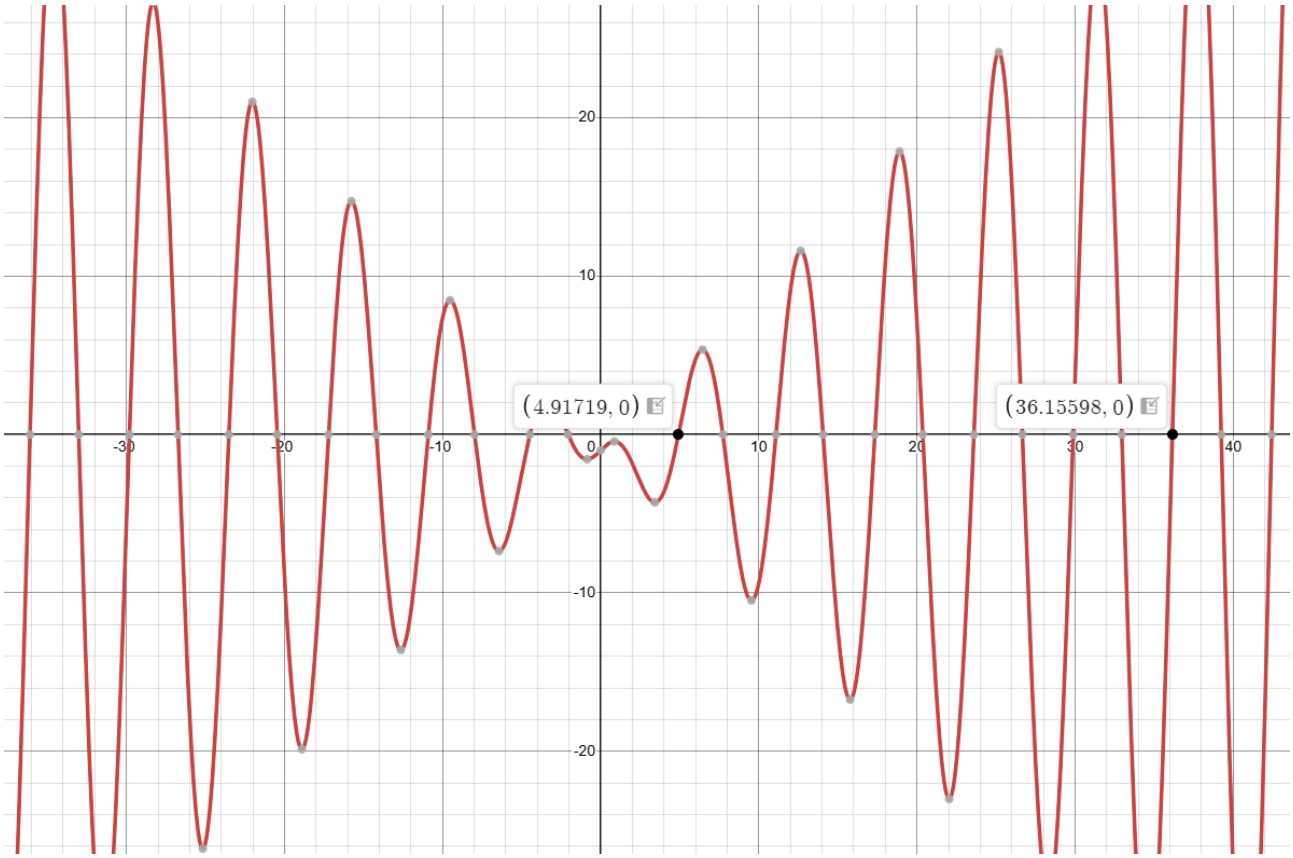
Metoda Newtona zbiega kwadratowo, gdy spełnione są następujące warunki:

1. Funkcja f jest ciągle różniczkowalna
2. Pierwiastek r jest prosty ($f'(r) \neq 0$)
3. Początkowe przybliżenie x_0 jest dostatecznie blisko r

Dla naszych równań zbieżność wygląda następująco:

4.1.5 Dla $x\cos(x) = 1$

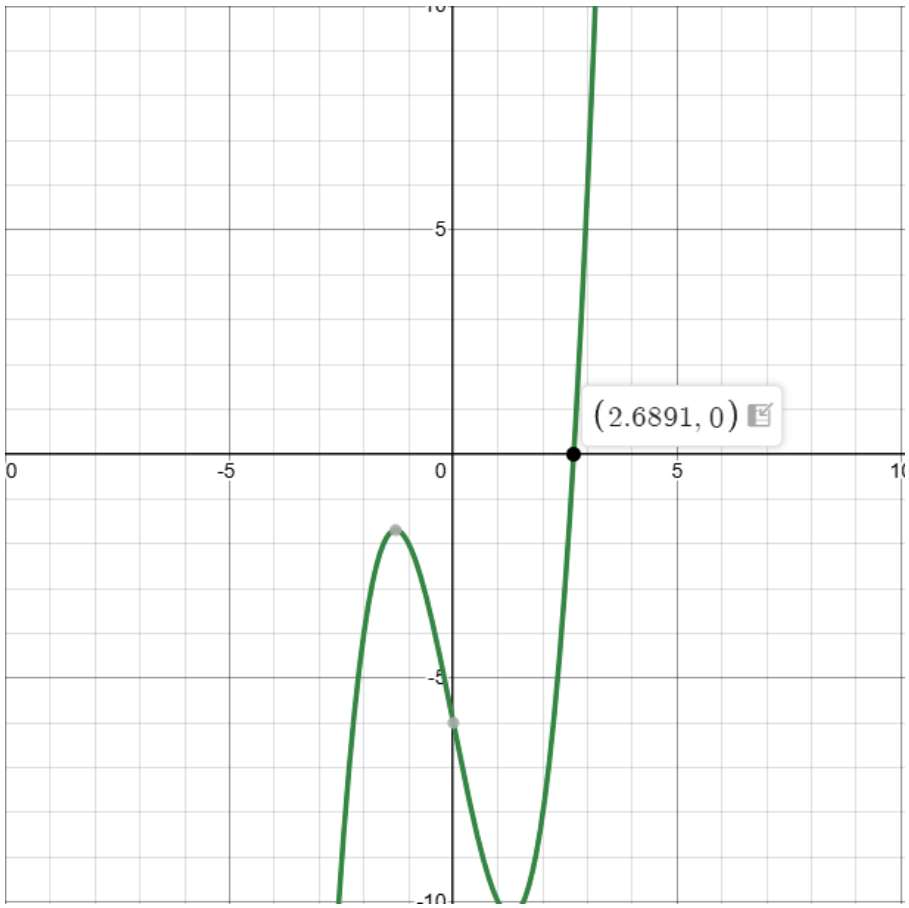
Kod znalazł pierwiastek w pobliżu $x \approx 4.9171859253$ i $x \approx 36.1559769881$. Metoda Newtona zwykle szybko zbiega przy początkowym przybliżeniu z przedziału $[0, 1]$. Jednakże pochodna może zanikać dla pewnych wartości x i dużej dokładności (tj. bardzo małej wartości) ϵ, co jest spowodowane dużą liczbą iteracji. Miejsce zerowe $x \approx 36.1559769881$ zostało znalezione dla wartości $x_0 = 1.0$ dla precyzji $\epsilon = 10^{-14}$, co sugeruje, że właśnie taki problem wystąpił.



Rysunek 1: Dwa znalezione miejsca zerowe dla równania $x \cdot \cos x = 1$. Widać, że istnieje ich nieskończenie wiele, jednak w tym przypadku algorytm Netwona schodzi do pewnego wybranego punktu.

4.1.6 Dla $x^3 - 5x - 6 = 0$

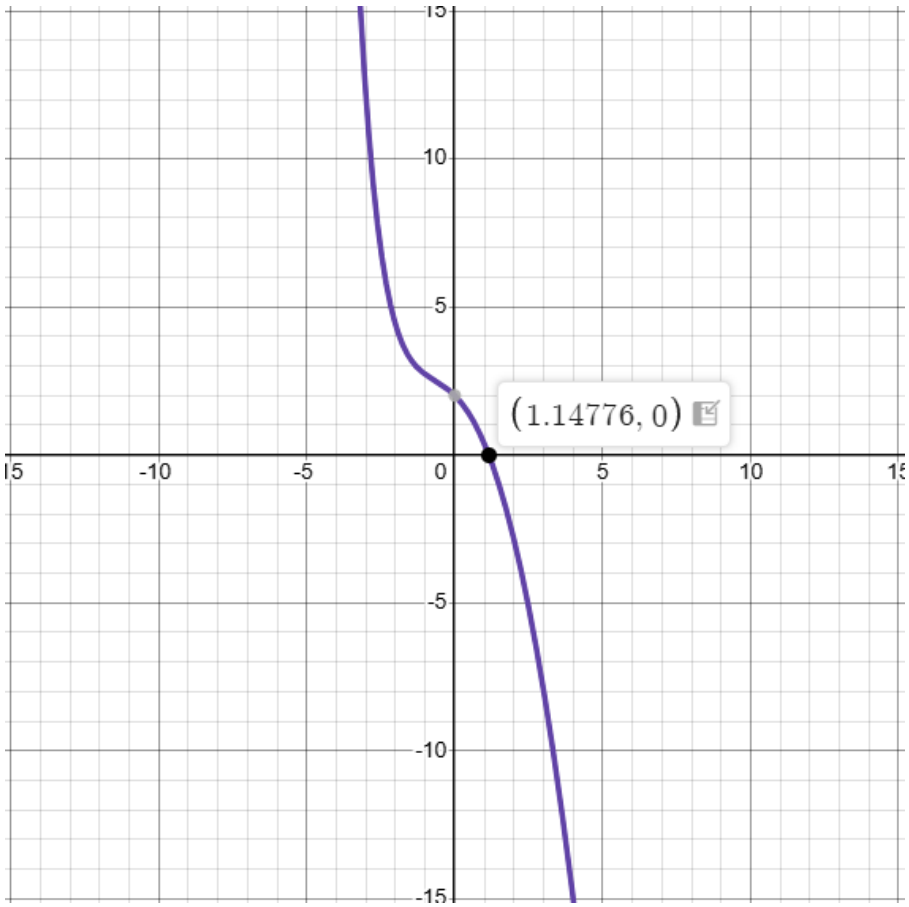
Wielomian ten ma pierwiastki rzeczywiste w $x \approx 2.6890953236$ i zespolone w $x \approx \pm\sqrt{\frac{5}{3}}i$. Metoda Newtona działa bardzo dobrze dla wielomianów i szybko zbiega do $x \approx 3$ przy rozsądnym starcie, np. $x_0 = 2$.



Rysunek 2: Znalezione miejsce zerowe na wykresie

4.1.7 Dla $e^{-x} = x^2 - 1$

Równanie to łączy składniki wykładnicze i wielomianowe. Ma pierwiastki w pobliżu $x \approx 1.1477576321$. Wybór odpowiedniego przybliżenia początkowego jest kluczowy dla zbieżności do żądanego pierwiastka.



Rysunek 3: Znalezione miejsce zerowe na wykresie

4.1.8 Uwagi dotyczące implementacji praktycznej

W implementacji w języku Python uwzględniono następujące zabezpieczenia:

- Ochrona przed dzieleniem przez pochodne bliskie zeru
- Ograniczenie liczby iteracji, aby uniknąć nieskończonej pętli
- Parametr tolerancji (ϵ) kontrolujący precyzję zbieżności
- Wiele punktów startowych, aby zwiększyć szansę znalezienia wszystkich pierwiastków

4.2 Kod w Python-ie iustrujący obliczenia

```
from numpy import cos, sin, exp
import matplotlib.pyplot as plt
import numpy as np

def newton_approx(f=lambda x: x, f_der=lambda x: 1, x0=1.0, epsilon=1e-16, max_iter=100):
    x = x0
    iterations = 0
    while abs(f(x)) > epsilon and iterations < max_iter:
        if abs(f_der(x)) < 1e-10:
            raise ValueError("Derivative too close to zero - divergent")
        x = x - f(x)/f_der(x)
        iterations += 1
    return x, iterations

def f1(x):
    return x * cos(x) - 1

def f1_der(x):
    return cos(x) - x * sin(x)

def f2(x):
    return x**3 - 5*x - 6

def f2_der(x):
    return 3 * x**2 - 5

def f3(x):
    return exp(-x) - x**2 + 1

def f3_der(x):
    return -exp(-x) - 2*x

def compare_methods(f, f_der, x0_values, epsilon_values, title):
    results = []
    for x0 in x0_values:
        for eps in epsilon_values:
            try:
                root, iterations = newton_approx(f, f_der, x0, eps)
                results.append({
                    'x0': x0,
                    'epsilon': eps,
                    'root': root,
                    'iterations': iterations,
                    'final_error': abs(f(root))
                })
            except ValueError as e:
                print(f"Error with x0={x0}, epsilon={eps}: {e}")

    # Print results in a table format
    print(f"\nResults for {title}:")
    print(f"{'x0':^10} | {'epsilon':^12} | {'root':^15} | {'iterations':^10} | {'final error':^12}")
    print("-" * 65)
    for r in results:
        print(f"{r['x0']:^10} | {r['epsilon']:^12.1e} | {r['root']:^15.10f} | {r['iterations']:^10} | {r['final_error']:^12.2e}")
    return results
```

```
56
57 def get_different_roots(results, epsilon=1e-4):
58     roots = []
59     for element in results:
60         root = element['root']
61         if len(roots) == 0:
62             roots.append(root)
63         for other in roots:
64             if abs(root - other) > epsilon:
65                 roots.append(root)
66     return roots
```

4.2.1 Wnioski

Metoda Newtona jest skuteczną techniką rozwiązywania równań nieliniowych, szczególnie gdy dostępne są dobre przybliżenia początkowe. W sprzyjających warunkach metoda wykazuje zbieżność kwadratową, co czyni ją znacznie szybszą niż prostsze metody, np. metoda bisekcji.

1. Dla trzech analizowanych równań:
   (a) $x\cos(x) = 1$ ma rozwiązanie w $x \approx 4.9171859253$ i $x \approx 36.1559769881$
   (b) $x^3 - 5x - 6 = 0$ ma rozwiązanie przy $x \approx 2.6890953236$, ze na to, że jest to równanie sześcienne, można je rozwiązać analitycznie [2]
   (c) $e^{-x} = x^2 - 1$ ma rozwiązania przy

Wszystkie te rozwiązania mogą być efektywnie znalezione metodą Newtona przy odpowiednich przybliżeniach początkowych.

Zadanie 2: Dowód i analiza uogólnionego wzoru Newtona

(a) Równoważność z metodą siecznych

Metoda siecznych dla równania $f(x) = 0$ jest dana iteracją

$$x_{k+1} \;=\; x_k - f(x_k)\,\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}.$$

Przekształćmy tę formułę:

$$x_{k+1} = \frac{x_k\big(f(x_k) - f(x_{k-1})\big) - f(x_k)\,(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})} \Rightarrow$$

$$\frac{x_k\big(f(x_k) - f(x_{k-1})\big) - f(x_k)\,(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})} = \frac{x_k f(x_k) - x_k f(x_{k-1}) - x_k f(x_k) + x_{k-1} f(x_k)}{f(x_k) - f(x_{k-1})} \Rightarrow$$

$$\frac{x_k f(x_k) - x_k f(x_{k-1}) - x_k f(x_k) + x_{k-1} f(x_k)}{f(x_k) - f(x_{k-1})} = \frac{x_{k-1} f(x_k) - x_k f(x_{k-1})}{f(x_k) - f(x_{k-1})},$$

co dokładnie daje formułę podaną w zadaniu.

(b) Numeryczna analiza wzoru

W arytmetyce zmiennoprzecinkowej o skończonej precyzji:

- **Zalety formy**
  - Unika mnożenia małych różnic typu $(x_k - x_{k-1})$, przenosząc je pod jeden ułamek.
  - Może zmniejszać lokalne wzmocnienie błędu, gdy $f(x_k) \approx f(x_{k-1})$.
- **Wady formy**
  - Więcej operacji mnożenia i odejmowania – większe ryzyko akumulacji zaokrągleń.
  - Gdy $f(x_k) - f(x_{k-1})$ jest bardzo małe, nadal dzielimy przez małą liczbę, co może prowadzić do znacznych błędów.
- **W praktyce** zwykle stosuje się pierwotny zapis $x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$, który jest nieco prostszy i często równie stabilny.

Zadanie 3: Rozwiązanie nieliniowego układu równań uogólnioną metodą Newtona

Dana jest funkcja wektorowa

$$\mathbf{F}(x_1, x_2) = \begin{pmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{pmatrix} = \begin{pmatrix} x_1^2 + x_1 x_2^3 - 9 \\ 3x_1^2 x_2 - x_2^3 - 4 \end{pmatrix}.$$

Macierz Jacobiego $J = \left(\partial f_i / \partial x_j\right)$ ma postać

$$J(x_1, x_2) = \begin{pmatrix} 2x_1 + x_2^3 & 3x_1\, x_2^2 \\ 6x_1\, x_2 & 3x_1^2 - 3x_2^2 \end{pmatrix}.$$

Iteracja Newtona wyraża się wzorem

$$\begin{aligned} J\big(x^{(k)}\big)\, \Delta x^{(k)} &= -\,\mathbf{F}\big(x^{(k)}\big), \\ x^{(k+1)} &= x^{(k)} + \Delta x^{(k)}, \end{aligned}^{x}$$

czyli jawnie

$$x^{(k+1)} = x^{(k)} - J\big(x^{(k)}\big)^{-1}\, \mathbf{F}\big(x^{(k)}\big).$$

Kroki obliczeń

Korki obliczeń wykonane przez komputer prezentują się następująco:

1. Punkt wyjściowy: $x_0 = 1.500000$, $y_0 = 1.000000$
2. Iteracja 1:
   - Punkt: $(x_0, y_0) = (1.500000, 1.000000)$
   - Wartości funkcji: $F(x, y) = [-5.250000, 1.750000]^T$
   - $||F(x, y)|| = 5.533986e + 00$

- Macierz Jakobiego: $J = \begin{bmatrix} 4.000000 & 4.500000 \\ 9.000000 & 3.750000 \end{bmatrix}$
- $\Delta = [-1.080882, 2.127451]^T$
- Następny punkt: $(x_1, y_1) = (0.419118, 3.127451)$

3. Iteracja 2:
   - Punkt: $(x_1, y_1) = (0.419118, 3.127451)$
   - Wartości funkcji: $F(x, y) = [3.996234, -32.941340]^T$
   - $||F(x, y)|| = 3.318285e + 01$
   - Macierz Jakobiego: $J = \begin{bmatrix} 31.427676 & 12.298106 \\ 7.864619 & -28.815870 \end{bmatrix}$
   - $\Delta = [0.289285, -1.064213]^T$
   - Następny punkt: $(x_2, y_2) = (0.708403, 2.063238)$
4. Iteracja 3:
   - Punkt: $(x_2, y_2) = (0.708403, 2.063238)$
   - Wartości funkcji: $F(x, y) = [-2.276187, -9.676890]^T$
   - $||F(x, y)|| = 9.940987e + 00$
   - Macierz Jakobiego: $J = \begin{bmatrix} 10.199910 & 9.046912 \\ 8.769625 & -11.265350 \end{bmatrix}$
   - $\Delta = [0.582712, -0.405378]^T$
   - Następny punkt: $(x_3, y_3) = (1.291115, 1.657860)$
5. Iteracja 4:
   - Punkt: $(x_3, y_3) = (1.291115, 1.657860)$
   - Wartości funkcji: $F(x, y) = [-1.449893, -0.265782]^T$
   - $||F(x, y)|| = 1.474053e + 00$
   - Macierz Jakobiego: $J = \begin{bmatrix} 7.138857 & 10.645886 \\ 12.842925 & -3.244567 \end{bmatrix}$
   - $\Delta = [0.047119, 0.104596]^T$
   - Następny punkt: $(x_4, y_4) = (1.338234, 1.762456)$
6. Iteracja 5:
   - Punkt: $(x_4, y_4) = (1.338234, 1.762456)$
   - Wartości funkcji: $F(x, y) = [0.117206, -0.005638]^T$
   - $||F(x, y)|| = 1.173414e - 01$
   - Macierz Jakobiego: $J = \begin{bmatrix} 8.151097 & 12.470671 \\ 14.151471 & -3.946139 \end{bmatrix}$
   - $\Delta = [-0.001880, -0.008170]^T$
   - Następny punkt: $(x_5, y_5) = (1.336354, 1.754286)$
7. Iteracja 6:

- Punkt: $(x_5, y_5) = (1.336354, 1.754286)$
- Wartości funkcji: $F(x, y) = [0.000618, -0.000210]^T$
- $||F(x, y)|| = 6.524147e - 04$
- Macierz Jakobiego: $J = \begin{bmatrix} 8.071557 & 12.337968 \\ 14.066086 & -3.875028 \end{bmatrix}$
- $\Delta = [0.000001, -0.000051]^T$
- Następny punkt: $(x_6, y_6) = (1.336355, 1.754235)$

8. Iteracja 7:

- Punkt: $(x_6, y_6) = (1.336355, 1.754235)$
- Wartości funkcji: $F(x, y) = [0.000000, -0.000000]^T$
- $||F(x, y)|| = 2.245655e - 08$
- Macierz Jakobiego: $J = \begin{bmatrix} 8.071091 & 12.337264 \\ 14.065690 & -3.874486 \end{bmatrix}$
- $\Delta = [0.000000, -0.000000]^T$
- Następny punkt: $(x_7, y_7) = (1.336355, 1.754235)$

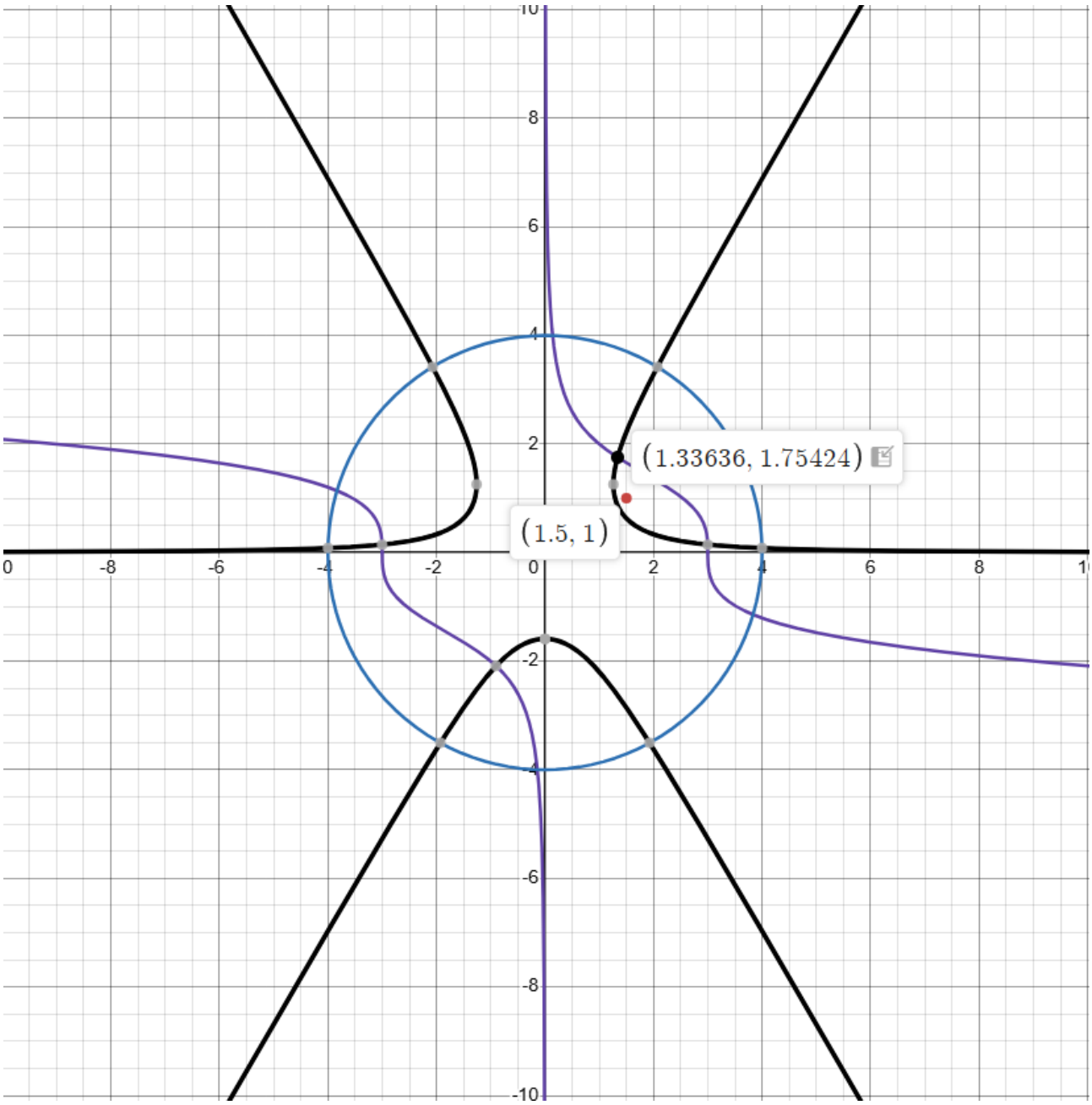
9. Iteracja 8:

- Punkt: $(x_7, y_7) = (1.336355, 1.754235)$
- Wartości funkcji: $F(x, y) = [0.000000, 0.000000]^T$
- $||F(x, y)|| = 3.552714e - 15$
- Macierz Jakobiego: $J = \begin{bmatrix} 8.071091 & 12.337264 \\ 14.065690 & -3.874486 \end{bmatrix}$
- $\Delta = [-0.000000, -0.000000]^T$
- Następny punkt: $(x_8, y_8) = (1.336355, 1.754235)$

10. Iteracja 9:

- Punkt: $(x_8, y_8) = (1.336355, 1.754235)$
- Wartości funkcji: $F(x, y) = [0.000000, 0.000000]^T$
- $||F(x, y)|| = 0.000000e + 00$
- Macierz Jakobiego: $J = \begin{bmatrix} 8.071091 & 12.337264 \\ 14.065690 & -3.874486 \end{bmatrix}$
- Warunek zbieżności: $||F(x, y)|| < 1e - 16$

Jak widać na rysunku 4, aby wygenerować wszystkie rozwiązania, należy wybrać parę punktów przykład leżących na okręgu oznaczonego na rysunku następnie skorzystać z algorytmu Metody Newtona i odfilrować powtarzające się (z zadaną precyzją) punkty.



Rysunek 4: Znalezione rozwiązanie z punktem początkowym.

Kod do programu generującego wszystkie kroki zamieszczam poniżej.

Kod w Pythonie ilustrujący obliczenia

```
    import numpy as np

def first_function(x, y):
    return x ** 2 + x * y ** 3 - 9

def first_function_der_x(x, y):
    return 2 * x + y ** 3

def first_function_der_y(x, y):
    return 3 * x * y ** 2

def second_function(x, y):
```

```
13      return 3 * x ** 2 * y - y ** 3 - 4
14
15  def second_function_der_x(x, y):
16      return 6 * x * y
17
18  def second_function_der_y(x, y):
19      return 3 * x ** 2 - 3 * y ** 2
20
21  def make_jacobian(x, y):
22      return np.array([
23          [first_function_der_x(x, y), first_function_der_y(x, y)],
24          [second_function_der_x(x, y), second_function_der_y(x, y)]
25      ])
26
27  def make_vector(x, y):
28      return np.array([first_function(x, y), second_function(x, y)])
29
30  def newton_method(x0, y0, tol=1e-16, max_iter=20):
31      x, y = x0, y0
32      print("\\begin{enumerate}")
33      print(f"\\item␣Punkt␣wyj ciowy:␣$x_0␣=␣{x0:.6f}$,␣$y_0␣=␣{y0:.6f}$")
34      for iteration in range(max_iter):
35          F = make_vector(x, y)
36          f_norm = np.linalg.norm(F)
37          J = make_jacobian(x, y)
38          print(f"\\item␣Iteracja␣{iteration␣+␣1}:")
39          print(f"␣␣\\begin{{itemize}}")
40          print(f"␣␣␣␣\\item␣Punkt:␣$(x_{iteration},␣y_{iteration})␣=␣({x:.6f
                }␣{y:.6f})$")
41          print(f"␣␣␣␣\\item␣Warto ci␣funkcji:␣$F(x,y)␣=␣[{F[0]:.6f},␣{F
                [1]:.6f}]^T$")
42          print(f"␣␣␣␣\\item␣$||F(x,y)||␣=␣{f_norm:.6e}$")
43          print(f"␣␣␣␣\\item␣Macierz␣Jakobiego:␣$J␣=␣\\begin{{bmatrix}}␣{J
                [0,0]:.6f}␣&␣{J[0,1]:.6f}␣\\\\␣{J[1,0]:.6f}␣&␣{J[1,1]:.6f}␣\\end
                {{bmatrix}}$")
44          if f_norm < tol:
45              print(f"␣␣␣␣\\item␣Warunek␣zbie no ci:␣$||F(x,y)||␣<␣{tol}$")
46              print(f"␣␣\\end{{itemize}}")
47              print("\\end{enumerate}")
48              return x, y, iteration, True
49          try:
50              delta = np.linalg.solve(J, -F)
51              print(f"␣␣␣␣\\item␣$\\Delta␣=␣[{delta[0]:.6f},␣{delta[1]:.6f}]^
                    T$")
52              x += delta[0]
53              y += delta[1]
54              print(f"␣␣␣␣\\item␣Nast pny␣punkt:␣$(x_{iteration+1},␣y_{
                    iteration+1})␣=␣({x:.6f},␣{y:.6f})$")
55              print(f"␣␣\\end{{itemize}}")
56          except np.linalg.LinAlgError:
57              print(f"␣␣␣␣\\item␣Macierz␣z␣zerowym␣wyznacznikiem␣-␣
                    rozbie no  ")
58              print(f"␣␣\\end{{itemize}}")
59              print("\\end{enumerate}")
60              return x, y, iteration, False
61      return x, y, max_iter, False
62
63  if __name__ == "__main__":
64      x0, y0 = 1.5, 1.0
65      x_sol, y_sol, iterations, converged = newton_method(x0, y0)
```

```
    if converged:
        print(f"\n% Summary")
        print(f"% Solution found after {iterations + 1} iterations:")
        print(f"% x = {x_sol:.10f}, y = {y_sol:.10f}")
        print(f"% Function values at solution:")
        print(f"% f1(x,y) = {first_function(x_sol, y_sol):.10e}")
        print(f"% f2(x,y) = {second_function(x_sol, y_sol):.10e}")
    else:
        print(f"\n% Failed to converge after {iterations + 1} iterations.")
        print(f"% Last values: x = {x_sol:.10f}, y = {y_sol:.10f}")
```

Wnioski

1. Metoda Newtona dla badanego układu nieliniowego zbiega bardzo szybko — w naszym przykładzie dostateczną precyzję osiągnięto już po 9 iteracjach, mimo początkowo dalekiego od rozwiązania punktu startowego.

2. Tempo zbieżności jest kwadratowe pod warunkiem, że macierz Jacobiego w pobliżu rozwiązania pozostaje nieosobliwa. Widzimy to wyraźnie w gwałtownym spadku normy wektora funkcji $\|\mathbf{F}(x^{(k)})\|$ od kroku 4 w dół.

3. Poprawny dobór punktu początkowego ma kluczowe znaczenie — przy innych punktach startowych możliwe byłyby zbieżność do innego rozwiązania lub brak zbieżności. Dlatego, aby uzyskać wszystkie rozwiązania układu, należy spróbować kilku różnych punktów startowych na wybranym obszarze (np. na okręgu pokazanym na rysunku 4).

4. Obliczanie i odwracanie macierzy Jacobiego w każdym kroku generuje istotny koszt obliczeniowy, co przy dużych układach mogłoby stać się wąskim gardłem. W praktyce można rozważyć modyfikacje metody Newtona (np. metody quasi-Newtona) lub przyspieszenie rozwiązywania układów liniowych.

5. Metoda okazała się stabilna wobec małych zakłóceń — po osiągnięciu obszaru lokalnej zbieżności kolejne poprawki $\Delta x^{(k)}$ są coraz mniejsze, aż do osiągnięcia granicy precyzji maszynowej.

Literatura

[1] Eric W. Weisstein. Newton's method. `https://mathworld.wolfram.com/NewtonsMethod.html`, n.d. From MathWorld–A Wolfram Web Resource.

[2] Wikipedia. Cubic equation — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Cubic\%20equation&oldid=1285341972`, 2025. [Online; accessed 07-May-2025].

[3] Wikipedia. Metoda Newtona — Wikipedia, the free encyclopedia. `http://pl.wikipedia.org/w/index.php?title=Metoda\%20Newtona&oldid=74009086`, 2025. [Online; accessed 06-May-2025].