# Biblioteki i polecenia w Pythonie do rozwiązywania równań

## 1) Rozwiązywanie symboliczne

**Biblioteka:** `sympy`

**Najważniejsze funkcje / polecenia**

- `sympy.symbols('x y')` — zadeklarowanie symboli.
- `sympy.solve(expr, x)` — ogólne rozwiązywanie (zwraca listę rozwiązań; dla układów lista słowników).
- `sympy.solveset(expr, x, domain=...)` — bardziej formalne, zwraca zbiory rozwiązań (np. `S.Reals`).
- `sympy.factor`, `sympy.simplify`, `sympy.expand` — upraszczanie przed/po.
- `sympy.nsolve(expr, x0)` — numeryczne rozwiązanie (przydatne gdy `solve` nie radzi sobie).
- `sympy.Matrix(A).solve(b)` — rozwiązywanie układów liniowych symbolicznie (jeśli da się).
- `sympy.Poly(...).nroots()` — numeryczne pierwiastki wielomianu (wysoka precyzja możliwa).

**Krótkie przykłady**

```
from sympy import symbols, solve, solveset, Eq
x = symbols('x')
solve(Eq(x**3-4*x,0), x)     # [0, -2, 2]
solveset(x**2-2, x)          # {-sqrt(2), sqrt(2)}
```

**Na co uważać (symbolicznie)**

- `solve` może zwracać konstrukcje symboliczne (parametryzacje) lub puste zbiory; `solveset` jest bardziej formalne.
- Dla równań transcendentalnych (np. `exp`, `sin`) rozwiązania mogą być wielowartościowe (używać `solveset` i podawać domenę).
- SymPy może być powolny lub nieporadny dla dużych układów — wtedy warto przejść na metody numeryczne.
- Upewnij się co do założeń na symbole (np. `positive=True`, `real=True`) jeśli chcesz tylko rozwiązania rzeczywiste.

## 2) Rozwiązywanie numeryczne — układy liniowe

**Biblioteki:** `numpy`, `scipy`

**Najważniejsze funkcje**

- `numpy.linalg.solve(A, b)` — bezpośrednie rozwiązanie $Ax = b$ (gdy $A$ jest kwadratowa i nieosobliwa).
- `numpy.linalg.lstsq(A, b)` — rozwiązanie najmniejszych kwadratów (nadokreślone / niedokładne).
- `scipy.linalg.solve(A, b)` — podobne do numpy, ale więcej opcji (np. symetryczne / triangularne).
- `scipy.sparse.linalg.spsolve(A_sparse, b)` — dla rzadkich macierzy.

- `scipy.sparse.linalg.cg`, `bicgstab` — iteracyjne metody dla dużych, rzadkich układów (wymagają preconditioning).

**Przykład**

```
import numpy as np
A = np.array([[3,1],[1,2]], dtype=float)
b = np.array([9,8], dtype=float)
x = np.linalg.solve(A, b)
```

**Na co uważać (liniowe numeryczne)**

- **Nigdy** nie używaj `np.linalg.inv(A) @ b` do rozwiązywania — to mniej stabilne i wolniejsze.
- Sprawdź `np.linalg.cond(A)` — źle uwarunkowane macierze dają niestabilne wyniki.
- Dla rzadkich / dużych układów używaj solverów rzadkich i preconditioningu.
- Uważaj na typy (`float64` vs `float32`) i skalowanie równań.

## 3) Rozwiązywanie numeryczne — pojedyncze równanie nieliniowe (skalarne)

**Biblioteki:** `scipy.optimize`, `mpmath` (wysoka precyzja)

**Najważniejsze funkcje**

- `scipy.optimize.root_scalar` — metoda ogólna dla jednego równania; ma metody: `'brentq'`, `'bisect'`, `'newton'`, `'secant'` itp.
- `scipy.optimize.brentq(f, a, b)` / `bisect` — wymagają przedziału, w którym f zmienia znak (gwarantowana zbieżność).
- `scipy.optimize.newton(f, x0, fprime=None)` — metoda Newtona; wymaga dobrego początkowego przybliżenia lub pochodnej (lub używa różnic skończonych).
- `mpmath.findroot` — znajdowanie pierwiastków z możliwością wysokiej precyzji i metod hybrydowych.

**Przykład**

```
from scipy.optimize import root\_scalar
f = lambda x: x**3 - 2*x - 5
res = root\_scalar(f, bracket=[2,3], method='brentq')
```

**Na co uważać (skalarne numeryczne)**

- `brentq` i `bisect` wymagają zmiany znaku na krańcach przedziału — to bezpieczne.
- `newton` zły start może nie zbiec lub zbiec do niepożądanego pierwiastka; gdy używasz `fprime=None` numeryczne pochodne mogą być niestabilne.
- Kilkukrotne pierwiastki (multiplicity $> 1$) spowalniają / zatrzymują Newtona — stosuj metody odporne lub transformacje.
- Ustal tolerancje (`xtol`, `rtol`, `maxiter`) i sprawdzaj `res.converged` / `res.root`.

## 4) Rozwiązywanie numeryczne — układy równań nieliniowych

**Biblioteki:** `scipy.optimize`, `mpmath`, ewentualnie `jax` / `autograd` dla dużych problemów z automatycznym różniczkowaniem

**Najważniejsze funkcje**

- `scipy.optimize.root(fun, x0, jac=None, method=...)` — ogólny solver dla wektorowych równań; metody: `'hybr'` (domyślna, od MINPACK), `'lm'` (Levenberg–Marquardt), `'broyden1'`, `'krylov'`, `'anderson'` itd.
- `scipy.optimize.fsolve` — wrapper do `hybr` (prostsze API).

- `scipy.optimize.least_squares` — do układów sformułowanych jako minimalizacja sumy kwadratów (Levenberg–Marquardt, `trf`, `dogbox`), pozwala dostarczyć Jacobian i ograniczenia.
- `mpmath.findroot` również obsługuje wektorowe układy.

**Przykład (root / least_squares)**

```
from scipy.optimize import root, least\_squares
def fun(x):
return [x[0] + x[1] - 3,
x[0]**2 + x[1]**2 - 9]

res = root(fun, x0=[1,2], method='hybr')
res.x

# lub jako least\_squares (jeli chcesz najmniejsze kwadraty)

res2 = least_squares(lambda x: fun(x), x0=[1,2], jac='2-point')
res2.x
```

**Na co uważać (układy nieliniowe)**

- **Początkowe przybliżenie** (`x0`) jest często krytyczne — złe `x0` → brak zbieżności lub zbieżność do innego rozwiązania.
- Dostarczenie **analitycznego Jacobiana** (argument `jac=`) znacząco zwiększa szybkość i stabilność; jeśli nie masz — użyj `least_squares` z `jac='3-point'` lub narzędzi autograd.
- Problemy wielu rozwiązań: metoda może znaleźć **inne** rozwiązanie niż oczekujesz. Przeszukuj przestrzeń startów lub stosuj globalne metody.
- Skalowanie równań: jeśli różne komponenty mają bardzo różne skale, algorytm może źle działać — przeskaluj układ lub użyj wag.
- Sprawdź `res.success` / `res.status` i nie polegaj tylko na wartości `x` — sprawdź residuum $|f(x)|$.
- Dla dużych rzadkich układów: użyj metod opartych na Jacobianie rzadkim i solverów rzadkich (np. `scipy.sparse` + `spsolve` lub iteracyjne metody Newtona z preconditioningiem).

## 5) Inne przydatne narzędzia / biblioteki

- `mpmath` — arytmetyka wielokrotnej precyzji, `mpmath.findroot` i `mpmath.polyroots`. Przydatne gdy potrzebna precyzja >64 bitów.
- `numdifftools` — dokładniejsze numeryczne różniczkowanie (jeśli nie ma jacobianu).
- `autograd`, `jax` — automatyczne różniczkowanie (świetne do dużych układów i gdy chcesz wyliczać dokładny jacobian automatycznie).
- `sympy` + `lambdify` — możesz przekształcić symboliczną funkcję do funkcji numerycznej (`lambdify`) i użyć jej w `scipy.optimize`.

## 6) Co wybrać w zależności od zadania

- **Symboliczne (dokładne)**: `sympy.solve`, `solveset` (gdy chcesz wzory analityczne).
- **Numeryczne — małe / gęste układy liniowe**: `numpy.linalg.solve`.
- **Numeryczne — duże / rzadkie układy liniowe**: `scipy.sparse.linalg.spsolve` lub metody iteracyjne (`cg`, `bicgstab`) + preconditioner.
- **Równanie nieliniowe (skalarne)**: `root_scalar` z `brentq` / `bisect` jeśli masz przedział; `newton` jeśli masz dobry start i/lub pochodną.
- **Układy nieliniowe**: `scipy.optimize.root` lub `least_squares` (jeśli to najmniejsze kwadraty) — dostarcz jacobian jeśli możesz.

- **Wysoka precyzja / wielokrotna precyzja**: `mpmath` lub `sympy.nsolve` z ustaleniem precyzji.

## 7) Praktyczne uwagii

1. **Początkowe przybliżenie**: krytyczne dla metod iteracyjnych — testuj różne starty.
2. **Bracketing**: jeśli możesz znaleźć przedział z różnicą znaku, używaj metod pewnych (brentq / bisect).
3. **Jacobian / pochodne**: analityczny jacobian znacząco poprawia zbieżność; jeśli brak — użyj wiarygodnego różnicowania numerycznego.
4. **Skalowanie**: normalizuj / skaluj zmienne i równania, żeby nie mieć skrajnych różnic skali.
5. **Sprawdzaj residuum i flagi konwergencji**: zawsze czytaj `res.success` i $|f(x)|$.
6. **Warunek macierzy** (dla liniowych): sprawdź `cond(A)`; przy złym warunku rozważ regularizację (Tikhonov), `lstsq` lub preconditioner.
7. **Symboliczne  zawsze możliwe**: SymPy może nie znaleźć zamkniętej formy lub być wolny; wtedy zmień na podejście numeryczne.
8. **Wiele rozwiązań**: równań nieliniowych często mają wiele rozwiązań — korzystaj z wielu startów, szkiców (plotów) i analizy.
9. **Precyzja numeryczna**: domyślnie `float64` — jeśli potrzebujesz więcej, użyj `mpmath` lub `sympy` z wysoką precyzją.
10. **Nie liczyć odwrotności macierzy** do rozwiązywania $Ax = b$ — użyj solverów (np. `linalg.solve`).