Non-linear equations

Towards the multi-dimensional case

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6BER03), 2024-2025

Today's outline

- Python solvers
- Newton-Raphson method
- Multi-dimensional Newton-Raphson

Non-linear Equation Solving in Python (1 var)

Single Variable Non-linear Zero Finding:

- Use the `root_scalar` function from `scipy.optimize` for finding zeros of a single-variable non-linear function.
- Be aware of the initial bracketing steps in `root_scalar`.

```python
from scipy.optimize import root_scalar

root_scalar(lambda x: -3*x**2 - 5*x + 2, method='brentq', bracket=[1, 4], xtol=1e-15)
```

```
     converged: True
          flag: converged
function_calls: 10
    iterations: 9
          root: 0.3333333333333333
```

Non-linear equation solver in Python ($\geq$ 2 var)

Solving Systems of Non-linear Equations (Multiple Variables):

- Use `fsolve` from `scipy.optimize` for systems involving multiple variables.
- Suitable for non-linear equations with two or more variables.

```python
from scipy.optimize import fsolve

def equations(x):
    return [2*x[0]*x[1] - x[1] + 2, 2*x[1] - 4*x[0] - 4]

fsolve(equations, [1, 1], xtol=1e-15)
```

Newton-Raphson Method

Algorithm:

- Requires evaluating both the function $f(x)$ and its derivative $f'(x)$ at arbitrary points.
- Extend the tangent line at the current point x_i until it intersects with zero.
- Set the next guess x_{i+1} as the abscissa of that zero crossing.
- For small enough δx and well-behaved functions, non-linear terms in the Taylor series become unimportant.

$$f(x) \approx f(x_i) + f'(x_i)\delta x + \mathcal{O}(\delta x^2) + \dots$$

$$0 \approx f(x_i) + f'(x_i)\delta x$$

$$\delta x \approx -\frac{f(x_i)}{f'(x_i)}$$

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}}$$

- Can be extended to higher dimensions.
- Requires an initial guess close enough to the root to avoid failure.

Newton-Raphson Method

Example with the Formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

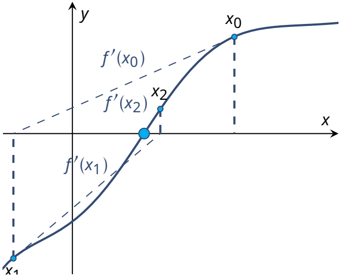
When it works:

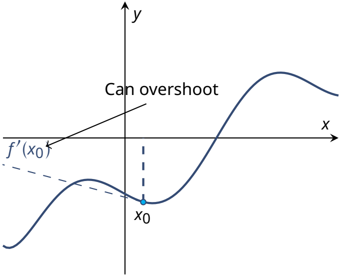
- Converges enormously fast when it functions correctly.

When it does not work:

- Underrelaxation can sometimes be helpful.
- Underrelaxation formula:

$$x_{n+1} = (1-\lambda)x_n + \lambda x_{n+1}$$
$$\lambda \in [0,1]$$





Newton-Raphson Method

Basic Algorithm:

Given initial x and a required tolerance $\varepsilon > 0$,

1. Compute $f(x)$ and $f'(x)$.
2. If $|f(x)| \leq \varepsilon$, return x.
3. Update x using the formula:

$$x \leftarrow x - \frac{f(x)}{f'(x)}$$

Repeat the above steps until a solution is found within the tolerance or the maximum number of iterations is exceeded.

Newton-Raphson Method

Exercise 5: Newton-Raphson Method in Excel

iteration	x	f	f'
0	-2	14	-8
1	-0.25	3.0625	-4.5
2	0.430556	0.463156	-3.13889
3	0.57811	0.021772	-2.84378
4	0.585766	5.86E-05	-2.82847
5	0.585786	4.29E-10	-2.82843
6	0.585786	0	-2.82843

Used formulas:

$$f(x) = x^2 - 4x + 2$$

$$f' = 2x - 4$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Newton-Raphson Method

Why is the Newton-Raphson so powerful?

- High rate of convergence
- Can achieve quadratic convergence!

Derivation of quadratic convergence:

1. Subtract solution
2. Define error
3. Express in terms of error
4. Use taylor expansion around solution
5. Rewrite in terms of error
6. Ignore higher order terms

$$x_{n+1} - x^* = x_n - x^* - f(x_n)/f'(x_n)$$

$$\varepsilon_n = x_n - x^*$$

$$\varepsilon_{n+1} = \varepsilon_n - f(x_n)/f'(x_n)$$

$$\varepsilon_{n+1} \approx \varepsilon_n - \frac{f(x^*) + f'(x^*)\varepsilon_n + f''(x^*)\varepsilon_n^2}{f'(x^*) + \mathcal{O}(\varepsilon_n^2)}$$

$$\varepsilon_{n+1} \approx -\frac{f''(x^*)\varepsilon_n^2 + \mathcal{O}(\varepsilon_n^3)}{f'(x^*) + \mathcal{O}(\varepsilon_n^2)}$$

$$\boxed{\varepsilon_{n+1} \approx -K\varepsilon_n^2}$$

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Newton-Raphson Method

Deriving the order of convergence

- The main issue with determining the order of convergence is that the solution is not known a priori
- To get around this issue it is possible to rewrite the problem in terms of known quantities.
- In the coming derivation, the following steps are taken to derive the order of convergence:
  1. The formal definition of K is given in terms of ε and the order of convergence m
  2. This formal definition is used to rewrite the fraction of successive errors
  3. Logarithms are used to isolate m
- Since the ε can't be computed without knowing the solution, the following approximation is made before plugging the final result:

$$\varepsilon_{n+1} \approx |x_{n+1} - x_n|$$

Newton-Raphson Method

1. Formal definition of K and m:

$$\lim_{n\to\infty} |\varepsilon_{n+1}| = K|\varepsilon_n|^m$$

2. Fraction of successive errors:

$$\frac{|\varepsilon_{n+1}|}{|\varepsilon_n|} = \frac{K|\varepsilon_n|^m}{K|\varepsilon_{n-1}|^m} \Rightarrow \left|\frac{\varepsilon_n}{\varepsilon_{n-1}}\right|^m$$

3. Extracting m:

$$\ln\left|\frac{\varepsilon_{n+1}}{\varepsilon_n}\right| = m\ln\left|\frac{\varepsilon_n}{\varepsilon_{n-1}}\right| \Rightarrow \boxed{m = \frac{\ln\left|\frac{\varepsilon_{n+1}}{\varepsilon_n}\right|}{\ln\left|\frac{\varepsilon_n}{\varepsilon_{n-1}}\right|}}$$

Newton-Raphson Method

Exercise 5: Newton-Raphson Method in Excel

- In this exercise, you will be working with the Newton-Raphson method implemented in Excel.
- The order of convergence (m) can be estimated using the relation:

$$m = \frac{\ln\left(\frac{\varepsilon_{n+1}}{\varepsilon_n}\right)}{\ln\left(\frac{\varepsilon_n}{\varepsilon_{n-1}}\right)}$$

  Where it is assumed that ε can be approximated by:

$$\varepsilon_{n+1} = |x_{n+1} - x_n|$$

- Solve a problem using the Newton-Raphson method in Excel and verify the order of convergence using the formulas above.

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Newton-Raphson Method

Exercise 5: Newton-Raphson Method in Excel solution

iteration	x	f	f'	eps	m
0	-2.000	14.000	-8.000	1.750	
1	-0.250	3.063	-4.500	0.681	1.619
2	0.431	0.463	-3.139	0.148	1.935
3	0.578	0.022	-2.844	0.008	1.998
4	0.586	0.000	-2.828	0.000	2.000
5	0.586	0.000	-2.828	0.000	
6	0.586	0.000	-2.828		

Used formulas:

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

$$m = \frac{\ln\left(\frac{\varepsilon_{n+1}}{\varepsilon_n}\right)}{\ln\left(\frac{\varepsilon_n}{\varepsilon_{n-1}}\right)}$$

$$\varepsilon_{n+1} = |x_{n+1} - x_n|$$

Newton-Raphson Method

Exercise 6: Newton-Raphson Method in Python

- Write a Python function to find the root of a function using the Newton-Raphson method.
- Assume that an initial guess x_0 is provided.
- The required tolerance for the solution should also be provided.
- Output the results of each iteration.
- Compute the order of convergence.

Newton-Raphson Method

Exercise 6: Newton-Raphson in Python solution

```python
def newton1D(f, df, x0, tol, max_iter):
  x = x0
  e = [0] * max_iter
  p = float('nan')
  for i in range(max_iter):
      x_new = x - f(x) / df(x)
      e[i] = abs(x_new - x)
      if i >= 2:
         p = (log(e[i]) - log(e[i - 1])) / (log(e[i - 1]) - log(e[i - 2]))
      print(f'x: {x_new:.10f}, e: {e[i]:.10f}, p: {p:.10f}')
      if e[i] < tol:
         break
      x = x_new
  return x
```

- Running the following command in Python yielded convergence in 6 iterations:

```python
newton1D(lambda x: x**2 - 4*x + 2, lambda x: 2*x - 4, 1, 1e-12, 100)
```

- Question: Why does it not work with an initial guess of $x_0 = 2$?
- This exercise encourages you to think about the influence of the initial guess on the convergence of the Newton-Raphson method.

Newton-Raphson Method

Modifications to the Basic Algorithm

- If $f'(x)$ is not known or is difficult to compute/program, a local numerical approximation can be used:

$$f'(x) \approx \frac{f(x+\delta x) - f(x)}{\delta x} \quad (\text{with } \delta x \sim 10^{-8})$$

- The chosen δx should be small but not too small to avoid round-off errors.
- The method should be combined with:
  - A bracketing method to prevent the solution from wandering outside of the bounds.
  - A reduced Newton step method for more robustness; don't take the full step if the error doesn't decrease sufficiently.
  - Sophisticated step size controls like local line searches and backtracking using cubic interpolation for global convergence.

Newton-Raphson Method in Python

Exercise 6: Numerical Differentiation

```python
from math import log
def newton1Dnum(f, h, x0, tol, max_iter):
  x = x0
  e = [0] * max_iter
  p = float('nan')
  for i in range(max_iter):
      x_new = x - f(x) / ((f(x + h) - f(x)) / h) # NUMERICAL DIFFERENTIATION
      e[i] = abs(x_new - x)
      if i >= 2:
         p = (log(e[i]) - log(e[i - 1])) / (log(e[i - 1]) - log(e[i - 2]))
      print(f'x: {x_new:.10f}, e: {e[i]:.10f}, p: {p:.10f}')
      if e[i] < tol:
         break
      x = x_new
  return x
```

- A command involving numerical differentiation in Python:

```python
newton1Dnum(lambda x: x**2 - 4*x + 2, 1e-7, 1, 1e-12, 100)
```

- This demonstrates that numerical differentiation can be utilized in the Newton-Raphson method to find the roots with the same efficiency in this specific case.

Newton-Raphson Method

How to Solve for Arbitrary Functions f: "Root Finding"

- **One-dimensional case**:
  - Move all terms to the left to have $f(x) = 0$.
  - Bracket or 'trap' a root between bracketing values, then hunt it down "like a rabbit."
- **Multi-dimensional case**:
  - Involving N equations in N unknowns.
  - It is not guaranteed to find a solution; it might not have a real solution or might have more than one solution.
  - Much more challenging compared to the one-dimensional case.
  - It is unpredictable to know if a root is nearby unless it has been found.

Newton-Raphson Method: Multi-dimensional Case (1)

- **Two-dimensional case:**

$$f(x,y) = 0,$$
$$g(x,y) = 0.$$

- **Multivariate Taylor series expansion:**

$$f(x+\delta x, y+\delta y) \approx f(x,y) + \frac{\partial f}{\partial x}\delta x + \frac{\partial f}{\partial y}\delta y + O(\delta x^2, \delta y^2) = 0$$

- **Neglecting higher order terms:**

$$g(x+\delta x, y+\delta y) \approx g(x,y) + \frac{\partial g}{\partial x}\delta x + \frac{\partial g}{\partial y}\delta y + O(\delta x^2, \delta y^2) = 0$$

- Leads to two linear equations in the unknowns δx and δy:

$$\frac{\partial f}{\partial x}\delta x + \frac{\partial f}{\partial y}\delta y = -f(x,y),$$
$$\frac{\partial g}{\partial x}\delta x + \frac{\partial g}{\partial y}\delta y = -g(x,y).$$

Newton-Raphson Method: Multi-dimensional Case (2)

In matrix notation:

$$\begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix} \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} = \begin{bmatrix} -f(x,y) \\ -g(x,y) \end{bmatrix}$$

Elements of this equation:

- Jacobian matrix:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix}$$

- The small displacement vector and **f**:

$$\delta\mathbf{x} = \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \qquad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f(x,y) \\ g(x,y) \end{bmatrix}$$

Solving equation by matrix inversion:

- Expressing the stepping equation in matrix notation:

$$\mathbf{J}(\mathbf{x}) \cdot \delta\mathbf{x} = -\mathbf{f}(\mathbf{x})$$

- Multiplying both sides by the inverse of **J**:

$$\delta\mathbf{x} = -\mathbf{J}^{-1}(\mathbf{x}) \cdot \mathbf{f}(\mathbf{x})$$

- Writing in terms of iteration number:

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}(\mathbf{x}_n) \cdot \mathbf{f}(\mathbf{x}_n)}$$

Newton-Raphson Method: Multi-dimensional Case (2)

In matrix notation:

$$\begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix} \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} = \begin{bmatrix} -f(x,y) \\ -g(x,y) \end{bmatrix}$$

Elements of this equation:

- Jacobian matrix:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix}$$

- The small displacement vector and **f**:

$$\delta\mathbf{x} = \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \qquad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f(x,y) \\ g(x,y) \end{bmatrix}$$

Solution via Cramer's rule:

- Determinant of the Jacobian $\mathtt{det}(\mathbf{J})$:

$$J = \mathtt{det}(\mathbf{J}) = \frac{\partial f}{\partial x}\frac{\partial g}{\partial y} - \frac{\partial f}{\partial y}\frac{\partial g}{\partial x}$$

- Solutions for δx and δy:

$$\delta x = \frac{-f(x,y)\frac{\partial g}{\partial y} + g(x,y)\frac{\partial f}{\partial y}}{J}$$

$$\delta y = \frac{f(x,y)\frac{\partial g}{\partial x} - g(x,y)\frac{\partial f}{\partial x}}{J}$$

EINDHOVEN UNIVERSITY OF TECHNOLOGY

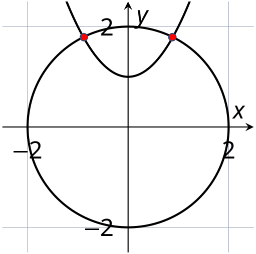
Newton-Raphson Method: multi-dimensional case

Example: *intersection of circle with parabola in matrix form*

$$\begin{aligned} x^2 + y^2 &= 4 \\ y &= x^2 + 1 \end{aligned} \quad \text{can be represented as} \quad \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ f(x) \end{bmatrix} = \begin{bmatrix} x - f(x) \\ x^2 + f(x^2) - 4 \end{bmatrix}$$

Iterations for solving:

i	$\mathbf{x}$	f	$\mathbf{J}$	$\delta\mathbf{x}$
1	$\begin{bmatrix} 1.00 \\ 2.00 \end{bmatrix}$	$\begin{bmatrix} 1.00 \\ 0.00 \end{bmatrix}$	$\begin{bmatrix} 2.00 & 4.00 \\ 2.00 & -1.00 \end{bmatrix}$	$\begin{bmatrix} -0.1 \\ -0.2 \end{bmatrix}$
2	$\begin{bmatrix} 0.90 \\ 1.80 \end{bmatrix}$	$\begin{bmatrix} 5.00 \\ 1.00 \end{bmatrix} \times 10^{-2}$	$\begin{bmatrix} 1.80 & 3.60 \\ 1.80 & -1.00 \end{bmatrix}$	$\begin{bmatrix} -0.01 \\ -8.7 \times 10^{-3} \end{bmatrix}$
3	$\begin{bmatrix} 0.89 \\ 1.79 \end{bmatrix}$	$\begin{bmatrix} 1.83 \\ 0.11 \end{bmatrix} \times 10^{-4}$	$\begin{bmatrix} 1.78 & 3.58 \\ 1.78 & -1.00 \end{bmatrix}$	$\begin{bmatrix} -6.99 \times 10^{-5} \\ -1.65 \times 10^{-5} \end{bmatrix}$
4	$\begin{bmatrix} 0.88 \\ 1.79 \end{bmatrix}$	$\begin{bmatrix} 5.16 \\ 4.89 \end{bmatrix} \times 10^{-9}$	$\begin{bmatrix} 1.78 & 3.58 \\ 1.78 & -1.00 \end{bmatrix}$	$\begin{bmatrix} -2.78 \times 10^{-9} \\ 5.94 \times 10^{-11} \end{bmatrix}$



Newton-Raphson Method: multi-dimensional case

Extensions to multi-dimensional case:
Check order of convergence:

it	x_1	x_2	eps1	eps2	m_1	m_2
1	1.0000	2.0000				
2	0.9000	1.8000	0.1000	0.2000		
3	0.8896	1.7913	0.0104	0.0087	1.9835	2.9482
4	0.8895	1.7913	0.0000699	0.0000165	2.0949	2.3208
5	0.8895	1.7913	0.0000000278	0.0000000059	2.0589	2.1382

Quadratic convergence

Doubling number of significant digits every iteration

$$m = \frac{\ln(\varepsilon_{n+1}) - \ln(\varepsilon_n)}{\ln(\varepsilon_n) - \ln(\varepsilon_{n-1})}$$

Newton-Raphson Method

Deriving the extension to more than two variables:

1. Generalization to the N-dimensional case
2. Define variables
3. Multi-variate Taylor series expansion
4. Define Jacobian matrix
5. Neglect higher-order terms
6. Express in terms of iterations

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}(\mathbf{x}_n)\cdot\mathbf{f}(\mathbf{x}_n)}$$

1. $f_i(x_1, x_2, \ldots, x_N) = 0$
2. $\mathbf{x} = [x_1, x_2, \ldots, x_N]$ $\mathbf{f} = [f_1, f_2, \ldots, f_N]$
3. $f_i(\mathbf{x} + \delta\mathbf{x}) = f_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial f_i}{\partial x_j}\delta x_j + O(\delta\mathbf{x}^2)$
4. $J_{ij} = \frac{\partial f_i}{\partial x_j}$ $f(\mathbf{x} + \delta\mathbf{x}) = f(\mathbf{x}) + \mathbf{J}\delta\mathbf{x} + O(\delta\mathbf{x}^2)$
5. $\mathbf{J}\cdot\delta\mathbf{x} = -\mathbf{f}(\mathbf{x})$

Newton-Raphson Method

Multi-variate Newton-Raphson in Python:

```python
def my_equations(X):
    F = np.zeros(2)
    F[0] = X[0]**2 + X[1]**2 - 4
    F[1] = X[0]**2 - X[1] + 1
    return F
```

```python
def my_jac(x):
    jac = np.zeros((2, 2))
    jac[0, 0] = 2 * x[0]
    jac[0, 1] = 2 * x[1]
    jac[1, 0] = 2 * x[0]
    jac[1, 1] = -1
    return jac
```

```python
import numpy as np
def newton_nd(f, J, x0, tol, max_iter):
    x = np.array(x0)
    err = np.zeros(max_iter)
    p = np.zeros(max_iter)
    for i in range(max_iter):
        delta_x = -np.linalg.solve(J(x), f(x))
        x += delta_x
        err[i] = np.linalg.norm(delta_x)
        if i > 0:
            p[i] = np.log(err[i]) / np.log(err[i-1])
        else:
            p[i] = float('nan')
        print(f'i = {i}: x = {x}, err = {err[i]:.6e}, p = {p[i]:.6f}')
        if err[i] < tol:
            break
    return x
```

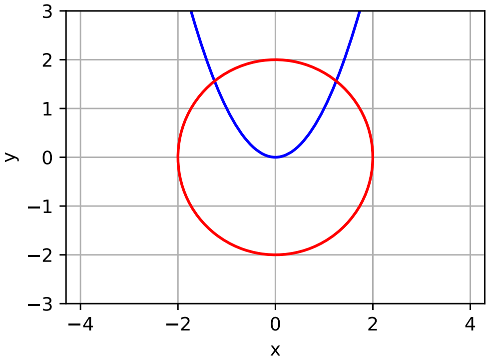
```python
newton_nd(my_equations, my_jac, [1, 2], 1e-12, 100)
```

Newton-Raphson Method

Multi-variate Newton-Raphson in Python:

Plotting the functions:

```
plot_implicit_function(lambda x,y: y-x**2, resolution=100, colors="blue")
plot_implicit_function(lambda x,y: y**2+x**2-4, resolution=100, colors="red")
```



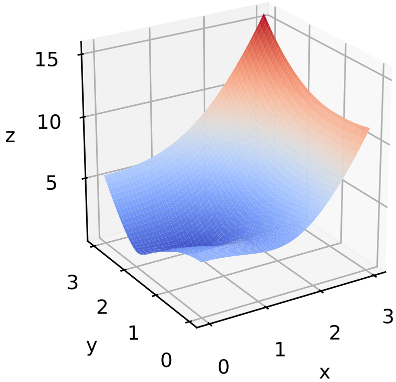
- Code can be found in `plot_implicit.py`
- Uses contour plot at $f(x,y) = 0$

Newton-Raphson Method

Multi-variate Newton-Raphson in Python:

Plotting the norm of the function:

```
plot_surface_function(lambda x,y: np.sqrt((x**2 + y**2 -4)**2+(x**2-y+1)**2),
                    (0,3),(0,3))
```

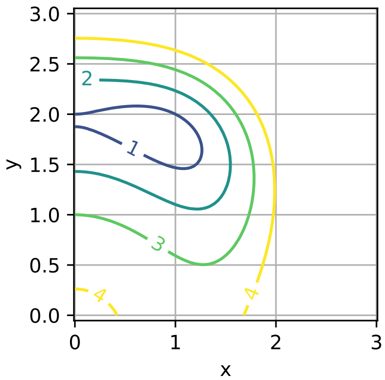


- Code can be found in `plot_implicit.py`
- Uses contour plot at $f(x,y) = 0$

Newton-Raphson Method

Multi-variate Newton-Raphson in Python:
Plotting the norm of the function:

```
plot_contours(lambda x,y: np.sqrt((x**2 + y**2 -4)**2+(x**2-y+1)**2),
              (0, 3), (0, 3), resolution = 100, levels=[0, 1, 2, 3, 4])
```



- Code can be found in `plot_implicit.py`
- Uses contour plot at $f(x,y) = 0$

Broyden's Method

Multi-dimensional secant method ('quasi-Newton'):

- Disadvantage of the Newton-Raphson method:
  - It requires the Jacobian matrix.
  - In many problems, no analytical Jacobian is available.
  - If the function evaluation is expensive, the numerical approximation using finite differences can be prohibitive.
- Solution: Use a cheap approximation of the Jacobian! (Secant or 'quasi-Newton' method)
- Comparison:

$$\text{Newton-Raphson:} \quad J_{ij}(\mathbf{x}) = \frac{\partial f_i}{\partial x_j}(\mathbf{x}) \quad \text{(Analytical)}$$

$$\text{Secant method:} \quad \mathbf{J}(\mathbf{x}) \quad \text{approximated by} \quad \mathbf{B} \quad \text{(Numerical)}$$

Broyden's Method

Approximating $\mathbf{B}^{n+1}$:

- Multi-dimensional secant method ('quasi-Newton'):
- Secant equation (generalization of 1D case):

$$\mathbf{B}^{n+1} \cdot \delta \mathbf{x}^n = \delta \mathbf{f}^n \quad \delta \mathbf{x}^n = \mathbf{x}^{n+1} - \mathbf{x}^n \quad \delta \mathbf{f}^n = \mathbf{f}^{n+1} - \mathbf{f}^n$$

- Underdetermined (not unique: -n equations with n unknowns), need another condition to pin down B^{n+1}.

Broyden's method:

- Determine $\mathbf{B}^{n+1}$ by making the least change to $\mathbf{B}^n$ that is consistent with the secant condition.
- Updating formula:

$$\mathbf{B}^{n+1} = \mathbf{B}^n + \frac{(\delta \mathbf{f}^n - \mathbf{B}^n \cdot \delta \mathbf{x}^n)}{\delta \mathbf{x}^n \cdot \delta \mathbf{x}^n} \otimes \delta \mathbf{x}^n$$

- Note: Sometimes δB_{n-1} is updated directly.

Broyden's Method

Background of Broyden's method:

- Secant equation:

$$\mathbf{B}^{n+1} \cdot \delta\mathbf{x}^n = \delta f_n$$

- Since there is no update on derivative info, why would $\mathbf{B}^n$ change in a direction orthogonal to $\delta\mathbf{x}^n$?

$$\Rightarrow (\delta\mathbf{x}^n)^T \delta\mathbf{w} = 0$$

$$\begin{aligned} \mathbf{B}^{n+1} \cdot \mathbf{w} &= \mathbf{B}^n \cdot \mathbf{w} \\ \mathbf{B}^{n+1} \cdot \delta\mathbf{x}^n &= \delta\mathbf{f}^n \end{aligned} \quad \Rightarrow \quad \mathbf{B}^{n+1} = \mathbf{B}^n + \frac{(\delta\mathbf{f}^n - \mathbf{B}^n \cdot \delta\mathbf{x}^n)}{\delta\mathbf{x}^n \cdot \delta\mathbf{x}^n} \otimes \delta\mathbf{x}^n$$

- Initialize $\delta\mathbf{x}^n$ and $\mathbf{B}_0$ with the identity matrix (or with finite difference approx.).

Broyden's Method

Python implementation of Broyden's method:

- Same example as before but now with Broyden's method.
- Slower convergence with Broyden's method should be offset by improved efficiency of each iteration!

```
broyden(@MyFunc,[1;2],1e-12,1e-12)
```

- Requires 11 iterations (compare with Newton: 5 iterations)
  But much fewer function evaluations per iteration!

```python
import numpy as np
from numpy.linalg import inv

def broyden(F, x0, tol=1e-6, max_iter=100):
    x = np.array(x0)
    B = np.eye(x.size)
    for i in range(max_iter):
        Fx = F(x)
        if np.linalg.norm(Fx) < tol:
            print(f"Converged after {i} iterations.")
            return x
        x_new = x - inv(B)@Fx
        delta_x = x_new - x
        delta_Fx = F(x_new) - Fx
        B += np.outer((delta_Fx - B@delta_x)/(delta_x@delta_x), delta_x)
        x = x_new
    print("Max iterations reached.")
    return x
```

Broyden's Method

- Same example as before but now with Broyden's method.
- Note how the approximate Jacobian (**B**) is updated over subsequent iterations:

$$\begin{bmatrix}1. & 0.\\ 0. & 1.\end{bmatrix} \rightarrow \begin{bmatrix}3. & -1.\\ 4. & -1.\end{bmatrix} \rightarrow \begin{bmatrix}-1.0 & -9.0\\ 3.4 & -2.2\end{bmatrix} \rightarrow \begin{bmatrix}-1.062 & -9.260\\ 3.411 & -2.154\end{bmatrix} \rightarrow$$

$$\begin{bmatrix}5.290 & -3.864\\ 2.493 & -2.934\end{bmatrix} \rightarrow \begin{bmatrix}7.363 & -1.931\\ 3.556 & -1.943\end{bmatrix} \rightarrow \begin{bmatrix}2.349 & -0.773\\ 3.547 & -1.941\end{bmatrix} \rightarrow \begin{bmatrix}-0.934 & -6.772\\ 2.351 & -4.124\end{bmatrix} \rightarrow$$

$$\begin{bmatrix}-0.384 & -5.879\\ 2.500 & -3.884\end{bmatrix} \rightarrow \begin{bmatrix}10.416 & 6.344\\ 5.947 & 0.018\end{bmatrix} \rightarrow \begin{bmatrix}9.781 & 5.515\\ 5.641 & -0.382\end{bmatrix} \rightarrow \begin{bmatrix}3.577 & 3.630\\ 3.362 & -1.074\end{bmatrix} \rightarrow$$

$$\begin{bmatrix}3.116 & 3.238\\ 2.912 & -1.458\end{bmatrix} \rightarrow \begin{bmatrix}1.992 & 3.272\\ 1.989 & -1.430\end{bmatrix} \rightarrow \dots \rightarrow \dots \rightarrow$$

- Compare with analytical jacobian: $\mathbf{B} = \begin{bmatrix}1.748 & 3.261\\ 1.736 & -1.439\end{bmatrix}$ $\mathbf{J} = \begin{bmatrix}1.779 & 3.583\\ 1.779 & -1\end{bmatrix}$
- Note that the approximate Jacobian (**B**) is not exact even when the solution has already been found!

Conclusions

- Recommendations for root finding:
  - One-dimensional cases:
    - If it is not easy/cheap to compute the function's derivative $\Rightarrow$ use Brent's algorithm.
    - If derivative information is available $\Rightarrow$ use Newton-Raphson's method + bookkeeping on bounds provided you can supply a good enough initial guess!!
    - There are specialized routines for (multiple) root finding of polynomials (but not covered in this course).
  - Multi-dimensional cases:
    - Use Newton-Raphson method, but make sure that you provide an initial guess close enough to achieve convergence.
    - In case derivative information is expensive $\Rightarrow$ use Broyden's method (but slower convergence!).