CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 9

Tuesday, October 5th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- Class methods, static methods and instance methods
- `python` modules
- `python` packages and the `python` package index (PyPI)
- Build a `python` package and publish on https://test.pypi.org/ (catch up today)

OUTLINE

- Towards automatic differentiation
  - The Jacobian and Newton's method
  - Numerical computation of derivatives

INTRODUCTION AND MOTIVATION

References for automatic differentiation:

- P. H.W. Hoffmann, *A Hitchhiker's Guide to Automatic Differentiation*, Springer 2015, doi:10.1007/s11075-015-0067-6 (You can access this paper through the Harvard network.)
- Griewank, A. and Walther, A., *Evaluating derivatives: principles and techniques of algorithmic differentiation*, SIAM 2008, Vol. 105

INTRODUCTION AND MOTIVATION

Differentiation is one of the most important operations in science.

- Finding extrema of functions and determining zeros of functions are central to optimization.
- Linearization of non-linear equations requires a prediction for a change in a small neighborhood which involves derivatives.
- *Numerically* solving differential equations forms a cornerstone of modern science and engineering and is intimately linked with predictive science.

INTRODUCTION AND MOTIVATION

Euler equations: a system of *partial differential equations* (PDEs) to describe compressible *fluid motion* in the form of *conservation laws*:

- Conservation of *mass* ρ
- Conservation of *momentum* $\rho\mathbf{u}$
- Conservation of *energy* E

The $\partial/\partial t$ and $\partial/\partial x$ are differential operators that describe the change in time and space of the *conserved* quantities ρ, ρu and E.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho u) = 0$$

$$\frac{\partial \rho u}{\partial t} + \frac{\partial}{\partial x}(\rho u^2 + p) = 0$$

$$\frac{\partial E}{\partial t} + \frac{\partial}{\partial x}\big((E + p)u\big) = 0$$

INTRODUCTION AND MOTIVATION

The Euler equations in the previous slide are highly *non-linear*. If we were to *linearize* the equations around a certain point q, we would need to find a so called ***Jacobian*** J of $f(q) \in \mathbb{R}^3$, where the input $q = [\rho, \rho u, E]^\intercal \in \mathbb{R}^3$ are the conserved variables.

We can then find the best linear approximation to $f(q + \Delta q)$ by projecting the Jacobian in the direction of a small change Δq, i.e., $f(q + \Delta q) \approx f(q) + J(q) \cdot \Delta q$. The Jacobian contains the first derivatives $J_{ij} = \partial f_i / \partial q_j$ and is a 3×3 matrix for this example.

INTRODUCTION AND MOTIVATION

A very frequent occurrence in science requires the scientist to find the zeros of a function $y = f(x)$. The input to the function is an m-dimensional vector $x \in \mathbb{R}^m$ and the function returns an n-dimensional vector $y \in \mathbb{R}^n$. We denote this mathematically as $f(x) : \mathbb{R}^m \mapsto \mathbb{R}^n$.

This expression is read: the function $f(x)$ maps $\mathbb{R}^m$ to $\mathbb{R}^n$.

EXAMPLE 1: NON-LINEAR SYSTEM

Consider the *system of non-linear* equations:

$$x_1 x_2^3 + \ln(x_3^2) = \sin(x_1 x_2 x_3)$$

$$x_1 + x_2 + \tan(x_3) = \frac{1}{x_1 x_2 x_3}.$$

We define the *vector*

$$x = [x_1, x_2, x_3]^\intercal = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix},$$

where we say $x \in \mathbb{R}^3$. Following the notation from above, $m = 3$.

EXAMPLE 1: NON-LINEAR SYSTEM

The function of interest is

$$f(x) = \begin{bmatrix} x_1 x_2^3 + \ln(x_3^2) - \sin(x_1 x_2 x_3) \\ x_1 + x_2 + \tan(x_3) - \dfrac{1}{x_1 x_2 x_3} \end{bmatrix}.$$

Thus $f(x)$ maps an input $x \in \mathbb{R}^3$ to $\mathbb{R}^2$ and we write $f(x) : \mathbb{R}^3 \mapsto \mathbb{R}^2$.

EXAMPLE 1: NON-LINEAR SYSTEM

If we plug-in numbers, say $x = [1, 2, 1]^\intercal$, and were to evaluate the non-linear system *at the point* x we find:

$$f\left(\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 7.09 \\ 4.06 \end{bmatrix}$$

NEWTON'S METHOD

We may have cause to find an x that renders $f(x) = 0$. This is *not so difficult for a linear system*, but *for a non-linear system it can be a major challenge*.

Newton's method is an algorithm with excellent convergence properties that allows us to find the ***roots*** x of a non-linear function f that satisfies $f(x) = 0$.

MATHEMATICAL TERMINOLOGY

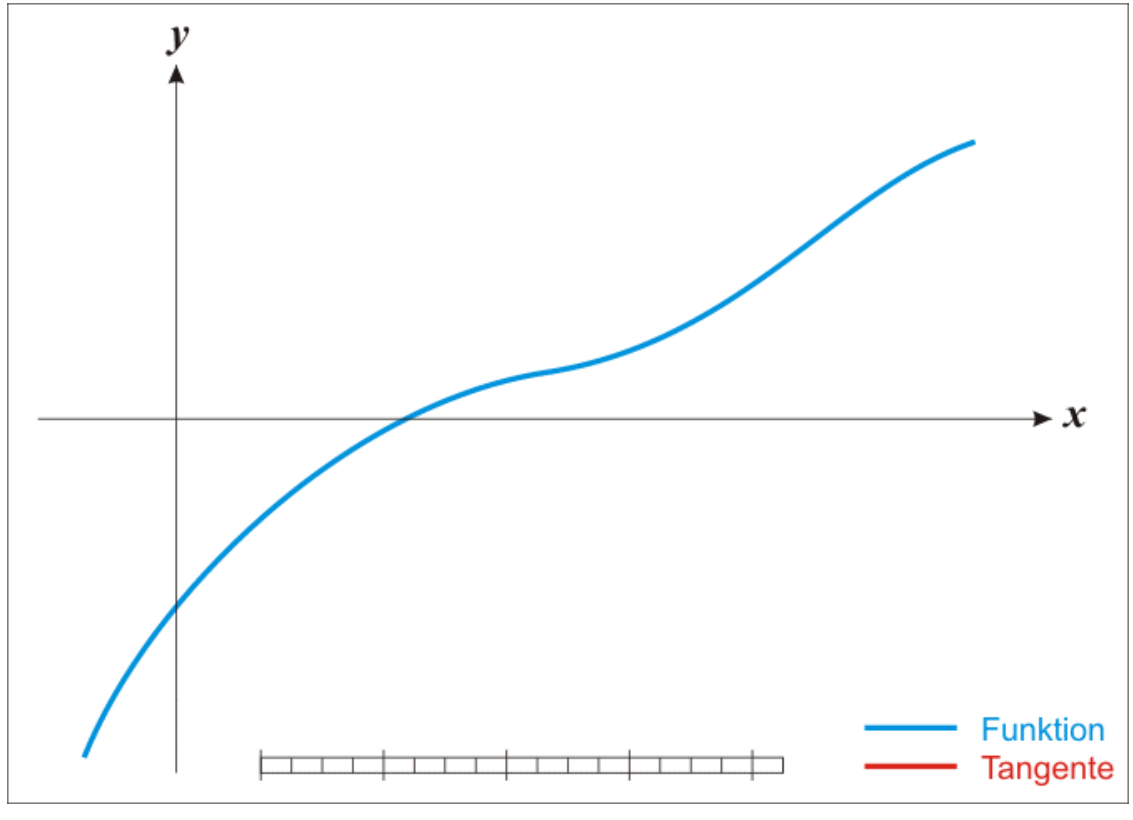
Spend ***10 minutes*** with your neighbors to discuss and understand the mathematical terminology just introduced.
Finding the roots of a function $f(x)$ has important practical applications.
Discuss some real applications where you would need to find roots.

DERIVATION OF NEWTON'S METHOD

The goal is to find $x \in \mathbb{R}^m$ such that $f(x) = 0$ for $f(x) \in \mathbb{R}^n$.

DERIVATION OF NEWTON'S METHOD

The algorithm visually:



Gif taken from Wikipedia

DERIVATION OF NEWTON'S METHOD

Step 1: choose an initial guess

Newton's method is an ***iterative*** method. We use the notation $x^{(k)}$ for the guess of the root at the k-th iteration. To start the algorithm you pick an *initial guess* at $k = 0$ for which most certainly $f(x^{(0)}) \neq 0$.

The method is not guaranteed to converge! Convergence depends on a good initial guess which requires some intuition and experience. When the method does converge, a solution with high accuracy can be found with only few iterations.

DERIVATION OF NEWTON'S METHOD

Step 2: explore the neighborhood

We look at a point just a little beyond $x^{(k)}$.
That is, we define the next iterate as

$$x^{(k+1)} = x^{(k)} + \Delta x^{(k)},$$

where $\Delta x^{(k)} = x^{(k+1)} - x^{(k)}$.

Note: we just introduce $\Delta x^{(k)}$ we do not know what its value should be.

DERIVATION OF NEWTON'S METHOD

Step 3: find a relationship between $f(x^{(k+1)})$ and $f(x^{(k)})$

Since we are looking in the neighborhood of $x^{(k)}$, the main tool we want here is a Taylor series expansion:

$$f(y) = \sum_{\kappa=0}^{\infty} \frac{f^{(\kappa)}(x)}{\kappa!}(y-x)^{\kappa}.$$

The notation $f^{(\kappa)}(x)$ means the κ-th derivative of f evaluated at x. It is a common mathematical notation and unrelated to $x^{(k)}$.

We substitute $y = x^{(k)} + \Delta x^{(k)}$ and $x = x^{(k)}$ and find:

$$f(x^{(k)} + \Delta x^{(k)}) = f(x^{(k)}) + \left.\frac{\partial f}{\partial x}\right|_{x=x^{(k)}} \Delta x^{(k)} + \text{h.o.t.}$$

DERIVATION OF NEWTON'S METHOD

Step 4: simplify

As our derivation is based on an *iterative* correction to $x^{(k)}$, we can argue that we may omit the higher order terms (h.o.t.) in our previous result at the cost of an *exact* relationship and possibly a few more iterations.

This simplifies to:

$$f(x^{(k)} + \Delta x^{(k)}) \approx f(x^{(k)}) + \left.\frac{\partial f}{\partial x}\right|_{x=x^{(k)}} \Delta x^{(k)}$$

DERIVATION OF NEWTON'S METHOD

Step 5: insert iteration criterion

We require a root for which $f(x^{(k+1)}) = 0$ which implies that $\Delta x^{(k)} = 0$ when converged (note that in the previous step we have sacrificed accuracy for simplicity but I will continue to use the '$=$' sign in the following).

$$f(x^{(k)}) + \left.\frac{\partial f}{\partial x}\right|_{x=x^{(k)}} \Delta x^{(k)} = 0$$

Note: this now allows us to solve for the unknown $\Delta x^{(k)}$.

DERIVATION OF NEWTON'S METHOD

Step 6: rearrange and interpret (scalar case)

Although it was stated at the beginning that $x \in \mathbb{R}^m$ and the image $f(x) \in \mathbb{R}^n$, it was silently assumed that $f(x)$ is a single variate scalar function to keep the Taylor series simple. In that case we can write the following iteration rule:

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})},$$

where $f'(x^{(k)}) \neq 0$ is the first derivative $\partial f / \partial x$ evaluated at the root $x^{(k)}$ of iteration k. To start the iterations we need an initial guess $x^{(0)}$.

DERIVATION OF NEWTON'S METHOD

Step 6: rearrange and interpret (general case)

In general we have $x \in \mathbb{R}^m$ and the image $f(x) \in \mathbb{R}^n$. We can then no longer simply divide by $f'(x)$ because we have a ***different structure***. Similar to the example of linearizing the Euler equations, the first order term in the Taylor series becomes $J(x^{(k)})\Delta x^{(k)}$ with $J(x^{(k)}) \in \mathbb{R}^{n \times m}$ the ***Jacobian*** of $f(x)$ evaluated at $x^{(k)}$ (now a $n \times m$ matrix with elements $\partial f_i / \partial x_j$).

DERIVATION OF NEWTON'S METHOD

Step 6: rearrange and interpret (general case)

In the general form of Newton's method we have to *solve a linear system* to obtain the correction $\Delta x^{(k)} \in \mathbb{R}^m$. The iteration rule now is:

$$J(x^{(k)})\Delta x^{(k)} = -f(x^{(k)}),$$
$$x^{(k+1)} = x^{(k)} + \Delta x^{(k)},$$

with some initial guess $x^{(0)}$. In every iteration k, we must solve a linear system for the m unknown corrections which we need to advance to $x^{(k+1)}$. The iterations are repeated until $\Delta x^{(k)} < \varepsilon$ where ε is a tolerance below which we consider the algorithm converged.

DERIVATION OF NEWTON'S METHOD

A few notes about what we just did:

- At the heart of Newton's method is the Jacobian J
- In order to use the algorithm, we need J which means we must compute derivatives and *evaluate* them at a point $x^{(k)}$.
- We can obtain J in different ways:
  - compute the derivatives manually
  - with a software for symbolic math
  - automatic differentiation
  - through a numerical approximation like Finite-Differences
- An accurate representation of J is key for good convergence behavior of the method.

EXAMPLE 2: INTERSECTION OF TWO LINES

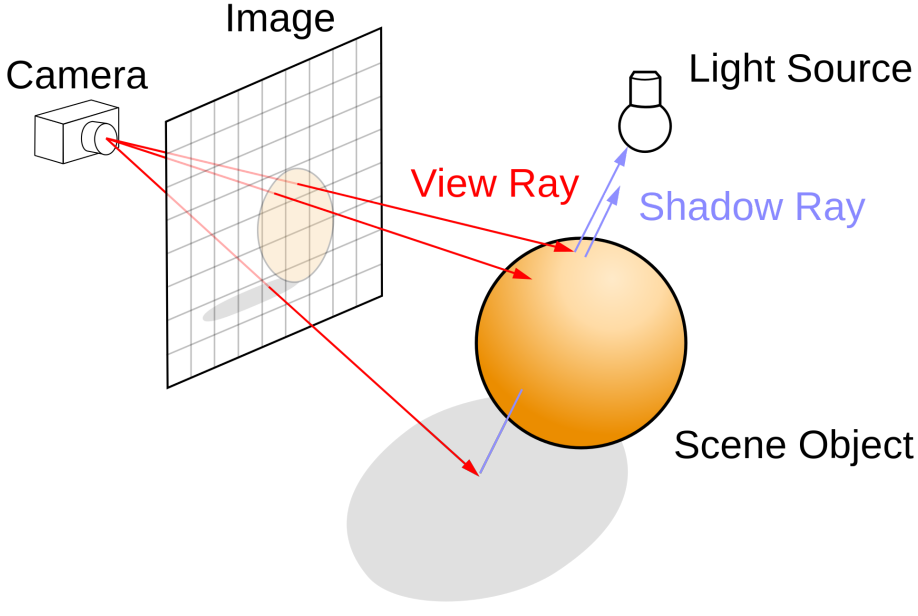
Given two functions $y_1 = x$ and $y_2 = \exp\left(-2\left(\sin(4x)\right)^2\right)$, find x such that $y_1 = y_2$.

This statement is equivalent to find x such that

$$f(x) = x - \exp\left(-2\left(\sin(4x)\right)^2\right) = 0.$$

EXAMPLE 2: INTERSECTION OF TWO LINES

A real world application is in ray-tracing to generate photo realistic images. Rays intersect with the surface of complex objects and are traced to compute an approximation of pixel color values.



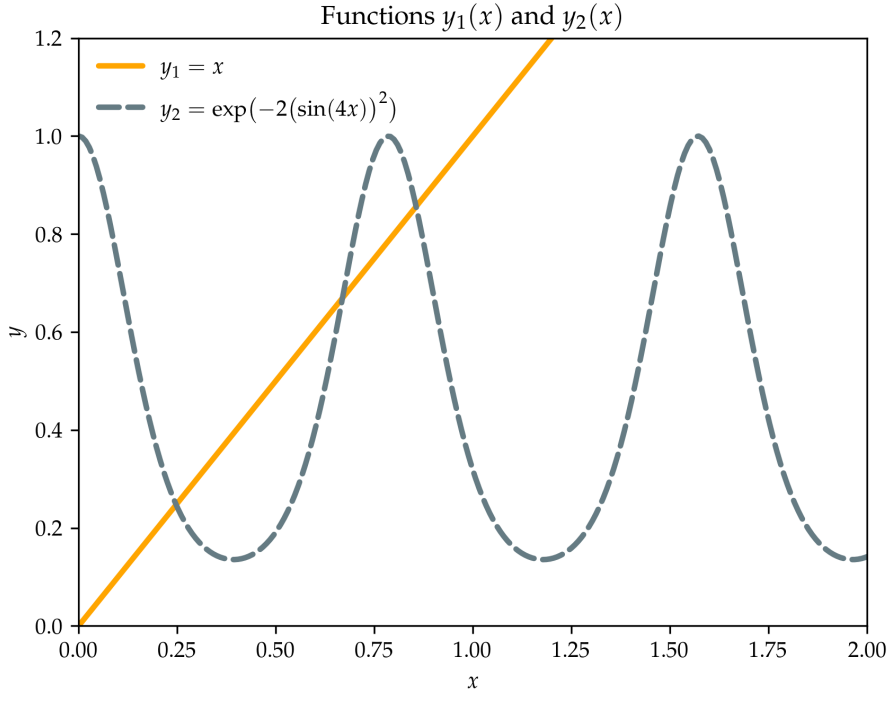
EXAMPLE 2: INTERSECTION OF TWO LINES

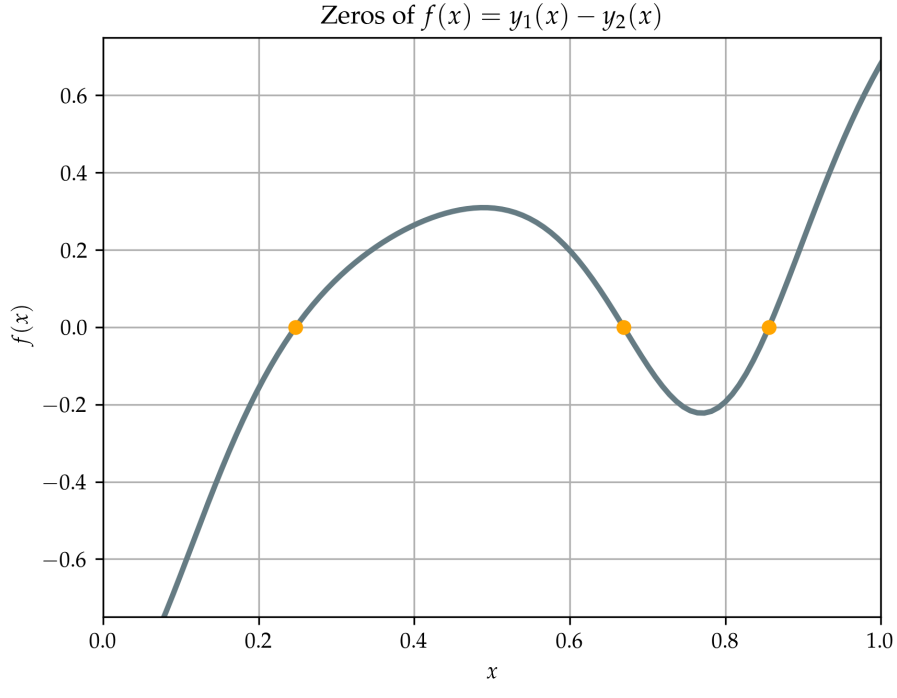
Before we start, it is a good idea to visualize our problem:

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 700)
y = np.exp(-2 * np.sin(4 * x)**2)
fig, ax = plt.subplots()
ax.plot(x, x, linewidth=2.5, label=r'$y_1=x$')
ax.plot(x,
        y,
        linewidth=2.5,
        linestyle='--',
        label=r'$y_2=\exp\bigl(-2\bigl(\sin(4x)\bigr)^2\bigr)$')
ax.set_xlim(0, 2)
ax.set_ylim(0, 1.2)
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.legend()
fig.savefig('example2_vis.png', dpi=300, bbox_inches='tight')
```

EXAMPLE 2: INTERSECTION OF TWO LINES

Before we start, it is a good idea to visualize our problem:





There are *three* zeros and we can not solve this problem by hand.
Let us try Newton's method.

EXAMPLE 2: INTERSECTION OF TWO LINES

Let us think about the design of our program:

- We need an initial guess
- We need some termination criterion
- We want to protect from infinite iterations if the algorithm diverges
- We would like to pass the parameter as arguments

Sketch:

```
x_k # initial guess
tol # convergence tolerance
max_it # maximum iterations
for k in range(max_it):
    dx_k = -f(x_k) / dfdx(x_k) # compute corr
    if abs(dx_k) < tol: # check for converger
        root = x_k + dx_k
        break
    x_k += dx_k # update the iteration variab
```

EXAMPLE 2: INTERSECTION OF TWO LINES

At this point we need to determine the Jacobian J of $f(x)$.

Given the function

$$f(x) = x - \exp\left(-2\left(\sin(4x)\right)^2\right),$$

compute the derivative df/dx:

1. By hand on a piece of paper
2. Check your calculus by using the `sympy` `python` package for symbolic math. Write a small `.py` script or Jupyter notebook and commit the code in your class repository under `lectures/lecture09` on your `main` or `master` branch. The online documentation can be found here or a pdf can be downloaded here. You can install the package with

```
python -m pip install [--user] sympy
```

You may collaborate with your neighbors (~15 minutes)

EXAMPLE 2: INTERSECTION OF TWO LINES

We add the functions $f(x)$ and $J(x)$ as anonymous `python` lambda's (you could also use normal function objects):

```python
import numpy as np

f = lambda x: x - np.exp(-2.0 * np.sin(4.0 * x) * np.sin(4.0 * x))
J = lambda x: 1.0 + 16.0 * np.exp(-2.0 * np.sin(4.0 * x)**2) * np.sin(4.0 * x) * np.cos(4.0 * x)

x_k # initial guess
tol # convergence tolerance
max_it # maximum iterations
for k in range(max_it):
    dx_k = -f(x_k) / J(x_k)
    if abs(dx_k) < tol:
        root = x_k + dx_k
        break
    x_k += dx_k
```

To handle arguments we can use the `argparse` `python` module.

EXAMPLE 2: INTERSECTION OF TWO LINES

Packing everything into a module:

```python
#!/usr/bin/env python3
import numpy as np

f = lambda x: x - np.exp(-2.0 * np.sin(4.0 * x) * np.sin(4.0 * x))
J = lambda x: 1.0 + 16.0 * np.exp(-2.0 * np.sin(4.0 * x)**2) * np.sin(4.0 * x) * np.cos(4.0 * x)

def newton(f, J, x_k, tol=1.0e-8, max_it=100):
    root = None
    for k in range(max_it):
        dx_k = -f(x_k) / J(x_k)
        if abs(dx_k) < tol:
            root = x_k + dx_k
            print(f"Found root {root:e} at iteration {k+1}")
            break
        print(f"Iteration {k+1}: Delta x = {dx_k:e}")
        x_k += dx_k
    return root

if __name__ == "__main__":
    import argparse
    def parse_args():
        parser = argparse.ArgumentParser(description="Newton-Raphson Method")
        parser.add_argument('-g', '--initial_guess', type=float, help="Initial guess", required=True)
        parser.add_argument('-t', '--tolerance', type=float, default=1.0e-8, help="Convergence tolerance")
        parser.add_argument('-i', '--maximum_iterations', type=int, default=100, help="Maximum iterations")
        return parser.parse_args()

    args = parse_args()
    newton(f, J, args.initial_guess, args.tolerance, args.maximum_iterations)
```

EXAMPLE 2: INTERSECTION OF TWO LINES

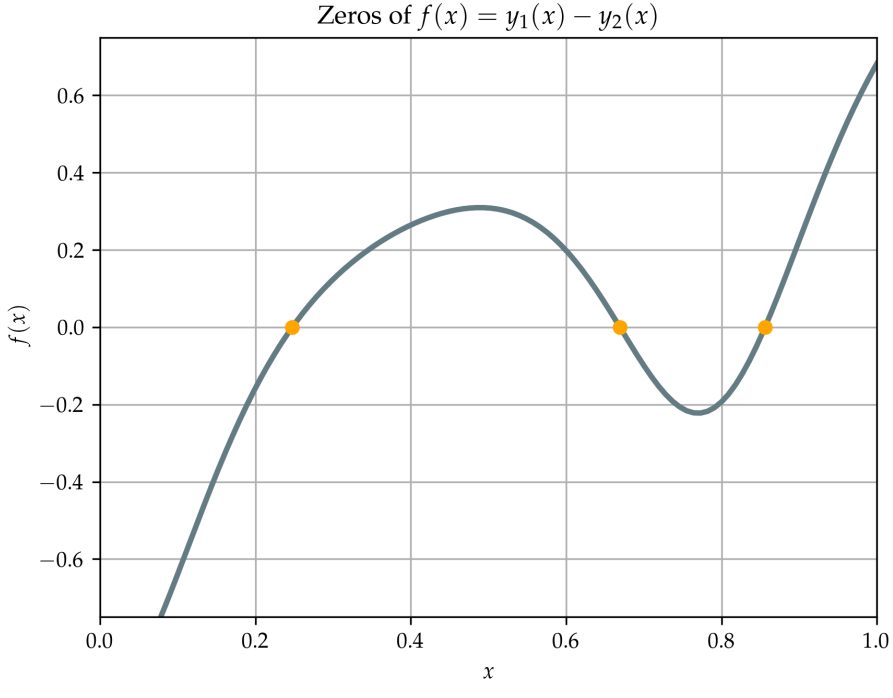
Recall: the check whether `__name__` corresponds to the `'__main__'` top-level scope allows us to run our module just like a program

```
$ chmod 755 newton.py
$ ./newton.py --help
usage: newton.py [-h] -g INITIAL_GUESS [-t TOLERANCE] [-i MAXIMUM_ITERATIONS]

Newton-Raphson Method

optional arguments:
  -h, --help            show this help message and exit
  -g INITIAL_GUESS, --initial_guess INITIAL_GUESS
                        Initial guess
  -t TOLERANCE, --tolerance TOLERANCE
                        Convergence tolerance
  -i MAXIMUM_ITERATIONS, --maximum_iterations MAXIMUM_ITERATIONS
                        Maximum iterations
$ ./newton.py --initial_guess 0.1
Iteration 1: Delta x = 1.218877e-01
Iteration 2: Delta x = 2.339599e-02
Iteration 3: Delta x = 2.066548e-03
Iteration 4: Delta x = 1.500080e-05
Found root 2.473652e-01 at iteration 5
```

For the initial guess `0.1`, the method seems to find a root at `2.473652e-01`.

EXAMPLE 2: INTERSECTION OF TWO LINES

Validation of the result:




```
>>> from newton import (f, J, newton)
>>> root = newton(f, J, 0.1)
Iteration 1: Delta x = 1.218877e-01
Iteration 2: Delta x = 2.339599e-02
Iteration 3: Delta x = 2.066548e-03
Iteration 4: Delta x = 1.500080e-05
Found root 2.473652e-01 at iteration 5
>>> f(root)
5.551115123125783e-17 # about zero
```

Note that the initial guess is crucial:

```
>>> newton(f, J, 0.6)
Found root 6.692328e-01 at iteration 5
>>> newton(f, J, 0.9)
Found root 8.560317e-01 at iteration 4
```

EXAMPLE 2: INTERSECTION OF TWO LINES

Summary:

- Derivatives are foundational in science and engineering.
- We illustrated a situation from optimization where we try to find the roots of a complicated, high-dimensional nonlinear function. The algorithm that we used was Newton's method, which requires evaluations of the Jacobian.
- We saw that the Jacobian also showed up when we *linearized* the Euler equations around a point q.
- The Jacobian requires the evaluation of derivatives for a given function at some point of interest x.
- We computed the derivatives ***by hand*** in our previous example. What if we can not do that or have other reasons of not doing it?

THE FINITE-DIFFERENCE METHOD

Suppose we want to avoid relying on the symbolic computation of the derivative. For the single-variate scalar function $f(x)$ we found the following relationship through the Taylor series expansion:

$$f(x+\varepsilon) = f(x) + \left.\frac{df}{dx}\right|_x \varepsilon + \text{h.o.t.},$$

where ε is a *small* parameter.

If we again drop the higher order terms, we get the following approximation for the derivative:

$$\left.\frac{df}{dx}\right|_x \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

THE FINITE-DIFFERENCE METHOD

We have introduced *another parameter* in order to approximate the derivative *numerically* with sole knowledge of $f(x)$. We do not know how to choose ε but it has to be small because our Taylor series Ansatz assumes we are looking in the close neighborhood of point x. Let's assume a value $\varepsilon = 10^{-2}$ and replace our previous Jacobian in our Newton module:

```
1 import numpy as np
2
3 f = lambda x: x - np.exp(-2.0 * np.sin(4.0 * x) * np.sin(4.0 * x))
4 J = lambda x, eps: (f(x + eps) - f(x)) / eps # Finite-Difference approximation of J
```

THE FINITE-DIFFERENCE METHOD

```
1 import numpy as np
2
3 f = lambda x: x - np.exp(-2.0 * np.sin(4.0 * x) * np.sin(4.0 * x))
4 J = lambda x, eps: (f(x + eps) - f(x)) / eps # Finite-Difference approximation of J
```

We now run Newton's method again, with our numerical approximation of $J(x)$:

```
 1 >>> from newton_fd import (f, J, newton)
 2 >>> root = newton(f, J, 0.1, eps=1.0e-2)
 3 Iteration 1: Delta x = 1.211561e-01
 4 Iteration 2: Delta x = 2.482629e-02
 5 Iteration 3: Delta x = 1.424802e-03
 6 Iteration 4: Delta x = -4.341516e-05
 7 Iteration 5: Delta x = 1.539820e-06
 8 Iteration 6: Delta x = -5.437925e-08
 9 Found root 2.473652e-01 at iteration 7
10 >>> f(root)
11 1.8454707206849719e-10
```

Compared to the previous case, we have lost *seven* orders of magnitude in accuracy and require *two* extra iterations.

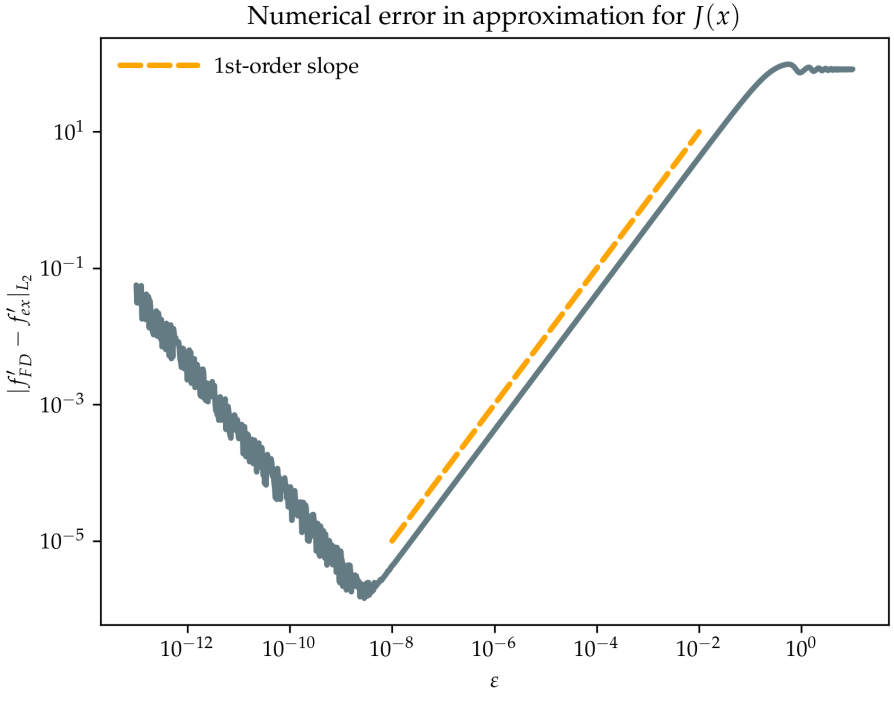
(See the `newton_fd.py` script on the lecture materials site.)

THE FINITE-DIFFERENCE METHOD

Since we know the exact form for $J(x)$ we can analyze the *numerical error* of our Finite-Difference approximation as we vary ε:

```
 1 import numpy as np
 2
 3 f = lambda x: x - np.exp(-2.0 * np.sin(4.0 * x) * np.sin(4.0 * x))
 4 J = lambda x: 1.0 + 16.0 * np.exp(-2.0 * np.sin(4.0 * x)**2) * np.sin(4.0 * x) * np.cos(4.0 * x)
 5 J_fd = lambda x, eps: (f(x + eps) - f(x)) / eps # Finite-Difference approximation of J
 6
 7 x = np.linspace(0.0, 2.0, 1000) # domain for f
 8 epsilon = np.logspace(-13, 1, 1000) # discretization \epsilon
 9 error = np.zeros(len(epsilon)) # array for L2 errors
10 for i, eps in enumerate(epsilon):
11     e = J_fd(x, eps) - J(x) # numerical error for all values in x
12     error[i] = np.linalg.norm(e) # compute L2 error norm
```

THE FINITE-DIFFERENCE METHOD



Observations:

- The numerical error for this approximation of $J(x)$ has a minimum around 10^{-6}
- The minimum error was *not* obtained at the smallest possible ε of about 10^{-16} for double precision according to the IEEE 754 standard (*machine precision*).
- Too small ε amplify the floating point error while ε too large does not provide a good approximation for the derivative.
- The method *reduces* the floating point error by one *decade* if we reduce ε by one decade (1st-order accurate).

THE FINITE-DIFFERENCE METHOD

It is not clear how to choose the best ε in general. Some results from numerical analysis suggest that it should be around $\sqrt{\varepsilon_{\text{machine}}}$ as a *rule of thumb* for a 1st-order method.

In the example before, the minimum numerical error was 1.438669×10^{-6} and corresponds to $\varepsilon = 2.860596 \times 10^{-9}$. If we compute the square root of $\varepsilon_{\text{machine}}$ in `python` we find:

```
>>> np.finfo(float).eps
2.220446049250313e-16
>>> np.sqrt(_)
1.4901161193847656e-08
```

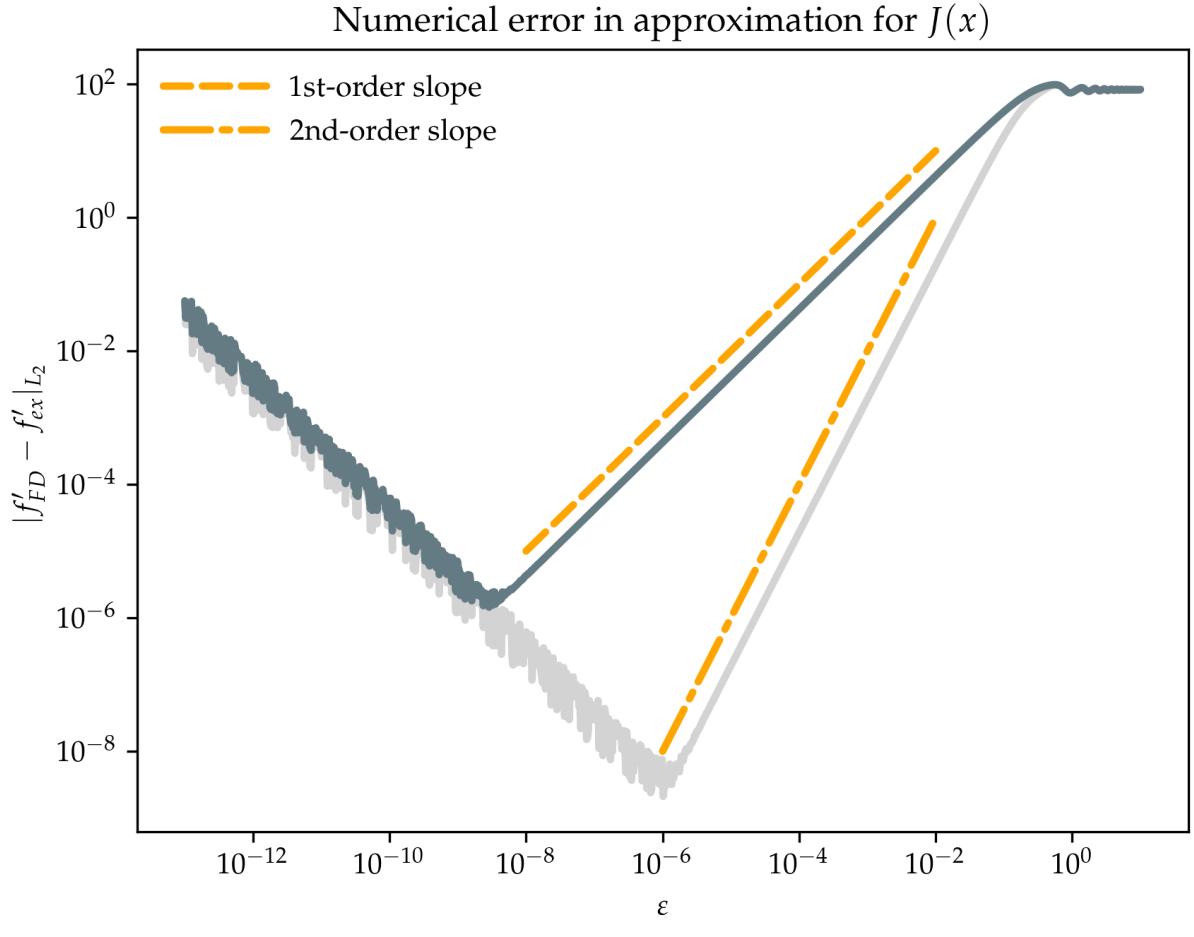
THE FINITE-DIFFERENCE METHOD (ADDITIONAL)

We have used the Taylor series expansion for $f(x+\varepsilon)$ for our previous method. We can do another Taylor series for $f(x-\varepsilon)$ and subtract the series from the previous one. This trick will *eliminate* the leading order term and we *gain an extra order of accuracy*. The method then becomes:

$$\left.\frac{df}{dx}\right|_x \approx \frac{f(x+\varepsilon)-f(x-\varepsilon)}{2\varepsilon}$$

THE FINITE-DIFFERENCE METHOD (ADDITIONAL)

Error analysis for this method reveals its superior accuracy:



THE FINITE-DIFFERENCE METHOD (ADDITIONAL)

If we return to the case where $x \in \mathbb{R}^m$ for $m > 1$, we can compute the partial derivative with respect to coordinate x_j by

$$\left.\frac{\partial f}{\partial x_j}\right|_x \approx \frac{f(x + \varepsilon e_j) - f(x)}{\varepsilon},$$

where e_j denotes the *unit vector* in the direction of x_j.

THE CONVERGENCE ORDER OF NEWTON'S METHOD?

Approximate the square root of an arbitrary number α using a few Newton iterations (~6-7). ***Plot the error between the current approximation and the true solution for each iteration*** in a plot with logarithmic y-axis. How much is the error reduced between consecutive iterations?

Write a small `python` module (`newton_sqrt.py`) with a function `newton_iter` that performs one Newton iteration each time it is called. Compute the error as

$$e = \left| \frac{v - v_{\text{exact}}}{v_{\text{exact}}} \right|.$$

Commit the code in your class repository under `lectures/lecture09` on your `main` or `master` branch. Name the module `newton_sqrt.py` and add a `if __name__ == "__main__":` statement at the end of your module.

You may collaborate with your neighbors (~15 minutes)

TOWARDS AUTOMATIC DIFFERENTIATION

- In the introduction, we motivated the need for computational techniques to compute derivatives.
- We focused on the Jacobian J, a $n \times m$ matrix with first derivatives of a mapping $f(x) : \mathbb{R}^m \mapsto \mathbb{R}^n$.
- We have discussed the computation of J with symbolic math which is accurate but may not always be applicable depending on $f(x)$ or may be too costly to evaluate.
- Numerical computation of J may be an alternative method at the cost of accuracy reduction and possible stability issues.
- Automatic differentiation (AD) overcomes both of these deficiencies. It is less costly than symbolic differentiation while evaluating derivatives at *machine precision*. There are *two* modes of AD: ***forward*** and ***reverse***, both involve the Jacobian J. The back-propagation algorithm in machine learning is a special case of the reverse AD mode.

RECAP

- Towards automatic differentiation
  - The Jacobian and Newton's method
  - Numerical computation of derivatives