SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 11

Fabian Wermelinger
Harvard University
CS107 / AC207

Thursday, October 6th 2022

LAST TIME

- Python packages and the Python Package Index
- Towards automatic differentiation
- Linearization of Euler equations as an example for the Jacobian
- Newton's method

TODAY

Main topics: ***Chain rule, Gradient operator and directional derivative, Automatic differentiation: forward mode***

Details:

- Example of Newton's method and numerical schemes to approximate derivatives.
- Evaluation trace
- The computational graph
- Computing derivatives of one variable using the forward mode

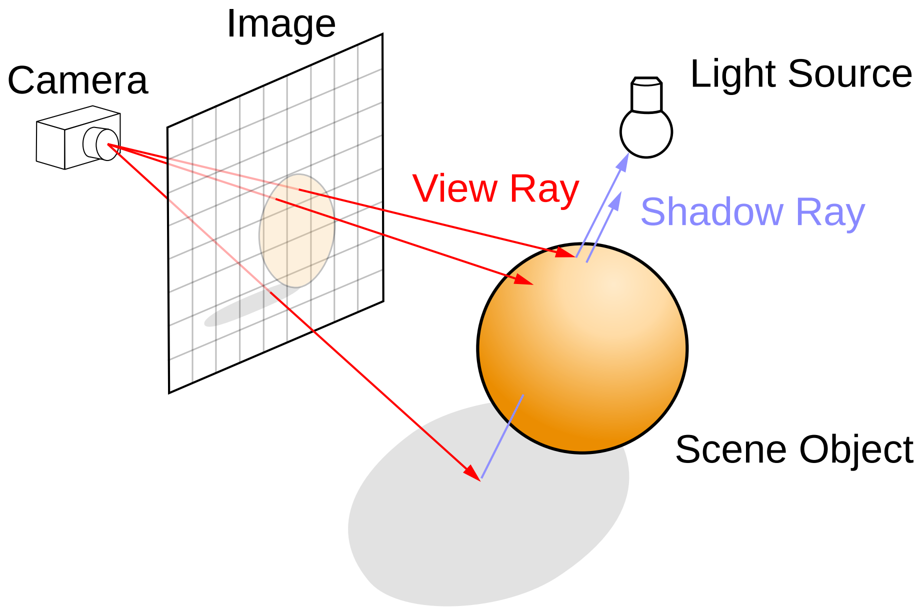
NEWTON'S METHOD: INTERSECTION OF TWO LINES

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

Find x such that $y_1 = y_2$. ***The statement is equivalent to:***

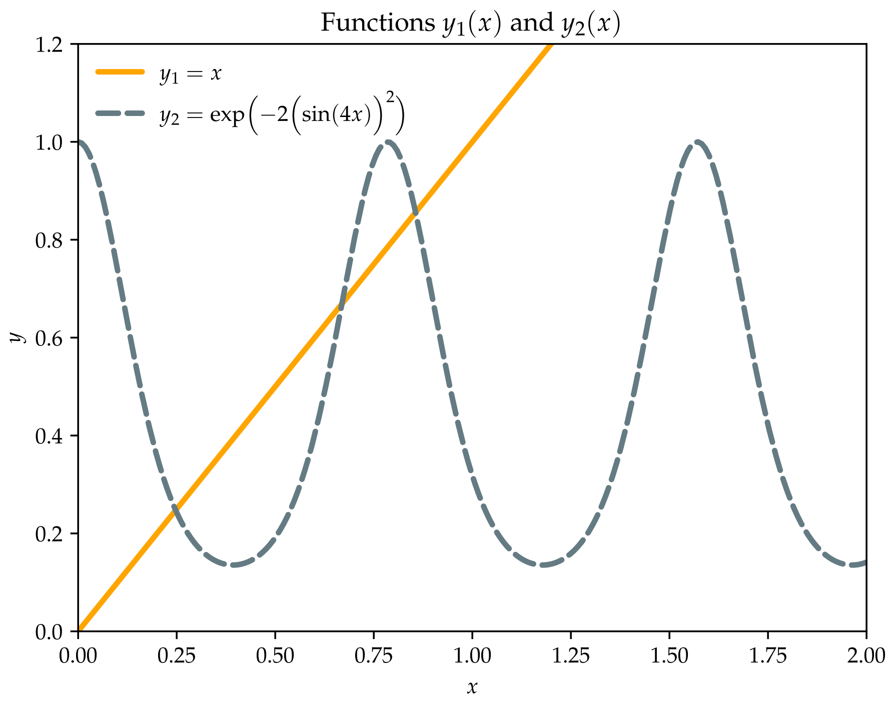
$$f(x) = x - \exp\left(-2\left(\sin(4x)\right)^2\right) = 0.$$

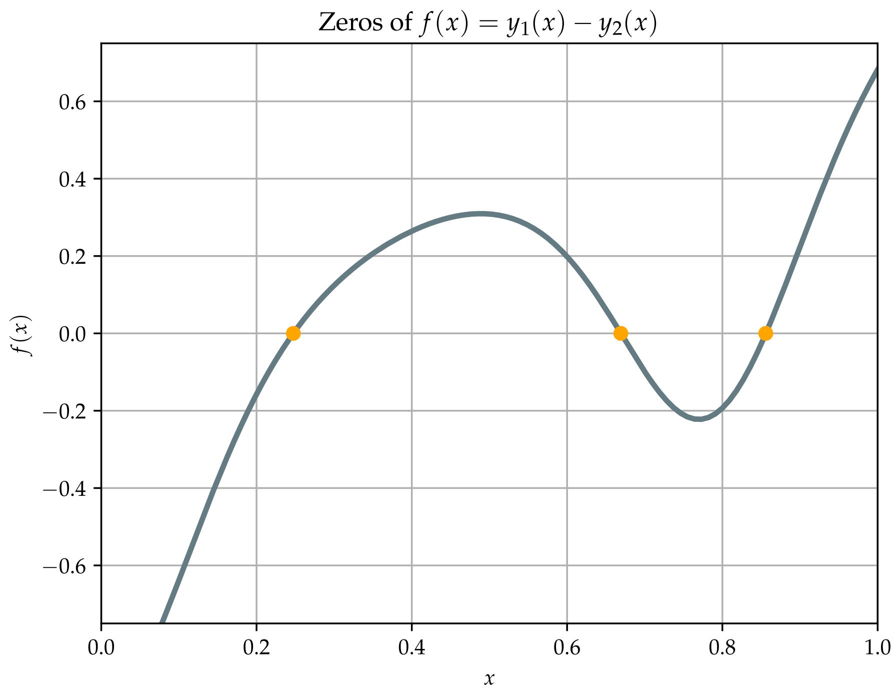
Real world application: ray-tracing



NEWTON'S METHOD: INTERSECTION OF TWO LINES

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





There are *three* zeros and we can not solve this problem by hand.

Method of choice → Newton's method

NEWTON'S METHOD: INTERSECTION OF TWO LINES

How should the algorithm in our program look like?

- Need an *initial guess*
- Need some *termination criterion*
- Want to protect from infinite iterations if there is *divergence*
- ***Optional:*** Would like to pass *parameters as arguments*

Algorithm sketch:

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

NEWTON'S METHOD: INTERSECTION OF TWO LINES

We need to determine the Jacobian J of $f(x)$:

Given the function

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

find the derivative df/dx (*for this scalar case $J(x) = df/dx$*).

Options:

1. On a piece of paper
2. Using software to calculate derivatives *analytically*, e.g. Mathematica. Python also supports this with the SymPy package for symbolic math. You can install the package with

   ```
   $ python -m pip install sympy
   ```

   See the online documentation for more. *You can find a Python script and a Jupyter notebook for this example in the lecture code handout.*

NEWTON'S METHOD: INTERSECTION OF TWO LINES

Once the Jacobian $J(x)$ is known we can complete Newton module:

```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)

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
```

(To handle arguments we can use the `argparse` module from the Python standard library)

You can find this code in the `newton.py` module in the class repository.

NEWTON'S METHOD: INTERSECTION OF TWO LINES

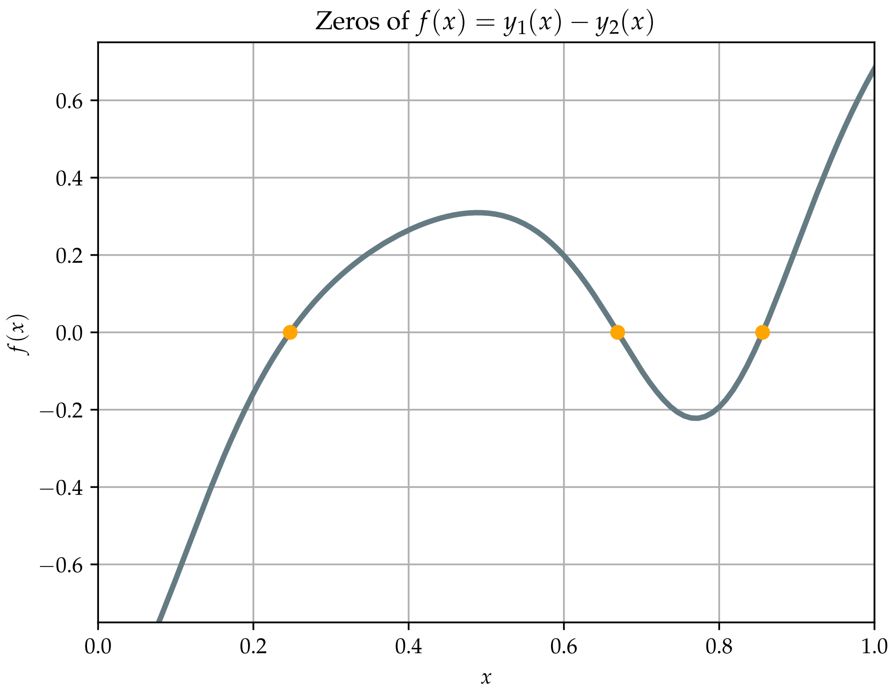
Recall: the check whether `__name__` corresponds to the `'__main__'` string in the top-level scope *allows us to run our module just like a program!*

```
$ 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
```

→ for the initial guess `0.1`, the method finds *one* root at `2.473652e-01`

NEWTON'S METHOD: 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 # converges to second root!
>>> newton(f, J, 0.9)
Found root 8.560317e-01 at iteration 4 # converges to third root!
```

NEWTON'S METHOD: INTERSECTION OF TWO LINES

Summary:

- We worked through a root finding problem to solve for the intersection of two lines based on a nonlinear function. An example application where this problem arises is ray-casting.
- Finding the solution to this problem required use of Newton's method which requires knowledge of the *derivative* of the system governing function $f(x)$.
- For a general setup in 3D space, there is not just one of these derivatives! In this case we call these "derivatives" the *Jacobian* of the system. The Jacobian also showed up when we *linearized* the Euler equations around a point q in the motivation of the previous lecture.
- In applications (e.g. root finding), the Jacobian needs to be *evaluated* at some point of interest x. To do this, we computed an analytic representation of Jacobian *by hand*.
- What if we cannot do that or have other reasons of not doing it?

THE FINITE-DIFFERENCE METHOD

- Suppose we want to avoid the calculation of the analytic form for derivative. For the single-variate scalar function $f(x)$ from the previous example, we found the following relationship through the Taylor series expansion:

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

where ε is a *small* parameter.

- If we again drop the higher order terms, we get the following ***finite difference*** approximation for the derivative, where ε is a *characteristic length scale* with units of x:

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

THE FINITE-DIFFERENCE METHOD

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

```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, eps: (f(x + eps) - f(x)) / eps # Finite-Difference approximation of J
```

THE FINITE-DIFFERENCE METHOD

- Let us assume a value $\varepsilon = 10^{-2}$ and replace the previous exact Jacobian in the Newton module:

```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, eps: (f(x + eps) - f(x)) / eps # Finite-Difference approximation of J
```

- If we run Newton's method again, with the numerical approximation of $J(x)$: *(see lecture code handouts)*

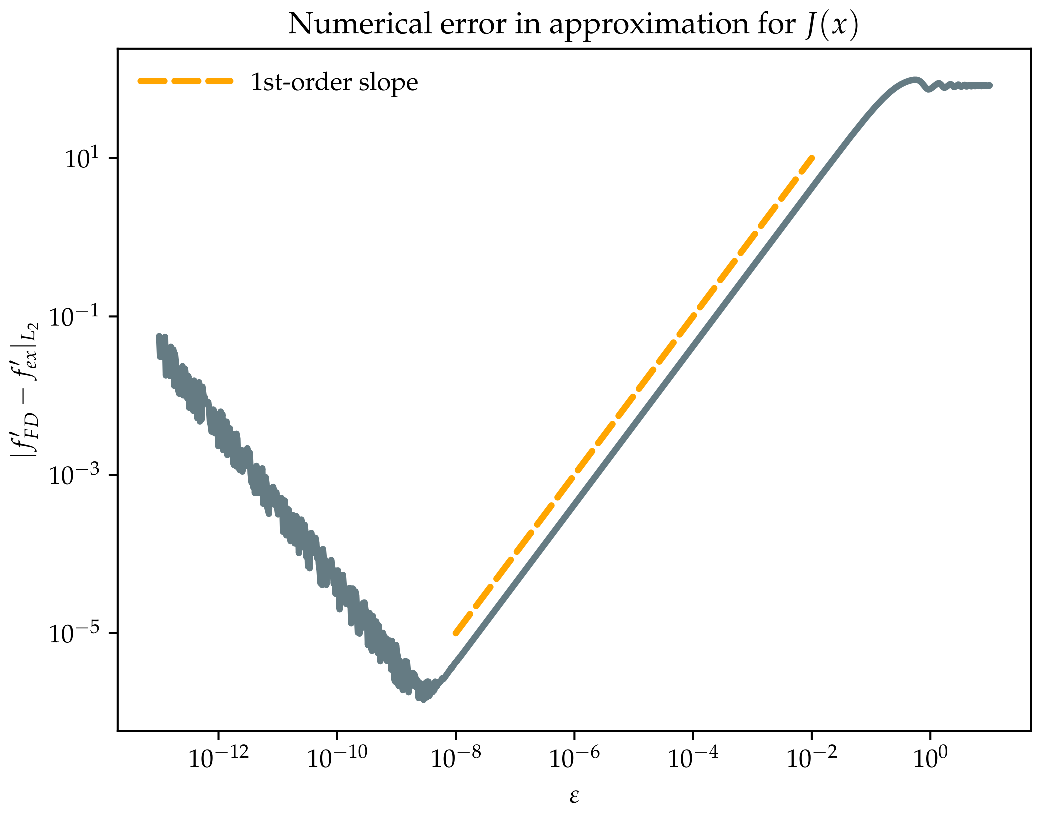
```
>>> from newton_fd import (f, J, newton)
>>> root = newton(f, J, 0.1, eps=1.0e-2)
Iteration 1: Delta x = 1.211561e-01
Iteration 2: Delta x = 2.482629e-02
Iteration 3: Delta x = 1.424802e-03
Iteration 4: Delta x = -4.341516e-05
Iteration 5: Delta x = 1.539820e-06
Iteration 6: Delta x = -5.437925e-08
Found root 2.473652e-01 at iteration 7 # compared to before: 2 extra iterations!
>>> f(root)
1.8454707206849719e-10 # compared to before: 7 orders of magnitude less accurate!
```

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:

```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) # exact J
J_fd = lambda x, eps: (
    f(x + eps) - f(x)
) / eps  # Finite-Difference approximation of J

x = np.linspace(0.0, 2.0, 1000)     # domain for f
epsilon = np.logspace(-13, 1, 1000) # discretization \epsilon
error = []                          # array for L2 errors
for eps in epsilon:
    e = J_fd(x, eps) - J(x)         # numerical error for all values in x
    error.append(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 at around 10^{-6}
- The minimum error was ***not*** obtained at the smallest possible ε of about 10^{-16} (*machine precision* → *for double precision based on to the IEEE 754 standard)*
- 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 $\sqrt{\varepsilon_{\mathrm{machine}}}$ as a *rule of thumb* for a *1st-order method*.
- ***Proof:*** 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 $\varepsilon_{\mathrm{machine}}$ in Python we find:

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

which is about the ε we can find in the plot on the previous slide.

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 order partial 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 mode*** and ***reverse mode***, both involve the Jacobian J. *The back-propagation algorithm in machine learning is a special case of reverse mode AD.*

THE BASIC IDEAS OF AUTOMATIC DIFFERENTIATION

- We have discussed the computation of the Jacobian J using symbolic math tools which is accurate but may not always be applicable depending on the complexity of $f(x)$ (or it may be too costly to evaluate).
- Numerical computation of J may be an alternative method at the cost of accuracy and possible stability issues.
- *Automatic differentiation (AD)* overcomes both of these deficiencies. It is:
  - less costly than symbolic differentiation
  - evaluates derivatives to machine precision
- There are two modes of AD: ***forward mode*** and ***reverse mode***. *The back-propagation algorithm in machine learning is a special case of reverse mode automatic differentiation.*
- ***Automatic differentiation is based on evaluating the chain rule step by step.***

REVIEW OF THE CHAIN RULE

At the heart of AD is the piecewise evaluation of the ***chain rule***.

- Suppose we have a function $f(u(t))$ and we want to compute the derivative of f with respect to t. This derivative is given by

$$\boxed{\frac{df}{dt} = \frac{\partial f}{\partial u}\frac{du}{dt}}$$

- ***Example:*** given $f(u(t)) = \sin(4t)$ with $u(t) = 4t$:

$$\frac{\partial f}{\partial u} = \cos(u), \qquad \frac{du}{dt} = 4 \qquad \Rightarrow \qquad \frac{df}{dt} = 4\cos(4t)$$

REVIEW OF THE CHAIN RULE

The total change of f is given by the sum of the partial changes in each coordinate direction.

- Suppose f has another coordinate $v(t)$ so that we have $f(u(t), v(t))$. Once again, we want to compute the derivative of f with respect to t. Applying the chain rule in this case gives

$$\frac{df}{dt} = \underbrace{\frac{\partial f}{\partial u}\frac{du}{dt}}_{\text{Change due to } u} + \underbrace{\frac{\partial f}{\partial v}\frac{dv}{dt}}_{\text{Change due to } v}$$

- Later we will extend this to an *arbitrary* number of coordinates u, v, w, ...

REVIEW OF THE CHAIN RULE

- *Examples:*

$$\boxed{\frac{df}{dt} = \underbrace{\frac{\partial f}{\partial u}\frac{du}{dt}}_{\text{Change due to } u} + \underbrace{\frac{\partial f}{\partial v}\frac{dv}{dt}}_{\text{Change due to } v}}$$

$$f(u(t), v(t)) = u + v \quad \Rightarrow \quad \frac{df}{dt} = \frac{du}{dt} + \frac{dv}{dt}$$

$$f(u(t), v(t)) = uv \quad \Rightarrow \quad \frac{df}{dt} = v\frac{du}{dt} + u\frac{dv}{dt}$$

$$f(u(t), v(t)) = \sin(uv) \quad \Rightarrow \quad \frac{df}{dt} = v\cos(uv)\frac{du}{dt} + u\cos(uv)\frac{dv}{dt}$$

REVIEW OF THE CHAIN RULE

The gradient operator ∇:

In vector calculus, the gradient describes the *steepest increase* of a scalar function $f(x)$. This steepest increase is along a certain *direction* given by coordinates $x \in \mathbb{R}^m$.

In our 3D world the dimension is $m = 3$. In general, the coordinate x is can be m-dimensional. Assuming a 3D space with coordinates $x = [x_1, x_2, x_3]^\intercal$, the ***gradient operator*** is given by

$$\nabla = \left[\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3}\right]^\intercal$$

REVIEW OF THE CHAIN RULE

The gradient operator ∇ (back to chain rule):

- What happens if we replace the scalar parameter $t \in \mathbb{R}$ from before with new coordinates $x \in \mathbb{R}^m$ *(a vector of coordinates)*?
- Now we compute the *gradient* of a *scalar function* f with respect to all coordinates x and write $f(u(x), v(x))$. To do so we replace the d/dt operator from before with the gradient ∇ operator:

$$\nabla_x f = \frac{\partial f}{\partial u}\nabla u + \frac{\partial f}{\partial v}\nabla v,$$

  where ∇_x on the left side emphasizes that the gradient is ***with respect to*** x.
- For $u = u(x)$ and $v = v(x)$ it is clear that the only possible gradient is with respect to x.

REVIEW OF THE CHAIN RULE

The gradient operator ∇ (back to chain rule):

***Single independent coordinate** t (f is a scalar function)*

$$\frac{df}{dt} = \frac{\partial f}{\partial u}\frac{du}{dt} + \frac{\partial f}{\partial v}\frac{dv}{dt}$$

***Vector of independent coordinates** x (f is a scalar function)*

$$\nabla_x f = \frac{\partial f}{\partial u}\nabla u + \frac{\partial f}{\partial v}\nabla v$$

The chain rule still holds, all we did is *replace the single coordinate t with an m-dimensional vector of coordinates x*. This required us to replace the *differential operator d/dt* with the *differential vector operator ∇*.

REVIEW OF THE CHAIN RULE

The gradient operator ∇ (back to chain rule):

$$\nabla_x f = \frac{\partial f}{\partial u}\nabla u + \frac{\partial f}{\partial v}\nabla v$$

Example:

- Let $x = [x_1, x_2]^\intercal \in \mathbb{R}^2$ *(two independent coordinates)*:
  - $u = u(x) = x_1 x_2$
  - $v = v(x) = x_1 + x_2$
- The function is given by: $f(u, v) = \sin(u) - \cos(v)$

$$\nabla u = \begin{bmatrix} x_2 \\ x_1 \end{bmatrix}, \nabla v = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \Rightarrow \nabla_x f = \cos(x_1 x_2)\begin{bmatrix} x_2 \\ x_1 \end{bmatrix} + \sin(x_1 + x_2)\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

REVIEW OF THE CHAIN RULE

The (almost) general chain rule:

- We now further generalize to not only account for $u = u(x)$ and $v = v(x)$ but *many functions* $y(x) = [y_1(x), \ldots, y_n(x)]^\intercal$ where all y_i take arguments $x \in \mathbb{R}^m$.
- The function $f = f(y(x))$ is a *scalar function* (therefore "*almost*" general chain rule) of n other functions y_i, each themselves a function of m variables. The gradient of f can then be written as the chain rule of n partial terms:

$$\nabla_x f = \sum_{i=1}^{n} \frac{\partial f}{\partial y_i} \nabla y_i(x)$$

- ***Example on previous slide:*** $m = 2$ and $n = 2 \rightarrow y_1 = u = x_1 x_2$ and $y_2 = v = x_1 + x_2$.

EVALUATION (FORWARD) TRACE OF A FUNCTION

- After the chain rule discussion above, let us apply the notation introduced and look at the evaluation trace of a scalar function $f(x)$ with a single argument $x \in \mathbb{R}$ ($m = 1$).
- Consider again the same function from the previous lecture:

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

- We would like to ***evaluate the function*** at an arbitrary point x_1. Let us define this point to be $x_1 = \frac{\pi}{16}$.

EVALUATION (FORWARD) TRACE OF A FUNCTION

The evaluation of $f(x_1)$ involves a *partial ordering* of the operations associated with the function f.

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

- ***For example:*** before we can evaluate $\sin(4x)$ we must evaluate the *intermediate* result $4x$ and before we can evaluate the exponential function we must evaluate the intermediate result $-2\left(\sin(4x)\right)^2$.
- The evaluation trace introduces *intermediate results* v_j *for* $j = 1, 2, \ldots$ *of elementary binary operations like multiplying two numbers together or unary operations like computing* $\sin(v_j)$.
- $\rightarrow$ we will use the notation v_1 for the first intermediate result, v_2 for the second and so on.

EVALUATION (FORWARD) TRACE OF A FUNCTION

A word on notation: the coordinates $x = [x_1, \ldots, x_m]^\mathsf{T}$ that is $x \in \mathbb{R}^m$ are called ***independent*** variables, whereas the *intermediate results* v_j are ***dependent*** variables, *they depend on* x.

We further *define the independent variables* as $v_{k-m} = x_k$ for $k = 1, 2, \ldots, m$ in the following evaluation trace.
For example: if $x \in \mathbb{R}^2$ then we would use the notation $v_{-1} = x_1$ and $v_0 = x_2$, whereas v_1 is the first intermediate result of the evaluation trace, v_2 the second and so on.

Recall: $f(x_1) = x_1 - \exp\left(-2\left(\sin(4x_1)\right)^2\right)$ and we are interested in the value of $f\left(x_1 = \frac{\pi}{16}\right)$:

EVALUATION (FORWARD) TRACE OF A FUNCTION

Recall: $f(x_1) = x_1 - \exp\left(-2\left(\sin(4x_1)\right)^2\right)$ and we are interested in the value of $f\left(x_1 = \frac{\pi}{16}\right)$:

Intermediate	Elementary Operation	Numerical value
$v_0 = x_1$	$\frac{\pi}{16}$	1.963495e-01
v_1	$4v_0$	7.853982e-01
v_2	$\sin(v_1)$	7.071068e-01
v_3	v_2^2	5.000000e-01
v_4	$-2v_3$	-1.000000e+00
v_5	$\exp(v_4)$	3.678794e-01
v_6	$-v_5$	-3.678794e-01
$v_7 = f(x_1)$	$v_0 + v_6$	-1.715299e-01

Input variables (*independent* variables)

Intermediate variables (*dependent* variables, $v_j = v_j(x)$)

COMPUTATIONAL (FORWARD) GRAPH

We can think of each intermediate result v_j as a *node in a graph*. By doing so, we can get a visual interpretation of the partial ordering of *elementary operations* in $f(x) = x - \exp\left(-2\left(\sin(4x)\right)^2\right)$:

$$\begin{tikzcd}
x_1 \arrow[r] & \boxed{v_0} \arrow[r, "\times"'] \arrow[rrrrrrr, bend left] & \boxed{v_1} \arrow[r, "\sin(\cdot)"] & \boxed{v_2} \arrow[r, "\mathrm{pow}(\cdot)"] & \boxed{v_3} \arrow[r, "\times"'] & \boxed{v_4} \arrow[r, "e^{(\cdot)}"] & \boxed{v_5} \arrow[r, "\times"'] & \boxed{v_6} \arrow[r, "+"] & \boxed{v_7} \arrow[r] & f(x_1) \\
 & \boxed{4} \arrow[ur] & & \boxed{2} \arrow[ur] & \boxed{-2} \arrow[ur] & & \boxed{-1} \arrow[ur] & & &
\end{tikzcd}$$

COMPUTATIONAL (FORWARD) GRAPH

The ***first key observation*** is that we worked ***from the inside out*** when developing the forward evaluation trace. We started from the value we want to evaluate $x_1 = \frac{\pi}{16}$ and worked from the inside outwards to the actual function value $f(x_1)$. The ***second key observation*** is that in each evaluation step, we only carried out ***elementary operations between intermediate results*** v_j.

(Later when we look at the reverse mode we will observe that this mode works in the opposite direction)

COMPUTING THE DERIVATIVE IN FORWARD TRACE

We are half-way through the forward mode of automatic differentiation:

- We have identified a partial ordering of elementary operations when evaluating an arbitrary function f.
- By breaking down the problem into smaller parts (elementary functions), we have computed intermediate results v_j for $j = 1, 2, \ldots$ with each $v_j = v_j(x)$ evaluated at point $x = x_1$.
- We have associated each v_j to a *node in a graph* for a visualization of the partial ordering. (Try to think about that in terms of a *data structures* → for example a ***tree***.)
- We did not compute any derivative so far.

COMPUTING THE DERIVATIVE IN FORWARD TRACE

Let us now return to the gradient ∇:

In the forward mode of automatic differentiation, we evaluate and carry forward a *directional derivative* of each intermediate variable v_j in a given direction $p \in \mathbb{R}^m$, *simultaneously* with the evaluation of v_j itself. (The latter is what we just did above.)

What does "direction" mean:

- Recall the linearization of the Euler equations (lecture 10): the ***direction*** was the one that gave the *best linear approximation*.
- Heat flows from ***hot to cold***: the "direction" is *dictated* by the *second law of thermodynamics* and given by the temperature gradient $\rightarrow -\nabla T$.
- In forward mode AD: the ***direction*** is the one of a particular ***derivative*** we are interested in. It can be *any linear combination of derivatives*. The direction here is a *parameter* that is chosen by you (not a physical law).

COMPUTING THE DERIVATIVE IN FORWARD TRACE

Directional derivative:

Let us ***define*** the gradient operator in a slightly different way than we did before. Instead of working with the *gradient vector*, we ***project*** it in the ***direction of a vector*** p (***the seed vector***):

$$\boxed{D_p y_i \overset{\text{def}}{=} (\nabla y_i)^\top p = \sum_{j=1}^{m} \frac{\partial y_i}{\partial x_j} p_j .}$$

Is the quantity $D_p y_i$ a ***vector*** or a ***scalar***?

COMPUTING THE DERIVATIVE IN FORWARD TRACE

$$D_p y_i \overset{\text{def}}{=} (\nabla y_i)^\top p = \sum_{j=1}^{m} \frac{\partial y_i}{\partial x_j} p_j.$$

We now return to our example function from before:

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

evaluated at the point $x = x_1 = \frac{\pi}{16}$. Recall that $x \in \mathbb{R}$ is one-dimensional ($m = 1$):

We can only choose one possible direction in this case!

Most natural choice is $p = 1$ and simplify:

$$D_p y_i = \frac{\partial y_i}{\partial x_1}$$

COMPUTING THE DERIVATIVE IN FORWARD TRACE

In forward mode AD, the derivative we compute is $D_p y_i$.

The y_i in $D_p y_i$ are *just functions* that depend on x (the independent variables).
Let us define these functions as follows:

$$y_i = v_{i-m} \quad \text{for } i = 1, 2, \ldots, n,$$

where the v_j are the variables we used in the evaluation trace before. The number n is the sum of *independent* variables and *intermediate* variables v_j for which $j > 0$.

In the forward trace example from earlier, we have $n = 8$ (1 independent variable and 7 intermediate variables). We therefore get $\rightarrow y_1 = v_0, y_2 = v_1$ up to $y_8 = v_7$. ***Note that y_i for $i = 1, 2, \ldots, m$ are exactly the m independent variables.***

COMPUTING THE DERIVATIVE IN FORWARD TRACE

Before we compute the forward trace including the derivatives of intermediate variables v_j, let us pick a few examples and look at them individually. Here is where you see how the *chain rule* enters forward mode AD.

Note: from now on we no longer write ∇_x to indicate that the differentials are with respect to the independent coordinates x (think of them as input variables).
$\rightarrow$ ***we assume this is always the case.***

COMPUTING THE DERIVATIVE IN FORWARD TRACE

What is the value of $D_p v_0$ in the evaluation trace of before?

- We know that $v_0 = x_1$.
- We have chosen our direction parameter to be $p = 1$.
- We are dealing with a 1D problem → the gradient operator simply is $\nabla := \partial / \partial x_1$.
- Putting this together we find:

$$\boxed{D_p v_0 = (\nabla v_0)^\intercal p = \frac{\partial x_1}{\partial x_1} \cdot 1 = 1}$$

COMPUTING THE DERIVATIVE IN FORWARD TRACE

What is the value of $D_p v_2$ in the evaluation trace of before?

- We know that $v_2 = v_2(v_1) = \sin(v_1)$.
- We have chosen our direction parameter to be $p = 1$.
- Because all v_j are functions of the *independent coordinates x*, ***the chain rule must be applied here:***

$$\nabla v_2 = \frac{\partial v_2}{\partial v_1} \nabla v_1 = \cos(v_1) \nabla v_1$$

- Putting this together we find:

$$\boxed{D_p v_2 = (\nabla v_2)^\intercal p = \cos(v_1)(\nabla v_1)^\intercal p = \cos(v_1) D_p v_1}$$

- ***Observe:*** we can compute the derivative of v_j with knowledge of v_i and $D_p v_i$ for $i < j$!

COMPUTING THE DERIVATIVE IN FORWARD TRACE

What is the value of $D_p v_7$ in the evaluation trace of before?

- We know that $v_7 = v_7(v_0, v_6) = v_0 + v_6$.
- Nothing new here ***except*** that we further know $v_7 = f(x_1)$ such that $\nabla v_7 = \nabla f$ and the directional derivative $D_p v_7 = D_p f$ is *exactly the derivative we are after, evaluated at x_1*:

$$\nabla v_7 = \frac{\partial v_7}{\partial v_0} \nabla v_0 + \frac{\partial v_7}{\partial v_6} \nabla v_6 .$$

- Projection in direction of p yields:

$$\boxed{D_p f = D_p v_7 = D_p v_0 + D_p v_6}$$

(***Note:*** the partial derivatives $\partial v_7 / \partial v_0$ and $\partial v_7 / \partial v_6$ are both equal to 1 in this case)

COMPUTING THE DERIVATIVE IN FORWARD TRACE

We now repeat the computation of the forward trace for our test function $f(x)$. What we did earlier is called the *forward primal trace*, we extend it this time with the *forward tangent trace* which corresponds to the *derivatives of the intermediate variables*.

In forward mode of automatic differentiation, we evaluate and *carry forward a directional derivative* $D_p v_j$ of each intermediate variable v_j in a given direction $p \in \mathbb{R}^m$, *simultaneously* with the evaluation of v_j itself.

Recall: $f(x_1) = x_1 - \exp\left(-2\left(\sin(4x_1)\right)^2\right)$ and we are interested in the value of $\left.\frac{\partial f}{\partial x}\right|_{x_1=\pi/16}$:

AUTOMATIC DIFFERENTIATION: FORWARD MODE

Recall: $f(x_1) = x_1 - \exp\left(-2\left(\sin(4x_1)\right)^2\right)$ and we are interested in the value of $\left.\frac{\partial f}{\partial x}\right|_{x_1=\pi/16}$:

Forward primal trace	Forward tangent trace	Numerical value: v_j; $D_p v_j$
$v_0 = x_1 = \frac{\pi}{16}$	$D_p v_0 = 1$	`1.963495e-01; 1.000000e+00`
$v_1 = 4v_0$	$D_p v_1 = 4D_p v_0$	`7.853982e-01; 4.000000e+00`
$v_2 = \sin(v_1)$	$D_p v_2 = \cos(v_1) D_p v_1$	`7.071068e-01; 2.828427e+00`
$v_3 = v_2^2$	$D_p v_3 = 2v_2 D_p v_2$	`5.000000e-01; 4.000000e+00`
$v_4 = -2v_3$	$D_p v_4 = -2D_p v_3$	`-1.000000e+00;-8.000000e+00`
$v_5 = \exp(v_4)$	$D_p v_5 = \exp(v_4) D_p v_4$	`3.678794e-01;-2.943036e+00`
$v_6 = -v_5$	$D_p v_6 = -D_p v_5$	`-3.678794e-01; 2.943036e+00`
$v_7 = f(x_1) = v_0 + v_6$	$D_p v_7 = \left.\frac{\partial f}{\partial x}\right\|_{x_1=\pi/16} = D_p v_0 + D_p v_6$	`-1.715299e-01; 3.943036e+00`

Input variables (*independent* variables)

Intermediate variables (*dependent* variables, $v_j = v_j(x)$)

AUTOMATIC DIFFERENTIATION: FORWARD MODE

Recall: $f(x_1) = x_1 - \exp\left(-2\left(\sin(4x_1)\right)^2\right)$ and we are interested in the value of $\left.\frac{\partial f}{\partial x}\right|_{x_1=\pi/16}$:

Forward primal trace	Forward tangent trace	Numerical value: v_j; $D_p v_j$
$v_0 = x_1 = \frac{\pi}{16}$	$D_p v_0 = 1$	`1.963495e-01; 1.000000e+00`
$v_7 = f(x_1) = v_0 + v_6$	$D_p v_7 = \left.\frac{\partial f}{\partial x}\right\|_{x_1=\pi/16} = D_p v_0 + D_p v_6$	`-1.715299e-01; 3.943036e+00`

We have computed the derivative before on paper and with `sympy`. You are encouraged to check that we indeed compute the correct result.

AUTOMATIC DIFFERENTIATION: FORWARD MODE

That is all there is to forward mode AD!

The key observations are the following:

- We have broken down the evaluation of an arbitrary function $f(x)$ into smaller pieces, each only consists of *elementary* operations like addition, multiplication, division, subtraction, exponentiation, trigonometric functions and so on.
- Forward mode works from the inside out.
- We have computed a *primal trace* of intermediate variables v_j and a *tangent trace* of their directional derivatives $D_p v_j$ both *simultaneously* in the same step.
- Since we only work with *elementary functions*, we know their derivatives and computing $D_p v_j$ is trivial!

RECAP

Automatic Differentiation: Forward Mode (basics)

- Example of Newton's method and numerical schemes to approximate derivatives.
- Evaluation trace
- The computational graph
- Computing derivatives of one variable using the forward mode

Further reading:

- 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 or find it in the class repository)
- Griewank, A. and Walther, A., *Evaluating derivatives: principles and techniques of algorithmic differentiation*, SIAM 2008, Vol. 105
- Nocedal, J. and Wright, S., *Numerical Optimization*, Springer 2006, 2nd Edition
- Finite difference method: https://en.wikipedia.org/wiki/Finite_difference_method