AUTOMATIC DIFFERENTIATION

PHILIPP MÜLLER

UNIVERSITY OF ZURICH

MOTIVATION

Derivatives are omnipresent in numerical algorithms

1st order derivatives

- Solving non-linear equations
  - E.g., by Newton’s method
- (Un-)constrained optimization
  - Gradient-Based optimization algorithms
  - Especially difficult for high dimensional variables, i.e., objective function $f: R^n \rightarrow R$
    - Structural sparsity can be key

2^{nd} order derivatives

- (Un-)constrained optimization

Higher order derivatives

- Higher-order differential equations

MOTIVATION

Suppose we want to solve the unconstrained optimization problem

$$\min_x f(x)$$

with $f: R \to R$ and $x \in R$.

Gradient-based optimization requires the gradient

$$\frac{\partial f}{\partial x}$$

FINITE DIFFERENCES

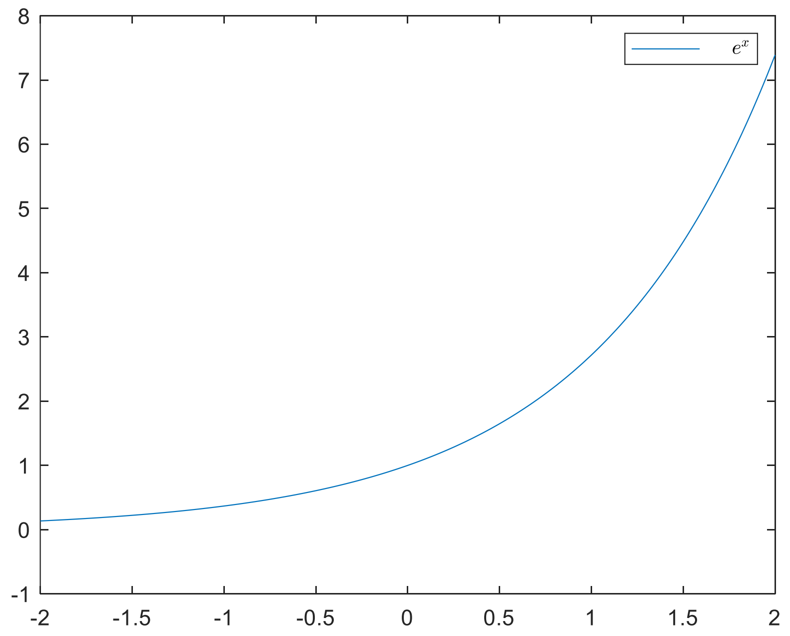
Recall: The *Taylor series expansion* of a real-valued function $f \in \mathcal{C}^n, n \geq 2$, around x and evaluated at a reads

$$f(a) = \sum_{|\alpha| \leq n} \frac{1}{\alpha!} \partial^\alpha f(x)(a-x)^\alpha + R$$

$$= f(x) + \frac{\partial f}{\partial x}(x)(a-x) + \frac{1}{2!}\frac{\partial^2 f}{\partial x^2}(x)(a-x)^2$$

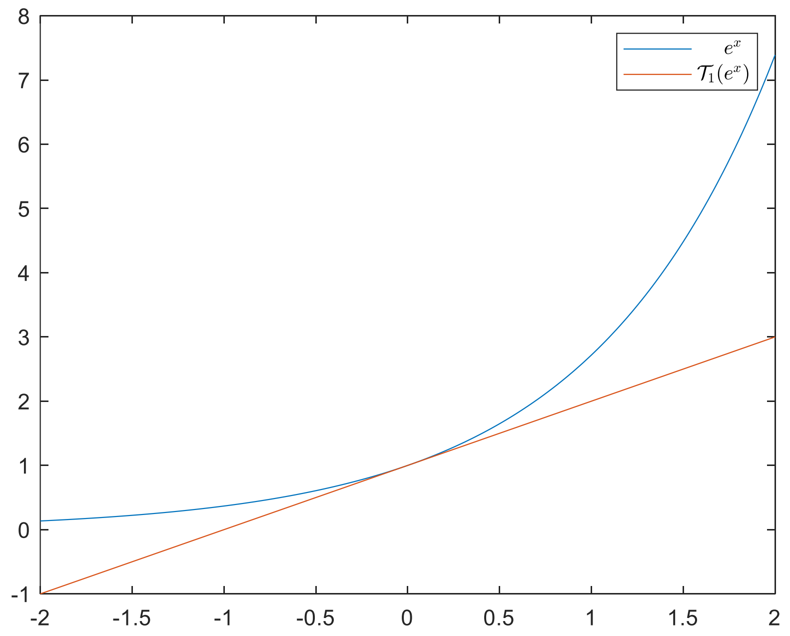
$$+ \mathrm{O}(|\mathrm{a} - \mathrm{x}|^3)$$

TAYLOR APPROXIMATION OF e^x AROUND 0



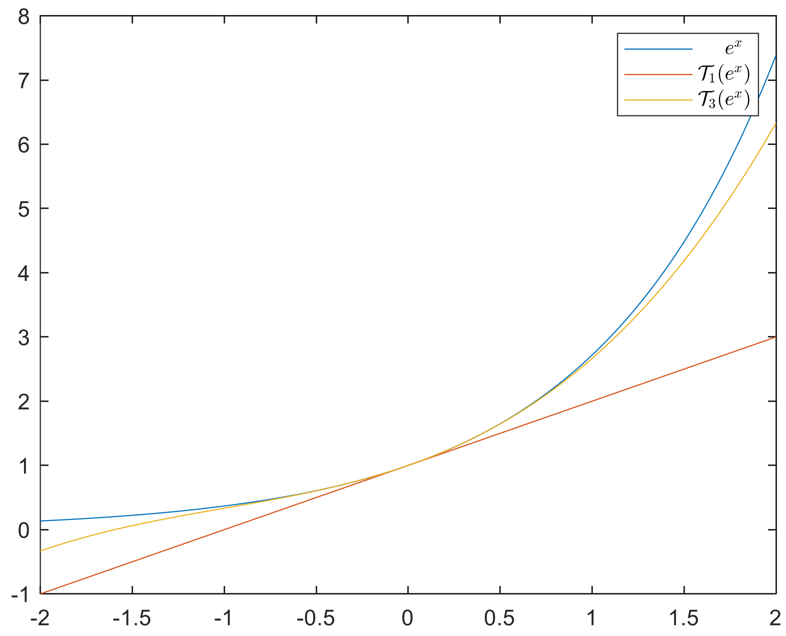
$$f(x) = e^x$$

TAYLOR APPROXIMATION OF e^x AROUND 0



$$\mathcal{T}f(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots$$

TAYLOR APPROXIMATION OF e^x AROUND 0



$$\mathcal{T}f(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \ \ldots$$

FINITE DIFFERENCES

Recall (Taylor Series):

$$f(a) \approx f(x) + \frac{\partial f}{\partial x}(x)(a-x) + \frac{1}{2!}\frac{\partial^2 f}{\partial x^2}(x)(a-x)^2$$

Truncate the Taylor series and set a $= x + h$ with a small h yields:

$$f(x+h) = f(x) + \frac{\partial f}{\partial x}(x)(x+h-x) + O(h^2)$$

$$\Leftrightarrow \frac{\partial f}{\partial x}(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

This results in the well-known *forward difference equation:*

$$\frac{\partial f}{\partial x}(x) \approx \frac{f(x+h) - f(x)}{h}$$

WHY WOULD WE NEED ANYTHING ELSE?

We derived $\frac{\partial f}{\partial x}$ by truncating the Taylor series resulting in the error $O(h)$. This truncation error decreases in the step size.

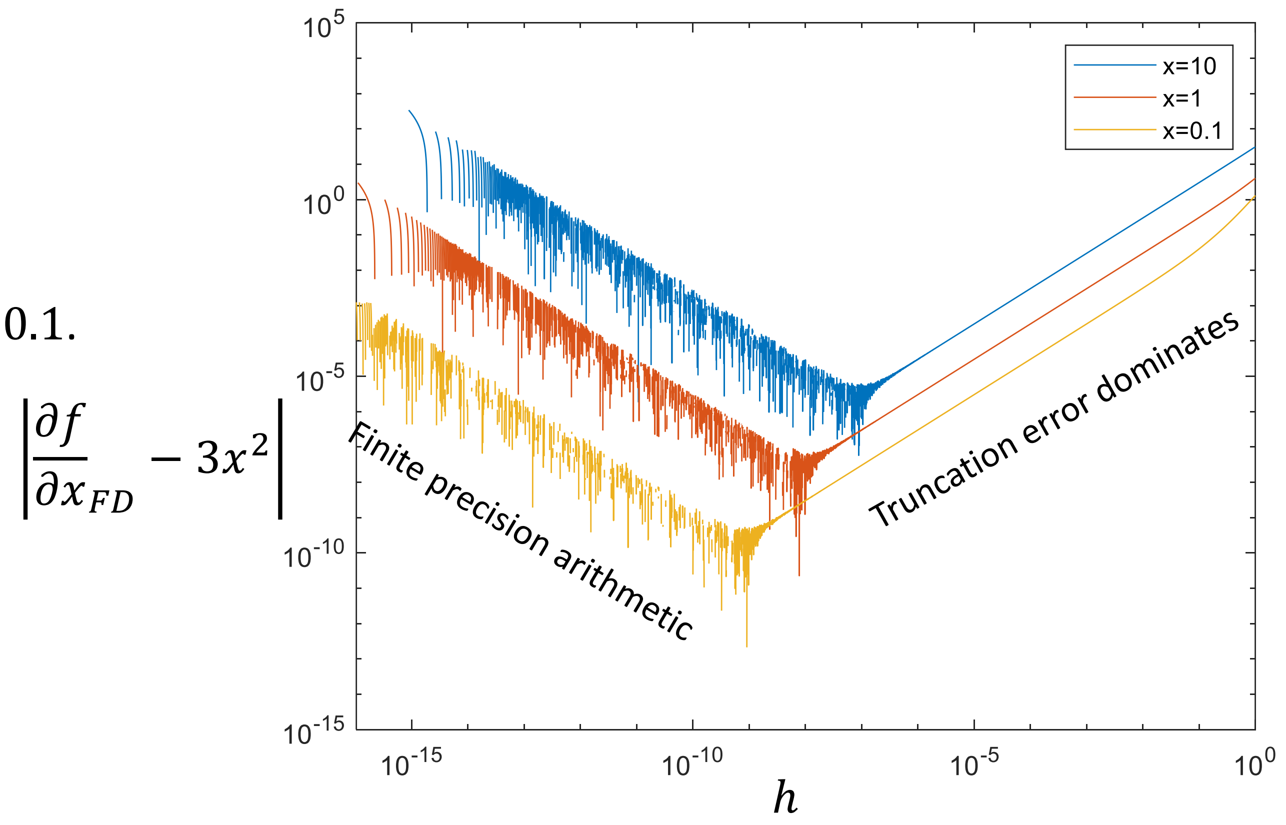
Accurate and efficient approximation of $\frac{\partial f}{\partial x}$ by choosing a very small h, i.e., $\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$?

PROBLEM SOLVED?

Apply forward differences to

$$f(x) = x^3$$

and increase the step size from 10^{-16} to 0.1.



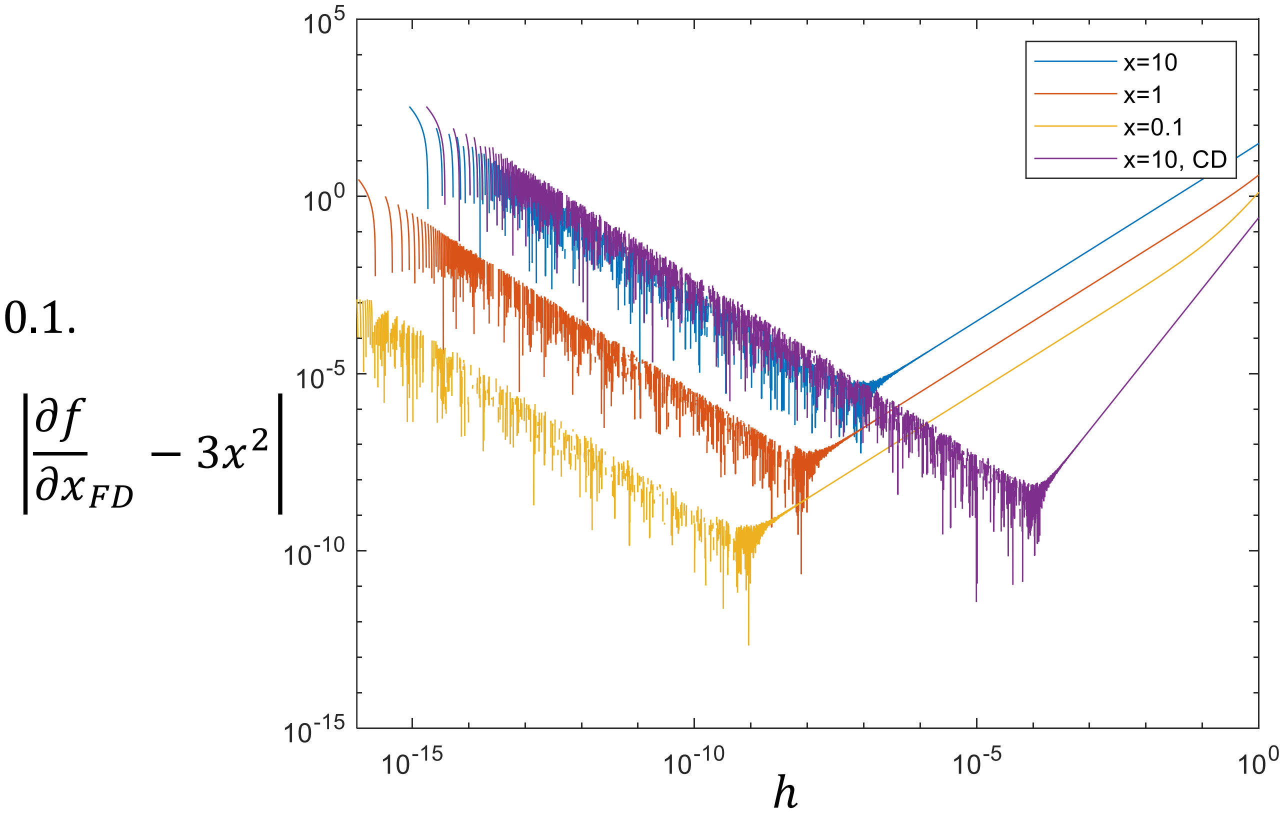
PROBLEM SOLVED?

Apply forward differences to

$$f(x) = x^3$$

and increase the step size from 10^{-16} to 0.1.

CD: $\frac{\partial f}{\partial x} = \frac{f\left(x+\frac{1}{2}h\right)-f\left(x-\frac{1}{2}h\right)}{h}$



NUMERICAL ERRORS IN FINITE PRECISION ARITHMETIC

Rounding error

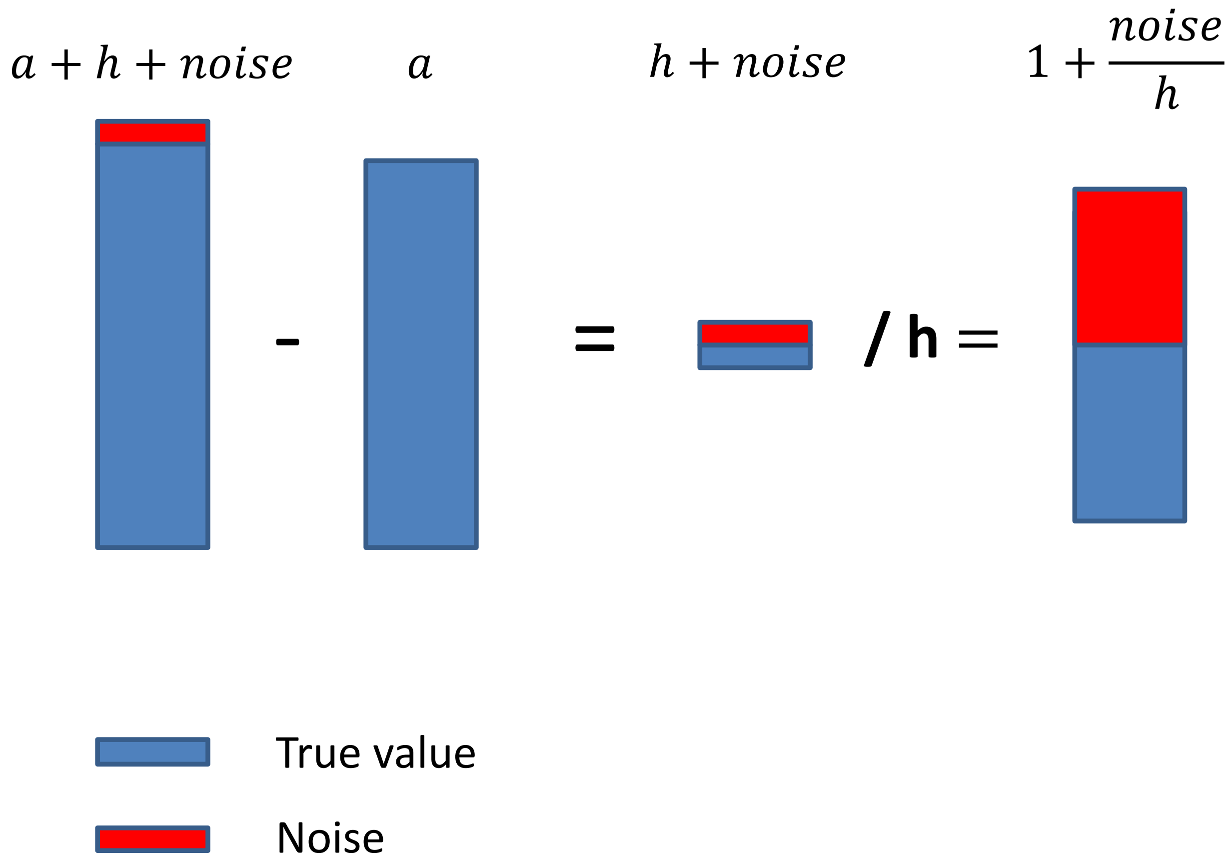
- Intermediate results are rounded
- Any rounding error propagates and amplifies

Truncation error

- Even if an algorithm is converging to the true solution, we are stopping it after some finite time.
- Mitigated by the appropriate convergence criteria as introduced by Ken.

Cancellation error

CANCELLATION ERROR



h = 1E-12

a = 1

b = a + h # add h to a

c = b – a # c should be equal to h

d = c/h # c = h, thus d should = 1

d = 0.999200722162641

FINITE DIFFERENCES FOR $f: R^n \to R$

Let's consider a n-dimensional unconstrained optimization problem

$$\min_x f(x)$$

with $f: R^n \to R$ and $x \in R^n$.

The finite difference quotients resemble ***directional derivatives*** for $f: R^n \to R^m$ and n > 1 ***:***

$$\frac{\partial f}{\partial x_i}(x) = \frac{f(x + h\boldsymbol{e_i}) - f(x)}{h}.$$

The cost of FD scales with n -> O(n) * cost(f).

SCALING

- Let's consider the Rosenbrock function $f: R^n \to R$ as benchmark

$$f(x) = \sum_{i=1}^{n-1} 10 * \left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2 .$$

- The runtimes are averaged across 1000 runs.

n	J_{FD} [s]	f [s]
10	6.7931e-05 (54x)	1.2472e-06
100	6.0959e-04 (332x)	1.8355e-06
1000	1.4839e-02 (1500x)	9.9629e-06
10000	1.3282 (20000x)	6.6663e-05
100000	?	9.0872e-04

AUTOMATIC DIFFERENTIATION

Basic Idea: Every computer program is a composition of differentiable elementary operations as,

- basic arithmetic operations as, e.g., +, -, and *,
- and basic functions as, e.g., sin, cos and tan.

Automatic differentiation can transform the source code of your function into the source code of the gradient.

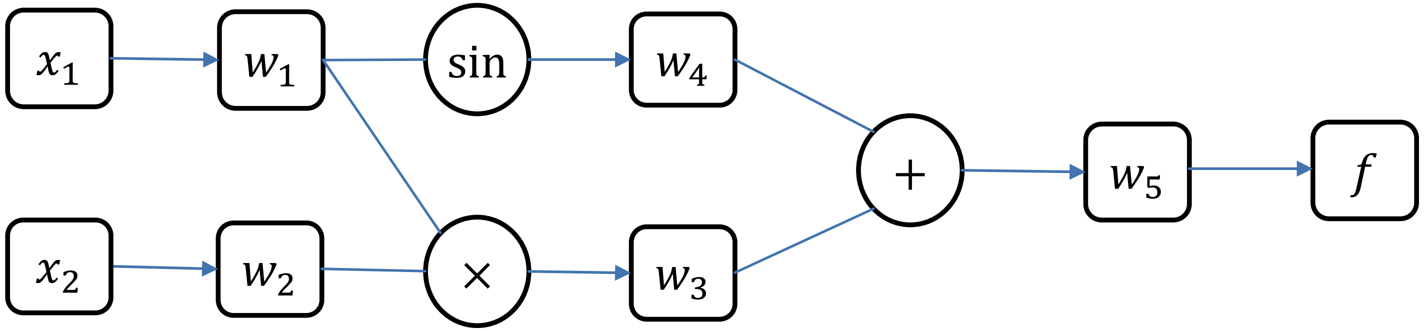
TOY EXAMPLE

Consider the function $f: R^2 \to R$

$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

This function can be discomposed in differentiable elementary operations:

$$\begin{aligned} w_1 &= x_1 \\ w_2 &= x_2 \\ w_3 &= w_1 w_2 \\ w_4 &= \sin(w_1) \\ w_5 &= w_3 + w_4 \\ f &= w_5 \end{aligned}$$



FORWARD MODE

Consider the function $f: R^2 \to R$

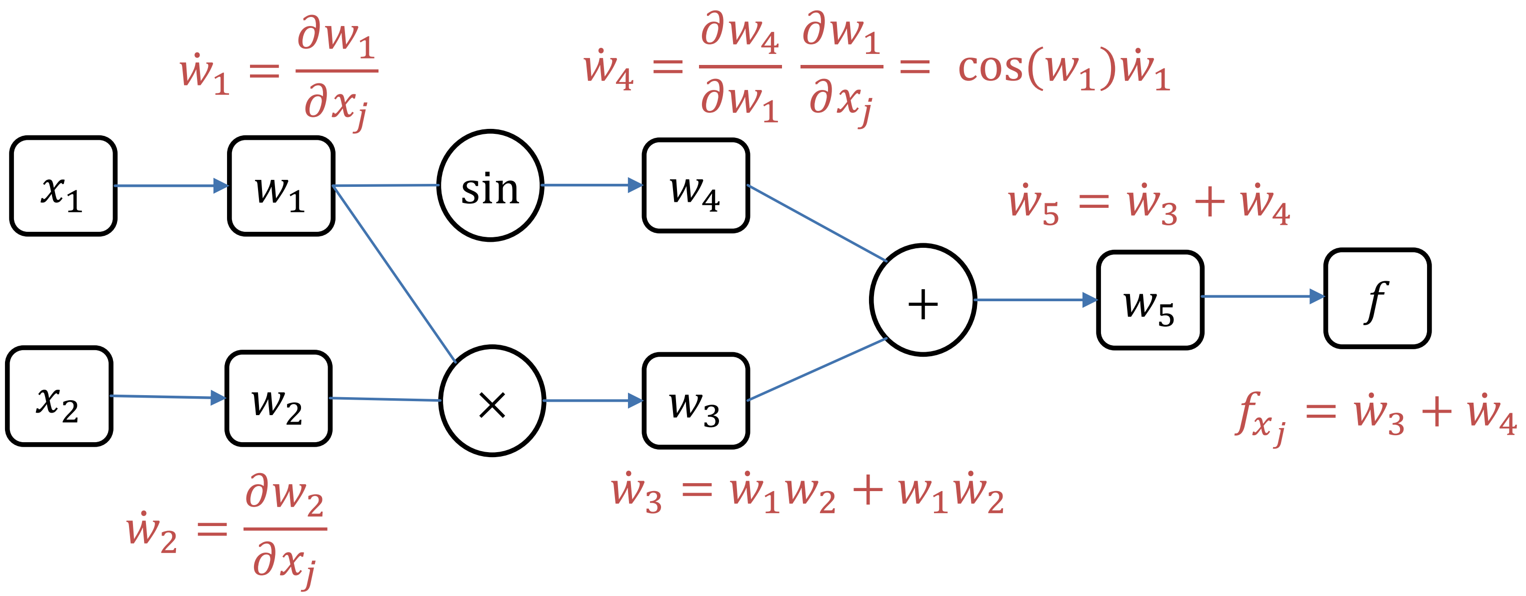
$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

To calculate the Gradient, calculate

$$\frac{\partial f(x_1, x_2)}{\partial x_1}$$

Choose input variable x_1 and calculate the sensitivity of each intermediate value as

$$\dot{w}_i = \frac{\partial w_i}{\partial x_j}$$



FORWARD MODE: EVALUATION

Suppose: $j = 1,\ x_1 = 1,\ x_2 = 2$

Consider the function $f: R^2 \to R$

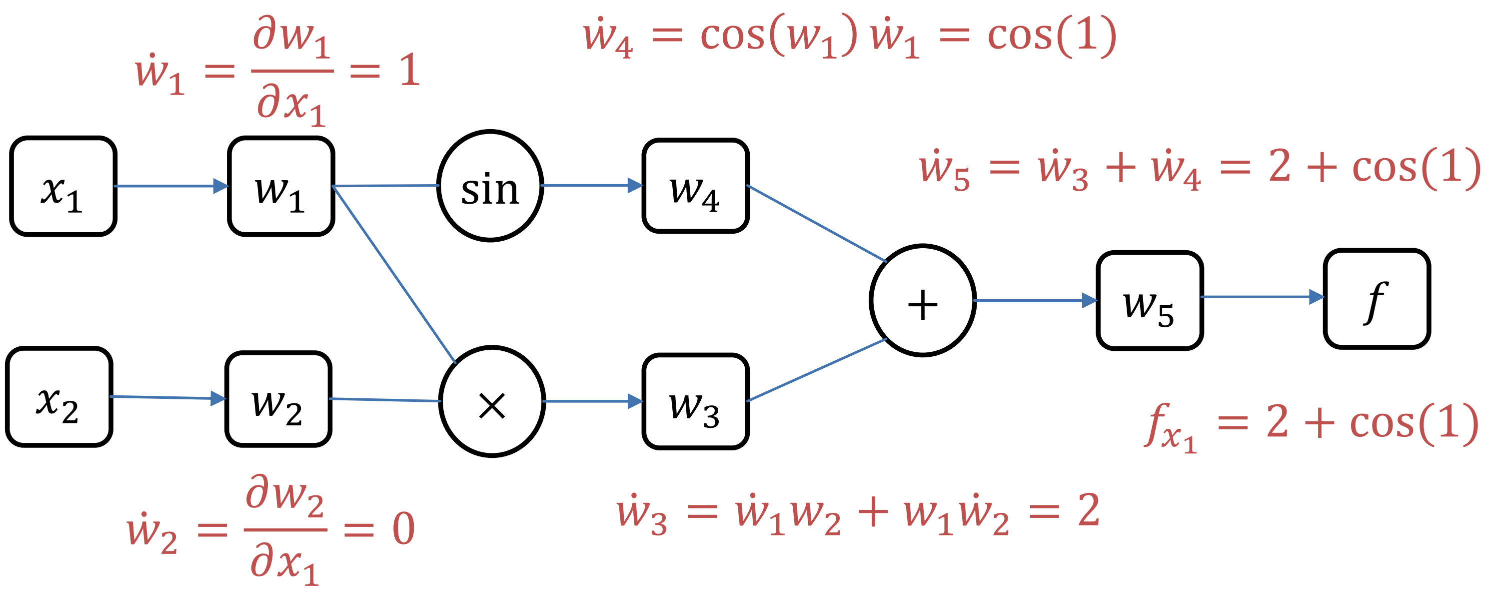
$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

To calculate the Gradient, calculate

$$\frac{\partial f(x_1, x_2)}{\partial x_1}$$

Choose input variable x_1 and calculate the sensitivity of each intermediate value as

$$\dot{w}_i = \frac{\partial w_i}{\partial x_1}$$



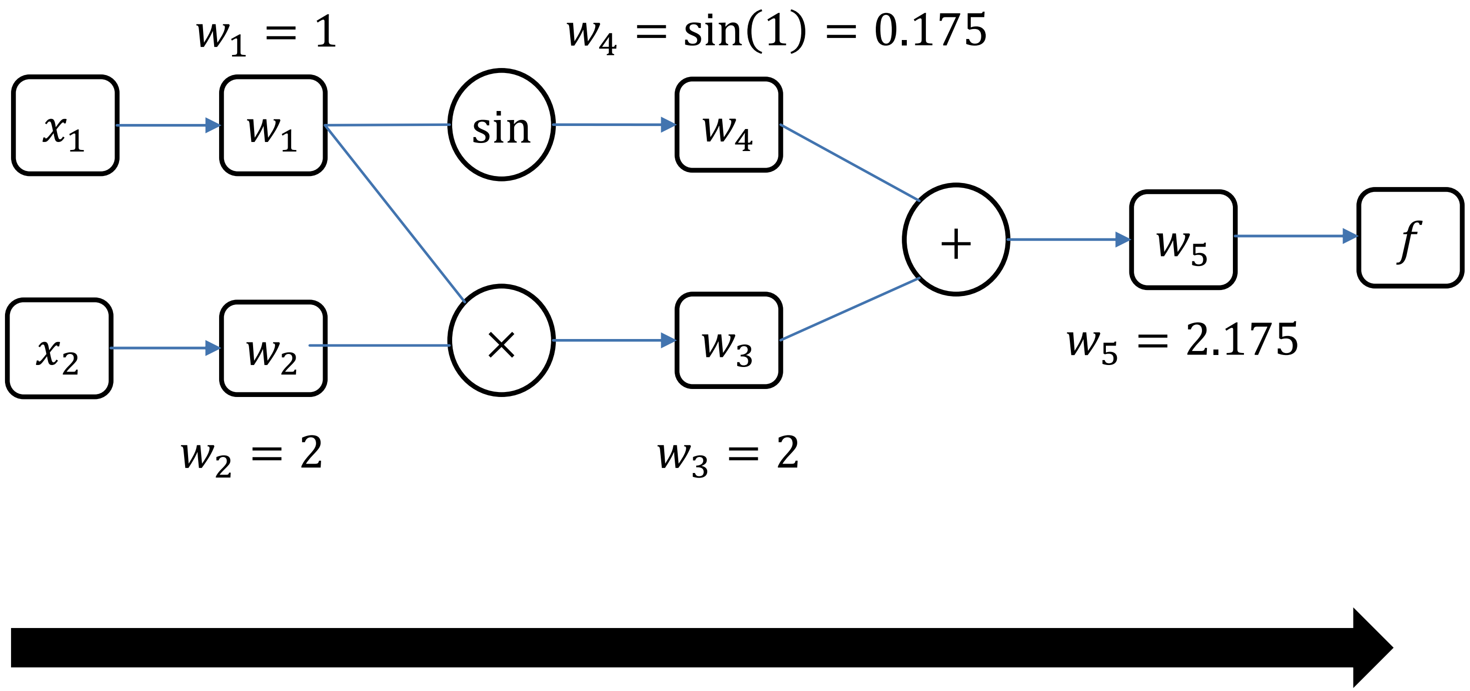
Accurate up to working precision, but still scales **linearly in n**. $Cost(J_f) = n\ c\ Cost(f)$

REVERSE MODE (ADJOINT MODE) – PRIMAL TRACE

Suppose: $x_1 = 1, \; x_2 = 2$

Consider the function $f: R^2 \to R$

$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$



Calculate the sensitivity of the output w.r.t. each intermediate value

$$\bar{w}_i = \frac{\partial f}{\partial w_j}$$

All intermediate values are stored. This leads to a high memory consumption, mitigated by good AD software

REVERSE MODE (ADJOINT MODE)

Consider the function $f: R^2 \rightarrow R$

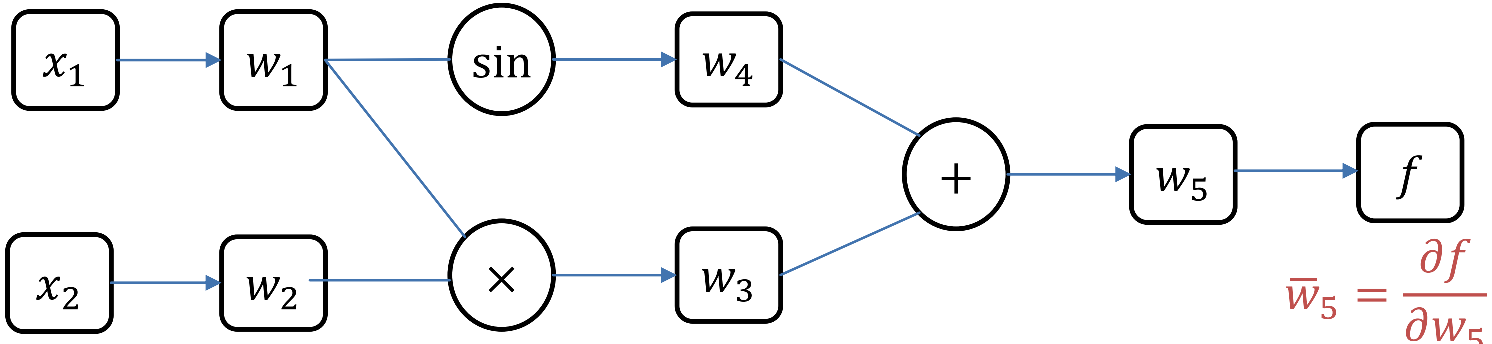
$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

Calculate the sensitivity of the output w.r.t. each intermediate value

$$\bar{w}_i = \frac{\partial f}{\partial w_j}$$

$$\bar{w}_1 = \frac{\partial f}{\partial w_4}\frac{\partial w_4}{\partial w_1} + \frac{\partial f}{\partial w_3}\frac{\partial w_3}{\partial w_1} = \bar{w}_4 \cos w_1 + \bar{w}_3 w_2$$

$$\bar{w}_4 = \frac{\partial f}{\partial w_4} = \frac{\partial f}{\partial w_5}\frac{\partial w_5}{\partial w_4} = \bar{w}_5 \cdot 1$$



$$\bar{w}_5 = \frac{\partial f}{\partial w_5}$$

$$\bar{w}_2 = \frac{\partial f}{\partial w_3}\frac{\partial w_3}{\partial w_2} = \bar{w}_3 w_1$$

$$\bar{w}_3 = \frac{\partial f}{\partial w_5}\frac{\partial w_5}{\partial w_3} = \bar{w}_5 \cdot 1$$

REVERSE MODE (ADJOINT MODE) – DUAL TRACE

Consider the function $f: R^2 \to R$

$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

Calculate the sensitivity of the output w.r.t. each intermediate value

$$\bar{w}_i = \frac{\partial f}{\partial w_j}$$

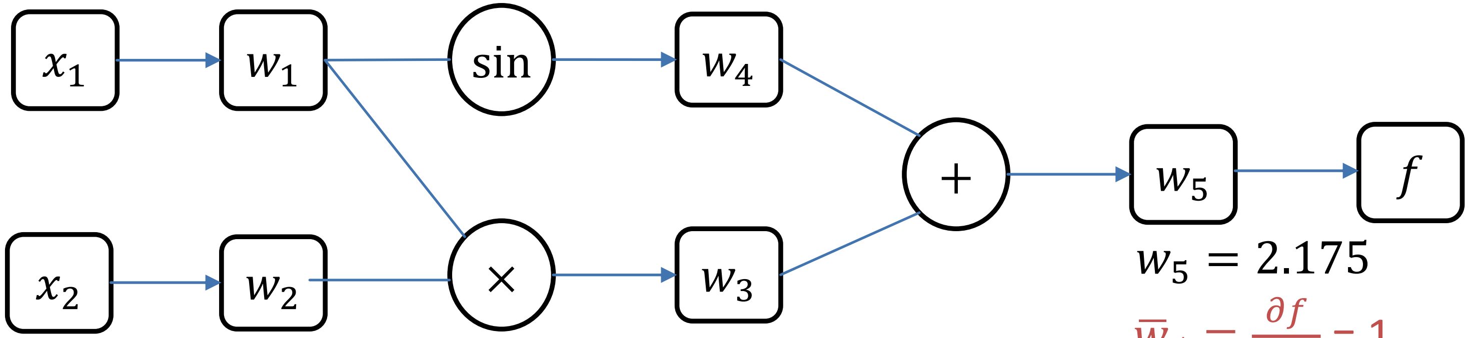
Suppose: $x_1 = 1, \; x_2 = 2$

$w_1 = 1$

$\bar{w}_1 = \bar{w}_4 \cos w_1 + \bar{w}_3 w_2 = \cos 1 + 2$

$w_4 = \sin(1) = 0.175$

$\bar{w}_4 = \bar{w}_5 = 1$



$w_5 = 2.175$

$\bar{w}_5 = \frac{\partial f}{\partial w_5} = 1$

$w_2 = 2$

$\bar{w}_2 = 1$

$w_3 = 2$

$\bar{w}_3 = \bar{w}_5 = 1$

Accurate up to working precision, scales **linearly in m**. $Cost(J_f) = m\, c_2\, Cost(f)$

SUMMARY

Finite differences

- The approximation error decreases as $O(h)$ for forward finite differences. BUT, the error due to the finite precision arithmetic can not be neglected.
- The time required to compute the Jacobian of $f: R^n \rightarrow R^m$ scales with $O(n) * cost(f)$.

AD - Forward mode

- The gradients are accurate up to machine precision.
- The time required to compute the Jacobian of $f: R^n \rightarrow R^m$ scales with $O(n) * cost(f)$

AD - Reverse mode

- The gradients are accurate up to machine precision. The memory requirement may be huge depending on the underlying implementation.
- The time required to compute the Jacobian of $f: R^n \rightarrow R^m$ scales with $O(m) * cost(f)$

AD TOOLS

CasADi

- Available for Python, Matlab, Octave and C++
- Includes interfaces to a lot of free as well as commercial optimizers (as e.g., IPOPT (IP), KNITRO (IP & SQP), WORHP (SQP), SNOPT (SQP))
- **Structural sparsity detection**

ADiMat

- Available for Matlab

PyTorch / Tensorflow

TUTORIAL SESSION

1. Implementation of the Rosenbrock function

$$f(x) = \sum_{i=1}^{n-1} 10 \,*\, \left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2 \,,$$

2. Implementation of the finite difference approximation and reverse mode AD of f(x). Comparison of their runtimes for $n = 10^i$, i = 1, 2, 3, 4, …
3. Optimization of the Rosenbrock function using **fminunc** using
   1. the finite difference approximation of f(x), and
   2. the reverse mode approximation of f(x).

CASADI

- Include the casadi directory in the Matlab path

```
import casadi.*
x_MX = MX.sym('some_name', size_rows, size_columns)          % create symbolic variable
d_rosenbrock_ = jacobian(rosenbrock(x_MX), x_MX)              % differentiate rosenbrock
d_rosenbrock = Function('some_name', {x_MX}, {d_rosenbrock_})  % create callable function
d_rosenbrock(x)
```