Algorithmic Differentiation: A Practical Approach for Engineers

Short Course (Day 1)

Pavanakumar Mohanamuraly

Excepts from “Numerical Optimisation” course by Dr. Jens D Mueller, Queen Mary University of London. (*notes and figures with permission*)

Introduction to AD tools used in the course

TAPENADE

Inria

python™

Github: https://github.com/pavanakumar/IITM_AD_Course

Exercises: `exercises/day1`

Reading Material: `reading_materials/*.pdf`

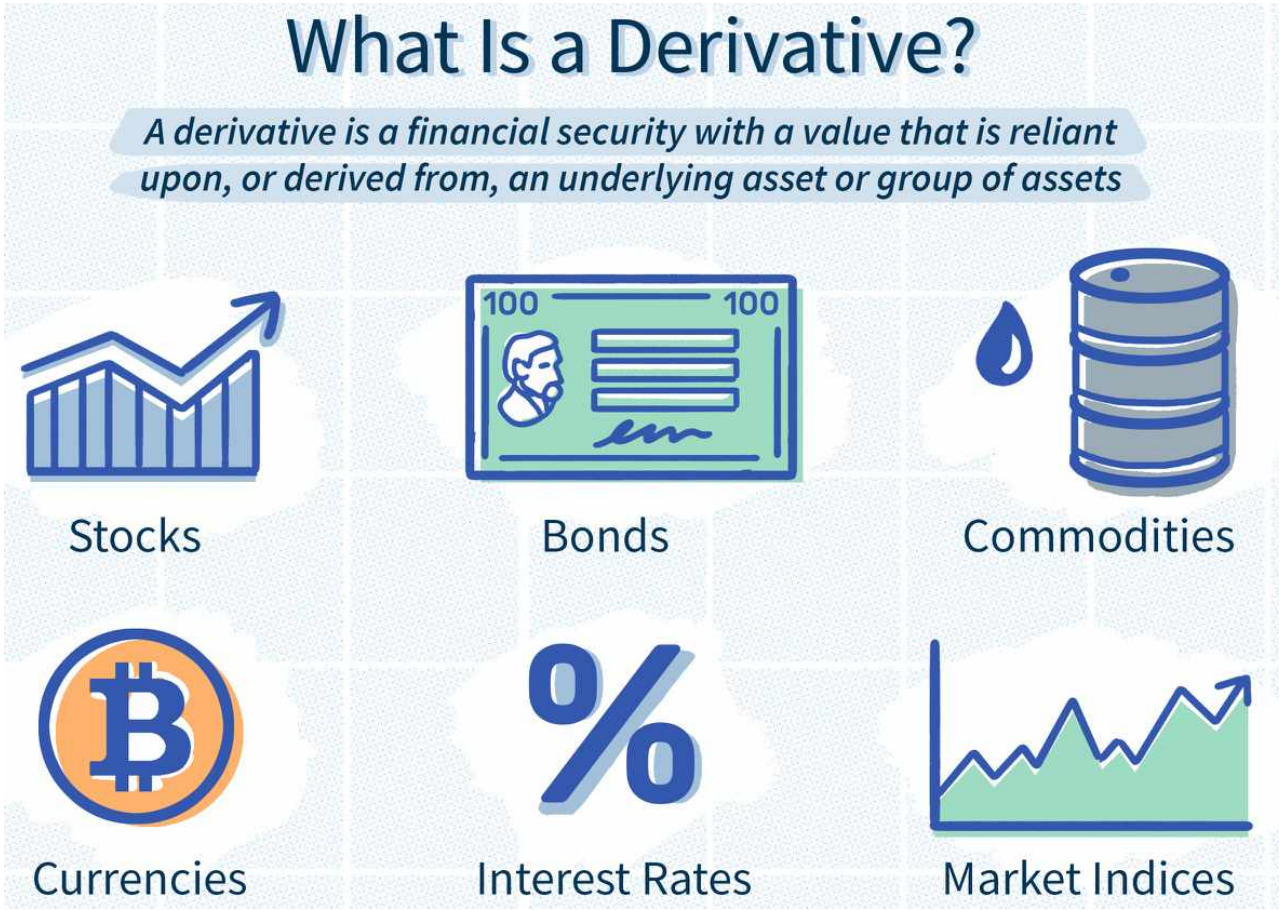
```
git clone https://github.com/pavanakumar/IITM_AD_Course.git
```

Motivation

Course deals with mathematical derivatives & application to optimisation/machine learning

$$\frac{dy}{dx} \qquad \nabla f$$

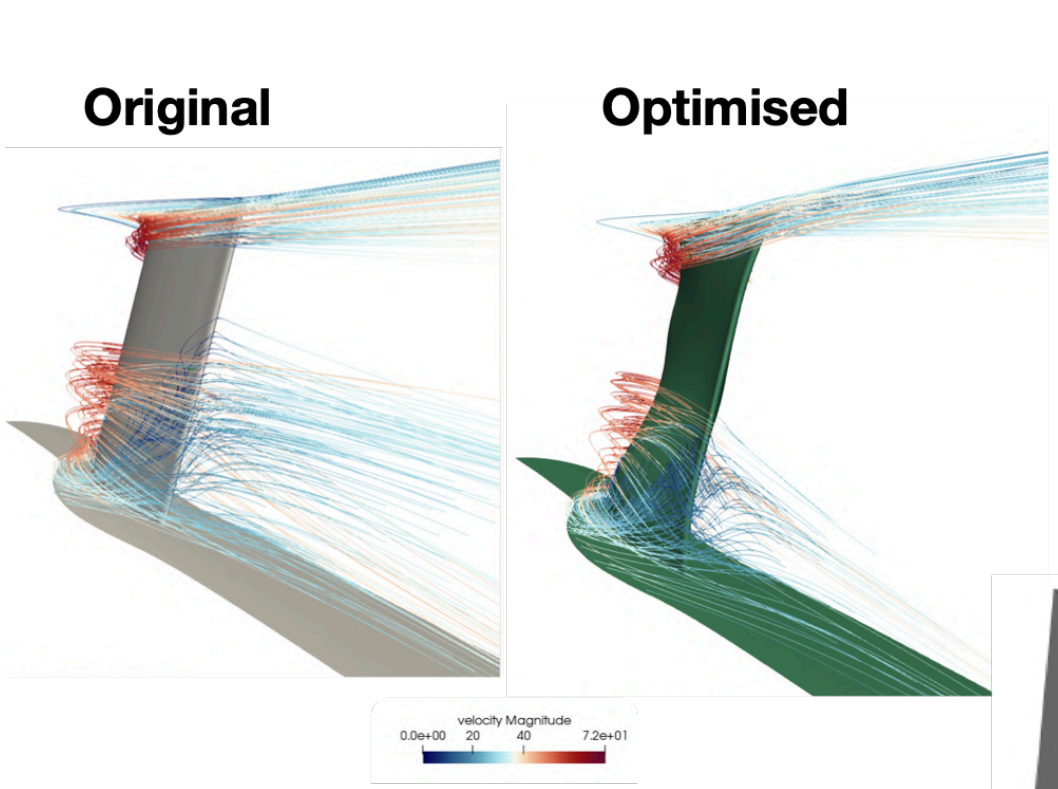
Why do we need derivatives?
Why are they important?

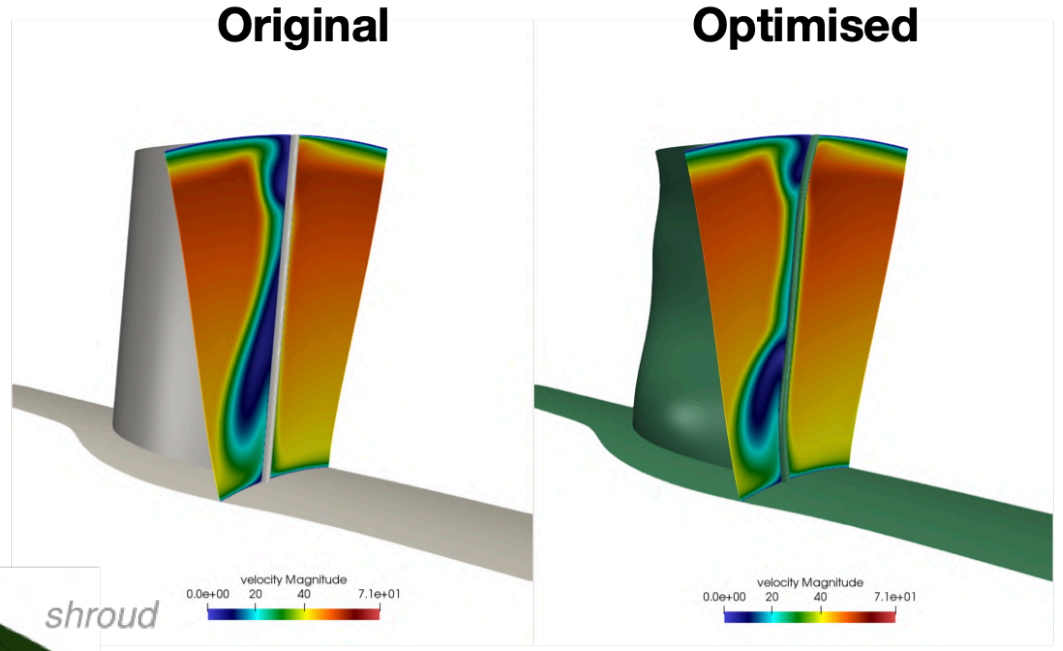


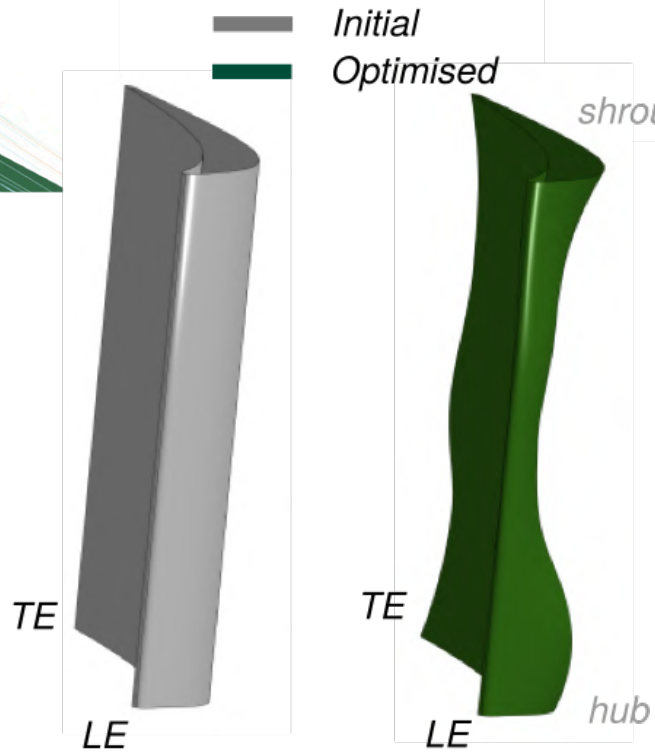
Not about financial derivatives*

**subject to market risks please read offer document carefully !!*

Aerodynamic Shape Optimisation





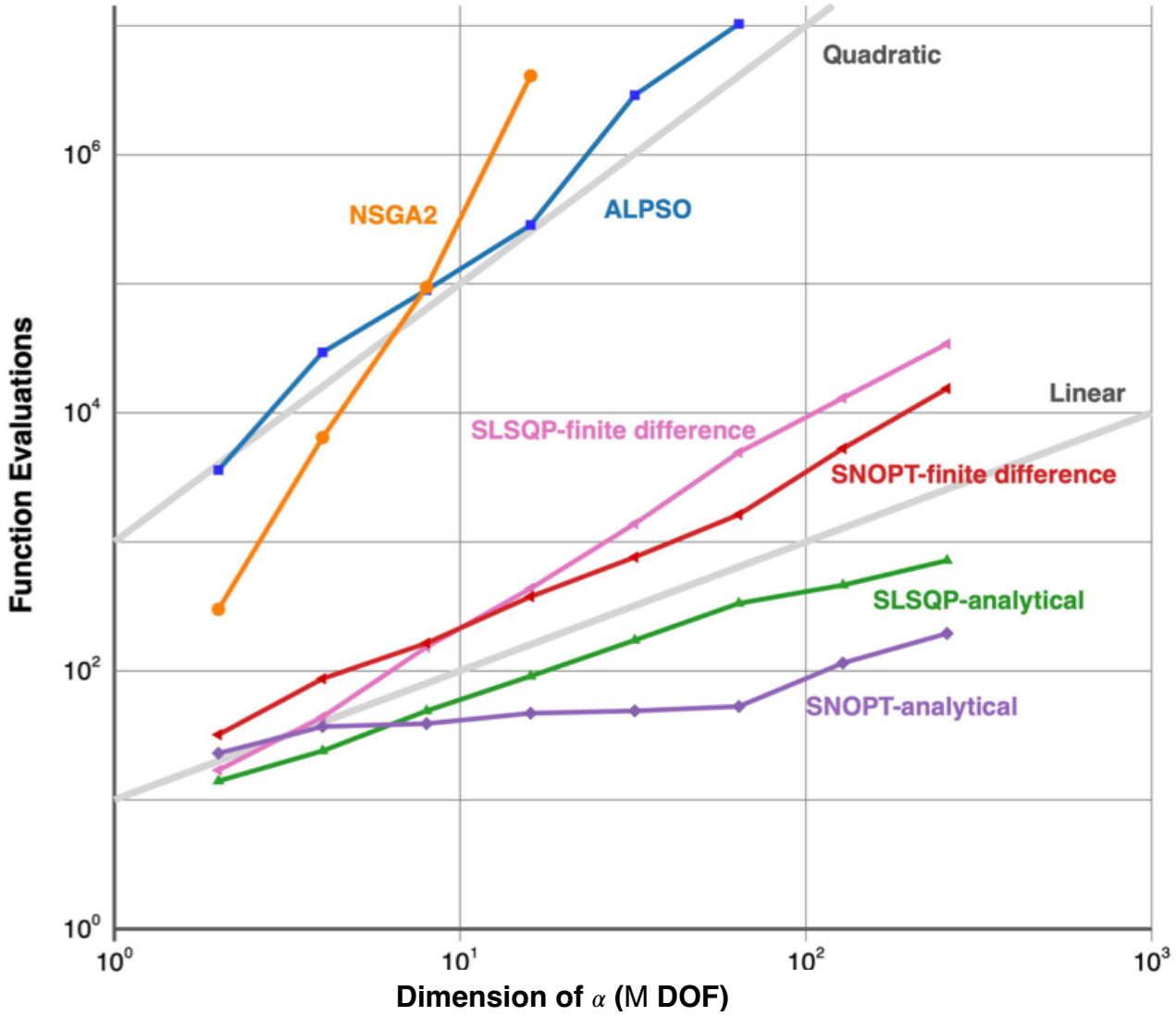


- 192 blade parameters (parametric CAD)
- 4 constraints

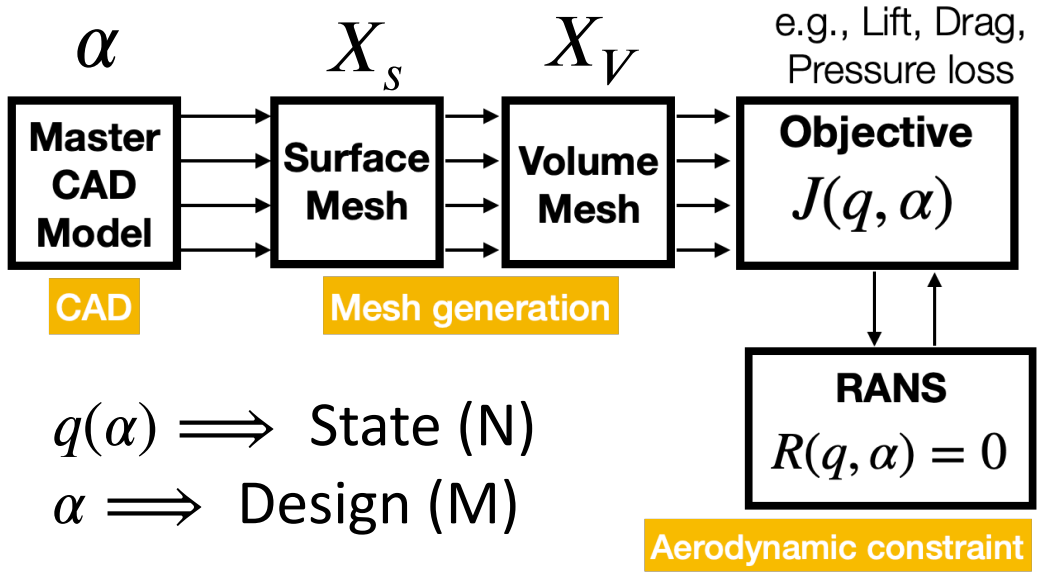
15% lower entropy loss (mass-flow and geometric constraints)

Source: O Mykhaskiv, M Banović, S Auriemma, P Mohanamuraly, et al, CAD 15(6), 2018.

Aerodynamic Shape Optimisation



* Lyu, Xu, and Martins. **Benchmarking optimization algorithms for wing aerodynamic design optimization.** ICCFD8-2014-0203.



$q(\alpha) \Longrightarrow$ State (N)

$\alpha \Longrightarrow$ Design (M)

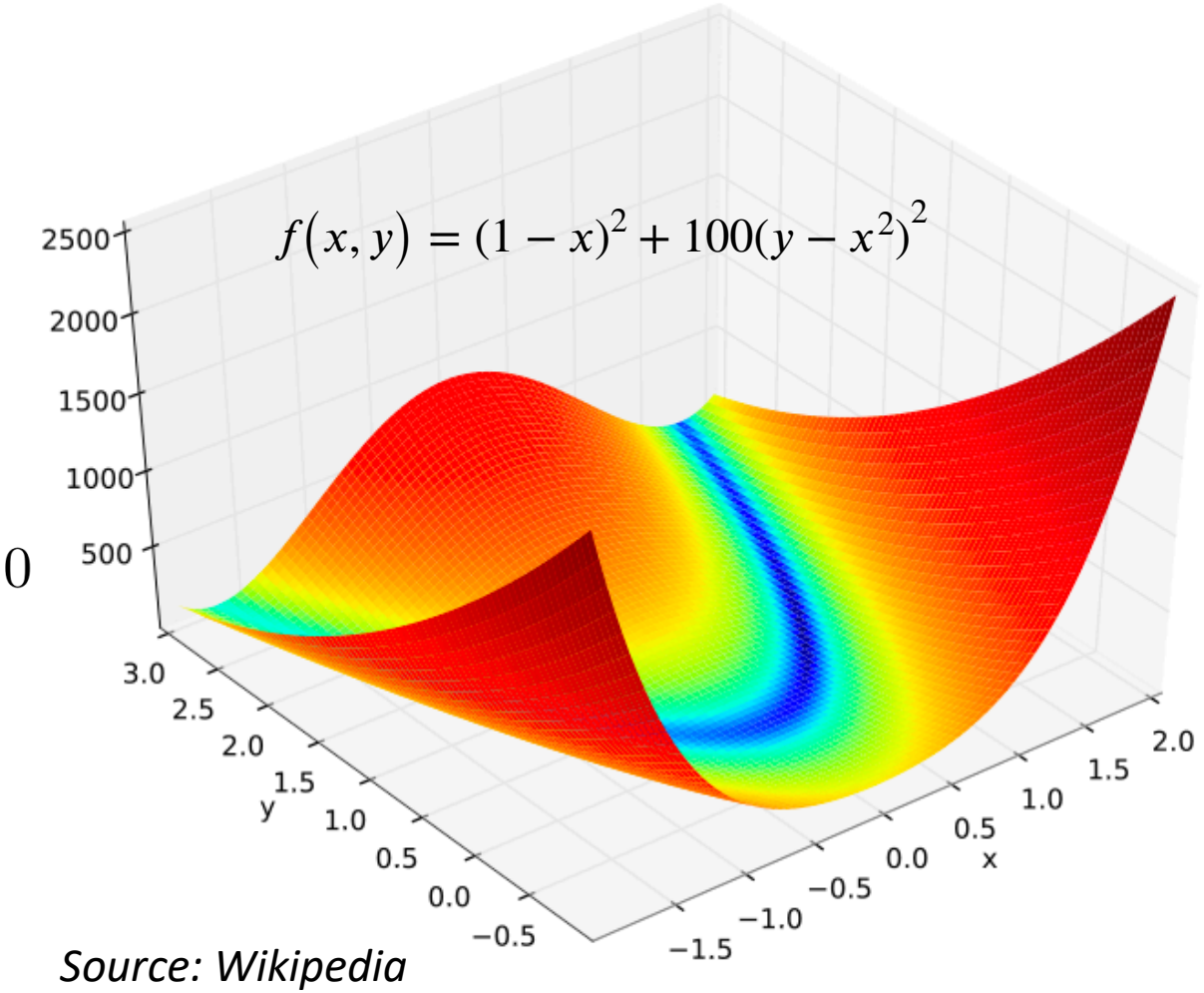
- Gradient-based methods have linear complexity to attain convergence of optimal
  - Well suited to large # DOF
  - Requires derivative or gradient of cost function J wrt α i.e., $\frac{dJ}{d\alpha}$
- Gradient-free have quadratic complexity for the same level of convergence
  - Suitable for smaller # DOF
  - Function evaluation is very cheap
  - Discrete variables or black-box models

Rosenbrock function

$$\text{minimize} \quad \sum_{i=1}^{n-1} 100\,(x_{i+1} - x_i^2)^2 + (x_i - 1)^2$$

$$\text{with respect to} \quad x \in \mathbb{R}^n$$

$$\text{subject to} \quad \hat{\sum}_{i=1}^{n-1} (1.1 - (x_i - 2)^3 - x_{i+1}) \geq 0$$



Source: Wikipedia

$N = 2$

$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$

Global minimum at $(1, 1)$ where $f = 0$

$N = 3$

$f(x, y, z) = (1 - x)^2 + (1 - y)^2 + 100\left[(y - x^2)^2 + (z - y^2)^2\right]$

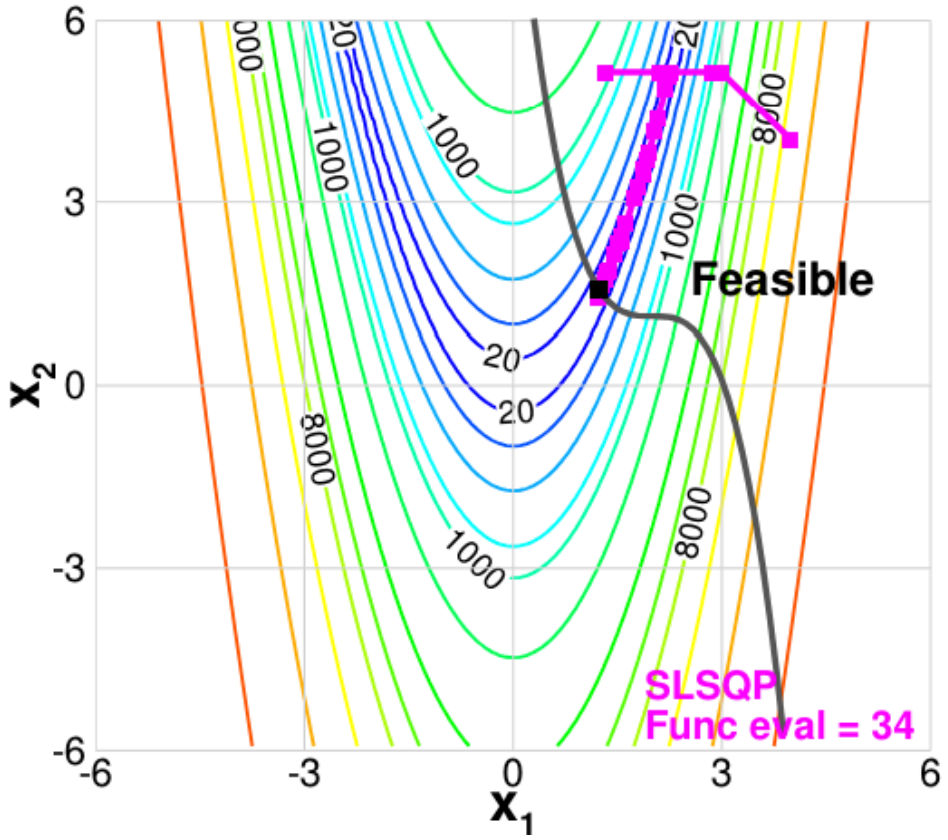
Global minimum at $(1, 1, 1)$

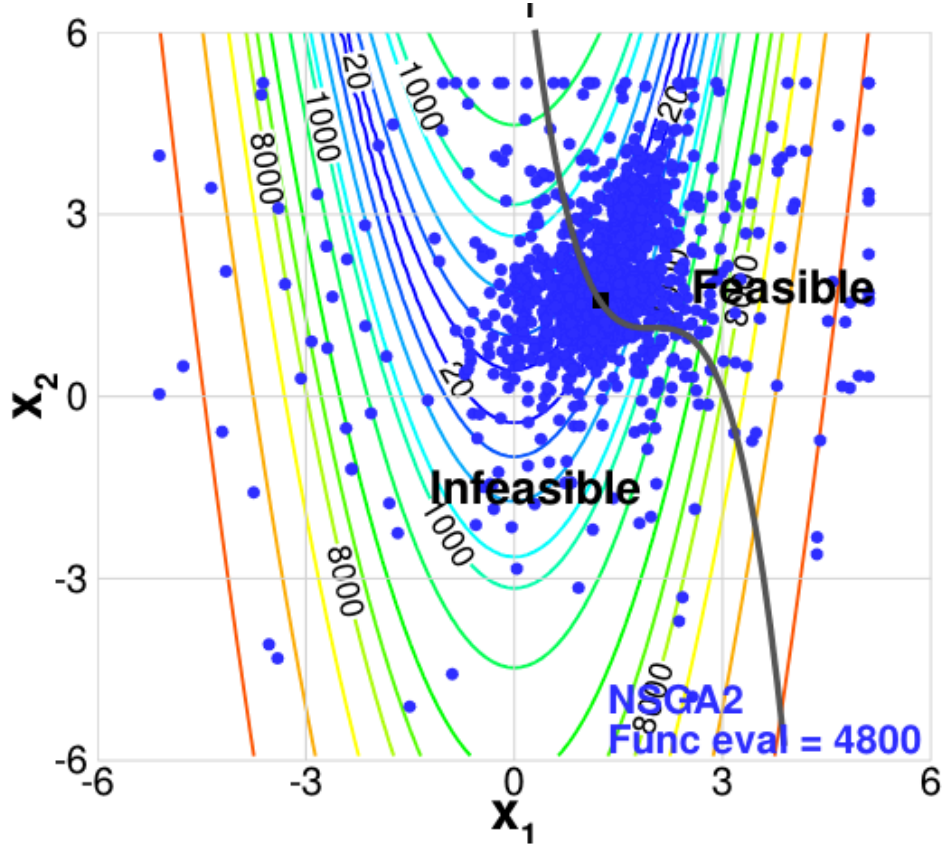
$4 \leq N \leq 7$

Two global minimums

$N \geq 7$

No analytical solution





* Lyu, Xu, and Martins. **Benchmarking optimization algorithms for wing aerodynamic design optimization.** ICCFD8-2014-0203.

- Gradient helps attain minimum faster (less function evaluations)
- Gradient-free explores a larger design space
  - Needs more function evaluation
  - Has a better chance of finding a global minimum

Deep learning and model approximation

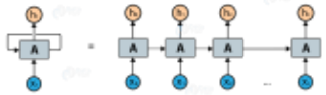
- Remarkably simple mathematical function approximation with tuneable parameters connected in a network

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$
$$\mathbf{z} = \sigma(\mathbf{y})$$

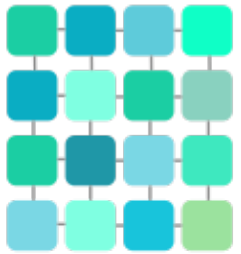
- Deep combinatorics of this kernel produces rich function representation (CNN, GraphNets, etc.)
- Network parameters are optimised to match data or some objective function using gradient-based stochastic optimisation methods

TEMPORAL/ SEQUENCE

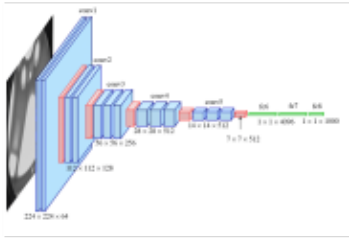
Have a nice day! :)



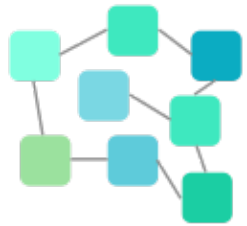
Recurrent neural network (RNN)



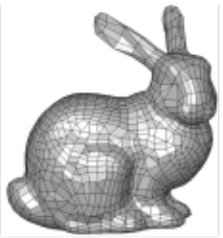
REGULAR TOPOLOGY

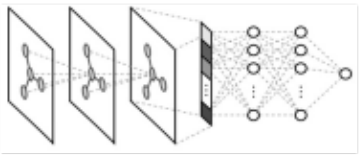


Convolutional neural network (CNN)



RELATIONAL DATA AND UNSTRCUTRED TOPOLOGY





Graph network (GNN/GCN)

Deep Learning and Derivatives

Linear Kernel

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Activation

$$\mathbf{z} = \sigma(\mathbf{y})$$

Layer normalisation

$$\hat{\mathbf{z}} = \frac{\mathbf{z} - \mu}{\sqrt{s + \epsilon}}\beta_\mu + \beta_s$$

Find parameters

$$\mathbf{w} = [\mathbf{W} \;\; \mathbf{b}]$$

that minimise cost-function

$$J(\mathbf{w})$$

given data (batch size $\boldsymbol{n}$)

$$J_i(\mathbf{w})$$

Stochastic gradient descent

$$\mathbf{w}^{new} = \mathbf{w}^{old} + \frac{\eta}{n}\sum_{i=1}^{n} \nabla J_i(\mathbf{w})$$

Algorithmic Differentiation and gradient-based optimization are behind virtually all recent successes in machine learning

Approaches to derivative calculation

- *Symbolic Differentiation*
  - Manually derive and code into program
  - Software based with code generation (Maxima)
- *Approximate Numerical Differentiation*
  - Forward or central difference
- *Complex-step method and (Hyper)Dual numbers*
- *Algorithmic Differentiation*
  - Overloading
  - Source Transformation

What isn’t Algorithmic Differentiation?

Symbolic Differentiation

Automatic manipulation of expressions for obtaining derivative expressions carried out by applying transformations representing rules of differentiation such as:

$$\frac{d}{dx}\left(f(x)+g(x)\right) \rightsquigarrow \frac{d}{dx}f(x)+\frac{d}{dx}g(x)$$

$$\frac{d}{dx}\left(f(x)\,g(x)\right) \rightsquigarrow \left(\frac{d}{dx}f(x)\right)g(x)+f(x)\left(\frac{d}{dx}g(x)\right)$$

- Mechanical process of symbolically differentiating an expression tree.
- Automation using computer algebra systems such as Maxima, PyTensor, etc.
- Valuable insight into the structure of the problem (analytical minimum!)
- Inefficient runtime calculation of derivative values (derivative becomes exponentially larger than the expression)
- Numerical stability issues.

What isn't Algorithmic Differentiation?

Finite-difference

Approximate the derivative using truncated Taylor series expansion. Simplest version is the forward difference,

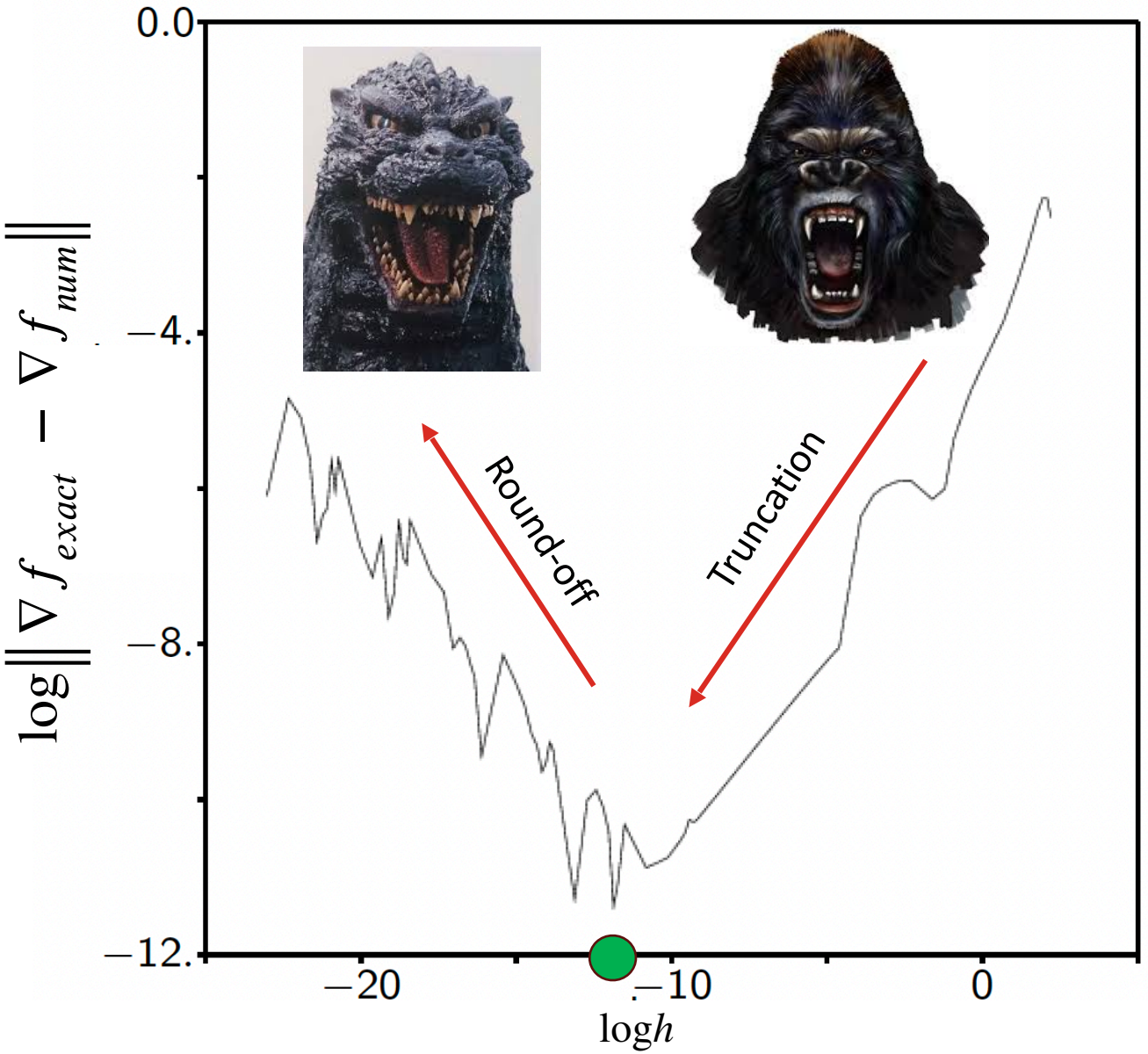
$$\frac{\partial f(\mathbf{x})}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h} + O(h)$$

with h a small perturbation size and $\boldsymbol{e}_i$ a vector of the same length as x with zeros everywhere, but one in position i. A better approximation is the central difference,

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

Can we let $h \to 0$ to make the error vanish?

Truncation vs. Round-off error (Godzilla vs. Kong)



- Results from typical CFD case (approximation using forward difference)
- Optimal h where truncation and round-off are minimal (lowest error)
- Problem/scale dependent (highly sensitivity to step-size h)
- Still a standard method of choice when one has a black-box model
- Useful for verification of gradients obtained using other methods (bring confidence)

Finite-difference for gradient computation

- Needs carefully setting optimal choice for step-size h
  - *Too small* $h \rightarrow$ *large round-off (cancellation)*
  - *Too large* $h \rightarrow$ *large truncation error*

- Forward difference: *T.E.* $\propto \mathcal{O}(h)$ *with one additional evaluation per parameter (or design variable)*

- Central difference: *T.E.* $\propto \mathcal{O}(h^2)$ *with two additional evaluation per parameter (or design variable)*

- Usually for n design variables require $n + 1$ function evaluations (for gradient) !

Derivatives using Complex-step method

Consider the Taylor series expansion of the real valued function $f(x)$ with step-size h in the complex plane $F(x+ih)$

$$F(x+ih) = F(x) + ihF'(x) - \frac{h^2}{2!}F''(x) - \frac{ih^3}{3!}F'''(x) + \mathcal{O}(h^4)$$

Taking the imaginary part and dividing by h we obtain,

$$\frac{1}{h}\mathrm{Im}\Big(F(x+ih)\Big) = F'(x) - \frac{h^3}{3!}F'''(x) + \mathcal{O}(h^4)$$

$$F'(x) = \frac{1}{h}\mathrm{Im}\Big(F(x+ih)\Big) + \mathcal{O}(h^2)$$

The complex variable "trick"

Properties of complex-step gradient computation

$$F'(x) = \frac{1}{h}\mathrm{Im}\Big(F\big(x + ih\big) \Big) + \mathcal{O}(h^2)$$

- No cancellation error problem, so choose h to be arbitrarily small to make T.E. as small as one wishes
- Approximation error: *T.E.* $\propto \mathcal{O}(h^2)$ *same as central difference*
- For n design variables require n function evaluations (linear)
- Cost of complex arithmetic is higher than real-valued computations (SIMD, Compiler/ Hardware optimisations, etc.)
- Can be implemented easily in languages that have support for complex arithmetic
  - Redefine non-analytic functions such as abs, $<$, $>$, etc.

What is Algorithmic Differentiation?

- Consider a computer program to compute the function $f(x)$

- The program can be viewed as a sequence of elementary operations or functions (for example $+,\ -,\ \div,\ \times,\ \sin,\ \cos,$ etc.)

$$f(\boldsymbol{x}) = f_n\Big(f_{n-1}\Big(\cdots f_2\big(f_1(\boldsymbol{x})\big)\Big)\Big),\quad \boldsymbol{x} = \{x_i\}$$

- Derivative of individual elementary operations $f_{n,\cdots,1}$ can be used to obtain the derivative of the program $f(x)$ using chain-rule of calculus (forward mode)

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} = \frac{\partial f_n}{\partial f_{n-1}} \bullet \frac{\partial f_{n-1}}{\partial f_{n-2}} \bullet \cdots \bullet \frac{\partial f_2}{\partial f_1} \bullet \frac{\partial f_1(\boldsymbol{x})}{\partial x_i} = E_n E_{n-1} \cdots E_2 E_1$$

- Process can be automated to compute derivative instruction by instruction (hence the name *Automatic Differentiation*)

But wait …

- With some extra effort we can obtain the transpose of the gradient (reverse or adjoint mode)

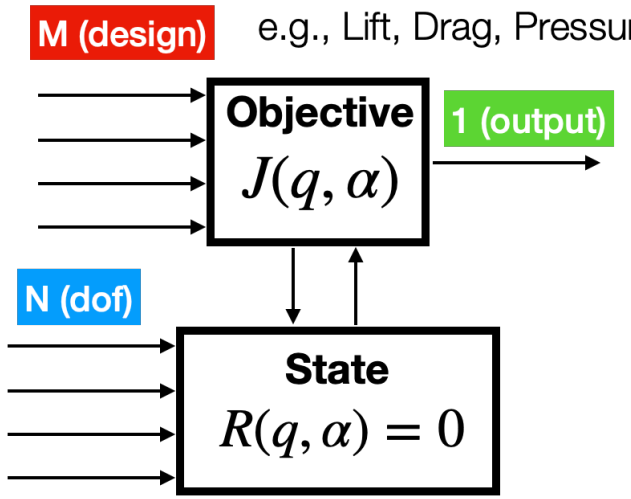
$$\left(\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right)^T = \left(\frac{\partial f_1(\boldsymbol{x})}{\partial x_i}\right)^T \bullet \left(\frac{\partial f_2}{\partial f_1}\right)^T \bullet \cdots \bullet \left(\frac{\partial f_{n-1}}{\partial f_{n-2}}\right)^T \bullet \left(\frac{\partial f_n}{\partial f_{n-1}}\right)^T = E_1^T E_2^T \cdots E_{n-1}^T E_n^T$$

- Magically the cost of computing gradient becomes $\mathcal{O}(1)$ wrt input parameters !

Adjoint primer

Constraint Optimisation

$$\min_{\alpha} J(q,\alpha)$$
$$\textbf{s.t. } R(q,\alpha)=0$$



e.g., Fluid RANS model

Optimisation requires Gradient (or Sensitivity)

$$\frac{dJ}{d\alpha}=\frac{\partial J}{\partial \alpha}+\frac{\partial J}{\partial q}\frac{\partial q}{\partial \alpha}$$

$$\left(\frac{dJ}{d\alpha}\right)^T=\left(\frac{\partial q}{\partial \alpha}\right)^T\left(\frac{\partial J}{\partial q}\right)^T=\underbrace{\left(\frac{\partial b}{\partial \alpha}\right)^T}_{\text{M×N}}\underbrace{\mathbf{A}^{-T}c}_{\text{N×N}\ \ \text{N×1}}$$

Adjoint Equation

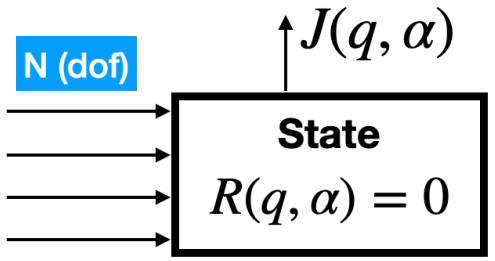
$$\mathbf{A}^T\bar{q}=c$$

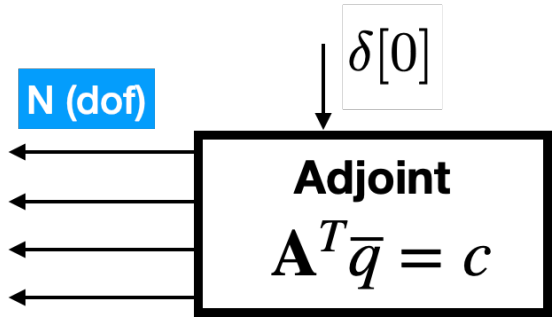
A simple example (linear objective and state)

$$R(q,\alpha):\mathbf{A}q=b \ \textbf{(or)}\ \mathbf{A}q-b=0$$
$$J(q,\alpha)\equiv c^Tq$$

Note: *$q=q(\alpha)$, forcing term $b=b(\alpha)$ and c is a constant vector*

Adjoint is simply the transposition of the state





Physical interpretation - greens function, impulse response

Cost of adjoint sensitivity same for 1 or 1B design parameter!

Summary

- Analytic derivatives are limited to small problems (expression blowup)
- Finite-differences are widely used, simple to implement, but have linear cost and require careful choice of step size
- Complex-step can be implemented effectively in some languages, if the source code is available. No truncation error, but linear cost
- Algorithmic differentiation (AD) needs application at source code level, are exact, cost is linear in forward mode
- AD can be extended to the reverse mode, which is exact and has constant cost

Forward mode using an example

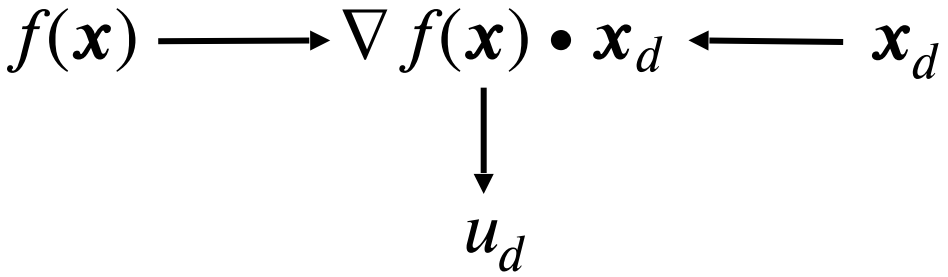
$$f(\boldsymbol{x}) = 3x_1 + 2x_2 + x_3$$

$$\boldsymbol{x} = \{x_1, x_2, x_3\}$$

$$\frac{\partial f(\boldsymbol{x})}{\partial x_1} = 3, \quad \frac{\partial f(\boldsymbol{x})}{\partial x_2} = 2, \quad \frac{\partial f(\boldsymbol{x})}{\partial x_3} = 1$$

```
subroutine f(u, x)
  real, intent(out) :: u
  real, intent(in)  :: x(3)
  u = 3*x(1) + 2*x(2) + x(3)
end subroutine f
```

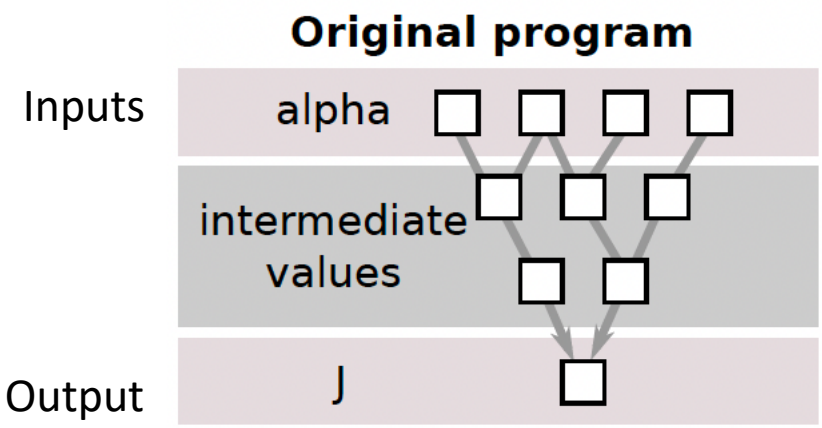
- Forward mode AD gives the directional gradient of function f
- Given the input perturbation $\boldsymbol{x}_d$ it returns the output perturbation $u_d = \nabla f(\boldsymbol{x}) \bullet \boldsymbol{x}_d$

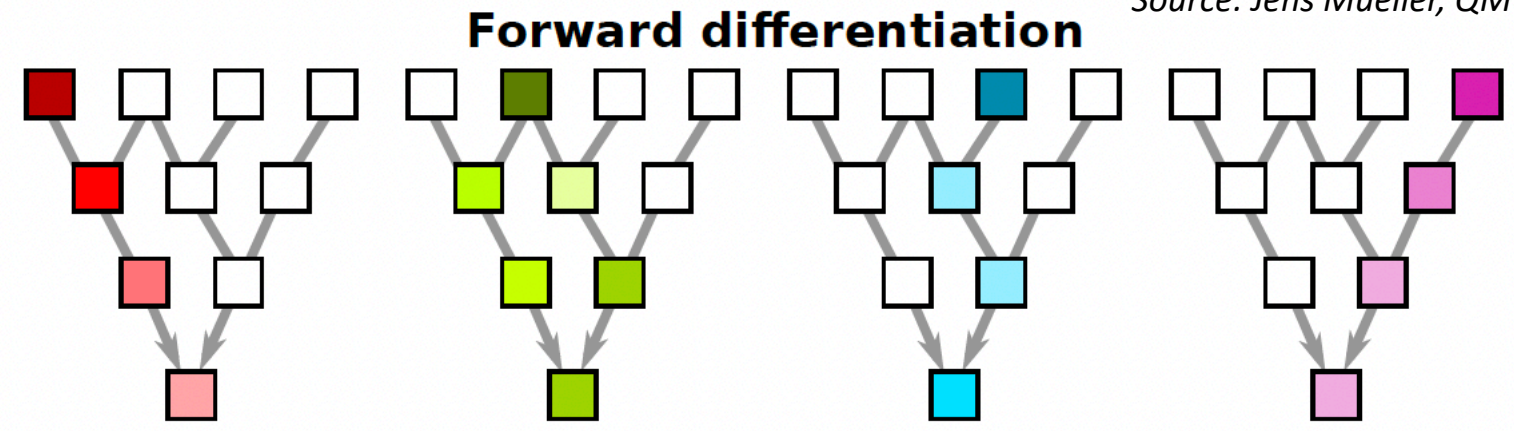



```
subroutine f_d(u, x, ud, xd)
  real, intent(out)   :: u
  real, intent(in)    :: x(3)
  real, intent(inout) :: ud
  real, intent(inout) :: xd(3)
  u = 3*x(1) + 2*x(2) + x(3)
  ud = 3*xd(1) + 2*xd(2) + xd(3)
end subroutine f_d
```

- $\boldsymbol{x}_d$ is called the seed vector and by choosing the right set of seeds we can assemble the complete gradient Jacobian i.e., $\boldsymbol{x}_d = \{1, 0, 0\}, \{0, 1, 0\}$ and $\{0, 0, 1\}$

Algorithmic Differentiation as a graph





Source: Jens Mueller, QMUL

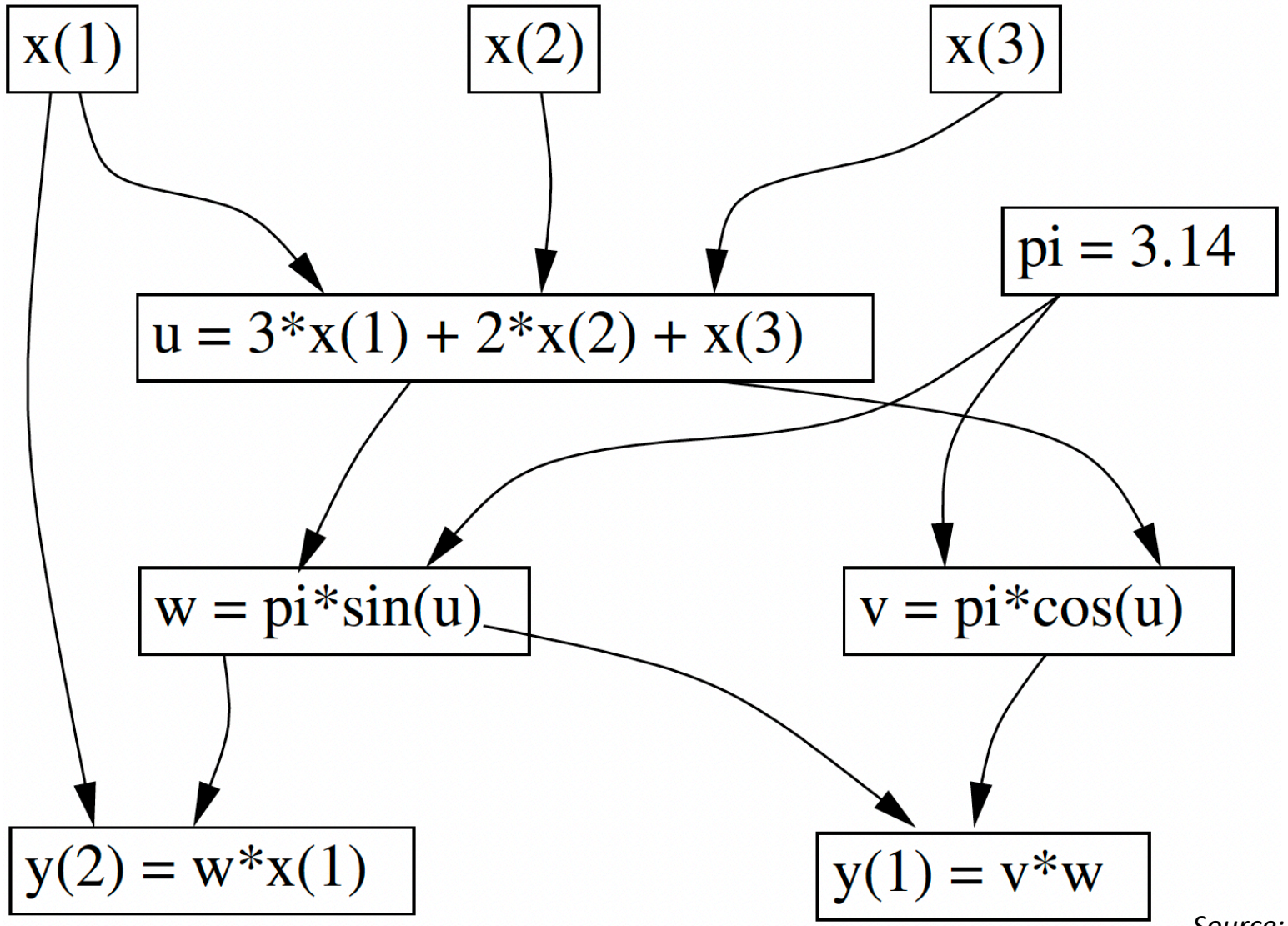
- Propagate the influence of each input α through the program (forward mode)
- Data flow graph clearly depicts the dependency of input, intermediate and output values
- Consider a slightly more complex example,

$$\boldsymbol{x} = \{x_1, x_2, x_3\}$$

$$f(\boldsymbol{x}) = 3x_1 + 2x_2 + x_3$$

$$g(\boldsymbol{y}) = \begin{bmatrix} w \\ v \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \bullet \cos(f) \bullet \pi \bullet \sin(f) \\ \pi \bullet \sin(f) \bullet x_1 \end{bmatrix}$$

Abstract graph of problem



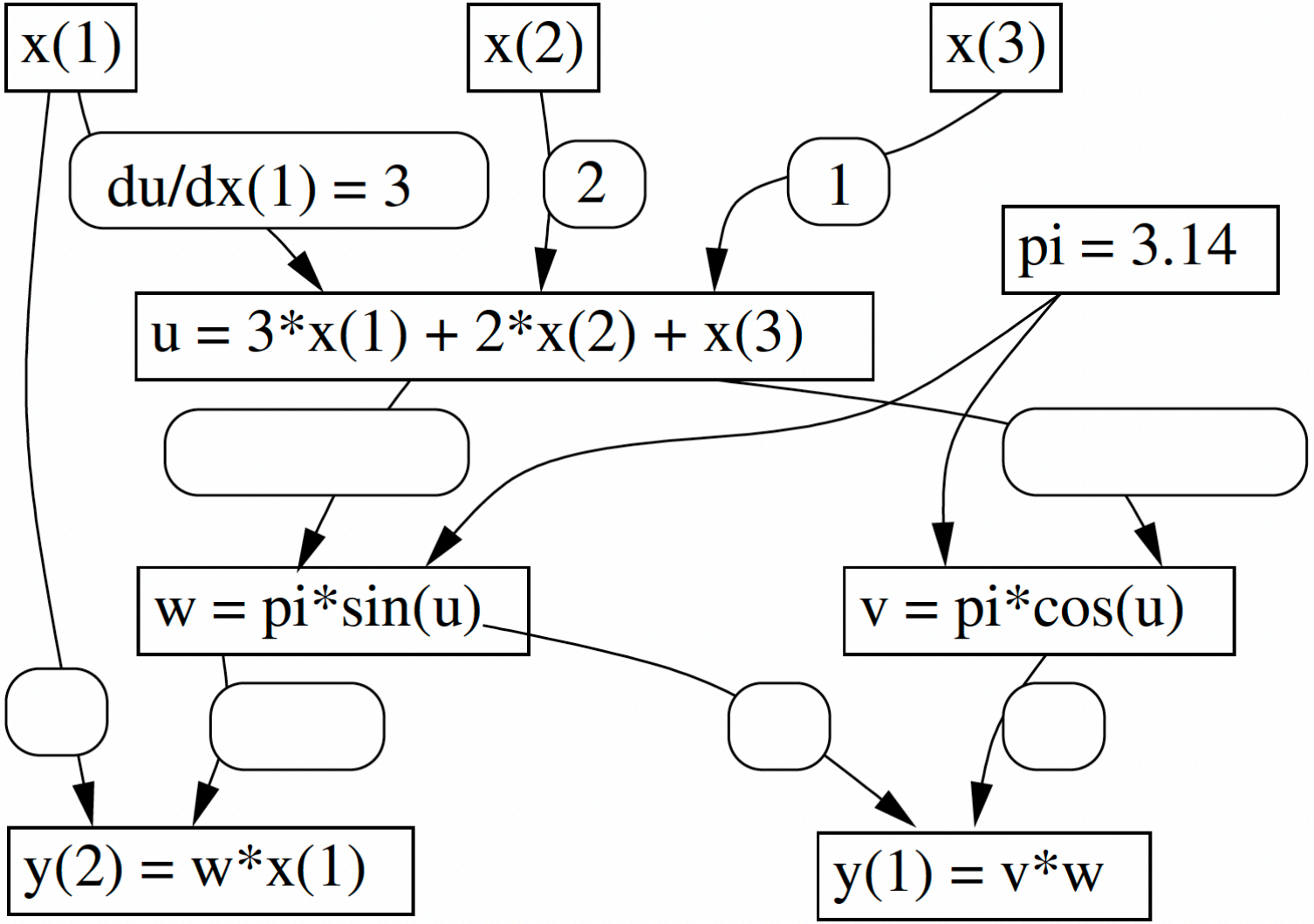
$$\boldsymbol{x} = \{x_1, x_2, x_3\}$$

$$f(\boldsymbol{x}) = 3x_1 + 2x_2 + x_3$$

$$g(\boldsymbol{y}) = \begin{bmatrix} w \\ v \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \bullet \cos(f) \bullet \pi \bullet \sin(f) \\ \pi \bullet \sin(f) \bullet x_1 \end{bmatrix}$$

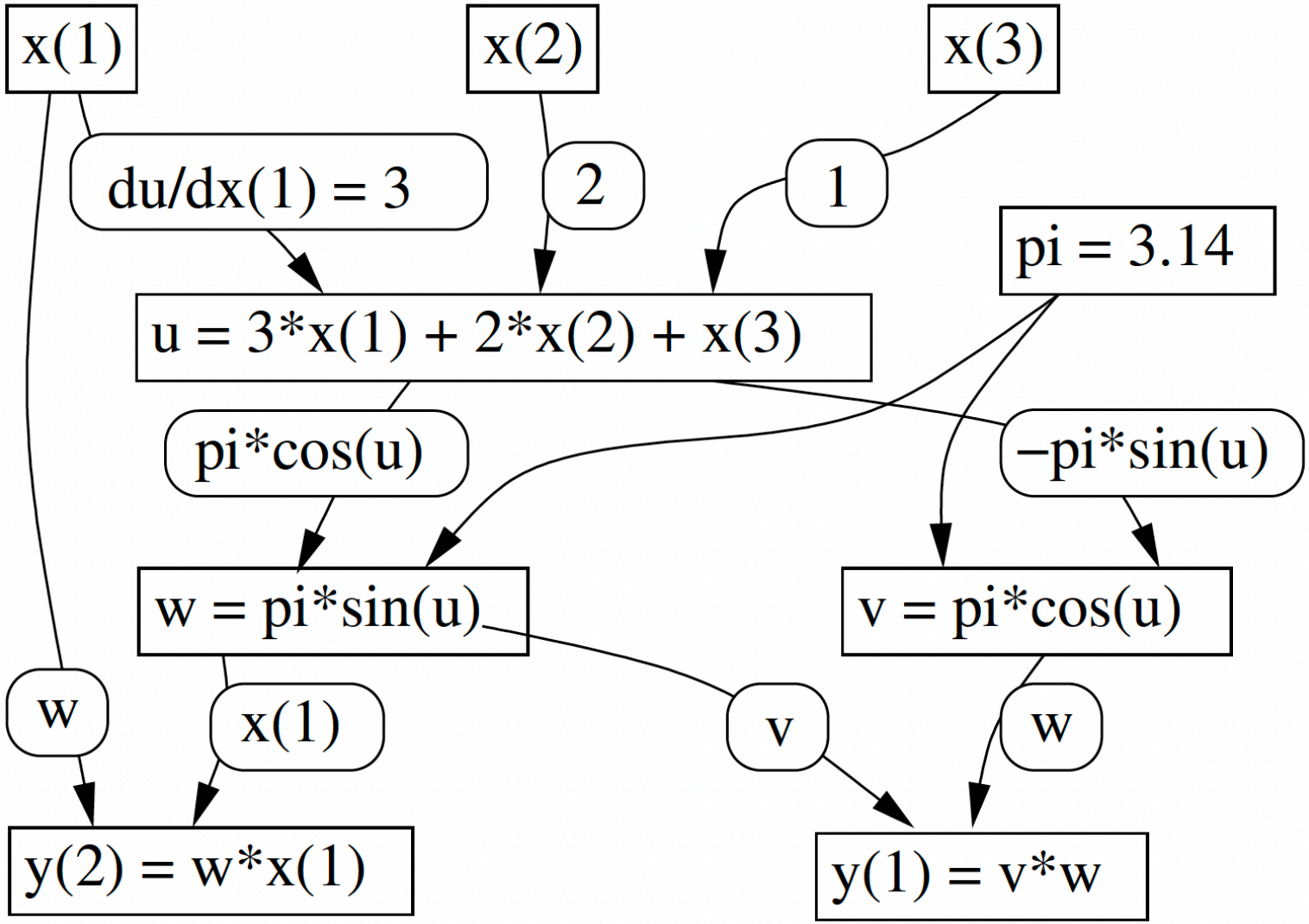
Source: Jens Mueller, QMUL

Add partial derivative calculation along graph edge



Source: Jens Mueller, QMUL

Add partial derivative calculation along graph edge



Source: Jens Mueller, QMUL

Matrix view of Forward AD

- Given a seed vector $\dot{\boldsymbol{x}}$, forward AD computes the Jacobian-vector of the matrix ∇f with the seed $\dot{\boldsymbol{x}}$: $E_n E_{n-1} \cdots E_2 E_1 \dot{\boldsymbol{x}} = E\dot{\boldsymbol{x}} = \nabla f \dot{\boldsymbol{x}}$

$$\nabla f \cdot \dot{x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdots & & & \\ \frac{\partial f_M}{\partial x_1} & \frac{\partial f_M}{\partial x_2} & \cdots & \frac{\partial f_M}{\partial x_n} \end{bmatrix} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_n \end{bmatrix} = \dot{y}$$

- AD computes a *directional derivative* (seed vector)
- Each run of the differentiated program computes all output perturbations in one column of ∇f
- For n inputs we need n invocations of the Forward to assemble the complete gradient ∇f (cost linear in #DOF)

Reverse mode AD

- Recall that the forward mode AD computes gradient of $y = f(x)$

$$\dot{\boldsymbol{y}} = \frac{\partial f}{\partial x}\dot{\boldsymbol{x}} = E_n E_{n-1} \cdots E_2 E_1 \dot{\boldsymbol{x}}$$

- Applying a simple transpose rule of matrix multiplication to $\bar{\boldsymbol{y}}\frac{\partial f}{\partial x}$ we get:

$$\left(\bar{\boldsymbol{y}}\frac{\partial f}{\partial x}\right)^T = E_1^T E_2^T \cdots E_{n-1}^T E_n^T \bar{\boldsymbol{y}}^T$$

- We apply the same differentiation operator E_i as forward mode but use their transpose (apply chain rule and accumulate starting from E_n)
- We follow the primal statement in reverse
- Another convention to represent reverse mode (transpose dropped):

$$\bar{\boldsymbol{x}} = \bar{\boldsymbol{x}} + \left(\frac{\partial f}{\partial x}\right)^T \bar{\boldsymbol{y}}$$

Matrix view of Reverse Derivative

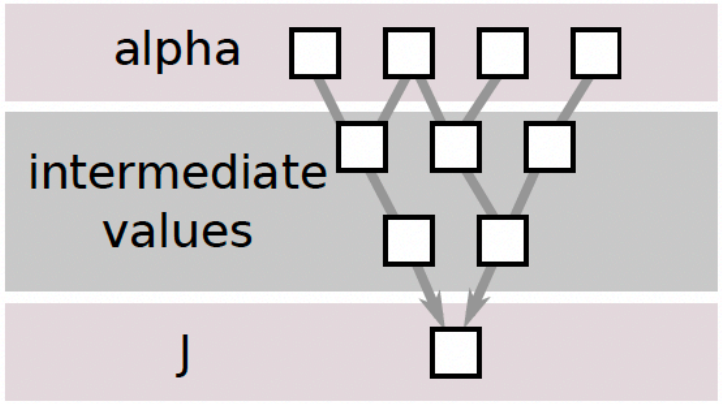
$$[\bar{y}_1, \bar{y}_2] \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_n} \end{bmatrix} = [\bar{x}_1, \bar{x}_2, \bar{x}_3]$$

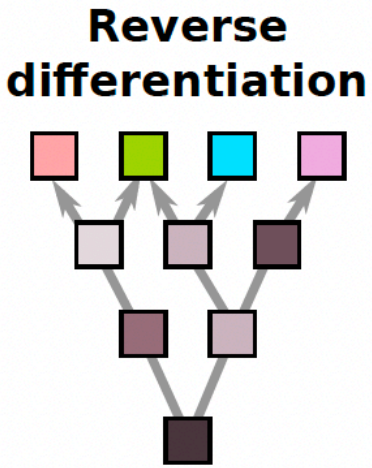
By seeding $\bar{y} = [1, 0]$ we obtain

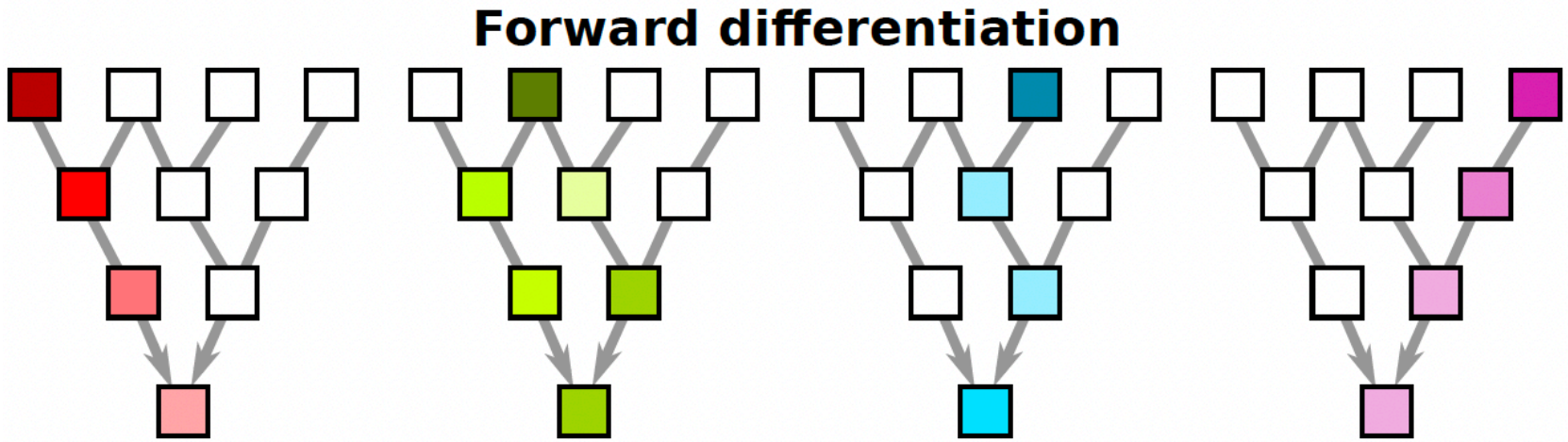
$$[1, 0] \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_3} \end{bmatrix} = \left[\frac{\partial y_1}{\partial x_1}, \frac{\partial y_1}{\partial x_1}, \frac{\partial y_1}{\partial x_3}\right]$$

- Recall in Forward mode we could obtain one column of the Jacobian by seeding
- In Reverse mode we can extract one row at a time by seeding

Graph representation of Reverse and Forward AD







- Forward: *Propagate input perturbations through the program finally reaching the output*
- Reverse: *Trace back the influence of output perturbation on to every input*
- One pass is all it takes to obtain all derivatives!

Source: Jens Mueller, QMUL

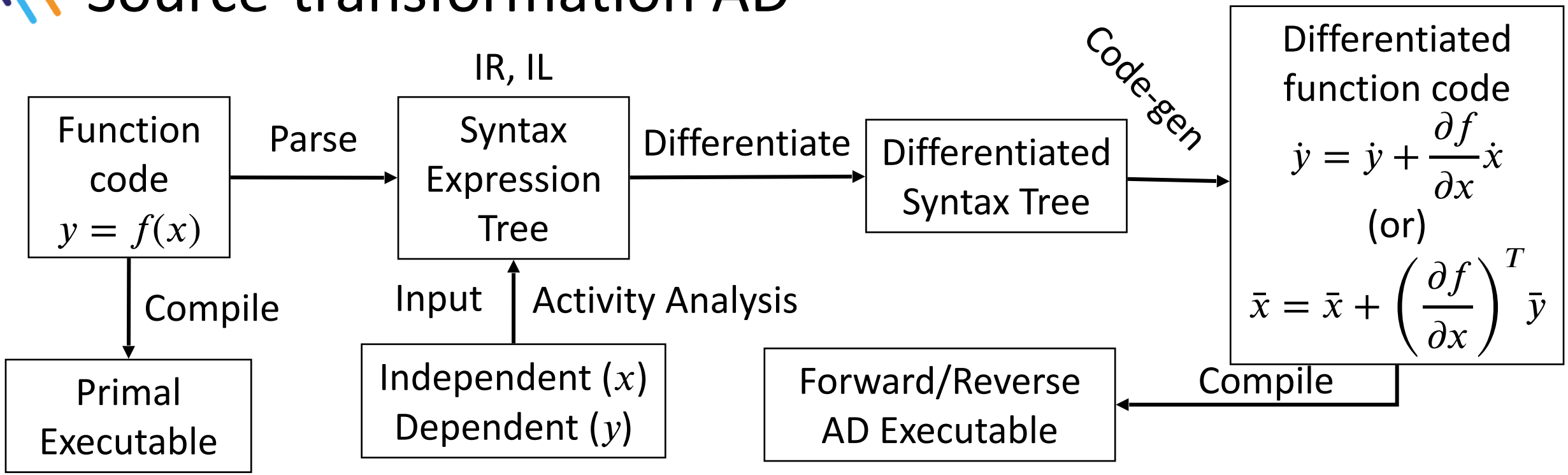
From Algorithmic to Automatic Differentiation

- Forward-mode steps through the statements in the same order, add a derivative computation statement before each primal statement.
- The reverse-mode records all statements, then accumulates their derivatives in reverse.
- Both are straightforward (i.e. rigorous, rule-based, mechanical) process why not have this done by software.

There are two main ways to apply automatic differentiation

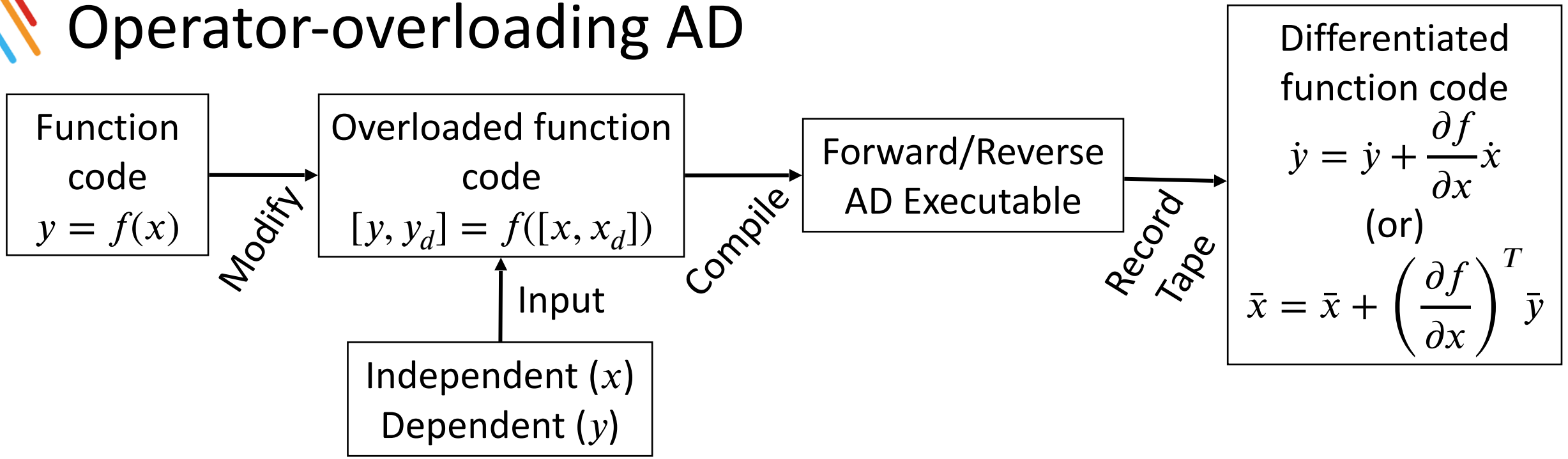
- *Source-transformation AD*
- *Operator-overloading AD*

Source-transformation AD



- Require complete compiler infrastructure (except machine code generation)
- Access to sources of the differentiated program (testing and quality control)
- Some AD tools
  - Tapenade (INRIA): Fortran, C, CUDA-C (exp), C++ (exp) – most popular
  - TAF, TAC (FastOpt, commercial): Fortran, C
  - Enzyme: C/C++, Julia (works on LLVM IR) – technically language agnostic

Operator-overloading AD



- Define special types and overload standard operations such as * or + for these types
- Naturally gives rise to a forward-mode differentiation (operators need overloading)
- Reverse mode needs a tape to store all intermediate operations (for playing back)

Example (C type)

```
struct {
  double val ;
  double val_d ;
} double_d
```

Redefinition of + operator

```
double_d operator *( double_d a, double_d b ) {
  double_d prod ;
  prod.val_d = a.val*b.val_d + a.val_d*b.val ;
  prod.val = a.val * b.val ;
return ( prod ) ; }
```

Operator-overloading vs. Source-transformation

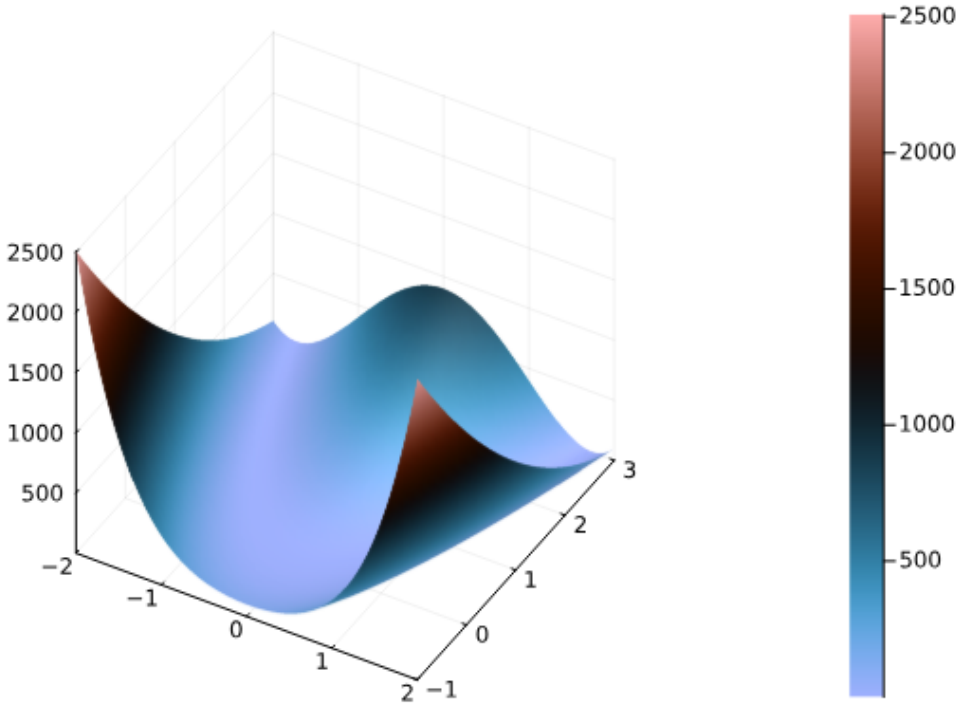
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`
- Static compile-time optimisation is not possible (minimised by C++ expression templates)
- S-T AD usually outperforms O-O AD.

```
struct {
  double val ;
  double val_d ;
} double_d
```

Some popular OO AD tools

- ADOL-C (https://github.com/coin-or/ADOL-C) support for Eigen lib
- CoDiPack (https://www.scicomp.uni-kl.de/software/codi/)
- Sacado (https://trilinos.github.io/sacado.html)

Hands-on exercise 1



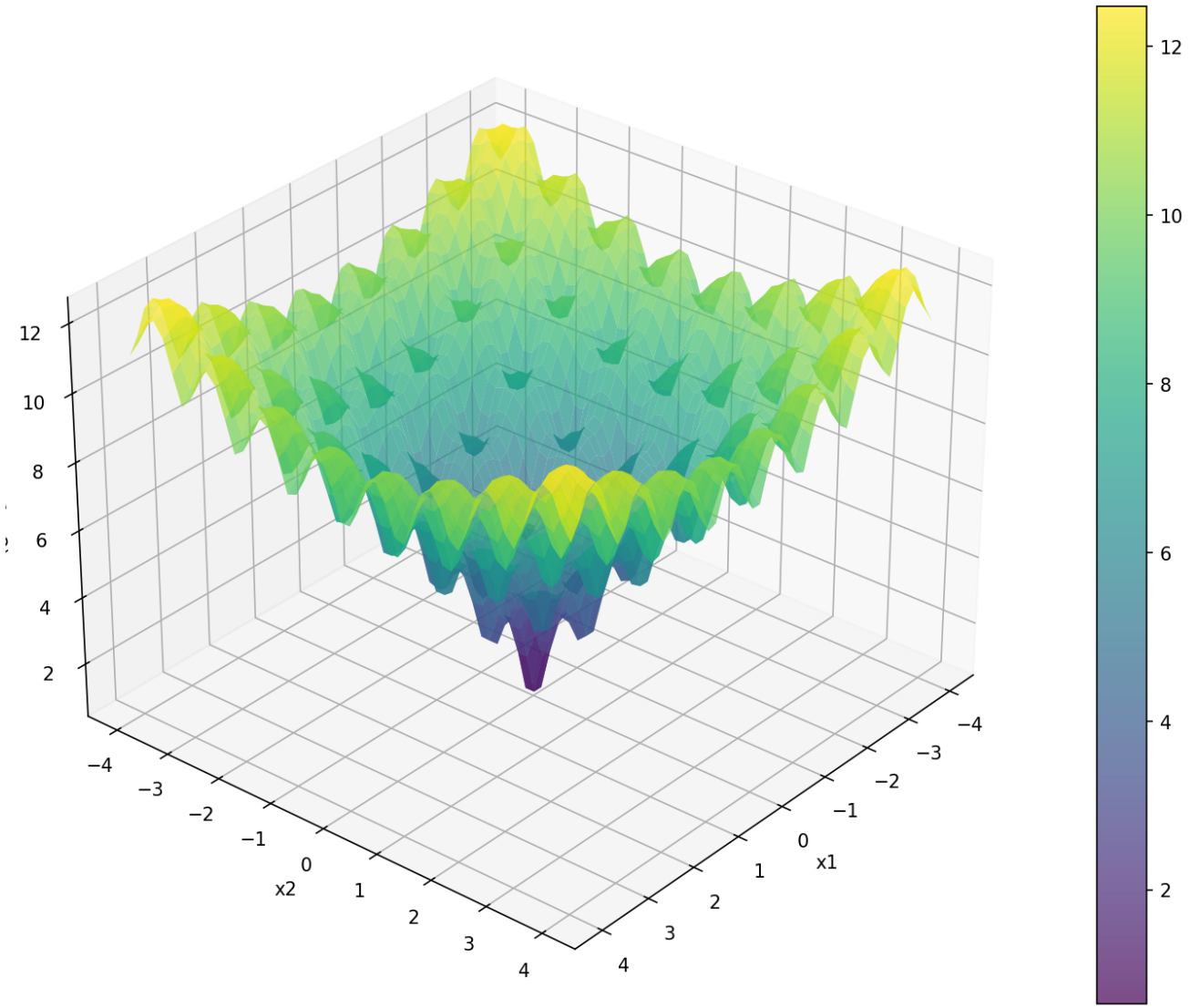
Optimisation using gradient-free (Genetic Algorithm)

Optimisation using gradient-based method (LBFGS)

Estimating gradients using finite-difference

$$\frac{\partial f}{\partial x_1} = \frac{f(x + h\mathbf{e}_1, p) - f(x, p)}{h}$$

Hands-on exercise 2



Optimisation using gradient-free (Genetic Algorithm)

Optimisation using gradient-based method (LBFGS)

Local vs. Global Optimal

Basin of Convergence

Randomised Restart (Probabilistic)

Hands-on exercise 3

For a vortex of strength Γ and centred at (x_0, y_0) and a characteric radius R, the perturbation velocities are given by,

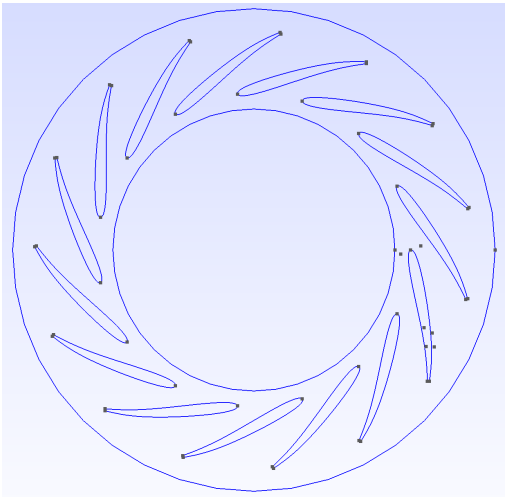
$$(u', v') = \frac{\Gamma}{2\pi R^2} \exp\left[\frac{1 - \left(\frac{r}{R}\right)^2}{2}\right] (y_0 - y, x - x_0)$$

The vorticity (ω) of the Isentropic vortex is obtained using the relation:

$$\omega = \nabla \times \mathbf{v} = \frac{\partial v'}{\partial x} - \frac{\partial u'}{\partial y}$$

Note that we need to obtain the derivatives of the perturbation velocity and let us use both analytically derived expression and complex-step method to estimate this value.

Hands-on exercise 4

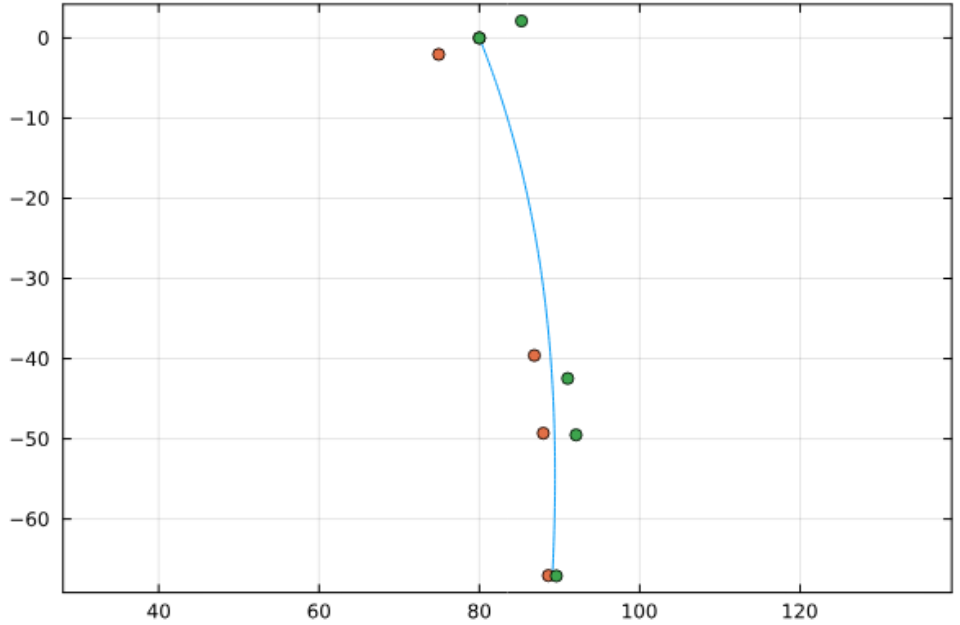


Bézier Curves are parametric curves used extensively in CAD modelling to obtain smooth curves with high degree of control on the curvature. It can be represented mathematically as the function $\mathbf{B}(t)$ with parameter $0 < t < 1$ defined as,

$$\mathbf{B}(t) = \sum_{i=0}^{n} \mathbf{P}_i b_{i,n}(t), \; t \in [0, 1]$$

The points $\mathbf{P}_i$ are called the control points of the curve that can be manipulated to control the curve shape and curvature. The Bernstein basis polynomial $b_{i,n}(t)$ is defined as,

$$b_{i,n}(t) = \binom{n}{i}(1-t)^{n-i}t^i$$



The curve or arc length of a parametric curve is defined as,

$$L = \int_{t_1}^{t_2} \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\, dt$$

We can use autodiff to obtain the derivatives $\frac{dx}{dt}$ and $\frac{dy}{dt}$ by differentiating

Hands-on exercise 5

In this hands-on we demonstrate a simple problem of fitting a 4 layer deep MLP to approximate the function,

$$f(x) = 2x - x^3 \qquad x \in (-2, 2)$$