Conception optimale pour l'ingénieur (Aerospace)

C3 by Prof. J. Morlier

2025

AU PROGRAMME

Python based

lundi 31 mars 2025			
		09h15 - 12h45	MORLIER Joseph
		14h00 - 16h15	MORLIER Joseph
mardi 01 avril 2025			
		09h15 - 12h45	MORLIER Joseph
		14h00 - 16h15	MORLIER Joseph
mercredi 02 avril 2025			
		09h15 - 12h45	MORLIER Joseph MURADÁS ODRIOZOLA Daniel
		14h00 - 16h15	MAS COLOMER JOAN MURADÁS ODRIOZOLA Daniel
jeudi 03 avril 2025			
		09h15 - 12h45	MAS COLOMER JOAN MURADÁS ODRIOZOLA Daniel

Intro: Sustainable Aviation (Materials) With Both Eyes Open
Design optimization 1: constrained optimization, MOO, Sensibility with examples
Project DO 1 2 3

Topology Optimization with examples
Material ecoselection, Ashby Diagram and more

Projet DO 1 2 3
Wrap up and demo from students

Intro to MDAO **Static Aeroelastic problem is a MDAO problem**
Airbus PROJECT by TEAM of 3 (marked*)

vendredi 04 avril 2025		**ORAL MARKED***	
		09h15 - 11h30	MORLIER Joseph MURADÁS ODRIOZOLA Daniel

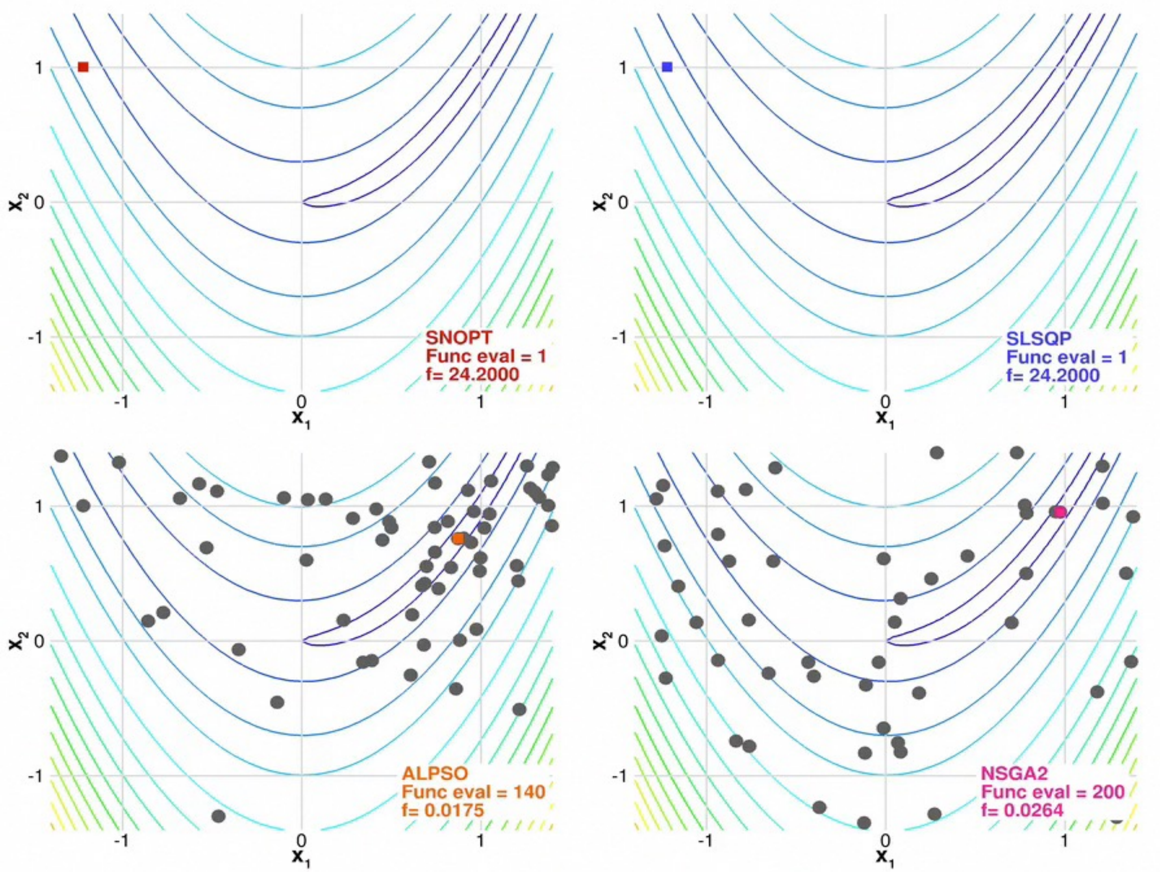
Part1 : On sensibility

Gradient, Hessian and many more?

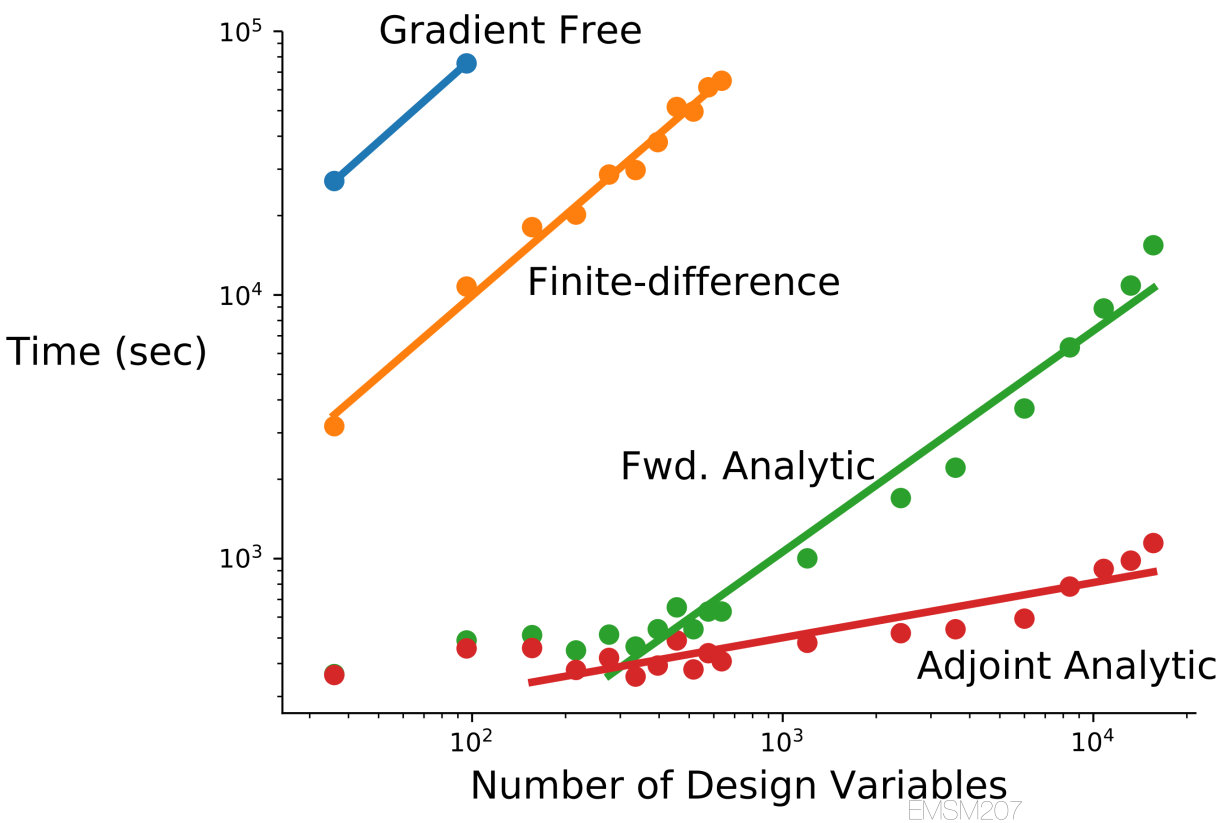
Gradient-based methods take a more direct path to … the optimum

Gradient-based

Gradient-free



Gradient-based optimization with analytic derivatives is our only hope for large-scale problems



100x-10,000x speedup for aerodynamic shape optimization vs. gradient-free[1]

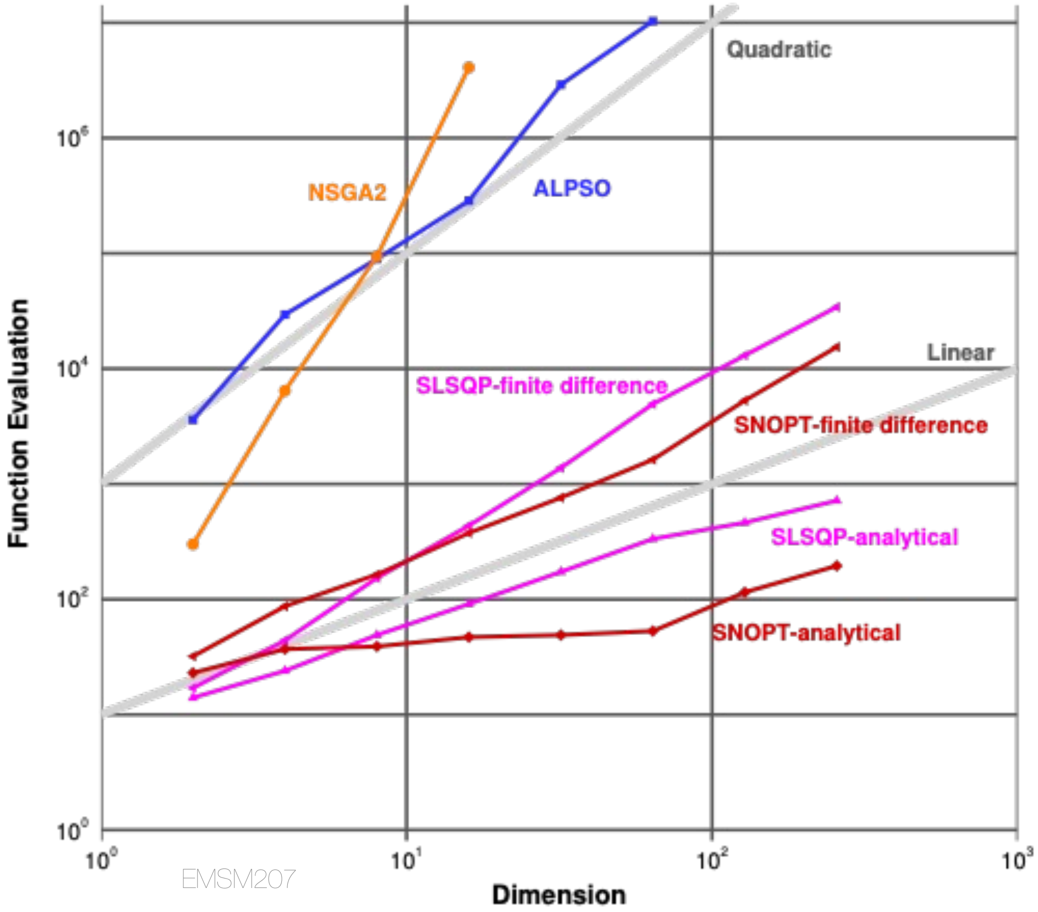
At least 5x-10x speedup vs. finite-difference[2]

[1] Lyu et al. ICCFD8-2014-0203
[2] Gray et al. Aviation 2014-2042

Gradient-based optimization is the only hope

for large numbers of design variables

Need accurate derivative



[Lyu et al. ICCFD8-2014-0203]

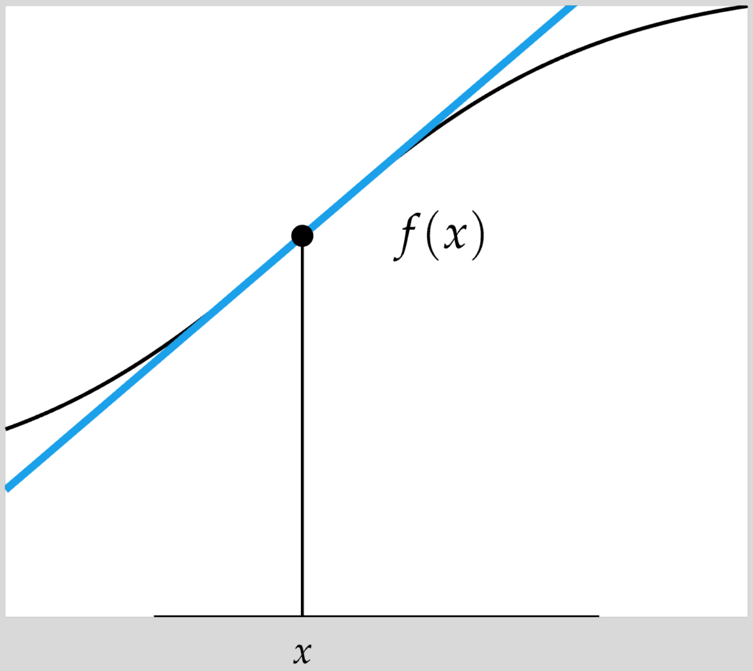
Derivatives

- Derivatives tell us which direction to search for a solution

Derivatives

- Slope of Tangent Line

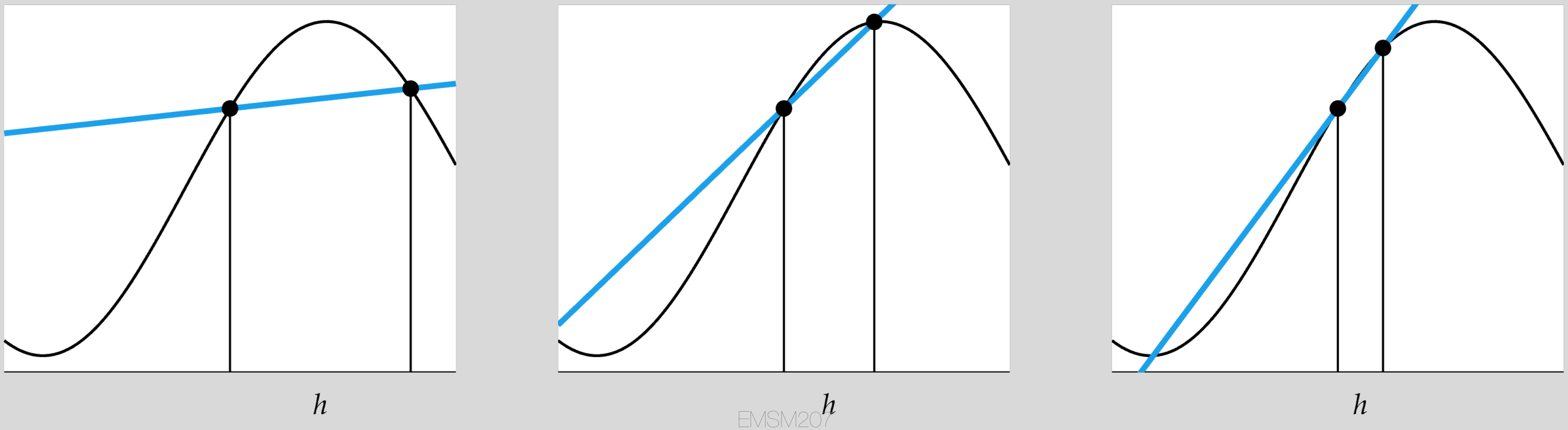
$$f'(x) \equiv \frac{df(x)}{dx}$$



Derivatives

$$f(x+\Delta x) \approx f(x) + f'(x)\Delta x$$

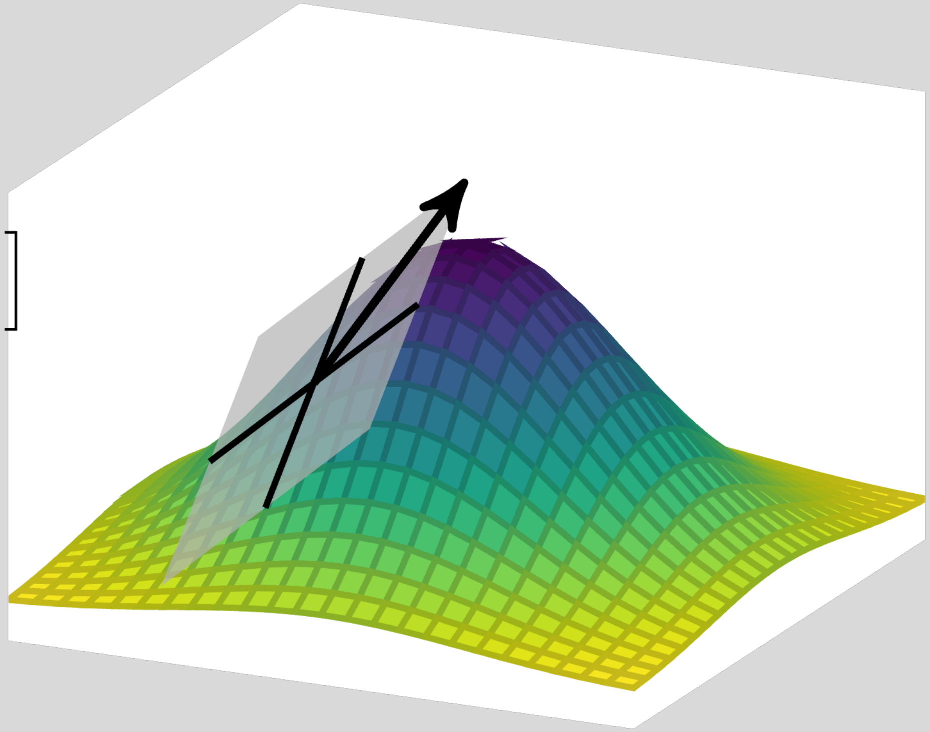
$$f'(x) = \frac{\Delta f(x)}{\Delta x}$$



Derivatives in Multiple Dimensions

- Gradient Vector

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \quad \frac{\partial f(\mathbf{x})}{\partial x_2}, \quad \ldots, \quad \frac{\partial f(\mathbf{x})}{\partial x_n}\right]$$



- Hessian Matrix

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ & \vdots & & \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_n} \end{bmatrix}$$

Derivatives in Multiple Dimensions

- Directional Derivative

$$\nabla_{\mathbf{s}} f(\mathbf{x}) \equiv \underbrace{\lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{s}) - f(\mathbf{x})}{h}}_{\text{forward difference}}$$

$$= \underbrace{\lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{s}/2) - f(\mathbf{x} - h\mathbf{s}/2)}{h}}_{\text{central difference}}$$

$$= \underbrace{\lim_{h \to 0} \frac{f(\mathbf{x}) - f(\mathbf{x} - h\mathbf{s})}{h}}_{\text{backward difference}}$$

Analytical sensitivities

If the objective function is known in closed form, we can often compute the gradient vector(s) in closed form (analytically):

Example: $J\,(x_1, x_2) = x_1 + x_2 + \dfrac{1}{x_1 \cdot x_2}$

Analytical Gradient: $\nabla J = \begin{bmatrix} \dfrac{\partial J}{\partial x_1} \\ \dfrac{\partial J}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 - \dfrac{1}{x_1^2 x_2} \\ 1 - \dfrac{1}{x_1 x_2^2} \end{bmatrix}$

Example

$x_1 = x_2 = 1$

$J(1,1) = 3$

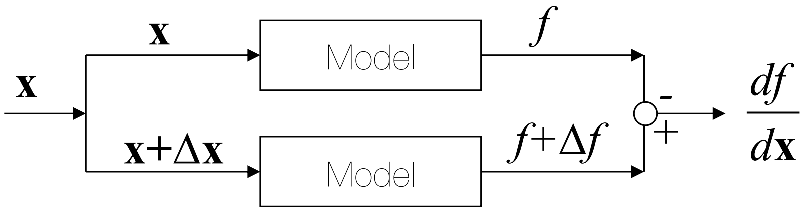
$\nabla J(1,1) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Minimum

For complex systems analytical gradients are rarely available

Sensitivity analysis approaches

Simpler approach (*default with fmincon*):



GFD

Matlab

How to proceed with PDE such as Kq=f?

Discrete

Nastran SOL200

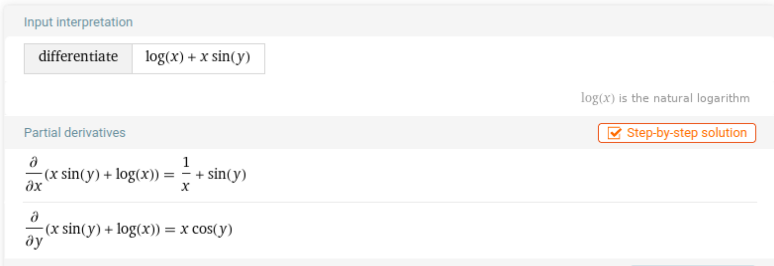
Symbolic differentiation

- Use symbolic mathematics programs
- e.g. MATLAB®, Maple®, Mathematica®

construct a symbolic object

```
» syms x1 x2
» J=x1+x2+1/(x1*x2);
» dJdx1=diff(J,x1)
dJdx1 =1-1/x1^2/x2
» dJdx2=diff(J,x2)
dJdx2 = 1-1/x1/x2^2
```

difference operator



Symbolic differentiation using WolframAlpha.

Automatic Differentiation

- Mathematical formulae are built from a finite set of basic functions, e.g. additions, sin x, exp x, etc.
- Using chain rule, differentiate analysis code: add statements that generate derivatives of the basic functions
- Tracks numerical values of derivatives, does not track symbolically as discussed before
- Outputs modified program = original + derivative capability
- e.g., ADIFOR (FORTRAN), TAPENADE (C, FORTRAN), TOMLAB (MATLAB), many more…
- Resources at http://www.autodiff.org/
- USE JULIA

https://sinews.siam.org/Details-Page/scientific-machine-learning-how-julia-employs-differentiable-programming-to-do-it-best

How Nastran
(a FE code)
is
doing this ?

- General case f is not depending on xi
- except for volumic force (i.e. gravity)

(2) Comment fait Nastran ?

→ approche semi-analytique

$$K(x)\,u(x) = f(x)$$

$$\frac{\partial K(x)}{\partial x_i}\,u(x) + K(x)\,\frac{\partial u(x)}{\partial x_i} = \frac{\partial f(x)}{\partial x_i}$$

$$K(x)\,\frac{\partial u(x)}{\partial x_i} = \frac{\partial f(x)}{\partial x_i} - \frac{\partial K(x)}{\partial x_i}\,u(x)$$

$$\tilde{K}\,\tilde{u} = \tilde{f}$$

$$\tilde{u} = \tilde{K}^{-1}\,\tilde{f} = K^{-1}\,\tilde{f}$$

$$\frac{\partial u(x)}{\partial x_i} = K^{-1}\left\{\frac{\partial f(x)}{\partial x_i} - \frac{\partial K(x)}{\partial x_i}\,u(x)\right\}$$

$$= K^{-1}\left\{\frac{f(x+\Delta x) - f(x)}{\Delta x} - \frac{K(x+\Delta x) - K(x)}{\Delta x}\,u(x)\right\}$$

Numerical Differentiation

- Finite Difference Methods
- Complex Step Method

Numerical Differentiation: Finite Difference

- Derivation from Taylor series expansion

$$f(x+h) = f(x) + \frac{f'(x)}{1!}h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \cdots$$

Numerical Differentiation: Finite Difference

- Neighboring points are used to approximate the derivative

$$f'(x) \approx \underbrace{\frac{f(x+h)-f(x)}{h}}_{\text{forward difference}} \approx \underbrace{\frac{f(x+h/2)-f(x-h/2)}{h}}_{\text{central difference}} \approx \underbrace{\frac{f(x)-f(x-h)}{h}}_{\text{backward difference}}$$

- h too small causes numerical cancellation errors

Numerical Differentiation: Finite Difference

- Error Analysis
  - Forward Difference: $O(h)$
  - Central Difference: $O(h^2)$

Numerical Differentiation: Complex Step

- Taylor series expansion using imaginary step

$$f(x+ih) = f(x) + ihf'(x) - h^2\frac{f''(x)}{2!} - ih^3\frac{f'''(x)}{3!} + \cdots$$

$$f'(x) = \frac{\mathrm{Im}(f(x+ih))}{h} + O(h^2) \text{ as } h \to 0$$

$$f(x) = \mathrm{Re}(f(x+ih)) + O(h^2)$$

Complex Step Derivative (see LIVESCRIPT ON LMS)

- Similar to finite differences, but uses an imaginary step

$$f'(x_0) \approx \frac{\text{Im}\left[f(x_0 + i\Delta x)\right]}{\Delta x}$$

Second order accurate

Canx use very small step sizes e.g. $\Delta x \approx 10^{-20}$

Doesn't have rounding error, since it doesn't perform subtraction

Limited application areas

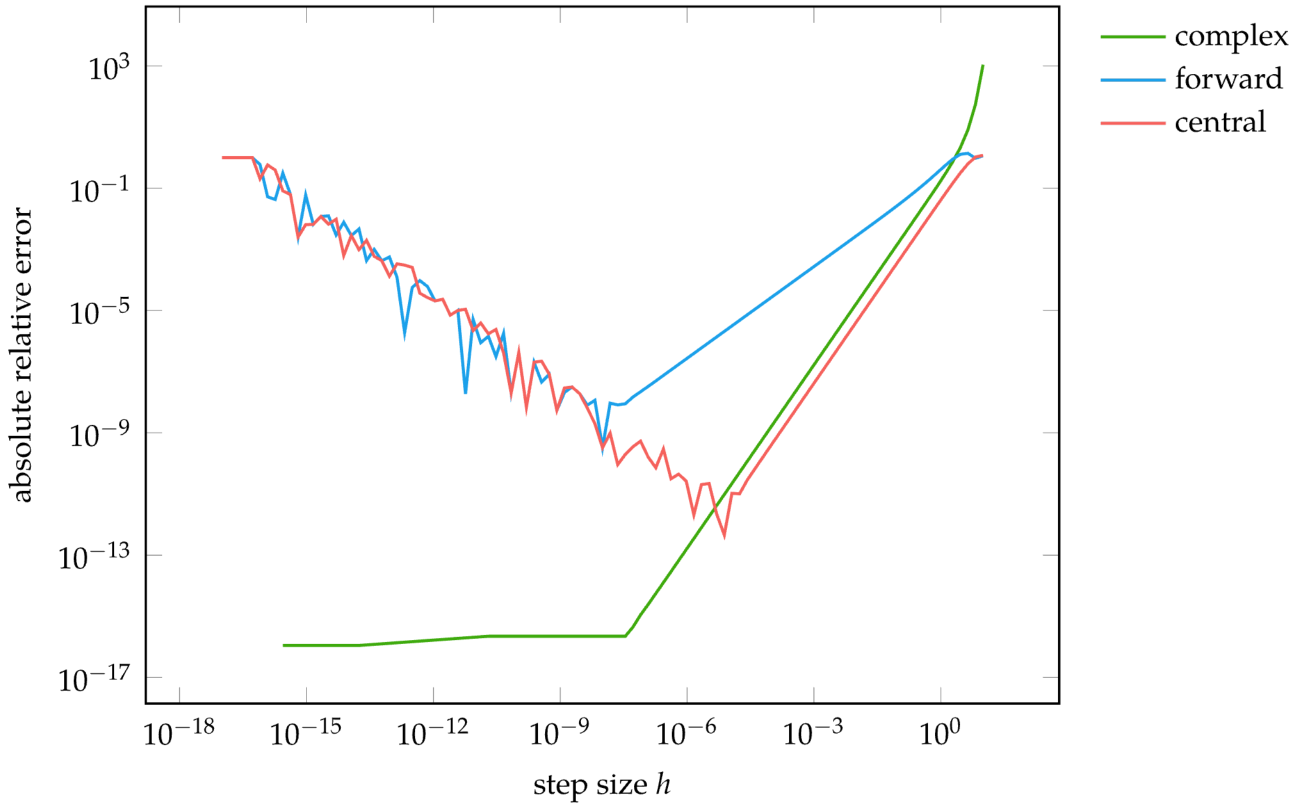
Code must be able to handle complex step values

J.R.R.A. MARTINS, I.M. KROO AND J.J. ALONSO, AN AUTOMATED METHOD FOR SENSITIVITY ANALYSIS USING COMPLEX VARIABLES, AIAA PAPER 2000-0689, JAN 2000

First Exercice

Read and finish the notebook called Complexstep_student.ipynb

Numerical Differentiation Error Comparison



EMSM207

Automatic Differentiation

- Evaluate a function and compute partial derivatives simultaneously using the chain rule of differentiation

$$\frac{d}{dx} f(g(x)) = \frac{d}{dx}(f \circ g)(x) = \frac{df}{dg}\frac{dg}{dx}$$

AD . . . is Computer Sciences

A program is composed of elementary operations like addition, subtraction, multiplication, and division.

Consider the function $f(a,b) = \ln(ab + \max(a,2))$. If we want to compute the partial derivative with respect to a at a point, we need to apply the chain rule several times:[9]

$$
\begin{aligned}
\frac{\partial f}{\partial a} &= \frac{\partial}{\partial a}\ln(ab+\max(a,2)) \\
&= \frac{1}{ab+\max(a,2)}\frac{\partial}{\partial a}(ab+\max(a,2)) \\
&= \frac{1}{ab+\max(a,2)}\left[\frac{\partial(ab)}{\partial a} + \frac{\partial \max(a,2)}{\partial a}\right] \\
&= \frac{1}{ab+\max(a,2)}\left[\left(b\frac{\partial a}{\partial a} + a\frac{\partial b}{\partial a}\right) + \left((2>a)\frac{\partial 2}{\partial a} + (2<a)\frac{\partial a}{\partial a}\right)\right] \\
&= \frac{1}{ab+\max(a,2)}[b + (2<a)]
\end{aligned}
$$

One example

- Forward Accumulation is equivalent to expanding a function using the chain rule and computing the derivatives inside-out
- Requires n-passes to compute n-dimensional gradient

$$\frac{\partial f}{\partial a}(3,2) \qquad f(a,b) = \ln(ab + \max(a,2))$$

AD computational graphs

$$\frac{\partial f(3,2)}{\partial a}$$

- Forward Accumulation

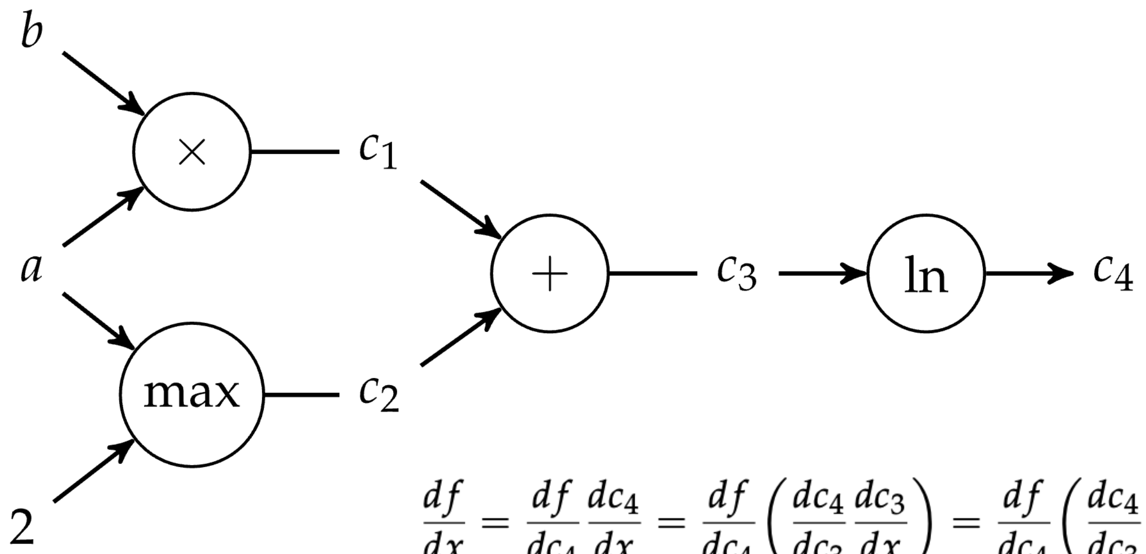
$$f(a,b) = \ln(\, ab + \max(a, 2))$$

$b = 2$

$\frac{\partial b}{\partial a} = \quad \dot{b} = 0$

$a = 3$

$\dot{a} = 1$



$$\frac{df}{dx} = \frac{df}{dc_4}\frac{dc_4}{dx} = \frac{df}{dc_4}\left(\frac{dc_4}{dc_3}\frac{dc_3}{dx}\right) = \frac{df}{dc_4}\left(\frac{dc_4}{dc_3}\left(\frac{dc_3}{dc_2}\frac{dc_2}{dx} + \frac{dc_3}{dc_1}\frac{dc_1}{dx}\right)\right)$$

Automatic Differentiation

- Forward Accumulation

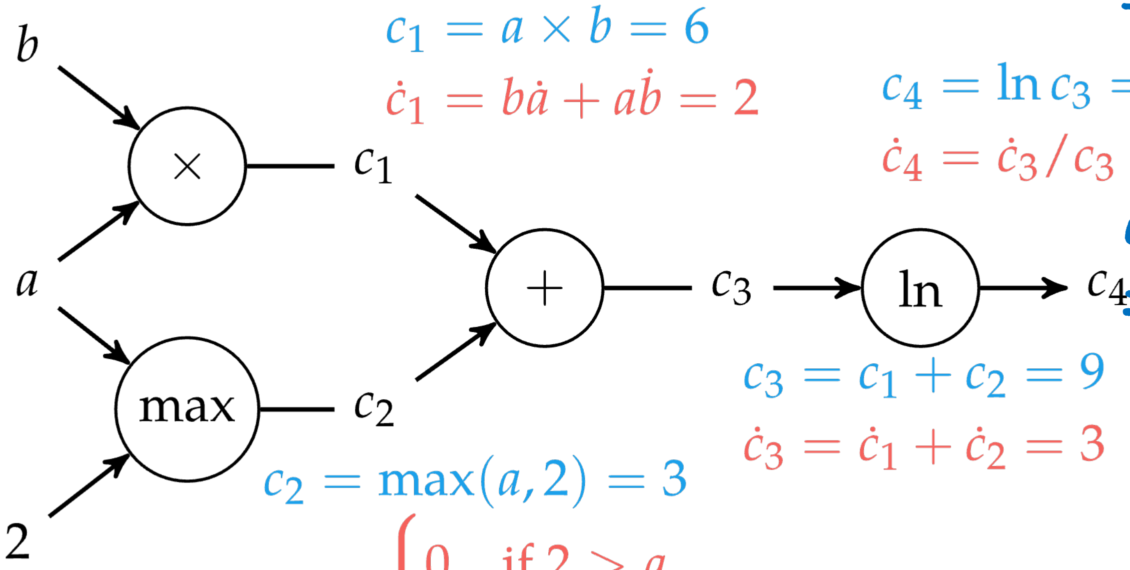
$$f(a,b) = \ln(\, ab + \max(a,2))$$

$b = 2$

$\dot{b} = 0$

$a = 3$

$\dot{a} = 1$



$c_1 = a \times b = 6$

$\dot{c}_1 = b\dot{a} + a\dot{b} = 2$

$c_2 = \max(a,2) = 3$

$$\dot{c}_2 = \begin{cases} 0 & \text{if } 2 > a \\ \dot{a} & \text{if } 2 < a \end{cases} = 1$$

$c_3 = c_1 + c_2 = 9$

$\dot{c}_3 = \dot{c}_1 + \dot{c}_2 = 3$

$c_4 = \ln c_3 = \ln 9$

$\dot{c}_4 = \dot{c}_3 / c_3 = \frac{1}{3}$

$f(3,2) = \ln 9$

$df(3,2)/da = 1/3$

In Julia

The `ForwardDiff.jl` package supports an extensive set of mathematical operations and additionally provides gradients and Hessians.

```
julia> using ForwardDiff
julia> a = ForwardDiff.Dual(3,1);
julia> b = ForwardDiff.Dual(2,0);
julia> log(a*b + max(a,2))
Dual{Nothing}(2.1972245773362196,0.3333333333333333)
```

In Julia

The `Zygote.jl` package provides automatic differentiation in the form of reverse-accumulation. Here the `gradient` function is used to automatically generate the backwards pass through the source code of `f` to obtain the gradient.

```
julia> import Zygote: gradient
julia> f(a, b) = log(a*b + max(a,2));
julia> gradient(f, 3.0, 2.0)
(0.3333333333333333, 0.3333333333333333)
```

What is JAX?

JAX = accelerated array computation + program transformation

```python
import jax.numpy as jnp
```

- JAX is NumPy on the CPU and GPU!
- JAX uses XLA (Accelerated Linear Algebra) to compile and run NumPy code, *lightning fast*

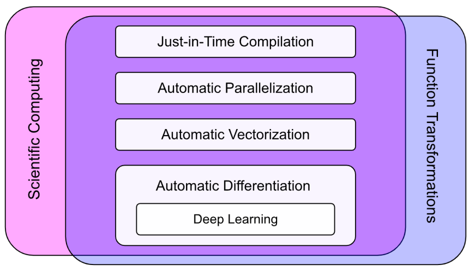


Image credit: AssemblyAI

- JAX can automatically *differentiate* and *parallelise* native Python and NumPy code

JAX

JAX is NumPy on the GPU

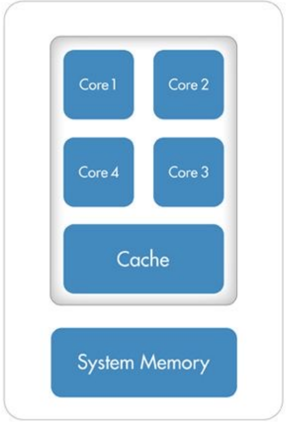
```python
import numpy as np

A = np.array([[1., 2., 3.],
              [1., 2., 3.],
              [1., 2., 3.],])

x = np.array([4.,5.,6.])

b = A @ x
print(b)

---
[32. 32. 32.]
```

(10,000 x 10,000) (10,000 x 10,000)
NumPy on CPU (Apple M1 Max):
7.22 s ± 109 ms

```python
import jax.numpy as jnp

A = jnp.array([[1., 2., 3.],
               [1., 2., 3.],
               [1., 2., 3.],])

x = jnp.array([4.,5.,6.])

b = A @ x
print(b)

---
[32. 32. 32.]
```

(10,000 x 10,000) (10,000 x 10,000)
JAX on GPU (NVIDIA RTX 3090):
56.9 ms ± 222 µs (**126x** faster)

CPU (Multiple Cores)



Low latency
Ideal for serial processing

GPU (Hundreds of Cores)

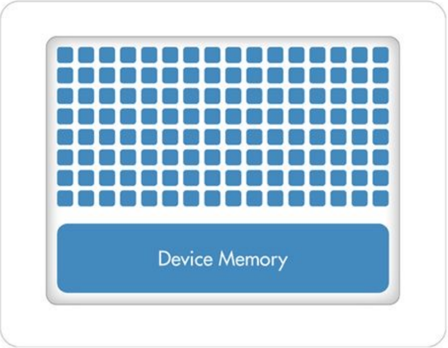


Image credit: MathWorks

High throughput
Ideal for parallel processing

JAX

What is a program transformation?

```
import jax
import jax.numpy as jnp

def f(x):
    return x**2

dfdx = jax.grad(f)# this returns a python function!

x = jnp.array(10.)

print(x)
print(dfdx(x))

---
10.0
20.0
```

Step 1: convert Python function into a simple intermediate language (jaxpr)

```
print(jax.make_jaxpr(f)(x))

---
{ lambda ; a:f32[]. let b:f32[] = integer_pow[y=2] a in (b,) }
```

Step 2: apply transformation (e.g. return the corresponding gradient function)

```
print(jax.make_jaxpr(dfdx)(x))

---
{ lambda ; a:f32[]. let
    _:f32[] = integer_pow[y=2] a
    b:f32[] = integer_pow[y=1] a
    c:f32[] = mul 2.0 b
    d:f32[] = mul 1.0 c
  in (d,) }
```

JAX Program transformations are composable

```
import jax
import jax.numpy as jnp

def f(x):
    return x**2

dfdx = jax.grad(f)# this returns a python function!

d2fdx2 = jax.grad(dfdx)# transformations are composable!

x = jnp.array(10.)

print(x)
print(d2fdx2(x))

---
10.0
2.0
```

- We can **arbitrarily compose** program transformations in JAX!
- This allows highly **sophisticated** workflows to be developed

JAX is not easy

- https://medium.com/swlh/solving-optimization-problems-with-jax-98376508bd4f

So start… with Autograd

- Read and finish the notebook called 03-introduction-to-autograd.ipynb and 04-autograd-applications.ipynb

Scipy and Sympy

https://members.cbio.mines-paristech.fr/~nvaroquaux/teaching/2016-image-xd/advanced/mathematical_optimization/index.html

https://members.cbio.mines-paristech.fr/~nvaroquaux/teaching/2016-image-xd/packages/sympy.html

Without knowledge of the gradient:	
	•In general, prefer BFGS (scipy.optimize.fmin_bfgs()) or L-BFGS (scipy.optimize.fmin_l_bfgs_b()), even if you have to approximate numerically gradients •On well-conditioned problems, Powell (scipy.optimize.fmin_powell()) and Nelder-Mead (scipy.optimize.fmin()), both gradient-free methods, work well in high dimension, but they collapse for ill-conditioned problems.
With knowledge of the gradient:	
	•BFGS (scipy.optimize.fmin_bfgs()) or L-BFGS (scipy.optimize.fmin_l_bfgs_b()). •Computational overhead of BFGS is larger than that L-BFGS, itself larger than that of conjugate gradient. On the other side, BFGS usually needs less function evaluations than CG. Thus conjugate gradient method is better than BFGS at optimizing computationally cheap functions.
With the Hessian:	
	•If you can compute the Hessian, prefer the Newton method (scipy.optimize.fmin_ncg()).
If you have noisy measurements:	
	•Use Nelder-Mead (scipy.optimize.fmin()) or Powell (scipy.optimize.fmin_powell()).