

# Automatic Differentiation for Computational Engineering

Kailai Xu and Eric Darve

# Outline

- 1 Overview
- 2 Computational Graph
- 3 Forward Mode
- 4 Reverse Mode
- 5 AD for Physical Simulation
- 6 AD Through Implicit Operators
- 7 Conclusion

# Overview

- Gradients are useful in many applications

- **Mathematical Optimization**

$$\min_{x \in \mathbb{R}^n} f(x)$$

Using the gradient descent method:

$$x_{n+1} = x_n - \alpha_n \nabla f(x_n)$$

- **Sensitivity Analysis**

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

- **Machine Learning**

Training a neural network using automatic differentiation (back-propagation).

- **Solving Nonlinear Equations** Solve a nonlinear equation  $f(x) = 0$  using Newton's method

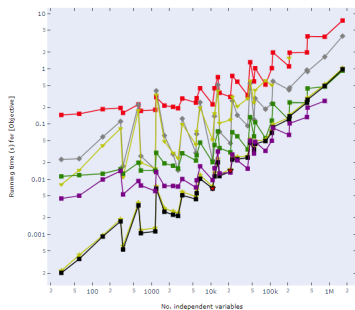
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

# Terminology

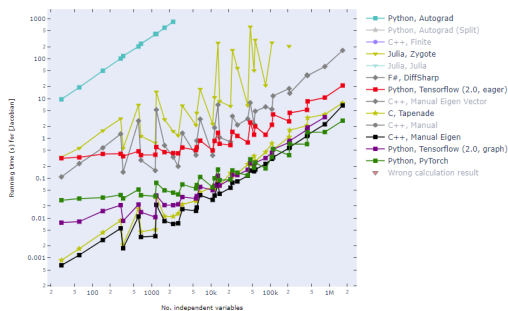
- Deriving and implementing gradients are a challenging and all-consuming process.
- Automatic differentiation: a set of techniques to numerically evaluate the derivative of a function specified by a computer program (Wikipedia). It also bears other names such as autodiff, algorithmic differentiation, computational differentiation, and back-propagation.
- There are a lot of AD softwares
  - ① TensorFlow and PyTorch: deep learning frameworks in Python
  - ② Adept-2: combined array and automatic differentiation library in C++
  - ③ autograd: efficiently derivatives computation of NumPy code.
  - ④ ForwardDiff.jl, Zygote.jl: Julia differentiable programming packages
- This lecture: how to compute gradients using automatic differentiation (AD)
  - Forward mode, reverse mode, and AD for implicit solvers

# AD Software

GMM (1k) [Objective] - Release



GMM (1k) [Jacobian] - Release



<https://github.com/microsoft/ADBench>

# Finite Differences

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \quad f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

- Derived from the definition of derivatives

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

- Conceptually simple.
- Curse of dimensionalities: to compute the gradients of  $f : \mathbb{R}^m \rightarrow \mathbb{R}$ , you need at least  $\mathcal{O}(m)$  function evaluations.
- Huge numerical error: roundoff error.

# Finite Difference

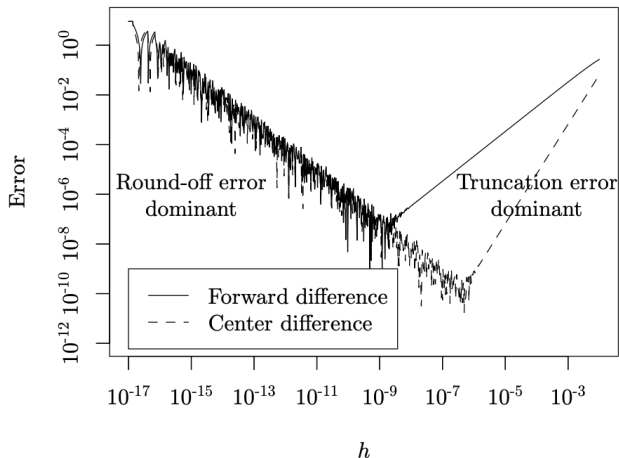
$$f(x) = \sin(x) \quad f'(x) = \cos(x) \quad x_0 = 0.1$$

```
f = x -> sin(x)
x0 = 0.1
e = cos(x0)
println("True derivative: $e")
printstyled("Forward Difference\tError\t\t\tCentral Difference\tError\n", bold=true)
for i = 1:10
    h = 1/10^i
    f1 = (f(x0+h)-f(x0))/h
    f2 = (f(x0+h)-f(x0-h))/2h
    e1 = abs(f1-e)
    e2 = abs(f2-e)
    println("$f1\t$e1\t$f2\t$e2")
end
```

True derivative: 0.9950041652780258

| Forward Difference | Error                  | Central Difference | Error                  |
|--------------------|------------------------|--------------------|------------------------|
| 0.9883591414823306 | 0.006645023795695204   | 0.9933466539753061 | 0.0016575113027197386  |
| 0.9944884190346656 | 0.00051574624336026    | 0.9949875819581878 | 1.6583319838003874e-5  |
| 0.994954082739849  | 5.008253817684327e-5   | 0.995003999444008  | 1.6583401785119634e-7  |
| 0.9949991719489237 | 4.993329102087607e-6   | 0.9950041636197504 | 1.6582754058802607e-9  |
| 0.9950036660946736 | 4.991833522094424e-7   | 0.9950041652613538 | 1.6671997711619105e-11 |
| 0.9950041153644618 | 4.991356405970038e-8   | 0.9950041652759256 | 2.1002088956834086e-1  |
| 2                  |                        |                    |                        |
| 0.9950041603146165 | 4.963409350189352e-9   | 0.9950041653106201 | 3.2594260623852733e-1  |
| 1                  |                        |                    |                        |
| 0.9950041651718422 | 1.0618361745429183e-10 | 0.9950041651718422 | 1.0618361745429183e-1  |
| 0                  |                        |                    |                        |
| 0.9950041623962845 | 2.8817412900394856e-9  | 0.9950041623962845 | 2.8817412900394856e-9  |
| 0.9950040791295578 | 8.614846802590392e-8   | 0.9950040791295578 | 8.614846802590392e-8   |

# Finite Difference



Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2017). Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1), 5595-5637.



# Symbolic Differentiation

- Symbolic differentiation computes exact derivatives (gradients): there is no approximation error.
- It works by recursively applying simple rules to **symbols**

$$\begin{aligned}\frac{d}{dx}(c) &= 0 & \frac{d}{dx}(x) &= 1 \\ \frac{d}{dx}(u + v) &= \frac{d}{dx}(u) + \frac{d}{dx}(v) & \frac{d}{dx}(uv) &= v \frac{d}{dx}(u) + u \frac{d}{dx}(v) \\ &\dots\end{aligned}$$

Here  $c$  is a variable independent of  $x$ , and  $u, v$  are variables dependent on  $x$ .

- There may not exist convenient expressions for the analytical gradients of some functions. For example, a blackbox function from a third-party library.

# Symbolic Differentiation

- Symbolic differentiation can lead to complex and redundant expressions

```
using SymPy
sigmoid = x -> 1/(1+exp(-x))
x,w1,w2,w3,b1,b2,b3 = @vars x w1 w2 w3 b1 b2 b3
y = w3*sigmoid(w2*sigmoid(w1*x+b1)+b2)+b3
dw1 = diff(y, w1)
```

$$\frac{w_2 w_3 x e^{-b_1 - w_1 x} e^{-b_2 - \frac{w_2}{e^{-b_1 - w_1 x} + 1}}}{(e^{-b_1 - w_1 x} + 1)^2 \left( e^{-b_2 - \frac{w_2}{e^{-b_1 - w_1 x} + 1}} + 1 \right)^2}$$

```
print(dw1)
```

```
w2*w3*x*exp(-b1 - w1*x)*exp(-b2 - w2/(exp(-b1 - w1*x) +
1))/((exp(-b1 - w1*x) + 1)^2*(exp(-b2 - w2/(exp(-b1 - w1*
x) + 1)) + 1)^2)
```

# Automatic Differentiation

- AD is neither finite difference nor symbolic differentiation.
- It works by recursively applies simple rules to **values**

$$\begin{aligned}\frac{d}{dx}(c) &= 0 & \frac{d}{dx}(x) &= 1 \\ \frac{d}{dx}(u + v) &= \frac{d}{dx}(u) + \frac{d}{dx}(v) & \frac{d}{dx}(uv) &= v \frac{d}{dx}(u) + u \frac{d}{dx}(v) \\ &\dots\end{aligned}$$

Here  $c$  is a variable independent of  $x$ , and  $u, v$  are variables dependent on  $x$ .

- It evaluates numerically gradients of “function units” using symbolic differentiation, and chains the computed gradients using the chain rule

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

- It is efficient (linear in the cost of computing the function itself) and numerically stable.

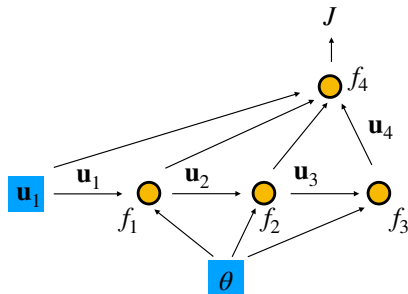
# Outline

- 1 Overview
- 2 Computational Graph**
- 3 Forward Mode
- 4 Reverse Mode
- 5 AD for Physical Simulation
- 6 AD Through Implicit Operators
- 7 Conclusion

# Computational Graph

- The “language” for automatic differentiation is computational graph.
  - The computational graph is a **directed acyclic graph (DAG)**.
  - Each **edge** represents the data: a scalar, a vector, a matrix, or a high dimensional tensor.
  - Each **node** is a function that consumes several incoming edges and outputs some values.

$$\begin{aligned}J &= f_4(u_1, u_2, u_3, u_4), \\u_2 &= f_1(u_1, \theta), \\u_3 &= f_2(u_2, \theta), \\u_4 &= f_3(u_3, \theta).\end{aligned}$$

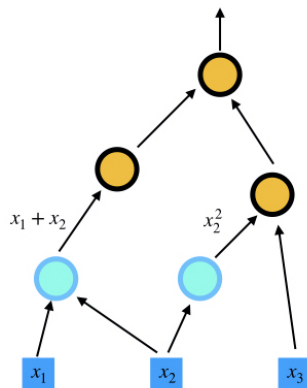


- Let's build a computational graph for computing

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

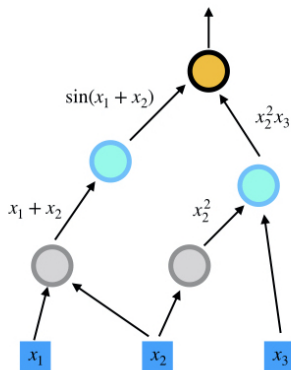
# Building a Computational Graph

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



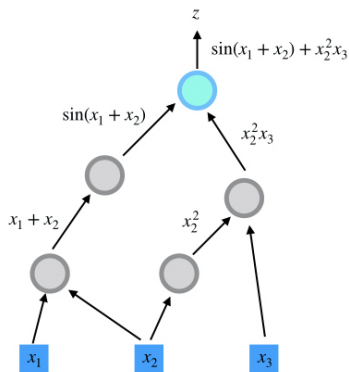
# Building a Computational Graph

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



# Building a Computational Graph

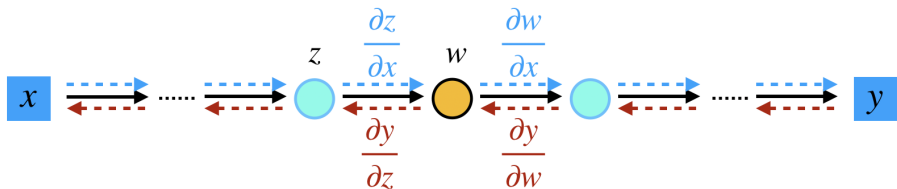
$$z = \sin(x_1 + x_2) + x_2^2 x_3$$





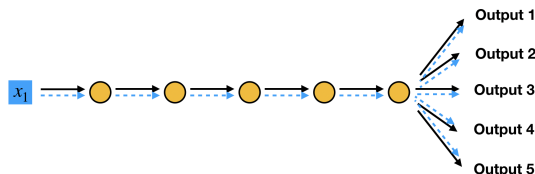
# Computing Gradients from a Computational Graph

- Automatic differentiation works by **propagating** gradients in the computational graph.
- Two basic modes: forward-mode and backward-mode. Forward-mode propagates gradients in the same direction as forward computation. Backward-mode propagates gradients in the reverse direction of forward computation.

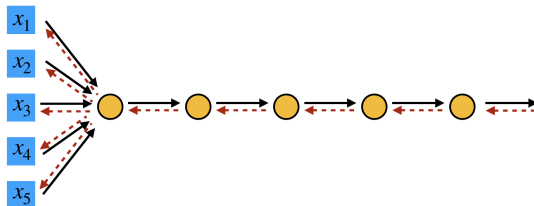


# Computing Gradients from a Computational Graph

- Different computational graph topologies call for different modes of automatic differentiation.
  - One-to-many: forward-propagation  $\Rightarrow$  forward-mode AD.



- Many-to-one: back-propagation  $\Rightarrow$  reverse-mode AD.



# Outline

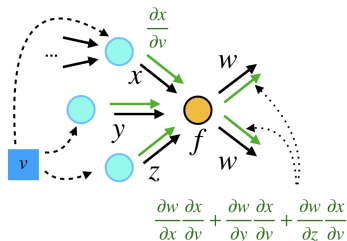
- 1 Overview
- 2 Computational Graph
- 3 Forward Mode**
- 4 Reverse Mode
- 5 AD for Physical Simulation
- 6 AD Through Implicit Operators
- 7 Conclusion

# Automatic Differentiation: Forward Mode AD

- The forward-mode automatic differentiation uses the chain rule to propagate the gradients.

$$\frac{\partial f \circ g(x)}{\partial x} = f'(g(x))g'(x)$$

- Compute in the same order as function evaluation.
- Each node in the computational graph
  - **Aggregate** all the gradients from up-streams.
  - **Forward** the gradient to down-stream nodes.



## Example: Forward Mode AD

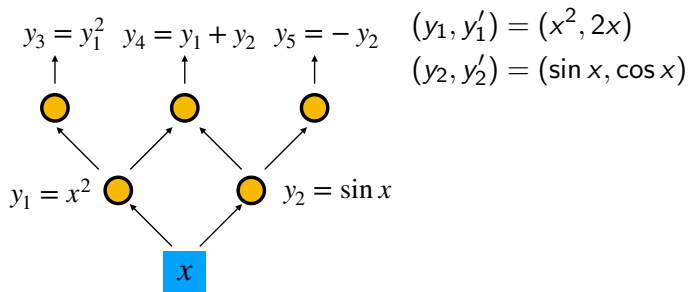
- Let's consider a specific way for computing

$$f(x) = \begin{bmatrix} x^4 \\ x^2 + \sin(x) \\ -\sin(x) \end{bmatrix}$$

## Example: Forward Mode AD

- Let's consider a specific way for computing

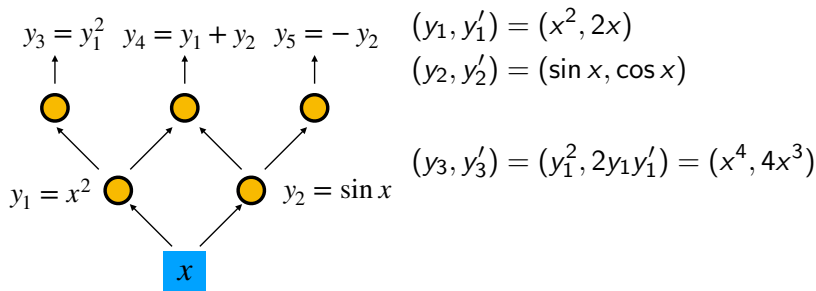
$$f(x) = \begin{bmatrix} x^4 \\ x^2 + \sin(x) \\ -\sin(x) \end{bmatrix}$$



## Example: Forward Mode AD

- Let's consider a specific way for computing

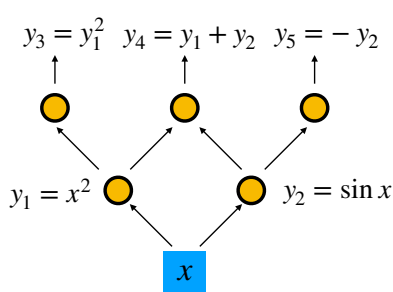
$$f(x) = \begin{bmatrix} x^4 \\ x^2 + \sin(x) \\ -\sin(x) \end{bmatrix}$$



## Example: Forward Mode AD

- Let's consider a specific way for computing

$$f(x) = \begin{bmatrix} x^4 \\ x^2 + \sin(x) \\ -\sin(x) \end{bmatrix}$$



$$(y_1, y'_1) = (x^2, 2x)$$

$$(y_2, y'_2) = (\sin x, \cos x)$$

$$(y_3, y'_3) = (y_1^2, 2y_1 y'_1) = (x^4, 4x^3)$$

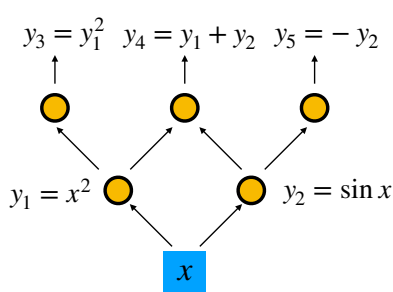
$$\begin{aligned} (y_4, y'_4) &= (y_1 + y_2, y'_1 + y'_2) \\ &= (x^2 + \sin x, 2x + \cos x) \end{aligned}$$



## Example: Forward Mode AD

- Let's consider a specific way for computing

$$f(x) = \begin{bmatrix} x^4 \\ x^2 + \sin(x) \\ -\sin(x) \end{bmatrix}$$



$$(y_1, y'_1) = (x^2, 2x)$$

$$(y_2, y'_2) = (\sin x, \cos x)$$

$$(y_3, y'_3) = (y_1^2, 2y_1 y'_1) = (x^4, 4x^3)$$

$$(y_4, y'_4) = (y_1 + y_2, y'_1 + y'_2) \\ = (x^2 + \sin x, 2x + \cos x)$$

$$(y_5, y'_5) = (-y_2, -y'_2) = (-\sin x, -\cos x)$$

# Summary

- Forward mode AD reuses gradients from upstreams. Therefore, this mode is useful for few-to-many mappings

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m, n \ll m$$

- Applications: sensitivity analysis, uncertainty quantification, etc.
  - Consider a physical model  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ , let  $x \in \mathbb{R}^n$  be the quantity of interest (usually a low dimensional physical parameter), uncertainty propagation method computes the perturbation of the model output (usually a large dimensional quantity, i.e.,  $m \gg 1$ )

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

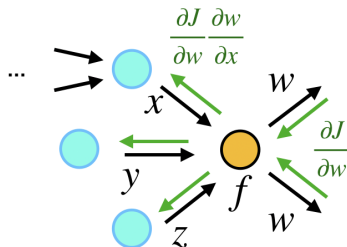
# Outline

- 1 Overview
- 2 Computational Graph
- 3 Forward Mode
- 4 Reverse Mode**
- 5 AD for Physical Simulation
- 6 AD Through Implicit Operators
- 7 Conclusion

# Reverse Mode AD

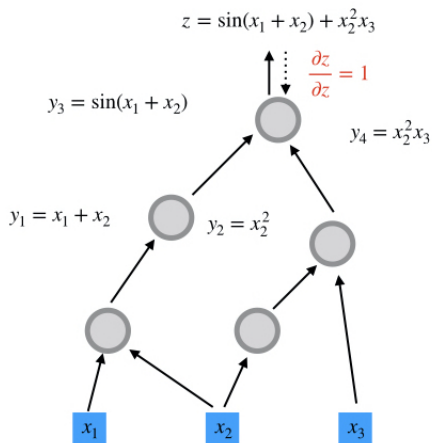
$$\frac{df(g(x))}{dx} = f'(g(x))g'(x)$$

- Computing in the reverse order of forward computation.
- Each node in the computational graph
  - **Aggregates** all the gradients from down-streams
  - **Back-propagates** the gradient to upstream nodes.



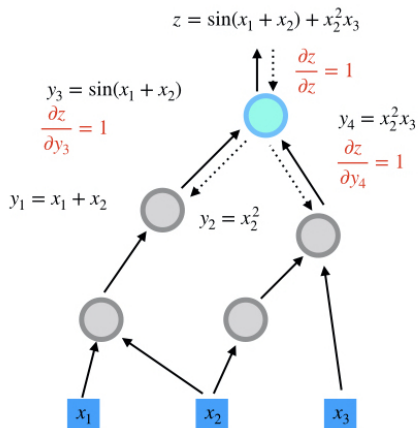
# Example: Reverse Mode AD

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



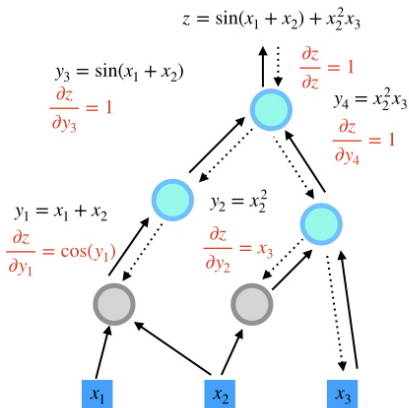
# Example: Reverse Mode AD

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



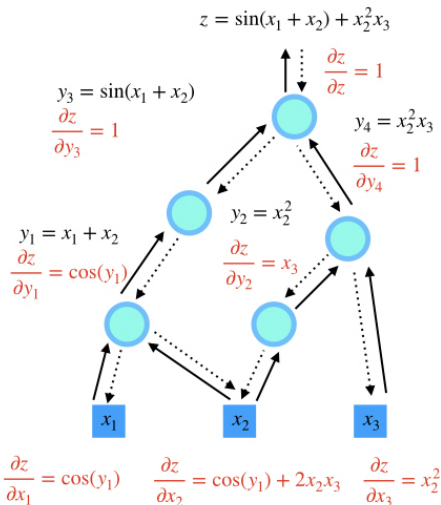
## Example: Reverse Mode AD

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



# Example: Reverse Mode AD

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





# Summary

- Reverse mode AD reuses gradients from down-streams. Therefore, this mode is useful for many-to-few mappings

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m, n \gg m$$

- Typical application:
  - Deep learning:  $n$  = total number of weights and biases of the neural network,  $m = 1$  (loss function).
  - Mathematical optimization: usually there are only a single objective function.

# Summary

- In general, for a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

| Mode    | Suitable for ... | Complexity <sup>1</sup>      | Application      |
|---------|------------------|------------------------------|------------------|
| Forward | $m \gg n$        | $\leq 2.5 \text{ OPS}(f(x))$ | UQ               |
| Reverse | $m \ll n$        | $\leq 4 \text{ OPS}(f(x))$   | Inverse Modeling |

- There are also many other interesting topics
  - Mixed mode AD: many-to-many mappings.
  - Computing sparse Jacobian matrices using AD by exploiting sparse structures.

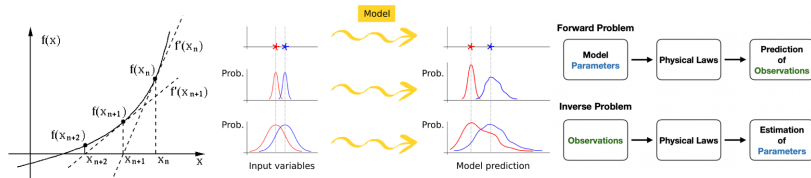
Margossian CC. A review of automatic differentiation and its efficient implementation. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery. 2019 Jul;9(4):e1305.

<sup>1</sup>OPS is a metric for complexity in terms of fused-multiply-adds.

# Outline

- 1 Overview
- 2 Computational Graph
- 3 Forward Mode
- 4 Reverse Mode
- 5 AD for Physical Simulation**
- 6 AD Through Implicit Operators
- 7 Conclusion

# The Demand for Gradients in Physical Simulation



- Solving nonlinear equations
- Uncertainty quantification/sensitivity analysis
- Inverse problems

Image source:

<https://mirams.wordpress.com/2016/11/23/uncertainty-in-risk-prediction/>,  
<http://fourier.eng.hmc.edu/e176/lectures/ch2/node5.html>

# Inverse Problem and Mathematical Optimization

- Consider a bar under heating with a source term  $f(x, t)$ . The right hand side has fixed temperature and the left hand side is insulated.
- The governing equation for the temperature  $u(x, t)$  is

$$\begin{aligned}\frac{\partial u(x, t)}{\partial t} &= \kappa(x) \Delta u(x, t) + f(x, t), \quad t \in (0, T), x \in \Omega \\ u(1, t) &= 0 \quad t > 0 \\ \kappa(0) \frac{\partial u(0, t)}{\partial x} &= 0 \quad t > 0\end{aligned}$$

- The diffusivity coefficient is given by

$$\kappa(x) = a + bx$$

where  $a$  and  $b$  are unknown parameters.

# Inverse Problem and Mathematical Optimization

- Goal: calibrate  $a$  and  $b$  from  $u_0(t) = u(0, t)$

$$\kappa(x) = a + bx$$

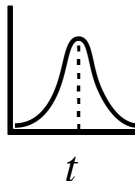
$$\frac{\partial u(x, t)}{\partial t} = \kappa(x) \Delta u(x, t) + f(x, t)$$

$$\kappa(0) \frac{\partial u(0, t)}{\partial x} = 0$$

$$u(1, t) = 0$$



$u(0, t)$



$a, b$

# Inverse Problem and Mathematical Optimization

- This problem is a standard inverse problem. We can formulate the problem as a PDE-constrained optimization problem

$$\begin{aligned} \min_{a,b} \quad & \int_0^t (u(0, t) - u_0(t))^2 dt \\ \text{s.t.} \quad & \frac{\partial u(x, t)}{\partial t} = \kappa(x) \Delta u(x, t) + f(x, t), \quad t \in (0, T), x \in (0, 1) \\ & -\kappa(0) \frac{\partial u(0, t)}{\partial x} = 0, t > 0 \\ & u(1, t) = 0, t > 0 \\ & u(x, 0) = 0, x \in [0, 1] \\ & \kappa(x) = ax + b \end{aligned}$$

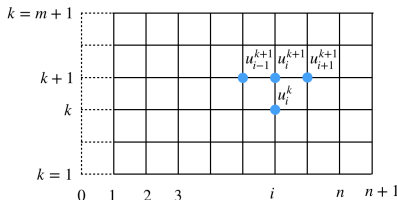
# Numerical Partial Differential Equation

- As with many physical modeling techniques, we discretize the PDE using numerical schemes. Here is a finite difference scheme for the PDE  $k = 1, 2, \dots, m, i = 1, 2, \dots, n$

$$\frac{u_i^{k+1} - u_i^k}{\Delta t} = \kappa_i \frac{u_{i+1}^{k+1} + u_{i-1}^{k+1} - 2u_i^{k+1}}{\Delta x^2} + f_i^{k+1}$$

For initial and boundary conditions, we have

$$\begin{aligned} -\kappa_1 \frac{u_2^k - u_0^k}{2\Delta x} &= 0 \\ u_{n+1}^k &= 0 \\ u_i^0 &= 0 \end{aligned}$$





# Numerical Partial Differential Equation

- Rewriting the equation as a linear system, we have

$$A(a, b)U^{k+1} = U^k + F^{k+1}, \quad U^k = \begin{bmatrix} u_1^k \\ u_2^k \\ \vdots \\ u_n^k \end{bmatrix}$$

Here  $\lambda_i = -\kappa_i \frac{\Delta t}{\Delta x^2}$  and

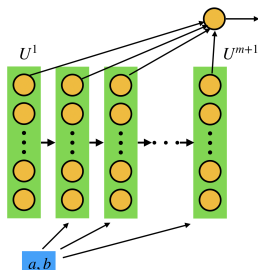
$$A(a, b) = \begin{bmatrix} 2\lambda_1 + 1 & -\lambda_1 & & & \\ -\lambda_2 & 2\lambda_2 + 1 & & & \\ & -\lambda_3 & 2\lambda_3 + 1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & -\lambda_{n-1} \\ -\lambda_n & & & & 2\lambda_n + 1 \end{bmatrix}, \quad F^k = \Delta t \begin{bmatrix} f_1^{k+1} \\ f_2^{k+1} \\ \vdots \\ f_n^{k+1} \end{bmatrix}$$

# Computational Graph for Numerical Schemes

- The discretized optimization problem is

$$\begin{aligned} \min_{a,b} \quad & \sum_{k=1}^m (u_1^k - u_0((k-1)\Delta t))^2 \\ \text{s.t.} \quad & A(a,b)U^{k+1} = U^k + F^{k+1}, k = 1, 2, \dots, m \\ & U^0 = 0 \end{aligned}$$

- The computational graph for the forward computation (evaluating the loss function) is



# Implementation using an AD system

```
function condition(i, u_arr)
    i<=m+1
end

function body(i, u_arr)
    u = read(u_arr, i-1)
    rhs = u + F[i]
    u_next = A\rhs
    u_arr = write(u_arr, i, u_next)
    i+1, u_arr
end

F = constant(F)
u_arr = TensorArray(m+1)
u_arr = write(u_arr, 1, zeros(n))
i = constant(2, dtype=Int32)
_, u = while_loop(condition, body, [i, u_arr])
u = set_shape(stack(u), (m+1, n))

uc = readlm("data.txt")[:]

loss = sum((uc-u[:,1])^2) * 1e10

sess = Session(); init(sess)
BFGS!(sess, loss)
```

Simulation Loop

**You will have chance to  
Practice in your homework!  
(TensorFlow/PyTorch, ADCME,  
or any other AD tools)**

Formulate  
Loss Function

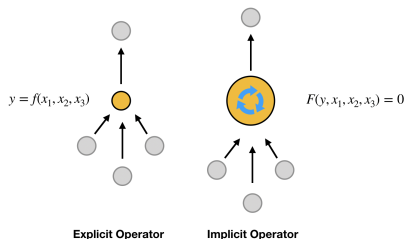
Gradient Computation  
Optimization

# Outline

- 1 Overview
- 2 Computational Graph
- 3 Forward Mode
- 4 Reverse Mode
- 5 AD for Physical Simulation
- 6 AD Through Implicit Operators**
- 7 Conclusion

# Challenges in AD

- Most AD frameworks only deal with explicit operators, i.e., the functions that has analytical derivatives, or composition of these functions.
- Many scientific computing algorithms are **iterative** or **implicit** in nature.



| Linear/Nonlinear | Explicit/Implicit | Expression    |
|------------------|-------------------|---------------|
| Linear           | Explicit          | $y = Ax$      |
| Nonlinear        | Explicit          | $y = F(x)$    |
| <b>Linear</b>    | <b>Implicit</b>   | $Ay = x$      |
| <b>Nonlinear</b> | <b>Implicit</b>   | $F(x, y) = 0$ |

## Example

- Consider a function  $f : x \rightarrow y$ , which is implicitly defined by

$$F(x, y) = x^3 - (y^3 + y) = 0$$

If not using the cubic formula for finding the roots, the forward computation consists of iterative algorithms, such as the Newton's method and bisection method

$$y^0 \leftarrow 0$$

$$k \leftarrow 0$$

**while**  $|F(x, y^k)| > \epsilon$  **do**

$$\delta^k \leftarrow F(x, y^k) / F'_y(x, y^k)$$

$$y^{k+1} \leftarrow y^k - \delta^k$$

$$k \leftarrow k + 1$$

**end while**

**Return**  $y^k$

$$l \leftarrow -M, r \leftarrow M, m \leftarrow 0$$

**while**  $|F(x, m)| > \epsilon$  **do**

$$c \leftarrow \frac{a+b}{2}$$

**if**  $F(x, m) > 0$  **then**

$$a \leftarrow m$$

**else**

$$b \leftarrow m$$

**end if**

**end while**

**Return**  $c$

## Example

- An efficient way is to apply the **implicit function theorem**. For our example,  $F(x, y) = x^3 - (y^3 + y) = 0$ , treat  $y$  as a function of  $x$  and take the derivative on both sides

$$3x^2 - 3y(x)^2 y'(x) - 1 = 0 \Rightarrow y'(x) = \frac{3x^2 - 1}{3y(x)^2}$$

The above gradient is **exact**.

# Implicit Operators in Physical Modeling

- Return to our bar problem, what if the material property is complex and has a temperature-dependent governing equation?

$$\frac{\partial u(x, t)}{\partial t} = \kappa(u) \Delta u(x, t) + f(x, t), \quad t \in (0, T), x \in \Omega$$

- An implicit scheme is usually a nonlinear equation, and requires an iterative solver (e.g., the Newton-Raphson algorithm) to solve

$$\frac{u_i^{k+1} - u_i^k}{\Delta t} = \kappa(u_i^{k+1}) \frac{u_{i+1}^{k+1} + u_{i-1}^{k+1} - 2u_i^{k+1}}{\Delta x^2} + f_i^{k+1}$$

- Typical AD frameworks cannot handle this operator. We need to differentiate through implicit operators.
- This topic will be covered in a future lecture: **physics constrained learning**.



# Outline

- 1 Overview
- 2 Computational Graph
- 3 Forward Mode
- 4 Reverse Mode
- 5 AD for Physical Simulation
- 6 AD Through Implicit Operators
- 7 Conclusion**

# Conclusion

- What's covered in this lecture
  - Reverse mode automatic differentiation;
  - Forward mode automatic differentiation;
  - Using AD to solve inverse problems in physical modeling;
  - Automatic differentiation through implicit operators.

# What's Next

- Physics constrained learning: inverse modeling using automatic differentiation through implicit operators;
- Neural networks and numerical schemes: substitute the unknown component in a physical system with a neural network and learn the neural network with AD;
- Implementation of inverse modeling algorithms in ADCME.