Introduction to Automatic Differentiation

Johannes Willkomm

PLEIAD Seminar, Universidad de Chile

27 Nov. 2009

Santiago de Chile

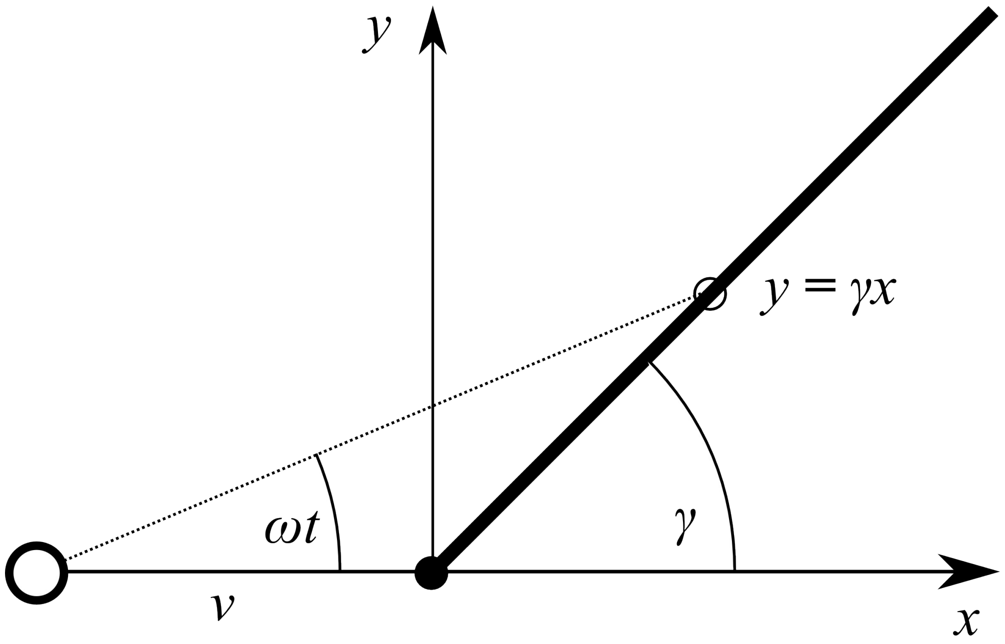
Outline of the talk

- Automatic Differentiation (AD)
  - Definition by example
  - Forward and reverse mode
  - Scalar and vector mode
- AD implementation
  - Source transformation and operator overloading
  - Reverse mode example
  - Tools
- Alternatives to AD
  - Divided differences, Complex-Variable method
  - Symbolic and manual differentiation
- Summary

- **Automatic** or **Algorithmic Differentiation** (AD)
  - Given a numeric program, that implements function F
  - AD creates a new program that computes F', the first order derivative of F
  - And sometimes also the higher order derivatives F'', F''', F^{IV}, etc.

Lighthouse example

- Consider the beam of a lighthouse rotating with angular velocity ω as it runs along a quay with slope γ at distance ν, as a function of time t



- The coordinates of the point where the light hits the quay are given by

$$x = \frac{\nu \tan(\omega t)}{\gamma - \tan(\omega t)} \qquad y = \frac{\gamma\nu \tan(\omega t)}{\gamma - \tan(\omega t)}$$

- A program implementing this function

$$\begin{aligned}
v_1 &= \omega * t;\\
v_2 &= \tan(v_1);\\
v_3 &= \gamma - v_2;\\
v_4 &= \nu * v_2;\\
x &= v_4 / v_3;\\
y &= \gamma * x;
\end{aligned}$$

Forward mode example

- Program code can be mechanically differentiated
  - Differentiate each statement and insert it before the original statement

$$\begin{aligned} v_1 &= \omega * t; \\ v_2 &= \tan(v_1); \\ v_3 &= \gamma - v_2; \\ v_4 &= \nu * v_2; \\ x &= v_4 / v_3; \\ y &= \gamma * x; \end{aligned}$$

$$\begin{aligned} \delta v_1 &= \delta\omega * t + \omega * \delta t; \\ v_1 &= \omega * t; \\ \delta v_2 &= \delta v_1 / \cos^2(v_1); \\ v_2 &= \tan(v_1); \\ \delta v_3 &= \delta\gamma - \delta v_2; \\ v_3 &= \gamma - v_2; \\ \delta v_4 &= \delta\nu * v_2 + \nu * \delta v_2; \\ v_4 &= \nu * v_2; \\ \delta x &= \delta v_4 / v_3 + v_4 \delta v_3 / v_3^2; \\ x &= v_4 / v_3; \\ \delta y &= \delta\gamma * x + \gamma * \delta x; \\ y &= \gamma * x; \end{aligned}$$

Running the AD code

- The AD code has new input and output variables

  δt, $\delta \gamma$, δv, and $\delta \omega$ are new inputs

  δx, δy are new results

- The user must set the input derivatives
  - $\delta t = \mathrm{d}t/\mathrm{d}p$, $\delta \gamma = \mathrm{d}\gamma/\mathrm{d}p$, $\delta v = \mathrm{d}v/\mathrm{d}p$, and $\delta \omega = \mathrm{d}\omega/\mathrm{d}p$, where p is the parameter to differentiate to

- Examples:
  - Setting $\delta t = 1$, $\delta \gamma = 0$, $\delta v = 0$, and $\delta \omega = 0$, the AD code computes $\mathrm{d}x/\mathrm{d}t$ and $\mathrm{d}y/\mathrm{d}t$
  - Setting $\delta t = 0$, $\delta \gamma = 1$, $\delta v = 0$, and $\delta \omega = 0$, the AD code computes $\mathrm{d}x/\mathrm{d}\gamma$ and $\mathrm{d}y/\mathrm{d}\gamma$, etc.

- To get all eight derivatives, the code must be run four times: this is the scalar forward mode

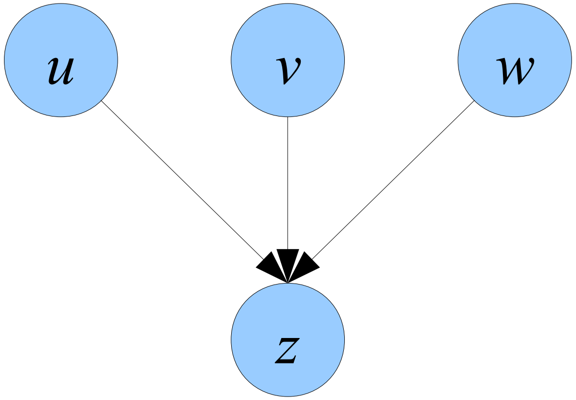
- We can also transform the derivative variables into vectors
  - Using 4-vectors we can compute all derivatives at once
- Example
  - Set $\delta t = [1,0,0,0]$, $\delta \gamma = [0,1,0,0]$, $\delta v = [0,0,1,0]$, and $\delta \omega = [0,0,0,1]$
  - As the result we obtain the full Jacobian matrix $J = \mathrm{D}F$

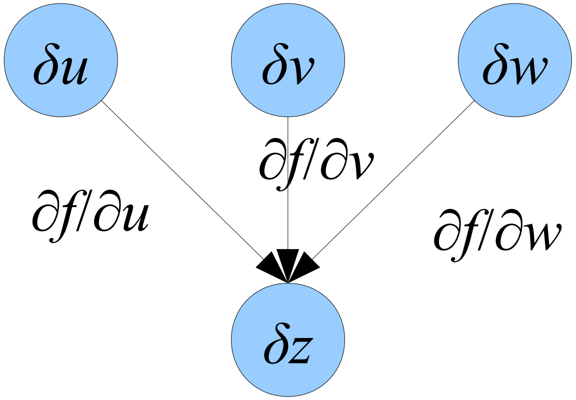
$$\delta x = \left[\frac{\mathrm{d}x}{\mathrm{d}t}, \frac{\mathrm{d}x}{\mathrm{d}\gamma}, \frac{\mathrm{d}x}{\mathrm{d}\nu}, \frac{\mathrm{d}x}{\mathrm{d}\omega}\right]$$

$$\delta y = \left[\frac{\mathrm{d}y}{\mathrm{d}t}, \frac{\mathrm{d}y}{\mathrm{d}\gamma}, \frac{\mathrm{d}y}{\mathrm{d}\nu}, \frac{\mathrm{d}y}{\mathrm{d}\omega}\right]$$

- To differentiate a program
  - Create new variable δv for each program variable v
  - Differentiate each statement and insert it before the original statement
  - Each δv holds the derivative $\mathrm{d}v/\mathrm{d}p$ of v w.r.t. the input parameter p

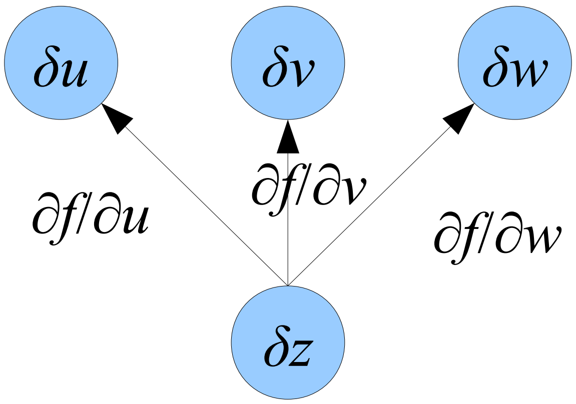
$$z = f(u, v, w); \qquad \delta z = \frac{\partial f}{\partial u}\delta u + \frac{\partial f}{\partial v}\delta v + \frac{\partial f}{\partial w}\delta w;$$





- AD is also possible by running the program backwards
- For each statement we propagate the derivative of the LHS to the derivatives of the variables on the RHS
  - Create the so-called adjoint statements

$$\delta u = \delta u + \frac{\partial f}{\partial u}\delta z;$$
$$\delta v = \delta u + \frac{\partial f}{\partial v}\delta z;$$
$$\delta w = \delta u + \frac{\partial f}{\partial w}\delta z;$$



- Forward sweep
  - The program is executed, saving all variable values
- Initialize adjoints
  - Initialize all derivative variables δv to zero
- Return sweep
  - Execute the adjoint statements in reverse order
  - Now, at any one time, δv contains the adjoint $\mathrm{d}f/\mathrm{d}v$ of v

Lighthouse in reverse

- Run code
- Zero adjoints
- Run adjoint code

$$
\begin{aligned}
v_1 &= \omega * t; \\
v_2 &= \tan(v_1); \\
v_3 &= \gamma - v_2; \\
v_4 &= \nu * v_2; \\
x &= v_4 / v_3; \\
y &= \gamma * x;
\end{aligned}
$$

$$
\begin{aligned}
\delta t &= 0; \\
\delta\gamma &= 0; \\
\delta\nu &= 0; \\
\delta\omega &= 0; \\
\delta v_1 &= 0; \\
\delta v_2 &= 0; \\
\delta v_3 &= 0; \\
\delta v_4 &= 0;
\end{aligned}
$$

$$
\begin{aligned}
\delta x &+= \gamma * \delta y; \\
\delta\gamma &+= x * \delta y; \\
\delta v_4 &+= \delta x / v_3; \\
\delta v_3 &+= -v_4 / v_3^2 * \delta x; \\
\delta\nu &+= v_2 \delta v_4; \\
\delta v_2 &+= \nu \delta v_4; \\
\delta\gamma &+= \delta v_3; \\
\delta v_2 &+= -\delta v_3; \\
\delta v_1 &+= \delta v_2 / \cos^2(v_1); \\
\delta\omega &+= t * \delta v_1; \\
\delta t &+= \omega * \delta v_1;
\end{aligned}
$$

Running reverse mode AD code

- The adjoint code has new in- and outputs

  δx, δy are new inputs

  δt, $\delta \gamma$, δv, and $\delta \omega$ are new results
- Values for δx and δy are supplied by the user

  $\delta x = \mathrm{d}x/\mathrm{d}r$ and $\delta y = \mathrm{d}y/\mathrm{d}r$ where r is the result to differentiate
- Example
  - Setting $\delta x = 1$ and $\delta y = 0$, the code computes $\mathrm{d}x/\mathrm{d}t$, $\mathrm{d}x/\mathrm{d}\gamma$, $\mathrm{d}x/\mathrm{d}v$, and $\mathrm{d}x/\mathrm{d}\omega$
  - Setting $\delta x = 0$ and $\delta y = 1$, the code computes $\mathrm{d}y/\mathrm{d}t$, $\mathrm{d}y/\mathrm{d}\gamma$, $\mathrm{d}y/\mathrm{d}v$, and $\mathrm{d}y/\mathrm{d}\omega$
- To get all eight derivatives, the code must be run twice, or with 2-vectors as input adjoints

- Given a function

$$\mathbf{y} = F(\mathbf{x}), \quad F : \mathbb{R}^n \to \mathbb{R}^m$$

  - First order AD computes the Jacobian

$$J = \mathrm{D}F \in \mathbb{R}^{m \times n}$$

  - Or products thereof

- AD in forward mode

$$J \cdot S, \quad S \in \mathbb{R}^{n \times p}$$

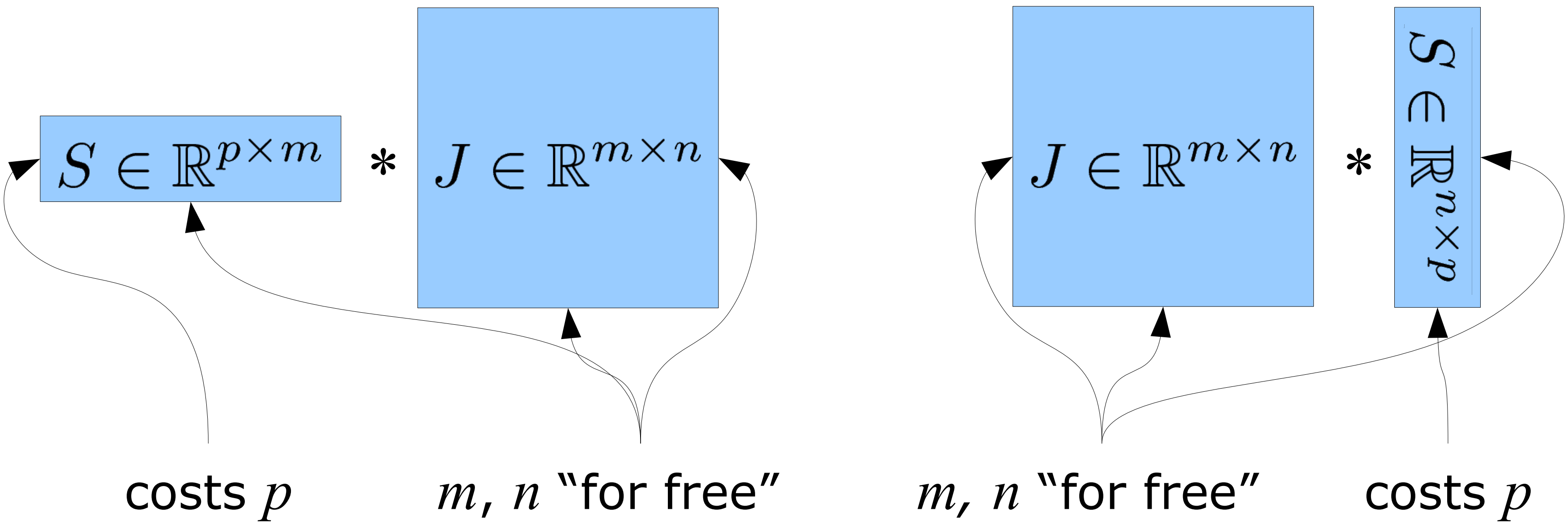
  - Computes Jacobian times vector or Jacobian time matrix products

- AD in reverse mode

$$S \cdot J, \quad S \in \mathbb{R}^{p \times m}$$

  - Computes vector times Jacobian or matrix times Jacobian products

- The time complexity depends on the number of rows or columns in S and the runtime T_F of F
  - Computing J has $T_F \mathrm{O}(m)$ in RM and $T_F \mathrm{O}(n)$ in FM
  - The c in O is $3 < \mathrm{c} < 50$, depending on tool & strategy



- Space complexity is $\mathrm{O}(T_F)$ in RM!

- Source transformation
  - New program text is generated
  - Higher order derivatives often not directly supported, but by repeatedly applying the tool
- Operator Overloading
  - Numeric data type (**double**) is replaced by new type
  - Tapeless: Derivatives are stored inside the active variables and updated on the fly
    - Forward mode only
  - With Taping: Computations are first recorded on a so-called Tape, which is then read (forwards or backwards) to compute the derivatives
  - Higher order derivatives are not much more difficult to implement than first order

- Compute polynomial of order n

$$F(x, \mathbf{c}) = \prod_{i=0}^{n} c_i x^i$$

- A C-style implementation in MATLAB
  - If x, c_i are all scalars that could also be a one-liner

```
function r = polynom(x, c)
  r = 0;
  powX = 1;
  for i=1:length(c)
    r = r + c(i) .* powX;
    powX = powX .* x;
  end
```

Source transformation RM

```
function [a_x a_c nr_r] = a_polynom(x, c, a_r)
    tmpc1 = 0;
    r = 0;
    powX = 1;
    tmpf1 = length(c);
    for i=1 : tmpf1
        push(tmpc1);
        tmpc1 = c(i) .* powX;
        push(r);
        r = r + tmpc1;
        push(powX);
        powX = powX .* x;
    end
    push(tmpf1);
    nr_r = r;
```

- Forward sweep
  - Run (canonicalized) code
  - Save all values overwritten
  - Save control flow

```
    [a_powX a_tmpc1] = a_zeros(powX, tmpc1);
    [a_x a_c] = a_zeros(x, c);
    if nargin < 3
        [a_r] = a_zeros(r);
    end
    [tmpf1] = pop;
    for i=fliplr(1 : tmpf1)
        [powX] = pop;
        a_x = a_x + adjred(x, powX .* a_powX);
        a_powX = adjred(powX, a_powX .* x);
        [r] = pop;
        a_tmpc1 = a_tmpc1 + adjred(tmpc1, a_r);
        a_r = adjred(r, a_r);
        [tmpc1] = pop;
        a_c(i) = a_c(i) + adjred(c(i), a_tmpc1 .* powX);
        a_powX = a_powX + adjred(powX, c(i) .* a_tmpc1);
        [a_tmpc1] = a_zeros(tmpc1);
    end
end
```

- Return sweep
  - Zero adjoints
  - Run backwards
  - Compute adjoints

AD tools

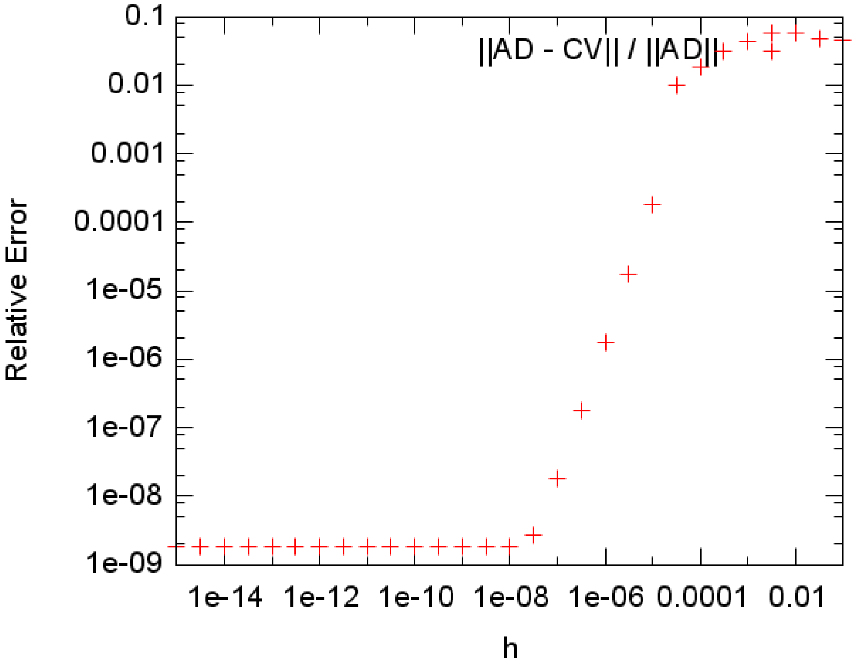
Tool	Language	FM	RM	ST	OO
ADOL-C	C/C++	✓	✓		✓
CppAD	C/C++	✓	✓		✓
ADiFor	Fortran 77	✓		✓	
Tapenade	Fortran 77, Fortran 90/95	✓	✓	✓	
ADiMat	Matlab	✓	✓	✓	
MAD	Matlab	✓			✓
ADiCape	CapeML	✓		✓	

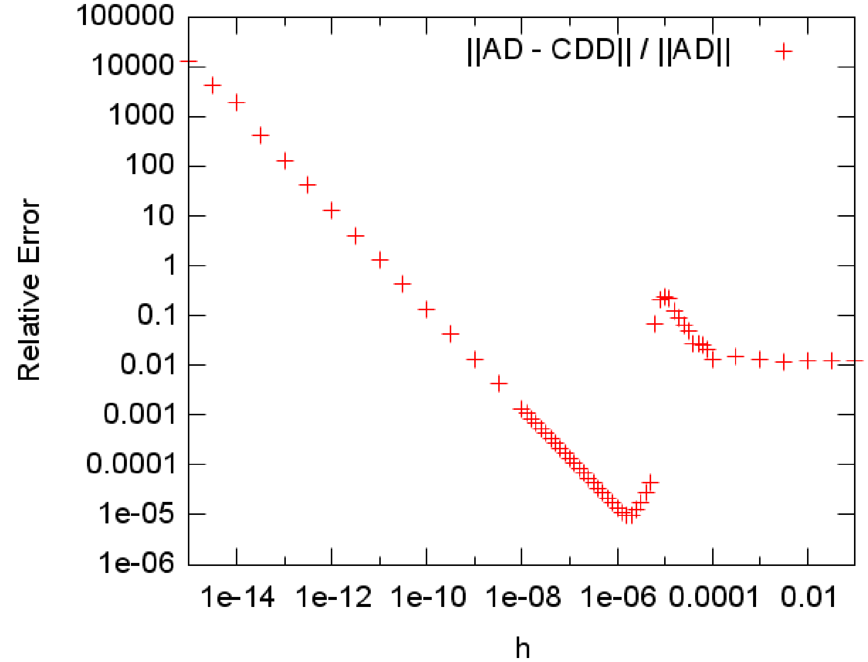
Alternative ways to compute derivatives

- Divided differences $$\frac{\mathrm{d}f}{\mathrm{d}x_i} \approx \frac{f(\mathbf{x}+h\mathbf{e}_i) - f(\mathbf{x}-h\mathbf{e}_i)}{2h}$$
  - Very inaccurate
  - Difficult to find the right value for h
  - ✔ Only function F is required
  - Only $J\mathbf{v}$ with complexity $\mathrm{O}(n)$

- Complex variable method $$\frac{\mathrm{d}f}{\mathrm{d}x_k} = \frac{\Im\{f(\mathbf{x}+h\mathbf{i}\mathbf{e}_k)\}}{h}$$
  - Program needs to be changed similar to AD with OO
  - ✔ Derivatives are exact, if h is just small enough
  - Need to provide new operations >, <, abs
  - Only $J\mathbf{v}$ with complexity $\mathrm{O}(n)$

CV-Method vs. DD





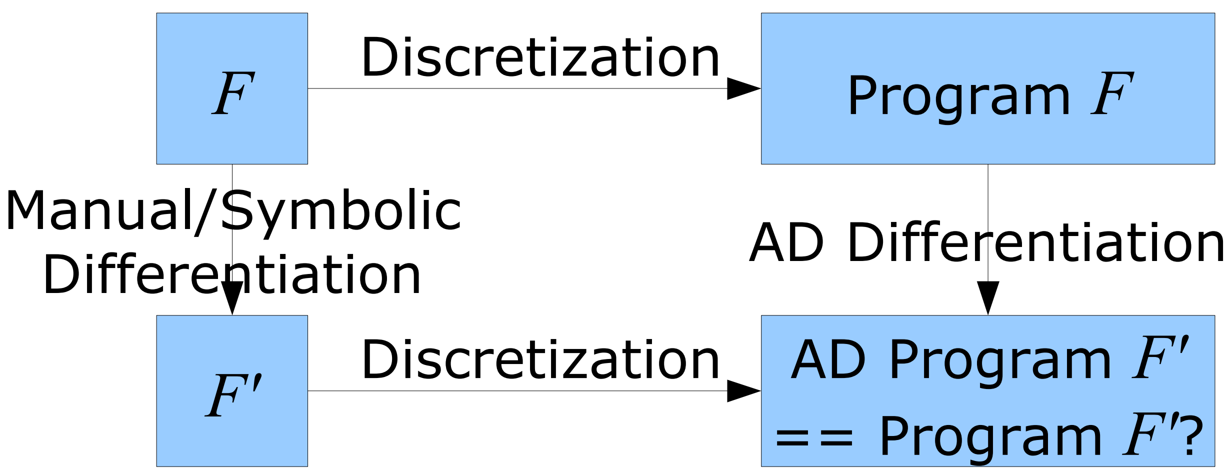
- The CV-Method is more precise
  - Usually up to machine precision
- And it is safer to use
  - Just set h to a very small value, e.g. $h = 10^{-60}$

Alternative ways to compute derivatives

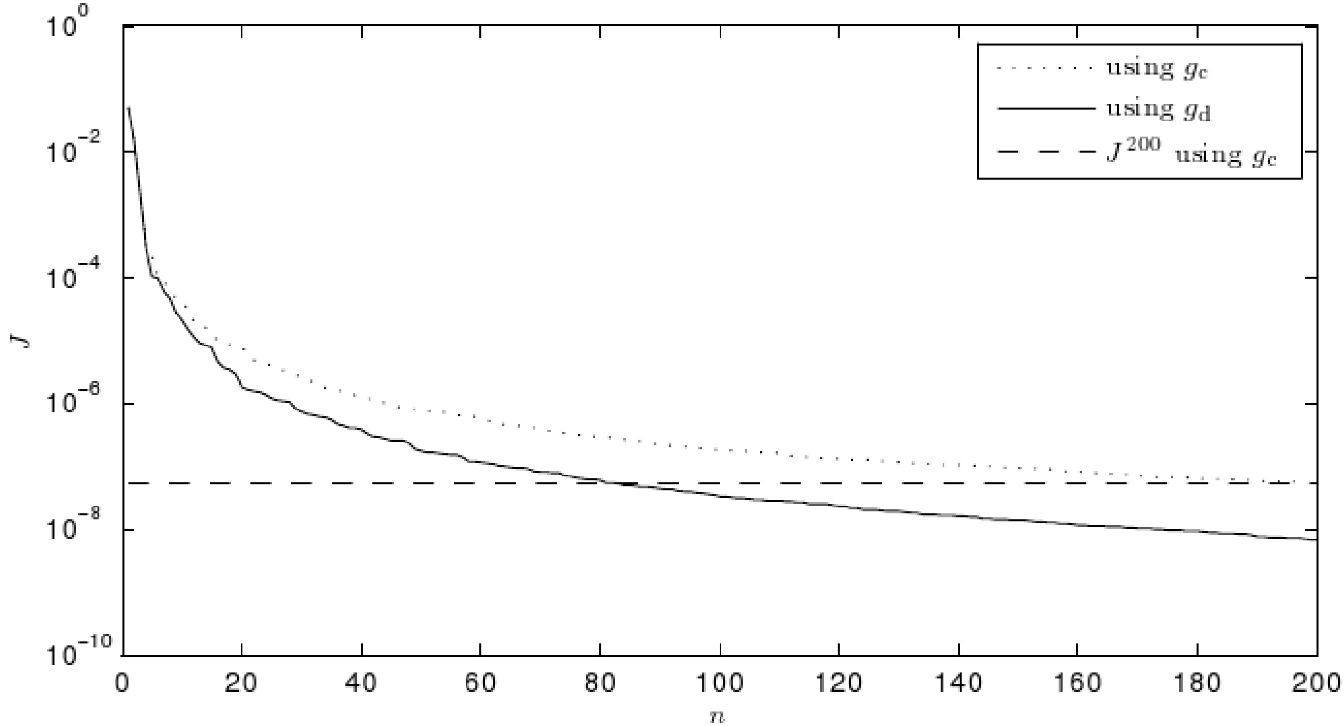
- Symbolic differentiation
  - May be difficult to write a whole program as one expression
  - Large derivative expressions with lots of repeated subexpressions
  - Often very large runtimes
    - Especially for higher order derivatives
    - Differentiation has to be done only once however
- Manual differentiation
  - Usually efficient derivative code
  - Often tedious and error-prone, especially when F is changed
  - Discretization of F and F' has to be taken into account

Differentiation-Discretization

- Let F be defined by a PDE
  - Usually implemented by discretization
  - e.g. using the Finite Element Method
- Derivative F' often by discretizing the adjoint PDE
  - The discretization introduces errors in both F and F'
  - AD of the discretized F differentiates through the discretization errors of F



- Solving Inverse Heat Conduction Problem with Conjugate-Gradient optimization using both AD gradient and gradient obtained from adjoint PDE
  - The objective function J drops faster with AD
  - "faster" means fewer number of iterations here



Conclusion

- AD advantages
  - AD can provide derivatives of that are efficient, precise, and reliable
  - AD is often easy to apply
- AD disadvantages
  - AD tools can be difficult to use and may lack support for language elements and/or higher order derivatives
  - Applying the reverse mode of AD needs special measures to cope with the memory requirements
    - Possible, but not discussed here
- When you need derivatives you should use AD
- You should consult with an AD expert

The AD book

“Evaluating Derivatives”, 2nd edition

Andreas Griewank & Andrea Walther

SIAM, Philadelphia 2008

ISBN 978-0-898716-59-7