

Finite Difference Approximation

- Solving the problem numerically → transform the continuous problem into a discrete one by:
 - Looking at finite time/space steps
- Assuming all functions are sufficiently smooth, a straightforward Taylor expansion gives:

$$u(t + \Delta t) = u(t) + u'(t)\Delta t + u''(t)\frac{\Delta t^2}{2!} + u'''(t)\frac{\Delta t^3}{3!} + \dots$$

- Thus $u'(t)$ is computed:

$$u'(t) = \frac{u(t + \Delta t) - u(t)}{\Delta t} + O(\Delta t^2)$$



Finite Difference Approximation (2)

The approximation is obtained by replacing a **differential operator** by a **finite difference**

$$u'(t) = \frac{u(t + \Delta t) - u(t)}{\Delta t}$$

- Substituting this in $u'(t) = f(t, u)$ gives

$$u(t + \Delta t) = u(t) + \Delta t f(t, u(t))$$

- If $t_0 = 0, t_{k+1} = t_k + \Delta t = \dots = (k + 1)\Delta t, u(t_k) = u_k$
- We thus get the **Explicit (forward) Euler** difference equation $u_{k+1} = u_k + \Delta t f(t_k, u_k)$



Explicit Euler: Alternative (easier) Formulation

121

- Considering the general first order differential equation $x'(t) = f(t, x(t))$ with some initial condition $x(0) = x_0$
- Euler's (explicit) method based on

$$x'(t) = \lim_{h \rightarrow 0} \frac{x(t+h) - x(t)}{h}$$

- Instead of letting, $h \rightarrow 0$ take a finite $h > 0$ and

$$x'(t) \approx \frac{x(t+h) - x(t)}{h}$$



Explicit Euler: Alternative (easier) Formulation (2)

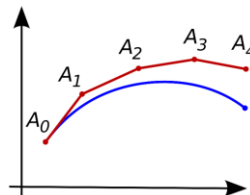
122

- Thus Euler's Explicit Method becomes

$$x(t+h) \approx x(t) + hx'(t) = x(t) + hf(t, x(t))$$

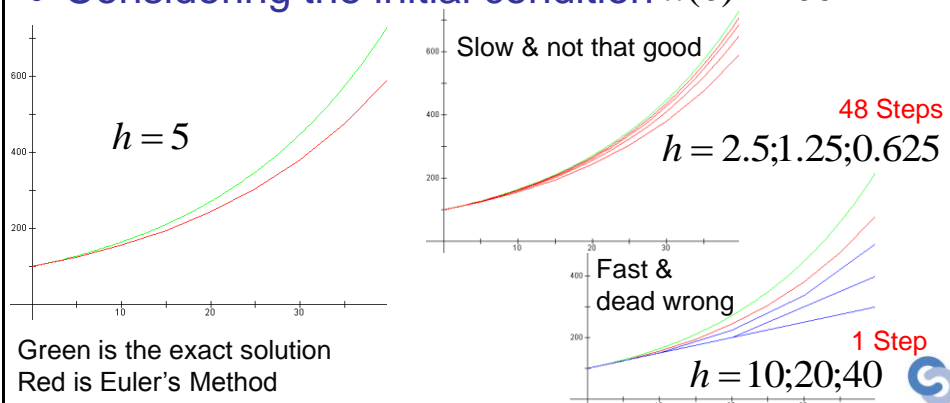
- Which is:

- Simple to program
- Very inefficient
- It sometimes gives totally erroneous results
- Highly dependent on the “right” choice for h
- Error is proportional in the first order with the step-size $h \rightarrow$ Euler is a “**first order method**”



Euler's Method - Example

- Let us apply Euler's method to the equation $x'(t) = 0.05 \cdot x$ which has as solution $x = 100 \cdot e^{0.05t}$
- Considering the initial condition $x(0) = 100$



Implicit Euler's method

- Instead of using the value of the derivative at the present point in time t , use the value of the derivative at the future point in time $t+h$:

$$x'(t) = \lim_{h \rightarrow 0} \frac{x(t) - x(t-h)}{h}$$

- Again, with a finite h we get an approximate equation


$$x'(t) \approx \frac{x(t) - x(t-h)}{h}$$

- And thus

$$x(t) \approx x(t-h) + hx'(t) = x(t-h) + hf(t, x(t))$$

Implicit Euler's method (2)


- As can be seen in

$$x(t) \approx x(t-h) + hx'(t) = x(t-h) + hf(t, x(t))$$
- The unknown value $x(t)$ appears on **both sides** of the equation, and as an argument for the function $f(t, x)$ **on the right hand side**.
- This method is known as the **Implicit Euler method**.
- Finding $x(t)$ requires solving (possibly numerically) the equation for $x(t)$
- However, if an analytic formula for $x(t)$ exists, the method is very easy to use
- To solve Implicit Euler you need an **iterative solver** 

Stability of Implicit Euler

- Considering $f(t, x) = -\lambda x$

$$x_{k+1} = x_k - \lambda h x_{k+1} \Leftrightarrow (1+h)x_{k+1} = x_k$$
- And therefore

$$x_{k+1} = \left(\frac{1}{1+\lambda h} \right) x_k = \left(\frac{1}{1+\lambda h} \right)^k x_0$$
- If $\lambda > 0$ which is the condition for a stable equation, we find that $x_k \rightarrow 0$ for all values of λ and t .
- This method is called **unconditionally stable**
- The main advantage of an implicit method over an explicit one is clearly the stability 

Stability of Implicit Euler (2)

- It is possible to take **larger time steps** without worrying about unphysical behavior
- Large time steps
 - Can make convergence to the steady state slower
 - But at least there will be no divergence
- Drawback – implicit methods are more complicated
 - Can involve nonlinear systems of equations to be solved in every time step



Other Methods – Runge-Kutta

- A slightly more complicated method for the solution of the generic first order differential equation $x'(t) = f(t, x(t))$ is the Runge-Kutta method
- A fundamental theorem of calculus reads

$$x(t_j + h) - x(t_j) = \int_{t_j}^{t_j+h} x'(\tau) d\tau = \int_{t_j}^{t_j+h} f(\tau, x(\tau)) d\tau$$

- An approximation for $x(t + h)$ is the approximation of the integral on the right hand side
- For numerical integration we apply the midpoint rule

$$\int_a^b g(\tau) d\tau \approx (b - a) g\left(\frac{a + b}{2}\right)$$



Other Methods – Runge-Kutta (2)

- First, this approximation leads to the following formula

$$x(t_j + h) \approx x(t_j) + hf(t_j + \frac{h}{2}, x(t_j + \frac{h}{2}))$$

- There is a big problem however:
 - We do not know the value of $x(t+h)$ at the midpoint $t_j + \frac{h}{2}$
- We must use an approximation, and we choose Euler's explicit method for this

$$x(t_j + \frac{h}{2}) \approx x(t_j) + \frac{h}{2} f(t_j, x(t_j))$$

- The final form for the new method thus reads

$$x(t_j + h) \approx x(t_j) + hf(t_j + \frac{h}{2}, x(t_j) + \frac{h}{2} f(t_j, x(t_j)))$$



Classical Predator-Prey Model

- Two species system:
 - Predators **P**
 - Preys **V**
- Predators are the only factor cutting down the prey population
- The remaining prey population breeds without limitations
- Predators' breeding rate is proportional to their catch



Classical Predator-Prey Model (2)

- The number of predators is limited by death rate

$$V' = kV - aVP$$

$$P' = bVP - dP$$

Also known as Lotka–Volterra equations

- Where:
 - **k** is the birth rate for prey
 - **a** is catch rate
 - **b** characterizes the efficiency with which predators use their catch to produce more predators
 - **d** is the death rate for predators



Classical Predator-Prey Model (3)

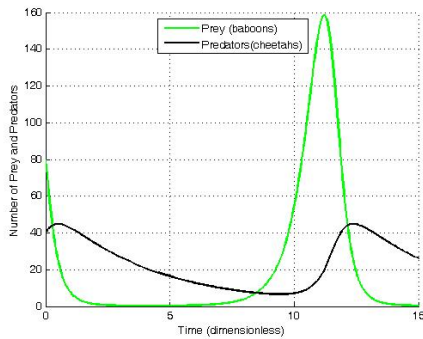
- The non-trivial equilibrium of the system

$$\begin{cases} kv - aVP = 0 \\ bVP - dP = 0 \end{cases} \Rightarrow \begin{cases} P = \frac{k}{a} = P_{eq} \\ V = \frac{d}{b} = V_{eq} \end{cases}$$

- Note the dependence on parameters at the equilibrium
- The lines $P = P_{eq}$ and $V = V_{eq}$ divide the first quadrant into four parts with different signs for the derivatives P' and V'

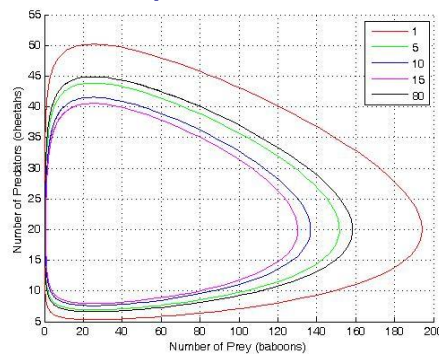


Predator-Prey Model Examples



- Problem at the lower limit of the prey population

- The solution is in a state of dynamic equilibrium – inside of the ellipses



Systems of Differential Equations

- Systems of differential equations describe the dynamics of complicated models
- There are usually two or more state variables $x_j(t)$ whose time development should be studied
 - For a specified time interval
 - For an arbitrary long period of time
- Some systems contain both **algebraic** and **differential equations** (DAEs)
- Two-dimensional models are easy to analyze
 - It is often possible to draw pictures of what is going on



Systems of Differential Equations (2)

- Generic form of a system of differential equations

$$x_1'(t) = f_1(t, x_1(t), x_2(t), \dots, x_n(t))$$

$$x_2'(t) = f_2(t, x_1(t), x_2(t), \dots, x_n(t))$$

...

$$x_n'(t) = f_n(t, x_1(t), x_2(t), \dots, x_n(t))$$

- If none of the functions f_j depend explicitly on t , the system is **autonomous**
- If every function f_j is linear, the system is said to be **linear**



Systems of Differential Equations (3)

- In order to have a unique solution
 - Initial and/or boundary conditions are needed
- Initial conditions are given at the moment $t = 0$
- Sometimes conditions are also given at the end point of the time interval
- It is also possible that part of the conditions are specified at the starting point and part at the end



General Model Behavior

- Systems with two state variables can exhibit: equilibrium and limit cycles
- Equilibrium behavior:
 - **Stable**: starting near an equilibrium, will keep you near that equilibrium
 - **Asymptotically stable**: starting near an equilibrium, will drift you closer and closer to the equilibrium
 - **Unstable**: starting exactly at the equilibrium, will keep you there. Any perturbation, however small, will drive you away
 - **Saddle point**: depending on the direction w.r.t. the equilibrium you can either drift towards it or away from it



General Model Behavior (2)

- Limit cycles:
 - **Neutral**: small perturbations will move you to another cycle
 - **Stable**: the effect of small perturbations will gradually disappear and the system drifts back to the original cycle
 - **Unstable**: small perturbations drive the system away from the cycle
- Possibly only numerical results
- Usually the long-term behavior is what matters



Table of Contents

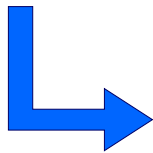
- Motivation & Trends in HPC
- R&D Projects @ PP
- Mathematical Modeling
- **Numerical Methods used in HPSC**
 - Systems of Differential Equations: ODEs & PDEs
 - **Automatic Differentiation**
 - Solving Optimization Problems
 - Solving Nonlinear Equations
 - Basic Linear Algebra, Eigenvalues and Eigenvectors
 - Chaotic systems
- HPSC Program Development/Enhancement: from Prototype to Production
- Debugging, Profiling, Performance Analysis & Optimization



Automatic Differentiation – Motivation

- *Design optimization*
- *Sensitivity analysis & Parameter identification*
- *Data assimilation problems*
- *Inverse problems (data assimilation)*
- *Solving ODE, PDE, DAE, ...*
- *Linear approximation & Curve fitting*

Derivative information required

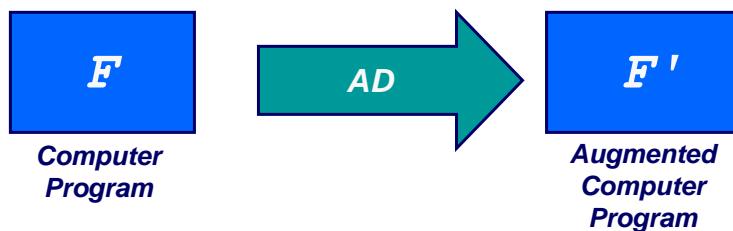


Numeric Differentiation
Symbolic Differentiation
Automatic Differentiation



Automatic Differentiation (AD)

- Semantic transformation:
 - Given a computer code F , AD generates a new program F' which applies the chain rule of differential calculus to elementary operations for which the derivatives are known

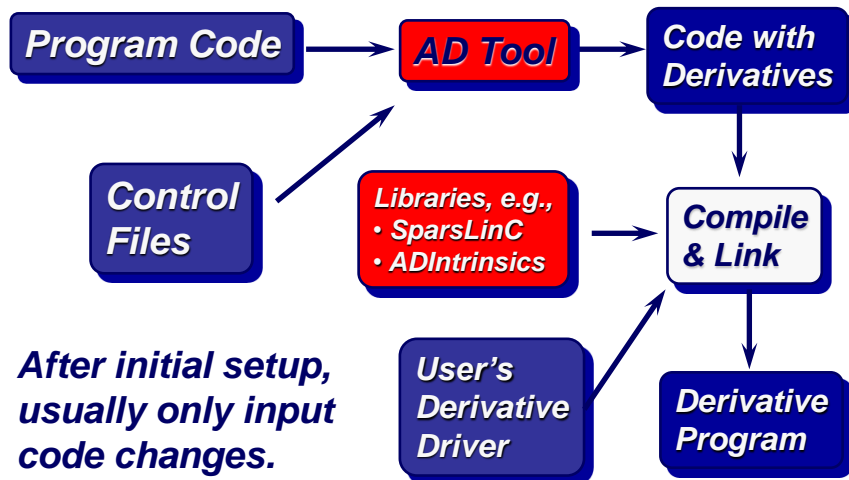


AD Advantages

- Generates truncation- and cancellation error-free derivatives
- Generates a program for the computation of derivative values, not of derivative formulae
- The associativity of the chain rule allows for a wide range of choices in accumulating derivatives → forward & reverse modes and generalizations



The AD Process



AD-Tool Implementations

- Source Transformation (ST)
 - Compiler-based technique → generate new code that **explicitly** computes derivatives
 - **Advantages:** entire context known at compile-time → efficient code, transparency
 - **Drawback:** difficult implementation
- Operator Overloading (OO)
 - Extends elementary operations & functions to also **implicitly** compute derivatives
 - **Requires:** redeclaration of active variables to the new overloaded types
 - **Advantages:** easy implementation & „same“ source code
 - **Drawback:** granularity → inefficient code



Some AD Tools

<http://www.autodiff.org>

Fortran 77, some Fortran 90:

- ADIFOR 3.0/ADJIFOR (RM, ∂f , $\partial^2 f$): Alan Carle & Mike Fagan (Rice)
- Tapenade (RM, ∂f): Laurent Hascoet (INRIA)
- TAMC/TAF (RM, ∂f): Ralf Giering (FastOpt GmbH).

ANSI-C/C++:

- ADOL-C: (FM/RM, $\partial^k f$): Andreas Griewank & Co. (TU Dresden)
- ADIC: (FM/RM, $\partial^k f$): Hovland (ANL)

Application Specific:

- Cosy-Infinity (Remainder Differential Algebra): Martin Berz (U Michigan)
- TOMLAB/MAD (AD of Matlab): Shaun Forth (RMCS Shrivvenham)
- ADiMat (AD of Matlab): Andre Vehreschild (RWTH Aachen)

OpenAD/F:

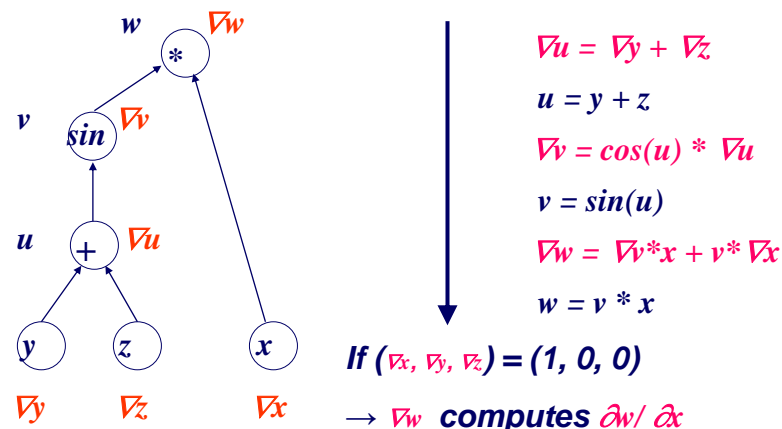
- AD Framework: J. Utke & U. Naumann (ANL & RWTH Aachen)



Forward Mode - FM

Example code f : $w = x * \sin(y+z)$

Original code and derivative statements:



Each variable t is associated with a derivative object ∇t



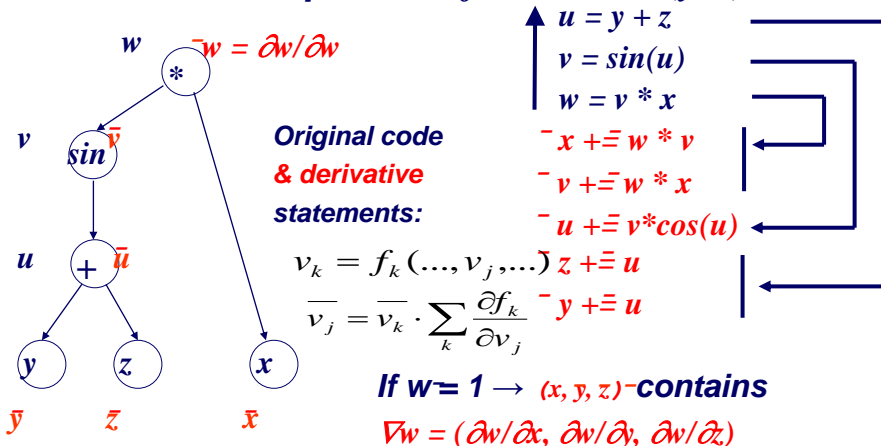
Forward Mode Facts

- Computes $\partial f / \partial t^* S$ (S is called “seed matrix” – $[\nabla_x, \nabla_y, \nabla_z]$) by propagating sensitivities of *intermediate values* with respect to *input values*
- For p input values of interest, **runtime** and **memory** scale approximately with p . May be much less e.g. for **sparse Jacobians**
- FM is appropriate for moderate p 's



Reverse Mode - RM

The same example code $f: w = x * \sin(y+z)$



Each variable t is associated with an adjoint object \bar{t} , the differentiated expressions are accumulated in the reverse order



Reverse Mode Facts

- Computes $W^T * \frac{\partial f}{\partial t}$ by propagating sensitivities of *output values* with respect to *intermediate values*
- For q output values of interest, the **runtime** scales with q . **Memory requirements** are harder to predict → greatly depend on the structure & implementation of program/AD-tool
- RM is great for computing „long gradients“ – small q 's and big p 's



Forward & Reverse Mode Example

given function $f(x, y, z) = (xy + \cos z)(x^2 + 2y^2 + 3z^2)$, the partial derivatives are

$$\frac{\partial f}{\partial x} = y \cdot (x^2 + 2y^2 + 3z^2) + (xy + \cos z) \cdot 2x = 3x^2y + 2y^3 + 3yz^2 + 2x \cos z,$$

$$\frac{\partial f}{\partial y} = x \cdot (x^2 + 2y^2 + 3z^2) + (xy + \cos z) \cdot 4y = x^3 + 6xy^2 + 3xz^2 + 4y \cos z,$$

$$\begin{aligned} \frac{\partial f}{\partial z} &= -\sin z \cdot (x^2 + 2y^2 + 3z^2) + (xy + \cos z) \cdot 6z \\ &= -x^2 \sin z - 2y^2 \sin z - 3z^2 \sin z + 6xyz + 6z \cos z. \end{aligned}$$

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \end{bmatrix}^T$$



Forward Mode



Code list + Gradient entries

$u_1 = x,$	$\nabla u_1 = [1, 0, 0],$
$u_2 = y,$	$\nabla u_2 = [0, 1, 0],$
$u_3 = z,$	$\nabla u_3 = [0, 0, 1],$
$u_4 = u_1 u_2,$	$\nabla u_4 = u_1 \nabla u_2 + u_2 \nabla u_1 = [0, u_1, 0] + [u_2, 0, 0] = [u_2, u_1, 0],$
$u_5 = \cos u_3,$	$\nabla u_5 = (-\sin u_3) \nabla u_3 = [0, 0, -\sin u_3],$
$u_6 = u_4 + u_5,$	$\nabla u_6 = \nabla u_4 + \nabla u_5 = [u_2, u_1, -\sin u_3],$
$u_7 = u_1^2,$	$\nabla u_7 = 2u_1 \nabla u_1 = [2u_1, 0, 0],$
$u_8 = 2u_2^2,$	$\nabla u_8 = 4u_2 \nabla u_2 = [0, 4u_2, 0],$
$u_9 = 3u_3^2,$	$\nabla u_9 = 6u_3 \nabla u_3 = [0, 0, 6u_3],$
$u_{10} = u_7 + u_8 + u_9,$	$\nabla u_{10} = \nabla u_7 + \nabla u_8 + \nabla u_9 = [2u_1, 4u_2, 6u_3],$
$u_{11} = u_6 u_{10},$	$\nabla u_{11} = u_6 \nabla u_{10} + u_{10} \nabla u_6 = [2u_6 u_1 + u_{10} u_2, 4u_6 u_2 + u_{10} u_1, 6u_6 u_3 - u_{10} \sin u_3].$

$$\begin{aligned} \nabla f(x, y, z) = \nabla u_{11} &= [3x^2 y + 2x \cos z + 2y^3 + 3yz^2, \\ &6xy^2 + 4y \cos z + x^3 + 3xz^2, \\ &6xyz + 6z \cos z - x^2 \sin z - 2y^2 \sin z - 3z^2 \sin z]. \end{aligned}$$

Correct!



Reverse Mode



Code list + Adjoints

$u_1 = x,$	$\frac{\partial u_{11}}{\partial u_1} = 1, \frac{\partial u_{11}}{\partial u_{10}} = u_6, \frac{\partial u_{11}}{\partial u_9} = \frac{\partial u_{11}}{\partial u_{10}} \frac{\partial u_{10}}{\partial u_9} = u_6,$
$u_2 = y,$	
$u_3 = z,$	$\frac{\partial u_{11}}{\partial u_8} = \frac{\partial u_{11}}{\partial u_{10}} \frac{\partial u_{10}}{\partial u_8} = u_6, \frac{\partial u_{11}}{\partial u_7} = \frac{\partial u_{11}}{\partial u_{10}} \frac{\partial u_{10}}{\partial u_7} = u_6,$
$u_4 = u_1 u_2,$	
$u_5 = \cos u_3,$	$\frac{\partial u_{11}}{\partial u_6} = u_{10}, \frac{\partial u_{11}}{\partial u_5} = \frac{\partial u_{11}}{\partial u_6} \frac{\partial u_6}{\partial u_5} = u_{10}, \frac{\partial u_{11}}{\partial u_4} = \frac{\partial u_{11}}{\partial u_6} \frac{\partial u_6}{\partial u_4} = u_{10},$
$u_6 = u_4 + u_5,$	
$u_7 = u_1^2,$	$\frac{\partial u_{11}}{\partial u_3} = \frac{\partial u_{11}}{\partial u_9} \frac{\partial u_9}{\partial u_3} + \frac{\partial u_{11}}{\partial u_5} \frac{\partial u_5}{\partial u_3} = 6u_6 u_3 - u_{10} \sin u_3,$
$u_8 = 2u_2^2,$	
$u_9 = 3u_3^2,$	$\frac{\partial u_{11}}{\partial u_2} = \frac{\partial u_{11}}{\partial u_4} \frac{\partial u_4}{\partial u_2} + \frac{\partial u_{11}}{\partial u_8} \frac{\partial u_8}{\partial u_2} = u_{10} u_1 + 4u_6 u_2,$
$u_{10} = u_7 + u_8 + u_9,$	
$u_{11} = u_6 u_{10},$	$\frac{\partial u_{11}}{\partial u_1} = \frac{\partial u_{11}}{\partial u_4} \frac{\partial u_4}{\partial u_1} + \frac{\partial u_{11}}{\partial u_7} \frac{\partial u_7}{\partial u_1} = u_{10} u_2 + 2u_6 u_1.$

$$\begin{aligned} \nabla f(x, y, z) &= \left[\frac{\partial u_{11}}{\partial u_1}, \frac{\partial u_{11}}{\partial u_2}, \frac{\partial u_{11}}{\partial u_3} \right] = [3x^2 y + 2x \cos z + 2y^3 + 3yz^2, \\ &6xy^2 + 4y \cos z + x^3 + 3xz^2, \\ &6xyz + 6z \cos z - x^2 \sin z - 2y^2 \sin z - 3z^2 \sin z]. \end{aligned}$$

Correct!



Divided Differences

First order differentiation

$$\text{Forward differentiation: } \frac{\partial f}{\partial x_m} = \frac{f(x_1, \dots, x_m + h, \dots, x_n) - f(x_1, \dots, x_m, \dots, x_n)}{h} + O(h)$$

$$\text{Backward differentiation: } \frac{\partial f}{\partial x_m} = \frac{f(x_1, \dots, x_m, \dots, x_n) - f(x_1, \dots, x_m - h, \dots, x_n)}{h} + O(h)$$

$$\text{Centered differentiation: } \frac{\partial f}{\partial x_m} = \frac{f(x_1, \dots, x_m + h, \dots, x_n) - f(x_1, \dots, x_m - h, \dots, x_n)}{2h} + O(h^2)$$

Second order differentiation

$$\frac{\partial^2 f}{\partial x_m^2} = \frac{f(x_1, \dots, x_m + h, \dots, x_n) - 2f(x_1, \dots, x_m, \dots, x_n) + f(x_1, \dots, x_m - h, \dots, x_n)}{h^2} + O(h^2)$$



Which mode to use?

- Use forward mode when
 - # **independents** is very small
 - Only a directional derivative $\mathbf{J}\mathbf{v}$ is needed
 - Reverse mode is not tractable
- Use reverse mode when
 - # **dependents** is very small
 - Only $\mathbf{J}^T\mathbf{v}$ is needed



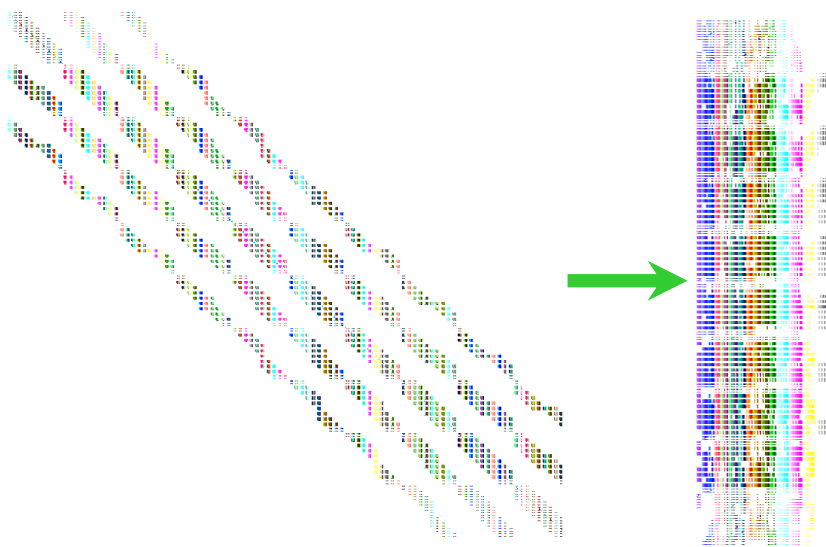
Case Study – Matrix Coloring

- Jacobian matrices are often sparse
- The forward mode of AD computes $J \times S$, where S is usually an identity matrix or a vector
- One can “**compress**” Jacobian by choosing S such that *structurally orthogonal* columns are combined
- A set of columns are structurally orthogonal if no two of them have nonzeros in the same row
- Equivalent problem: color the graph whose adjacency matrix is $J^T J$
- Equivalent problem: distance-2 color the bipartite graph of J

©Paul Hovland @ANL



Compressed Jacobian

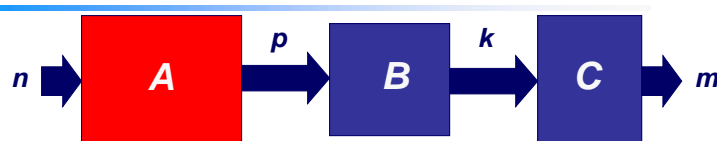


What is feasible & practical

- Key point:
 - Forward mode computes JS at cost proportional to number of columns in S
 - Reverse mode computes $J^T W$ at cost proportional to number of columns in W
- Jacobians of functions
 - Small number (1—1000) of **independent** variables (FM)
 - Small number (1—100) of **dependent** variables (RM)
- Extremely large, but (very) sparse Jacobians and Hessians
- Jacobian-vector products (forward mode)
- Transposed-Jacobian-vector products (adjoint mode)
- Hessian-vector products (forward + adjoint modes)



Scenarios



- n small: use FM on full computation
- m small: use RM on full computation
- m & n large, P small: use RM on A, FM on B&C
- m & n large, K small: use RM on A&B, FM on C
- n, p, k, m large, Jacobians of A, B, C sparse: compressed FM
- n, p, k, m large, Jacobians of A, B, C low rank: scarce FM
- n, p, k, m large: Jacobians of A, B, C dense: what to do?



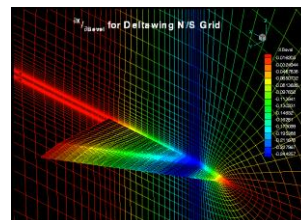
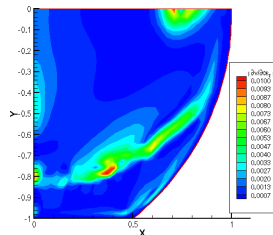
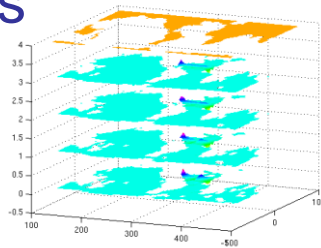
Issues with Black Box Differentiation

- Source code may not be available or may be difficult to work with
- Simulation may not be (chain rule) differentiable
 - Feedback due to adaptive algorithms
 - Non-differentiable functions
 - Noisy functions
 - Convergence rates
 - Etc.
- Accurate derivatives may not be needed – FD might be cheaper
- Differentiation and discretization do not commute



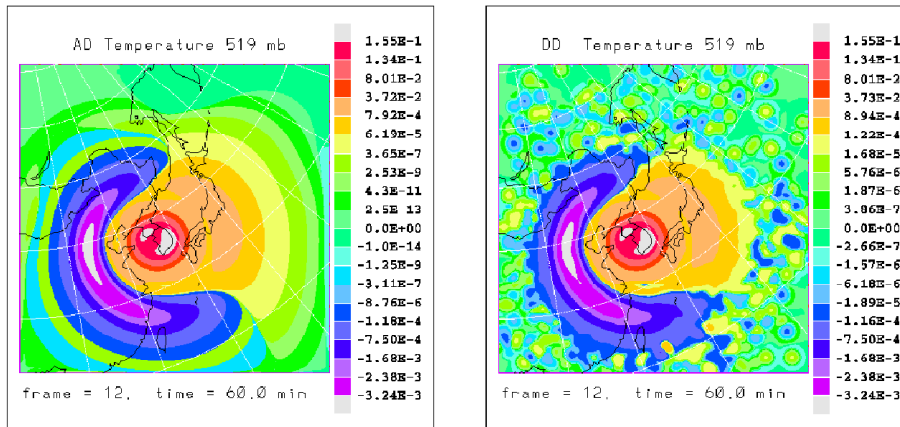
Application highlights

- Atmospheric chemistry
- Breast cancer biostatistical analysis
- CFD: CFL3D, NSC2KE, Fluent 4.52
- Chemical kinetics
- Climate and weather: MITGCM, MM5, CICE
- Semiconductor device simulation
- Water reservoir simulation



Sensitivity Analysis: Mesoscale Weather Modeling

161



AD Conclusions & Future Work

162

- Automatic differentiation research involves a wide range of combinatorial problems
- AD is a powerful tool for scientific computing
- Modern automatic differentiation tools are robust and produce efficient code for complex simulation codes
 - Requires an industrial-strength compiler infrastructure
 - Efficiency requires sophisticated compiler analysis
- Effective use of automatic differentiation depends on insight into problem structure
- Future Work
 - Further develop and test techniques for computing Jacobians that are effectively sparse or effectively low rank
 - Develop techniques to automatically generate complex and adaptive checkpointing strategies

