Automatic Differentiation

Hamid Reza Ghaffari , Jonathan Li, Yang Li, Zhenghua Nie

Instructor: Prof. Tamas Terlaky
School of Computational Engineering and School
McMaster University

March. 23, 2007

Outline

1 Introductions
2 Forward and Reverse Mode
- Forward methods
- Reverse methods
- Comparison
- Extended knowledge
- Case Study
3 Complexity Analysis
- Forward Mode Complexity
- Reverse Mode Complexity
4 AD Softwares
- AD tools in MATLAB
- AD in C/C++ (ADIC)
  - Developers
  - introduction
  - ADIS Anatomy
  - ADICProcess
  - Example
  - Handling Side Effects
  - References

Why Do we Need Derivatives?

- *Optimization via gradient method.*
  - Unconstrained Optimization minimize $y = f(x)$ requires gradient or hessian.
  - Constrained Optimization minimize $y = f(x)$ such that $c(x) = 0$ also requires Jacobian $Jc(x) = [\partial c_j / \partial x_i]$.
- *Solution of Nonlinear Equations $f(x) = 0$ by Newton Method*

$$x^{n+1} = x^n - \left[\frac{\partial f(x^n)}{\partial x}\right]^{-1} f(x^n)$$

  requires Jacobian $JF = [\partial f / \partial x]$.
- Parameter Estimation, Data Assimilation, Sensitivity Analysis, Inverse Problem, ...

Why Do we Need Derivatives?

- *Optimization via gradient method.*
  - Unconstrained Optimization minimize $y = f(x)$ requires gradient or hessian.
  - Constrained Optimization minimize $y = f(x)$ such that $c(x) = 0$ also requires Jacobian $Jc(x) = [\partial c_j/\partial x_i]$.
- *Solution of Nonlinear Equations* $f(x) = 0$ by Newton Method

$$x^{n+1} = x^n - \left[\frac{\partial f(x^n)}{\partial x}\right]^{-1} f(x^n)$$

  requires Jacobian $JF = [\partial f/\partial x]$.
- Parameter Estimation, Data Assimilation, Sensitivity Analysis, Inverse Problem, ...

Why Do we Need Derivatives?

- *Optimization via gradient method.*
  - Unconstrained Optimization minimize $y = f(x)$ requires gradient or hessian.
  - Constrained Optimization minimize $y = f(x)$ such that $c(x) = 0$ also requires Jacobian $Jc(x) = [\partial c_j / \partial x_i]$.
- *Solution of Nonlinear Equations* $f(x) = 0$ by Newton Method
$$x^{n+1} = x^n - \left[\frac{\partial f(x^n)}{\partial x}\right]^{-1} f(x^n)$$
requires Jacobian $JF = [\partial f / \partial x]$.
- Parameter Estimation, Data Assimilation, Sensitivity Analysis, Inverse Problem, ...

How Do We Obtain Derivatives?

- **Reliability:** the correctness and numerical accuracy of the derivative results;
- **Computational Cost:** the amount of runtime and memory required for the derivative code;
- **Development Time:** the time it takes to design, implement, and verify the derivative code, beyond the time to implement the code for the computation of underlying function.

How Do We Obtain Derivatives?

- **Reliability:** the correctness and numerical accuracy of the derivative results;
- **Computational Cost:** the amount of runtime and memory required for the derivative code;
- **Development Time:** the time it takes to design, implement, and verify the derivative code, beyond the time to implement the code for the computation of underlying function.

How Do We Obtain Derivatives?

- **Reliability:** the correctness and numerical accuracy of the derivative results;
- **Computational Cost:** the amount of runtime and memory required for the derivative code;
- **Development Time:** the time it takes to design, implement, and verify the derivative code, beyond the time to implement the code for the computation of underlying function.

Main Approaches

- Hand Coding
- Divided Differences
- Symbolic Differentiation
- Automatic Differentiation

Hand Coding

An analytic expression for the derivative is identified first and then implemented by hand using any high-level programming language.

- Advantages
  - Accuracy up to machine precision, if care is taken.
  - Highly-optimized implementation depending on the skill of the implementer.
- Disadvantages
  - Only applicable for "simple" functions and error-prone.
  - Requires considerable human effort.

Hand Coding

An analytic expression for the derivative is identified first and then implemented by hand using any high-level programming language.

- Advantages
  - Accuracy up to machine precision, if care is taken.
  - Highly-optimized implementation depending on the skill of the implementer.
- Disadvantages
  - Only applicable for "simple" functions and error-prone.
  - Requires considerable human effort.

Hand Coding

An analytic expression for the derivative is identified first and then implemented by hand using any high-level programming language.

- Advantages
  - Accuracy up to machine precision, if care is taken.
  - Highly-optimized implementation depending on the skill of the implementer.
- Disadvantages
  - Only applicable for "simple" functions and error-prone.
  - Requires considerable human effort.

Hand Coding

An analytic expression for the derivative is identified first and then implemented by hand using any high-level programming language.

- Advantages
  - Accuracy up to machine precision, if care is taken.
  - Highly-optimized implementation depending on the skill of the implementer.
- Disadvantages
  - Only applicable for "simple" functions and error-prone.
  - Requires considerable human effort.

Hand Coding

An analytic expression for the derivative is identified first and then implemented by hand using any high-level programming language.

- Advantages
  - Accuracy up to machine precision, if care is taken.
  - Highly-optimized implementation depending on the skill of the implementer.
- Disadvantages
  - Only applicable for "simple" functions and error-prone.
  - Requires considerable human effort.

Hand Coding

An analytic expression for the derivative is identified first and then implemented by hand using any high-level programming language.

- Advantages
  - Accuracy up to machine precision, if care is taken.
  - Highly-optimized implementation depending on the skill of the implementer.
- Disadvantages
  - Only applicable for "simple" functions and error-prone.
  - Requires considerable human effort.

Divided Differences

Approximate the derivative of a function f w.r.t the ith component of x at a particular point x_0 by difference numerically, e.g

$$\left.\frac{\partial f(x)}{\partial x_i}\right|_{x_0} \approx \frac{f(x_0 + he_i) - f(x_0)}{h}$$

where e_i is the ith Cartesian unit vector.

Divided Differences(Ctd.)

$$\left.\frac{\partial f(x)}{\partial x_i}\right|_{x_0} \approx \frac{f(x_0 + he_i) - f(x_0)}{h}$$

- Advantage:
  - only f is needed, easy to be implemented, used as a "black box"
  - easy to parallelize
- Disadvantage:
  - Accuracy hard to assess, depending on the choice of h
  - Computational complexity bounded below: $(n+1) \times cost(f)$

Divided Differences(Ctd.)

$$\left.\frac{\partial f(x)}{\partial x_i}\right|_{x_0} \approx \frac{f(x_0 + he_i) - f(x_0)}{h}$$

- Advantage:
  - only f is needed, easy to be implemented, used as a "black box"
  - easy to parallelize
- Disadvantage:
  - Accuracy hard to assess, depending on the choice of h
  - Computational complexity bounded below: $(n+1) \times cost(f)$

Divided Differences(Ctd.)

$$\left.\frac{\partial f(x)}{\partial x_i}\right|_{x_0} \approx \frac{f(x_0 + he_i) - f(x_0)}{h}$$

- Advantage:
  - only f is needed, easy to be implemented, used as a "black box"
  - easy to parallelize
- Disadvantage:
  - Accuracy hard to assess, depending on the choice of h
  - Computational complexity bounded below: $(n+1) \times cost(f)$

Divided Differences(Ctd.)

$$\left.\frac{\partial f(x)}{\partial x_i}\right|_{x_0} \approx \frac{f(x_0 + he_i) - f(x_0)}{h}$$

- Advantage:
  - only f is needed, easy to be implemented, used as a "black box"
  - easy to parallelize
- Disadvantage:
  - Accuracy hard to assess, depending on the choice of h
  - Computational complexity bounded below: $(n+1) \times cost(f)$

Divided Differences(Ctd.)

$$\left.\frac{\partial f(x)}{\partial x_i}\right|_{x_0} \approx \frac{f(x_0 + he_i) - f(x_0)}{h}$$

- Advantage:
  - only f is needed, easy to be implemented, used as a "black box"
  - easy to parallelize
- Disadvantage:
  - Accuracy hard to assess, depending on the choice of h
  - Computational complexity bounded below: $(n+1) \times cost(f)$

Divided Differences(Ctd.)

$$\left.\frac{\partial f(x)}{\partial x_i}\right|_{x_0} \approx \frac{f(x_0 + he_i) - f(x_0)}{h}$$

- Advantage:
  - only f is needed, easy to be implemented, used as a "black box"
  - easy to parallelize
- Disadvantage:
  - Accuracy hard to assess, depending on the choice of h
  - Computational complexity bounded below: $(n+1) \times cost(f)$

Symbolic Differentiation

Find an explicit derivative expression by computer algebra systems.

- Disadvantages:
    - The length of the representation of the resulting derivative expressions increases rapidly with the number, n, of independent variables;
    - Inefficient in terms of computing time due to the rapid growth of the underlying expressions;
    - Unable to deal with constructs such as branches, loops, or subroutines that are inherent in computer codes.

Automatic Differentiation

- What is Automatic Differentiation?
  Algorithmic, or automatic, differentiation (AD) is concerned with the accurate and efficient evaluation of derivatives for functions defined by computer programs. No truncation errors are incurred, and the resulting numerical derivative values can be used for all scientific computations that are based on linear, quadratic, or even higher order approximations to nonlinear scalar or vector functions.

Automatic Differentiation (Cont.)

- What's the idea behind Automatic Differentiation?
  Automatic differentiation techniques rely on the fact that every function no matter how complicated is executed on a computer as a (potentially very long) sequence of elementary operations such as additions, multiplications, and elementary functions such as *sin* and *cos*. By repeated application of the chain rule of derivative calculus to the composition of those elementary operations, one can computes in a completely mechanical fashion.

How good AD is?

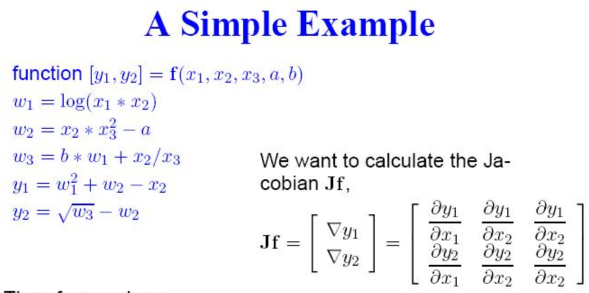
- **Reliability**
  Accurate to machine precision, no truncation error exists.
- **Computational Cost**
  Forward Mode: $2 \sim 3n \times cost(f)$
  Reverse Mode: $5 \times cost(f)$
- **Human Effort**
  Spend less time in preparing a code for differentiation, in particular in situations where computer models are bound to change frequently.

How widely is AD used?

- Sensitivity Analysis of a Mesoscale Weather Model
  Application Area: Climate Modeling
- Data assimilation for ocean circulation
  Application Area: Oceanography
- Intensity Modulated Radiation Therapy
  Application Area: Biomedicine
- Multidisciplinary Design of Aircraft
  Application Area: Computational Fluid Dynamics
- The NEOS server
  Application Area: Optimization
- ......

Source: http://www.autodiff.org/?module=Applications&submenu=& category=all

AD methods : SimpleExample



A Simple Example

function $[y_1, y_2] = \mathrm{f}(x_1, x_2, x_3, a, b)$

$w_1 = \log(x_1 * x_2)$

$w_2 = x_2 * x_3^2 - a$

$w_3 = b * w_1 + x_2/x_3$

$y_1 = w_1^2 + w_2 - x_2$

$y_2 = \sqrt{w_3} - w_2$

We want to calculate the Jacobian $\mathbf{Jf}$,

$$\mathbf{Jf} = \begin{bmatrix} \nabla y_1 \\ \nabla y_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_2} \end{bmatrix}$$

Therefore we have

- independent variables: $\mathbf{x} = (x_1, x_2, x_3)$
- dependent variables: $\mathbf{y} = (y_1, y_2)$
- intermediate variables: $\mathbf{w} = (w_1, w_2, w_3)$
- active variables $\mathbf{x}, \mathbf{y}, \mathbf{w}$
- inactive variables a, b

SimpleExample

Unify all the variable..

$$\left.\begin{aligned}
u_1 &= x_1 \\
u_2 &= x_2 \\
u_3 &= x_3 \\
u_4 &= \Phi_4(u_1, u_2) && = \log(u_1 * u_2) \\
u_5 &= \Phi_5(u_2, u_3) && = u_2 * u_3^2 - a \\
u_6 &= \Phi_6(u_2, u_3, u_4) && = b * u_4 + u_2 / u_3 \\
u_7 &= \Phi_7(u_2, u_4, u_5) && = u_4^2 + u_5 - u_2 \\
u_8 &= \Phi_8(u_5, u_6) && = \sqrt{u_6} - u_5
\end{aligned}\right\}$$

Forward method

- **Forward method**
  Differentiate the Code:

$$u_i = x_i \quad i = 1, \ldots n,$$

$$u_i = \Phi(\{u_j\}_{j<i}) \quad i = n+1, \ldots, N$$

Differentiate:

$$\nabla u_i = e_i \quad i = 1, \ldots, n$$

$$\nabla u_i = \sum_{j<i} c_{i,j} * \nabla u_j \quad i = n+1, \ldots, N$$

Reverse method

- **Reverse method**
  Compute the Adjoint of the Code

$$\overline{u}_j = \frac{\partial y}{\partial u_j} = \frac{\partial(y_1, y_2, ..y_m)}{\partial u_j}$$

  Compute for dependent variables

$$\overline{u}_{n+p+j} = \frac{\partial(y_1, y_2, ..y_m)}{\partial u_j} = e_j \quad j = 1, ..., m$$

  Compute for intermediates and independents u_j, $j = n + p, ..., 1$

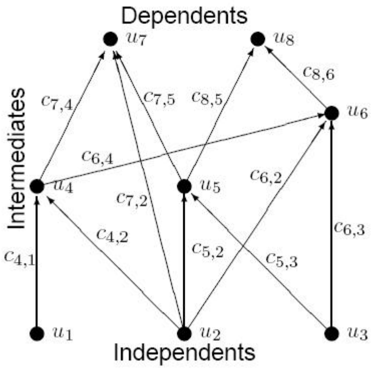
$$\overline{u}_j = \frac{\partial y}{\partial u_j} = \sum_{i>j} \overline{u}_i c_{i,j}$$

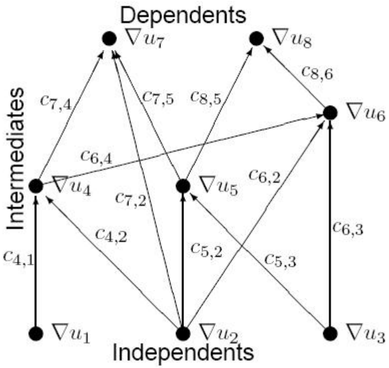
Forward methods

- **Forward method**
- ***Method :*** Compute the gradient of each variable, and use the chain rule to pass the gradient
- ***The size of computed object***: In each computation, it computes the vectors with input size n.
- The computation of gradient of each variable proceeds with the computation of each variable
- Easily implement

Forward methods

- **Computing Variable Value   Computing Gradient Value**





Reverse methods

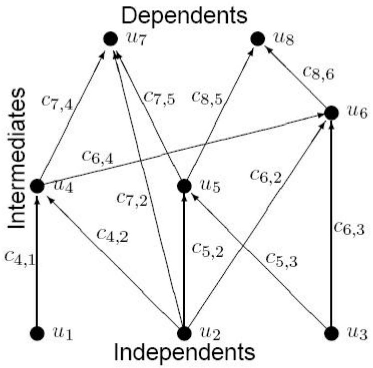
- **Reverse method**
- ***Method*** : Compute Adjoint of each variable, pass the Adjoint
- ***The size of computed object***: In each computation, it computes the vectors with output size m. (Note,usually the output size is 1 in optimization application.)
- The computation of Adjoint of each variable proceed after the completion of the computation of all variables.

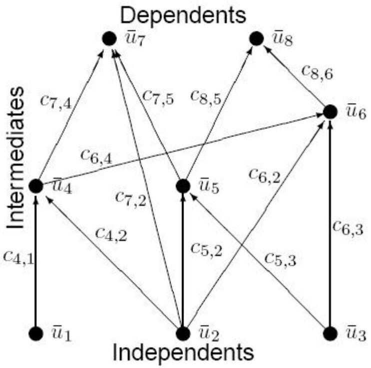
Reverse methods

- **Reverse method**
- Traverse through the Computational Graph reversely and get the parents of each variable so as to compute the Adjoint.
- Obtain the gradient by compute each partial deriviate one by one
- Harder to implement

Reverse methods

- **Computing Variable Value   Computing Adjoint Value**





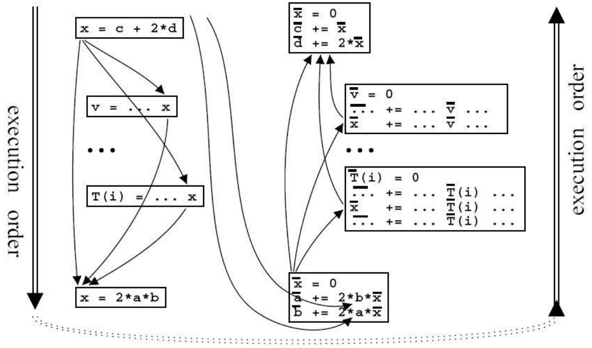
Implementation of Reverse mode

- Implementation of Reverse mode
- As mentioned above, the implementation in Forward mode is relatively straightforward. We only propose the comparison of important feature between ***Source Transformation*** and ***Operator Overloading***:
- Using Source Transformation: Re-ordering the code upside down
- Using Operator Overloading: Record computation on a "tape"

Implementation of Reverse mode

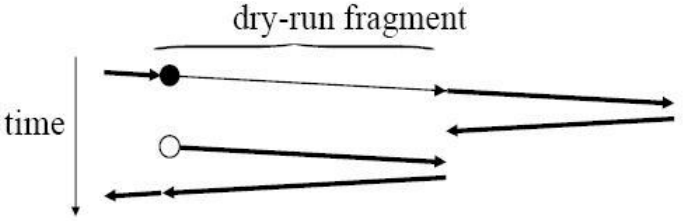
- Re-ordering the code upside down:

Hascoët: Adjoining a Data-Dependence Graph



Implementation of Reverse mode

- Record computation on a "tape"
  Record:Operation,operands
- Related technique: Checkpointing
  If the number of operations going large, Checkpointing prevent the program from exhausting all the memory



Comparison

- The following topic is discussed in the comparison between Forward mode and backward mode
- Computational Complexity
- Memory Required
- Time to develop

Cost of Forward Propagation of Derivs.

- Define $\begin{cases} N_{|c|=1} : & \textit{No. of unit local derivatives } c_{i,j} = \pm 1 \\ N_{|c|\neq 1} : & \textit{No. of nonunit local derivatives } c_{i,j} \neq 0,\ \pm 1 \end{cases}$
- Solve for derivatives in forward order $\bigtriangledown u_{n+1}, \bigtriangledown u_{n+2}, \ldots, \bigtriangledown u_N$

$$\bigtriangledown u_i = \sum_{j \prec i} c_{i,j} * \bigtriangledown u_j, \ i = n+1, \ldots, N,$$

  with each $\bigtriangledown u_i = (\partial u_i / \partial x_1, \ldots, \partial u_i / \partial x_n)$, a length n vector.
- Flop count $\textit{flops}(\textit{fwd})$ given by,

$$\begin{array}{rcll} \textit{flops}(\textit{fwd}) & = & nN_{|c|\neq 1} & (\textit{mults.} c_{i,j} * \bigtriangledown u_j,\ c_{i,j} \neq 1, 0) \\ & & +n(N_{|c|\neq 1} + N_{|c|=1}) & (\textit{adds./subs.} + c_{i,j} \bigtriangledown u_j) \\ & & -n(p+m) & (\textit{first } n \textit{ adds./subs.}) \end{array}$$

$$\textit{flops}(\textit{fwd}) = n(2N_{|c|\neq 1} + N_{|c|=1} - p - m)$$

Cost of Reverse Propagation of Adjoints

- Solve for adjoints in reverse order $\bar{u}_{n+p}, \bar{u}_{n+p-1}, \dots, \bar{u}_1$

$$\bar{u}_j = \sum_{i \succ j} \bar{u}_i c_{i,j}.$$

with $\bar{u}_j = \frac{\partial}{\partial u_j}(y_1, y_2, \dots, y_m)$ is a length m vector.

- Flop count $flops(rev)$ given by,

$$\begin{aligned} flops(rev) &= mN_{|c|\neq 1} && (mults. \bar{u}_i * c_{i,j},\ c_{i,j} \neq \pm 1, 0) \\ &= +m(N_{|c|=1} + N_{|c|\neq 1}) && (adds./subs.\ + (\bar{u}_i * c_{i,j})) \end{aligned}$$

$$flops(rev) = m(2N_{|c|\neq 1} + N_{|c|=1}).$$

Memory Required

- ***Used Storage***:
  It's uncertain that which mode takes more memory, usually, reverse mode takes more.
  The cost of memory for Forward mode is from:
  Storing size (1) in each variable
  Storing input size n in each gradient variable
  The cost of memory for Reverse mode is from:
  Storing size (1) in each variable
  Storing output size m in each Adjoint variable
  Storing DAG(directed acyclic graph,which present the function)

Memory Required

- It's more likely to have less memory used while using forward mode:
  1.If there exists reused variable in original function
  2.If n is so large that Reverse requires lots of memory to store DAG.
  It's more likely to have less memory used while using reverse mode:
  1.If n is relatively large, so the storage required for storing gradient is more than storing Adjoint

Time to develop

- ***Time to develop***: Usually, it's hard to develop Reverse code than Forward one, especially using Source Transformation technique.

Time to develop

- ***Conclusion***:
  Using Forward mode when $n \gg m$, such as optimization
  Using Reverse mode when $m \gg n$, such as Sensitivity Analysis

Extended knowledge

- ***Directional Derivatives***
  Forward mode:
  seed $\mathbf{d} = (d_1, ...d_n)_T$
  seeding $\nabla x_i = d_i$
  calculates $Jf * d$
  Multi-directional derivatives : replace d by D,where
  $D = [d_{ij}]_{i=1,..n,j=1,..q}$

Extended knowledge

- ***Directional Adjoints***
  Reverse mode:
  seed $\mathbf{v} = (v_1, ...v_m)$
  seeding $\overline{y}_j = v_j$
  calculates $v * Jf$
  Multi-directional Adjoint : replace v by V,where
  $V = [v_{ij}]_{i=1,..q,j=1,..m}$

Case Study

- ***Using FADBAD++***:
- FADBAD++ were developed by Ole Stauning and Claus Bendtsen.
- Flexible automatic differentiation using templates and operator overloading in ANSI C++
- Only with source code, no additional library required.
- Free to use

Case Study

- ***Using FADBAD++***:
  Test function : $f(x) = \prod x_i$
  Objective: Testing different coding of the function in Forward mode, try to reuse the variable
  Result : Basically, no matter how you code,the memory cost as much as $n * n * 8byte$, no different between reuse variable or not

Case Study

- ***Using FADBAD++***:
  Test function : $f(x) = \prod x_i$
  Objective: Testing Reverse mode
  Result : test until $n = 6500$, Using Forward mode out of memory. Reverse is 127 times faster, and only take few MB.
  Remark : Couldn't see how the DAG take the memory from using reverse mode, it's more likely to observe by using fewer independent variables but more complicated function.

Code List

Code-List given by re-writing the code into elemental binary and unary operations/functions, e.g.

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} log^2(x_1 x_2) + x_2 x_3^2 - a - x_2 \\ \sqrt{b \cdot log(x_1 x_2) + x_2/x_3} - x_2 x_3^2 + a \end{bmatrix}$$

$$\begin{array}{lll} v_1 = x_1 & v_7 = v_6 * v_2 & v_{13} = v_8 - v_2 \\ v_2 = x_2 & v_8 = v_7 - a & v_{14} = v_5^2 \\ v_3 = x_3 & v_9 = 1/v_3 & v_{15} = \sqrt{v_{12}} \\ v_4 = v_1 * v_2 & v_{10} = v_2 * v_9 & v_{16} = v_{14} + v_{13} \\ v_5 = log(v_4) & v_{11} = b * v_5 & v_{17} = v_{15} - v_8 \\ v_6 = v_3^2 & v_{12} = v_{11} + v_{10} & \end{array}$$

Code-list (ctd.)

- Assume code-list contains
  - $N_{\pm}$ addition/substractions e.g $v_{14} + v_{13}$
  - N_* multiplications e.g. $v_1 * v_2$
  - N_f nonlinear functions/operations e.g. $log(v_4)$, $1/v_3$
  - Total of $p + m = N_{\pm} + N_* + N_f$ statements
- Then
  - Each addition/subtraction generates two $c_{i,j} = \pm 1$
  - Each multiplication generates two $c_{i,j} \neq \pm 1, 0$
  - Each nonlinear function generates one $c_{i,j} \neq 1, 0$ requiring one nonlinear function evaluation e.g. $v_5 = log(v_4)$ gives $c_{5,4} = 1/v_4$.
- So we have,

$$\begin{aligned} N_{|c|=1} &= 2N_{\pm} \\ N_{|c|\neq 1} &= 2N_* + 1N_f \end{aligned}$$

Complexity of Forward Mode

$$flops(Jf) = flops(f) + flops(c_{i,j}) + flops(fwd)$$

- Assume $flops(nonlinear\ function) = w,\ w > 1.$
- Cost of evaluation function is,

$$flops(f) = N_* + N_\pm + wN_f$$

- Cost of evaluation local derivatives $c_{i,j}$ is,

$$flops(c_{i,j}) = wN_f.$$

- Cost of forward propagation of derivatives is

$$\begin{aligned} flops(fwd) &= n(2N_{|c|\neq 1} + N_{|c|=1} - p - m) \\ &= n(3N_* + N_\pm + N_f) \end{aligned}$$

Complexity of Forward Mode (Ctd.)

Then for forward mode

$$\begin{aligned}\frac{flops(Jf)}{flops(f)} &= 1 + \frac{wN_f + n(3N_* + N_\pm + N_f)}{N_* + N_\pm + wN_f} \\ &= 1 + 3n\widehat{N}_* + n\widehat{N}_\pm + n(\tfrac{1}{w} + \tfrac{1}{n})\widehat{wN_f}\end{aligned}$$

where,

$$(\widehat{N}_*, \widehat{N}_\pm, \widehat{wN_f}) = \frac{(N_*, N_\pm, wN_f)}{N_* + N_\pm + wN_f}.$$

Since $\widehat{N}_* + \widehat{N}_\pm + \widehat{wN_f} = 1$ and all coefficients positive,

$$\frac{flops(Jf)}{flops(f)} \leq 1 + n * max(3, 1, (\frac{1}{w} + \frac{1}{n})) = 1 + 3n.$$

$n << m$, Forward Mode preferred.

Complexity of Reverse Mode

$$flops(rev) = m(4N_* + 2N_\pm + 2N_f),$$

giving,

$$\frac{flops(Jf)}{flops(f)} = 1 + 4m\widehat{N}_* + 2m\widehat{N}_\pm + m(\tfrac{2}{w} + \tfrac{1}{m})\widehat{wN}_f$$

and

$$\frac{flops(Jf)}{flops(f)} \leq 1 + m * max(4, 2, (\frac{2}{w} + \frac{1}{m})) = 1 + 4m$$

For $m = 1$

$$flops(\bigtriangledown f) \leq 5 flops(f)$$

Differentiation Arithmetic

$$\overrightarrow{u} = (u, u'),$$

where u denotes the value of the function u: $\mathbb{R} \to \mathbb{R}$ evaluated at the point x_0, and where u' denotes the value $u'(x_0)$.

$$\begin{aligned}
\overrightarrow{u} + \overrightarrow{v} &= (u + v, u' + v') \\
\overrightarrow{u} - \overrightarrow{v} &= (u - v, u' - v') \\
\overrightarrow{u} \times \overrightarrow{v} &= (uv, uv' + u'v) \\
\overrightarrow{u} \div \overrightarrow{v} &= (u/v, u' - (u/v)v'/v) \\
\overrightarrow{x} &= (x, 1) \\
\overrightarrow{c} &= (c, 0)
\end{aligned}$$

Ref:http://www.math.uu.se/ warwick/vt07/FMB/avnm1.pdf

Differentiation Arithmetic

$$\overrightarrow{u} = (u, u'),$$

where u denotes the value of the function u: $\mathbb{R} \rightarrow \mathbb{R}$ evaluated at the point x_0, and where u' denotes the value $u'(x_0)$.

$$\begin{aligned}
\overrightarrow{u} + \overrightarrow{v} &= (u + v, u' + v') \\
\overrightarrow{u} - \overrightarrow{v} &= (u - v, u' - v') \\
\overrightarrow{u} \times \overrightarrow{v} &= (uv, uv' + u'v) \\
\overrightarrow{u} \div \overrightarrow{v} &= (u/v, u' - (u/v)v'/v) \\
\overrightarrow{x} &= (x, 1) \\
\overrightarrow{c} &= (c, 0)
\end{aligned}$$

Ref:http://www.math.uu.se/ warwick/vt07/FMB/avnm1.pdf

Differentiation Arithmetic

$$\overrightarrow{u} = (u, u'),$$

where u denotes the value of the function u: $\mathbb{R} \to \mathbb{R}$ evaluated at the point x_0, and where u' denotes the value $u'(x_0)$.

$$\begin{aligned}
\overrightarrow{u} + \overrightarrow{v} &= (u + v, u' + v') \\
\overrightarrow{u} - \overrightarrow{v} &= (u - v, u' - v') \\
\overrightarrow{u} \times \overrightarrow{v} &= (uv, uv' + u'v) \\
\overrightarrow{u} \div \overrightarrow{v} &= (u/v, u' - (u/v)v'/v) \\
\overrightarrow{x} &= (x, 1) \\
\overrightarrow{c} &= (c, 0)
\end{aligned}$$

Ref:http://www.math.uu.se/ warwick/vt07/FMB/avnm1.pdf

Example of a Rational Function

$$f(x) = \frac{(x+1)(x-2)}{x+3}$$
$$f(3) = 2/3,\ f'(3) = ?$$

$$\vec{f}(\vec{x}) = \frac{(\vec{x} + \vec{1})(\vec{x} - \vec{2})}{(\vec{x} + \vec{3})} = \frac{((x,1)+(1,0)) \times ((x,1)-(2,0))}{((x,1)+(3,0))}$$

Inserting the value $\vec{x} = (3,1)$ into $\vec{f}$ produces

$$\begin{aligned}\vec{f}(3,1) &= \frac{((3,1)+(1,0)) \times ((3,1)-(2,0))}{((3,1)+(3,0))} \\ &= \frac{(4,1) \times (1,1)}{(6,1)} \\ &= \frac{(4,5)}{(6,1)} = \left(\frac{2}{3}, \frac{13}{18}\right)\end{aligned}$$

Example of a Rational Function

$$f(x) = \frac{(x+1)(x-2)}{x+3}$$
$$f(3) = 2/3,\ f'(3) = ?$$

$$\vec{f}(\vec{x}) = \frac{(\vec{x} + \vec{1})(\vec{x} - \vec{2})}{(\vec{x} + \vec{3})} = \frac{((x,1)+(1,0)) \times ((x,1)-(2,0))}{((x,1)+(3,0))}$$

Inserting the value $\vec{x} = (3,1)$ into $\vec{f}$ produces

$$\begin{aligned}\vec{f}(3,1) &= \frac{((3,1)+(1,0)) \times ((3,1)-(2,0))}{((3,1)+(3,0))} \\ &= \frac{(4,1)\times(1,1)}{(6,1)} \\ &= \frac{(4,5)}{(6,1)} = \left(\frac{2}{3}, \frac{13}{18}\right)\end{aligned}$$

Example of a Rational Function

$$f(x) = \frac{(x+1)(x-2)}{x+3}$$
$$f(3) = 2/3,\ f'(3) = ?$$

$$\vec{f}(\vec{x}) = \frac{(\vec{x} + \vec{1})(\vec{x} - \vec{2})}{(\vec{x} + \vec{3})} = \frac{((x,1) + (1,0)) \times ((x,1) - (2,0))}{((x,1) + (3,0))}$$

Inserting the value $\vec{x} = (3,1)$ into $\vec{f}$ produces

$$\begin{aligned} \vec{f}(3,1) &= \frac{((3,1) + (1,0)) \times ((3,1) - (2,0))}{((3,1) + (3,0))} \\ &= \frac{(4,1) \times (1,1)}{(6,1)} \\ &= \frac{(4,5)}{(6,1)} = \left(\frac{2}{3}, \frac{13}{18}\right) \end{aligned}$$

Derivatives of Element Functions

Chain Rule:

$$(g \circ u)'(x) = u'(x)(g' \circ u)(x)$$

$$\vec{g}(\vec{u}) = \vec{g}((u, u')) = (g(u), u'g'(u))$$

$$\begin{aligned}
\sin \vec{u} &= \sin(u, u') = (\sin u, u' \cos u) \\
\cos \vec{u} &= \cos(u, u') = (\cos u, -u' \sin u) \\
e^{\vec{u}} &= e^{(u,u')} = (e^u, u'e^u) \\
&\vdots
\end{aligned}$$

Derivatives of Element Functions

Chain Rule:

$$(g \circ u)'(x) = u'(x)(g' \circ u)(x)$$

$$\vec{g}(\vec{u}) = \vec{g}((u, u')) = (g(u), u'g'(u))$$

$$\begin{aligned}
\sin \vec{u} &= \sin(u, u') = (\sin u, u' \cos u) \\
\cos \vec{u} &= \cos(u, u') = (\cos u, -u' \sin u) \\
e^{\vec{u}} &= e^{(u, u')} = (e^u, u' e^u) \\
&\vdots
\end{aligned}$$

Derivatives of Element Functions

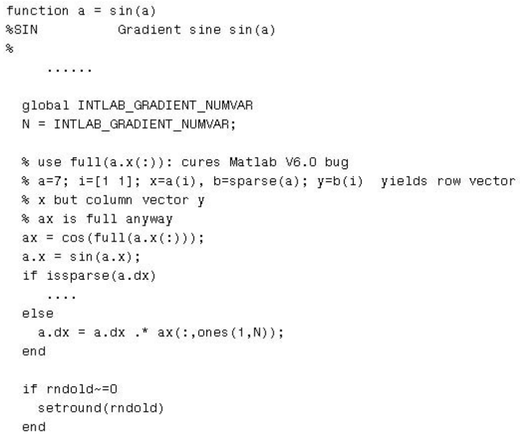
Chain Rule:

$$(g \circ u)'(x) = u'(x)(g' \circ u)(x)$$

$$\overrightarrow{g}(\overrightarrow{u}) = \overrightarrow{g}((u, u')) = (g(u), u'g'(u))$$

$$\begin{aligned}
\sin \overrightarrow{u} &= \sin(u, u') = (\sin u, u' \cos u) \\
\cos \overrightarrow{u} &= \cos(u, u') = (\cos u, -u' \sin u) \\
e^{\overrightarrow{u}} &= e^{(u,u')} = (e^u, u'e^u) \\
&\vdots
\end{aligned}$$

Example of Sin



```
function a = sin(a)
%SIN          Gradient sine sin(a)
%
     ......

  global INTLAB_GRADIENT_NUMVAR
  N = INTLAB_GRADIENT_NUMVAR;

  % use full(a.x(:)): cures Matlab V6.0 bug
  % a=7; i=[1 1]; x=a(i), b=sparse(a); y=b(i)  yields row vector
  % x but column vector y
  % ax is full anyway
  ax = cos(full(a.x(:)));
  a.x = sin(a.x);
  if issparse(a.dx)
     ....
  else
    a.dx = a.dx .* ax(:,ones(1,N));
  end

  if rndold~=0
    setround(rndold)
  end
```

From ../*Intlab*/*gradient*/@*gradient*/*sin*.*m*

Example for Element Functions

Evaluate the derivative at x=0.

$$\begin{aligned}
f(x) &= (1 + x + e^x)\sin x \\
\overrightarrow{f}(\overrightarrow{x}) &= (\overrightarrow{1} + \overrightarrow{x} + e^{\overrightarrow{x}}) sin \overrightarrow{x} \\
\overrightarrow{f}(0,1) &= \left((1,0) + (0,1) + e^{(0,1)}\right)\sin(0,1) \\
&= \left((1,1) + (e^0, e^0)\right)(\sin 0, \cos 0) \\
&= (2,2)(0,1) = (0,2).
\end{aligned}$$

Example for Element Functions

Evaluate the derivative at x=0.

$$
\begin{aligned}
f(x) &= (1 + x + e^x)\sin x \\
\overrightarrow{f}(\overrightarrow{x}) &= (\overrightarrow{1} + \overrightarrow{x} + e^{\overrightarrow{x}})sin\overrightarrow{x} \\
\overrightarrow{f}(0,1) &= \left((1,0) + (0,1) + e^{(0,1)}\right)\sin(0,1) \\
&= \left((1,1) + (e^0, e^0)\right)(\sin 0, \cos 0) \\
&= (2,2)(0,1) = (0,2).
\end{aligned}
$$

Example for Element Functions

Evaluate the derivative at x=0.

$$\begin{aligned}
f(x) &= (1 + x + e^x)\sin x \\
\overrightarrow{f}(\overrightarrow{x}) &= (\overrightarrow{1} + \overrightarrow{x} + e^{\overrightarrow{x}}) sin \overrightarrow{x} \\
\overrightarrow{f}(0,1) &= \Big((1,0) + (0,1) + e^{(0,1)}\Big) \sin(0,1) \\
&= \Big((1,1) + (e^0, e^0)\Big)(\sin 0, \cos 0) \\
&= (2,2)(0,1) = (0,2).
\end{aligned}$$

High-order Derivatives

$$\overrightarrow{u} = (u, u', u''),$$

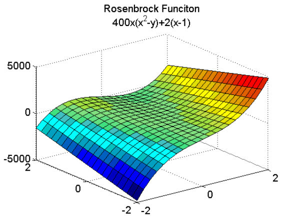
$$\begin{aligned}
\overrightarrow{u} + \overrightarrow{v} &= (u + v, u' + v', u'' + v'') \\
\overrightarrow{u} - \overrightarrow{v} &= (u - v, u' - v', u'' - v'') \\
\overrightarrow{u} \times \overrightarrow{v} &= (uv, uv' + u'v, uv'' + 2u'v' + u''v') \\
\overrightarrow{u} \div \overrightarrow{v} &= (u/v, u' - (u/v)v'/v, (u'' - 2(u/v)'v' - (u/v)v'')/v) \\
\cdots\cdots &
\end{aligned}$$

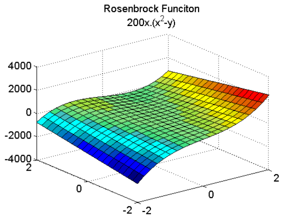
INTLab

Developers:	Institute for Reliable Computing, Hamburg University of Technology
Mode:	Forward
Method:	Operator overloading
Language:	MATLAB
URL:	http://www.ti3.tu-harburg.de/rump/intlab/
Licensing:	Open Source

Rosenbrock Function

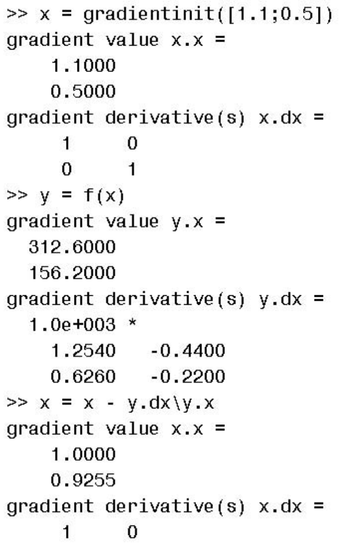
$$\begin{aligned} y1 &= 400x_1(x_1^2 - x_2) + 2(x_1 - 1) \\ y2 &= 200(x_1^2 - x_2) \end{aligned}$$





```
f = inline('[ 400*x(1)*(x(1)*x(1)-x(2)) + 2*(x(1)-1); 200*x(1)*(x(1)*x(1)-x(2))]')
```

One Step of Newton Method with INTLab

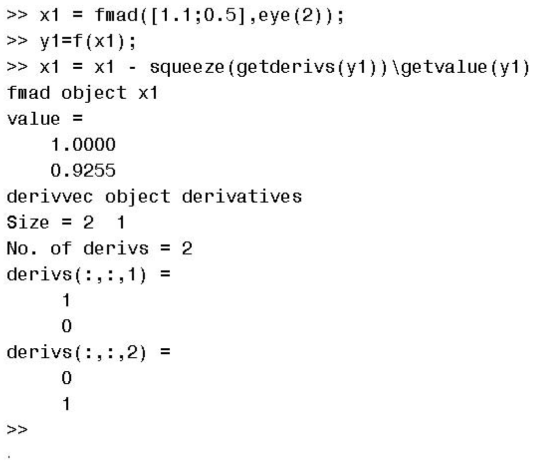


```
>> x = gradientinit([1.1;0.5])
gradient value x.x =
    1.1000
    0.5000
gradient derivative(s) x.dx =
     1     0
     0     1
>> y = f(x)
gradient value y.x =
  312.6000
  156.2000
gradient derivative(s) y.dx =
  1.0e+003 *
    1.2540   -0.4400
    0.6260   -0.2200
>> x = x - y.dx\y.x
gradient value x.x =
    1.0000
    0.9255
gradient derivative(s) x.dx =
     1     0
```

TOMLAB/MAD

Developers:	Marcus M. Edvall and Kenneth Holmstrom, Tomlab Optimization Inc. (TOMLAB /MAD integration) Shaun A. Forth and Robert Ketzscher, Cranfield University (MAD)
Mode:	Forward
Method:	Operator overloading
Language:	MATLAB
URL:	http://tomlab.biz/products/mad/
Licensing:	License

One Step of Newton Method with MAD



```
>> x1 = fmad([1.1;0.5],eye(2));
>> y1=f(x1);
>> x1 = x1 - squeeze(getderivs(y1))\getvalue(y1)
fmad object x1
value =
    1.0000
    0.9255
derivvec object derivatives
Size = 2   1
No. of derivs = 2
derivs(:,:,1) =
     1
     0
derivs(:,:,2) =
     0
     1
>>
```

ADiMat

Developers:	Andre Vehreschild, Institute for Scientific Computing, RWTH Aachen University
Mode:	Forward
Method:	Source transformation Operator overloading
Language:	MATLAB
URL:	http://www.sc.rwth-aachen.de/vehreschild/adimat.html
Licensing:	under discussion

ADiMat's Example

```
function [result1, result2]= f(x)
  % Compute the sin and square-root of x*2.
  % Very simple example for ADiMat website.
  % Andre Vehreschild, Institute for
  % Scientific Computing,
  % RWTH Aachen University, D-52056 Aachen,
  % Germany.
  % vehreschild@sc.rwth-aachen.de

  result1= sin(x);
  result2= sqrt(x*2);
```

Source:http://www.sc.rwth-aachen.de/vehreschild/adimat/example1.html

ADiMat's Example (cont.)

```
>> addiff(@f, 'x', 'result1,result2');
>> p=magic(5);
>> g_p=createFullGradients(p);
>> [g_r1, r1, g_r2, r2]= g_f(g_p, p);
>> J1= [g_r1{:}]; % and
>> J2= [g_r2{:}];
```

Source: http://www.sc.rwth-aachen.de/vehreschild/adimat/example1.html

ADiMat's Example (cont.)

```
function [g_result1, result1, g_result2, result2] = g_f(
    % Compute the sin and square-root of x*2.
    % Very simple example for ADiMat website.
    % Andre Vehreschild, Institute for Scientific Computi
    % RWTH Aachen University, D-52056 Aachen, Germany.
    % vehreschild@sc.rwth-aachen.de
g_result1= ((g_x).* cos(x));
result1= sin(x);
g_tmp_f_00000= g_x* 2;
tmp_f_00000= x* 2;
g_result2= ((g_tmp_f_00000)./ (2.*
sqrt(tmp_f_00000)));
result2= sqrt(tmp_f_00000);
```

Source:http://www.sc.rwth-aachen.de/vehreschild/adimat/example1.html

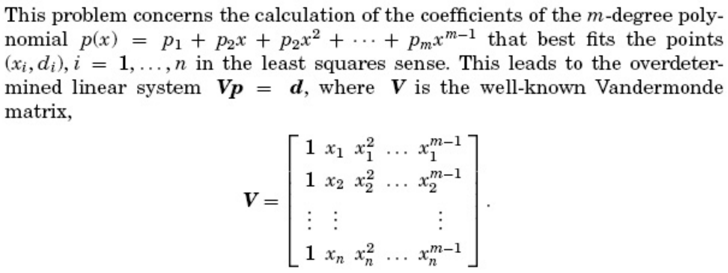
Matrix Calculus

Definition: If X is $p \times q$ and Y is $m \times n$, then dY: = dY/dX dX: where the derivative dY/dX is a large $mn \times pq$ matrix.

$$\begin{aligned} d(X^2): &= (XdX + dXX): \\ d(det(X)) &= d(det(X^T)) = det(X)(X^{-T}):^T dX: \\ d(ln(det(X))) &= (X^{-T}):^T dX: \end{aligned}$$

Ref: http://www.ee.ic.ac.uk/hp/staff/dmb/matrix/calculus.html

Vandermonde Function



This problem concerns the calculation of the coefficients of the m-degree polynomial $p(x) = p_1 + p_2 x + p_2 x^2 + \cdots + p_m x^{m-1}$ that best fits the points $(x_i, d_i), i = 1, \ldots, n$ in the least squares sense. This leads to the overdetermined linear system $\boldsymbol{V}\boldsymbol{p} = \boldsymbol{d}$, where $\boldsymbol{V}$ is the well-known Vandermonde matrix,

$$\boldsymbol{V} = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{m-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{m-1} \\ \vdots & \vdots & & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{m-1} \end{bmatrix}.$$

Source: Shaun A. Forth *An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB* ACM Transactions on Mathematical Software, Vol. 32,No.2, 2006, P195-222

Vandermonde Function (cont.)

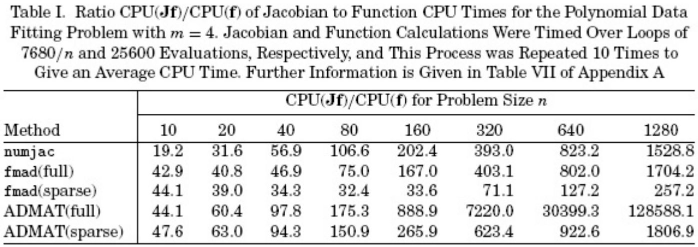


Table I. Ratio CPU(**Jf**)/CPU(**f**) of Jacobian to Function CPU Times for the Polynomial Data Fitting Problem with $m = 4$. Jacobian and Function Calculations Were Timed Over Loops of $7680/n$ and 25600 Evaluations, Respectively, and This Process was Repeated 10 Times to Give an Average CPU Time. Further Information is Given in Table VII of Appendix A

	CPU(**Jf**)/CPU(**f**) for Problem Size n							
Method	10	20	40	80	160	320	640	1280
`numjac`	19.2	31.6	56.9	106.6	202.4	393.0	823.2	1528.8
`fmad`(full)	42.9	40.8	46.9	75.0	167.0	403.1	802.0	1704.2
`fmad`(sparse)	44.1	39.0	34.3	32.4	33.6	71.1	127.2	257.2
ADMAT(full)	44.1	60.4	97.8	175.3	888.9	7220.0	30399.3	128588.1
ADMAT(sparse)	47.6	63.0	94.3	150.9	265.9	623.4	922.6	1806.9

Experiment on a PIV 3.0Ghz PC (Windows XP), Matlab Version: 6.5

Source: Shaun A. Forth *An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB* ACM Transactions on Mathematical Software, Vol. 32,No.2, 2006, P195-222

Vandermonde Function (cont.)

Method	10	20	40	80	160	320	640	1280
Function	0.000	0.000	0.000	0.000	0.000	0.010	0.000	0.000
MAD(Full)	0.070	0.060	0.070	0.130	0.581	2.664	10.535	45.535
MAD(Sparse)	0.071	0.050	0.060	0.060	0.060	0.070	0.100	0.881
INTLab	0.050	0.040	0.040	0.090	0.040	0.050	0.071	0.120
ADiMat	0.231	0.140	0.271	0.601	1.362	3.044	7.340	21.611

Unit of CPU time is second. Experiment on a PIII1000Hz PC (Windows 2000), Matlab Version: 7.0.1.24704 (R14)

Service Pack 1, TOMLAB v5.6, INTLAB Version 5.3, ADiMat (beta) 0.4-r9.

Arrowhead Function

$$\left.\begin{array}{ll} f_1 & = 2x_1^2 + \sum_{i=1}^n x_i^2 \\ f_i & = x_1^2 + x_i^2, \qquad\qquad i = 2, \dots, n \end{array}\right\},$$

for $n = 7$, the Jacobian has sparsity pattern,

$$\mathbf{Jf}(\boldsymbol{x}) = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & & & & & \\ \bullet & & \bullet & & & & \\ \bullet & & & \bullet & & & \\ \bullet & & & & \bullet & & \\ \bullet & & & & & \bullet & \\ \bullet & & & & & & \bullet \end{bmatrix},$$

Source: Shaun A. Forth *An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB* ACM Transactions on Mathematical Software, Vol. 32,No.2, 2006, P195-222

Arrowhead Function (cont.)

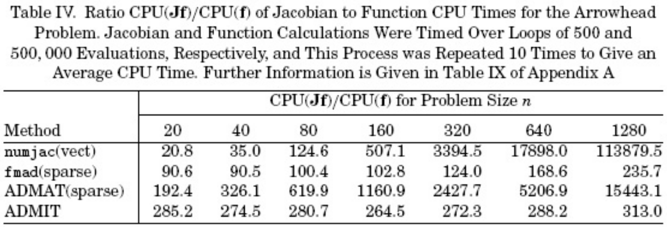


Table IV. Ratio CPU(**Jf**)/CPU(**f**) of Jacobian to Function CPU Times for the Arrowhead Problem. Jacobian and Function Calculations Were Timed Over Loops of 500 and 500, 000 Evaluations, Respectively, and This Process was Repeated 10 Times to Give an Average CPU Time. Further Information is Given in Table IX of Appendix A

	CPU(**Jf**)/CPU(**f**) for Problem Size n						
Method	20	40	80	160	320	640	1280
`numjac`(vect)	20.8	35.0	124.6	507.1	3394.5	17898.0	113879.5
`fmad`(sparse)	90.6	90.5	100.4	102.8	124.0	168.6	235.7
ADMAT(sparse)	192.4	326.1	619.9	1160.9	2427.7	5206.9	15443.1
ADMIT	285.2	274.5	280.7	264.5	272.3	288.2	313.0

Experiment on a PIV 3.0Ghz PC (Windows XP), Matlab Version: 6.5

Source: Shaun A. Forth *An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB* ACM Transactions on Mathematical Software, Vol. 32,No.2, 2006, P195-222

Arrowhead Function (cont.)

Method	20	40	80	160	320	640	1280
Function	0.010	0.000	0.000	0.000	0.000	0.000	0.000
MAD(Full)	0.180	0.050	0.070	0.200	1.111	4.367	17.796
MAD(Sparse)	0.060	0.060	0.060	0.070	0.080	0.100	0.160
INTLab	0.090	0.051	0.050	0.050	0.081	0.140	0.340
ADiMat	0.911	0.311	0.651	1.262	2.704	6.028	14.581

Unit of CPU time is second. Experiment on a PIII1000Hz PC (Windows 2000), Matlab Version: 7.0.1.24704 (R14)

Service Pack 1, TOMLAB v5.6, INTLAB Version 5.3, ADiMat (beta) 0.4-r9.

BDQRTIC mod

```
  function  y = bdqrtic(x)
%  http://www.sor.princeton.edu/~rvdb/ampl/nlmodels/cute/bdqrtic.mod
% Source: Problem 61 in
% A.R. Conn, N.I.M. Gould, M. Lescrenier and Ph.L. Toint,
% "Performance of a multifrontal scheme for partially separable
% optimization",
% Report 88/4, Dept of Mathematics, FUNDP (Namur, B), 1988.
% Copyright (C) 2001 Princeton University
% All Rights Reserved
    N = length(x);
    I = 1:N-4;
    y = sum( (-4*x(I)+3.0).^2 ) + sum( ( x(I).^2 + 2*x(I+1).^2 +...
        3*x(I+2).^2 + 4*x(I+3).^2 + 5*x(N).^2 ).^2 );
```

BDQRTIC mod (cont.)

Method	20	40	80	160	320	640	1280
Function	12.809	0.010	0.000	0.000	0.000	0.010	0.000
MAD(Full)	2.604	0.121	0.150	0.490	2.513	10.926	43.162
MAD(Sparse)	0.270	0.120	0.130	0.150	0.201	0.260	0.371
INTLab	2.293	0.080	0.100	0.110	0.150	0.230	0.481
ADiMat	3.455	0.621	1.152	2.544	5.778	14.641	42.671

Unit of CPU time is second. Experiment on a PIII1000Hz PC (Windows 2000), Matlab Version: 7.0.1.24704 (R14)

Service Pack 1, TOMLAB v5.6, INTLAB Version 5.3, ADiMat (beta) 0.4-r9.

Summary of AD softwares in MATLab

- Operator overloading method for AD forward mode is easy to implement by differentiation arithmetic.
- All of AD tools in Matlab are easy to use.
- Sparse storage provides a good way to improve the performance of AD tools.

The Computational Differentiation Group at Argonne National Laboratory

ADIC introduced in 1997 by:

Chrirtian Bischof
Scientific Computing at
RWTH Aachen University

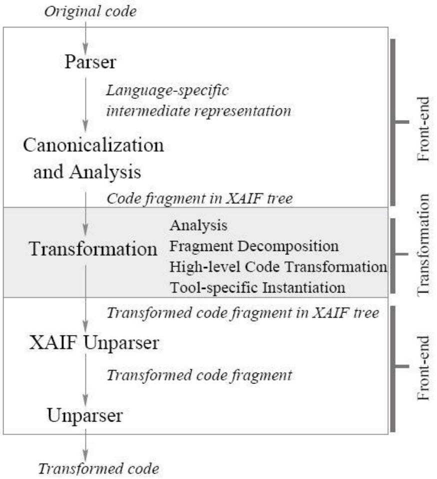
Lucas Roh
founder, president and
CEO of Hostway Co.

and the other team members.

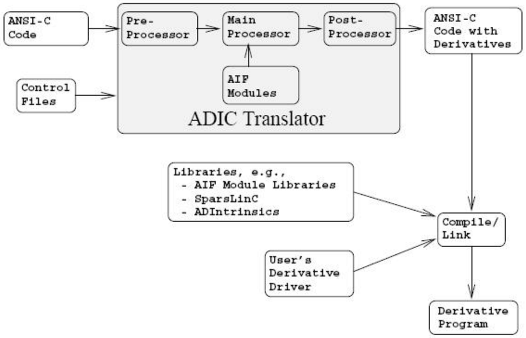
State of ADIS

- ADIC is an **A**utomatic **D**ifferentiation tools **I**n ANSI **C**/C++.
- ADIC was introduced in 1966.
- Last updated: June 10, 2005.
- Official web site www-new.mcs.anl.gov/adic/down-2.htm.
- ADIC is using forward method.
- Supported Platforms: Unix/Linux.
- Selected Application: NEOS
- Related Research Group: Argonne National Laboratory, USA

ADICAnatomy


Original code
Parser
Language-specific intermediate representation
Canonicalization and Analysis
Code fragment in XAIF tree
Transformation
Analysis
Fragment Decomposition
High-level Code Transformation
Tool-specific Instantiation
Transformed code fragment in XAIF tree
XAIF Unparser
Transformed code fragment
Unparser
Transformed code
Front-end
Transformation
Front-end

ADICProcess



func.c

```
#include "func.h"
#include <math.h>

void func(data_t * pdata)
{
   int i;
   double *x = pdata->x;
   double *y = pdata->y;
   double s, temp;


   i=0;
   for (;i < pdata->len ;){
        s = s + x[i]*y[i];
        i++;
   }

   temp = exp(s);

   pdata->r = temp;
}
```

driver.c

```
#include "ad_deriv.h"
#include<stdio.h>

#define MAXLEN 100

typedef struct {
        int     len;
        DERIV_TYPE *x, *y, r;
} data_t;
void ad_func(data_t *);

int main(){
   int i, n;
   double grad[ad_GRAD_MAX], t1, t2;
   data_t data;
   DERIV_TYPE x[MAXLEN], y[MAXLEN], r;

   ad_AD_Init(ad_GRAD_MAX);

   scanf("%d", &n);
   for (i=0; i< n ; i++) {
        scanf("%lf %lf", &t1, &t2);
        ad_grad_axpy_0(&(x[i]));
        DERIV_val(x[i])=t1;
        ad_grad_axpy_0(&(y[i]));
        DERIV_val(y[i])=t2;
   }
   data.len = n;
   data.x = x;
   data.y = y;
```

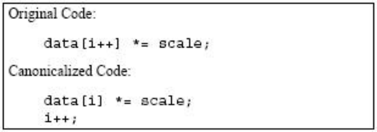
```
   ad_AD_SetIndepArray(x,n);
   ad_AD_SetIndepArray(y,n);
   ad_AD_SetIndepDone();

   ad_func(&data);

   ad_AD_ExtractGrad(grad, data.r);

   printf("%le\n",DERIV_val(data.r));
   for (i=0; i<n ; i++){
       printf("%le\n",grad[i]);
   }

   ad_AD_Final();
   return 0;
}
```

Commands

```
% adiC -vd gradient func.c

% gcc -I$ADIC/include -o ad_func func.ad.c func.c func_driver.c
        -L$ADIC/lib/$ADI_ARCH -lADIntrinsics-C -laif_grad -lm
```

- The first command generates the header file ad_deriv.h and derivative function func.ad.c;
- The second command compiles and links all needed functions and generates ad_func;

Handling Side Effects

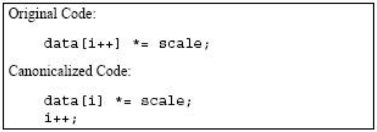

Original Code:

```
    data[i++] *= scale;
```

Canonicalized Code:

```
    data[i] *= scale;
    i++;
```

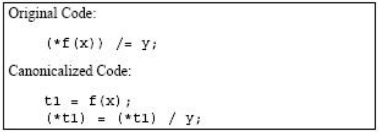
Handling Side Effects


Original Code:

```
    data[i++] *= scale;
```

Canonicalized Code:

```
    data[i] *= scale;
    i++;
```


Original Code:

```
    (*f(x)) /= y;
```

Canonicalized Code:

```
    t1 = f(x);
    (*t1) = (*t1) / y;
```

Handling Side Effects

Original Code:

```
    data[i++] *= scale;
```

Canonicalized Code:

```
    data[i] *= scale;
    i++;
```

Original Code:

```
    (*f(x)) /= y;
```

Canonicalized Code:

```
    t1 = f(x);
    (*t1) = (*t1) / y;
```

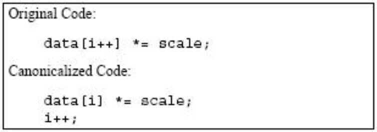
Original Code:

```
  unsigned long get_information (int key)
  double x,y;
  int key;

  y = x * (double) get_information (key);
```

Canonicalized Code:

```
  unsigned long get_information (int key)
  double tmp,x,y;
  int key;

  tmp = (double) get_information (key);
  y = x * tmp;
```

Handling Side Effects

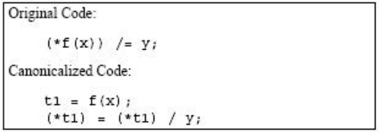


Original Code:

```
    data[i++] *= scale;
```

Canonicalized Code:

```
    data[i] *= scale;
    i++;
```

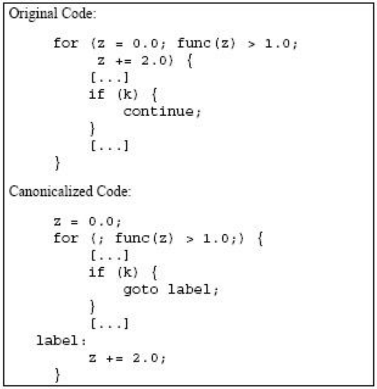


Original Code:

```
    (*f(x)) /= y;
```

Canonicalized Code:

```
    t1 = f(x);
    (*t1) = (*t1) / y;
```

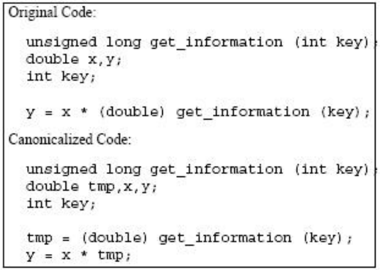


Original Code:

```
    for (z = 0.0; func(z) > 1.0;
         z += 2.0) {
        [...]
        if (k) {
            continue;
        }
        [...]
    }
```

Canonicalized Code:

```
    z = 0.0;
    for (; func(z) > 1.0;) {
        [...]
        if (k) {
            goto label;
        }
        [...]
  label:
        z += 2.0;
    }
```



Original Code:

```
  unsigned long get_information (int key)
  double x,y;
  int key;

  y = x * (double) get_information (key);
```

Canonicalized Code:

```
  unsigned long get_information (int key)
  double tmp,x,y;
  int key;

  tmp = (double) get_information (key);
  y = x * tmp;
```

For Further Reading in ADIC

Christian H. Bischof, Paul D. Hovland, Boyana Norris
Implementation of Automatic Differentiation Tools.
PEPM Š02, Jan. 1415, 2002 Portland, OR, USA

Paul D. Hovlan and Boyana Norris
Users' Guide to ADIC 1.1.
UsersŠ Guide to ADIC 1.1

C. H. Bischof, L. Roh, A. J. Mauer-Oats
ADIC: an extensible automatic differentiation tool for ANSI-C.
Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA

Reference

- C.H. Bischof and H. M. Bucker. *Computing Derivatives of Computer Programs*, in Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition, edited by J. Grotendorst, NIC-Directors,2000, pages 315-327
- C. Bischof, A. Carle, P. Khademi, and G. Pusch. *Automatic Differentiation: Obtaining Fast and Reliable Derivatives-Fast*, in Control Problems in Industry, edited by I. Lasiecka and B. Morton,1995, pages 1-16
- Andreas Griewank. *On Automatic Differentiation*, in Mathematical Programming: Recent Developments and Applications, edited by M. Iri and K. Tanabe, Kluwer Academic Publishers, 1989.
- Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, Penn., 2000.
- Shaun Forth. *Introduction to Automatic Differentiation*, presentation slide for The 4th International Conference on Automatic Differentiation. July 19-23 University of Chicago, Gleacher Centre, Chicago USA, 2004.
- G. F. Corliss, *Automatic Differentiation*.
- Warwick Tucker, http://www.math.uu.se/ warwick/vt07/FMB/avnm1.pdf
- http://www.autodiff.org/
- http://www.ti3.tu-harburg.de/rump/intlab/
- http://tomopt.com/tomlab/products/mad/
- http://www.sc.rwth-aachen.de/vehreschild/adimat/index.html
- Shaun A. Forth *An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB* ACM Transactions on Mathematical Software, Vol. 32,No.2, 2006, P195 $-$ 222
- Siegfried M. Rump *INTLAB $--$ INTerval LABoratory* Developments in Reliable Computing, Kluwer Academic Publishers, 1999, p77 $-$ 104
- Christian H. Bischof, H. Martin Bucker, Bruno Lang, A. Rasch, Andre Vehreschild *Combining Source Transformation and Operator Overloading Techniques to Compute Derivatives for MATLAB Programs* Conference proceeding, Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002), IEEE Computer Society, 2002

Thanks!

Questions?