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.
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 $i$th 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 $i$th Cartesian unit vector.
Divided Differences (Ctd.)

\[
\frac{\partial f(x)}{\partial x_i} \bigg|_{x_0} \approx \frac{f(x_0 + he_i) - f(x_0)}{h}
\]

- **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: \((n + 1) \times \text{cost}(f)\)
Divided Differences (Ctd.)

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

- **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: $(n + 1) \times \text{cost}(f)$
Divided Differences (Ctd.)

\[ \frac{\partial f(x)}{\partial x_i} \bigg|_{x_0} \approx \frac{f(x_0 + he_i) - f(x_0)}{h} \]

- **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: \((n + 1) \times \text{cost}(f)\)
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: \((n+1) \times \text{cost}(f)\)
Divided Differences (Ctd.)

\[ \frac{\partial f(x)}{\partial x_i} \bigg|_{x_0} \approx \frac{f(x_0 + he_i) - f(x_0)}{h} \]

- **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: \( (n + 1) \times \text{cost}(f) \)
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.
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.
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** \([y_1, y_2] = f(x_1, x_2, x_3, a, b)\)

\[w_1 = \log(x_1 * x_2)\]
\[w_2 = x_2 * x_3^2 - a\]
\[w_3 = b * w_1 + x_2 / x_3\]
\[y_1 = w_1^2 + w_2 - x_2\]
\[y_2 = \sqrt{w_3 - w_2}\]

We want to calculate the Jacobian \(J_f\),

\[
J_f = \begin{bmatrix}
\nabla y_1 \\
\nabla y_2
\end{bmatrix} = \begin{bmatrix}
\frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\
\frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3}
\end{bmatrix}
\]

Therefore we have

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

Unify all the variable..

\[
\begin{align*}
    u_1 &= x_1 \\
    u_2 &= x_2 \\
    u_3 &= x_3 \\
    u_4 &= \Phi_4(u_1, u_2) = \log(u_1 \cdot u_2) \\
    u_5 &= \Phi_5(u_2, u_3) = u_2 \cdot u_3^2 - a \\
    u_6 &= \Phi_6(u_2, u_3, u_4) = b \cdot u_4 + u_2 / u_3 \\
    u_7 &= \Phi_7(u_2, u_4, u_5) = u_4^2 + u_5 - u_2 \\
    u_8 &= \Phi_8(u_5, u_6) = \sqrt{u_6 - u_5}
\end{align*}
\]
Automatic Differentiation
Forward and Reverse Mode

Forward method

- **Forward method**
  Differentiate the Code:

  \[ u_i = x_i \quad i = 1, ... n, \]

  \[ u_i = \Phi(\{u_j\}_{j<i}) \quad i = n + 1, ..., N \]

  Differentiate:

  \[ \nabla u_i = e_i \quad i = 1, ..., n \]

  \[ \nabla u_i = \sum_{j<i} c_{i,j} \ast \nabla u_j \quad i = n + 1, ..., N \]
Reverse method

Compute the Adjoint of the Code

\[ \bar{u}_j = \frac{\partial y}{\partial u_j} = \frac{\partial (y_1, y_2, \ldots, y_m)}{\partial u_j} \]

Compute for dependent variables

\[ \bar{u}_{n+p+j} = \frac{\partial (y_1, y_2, \ldots, y_m)}{\partial u_j} = e_j \quad j = 1, \ldots, m \]

Compute for intermediates and independents \( u_j, j = n + p, \ldots, 1 \)

\[ \bar{u}_j = \frac{\partial y}{\partial u_j} = \sum_{i > j} \bar{u}_i c_{i,j} \]
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
Automatic Differentiation
Forward and Reverse Mode
Forward methods

- Computing Variable Value
- Computing Gradient Value

Diagram showing the flow of computation and gradient calculation in forward methods.
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 derivative one by one
  - Harder to implement
Reverse methods

- Computing Variable Value
- Computing Adjoint Value
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
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 \[ \begin{cases} N_{|c|=1} : \text{No. of unit local derivatives } c_{i,j} = \pm 1 \\ N_{|c|\neq 1} : \text{No. of nonunit local derivatives } c_{i,j} \neq 0, \pm 1 \end{cases} \]

Solve for derivatives in forward order \( \nabla u_{n+1}, \nabla u_{n+2}, \ldots, \nabla u_N \)

\[ \nabla u_i = \sum_{j<i} c_{i,j} \ast \nabla u_j, \ i = n+1, \ldots, N, \]

with each \( \nabla u_i = (\partial u_i/\partial x_1, \ldots, \partial u_i/\partial x_n) \), a length \( n \) vector.

Flop count \( \text{flops}(fwd) \) given by,

\[
\text{flops}(fwd) = \begin{cases} 
    nN_{|c|\neq 1} 
    + n(N_{|c|\neq 1} + N_{|c|=1}) 
    - n(p + m) 
    & (\text{mults. } c_{i,j} \ast \nabla u_j, \ c_{i,j} \neq 1, 0) 
    
    & (\text{adds./subs. } c_{i,j} \nabla u_j) 
    
    & (\text{first } n \text{ adds./subs.}) 
    
    n(2N_{|c|\neq 1} + N_{|c|=1} - p - m) 
    & 
\end{cases}
\]
Cost of Reverse Propagation of Adoints

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

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

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

- Flop count $\text{flops}(rev)$ given by,

$$\text{flops}(rev) = m N_{|c|\neq 1} \quad (\text{mults.} \bar{u}_i \ast c_{i,j}, \ c_{i,j} \neq \pm 1, 0)$$
$$= +m(N_{|c|=1} + N_{|c|\neq 1}) \quad (\text{adds./subs.} + (\bar{u}_i \ast c_{i,j}))$$

$$\text{flops}(rev) = m(2N_{|c|\neq 1} + N_{|c|=1}).$$
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.
Conclusion:
Using Forward mode when $n \gg m$, such as optimization
Using Reverse mode when $m \gg n$, such as Sensitivity Analysis
Directional Derivatives

Forward mode:
seed \( \mathbf{d} = (d_1, \ldots d_n)^T \)

seeding \( \nabla x_i = d_i \)
calculates \( Jf \ast d \)
Multi-directional derivatives: replace \( \mathbf{d} \) by \( \mathbf{D} \), where
\[
D = [d_{ij}]_{i=1,..n, j=1,..q}
\]
Directional Adjoints

Reverse mode:
seed \( \mathbf{v} = (v_1, \ldots, v_m) \)
seeding \( \mathbf{y}_j = v_j \)
calculates \( \mathbf{v} \ast Jf \)
Multi-directional Adjoint: replace \( \mathbf{v} \) by \( \mathbf{V} \), where
\( \mathbf{V} = [v_{ij}]_{i=1,\ldots,q,j=1,\ldots,m} \)
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
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 \( n \times n \times 8 \text{byte} \), 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 given by re-writing the code into elemental binary and unary operations/functions, e.g.

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

\[
\begin{align*}
  v_1 &= x_1 & v_7 &= v_6 \cdot v_2 & v_{13} &= v_8 - v_2 \\
  v_2 &= x_2 & v_8 &= v_7 - a & v_{14} &= v_5^2 \\
  v_3 &= x_3 & v_9 &= 1 / v_3 & v_{15} &= \sqrt{v_{12}} \\
  v_4 &= v_1 \cdot v_2 & v_{10} &= v_2 \cdot v_9 & v_{16} &= v_{14} + v_{13} \\
  v_5 &= \log(v_4) & v_{11} &= b \cdot v_5 & v_{17} &= v_{15} - v_8 \\
  v_6 &= v_3^2 & v_{12} &= v_{11} + v_{10}
\end{align*}
\]
Assume code-list contains
- $N_\pm$ addition/subtractions e.g. $v_{14} + v_{13}$
- $N_*$ multiplications e.g. $v_1 \times v_2$
- $N_f$ 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,

\[
\begin{align*}
N_{|c|=1} &= 2N_\pm \\
N_{|c|\neq1} &= 2N_* + N_f
\end{align*}
\]
Complexity of Forward Mode

\[ \text{flops}(J\mathbf{f}) = \text{flops}(f) + \text{flops}(c_{i,j}) + \text{flops}(\text{fwd}) \]

- Assume \( \text{flops(\text{nonlinear function})} = w, w > 1. \)
- Cost of evaluation function is,
\[ \text{flops}(f) = N_\ast + N_\pm + wN_f \]
- Cost of evaluation local derivatives \( c_{i,j} \) is,
\[ \text{flops}(c_{i,j}) = wN_f. \]
- Cost of forward propagation of derivatives is
\[ \text{flops}(\text{fwd}) = n(2N_{|c|\neq 1} + N_{|c|=1} - p - m) \]
\[ = n(3N_\ast + N_\pm + N_f) \]
Then for forward mode

\[
\frac{\text{flops}(J_f)}{\text{flops}(f)} = 1 + \frac{wn_f + n(3N_* + N_\pm + N_f)}{N_* + N_\pm + wn_f}
\]

\[
= 1 + 3n\hat{N}_* + n\hat{N}_\pm + n\left(\frac{1}{w} + \frac{1}{n}\right)\hat{w}N_f
\]

where,

\[
(\hat{N}_*, \hat{N}_\pm, \hat{w}N_f) = \left(\frac{N_*, N_\pm, wN_f}{N_* + N_\pm + wN_f}\right).
\]

Since \(\hat{N}_* + \hat{N}_\pm + \hat{w}N_f = 1\) and all coefficients positive,

\[
\frac{\text{flops}(J_f)}{\text{flops}(f)} \leq 1 + n \times \max(3, 1, \left(\frac{1}{w} + \frac{1}{n}\right)) = 1 + 3n.
\]

\(n << m\), Forward Mode preferred.
Complexity of Reverse Mode

\[ \text{flops} (\text{rev}) = m(4N_* + 2N_\pm + 2N_f), \]

giving,

\[ \frac{\text{flops}(Jf)}{\text{flops}(f)} = 1 + 4m\hat{N}_* + 2m\hat{N}_\pm + m\left(\frac{2}{w} + \frac{1}{m}\right)\hat{w}\hat{N}_f \]

and

\[ \frac{\text{flops}(Jf)}{\text{flops}(f)} \leq 1 + m \cdot \max(4, 2, \frac{2}{w} + \frac{1}{m}) = 1 + 4m \]

For \( m = 1 \)

\[ \text{flops}(\nabla f) \leq 5\text{flops}(f) \]
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 \( u'(x_0) \).

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

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

\( \mathbf{u} = (u, u') \),

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 \( u'(x_0) \).

\[
\begin{align*}
\mathbf{u} + \mathbf{v} &= (u + v, u' + v') \\
\mathbf{u} - \mathbf{v} &= (u - v, u' - v') \\
\mathbf{u} \times \mathbf{v} &= (uv, uv' + u'v) \\
\mathbf{u} \div \mathbf{v} &= (u/v, u' - (u/v)v'/v) \\
\mathbf{x} &= (x, 1) \\
\mathbf{c} &= (c, 0)
\end{align*}
\]

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

\[ \vec{u} = (u, u'), \]

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 \( u'(x_0) \).

\[ \begin{align*}
\vec{u} + \vec{v} &= (u + v, u' + v') \\
\vec{u} - \vec{v} &= (u - v, u' - v') \\
\vec{u} \times \vec{v} &= (uv, uv' + u'v) \\
\vec{u} \div \vec{v} &= (u/v, u' - (u/v)v'/v) \\
\vec{x} &= (x, 1) \\
\vec{c} &= (c, 0)
\end{align*} \]

Ref: http://www.math.uu.se/warwick/vt07/FMB/avnm1.pdf
Example of a Rational Function

\[ f(x) = \frac{(x+1)(x-2)}{x+3} \]

\[ f(3) = \frac{2}{3}, \quad f'(3) = ? \]

\[ \overrightarrow{f}(\overrightarrow{x}) = \frac{(\overrightarrow{x} + \overrightarrow{1})(\overrightarrow{x} - \overrightarrow{2})}{(\overrightarrow{x} + \overrightarrow{3})} = \frac{((x, 1) + (1, 0)) \times ((x, 1) - (2, 0))}{((x, 1) + (3, 0))} \]

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

\[ \overrightarrow{f}(3, 1) = \frac{((3, 1) + (1, 0)) \times ((3, 1) - (2, 0))}{((3, 1) + (3, 0))} \]

\[ = \frac{(4, 1) \times (1, 1)}{(6, 1)} \]

\[ = \frac{(4, 5)}{(6, 1)} = \left( \frac{2}{3}, \frac{13}{18} \right) \]
Example of a Rational Function

\[
f(x) = \frac{(x+1)(x-2)}{x+3}
\]

\[
f(3) = \frac{2}{3}, \quad f'(3) = ?
\]

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

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

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

\[
= \frac{(4, 1) \times (1, 1)}{(6, 1)}
\]

\[
= \frac{(4, 5)}{(6, 1)} = \left( \frac{2}{3}, \frac{13}{18} \right)
\]
Example of a Rational Function

\[ f(x) = \frac{(x+1)(x-2)}{x+3} \]

\[ f(3) = \frac{2}{3}, \quad f'(3) = ? \]

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

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

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

\[ = \frac{(4, 1) \times (1, 1)}{(6, 1)} \]

\[ = \frac{(4, 5)}{(6, 1)} = \left( \frac{2}{3}, \frac{13}{18} \right) \]
Derivatives of Element Functions

Chain Rule:

\[(g \circ u)'(x) = u'(x)(g' \circ u)(x)\]

\[\ddot{g}(\dddot{u}) = \ddot{g}((u, u')) = (g(u), u'g'(u))\]

\[
sin \dddot{u} = sin(u, u') = (\sin u, u' \cos u)\]

\[
cos \dddot{u} = cos(u, u') = (\cos u, -u' \sin u)\]

\[
e^\dddot{u} = e^{(u,u')} = (e^u, u'e^u)\]

\[
\vdots\]
Derivatives of Element Functions

Chain Rule:

\[(g \circ u)'(x) = u'(x)(g' \circ u)(x)\]

\[\vec{g}(\vec{u}) = \vec{g}((u, u')) = (g(u), u'g'(u))\]

\[
\begin{align*}
\sin \vec{u} &= \sin(u, u') = (\sin u, u' \cos u) \\
\cos \vec{u} &= \cos(u, u') = (\cos u, -u' \sin u) \\
e^{\vec{u}} &= e^{(u,u')} = (e^u, u'e^u)
\end{align*}
\]

\[\vdots\]
Derivatives of Element Functions

Chain Rule:

\[(g \circ u)'(x) = u'(x)(g' \circ u)(x)\]

\[\vec{g}(\vec{u}) = \vec{g}((u, u')) = (g(u), u'g'(u))\]

\[
\begin{align*}
\sin \vec{u} &= \sin(u, u') = (\sin u, u' \cos u) \\
\cos \vec{u} &= \cos(u, u') = (\cos u, -u' \sin u) \\
e^\vec{u} &= e^{(u, u')} = (e^u, u' e^u) \\
\end{align*}
\]
Example of Sin

```matlab
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 $x=0$.

$$f(x) = (1 + x + e^x) \sin x$$

$$\overrightarrow{f(x)} = (1 + x + e^x) \sin x$$

$$\overrightarrow{f(0, 1)} = \left( (1, 0) + (0, 1) + e^{(0,1)} \right) \sin(0, 1)$$

$$= \left( (1, 1) + (e^0, e^0) \right) (\sin 0, \cos 0)$$

$$= (2, 2)(0, 1) = (0, 2).$$
Example for Element Functions

Evaluate the derivative at $x=0$.

\[
\begin{align*}
  f(x) & = (1 + x + e^x) \sin x \\
  \vec{f}(\vec{x}) & = (1 + \vec{x} + e^{\vec{x}}) \sin \vec{x} \\
  \vec{f}(0, 1) & = \left( (1, 0) + (0, 1) + e^{(0,1)} \right) \sin(0, 1) \\
  & = \left( (1, 1) + (e^0, e^0) \right) (\sin 0, \cos 0) \\
  & = (2, 2)(0, 1) = (0, 2).
\end{align*}
\]
Example for Element Functions

Evaluate the derivative at $x=0$.

$$f(x) = (1 + x + e^x) \sin x$$

$$\overrightarrow{f}(\overrightarrow{x}) = (1 + \overrightarrow{x} + e^\overrightarrow{x})\sin\overrightarrow{x}$$

$$\overrightarrow{f}(0, 1) = \left((1, 0) + (0, 1) + e^{(0,1)}\right) \sin(0, 1)$$

$$= \left((1, 1) + (e^0, e^0)\right) (\sin 0, \cos 0)$$

$$= (2, 2)(0, 1) = (0, 2).$$
High-order Derivatives

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

$$\vec{u} + \vec{v} = (u + v, u' + v', u'' + v'')$$
$$\vec{u} - \vec{v} = (u - v, u' - v', u'' - v'')$$
$$\vec{u} \times \vec{v} = (uv, uv' + u'v, uv'' + 2u'v' + u''v')$$
$$\vec{u} \div \vec{v} = (u/v, u' - (u/v)v'/v, (u'' - 2(u/v)'v' - (u/v)v'')/v)$$

......
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

\[ y_1 = 400x_1(x_1^2 - x_2) + 2(x_1 - 1) \]
\[ y_2 = 200(x_1^2 - x_2) \]

```matlab
f = inline(['[ 400*x(1)*(x(1)^2-x(2)) + 2*(x(1)-1); 200*x(1)*(x(1)^2-x(1)-x(2))]']
```
One Step of Newton Method with INTLab

```matlab
>> 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
```
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

```matlab
>> 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
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.)

```matlab
>> 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
function [g_result1, result1, g_result2, result2] = g_f(g_x, 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

    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
**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.

\[
\begin{align*}
    d(X^2) & : = (XdX + dXX) : \\
    d(det(X)) & = d(det(X^T)) = det(X)(X^{-T}) :^T dX : \\
    d(ln(det(X))) & = (X^{-T}) :^T dX :
\end{align*}
\]

Ref: http://www.ee.ic.ac.uk/hp/staff/dmb/matrix/calculus.html
This problem concerns the calculation of the coefficients of the $m$-degree polynomial $p(x) = p_1 + p_2x + p_2x^2 + \cdots + p_mx^{m-1}$ 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 $Vp = d$, where $V$ is the well-known Vandermonde matrix,

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

<table>
<thead>
<tr>
<th>Method</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
<th>640</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>numjac</td>
<td>19.2</td>
<td>31.6</td>
<td>56.9</td>
<td>106.6</td>
<td>202.4</td>
<td>393.0</td>
<td>823.2</td>
<td>1528.8</td>
</tr>
<tr>
<td>fmad(full)</td>
<td>42.9</td>
<td>40.8</td>
<td>46.9</td>
<td>75.0</td>
<td>167.0</td>
<td>403.1</td>
<td>802.0</td>
<td>1704.2</td>
</tr>
<tr>
<td>fmad(sparse)</td>
<td>44.1</td>
<td>39.0</td>
<td>34.3</td>
<td>32.4</td>
<td>33.6</td>
<td>71.1</td>
<td>127.2</td>
<td>257.2</td>
</tr>
<tr>
<td>ADMAT(full)</td>
<td>44.1</td>
<td>60.4</td>
<td>97.8</td>
<td>175.3</td>
<td>888.9</td>
<td>7220.0</td>
<td>30399.3</td>
<td>128588.1</td>
</tr>
<tr>
<td>ADMAT(sparse)</td>
<td>47.6</td>
<td>63.0</td>
<td>94.3</td>
<td>150.9</td>
<td>265.9</td>
<td>623.4</td>
<td>922.6</td>
<td>1806.9</td>
</tr>
</tbody>
</table>

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

Vandermonde Function (cont.)

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

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

\[ f_1 = 2x_1^2 + \sum_{i=1}^{n} x_i^2, \quad f_i = x_1^2 + x_i^2, \quad i = 2, \ldots, n \}\]

for \( n = 7 \), the Jacobian has sparsity pattern,

\[ Jf(x) = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}, \]

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

<table>
<thead>
<tr>
<th>Method</th>
<th>CPU(Jf)/CPU(f) for Problem Size n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td>numjac(vect)</td>
<td>20.8</td>
</tr>
<tr>
<td>fmad(sparse)</td>
<td>90.6</td>
</tr>
<tr>
<td>ADMAT(sparse)</td>
<td>192.4</td>
</tr>
<tr>
<td>ADMIT</td>
<td>285.2</td>
</tr>
</tbody>
</table>

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

## Arrowhead Function (cont.)

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

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.
function y = bdqrtc(x)
% http://www.sor.princeton.edu/~rvdb/ampl/n1models/cute/bdqrtc.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”,
% 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.)

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

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 **Automatic Differentiation** tools **in** ANSI C/C++.
- ADIC was introduced in 1966.
- Last updated: June 10, 2005.
- ADIC is using forward method.
- Supported Platforms: Unix/Linux.
- Selected Application: NEOS
- Related Research Group: Argonne National Laboratory, USA
ADICProcess
#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;
}
#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("%e\n",DERIV_val(data.r));
   for (i=0; i<n; i++){
      printf("%e\n",grad[i]);
   }
   ad_AD_Final();
   return 0;
}
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:
```c
    data[i++] *= scale;
```

### Canonicalized Code:
```c
    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:
```c
data[i++] *= scale;
```
Canonicalized Code:
```c
data[i] *= scale;
i++;
```

Original Code:
```c
(*f(x)) /= y;
```
Canonicalized Code:
```c
t1 = f(x);
(*t1) = (*t1) / y;
```

Original Code:
```c
unsigned long get_information (int key);
double x, y;
int key;
y = x * (double) get_information (key);
```
Canonicalized Code:
```c
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:
```c
data[i++] *= scale;
```
Canonicalized Code:
```c
data[i] *= scale;
i++;
```

Original Code:
```c
(*f(x)) /= y;
```
Canonicalized Code:
```c
t1 = f(x);
(*t1) = (*t1) / y;
```

Original Code:
```c
unsigned long get_information (int key);
double x,y;
int key;

y = x * (double) get_information (key);
```
Canonicalized Code:
```c
unsigned long get_information (int key);
double tmp,x,y;
int key;

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

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


G. F. Corliss, *Automatic Differentiation*.


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


Thanks!

Questions?