Derivatives Needed!!! — Continued

So far, we looked at using finite difference methods for computing good numerical estimates for “missing” derivatives.

The finite difference approach comes at the price of —

(i) introducing some numerical error in the derivative expressions

(ii) a need for extra evaluations of the function (objective) and/or the gradient.

Finite difference approximations work very well, but if the analytical expressions for the gradient and Hessian can be provided it is the way to go!

Next, we look at Automatic Differentiation Techniques.

“Automatic differentiation (AD) is a technique for augmenting computer programs with derivative computations. It exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations such as additions or elementary functions such as \( \exp() \). By applying the chain rule of derivative calculus repeatedly to these operations, derivatives of arbitrary order can be computed automatically, and accurate to working precision.”


We will here take a very brief look at AD. Even more references and pointers can be found at http://www.AutoDiff.org/, including tools for MATLAB and python.
Micro-History of Automatic Differentiation

- First publications: early 1950s
- 1st International Conference on AD: USA, 1991
- IMA Special Workshop, Minneapolis, USA, 1997
- First European AD Workshop: France, 2005
- AD2016 - 7th International Conference on Algorithmic Differentiation September 2016, UK

The Chain Rule — Forgotten(?) Calculus

The **Chain Rule** in its full vector glory takes the form

$$\nabla_{\bar{x}} h(\bar{y}(\bar{x})) = \sum_{i=1}^{m} \frac{\partial h}{\partial y_i} \nabla y_i(\bar{x}).$$

Automatic differentiation is essentially applying the chain rule **at the code level**. — There are two **modes** of AD, the **forward** and the **reverse** mode. We follow the example from Nocedal-Wright.

Example Problem

We consider a function $f : \mathbb{R}^3 \to \mathbb{R}$

$$f(\bar{x}) = (x_1 x_2 \sin(x_3) + e^{x_1 x_2})/x_3$$

This function can be evaluated in several different ways.

The computational environment will apply certain restrictions (aiming for efficiency), e.g. “the multiplication of $x_1 x_2$ must take place before the exponentiation $e^{x_1 x_2}$.” We illustrate one possibility with a (partially ordered) **computational graph** and several **intermediate variables**.

Language

Some **graph theoretical language**:

- node#i is the **parent** of node#j (and hence node#j the **child** of node#i) if there is a directed arc from $i$ to $j$.

- A node can be evaluated when all its parents are known, so the computation flows left-to-right (see next slide). This is known as a **forward sweep**.
In forward mode, a **directional derivative**, in the direction \( \overline{p} \in \mathbb{R}^n \), of each intermediate value \( x_i \) is evaluated and carried forward simultaneously with the evaluation of \( x_i \) itself.

**Notation**

The directional derivative for \( \overline{p} \in \mathbb{R}^n \) associated with \( x_i \)

\[
D_{\overline{p}}x_i \overset{\text{def}}{=} [\nabla x_i]^T \overline{p} = \sum_{j=1}^{n} \frac{\partial x_j}{\partial x_i} p_j, \quad \forall i
\]

In our example \( n = 3, i = 1, 2, \ldots, 9 \), and our goal is to evaluate \( D_{\overline{p}}x_9 \equiv \nabla f(\overline{x})^T \overline{p} \).

**Note:** \( D_{\overline{p}}x_1 = p_1 \), \( D_{\overline{p}}x_2 = p_2 \), and \( D_{\overline{p}}x_3 = p_3 \). \( \overline{p} \) is known as a **seed vector**.

As soon as the value of \( x_i \) is known at a node, we can find \( D_{\overline{p}}x_i \) using the chain rule. *E.g.* when \( x_7 = x_4 \times x_5 \) is known, we have

\[
\nabla x_7 = \frac{\partial x_7}{\partial x_4} \nabla x_4 + \frac{\partial x_7}{\partial x_5} \nabla x_5 = x_5 \nabla x_4 + x_4 \nabla x_5
\]

Hence, the directional derivative \( D_{\overline{p}}x_7 = x_5 D_{\overline{p}}x_4 + x_4 D_{\overline{p}}x_5 \) can be evaluated.

The accumulation of the directional derivative follows the forward sweep, and at the end we have \( D_{\overline{p}}x_9 = D_{\overline{p}}f = \nabla f(\overline{x})^T \overline{p} \).

The key to **practical implementation** of forward-mode automatic differentiation is the concurrent evaluation of \( x_i \) and \( D_{\overline{p}}x_i \).

To obtain the complete gradient vector, the procedure is carried out simultaneously for \( n \) seed vectors \( \overline{p} = \{ \overline{e}_1, \overline{e}_2, \ldots, \overline{e}_n \} \).

The computational cost can be significant, for instance a single division operation \( w/y \) induces approximately \( 2n \) multiplications and \( n \) additions in the gradient calculation.
An exact bound on the increase in computation (especially for a complicated expression) is hard to obtain [Worst-Case Bounds are not very useful here], since we have to take into account the cost of storing and retrieving the extra quantities $D_{0i}x_i$.

AD/FM can be implemented in terms of a pre-compiler which transforms function evaluation code into augmented code that evaluates derivatives as well. Alternatively operator-overloading (in e.g. C++) can be used to transparently extend data structures and operations to perform the necessary bookkeeping and computations.

In reverse mode AD, the function value $f$ is first computed in a **forward sweep**, then in a second **reverse sweep** the derivatives of $f$ with respect to each variable $x_i$ (independent and intermediate) are recovered.

We associate an adjoint variable $\bar{x}_i$ with each node in the computational graph. In these adjoint variables we accumulate the partial derivative information $\frac{\partial f}{\partial x_j}$ during the reverse sweep. They are initialized $\bar{x}_1 = \bar{x}_2 = \cdots = \bar{x}_{n-1} = 0$, and $\bar{x}_n = 1$.

The reverse sweep is built on the chain rule:

$$\bar{x}_i = \frac{\partial f}{\partial x_i} = \sum_{j \text{ child of } i} \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial x_i}$$

When a term in the sum becomes known, we accumulate it in $\bar{x}_i$:

$$\bar{x}_i + = \frac{\partial f}{\partial x_i} \frac{\partial x_i}{\partial x_j} \frac{\partial x_j}{\partial x_i}$$

Once $\bar{x}_i$ has received contributions from all its children, it is finalized and can communicate its value to its parent(s).

During the reverse sweep we work directly with **numerical values**, not with formulas or computer code describing the variables $x_i$ and/or the partial derivatives $\frac{\partial f}{\partial x_i}$.

Let’s work through the example...
The reverse sweep

We have (

Next, we use #6 and #7, to finalize #4 and #5. (x₆ = eᵣ₄, x₇ = x₄ * x₅).

We now have the final pieces to complete the reverse sweep. (x₄ = x₁ * x₂, x₅ = sin(x₃)).

We now update the parents of #8 — #6 and #7, finalizing both. (x₉ = x₆ + x₇).

### The reverse sweep

\[ x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 0, x_7 = 0, x_8 = 0, x_9 = 1 \]

Node #9 is the child of nodes #3 and #8, we update the associated adjoint variables, finalizing node #8. (x₈ = x₈ / x₃)

\[
\begin{align*}
x_3 &\;= \; \frac{2 + e^2}{\pi} / (\pi/2)^2 = \frac{-8 - 4e^2}{\pi^2} \\
x_8 &\;= \; \frac{\partial f}{\partial x_9} / \partial x_3 = \frac{1}{\pi} / 2 = \frac{2}{\pi}
\end{align*}
\]

Next, we use #6 and #7, to finalize #4 and #5. (x₆ = eᵣ₄, x₇ = x₄ * x₅).

\[
\begin{align*}
x_4 &\;= \; \frac{\partial f}{\partial x_6} + \frac{\partial f}{\partial x_7} \cdot x_7 \cdot 1 = \frac{2e^2 + 2}{\pi} \\
x_5 &\;= \; \frac{\partial f}{\partial x_7} \cdot 2 = \frac{4}{\pi}
\end{align*}
\]

We now have the final pieces to complete the reverse sweep. (x₄ = x₁ * x₂, x₅ = sin(x₃)).

\[
\begin{align*}
x_3 &\;= \; \frac{\partial f}{\partial x_5} / \partial x_3 = \frac{x_5 \cdot (-\cos(\pi/2))}{0} \\
x_2 &\;= \; \frac{\partial f}{\partial x_4} / \partial x_2 = \frac{2e^2 + 2}{\pi} \\
x_1 &\;= \; \frac{\partial f}{\partial x_4} / \partial x_1 = \frac{x_4 \cdot 2}{\pi} = \frac{4e^2 + 4}{\pi}
\end{align*}
\]

We have \( \nabla f(\bar{x}) \bar{x} = [1, 2, \pi/2] \) = [x₁, x₂, x₃].
Reverse Mode Automatic Differentiation: Comments

**Main appeal of AD/RM:** low computational complexity for scalar functions. Extra arithmetic is at most 4–5 times the “pure” function evaluation.

**Main drawback of AD/RM:** the entire computational graph must be stored for the reverse sweep. Implementation and access patterns are quite straightforward, but storage can be a problem:
- If each node can be stored in 20 bytes, then a function that requires one second of evaluation on a 100 megaflop computer may generate a graph which requires 2 Gb of storage!
- The storage requirement can be reduced at the cost of extra arithmetics by partial forward and reverse sweeps; re-evaluating portions instead of storing the whole graph. This is known as checkpointing.

Extending AD to Vector Functions

The idea of automatic differentiation is quite easily extended to vector valued functions,

\[ F : \mathbb{R}^n \rightarrow \mathbb{R}^m, \]

so that the Jacobian matrix of first derivatives

\[ J(\bar{x}) = \begin{bmatrix} \frac{\partial F_j}{\partial x_i} \end{bmatrix}_{j=1,2,\ldots,m}^{i=1,2,\ldots,n} \]

can be formed using automatic differentiation.

All we need is additional bookkeeping for the \( m \) components of \( F \).

Extending AD to the Hessian (2nd Derivatives)

The technique can also be adapted to evaluate the Hessian. In **forward mode** we need 2 seed vectors \( \bar{p} \) and \( \bar{q} \), and we define

\[ D_{\bar{p}\bar{q}}x_i = \bar{p}^T [\nabla^2 x_i] \bar{q} \]

these 2nd derivative quantities are propagated (using the chain rule) just like the 1st derivatives were propagated in our previous example.

At the end of the sweep the terminal node contains the value of

\[ D_{\bar{p}\bar{q}}x_n = \bar{p}^T [\nabla^2 x_n] \bar{q} = \bar{p}^T [\nabla^2 f(\bar{x})] \bar{q} \]

Each pair \((\bar{p}, \bar{q}) = (\bar{e}_i, \bar{e}_j)\) gives the \( H_{ij} \) entry of the Hessian matrix.

Forward-Mode Hessian

Due to symmetry we only need to compute the \( j \leq i \) entries \( H_{ij} \), and it is quite clear how to exploit sparsity.

The increase in the number of arithmetic computations compared with evaluation of \( f \) alone is a multiplicative factor

\[ \sim [1 + n + N_x(\nabla^2 f)] \]

where \( N_x(\nabla^2 f) \) is the number of entries of the Hessian we decide to compute.

In Newton-CG algorithms we only need the effect of the Hessian-vector product \( \nabla^2 f(\bar{x})\bar{r} \), in this case we simply fix the second seed vector \( \bar{q} \equiv \bar{r} \) and compute the remaining \( n \) entries

\[ \bar{e}_j^T [\nabla^2 f(\bar{x})] \bar{r} = [\nabla^2 f(\bar{x})\bar{r}]_j, \quad j = 1, 2, \ldots, n \]

using the forward sweep.
Reverse-Mode Hessian

It is also possible to implement evaluation of the Hessian, \( \nabla^2 f(\tilde{x}) \), or the Hessian-vector product, \( \nabla^2 f(\tilde{x}) \tilde{r} \), in reverse mode.

In the latter case, first both \( f(\tilde{x}) \) and \( \nabla f(\tilde{x})^T \tilde{r} \) are propagated during the forward sweep and the values accumulated in \( (x_i, D_i \tilde{x}_i) \).

Then we apply the reverse sweep to the computed \( \nabla f(\tilde{x})^T \tilde{r} \). At the completion of the reverse sweep we have

\[
\frac{\partial}{\partial x_i} \left[ \nabla f(\tilde{x})^T \tilde{r} \right] = \left[ \nabla^2 f(\tilde{x}) \tilde{r} \right]_{i=1,2,\ldots,n}
\]

in the nodes corresponding to the independent variables.

The increase in work over evaluation of \( f \) alone is a multiplicative factor not greater than

\[
\sim 12 \mathcal{N}_c(\nabla^2 f(\tilde{x}))
\]

where \( \mathcal{N}_c(\nabla^2 f(\tilde{x})) \) = \# of (right-)seed vectors used in computation.

Limitation of Automatic Differentiation

1. \( f(x) \) depends on the numerical solution of a PDE

In this case \( f(\tilde{x}) \) may contain truncation errors due to the scheme used to solve the PDE. Even though the truncation errors are small \( |\tau(\tilde{x})| < \epsilon \), we cannot control the derivative (gradient) \( \tau'(\tilde{x}) \), hence errors in AD-computed \( f'(\tilde{x}) \) can potentially be large.

2. Perverse code

Due to branching (if-statements, etc.) a function evaluation may be equivalent to the following valid but nasty piece of code:

```plaintext
if (x == 1.0) then {
    f = 0.0;
} else {
    f = x - 1.0;
}
```

Automatic differentiation would most likely give us \( f'(1) = 0 \), which does not seem like a Good Idea™.

Bottom Line

- Automatic differentiation provide a set of powerful tools/ideas.
- AD can enhance optimization algorithms, and can be applied successfully in many applications involving highly complex functions.
- AD facilitates interpretations of the computed optimal solutions, allowing the user to extract more information from the result.
- AD does not absolve the user altogether from the responsibility of thinking about derivative calculations, and correctly interpreting and validating the results.