Algorithmic Differentiation III

First Adjoint Derivatives of Multivariate Vector Functions

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen University

Contents

Outline

Objective and Learning Outcomes

Objective

- Introduction to first-order adjoint algorithmic differentiation of multivariate vector functions

Learning Outcomes

- You will understand
  - first-order adjoints
  - link with first-order tangents through differential invariant.
- You will be able to
  - use dco/c++ to compute first-order adjoints and Jacobians
  - validate adjoints against tangents and corresponding finite difference approximations.

Outline

Recall

Tangents

Consider a numerical program implementing

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

where the superscript $*$ marks passive arguments / results of F We are interested in the first [partial] directional derivative of $\mathbf{y}$ wrt. $\mathbf{x}$ at a given point $(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}^*)$. An approximation can be computed by finite differences.

First-order tangent mode algorithmic differentiation (AD) accumulates Jacobian $\times$ vector products (first [partial] directional derivatives) with machine accuracy as

$$\mathbf{y}^{(1)} := \mathbf{y}^{(1)} + F^{(1)}(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}^*, \mathbf{x}^{(1)}) \equiv \mathbf{y}^{(1)} + \frac{\partial F}{\partial \mathbf{x}}(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}^*) \cdot \mathbf{x}^{(1)}$$

Notation: First-order tangents are marked with superscripts $.^{(1)}$.

Recall

Tangent [Forward] Mode AD

For a given DAG $G = G(\mathbf{x}, \mathbf{x}^*)$ or $F(\mathbf{x}, \mathbf{x}^*)$ the DAG $\times$ vector product $F(\mathbf{x}, \mathbf{x}^*) \cdot \mathbf{x}^{(1)}$ is evaluated as

$$v_i^{(1)} := v_i^{(1)} + \frac{d\varphi_i(v_k)_{k \prec i}}{dv_j} \cdot v_j^{(1)} \tag{1}$$

for $i = n + n^*, \ldots, n + q - 1$, $j \prec i$, and where $\mathbf{x}^{*(1)} = 0$. Both $\mathbf{y}^{(1)}$ and $\mathbf{y}^{*(1)}$ are computed. The elemental partial derivatives are used in the order of their computation by the linearized single assignment code allowing for each element function to be augmented locally by its tangent.

Eqn.(1) lends itself to implementation by operator and function overloading (e.g, in C++). The entire arithmetic can be overloaded for a custom data type $(v, v^{(1)})$ comprising both value and tangent.

Outline

Adjoints

Vector $\times$ Jacobian Products

First-order adjoints of

$$F : R^n \times R^{n^*} \rightarrow R^m \times R^{m^*} : \quad (\mathbf{y}, \mathbf{y}^*) = F(\mathbf{x}, \mathbf{x}^*) \ ,$$

can be evaluated in adjoint [reverse] mode AD with machine accuracy as vector $\times$ Jacobian products

$$\mathbf{x}_{(1)} := \mathbf{x}_{(1)} + F_{(1)}(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}^*, \mathbf{y}_{(1)}) \equiv \mathbf{x}_{(1)} + \mathbf{y}_{(1)} \cdot \frac{\partial F}{\partial \mathbf{x}}(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}^*)$$

for $\mathbf{y}_{(1)} \in R^{1\times m}$ and $\mathbf{x}_{(1)} \in R^{1\times n}$.

Notation: First-order adjoints are marked with subscripts $\cdot_{(1)}$.

Approximation by finite differences requires accumulation of the entire Jacobian. It does not yield the same computational complexity as adjoint mode AD.

Adjoints

Cost of [Partial] Jacobian Accumulation

The partial Jacobian $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ can be accumulated at $(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}^*)$ by m evaluations[1] of $F_{(1)}$ with $\mathbf{y}_{(1)}$ ranging over the Cartesian basis vectors in $\boldsymbol{R}^m$. Hence,

$$\text{Cost}\left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right) = O(m) \cdot \text{Cost}(F_{(1)}) = O(m) \cdot \text{Cost}(F) \; .$$

The second identity follows immediately from the fact that adjoints are implemented as vector $\times$ DAG products; see below.

If $n^* = m^* = 0$ then $\mathbf{y} = F(\mathbf{x})$ and

$$\mathbf{x}_{(1)} := F_{(1)}(\tilde{\mathbf{x}}, \mathbf{y}_{(1)}) \equiv \mathbf{y}_{(1)} \cdot \frac{d\mathbf{y}}{d\mathbf{x}}(\tilde{\mathbf{x}}) \; .$$

[1] This number can be decreased by detecting and exploiting potential sparsity.

Adjoints

Adjoint [Reverse] Mode AD

The vector $\times$ DAG product $\mathbf{y}_{(1)} \cdot G(\mathbf{x}, \mathbf{x}^*)$ is evaluated as

$$v_{i_{(1)}} = v_{i_{(1)}} + v_{j_{(1)}} \cdot \frac{d\varphi_j(v_k)_{k \prec j}}{dv_i} \tag{2}$$

for $j = n + n^* + q - 1, \ldots, n + n^*$, $i \prec j$, and where $\mathbf{y^*}_{(1)} = 0$. Both $\mathbf{x}_{(1)}$ and $\mathbf{x^*}_{(1)}$ are computed.[2]

Each element function is matched by its adjoint. The elemental partial derivatives are used in reverse wrt. to the order of their computation by the linearized single assignment code.

Implementation of reverse mode AD by overloading records the DAG (also: *tape*) followed by the evaluation of Eqn.(2) to compute $\mathbf{y}_{(1)} \cdot G(\mathbf{x}, \mathbf{x}^*)$.

[2] Activity analysis allows for passive operations to be avoided.

Adjoint Mode AD

Vector $\times$ DAG product

The vector $\times$ Jacobian product

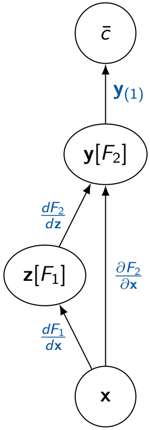
$$\mathbf{x}_{(1)} = \mathbf{x}_{(1)} \cdot \frac{dF}{d\mathbf{x}}(\tilde{\mathbf{x}})$$

of $\mathbf{y} = F(\mathbf{x})$ at $\tilde{\mathbf{x}}$ can be represented as the derivative of an auxiliary variable $\bar{c} = \bar{c}(\mathbf{y}(\mathbf{x}))$ wrt. $\mathbf{x}$ at $\tilde{\mathbf{x}}$ such that

$$\frac{d\bar{c}}{d\mathbf{y}} = \mathbf{y}_{(1)} \,.$$

The chain rule yields

$$\mathbf{x}_{(1)} \equiv \frac{d\bar{c}}{d\mathbf{x}}(\tilde{\mathbf{x}}) = \frac{d\bar{c}}{d\mathbf{y}} \cdot \frac{dF}{d\mathbf{x}}(\tilde{\mathbf{x}}) = \mathbf{y}_{(1)} \cdot \frac{dF}{d\mathbf{x}}(\tilde{\mathbf{x}}) \,.$$



$\bar{c}$
$\mathbf{y}_{(1)}$
$\mathbf{y}[F_2]$
$\frac{dF_2}{d\mathbf{z}}$
$\mathbf{z}[F_1]$
$\frac{\partial F_2}{\partial \mathbf{x}}$
$\frac{dF_1}{d\mathbf{x}}$
$\mathbf{x}$

Think of $\bar{c} = \mathbf{y}_{(1)} \cdot \mathbf{y}$.

Adjoints

Example: $y = f(\mathbf{x}) = e^{\sin\left(x_0^2+x_1^2\right)}$

Seeding a Cartesian basis vector (here scalar) $v_{6(1)} = y_{(1)} = 1$ and running the primal single assignment code

$$v_2 = v_0^2;\ \ v_3 = v_1^2;\ \ v_4 = v_2 + v_3;\ \ v_5 = \sin(v_4);\ \ v_6 = e^{v_5}$$

followed by the adjoint single assignment code

$$\begin{aligned}
v_{5(1)} &= v_{5(1)} + v_6 \cdot v_{6(1)} \\
v_{4(1)} &= v_{4(1)} + \cos(v_4) \cdot v_{5(1)} \\
v_{3(1)} &= v_{3(1)} + v_{4(1)} \\
v_{2(1)} &= v_{2(1)} + v_{4(1)} \\
v_{1(1)} &= v_{1(1)} + 2 \cdot v_1 \cdot v_{3(1)} \\
v_{0(1)} &= v_{0(1)} + 2 \cdot v_0 \cdot v_{2(1)}
\end{aligned}$$

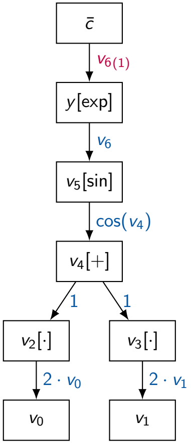
yields the corresponding row of the Jacobian (here the entire gradient) in $\mathbf{x}_{(1)} = \left(x_{0(1)}, x_{1(1)}\right) = \left(v_{0(1)}, v_{1(1)}\right)$.

Adjoints

Example: $y = f(\mathbf{x}) = e^{\sin\left(x_0^2 + x_1^2\right)}$

$$
\begin{aligned}
v_2 &= v_0^2 \\
v_3 &= v_1^2 \\
v_4 &= v_2 + v_3 \\
v_5 &= \sin(v_4) \\
v_6 &= e^{v_5}
\end{aligned}
$$

$$
\begin{aligned}
v_{5(1)} &= v_{5(1)} + v_6 \cdot v_{6(1)} \\
v_{4(1)} &= v_{4(1)} + \cos(v_4) \cdot v_{5(1)} \\
v_{3(1)} &= v_{3(1)} + v_{4(1)} \\
v_{2(1)} &= v_{2(1)} + v_{4(1)} \\
v_{1(1)} &= v_{1(1)} + 2 \cdot v_1 \cdot v_{3(1)} \\
v_{0(1)} &= v_{0(1)} + 2 \cdot v_0 \cdot v_{2(1)}
\end{aligned}
$$


$$\begin{tikzcd}
& \boxed{\bar{c}} \arrow[d, "v_{6(1)}"] & \\
& \boxed{y[\exp]} \arrow[d, "v_6"] & \\
& \boxed{v_5[\sin]} \arrow[d, "\cos(v_4)"] & \\
& \boxed{v_4[+]} \arrow[dl, "1"] \arrow[dr, "1"] & \\
\boxed{v_2[\cdot]} \arrow[d, "2 \cdot v_0"] & & \boxed{v_3[\cdot]} \arrow[d, "2 \cdot v_1"] \\
\boxed{v_0} & & \boxed{v_1}
\end{tikzcd}$$

Adjoints

Implementation with dco/c++ ($f : R^n \rightarrow R$)

```
#include "f.hpp"
#include "Eigen/Dense"
#include "dco.hpp"

template<typename T, int N>
void f_a(const Eigen::Matrix<T,N,1>& x_v, T& y_v, Eigen::Matrix<T,N,1>& dydx) {
  using DCO_M=typename dco::ga1s<T>;
  using DCO_T=typename DCO_M::type;
  auto n=x_v.size();
  Eigen::Matrix<DCO_T,N,1> x(n); DCO_T y=0;
  DCO_M::global_tape=DCO_M::tape_t::create();
  for (auto i=0;i<n;i++) {
    x(i)=x_v(i);
    DCO_M::global_tape->register_variable(x(i));
  }
  f(x,y);
  y_v=dco::value(y);
  dco::derivative(y)=1;
  DCO_M::global_tape->interpret_adjoint();
  for (auto i=0;i<n;i++) dydx(i)=dco::derivative(x(i));
  DCO_M::tape_t::remove(DCO_M::global_tape);
}
```

Adjoints

Implementation with dco/c++ ($f : \mathbb{R}^n \rightarrow \mathbb{R}^m$)

```cpp
template<typename T, int N, int M>
void f_a(const Eigen::Matrix<T,N,1>& x_v,
   Eigen::Matrix<T,M,1>& y_v, Eigen::Matrix<T,M,N>& dydx) {
  using DCO_M=typename dco::ga1s<T>; using DCO_T=typename DCO_M::type;
  auto n=x_v.size(), m=y_v.size();
  Eigen::Matrix<DCO_T,N,1> x(n); Eigen::Matrix<DCO_T,M,1> y(m);
  for (auto j=0;j<m;j++) y(j)=y_v(j);
  DCO_M::global_tape=DCO_M::tape_t::create();
  for (auto i=0;i<n;i++) {
    x(i)=x_v(i);
    DCO_M::global_tape->register_variable(x(i));
  }
  f(x,y);
  for (auto j=0;j<m;j++) y_v(j)=dco::value(y(j));
  for (auto j=0;j<m;j++) {
    dco::derivative(y(j))=1;
    DCO_M::global_tape->interpret_adjoint();
    for (auto i=0;i<n;i++) dydx(j,i)=dco::derivative(x(i));
    DCO_M::global_tape->zero_adjoints();
  }
  DCO_M::tape_t::remove(DCO_M::global_tape);
}
```

Outline

Differential Invariant

Validation of Adjoints

$$\mathbf{x}_{(1)} \cdot \mathbf{x}^{(1)} = \mathbf{y}_{(1)} \cdot F'(\mathbf{x}) \cdot \mathbf{x}^{(1)} = \mathbf{y}_{(1)} \cdot \mathbf{y}^{(1)}$$

Note use of differential invariant for

- validation of adjoints against tangents
- validation of adjoints against approximate tangents
- debugging of derivative code.

Outline

Summary and Next Steps

Summary

- Adjoint mode algorithmic differentiation (with dco/c++)
- Differential invariant

for multivariate vector functions

Next Steps

- Practice adjoint single assignment code.
- Run (own) examples and validate results.
- Continue the course to find out more ...