Algorithmic Differentiation II

First Directional 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 approximation of first directional derivatives and to first-order tangent algorithmic differentiation of multivariate vector functions

Learning Outcomes

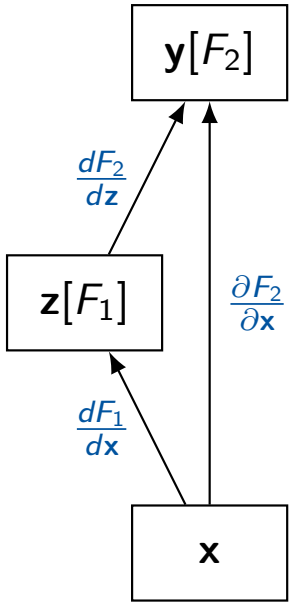
- You will understand
  - first-order finite differences
  - first-order tangents.
- You will be able to
  - approximate first directional derivatives and Jacobians by finite differences
  - use dco/c++ for the computation of first-order tangents and Jacobians.

Outline

Let $\mathbf{y} = F(\mathbf{x}) : \boldsymbol{R}^n \to \boldsymbol{R}^m$ be such that

$$\mathbf{y} = F(\mathbf{x}) = F_2(F_1(\mathbf{x}), \mathbf{x}) = F_2(\mathbf{z}, \mathbf{x})$$

with (continuously) differentiable $F_1 : \boldsymbol{R}^n \to \boldsymbol{R}^p$ and $F_2 : \boldsymbol{R}^p \times \boldsymbol{R}^n \to \boldsymbol{R}^m$.



Then F is continuously differentiable over $\boldsymbol{R}^n$ and

$$\frac{dF}{d\mathbf{x}}(\tilde{\mathbf{x}}) = \frac{dF_2}{d\mathbf{x}}(\tilde{\mathbf{z}}, \tilde{\mathbf{x}}) = \frac{dF_2}{d\mathbf{z}}(\tilde{\mathbf{z}}, \tilde{\mathbf{x}}) \cdot \frac{dF_1}{d\mathbf{x}}(\tilde{\mathbf{x}}) + \frac{\partial F_2}{\partial \mathbf{x}}(\tilde{\mathbf{z}}, \tilde{\mathbf{x}})$$

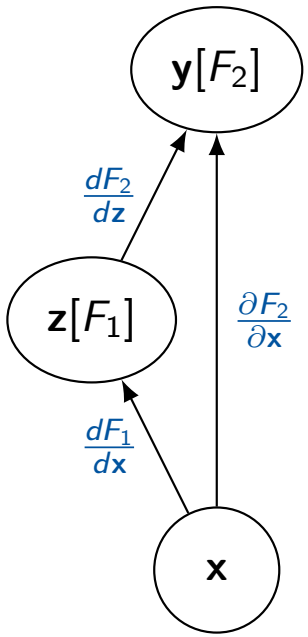
for all $\tilde{\mathbf{x}} \in \boldsymbol{R}^n$ and $\tilde{\mathbf{z}} = F_1(\tilde{\mathbf{x}})$.

Notation: $\frac{\partial F_2}{\partial \mathbf{x}}$ partial derivative; $\frac{dF_2}{d\mathbf{x}}$ total derivative

A composite function $\mathbf{y} = F(\mathbf{x})$ such as

$$\mathbf{z} = F_1(\mathbf{x})$$
$$\mathbf{y} = F_2(\mathbf{z}, \mathbf{x})$$

induces a directed acyclic graph (DAG) $G = (V, E)$ with vertices in V representing variables (e.g, $\mathbf{x}$, $\mathbf{z}$ and $\mathbf{y}$) and with local (partial) derivatives associated with the edges in E.



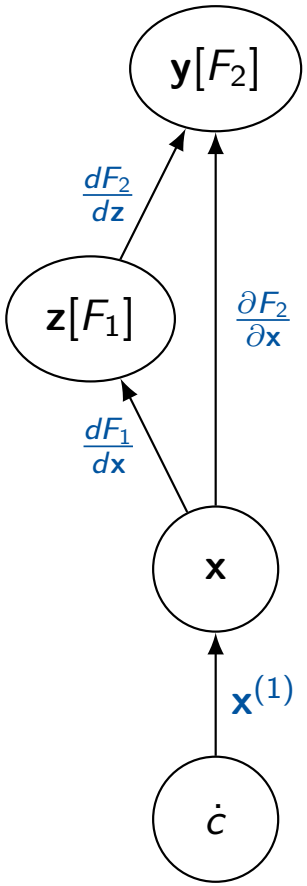
$$F'(\mathbf{x}) \equiv \frac{d\mathbf{y}}{d\mathbf{x}} = \sum_{\text{path} \in \text{DAG}} \prod_{(i,j) \in \text{path}} \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} + \frac{\partial \mathbf{y}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} + \frac{d\mathbf{y}}{d\mathbf{z}} \cdot \frac{d\mathbf{z}}{d\mathbf{x}}$$

The directional derivative (Jacobian $\times$ vector product)

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

of $\mathbf{y} = F(\mathbf{x})$ at $\tilde{\mathbf{x}}$ can be represented as the derivative of $\mathbf{y} = \mathbf{y}(\mathbf{x}(\dot{c}))$ with respect to (wrt.) an auxiliary variable $\dot{c} \in \boldsymbol{R}$ at $\tilde{\mathbf{x}}$ such that

$$\frac{d\mathbf{x}}{d\dot{c}} = \mathbf{x}^{(1)} \ .$$



The chain rule yields

$$\mathbf{y}^{(1)} \equiv \frac{dF}{d\dot{c}} = \frac{dF}{d\mathbf{x}}(\tilde{\mathbf{x}}) \cdot \frac{d\mathbf{x}}{d\dot{c}} = \frac{dF}{d\mathbf{x}}(\tilde{\mathbf{x}}) \cdot \mathbf{x}^{(1)} \ .$$

Directional derivatives are marked with the superscript $^{(1)}$.

Think of $\mathbf{x} = \mathbf{x} + \mathbf{x}^{(1)} \cdot \dot{c}$ at $\mathbf{x} = \tilde{\mathbf{x}}$ and $\dot{c} = 0$.

In scientific computing the multivariate vector functions

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

of interest are implemented as differentiable computer programs.

Such programs decompose into sequences of $q = p + m$ differentiable elemental functions φ_j evaluated as a single assignment code[1]

$$v_j = \varphi_j(v_k)_{k \prec j} \quad \text{for } j = n, \ldots, n + q - 1$$

and where $v_i = x_i$ for $i = 0, \ldots, n-1$, $y_k = v_{n+p+k}$ for $k = 0, \ldots, m-1$ and $k \prec j$ if v_k is an argument of φ_j.

A DAG $G = (V, E)$ is induced. Partial derivatives of the elemental functions wrt. their arguments are associated with the corresponding edges.

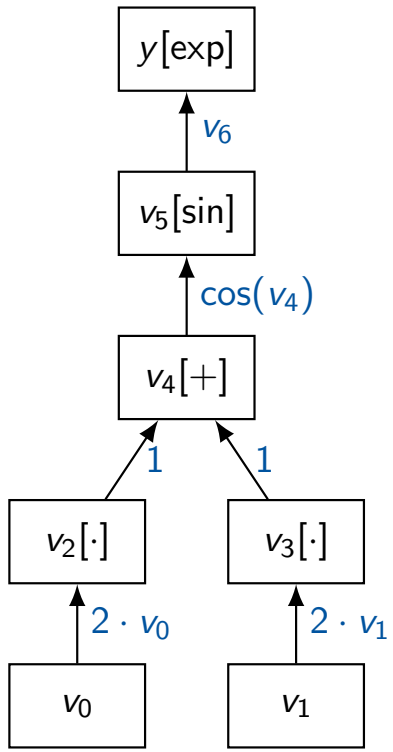
[1] Variables are written once.

Single Assignment Code ⇒ DAG

Example

$$y = f(\mathbf{x}) = e^{\sin\left(\|\mathbf{x}\|_2^2\right)} = e^{\sin\left(\mathbf{x}^T \cdot \mathbf{x}\right)} = e^{\sin\left(\sum_{i=0}^{n-1} x_i^2\right)}, \quad n = 2$$

$$\begin{aligned}
v_0 &= x_0 \\
v_1 &= x_1 \\
v_2 &= v_0^2; & \frac{dv_2}{dv_0} &= 2 \cdot v_0 \\
v_3 &= v_1^2; & \frac{dv_3}{dv_1} &= 2 \cdot v_1 \\
v_4 &= v_2 + v_3; & \frac{dv_4}{dv_2} &= \frac{dv_4}{dv_3} = 1 \\
v_5 &= \sin(v_4); & \frac{dv_5}{dv_4} &= \cos(v_4) \\
v_6 &= e^{v_5}; & \frac{dv_6}{dv_5} &= v_6 \\
y &= v_6
\end{aligned}$$



The DAG $G = G(\tilde{\mathbf{x}})$ of F induces a linear mapping (generalized Jacobian × Vector Product)

$$G : \boldsymbol{R}^n \to \boldsymbol{R}^m : \quad \mathbf{y}^{(1)} = G \cdot \mathbf{x}^{(1)}$$

defined by the chain rule applied to $F(\mathbf{x}(\dot{c}))$ at $\mathbf{x} = \tilde{\mathbf{x}}$ and for

$$\frac{d\mathbf{x}}{d\dot{c}} \equiv \mathbf{x}^{(1)} \in \boldsymbol{R}^n .$$

This DAG × vector product is evaluated as

$$v_i^{(1)} = \sum_{j \prec i} \frac{d\varphi_i(v_k)_{k \prec i}}{dv_j} \cdot v_j^{(1)} \quad \text{for } i = n, \ldots, n+q-1$$

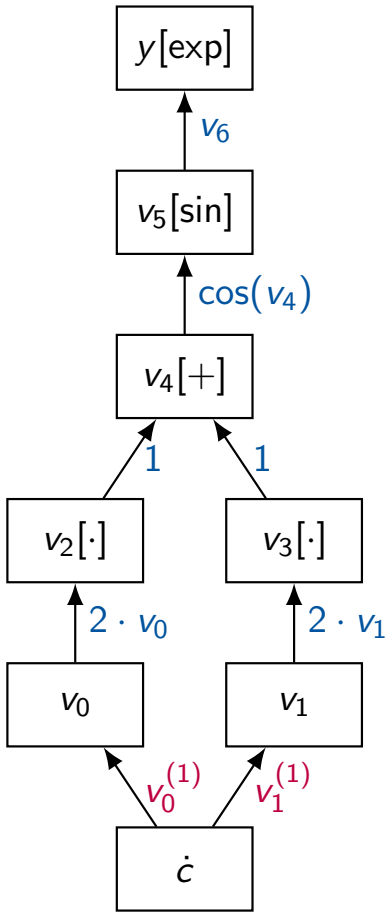
and where $v_i^{(1)} = x_i^{(1)}$ for $i = 0, \ldots, n-1$ and $y_k^{(1)} = v_{n+p+k}^{(1)}$ for $k = 0, \ldots, m-1$.

DAG × Vector Product

Example

$$y = f(\mathbf{x}) = e^{\sin\left(\|\mathbf{x}\|_2^2\right)} = e^{\sin\left(\mathbf{x}^T \cdot \mathbf{x}\right)} = e^{\sin\left(\sum_{i=0}^{n-1} x_i^2\right)}, \quad n = 2$$

$$\begin{aligned}
v_0 &= x_0 & v_0^{(1)} &= x_0^{(1)} \\
v_1 &= x_1 & v_1^{(1)} &= x_1^{(1)} \\
v_2 &= v_0^2 & v_2^{(1)} &= 2 \cdot v_0 \cdot v_0^{(1)} \\
v_3 &= v_1^2 & v_3^{(1)} &= 2 \cdot v_1 \cdot v_1^{(1)} \\
v_4 &= v_2 + v_3 & v_4^{(1)} &= v_2^{(1)} + v_3^{(1)} \\
v_5 &= \sin(v_4) & v_5^{(1)} &= \cos(v_4) \cdot v_4^{(1)} \\
v_6 &= e^{v_5} & v_6^{(1)} &= v_6 \cdot v_5^{(1)} \\
y &= v_6 & y^{(1)} &= v_6^{(1)}
\end{aligned}$$



- $f : \boldsymbol{R} \to \boldsymbol{R}$

$$f(x+\Delta x) = f(x)+f'(x)\cdot\Delta x+\frac{1}{2}\cdot f''(x)\cdot\Delta x^2+\frac{1}{6}\cdot f'''(x)\cdot\Delta x^3+O(|\Delta x|^4)$$

- $f : \boldsymbol{R}^n \to \boldsymbol{R}$

$$f(\mathbf{x}+\Delta\mathbf{x}) = f(\mathbf{x})+f'(\mathbf{x})^T\cdot\Delta\mathbf{x}+\frac{1}{2}\cdot\Delta\mathbf{x}^T\cdot f''(\mathbf{x})\cdot\Delta\mathbf{x}+O(\|\Delta\mathbf{x}\|_2^3)$$

- $F : \boldsymbol{R}^n \to \boldsymbol{R}^n$

$$F(\mathbf{x}+\Delta\mathbf{x}) = F(\mathbf{x})+F'(\mathbf{x})\cdot\Delta\mathbf{x}+O(\|\Delta\mathbf{x}\|_2^2)$$

Higher-order terms are omitted to avoid tensor notation.

Outline

Similar to the scalar case, many numerical methods for nonlinear multivariate vector functions are built on local (at $\tilde{\mathbf{x}}$) replacement of the target function with a linear (affine; in $\Delta\mathbf{x}$) approximation derived from the truncated Taylor series expansion and "hoping" that

$$F(\tilde{\mathbf{x}} + \Delta\mathbf{x}) \approx \bar{F}(\Delta\mathbf{x}) = F(\tilde{\mathbf{x}}) + F'(\tilde{\mathbf{x}}) \cdot \Delta\mathbf{x} \; .$$

The derivative of the linear function $\bar{F}(\Delta\mathbf{x})$ wrt. $\Delta\mathbf{x}$

$$\bar{F}'(\Delta\mathbf{x}) = \bar{F}'(\tilde{\mathbf{x}}) \approx F'(\tilde{\mathbf{x}})$$

can be used to approximate the Jacobian $F'(\tilde{\mathbf{x}})$ of the nonlinear function F.

Again, differentiation of $\bar{F}(\Delta\mathbf{x})$ wrt. $\Delta\mathbf{x}$ amounts to the evaluation of finite differences.

Linearization of scalar problems yield linear equations $a \cdot x = b$. Small changes in $b \in \boldsymbol{R}$ imply small (same order) changes in $x \in \boldsymbol{R}$.

Linearization of multivariate vector functions yields systems of linear equations $A \cdot \mathbf{x} = \mathbf{b}$. Small changes in $\mathbf{b} \in \boldsymbol{R}^n$ can yield large changes in $\mathbf{x} \in \boldsymbol{R}^n$ due to poor conditioning of $A \in \boldsymbol{R}^{n \times n}$. For example,

$$\begin{aligned} x + y &= 2 \\ x + 1.001 \cdot y &= 2 \end{aligned} \quad \Rightarrow \quad \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

while

$$\begin{aligned} x + y &= 2 \\ x + 1.001 \cdot y &= 2.001 \end{aligned} \quad \Rightarrow \quad \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

due to near singularity of A ($\mathrm{cond}(A) \approx 4004$).

The directional derivative of F in direction $\mathbf{x}^{(1)} \in R^n$ (product of the Jacobian of F with $\mathbf{x}^{(1)}$) can be approximated at $\tilde{\mathbf{x}} \in \boldsymbol{R}^n$ by finite difference quotients as

$$F'(\tilde{\mathbf{x}}) \cdot \mathbf{x}^{(1)} \approx_1 \left(\frac{F(\tilde{\mathbf{x}} + \Delta \mathbf{x}_i \cdot \mathbf{x}^{(1)}) - F(\tilde{\mathbf{x}})}{\Delta \mathbf{x}_i} \right)_{i=0}^{n-1} \quad \text{(forward)}$$

$$\approx_1 \left(\frac{F(\tilde{\mathbf{x}}) - F(\tilde{\mathbf{x}} - \Delta \mathbf{x}_i \cdot \mathbf{x}^{(1)})}{\Delta \mathbf{x}_i} \right)_{i=0}^{n-1} \quad \text{(backward)}$$

$$\approx_2 \left(\frac{F(\tilde{\mathbf{x}} + \Delta \mathbf{x}_i \cdot \mathbf{x}^{(1)}) - F(\tilde{\mathbf{x}} - \Delta \mathbf{x}_i \cdot \mathbf{x}^{(1)})}{2 \cdot \Delta \mathbf{x}_i} \right)_{i=0}^{n-1} \quad \text{(central)}$$

where $\Delta \mathbf{x} = \Delta \mathbf{x}(\tilde{\mathbf{x}}) \in \boldsymbol{R}^n$ is a vector of suitable perturbations.

Forward and backward finite differences exhibit first-order accuracy ($\approx_1$) while central finite differences turn out to be second-order accurate ($\approx_2$); see below.

Individual columns of the Jacobian of F can be approximated at $\tilde{\mathbf{x}} \in \boldsymbol{R}^n$ as

$$F'(\tilde{\mathbf{x}}) \approx_1 \left(\frac{F(\tilde{\mathbf{x}} + \Delta \mathbf{x}_i \cdot \mathbf{e}_i) - F(\tilde{\mathbf{x}})}{\Delta \mathbf{x}_i} \right)_{i=0}^{n-1} \quad \text{(forward)}$$

$$\approx_1 \left(\frac{F(\tilde{\mathbf{x}}) - F(\tilde{\mathbf{x}} - \Delta \mathbf{x}_i \cdot \mathbf{e}_i)}{\Delta \mathbf{x}_i} \right)_{i=0}^{n-1} \quad \text{(backward)}$$

$$\approx_2 \left(\frac{F(\tilde{\mathbf{x}} + \Delta \mathbf{x}_i \cdot \mathbf{e}_i) - F(\tilde{\mathbf{x}} - \Delta \mathbf{x}_i \cdot \mathbf{e}_i)}{2 \cdot \Delta \mathbf{x}_i} \right)_{i=0}^{n-1} \quad \text{(central)}$$

where $\mathbf{e}_i$ denotes the i-th Cartesian basis vector in $\boldsymbol{R}^n$.

Forward and backward finite difference quotients follow immediately from the Taylor series expansion

$$F(\mathbf{x} \pm \Delta\mathbf{x}) = F(\mathbf{x}) \pm F'(\mathbf{x}) \cdot \Delta\mathbf{x} + O(\|\Delta\mathbf{x}\|_2^2)$$

truncated after the first-order term. The error decreases quadratically with $0 < \|\Delta\mathbf{x}\|_2 \ll 1$ yielding first-order accuracy.

The central finite difference quotient is obtained by truncation of the difference

$$F(\mathbf{x} + \Delta\mathbf{x}) - F(\mathbf{x} - \Delta\mathbf{x})$$

of both Taylor series expansions after the first-order term. The error turns out to decrease cubically with $0 < \|\Delta\mathbf{x}\|_2 \ll 1$ yielding second-order accuracy.

Finite Differences

Implementation ($f : R^n \to R$)


```
1  #include "f.hpp" // void f(const Eigen::Matrix<T,N,1>&,T&);
2  #include "Eigen/Dense"
3
4  template<typename T, int N>
5  void f_cfd(Eigen::Matrix<T,N,1> x, T& y, Eigen::Matrix<T,N,1>& dydx) {
6    for (auto i=0;i<x.size();i++) {
7      T dx=fabs(x(i))<1 ? sqrt(std::numeric_limits<T>::epsilon())
8          : sqrt(std::numeric_limits<T>::epsilon())*fabs(x(i));
9      T yp,ym;
10     x(i)+=dx; f(x,yp); x(i)-=2*dx; f(x,ym); x(i)+=dx;
11     dydx(i)=(yp-ym)/(2*dx);
12   }
13   f(x,y);
14 }
```

Finding a `dx` for a good approximation of the derivative is often nontrivial. Typically, perturbation of the midpoint of the mantissa

```
sqrt(std::numeric_limits<T>::epsilon()) [*fabs(x(i))]
```

yields a reasonable compromise between accuracy and numerical stability.

Finite Differences

Implementation ($f : R^n \to R^m$)


```
1  #include "f.hpp"// void f(const Eigen::Matrix<T,N,1>&,
2                                  Eigen::Matrix<T,N,1>&);
3  #include "Eigen/Dense"
4
5  template<typename T, int N, int M>
6  void f_cfd(Eigen::Matrix<T,N,1>& x,
7             Eigen::Matrix<T,M,1>& y, Eigen::Matrix<T,M,N>& dydx) {
8    for (auto i=0;i<x.size();i++) {
9      T dx=fabs(x(i))<1 ? sqrt(std::numeric_limits<T>::epsilon())
10        : sqrt(std::numeric_limits<T>::epsilon())*fabs(x(i));
11     Eigen::Matrix<T,M,1> yp(y),ym(y);
12     x(i)+=dx; f(x,yp); x(i)-=2*dx; f(x,ym); x(i)+=dx;
13     for (auto j=0;j<y.size();j++) dydx(j,i)=(yp(j)-ym(j))/(2*dx);
14   }
15   f(x,y);
16 }
```

Outline

Consider a numerical program implementing

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

where the superscript $*$ marks arguments / results of F which are not subject to differentiation.[2] 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 × 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 $.^{(1)}$.

[2] They are referred to as passive as opposed to active variables.

The partial Jacobian $F_{\mathbf{x}} \equiv \frac{\partial F}{\partial \mathbf{x}}$ can be accumulated at $(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}^*)$ by n evaluations of $F_{\mathbf{x}}^{(1)}$ with $\mathbf{x}^{(1)}$ ranging over the Cartesian basis vectors in $\boldsymbol{R}^n$.[3] Hence,

$$\text{Cost}(F_{\mathbf{x}}) = O(n) \cdot \text{Cost}(F_{\mathbf{x}}^{(1)}) = O(n) \cdot \text{Cost}(F) \ .$$

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

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

$$\mathbf{y}^{(1)} := F^{(1)}(\tilde{\mathbf{x}}, \mathbf{x}^{(1)}) \equiv \frac{dF}{d\mathbf{x}}(\tilde{\mathbf{x}}) \ .$$

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

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 $i = n + n^*, \ldots, n + n^* + q - 1$, $j \prec i$, and where $\mathbf{x}^{*(1)} = 0$. Both $\mathbf{y}^{(1)}$ and $\mathbf{y}^{*(1)}$ are computed.[4] 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.

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

Seeding a Cartesian basis vector, e.g, $v_0^{(1)} = x_0^{(1)} = 1$ and $v_1^{(1)} = x_1^{(1)} = 0$, and initializing $v_i^{(1)} = 0$, $i = 2, \ldots, 6$ followed by propagating tangents alongside the primal function values as

$$
\begin{aligned}
v_2 &:= v_0^2; & v_2^{(1)} &:= v_2^{(1)} + 2 \cdot v_0 \cdot v_0^{(1)} \\
v_3 &:= v_1^2; & v_3^{(1)} &:= v_3^{(1)} + 2 \cdot v_1 \cdot v_1^{(1)} \\
v_4 &:= v_2 + v_3; & v_4^{(1)} &:= v_4^{(1)} + v_2^{(1)};\ v_4^{(1)} := v_4^{(1)} + v_3^{(1)} \\
v_5 &:= \sin(v_4); & v_5^{(1)} &:= v_5^{(1)} + \cos(v_4) \cdot v_4^{(1)} \\
v_6 &:= e^{v_5}; & v_6^{(1)} &:= v_6^{(1)} + v_6 \cdot v_5^{(1)}
\end{aligned}
$$

yields the corresponding column of the Jacobian, e.g, the first gradient entry, to be harvested from $y^{(1)} = v_6^{(1)}$.

Tangents

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


```
1  #include "f.hpp"
2  #include "dco.hpp"
3  #include "Eigen/Dense"
4
5  template<typename T, int N=Eigen::Dynamic>
6  void f_t(const Eigen::Matrix<T,N,1>& x_v, T& y_v, Eigen::Matrix<T,N,1>& dydx) {
7    typedef typename dco::gt1s<T>::type DCO_T;
8    size_t n=x_v.size();
9    Eigen::Matrix<DCO_T,N,1> x(n); DCO_T y=0;
10   for (size_t i=0;i<n;i++) x(i)=x_v(i);
11   for (size_t i=0;i<n;i++) {
12     dco::derivative(x(i))=1;
13     f(x,y);
14     dydx(i)=dco::derivative(y);
15     dco::derivative(x(i))=0;
16   }
17   y_v=dco::value(y);
18 }
```

Tangents

Implementation with dco/c++ ($F : R^n \to R^m$)


```
1  #include "f.hpp"
2  #include "dco.hpp"
3  #include "Eigen/Dense"
4
5  template<typename T, int N, int M>
6  void f_t(const Eigen::Matrix<T,N,1>& x_v,
7           Eigen::Matrix<T,M,1>& y_v, Eigen::Matrix<T,M,N>& dydx) {
8    using DCO_T=typename dco::gt1s<T>::type;
9    auto n=x_v.size(), m=y_v.size();
10   Eigen::Matrix<DCO_T,N,1> x(n);
11   Eigen::Matrix<DCO_T,M,1> y(m);
12   for (auto i=0;i<n;i++) x(i)=x_v(i);
13   for (auto i=0;i<n;i++) {
14     for (auto j=0;j<m;j++) y(j)=y_v(j);
15     dco::derivative(x(i))=1;
16     f(x,y);
17     for (auto j=0;j<m;j++) dydx(j,i)=dco::derivative(y(j));
18     dco::derivative(x(i))=0;
19   }
20   for (auto j=0;j<m;j++) y_v(j)=dco::value(y(j));
21 }
```

Outline

Summary

- Finite differences
- Tangent mode algorithmic differentiation (with dco/c++)

for multivariate vector functions

Next Steps

- Practice tangent single assignment code
- Run (own) sample code and compare results.
- Continue the course to find out more ...