Introduction to Algorithmic Differentiation

AD by Hand (Tangent Code)

Uwe Naumann

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

RWTH Aachen

Contents

Outline

Recall

- Sigmoidal Smoothing
- Newton's Method

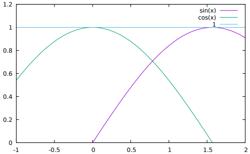
Tangent Code Generation Rules

Examples

- Tangent Straight-Line Code
- Tangent Intraprocedural Code
- Tangent Interprocedural Code

Let $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ be defined as

$$f(x,p) = \begin{cases} f_1(x) & x < p \\ f_2(x) & x \geq p \ . \end{cases}$$



with differentiable univariate scalar f_1 and f_2.

Depending on the choice of f_1 and f_2 the function f can be nondifferentiable or even discontinuous at $x = p$.

Examples:

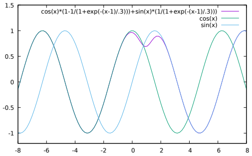
- $f_1 = \cos,\ f_2 = \sin \Rightarrow$ discontinuous at $x = p = 1$
- $f_1 = \cos,\ f_2 = \sin \Rightarrow$ nondifferentiable at $x = p = \frac{\pi}{4}$
- $f_1 = 1,\ f_2 = \cos \Rightarrow$ differentiable at $x = p = 0$

Sigmoidal smoothing replaces f with $\tilde{f} : \boldsymbol{R}^3 \rightarrow \boldsymbol{R}$ defined as

$$\tilde{f}(x, p, w) = (1 - \sigma(x, p, w)) \cdot f_1(x) + \sigma(x, p, w) \cdot f_2(x) \; ,$$

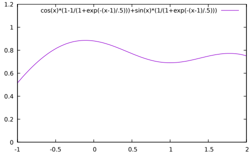
where

$$\sigma(x, p, w) = \frac{1}{1 + e^{-\frac{x-p}{w}}} \; .$$

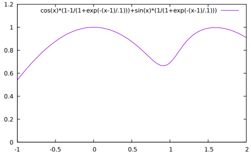


$w = 0.3$

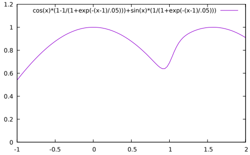
Example: $f_1 = \cos$, $f_2 = \sin$ at $x = p = 1$



$w = 0.5$



$w = 0.1$



$w = 0.05$

nondifferentiable ⇒ differentiable

```
#pragma once

#include <cmath>

template<typename T>
void g(T &x, const T &p) {
  if (x<p)
    x=sin(x);
  else
    x=cos(x);
}
```

⇒

```
#pragma once

#include <cmath>

template<typename T>
void f(T &x, const T &p, const T &w) {
  T f1=sin(x);
  T f2=cos(x);
  x=1./(1.+exp(-(x-p)/w));
  x=f1*(1-x)+f2*x;
}
```

Consider a nonlinear equation $y = f(x) = 0$ at some (starting) point x.

Building on the assumption that $f(x + \Delta x) \approx f(x) + f'(x) \cdot \Delta x$ the root finding problem for f can be replaced locally by the root finding problem for the linearization

$$\bar{f}(\Delta x) = f(x) + f'(x) \cdot \Delta x \ .$$

The right-hand side is a straight line intersecting the y-axis in $(\Delta x = 0, \bar{f}(\Delta x) = f(x))$.

Solution of

$$\bar{f}(\Delta x) = f(x) + f'(x) \cdot \Delta x = 0$$

for Δx yields

$$\Delta x = -\frac{f(x)}{f'(x)}$$

implying $f(x + \Delta x) \approx 0$.

If the new iterate is not close enough to the root of the nonlinear function, i.e, $|f(x+\Delta x)| > \epsilon$ for some measure of accuracy of the numerical approximation $\epsilon > 0$, then it becomes the starting point for the next iteration yielding the recurrence

$$x = x - \frac{f(x)}{f'(x)}$$

Convergence of this method is not guaranteed in general. Damping of the magnitude of the next step may help.

$$x = x - \alpha \cdot \frac{f(x)}{f'(x)} \quad \text{for } 0 < \alpha \leq 1 \,.$$

The damping parameter α is often determined by line search (e.g, recursive bisection yielding $\alpha = 1,\ 0.5,\ 0.25,\ \ldots$) such that decrease in absolute function value is ensured.

Newton's Method

Implementation (E.g, $x^2 - p = 0$)

The following iteration terminates if the residual is close enough to zero or if a given number (maxit) of iterations was performed.

```
template<typename T, typename PT>
void newton(T &x, const T &p, const PT
    &eps, const unsigned int maxit) {
  unsigned int it=0;
  T f=pow(x,2)-p;
  do {
    T dfdx=2*x;
    x-=f/dfdx;
    f=pow(x,2)-p;
    if (++it==maxit) break;
  } while(fabs(f)>eps);
}
```

Alternatively, ...

```
template<typename T, typename PT>
T f(T &x, const PT &p) {
  return pow(x,2)-p;
}

template<typename T, typename PT>
T dfdx(T &x, const PT &) { return 2*x; }

template<typename T, typename PT>
void newton(T &x, const T &p, const PT
    &eps, const unsigned int maxit) {
  unsigned int it=0;
  T y=f(x,p);
  do {
    x-=y/dfdx(x,p);
    y=f(x,p);
    if (++it==maxit) break;
  } while(fabs(y)>eps);
}
```

We consider differentiable numerical programs

$$\begin{pmatrix} y \\ \tilde{y} \end{pmatrix} = F(x, \tilde{x}) : \boldsymbol{R}^n \times \boldsymbol{R}^{\tilde{n}} \to \boldsymbol{R}^m \times \boldsymbol{R}^{\tilde{m}}$$

mapping active ($x \in \boldsymbol{R}^n$) and passive ($\tilde{x} \in \boldsymbol{R}^{\tilde{n}}$) inputs onto active ($y \in \boldsymbol{R}^m$) and passive ($\tilde{y} \in \boldsymbol{R}^{\tilde{m}}$) outputs. The active output y is assumed to be differentiable with respect to x with

$$F' \equiv F'(x, \tilde{x}) = \frac{dy}{dx} \; .$$

The corresponding (first-order) tangent program computes

$$\begin{pmatrix} y \\ \tilde{y} \\ y^{(1)} \end{pmatrix} = F^{(1)}(x, \tilde{x}, x^{(1)}) \equiv \begin{pmatrix} F(x, \tilde{x}) \\ F' \cdot x^{(1)} \end{pmatrix} \; .$$

- all arguments active for $\frac{dx}{d(x\ p\ w)^T}$ or $\frac{dx}{d(p\ w)^T}$

```
template<typename T>
void f_t(T &x, T &x_t, const T &p, const T &p_t, const T &w, const T &w_t) {
  ...
}
```

- p passive for $\frac{dx}{d(x\ w)^T}$ or $\frac{dx}{dw}$

```
template<typename T>
void f_t(T &x, T &x_t, const T &p, const T &w, const T &w_t) {
  ...
}
```

- w passive for $\frac{dx}{d(x\ p)^T}$ or $\frac{dx}{dp}$

```
template<typename T>
void f_t(T &x, T &x_t, const T &p, const T &p_t, const T &w) {
  ...
}
```

For given values of the inputs $\mathrm{x} = (x_i)_{i=0}^{n-1}$ and $\tilde{\mathrm{x}}$ the active section

$$\mathrm{y} = (y_k)_{k=0}^{m-1} = \mathrm{y}(\mathrm{x}, \tilde{\mathrm{x}})$$

of a differentiable program $F(\mathrm{x}, \tilde{\mathrm{x}})$ decomposes into a sequence of $q = p + m$ differentiable elemental functions φ_j evaluated as a single assignment code

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

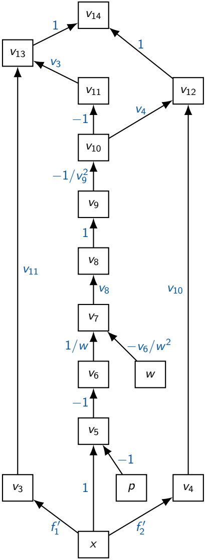
over active variables v_j, $j = 0, \ldots, n+q-1$ and where $v_i = x_i$ for $i = 0, \ldots, n-1$, $y_k = v_{n+p+k}$ for $k = 0, \ldots, m-1$.

The notation $k \prec j$ ($j \succ k$) marks v_k as an argument of φ_j.

Single Assignment Code

Example: Sigmoidal Smoothing

```
template<typename T>
void f(T &x, const T &p, const T &w) {
  std::vector<T> v(15);
  v[0]=x;
  v[1]=p,
  v[2]=w;
  v[3]=sin(v[0]);
  v[4]=cos(v[0]);
  v[5]=v[0]-v[1];
  v[6]=-v[5];
  v[7]=v[6]/v[2];
  v[8]=exp(v[7]);
  v[9]=1.0+v[8];
  v[10]=1.0/v[9];
  v[11]=1.0-v[10];
  v[12]=v[3]*v[11];
  v[13]=v[4]*v[10];
  v[14]=v[12]+v[13];
  x=v[14];
}
```



Assuming differentiability of all elemental functions the differentiation of

$$v_k = \varphi_k \left(\varphi_j(v_i, (v_\mu)_{i \neq \mu \prec j}), v_i, (v_\nu)_{\{i,j\} \neq \nu \prec k}\right)$$

with respect to v_i yields

$$\frac{dv_k}{dv_i} = \frac{dv_k}{dv_j} \cdot \frac{dv_j}{dv_i} + \frac{\partial v_k}{\partial v_i}$$

where $v_j = \varphi_j(v_i, (v_\mu)_{i \neq \mu \prec j})$.

The corresponding contribution to the directional derivative (tangent) of v_k becomes equal to

$$\frac{dv_k}{dv_i} \left(v_i, (v_\mu)_{i \neq \mu \prec j}, (v_\nu)_{\{i,j\} \neq \nu \prec k}\right) \cdot v_i^{(1)} \; .$$

Example: v_t[7]+=v_t[6]/v[2]; v_t[7]+=−v[6]∗v_t[2]/pow(v[2],2);

Tangent Single-Assignment Code

As an immediate consequence of the chain rule the directional derivative (tangent) of

$$\mathrm{y} = (y_k)_{k=0}^{m-1} = \mathrm{y}(\mathrm{x}, \tilde{\mathrm{x}})$$

with respect to $\mathrm{x} = (x_i)_{i=0}^{n-1}$ in direction $\mathrm{x}^{(1)} = (x_i^{(1)})_{i=0}^{n-1}$ is computed for given $v_i^{(1)} = x_i^{(1)}$ as

$$v_j^{(1)} = v_j^{(1)} + \frac{d\varphi_j}{dv_i}(v_k)_{k \prec j} \cdot v_i^{(1)} \quad \forall\ i \prec j$$

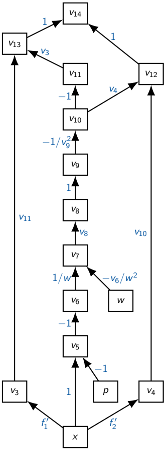
for $j = n, \ldots, n+q-1$ and all $v_j^{(1)}$ equal to zero initially.

The directional derivative is returned through $\mathrm{y}^{(1)} = \left(y_k^{(1)}\right)_{k=0}^{m-1}$ where $y_k^{(1)} = v_{n+p+k}^{(1)}$. Tangent arithmetic lends itself to implementation by overloading as a simple augmentation of the primal arithmetic with the computation of directional derivatives.

Tangent Single Assignment Code

Example: Sigmoidal Smoothing

```
template<typename T>
void f_t(T &x, T &x_t,
         const T &p, const T &p_t, const T &w, const T &w_t) {
  std::vector<T> v(15), v_t(15,0);
  v[0]=x; v_t[0]=x_t;
  v[1]=p, v_t[1]=p_t,
  v[2]=w; v_t[2]=w_t;
  v[3]=sin(v[0]); v_t[3]+=cos(v[0])*v_t[0];
  v[4]=cos(v[0]); v_t[4]+=-sin(v[0])*v_t[0];
  v[5]=v[0]-v[1]; v_t[5]+=v_t[0]; v_t[5]+=-v_t[1];
  v[6]=-v[5]; v_t[6]+=-v_t[5];
  v[7]=v[6]/v[2]; v_t[7]+=v_t[6]/v[2]; v_t[7]+=-v[6]*v_t[2]/pow(v[2],2);
  v[8]=exp(v[7]); v_t[8]+=v[8]*v_t[7];
  v[9]=1.0+v[8]; v_t[9]+=v_t[8];
  v[10]=1.0/v[9]; v_t[10]+=-v_t[9]/pow(v[9],2);
  v[11]=1.0-v[10]; v_t[11]+=-v_t[10];
  v[12]=v[3]*v[11]; v_t[12]+=v_t[3]*v[11]; v_t[12]+=v[3]*v_t[11];
  v[13]=v[4]*v[10]; v_t[13]+=v_t[4]*v[10]; v_t[13]+=v[4]*v_t[10];
  v[14]=v[12]+v[13]; v_t[14]+=v_t[12]; v_t[14]+=v_t[13];
  x=v[14]; x_t=v_t[14];
}
```



Outline

Recall

- Sigmoidal Smoothing
- Newton's Method

Tangent Code Generation Rules

Examples

- Tangent Straight-Line Code
- Tangent Intraprocedural Code
- Tangent Interprocedural Code

Tangent Code

Rules

TR1 The active data segment must be duplicated. Each active primal variable is matched by its tangent [of same type and shape].

TR2 Assignment-level tangent code must precede the respective primal assignments. Local tangent single assignment code simplifies differentiation of complex expressions.

TR3 The tangent flow of control is equal to the primal flow of control. Mind potential non-differentiability due to branching.

TR4 Calls to primal subprograms must be replaced with calls to the corresponding tangent subprograms. This rule generalizes to polymorphism (overloading, class hierarchies).

TR5 Drivers are required, e.g, for the accumulation of the gradient.

Tangent Code

Rule TR5: Driver

```
template<typename T>
void gradient(const std::vector<T> &x, T& y,
              std::vector<T> &grad) {
  size_t n=x.size();
  std::vector<T> x_t(n,0);
  for (size_t i=0;i<n;i++) {
    x_t[i]=1;
    f_t(x,x_t,y,grad[i]);
    x_t[i]=0;
  }
}
```

- A driver is required to extract the appropriate derivatives from the tangent code in the given context; e.g, the gradient element-wise as directional derivatives in the Cartesian basis directions.
- Directional derivatives in other directions may be required.

Tangent Code

Rule TR1: Duplication of Active Data Segment

```
void f(float x, float &y) {
  float z=x*x; y=sin(z);
}

void f_t(float x, float x_t,
         float &y, float &y_t) {
  float z_t=2*x*x_t;
  float z=x*x;
  y_t=cos(z)*z_t;
  y=sin(z);
}
```

- The signature is augmented with tangents `x_t` and `y_t` for the active arguments `x` and `y`.
- `x` is passed by value; so is `x_t` yielding two local variables of type **float**.
- `y` is passed by reference; so is `y_t` which needs to be declared outside of `f_t`.
- The local variable `z` is augmented with its tangent `z_t`.
- Both primal assignments are preceded by their (trivial) tangent assignments.

Tangent Code

Rule TR2: Assignment-Level Tangent Code

```
void f(float x, float &y) {
  y=sin(x*x);
}

void f_t(float x, float x_t,
         float &y, float &y_t) {
  float v_t=2*x*x_t;
  float v=x*x;
  y_t=cos(v)*v_t;
  y=sin(v);
}
```

- Each (one in this case) primal assignment is decomposed into a single assignment code augmented with its corresponding tangents.
- Optimization by *copy propagation* eliminates v_t yielding

```
T v=x*x;
y_t=cos(v)*2*x*x_t;
y=sin(v);
```

Tangent Code

Rule TR3: Tangent = Primal Flow of Control

```
template<typename T>
void f_t(const std::vector<T> &x, const
    std::vector<T> &x_t, T &y, T &y_t) {
  T xTx_t=0, xTx=0;
  for (size_t i=0;i<x.size();i++)
    if (i==0) {
      xTx_t=2*x[i]*x_t[i];
      xTx=pow(x[i],2);
    } else {
      xTx_t+=2*x[i]*x_t[i];
      xTx+=pow(x[i],2);
    }
  y_t=cos(xTx)*xTx_t; y=sin(xTx);
}
```

- For a given primal implementation of $y = \sin(x^T \cdot x)$ as

```
template<typename T>
void f(const std::vector<T> &x, T &y)
  {
  T xTx=0;
  for (size_t i=0;i<x.size();i++)
    if (i==0)
      xTx=pow(x[i],2);
    else
      xTx+=pow(x[i],2);
  y=sin(xTx);
}
```

  we obtain the tangent code on the left.

- Special care must be taken if the flow of control yields nondifferentiability; e.g, (sigmoidal) smoothing. We focus on differentiable programs.

Tangent Code

Rule TR4: Tangent Subprograms

```
void g(float x, float &y) {
  y=x*x;
}

void g_t(float x, float x_t,
         float &y, float &y_t) {
  y_t=2*x*x_t; y=x*x;
}

void f(float x, float &y) {
  float z; g(x,z); y=sin(z);
}

void f_t(float x, float x_t,
         float &y, float &y_t) {
  float z,z_t;
  g_t(x,x_t,z,z_t);
  y_t=cos(z)*z_t; y=sin(z);
}
```

- Calls to primal subprograms need to be replaced by calls to their tangents.
- Potentially induced nondifferentiability (e.g, due to recursion) needs to be dealt with. We focus on differentiable programs.

Outline

Recall
- Sigmoidal Smoothing
- Newton's Method

Tangent Code Generation Rules

Examples
- Tangent Straight-Line Code
- Tangent Intraprocedural Code
- Tangent Interprocedural Code

Tangent Code

Examples

In the following we apply the tangent code generation rules to “slightly more real-world” examples, namely the previously introduced implementations of

- sigmoidal smoothing to illustrate tangent straight-line code;
- Newton’s method to illustrate tangent
  - intraprocedural
  - interprocedural

  code.

Tangent Straight-Line Code

Example: Sigmoidal Smoothing

```
template<typename T>
void f_t(T &x, T &x_t, const T &p, const T &p_t, const T &w, const T &w_t) {
  T f1_t=cos(x)*x_t; T f1=sin(x);
  T f2_t=-sin(x)*x_t; T f2=cos(x);
  T aux=exp(-(x-p)/w);
  x_t=-pow(1./(1.+aux),2)*aux*(-x_t/w+p_t/w+w_t*(x-p)/pow(w,2));
  x=1./(1.+aux);
  x_t=f1_t*(1-x)-f1*x_t+f2_t*x+f2*x_t;
  x=f1*(1-x)+f2*x;
}
```

Tangent Intraprocedural Code

Example: Newton's Method

```
template<typename T, typename PT>
void f_t(T &x, T &x_t, const T &p, const T &p_t, const PT &eps, const unsigned int
    maxit) {
  unsigned int it=0;
  T f_t=2*x*x_t-p_t;
  T f=pow(x,2)-p;
  do {
    T dfdx_t=2*x_t;
    T dfdx=2*x;
    x_t-=f_t/dfdx-f*dfdx_t/pow(dfdx,2);
    x-=f/dfdx;
    f_t=2*x*x_t-p_t;
    f=pow(x,2)-p;
    if (++it==maxit) break;
  } while(fabs(f)>eps);
}
```

Tangent Interprocedural Code

Example: Newton's Method

```
template<typename T, typename PT>
void f_t(T &x, T& x_t, const PT &p, const PT &p_t, T &r, T &r_t);

template<typename T, typename PT>
void dfdx_t(T &x, T &x_t, const PT &, const PT &, T &drdx, T& drdx_t);

template<typename T, typename PT>
void newton_t(T &x, T &x_t, const T &p, const T &p_t, const PT &eps, const unsigned
     int maxit) {
  unsigned int it=0;
  T y,y_t;
  f_t(x,x_t,p,p_t,y,y_t);
  do {
    T dydx,dydx_t;
    dfdx_t(x,x_t,p,p_t,dydx,dydx_t);
    x_t-=y_t/dydx-y*dydx_t/pow(dydx,2);
    x-=y/dydx;
    f_t(x,x_t,p,p_t,y,y_t);
    if (++it==maxit) break;
  } while(fabs(y)>eps);
}
```

- association of tangents by address
- vector tangent mode
- assignment-level adjoint code
- lower-precision tangents

Summary

Recall

- Sigmoidal Smoothing
- Newton's Method

Tangent Code Generation Rules

Examples

- Tangent Straight-Line Code
- Tangent Intraprocedural Code
- Tangent Interprocedural Code