Introduction to Algorithmic Differentiation

AD by Hand (Adjoint Straight-Line Code)

Uwe Naumann

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

RWTH Aachen
Recall

Adjoint Programs
Adjoint Single-Assignment Code

Adjoint Programs Revisited

Adjoint Straight-Line Code
Adjoint Code Generation Rules
Example
Outline

Recall
  Adjoint Programs
  Adjoint Single-Assignment Code

Adjoint Programs Revisited

Adjoint Straight-Line Code
  Adjoint Code Generation Rules
  Example
Recall

Adjoint Programs

We consider differentiable numerical programs

\[
\begin{pmatrix}
  y \\
  \tilde{y}
\end{pmatrix}
:= F(x, \tilde{x}) : R^n \times R^{\tilde{n}} \rightarrow R^m \times R^{\tilde{m}}
\]

mapping active \((x \in R^n)\) and passive \((\tilde{x} \in R^{\tilde{n}})\) inputs onto active \((y \in R^m)\) and passive \((\tilde{y} \in R^{\tilde{m}})\) outputs. The active output \(y\) is assumed to be differentiable with respect to \(x\) with Jacobian \(F' = \frac{dy}{dx} \in R^{m \times n}\).

The corresponding (first-order) adjoint program computes

\[
\begin{pmatrix}
  y \\
  \tilde{y} \\
  x(1)
\end{pmatrix}
:= F_{(1)}(x, \tilde{x}, y(1)) \equiv \begin{pmatrix}
  F(x, \tilde{x}) \\
  y(1) \cdot F'
\end{pmatrix},
\]

for given \(y(1) \in R^{1 \times m}\) yielding \(x(1) \in R^{1 \times n}\).
Recall
Single Assignment Code

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

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

of a differentiable program \( F(x, \tilde{x}) \) decomposes into a sequence of \( q = p + m \) differentiable elemental functions \( \varphi_j \) 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 \( \varphi_j \).
Single Assignment Code
Example: Sigmoidal Smoothing

```c++
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[8] = exp(v[7]);
    v[9] = 1.0 + v[8];
    v[10] = 1.0 / v[9];
    x = v[14];
}
```
Recall

Chain Rule

Assuming differentiability of all elemental functions the differentiation of

\[ v_k := \varphi_k \left( \varphi_j(v_i, (v_\mu)_{i \neq \mu < j}), v_i, (v_\nu)_{\{i,j\} \neq \nu < k} \right) \]

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

The corresponding contribution to the adjoint of \( v_i \) becomes equal to

\[ \frac{dv_k}{dv_i} \left( v_i, (v_\mu)_{i \neq \mu < j}, (v_\nu)_{\{i,j\} \neq \nu < k} \right) \cdot v_k(1) \cdot \]

Example:

\[ v \_ a[6] += v \_ a[7]/v[2]; \ v \_ a[2] += -v \_ a[7]*v[6]/\text{pow}(v[2],2); \]
As an immediate consequence of the chain rule the adjoint of
\[ y = (y_k)_{k=0}^{m-1} = y(x, \tilde{x}) \]
with respect to \( x = (x_j)_{j=0}^{n-1} \) in direction \( y(1) = (y_i(1))_{i=0}^{m-1} \) is computed for given \( v_{i+n+p(1)} := y_i(1) \) as

\[ v_{j(1)} := v_{j(1)} + \frac{d\varphi_i}{dv_j} (v_k)_{k>j} \cdot v_i(1) \quad \forall \ i > j \]

for \( j = n + p, \ldots, 0 \) and all \( v_{j(1)} \) equal to zero initially.

The adjoint is returned through \( x(1) = (x_i(1))_{i=0}^{n-1} \) where \( x_i(1) := v_i(1) \).

Implementation of adjoint arithmetic by overloading requires recording and reverse interpretation of a tape.
Adjoint Single Assignment Code

Example: Sigmoidal Smoothing

```cpp
template<typename T>
void f_a(T &x, T &x_a, const T &p, T &p_a, const T &w, T& w_a) {
    std::vector<T> v(15), v_a(15,0);
    v[0]=x; v[1]=p, v[2]=w; ... x=v[14];

    v_a[14]=x_a;
    v_a[13]+=v_a[14]; v_a[12]+=v_a[14];
    v_a[10]+=−v_a[11];
    v_a[9]+=−v_a[10]/pow(v[9],2);
    v_a[8]+=v_a[9];
    v_a[7]+=v[8]*v_a[8];
    v_a[6]+=v_a[7]/v[2]; v_a[2]+=−v_a[7]*v[6]/pow(v[2],2);
    v_a[5]+=−v_a[6];
    v_a[1]+=−v_a[5]; v_a[0]+=v_a[5];
    v_a[0]+=−sin(v[0])*v_a[4];
    v_a[0]+=cos(v[0])*v_a[3];
    w_a=v_a[2];
    p_a=v_a[1];
    x_a=v_a[0];
}
```
Outline

Recall
  Adjoint Programs
  Adjoint Single-Assignment Code

Adjoint Programs Revisited

Adjoint Straight-Line Code
  Adjoint Code Generation Rules
  Example
Implementations of adjoints by overloading typically yield

\[
\begin{pmatrix}
  y \\
  \tilde{y}
\end{pmatrix}
= 
\begin{pmatrix}
  F(x, \tilde{x}) \\
  y(1) \cdot F'
\end{pmatrix}
.
\]

Potential incrementation of initially nonzero \( x(1) \) (context-sensitivity) needs to be implemented explicitly by the user. Manual or automatic source code transformation usually yields context-sensitive adjoint code as

\[
\begin{pmatrix}
  y \\
  \tilde{y}
\end{pmatrix}
=
\begin{pmatrix}
  0 \\
  x(1)
\end{pmatrix}
+ 
\begin{pmatrix}
  F(x, \tilde{x}) \\
  y(1) \cdot F'
\end{pmatrix}
\]

\( y(1) := 0 \)

assuming unaliased \( x(1) \) and \( y(1) \) (due to unaliased \( x \) and \( y \)); see below for discussion.
Outline

Recall
  Adjoint Programs
  Adjoint Single-Assignment Code

Adjoint Programs Revisited

Adjoint Straight-Line Code
  Adjoint Code Generation Rules
  Example
Adjoint Straight-Line Code

Rules

AR1  The active data segment must be duplicated. Adjoint variables must be initially equal to zero.

AR2  The (augmented) primal statements must be followed by the corresponding adjoint statements in reverse order. → augmented primal and adjoint sections

AR3  Adjoints of results of primal assignments must be set equal to zero after execution of the corresponding adjoint assignments.

AR4  Values required by the adjoint and lost in [augmented] primal section must be recovered. → data flow reversal

AR5  Primal results lost due to data flow reversal should be recovered.

AR6  Feasible interleavings of [augmented] primal and adjoint statements should be exploited for improved efficiency.

AR7  Aliasing must be dealt with.

AR10 Drivers are required, e.g, for the accumulation of the gradient.
Products of the transposed Jacobian with a given vector of adjoints of the primal results are computed.

Whole Jacobians can be accumulated by letting those directions range over the corresponding Cartesian basis directions.

Gradients can be obtained cheaply.
void f_a(float x, float &x_a,
    float &y, float &y_a) {
  // (augmented) primal section
  float z=x*x;
  y=sin(z);
  // adjoint section
  float z_a=0;
  z_a+=cos(z)*y_a;
  x_a+=2*x*z_a;
}

▶ The signature is augmented with adjoint $x_a$ and $y_a$ for the active arguments $x$ and $y$. Intents are defined according to data flow reversal.

▶ $x$ is pure input and $y$ is pure output $\Rightarrow x_a$ and $y_a$ are both input and output.

▶ The local variable $z$ is augmented with its adjoint $z_a$ which is initialized to zero.

▶ The (not yet augmented) primal section is followed by the adjoint section (adjoints of primal statements in reverse order).
Program variables can represent unrelated mathematical variables.

Illustration:

```c
// primal
y = x;  
x = z;  

// adjoint
z_a += x_a;  
x_a += y_a;  

⇒ wrong
```

⇒ right
Required values can be lost as the result of overwriting of variables. Recovery can be implemented by storage/restorage (store-all), recomputation from stored inputs (recompute-all), or combinations thereof. In the following we adopt the store-all strategy by pushing required values to and popping them from stacks of appropriate element types.

Illustration: Let $x, y, z$ have type $T$.

```
// primal
y = \sin(x);
x = z;

// adjoint
z_a += x_a; x_a = 0;
x_a += \cos(x) \ast y_a; y_a = 0;
```

```
// augmented primal
y = \sin(x);
tbr_T.push(x);
x = z;

// adjoint
x = tbr_T.top(); tbr_T.pop();
z_a += x_a; x_a = 0;
x_a += \cos(x) \ast y_a; y_a = 0;
```

$\Rightarrow$ wrong

$\Rightarrow$ right
Required values can be lost as the result of local variables leaving their scopes.

Illustration: Let $x, y, z$ have type $T$.

```
// primal
{ T x=...; y=sin(x); }
z=y;
...

// adjoint
...
y_a+=z_a; z_a=0;
{ T x; x_a+=cos(x)*y_a; y_a=0; ... }

⇒ wrong
```

```
// augmented primal
{ T x=...; y=sin(x); tbr_T.push(x); }
z=y;
...

// adjoint
...
y_a+=z_a; z_a=0;
{ T x=tbr_T.top(); tbr_T.pop();
  x_a+=cos(x)*y_a; y_a=0; ... }

⇒ right
```
The nonlinear use of $x$ in $y = x \times x$ defines the input value of $x$ as “to be recovered,” similarly: $y$ in $x = \sin(y)$.

Recovery must take place prior to first use of the lost primal value (extreme case: prior to the adjoint of the overwriting assignment itself in case of aliasing between left- and right-hand sides of the overwriting assignment).

Seeding of $x_a$ as the adjoint of an intermediate variable should be prohibited unless $x$ is regarded as an output of $f$ (default for pass by reference) instead of a nonpure input.
Primal results can be [partially] lost due to data flow reversal. Recovery can be implemented by storage/restorage.

Illustration: Let $x,y,z$ have type $T$.  

```cpp
1  std::stack<T> tbr_T;
2  // augmented primal
3  y = sin(x);
4  tbr_T.push(x);
5  x = z;
6  
7  // adjoint
8  x = tbr_T.top(); tbr_T.pop();
9  z.a += x.a; x.a = 0;
10  x.a += cos(x) * y.a; y.a = 0;
11
\Rightarrow primal wrong
```

\textcolor{red}{\Rightarrow primal right}\textcolor{red}{\Rightarrow primal right}
void f_a(float x, float &x_a,
    float &y, float &y_a) {
    std::stack<float> tbr;
    // augmented primal section
    y=x*x;
    tbr.push(x);
    x=sin(y);
    tbr.push(y);
    y=x;
    float out=y; // store primal result
    // adjoint section
    assert(x_a==0);
    y=tbr.top(); tbr.pop();
    x_a+=y_a; y_a=0;
    x=tbr.top(); tbr.pop();
    y_a+=cos(y)*x_a; x_a=0;
    x_a+=2*x*y_a; y_a=0;
    y=out; // restore primal result
    }

The value of y as the primal result is lost due to recovery of the required value of y as a nonlinear argument in x=sin(y).

storage at the end of the augmented primal section and restorage at the end of the adjoint section ensures correctness of y as the primal result to be returned by the adjoint subprogram.
Missing data dependence may allow for interleavings of augmented primal and adjoint statements.

Illustration:

```plaintext
1 // primal
2 y = sin(x);
3 x = z;

4 // adjoint
5 z_a += x_a; x_a = 0;
6 x_a += cos(x) * y_a; y_a = 0;

7 // primal
8 y = sin(x);

⇒ wrong

1 // primal
2 y = sin(x);
3 x = z;

4 // adjoint
5 z_a += x_a; x_a = 0;
6 x_a += cos(x) * y_a; y_a = 0;

7 // primal
8 y = sin(x);
9 x = z;

⇒ right
```
Adjoint Straight-Line Code

AR6: Example

```
void f_a(float x, float &x_a,
        float &y, float &y_a) {
    // (here not augmented) primal section
    y=x*x;
    // adjoint section interleaved
    assert(x_a==0);
    x_a+=y_a; y_a=0;
    y_a+=cos(y)*x_a; x_a=0;
    x_a+=2*x*y_a;
    // primal section (not augmented!)
    x=sin(y);
    y=x;
}
```

- Neither values of \(x\) or \(y\) computed in lines 11 and 12, respectively, are required by the adjoint section; they can be moved to the end of the subprogram.

- Consequently, no required value is lost prior to the adjoint section. The tbr stack can be eliminated entirely.
Adjoints among arguments and results of primal assignments yield custom variants of the adjoint code generation rules. Proofs follow immediately from local decompositions into alias-free sequences of primal assignments.

Illustration:

```
// primal
y+=x;

// adjoint
y_a+=sin(x)*x_a;
x_a+=cos(x)*y*x_a; x_a=0;
x_a+=y_a; y_a+=y_a; y_a=0;

// primal
x=sin(x)*y;
```

⇒ wrong

```
// primal
y+=x;

// adjoint
y_a+=sin(x)*x_a; x_a=cos(x)*y*x_a;
x_a+=y_a;

// primal
x=sin(x)*y;
```

⇒ right
Adjoint Straight-Line Code

AR7: Example

```c
void f_a(float x, float &x_a, float &y, float &y_a) {
    // primal section
    float tbr=x;
    x=x*x;
    // adjoint section interleaved
    assert(x_a==0);
    x_a+=cos(x)*y_a;
    // primal section
    y=sin(x);
    // adjoint section
    x=tbr;
    x_a+=2*x*x*y_a;
    x_a=2*x*x*x_a;
}
```

- Decomposing \(x=x*\) as \(v=x*\); \(x=v\); yields the adjoint

- The interleaved adjoint code ensures the correct primal result.

- Decomposing \(x=x*\) as \(v=x*\); \(x=v\); yields the adjoint

- The interleaved adjoint code ensures the correct primal result.
Adjoint Straight-Line Code
Example: Sigmoidal Smoothing

```cpp
#pragma once

#include <cmath>
#include <stack>

template<typename T>
void f_a(T &x, T &x_a, const T &p, T &p_a, const T &w, T &w_a) {
    std::stack<T> tbr_T;
    T f1=sin(x); T f2=cos(x); T aux=exp(-(x-p)/w);
    tbr_T.push(x); x=1./(1.+aux);
    tbr_T.push(x); x=f1*(1-x)+f2*x;
    T y=x;
    x=tbr_T.top(); tbr_T.pop(); T f1_a=0, f2_a=0, aux_a=0;
    f2_a+=x*x_a; f1_a+=-(1-x)*x_a; x_a=f2*x_a-f1*x_a;
    x=tbr_T.top(); tbr_T.pop(); aux_a+=-pow(1./(1.+aux),2)*x_a; x_a=0;
    x_a+=aux*x_a/w; p_a+=aux*x_a/w; w_a+=aux*(x-p)/pow(w,2)*aux_a; aux_a=0;
    x_a+=-sin(x)*f2_a;
    x_a+=cos(x)*f1_a;
    x=y;
}
```
Summary

Recall
Adjoint Programs
Adjoint Single-Assignment Code

Adjoint Programs Revisited

Adjoint Straight-Line Code
Adjoint Code Generation Rules
Example