Introduction to Algorithmic Differentiation

AD by Hand (Adjoint Straight-Line Code)

Uwe Naumann

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

RWTH Aachen

Contents

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

We consider differentiable numerical programs

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

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 Jacobian $F' = \frac{dy}{dx} \in \boldsymbol{R}^{m \times n}$.

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 $y_{(1)} \in \boldsymbol{R}^{1 \times m}$ yielding $x_{(1)} \in \boldsymbol{R}^{1 \times n}$.

Software and Tools for Computational Engineering
RWTH AACHEN UNIVERSITY

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

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

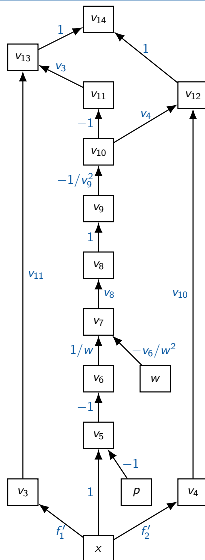
over active variables v_j, $j = 0, \ldots, n+q-1$ and where $v_i = x_i$ for $i = 0, \ldots, n-1$, $y_k = v_{n+p+k}$ for $k = 0, \ldots, m-1$.

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

Single Assignment Code

Example: Sigmoidal Smoothing

```
 1  template<typename T>
 2  void f(T &x, const T &p, const T &w) {
 3    std::vector<T> v(15);
 4    v[0]=x;
 5    v[1]=p,
 6    v[2]=w;
 7    v[3]=sin(v[0]);
 8    v[4]=cos(v[0]);
 9    v[5]=v[0]-v[1];
10    v[6]=-v[5];
11    v[7]=v[6]/v[2];
12    v[8]=exp(v[7]);
13    v[9]=1.0+v[8];
14    v[10]=1.0/v[9];
15    v[11]=1.0-v[10];
16    v[12]=v[3]*v[11];
17    v[13]=v[4]*v[10];
18    v[14]=v[12]+v[13];
19    x=v[14];
20  }
```



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 adjoint of v_i becomes equal to

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

Example: `v_a[6]+=v_a[7]/v[2]; v_a[2]+=-v_a[7]*v[6]/pow(v[2],2);`

As an immediate consequence of the chain rule the adjoint of

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

with respect to $\mathbf{x} = (x_j)_{j=0}^{n-1}$ in direction $\mathbf{y}_{(1)} = (y_{i(1)})_{i=0}^{m-1}$ is computed for given $v_{i+n+p(1)} := y_{i(1)}$ as

$$v_{j(1)} := v_{j(1)} + \frac{d\varphi_i}{dv_j}(v_k)_{k \succ j} \cdot v_{i(1)} \quad \forall\ i \succ j$$

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

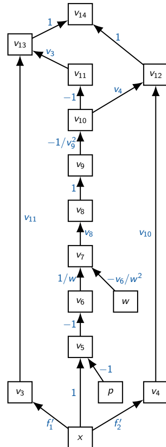
The adjoint is returned through $\mathbf{x}_{(1)} = \left(x_{i(1)}\right)_{i=0}^{n-1}$ where $x_{i(1)} := v_{i(1)}$. Implementation of adjoint arithmetic by overloading requires recording and reverse interpretation of a tape.

Adjoint Single Assignment Code

Example: Sigmoidal Smoothing

```
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[4]*v_a[13]; v_a[4]+=v[10]*v_a[13];
  v_a[11]+=v[3]*v_a[12]; v_a[3]+=v[11]*v_a[12];
  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

$$\begin{pmatrix} y \\ \tilde{y} \\ x_{(1)} \end{pmatrix} := \begin{pmatrix} F(x,\tilde{x}) \\ y_{(1)} \cdot F' \end{pmatrix} .$$

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

$$\begin{pmatrix} y \\ \tilde{y} \\ x_{(1)} \end{pmatrix} := \begin{pmatrix} 0 \\ x_{(1)} \end{pmatrix} + \begin{pmatrix} F(x,\tilde{x}) \\ y_{(1)} \cdot F' \end{pmatrix}$$

$$y_{(1)} := 0$$

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.

```
1  #include "f_a.hpp"
2
3  template<typename T>
4  void gradient(const std::vector<T> &x,
5          T& y, std::vector<T> &grad) {
6    size_t n=x.size();
7    assert(n==grad.size());
8    grad.clear();
9    T y_a=1;
10   f_a(x,grad,y,y_a);
11 }
```

- 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.

Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

```
1  void f_a(float x, float &x_a,
2       float &y, float &y_a) {
3    // (augmented) primal section
4    float z=x*x;
5    y=sin(z);
6    // adjoint section
7    float z_a=0;
8    z_a+=cos(z)*y_a;
9    x_a+=2*x*z_a;
10 }
```

- 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).

Adjoint Straight-Line Code

AR3: Reset Adjoints of Primal Results

Software and Tools for Computational Engineering | RWTH AACHEN UNIVERSITY

Program variables can represent unrelated mathematical variables.

Illustration:

```
1  // primal
2  y=x;
3  x=z;
4
5  // adjoint
6  z_a+=x_a;
7  x_a+=y_a;
```

⇒ wrong

```
1  // primal
2  y=x;
3  x=z;
4
5  // adjoint
6  z_a+=x_a; x_a=0;
7  x_a+=y_a; y_a=0;
```

⇒ 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.

```
1 // primal
2 y=sin(x);
3 x=z;
4
5 // adjoint
6 z_a+=x_a; x_a=0;
7 x_a+=cos(x)*y_a; y_a=0;
```

⇒ wrong

```
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;
```

⇒ right

Adjoint Straight-Line Code

AR4: Data Flow Reversal

Required values can be lost as the result of local variables leaving their scopes.

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

```
1  // primal
2  { T x=...; y=sin(x); }
3  z=y;
4  ...
5
6  // adjoint
7  ...
8  y_a+=z_a; z_a=0;
9  { T x; x_a+=cos(x)*y_a; y_a=0; ... }
```

⇒ wrong

```
1  std::stack<T> tbr_T;
2  // augmented primal
3  { T x=...; y=sin(x); tbr_T.push(x); }
4  z=y;
5  ...
6
7  // adjoint
8  ...
9  y_a+=z_a; z_a=0;
10 {
11   T x=tbr_T.top(); tbr_T.pop();
12   x_a+=cos(x)*y_a; y_a=0; ...
13 }
```

⇒ right

Adjoint Straight-Line Code

AR4: Example

Software and Tools for Computational Engineering | RWTH AACHEN UNIVERSITY

```
1  void f_a(float x, float &x_a,
2           float &y, float &y_a) {
3    std::stack<float> tbr;
4    // augmented primal section
5    y=x*x;
6    tbr.push(x);
7    x=sin(y);
8    tbr.push(y);
9    y=x;
10   // adjoint section
11   assert(x_a==0); // intermediate x!
12   y=tbr.top(); tbr.pop();
13   x_a+=y_a; y_a=0;
14   x=tbr.top(); tbr.pop();
15   y_a+=cos(y)*x_a; x_a=0;
16   x_a+=2*x*y_a;
17 }
```

- The nonlinear use of `x` in `y=x*x` defines the input value of `x` as "to be reco[vered/rded];" 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.

Adjoint Straight-Line Code

AR5: Recover Lost Primal Results

Software and Tools for Computational Engineering — RWTH AACHEN UNIVERSITY

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.

```
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;
```

⇒ primal wrong

```
1  std::stack<T> tbr_T;
2  // augmented primal
3  y=sin(x);
4  tbr_T.push(x);
5  x=z;
6
7  T r=x;
8
9  // adjoint
10 x=tbr_T.top(); tbr_T.pop();
11 z_a+=x_a; x_a=0;
12 x_a+=cos(x)*y_a; y_a=0;
13
14 x=r;
```

⇒ primal right

Adjoint Straight-Line Code

AR5: Example


```
1  void f_a(float x, float &x_a,
2           float &y, float &y_a) {
3    std::stack<float> tbr;
4    // augmented primal section
5    y=x*x;
6    tbr.push(x);
7    x=sin(y);
8    tbr.push(y);
9    y=x;
10   float out=y; // store primal result
11   // adjoint section
12   assert(x_a==0);
13   y=tbr.top(); tbr.pop();
14   x_a+=y_a; y_a=0;
15   x=tbr.top(); tbr.pop();
16   y_a+=cos(y)*x_a; x_a=0;
17   x_a+=2*x*y_a; y_a=0;
18   y=out; // restore primal result
19 }
```

- 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.

Adjoint Straight-Line Code

AR6: Exploit Interleavings

Missing data dependence may allow for interleavings of augmented primal and adjoint statements.

Illustration:

```
1 // primal
2 y=sin(x);
3 x=z;
4
5 // adjoint
6 z_a+=x_a; x_a=0;
7 x_a+=cos(x)*y_a; y_a=0;
```

⇒ wrong

```
1 // primal
2 y=sin(x);
3
4 // adjoint
5 z_a+=x_a; x_a=0;
6 x_a+=cos(x)*y_a; y_a=0;
7
8 // primal
9 x=z;
```

⇒ right

Adjoint Straight-Line Code

AR6: Example

```
1  void f_a(float x, float &x_a,
2             float &y, float &y_a) {
3    // (here not augmented) primal section
4    y=x*x;
5    // adjoint section interleaved
6    assert(x_a==0);
7    x_a+=y_a; y_a=0;
8    y_a+=cos(y)*x_a; x_a=0;
9    x_a+=2*x*y_a;
10   // primal section (not augmented!)
11   x=sin(y);
12   y=x;
13 }
```

- 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.

Aliasing 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:

```
1  // primal
2  y+=x;
3
4  // adjoint
5  y_a+=sin(x)*x_a;
6  x_a+=cos(x)*y*x_a; x_a=0;
7  x_a+=y_a; y_a+=y_a; y_a=0;
8
9  // primal
10 x=sin(x)*y;
```

⇒ wrong

```
1 // primal
2 y+=x;
3
4 // adjoint
5 y_a+=sin(x)*x_a; x_a=cos(x)*y*x_a;
6 x_a+=y_a;
7
8 // primal
9 x=sin(x)*y;
```

⇒ right

Adjoint Straight-Line Code

AR7: Example


```
1   void f_a(float x, float &x_a, float &y,
      float &y_a) {
2     // primal section
3     float tbr=x;
4     x=x*x;
5     // adjoint section interleaved
6     assert(x_a==0);
7     x_a+=cos(x)*y_a;
8     // primal section
9     y=sin(x);
10    // adjoint section
11    x=tbr;
12    x_a=2*x*x_a;
13  }
```

- Decomposing x=x*x as v=x*x; x=v; yields the adjoint

```
v=x*x;
tbr=x;
x=v;
x=tbr;
v_a=0;
v_a+=x_a; x_a=0;
x_a+=2*x*v_a; v_a=0;
```

simplifying to

```
tbr=x;
x=x*x;
x=tbr;
x_a=2*x*x_a;
```

- The interleaved adjoint code ensures the correct primal result.

Adjoint Straight-Line Code

Example: Sigmoidal Smoothing

Software and Tools for Computational Engineering
RWTH AACHEN UNIVERSITY

```
1  #pragma once
2
3  #include <cmath>
4  #include <stack>
5
6  template<typename T>
7  void f_a(T &x, T &x_a, const T &p, T &p_a, const T &w, T &w_a) {
8    std::stack<T> tbr_T;
9    T f1=sin(x); T f2=cos(x); T aux=exp(-(x-p)/w);
10   tbr_T.push(x); x=1./(1.+aux);
11   tbr_T.push(x); x=f1*(1-x)+f2*x;
12   T y=x;
13   x=tbr_T.top(); tbr_T.pop(); T f1_a=0, f2_a=0, aux_a=0;
14   f2_a+=x*x_a; f1_a+=(1-x)*x_a; x_a=f2*x_a-f1*x_a;
15   x=tbr_T.top(); tbr_T.pop(); aux_a+=-pow(1./(1.+aux),2)*x_a; x_a=0;
16   x_a+=aux*-aux_a/w; p_a+=aux*aux_a/w; w_a+=aux*(x-p)/pow(w,2)*aux_a; aux_a=0;
17   x_a+=-sin(x)*f2_a;
18   x_a+=cos(x)*f1_a;
19   x=y;
20 }
```

Summary

Recall

- Adjoint Programs
- Adjoint Single-Assignment Code

Adjoint Programs Revisited

Adjoint Straight-Line Code

- Adjoint Code Generation Rules
- Example