Introduction to Algorithmic Differentiation

AD by Hand (Adjoint Intra- and Interprocedural Code)

Uwe Naumann

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

RWTH Aachen

Contents

Outline

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. $\rightarrow$ 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. $\rightarrow$ 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.

$\vdots$

AR10 Drivers are required, e.g, for the accumulation of the gradient.

Outline

AR8 The primal flow of control must be reversed.

- AR8.1 Reverse irreducible flow of control by enumeration of basic blocks.
- AR8.2 Reverse reducible flow of control by counting loop iterations and by enumerating branches.
- AR8.3 Reverse simple **for**-loops explicitly.

AR9 Calls to subprograms need to be replaced in the augmented primal / adjoint section of the caller with calls to their augmented primal / adjoint sections.

Outline

Adjoint Code

Rule AR8.1


```
1  template<typename T>
2  void f_a(const std::vector<T> &x,
3           std::vector<T> &x_a, T &y, T &y_a) {
4    std::stack<size_t> bbs,tbr;
5    // augmented primal section
6    T xTx=42;
7    bbs.push(0);
8    for (size_t i=0;i<x.size();i++) {
9      if (i==0) {
10       xTx=pow(x[i],2);
11       bbs.push(1);
12     }
13     else {
14       xTx+=pow(x[i],2);
15       bbs.push(2);
16     }
17     tbr.push(i);
18     bbs.push(3);
19   }
20   y=exp(sin(xTx));
21   bbs.push(4);
22 ...
```

- Basic blocks are enumerated and their indexes are pushed onto the bbs stack whenever the respective basic block is executed.
- Primal values which need to be recorded (here only i) are pushed onto the tbr stack prior to being overwritten.

Adjoint Code

Rule AR8.1

```
1  ...
2    // adjoint section
3    T xTx_a=0; size_t i=0;
4    while (!bbs.empty()) {
5      switch (bbs.top()) {
6        case 0: break;
7        case 1:
8          x_a[i]+=2*x[i]*xTx_a; xTx_a=0;
9          break;
10       case 2:
11         x_a[i]+=2*x[i]*xTx_a;
12         break;
13       case 3:
14         i=tbr.top(); tbr.pop();
15         break;
16       case 4:
17         xTx_a+=y*cos(xTx)*y_a; y_a=0;
18         break;
19     }
20     bbs.pop();
21   }
22 }
```

- Indexes of basic blocks executed in the augmented primal section are recovered from the bbs stack. Last-in-first-out ensures the reversal of the flow of control.
- Rules for the generation of adjoint straight-line code apply inside of basic blocks.

Adjoint Code

Rule AR8.2


```
1  template<typename T>
2  void f_a(const std::vector<T> &x,
3           std::vector<T> &x_a, T &y, T &y_a) {
4    std::stack<size_t> cf,tbr;
5    // augmented primal section
6    T xTx=42;
7    size_t lc=0;
8    for (size_t i=0;i<x.size();i++) {
9      if (i==0) {
10       xTx=pow(x[i],2);
11       cf.push(1);
12     } else {
13       xTx+=pow(x[i],2);
14       cf.push(0);
15     }
16     lc++;
17     tbr.push(i);
18   }
19   cf.push(lc);
20   y=exp(sin(xTx));
21 ...
```

- Numbers of loop iterations are counted and pushed onto the cf stack right after the loop.
- Branches are enumerated and their indexes are pushed onto the cf stack in case of execution as part of the augmented primal section. Pushes need to take place at the end of the branch to ensure correctness of the control flow reversal.

Adjoint Code

Rule AR8.2


```
1  ...
2    // adjoint section
3    T xTx_a=y*cos(xTx)*y_a; y_a=0;
4    lc=cf.top(); cf.pop();
5    for (size_t j=0;j<lc;j++) {
6      size_t i=tbr.top(); tbr.pop();
7      size_t b=cf.top(); cf.pop();
8      if (b) {
9        x_a[i]+=2*x[i]*xTx_a; xTx_a=0;
10     } else
11       x_a[i]+=2*x[i]*xTx_a;
12   }
13 }
```

- The order of statements is reversed at all levels of the flow of control hierarchy (levels defined by nesting of loops and branches).
- Loop iterations are reversed by executing the adjoint loop body as often as recorded by the augmented primal section $\rightarrow$ lc.
- Adjoint branch statements (**if**, **switch**) execute the adjoint of the branch taken by the augmented primal section.

Adjoint Code

Rule AR8.3


```
1  template<typename T>
2  void f_a(const std::vector<T> &x,
3           std::vector<T> &x_a, T &y, T &y_a) {
4    // augmented primal section
5    T xTx=42;
6    for (size_t i=0;i<x.size();i++)
7      if (i==0)
8        xTx=pow(x[i],2);
9      else
10       xTx+=pow(x[i],2);
11   y=exp(sin(xTx));
12   // adjoint section
13   T xTx_a=y*cos(xTx)*y_a; y_a=0;
14   for (size_t i=x.size();i>0;i--)
15     x_a[i-1]+=2*x[i-1]*xTx_a; // note offset
16 }
```

- Simple **for**-loop allow for reversal in the context of adjoints through reversal of the loop counter evolution, e.g, i++ $\Rightarrow$ i−−.
- Validity of the adjoint loop termination condition in the presence of **unsigned** loop counter types needs to be ensures, e.g. size_t i=0; i−− $\Rightarrow$ i>0.

Adjoint Intraprocedural Code

Example: Newton's Method (Augmented Primal)


```
1  template<typename T, typename PT>
2  void f_a(T &x, T &x_a, const T &p, T &p_a, const PT &eps, const unsigned int maxit) {
3    std::stack<T> tbr;
4    unsigned int it=0;
5    T f=pow(x,2)-p;
6    do {
7      T dfdx=2*x;
8      tbr.push(x);
9      x-=f/dfdx;
10     tbr.push(f);
11     f=pow(x,2)-p;
12     tbr.push(dfdx);
13     if (++it==maxit) break;
14   } while(fabs(f)>eps);
15   T x_out=x;
16 ...
```

Adjoint Intraprocedural Code

Example: Newton's Method (Adjoint)


```
1  ...
2    T f_a=0;
3    for (auto i=it;i>0;--i) {
4      T dfdx_a=0;
5      T dfdx=tbr.top(); tbr.pop();
6      f=tbr.top(); tbr.pop();
7      p_a+=-f_a; x_a+=2*x*f_a; f_a=0;
8      x=tbr.top(); tbr.pop();
9      f_a+=-x_a/dfdx; dfdx_a+=x_a*f/pow(dfdx,2);
10     x_a+=2*dfdx_a;
11   }
12   p_a+=-f_a; x_a+=2*x*f_a;
13   x=x_out;
14 }
```

Outline

Adjoint Code

AR9


```
1  enum mode { primal, adjoint };
2
3  template<typename T>
4  void g_a(mode m, const T &x,
5           T &x_a, T &y, T &y_a) {
6    if (m==mode::primal) {
7      y+=pow(x,2);
8    } else {
9      x_a+=2*x*y_a;
10   }
11 }
```

- Augmented primal and adjoint sections of non-top-level subprograms need to be called separately → mode.

Adjoint Code

AR9


```
1  template<typename T>
2  void f_a(const std::vector<T> &x, i
3          std::vector<T> &x_a, T &y, T &y_a) {
4    // augmented primal section
5    T xTx=42, xTx_a=0;
6    for (size_t i=0;i<x.size();i++)
7      if (i==0)
8        xTx=pow(x[i],2);
9      else
10       g_a(mode::primal,x[i],x_a[i],xTx,xTx_a);
11   y=exp(sin(xTx));
12   // adjoint section
13   xTx_a+=y*cos(xTx)*y_a;
14   for (size_t i=x.size();i>0;i--) // note offset
15     if (i==0)
16       x_a[i-1]+=2*x[i-1]*xTx_a;
17     else
18       g_a(mode::adjoint,x[i-1],x_a[i-1],xTx,xTx_a);
19 }
```

- Callers call augmented primal sections of callees in their augmented primal sections; they call adjoint sections of callees in their adjoint sections.
- Note: Required values may get lost on exit from the callee due to local nonstatic variables running out of scope.

Adjoint Interprocedural Code

Example: Newton's Method (Residual)


```
1  enum mode { primal, adjoint };
2
3  template<typename T, typename PT>
4  T f(const T &x, const PT &p) {
5      return pow(x,2)-p;
6  }
7
8  template<typename T, typename PT>
9  void f_a(mode m, const T &x, T &x_a, const PT &p, PT &p_a, T &y, T &y_a) {
10   static std::stack<T> tbr;
11   if (m==mode::primal) {
12     tbr.push(y);
13     y=pow(x,2)-p;
14   } else {
15     y=tbr.top(); tbr.pop();
16     p_a+=-y_a; x_a+=2*x*y_a; y_a=0;
17   }
18 }
```

Adjoint Interprocedural Code

Example: Newton's Method (Derivative of Residual)


```
1  template<typename T, typename PT>
2  T dfdx(T &x, const PT &) {
3    return 2*x;
4  }
5
6  template<typename T, typename PT>
7  void dfdx_a(mode m, const T &x, T &x_a, const PT &, PT &, T &dydx, T &dydx_a) {
8    static std::stack<T> tbr;
9    if (m==mode::primal) {
10     tbr.push(dydx);
11     dydx=2*x;
12   } else {
13     dydx=tbr.top(); tbr.pop();
14     x_a+=2*dydx_a; dydx_a=0;
15   }
16 }
```

Adjoint Interprocedural Code

Example: Newton's Method (Primal)


```
1  template<typename T, typename PT>
2  void newton(T &x, const PT &p, const PT &eps, const unsigned int maxit) {
3    unsigned int it=0;
4    T y=f(x,p);
5    do {
6      x-=y/dfdx(x,p);
7      y=f(x,p);
8      if (++it==maxit) break;
9    } while(fabs(y)>eps);
10 }
```

Adjoint Interprocedural Code

Example: Newton's Method (Augmented Primal)


```
1  template<typename T, typename PT>
2  void newton_a(T &x, T &x_a, const PT &p, PT &p_a, const PT &eps, const unsigned int
       maxit) {
3    std::stack<T> tbr;
4    unsigned int it=0;
5    T y,y_a,dydx,dydx_a;
6    f_a(mode::primal,x,x_a,p,p_a,y,y_a);
7    do {
8      dfdx_a(mode::primal,x,x_a,p,p_a,dydx,dydx_a);
9      tbr.push(x);
10     x-=y/dydx;
11     f_a(mode::primal,x,x_a,p,p_a,y,y_a);
12     if (++it==maxit) break;
13   } while(fabs(y)>eps);
14   T x_out=x;
15 ...
```

Adjoint Interprocedural Code

Example: Newton's Method (Adjoint)


```
1  ...
2    y_a=0; dydx_a=0;
3    for (auto i=it;i>0;--i) {
4      f_a(mode::adjoint,x,x_a,p,p_a,y,y_a);
5      x=tbr.top(); tbr.pop();
6      y_a+=-x_a/dydx; dydx_a+=x_a*y/pow(dydx,2);
7      dfdx_a(mode::adjoint,x,x_a,p,p_a,dydx,dydx_a);
8    }
9    f_a(mode::adjoint,x,x_a,p,p_a,y,y_a);
10   x=x_out;
11 }
```

Adjoint Driver

Example: Newton's Method


```
1 template<typename T, typename PT>
2 std::vector<T> gradient(T &x, const T &p, const PT &eps, const unsigned int maxit) {
3   std::vector<T> g;
4   T x_a=1,p_a=0;
5   newton_a(x,x_a,p,p_a,eps,maxit);
6   g.push_back(x_a);
7   g.push_back(p_a);
8   return g;
9 }
```

- association of adjoints by address
- vector adjoint mode
- lower-precision adjoints
- interval (etc.) adjoints

Summary

Recall

Adjoint Code Generation Rules

Adjoint Intraprocedural Code

- Rules
- Example: Newton's Method

Adjoint Interprocedural Code

- Rules
- Example: Newton's Method