Season's Greetings
by Algorithmic Differentiation

Uwe Naumann

Software and Tools for Computational Engineering, RWTH Aachen University, Germany

December 2023

Abstract

Algorithmic Differentiation (AD) is used to implement type-generic tangent and adjoint versions of

$$y = \sum_{i=0}^{n-1} x_{2i} \cdot x_{2i+1}$$

in C++. Instantiations with the data type of **x** equal to **char** and output of the gradient at $(101\ 77\ 114\ 114\ 32\ 121\ 109\ 88\ 115\ 97)^T$ to std :: cout yields "Merry Xmas".

Similarly, type-generic sparsity-aware second-order tangent and second-order adjoint versions of

$$y = \frac{1}{6} \cdot \sum_{i=0}^{n-1} x_i^3$$

yield "Happy 2024" at $(72\ 97\ 112\ 112\ 121\ 32\ 50\ 48\ 50\ 52)^T$. Zeros can be added to the input vector to explore the significantly varying run times of the different derivative codes while not modfiying its output. The source code can be found on `https://github.com/un110076/SeasonsGreetings`.

Contents

1 AD wishes Merry Xmas! 1
- 1.1 The Maths . . . 1
- 1.2 The Code . . . 2
- 1.3 The Run Times . . . 6

2 AD wishes Happy 2024! 6
- 2.1 The Maths . . . 6
- 2.2 The Code . . . 7
- 2.3 The Run Times . . . 11

3 Conclusion: Merry Xmas and Happy 2024! 12

1 AD wishes Merry Xmas!

1.1 The Maths

The gradient of the function $f : \mathbb{R}^N \to \mathbb{R} : y = f(\mathbf{x})$,

$$y = \sum_{i=0}^{n-1} x_{2i} \cdot x_{2i+1} \ ,$$

where $\mathbf{x} = (x_j)_{j=0}^{N-1}$ and $N = 2n$ for given $n \geq 0$, is easily found to be equal to

$$f'(\mathbf{x}) = \left(\left(\begin{pmatrix} x_{2i+1} \\ x_{2i} \end{pmatrix} \right)_{i=0}^{n-1} \right)^T \in \mathbb{R}^{1 \times N} \ .$$

For example, $f'(\mathbf{x}) = (1\ 2\ 3\ 4)$ at $\mathbf{x} = (2\ 1\ 4\ 3)^T$.

A (first-order) *tangent* version

$$f^{(1)} : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R} : y^{(1)} = f^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$$

of f resulting from tangent AD [1] applied to a given differentiable implementation of f computes

$$y^{(1)} \equiv f'(\mathbf{x}) \cdot \mathbf{x}^{(1)}$$

without explicit accumulation of the gradient. The latter results from letting $\mathbf{x}^{(1)}$ range over the Cartesian basis vectors $\mathbf{e}_i \in \mathbb{R}^N$, resulting in N calls of $f^{(1)}$ for $i = 0, \ldots, N-1$. For example,

$$y^{(1)} = f'(\mathbf{x}) \cdot \mathbf{e}_2 = (1\ 2\ 3\ 4) \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = 3 \ .$$

The notation is adopted from [2], where the superscript $^{(1)}$ marks tangent versions of the original function f and of its variables $\mathbf{x}$ and y.

A (first-order) *adjoint* version

$$f_{(1)} : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N : \mathbf{x}_{(1)} = f_{(1)}(\mathbf{x}, y_{(1)})$$

resulting from adjoint AD [1] applied to a given differentiable implementation of f computes

$$\mathbf{x}_{(1)} \equiv y_{(1)} \cdot f'(\mathbf{x})$$

without prior accumulation of the gradient. Set $y_{(1)} = 1$ to obtain the latter by a single evaluation of $f_{(1)}$, for example, $\mathbf{x}_{(1)} = 1 \cdot f'(\mathbf{x}) = 1 \cdot (1\ 2\ 3\ 4) = (1\ 2\ 3\ 4)$. As in [2], the subscript $_{(1)}$ marks adjoint versions of the original function f and of its variables $\mathbf{x}$ and y.

1.2 The Code

Consider the following implementation of f in C++.

```
template<int N, typename T>
void f(T x[], T &y) {
  y=0;
  for (int i=0;i<N;i+=2) {
    y+=x[i]*x[i+1];
  }
}
```

Instantiation with a given size N for the input vector x with elements of a given type T triggers evaluation of f in T-arithmetic. All operators (=,+=,*) need to be defined for type T, which is certainly the case for all built-in arithmetic types, such as **float**, **int**, ... as well as **char**.

Tangent AD yields $f^{(1)}$ as follows:

```
// tangent version of f
template<int N, typename T>
void f_t1(T x[], T x_t1[], T &y_t1) {
  y_t1=0;
  for (int i=0;i<N;i+=2) {
    // product rule
    y_t1+=x_t1[i]*x[i+1]+x[i]*x_t1[i+1];
  }
}
```

Given values of the two input vectors x and x_t1 define the scalar output y_t1. The superscript $^{(1)}$ in the mathematical notation is represented by _t1 in the C++ source code. Well-known tangent code generation rules [2], for example, the product rule, are applied.

Adjoint AD yields $f_{(1)}$ as follows:

```
// adjoint version of f
template<int N, typename T>
void f_a1(T x[], T y_a1, T x_a1[]) {
  for (int i=N-2;i>=0;i-=2) {
    // adjoint product rule
    x_a1[i]=y_a1*x[i+1];
    x_a1[i+1]=y_a1*x[i];
  }
}
```

Given values of the two inputs x and y_a1 define the scalar output vector x_a1. The subscript $_{(1)}$ in the mathematical notation is represented by _a1 in the C++ source code. Well-known adjoint code generation rules are applied; see also [2].

The gradient is printed to std::cout in tangent mode by the following driver routine:

```
// gradient in tangent mode
template<int N, typename T>
void dfdx_t1(T x[]) {
  // declare tangent variables
  T x_t1[N], y_t1;
  // prepare for Cartesian basis vectors e_i
  for (int i=0;i<N;++i) x_t1[i]=0;
  for (int i=0;i<N;++i) {
    // set tangent of input equal to e_i
    x_t1[i]=1;
    // call tangent of f
    f_t1<N>(x,x_t1,y_t1);
    // print tangent of output
    std::cout << y_t1;
    // prepare for next Cartesian basis vector
    x_t1[i]=0;
  }
}
```

All steps are documented by comments in the source code. Following initializations, N evaluations of the tangent of f are performed with tangents of the input ranging over the corresponding Cartesian basis vectors. The gradient

$$f'(\mathbf{x}) = (77\ 101\ 114\ 114\ 121\ 32\ 88\ 109\ 97\ 115)$$

is computed and printed entry by entry.

The same task is completed more efficiently in adjoint mode by the following driver:

```
// gradient in adjoint mode
template<int N, typename T>
void dfdx_a1(T x[]) {
  // declare adjoint variables
  T x_a1[N]; T y_a1;
  // set adjoint of output equal to one
  y_a1=1;
  // call adjoint of f
  f_a1<N>(x,y_a1,x_a1);
  // print adjoint of input
  for (int i=0;i<N;++i) std::cout << x_a1[i];
}
```

A single evaluation of the adjoint function is performed with the adjoint of the output set equal to one. All entries of the gradient are computed simultaneously.

All of the above is stored in the C++ header file f1.h. It is included into the following source of the entire program.

```
#include "f1.h"
```

```
#include <iostream>
#include <chrono>

int main() {
  using namespace std;
  using namespace std::chrono;
  // increase to experience superior run time of adjoint
  const int N=231224; // e.g., YYMMDD; must be even
  // code to be executed in integer arithmetic
  using T=char;
  // size of x equal to size of gradient
  T x[N];
  // pairs of custom input values distributed uniformly
  T xv[]={101,77,114,114,32,121,109,88,115,97};
  for (int i=0,j=0;i<N;++i) {
    if ((!(i%(N/5)))&&(j<10)) {
      x[i]=xv[j++];
      x[i+++1]=xv[j++];
    } else {
      x[i]=0;
    }
  }
  // Season's Greetings by Tangent AD ...
  cout << "Tangent AD wishes ";
  // start time measurement
  auto t_begin=system_clock::now();
  // compute gradient in tangent mode
  dfdx_t1<N>(x);
  // finish time measurement
  auto t_end=system_clock::now();
  // report run time
  cout << "! (taking "
    << duration_cast<milliseconds>(t_end-t_begin).count()
    << "ms)" << endl;
  // Season's Greetings by Tangent AD ...
  cout << "Adjoint AD wishes ";
  // start time measurement
  t_begin=system_clock::now();
  // compute gradient in adjoint mode
  dfdx_a1<N>(x);
  // finish time measurement
  t_end=system_clock::now();
  // report run time
  cout << "! (taking "
    << duration_cast<milliseconds>(t_end-t_begin).count()
```

```
    << "ms)" << endl;
  return 0;
}
```

Five consecutive value pairs from

$$(101\ 77\ 114\ 114\ 32\ 121\ 109\ 88\ 115\ 97)^T$$

are distributed uniformly over an otherwise zero vector $\mathbf{x} \in \mathbb{R}^N$. The elapsed run times of the computation of the gradient in tangent and adjoint modes is measured for given even $N \geq 10$.

1.3 The Run Times

Both alternatives wish Merry Xmas!

The code is compiled using `g++ -O3` on our laptop. Leading zeros in $\mathbf{x}$ yield zero entries in the gradient. Their output to std::cout has no visible effect due to interpretation as empty 0-terminated C-strings.

Tangent mode takes more than six seconds to complete the task for $N = 231224$. Adjoint mode gets the job done without noticeable delay. You are encouraged to let your own experiments illustrate the $\mathcal{O}(N)$ growth of this gap in run time.

2 AD wishes Happy 2024!

2.1 The Maths

The Hessian of the function $f : \mathbb{R}^N \to \mathbb{R} : y = f(\mathbf{x})$,

$$y = \frac{1}{6} \cdot \sum_{i=0}^{N-1} x_i^3 \ ,$$

where $\mathbf{x} = (x_j)_{j=0}^{N-1}$ and $N \geq 0$, is easily found to be equal to

$$f''(\mathbf{x}) = (h_{j,i})_{j=0,\ldots,n-1}^{i=0,\ldots,n-1}, \ \text{ where } \ h_{j,i} = \begin{cases} x_i & i = j \\ 0 & i \neq j \end{cases} \ .$$

For example,

$$f''(\mathbf{x}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

at $\mathbf{x} = (1\ 2\ 3\ 4)^T$.

A *second-order tangent* version

$$f^{(1,2)} : \mathbb{R}^N \times \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R} : y^{(1,2)} = f^{(1,2)}(\mathbf{x}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)})$$

of f computes

$$y^{(1,2)} \equiv \mathbf{x}^{(1)^T} \cdot f''(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

without explicit accumulation of the Hessian. As in [2], we denote tangents due to the application of AD to f by the superscript $^{(1)}$. Reapplication of tangent AD to $f^{(1)}$ appends the superscript $^{(2)}$ to the respective variable and function names. Sequences of superscripts are fused into $^{(1,2)} \equiv {}^{(1)^{(2)}}$.

A dense Hessian requires $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ to range independently over the Cartesian basis vectors in $\mathbb{R}^N$. Potential symmetry should be exploited. $f^{(1,2)}$ needs to be evaluated $\frac{N(N+1)}{2}$ times in this case. In the given special case the diagonal structure of f'' allows for its accumulation with only N evaluations of $f^{(1,2)}$ by letting $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ range simultaneously over the Cartesian basis vectors in $\mathbb{R}^N$. For example,

$$\mathbf{e}_2^T \cdot f''(\mathbf{x}) \cdot \mathbf{e}_2 = \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = 3$$

at $\mathbf{x} = (1\ 2\ 3\ 4)^T$.

A *second-order adjoint* version

$$f_{(1)}^{(2)} : \mathbb{R}^N \times \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}^N : \mathbf{x}_{(1)}^{(2)} = f_{(1)}^{(2)}(\mathbf{x}, y_{(1)}, \mathbf{x}^{(2)})$$

of f computes

$$\mathbf{x}_{(1)}^{(2)} \equiv y_{(1)} \cdot f''(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

without prior accumulation of the Hessian. As in [2], we denote adjoints due to the application of AD to f by the subscript $_{(1)}$. Subsequent application of tangent AD to $f_{(1)}$ appends the superscript $^{(2)}$ to the respective variable and function names.

Set $y_{(1)} = 1$ and let $\mathbf{x}^{(2)}$ range over the Cartesian basis vectors in $\mathbb{R}^N$ to accumulate the Hessian column-wise by N evaluations of $f_{(1)}^{(2)}$. For example,

$$1 \cdot f''(\mathbf{x}) \cdot \mathbf{e}_2 = 1 \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 3 \\ 0 \end{pmatrix}$$

at $\mathbf{x} = (1\ 2\ 3\ 4)^T$.

Direct compression [3] of a diagonal Hessian exploits the fact that all columns are structurally orthogonal as no two of them contain nonzero entries in the same row. The sum of all columns amounts to the diagonal. It is obtained by a single evaluation of $f_{(1)}^{(2)}$ with all entries of $\mathbf{x}^{(2)}$ set equal to one. For example,

$$1 \cdot f''(\mathbf{x}) \cdot \sum_{i=0}^{N} \mathbf{e}_i = 1 \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

at $\mathbf{x} = (1\ 2\ 3\ 4)^T$.

2.2 The Code

Consider the following implementation of f in C++.

```
template<int N, typename T>
void f(T x[], T &y) {
  y=0;
  for (int i=0;i<N;++i) {
    y+=pow(x[i],3);
  }
  y/=6;
}
```

Tangent AD yields $f^{(1)}$ as follows:

```
// first-order tangent version of f
template<int N, typename T>
void f_t1(T x[], T x_t1[], T &y_t1) {
  // initially vanishing derivative of result
  y_t1=0;
  for (int i=0;i<N;++i) {
    // tangent of cubic function; factor 3 pulled out
    y_t1+=pow(x[i],2)*x_t1[i];
  }
  // factor 3 applied
  y_t1/=2;
}
```

Analogously, adjoint AD yields $f_{(1)}$ as follows:

```
// first-order adjoint version of f
template<int N, typename T>
void f_a1(T x[], T y_a1, T x_a1[]) {
  for (int i=N-1;i>=0;--i) {
    // adjoint of cubic function; factor 1/6 pulled in
    x_a1[i]=pow(x[i],2)/2*y_a1;
  }
}
```

Application of tangent AD to $f^{(1)}$ yields the second-order tangent $f^{(1,2)}$ as follows:

```
// second-order tangent version of f
template<int N, typename T>
void f_t1_t2(T x[], T x_t1[], T x_t2[], T &y_t1_t2) {
  // initially vanishing derivative of result
  y_t1_t2=0;
```

```
  for (int i=0;i<N;++i) {
    // tangent of quadratic function; factor 1/2 applied
    y_t1_t2+=x[i]*x_t1[i]*x_t2[i];
  }
}
```

Analogously, application of tangent AD to $f_{(1)}$ yields the second-order adjoint $f_{(1)}^{(2)}$ as follows:

```
// second-order adjoint version of f
template<int N, typename T>
void f_a1_t2(T x[], T y_a1, T x_t2[], T x_a1_t2[]) {
  for (int i=N-1;i>=0;--i) {
    // adjoint of quadratic function; factor 1/2 applied
    x_a1_t2[i]=x[i]*y_a1*x_t2[i];
  }
}
```

Without exploitation of sparsity the second-order tangent function is called $\frac{N(N+1)}{2}$ times to compute the lower triangular submatrix of the Hessian

$$f''(\mathbf{x}) = \begin{pmatrix} 72 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 97 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 112 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 112 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 121 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 32 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 50 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 48 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 50 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 52 \end{pmatrix}.$$

The following driver computes the individual entries of the diagonal by letting $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ range simultaneously over the Cartesian basis vectors in $\mathbb{R}^N$.

```
// Diagonal Hessian in second-order tangent mode
template<int N, typename T>
void ddfdxx_t1_t2_sparse(T x[]) {
  // declare tangent variables
  T x_t1[N], x_t2[N], y_t1_t2;
  // prepare for Cartesian basis vectors e_i
  for (int i=0;i<N;++i) x_t1[i]=x_t2[i]=0;
  // loop over diagonal entries
  for (int i=0;i<N;++i) {
    // set both tangents of inputs equal to e_i
    x_t1[i]=x_t2[i]=1;
    // call second-order tangent version of f
    f_t1_t2<N>(x, x_t1, x_t2, y_t1_t2);
```

```
    // print i-th diagonal entry of Hessian
    std::cout << y_t1_t2;
    // prepare for next Cartesian basis vector
    x_t1[i]=x_t2[i]=0;
  }
}
```

The second-order tangent function is called N times.

Direct compression [3] in second-order adjoint mode yields the sum of all columns of the Hessian as follows:

```
// Column-compressed Hessian in second-order adjoint mode
template<int N, typename T>
void ddfdxx_a1_t2_compressed(T x[]) {
  // declare tangent and adjoint variables
  T x_t2[N], x_a1_t2[N], y_a1;
  // set all entries of tangent of input equal to one
  for (int i=0;i<N;++i) x_t2[i]=1;
  // set adjoint of output equal to one
  y_a1=1;
  // call second-order adjoint f
  f_a1_t2<N>(x,y_a1,x_t2,x_a1_t2);
  // print sum of all columns of Hessian
  for (int j=0;j<N;++j) std::cout << x_a1_t2[j];
}
```

A single call of the second-order adjoint function yields all diagonal entries of the Hessian.

All of the above is stored in the C++ header file f2.h. It is included into the following source of the entire program.

```
#include "f2.h"

#include <iostream>
#include <chrono>

int main() {
  using namespace std;
  using namespace std::chrono;
  // increase to compare run times of various approaches
  const int N=240101; // e.g., YYMMDD
  // code to be executed in integer arithmetic
  using T=char;
  // size of x yields squared number of Hessian entries
   T x[N];
  // pairs of custom input values distributed uniformly
  T xv[]={72,97,112,112,121,32,50,48,50,52};
  for (int i=0,j=0;i<N;++i) {
```

```
    if (((!(i%(N/5)))&&(j<10))) {
      x[i]=xv[j++];
      x[i+++1]=xv[j++];
    } else {
      x[i]=0;
    }
  }

  // Season's Greetings ...
  cout << "Second-Order Tangent AD wishes ";
  // start time measurement
  auto t_begin=system_clock::now();
  // diagonal of the Hessian in tangent of tangent mode
  ddfdxx_t1_t2_sparse<N>(x);
  // finish time measurement
  auto t_end=system_clock::now();
  // report run time
  cout << "! (taking "
    << duration_cast<milliseconds>(t_end-t_begin).count()
    << "ms)" << endl;
  cout << "Second-Order Adjoint AD wishes ";
  // start time measurement
  t_begin=system_clock::now();
  // diagonal of the Hessian in tangent of adjoint mode
  ddfdxx_a1_t2_compressed<N>(x);
  // finish time measurement
  t_end=system_clock::now();
  // report run time
  cout << "! (taking "
    << duration_cast<milliseconds>(t_end-t_begin).count()
    << "ms)" << endl;
  return 0;
}
```

Five consecutive value pairs from

$$(72\ 97\ 112\ 112\ 121\ 32\ 50\ 48\ 50\ 52)^T$$

are distributed uniformly over an otherwise zero vector $\mathbf{x} \in \mathbb{R}^N$. The elapsed run times of the computation of the diagonal of the Hessian in sparsity-aware second-order tangent and adjoint modes is measured for given $N \geq 10$.

2.3 The Run Times

Both alternatives wish Happy 2024!

On our laptop, sparse second-order tangent mode takes nearly ten seconds to complete this task for $N = 240101$. Direct compression in second-order adjoint

mode gets the job done almost immediately. Again, you are encouraged to let your own experiments illustrate the $\mathcal{O}(N)$ growth of this gap in run time.

3 Conclusion: Merry Xmas and Happy 2024!

References

[1] A. Griewank and A. Walther, *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation, Second Edition*, no. OT105 in Other Titles in Applied Mathematics, SIAM, 2008.

[2] U. Naumann, *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.*, no. SE24 in Software, Environments, and Tools, SIAM, 2012.

[3] A. Gebremedhin, F. Manne, and A. Pothen, *What color is your Jacobian? Graph coloring for computing derivatives*, SIAM Review, 47 (2005), pp. 629–705.