An Introduction to Automatic Differentiation in Eigen

Evan Drumwright*

Abstract

Automatic differentiation is a convenient way to compute derivatives that differs from *analytical differentiation*—which uses a human (or computer) to determine derivatives offline and then turn that output into programming code—and *numerical differentiation*, which uses the limit function definition of a derivative and needs to use only function evaluations. Numerical differentiation is slow if the function evaluation is slow, and is capable of only limited floating point accuracy for computing the derivatives. Analytical differentiation is tedious, slow, and error prone. Automatic differentiation bridges both worlds: it provides high numerical accuracy and avoids potential errors from analytic computation of derivatives. It does this by exploiting that computer programs execute sequences of elementary functions and arithmetic operations, and by using the "chain rule" from differential calculus.

This document focuses on the "unsupported" automatic differentiation module for the Eigen matrix/vector arithmetic and linear algebra library for C++. Eigen makes it relatively straightforward to integrate automatic differentiation for scalar types into your code; its support for matrices simplifies computation of gradient vectors and Jacobian and Hessian matrices as well. This document exists because the Eigen documentation on automatic differentiation is currently nonexistent.

I. Introduction

Let us start with a simple example of automatic differentiation (hereafter denoted "AutoDiff"):

```
1  #include <iostream>
2  #include <Eigen/Core>
3  #include <unsupported/Eigen/AutoDiffScalar.h>
4
5  template <class T>
6  T sin_T(const T& a) {
7    using std::sin;
8    return sin(a);
9  }
10
11 int main() {
12   typedef Eigen::AutoDiffScalar<Eigen::Matrix<double, 1, 1>> AScalar;
13   AScalar x = 3*M_PI/4.0;
14   x.derivatives()(0) = 1;
15   cout << "sin(x) = " << sin_T(x).value() << " for x = " << x.value() << endl;
16   cout << "d/dx sin = cos(" << x.value() << ") = " <<
17     sin_T(x).derivatives() << endl;
18 }
```

The output from running this code:

```
sin(x) = 0.707107 for x = 2.35619
d/dx sin = cos(2.35619) = -0.707107
```

which is as we should expect.

Some points of interest are listed below:

Line 7 This statement may be necessary for your compiler to distinguish between the AutoDiff'd `sin` function and the C++ standard library version.

Line 13 This sets the value of the independent variable, x, *at the point at which the derivative is to be evaluated.*

Line 14 This line is necessary to obtain the derivative of `sin` with respect to `x` on return from `sin_T`. I'll discuss below why it needs to be set to this seemingly arbitrary value.

* Toyota Research Institute, Palo Alto, CA, USA

Notice that `sin_T(x).value()` contains the output of the sine function evaluated at `x` while `sin_T(x).derivatives()` contains the derivative of the sine function with respect to `x`, evaluated at the current value of `x`.

II. How does AutoDiff work?

An examination of the header file `unsupported/Eigen/AutoDiff` reveals the functionality of AutoDiff:

```
EIGEN_AUTODIFF_DECLARE_GLOBAL_UNARY(sin,
  using std::sin;
  using std::cos;
  return Eigen::MakeAutoDiffScalar(sin(x.value()),x.derivatives() * cos(x.value()));
  )
```

Noting that the first argument to `MakeAutoDiffScalar` is the output of the function, we can guess that the second argument is the derivative of the function, illustrating the chain rule from calculus:
For independent variable `x`, `derivatives()` should be set to one.
Defining a function $\mathtt{x}(t)$ of independent variable t, for example:

$$y(t) = t^2 \tag{1}$$

we have (using the chain rule):

$$\frac{\mathrm{d}}{\mathrm{d}t}\sin y(t) = \cos y(t)\frac{\mathrm{d}}{\mathrm{d}t}y(t) \tag{2}$$

$$= \cos y(t) \cdot 2t \cdot \frac{\mathrm{d}}{\mathrm{d}t}t \tag{3}$$

$$= \cos t^2 \cdot 2t \cdot 1 \tag{4}$$

The following code implements this as an AutoDiff:

```
#include <iostream>
#include <Eigen/Core>
#include <unsupported/Eigen/AutoDiff>

using namespace std;

template <class T>
T sin_T(const T& a) {
return sin(a);
}

template <class T>
T y(const T& t) { return t*t; }

int main() {
  typedef Eigen::Matrix<double, 1, 1> Vector1d;
  typedef Eigen::AutoDiffScalar<Vector1d> AScalar;
  AScalar t = 3*M_PI/4.0;
  t.derivatives()(0) = 1;
  cout << "d/dt sin(y(t)) = " << sin_T(y(t)).derivatives() << " (according to AutoDiff)" <<
    endl;
  cout << "d/dt sin = cos(" << t.value() << "^2) * 2 * " << t.value() << " = " <<
    std::cos(t.value() * t.value()) * t.value() * 2 << endl;
}
```

and produces the output:

```
d/dt sin(y(t)) = 3.50673 (according to AutoDiff)
d/dt sin(y(t)) = d/dt sin(t^2) = 2.35619^2 = 3.50673
```

If you examine the intermediate output of `sin_T(y(t))`, you will see that `y(t).value()` $= (3\pi/4)^2$ and `y(t).derivatives()` $= 2 \cdot (3\pi/4)$.

I hope I have made the mechanism straightforward at this point: *set* `derivatives()(0)` $= 1$ *for the AutoDiff scalar representing your independent variable.* `derivatives()(0)` $\neq 1$ is indicative of a final result or of an intermediate result (in the context of composite functions, like in the example above). If you neglect to set the AutoDiff scalar representing your independent variable, you can expect its value to be zero, which is equivalent to specifying that the variable you would like to differentiate is a constant.

III. How would the template argument for an AutoDiff scalar be a vector? On computing gradients

Consider the following code example:

```
#include <iostream>
#include <functional>
#include <Eigen/Core>
#include <unsupported/Eigen/AutoDiff>

// Typedef for an AutoDiff scalar holding an n-dimensional vector of derivatives.
typedef Eigen::AutoDiffScalar<Eigen::VectorXd> AScalar;

// Typedef for a vector of two autodiff scalars (feature requires C++11).
template <class T> using Vector2 = Eigen::Matrix<T, 2, 1>;
typedef Vector2<Eigen::AutoDiffScalar<Vector2<double>>> XScalar;

// This function illustrates using automatic differentiation on a vector
// function using a single autodiff scalar.
AScalar sum_square(const XScalar& a) { return a(0)*a(0) + a(1)*a(1); }

int main() {
  XScalar b;
  b(0).value() = 2.0;
  b(1).value() = 3.0;
  b(0).derivatives()(0) = 1.0;   b(0).derivatives()(1) = 0.0;
  b(1).derivatives()(0) = 0.0;   b(1).derivatives()(1) = 1.0;
  AScalar a = sum_square(b);

  std::cout << "The output of (x^2 + y^2) at x=2, y=3 is: " << a.value() << std::endl;
  std::cout << "The gradient of (x^2 + y^2) at x=2, y=3 is: " << a.derivatives() << std::endl;
}
```

Examine the AutoDiff scalar output of `sum_square`, a vector-valued function, which is a scalar with a vector of derivatives. Clearly the "scalar" and the derivatives need not be of identical dimension. Additionally, it is clear that `sum_square` needs two inputs.
This example also illustrates how to compute a gradient of a function.

IV. Generic functions

C++ allows generic functions to be written that accept either plain-old-data scalars or AutoDiff scalars. Using the function `sin_t` defined above:

```
...

int test() {
  typedef Eigen::AutoDiffScalar<Eigen::Matrix<double, 1, 1>> AScalar;
  const double theta = 3*M_PI/4.0;
  AScalar x = theta;
  cout << "sin(" << theta << ") = " << sin_T(theta) << " = " << sin_T(x).value() << endl;
}
```

Thhis generalization makes it relatively straightforward to convert already working code to using AutoDiff.

V. Computing second (and higher) derivatives

Computing second derivatives, and higher, is where reading and writing AutoDiff code becomes less intuitive. Such higher derivatives use recursive AutoDiff scalars, as illustrated in the code below (see `AScalar` and `DScalar`, where `DScalar` is used for computing a second derivative). Figure 1 shows the hidden code that is generated by the C++ compiler as it computes the second derivative within this function.

```
1  #include <iostream>
2  #include <functional>
3  #include <Eigen/Core>
4  #include <unsupported/Eigen/AutoDiff>
5
6  // Typedef for a vector of two autodiff scalars (feature requires C++11).
7  template <class T> using Vector1 = Eigen::Matrix<T, 1, 1>;
8
9  // Typedef for an AutoDiff scalar holding an scalar derivative.
10 typedef Eigen::AutoDiffScalar<Vector1<double>> AScalar;
11
12 // Typedef for a recursive AutoDiff scalar.
13 typedef Eigen::AutoDiffScalar<Vector1<AScalar>> DScalar;
14
15 template <class T>
16 T sin_T(const T& a) {
17   return sin(a);
18 }
19
20 int main() {
21   // Computes the second derivative of sin() evaluated at M_PI/4     2.35619...
22   DScalar a;
23   a.value() = M_PI/4.0;
24
25   // Compute and output first and second derivatives
26   a.derivatives()(0).value() = 1;
27   a.value().derivatives()(0) = 1;
28   std::cout << "d/dx sin(x) = cos(x) = " << sin_T(a).derivatives() << " for x = " << a.value()
       << std::endl;
29   std::cout << "d^2/dx^2 sin(x) = -sin(x) = " << sin_T(a).derivatives()(0).derivatives() << "
       for x = " << a.value() << std::endl;
30 }
```

The output of running this program is:

```
d/dx sin(x) = cos(x) = 0.707107 for x = 0.785398
d^2/dx^2 sin(x) = -sin(x) = -0.707107 for x = 0.785398
```

The statements on Lines 26 and 27 are critical to computing the second derivative. **Without these statements, the program produces the following output**:

```
d/dx sin(x) = cos(x) = 0.707107 for x = 0.785398
d^2/dx^2 sin(x) = -sin(x) = 0 for x = 0.785398
```

Additionally, if you compare this code to that which was presented before, you will notice a difference. We set `a.value().derivatives()(0) = 1` and `a.derivatives()(0).value() = 1` on Lines 26 and 27 rather than `a.derivatives()(0).derivatives()(0) = 1`. *The second derivative is retrieved from* `a.derivatives()0).derivatives()(0)`*, however.*

Semantics of the various fields in the recursion

Where the first-derivative possessed clear semantices for `AScalar::derivatives()` on entry (i.e., it is equivalent to the output from the chain rule, as described in §II), the semantics of the fields for the recursive AutoDiff scalar are less meaningful.

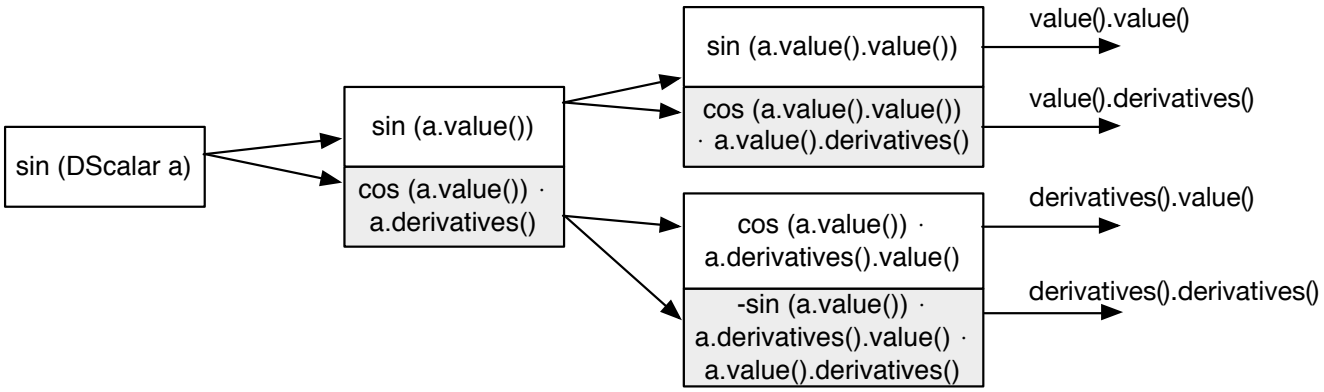

Fig. 1: C++ code generated by calling `sin` on a recursive AutoDiff (`DScalar`, defined in Section V). The middle box shows how templatizing the code in C++ generates `sin(AScalar)` and `cos(AScalar) * AScalar`, where `AScalar` is also defined in Section V.White boxes correspond to the `value()` "field", while gray boxes correspond to the `derivative` "field" returned at each level.

For computing the second derivative of non-composite function $f(x)$, where x is an independent variable: We assume below that the AutoDiff scalar is named `a`. "Input" shows the proper settings for `a` on entry and "Output" shows the outputs on return from the C++ implementation of `f(a)`.

	Input (`a`)	Output (`f(a)`)
`a.value().value()`	x	$f(x)$
`a.value().derivatives()`	1	$\frac{df}{dx}(x)$
`a.derivatives()(0).value()`	1	$\frac{df}{dx}(x)$
`a.derivatives()(0).derivatives()`	—	$\frac{d^2f}{dx^2}(x)$

For computing the second derivative of composite function $g(u)$ where $u = f(x)$ and x is an independent variable: We assume below that there are two AutoDiff scalars, named `a` and `b`. "Input" shows the proper settings for `a` and `b` on entry and "Output" shows the outputs on return from the C++ implementations of `f(a)` and `g(b)`.

	Input (`a`)	Output (`f(a)`)
`a.value().value()`	x	$f(x)$
`a.value().derivatives()`	1	$\frac{df}{dx}(x)$
`a.derivatives()(0).value()`	1	$\frac{df}{dx}(x)$
`a.derivatives()(0).derivatives()`	0	$\frac{d^2f}{dx^2}(x)$

Note that the zero input setting for `a.derivatives()(0).derivatives()` is not a misprint because $\frac{d^2}{dx^2}x = \frac{d}{dx}1 = 0$. We now consider `b` and `f(b)`:

	Input (`b`)	Output (`g(b)`)
`b.value().value()`	$u = f(x)$	$g(u)$
`b.value().derivatives()`	$\frac{du}{dx} = \frac{df}{dx}(x)$	$\frac{dg}{du}\frac{du}{dx}(x)$
`b.derivatives()(0).value()`	$\frac{du}{dx} = \frac{df}{dx}(x)$	$\frac{dg}{du}\frac{du}{dx}(x)$
`b.derivatives()(0).derivatives()`	$\frac{d^2u}{dx^2}(x)$	$\frac{d^2g}{dx^2}(x)$

The following code illustrates these concepts:

```
#include <iostream>
#include <functional>
#include <Eigen/Core>
#include <unsupported/Eigen/AutoDiff>
```

```
5
6  // Typedef for a vector of two autodiff scalars (feature requires C++11).
7  template <class T> using Vector1 = Eigen::Matrix<T, 1, 1>;
8
9  // Typedef for an AutoDiff scalar holding an scalar derivative.
10 typedef Eigen::AutoDiffScalar<Vector1<double>> AScalar;
11
12 // Typedef for a recursive AutoDiff scalar.
13 typedef Eigen::AutoDiffScalar<Vector1<AScalar>> DScalar;
14
15 template <class T>
16 T f(const T& x) {
17   return x*x*x;
18 }
19
20 template <class T>
21 T g(const T& u) {
22   return sin(u);
23 }
24
25 int main() {
26   // Computes the second derivative of g(f(x)) evaluated at x = 4
27   DScalar a;
28   a.value() = 4;
29
30   // Compute and output first and second derivatives from b = f(a)
31   a.derivatives()(0).value() = 1;
32   a.value().derivatives()(0) = 1;
33   DScalar b = f(a);
34   std::cout << "u = f(x) = x^3 = " << b.value().value() << " for x = " << a.value() << std::
       endl;
35   std::cout << "du/dx = d/dx x^3 = 3*x^2 = " << b.derivatives()(0).value() << " = " << b.value
       ().derivatives()(0) << " for x = " << a.value() << std::endl;
36   std::cout << "d^2u/dx^2 = d^2/dx^2 x^3 = 6*x = " << b.derivatives()(0).derivatives() << " for
        x = " << a.value() << std::endl;
37
38   // Compute and output first and second derivatives from g(b)
39   std::cout << "g(u) = sin(u) = " << g(b).value().value() << " for u = " << b.value().value()
       << std::endl;
40   std::cout << "dg/dx = cos(u) * du/dx (x) = " << g(b).value().derivatives()(0) << " = " << g(b
       ).derivatives()(0).value() << " at x = " << a.value() << std::endl;
41   std::cout << "d^2g/dx^2 = -sin(u) * (du/dx)^2 + cos(u) * d^2u/dx^2 (x) = " << g(b).
       derivatives()(0).derivatives()(0) << " at x = " << a.value() << std::endl;
42 }
```

The output of running this program is:

```
1 u = f(x) = x^3 = 64 for x = 4
2 du/dx = d/dx x^3 = 3*x^2 = 48 = 48 for x = 4
3 d^2u/dx^2 = d^2/dx^2 x^3 = 6*x = 24 for x = 4
4 g(u) = sin(u) = 0.920026 for u = 64
5 dg/dx = cos(u) * du/dx (x) = 18.8091 = 18.8091 at x = 4
6 d^2g/dx^2 = -sin(u) * (du/dx)^2 + cos(u) * d^2u/dx^2 (x) = -2110.34 at x = 4
```

Please verify for yourself that these calculations match.