DVDA Verifiably Differentiates Algorithmically

Greg Horn

December 15, 2011

Abstract

I wrote a Haskell library for symbolic manipulation and automatic differentiation. It is strongly influenced by the Theano [1] library in Python. DVDA is available at http://github.com/ghorn/dvda. This paper provides a brief explanation of automatic differentiation and its implementation in DVDA.

Contents

List of Figures

1 Algorithmic Differentiation

The fast, robust numeric minimization of smooth nonlinear $\mathbb{R}^n \rightarrow \mathbb{R}$ functions can often be dramatically improved when efficient, accurate gradient information is available. Hand-coding gradients can be time-consuming and a source of errors, and it prohibits rapid iteration. It is a huge productivity aid when an engineer only has to code up a function and a library automatically provides fast, exact gradients.

1.1 Numeric Differentiation

A very simple technique for algorithmically differentiating is numerical differentiation.

1.1.1 Forward Differencing

Taking the first two terms of a Taylor series

$$f(x_0 + h) = f(x_0) + f'(x_0)h + O(h^2) \tag{1}$$

and solving for $f'(x_0)$ yields

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + O(h), \tag{2}$$

a simple and well known gradient approximation formula with linear error in step size h.

1.1.2 Central Differencing

A formula with quadratic error is derived by taking an addition term in the Taylor series for $f(x_0 + h)$ and $f(x_0 - h)$:

$$f(x_0 + h) = f(x_0) + f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + O(h^3) \tag{3}$$

$$f(x_0 - h) = f(x_0) - f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + O(h^3) \tag{4}$$

Subtracting these equations cancels the second order term

$$f(x_0 + h) - f(x_0 - h) = 2f'(x_0)h + O(h^3) \tag{5}$$

and solving for the derivative yields

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + O(h^2) \tag{6}$$

Central differencing requires twice as many function evaluations as forward differencing.

Both finite and central difference formulas suffer from truncation error as the step size approaches 0 (Figure 1). A good implementation (e.g. [2]) will automatically choose step size, balancing convergence with truncation error.

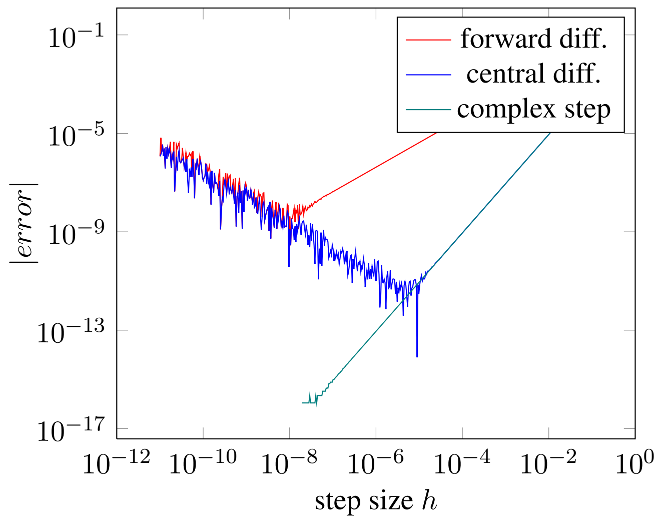


Figure 1: Truncation error in finite step approximations as step size h approaches 0

1.1.3 Complex Step

One last method worth mentioning is the complex step. Brought back to mainstream attention in [7], the complex step method has quadratic error in h and no truncation error. The Taylor series of $f(x_0 + ih)$ is taken:

$$\begin{aligned} f(x_0+ih) &= f(x_0) + f'(x_0)ih + \frac{1}{2}f''(x_0)(ih)^2 + O(h^3) && (7)\\ &= f(x_0) + f'(x_0)ih - \frac{1}{2}f''(x_0)h^2 + O(h^3) && (8) \end{aligned}$$

Taking the imaginary part and solving for $f'(x_0)$:

$$Im[f(x_0+ih)] = f'(x_0)h + O(h^3) \tag{9}$$

$$f'(x_0) = \frac{Im[f(x_0+ih)]}{h} + O(h^2) \tag{10}$$

yields an elegant approximation for the derivative.

1.1.4 Complexity

All of these finite step methods require $O[n]$ evaluations to compute the gradient of $f : \mathbb{R}^n \to \mathbb{R}$, because a separate step must be taken in each dimension, e.g. for forward step:

$$\nabla f(\vec{x_0}) \approx \frac{1}{h} \begin{bmatrix} f\left(\vec{x_0} + \begin{bmatrix} h \\ 0 \\ \vdots \\ 0 \end{bmatrix}\right) - f(\vec{x_0}) \\ f\left(\vec{x_0} + \begin{bmatrix} 0 \\ h \\ \vdots \\ 0 \end{bmatrix}\right) - f(\vec{x_0}) \\ \vdots \\ f\left(\vec{x_0} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ h \end{bmatrix}\right) - f(\vec{x_0}) \end{bmatrix} \tag{11}$$

1.2 Symbolic Differentiation

Symbolic differentiation refers to the process of differentiating a function symbolically with a library like SymPy [8] and then evaluating the result, often through code generation. For example the functions

```haskell
f :: Floating a => (a,a) -> a
f (x0,x1) = sin (x0*x1)

g :: Floating a => a -> (a,a)
g x = (sx, sx*x)
  where
    sx = sin x
```

would be transformed into

```haskell
df :: Floating a => (a,a) -> (a,a)
df (x0,x1) = (cos(x0*x1)*x1, cos(x0*x1)*x0)

dg :: Floating a => a -> (a,a)
dg x = (cos(x), cos(x)*x + sin(x))
```

This is inefficient because it does not eliminate common sub-expressions. What we really want is:

```
df :: Floating a => (a,a) -> (a,a)
df (x0,x1) = (cx0x1 * x1, cx0x1 * x0)
  where
    cx0x1 = cos (x0 * x1)

dg :: Floating a => a -> (a,a)
dg x = (cx, cx*x + (sin x))
  where
    cx = cos x
```

While an optimizing compiler will certainly do this internally, it is not guaranteed to catch every repeated operation in more complicated functions and compile times and memory consumption certainly suffer. Using symbolic differentiation on large functions leads to larger memory usage and slower speed in the source-to-source analytic differentiation, the compilation, and the (unoptimized) evaluation.

1.3 Automatic Differentiation

Automatic differentiation eliminates repeated operations by maintaining a dual number which is a tuple containing both the primal function value and the infinitesimal sensitivity or perturbation value. This dual number is propagated through evaluation using the chain rule.

1.3.1 Forward Mode

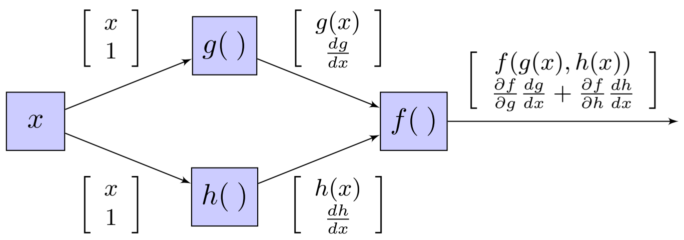

x
$g(\,)$
$h(\,)$
$f(\,)$
$\begin{bmatrix} x \\ 1 \end{bmatrix}$
$\begin{bmatrix} x \\ 1 \end{bmatrix}$
$\begin{bmatrix} g(x) \\ \frac{dg}{dx} \end{bmatrix}$
$\begin{bmatrix} h(x) \\ \frac{dh}{dx} \end{bmatrix}$
$\begin{bmatrix} f(g(x),h(x)) \\ \frac{\partial f}{\partial g}\frac{dg}{dx} + \frac{\partial f}{\partial h}\frac{dh}{dx} \end{bmatrix}$

Figure 2: Dual variables in a FAD expression graph

The forward mode of operation is suitable for differentiating functions of $\mathbb{R} \to \mathbb{R}^m$, and is elegantly implemented and readily available [5]. Rigorous treatments are given in [4] and [6].

A dual data type is implemented and operations are overloaded using analytic derivative formulas:

```
data Dual a = Dual a a deriving (Show, Eq)

instance Num a => Num (Dual a) where
  (Dual x x') * (Dual y y') = Dual (x * y) (x*y' + x'*y)
  (Dual x x') + (Dual y y') = Dual (x + y) (x' + y')
  (Dual x x') - (Dual y y') = Dual (x - y) (x' - y')
  negate (Dual x x') = Dual (-x) (-x')
  fromInteger x = Dual (fromInteger x) 0

instance Floating a => Floating (Dual a) where
  sin (Dual x x') = Dual (sin x) (cos x * x')
  -- etc.
```

A function and its derivatives are then evaluated simultaneously:

```
-- Evaluate R -> R^n function and it's derivitives
evalFunAndDerivs :: Num a => (a -> [a]) -> a -> [Dual a]
evalFunAndDerivs f x = f (Dual x 1)
```

Notice that each overloaded dual operation is a small constant factor more expensive than the corresponding primal operation. For example multiplication on a dual number is three floating multiplies and an add. The worst operation in Num/Fractional/Floating is logBase:

```
logBase (Dual b b') (Dual e e') = Dual primal pert
  where
    primal = logBase b e
    pert = (e'/e - primal*b'/b) / log b
```

which is still only a small constant factor slower than the primal operation.

This means that the added time and memory cost of evaluating the derivatives of an $\mathbb{R} \to \mathbb{R}^m$ function is independent of m, unlike in numerical or symbolic differentiation.

The computational graph of FAD on the previous example $g(x) = (sx, sx * x)$ with $sx = sin(x)$ is shown in Figure 3. The C source code generated by DVDA is:

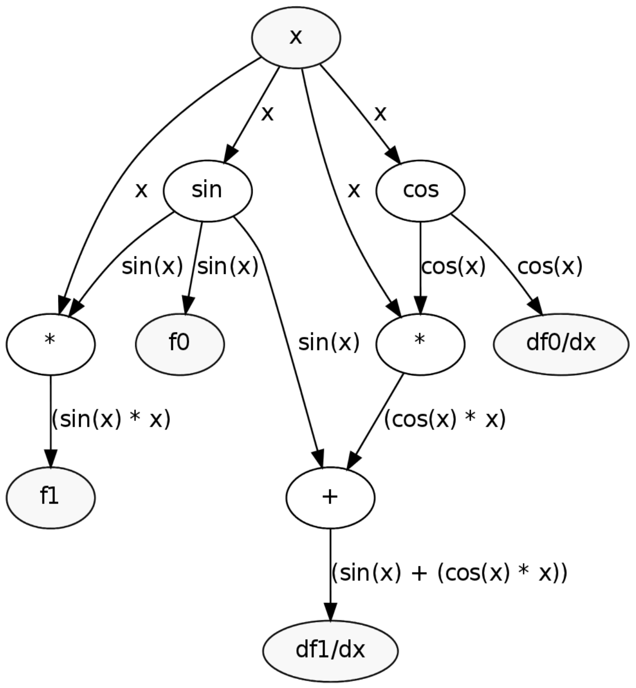
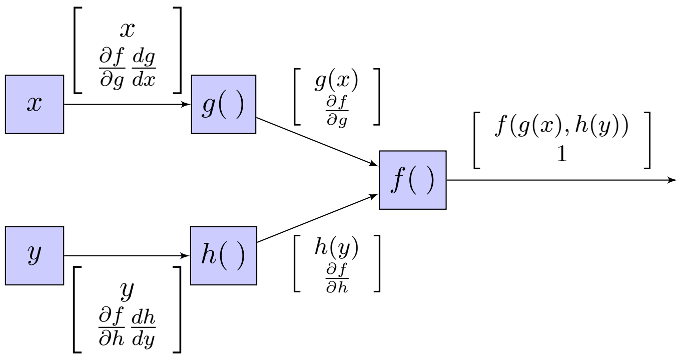


Figure 3: FAD computation of $(f_0, f_1, \frac{df_0}{dx}, \frac{df_1}{dx})$ for $f = (sin(x),\ x\ sin(x))$

```
// call_f6dba2fb379ec87fba2d70e0df484b71.c

#include "math.h"
#include "string.h"
#include "call_f6dba2fb379ec87fba2d70e0df484b71.h"

void call_f6dba2fb379ec87fba2d70e0df484b71(const double * const in[],
                                           double * const out[])
{
    // input declarations:
    const double x = *(in[0]);

    // body:
    const double t1 = x;
    const double t0 = sin(t1);
    const double t3 = t0 * t1;
    const double t5 = cos(t1);
    const double t8 = t5 * t1;
    const double t7 = t0 + t8;
    out[0][0] = t0; // out0, output node: 2
    out[1][0] = t3; // out1, output node: 4
    out[2][0] = t5; // out2, output node: 6
    out[3][0] = t7; // out3, output node: 9
}
```

1.3.2 Reverse Mode

The efficient differentiation of $\mathbb{R}^n \to \mathbb{R}$ functions is trickier, but much more useful for many optimization problems. A complete explanation is given in [4]. The *ad* library by Edward Kmett [3] implements both forward and reverse modes.



x
$\begin{bmatrix} x \\ \frac{\partial f}{\partial g}\frac{dg}{dx} \end{bmatrix}$
$g(\,)$
$\begin{bmatrix} g(x) \\ \frac{\partial f}{\partial g} \end{bmatrix}$
$\begin{bmatrix} f(g(x),h(y)) \\ 1 \end{bmatrix}$
$f(\,)$
y
$h(\,)$
$\begin{bmatrix} h(y) \\ \frac{\partial f}{\partial h} \end{bmatrix}$
$\begin{bmatrix} y \\ \frac{\partial f}{\partial h}\frac{dh}{dy} \end{bmatrix}$

Figure 4: Dual variables in a RAD expression graph

In reverse mode an expression graph is first built up in a "tape". The output is then seeded with unit sensitivity and sensitivities are backpropagated through the graph using the chain rule. Since each function node only applies the chain rule once, the reverse mode has the property that the additional cost of computing a gradient is independent of n.

The DVDA implementation of a tape is a symbolic expression:

```haskell
data Expr a = ENum a
            | EInt Int
            | ESym String
            | EUnary UnaryType (Expr a)
            | EBinary BinaryType (Expr a) (Expr a)
```

Partial example instances of Num/Floating are provided in Appendix A.

The unary and binary enums are defined:

```haskell
data UnaryType = Sin
               | Cos
               | Neg

applyUnary :: Floating a => UnaryType -> a -> a
applyUnary Neg = negate
applyUnary Sin = sin
applyUnary Cos = cos
-- etc.
```

```haskell
data BinaryType = Mul
                | Add
                | Div
                | Sub

applyBinary :: Floating a => BinaryType -> a -> a -> a
applyBinary Mul = (*)
applyBinary Add = (+)
applyBinary Div = (/)
applyBinary Sub = (-)
-- etc.
```

The backpropagation of sensitivities is then:

```haskell
-- backpropogate sensitivities
-- return list of symbolic variables with their sensitivities
getSensitivities :: Floating a => Expr a -> Expr a -> [(Expr a, Expr a)]
getSensitivities (EUnary unType g) sens =
  getSensitivities g (sens*dfdg)
  where
    dfdg = pert $ applyUnary unType (Dual g 1)
getSensitivities (EBinary binType g h) sens =
  getSensitivities g (sens*dfdg) ++ getSensitivities h (sens*dfdh)
  where
    dfdg = pert $ applyBinary binType (Dual g 1) (Dual h 0)
    dfdh = pert $ applyBinary binType (Dual g 0) (Dual h 1)
getSensitivities primal@(ESym _ _) sens = [(primal, sens)]
getSensitivities _ _ = []

-- get the perturbation part of the dual
pert :: Dual a -> a
pert (Dual _ x) = x
```

and finally reverse automatic differentiation is performed:

```haskell
-- Take the gradient of an expression w.r.t. list of inputs
rad :: Floating a => Expr a -> [Expr a] -> [Expr a]
rad expr args = map getXSens args
  where
    -- gradient w.r.t. ALL inputs in expression
    sens = getSensitivities expr 1

    -- filter inputs requested by user
    getXSens x = sum $ map snd $ filter (\y -> x == fst y) sens
```

This implementation does not have the property that gradient computation is the same order of time complexity as function computation, because the gradient is returned as a list of expression trees which

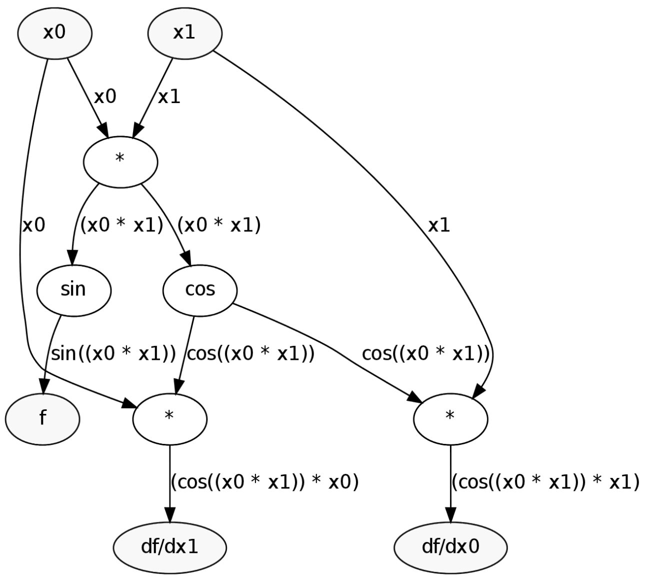


Figure 5: RAD computation of $(f, \frac{df}{dx_0}, \frac{df}{dx_1})$ for $f = sin(x_0 * x_1)$

require common sub-expression elimination. In practice DVDA performs CSE quickly and the generated C code recovers this crucial property.

The computational graph of FAD on the previous example $f(x) = sin(x_0 * x_1)$ is shown in Figure 5. The C source generated by DVDA is:

```
// call_5aa20fed1439bec521f9a7ef42485a12.c

#include "math.h"
#include "string.h"
#include "call_5aa20fed1439bec521f9a7ef42485a12.h"

void call_5aa20fed1439bec521f9a7ef42485a12(const double * const in[],
                                             double * const out[])
{
    // input declarations:
    const double x0 = *(in[0]);
    const double x1 = *(in[1]);

    // body:
    const double t2 = x0;
    const double t3 = x1;
    const double t1 = t2 * t3;
    const double t0 = sin(t1);
    const double t6 = cos(t1);
    const double t5 = t6 * t3;
    const double t8 = t6 * t2;
    out[0][0] = t0; // out0, output node: 4
    out[1][0] = t5; // out1, output node: 7
    out[2][0] = t8; // out2, output node: 9
}
```

References

[1] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[2] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. GNU Scientific Library Reference Manual (3rd Edition). http://www.gnu.org/software/gsl/.

[3] E. A. Kmett. ad. http://hackage.haskell.org/package/ad/, 2010-2011.

[4] B. A. Pearlmutter and J. M. Siskind. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30:7:1–7:36, March 2008.

[5] B. A. Pearlmutter and J. M. Siskind. fad. http://hackage.haskell.org/package/fad/, 2008-2009.

[6] J. M. Siskind and B. A. Pearlmutter. Nesting forward-mode ad in a functional framework. *Higher Order Symbol. Comput.*, 21:361–376, December 2008.

[7] W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Rev.*, 40:110–112, March 1998.

[8] SymPy Development Team. *SymPy: Python library for symbolic mathematics*, 2011.

A Num/Floating instances for Expr

This is a partial implementation, showing how the identities $+0$ and $*1$ are pruned, and how $*0$ is simplified:

```haskell
instance Num a => Num (Expr a) where
  -- simplifications
  negate (EInt 0) = EInt 0

  (ENum x) + (ENum y) = ENum (x + y)
  (EInt x) + (EInt y) = EInt (x + y)
  (EInt 0) + y = y
  x + (EInt 0) = x

  (ENum x) - (ENum y) = ENum (x - y)
  (EInt x) - (EInt y) = EInt (x - y)
  (EInt 0) - y = negate y
  x - (EInt 0) = x

  (ENum x) * (ENum y) = ENum (x * y)
  (EInt x) * (EInt y) = EInt (x * y)
  (EInt 1) * y = y
  x * (EInt 1) = x
  (EInt 0) * y = 0
  x * (EInt 0) = 0

  -- normal operations
  fromInteger = EInt
  negate x = EUnary UnaryType Neg x
  x + y = EBinary BinaryType Add x y
  x - y = EBinary BinaryType Sub x y
  x * y = EBinary BinaryType Mul x y

instance (Floating a) => Floating (Expr a) where
  sin x = EUnary UnaryType Sin x
  cos x = EUnary UnaryType Cos x

-- evaluate (Expr a) to a numeric value
eval :: Floating a => Expr a -> [a]
eval (ENum x) = x
eval (EInt x) = fromIntegral x
eval (EUnary unType x) = (applyUnary unType) (eval x)
eval (EBinary binType x y) = (applyBinary binType) (eval x) (eval y)
```