Gradients via Reverse Accumulation

John Mount*

July 14, 2010

Abstract

In our earlier article Automatic Differentiation with Scala[1] we demonstrated how to use Scala to compute the gradient of function (a step useful in solving optimization problems). In this article[2] outline an often faster automatic differentiation technique called reverse accumulation.

Contents

1 Introduction

Consider the simple (but silly) function given by Equation 1:

$$f(v_1, v_2, v_3) = (\sin(v_1) - \tan(v_2))^2 / v_3 \tag{1}$$

This equation represents a simple function of the three variables v_1, v_2, v_3 and will serve as our example problem. What we want to do is compute the gradient of this function which is the vector: $(\frac{\partial f()}{\partial v_1}, \frac{\partial f()}{\partial v_2}, \frac{\partial f()}{\partial v_3})$ indicating the direction of most rapid decrease in $f()$ as we change the values of the vs.

This function is deliberately chosen to be trivial (something we could punch-in on a calculator). More realistic problems would include cost of complicated wiring diagrams (as discussed in our previous article), maximum likelihood inference of statistical parameters, minimization of energy of molecular confirmations, training of neural networks, solutions of generalized linear models and so on. The ability to efficiently estimate gradients is a powerful method that allows one to bring a number of other tools (such as conjugate gradient based optimizers) to bear on problems.

*email: mailto:jmount@win-vector.com web: http://www.win-vector.com/

[1]URL: http://www.win-vector.com/blog/2010/06/automatic-differentiation-with-scala/

[2]URL: http://www.win-vector.com/dfiles/ReverseAccumulation.pdf

2 Straightforward Coding

What we want is a system that given a function can automatically produce an implementation of the following Scala trait (that declares both how to calculate the function and its gradient):

```
trait GFunction {
  def apply(x:Array[Double]):Double                      // f(x)
  def gradEval(x:Array[Double]):(Double,Array[Double]) // f(x) gradient(f(x))
}
```

We can achieve a very fast numeric estimate of the gradient with code like the following:

```
class NumDiff(f:Array[Double]=>Double) extends GFunction {

  def apply(x: Array[Double]): Double = f(x)

  def gradEval(x: Array[Double]):(Double,Array[Double]) = {
    val xdim = x.length
    val x2 = copy(x)
    val grad = new Array[Double](xdim)
    val delta:Double = 1.0e-6
    val fbase = f(x)
    for(i <- 0 until xdim) {
      val xorig = x(i)
      x2(i) = x(i) + delta
      val fplus = f(x2)
      x2(i) = xorig
      grad(i) = (fplus-fbase)/delta
    }
    (fbase,grad)
  }
}
```

This trick of computing the function $n+1$ times to form divided difference (where n is the dimension of vectors we are working with) often vastly out-performs non-gradient based optimization. In our article on automatic differentiation we introduced the dual numbers which are written as pairs (a, b) and have the property that for a function properly extended from regular numbers to the dual numbers we have:

$$f((a,b)) = (f(a), bf'(a))$$

So we can read-off exact[3] derivatives and gradients by evaluating the function n times over the dual numbers.

All of the techniques given above are very similar and rely on an idea called "forward accumulation", which we discuss in the next section.

[3] To the limits of machine arithmetic.

3 Forward Accumulation

In forward accumulation we compute derivatives and gradients of a function by using partial results from earlier stages in evaluating the function. We demonstrate this by converting Equation 1 into a notation called a "straight line program." A straight line program is an ordered sequence of extremely simple steps (such as adding or multiplying two numbers). The straight line program for Equation 1 is given below:

$$
\begin{array}{lcll}
v_1 & & \rightarrow & 3.1 \\
v_2 & & \rightarrow & -2.2 \\
v_3 & & \rightarrow & 2.0 \\
v_4 & = \sin(v_1) & \rightarrow & 0.04158 \\
v_5 & = \tan(v_2) & \rightarrow & 1.374 \\
v_6 & = v_4 - v_5 & \rightarrow & -1.332 \\
v_7 & = v_6 \times v_6 & \rightarrow & 1.775 \\
v_8 & = v_7/v_3 & \rightarrow & 0.8874
\end{array}
$$

This straight line program has 8 steps. Each step is the assignment of a value to new variable (v_1 through v_8). The first three variables v_1, v_2, v_3 represent our original inputs and the final variable v_8 represents the function result. So our example straight line program shows the steps in calculating $f(v_1, v_2, v_3)$ at the point $(3.1, -2.2, 2.0)$. In this notation the gradient we desired is written as: $(\frac{\partial v_8}{\partial v_1}, \frac{\partial v_8}{\partial v_2}, \frac{\partial v_8}{\partial v_3})$.

We introduce a table notation to show all possible partial derivatives $\frac{\partial v_i}{\partial v_j}$ where the table row-label denotes the variable v_i and the table column label denotes the operator $\frac{\partial}{\partial v_j}$. In the table below we have indicated the 3 cells that represent the values we are interested in (all other cells left empty).

	$\frac{\partial}{\partial v_1}$	$\frac{\partial}{\partial v_2}$	$\frac{\partial}{\partial v_3}$	$\frac{\partial}{\partial v_4}$	$\frac{\partial}{\partial v_5}$	$\frac{\partial}{\partial v_6}$	$\frac{\partial}{\partial v_7}$	$\frac{\partial}{\partial v_8}$
v_1								
v_2								
v_3								
v_4								
v_5								
v_6								
v_7								
v_8	$\frac{\partial v_8}{\partial v_1}$	$\frac{\partial v_8}{\partial v_2}$	$\frac{\partial v_8}{\partial v_3}$					

We don't know the values of these 3 cells so we just filled them in with their symbolic notations (as given from the row and column labels). A natural question is- what cells do we know? The table below shows what is "obvious": each variable depends on itself in a 1 to 1 manner, each input variable is treated as being independent of all other input variables and because our straight line program does not allow forward references $\frac{\partial v_i}{\partial v_j} = 0$ if $j > i$:

	$\frac{\partial}{\partial v_1}$	$\frac{\partial}{\partial v_2}$	$\frac{\partial}{\partial v_3}$	$\frac{\partial}{\partial v_4}$	$\frac{\partial}{\partial v_5}$	$\frac{\partial}{\partial v_6}$	$\frac{\partial}{\partial v_7}$	$\frac{\partial}{\partial v_8}$
v_1	1	0	0	0	0	0	0	0
v_2	0	1	0	0	0	0	0	0
v_3	0	0	1	0	0	0	0	0
v_4				1	0	0	0	0
v_5					1	0	0	0
v_6						1	0	0
v_7							1	0
v_8								1

Unfortunately none of the values we want to know were filled in with "obvious" values. But we can look at this set-up as a dynamic programming table. We know some boundary values of the table and we just need an incremental scheme to fill out more of the table until we reach all of the values we are interested in.

The standard way to fill out this table is called "forward accumulation" and it is very close to the calculations implemented by the divided differences trick mentioned before and to the dual number calculation. The idea is explained by the "chain rule" from calculus which is of the form:

$$\frac{\partial}{\partial x} f(y) = f'(y) \frac{\partial y}{\partial x}$$

Or in words: "the partial derivative of f(y) with respect to x is the derivative of f() evaluated at y multiplied by the partial derivative of y with respect to x." This is all we need to calculate forward. For example we can compute $\frac{\partial v_5}{\partial v_2} = (1/\cos(-2.2)^2) \times \frac{\partial v_2}{\partial v_2} = 2.887 \times 1$ because we know $\tan(x)' = 1/\cos(x)^2$ and $\frac{\partial v_2}{\partial v_2} = 1$.

This sort of rule lets us work down any column in the table to get partial results:

	$\frac{\partial}{\partial v_1}$	$\frac{\partial}{\partial v_2}$	$\frac{\partial}{\partial v_3}$	$\frac{\partial}{\partial v_4}$	$\frac{\partial}{\partial v_5}$	$\frac{\partial}{\partial v_6}$	$\frac{\partial}{\partial v_7}$	$\frac{\partial}{\partial v_8}$
v_1	1	0	0					
v_2	0	1	0					
v_3	0	0	1					
v_4		0						
v_5		2.887						
v_6								
v_7								
v_8								

And a complete final result:

	$\frac{\partial}{\partial v_1}$	$\frac{\partial}{\partial v_2}$	$\frac{\partial}{\partial v_3}$	$\frac{\partial}{\partial v_4}$	$\frac{\partial}{\partial v_5}$	$\frac{\partial}{\partial v_6}$	$\frac{\partial}{\partial v_7}$	$\frac{\partial}{\partial v_8}$
v_1	1	0	0					
v_2	0	1	0					
v_3	0	0	1					
v_4	−0.9991	0	0					
v_5	0	2.887	0					
v_6	−0.9991	−2.887	0					
v_7	2.662	7.693	0					
v_8	1.331	3.847	−0.4437					

Yielding the desired gradient: $(1.331, 3.847, -0.4437)$.

To state the general rule we will introduce a bit of notation. Let $D_p[f()]$ represent the derivative of the function $f()$ with respect to its pth argument. In this notation we would write our example step as: $\frac{\partial v_5}{\partial v_2} = D_1[\cos(-2.2)] \times \frac{\partial v_2}{\partial v_2}$. And the chain rule as taught in calculus would be written as:

$$\frac{\partial v_i}{\partial v_j} = \sum_{\substack{(k,p) \\ j<k<i \\ v_i():p=v_k}} D_p[v_i()] \times \frac{\partial v_k}{\partial v_j}$$

where $v_i()$ is the function on the ith step of our calculation and $v_i() : p$ denotes the variable that is the pth position argument for the function $v_i()$. This notation is ugly, but it is needed to separate the partial derivatives (which we are looking up in our table) and the functional derivatives (which we would need to calculate). The sum indicates that we must sum across all nodes-k that are between node-i and node-j where node-j influences node-k's value and node-k influences node-i's value. Or we must take all indirect influences into account. Figure 1 shows how to compute $\frac{\partial v_8}{\partial v_1}$ using $\frac{\partial v_7}{\partial v_1}$. diagrammatically. To compute the influence of v_1 on v_8 we must find the minimal set of nodes directly influencing v_8 (in this case v_7) and then use our table to account for all the long-range interactions from v_1 to the cut nodes.

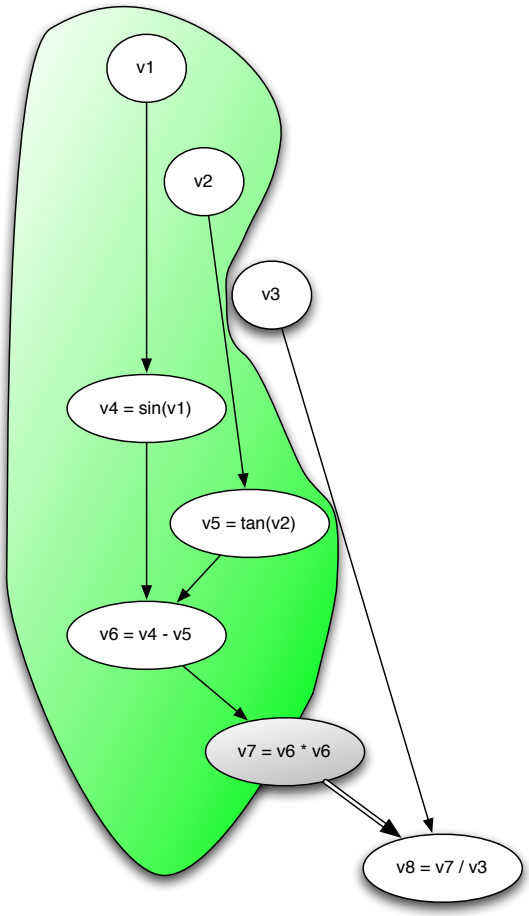

Figure 1: Forward Cut

Again, this seems complicated- but this is what is taught in calculus as the chain rule. In our table-notation it is simpler to say: compute down columns to the values you want.

4 Reverse Accumulation

The only problem with forward accumulation is that if we are working with a m step calculation over n input variables we fill out an $m \times n$ portion of the partial derivative table. Reverse accumulation limits itself to filling out only m cells of the table- no greater than the number of steps in the calculation.

The idea of reverse accumulation is again dynamic programming- but working back from the known value $\frac{\partial v_8}{\partial v_8}$ across a row.

To do this we bring in a slight variation on the standard chain rule:

$\frac{\partial v_8}{\partial v_7} = D_1[v_8()]\frac{\partial v_8}{\partial v_8} = (1/v_3) \times 1 = 0.5$

This step lets us work left in row-8 and fill in $\frac{\partial v_8}{\partial v_7}$ as pictured below.

	$\frac{\partial}{\partial v_1}$	$\frac{\partial}{\partial v_2}$	$\frac{\partial}{\partial v_3}$	$\frac{\partial}{\partial v_4}$	$\frac{\partial}{\partial v_5}$	$\frac{\partial}{\partial v_6}$	$\frac{\partial}{\partial v_7}$	$\frac{\partial}{\partial v_8}$
v_1								
v_2								
v_3								
v_4								
v_5								
v_6								
v_7								
v_8							0.5	1

We can continue right to left and complete the bottom row of the table with out need for any other rows:

	$\frac{\partial}{\partial v_1}$	$\frac{\partial}{\partial v_2}$	$\frac{\partial}{\partial v_3}$	$\frac{\partial}{\partial v_4}$	$\frac{\partial}{\partial v_5}$	$\frac{\partial}{\partial v_6}$	$\frac{\partial}{\partial v_7}$	$\frac{\partial}{\partial v_8}$
v_1								
v_2								
v_3								
v_4								
v_5								
v_6								
v_7								
v_8	1.331	3.847	-0.4437	-1.332	1.332	-1.332	0.5	1

The general rule is given by:

$$\frac{\partial v_i}{\partial v_j} = \sum_{\substack{(k,p) \\ j<k\leq i \\ v_k():p=v_j}} D_p[v_k()] \times \frac{\partial v_i}{\partial v_k}$$

This rule is based on the idea of cutting variable away from the result by tracking direct links from variables and tracking indirect influence *forward* using a single row of our table. Figure 2 shows the cut used to calculate $\frac{\partial v_8}{\partial v_1}$ using $\frac{\partial v_8}{\partial v_4}$.

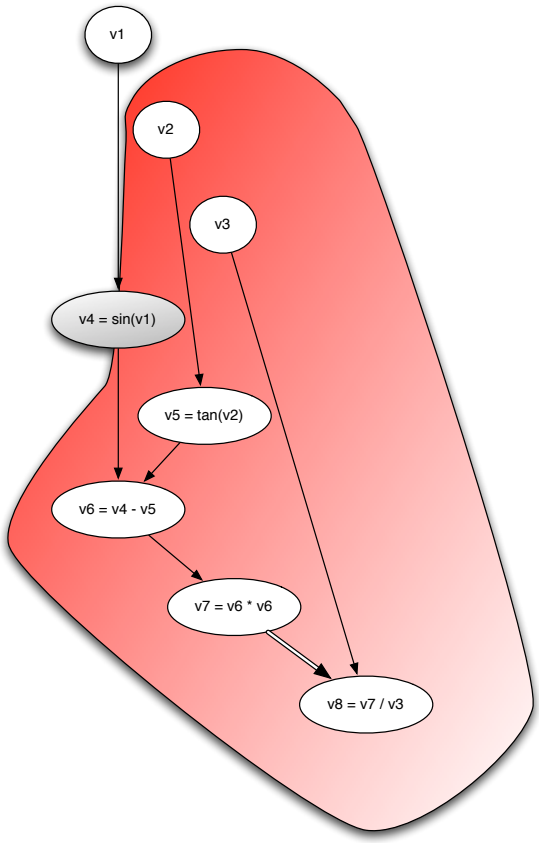

Figure 2: Reverse Cut

5 Implementation

To implement reverse accumulation you must capture an explicit representation of the straight line program representing your calculation. For forward automatic differentiation we could hide the bookkeeping with the dual numbers or by carrying gradient vectors forward. Capturing the calculation history takes memory and introduces overhead. In Figure 3 we show the time in MS to calculate a gradient for a wiring length problem in n-variables (n varying from 10 to 120). For each problem size we show the run time for four algorithms:

- fwd: Time for the dual number based forward differentiation to calculate the gradient.
- num: Time for the numeric divided difference method to calculate the gradient.
- rev: Time for the reverse accumulation method to calculate the gradient.
- sum: Time for special reverse accumulation method to calculate the gradient (discussed later).

Notice that all of the algorithms are significantly more expensive than the numeric method. However the numeric method is a somewhat weaker estimate of the gradient. Also the runtime of "rev" and "sum" do not include the time to capture the initial straight line

program. We timed re-evaluating an already captured straight-line program and gradient at a new point (which is correct as long as basic conditions such as the original calculation have no number driven if statements hold).

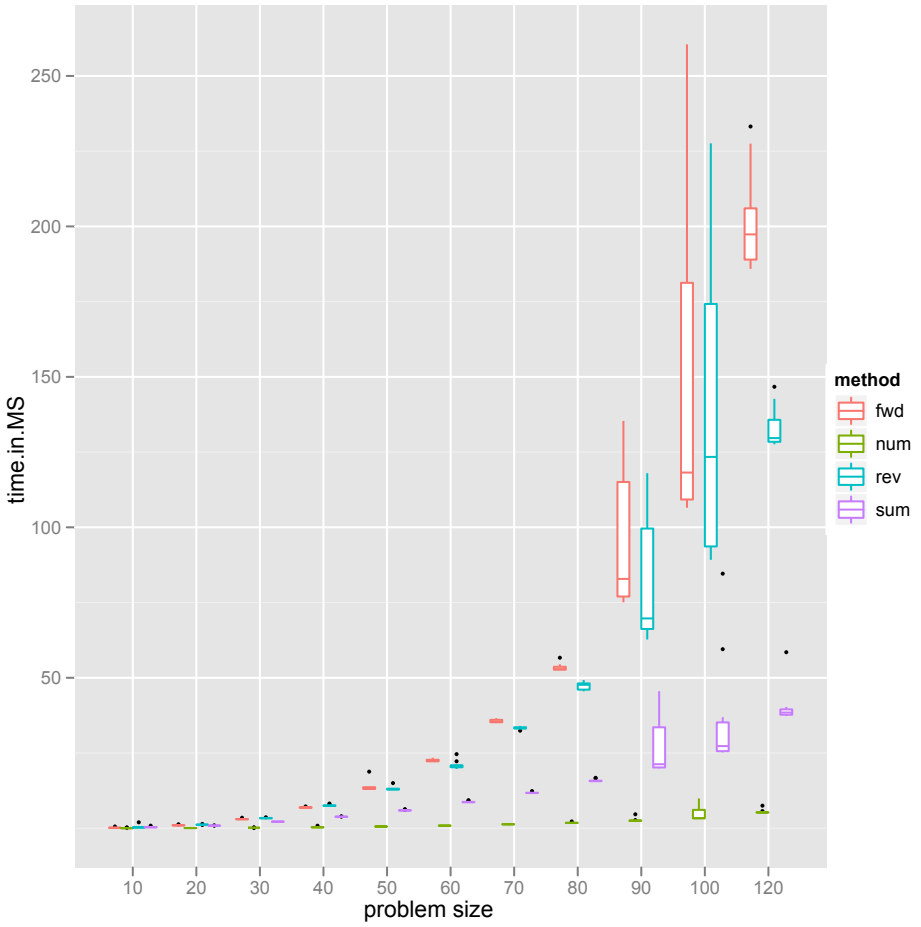

Figure 3: Runtimes for Forward, Numeric, Reverse and Summed Reverse differentiation

6 Improvements

Both the speed and memory footprint of reverse accumulation can be vastly improved for a very common class of functions. The function class is functions of the form:

$$f(x) = \sum_y g(x, y)$$

That is, functions that can be written as a sum. Many problems have this form:

- Our total wiring cost problem can be written as sum of point to point costs.
- Any maximum likelihood problem (like fitting a machine learning model) can be written has the sum of log-likelihoods over all data.

The trick is to notice that since $f()$ can be written as a sum, so can its gradient:

$$\nabla_x f(x) = \sum_y \nabla_x g(x, y).$$

So what we do is capture the much smaller straight line program for $g(x, y)$ where both x and y are treated as variables. Then, for a given x and every y, compute $g(x, y)$ and $\nabla_x g(x, y)$ and keep a running sum of these quantities. To declare such a function we ask the user to implement their function as an extension of the following trait:

```
abstract trait SummableFN {
  def apply[Y<:NumberBase[Y]](parameterArg:Array[Y],varyingArg:Array[Y]):Y
}
```

By convention we treat the first argument as the true parameters (the variables we wish a gradient with respect to) and the second argument as the values we will use data to sum-out. This notation is very succinct in that the user just specifies how to score one example and the system implicitly sums over all examples for both function and gradient calculation. The "sum" algorithm mentioned before is in fact this "sum over reverse accumulation" trick. For larger problems the lower complexity of this method eventually dominates the extra bookkeeping, indexing and object costs.

Figure 4 show the logarithm of the runtime in MS (to compress the range for viewability) for the standard numeric divided difference method "num" and the sum over reverse accumulation method "sum". The cross-over point is around 1000 variables. Above 1000 variables the sum over reverse accumulation method is reliably faster than the divided differences method. Given this is the era of numeric models over big data- this is exactly when we need this kind of improvement. This technique is essentially the famous "back-prop" algorithm from neural net training, but it is actually easier to discuss here where it has not been needlessly specialized over a single class of functions and with a single optimization strategy.

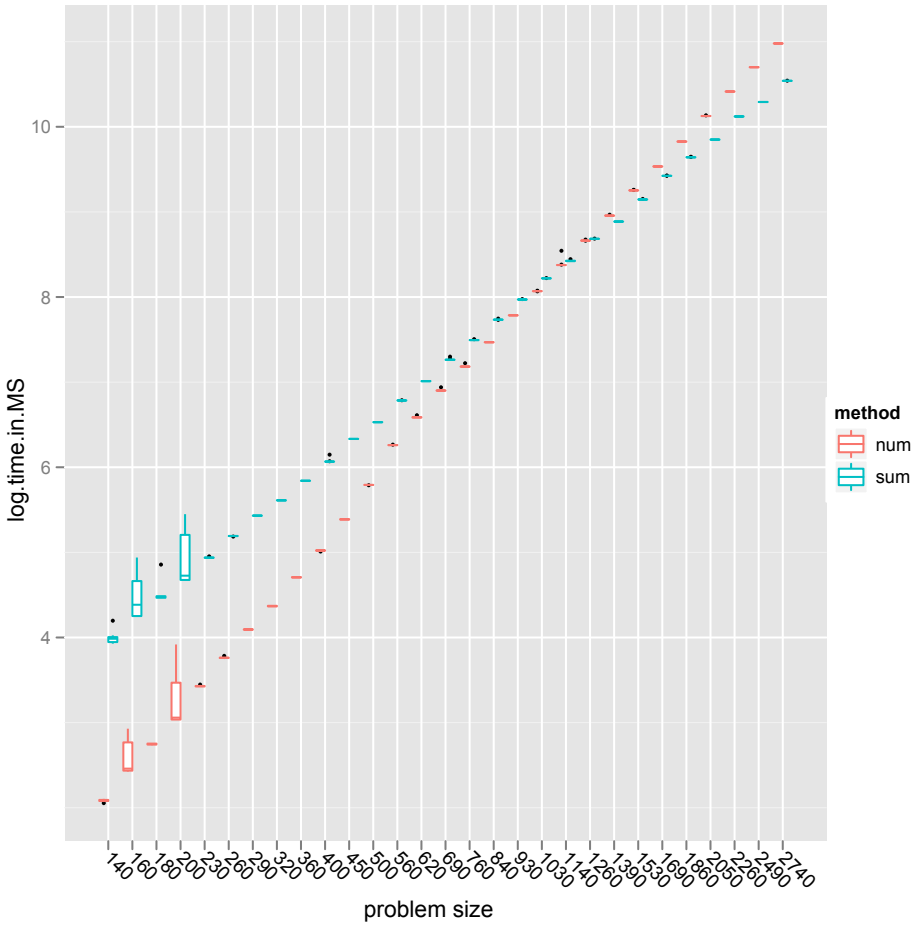

Figure 4: Runtimes for Numeric and Summed Reverse differentiation

7 Reference Code

As before we supply reference code in Scala: AutoDiff-0.3.jar[4], AutoDiff-0.3.advert[5], AutoDiff-0.3.sbp[6].

Note current code now distributed from: github.com/WinVector/AutoDiff[7].

The commands we use to build the system, print an example, get timings and run all the JUnit tests are given below:

```
scalac -g:none -optimise -classpath lib/junit-4.8.2.jar -d bin
   -sourcepath src:test ‘find src test -name \*.scala‘
scala -classpath bin com.winvector.demo.Print
JAVA_OPTS="-Xmx2g" scala -classpath bin:lib/junit-4.8.2.jar
   com.winvector.demo.GradSpeed
scala -classpath bin:lib/junit-4.8.2.jar com.winvector.test.TestAll test
```

[4]URL: http://www.win-vector.com/dfiles/AutoDiff-0.3.jar
[5]URL: http://www.win-vector.com/dfiles/AutoDiff-0.3.advert
[6]URL: http://www.win-vector.com/dfiles/AutoDiff-0.3.sbp
[7]URL: https://github.com/WinVector/AutoDiff