Algorithm Differentiation in Lisp: ADIL

Richard Fateman
Computer Science
University of California
Berkeley, CA, USA

September 14, 2014

Abstract

Algorithm differentiation (AD) is a technique used to transform a program F computing a numerical function of one argument $F(x)$ into another program $G(p)$ that returns a pair, $\langle F(p),\ F'(p)\rangle$ where by $F'(p)$ we mean the derivative of F with respect to its argument x, evaluated at $x = p$. That is, we have a program AD that takes as input a program, and returns another: $G := \text{AD}(F)$. Over the years AD programs have been developed to allow F to be expressed in some specialized variant of a popular programming language L (FORTRAN, C, Matlab, Python) and where G is delivered in that language L or some other. Alternatively, executing $F(p)$ is some environment will deliver $\langle F'(p),\ F(p)\rangle$ directly. AD tools have also been incorporated in computer algebra systems (CAS) such as Maple. A CAS is hardly necessary for the task of writing the AD program, since the main requirement is a set of tools for manipulation of an internal (typically tree) form of a program. In Lisp, a normal program is already in this form and so the AD program in Lisp (ADIL), the target F and the product G can all be expressed compactly in Lisp. In spite of the brevity and extensibility of the ADIL program, we can provide features which are unsupported in other AD programs. In particular, recursive functions are easily accommodated. Our perspective here is to point out that for scientists who write programs in Lisp or any language that can be converted to Lisp, AD is easily at hand.

1 Introduction to Automatic Algorithm Differentiation (AD)

For a complete introduction to the topic of Automatic or Algorithm Differentiation as used in this paper we recommend a visit to the website `www.autodiff.org`. Here you can find a collection of links to the standard literature, including AD systems and prominent applications. There are also links to the historical record of conference publications [6, 1] and books.

There are two major variation of AD techniques, forward and "reverse" differentiation. Although we have programmed both, for simplicity and brevity this paper deals only with the forward version.

1.1 A brief tangent

AD is not the same as "symbolic differentiation of algebraic expressions," a touch-stone task for symbolic languages such as Lisp. That is, the compact representation for a symbolic differentiation program is cited as one driving application for the original Lisp language design, and is often shown as an example in other symbolic languages. In a conventional setup for Lisp, $x \sin x \log x + 3$ would be written as `(+ (* x (sin x) (log x)) 3)`[1]. A brief program[2] can differentiate this with respect to `x` to get

```
(+ (* (* x (sin x) (log x))
      (+ (* 1 (expt x -1)) (* (* (cos x) 1) (expt (sin x) -1))
```

[1]If you are uncomfortable with this notation, many parsers from more "conventional" infix notation are available. The interface may be as simple as using a normal Lisp program text, but enclosing such infix expressions in markers, for example `[x*sin(x)*log(x)+3]`.

[2]see Appendix 1

```
        (* (* (expt x -1) 1) (expt (log x) -1))))
  0)
```

an answer which is correct but unsimplified and clumsy in appearance, even if it were converted to more conventional infix.

A proper "computer algebra system" or CAS would contain a (much longer) simplification program to take the symbolic result and reduce it to simpler terms, removing multiplications by 1 and additions with 0, perhaps finding common factors, etc.

1.2 An analogy in CAS

AD does not simulate the CAS feature just described. AD does not offer explicit symbolic differentiation. AD technology is far more analogous to another CAS feature: arithmetic with truncated Taylor series. AD converts a conventional program F treating each conventional scalar value u to a program G operating on a pair such as $p = \langle u,\ v\rangle$ representing that scalar value u and its derivative v with respect to an implicit parameter, say t. In particular, a constant has $v = 0$, and the parameter t has derivative $v = 1$. As a series, the pair p represents $s = u + v\,t + \cdots$. The equivalence to computation with Taylor series provides guidance as to how to compute with these pairs. (If there is any question, a CAS Taylor series program can generally provide an appropriate result.) The Taylor expansion of $\cos s$ is $\cos u - \sin u\, v\, t + \cdots$ and so $\cos p = \langle \cos u,\ -v \times \sin u\rangle$.

The extension of this idea to higher derivatives is straightforward although tedious by hand. Here is how it would work: we use triples, e.g. where the second derivative is called w: If $p = \langle u,\ v,\ w\rangle$, the operation of cos yields:

$$\cos p = \cos u - \sin u\, v\, t - \frac{\left(\cos u\, v^2 + 2\, \sin u\, w\right)\, t^2}{2} + \cdots$$

The result triple consists of picking the derivatives from the coefficients of different powers of t. $n!$ cancelled). An efficient program computing those coefficients naturally will need to compute $\sin u$ and $\cos u$ only once.

2 AD in Lisp

There are two fundamentally different approaches to building a forward AD system, and a third that looks plausible, at least briefly, for Lisp.

1. We can take a program P in (more-or-less) unchanged form and by *overloading* each of its operations make P executable in another domain in which scalars are replaced by pairs.

2. We can transform the source code of P into another program in which each line of code (or equivalent expression) is expanded into two lines (or equivalent) in a particular way so as to compute the original function and secondarily, the requested derivative.

3. A third technique, usually ignored, could be used in Lisp: write a new version of Lisp's `eval` to do AD. This is not a good idea, even in Lisp, because most Lisp programs don't use `eval`: they are compiled into assembler. The technique of overloading operators is conceptually similar to patching the `eval` program anyway, and so we discard this option.

We provide code for the first two approaches. Because it inherits all the Lisp facilities not specifically shadowed by AD variations, the overloading method has the advantage of covering just about everything one can do in Lisp. The overloading code is structured in a way that is fairly easy to understand; this overloading idea in various guises appears in many modern programming languages, sometimes called object-oriented programming.

Overloading is unfortunately not as efficient at run-time as the second method, which tends to run at a (small) multiple of the original code's speed. But source code transformation is ticklish, requiring attention to transformations for almost every aspect of the language. Thus our compiler-based version `dcomp` does not cover as much of Lisp as overloading. Both techniques can, fortunately, be used together. That is, a large program can be run mostly unchanged via overloading. If there is a bottleneck, some sub-part can be compiled via `dcomp` to reduce the run-time overhead. (Another variant of AD, reverse mode can in some situations be yet more efficient, and can also, in principle, be inserted in this way. An exposition of reverse mode in Lisp is given by Pearlmutter and Siskind [9].)

Regardless of the programming technique, one part of the task is to represent the pair as some kind of compound object, perhaps a `vector`, a `defstruct`, a Common Lisp Object System (CLOS) object, or just a list. Each would be easily distinguished from a conventional scalar should that be necessary, and each has space for a value and one (or possibly more) derivatives. For source-code transformation it may be adequate to just make near-duplicate names, e.g. if the original program has variable `v`, generate `v_diff_x` (etc.)

2.1 Overloading

Here we overload the arithmetic and numeric relational operators and let the rest of the language be inherited from Lisp's standard implementation. In ANSI standard Common Lisp, the accepted route to this is to establish a `package` to "shadow" the overloaded operations. We named the package `ga` for Generic Arithmetic, and can specify all the operations that are distinct as implemented for AD. We use the same base `ga` package for other arithmetic variants such as interval arithmetic, polynomial arithmetic, arbitrary-precision software floats, and combinations of such things[3]. All else is the same as in the standard `Common-Lisp` package.

The implementation of a consistent shadowing of Lisp arithmetic + and *, as well as comparisons is complicated by the fact that these operators can take any number of operands (including in some cases zero), in the "obvious" extension of the concept. Thus we wrote generic programs like `two-arg-*` for multiplying each possible pair of types, and then melded them together with an n-ary programs like `*`. Since the programs for each operator are very nearly the same, we wrote macro programs (program-writing programs) that helped us along. Thus to write the bulk of the "*" sub-programs we wrote `(defarithmetic *)`. The comparison and single-argument programs are easier. We include each of them by, for example `(defcomparison /=)` for the not-equal, and we wrote a program `r` to insert new information into the ADIL database. `(r sin (cos x))` encodes the needed information about sin and its derivative.

2.2 Using ADIL

Consider the function $f(x) = x \times \sin x \times \log x + 3$ written in Lisp as
`(defun f(x)(+ (* x (sin x) (log x)) 3))`.
We can evaluate f at a point p where p is the single-precision 1.23: `(f 1.23)` returns 3.2399836.

We can evaluate f and its first derivative at the point 1.23 by `(f(df 1.23 1.0))` where the second value 1.0 is the derivative of x with respect to x. This returns

```
<3.2399836, 1.2227035>
```

In view of the support for "generic" arithmetic in Lisp, we can compute `f` in double precision: `(f (df 1.23d0 1.0))` which returns

```
<3.23998349987768d0, 1.2227034313304448d0>
```

As illustrated above, in this system we define `df` as a constructor for a pair of numbers. This pair is displayed in angle-brackets. $p = \langle 1.23, 1.0 \rangle$. Note that the program `f` has not changed *at all*; applying the function to a different type of argument, and running the program within the `:ga` package is all that is needed.

A more interesting program is this one.

```
(defun s(x) (if (< (abs x) 1.0d-5) x
              (let ((z (s (* -1/3 x))))
                 (-(* 4 (expt z 3))
                   (* 3 z)))))
```

[3]requiring some delicacy in coercions between formats

While it is not obvious, this computes an approximation to $\sin x$ by applying an identity recursively. That is, $\sin x$ is a polynomial $4z^3 - 3z$ in $z = \sin(-x/3)$. For small enough x, $\sin x$ can be approximated by x. Let us try it out by typing `(s (df 1.23 1))`. We get

```
<0.942489, 0.33423734>
```

This not only provides a value for sin(1.23) but also computes 0.33423734, the second part of the pair, which happens to be close to cos(1.23). Closer values can be obtained by replacing the error bound 1.0d-5 with a smaller number, and using double-floats. ADIL, starting with `s`, ran the `s` program that approximated sin() and also ran a program that consequently computed cos() because it ran the *derivative of the program that computes sin.*

The classic recursive program in Lisp is factorial:
`(defun fact(x) (if (= x 1) 1 (* x (fact (1- x)))))`
which we modify to
`(defun fact(x) (if (= x 1) (df 1 0.422784335098d0) (* x (fact (1- x)))))`
whose base case is now the value one, with a peculiar derivative. We do this so as to make it correspond to the derivative of the Gamma function[4].

With this form we can provide not only the factorial but its "derivative" (at least at integer points).

In these examples we have not modified Lisp syntax at all. Anything *else* from Lisp is imported without attention or comment. Thus to compute and print ten $\sin x$ values, we can use the iterator `dotimes` as in `(dotimes (i 10) (print (s (df i 1))))`.

2.3 Newton Iteration, or, Why is AD useful?

It should be apparent so far that if we are given a complicated function $F: R \to R$ arranged as an expression, and all the sub-functions "cooperate" properly, we can feed in a pair $\langle c, 1\rangle$, representing the expression x and its derivative with respect to x, namely 1, each evaluated at $x = c$. We get out pairs $\langle f, f'\rangle = F(\langle c, 1\rangle)$ where the latter is the pair $f = F(c)$, and $f' = D_x(F(x))|_{x=c}$. This is sometimes exactly what we want in an application. These are discussed in papers available via `www.autodiff.org`. Here we take one of the simplest non-trival examples, and perhaps the easiest to motivate: the quadratically convergent Newton iteration.

To converge to a root in a Newton iteration for $f(z) = 0$ given an initial guess c_0 or $t_0 = \langle c_0, 0\rangle$, we compute $F(t_i) = \langle f(t_i), f'(t_i)\rangle$. Then the next iteration $t_{i+1} = t_i - f(t_i)/f'(t_i)$. If this is not sufficiently accurate we consider repeating with $\langle f, f'\rangle = F(t_{i+1})$, etc.

In this Newton iteration program, which by its nature must know that it is getting a value and derivative, we have used three functions particular to ADIL, namely `df-p`, a predicate which will return true if applied to a `df` object, as well as the two selection functions, `df-f` and `df-d` for extracting the value and derivative respectively from a `df` object.

The program is shorter than the explanation.

```
 Newton iteration: (ni fun guess) usage: fun is a
;; function of one argument guess is an estimate of solution of
;; fun(x)=0 output: a new guess. (Not a df structure, just a number)

(defun ni (f z) ;one Newton iteration step
  (let* ((pt (if (df-p z) z (df z 1)))  ; make sure init point is a df
         (v (funcall f pt))) ;compute f, f' at pt
   (df-f (- pt (/ (df-f v)(df-d v)))))) ; return the improved guess
```

As a simple example, consider $f(x) = \sin(1 + 2x)$ or
`(defun f(x)(sin(+ 1 (* 2 x))))`.
Then

[4]a continuous (for $x > 1$) version of factorial prominent among the "special functions" of physics. $\Gamma(n + 1) = n!$, The derivative of $\Gamma(2)$ is 0.422..

```
(setf h 2.0d0)      ;; just a guess
(setf h (ni 'f h))  ;; one step
(setf h (ni 'f h))  ;; another step
;; h converges in 6 steps to 4.21238898038469d0
```

A more useful version of Newton iteration might return the value of $f(h)$, too. Then the "residual" and the derivative can be taken into account in testing whether the Newton iteration has converged sufficiently. That program would look like:

```
(defun ni2 (f z)
  (let* ((pt (if (df-p z) z (df z 1)))
         (v (funcall f pt))) ;compute f, f' at pt
  (values
   (df-f (- pt (/ (df-f v)(df-d v)))) ;the next guess
    v))) ; the residual and derivative
```

;; A harder test for Newton iteration is this function,

```
(defun test(x)(+ (* 1/3 x) 1 (sin x))) ;; f(x) = x/3+1+sin x.
;; which is good if you start close enough to -0.8 or -3.2 or -5.4
;; but ni fails for some other initial guesses.
```

Finding zeros this way is a sometimes ticklish proposition because this iteration may not converge. There is a large literature on guaranteed root-finding procedures somewhat trading off speed and guaranteed convergence. A naive improvement on this method is to stop when two successive iterations are close in terms of relative or absolute error, or to halt if the number of iterations exceeds some bound.

This next program uses `ni2` which returns the value of the residual, and so we test this residual, or quit after some count is exceeded.

```
(defun run-newt2(f guess &key (abstol 1.0d-8) (count 18)) ;; Solve f=0
  ;; It looks only at the residual.
    (dotimes (i count  ;; do at most count times. failure prints msg
               (error "~%Newton quits after ~s iterations: ~s" count guess))
      (multiple-value-bind
           (newguess v)
           (ni2 f guess)
         (if (< (abs (df-f v)) abstol)
           (return newguess)
           (setf guess newguess)))))
```

2.4 What about speed?

The generic arithmetic (GA) system produces code that is slower than ordinary Lisp code, especially if that ordinary Lisp is "optimized"[5].

With appropriate instructions to the compiler, Common Lisp arithmetic can be compiled to "non-generic" straight-line code, say double-precision floating-point, comparable to that of other high-level languages.

How much of a speed difference is there? Partly this depends on the Lisp implementation in use, and the computer hardware. On some benchmarks that appear to heavily involve arithmetic computations, much of the time is spent figuring how to dispatch-to-the-right-method, using the generic arithmetic for ADIL. On one system we used for testing (Allegro CL 9.0 running on a Pentium 4) we found the generic dispatch was about a factor of 2 slower compared to "normal Lisp" and perhaps a factor of 5 slower than "optimized"

[5]Note that without type declarations and compilation, ordinary Lisp already has its own burden of generic arithmetic. Short and long (arbitrary length) integers, single- or double-floats, rationals, and complex numbers, as well as all plausible combinations are handled by standard ANSI Common Lisp. Indeed, code in the ADIL system using overloading also works for mixes of all these number types as much as in Common Lisp.

Lisp (where the optimized code was constrained to double-floats.) On tests where most of the work is done in numeric subroutine libraries such as sin or log, the difference is modest since the log routine takes the same time whether it is called from the `ga` package or from the `user` package. On tests where most of the operations are *other than arithmetic* such as looping over integer indexes, the `ga` programs mostly run at full speed because they are in fact mostly identical.

2.5 Source Code Transformation

As mentioned earlier, there is another part of ADIL. We wrote a program called `dcomp` that can compile programs, in-line, in a restricted language subset of Lisp. The subset is essentially that of functional-style arithmetic programs. In this situation, benchmarking a simple example suggests that the comparison between the ordinary Lisp for computing a function f and the ADIL Lisp for computing a function f and also its derivative is about a factor of two longer, which is about the best you would expect if you assume that evaluating the derivative is roughly similar in cost to the function itself.

This experiment computes $3 + z * (4 + z)$ where $z = \sin x$. In Lisp this looks like
`(defun kk(x) (setf z (sin x))(+ 3.0d0 (* z (+ 4.0d0 z))))`.
A nicer version binds `z` locally, as shown below. The `dcomp` version is shown with the `defdiff` defining form. All code is run through the Lisp compiler before timing.

```
;; each timing is a run of 100000 in a compiled loop



first, an optimized ordinary lisp double-float function.
 z = sin(x)
return 3+z*(4+z)

user:(defun kko(x &aux z)
        (declare (double-float x z) (optimize (speed 3)(safety 0)))
       (setf z (sin x))
       (+ 3.0d0 (* z (+ 4.0d0 z))))

The test:  (kk0 1.23d0),  100000 times.
time: 79 ms.

next, the same computation, done using generic arithmetic, but ultimately
executed in double float.  It suggests that using ga is 1.6X slower.

ga:   (defun kk(x &aux z) (setf z (sin x))(+ 3.0d0 (* z (+ 4.0d0 z))))

The test:  (kk 1.23d0), 100000 times.
time: 125 ms.

next, use kk on the derivative form  h=(df 1.23d0 1.0). The result is a pair,
value and derivative. It seems that this is about 5X slower than the fastest.
The test:  (kk h), 100000 times.
time: 407 ms.

next, compile a function that returns 2 values, (Lisp can do this) without
putting them in a pair. About 4.7X slower.


user: (defdiff kz(x)(progn (setf z (sin x))(+ 3.0d0 (* z (+ 4.0d0 z)))))
```

```
The test (kz 1.23d0), 100000 times.  Each call returns 2 values, function and derivative.
time: 375 ms
```

The body of `kz` can be examined by tracing an internal program `dc` which is the main program inside `dcomp`:

```
(lambda (g94)
       "(progn (setf z (sin x)) (+ 3.0d0 (* z (+ 4.0d0 z)))) wrt x"
       (declare (double-float g94))
       (declare (optimize (speed 3) (debug 0) (safety 0)))
       (let ((t102 0.0d0) (f101 0.0d0)
             (t100 0.0d0) (f99 0.0d0)
             (t98 0.0d0)  (f97 0.0d0)
             (z_DIF_x 0.0d0) (t96 0.0d0)
             (f95 0.0d0))
         (declare (double-float t102 f101 t100 f99 t98 f97 z_DIF_x t96 f95))
         (setf f95 (sin g94))
         (setf t96 (cos g94))
         (setf z f95)
         (setf z_DIF_x t96)
         (setf t98 0.0d0 f97 3.0d0)
         (setf t100 z_DIF_x f99 z)
         (setf t102 0.0d0 f101 4.0d0)
         (setf t102 (+ z_DIF_x t102))
         (setf f101 (+ z f101))
         (setf t100 (+ (* f101 t100) (* t102 f99)))
         (setf f99 (* f101 f99))
         (setf t98 (+ t100 t98))
         (setf f97 (+ f99 f97))
         (df f97 t98)))
```

What else can `dcomp` do? In addition to the usual arithmetic operations and built-in functions (sin, cos, log, etc.) as declared in the generic arithmetic, `dcomp` can handle `if, progn, setf`. Functions defined with `defdiff` take only scalar arguments, not `df` structures. They return only `df` structures. The derivative is always with respect to the first formal argument. Thus `(defdiff f(x y z)(cos (+ x y z)))` computes the derivative with respect to `x` only. It would be possible to have functions of many parameters and to specify the computation of partial derivatives with respect to some or all of the parameters, as with some other AD tools; as currently implemented, auxiliary parameters can be passed by lexical scope, or identified with global variables if appropriate. For example, if functions are operating on n by m matrices, presumably one is not interested in the "derivative with respect to n".

Initially we wrote a version of `dcomp` which systematically allowed recursion, but full support led to too much additional complexity and we removed it. A function like `f` above can be used directly by the generic arithmetic system, and if suitable, will be substantially faster. The `dcomp file` is surprisingly short, and takes full advantage of the rarely-used property of Lisp programs that such programs, prior to compiling, can be treated as data. The `dcomp file` is about 268 lines of text, including comments.

3 Why use Lisp?

For fans of Lisp, there is no question that one motivation is to show how easy it is to implement AD in Lisp. Lisp provides a natural representation for programs as data and a natural form for writing programs that write programs, which is what we do in ADIL. The code is short, and is in ANSI standard Common Lisp. This code is not "naive" – that is, it is not written in just the obvious idioms of introductory Lisp. It makes substantial use of macros and the object-oriented system. It illustrates, for those with only a cursory familiarity, that Lisp is more than `CAR` and `CDR`. For persons familiar with AD implementation elsewhere, a brief glance at the Lisp code shows that some of the ideas of AD can be expressed very concisely.

For those who care not a whit about Lisp or implementation strategies, you may prefer to refer to the online documentation for ADIFOR. I looked at 2.0 version D, some 99 pages, where one can read in detail how programs to be differentiated must be distinguished from ordinary FORTRAN (77) programs. Using ADIFOR requires declaring and marking program variables, adjusting numerous parameters, perhaps revising the FORTRAN in order to obey various restrictions, and following detailed guidelines on the use of these programs. The corresponding restrictions for ADIL and Lisp are quite modest, and certainly not 99 pages worth.

Since the Lisp programs are relatively short and entirely open to view and written in standard ANSI Common Lisp, they should run in any ANSI-conforming implementation. The flexibility one gets is apparent by looking at the code in which Lisp macro-definitions have made the addition of new derivative information simple. For example, to insert a new rule for differentiation of $\tanh x$ one adds the text`"tanh"` to the `shadow` list, and adds to the executable code in the file
```
(r tanh (expt (cosh x) -2)).
```
By comparison, other AD systems are relatively rigid, and require some effort to port, extend or optimize: the original designers must anticipate the spectrum of possible choices, perhaps freezing some choices, and then describe all the variants in detail. The user must then read the details, and hope there are no bugs. The user is unlikely to be able, in any case, to repair them. Some of the restrictions of the large systems seem to be arbitrary–perhaps a small point, but it did not occur to us to have to exclude recursive functions. Recursion is a feature lacking in ADIFOR. Would a typical programmer feel comfortable modifying Lisp code? Consider that a user of ADIL is perhaps not “typical” but is writing Lisp code to run through ADIL!

Finally, we wish to point out that contrary to common stereotypes, Lisp has been shown to be perfectly adequate for expressing numeric computation; some sophisticated compilers are available for generating efficient floating-point code.

3.1 Comments on the usefulness of AD

AD comes into the picture when computational scientists wish to use a large existing program P as a module in an optimization tool, or for sensitivity analysis. The optimization tool may require separate modules to compute values for functions and derivatives, which in the absence of AD, might require understanding P and writing and debugging a new program for the derivative PPRIME. AD promises to write PPRIME correctly with low intellectual effort, through tools like ADIFOR, for FORTRAN, and ADOL-C for C or C++.

The AD programs for FORTRAN, C, C++ generally work on “black box” programs in those languages, but not entirely automatically. They may reduce the programming effort to take a program and produce a “derivative” program considerably, say reducing the time from a year to a week. A visit to the previously mentioned `http://www.autodiff.org` website has links to some applications illustrating the level of human effort to build automation tools, document them for others to use, and then the continued effort and partnerships needed to use them. While the goal has been to differentiate nearly any program written in FORTRAN or C, it seems that some attention is required for success.

As for our own tools, we could, oddly enough, convert FORTRAN to Common Lisp using a program `f2cl` [2] and try to differentiate that, and `dcomp` could be modified to produce FORTRAN or C (etc.) code; this could be dropped into a FORTRAN (etc) source code file. But consider that a Lisp compiler will convert the ADIL results into assembly language without a stop-over in some other language.

There may be additional elaborations needed to make good use of ADIL. We hope that these will be motivated by applications of AD. We are confident that the structure we have set out is consistent with the fundamental needs for implementing forward-differentiation AD in Lisp. This system can naturally be loaded into a computer algebra system that is already written in Lisp, so that a computational scientist using a CAS may find Lisp surprisingly nearby.

To be realistic we should mention that a computational scientist may be simply unfamiliar with AD. There are conceptually simpler (but generally far less accurate and perhaps slower) ways of approximating derivatives of “programs” using finite differences. There is also a complex-step evaluation strategy [8], which takes advantage of the fact that for an analytic function f and an appropriately chosen small value of h, $\mathrm{Imag}(f(x+ih))/h$ is approximately $f'(x)$ For our earlier example, if we compute

`(imagpart(/ (f (+ 1.23d0 (complex 0 1.0e-8))) 1.0e-8))` we get 1.2227034316593401d0 which agrees to about 8 digits in the expected value. Choosing h may be ticklish.

Complicating *our* particular "sales pitch" is the need for a user of ADIL to, directly or indirectly, present the computation to be differentiated as a Lisp program.

4 A Brief Explanation of Forward Differentiation for ADIL

Consider a space of "differentiable functions evaluated at a point c." In this space we can represent a "function f at a point c" by a pair $\langle f(c),\ f'(c)\rangle$. That is, in this system every object is a pair: a value $f(c)$ and the derivative with respect to its argument $D_x f(x)$, evaluated at c. (written as $f'(c)$).

For a start, note that every number n is really a special case of its own Constant function, $C_n(x)$ such that $C_n(x) = n$ for all x. $C_3(x)$ is thus $\langle 3,\ 0\rangle$. The constant π is $t_1 = \langle 3.14159265\cdots, 0.0\rangle$, which represents a function that is always π and has zero slope. The object $t_2 = \langle c, 1\rangle$ represents the function $f(x) = x$ evaluated at c. At this point we must be clear that all our functions are functions of the same variable, and that furthermore we will be fixing a point $x = c$ of interest. It does not make sense to operate collectively on $\langle f(y), D_y f(y)\rangle$ at $y = a$ and $\langle g(x), D_x g(x)\rangle$ at $x = b$.

We can operate on these objects. For example, sin operating on t_2 is the pair $\langle \sin(c), \cos(c)\rangle$. In general, $\sin(\langle a, a'\rangle)$ is $\langle \sin(a)\ ,\ \cos(a) \times a'\rangle$.

We can compute other operations unsurprisingly, as, for example the sum of two pairs: $\langle a, a'\rangle + \langle b, b'\rangle = \langle a + b\ ,\ a' + b'\rangle$. *Note, we have abused notation somewhat: The "+" on the left is adding in our pair-space, the "+" on the right is adding real numbers. Such distinctions are important when you write programs!* Similarly, the product of two pairs in this space is $\langle a, a'\rangle \times \langle b, b'\rangle = \langle a \times b,\ a \times b' + a' \times b\rangle$. This can be extended to many standard arithmetic operations in programming languages, at least the differentiable ones [6]. AD implementors seek to find some useful analogy for other operations which do not have obvious derivatives. Fallling in this category are most data structure manipulations, calls to external routines, loops, creating arrays, etc.[6] In ADIL, these mostly come free.

5 Is AD Patented?

At least two US patents have been issued on AD: 6,223,341 and 6,397,380; also see application 20030023645 (2001). There are free on-line copies of these and all other patents.

6 Conclusion and Some History

AD programs can be easily written and executed in Lisp. This is not surprising. What is perhaps surprising is that no one has written this paper earlier.

We have experimented with alternative approaches, both for forward and reverse differentiation, but this approach seems to be a good combination of brevity, clarity, extensibility, generality, and efficiency.

The initial draft of this paper and the associated program was completed circa January, 2001. The paper and programs were slightly revised in 2005. A September, 2014 Google search for {"automatic differentiation" lisp} obtains about 46,800 hits, some of them to implementations of AD in Lisp or the Lisp dialect Scheme, as well as an earlier draft of this paper. Search on the phrase "algorithm differentiation" finds 121 hits.

The autodiff.org web site remains an excellent resource.

[6]It is a mistake to `declare` variables to be (say) double-floats, when in the ADIL framework they will be `df` structures. We are not aware of any other systematic problems.

7 Acknowledgments

Thanks to the CCA referee and editor for numerous useful suggestions. Among them, for purposes of relating the computer algebra community to AD, see the early work using REDUCE, [4] and Maple in [1]. What is interesting is that if one requires input in a CAS language and output in that CAS language or even in(say) FORTRAN, the task becomes more difficult than just using Lisp for it all.

Appendix 1

First we display a seven line Lisp differentiation program (similar to many others written over the years) that is distinguished by brevity. This one was posted on a Lisp newsgroup by Pisin Bootvong, some time ago, and makes use of the Common Lisp object system (CLOS) and `destructuring-bind` nicely:

```
(defmethod d ((x symbol) var) (if (eql x var) 1 0))  ;;dx/dx =1
(defmethod d ((x number) var) 0)                      ;;d(number)/dx = 0
(defmethod d ((expr list) var)
   (destructuring-bind (op e1 e2) expr
     (case op
       (+ `(+ ,(d e1 var) ,(d e2 var)))               ;; d(a+b) = d(a)+d(b)
       (* `(+ (* ,(d e1 var) ,e2) (* ,e1 ,(d e2 var))))))  ;; d(a*b)=a*d(b)+b*d(a)
```

Here's a more elaborate, but still short Lisp program with more capabilities and greater extensibility [3].

```
(defun d(e v)(if(atom e)(if(eq e v)1 0)
               (funcall(or(get(car e)'d)#'undef)e v)))

(defun undef(e v) `(d ,e ,v)) ;;anything unknown goes here

(defmacro r(op s)`(setf(get ',op 'd) ;;define a rule to diff operator op!
                (compile() '(lambda(e v)
                             (let((x(cadr e)))
                               (list '* (subst x 'x ',s) (d x v)))))))
(r cos (* -1 (sin x)))
(r sin (cos x))
(r exp (exp x))
(r log (expt x -1)) ;; etc,
(setf(get '+ 'd)  ;; rules for +, *, expt must handle n args, not just 1
  #'(lambda(e v) `(+,@(mapcar #'(lambda(r)(d r v))(cdr e)))))
(setf(get '* 'd)
  #'(lambda(e v) `(*,e(+,@(mapcar #'(lambda(r) `(*,(d r v)(expt,r -1)))(cdr e))))))
(setf(get 'expt 'd)
  #'(lambda(e v) `(*,e,(d `(*,(caddr e)(log,(cadr e)))v))))
```

Other programming languages, especially ones with a functional approach, can usually handle this task nicely, but the major issue (and one not addressed here) is simplification of the result. Trying to write a simplifier adds substantially to the programmer's burden. The differentiation program in a computer algebra system like Maxima is much larger because it handles a larger class of functions but even so should be ultimately faster by generating more compact simplified forms.

Another program whose specifications seem superficially like the previous one does not build any list structure. It assume that the result of interest is the value of the derivative at a point, not its symbolic representation. Thus the `d` function takes another argument, the point `p`. It returns the derivative of the `expr` with respect to the `var` at the point `p`. We are no longer returning lists. Note that we now have to evaluate expressions involving the variable, for which we have defined the `val` method. Although such a program can be written entirely using the skeleton of the previous few programs, we illustrate a different approach using generic programming (a handle on object-oriented programs) supported in Common Lisp.

```
(defmethod d ((x symbol) var p) (if (eql x var) 1 0))
(defmethod d ((x number) var p) 0)
(defmethod d ((expr list) var p)
   (destructuring-bind (op e1 e2) expr
     (case op
       (+ (+ (d e1 var p) (d e2 var p)))
       (* (+ (* (d e1 var p) (val e2 var p)) (* (val e1 var p) (d e2 var p)))))))

(defun val(expr var p)(funcall  `(lambda (,var) ,expr) p))
```

This program's `case` statement would have to be expanded for other two-argument functions, and would also need to be altered for one-argument functions like sin and cos.

Appendix 2 All the Code

An earlier version of this article included program source code for ADIL, the overloading version and generic arithmetic, constituting about 542 source-code lines and `dcomp`, about 268 lines. These line counts include comments, examples, and some timing data. These are instead collected in the single file

`http://www.cs.berkeley.edu/~fateman/generic/all-adil.lisp`

References

[1] M. Berz, C.H. Bischof, G. Corliss (eds), "Computational Differentiation: Techniques, Applications and Tools," SIAM Proceedings in Applied Mathematics Series, Vol 89, SIAM 1996.

[2] K.A. Broughan, D. M. K. Willcock, "Fortran to Lisp translation using f2cl", *SoftwarePractice & Experience, Volume 26 Issue 10*, Oct. 1996, pp. 1127-1139.

[3] R. Fateman, "A Short Note on Short Differentiation Programs in Lisp, and a Comment on Logarithmic Differentiation," *SIGSAM Bulletin Volume 32, Number 3*, Sept., 1998, pp. 2-7. `www.cs.berkeley.edu/~fateman/papers/deriv.pdf`.

[4] Victor V. Goldman, van Hulzen, J. A. and Jan H. J. Molenkamp, "Efficient Numerical Program Generation and Computer Algebra Environments", in [6] pp. 74-83.

[5] A. Griewank, "On Automatic Differentiation," in M. Iri & K. Tanabe (Eds.) *MATHEMATICAL PROGRAMMING,* Kluwer Academic Publishers, 1989, pp. 83-107.

[6] A. Griewank and G. Corliss (eds), *Automatic Differentiation of Algorithms: Theory, Implementation and Application,* Proceedings of the First SIAM Workshop on Automatic Differentiation, SIAM, 1991, (see esp. paper by L. Rall.)

[7] D. Kalman, "Doubly Recursive Multivariate Automatic Differentiation", *Mathematics Magazine, vol 75,* no 3, June 2002. pp. 187-202

[8] L. F. Shampine, "Accurate numerical derivatives in MATLAB".*ACM Trans. Math. Softw. vol 33*, no 4, (August 2007).

[9] Pearlmutter, B.A. and Siskind, J.M., "Reverse-Mode AD in a Functional Framework: Lambda the Ultimate Backpropagator,"*ACM Transactions on Programming Languages and Systems (TOPLAS), 30* (2) pp. 1-36, April 2008.