Symbolic differentiation in Tcl: reusing the Tcl parser for symbolic algebra

Kevin B. Kenny
Computational Biology Laboratory, GE Global Research Center, Niskayuna, NY
kennykb@research.ge.com

Many mathematical algorithms are improved by the ability to provide the derivatives of functions. This paper presents a symbolic differentiator for Tcl: a Tcl function that operates on Tcl expressions and produces expressions that compute their derivatives. The differentiator exploits the little-used API's to the Tcl parser to analyze the input expressions, an unusual use of procedures that were intended for an entirely different purpose (code instrumentation for profiling and debugging).

1. Introduction

Many algorithms for mathematical analysis and optimization require, or at the very least are improved, by knowledge of the derivatives of functions. Minimization and maximization of functions is a case in point: algorithms such as the conjugate-gradient method are usually much faster than "derivative-free" algorithms such as Powell's method.[1] Similar improvements are achieved in root finding, curve fitting, and the solution of ordinary differential equations. Nevertheless, users of routines that carry out these tasks often cry out for the "derivative-free" methods, simply because it is a tedious task to compute the derivatives.

The desire for derivative-free interfaces is understandable in languages like C or Fortran, whose capacity to deal with symbolic computation is limited. By contrast, Tcl has extensive features for string manipulation and symbolic calculation. Surely we can do better. This paper presents one approach: using the existing Tcl parser to process mathematical expressions and generate symbolic calculations for their derivatives.

2. Symbolic differentiation

Symbolic differentiation is one of the easier problems in symbolic algebra; it is often presented as a student exercise in artificial-intelligence courses[2]. A symbolic differentiator operates by traversing the parse tree of an expression recursively, differentiating each node in turn. At each step, a simple rule applies. The method is perhaps best illustrated by considering a concrete problem, such as

> Find the derivative of `sin($x)+cos($x)*sin($y)` with respect to `x`.

[1] Readers unfamiliar with these methods can get an introduction in [PTVF92], sections 10.5-10.6.
[2] For instance, it appears as "Problem 16-1" in [WiHo81], pp. 215-6.

A possible parse tree of the given expression is shown in Figure 1. This particular tree has nodes of only five different types:

1. The operator ‘+’;
2. The operator ‘*’;
3. The *apply* operator, which applies a function to a subexpression;
4. Function names (‘cos’ and ‘sin’).
5. Variable references (‘$x’ and ‘$y’).

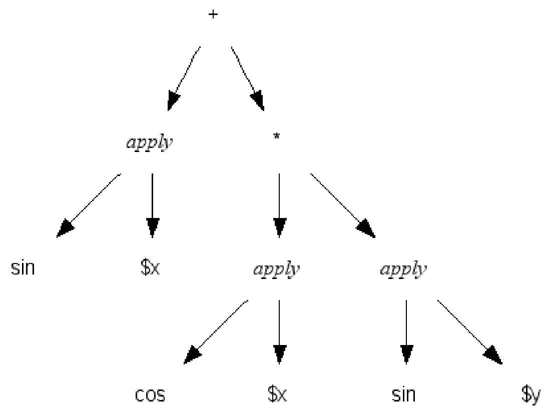

Figure 1. Parse tree of the expression ‘sin($x) + cos($x) * sin($y)’

Hence, differentiating the expression, given the parse tree, requires only four rules, shown in Table 1; plus two auxiliary rules that give the derivatives of `sin` and `cos`. (In the table, the prime symbol (‘′’) represents the result of recursively computing the derivative of the given subexpression.

Table 1. Differentiation rules needed for the parse tree of Figure 1.

Expression	**Derivative with respect to ‘x’**
`$`*v*	`1` if *v* is `x`, `0` otherwise.
u + *v*	*u*′ + *v*′ (Sum Rule)
u * *v*	*u*′ * *v* + *u* * *v*′ (Product Rule)
Apply: f(u)	*f*′(*u*) * *u*′ (Chain Rule)
`sin($x)`	`cos($x)`
`cos($x)`	`-sin($x)`

Given these rules, we can compute `[derivative {sin($x)+cos($x)*sin($y)}]` by applying them recursively as shown in Figure 2. The process is entirely mechanical; given the parse, it can be done by simple string replacements. Moreover, it is easy to see how Table 1 could be expanded to cover the remainder of Tcl’s mathematical operators and functions (at least, those that have analytical derivatives).

```
derivative {sin($x)+cos($x)*sin($y)}
  derivative {sin($x)}
    functional-derivative sin(·)
    ==> cos(·)
    derivative {$x}
    ==> 1
  ==> cos($x)*1
  derivative {cos($x)*sin($y)}
    derivative {cos($x)}
      functional-derivative cos(·)
      ==> -sin(·)
      derivative {$x}
      ==> 1
    ==> -sin($x)*1
    derivative {sin($y)}
      functional-derivative sin(·)
      ==> cos(·)
      derivative {$y}
      ==> 0
    ==> cos($y)*0
  ==> -sin($x)*1*sin($y) + cos($x)*0*cos($y)
==> cos($x)*1+((-sin($x)*1)*sin($y) + cos($x)*(0*cos($y)))
```

Figure 2. Naïve differentiation of sin($x)+cos($x)*sin($y)

The calculation shown in the figure illustrates one basic trouble: the computation results in a horrendous expression because there is no algebraic simplification. Fortunately, only a few rules suffice to improve it to something that is at least reasonable:

1. Addition of zero (on either side of the plus sign) may be eliminated by replacing the sum with the non-zero operand.
2. Multiplication by zero (on either side of the multiplication sign) may be eliminated by replacing the product with zero.
3. Multiplication by one (on either side of the multiplication sign) may be eliminated by replacing the product with the other operand.

Applying these rules at each step as the derivative is constructed simplifies the result to:

```
cos($x)+(-sin($x)*sin($y))
```

The latter expression is at least legible, if not elegant.

3. Exploiting Tcl's own parser

We have just seen how a handful of rules for computing derivatives, plus another handful for performing rudimentary algebraic simplification, can produce a useful symbolic differentiator that operates on parse trees. We are now left with the problem of producing an appropriate parse tree from an input expression. This problem is the more difficult of the two.

Fortunately, Tcl, being an interpretive language, of necessity embeds a parser for expressions that is available at run time. The parser has only a C API [Tcl06a], so some means has to be found to make the information available at script level. This problem too, is already solved, in the parser interface originally developed for the now-obsolete TclPro tool suite [Ajub03]. The parser was originally intended for use in syntax checking, and in code instrumentation for profiling and debugging. It is nevertheless serviceable for the differentiator.

The parser from TclPro has a somewhat awkward interface, owing to the fact that it was intended for parsing large files or portions of large files. It presents a `[parse]` command that accepts the type of object to be parsed (for our purposes, only '`expr`' need be considered), the string to be parsed, and a two-element list comprising the character indices that contain the object in question. Its return value is a *subtree:* a three-element list that contains a token type, the range of character indices that the subtree represents, and a list of components that make up the subtree. Each of the components is itself a subtree. Token types of interest include:

- `subexpr`, which denotes a compound expression of some sort;
- `operator`, which denotes the application of an operator such as '`+`' or of a mathematical function;
- `variable`; which denotes a variable reference such as '`$x`'; and
- `text`; which denotes a constant string.

While it would be possible to generate derivatives by walking this structure recursively, the differentiator takes a slightly different approach. Since the expressions to be differentiated are small, the cost of duplicating the constant strings in `text` subtrees is likewise small. The parser therefore rewrites this list into a form suitable for evaluation as a Tcl command, with the character ranges removed and the text simply embedded as parameters. For the expression that we have been considering, the rewritten command (with backslashes, newlines and indentations added for clarity) is:

```
{operator +} \
    {{operator sin} {var x}} \
    {{operator *} \
        {{operator cos} {var x}} \
        {{operator sin} {var y}}}
```

4. Details of the differentiator

Given the parse tree in command form, the differentiator inserts the name of the variable with respect to which the derivative is being computed after the command name, and evaluates the resulting command in the `math::symdiff::differentiate` namespace:

```
namespace eval math::symdiff::differentiate \
    [linsert $parseTree 1 $varName]
```

The result of the evaluation is expected to be a parse tree in the same form: second and higher derivatives can be obtained by evaluating the result with another variable name inserted.

The differentiator includes the rules for finding the derivatives of sums, differences, products, quotients and powers. It also has the basic rules for differentiating the built-in functions, and a number of these rules also invoke common code for the Chain Rule. A typical rule, in fact an unusually complex one, is the one for the two-argument arc-tangent function, shown in Figure 3.

```
proc {math::symdiff::differentiate::operator atan2} {var f g} {
    set df [eval [linsert $f 1 $var]]
    set dg [eval [linsert $g 1 $var]]
    return [MakeQuotient \
                [MakeDifference \
                     [MakeProd $df $g] \
                     [MakeProd $f $dg]] \
                [MakeSum \
                     [MakeProd $f $f] \
                    [MakeProd $g $g]]]
}
```

Figure 3. Typical differentiation rule.

Here we see the recursive nature of the differentiator at work: it begins by differentiating the two arguments to `atan2` with respect to the given variable, and then applies the rule:

$$\frac{d}{dt}\tan^{-1}\frac{f}{g} = \left(g\frac{df}{dt} - f\frac{dg}{dt}\right)\Big/\left(f^2 + g^2\right).$$

A handful of functions, no more than a couple of hundred lines of code in all, implement this part of the differentiator.

The `[MakeSum]`, `[MakeDifference]`, `[MakeProd]`, … functions, in the simplest form, could have been implemented simply as invocations of `[list]`. In the actual implementation, though, they include the algebraic simplifications discussed above. `[MakeSum]`, for instance, has special cases:

- If either operand begins with a unary minus, the sum is rewritten as a difference.
- If either operand is a constant 0, the sum is rewritten as the other operand.
- If both operands are constants, the sum is folded to a constant representing their sum.

Similarly, `[MakeProduct]` has special cases to lift unary minus out of products; to simplify multiplications by zero, one and –1; and to reduce the product of two constants to a constant. The other expression constructors have similar peephole optimizations.

A simple (twenty-line or so) Tcl script then converts the list representation back to Tcl's notation so that `[eval]` can deal with it.

5. Experience with the differentiator.

The resulting derivatives are hardly a minimal representation, but they are good enough for numeric evaluations. The differentiator has been integrated successfully with:

- A multidimensional root-finder using modified Newton-Raphson iteration.
- A function minimizer using the conjugate-gradient method and explicit derivatives.
- A non-linear least-squares curve fitter using the Levenberg-Marquardt algorithm.
- A solver for stiff ODE's based on the Fortran code LSODAR (and described in a companion paper).

6. Directions for future work.

The differentiator as it stands is serviceable, but there are ample opportunities for future work. Among the most obvious challenges are:

- Better algebraic simplification for the output. This sort of technique also moves Tcl toward the realm of a true symbolic-algebra system that could be used for more than just differentiation.
- Cleaning up the horrible API of the TclPro parsing extension and integrating it more tightly into the Tcl core (failing to expose the parser at script level is an egregious oversight).
- Reworking the Tcl-command representation to use the "math functions as commands" and "math operators as commands" syntax of Tcl 8.5. This change would (in 8.5) make it possible to evaluate the generated derivatives directly without reconverting to infix notation.
- At the same time that such a change is made to use `tcl::mathfunc::F` notation for the built-in mathematical functions, the ability for users to extend the set of supported functions dynamically should be included. Such an extension would allow special functions such as exponential integrals, elliptic integrals and Jacobian elliptic functions, Bessel functions, and so on to be differentiated symbolically.

Before tackling these issues, though, it would be worthwhile considering whether symbolic differentiation is, in fact, the best approach to the problem of computing derivatives. Alternatives are numeric differentiation (through some finite difference method) and *automatic differentiation* [Kede85].

For the most part, numeric differentiation can be disregarded as being no improvement over the derivative-free methods; indeed, many of them resort to evaluating derivatives by finite differences. Automatic differentiation is, however, worth considering. It has the advantage that it can calculate derivatives for functions that are more complex than single expressions; indeed, it might be able to generate derivatives for entire Tcl procedures (albeit simple ones).

Automatic differentiation works by performing all computations that would be on the reals with an algebra of *dual numbers;* each real number *x* is replaced with an ordered pair $\left\langle f, f' \right\rangle$. The second member of the pair carries derivative information. A real constant *c* can be represented as $\left\langle c,0 \right\rangle$; a variable can be represented as $\left\langle v,0 \right\rangle$ if it is not the variable with respect to which the derivative is being extracted, or $\left\langle v,1 \right\rangle$ if it is. Operators and math functions can be evaluated according to rules similar to those in Table 1; for instance, $\left\langle x, x' \right\rangle + \left\langle y, y' \right\rangle = \left\langle x + y, x' + y' \right\rangle$ and $\sin(\left\langle x, x' \right\rangle) = \left\langle \sin(x), x' \cos(x) \right\rangle$.

Given these rules, and using the Tcl parser for commands as well as the one for expressions, it is possible to imagine an automatic rewrite of a procedure like the following one for computing the arithmetic-geometric mean of two numbers:

```
proc agm {x y} {
    while {abs($x-$y) > 1e-12} {
       set g [expr {sqrt($x*$y)}]
       set x [expr {0.5 * ($x + $y)}]
       set y $g
    }
    return $x
}
```

into one like:

```
proc dagm {x y} {
    while {[d> [dabs [d- $x $y]] {1e-12 0.}]} {
       set g [dsqrt [d* $x $y]]
       set x [d* {0.5 0.} [d+ $x $y]]
       set y $g
    }
    return $x
}
```

Given appropriate definitions of the procedures `[d+]`, `[d*]`, `[dreal]`, `[dsqrt]`, and so on, the latter procedure does indeed compute the arithmetic-geometric mean of its dual-number arguments. If one of its arguments is augmented with 1.0, it also computes the derivative with respect to that argument, as we can see by comparing the result with a finite difference approximation:

```
% puts [list [agm 1. 2.] \
         [expr {500.0 * ([agm 1.001 2.] - [agm 0.999 2.])}]]
1.4567910310469068 0.60520928884233438
% puts [dagm {1. 1.} {2. 0.}]
1.4567910310469068 0.60520923913814795
% puts [list [agm 1. 2.] \
         [expr {500.0 * ([agm 1. 2.001] - [agm 1. 1.999])}]]
1.4567910310469068 0.42579090127115027
% puts [dagm {1. 0.} {2. 1.}]
1.4567910310469068 0.42579089595437941
```

Note that the technique works even though the procedure computes the function by successive approximation. A great many control structures can be accommodated. The automatic-differentiation approach is extremely powerful; the chief drawback is that floating-point values throughout the program must be replaced with dual numbers. Explicit initialization of floating point values in lists, dictionaries, and other non-scalar objects would require sophisticated data flow analysis to accommodate.

7. Conclusions

Tcl 8.5 is not far from being an extremely capable system for ad-hoc mathematical calculations. With the ability to leverage evaluation codes programmed in low-level languages such as C or Fortran, ad-hoc entry of mathematical formulas for evaluation at run time, and graphics (either on the Tk canvas, or using extensions such as BLT, Zinc, Tk3D, or Vtk), it can be used to develop quite powerful modeling and simulation environments. The addition of symbolic calculations (in addition to derivatives, other calculations such as simple root-finding [Kenn96] are possible) provides another layer of richness that has yet to be fully exploited.

References

[Ajub03] Ajuba Solutions. "TclPro." Software released to the open source and available from `http://sourceforge.net/projects/tclpro/`.

[Kede85] Kedem, Gershon. "Automatic differentiation of computer programs." *ACM Trans. on Math. Software 6:* 2 (June, 1980), pp. 150-165. Available online to subscribers at `http://doi.acm.org/10.1145/355887.355890`.

[Kenn96] Kenny, Kevin B. "TclSolver: An algebraic constraint manager for Tcl." Proc. 4th USENIX Tcl/Tk Workshop. Monterey, Calif.: USENIX, July, 1996. Available online at `http://www.usenix.org/publications/library/proceedings/tcl96/kenny.html`

[PTVF92] Press, William H, Saul A Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C: the art of scientific computing.* 2nd ed., Cambridge, England: Cambridge University Press, 1992. ISBN 0-521-43108-5. Available on-line at `http://www.nrbook.com/a/bookcpdf.php`.

[Tcl06a] Tcl Core Team. Manual page for `Tcl_ParseExpr`, `http://www.tcl.tk/man/tcl8.5/TclLib/ParseCmd.htm`, dated 3 November 2006 and retrieved 22 August 2007.

[WiHo84] Winston, Patrick Henry and Berthold Klaus Paul Horn. *LISP*. Reading, Mass.: Addison-Wesley, 1981. ISBN 0-201-08329-9.