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

(Extended abstract)

Symbolic differentiation is one of the easier problems in symbolic algebra; it is often presented as a student exercise in artificial-intelligence courses. Even though it is easy, it remains useful (and often underutilized) for mathematical computations such as root-finding, minimization and maximization of functions, and solving ordinary differential equations.

The most time-consuming part of writing a symbolic differentiator, in many languages, is writing a parser for the expressions to be differentiated. Fortunately, Tcl, being an interpretive language, comes with a parser for expressions that is available at run time. While the parser is not normally exported to scripts, the parser interface that the instrumentor in TclPro uses allows for script access via an extension. One advantage to using the built-in parser is that the programmer can be certain that the language of expressions to be differentiated is exactly the language of expressions to be evaluated.

The differentiator begins with a Tcl expression whose derivative is to be found, and the variable with respect to which it is being differentiated. The first thing that it does is to se the 'parser' extension to parse the expression. It then rewrites the parse tree into a form that is suitable for evaluation as a Tcl command. For instance, the Tcl expression,

```
2 * sin($x) * cos($x)
```

would be rewritten into the Tcl command:

```
{operator *} \
    {{operator *} \
        {constant 2.} \
        {{operator sin} \
            {var x}}} \
    {{operator cos} \
        {var x}}
```

Various rewritings are then available by evaluating the command in various namespaces. In particular, the command:

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

differentiates the given expression with respect to the given variable. The result is a tree in the same form: second and higher derivatives can be obtained by the same method.

The differentiator itself is fairly stupid. It 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:

```
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]]]
}
```

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 could have been implemented simply as invocations to [list]. In the actual implementation, though, they are done with a modicum of "peephole optimization." [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.

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).

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.

Tcl 8.5 is not far from being an extremely capable system for ad-hoc mathematical calculations; extensions such as this one point the way.