Mathematical Expression Handling In Perl

Jonathan Worthington (jonathan@jwcs.net)

September 1, 2005

These notes and the talk that goes with them present a method of working with mathematical expressions analytically. This ranges from simply evaluating an expression by substituting values in for its variable(s) and named constants to more complex manipulations, such as finding an expression that represents the derivative of the original expression.

1 The Analytical Approach

Computers are great at working with numbers, but sometimes there is a need to evaluate mathematical expressions that will be unknown until runtime. Further, sometimes it is desirable to do analytical manipulations - that is, manipulations that take one expression and give back another one. This potentially allows for no numerical errors to be introduced in the manipulation. A numerical result will usually be required at some point, however, so a way of evaluating the expression is required.

2 The Internal Representation

Mathematical expressions are written in mathematical notation.

$$\cos(\frac{\exp(a^2 - x^2)}{x + 2a})$$

This is usually translated to a form that is easy to type on a standard keyboard by using `/` to denote division, `^` to denote exponentiation and, maybe optionally, putting a `*` to explicitly denote multiplication. The above expression then becomes

```
cos(exp(a^2 - x^2) / (x + 2*a))
```

This is the notation that the series of Perl modules that will be discussed take as input and hand back as output when analytical manipulation is taking place.

2.1 Weaknesses Of A Textual Representation

An early attempt at manipulating expressions involved working with them in their textual form. Essentially the manipulation of the expression became a series of string manipulations. With Perl being so good at string manipulation, an inexperienced glance at the problem suggests this might be a good idea. The reality is that it is not.

A lot of Perl's strength in string manipulation comes from its regex support. While Perl's regexes go some way beyond the capabilities of real regular expressions, using them to match non-regular languages is often complicated or infeasible. Mathematical expressions do not form a regular language; properly nested brackets cannot be matched by a regular expression (proof available through the pumping lemma).

Working with expressions in a textual representation quickly made the code complex and difficult to extend without introducing bugs. Presenting a simple way of allowing other developers to build upon this work to implement their own manipulations was difficult too as the parsing and manipulation were too closely related.

The solution was to parse expressions once, turning them into an internal representation that was easier to manipulate. Another routine could turn this internal representation back into the familiar human-readable text form.

2.2 Representing Expressions With A Tree

The method chosen by the author to represent expressions involves using a non-strict binary tree (that is, a tree structure where each node has a maximum of two branches).

A node with no branches represents either a variable (x), a named constant (c) or a numerical constant (42). A node with 1 branch represents a function (sin, cos, exp, ln, etc.), the single branch being the expression that is the parameter of the function. A node with two branches represents an operator (+, -, *, / and ^), the two branches being the operands (expressions that the operators act upon).

Therefore the expression

$$\cos(\frac{\exp(a^2 - x^2)}{x + 2a})$$

Would be represented by the tree in Figure 1.

Notice that there is no need to store any details about bracket placement as order of evaluation is encoded in the tree structure; evaluation is simply done depth first. Tree structures are also simple to manipulate using recursion. Both evaluation and differentiation are problems that can be expressed neatly using recursion.

Finally, note that the end user of any modules that manipulate expressions need never be exposed to this internal representation. The only thing

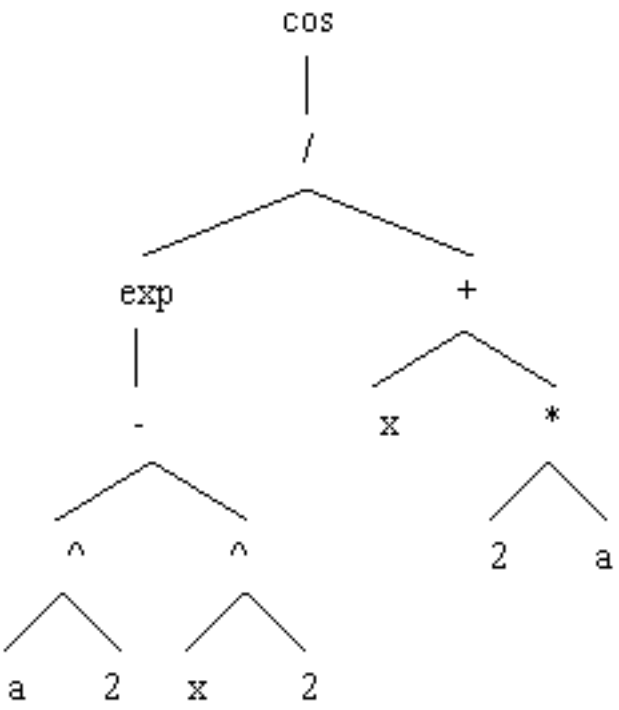

Figure 1: Figure 1: Example expression tree

they need to be prepared for is that white space and bracketing will not be preserved, though sufficient bracketing will always be generated such that the meaning of the expression stays the same.

3 Math::Calculus::Expression

This module implements a parser that transforms expressions in text form to an internal tree-based representation, transforms the tree back into a text form again and provides some basic operations, including:

- Evaluating the expression by substituting in values for all variables and named constants.
- Doing some basic simplifications (for example, recognising that "2 + 2" can be simplified to "4", that anything multiplied by zero disappears, etc.)
- Checking if two expressions have the same representation; if they do, they are equivalent; if they do not, then they may or may not be equal - more on this later.

- Cloning the expression to create a new expression object containing the same expression

The module is object oriented and is intended to be subclassed to provide additional functionality. If an undefined method is called on an expression object then it will attempt to load another module that provides that method and call it. For example, if the "differentiate" method is called then "Math::Calculus::Differentiate" would be loaded. This is done through implementing the required logic in the magical sub AUTOLOAD.

An example of using the Expression module is given below. It takes an expression and then evaluates it.

```
# Create an expression object.
use Math::Calculus::Expression;
my $exp = Math::Calculus::Expression->new;

# Set an expression and set its variable.
$exp->setExpression('2*x^2 + sin(2*t - x) + 10');
$exp->addVariable('x');

# Evaluate it with x = 4, t = 2.
my $val = $exp->evaluate(
x => 4,
t => 2
);
print "Evaluates to $val"; # 42
```

Each node of the expression tree is represented by a hash. Branches are simply hashrefs. At the bottom of the tree, instead of having a hashref to another node, a letter or number is stored. Thus it is possible to check if the branch is a subtree simply by using `ref`.

Using hashes to represent each node is not the cheapest solution, but it makes for easy to read code. It also means that it is easy to add extra data to a node, which allows the tree to be augmented.

4 Math::Calculus::Differentiate

This module inherits from Math::Calculus::Expression and adds the differentiate method that transforms the currently represented expression into its derivative. The derivative of an expression describes the gradient - how steep the curve is at each point. At this time the module only does partial differentiation - that is, differentiation with respect to a single variable.

Differentiation is implemented recursively. This works well, as many rules of differentiation work by taking part of the expression being differentiated and use the derivative of that part of the expression to build the

final solution. For example, the rule for differentiating an expression to a constant power is

$$\frac{d}{dx}[(e)^n] = n\frac{d}{dx}[e](e)^{(n-1)}$$

This is implemented as follows.

```
# d[(f(x))^n] = n*f'(x)*f(x)^(n-1)
return {
    operation => '*',
    operand1 => $tree->{'operand2'},
    operand2 => {
        operation => '*',
        operand1 => $self->differentiateTree
                            ($variable, $tree->{'operand1'}),
        operand2 => {
            operation => '^',
            operand1 => $tree->{'operand1'},
            operand2 => {
                operation => '-',
                operand1 => $tree->{'operand2'},
                operand2 => 1
            }
        }
    }
};
```

Note how there is no need to do any parsing or care about precedence here, and how this results in clear, easy to follow code.

Using the module is simple, as illustrated by the following code snippet. Note also the use of the simplifier.

```
# Create an expression object. Note we could have used the
# Expression module, and it would have loaded the subclass
# for us.
use Math::Calculus::Differentiate;
my $exp = Math::Calculus::Differentiate->new;

# Set an expression and set its variable.
$exp->setExpression('2*x^2 + sin(2*t - x) + 10');
$exp->addVariable('x');

# Differentiate with respect to x. This prints out:
# 2*2*1*x^(2 - 1) + (2*t - 1)*cos(2*t - x) + 0
```

```
$exp->differentiate('x');
print $exp->getExpression . "\n";

# If we simplify it, things get cleaner. This prints out:
# 4*x + (2*t - 1)*cos(2*t - x)
$exp->simplify.
print $exp->getExpression . "\n";
```

5 Math::Calculus::NewtonRaphson

The Newton Raphson method can be used to find a solution to an equation or system of equations. It is a numerical method, and the module described here demonstrates how numerical and analytical methods can come together. The condition for Newton Raphson to solve an equation is that it has to be in the form

$$f(x) = 0$$

Therefore the left hand side is an expression, and can be represented and evaluated. It uses the iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Where f' is the derivative of f. Note that analytical differentiation can be (and is) used, which eliminates the errors that would have been incurred through numerical differentiation.

As with any iteration, Newton Raphson requires an initial guess. The user of the module must supply this. The following example solves

$$x^x + sin(2x) = 7$$

Which has been rearranged to give

$$x^x + sin(2x) - 7 = 0$$

```
# Create an expression object.
use Math::Calculus::NewtonRaphson;
my $exp = Math::Calculus::NewtonRaphson->new;

# Set an expression and set its variable.
$exp->setExpression('x^x + sin(2*x) - 7');
$exp->addVariable('x');

# Attempt to solve it with initial guess 3.
my $sol = $exp->newtonRaphson('x', 3);
print "Solution is $sol\n"; # 2.38828587710838
```

6 Equivalence Of Expressions

Two mathematical expressions are equivalent if they are equal when evaluated for all possible values of their variable(s). As evaluating an expression for every value of its variables is not a feasible implementation of equivalence testing, something else is needed.

One thing that is obvious is that two expressions that have the same representation are equivalent. For example, if "$2 + x$" and "$(2) + (x)$" are parsed, they lead to the same tree. This is too tight as a condition for equivalence, however. Consider "$2+x$" and "$x+2$". These will have different tree representations, but are clearly equivalent. This could be solved by trying to impose some kind of ordering, but that quickly falls down when it comes to testing if "$(x + 1)(x - 1)$" is equivalent to "$x^2 - 1$". This latest problem could be solved by also implementing multiplying out of brackets, so the expression is a sum of products.

Despite the growing complexity of the current solution, it still fails to reveal that "$sin(x)/cos(x)$" is equivalent to "$tan(x)$". Trying to implement all of the trigonometric and hyperbolic identities would be the next step. Clearly, trying to massage the two trees so they become the same is difficult and messy.

A better approach is to find a canonical form for representing expressions. A canonical form is one where all equivalent expressions have the same representation. A Taylor Series is such a form; it reduces the expression to an infinite polynomial ("$k_0 + k_1x + k_2x^2 + k_3x^3 + ...$"). Finding the n^{th} coefficient involves evaluating the n^{th} derivative at a chosen point, so the expression needs to be infinitely differentiable.

If the Taylor Series of two expressions are equivalent, it can be said that the expressions themselves are equivalent. Obviously, it is not possible to compare the coefficients of every term in the series, so only the first N are tested, where the chosen value of N depends on how similar the expressions must be.

Using Taylor Series here has its downfalls; evaluating a large number of coefficients becomes expensive. Computing the derivatives is time consuming and for some functions the size of the derivative expression, under the currently available expression simplifier on CPAN at the time of writing, blows up exponentially. Also, finding the n^{th} coefficient involves computing $n!$, which grows very quickly.

The module Math::Calculus::TaylorEquivalent, available from CPAN, implements this and will happily find all the earlier examples to be equivalent. It was also trivial to implement.

7 Conclusions

The modules discussed in this paper support tasks from acceptance and evaluation of numerical expressions to performing more advanced manipulations. Representing expressions as these modules do enables a range of analytical manipulations to be implemented relatively simply. Using analytical methods may reduce error compared to using numerical method, but will also decrease performance greatly.

While these particular modules may not be of use to many, the concept of using an internal representation to provide an easier development model and reduce implementation complexity is more widely applicable.