Algebraic Simplification and Automatic Differentiation

Stephen M. Watt
(smwatt@watson.ibm.com)

IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

This talk presents algebraic simplification techniques of computer algebra and optimizing compilers that can be usefully applied in the field of automatic differentiation.

We preface the presentation with an example of how analytic derivatives improve upon finite differencing in a real-world application: IBM Burlington has several mainframes devoted to simulating new semiconductor technologies before they are manufactured. Converting the device models with a custom piece of automatic differentiation software employing algebraic simplifications led to an estimated \$1,000,000 in monthly savings in terms of CPU time.

We note that the meaning of algebraic simplification depends on the class of expressions and on the measure of simplicity. Even for the relatively simple class of univariate polynomials, there is no single best definition: a factored or expanded representation might be "simpler" on a case-by-case basis. A well-defined subproblem of simplification is to ask whether an expression is equivalent to zero. This question is easier and sufficient for many purposes. Depending on the class of expressions, however, even this question can be provably unsolvable.

Some classes of expressions admit "normal" or "canonical" forms. Simplification to a normal form converts a zero-equivalent expression to 0. Simplification to a canonical form converts mathematically equivalent expressions to a unique representative. Conversion to these forms requires exact coefficient arithmetic, so implementations usually convert floating-point numbers to rational numbers beforehand.

In computer algera, various normal and canonical forms are used for polynomials and rational functions. The class of functions can be extended by treating $\sin(x)$, $\cos(x)$, $\sin(2x)$, etc., as new indeterminates. In this setting, many identities are algebraic relations (e.g., $\sin^2(x) + \cos^2(x) - 1 = 0$). There may be a structure theorem for a class of functions that can be used to express a given set of extensions in terms of an algebraically independent set. This leads to expressions that are in one sense simpler, even though the resulting form is usually more lengthy. It is often more convenient to retain a set of algebraic relations and to simplify expressions modulo this set. An important technique in computer algebra is to simplify an expression modulo a set of polynomials of this sort. We describe how this can be done using Gröbner bases. We then describe other simplification techniques including factorization, functional decomposition, radical transformation and rule-based methods.

Various computational techniques of computer algebra are based on algebraic properties. One standard method is to solve and combine several simpler problems. An example of this is the computation of several modular images that are combined by Chinese remaindering.

One might distinguish between "black box" and "clear box" function representations. In a "black box" representation, one can compute only values of a function — one cannot see any explicit expression structure. This form is the standard in numerical computing, but it has only recently found effective use in computer algebra. In a "clear box" representation, a function is represented as an expression tree or sequence. This is the standard form in computer algebra, but it is relatively uncommon in numeric computing, with automatic differentiation being one example.

Compiler optimizations can be seen as algebraic simplification of "clear box" function representations. We describe a number of these optimization techniques applicable to automatic differentiation. We begin with a description of basic blocks and flow graphs. We then give a detailed description, global data-flow analysis and illustrate how it can be used to eliminate variables or eliminate common subexpressions. We also discuss transformation of programs to static single assignment form and data structure elimination.

In conclusion, we present a list of "dos" and "don'ts":

- *Do* exploit algebraic relationships (e.g., sin, cos, matrix ops.).
- *Do not* blindly trust loosely defined simplifications.
- *Do* use term orderings to eliminate more expensive function calls.
- *Do not* expect miracles from expression simplification.
- *Do* use data-flow analysis for dependency information, common subexpression elimination, etc.
- *Do not* expect compiler providers to include specific automatic differentiation methods.
- *Do* ask compiler providers to architect application-oriented back doors into their optimizers.