Automatic Differentiation System For Compiled Functions

Luc Maisonobe

1.Overview

This paper describes the concepts underlying an innovative automatic differentiator for mathematical functions.

Just like the mathematical Nabla operator transforms a function into its differential, an automatic differentiator transforms an existing software implementation of a mathematical function `f` into another software implementation that either in addition to or instead of computing the same value as the original implementation, computes the value of its derivative. The created implementation is built automatically by applying the classical exact differentiation rules to the function underlying expressions. There are *no* approximations and *no* step sizes.

This approach has the following benefits:

- differentiation is exact
- there are no problem-dependent step size to handle:
  - no problems of physical units
  - no problems of order of magnitude
  - no *a priori* knowledge of the behavior required
- there are no configuration parameters
- differentiation can be computed even at domains boundaries

2.Prior art

Automatic differentiation is not new[1]. Several tools exist, they mainly fall in three categories:

1. systems that rely on dedicated types and overloaded mathematical operators
2. systems that rely on a high level mathematical representation of the function and generate source code
3. systems that analyze source code and generate source code

The first category is often encountered in custom-developed programs when the programming language provides support for overloading like C++ or Fortran95. They need specific development of the original function using specific types instead primitive real variables.

The second category replaces dedicated types in the programming language by automatic generation of

1 http://www.autodiff.org/

source code using Computer Algebra Systems[2] like Mathematica[3], Maxima[4] ... The function to be differentiated is written directly in the CAS system language. Symbolic differentiation is used and the resulting expression is exported outside of the CAS system by source code generation. Supported languages for generation are often limited to Fortran and C. The differentiation is therefore an ad-hoc process manually merged in the development process of the program. The original function must be imported into the computer algebra system mainly by rewriting it from scratch, and the generated function must be imported back into the program often with manual changes to fit it to the high-level program since there is no way to specify a desired interface.

The third category is an evolution of the second one that removes some of its drawbacks by reading directly source code in some supported programming language and generating code with interfaces similar to the original ones. Tools exist for C, C++, Fortran77, Fortran95 and Matlab. These tools still need the complete source code to operate.

3.New approach

The new approach described here is an evolution of the third category described previously. It goes further in the way towards transparency from a user's point of view by removing all source processing. It does this by analyzing and generating directly the low level representation of the program, not its source. This approach has the following benefits:

- there is no special handling of source:
    - no symbolic package with its own language
    - no limitation to single self-contained expressions
    - no source code generation
    - no manual integration with the rest of application
- one writes and maintains only the basic equation and get the differential for free
- it can be applied to already existing code without refactoring
- it is effective even when source code is not available

This approach also simplifies the analysis part since there is no need to implement a source parser but relying on work already done by the compiler. Low level representation is often much easier to analyze automatically (but not humanly!) since it is already decomposed in very small operations and basic data structures.

4.Dynamic environments

This approach has additional benefits for programs running in dynamic environments like the Java Virtual Machine. In these environments, on-demand loading allow the differentiation to be done entirely at run time.

---

2 http://en.wikipedia.org/wiki/Computer_algebra_system
3 http://www.wolfram.com/products/mathematica/index.html
4 http://maxima.sourceforge.net/

Differentiation done at runtime implies it is done on already existing instances. The result of differentiation is a generated object which must be linked to the primitive instances. The link between the primitive and generated objects is very tight: the primitive object is referenced from the generated one and no data is copied at all. This way, if some parts of the code not using derivatives change the state of the primitive object, the parts of the code using derivatives will use the changed state immediately, even in multi-threaded environments.

Another feature brought by dynamic environments is that it allows differentiation to be used in *callback* contexts. A typical example is a user program implementing some function `f(x)` and needing to find its maximum. The user program will call an third party mathematical library for this and not bother how this library computes this maximum. The library will only need to be provided an object implementing the `f(x)` function to call it back. With automatic differentiation at run time without source code, the third party library may take the object, differentiate the function, and use an efficient algorithm using the derivative to compute very quickly the maximum. Without these features, the third party library would either be forced to use less effective algorithms without derivative or to ask the user to provide the derivative himself.

5.Reference implementation

A reference implementation of the ideas exposed in this paper has been done. It is an open-source project called Nabla[5]. This reference implementation uses the Java platform. The complete source code and explanations are available on the project web site.

Nabla computes the derivatives applying the classical differentiation rules at bytecode level. When an instance of a class implementing `UnivariateDifferentiable` is passed to its `differentiate` method, Nabla tracks the mathematical operations flow that leads from the `t` parameter to the return value of the function. At the bytecode instructions level, the operations are elementary ones. Each elementary operation is then changed to compute both a value and a derivative. Nothing is changed to the control flow instructions (loops, branches, operations scheduling).

All the changed operations belong to a small subset of the virtual machine instructions. This set contains basic arithmetic operations (addition, subtraction ...), conversion operations (double to int, long to double ...), storage instructions (local variables, functions parameters, instance or class fields ...) and calls to elementary functions defined in the `Math` and `StrictMath` classes. There is really nothing more!

For each one of these basic bytecode instructions, we know how to map it to a mathematical equation and we can combine this equation with its derivative to form a pair of equations we will use later. For example, a `DADD` bytecode instruction corresponds to the addition of two real numbers and produces a third number which is their sum. So we map the instruction to the equation: $c=a+b$ and we combine this with its derivative to form the pair: $\left(c=a+b, \frac{dc}{dt}=\frac{da}{dt}+\frac{db}{dt}\right)$.

In this example, we have simply used the linearity property of differentiation which implies that the derivative of a sum is the sum of the derivatives. Similar rules exist for all arithmetic instructions, and the derivative of all basic functions in the `Math` and `StrictMath` is known.

5 http://commons.apache.org/sandbox/nabla/

Lets consider a more extensive example:

```
public class Linear implements UnivariateDifferentiable {
    public double f(double t) {
        double result = 1;
        for (int i = 0; i < 3; ++i) {
            result = result * (2 * t + 1);
        }
        return result;
    }
}
```

A class compiled from the above source would be converted by Nabla to a code roughly similar to the one that would result from compiling the code below. The major difference is that Nabla reads and generates bytecode directly, it does not use source at all.

```
public DifferentialPair f(DifferentialPair t) {

    // source roughly equivalent to code generated at method entry
    double t0 = t.getValue();
    double t1 = t.getFirstDerivative();

    // source roughly equivalent to conversion of the result affectation
    double result0  = 1;
    double result1  = 0;

    // this loop handling code is not changed at all
    for (int i = 0; i < 3; ++i) {

        // source roughly equivalent to conversion of "2 * ..."
        double tmpA0 = 2 * t0;
        double tmpA1 = 2 * t1;

        // source roughly equivalent to conversion of "... + 1"
        tmpA0 += 1;

        // source roughly equivalent to conversion of "result * ..."
        double tmpB0 = result0 * tmpA0;
        double tmpB1 = result0 * tmpA1 + result1 * tmpA0;

        // source roughly equivalent to conversion of "result = ..."
        result0 = tmpB0;
        result1 = tmpB1;

    }

    // source equivalent to code generated at method exit
    return new DifferentialPair(result0, result1);

}
```

As shown by this example, the only things that need to be changed for differentiating the `f` method are the `t` parameter and the `result` local variable whose types are changed from double to `DifferentialPair`, the two multiplications and the addition. In order to generate the new function, Nabla converts these elements one at a time, starting from the parameter change and propagating the change using a simple data flow analysis. There is no need to analyze the global structure of the code, and no need to change anything in the loop handling instructions.

This example shows that the instructions conversions have local scope. No global tree representation of the method is needed at all.

6. No intermediate DifferentialPair instances

Another thing that is shown in the previous example is that `DifferentialPair` instances appear only at method entry and exit. They are not used internally for elementary conversions, only pairs of local variables (or operand stack cells as we will see later) are used.

The only methods from the `DifferentialPair` class that are used by the automatic differentiator are the two arguments constructor, the `getValue` method and the `getFirstDerivative` method.

For differentiable functions that call other differentiable functions, additional instances of `DifferentialPair` will be used. Some will be built in the caller to pass parameters to the callee, and others will be build in the callee to return results to the caller. This is *not* the case for calls to the elementary mathematical functions defined in the `Math` and `StrictMath` classes. For the known functions the derivative computation are inlined. For example a call to the `Math.cos` function will be inlined as a call to `Math.cos`, a call to `Math.sin` and some intermediate arithmetic operations.

7. Data flow analysis

As shown in the example above, the original method bytecode contains both immutable parts that must be preserved and parts belonging to what we will call the *computation path* from the `t` parameter to the result that must be differentiated. The instructions that must be converted are identified by a data flow analysis seeded with the `t` parameter, which is changed from a primitive `double` in the original method to a `DifferentialPair` in the generated derivative.