Developing Tools for Floating-point Debugging

Grey Ballard
CS 263 Design and Analysis of Programming Languages
May 4, 2010

1 Introduction

Floating-point arithmetic exhibits subtle behavior and the growing majority of users of numerical software are not experts in roundoff error analysis. Even for experts, locating the source of floating-point exceptions and numerical instabilities in real world programs can be difficult. Further, many of the hardware features required or suggested by the IEEE Standard 754 which aid in the debugging of numerical programs are either not accessible to programming languages or simply not implemented in hardware at all [Kah06]. The objective in this project is to help develop a tool for debugging floating-point programs by using the techniques of “delta debugging” [ZH02] to localize numerically sensitive operations.

1.1 Tools for Floating-point Debugging

This class project fits into a more general research project at Berkeley on which David Bailey, James Demmel, William Kahan, Guillaume Revy, and Koushik Sen are all currently working. They have submitted an abstract to the Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2010) [BDK+10]. The main objective of the greater project is to reduce the difficulty of debugging roundoff and other numerical exception problems that arise in LAPACK [ABB+92] and other scientific computing codes. Possible numerical errors include overflow, underflow, invalid operations (involving NaN’s), as well as simply the return of an inaccurate result (raising no exceptions). In the case of LAPACK codes, these inaccuracies are often detected for anomalous input matrices by built-in checks, but determining the source of the instability is still very difficult.

There are various techniques for determining sensitivity to roundoff errors [Kah06]. One possibility is to perform the computation on slightly different inputs and check how widely the results vary. Another possibility is to perform the computation multiple times on the same input, each with a different rounding mode (up, down, to zero, as well as the default “to nearest”), and again check how widely the results vary. A similar approach (though more difficult to implement) is to perform the computation using interval arithmetic. Finally, one can also perform the computation at increasing levels of precision, trusting those digits in the answer that do not change as the precision changes. Hardware implementations generally include single and double precision (roughly 8 and 16 decimal digits, respectively) and sometimes an extended...
or “long double” precision. Computation in higher precision generally requires using a software package with special implementations.

1.2 Delta Debugging

Determining that a floating-point program is sensitive to roundoff is not sufficient for improving the robustness of the program. In order to fix the problem, we want to know exactly where in the code the sensitive floating-point operations lie. Delta debugging is a technique for simplifying the debugging effort by isolating the failure-inducing differences between a program which exhibits the problem and one that does not [ZH02]. That is, if a program produces an inaccurate result if run in single precision but a correct result if run in double precision, then delta debugging will find a minimal set of operations that must be performed in double precision in order to produce an accurate result. Note that this minimal set may not be unique to achieve the desired threshold, and it also may not represent the global minimum number of operations. This process localizes the numerical sensitivities, and it also suggests a cheap fix to the numerical bug (if the set of operations is small or the operations are not executed often). Delta debugging works by choosing a subset of the possible changes to the original code (e.g. changing the declaration of a single precision variable to double precision), implementing the changes, performing the computation, and testing the results. Section 2 discusses the details of delta debugging.

1.3 CIL

In order to implement the changes in the original code, we use the C Intermediate Language (CIL) and its tools [NMRW02]. CIL is a high-level representation of a C code that allows for easy manipulation and source-to-source transformation. The CIL framework allows a user to implement a “visitor” which traverses the CIL program tree and applies transformations where desired. Each node in the tree represents a variable, constant, operation, etc. whose attributes (e.g. precision type, rounding mode) can be changed. The output of the process is a human-legible, compilable C file. The transformed code can be compiled and executed and the results tested.

1.4 Double to Double-Double

Given the delta debugging infrastructure and a “visitor” class for each type of transformation, many different analyses can be performed. Guillaume Revy has developed a delta debugging infrastructure as well as written transformations for changing single precision to double precision, altering rounding modes, swapping function calls between two possible functions, and checking/resetting exception flags. My addition to the greater project is to add the functionality of transforming single or double precision code to higher (possibly arbitrary) precisions (using higher precision libraries developed by David Bailey and others).

Arbitrary precision software packages allow the user to specify both precision and range of floating-point numbers, which allows finer control over the code transformation. For example, to track an overflow or underflow bug, it might be important to alter the exponent range of a variable without changing the number of bits in its mantissa (the fractional part) so the accuracy
of the computed result doesn’t change. Simply changing from single to double increases both the exponent range and available precision. Numerical inaccuracies in double precision code may also require transformations to higher precision software implementations (when “long double” or extended precisions are either not implemented in hardware or are not sufficient for the problem).

I have implemented a tool which uses transformations from double precision to “double-double” precision. The higher precision package I used is available at David Bailey’s website [HLB08]. The double-double precision format stores floating-point numbers as the implicit sum of two double precision numbers (implemented as a struct). One of the double precision numbers stores the first 53 significant precision bits and the second stores the last 53 bits. Note that this doubles the precision but does not increase the range of a double precision number. The package includes overloaded operators for all the math library functions. Thus, in order to convert a double precision program to double-double precision, the programmer need only (in principle) add a header file and change the variable declaration types. Note that in order to exploit operator overloading, using the package implies that the program can no longer be compiled as C code. Creating mixed-precision code is more complicated— in addition to changing variable declarations, specifying the type of intermediate computations also has an important effect on the numerical precision of the final result.

The techniques used to coordinate the original C code with the delta debugging infrastructure, the CIL tools, and the double-double precision library can easily be generalized to handle arbitrary precision packages. Section 3 discusses the implementation details and the example codes on which the tool was run.

2 Delta Debugging

The delta debugging approach, presented in [ZH02], was first used as a tool for the simplification of bugs reported to the Mozilla web browser project. From thousands of bug reports (e.g. HTML pages that caused the browser to crash or lists of user actions that caused abnormal behavior), the goal was to develop a tool that would automatically reduce the size of each bug report (e.g. figure out which particular lines of code actually caused the crash) as well as reduce the number of distinct bug reports (i.e. determine equivalence classes of bug reports that share failure-inducing input). In the case of isolating particular lines of HTML code, the approach is to start with an empty document and test different subsets of lines until the browser crashes. The delta debugging algorithm determines a small number of lines of code that cause a crash such that if any of those lines were removed, the browser would not crash. With this smaller buggy HTML document, a human can more easily determine the actual bug in Mozilla. If multiple seemingly different pages reduce to the same isolated code, the extra bug reports can be removed from the database.

We will illustrate the delta debugging algorithm with the transformation of double precision to double-double precision in order to localize the operation(s) producing a numerical inaccuracy; the approach is nearly the same for the other numerical transformations. Assume that a double precision program produces a result (which can be checked for accuracy) whose error does not fall below a given threshold $\tau$, and assume that if the entire computation is performed with double-double precision, then the error of the result is below the threshold. In this case,
we start with a code that exhibits a numerical bug and incrementally add precision until the desired accuracy is achieved (note that this is the opposite of the approach used in isolating Mozilla bugs).

Let \( S \) be the set of changes which transforms the double precision code to double-double, including changes of variable declarations, operation types, and function return types (note that \( S \) does not include adding header files or other operations that link the code with the higher precision software). The goal is to find a minimal subset \( S' \) of \( S \) such that performing the changes in \( S' \) to the original double precision code (and compiling and executing) produces a numerical result with error less than \( \tau \). A “minimal” subset is a subset \( S' \subseteq S \) such that no proper subset of changes in \( S' \) will produce a result with the desired accuracy.

Algorithm 1 presents the \( \text{ddmin} \) function which is used to obtain the minimal subset \( S' \) as quickly as possible. In short, it is given as input a set of changes that produce an accurate result, and it tries to find a proper subset of those changes that still produces an accurate result (and then it recurses on the subset). In Algorithm 1, \( \tau \) is the error threshold, and the “error” function takes as input a set of changes, performs those changes, compiles and executes, and outputs the numerical error. The \( \text{ddmin} \) function is first called with the set of all possible changes \( S \) and a granularity of 2. If there exists a singleton change that produces an accurate result, then the algorithm will determine the change in the time of binary search–\( O(\log |S|) \). In the worst case, the complexity is \( O(|S|^2) \) (this complexity does not include the time required to execute the numerical computation).

3 Implementation

Given an implementation of the delta debugging algorithm (written in C++), the double-double precision software package, the CIL tool suite, and an existing implementation of a CIL visitor (written in OCaml) which transformed single precision variables and operations to double precision, my goal was to successfully execute the delta debugging algorithm on a double precision C code that computed a result above the desired accuracy threshold and
find the minimum number of operations that must be performed in double-double precision to achieve a sufficiently small error in the final result.

3.1 Implementation difficulties

The primary difficulty in the implementation was to reconcile the needs of CIL and the double-double software. On the one hand, the CIL tools can translate a C file into CIL in order to perform the transformations, but they cannot handle C++ code. On the other hand, any higher precision library uses operator overloading to ease the transition of double precision codes to use of higher precision types (it eases the transition by hand, at least), and so the header files necessary to compile are inherently C++ codes.

Taking the following steps avoids the C/C++ compilation issues: preprocess the original C file with `gcc`, apply the transformations using CIL tools and generate transformed C code (which is not compilable as it includes unknown type declarations), add `#include` statements for double-double library header files, and then compile using `g++`, linking the double-double library.

The name of the `struct` for storing numbers in double-double precision for this library is `dd_real`. In order to allow for the transformation of the declaration of a variable as a `double` to a `dd_real`, I added the data type `dd_real` to CIL. CIL defines a floating-point type with further classification of its precision, so in fact I added `FDoubleDouble` to the language of `fkind` (the classification of the floating-point type `TFloat`). In this way the CIL tools could generate C code with declarations and typecasts with the keyword `dd_real`.

The double-double library is designed for most codes to work with no further modifications (due to operator overloading) other than adding the header file and linking the library, but I experienced some difficulty in compiling the transformed code because the library did not include a typecast from double-double precision to double. I added this functionality by hand by simply dropping the least significant double value, similar to a cast of a floating-point number to an integer. Note that this implies that the double value returned is not always correctly rounded, but in the case of my examples, casting values as doubles was only used for `printf` statements. Correctly rounded casts to double would require more careful implementation.

3.2 Simple Example

As a simple example, I started with the following C program, `simple.c`:

```c
#include <stdio.h>

int main()
{
    double a = 1e17;
    double b = 1;
    double c = a + b;
    double d = c - a;
    printf(" => total error: %.16e\n", fabs(1.0-d));
    return 0;
}
```

In double precision, this program will compute `1e17 + 1` as `1e17` and then set `d` to be 0. In exact arithmetic (or in double-double precision), `d` is equal to `b`, or 1.
The output of running the delta debugging tool on this code is given in Appendix A. The numerical error occurs at the assignment \(c = a + b\), and two changes are necessary. First, the addition must be done in higher precision, and second, \(c\) must be stored as a double-double to prevent loss of information. The delta debugging algorithm chooses to declare both \(b\) and \(c\) as double-double precision variables, but there are other pairs of changes that would produce an accurate result.

The tool counted 15 possible changes, so to test every possible subset of changes would require \(2^{15}\) tests. Instead, the algorithm found a minimal change in 8 tests. In this simple example, it is easy for a human to spot the source of the numerical inaccuracy, but in the case of a complicated matrix algorithm with a specific anomalous input matrix, determining the operation causing the problem is much more difficult. Localizing the error allows the human debugger to focus her attention on one property of the numerical algorithm and decide if the problem is a programming error or a deficiency of the algorithm itself.

3.3 More Realistic Example

The second example to which I applied the delta debugging tool is taken from [Bai08]. It is an example of a numerical computation that is inherently sensitive to roundoff such that judicious use of higher precision arithmetic will increase the accuracy with relatively little cost in performance. Since double-double arithmetic and other higher precision libraries are software implementations, their performance degrades more quickly than the precision improves (as opposed to the relationship between double and single precision which are both implemented in hardware, where single precision arithmetic can be done twice as fast with half the precision). If double precision is not sufficient for the accuracy required in a certain computation and higher precision is required, it can be very advantageous to performance to apply higher precision in as few computations as possible. We use delta debugging to determine the necessary changes and compare the results to a numerical analyst’s choice.

The computation computes the arc length of an irregular function,

\[ g(x) = x + \sum_{n=0}^{5} 2^{-k} \sin(2^k x) \]

over the interval \((0, \pi)\). The numerical approach is to break the interval into \(\pi/h\) subintervals and sum up the length of the function over each subinterval after linearizing the function over the subinterval. Thus, the computation is summing up the value \(\sqrt{h^2 + (g(x_k + h) - g(x_k))^2}\) over all \(x_k = kh\) in the interval \((0, \pi)\). The original double precision C code funarc.c is given in Appendix B (it is translated from the Fortran-90 code funarc.f90 found on David Bailey’s website (available at the following URL):

http://crd.lbl.gov/~dhbailey/dhbpapers/numerical-bugs.tar.gz

As argued in [Bai08], performing the entire computation in double-double runs about 20 times as slow as double precision (but the error achieved is on the order of double precision, whereas several digits of precision are lost if the original code is used). If the variable used to accumulate the sum is stored and incremented in double-double precision, then full double precision
accuracy is maintained. Since most of the work occurs in the function evaluation, there is only a slight performance penalty.

The delta debugging tool identified 57 possible changes of double to double-double precision. Because there exists a single change that produces sufficient accuracy, the algorithm became a binary search and required only 10 tests before determining the correct change to make. The advantage of this approach is that the user needs no understanding of the numerical properties of the algorithm or implementation. With the delta debugging tool one needs only a mechanism for checking the error of the numerical result (the “correct” answer can be computed using all higher precision arithmetic if necessary).

4 Conclusion/Future Work

Applying the delta debugging approach to numerical programs is an important step in reducing the difficulty of correcting codes that exhibit numerical sensitivities which can arise in various and subtle ways. It can be used by a numerical analyst as a tool to isolate the computations relevant to the inaccuracy, or it can be used by a non-expert to naively improve accuracy at minimal cost to performance. As many code transformations are possible and relevant to numerical debugging (and several have already been implemented by Guillaume Revy), the implementation of double to double-double transformations presented here is an important addition to the general endeavor. With the extra effort of reconciling existing implementations of the delta debugging algorithm and the CIL tools with higher precision libraries, we add the ability to simplify numerical bugs with double precision code and open the possibility of finer control by independently changing precision and range of floating-point numbers.

The next step in this direction will be to apply the tool to isolating actual bugs in the double precision LAPACK routines. Because of the breadth and depth of the interconnections between LAPACK and BLAS routines, the compilation and compatibility difficulties involved in getting the tool to work will be more involved than with the simpler standalone codes tested in this project. As LAPACK bugs are considered and resolved by the small set of numerical analysts in the community (who have plenty of other things to do with their time), I hope this tool will greatly increase the efficiency of their debugging efforts.
References


A  Transformed simple.c code

```c
#include "qd/dd_real.h"
#include "stdio.h"
/* Generated by CIL v. 1.3.7 */
/* print_CIL_Input is true */

extern int printf(char const * __restrict __format , ...);
extern __attribute__((__nothrow__)) double fabs(double __x ) __attribute__((__const__)) ;

int main(void)
{ double a ;
  dd_real b ;
  dd_real c ;
  double d ;
  double tmp ;

  a = 1e17;
  b = (double )1;
  c = a + b;
  d = c - a;
  tmp = fabs(1.0 - d);
  printf((char const * __restrict )" => total error: %1.16e\n", tmp);
  return (0);
}
```

B  Original funarc.c code

```c
#define _GNUCC
#include<math.h>
#include<stdio.h>
#endif
double fun( double x )
{
  int k;
  double ti, di = 1.0;

  ti = x;
  for( k = 1; k <= 5; k++ )
  {
    di = 2.0 * di;
    ti = ti + sin (di * x) / di;
  }

  return ti;
}

int main( int argc, char **argv)
{ int i, j, k, m = 1000000;
  double b, s1, t1, t2, dp, ans = 5.795776322412856;
  long double threshold = 1e-14;
  ```
t1 = -1.0;
dppl = acos(t1);
s1 = 0.0;
t1 = 0.0;
h = dppl / n;

for( i = 1; i <= n; i++ )
{
    t2 = fun(i * h);
    s1 = s1 + sqrt(h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
}

printf(" => total error: %d %1.15Le %1.15Le\n",
    fabs(ans-s1)>threshold, fabs(ans-s1), threshold);

return 0;