Tracking the Maslov Index

David W.H. Swenson

April 2008

Abstract

In what is surely one of the great acts of premature optimization (also known as "fixing something that isn't really broken"), the author presents an optimally fast method for tracking the Maslov index. The time that will be saved by all people using this new method (integrated over the entire future) will doubtless be miniscule compared to the amount of time it took to write this up.

1 The Maslov Index

The Maslov index is used to track how many times a complex function has crossed a branch cut (specifically, the branch cut of the square root function). To briefly explain the meaning of the branch cut, consider the familiar case of squaring real numbers. For example, both $(-4)^2$ and 4^2 are equal to 16. So if you want to take the square root of 16, you could end up with either 4 or -4. Although both are mathematically reasonable, we usually take the positive root (and most computer functions to get the square root choose that one as well). When we generalize the square root to complex numbers, we can use the Maslov index to select the physically meaningful solution.

1.1 Why Track the Maslov Index

We have a situation where we are tracking a complex quantity (the Herman-Kluk prefactor) which is given as the square root of a complex number. We expect this quantity to make a continuous path in the complex plane. However, our complex square root function always takes the root with a positive real part.

This is okay when the argument of the square root is in the first or second quadrant (having started on the positive real axis). However, once it crosses into the third quadrant (that is, once it crosses the branch cut along the negative real axis) the square root jumps from imaginary axis in the first quadrant down to the imaginary axis in the fourth quadrant, instead of continuing into the second quadrant. We can obtain the expected result by multiplying by -1. In fact, we maintain a continuous path if we multiply the result of the square root by a factor of -1 for every time the branch cut has been crossed. The Maslov index is the number of branch cut crossings.

1.2 The Simplest Implementation

The most direct way to implement the above is simply to add one to the Maslov index every time the radicand (`rad`) crosses the branch cut. For the ith step:

```
if ((Real(rad[i])<0.0) && (Imag(rad[i])*Imag(rad[i-1])<0.0)) {
    maslov_index = maslov_index + 1;
}
maslov = (-1)^maslov_index;
num[i] = maslov * sqrt(rad[i]);
```

This can be improved immediately by removing the $(-1)^{\texttt{maslov\_index}}$ line and replacing the interior of the `if`-statement with `maslov = -maslov`, having set `maslov = 1` on initialization of the trajectory.

1.3 Definitions of Terms

Since many of the terms we'll use are very similar, we'd like to define them carefully here.

The `maslov_index` is the count of the number of times we have crossed the branch cut. We distinguish that from `maslov` which we'll use to denote the correct sign of the result, *i.e.*, $\texttt{maslov} = (-1)^{\texttt{maslov\_index}}$.

We should also note the difference between a "branch cut" and a "branch statement." The branch cut is the mathematical idea described above, which is the negative real axis in our case. A branch statement is a computational statement (essentially, one of a set of commands in assembly language), and will be described in section 2.1. The similarity of these two names is an unfortunate coincidence — the concepts are completely unrelated.

2 Computational Ideas

While the pseudocode in the previous section is adequate, it is not the most efficient way to implement tracking of the Maslov index. Without some major help from a very smart compiler, that code is, in fact, one of the worst ways to implement Maslov tracking. To see why, we need to go into some details about what happens at the computer science level.

2.1 Branches in Computation

Modern processors use tricks to line a bunch of instructions up in order to move through them more quickly. Branch statements are situations in a program where there is more than one possible next instruction. For example, an `if` statement causes a branch in the code: one path of instructions is followed if the condition is true; another is followed if not.

Modern processors use sophisticated techniques to try to guess which instruction branch will be followed. However, the guess will occasionally be wrong, and when it is wrong, the line of instructions sent to the processor will be wrong. The processor will have to go back and correct it, and this takes a lot of time.

As an analogy, consider the process of putting grades into a gradebook. If you're like me, you do that by first alphabetizing the graded papers. This is analogous to the processor lining up the instructions. But what if you messed up the alphabetization? Perhaps in some moment of confusion, you thought that K came before H. When you realize this, you have to stop entering the grades (processing instructions) and re-alphabetize the papers (set up a new line of instructions). Just as this can cause a substantial slowdown in recording grades, it can also cause a very substantial performance hit in a computer program. So a general principle of computation is that it is best to implement methods that avoid branch statements.

The actual code I have been using (in C) for my Maslov tracking looks more like:

```
maslov *= (1-2*(Real(rad[i]<0.0) && (Imag(rad[i])*Imag(rad[i-1])<0.0)));
num[i] = maslov * sqrt(rad[i]);
```

Despite the fact that this has removed the "if" statment, it still does not avoid the problem with branches. This is because C uses something called "short-circuit" logic, meaning that the expression after the "`&&`" is only evaluated if the previous expression is true. Therefore, it still needs a branch.[1]

2.2 The IEEE Floating Point Standard

The Institute of Electrical and Electronics Engineers (IEEE) developed a standard method for representing "decimal" numbers on computers. More correctly, these are called "floating point" numbers. For purposes of this article, we only need to know that one of the features of IEEE floating point numbers is the existence of a sign bit. If this bit is 1, then the number is negative. If the bit is 0, the number is positive.

3 The Best Implementation

The last thing to mention is that all operations on computers are built up from simple logical operators (like AND and XOR) and that directly accessing these bitwise logical operations tends to be faster than doing floating point (or even integer) arithmetic.

3.1 The Algorithm

First, we'll generate a mask. The mask should have a value of one in the sign bit, and zero in all other bits. For IEEE-compliant systems, this will be a hexadecimal value of of `0x80...0` (with the appropriate number of zeroes filled in for the specific type of float). For any system with a sign bit, this constant can be determined by the value of (for example) `XOR(-1.0,1.0)`.

Once we have this mask (stored in the constant called `mask`), we can write the general implementation for the Maslov index:

```
r1 = XOR(Imag(rad[i]), Imag(rad[i-1]));
// the sign of r1 is the sign of rad[i]*rad[i-1]

r2 = AND(Read(rad[i]), r1);
// the sign bit of r2 is 1 if we've crossed the branch cut (i.e., if we need
// to increase the maslov index) and 0 otherwise

ifclause = AND(mask, r2);
// result of the if-clause in the sign bit; other bits are zero

maslov = XOR(maslov, ifclause); // switch sign if necessary

num[i] = XOR(maslov, sqrt(rad[i]));
// All bits except the sign bit are zero in maslov, therefore XOR keeps all
// those. For the sign bit, XOR acts as multiplication.
```

The preceding listing is our ideal method for tracking the Maslov index. We have replaced the three comparisons (two "less thans" and one logical "and") and a floating point multiplication with the first two bitwise logical operations. The branch jump has been replaced by another two logical operations, and the final multiplication has also been replaced by a bitwise

[1] I think. Since I haven't actually written my own C compiler, I wouldn't stake my life on that statement. But you get the point.

location operation. In all, we cut two floating point multiplications, a branching statement, and two comparisons down to six bitwise logical operations.

3.2 Computational Results

Someday I'll run timing tests on these, trying VerySlowMaslov (if-then-else with powers), SlowMaslov (if-then-else with multiplication), Maslov (my current code), and FastMaslov (new implementation described herein). I'll also try it with varying levels of optimization in gcc. See how good gcc optimization is. Present the whole thing as a bar graph, where smaller bars are better.