Chapter 1

Numeric Artifacts

1.1 Introduction

Virtually all solutions to problems in electromagnetics require the use of a computer. Even when an analytic or "closed form" solution is available which is nominally exact, one typically must use a computer to translate that solution into numeric values for a given set of parameters. Because of inherent limitations in the way numbers are stored in computers, some errors will invariably be present in the resulting solution. These errors will typically be small but they are an artifact about which one should be aware. Here we will discuss a few of the consequences of finite precision.

Later we will be discussing numeric solutions to electromagnetic problems which are based on the finite-difference time-domain (FDTD) method. The FDTD method makes approximations that force the solutions to be approximate, i.e., the method is inherently approximate. The results obtained from the FDTD method would be approximate even if we used computers that offered infinite numeric precision. The inherent approximations in the FDTD method will be discussed in subsequent chapters.

With numerical methods there is one note of caution which one should always keep in mind. Provided the implementation of a solution does not fail catastrophically, a computer is always willing to give you a result. You will probably find there are times when, to get your program simply to run, the debugging process is incredibly arduous. When your program does run, the natural assumption is that all the bugs have been fixed. Unfortunately that often is not the case. Getting the program to run is one thing, getting correct results is another. And, in fact, getting accurate results is yet another thing—your solution may be correct for the given implementation, but the implementation may not be one which is capable of producing sufficiently accurate results. Therefore, the more ways you have to test your implementation and your solution, the better. For example, a solution may be obtained at one level of discretization and then another solution using a finer discretization. If the two solutions are not sufficiently close, one has not yet converged to the "true" solution and a finer discretization must be used or, perhaps, there is some systemic error in the implementation. The bottom line: just because a computer gives you an answer does not mean that answer is correct.

Lecture notes by John Schneider. `numerical-issues.tex`

1.2 Finite Precision

If we sum one-eleventh eleven times we know that the result is one, i.e., $1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 = 1$. But is that true on a computer? Consider the C program shown in Program 1.1.

Program 1.1 `oneEleventh.c`: Test if $1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11$ equals 1.

```
1  /* Is summing 1./11. ten times == 1.0? */
2  #include <stdio.h>
3
4  int main() {
5    float a;
6
7    a = 1.0 / 11.0;
8
9    if (a + a + a + a + a + a + a + a + a + a + a == 1.0)
10     printf("Equal.\n");
11   else
12     printf("Not equal.\n");
13
14   return 0;
15 }
```

In this program the float variable `a` is set to one-eleventh. In line 9 the sum of eleven `a`'s is compared to one. If they are equal, the program prints "Equal" but prints "Not equal" otherwise. The output of this program is "Not equal." Thus, to a computer (at least one running a language typically used in the solution of electromagnetics problems), the sum of one-eleventh eleven times is not equal to one. It is worth noting that had line 9 been written `a=1/11;`, `a` would have been set to zero since integer math would be used to evaluate the division. By using `a = 1.0 / 11.0;`, the computer uses floating-point math.

The floating-point data types in C or FORTRAN can only store a finite number of digits. On most machines four bytes (32 binary digits or bits) are used for single-precision numbers and eight bytes (64 digits) are used for double precision. Returning to the sum of one-elevenths, as an extreme example, assumed that a computer can only store two decimal digits. One eleventh is equal to 0.09090909... Thus, to two decimal places one-eleventh would be approximated by 0.09. Summing this eleven times yields

$$0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 = 0.99$$

which is clearly not equal to one. If the number is stored with more digits, the result becomes closer to one, but it never gets there. Both the decimal and binary floating-point representation of one-eleventh have an infinite number of digits. Thus, when attempting to store one-eleventh

in a computer the number has to be truncated so that the computer stores an approximation of one-eleventh. Because of this truncation summing one-eleventh eleven times does not yield one.

Since $1/10$ is equal to 0.1, it might appear this number can be stored with a finite number of digits. Although one-tenth has a finite number of digits when written in base ten (decimal representation), it has an infinite number of digits when written in base two (binary representation).

In a floating-point decimal number each digit represents the number of a particular power of ten. Letting a blank represent a digit, a decimal number can be thought of in the follow way:

$$\cdots \underline{\quad} \; \underline{\quad} \; \underline{\quad} \; \underline{\quad} \; . \; \underline{\quad} \; \underline{\quad} \; \underline{\quad} \; \underline{\quad} \cdots$$
$$10^3 \quad 10^2 \quad 10^1 \quad 10^0 \quad\quad 10^{-1} \; 10^{-2} \; 10^{-3} \; 10^{-4}$$

Each digits tells how many of a particular power of 10 there is in a number. The decimal point serves as the dividing line between negative and non-negative exponents. Binary numbers are similar except each digit represents a power of two:

$$\cdots \underline{\quad} \; \underline{\quad} \; \underline{\quad} \; \underline{\quad} \; . \; \underline{\quad} \; \underline{\quad} \; \underline{\quad} \; \underline{\quad} \cdots$$
$$2^3 \quad 2^2 \quad 2^1 \quad 2^0 \quad\quad 2^{-1} \; 2^{-2} \; 2^{-3} \; 2^{-4}$$

The base-ten number 0.1_{10} is simply 1×10^{-1}. To obtain the same value using binary numbers we have to take $2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \ldots$, i.e., an infinite number of binary digits. Another way of writing this is

$$0.1_{10} = 0.0001100110011001100110011\ldots_2 .$$

As before, when this is stored in a computer, the number has to be truncated. The stored value is no longer precisely equal to one-tenth. Summing ten of these values does not yield one (although the difference is very small).

The details of how floating-point values are stored in a computer are not a primary concern. However, it is helpful to know how bits are allocated. Numbers are stored in exponential form and the standard allocation of bits is:

	total bits	sign	mantissa	exponent
single precision	32	1	23	8
double precision	64	1	52	11

Essentially the exponent gives the magnitude of the number while the mantissa gives the digits of the number—the mantissa determines the precision. The more digits available for the mantissa, the more precisely a number can be represented. Although a double-precision number has twice as many total bits as a single-precision number, it uses 52 bits for the mantissa whereas a single-precision number uses 23. Therefore double-precision numbers actually offer more than twice the precision of single-precision numbers. A mantissa of 23 binary digits corresponds to a little less than seven decimal digits. This is because 2^{23} is $8{,}388{,}608$, thus 23 binary digits can represent numbers between 0 and $8{,}388{,}607$. On the other hand, a mantissa of 52 binary digits corresponds to a value with between 15 and 16 decimal digits ($2^{52} = 4{,}503{,}599{,}627{,}370{,}496$).

For the exponent, a double-precision number has three more bits than a single-precision number. It may seem as if the double-precision exponent has been short-changed as it does not have twice as many bits as a single-precision number. However, keep in mind that the exponent represents the size of a number. Each additional bit essentially doubles the number of values that can be represented. If the exponent had nine bits, it could represent numbers which were twice as large

as single-precision numbers. The three additional bits that a double-precision number possesses allows it to represent exponents which are eight times larger than single-precision numbers. This translates into numbers which are 256 times larger (or smaller) in magnitude than single-precision numbers.

Consider the following equation

$$a + b = a.$$

From mathematics we know this equation can only be satisfied if b is zero. However, using computers this equation can be true, i.e., b makes no contribution to a, even when b is non-zero.

When numbers are added or subtracted, their mantissas are shifted until their exponents are equal. At that point the mantissas can be directly added or subtracted. However, if the difference in the exponents is greater than the length of the mantissa, then the smaller number will not have any affect when added to or subtracted from the larger number. The code fragment shown in Fragment 1.2 illustrates this phenomenon.

Fragment 1.2 Code fragment to test if a non-zero b can satisfy the equation $a + b = a$.

```
1   float a = 1.0, b = 0.5, c;
2
3   c = a + b;
4
5   while(c != a) {
6     b = b / 2.0;
7     c = a + b;
8   }
9
10  printf("%12g %12g %12g\n",a,b,c);
```

Here a is initialized to one while b is set to one-half. The variable c holds the sum of a and b. The while-loop starting on line 5 will continue as long as c is not equal to a. In the body of the loop, b is divided by 2 and c is again set equal to $a + b$. If the computer had infinite precision, this would be an infinite loop. The value of b would become vanishingly small, but it would never be zero and hence $a + b$ would never equal a. However, the loop does terminate and the output of the `printf()` statement in line 10 is:

```
           1   5.96046e-08   1
```

This shows that both a and c are unity while b has a value of 5.96046×10^{-8}. Note that this value of b corresponds to 1×2^{-24}. When b has this value, the exponents of a and b differ by more than 23 (a is 1×2^0).

One more example serves to illustrate the less-than-obvious ways in which finite precision can corrupt a calculation. Assume the variable a is set equal to 2. Taking the square root of a and then squaring a should yield a result which is close to 2 (ideally it would be 2, but since $\sqrt{2}$ has an infinite number of digits, some accuracy will be lost). However, what happens if the square root is

taken 23 times and then the number is squared 23 times? We would hope to get a result close to two, but that is not the case. The program shown in Program 1.3 allows us to test this scenario.

Program 1.3 `rootTest.c`: Take the square root of a number repeatedly and then square the number an equal number of times.

```
/* Square-root test. */
#include <math.h>  // needed for sqrt()
#include <stdio.h>

#define COUNT 23

int main() {
  float a = 2.0;
  int i;

  for (i = 0; i < COUNT; i++)
    a = sqrt(a);  // square root of a

  for (i = 0; i < COUNT; i++)
    a = a * a;       // a squared

  printf("%12g\n",a);

  return 0;
}
```

The program output is one, i.e., the result is $a = 1.0$. Each time the square root is taken, the value gets closer and closer to unity. Eventually, because of truncation error, the computer thinks the number is unity. At that point no amount of squaring the number will change it.

1.3 Symbolic Manipulation

When using languages which are typically used in numerical analysis (such as C, C++, FORTRAN, or even MATLAB), truncation error is unavoidable. The ratio of the circumference of a circle to its diameter is the number $\pi = 3.141592\ldots$ This is an irrational number with an infinite number of digits. Thus one cannot store the exact numeric value of π in a computer. Instead, one must use an approximation consisting of a finite number of digits. However, there are software packages, such as Mathematica, that allow one to manipulate symbols. Within Mathematica, if a person writes `Pi`, Mathematica "knows" symbolically what that means. For example, the cosine of `10000000001*Pi` is identically negative one. Similarly, one could write `Sqrt[2]`. Mathematica knows that the square of this is identically 2. Unfortunately, though, such symbolic manipulations are incredibly expensive in terms of computational resources. Many cutting-edge

problems in electromagnetics can involve hundreds of thousand or even millions of unknowns. To deal with these large amounts of data it is imperative to be as efficient—both in terms of memory and computation time—as possible. Mathematica is wonderful for many things, but it is not the right tool for solving large numeric problems.

In MATLAB one can write `pi` as a shorthand representation of π. However, this representation of π is different from that used in Mathematica. In MATLAB, `pi` is essentially the same as the numeric representation—it is just more convenient to write `pi` than all the numeric digits. In C, provided you have included the header file `math.h`, you can use `M_PI` as a shorthand for π. Looking in `math.h` reveals the following statement:

```
# define M_PI           3.14159265358979323846  /* pi */
```

This is similar to what is happening in MATLAB. MATLAB only knows what the numeric value of `pi` is and that numeric value is a truncated version of the true value. Thus, taking the cosine of `10000000001*pi` yields -0.99999999999954 instead of the exact value of -1 (but, of course, the difference is trivial in this case).