What Your Computer's Doing

Mech 511

1 Floating Point Arithmetic

When we think about arithmetic, we typically think in terms of infinite precision numbers. For instance, we know that

$$\frac{1}{3}+\frac{1}{3}+\frac{1}{3}+\frac{1}{3}+\frac{1}{3}+\frac{1}{3}=2$$

exactly, because we know about fractions. However, if you tell a computer to evaluate this, using double precision arithmetic[1], you'll get 1.99999999999999778. More precisely, you'll get $2-2^{-52}$. That's because your computer can't represent 1/3 exactly in binary using a finite number of digits, just as we can't in decimal.

1.1 A Decimal Floating Point Model

As an easier-to-follow analog to computer floating-point arithmetic, let's use a really simple system in which we stick with decimal number, but with a twist. We can only write down four digits, *a.bcd*, with a two-digit exponent; each part can have a sign attached. In other words, we can write $-1.234 \cdot 10^{56}$, or $3.142 \cdot 10^{0}$. That last is the closest we can get to π. So the largest number we can represent is $9.999 \cdot 10^{99}$, and the smallest positive number is $1.000 \cdot 10^{-99}$. Actually, if we don't insist on having a non-zero digit to the left of the decimal point, we can go as close to zero as $0.001 \cdot 10^{-99}$; this is called a *de-normalized number.*

There are a lot of number we can't represent exactly, like π. We can't even represent 10,005 exactly! When we convert a real number to our decimal floating point system, we have to round, like we did with π. In that case, we weren't exactly halfway between two floating point numbers; we would need a rule to tell us whether 10,005 should be represented as $1.000 \cdot 10^{4}$ (10,000) or $1.001 \cdot 10^{4}$ (10,010). For now, let's pick "round to nearest": we'll round representations that are too long to the next larger number, as we did with π, above.[2]

So what happens when we try to represent fractions (whose denominators aren't only powers of 2 and 5)? Let's try with $\frac{1}{3}$. When we write this as a decimal, we get

$$\frac{1}{3}=0.333\overline{3}$$

[1](C double or Fortran REAL*8 and its descendants)

[2]Other options in the IEEE floating point standard are round up (towards $+\infty$, regardless of the sign of the number), round down (towards $-\infty$), and round towards zero,

which in our floating point system would be $3.333 \cdot 10^{-1}$. In this case, we round down. On the other hand, 2/3 would round up to $6.667 \cdot 10^{-1}$. What happens if we add

$$\begin{aligned}\frac{1}{3}+\frac{1}{3}+\frac{1}{3} &= 3.333 \cdot 10^{-1} + 3.333 \cdot 10^{-1} + 3.333 \cdot 10^{-1} \\ &= 9.999 \cdot 10^{-1}\end{aligned}$$

Yes, that's right. Because of *roundoff error*, we end up not getting exactly 1. We're off by (in this case) one *unit in the least-significant place* (ULP). This quantity (10^{-4}) is the formal definition of machine ϵ: the smallest number which, when subtracted from 1, the result can be distinguished from 1. An alternate definition, also in wide use, is that machine ϵ is the smallest number which, when *added* to 1, the result can be distinguished from 1; for our system, that would be 10^{-3}: $1.000 \cdot 10^0 + 1.000 \cdot 10^{-3} = 1.001 \cdot 10^0$.

Now let's try some more arithmetic. Suppose we try to add

$$\frac{7}{6}+\frac{1}{7}=\frac{55}{42}$$

In floating point arithmetic, we'll have $1.167 \cdot 10^0 + 1.429 \cdot 10^{-1}$, or

$$\begin{array}{r} 1.167 \\ \underline{+0.1429} \\ 1.3099 \end{array}$$

which we'd represent as $1.310 \cdot 10^0$. A couple of points are in order here. First, this is actually the closest representation of 55/42, which is good. Second, internally, your CPU carries extra bits, just as we kept an extra digit and rounded at the end, when we "stored" the result.[3] A lot of the time, floating point arithmetic gives us the right value for as many digits as we can possibly get, but not always, as we saw when we added 1/3 three times.

Multiplication is also challenging, because the extra bits it produces can cause rounding the wrong way. Suppose we try

$$\frac{7}{6} \times \frac{1}{7} = \frac{1}{6}$$

In our floating point system, we'll get

$$1.167 \times 0.1429 = 0.1667643$$

which will round to $1.668 \cdot 10^{-1}$, which is slightly larger (1 ULP) than the best representation for 1/6. It's also easy to get roundoff error the other way; for instance, $\frac{4}{3} \times \frac{2}{7}$ will round to be 1 ULP too low.

Suppose that we wanted to compute 70! (that's $1 \times 2 \times 3 \times \cdots \times 69 \times 70$). The value for this, to ten significant digits, is $1.197857167 \cdot 10^{100}$. We can't represent this number, because we only have two digits of exponent; this is an example of *overflow*. Similarly, we can't represent its inverse (which is $8.348240738 \cdot 10^{-101}$) with a non-zero to the left of the decimal point. But we could write it as a de-normalized number: $0.083 \cdot 10^{-99}$, with a reduction in precision. Eventually, though, we run out of digits entirely: if we divide now by 100, we get a number too small to represent, even as a de-normalized number; this is called *underflow*.

[3]In fact, Intel and AMD CPU's use 80-bit floating point numbers internally and convert to single or double precision when writing to memory.

1.2 Computer Floating Point Representation

Your computer represents numbers similarly, but in binary instead of decimal. The IEEE floating point standard defines two commonly implemented types of floating point numbers, single precision (float) and double precision (double). Each represents floating point numbers the same way:

$$\pm 1.[\text{digits of mantissa}]\cdot 2^{\pm\text{digits of exponent}}$$

In practice, the exponent is written as a positive binary integer, and a bias is automatically subtracted from it in use. The storage order is: sign, biased exponent, mantissa; this means that double precision numbers often start with 7F (positive) or FF (negative) when you look at their hexadecimal representation.

The ranges of values are:

Precision	Single	Double
Size of mantissa	23 bits	52 bits
Size of exponent	8 bits	11 bits
Exponent bias	127	1023
Largest number	$\pm 10^{38.5}$ $\pm 2^{128}$	$\pm 10^{308.3}$ $\pm 2^{1024}$
Smallest normalized number	$\pm 2^{-126}$	$\pm 2^{-1022}$
Smallest denormalized number	$\pm 2^{-149}$	$\pm 2^{-1074}$
Machine ϵ: $1+\epsilon \neq 1$	$2^{-23} \approx 1.19 \cdot 10^{-7}$	$2^{-52} \approx 2.22 \cdot 10^{-16}$

That value for machine ϵ explains why using single precision in CFD can lead to problems: you may need to converge more deeply than that to get accurate values for some quantity of interest. There are even situations in which double precision roundoff matters; unstructured mesh generation is one area where this comes up a lot.

The IEEE floating point standard also gives specific ways to represent $\pm\infty$ and invalid results (like $\infty \times 0$ or $\sqrt{-2}$: the infamous not-a-number result). The internal representation of these isn't important for us, but it is important to know that these unfortunate values can be represented.

2 Building Your Program

If you're using an integrated development environment, much of this will be done without your direct intervention, but it's still useful to understand what's going on.

2.1 From Source Code to Executable Program

Your compiler actually goes through (at least) four stages to turn your high-level language source code into an executable program. We often refer to the three stages in that process from C source code to an object file as "compilation", even though strictly speaking compilation is only one stage of that process.

2.1.1 Pre-processing

The first step in building your program is to pre-process the source code. All the lines in your code that start with `#` are actually directives for the preprocessor.[4] These fall into several common categories:

- `#include` directives are replaced by the full text of the file that's named in the directive. The pre-processor is run on the contents of the included file as well. There are two flavors of include directives:

  - `#include <filename.h>` (or `#include <filename>`, for C++ files) should be used for system header files, like `stdio.h`, `algorithm`, `math.h`, etc.
  - `#include “filename.h”` should be used for user-space header files.

  The difference between the two is the search path used. Your pre-processor has a list of directories in which it searches for system header files. You can also define a list of additional directories to search for user-defined header files. If you use `#include <...>`, the pre-processor will skip checking the additional directories.

- `#define` / `#undef` directives. These declare macros, with the first token after define being the name, and the rest of the line being the body of the macro. A macro can be as simple as

  ```
  #define NMAX 20
  ```

  or it can define whole blocks of code or even functions, including arguments. Regardless of what the definition is, the pre-processor will replace all later uses of the name of the macro with its definition (whole tokens only: so `int T[NMAX]` $\rightarrow$ `int T[20]`, but `int NMAX_SQUARED` is left alone).

  `#undef NMAX` would remove any previous definition.

  Be careful! First, you need to avoid unintended consequences. For instance,

  ```
  #define NCELLS IMAX*JMAX
  double norm = sum / NCELLS;
  ```

  expands the second line to

  ```
  double norm = sum / IMAX*JMAX;
  ```

  when you probably want

  ```
  double norm = sum / (IMAX*JMAX);
  ```

  So any time you do any arithmetic operations in a macro, be sure to wrap it in parentheses. Similarly, it's pretty common practice to write blocks of macro code like this:

[4]This is a lie in the service of the truth. There are some exceptions, which we'll come back to later in the term.

```
#define VECTOR_MAG(vec, n, result) do { \
  int ii_; \
  double sum_ = 0; \
  for (ii_ = 0; ii_ < n; ii_++) { \
    sum_ += vec[ii_]*vec[ii_]; \
  } \
  result = sqrt(sum_); \
} while (0)
```

Note the backslash (\) at the end of each line; this makes the whole block a single line, so the pre-processor will substitute all of it each time `VECTOR_MAG` appears. Second, having the whole things in a `do { } while (0)` makes it so that you have a context in which to define local variables and also makes it so that when you naturally type `VECTOR_MAG(T, NMAX, magT);` you'll get ... `while (0);` , which is a well-formed C/C++ statement.

- Conditional compilation. For example,

```
#ifdef HAVE_BOOL
  bool isConverged = false;
#else
  #define true 1
  #define false 0
  int isConverged = false;
#endif
```

or

```
#if LIBRARY_VERSION > 5
  solveProblem(soln, N, alpha);
#else
  double beta = 3;
  solveProblem(soln, N, alpha, beta);
#endif
```

When the pre-processor is done, it will have included all files and expanded all macros. The result will be a (much longer) stream of C/C++/Fortran code that is fed to the compiler proper.

2.1.2 Compiling

The compiler's job is to take the high-level language code and convert it to assembler source code.

Your source code contains many things: variable names, flow control words (like `for` and `if`), type names (`int`, `double`, etc), other reserved words (`class`, `const`, etc), and constant values (`M_PI`, `42`, etc). These are called *tokens*; they are analogous to words in human languages. Just like human languages, computer languages have a grammar that describes the correct statements (analogous to sentences), blocks (a for loop, for instance, or a subroutine; analogous to paragraphs) and programs (analogous to a book). One important difference is that computer languages have grammars that are explicitly defined and have no exceptions.[5]

The compiler's job is to take the character soup it gets in a high-level language and turn it into hardware-specific assembler code. There are several stages in this process, and it can be quite complicated, but simplest version is roughly this:

- Parsing the source code into tokens. That is, read sequentially through the source code and identify that

  ```
  int i = 3;
  ```

  has five tokens: the type identifier "`int`"; the variable name "`i`"; the assignment operation "`=`"; the constant "`3`"; and the end-of-statement marker "`;`". And then do this again for the next line, and the next.

- Using lexical analysis, put those tokens (words) together into a coherent story: statements (sentences), blocks, functions, and finally a program.

- At this point, your program is typically represented inside the compiler using a tree structure, where each branching in the tree is an operation and each next-level item is an operand. In the simple example above, "`=`" (assignment) is the operation, and "`i`" and "3" would be operands. Your compiler can, on request, optimize this tree for faster execution, smaller memory usage, or both. In the former case (which is generally the one we care about — smaller memory usage won't make big arrays small, because you need all that data at the same time, typically), the compiler will re-order operations, eliminate redundant calculations, etc, to reduce the computational cost of the assembler code it's about to produce.

- Finally, the compiler writes out assembler code. This is written to a temporary file or pipe, unless you specifically request that it be written to a permanent file. The instructions in the assembler code are extremely low level. For instance, the C source code

  ```
  void simple(int array[]) {
    int i, j = 0;
    for (i = 0; i < 10; i++) {
      j += i;
      array[i] = j;
    }
  }
  ```

[5]Though there are some places where computer language specifications say, "If you do this, the behavior of the program is undefined." Avoid those situations, because your program almost certainly won't work reliably.

compiles to produce this assembly code:

```
simple:
.LFB0:
.cfi_startproc
xorl %eax, %eax
xorl %edx, %edx
.p2align 4,,10
.p2align 3
.L2:
addl %eax, %edx
movl %edx, (%rdi,%rax,4)
addq $1, %rax
cmpq $10, %rax
jne .L2
rep ret
.cfi_endproc
```

2.1.3 Linking

After the assembler is done, you have an object file. If you're only compiling a single source file, you never see that object file, because the compiler then links it immediately to system libraries and produces the final executable. If you've got a larger project, with multiple source files, your compiler can easily be told to produce an object file for each (the -c compiler option typically does this). Then you have to link them together to get an executable program. What the linker does is identify where it can find functions you refer to but don't define in the same file, and replace calls to those functions with actual address information, telling where that function can be found in the binary image of the program in memory (relative to the beginning of the programs).

This mechanism also allows you to resolve library functions (including favorites like `cos`, `sqrt`, and `printf`). Your linker can work in two modes for library functions: static and dynamic linkage. With static linkage, copies of all the library functions are included in the executable; this makes the executable larger, but also makes it more portable, as you don't need to have the library available at run time. With dynamic linkage, the linker makes a note of which library resolves a function call, but doesn't copy the function into the executable; in this scenario, the shared library is read at run time to retrieve a copy of the function. Dynamic linkage makes for smaller executables, but the libraries must all be present (and in the appropriate search path) at run time.

2.2 An Example

My source code for the final project from Mech 510 consists of four source files: `block_tri.c`, `fluxes.c`, `implicit.c`, and `ins2d.c`. Leaving aside functions declared static (which be definition aren't visible outside the file they're defined in), these files define and use the following functions:

File	Defined	Using	
block_tri.c	vSolveBlockTri		
fluxes.c	vResid		
implicit.c	vApproxFactorPass	cos	vResid
	vColumnLHS	sqrt	vSolveBlockTri
	vRowLHS	calloc	
	vGhost		
ins2d.c	main	calloc	puts
		fclose	sin
		fopen	sincos
		fputc	sqrt
		free	stderr
		fwrite	vApproxFactorPass
		log10	vGhost

To compile this, I run:

```
gcc -O3 -DNDEBUG -Wall -c ins2d.c
gcc -O3 -DNDEBUG -Wall -c fluxes.c
gcc -O3 -DNDEBUG -Wall -c implicit.c
gcc -O3 -DNDEBUG -Wall -c block_tri.c
gcc -o ins2d ins2d.o fluxes.o implicit.o block_tri.o -lm
```

The first four lines compile the four source files and produce `ins2d.o`, `fluxes.o`, `implicit.o`, and `block_tri.o`. The last line links the four together to produce the executable `ins2d`. The `-lm` at the end links in the math library. (The standard C library is linked automatically.)

2.3 Automating the Build Process

Nobody wants to have to type even the five lines I just showed over and over again. Not only that, but in complex programs with lots of include files (and include files including include files, etc), you can't always easily tell which files need to be re-compiled. This is where a tool called `make` comes in; your IDE is using `make` or something similar behind the scenes, even if you're unaware of that. With make, you create a set of rules describing the dependencies in your code (which files should be updated if they're older than which other files) and rules describing how to build a file. These rules are usually stored in a file called `Makefile`; other names can be used, but this is the file `make` looks for by default.

We'll skip the details of `make` syntax, and instead look at a simple example. My `Makefile` for the 510 final project looks like this:

```
ins2d: ins2d.o fluxes.o implicit.o block_tri.o
    gcc -o ins2d ins2d.o fluxes.o implicit.o block_tri.o -lm

clean:
    -rm -f *~ *.o *junk*
```

```
.c.o:
    gcc -O3 -DNDEBUG -Wall -c $*.c

depend:
    makedepend -I. -Y. *.c 2> /dev/null

# DO NOT DELETE

block_tri.o: defines.h momentum.h
fluxes.o: defines.h momentum.h
implicit.o: defines.h momentum.h
ins2d.o: momentum.h defines.h
validate.o: momentum.h defines.h
```

The first rule tells how to build `ins2d` from the object files. The list of files after the ":" are files that `ins2d` depends on; if one of those has changed, `ins2d` must be updated.

The second rule (`clean`) is a housekeeping thing, making it easy to clean up extraneous files, and to force a complete rebuild, because it deletes all the object files.

The third rule (`.c.o`) is a *suffix rule*, so called because it specifies how to turn any C source file (ending in `.c`) into an object file (ending in `.o`). Yes, I do turn on warnings (`-Wall`); in fact, usually I turn on more than this, because `-Wall` doesn't actually turn on *all* of gcc's warnings. If your compiler is warning you about something, that's a sign your syntax is questionable or ambiguous (to humans; to the compiler, there's only one possible meaning). You should fix *all* of these, unless they're warnings about things in system include files that you have no control over.

The fourth rule (depend) tells how to automatically check for the dependencies of files you're compiling. `-I.` tells `makedepend` to look for `#include "..."` files in the current directory. `-Y.` tells it the same thing for `#include <...>` files; this prevents `makedepend` from cluttering your list of dependent files (the stuff below "`# DO NOT DELETE`", which is where `makedepend` puts its output) with system files that aren't changing anyway. Finally, "`2> /dev/null`" diverts error output so that it doesn't print on-screen.

So to be sure my executable `ins2d` is up-to-date with my source code, I simply type

```
make ins2d
```

While make is used most frequently for compiling and linking programs, that's not it's only possible use. I've used it a fair bit to take output files and automatically generate plots. If you have a script that can create the plot, then it's simple enough to tell make that the output plot file depends on a list on input data files, and to run a specific script to produce the new plot.