Lesson 2

CSPP58001

January 13, 2013

1 Floating point

In lesson 1 we focused on solution methods for simultaneous systems of linear equations. The basic idea was to rewrite the linear system as an equivalent upper diagonal system via a sequence of operations that were guaranteed to preserve the solution. As we saw this algorithm – called *Gaussian Elimination* (GE) – was simple to implement in any scalar or array based language. Gaussian elimination is not the beginning and end of all direct solvers, but it is often good enough, and the same basic ideas underlie a number of more sophisticated techniques.

We also mentioned that with GE, like most algorithms we'll study in this class. their a new set of issues arise when we consider their accuracy on a digital computer. That is, most decimal numbers cannot be represented exactly on a digital computer, and most numerical operations are subject to *roundoff error*. When carrying out GE small errors can accumulate quickly and the result can be highly inaccurate. This is a loose conceptual description of the problem. We quantify the ideas of *stability* and *accuracy* later in the course, when we have a broader theoretical basis to build upon. For now I hope to convey qualitatively that roundoff error is a significant issue that must be addressed within the algorithm itself. For GE our rules of thumb were at minimum partial pivoting to bring the largest column value into the next pivot location.

1.1 Integer representation

Before discussing floating point representation we start with the basic integer representation that underlies it. On a base-2 digital computer, integers can be represented exactly within a given range. For example, in C the following are typical

type	value range	bits
char:	$-128 \leq x \leq 127$	8
short:	$-32,768 \leq x \leq 32,767$	16
uchar:	$0 \leq x < 255$	8
int:	$-2,147,483,648 \leq x < 2,147,483,647$	32

One issue that is easily dealt with but the programmer often must be aware of whether a specific processor uses *big endian* vs. *little endian* layout. In big endian, the most significant byte of a multi-byte numeric representation has the lowest address, such as we typically write numbers on paper. For example the signed int 255 would look like:

00000000 00000000 00000000 11111111 (big endian)
11111111 00000000 00000000 00000000 (little endian)

There are endless debates about which format is better. For the most part, though, for the purposes of scientific computing, this is just a detail that we need to be aware of to ensure portability of our algorithms. Further details on integer representation can easily be worked out on the piece of paper or with the help of hundreds of text or web references. There is no sense in reproducing much detail here.

In terms of carrying out integer manipulations, a few simple things should be kept in mind that are relevant for our discussion of floating point. First, bit shifting is equivalent to multiplying or dividing by 2 (this follows simply from the definition of a binary number).

$$\begin{aligned} a << b &: a \times 2^b \\ a >> b &: a \times 2^{-b} \end{aligned} \tag{1}$$

where the $>>$ notation is commonly used for ”bit-shift” in popular programming languages (C, Python, Java, etc).

Another reminder is that bit-wise operations (typically ’&’ for bitwise-and and ’||’ for bitwise-or) can be used for lower level control.

Finally, bitwise addition of two binary integers is exact. We follow the same process as with decimal addition

$$\begin{aligned} 0+0 &\rightarrow 0 \\ 1+0 &\rightarrow 1 \\ 0+1 &\rightarrow 1 \\ 1+1 &\rightarrow 0(carry) \end{aligned} \tag{2}$$

For example, to add $7 + 9 = 16$ in binary we do:

$$\begin{array}{rr} & 0111_2 \\ + & 1001_2 \\ \hline & 10000_2 \end{array}$$

If you never have before, it's a good idea to play around for a few hours in binary representation to get accustomed to its basic properties. On the dropbox site I've included a simple C code to print the bit pattern of any arbitrary integer type.

2 Floating point representation

The more relevant issue for Numerical Methods is the approximate nature of decimal representation. First, we note that *decimal* and *floating point* are often used interchangeably to refer to the internal representation of non-integral values. As we'll see floating point is one common technique to represent decimals (but not the only way).

Unlike integers, most decimal values cannot be represented exactly on a digital computer. Furthermore, even for the small subset of numbers that can be represented exactly, the result of numerical operations is typically not exact. Often these errors can be hidden by using very high precision numerical representation (e.g. 8 or 16-byte), but potential problems lurk in many places when carrying out typical numerical operations, and errors that are small and not apparent can accumulate and become non-trivial very quickly. Understanding the limitations of floating point is a key aspect of being a credible computer modeler.

First, we look at how decimal values are represented in binary. The most common approach is to represent decimal numbers in binary scientific notation:

$$s \times M \times B^{e-E}$$

s: single sign bit
M: "mantissa": integer representation of number with magnitude removed
B: "base" : typically 2
e: "exponent":
E: "bias"

To make this more concrete let's focus on a specific standard (the most common and most pratically relevant) – ieee 32-bit single precision:

ieee 32-bit
s: 0 or 1
M: 23-bit *normalized*
B: 2
e: 8-bit
E: 127

Example: Represent 63.25 in 32-bit ieee floating point:

- First, write down 63.25 in standard base 2:
  ... 0 0 1 1 1 1 1 1 . 0 1 0 0 ...
- Next, normalize the representation
  1 . 1 1 1 1 1 0 1 $\times 2^5$
- $e - E = 5 \rightarrow e = 132$
- s=0 (since positive)
- M = (1) 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

note that the first 1 (in parentheses) will always be there (since we normalize in ieee floating point) and thus it is not explicitly stored. this gives us one extra bit of precision in the mantissa. Putting this all together we have

$$63.25 \rightarrow 0 \ \ 10000100 \ \ 11111010000000000000000$$

In this case we conveniently chose a number with an exact base-2 representation (the fraction is $.25 = 2^{-2}$). In general this won't be the case:

Example: Represent 100.10 in 32-bit ieee floating point:
$100 \rightarrow 1100100$
$.10 \rightarrow .00011001100110011... \equiv .0\overline{0011}$
$1100100.0\overline{0011} \rightarrow 1.1001000\overline{0011} \times 2^6$
$e = 133$
$s = 0$
$M = 10010000011001100110011$
$100.10 \rightarrow 0 \ \ 10000101 \ \ 10010000011001100110011$

Question: how do we approximate decimal .1 in binary notation?

(a) Begin with the decimal fraction and multiply by 2. The whole number part of the result is the first binary digit to the right of the point.

Because .1 x 2 = 0.2, the first binary digit to the right of the point is a 0. So far, we have .1 (decimal) = .0??? . . . (base 2) .

(b) Next we disregard the whole number part of the previous result (0 in this case) and multiply by 2 once again. The whole number part of this new result is the second binary digit to the right of the point. We will continue this process until we get a zero as our decimal part or until we recognize an infinite repeating pattern.

Because .2 x 2 = 0.4, the second binary digit to the right of the point is also a 0. So far, we have .1 (decimal) = .00?? . . . (base 2) .

(c) Disregarding the whole number part of the previous result (again a 0), we multiply by 2 once again. The whole number part of the result is now the next binary digit to the right of the point.

Because .4 x 2 = 0.8, the third binary digit to the right of the point is also a 0. So now we have .1 (decimal) = .000?? . . . (base 2) .

(d) We multiply by 2 once again, disregarding the whole number part of the previous result (again a 0 in this case).

Because .8 x 2 = 1.6, the fourth binary digit to the right of the point is a 1. So now we have .1 (decimal) = .0001?? . . . (base 2) .

(e) We multiply by 2 once again, disregarding the whole number part of the previous result (a 1 in this case).

Because .6 x 2 = 1.2, the fifth binary digit to the right of the point is a 1. So now we have .1 (decimal) = .00011?? . . . (base 2) .

(f) We multiply by 2 once again, disregarding the whole number part of the previous result. Let's make an important observation here. Notice that this next step to be performed (multiply 2. x 2) is exactly the same action we had in step 2. We are then bound to repeat steps 2-5, then return to Step 2 again indefinitely. In other words, we will never get a 0 as the decimal fraction part of our result. Instead we will just cycle through steps 2-5 forever. This means we will obtain the sequence of digits generated in steps 2-5, namely 0011, over and over. Hence, the final binary representation will be.

.1 (decimal) = .00011001100110011 . . . (base 2) .

The repeating pattern is more obvious if we highlight it in color as below:

.1 (decimal) = .00011001100110011 . . . (base 2) .

2.1 Adding and subtracting floating point numbers

How are floating point numbers added? The numbers are first written with the same exponent and the mantissas are added. To make our example simpler we'll make up an 8-bit floating point format with

$$s = \pm 1$$
$$B = 2$$
$$E = 7$$
$$e = 4bits$$
$$M = 3bits$$

Example Add the two 1-byte binary floating point values
0 1001 110
0 0111 000
We start by rewriting exposing the hidden bit and explicitly expressing the exponent:
$+1.110 \times 2^2$
$+1.000 \times 2^0$
Then, shift the exponent of the smaller term to match the exponent of the larger term:
$.01000 \times 2^2$
Now, add the mantissas:

$$\begin{array}{rr} & 1.110_2 \\ + & 0.010_2 \\ \hline & 10.000_2 \end{array}$$

Finally, renormalize:
1.00×2^3

We will go over several such examples in class and demonstrate to what extent precision can be lost when e.g. adding numbers of very different orders of magnitude. Considering this idea more generally is the subject of one of your homework problems, so we won't discuss it in more depth here. The main ideas, though, should be apparent. It is critical to realize that your floating point operations are approximations, and that those approximations can diverge from the true answer over many operations if you're not careful.

3 Applications of Linear Systems

Here we move away from solution techniques and consider some fun situations where linear systems arise. The first is a toy problem designed to shed light on linear independence and new concepts such as *reduced row echelon form* and the *rank* of a matrix.

3.1 A Puzzle

Example: Consider the following 3x3 matrix:

$$\begin{bmatrix} 8 & ? & ? \\ ? & 5 & ? \\ ? & 1 & ? \end{bmatrix}$$

To solve this puzzle, complete the matrix according to the following rules:

1. All rows, columns and diagonals sum to the same quantity

2. All elements together sum to 45.

Even at 3x3 the solution to the puzzle is probably difficult to do by inspection or trial and error (and completely impractical for anything reasonably large). Instead, we write a linear system and use elimination to solve for the solution.

This is a good exercise because it forces one to confront what is meant by "linear independence". In expressing the constraints mathematically, we know that 6 independent conditions are needed to solve for the six unknowns. What is independent isn't always as obvious as you might suspect – some conditions that appear to be independent may actually imply one another. Fortunately, there is a systematic way to address this issue using elimination. Performing full Gauss-Jordan elimination on A (eliminating above and below diagonal) automatically tells us how many independent equations we have and, in the case of an undetermined system, properties of the family of solutions. This process produces what is referred to as the *reduced row echelon form* of A. We won't go into great depth here but illustrate the main concept via this example.

Let's try to express the constraints. First, we identify our unknowns in the regular way. To be complete we identify all 9 and express the three known ones as part of the linear system:

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}$$

There is no unique way to express our constraints, but we try a reasonable approach.

$$\begin{aligned}
x_1+x_2+x_3+x_4+x_5+x_6+x_7+x_8+x_9 &= 45 \\
x_1+x_2+x_3-x_4-x_5-x_6 &= 0 \\
x_4+x_5+x_6-x_7-x_8-x_9 &= 0 \\
x_7+x_8+x_9-x_1-x_4-x_7 &= 0 \\
x_1+x_4+x_7-x_2-x_5-x_8 &= 0 \\
x_2+x_5+x_8-x_3-x_6-x_9 &= 0 \\
x_3+x_6+x_9-x_1-x_5-x_9 &= 0 \\
x_1 &= 8 \\
x_5 &= 5 \\
x_8 &= 1
\end{aligned}$$

In matrix notation this becomes:

$$\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & -1 & -1 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & -1 & -1 & -1 \\
-1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 1 \\
1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 \\
0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 \\
-1 & 0 & 1 & 0 & -1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix}
=
\begin{bmatrix} 45 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 8 \\ 5 \\ 1 \end{bmatrix}$$

Note first that the matrix is no longer square – there are more constraints (10) than unknowns (9). This is correctly expressed as a legal matrix operation, though, since the number of columns of A equals the number of rows of x. The question is whether the system is over-constrained and there is no solution. Is this necessarily the case? In class we use Gauss-Jordan to derive the reduced row echelon form (rref command in Matlab). This gives us all of the information we need about the system. In this case a single unique solution does exist – our 10 equations really only constitute 9 linearly independent ones!

3.2 The discretized heat equation

We now introduce the first part of our class application. Very large linear systems typically arise from the discretization of differential equations that describe (predict) physical processes. A deep understanding of differential equations is not necessary to understand the numerical aspect of the application; however, a basic review will be helpful to getting the most out of the course. I will motivate the examples based on intuition more than rigor.

We begin with a simple form of the *heat equation* in one dimenion:

$$\begin{aligned}
&\frac{\partial T}{\partial t} = \alpha \frac{\partial(\frac{\partial T}{\partial x})}{\partial x} \equiv \alpha \frac{\partial^2 T}{\partial x^2} \\
&T(t=0,x) = T_0(x) \\
&T(x=0,t) = T_l; T(x=L,t) = T_r
\end{aligned} \tag{3}$$

In (**??**), T denotes the temperate of the (1-D) medium in question, t denotes time, α is the (assumed) constant *diffusivity* of the medium (rate at which it diffuses heat), and x is the spatial dimension. Conceptually it is best to imagine e.g. a long, thin metal bar. Initially the bar has temperature T_0 (can be a function of x); equation (**??**) predicts its temperature at every point at any time in the future. That is, a solution to (**??**) yields an expression for $T(t,x)$.

Most differential equations that describe physical processes cannot be solved in general – that is, $T(t,x)$ cannot be determined. It turns out that for the simple 1d equation above analytical solutions are easy to derive. As we move to more complex forms of the equation and more complex geometries, though, we'll find that there is no known analytical solution. In that case we attempt to derive approximate solutions numerically with the use of a digital computer. In this simple case we can use the analytical solution as a basis of comparison for our numerically derived answer.

The basis of the simplest procedures to numerically solve (**??**) is to express the derivates as finite differences based on the definition of a derivative.

$$\begin{aligned}\frac{\partial T}{\partial t} &\approx \frac{T_j^{i+1} - T_j^i}{\Delta t}\\ \frac{\partial^2 T}{\partial x^2} &\approx \frac{T_{j+1}^i - 2T_j^i - +T_{j-1}^i}{\Delta x^2}\end{aligned}$$

Note that the this particular choice of discretization is not unique (nor optimal, as well see). For now consider it one reasonable choice that will illustrate the general technique. We will discuss the *stability* and *accuracy* of this choice later.

Given this discretization we can solve for the temperature at any time increment as:

$$T_j^{i+1} = T_j^i + \frac{\alpha \Delta t}{\Delta x^2}\left[T_{j+1}^i - 2T_j^i - +T_{j-1}^i\right] \tag{4}$$

In (**??**) for simplicity we assume equal grid space – it $\Delta x = \frac{L}{n}$ where n is the number of spatial gridpoints so that $1 \leq j \leq n$. Similarly, we choose equally space timesteps Δt, so that the values of i correspond to $[0, \Delta t, 2\Delta t, ...]$.

In class we write a simple matlab and C code to solve for arbitrary boundary and initial conditions.

3.2.1 implicit methods

For reasons we will understand later in the course, it turns out that (**??**) is only stable for $\frac{\alpha \Delta t}{\Delta x^2} < 1$. A discretization with unconditional stability can be obtained as follows:

$$T_j^{i+1} = T_j^i + \frac{\alpha \Delta t}{\Delta x^2}\left[T_{j+1}^{i+1} - 2T_j^{i+1} - +T_{j-1}^{i+1}\right] \tag{5}$$

Specifically, we replace the T values on the right hand side with the same values evaluated at time $i + 1$. Note that this equation is no longer *explicit* – that is, we can no longer compute T^{i+1} in terms of previous values of T. Instead, what we have is a large linear system of n equations that has to be solved simultaneously. We derive this system as a simple in class exercise.

An alternate formulation which contains good aspects of FTCD and backward Euler is the so-called *Crank Nicholson* method. This method uses the average of the right hand sides of the two previous method:

$$T_j^{i+1} = T_j^i + \frac{\alpha \Delta t}{2\Delta x^2}\left[T_{j+1}^{i+1} - 2T_j^{i+1} - +T_{j-1}^{i+1} + T_{j+1}^i - 2T_j^i - +T_{j-1}^i\right] \tag{6}$$

This also forms a tri-diagonal linear system and is easily solved. We write solvers for each of these methods as in-class exercises.

3.3 tridiagonal systems

We saw that for both backward Eueler and Crank Nicholson the linear systems that emerged were highly *sparse* – ie most of the entries to A are zero. In such a case one does not waste the memory of literally building the entire A and passing it as a parameter to e.g. our Gaussian elimination method. Instead, depending on the structure of the matrix one of several alternatives is employed.

It turns out the *tridiagonal* systems are extremely common and merit their own special solver. Tridiagonal systems look e.g. like this (for a 5× case):

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 \\ 0 & 0 & a_4 & b_4 & c_4 \\ 0 & 0 & 0 & a_5 & b_5 \end{bmatrix}$$

For tridiagonal systems we only require as input three 1d arrays of n a's, b's, and c's. We do elimination directly only on the known non-zero coefficients. As in class exercise we code a simple tridiagonal solver as part of our solution to the implicit 1-D heat equation.