Lecture 7, Sci. Comp. for DPhil Students

Nick Trefethen, Tuesday 5.11.19

Today

- II.6 Floating point arithmetic
- II.7 Stability, conditioning, and backward error analysis

Handouts

- Assignment 2 solutions
- "Discrete or continuous?" essay from *SIAM News*, 2012
- `m12_floating.m`, `m13_stability.m`, `m14_conditioning.m`, `m15_backwardstab.m`

Announcements

- Read: Trefethen & Bau chapter 13 (pp. 97–101).
- Assignment 2 due now. Assignment 3 will be due in two weeks.
- Pass around copies of three great books:

Overton, *Numerical Computing with IEEE Floating Point Arithmetic*
Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed.
Muller et al.: *Handbook of Floating-Point Arithmetic*, 2nd ed.

II.6 Floating point arithmetic

The physical world

The laws of classical physics describe the motion and deformation of fluids and solids. They involve quantities such as density, pressure, and temperature, and they are typically written as PDEs.

Of course this is an approximation, for the world is not continuous but is made of discrete atoms and molecules. Density is an average, pressure is an average, temperature is an average.... But this is certainly the "right" thing to do for most applications in science and engineering: to ignore the atoms and molecules and regard the physical world as continuous.

Of course physicists well understand how the continuum is built up from discrete particles. Maxwell and Boltzmann in the 19th century were key developers of **statistical mechanics**. For example:

- If you halve the volume of a box, keeping temperature constant, the pressure of a gas inside doubles. Reason: twice as many impacts of molecules per unit cross-section per unit time. [Boyle's Law – 1662, here on the High Street in Oxford, with help from Towneley and Power]
- If you double the temperature of a box, keeping volume constant, the pressure doubles. Reason: the momentum of each particle increases by a factor of $\sqrt{2}$, and the number of impacts per unit cross-section per unit time also goes up by $\sqrt{2}$.

These are, as it were, "implementation details" for the laws of physics.

How fine is the physical continuum? **Avogadro's number** is the number of molecules in a mole of a substance; it is about 6×10^{23}. There are about 50 moles of gas in a cubic meter at ordinary conditions, so this comes to about 3×10^{25} molecules per cubic meter in a gas at ordinary conditions [**Loschmidt's constant**].

The cube root of 3×10^{25} is about 3×10^{8}. Thus there are about 3×10^{8} molecules per linear meter in an ordinary gas. For a solid, the figure is about ten times higher: 3×10^{9}.

Thus, roughly speaking,

A gas or solid has around 10^9 *particles per meter.*

That's 10^7 per centimeter; or 10^{11} across a football pitch. This is how fine the "discretisation" is in our physical world.

It is with these facts in mind that I'd like us to think about floating point arithmetic.

Computers represent the real line by a discrete approximation. The numbers are evenly spaced in $[1, 2)$, where we find exactly $2^{52} \approx 4.5 \times 10^{16}$ **floating point numbers**.

Everything is scale-invariant, so we find the same picture, rescaled, in $[1/2, 1)$ or in $[8, 16)$, etc.:

```
...XXXXX-X-X-X-X---X---X---X---X-------X-------X-------X-------x...

      1/2      1               2                               4
                        \  /
                          -52
                         2
```

This picture persists far out to the **underflow** and **overflow** limits, around $2^{-1024} \approx 10^{-308}$ and $2^{1024} \approx 10^{308}$. Thus:

Computer arithmetic is a million times finer than physics.

If we gave floating point arithmetic coordinates to this room or to this podium, we'd find there were about a million coordinate points between each adjacent pair of molecules. In fact, 10^{16} is more or less the number of molecules in a line from one point on earth through to its antipodal point.

Computers came on the scene around 1950, and for decades it was not so simple. Different manufacturers had different number systems, including bases 2,3,8,10, and 16. In the 1980s, however, thanks to the driving force of Prof. William Kahan of Berkeley, the **IEEE floating point arithmetic standard** was adopted. Nowadays virtually all machines have almost exactly the same arithmetic (though new hardware with lower precision to save time and power are an exciting topic of current research, including at Oxford with Tim Palmer and colleagues in Physics).

To be precise, we're talking of **IEEE double precision arithmetic**, which is the standard for numerical computation. The picture above is implemented by:

```
1 flt-pt real number  =  2 words  =  8 bytes  =  64 bits:
```

```
 1 bit  for sign
11 bits for base-2 exponent
52 bits for fraction
```

For the details, you can't do better than Overton's marvelous SIAM book *Numerical Computing with IEEE Floating Point Arithmetic* (see Books & Journals at our Web site). For a fuller treatment see Muller et al., *Handbook of Floating-Point Arithmetic,* 2nd ed.

All this is just how we *represent* real numbers.

Of course in addition we have to *compute* with them.

IEEE arithmetic follows a simple principle:

```
-------------------------------------------------------
|                                                     |
| If  x  and  y  are floating point numbers and       |
|                                                     |
| x+y, x-y, x*y, or x/y  is computed on the machine,  |
|                                                     |
| the result is the exact answer, correctly rounded.  |
|                                                     |
-------------------------------------------------------
```

(There are different rounding modes, e.g. round-to-nearest, which we won't discuss.)

Corollary

```
-------------------------------------------------------
|                                                     |
| Each operation yields                               |
|                                                     |
| computed(x op y) = (x op y) (1+eps)                 |
|                                                     |
| for some eps with  |eps| <= machine-eps             |
|                                                     |
|                           = 2^(-53) ~= 1.1e-16 .    |
|                                                     |
-------------------------------------------------------
```

This is a very powerful statement and worth mulling over. At every step along the way, whether you do ten operations or ten trillion, a small **rounding error** is introduced down at the 16th decimal place.

Assuming you avoid underflow and overflow, which usually isn't hard, that's all that you usually need to know of the underlying processes of floating-point arithmetic.

In the past, machines have not always lived up to this clean behaviour! For example, Cray machine subtraction used to have machine-eps = 1. And the 1994 Pentium bug that some of you may have heard of led to division with machine-eps $\approx 6 \times 10^{-5}$.

`[m12_floating.m]`

II.7 Stability, conditioning and backward error analysis

OK, now we know what computers are doing at a low level. What effect does this have on the correctness of our computations at a high level?

Analogously we might ask, when does a physicist notice macroscopic effects of the underlying atomic and molecular structure?

There are certainly cases where such effects appear. Superconductivity, superfluidity, and Bose-Einstein condensation are examples of microscopic quantum effects scaling up to macroscopic surprises. Quantum computing?

For the numerical analyst, microscopic effects cause trouble rather more often. This is called **instability** of a numerical algorithm.

Classic example of an unstable algorithm: computing $\exp(-40)$ via Taylor series. (Demonstration in a moment.) The cause here is cancellation error, which is at fault often but *not always!*

The following idea of **backward stability**, perhaps introduced by Turing and certainly made famous by Wilkinson, turns out to be a crucial notion.

A **stable algorithm** *is one that delivers the exact answer to a slightly perturbed problem.*

Perturbations are relative. "Slight" might mean, say, factor of $1+\epsilon$ with $\epsilon = 1000 \times$ machine-eps. That number 1000 grows with problem size, but slowly.

Backward stability is a powerful idea all across numerical analysis. In numerical linear algebra it is indispensable.

In general it is not reasonable to ask for more than backward stability. Thus $\sin(10^{20}\pi) = -0.3941$ is a perfectly respectable answer! — for it's exactly right for data perturbed relatively by around 10^{-16}.

In particular, it is not reasonable to expect accuracy for a problem that is highly **ill-conditioned** – sensitive to perturbations. E.G., if $\mathrm{cond}(A) = 10^{10}$, solutions to $Ax = b$ will be inaccurate.

---

ill-conditioned problem vs. *unstable algorithm*

Conversely, it is reasonable to ask for stability, and all across scientific computing, fast stable algorithms have been developed for all kinds of problems.

`[m13_stability.m, m14_conditioning.m, m15_backwardstab.m]`