Chapter 1 - Introduction to Numerical Computing and Matlab

Raymond J. Spiteri

Lecture Notes for Math 211:
Numerical Analysis 1
(Introductory Numerical Analysis)

University of Saskatchewan

January, 2013

1.1 Motivation for the Course

This course is about *numerical analysis*.

The associated computations are done using MATLAB.

- Numerical analysis is concerned with the solution of mathematically formulated problems via computer.

- It is what you do when you can't solve a mathematically formulated problem with pencil and paper alone.

- Numerical computing is being done all around us. ("We all use math everyday.")

- The problems themselves can come from many fields of application, e.g., biology, chemistry, physics, engineering, medicine, education, entertainment, the internet, forensics, financial markets, . . .

- The stakes are often high, so the results must be accurate, reliable, and robust.

What is Matlab*?*

- MATLAB = MATrix LABoratory
- MATLAB is a *problem-solving environment* (PSE).
- PSEs are meant to allow the student / researcher to focus on the numerical analysis more than the details of computer programming.
- It was developed by Cleve Moler in the 1970s as a teaching tool.
- Today it is used almost everywhere, from universities to government research labs to industry.

We will learn about these things mostly *by example*.

Numerical analysis is not a spectator sport! You are strongly encouraged to experiment with the demo programs until you get a feel for what is happening!

Why Matlab*?*

- It is powerful and yet relatively easy to learn.
- Knowledge of the use of MATLAB is a marketable skill. (It is also a necessary skill for many jobs!)

Multiplying two matrices $\mathbf{A} \in \Re^{l \times m}$ and $\mathbf{B} \in \Re^{m \times n}$ to get a matrix $\mathbf{C} \in \Re^{l \times n}$ using FORTRAN:

```
   DO 10 I=1,L
      DO 20 J=1,N
         C(I,J) = 0
         DO 30 K=1,M
            C(I,J) = C(I,J) + A(I,M)*B(M,J)
30       CONTINUE
20    CONTINUE
10 CONTINUE
```

(This does not include variable declarations, etc.)

In MATLAB:

```
C=A*B;
```

Poor computing: cost and opportunity

- A report from 2002 estimated that software bugs cost the U.S. economy $60B each year, $22B of which could be eliminated by improved practices.

- A recent article estimates that finding and fixing bugs costs the global economy $312B annually.

- Computer system errors are usually rooted in human error, not shortcomings in technology.

- In this course, we will learn some basic concepts related to understanding the kinds of problems that can occur in numerical computing as well as obtaining insight into how to obtain accurate and reliable results from a computer.

- You can easily imagine that this kind of training is very valuable to potential employers / customers.

Some infamous software bugs

- Mariner 1, July 22, 1962

  The Mariner 1 spacecraft had to be destroyed 294.5s after launch due to a missing overbar: $\dot{R}_n$ was used instead of $\bar{\dot{R}}_n$; i.e., the measured value of the rate of change of radius R at time t_n was used instead of its smoothed (time-averaged) value.

- The first computer worm, November 2, 1988

  While a computer science graduate student at Cornell University, Robert Tappan Morris made a coding error that unwittingly released the first computer worm, disabling around 10% of the computers on the Internet. Morris was fined $10,000 for his mistake, the first person convicted under the 1986 Computer Fraud and Abuse Act (US). He is now a professor at MIT.

Disasters caused by poor numerical computing

Here are some examples of real life disasters that occurred because of poor computing practices (and not because of programming errors per se).

- Vancouver Stock Exchange "Crash", 1982

  The index calculation always *truncated the updated value to 3 decimal places after each transaction*. After 22 months, the index went from its starting value of 1000.000 to 524.881 when it should have been at 1098.892.

- CSI: Round-off error (aka Salami attacks)

  Hackers diverted round-off errors from normal bank calculations to their own accounts, netting them substantial sums of money over time.

- Patriot missile failure, February 25, 1991

  A Scud missile killed 28 soldiers in Dhahran, Saudi Arabia, when the Patriot missile meant to intercept it miscalculated the Scud's position. The error came from an integer to floating-point conversion of 9.5×10^{-8} per tenth of a second multiplied by the product of the up-time of the Patriot (100 hours) and the velocity of the Scud (1676 metres per second); i.e., 573 metres.

- Mars Climate Orbiter crash, September 23, 1999

  A subcontractor failed to respect the specification that SI units be used instead of Imperial units. The Orbiter overshot its orbit and crashed into the Martian surface, putting a swift and disappointing end to a $655M investment.

- Y2K, January 1, 2000

  Almost $300B was spent worldwide from 1995 to 2000 to keep systems from miscalculation all because people did not use 4 digits to store the year. A further $21B were lost to outages.

- Toyota hybrid vehicle recall, February 8, 2010

  Toyota recalled more than 400,000 hybrid vehicles in 2010 because the software that controlled the anti-lock brakes was not responsive enough in regenerating braking after hitting a bump. This software glitch cost Toyota an estimated $3B.

- Knight Capital, August 1, 2012

  One incorrect line of code caused Knight Capital to lose $440M in about 30 minutes. This amount represents 3–4 times the amount of money made by the company in the previous year. The most likely cause is the common "rush to production" scenario in which proper testing is foregone in the name of expediency. The company is now struggling to remain solvent.

A Matlab Tutorial

See `intro.m`

See also deckShuffling.m, perfectShuffle.m

1.2 Floating-Point Arithmetic

A basic understanding of floating-point arithmetic is essential when solving problems numerically. This is because certain things happen in a floating-point environment that might surprise you if you have no appreciation for them.

The reason we have to study floating-point arithmetic at all is because of how numbers are stored and operated on in a computer.

In everyday life, we are used to dealing with integers and numbers that have 1 or 2 decimal places. We generally don't even like fractions.

But in science it does not take long before we encounter concepts like *irrational numbers* (e.g., $\sqrt{2}$), *transcendental numbers* (e.g., π), infinity (∞), and *infinitesimals* (arbitrarily small numbers).

On the other hand, a computer can only allocate a finite amount of memory to store any number, and it only has a finite amount of memory in total.

So we must work with a finite number of digits for a finite number of numbers!

These facts quickly lead to the phenomena of

- *round-off error*: not all numbers can be stored exactly; they must be rounded off so that it can be represented by the computer,

- *overflow*: when a number is larger than the largest number that can be represented (results in `Inf`),

- *underflow*: when a number has a smaller magnitude than the smallest that can be represented (number is usually set to 0).

We are able to study floating-point arithmetic in a systematic way because in 1985 it was agreed that all computers adopt the IEEE convention for floating-point arithmetic.

Before then, every computer could (and often did!) have its own brand of floating-point arithmetic!

This allows us to study floating-point arithmetic without referring to a specific computer.

Note 1. *This does not mean every computer will give the same answer to the same sets of floating-point operations!*

There is still some room for interpretation, even in the IEEE standard!

By default, MATLAB uses the IEEE *double-precision* format for its floating-point operations.

When double precision does not suffice, there is also *quadruple precision* (but not in MATLAB).

Software also exists that perform calculations in *multiple* (arbitrary) precision; e.g., in Maple by using the `Digits` command or in MATLAB using the `vpa` function from the Symbolic Math Toolbox.

We use numbers without thinking

We are so used to using numbers that we usually do so without thinking.

However, some thought about how we use numbers can add a lot of insight into how computers use them.

Historical Notes: The Sumerian[1] number system, the first known number system that existed in written form, used the (main) base 60 (well before 3000 BC).

The Aztec civilization (1300–1500 AD), like all other South and Central American civilizations, used the main base 20.

The sexagesimal system of Sumer was displaced by the decimal system of Egypt. Both were invented around the same time.

Base 5 was used by the Api people from Vanuatu in the South Pacific.

[1] Sumer was a city in Mesopotamia.

We use **base 10** for our number system.

So for example the number 123.456 means

$$1\times10^2+2\times10^1+3\times10^0+4\times10^{-1}+5\times10^{-2}+6\times10^{-3}.$$

We have also a scientific notation that *normalizes* each number so that the decimal place always appears in the same place.

(We will assume this is after the first significant digit.)

So instead of writing 123.456, 12.3456×10^1, etc., we agree to write this number using scientific notation as

$$1.23456\times10^2.$$

Thus any number is expressible in the form

$$\pm d_1.d_2d_3\ldots \times 10^e,$$

where each *digit* d_i satisfies $0 \leq d_i \leq 9$, $i = 1, 2, \ldots$, and $d_1 \neq 0$; e is called the exponent and in the real number system can be any integer.

Of course, some numbers are not expressible as a finite decimal expansion in base 10; e.g.,

$$\frac{1}{3} = \frac{3}{10} + \frac{3}{10^2} + \frac{3}{10^3} + \ldots.$$

If we terminate this expansion after any finite number of digits, we will be making a *round-off error*.

In general, if we terminate the expansion of any number at any point before it is complete, we will be making a round-off error.

Thus, if we can only store a fixed number of digits, not every number will be representable exactly.

For example, if we can store 3 digits, then 2.17 is exactly representable, but 2.1415 must be represented approximately, e.g., by 2.14.

Note that when we do computations with such numbers, intermediate results must be rounded at each step.

This means that exactly how you perform a computation may matter!

For example, in 5-digit arithmetic,

$$25.874 + 37654 - 37679 = 37680 - 37679 = 1.0000,$$

but

$$25.874 + (37654 - 37679) = 25.874 - 25.000 = 0.87400.$$

The difference is fairly substantial (one is exact, the other is off by more than 10%), especially if this number needs to be used in a future calculation!

On errors . . .

Let $\hat{x}$ be an approximation to a real number x.

Then the *absolute error* in $\hat{x}$ is defined to be

$$E_{\text{abs}} = |x - \hat{x}|,$$

and the *relative error* is defined to be

$$E_{\text{rel}} = \frac{|x - \hat{x}|}{|x|}.$$

Notes:

1. E_{rel} is not defined for $x = 0$.

2. The absolute value ($|\cdot|$) can be removed in the definition of E_{abs} if the sign of the error is important.

3. In practice, often a mix of absolute and relative errors are used to assess accuracy (or set tolerances).

Floating-point numbers

The chief advantage of the floating-point representation is that it can encompass a large range of numbers with a fixed amount of information.

For example, suppose we can store 6-digit numbers with 5 digits after the decimal place.

Then the largest number we can store is $9.99999 \approx 10$ and the smallest is $0.00001 = 10^{-5}$.

However, if we allocate 2 digits to represent an exponent (to base 10), then we can represent a much larger range of numbers.

The largest representable number is now 9.999×10^{99}.

The smallest representable number is 1.000×10^{-99}.

The price we have paid to get the increase in range is that we only have 4 significant digits instead of 6.

However, this tradeoff is favourable.

The concepts behind floating-point arithmetic on a computer follow these principles, except things are done in **base 2**.

The rest of our study of floating-point arithmetic will focus on the various nuances of dealing with real computer hardware.

In case you were wondering, base 2 has been adopted as the base for computer arithmetic primarily for economic reasons (although it turns out that small bases are a good idea for other reasons).

It is cheapest to manufacture transistors and detect whether they have a voltage on them (1) or not (0).

This leads to 2 (binary) voltage states that can be reliably created and measured, and hence base 2.

In contrast, *ternary* (or base 3) arithmetic could rely on whether a transistor had a certain voltage (+1), no voltage (0), or a reverse voltage (–1).

Unfortunately, such transistors are not as cheap to manufacture, nor is the production and detection of the voltage states as reliable.

Floating-Point Numbers

Because computers use base 2 to represent numbers, most real nonzero floating-point numbers are normalized to the form:

$$x = \pm(1 + f) \cdot 2^e.$$

The quantity f is called the *fraction* or mantissa, and e is called the *exponent*. The fraction satisfies

$$0 \leq f < 1.$$

The finiteness of f is a limitation on *precision* (how many significant digits can be stored).

The finiteness of e is a limitation on *range* (what numbers can be represented by the computer).

Any numbers that do not meet these limitations must be approximated by ones that do.

A Toy Floating-Point Number System

Imagine a floating-point number system using base 2.

Let f consist of t bits (binary digits; 0's or 1's), and let

$$e_{\text{min}} \leq e \leq e_{\text{max}}.$$

Suppose $t = 2$, $e_{\text{min}} = -2$, and $e_{\text{max}} = 1$.

Then all the positive numbers in this floating-point number system are all the possible combinations of the binary numbers

$$1.00, 1.01, 1.10, 1.11$$

and the exponents

$$-2, -1, 0, 1.$$

This leads to a total of 16 positive floating-point numbers in our number system. See floatgui.m

Some things to notice:

- Within each (binary) interval, i.e., intervals of the form $2^e \leq x \leq 2^{e+1}$, the floating-point numbers are equally spaced, with an increment of 2^{e-t}.

  For example, if $e = 0$ and $t = 2$, the spacing of the floating-point numbers between 1 and 2 is $1/4$.

- As e increases, the (absolute) spacing between floating-point numbers increases.

- With the logarithmic scale, it is more apparent that the distribution in each binary interval is the same; i.e., the relative spacing is constant.

- The distance between 1 and the next larger floating-point number is $\epsilon_{\text{machine}} = 2^{-t}$; $\epsilon_{\text{machine}}$ is known as machine epsilon.

- The smallest number that can be added to 1 and results in a number greater than 1 is $u = 2^{-(t+1)}$; u_e is known as unit round-off.

Double Precision

Double-precision floating-point numbers are stored in a 64-bit word, with 52 bits for f, 11 bits for e, and 1 bit for the sign of the number.

f must be representable in binary (base 2) using at most 52 bits. In other words, $2^{52}f$ is an integer satisfying

$$0 \leq 2^{52}f < 2^{52}.$$

The exponent e is an integer in the interval

$$-1022 \leq e \leq 1023.$$

The sign of e is accommodated by actually storing $e + 1023$, which is a number between 1 and $2^{11} - 2$.

The two extreme values for the exponent field (0 and $2^{11} - 1$) are reserved for exceptional floating-point numbers that we will describe later.

Note 2. *The entire fractional part of a floating-point number is not f, but $1+f$, which has 53 bits. However the leading 1 doesn't need to be stored.* In effect, the IEEE double-precision format packs 65 bits of information into a 64-bit word.

How much sleep should we lose over the gaps in the IEEE floating-point number system?

In one sense, not much. Here is a way to see it.

According to molecular physics, there are approximately 3×10^8 molecules per meter in a gas at atmospheric conditions — essentially the cube root of Avogadro's number. The circumference of the Earth is 4×10^7 meters; so in a circle around the Earth, there are around 10^{16} molecules.

In IEEE double-precision arithmetic, there are 2^{52} numbers between 1 and 2, and $2^{52} \approx 10^{16}$.

So if we put a giant circle around the Earth to represent the range from 1 to 2, the spacing of the double-precision floating-point numbers along the circle is about the same as the spacing of the air molecules.

Ubiquitous, innocuous round-off error

Round-off error even occurs with the innocuous-looking MATLAB statement

```
t = 0.1
```

The value `t` stored is not exactly 0.1 because $0.1 = 1/10$ requires an **infinite series** in binary:

$$\begin{aligned}\frac{1}{10} = \frac{0}{2^1} + \frac{0}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} \\ + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} \\ + \frac{0}{2^{10}} + \frac{0}{2^{11}} + \frac{1}{2^{12}} + \frac{1}{2^{13}} + \cdots\end{aligned}$$

Note that after the first term, the sequence of coefficients $\{0, 0, 1, 1\}$ is repeated infinitely often.

So the value stored in `t` is very close to, but not exactly equal to, 0.1.

This distinction can occasionally be important!

A simple example of round-off error

Consider the following MATLAB code segment:

```
format long
a = 4/3
b = a - 1
c = 3*b
e = 1 - c
```

In exact arithmetic, e $= 0$. What happens in MATLAB?

Note 3. *Before the IEEE standard, this code segment was used as a quick (and dirty) way to estimate* $\epsilon_{\text{machine}}$ *on a given computer.*

Machine Epsilon

A very important quantity associated with floating-point arithmetic is *machine epsilon*, $\epsilon_{\text{machine}}$.

In MATLAB, it is called `eps`.

Definition 1. $\epsilon_{\text{machine}}$ *is defined to be the distance from 1 to the next larger floating-point number.*

For the `floatgui` model floating-point system, $\epsilon_{\text{machine}} = 2^{-t}$.

Before the IEEE standard, different machines had different values of $\epsilon_{\text{machine}}$.

Now, for IEEE double-precision,

$$\epsilon_{\text{machine}} = 2^{-52} \approx 2.2204 \times 10^{-16}$$

(i.e., numbers are stored to about 16 digits).

$\epsilon_{\text{machine}}$ is not "floating-point zero"

There are many floating-point numbers between $\epsilon_{\text{machine}}$ and 0.

The smallest positive normalized floating-point number has $f = 0$ and $e = -1022$. MATLAB calls this number `realmin`.

The largest floating-point number has f a little less than 1 and $e = 1023$. MATLAB calls this number `realmax`.

In summary,

$$\texttt{eps} = 2^{-52} \approx 2.2204 \times 10^{-16}$$

$$\texttt{realmin} = 2^{-1022} \approx 2.2251 \times 10^{-308}$$

$$\texttt{realmax} = (2 - \epsilon_{\text{machine}}) * 2^{1023} \approx 1.7977 \times 10^{+308}$$

- Recall that if any computation is asked to produce a value larger than `realmax`, it is said to *overflow*.

  The result is an exceptional floating-point value called *infinity* or `Inf` in MATLAB.

  `Inf` is represented by taking $e = 1024$ and $f = 0$.

  `Inf` can make sense in calculations;
  e.g., $1/\texttt{Inf} = 0$ and $\texttt{Inf}+\texttt{Inf} = \texttt{Inf}$.

- If any computation is asked to produce a value that is **undefined** in the real number system, the result is an exceptional value called *Not-a-Number* (`NaN`).

  Examples include $0/0$ and `Inf`–`Inf`.

  The value `NaN` is represented by taking $e = 1024$ and any $f \neq 0$.

- If any computation is asked to produce a value smaller than `realmin`, it is said to *underflow*.

  This involves *subnormal* (or denormal) floating-point numbers in the interval between `realmin` and $\epsilon_{\text{machine}}$*`realmin`.

  The smallest positive subnormal number is $2^{-52-1022} \approx 4.94 \times 10^{-324}$.

  Any results smaller than this are set to 0.

  On machines without subnormal numbers, any results less than `realmin` are set to 0.

  The subnormal numbers live between 0 and the smallest positive normalized number (recall `floatgui.m`).

  They provide an elegant way to handle underflow, but their importance for computation is rare.

  Subnormal numbers are represented by the special value $e = -1023$, so the biased exponent $(e+1023)$ is 0.

- MATLAB uses the floating-point system to handle integers.

  Mathematically, the numbers 3 and 3.0 are equivalent, but many programming languages would use different representations for the two.
  MATLAB *does not make such a distinction!*

  We sometimes use the term *flint* to describe a floating-point number whose value is an integer.

  Floating-point operations on flints do not introduce any round-off error, as long as the results are not too large.

  Specifically:

  – Individual addition, subtraction, and multiplication of flints produce the exact flint result, if the result is not larger than 2^{53}.

    Beware of intermediate steps exceeding this limit!

  – Division and square root involving flints also produce a flint if the result is an integer. For example, sqrt(363/3) produces 11, with no round-off error.

Computing with Polynomials and Horner's Method

Polynomials are one of the most common tools in applied mathematics.

Dealing with them properly in floating-point arithmetic is important!

Consider the following MATLAB code segment:

```
x = 0.988:.0001:1.012;
y = x.^7-7*x.^6+21*x.^5-35*x.^4+35*x.^3-21*x.^2+7*x-1;
plot(x,y)
```

What is this code segment supposed to do?

Consider replacing the second line with

```
y = (x-1).^7;
```

or

```
y=-1+x.*(7+x.*(-21+x.*(35+x.*(-35+x.*(21+x.*(-7+x))))));
```

See hornerDemo.m

Horner's Method

To reduce round-off error and computational effort, it is best to use Horner's method to evaluate polynomials.

In order to evaluate the polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n,$$

we re-write it and evaluate it as

$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + a_n x) \ldots)).$$

Easy as 1, 2, 3

> *I do hate sums. There is no greater mistake than to call arithmetic an* exact *science. There are . . . hidden laws of Number which it requires a mind like mine to perceive. For instance, if you add a sum from the bottom up, and then again from the top down, the result is always different.*
> *— Mrs. La Touche (1924)*

We have seen that floating-point operations do not satisfy the axioms of operations with real numbers.

So even the order in which you add numbers matters!

Easy as 1, 2, 3

Consider the natural implementation (known as *recursive summation*) of $\sum_{i=1}^{n} x_i$ in MATLAB as

```
s = 0;
for i=1:n,
  s = s + x(i);
end
```

yields an answer that depends on the order of the x_i.

There is a famous algorithm due to Kahan (1965) that estimates the error produced when summing two numbers and accumulates this error during the computation.

It is based on the observation that the round-off error in the sum of two numbers can be estimated by subtracting one of the numbers from the sum.

Compensated Summation

The way this works can roughly be described as follows:

Imagine two numbers a and b such that $|a| \geq |b|$.

Their exact sum is $s = a + b = \hat{s} + e$.

Let $a = a_1 + a_2$ and $b = b_1 + b_2$, where a_2 and b_1 are determined by the *significance overlap* of the numbers.

For example, in 5-digit arithmetic

$$3.1416+0.023072 = (3.1+0.0416)+(0.02307+0.000002).$$

Then

$$\hat{s} = a_1 + (a_2 + b_1).$$

Now

$$\hat{s} - a = b_1 + 0,$$

and thus

$$(\hat{s} - a) - b = -b_2 + 0 = -e.$$

Compensated Summation

Here is the algorithm:

```
s = 0; e = 0;           % initialize sum and error

for i=1:n,
  temp = s;             % temporary sum
  y = x(i) + e;         % read in the next element and compensate
  s = temp + y;         % form sum from temporary element and
                        % compensated element
  e = (temp - s) + y;   % compute estimated error and carry forward
end
```

Two things to note, however:

1. The error is only estimated; the exact answer will not be magically produced as it would be if we had the exact error.

2. It costs extra to compute and store the error.

See cSumDemo.m

Compensated Summation

Note 4. *The pinnacle of this type of algorithm is a doubly compensated summation by Priest (1992), which under reasonable assumptions satisfied by IEEE arithmetic with t digits, base β, and provided $n \leq \beta^{t-3}$, the computed sum $\hat{s}_n$ satisfies*

$$|s_n - \hat{s}_n| \leq 2u_e|s_n|.$$

This is basically accuracy to full precision.