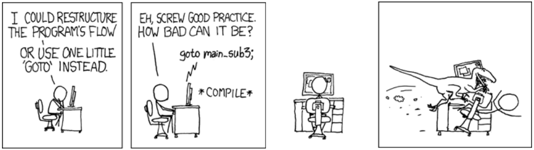
CS319: Scientific Computing (with MATLAB)

Numbers & errors; input & output

Week 2: **9am and 4pm**, 18 Jan 2023

Niall Madden



Source: xkcd (292)

Schedule: I'm still working on it....

Bitbucket

You should have access to the CS319 git repository at
`https://bitbucket.org/niallmadden/2223-cs319/src`
I sent out invitations last Tuesday. Check your email.

Register for your (free!) use of MATLAB.
Then you can

- Use the web-based version at `https://matlab.mathworks.com` (**recommended**)
- Download and install it on your own computer. But only do this if you have lots of free disk-space, and, to start with, don't include any extra toolboxes. These can be added later if needed.

Today:

1. 1. Errors
2. 2. Numbers
   - A close look at integers
3. 3. Floating point numbers
   - single
   - Example: comparing floats
   - `double`
4. 4: Output/Input
   - Output
   - Input

Other reading:

- Chapter 1 of The MATLAB Guide: https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9781611974669.ch1
- Chapter 1 of Learning MATLAB: https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9780898717662

1. Errors

(This discussion is based on Section 1.1 of Scientific Computing With Case Studies (https://epubs-siam-org.nuigalway.idm.oclc.org/doi/book/10.1137/9780898717723).

Computational modelling involves a number of steps:

1. Constructing a module of a physical situation;
2. Determining inputs for the equations, through physical measurement;
3. Selecting and coding an algorithm to compute useful solutions to the equations;
4. Running the program.

Each of those steps introduces an **error**:

1. Modelling error;
2. Measurement error;
3. Approximation error;
4. Rounding error.

It is important not to conflate these errors with **mistakes**: the errors are known and tolerated. But it is important to understand their implications. In scientific computing, we particularly care about (iii) and (iv).

2. Numbers

Variables are used to temporarily store values (numerical, text, etc, ...).

In MATLAB, a variable can be created just by assigning it a value.

Every variable has a **type**. Most of the time, unless we specifically state otherwise, variables in MATLAB are stored as **floating point numbers** called `double`.

Since it is the most important, we'll study it in depth presently.

2. Numbers

There are other numbers types in MATLAB

- **integers:** these are whole numbers. There are various types: `int8`, `int16`, `int32`, `int64`, which are stored in 8, 16, 32 and 64 bits.
- **unsigned integers:** these are non-negative whole numbers. Types are `uint8`, `uint16`, `uint32`, and `uint64`.
- **`logical`:** a variable that takes the value of `true` or `false`
- **`char`:** stores alphabetic or numeric symbols.

It is important for a course in Scientific Computing that we understand how numbers are stored and represented on a computer.

The most fundamental unit of storage on a computer is a **bit**. It is either **zero** or **one**.

The `logical` data type uses a single bit.

Your computer stores numbers collections of bits, and does so in the **binary** format; that is, in **base 2**.

Although they are not the most important, the easiest examples of binary consider are **integers**

Examples:

The smallest collection (other than a single bit) is called a **byte**, which has 8 bits.

The simplest integer type in MATLAB is `uint8` which stores an unsigned integer in a single byte.

The largest number representable is:

At the other extreme, an integer of type `int64` is stored in 64 bits (= 8 bytes).

One of 64 bits is used for the sign. Therefore the largest `integer` we can store is $2^{63} \approx 9.22 \times 10^{18}$

One can use the builtin MATLAB functions, `intmin` and `intmax` to check this.

Presently, we'll see a **MATLAB script** to do this.

MATLAB Scripts

A **scripts** is a list of MATLAB instructions stored in a file.
The run the script on the next slide, download it, without changing the name.
Then type the scripts name in the MATLAB command window.
If using Online MATLAB, you have to upload to that. Or try accessing at
https://drive.matlab.com/sharing/e3158c7e-feac-4ff9-878f-4a7898427550

Eg01_MyIntMax.m

```
%% Example 01 from Week 2: Check int max
% Who: Niall Madden
% What: manually checks the value of the largest
%  integer of type int18 that can be stored.
% When: Jan 2023

clear; % clear any previously defined variables
fprintf('\n-----------------------------\n');
fprintf('Example 1 from CS319-Week 2\n');

a=uint8(0); % Set a to zero in int16
b=a+1;
while( b>a )
   a=a+1;
   b=a+1;
end
fprintf('Largest int16=%d\n', a);
```

3. Floating point numbers

MATLAB (and just about every language you can think of) uses IEEE Standard Floating Point Arithmetic to approximate the real numbers. This short outline is based on Chapter 1 of O'Leary "*Scientific Computing with Case Studies*".

A floating point number ("**float**") is one represented as, say, 1.23×10^2. The "fixed" point version of this is 123.

Other examples:

3. Floating point numbers

The format of a float is

$$x = (-1)^{Sign} \times (Significant) \times 2^{(offset+Exponent)},$$

where

- *Sign* is a single bit that determines of the float is positive or negative;
- the *Significant* (also called the "**mantissa**") is the "fractional" part, and determines the precision;
- the *Exponent* determines how large or small the number is, and has a fixed offset (see below).

MATLAB has two types of float: `single` and `double`, which are stored in 4 and 8 bytes, respectively.

A variable of type `single` is stored using 4 bytes ($= 32$ bits). These 32 bits are allocated as:

- 1 bit for the *Sign*;
- 23 bits for the *fraction* (leading "implied bit" is free); and
- 8 bits for the *Exponent*, which has an offset of $e = -127$.

So this means that we write x as

$$x = \underbrace{(-1)^{Sign}}_{\text{1 bit}} \times 1.\underbrace{abcdefghijklmnopqrstuvw}_{\text{23 bits}} \times 2^{(-127+\underbrace{Exponent}_{\text{8 bits}})}$$

Since the *Significant* starts with the implied bit, which is always 1, it can never be zero. We need a way to represent zero, so that is done by setting all 32 bits to zero.

The smallest the *Significant* can be is

$$1.\underbrace{0000000000000000000000}_{\text{22 zeros}}1 \approx 1.$$

The largest it can be is

$$1.\underbrace{11111111111111111111111}_{\text{23 ones}} = 2 - 2^{-23} \approx 2.$$

Here it helps to remember that the binary faction 1.1 means (in decimal) $1 + \frac{1}{2}$, 1.11 means $1 + \frac{1}{2} + \frac{1}{4}$, etc.

The *Exponent* has 8 bits, but since they can't all be zero (as mentioned above), the smallest it can be is $-127 + 1 = -126$.

Together, all this means that the smallest positive `single` one can represent is
$x = (-1)^0 \times 1.000 \cdots 1 \times 2^{-126} \approx 2^{-126} \approx 1.1755 \times 10^{-38}$.

We also need a way to represent ∞ ("Inf"), which is done by setting all the bits to 1.

So the largest *Exponent* can be is $-127 + 254 = 127$.

That means if a number is of type `single`, the largest positive number it can represent is
$x = (-1)^0 \times 1.111\cdots 1 \times 2^{127} \approx 2 \times 2^{127} \approx 2^{128} \approx 3.4028 \times 10^{38}$.

This can be checked: `>> realmax('single')`

As well as working out how small or large a `single` can be, one should also consider how **precise** it can be. That often referred to as the **machine epsilon**, can be thought of as *eps*, where $1 - eps$ is the largest number that is less than 1 (i.e., $1 - eps/2$) would get rounded to 1.

The value of *eps* is determined by the *Significant*.

For a `single`, this is $x = 2^{-23} \approx 1.192 \times 10^{-7}$.

The MATLAB function `eps` can be used to check this.

As a rule, if `a` and `b` are floats, and we want to check if they have the same value, we don't use `a==b`.

This is because the computations leading to `a` or `b` could easily lead to some round-off error.

So, instead, should only check if they are very "similar" to each other: `abs(a-b) <= 10*eps('single')`

The default data-type in MATLAB is `double`:

- stored in 64 bits.
- 1 bit for the *Sign*;
- 52 bits for the *Significant* (as well as an leading implied bit); and
- 11 bits for the *Exponent*, which has an offset of $e = -1023$.

The smallest positive double that can stored is $2^{-1022} \approx 2.2251e-308$, and the largest is

$$\begin{aligned} 1.111111\cdots 111 \times 2^{2046-1023} &= \left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\dots\right)\times 2^{2046-1023} \\ &\approx 2\times 2^{1023} \approx 1.7977e+308. \end{aligned}$$

For a `double`, machine epsilon is $2^{-53} \approx 1.1102 \times 10^{-16}$.

Eg02_Rounding.m

```
%% Example 02 from Week 2: showing rounding errors
% CS319. Jan 2023

clear; % clear any previously defined variables
fprintf('\n------------------------------\n');
fprintf('Example 2 from CS319-Week 2\n');

n = 6; % Try different values!
h = 1/n;
x = 0.0;

for i=1:n
    x = x+h;
end
fprintf('I get x=%17.16f, and abs(1-x)=%8.5e\n', ...
    x, abs(1-x));
```

What results to we get with different n?

There are several different ways of getting output from MATLAB:

- When running a command, just omit the semi-colon from the line.
- Use the `disp()` function which outputs a single variable.
- The results of the above two can be controlled using the `format` function. Try

```
disp(pi)
format long
disp(pi)
format shortE
disp(pi)
```

- Use the fancy `fprintf` function. This is especially useful, because we can mix text and values, can specify how many decimal places, to output to, etc. Also, this is used to write to files.

Examples:

```
fprintf('pi = %f\n'); % pi = 3.141593
fprintf('pi = %7.1f\n', pi)
fprintf('pi = %7.3f\n', pi)
fprintf('pi = %7.5f\n', pi)
fprintf('pi = %7.7f\n', pi)
fprintf('pi = %7.6e\n', pi)
```

Other useful conversion characters:

- `c` or `s` single character or string
- `d` or `i` integer
- `g` or `G` let MATLAB guess if `f` or `e` is better.

Note the use of single quotes. One can also use double quotes (in more recent versions of MATLAB).

Since MATLAB is an interactive system, reading input in a script is not very common. But if we must:

```
x = input('Tell me something: ')
```

```
x = input('Tell me something: ', 's')
```