AMS 595.03 MATLAB Lecture 1

Table of Contents

Author: Oliver Yang

For further reading, Ch. 1 of *Attoway* and Chs. 2-3 of *Lockhart/Tilleson* also cover some of the topics of this lecture.

The MATLAB Desktop

A lot of people like to hate on MATLAB because they do not consider it to be a "hard-core" programming language. It is easier to use than C, but that in itself does not make it any worse. MATLAB is actually widely used by numerical analysts, including very top ones. (**Numerical analysis** is the study of computational methods in mathematics, as opposed to cominatorial algorithms, which are the subject of computer science.) It is also used as the default language of our research group.

Start the MATLAB Desktop.

One of the objectives of the MATLAB unit is to teach you not only the MATLAB programming language, but also how to use the MATLAB programming environment, which is an integrated development environment (IDE), i.e. it contains a terminal, code editor, and debugger.

By default, the Toolstrip is visible at the top of the MATLAB Desktop. I like to select Layout > Three Column in the Environment section of the Toolstrip.

In that case, under the Toolstrip, the left pane is the Current Folder. Programming languages allow you to issue instructions to the computer with source code written in that language. MATLAB stores code in script and function files. Note that by default, MATLAB only accesses files in Current Folder. To access files in other folders, you will have to add the directory to the MATLAB path. Folders (and files within them) that are not in the MATLAB path are transparent. Add a folder by right-clicking it and selecting Add to Path or by going to Set Path in the Environment section of the Toolstrip.

The top of the center pane is the Editor. To give more space for viewing code, you can minimize the Toolstrip by clicking the arrow at the bottom right of the Toolstrip. Or you can undock the Editor by clicking the arrow at the upper right corner of the Editor window and selecting Undock. This opens the Editor in a new window.

The bottom of the center pane is the Command Window, i.e. terminal, which is what we will use today. Next week, we will start using the Editor and debugger. You type commands at the prompt, indicated by >>. The fx button to its left allows you to select the MATLAB built-in functions.

By default, the right pane is the Workspace. Computer programs store data that is currently being used in **variables**. Variables are listed in the Workspace.

The Command History can be made to popup from the Command Window by selecting Layout > Command History > Popup in the Environment section of the Toolstrip. It can also be docked in the right pane by selecting Layout > Command History > Docked.

Variables

The Command Window and Formatting

Functions take variables as inputs, compute with them or on them, and return values in other variables. They are called procedures in Fortran and methods in Java. Next week, we will learn more about functions and write user-defined functions.

Now let's type a statement at the prompt and execute it by pressing Enter:

```
1
```

```
ans =
     1
```

This is an example of a numeric **literal**, a special type of expression whose value does not depend on variables or require computation. Note that a variable named `ans` now appears in the Workspace with the value 1. This is the default variable that expressions are assigned to when no variable is specified. The result is also displayed on the screen. To disable this, add a semicolon at the end of the statement:

```
2;
ans
```

```
ans =
     2
```

The semicolon is necessary to end a statement in C/C++, but not in Fortran, MATLAB, or Python. The value stored in `ans` is now 2. The value of any variable is displayed on the screen by using the variable name as a statement.

I don't like to display blank lines in the Command Window, so I use the `format` function to change the formatting of output in the Command Window:

```
format( 'compact')
```

The sequence of input arguments to a function is enclosed by a pair of parentheses after the function name. (I like to add a space after the first parenthesis to make the code easier to read. It is a matter of programming style.)

The `format` function takes a **character array** as input. Character literals are enclosed by single quotes.

Note that MATLAB ignores unnecessary spaces when executing code. In Fortran 77 and Python, on the other hand, the position of text in a line carries meaning.

MATLAB also ignores comments, which are denoted by lines beginning with the percent sign. You can also have inline comments (MATLAB ignores everything after `%` in a line of code). Use `%{` and `%}` on separate lines to comment all the lines between them (block comments):

```
% This is a comment on its own line.
a = 1; % This is an inline comment.
%{
This is a block comment.
%}
```

An alternative syntax for character array inputs to functions is:

```
format compact
```

I, and most other programmers, usually don't remember the required input arguments of built-in functions which are seldomly used. To review this information, you need to consult the documentation. (This is precisely the reason to include documentation in your own code. How else will others be able to use your code?)

To open the MATLAB documentation about any function in a new window, with descriptions of input and output arguments, as well as examples and related functions, use the `doc` function with the function name as input, or right-click the function name and select Help on Selection, or type the function name into Search Documentation at the top right corner of the Toolstrip:

```
doc( 'format')
```

The `help` function displays an abridged version in the Command Window:

```
help( 'format')
```

```
 format Set output format.
    format with no inputs sets the output format to the default appropriate
    for the class of the variable. For float variables, the default is
    format SHORT.

    format does not affect how MATLAB computations are done. Computations
    on float variables, namely single or double, are done in appropriate
    floating point precision, no matter how those variables are displayed.
    Computations on integer variables are done natively in integer. Integer
    variables are always displayed to the appropriate number of digits for
    the class, for example, 3 digits to display the INT8 range -128:127.
    format SHORT and LONG do not affect the display of integer variables.

    format may be used to switch between different output display formats
    of all float variables as follows:
      format SHORT     Scaled fixed point format with 5 digits.
      format LONG      Scaled fixed point format with 15 digits for double
                       and 7 digits for single.
      format SHORTE    Floating point format with 5 digits.
      format LONGE     Floating point format with 15 digits for double and
                       7 digits for single.
      format SHORTG    Best of fixed or floating point format with 5
                       digits.
      format LONGG     Best of fixed or floating point format with 15
                       digits for double and 7 digits for single.
      format SHORTENG  Engineering format that has at least 5 digits
                       and a power that is a multiple of three
      format LONGENG   Engineering format that has exactly 16 significant
                       digits and a power that is a multiple of three.

    format may be used to switch between different output display formats
    of all numeric variables as follows:
      format HEX     Hexadecimal format.
      format +       The symbols +, - and blank are printed
```

```
                     for positive, negative and zero elements.
                     Imaginary parts are ignored.
      format BANK    Fixed format for dollars and cents.
      format RAT     Approximation by ratio of small integers.  Numbers
                     with a large numerator or large denominator are
                     replaced by *.

    format may be used to affect the spacing in the display of all
    variables as follows:
      format COMPACT Suppresses extra line-feeds.
      format LOOSE   Puts the extra line-feeds back in.

    Example:
       format short, pi, single(pi)
    displays both double and single pi with 5 digits as 3.1416 while
       format long, pi, single(pi)
    displays pi as 3.141592653589793 and single(pi) as 3.1415927.

       format, intmax('uint64'), realmax
    shows these values as 18446744073709551615 and 1.7977e+308 while
       format hex, intmax('uint64'), realmax
    shows them as ffffffffffffffff and 7fefffffffffffff respectively.
    The HEX display corresponds to the internal representation of the value
    and is not the same as the hexadecimal notation in the C programming
    language.

    See also disp, display, isnumeric, isfloat, isinteger.

    Reference page for format
```

To display input argument hints for a function in a popup, type the function name, followed by a left parenthesis, then wait for about two seconds.

You may open the complete MATLAB in-product documentation by selecting the question mark button at the top of the Toolstrip.

Since I do a lot of scientific computing, I also usually like to display numbers in scientific notation with full precision. To do this, use the `format` function again:

```
format( 'longE')
```

Variable Assignment

Let's assign a value to a variable named a by using the assignment **operator** =, not to be confused with the equality operator ==.

```
a = 1
```

```
a = 
     1
```

Variables can be renamed and duplicated by right-clicking the variable in the Workspace.

Now let's assign a to another variable named b:

```
b = a
```

```
b = 
     1
```

(I like to add a space before and after operators to make the code easier to read. It is a matter of programming style.)

Note that `b` has the same value as `a`. Now let's assign another value to `a`:

```
a = ...
    2
```

```
a =
    2
```

Long lines can be broken and continued on the next line for ease of reading by using the ellipsis. Pressing Enter after an ellipsis in the Command Window causes MATLAB to display "Continue entering statement" in the status bar at the bottom of the Desktop and wait for further input. The ellipsis will be equivalent to a space. The correpsonding symbol in Fortran is `&`, and in Python, it is `\`. In C/C++, no symbol is necessary, but you may add a `\` if you wish. It is also good style to indent the continued lines.

Note that the value in `b` remains the same. Therefore `b` does not just refer to `a`. Rather, MATLAB created a separate new object in memory called `b`, with the same value as `a`. This is called a **deep copy** as opposed to a **shallow copy**. In other languages (e.g. C/C++), it is possible to just have `b` refer to `a` (a shallow copy), so that when the value of `a` changes, so does the value of `b`. You will see this concept many times in this course.

Variable Names

Every programming language has reserved words called **keywords** that cannot be used as variable names. MATLAB as 20 of them and they are always displayed in blue. We list them using the `iskeyword` function:

```
iskeyword()
```

```
ans = 20×1 cell array
    {'break'     }
    {'case'      }
    {'catch'     }
    {'classdef'  }
    {'continue'  }
    {'else'      }
    {'elseif'    }
    {'end'       }
    {'for'       }
    {'function'  }
    {'global'    }
    {'if'        }
    {'otherwise' }
    {'parfor'    }
    {'persistent'}
    {'return'    }
    {'spmd'      }
    {'switch'    }
    {'try'       }
    {'while'     }
```

Functions without any inputs are invoked with empty parentheses, or just the function name:

```
iskeyword
```

```
ans = 20×1 cell array
    {'break'     }
    {'case'      }
    {'catch'     }
    {'classdef'  }
    {'continue'  }
    {'else'      }
    {'elseif'    }
    {'end'       }
    {'for'       }
    {'function'  }
    {'global'    }
    {'if'        }
    {'otherwise' }
    {'parfor'    }
    {'persistent'}
    {'return'    }
    {'spmd'      }
    {'switch'    }
    {'try'       }
    {'while'     }
```

You can check if a name is in use with the `exist` function:

```
exist( 'a')
```

```
ans =
     1
```

This statement would give an error:

```
if = 1
```

Variable names must start with a letter. This would also produce an error:

```
1a = 1
```

They may only contain letters, digits, and underscores. This also produces an error:

```
a@ = 1
```

MATLAB, as most programming languages (e.g. C/C++, Java, Python), is case-sensitive, although Fortran, most notably, is not. So `A` will be a separate variable than `a`:

```
A = 3
```

```
A =
     3
```

```
a
```

```
a =
     2
```

It is good style to use abbreviations and underscores in variable names, for example:

```
vec1 = 1;
row_ptr = 1;
```

Deleting, Saving, and Loading Variables

To remove a variable from the Workspace, use the `clear` function with the variable name as input:

```
clear( 'row_ptr')
```

To remove all the variables, just use `clear`, or right-click in the Workspace and select Clear Workspace, or Clear Workspace in the Variable section of the Toolstrip:

```
clear
```

To clear the screen, use the `clc` function or right-click in the Command Window and select Clear Command Window:

```
clc
```

To see the entire Command History, use the up and down arrow keys at an empty prompt. To see previous commands starting with certain characters, for example, commands starting with "a" (there have been a few so far), type those characters at the prompt and then use the arrow keys.

The Workspace is cleared when you terminate the MATLAB desktop. To save a variable, use the `save` function with the variable name as input. This saves the variable to variable_name.mat. MAT is MATLAB's compressed file format. Following von Neumann's model of computation, code and data are both stored in the same memory.

```
save( 'a')
```

You must be careful that some functions take the name of the variable as a character array and other functions take the variable itself. Multiple variables can be selected in the Workspace and saved by right-clicking and selecting Save As...

The `save` function without any inputs saves the entire Workspace to matlab.mat, or you could use Save Workspace in the Workspace section of the Toolstrip:

```
save
```

```
Saving to: /Users/oyang/NumGeom/matlab.mat
```

To load a file, use the `load` function, or double-click it in Current Folder, or use Import Data in the Workspace section of the Toolstrip to select which variables in the file to load:

```
load( 'a')
load( 'matlab')
```

The `diary` function toggles logging. Logging writes all the subsequent Command Window input and output to a text file whose name is the input character array:

```
diary( 'log_file')
```

This is an easter egg in MATLAB that displays random sentences:

```
why
```

```
The bald and not excessively bald and not excessively smart hamster obeyed a terrified and not excessively terrifie
```

```
why
```

```
To fool the tall good and smart system manager.
```

```
why
```

```
The rich rich and tall and good system manager suggested it.
```

```
diary off
```

The `type` function displays the contents of a file. It takes the filename as input:

```
type( 'log_file')
```

Data Types

Floating-Point Numbers

Computers only work with discrete objects, not continuous ones. Therefore, they can only represent rational numbers, and moreover, not all rational numbers -- only those to a specified finite **precision** (e.g. decimals such as 1.0, 0.5, or -2.3333333333333333). They are called **floating-point numbers**.

Irrational numbers such as π are truncated and the last digit is rounded:

```
pi
```

```
ans =
     3.141592653589793e+00
```

On the real line, there is an infinite sequence of numbers approaching a given number from either side. This is not true in floating-point. The `eps` function tells you the gap between the input number and the next larger floating-point number:

```
eps( pi)
```

```
ans =
     4.440892098500626e-16
```

The gap increases as the numbers get larger:

```
eps( 2 * pi)
```

```
ans =
     8.881784197001252e-16
```

`eps( 1)`, or simply `eps`, is called **machine epsilon**.

Double-precision floating-point numbers are stored using 8 **bytes**, or 64 **bits**. One bit stores one binary number, i.e. 0 or 1. Floating-point numbers are always represented with 1 as the leading digit (since otherwise, the number may be scaled so that 1 is the leading digit) and a decimal point afterwards. 52 bits store the fractional part, 11 bits store the exponent, and 1 bit stores the sign. In base 10, this gives about 16 digits of precision.

These are the largest and smallest magnitudes that can be represented in double-precision, roughly 2^{1023} and 2^{-1023}:

```
realmax
```

```
ans =
    1.797693134862316e+308
```

```
realmin
```

```
ans =
    2.225073858507201e-308
```

Computation resulting in numbers greater than `realmax` and lower than `realmin` is called **overflow** and **underflow**, respectively. In scientific computing, you always need to watch out for overflow and underflow.

This is machine epsilon in double-precision:

```
eps( 1)
```

```
ans =
     2.220446049250313e-16
```

Note that machine epsilon is a property of your CPU, not of the programming language. Every CPU with the IEEE floating-point systems (essentially all computers nowadays) will have this same machine epsilon.

Singles are stored using 4 bytes, or 32 bits. 23 bits store the fractional part, 8 bits store the exponent, and 1 bit stores the sign. In base 10, this gives about 8 digits of precision.

These are the largest and smallest magnitudes that can be represented in double-precision, roughly 2^{127} and 2^{-127}:

```
realmax( 'single')
```

```
ans = single

   3.4028235e+38
```

```
realmin( 'single')
```

```
ans = single

   1.1754944e-38
```

This is machine epsilon in single-precision:

```
eps( single( 1))
```

```
ans = single

   1.1920929e-07
```

Every **object** in MATLAB is an **instance** of a certain **class**. For example, every double variable is an instance of the double class. Every instance of a class has the properties that define the class. A class is the programming equivalent of a category in mathematics. You will learn more about classes and object-oriented programming in the Python and C++ units. The `class` function returns a character array of the class of its input:

```
class( 1)
```

```
ans =
'double'
```

Note that in MATLAB, values of different data types can be assigned and reassigned to the same variable anywhere in the code:

```
a = 1
```

```
a =
     1
```

```
a = single( 1)
```

```
a = single

     1
```

This cannot be done in C or Fortran, in which all variables have a fixed data type and must be declared at the beginning of a program or function.

Whenever you type a number in MATLAB, MATLAB treats it as double-precision floating-point by default. In C and Fortran, on the other hand, floating-point numbers must be specified by the presence of a decimal point. Also, in Fortran, they are single-precision by default.

It is important to note that double and single-precision arithmetic are built into the hardware. That is, there are actually circuits that perform addition, for example, for doubles. Arithmetic in other precisions have to done at the software level, which is much slower, although there are now proposals to include 16-bit and quadruple-precision arithmetic into new chips.

There are several advantages to using single-precision. Singles require only half the memory of doubles. Depending on the **arithmetic-logic unit** (ALU) of your CPU, single-precision arithmetic is usually 2-4 faster than double. There is also speedup in the data throughput into the processor, since twice as many singles can be moved as doubles. (Computer architecture is covered in CSE 502.)

The disadvantage, of course, is that single-precision computations are not as accurate. Whether or not single-precision will suffice depends on the scientific application at hand. For highly ill-conditioned problems, single-precision may not give enough digits of **accuracy** for numerical methods to converge.

There are functions to **cast** their inputs into other numeric types. Casting a double as a single results in loss of precision:

```
single( pi)
```

```
ans = single

   3.1415927e+00
```

Note that casting it back to a double results in a discrepancy with the original at the 8th digit:

```
double( single( pi))
```

```
ans =
     3.141592741012573e+00
```

```
pi
```

```
ans =
     3.141592653589793e+00
```

Integers

MATLAB has 1, 2, 4, and 8-byte **integer** types (e.g. for numbers such as 2, 0, or -1) with and without sign:

```
int8( 1)
```

```
ans = int8

   1
```

```
int16( 1)
```

```
ans = int16

   1
```

```
int32( 1)
```

```
ans = int32

   1
```

```
int64( 1)
```

```
ans = int64

   1
```

```
uint8( 1)
```

```
ans = uint8

   1
```

```
uint16( 1)
```

```
ans = uint16

   1
```

```
uint32( 1)
```

```
ans = uint32

   1
```

```
uint64( 1)
```

```
ans = uint64

   1
```

The reason that types like int8 exist is for use in **embedded** systems, like a handheld calculator, in which the amount of memory available is small.

Note that int32 and int64 take up the same amount of memory as single and double, respectively, but they are, of course, different data types. The integer data types can only represent integers, not floating-point numbers.

The unsigned types can represent positive integers twice as large, since they do not use an extra bit to store the sign.

```
intmax( 'int8')
```

```
ans = int8

   127
```

```
intmax( 'int16')
```

```
ans = int16

   32767
```

```
intmax( 'int32')
```

```
ans = int32

   2147483647
```

```
intmax( 'int64')
```

```
ans = int64

   9223372036854775807
```

```
intmax( 'uint8')
```

```
ans = uint8

   255
```

```
intmax( 'uint16')
```

```
ans = uint16

   65535
```

```
intmax( 'uint32')
```

```
ans = uint32

   4294967295
```

```
intmax( 'uint64')
```

```
ans = uint64

   18446744073709551615
```

<u>Rounding</u>

By default, values are rounded to the nearest integer when casting to integer types:

```
int32( 0.49)
```

```
ans = int32

   0
```

```
int32( 0.5)
```

```
ans = int32

   1
```

```
int32( 0.51)
```

```
ans = int32

   1
```

This is the same behavior as the `round` function:

```
round( 0.49)
```

```
ans =
     0
```

```
round( 0.5)
```

```
ans =
     1
```

```
round( 0.51)
```

```
ans =
     1
```

MATLAB also has rounding functions with other behavior:

The `floor` function always its input down:

```
floor( 0.49)
```

```
ans =
     0
```

```
floor( 0.5)
```

```
ans =
     0
```

```
floor( 0.51)
```

```
ans =
     0
```

The `ceil` function always rounds its input up:

```
ceil( 0.49)
```

```
ans =
     1
```

```
ceil( 0.5)
```

```
ans =
     1
```

```
ceil( 0.51)
```

```
ans =
     1
```

The `fix` function always rounds its input towards 0:

```
fix( 0.49)
```

```
ans =
     0
```

```
fix( 0.5)
```

```
ans =
     0
```

```
fix( 0.51)
```

```
ans =
     0
```

```
fix( -0.49)
```

```
ans =
     0
```

```
fix( -0.5)
```

```
ans =
     0
```

```
fix( -0.51)
```

```
ans =
     0
```

Values above the maximum or below the minimum representable numbers are reduced to the maximum and minimum, respectively, when casting to integer types:

```
int8( 200)
```

```
ans = int8
```

```
    127
```

```
int8( -200)
```

```
ans = int8

    -128
```

Numeric types can be converted to characters by the functions `num2str` and `int2str`:

```
num2str( pi)
```

```
ans = 
'3.1416'
```

The precision can be modified by a second input argument. If there is more than one input argument, the arguments are separated by commas:

```
num2str( pi, 10)
```

```
ans = 
'3.141592654'
```

```
int2str( 3)
```

```
ans = 
'3'
```

These functions are useful for concatenating character arrays to be printed.

Logicals

Logical variables have two values, which translate to 1 and 0:

```
true
```

```
ans = logical

    1
```

```
false
```

```
ans = logical

    0
```

At the circuit level, they represent whether current is on or off.

There are various functions, which are useful for **debugging**, that check the data type of a variable:

```
isnumeric( 1)
```

```
ans = logical

    1
```

```
isa( int32( 1), 'integer')
```

```
ans = logical

   1
```

```
isa( 1, 'float')
```

```
ans = logical

   1
```

```
isa( 1, 'double')
```

```
ans = logical

   1
```

```
isa( single( 1), 'single')
```

```
ans = logical

   1
```

```
ischar( 'abc')
```

```
ans = logical

   1
```

```
islogical( true)
```

```
ans = logical

   1
```

(Bugs are mistakes in the code, such that the code as it is written does not reflect the programmer's intentions.)

Random Number Generation

There are also some commonly used built-in functions to generate numbers:

The `rand` function samples from the uniform distribution on the interval $(0, 1)$:

```
rand
```

```
ans =
     7.512670593056529e-01
```

```
rand
```

```
ans =
     2.550951154592691e-01
```

```
rand
```

```
ans =
     5.059570516651424e-01
```

There are various algorithms to generate **pseudorandom** numbers, with various tradeoffs. Random number generation is a deep and difficult field: search for "Creating and Controlling a Random Number Stream" in the documentation.

It is easy to scale the return values of rand to cover a larger interval:

```
2 * rand
```

```
ans =
     1.398153445313372e+00
```

The `randn` function samples from the standard normal distribution (i.e. with mean 0 and standard deviation 1):

```
randn
```

```
ans =
     1.100610217880866e+00
```

```
randn
```

```
ans =
     1.544211895503951e+00
```

```
randn
```

```
ans =
     8.593113317542546e-02
```

The return value of randn can be translated so that the mean is different:

```
randn + 1
```

```
ans =
    -4.915903106376094e-01
```

The `randi` function samples from the discrete uniform distribution on the integers from 1 to its input:

```
randi( 10)
```

```
ans =
     2
```

```
randi( 10)
```

```
ans =
     3
```

```
randi( 10)
```

```
ans =
     9
```

Operators

Arithmetic Operators

We list all the familiar **arithmetic** operators with their names:

```
1 + 1 % plus
```

```
ans =
     2
```

```
+1 % uplus
```

```
ans =
     1
```

```
1 - 1 % minus
```

```
ans =
     0
```

```
-1 % uminus
```

```
ans =
    -1
```

```
2 * 2 % mtimes
```

```
ans =
     4
```

Addition (or subtraction) and multiplication are called **floating-point operations** (**flops**).

```
1 / 2 % mrdivide
```

```
ans =
     5.000000000000000e-01
```

Note that MATLAB has a left-divide, which will be useful for matrices:

```
1 \ 2 % mldivide
```

```
ans =
     2
```

Depending on the ALU, addition and multiplication may take the same number of clock cycles, or may be somewhat different. Division, however, is usually more expensive, by a factor of about 4.

```
2 ^ 2 % mpower
```

```
ans =
     4
```

Note that **rounding errors** in floating-point numbers propagate and may become larger with floating-point operations:

```
1 + 1e-16
```

```
ans =
     1
```

Note that 1e-16 is less than half of machine epsilon:

```
eps
```

```
ans = 
     2.220446049250313e-16
```

Therefore the previous result was rounded down to 1.

As another example, `sin` is one of the built-in mathematical functions in MATLAB:

```
sin( pi)
```

```
ans = 
     1.224646799147353e-16
```

Note that the answer in floating point is not 0, but something close to machine epsilon.

A large part of numerical analysis is deriving upper bounds for how large rounding errors become in numerical algorithms.

A common way in which digits of accuracy are lost is cancellation through subtraction:

```
1.1111111111111111 - 1.1100000000000000
```

```
ans = 
     1.111111111111063e-03
```

Any remaining digits, after the digits of accuracy, are determined by rounding errors and are random.

If one operand is a single and the other is a double, then the result is converted to a single, since the result will only be accurate to single-precision:

```
single( 1) + double( 1)
```

```
ans = single

     2
```

If one operand is an integer and the other is a double, then the result is converted to an integer of the same type:

```
double( 1) - int32( 1)
```

```
ans = int32

   0
```

In general, MATLAB returns the type with lower precision.

An error is thrown if one operand is an integer and the other is a single:

```
int32( 1) * single( 1)
```

The operands also cannot be integers of different type:

```
int8( 1) / int16( 1)
```

In these cases, you must cast one operand into the type of the other in order to use binary operators.

Special Values

There are some other built-in functions that provide important constants:

`Inf` is the result of expressions that are larger than `realmax` or of division by 0:

```
1 / 0
```

```
ans =
   Inf
```

```
Inf / 0
```

```
ans =
   Inf
```

```
Inf + Inf
```

```
ans =
   Inf
```

Note that there is also a negative `Inf`:

```
-Inf * -Inf
```

```
ans =
   Inf
```

```
-2 * realmax
```

```
ans =
  -Inf
```

```
2 ^ 10000
```

```
ans =
   Inf
```

`NaN` is the result of indeterminate expressions:

```
0 / 0
```

```
ans =
   NaN
```

```
Inf / Inf
```

```
ans =
   NaN
```

```
Inf - Inf
```

```
ans =
   NaN
```

If you get `Inf` or `NaN`, it usually means that something has gone wrong in your computations.

The following functions, which are useful for debugging, exist for testing types:

```
isnan( NaN)
```

```
ans = logical

   1
```

```
isinf( Inf)
```

```
ans = logical

   1
```

```
isfinite( 1)
```

```
ans = logical

   1
```

Complex Numbers

MATLAB contains complex number types, but only for floating-point components.

```
a = 1 + 1i
```

```
a = 
      1.000000000000000e+00 + 1.000000000000000e+00i
```

Note that either `i` or `j` can be used for the imaginary unit:

```
single( 1) + single( 1) * j
```

```
ans = single

  1.0000000e+00 + 1.0000000e+00i
```

It is good practice to put a floating-point number in front of `i` or `j` to distinguish them from any variables that have been declared with those names (e.g. loop indices).

They can also be created by the `complex` function, whose input arguments are their real and imaginary parts:

```
complex( 1, 1)
```

```
ans = 
      1.000000000000000e+00 + 1.000000000000000e+00i
```

The `real` and `imag` functions return these respective parts of their input numbers:

```
real( a)
```

```
ans = 
     1
```

```
imag( a)
```

```
ans = 
     1
```

The following function exists for testing types:

```
isreal( 1 + 1i)
```

```
ans = logical

   0
```

The `abs` function gives the absolute value of its input:

```
abs( 1 + 1i)
```

```
ans =
     1.414213562373095e+00
```

The `sqrt` and `log` built-in functions will return complex values.

```
sqrt( -1)
```

```
ans =
      0.000000000000000e+00 + 1.000000000000000e+00i
```

```
log( -1)
```

```
ans =
      0.000000000000000e+00 + 3.141592653589793e+00i
```

Relational Operators

These are the **relational** operators:

```
int32( 1) == int32( 1) % eq
```

```
ans = logical

   1
```

Note that the equality operator should never be used to compare floating-point numbers that have been computed, since, even if they ought to be equal, they might not be in reality due to rounding errors. In **compiled** languages, the compiler usually issues a warning about this. You should always compare the magnitude of their difference with epsilon at that number: `abs( a - 2) <= eps( 2)`.

```
int32( 1) ~= int32( 2) % ne
```

```
ans = logical

   1
```

Relational operators may be used with different numeric types:

```
int32( 2) > int16( 1) % gt
```

```
ans = logical

   1
```

```
int32( 1) >= int32( 1) % ge
```

```
ans = logical

   1
```

```
int32( 1) < int32( 2) % lt
```

```
ans = logical

   1
```

```
int32( 1) <= int32( 1) % le
```

```
ans = logical

   1
```

Infinities are equal, but not-a-numbers are not:

```
Inf == Inf
```

```
ans = logical

   1

      NaN == NaN
```

Relational operators for complex numbers only consider the real part:

```
a = 2 + 1i
```

```
a =
      2.000000000000000e+00 + 1.000000000000000e+00i
```

```
b = 1 + 3i
```

```
b =
      1.000000000000000e+00 + 3.000000000000000e+00i
```

```
a > b
```

```
ans = logical

   1
```

Logical Operators

These are the logical operators:

```
true & true % and
```

```
ans = logical

   1
```

```
true | false % or
```

```
ans = logical
```

```
    1
```

```
~0 % not
```

```
ans = logical

    1
```

Note that we can use the numerical equivalents of `true` and `false` as well. There are also functions that perform the same operations:

```
and( true, false)
```

```
ans = logical

    0
```

```
or( true, true)
```

```
ans = logical

    1
```

```
not( 1)
```

```
ans = logical

    0
```

There are also "short-circuited" versions. The && operator immediately returns false if the first operand is false:

```
false && true
```

```
ans = logical

    0
```

The || operator immediately returns true if the first operand is true:

```
true || false
```

```
ans = logical

    1
```

Note that for the exclusive "or", there is no operator, but only a function:

```
xor( true, true)
```

```
ans = logical

    0
```

Precedence

MATLAB follows the usual order of operations. The exact list is somewhat complicated: search for "Operator Precedence" in the documentation.

Precedence rules can be overriden by using parentheses:

```
1 + 2 * 3
```

```
ans =
     7
```

```
(1 + 2) * 3
```

```
ans =
     9
```

It is important to note that & is always given precedence over |, and precendence is not determined left to right in this case:

```
(true | false) & false % == false
```

```
ans = logical

   0
```

```
true | (false & false) % == true
```

```
ans = logical

   1
```

```
true | false & false % == true
```

```
ans = logical

   1
```