MATLAB Basics
As a Programming Language
(2)

Some Basic Math Functions

These functions are element-wise when applied to arrays:

- Rounding functions: **`round`**, **`ceil`**, **`floor`**
- The use of **`abs`** (different meanings for real and complex numbers)
- Functions: **`exp`** and **`log`**, (also **`log2`** and **`log10`**)
- Functions: **`sin`**, **`cos`**, **`tan`**, **`asin`**, **`acos`**, **`atan`**, **`atan2`**
  - Degree-based versions by attaching **`d`** to triangular function names (such as **`sind`**)
- A variable with default value: **`pi`**

Basic Vector/Array Operations

- Vector functions: **`sum`** and **`mean`**
- Vector functions: **`cumsum`** and **`cumprod`**
- Using vector functions on multi-dimensional arrays:
  - Specifying the dimension
  - Whole-array sum and mean
- Functions: **`min`** and **`max`**
  - Array-and-scalar or array-and-array (element-wise)
  - Single-array vector-wise operation: **`min(X,[],dim)`**
- Sorting: **`sort`**
  - Getting the index
- Functions for logical vectors: **`any`** and **`all`**

Logical Indexing

- Logical indexing: Using logical values (true or false) to indicate which array elements to select.
  - Selecting array elements by logical indexing
  - Assignment with logical indexing
  - An example: Thresholding.
- Function: **`find`**
  - linear index output
  - subscripts output

Basic Matrix Operations

- You've got to know some linear algebra …
- Some useful operations:
  - For vectors: functions **`dot`** and **`cross`**
  - For square matrices: functions **`diag`** and **`trace`**
  - transpose: operators **`'`** (complex conjugate transpose) and **`.'`** (regular transpose); they are the same for real numbers.
  - Functions: **`fliplr`** and **`flipud`**
  - Matrix operations: **`*`**, **`/`**, **`^`**
  - Solving a set of linear equations: Operator **`\`**
  - Also used a lot: functions **`norm`**, **`inv`** and **`eig`**
  - Many more …

Using Matrix Operator '\'

- Solving a set of multi-variable linear equations:
  - The set of equations is expressed as $\boldsymbol{Az=y}$ ($\boldsymbol{A}$ is a square matrix; $\boldsymbol{z}$ and $\boldsymbol{y}$ are column vectors)
  - $\boldsymbol{z}$ (the unknowns) is solved by $\boldsymbol{A\backslash y}$
- Min-squared-error approximation (example: line fitting):
  - To find a line $y=ax+b$ that approximates the points (x_1,y_1), (x_2,y_2), … (x_n,y_n), where $n>2$.
  - Set of equations in the form of $\boldsymbol{Az=y}$:

    $\boldsymbol{A} = [x_1\ 1;\ x_2\ 1;\ \ldots\ ;\ x_n\ 1]$

    $\boldsymbol{y} = [y_1;\ y_2;\ \ldots\ y_n]$

    $\boldsymbol{z} = [a;\ b]$
  - $\boldsymbol{z}$ (the unknowns) is solved by $\boldsymbol{A\backslash y}$

Vectors as Sets

- Set operations: Functions: **`intersect`**, **`union`**, **`setdiff`**, **`setxor`**
  - Using returned indices
- A related function: **`unique`**

Scripts and m files

- Scripts: collections of statements saved in a file (**m** file)
- Scripts use the global scope (i.e., the workspace) for their variables, so they share variables with interactive (command line) statements.
- Using the editor
- Comments (symbol %) and block comments
- Line continuation (symbol ...)

Program Flow Control Overview

- Usage similar to C.
- Conditional branching:
  - `if … elseif … else … end`
  - `switch … case … otherwise … end`
- Loops:
  - `while … end`
  - `for … end`

Program Flow Control

- Using **`if … elseif … else … end`**:

```
if expression
   statements
elseif expression
   statements
…
else
   statements
end
```

- Can be nested.
- Use scalar for the conditions. (Vector conditions are handled with AND.)
- Numerical conditions are treated as true if non-zero and false if zero.

Program Flow Control

- Using **`switch … case … otherwise … end`**:

```
switch switch_expression
case case_expression
   statements
case case_expression
   statements
…
otherwise
   statements
end
```

Program Flow Control

- Using **`switch`** … **`case`** … **`otherwise`** … **`end`**:
  - Only the statements under one **`case`** (or **`otherwise`**) are executed (different from C)
  - **`otherwise`** is optional
  - A case expression can contain multiple choices to be matched to the switch expression.
    - Example: **`case {1, 2}`**
    - That is actually a cell array (to be discussed later).
  - **`switch`** and **`case`** expressions can be strings (to be discussed later)

Program Flow Control

- Using **`while`** … **`end`**:

```
while expression
   statements
end
```

- Same as in C

Program Flow Control

- Using **`for … end`**:

```
for control_variable = values
   statements
end
```

- Here **`control_variable`** is a variable name. (Avoid using **`i`** and **`j`**, as common in C, as they also represent the imaginary number **sqrt(-1)**.)
- In the k^{th} iteration, index is the k^{th} column (a row vector) of ***`values`***.
  - For normal use, set ***`values`*** to a row vector to avoid confusion. This makes **`control_variable`** a scalar.

Program Flow Control

- Using **`break`**: Terminate the current (inner-most) **`while`** or **`for`** loop.
- Using **`continue`**: Terminate the current iteration of the current **`while`** or **`for`** loops, and start next iteration.
- Same as in C

Vectorization

- MATLAB uses an interpreter, and it has a lot more overhead when executing a statement than when the program is compiled as in C.
- Built-in vector and matrix operations, on the other hand, have mostly been optimized with implementation in C.
- The motto here: Whenever possible, "vectorize" your operations for better efficiency, even if this appears less efficient on the surface.
- Example: Comparison of two equivalent implementations:

```
A = rand(1,2e8);  B = zeros(size(A));
for ii=1:2e8; B(ii)=A(ii)*A(ii); end % loop
B=A.*A; % vectorized
```