Unit 3: Custom Functions

Custom Functions in MATLAB

- Put functions in **m files**.
  - The file name is the name of the first function by default.
- Function header format: `function [a,b,c]=fn(d,e,f)`
  - Left side: output arguments (optional)
  - Right side: input arguments (optional)
- Adding **help text**: (comments at the top of the function code)
- To exit early: keyword `return`
- Error message: function `error`
- Recursion:

Custom Functions in MATLAB

- Each function has its own scope of variables.
  - Variables can be shared among multiple functions using the keyword **`global`**: Functions that declare the same variable name as global will share that variable. (Use with caution!)
- Variable passing (not exactly, just to understand the effect):
  - Input argument not in output and changed inside the function: **Pass by value**
  - Otherwise: **Pass by reference**
- Variable-length argument lists: the use of functions **`nargin`** and **`nargout`**

An Example Function

```
function [a, b] = fn(x, y)
if nargin == 2
  if nargout == 1
    a = x + y;
  else
    a = x; b = y;
  end
else
  a = x; b = x * 2;
end
```

An Example Function

- Try calling `fn` in the previous slide with 0, 1, or 2 input and output arguments.
- This is how MATLAB handles "function overloading": by checking input and output arguments.
  - You can do additional argument checking (for example, to see whether a certain input is a scalar or not).
  - Meanings of the output arguments can be affected by the number of output arguments.
- Many examples in MATLAB documentations.
  - For example, how will you implement the various ways of using the function `max`?

Cell Arrays

- (We introduce cell arrays here so that you can understand the syntax for handling variable-length function arguments. However, they are useful in many other ways.)
- A cell array is different from a regular array in that its elements can be about anything:
  - Scalars, other arrays (regular or cell), structures (discussed in the future), strings, etc.
  - Can be of multiple dimensions.
  - Use `{}` as the indexing operator for accessing individual elements.
  - Use `()` as the indexing operator for accessing sub-arrays.
  - Can be used to store "array of arrays", such as the courses taken by the students in a class, or the neighbors of the vertices in a graph.

Variable-Length Inputs

- Use **`varargin`** as the last input argument in the function header.
  - **`varargin`** will be a cell array containing all the extra inputs in the function call.
- Example function:

```
function v = fn(a, varargin)
v = a;
for ii = 1:length(varargin)
  v = v + varargin{ii};
end
```

Variable-Length Outputs

- Use **`varargout`** as the last output argument in the function header.
  - **`varargout`** is a cell array containing all the extra outputs requested in the function call.
- Example function:

```
function [v, varargout] = fn(a)
nout = nargout - 1;
v = a;
t = a;
for ii = 1:nout
  t = t * a;
  varargout{ii} = t;
end
```

Ignoring Outputs in Function Calls

- When calling a function, if some of the outputs are not needed, you can use the tilde (**~**) operator in their places instead of saving those unwanted outputs to dummy variables.

Local Functions

- The first function in a function (non-script) m-file is the only function visible outside the file.
  - A script m-file can not contain functions.
- Additional functions within the same file are local functions that are visible only within this file.
  - Orders are not important.
  - Individual functions can be terminated by keywords **`end`** or **`return`** for readability, but these are not required.

Unit 4: String Processing

Basics of String Processing

- Strings are arrays, too! (Specifically, they are called **character arrays** in MATLAB.)
  - Each character is an array element.
  - Consider **character** as a MATLAB data type.
  - Character arrays can have more than one dimensions.
  - Many array operations (concatenation, sub-array, etc.) are applicable to character arrays.
- Direct specification of strings: single quotation marks.
- Pre-allocation of character arrays: Function **`blanks`**

Basics of String Processing

- Check data type: Function **`ischar`**
- Type conversion (casting) between numerical and character data:
  - Numerical to character: Function **`char`**
  - Character to numerical: **`int8`**, **`double`**, etc.
  - Encoding: ASCII for 0-127, the rest depends on system locale setting.

Strings from Other Data Types

- Strings from simple numerical data (default formatting): Function **`num2str`**
- Generating formatted strings: Function **`sprintf`**
  - C-style formatting
  - Repeated use of the format string (the same as **`fprintf`**).

Strings to Other Data Types

- Strings to numerical data (default format specifier):
  - **`str2double`**
  - **`str2num`**
- Data from formatted strings: Function **`sscanf`**
  - C-style formatting
  - Repeated use of the format string.
  - Output is a single array; the data type depends on the format specifier.

Representing Multiple Strings

- Method 1: 2-D character array
  - Each row is a string.
  - Shorter strings are padded with blanks.
- Method 2: cell array of strings
  - Each element is a string.
  - Example: `{'apple', 'orange', 'kiwi'}`
  - Easier to process

Useful String Functions

- Element-wise property: **`isletter`** and **`isspace`**
- Find and replace: **`strfind`** and **`strrep`**
- Splitting by delimiters:
  - **`strsplit`** (output is a cell array of strings)
  - **`strtoken`** (output only one substring)
- Simple formatting:
  - Removing white spaces: **`deblank`** and **`strtrim`**
  - Change case: **`lower`** and **`upper`**
  - Justification: **`strjust`**

Useful String Functions

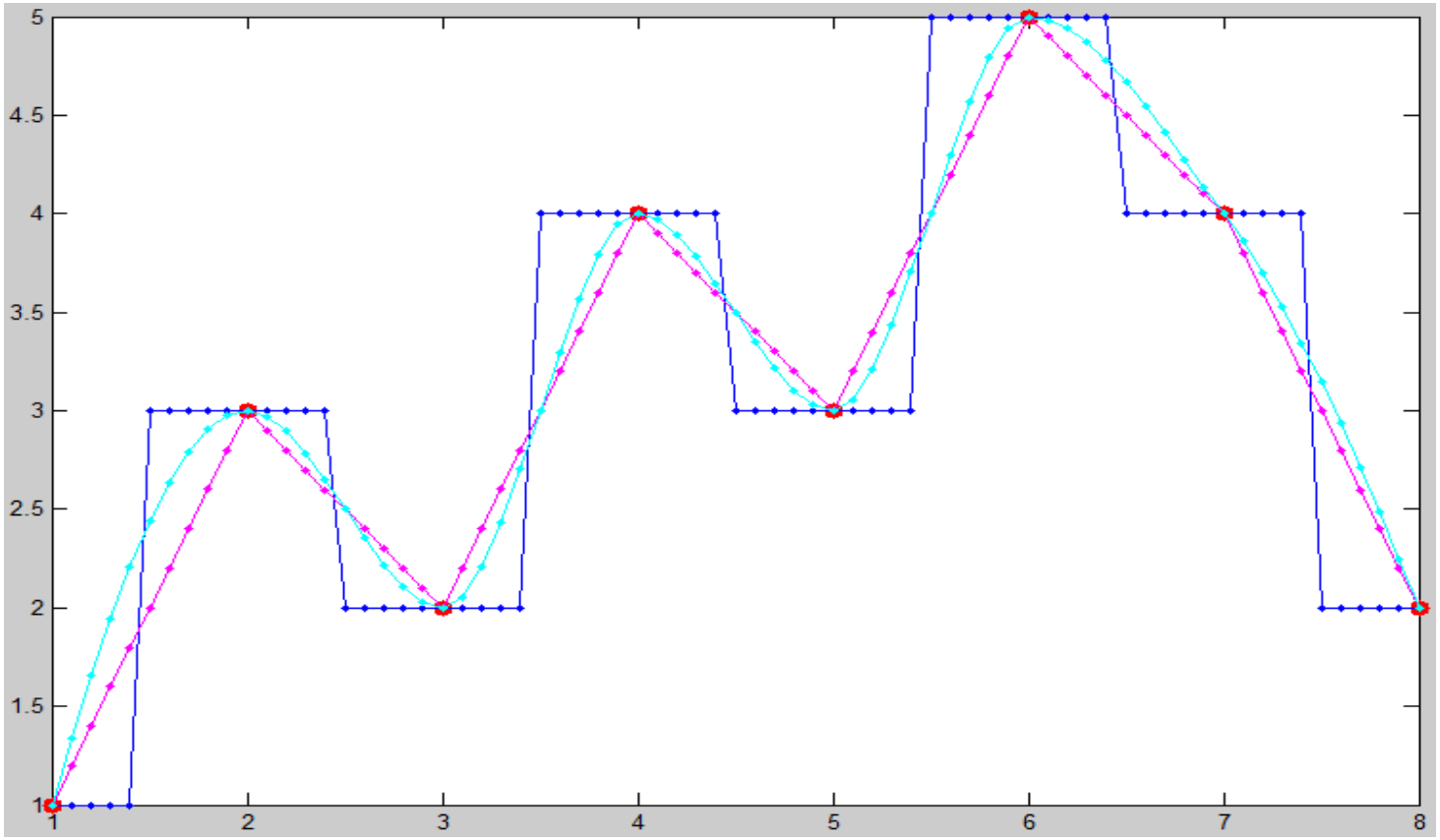
- Comparing strings:
  - **`strcmp`**, **`strcmpi`**, **`strncmp`**, **`strncmpi`**
    - Output is logical
    - Can compare a string with a cell array of strings
  - **`validatestring`** (mainly used for checking function input arguments that are strings)

Unit 5: Some Miscellaneous but Useful Topics

Interpolation

- Example with **`interp1`** (1-D):
  - Syntax: **`yy=interp1(x,y,xx,method)`** interpolates point pairs in **`x`** and **`y`** to point pairs in **`xx`** and **`yy`**.
  - Input **`method`** (a string) is optional (the default is **`'linear'`**). Common choices are: **`'nearest'`**, **`'linear'`**, **`'cubic'`**, **`'spline'`**.
- For 2-D and 3-D: **`interp2`** and **`interp3`**.

Interpolation




```
x=1:8; y=[1 3 2 4 3 5 4 2];
figure; clf; plot(x,y,'.r','markersize',24); hold on;
xx=1:.1:8; yy=interp1(x,y,xx,'nearest'); plot(xx,yy,'-b.');
xx=1:.1:8; yy=interp1(x,y,xx,'linear'); plot(xx,yy,'-m.');
xx=1:.1:8; yy=interp1(x,y,xx,'cubic'); plot(xx,yy,'-c.');
```

Polynomials

- A polynomial is represented by a vector containing its coefficients (from high to low-order terms).
  - Example: `[3 0 -1 2]` represents $3x^3-x+2$.
- Function **`polyval`**: Evaluate a polynomial for a given value of its variable.
  - Example: **`v=polyval([3 1 5],2);`**
- Function **`polyfit`**: Fitting a polynomial to a set of points.
  - Syntax: **`p=polyfit(x,y,deg);`**
  - Example: **`p=polyfit([1 1 2 3],[1 2 3 4],2);`**

Basic Timing

- Using `tic` and `toc` to get rough running time.
- Lets do some examples here …

More on Random Numbers

- We have seen the use of the function **`rand`**:
- Two other useful functions:
  - **`randn`**: normal distributed random numbers
  - **`randperm`**: random permutation of **`1:n`** (can be used to create random permutation of other vectors)
- Initializing a random number generator: Function **`rng`**.

Basics of File Processing

- Three different types of files:
  - MAT files for MATLAB variables
  - Text files
  - Binary files

MAT Files

- Matlab's own file format for Matlab variables
- Writing: `save(file_name, 'variable_name', …);`
  - Can save any number of variables; variable names are saved as well as the data.
- Reading: `load(file_name, 'variable_name', …);`
  - Loaded variables put in current work space.
  - Variable names are optional; if not specified, all the variables in the file are loaded.
  - `A = load(…)`: the loaded variables become fields of the structure variable `A` (to be discussed later).

Text Files

- Regular text files that can be used by other applications.
- Open a file: `fid = fopen(file_name, mode)`
  - mode: `'rt'` for reading, `'wt'` for writing; see documentation for other modes.
  - Usage very similar to C.
- Close a file: `fclose(fid)`
- Writing: `fprintf(fid, format_string, …);`
  - When `fid` is omitted, it writes to the screen.

Text Files

- Reading:
  - `s = fgetl(fid)`: Read a line.
  - `A = fscanf(fid, format_string)`: Read multiple values into a double array `A`; reuse `format_string` if necessary.
  - `C = textscan(fid, format_string)`: `C` is an array of cell arrays; each element (a cell array) in `C` represents the values corresponding to one place holder in `format_string`.
    - Suitable for reading in table-like text files.

Binary Files

- Open a file: `fid = fopen(file_name, mode)`
  - mode: `'rb'` for reading, `'wb'` for writing; see documentation for other modes.
  - Usage very similar to C.
- Close a file: `fclose(fid)`
- Writing: `fwrite`
- Reading: `fread`
  - Data formats can be specified (and converted) during reading/writing. See documentation.