Custom Functions

From now on, if you've only got limited programming experience, you may encounter unfamiliar concepts and terminologies from time to time. As we do not have time to cover them sufficiently, you are advised to learn about them by yourself.

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 of "`=`": output arguments (optional)
  - Right side of "`=`": input arguments (optional)
- Adding **help text**: (comments at the top of the function code)
- To exit early: keyword `return`
- To exit by throwing an error: function `error` (You can also include a text message to be displayed.)
  - Exception handling to be discussed later

Example Functions

```
% Return the max and min of two inputs
function [a, b] = max_and_min(x, y)
  a = max(x, y);
  b = min(x, y);

% Euclidian algorithm for
% greatest common divisor
function g = gcd(x, y)
  [a, b] = max_and_min(x, y);
  while b > 1
    t = mod(a, b);
    a = b;
    b = t;
  end
  g = a;
```

Optional Arguments

- When some arguments are optional: Use functions **`nargin`** and **`nargout`** to get the actual numbers of input and output arguments at the function call.
  - The function definition should include the maximum numbers of input/output arguments allowed.
- 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 actual numbers of input/output arguments.
- Many examples in standard MATLAB functions, such as **`max`** or **`find`**.

Example Functions

```
% Return the max and min of two inputs
% If there is only one input that is
% an array, it max and min are returned.
function [a, b] = max_and_min(x, y)
  if nargin == 2
    a = max(x, y);
    b = min(x, y);
  else % nargin == 1
    a = max(x(:));
    b = min(x(:));
  end
```

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 almost anything:

- Scalars, other arrays (regular or cell), structures and objects (discussed later), 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 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:

  (at least one input argument required in this example)

```
function v = var_sum(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] = var_pow(a)
  varargout = cell(1, nargout-1);
  v = a;  t = a;
  for ii = 1:nargout-1
    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.
- An example, when we only want the index of the largest element in a vector, but not its value:

```
A = rand(1, 5)
[~, k] = max(A)
```

Local Functions

- The first function in a function (non-script) m-file is the only function visible from outside of the file.
  - Note: 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.
  - Each new **`function`** keyword starts a new function within the file.
  - Individual functions can be terminated by keywords **`end`** or **`return`** for readability, but these are not required.

Function Recursion

- In programming, **recursion** represents techniques that let a function call itself either directly or indirectly.
- Important: Each call to the same function has its own set of local variables.
- Due to the memory overhead associated with function calls, there is a limit on the maximum levels of recursion.
- Example: The Euclidean Algorithm of finding the greatest common divisor (gcd):

```
function g = gcd_recursion(a, b)
  [a, b] = max_and_min(x, y);
  if b == 0;  g = a;  return;  end
  g = gcd_recursion(b, mod(a,b));  % recursion
```

Input Argument Validation

- This is important for writing robust functions that do not crash or do weird things when given unexpected inputs.
- The purpose: Stop the execution of the function when some input arguments are not acceptable for valid behaviors.
- While we can use many conditional statements (**`if`**, **`switch`**) for this purpose, function **`validateattributes`** provides a systematic way of validating individual inputs.

Input Argument Validation

```
function g = gcd(a, b)
  validateattributes(a, ...
        {'numeric'}, ...
        {'scalar','integer','positive'});
  validateattributes(b, ...
        {'numeric'}, ...
        {'scalar','integer','>=',1});
  [a, b] = max_and_min(x, y);
  while b > 1
    t = mod(a, b);
    a = b;
    b = t;
  end
  g = a;
```

Additional Function Concepts

- 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 arguments not in output and changed inside the function: **Pass by value**
  - Otherwise: **Pass by reference**