Functions

Function Handles (@)

- Callable association to MATLAB function stored in variable
  - Enables invocation of function outside its normal scope
  - Invoke function indirectly
  - Variable

Function Handles (@)

- Callable association to MATLAB function stored in variable
  - Enables invocation of function outside its normal scope
  - Invoke function indirectly
  - Variable
- Capture data for later use

Function Handles (@)

- Callable association to MATLAB function stored in variable
  - Enables invocation of function outside its normal scope
  - Invoke function indirectly
  - Variable
- Capture data for later use
- Enables passing functions as arguments

Function Handles (@)

- Callable association to MATLAB function stored in variable
  - Enables invocation of function outside its normal scope
  - Invoke function indirectly
  - Variable
- Capture data for later use
- Enables passing functions as arguments
  - Optimization
  - Solution of nonlinear systems of equations
  - Solution of ODEs
  - Numerical Integration

Function Handles (@)

- Callable association to MATLAB function stored in variable
  - Enables invocation of function outside its normal scope
  - Invoke function indirectly
  - Variable
- Capture data for later use
- Enables passing functions as arguments
  - Optimization
  - Solution of nonlinear systems of equations
  - Solution of ODEs
  - Numerical Integration
- Function handles must be scalars, i.e. can't be indexed with ()

Example

- Trapezoidal rule for integration

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n_{el}} \frac{b-a}{2n_{el}} \left[f(x_{i+1/2}) + f(x_{i-1/2}) \right]$$

```
function int_f = trap_rule(f,a,b,nel)

x=linspace(a,b,nel+1)';
int_f=0.5*((b-a)/nel)*sum(f(x(1:end-1))+f(x(2:end)));

end
```

```
>> a = exp(1);
>> f = @(x) a*x.^2;
>> trap_rule(f,-1,1,1000) % (2/3)*exp(1) = 1.8122
ans =
    1.8122
```

Outline

1. Data Types
   - Numeric Arrays
   - Cells & Cell Arrays
   - Struct & Struct Arrays
   - Function Handles
2. Functions and Scripts
   - Function Types
   - Workspace Control
   - Inputs/Outputs
   - Publish
3. MATLAB Tools
4. Code Performance

Scripts vs. Functions

- *Scripts*
  - Execute a series of MATLAB statements
  - Uses *base* workspace (does not have own workspace)
  - Parsed and loaded into memory every execution

Scripts vs. Functions

- *Scripts*
  - Execute a series of MATLAB statements
  - Uses *base* workspace (does not have own workspace)
  - Parsed and loaded into memory every execution
- *Functions*
  - Accept inputs, execute a series of MATLAB statements, and return outputs
  - *Local* workspace defined only during execution of function
    - `global`, `persistent` variables
    - `evalin`, `assignin` commands
  - Local, nested, private, anonymous, class methods
  - Parsed and loaded into memory during *first* execution

Anonymous Functions

- Functions without a file
  - Stored directly in function handle
  - Store expression and required variables
  - Zero or more arguments allowed
  - Nested anonymous functions permitted
- Array of function handle not allowed; function handle may return array

Anonymous Functions

- Functions without a file
  - Stored directly in function handle
  - Store expression and required variables
  - Zero or more arguments allowed
  - Nested anonymous functions permitted
- Array of function handle not allowed; function handle may return array

```
>> f1 = @(x,y) [sin(pi*x), cos(pi*y), tan(pi*x*y)];
>> f1(0.5,0.25)
ans =
    1.0000    0.7071    0.4142
>> quad(@(x) exp(1)*x.^2,-1,1)
ans =
    1.8122
```

Local Functions

- A given MATLAB file can contain multiple functions
  - The first function is the *main* function
    - Callable from anywhere, provided it is in the search path
  - Other functions in file are *local* functions
    - Only callable from main function or other local functions in *same* file
    - Enables modularity (large number of small functions) without creating a large number of files
    - Unfavorable from code reusability standpoint

Local Function Example

Contents of `loc_func_ex.m`

```
function main_out = loc_func_ex()
main_out = ['I can call the ',loc_func()];
end

function loc_out = loc_func()
loc_out = 'local function';
end
```

Command-line

```
>> loc_func_ex()
ans =
I can call the local function

>> ['I can''t call the ',loc_func()]
??? Undefined function or variable 'loc_func'.
```

Nested Functions

- A *nested function* is a function *completely* contained within some parent function.
- Useful as an alternative to *anonymous* function that can't be confined to a single line
- Can't be defined within MATLAB control statements
  - `if`/`elseif`/`else`, `switch`/`case`, `for`, `while`, or `try`/`catch`
- Variables *sharable* between parent and nested functions
- If variable in nested function not used in parent function, it remains local to the nested function
- Multiple levels of nesting permitted
  - Nested function available from
    - Level immediately above
    - Function nested at same level with same parent
    - Function at any lower level

Nested Functions: Example

- Parent and nested function can share variables

```
function nested_ex1
x = 5;
nestfun1;
   function nestfun1
     x = x + 1;
   end
   function nestfun2
     y = 4;
   end
disp(x)   % x = 6
nestfun2
disp(y+1) % y = 5
end
```

Private Functions

- Private functions useful for limiting scope of a function
- Designate a function as `private` by storing it in a subfolder named *private*
- Only available to functions/scripts in the folder immediately above the private subfolder

Evaluate/Assign in Another Workspace

- Eval expression in other workspace (`'caller'`, `'base'`)
  - `evalin`(ws,*expression*)
  - Useful for evaluating expression in caller's workspace *without* passing name variables as function arguments
- Assign variable in other workspace (`'caller'`, `'base'`)
  - `assignin`(ws, `'var'`, val)
  - Useful for circumventing local scope restrictions of functions

Variable Number of Inputs/Outputs

- Query number of inputs passed to a function
  - `nargin`
  - Don't try to pass more than in function declaration
- Determine number of outputs requested from function
  - `nargout`
  - Don't request more than in function declaration

Variable Number of Inputs/Outputs

- Query number of inputs passed to a function
  - `nargin`
  - Don't try to pass more than in function declaration
- Determine number of outputs requested from function
  - `nargout`
  - Don't request more than in function declaration

```
function  [o1,o2,o3] = narginout_ex(i1,i2,i3)
fprintf('Number inputs = %i;\t',nargin);
fprintf('Number outputs = %i;\n',nargout);
o1 = i1; o2=i2; o3=i3;
end
```

```
>> narginout_ex(1,2,3);
Number inputs = 3;  Number outputs = 0;
>> [a,b]=narginout_ex(1,2,3);
Number inputs = 3;  Number outputs = 2;
```

Variable-Length Input/Output Argument List

- Input-output argument list length unknown or conditional
  - Think of `plot, get, set` and the various Name-Property pairs that can be specified in a given function call

Variable-Length Input/Output Argument List

- Input-output argument list length unknown or conditional
  - Think of `plot, get, set` and the various Name-Property pairs that can be specified in a given function call
- Use `varargin` as last function input and `varargout` as last function output for input/output argument lists to be of variable length

Variable-Length Input/Output Argument List

- Input-output argument list length unknown or conditional
  - Think of `plot, get, set` and the various Name-Property pairs that can be specified in a given function call
- Use `varargin` as last function input and `varargout` as last function output for input/output argument lists to be of variable length
- All arguments prior to `varargin`/`varargout` will be matched one-to-one with calling expression

Variable-Length Input/Output Argument List

- Input-output argument list length unknown or conditional
  - Think of `plot, get, set` and the various Name-Property pairs that can be specified in a given function call
- Use `varargin` as last function input and `varargout` as last function output for input/output argument lists to be of variable length
- All arguments prior to `varargin/varargout` will be matched one-to-one with calling expression
- Remaining input/outputs will be stored in a cell array named `varargin/varargout`

Variable-Length Input/Output Argument List

- Input-output argument list length unknown or conditional
  - Think of `plot, get, set` and the various Name-Property pairs that can be specified in a given function call
- Use `varargin` as last function input and `varargout` as last function output for input/output argument lists to be of variable length
- All arguments prior to `varargin/varargout` will be matched one-to-one with calling expression
- Remaining input/outputs will be stored in a cell array named `varargin/varargout`
- `help varargin`, `help varargout` for more information

varargin, varargout Example

```
1 function [b,varargout] = vararg_ex(a,varargin)
2
3 b = a^2;
4 class(varargin)
5 varargout = cell(length(varargin)-a,1);
6 [varargout{:}] = varargin{1:end-a};
7
8 end
```

```
>> [b,vo1,vo2] = ...
    vararg_ex(2,'varargin','varargout','example','!');
ans =
cell
vo1 =
varargin
vo2 =
varargout
```

Publishing Scripts and Functions

- Generate view of MATLAB file in specified format
  - `publish(file,format)`
- Generate view of MATLAB file, fine-grained control
  - `publish(file,Name,Value)`, `publish(file,options)`
- By default, publishing MATLAB file runs associated code
  - Problematic when publishing functions
  - Set `'evalCode'` to `false`
  - File → Publish Configuration for <filename>

Outline

1. Data Types
   - Numeric Arrays
   - Cells & Cell Arrays
   - Struct & Struct Arrays
   - Function Handles
2. Functions and Scripts
   - Function Types
   - Workspace Control
   - Inputs/Outputs
   - Publish
3. MATLAB Tools
4. Code Performance

Debugger

- Breakpoint
- Step, Step In, Step Out
- Continue
- Tips/Tricks
  - Very useful!
  - Error occurs only on 10031 iteration. *How to debug?*

Debugger

- Breakpoint
- Step, Step In, Step Out
- Continue
- Tips/Tricks
  - Very useful!
  - Error occurs only on 10031 iteration. *How to debug?*
    - Conditional breakpoints
    - Try/catch
    - If statements

Profiler

- Debug and optimize MATLAB code by tracking execution time
  - Itemized timing of individual functions
  - Itemized timing of individual lines within each function
  - Records information about execution time, number of function calls, function dependencies
  - Debugging tool, understand unfamiliar file
- `profile` (`on`, `off`, `viewer`, `clear`, `-timer`)
- `profsave`
  - Save profile report to HTML format
- **Demo**: `profiler_ex.m`
- Other performance assessment functions
  - `tic`, `toc`, `timeit`, `bench`, `cputime`
  - `memory`

Outline

1. Data Types
   - Numeric Arrays
   - Cells & Cell Arrays
   - Struct & Struct Arrays
   - Function Handles
2. Functions and Scripts
   - Function Types
   - Workspace Control
   - Inputs/Outputs
   - Publish
3. MATLAB Tools
4. Code Performance

Performance Optimization

- Optimize the algorithm itself
- Be careful with matrices!
  - Sparse vs. full
  - Parentheses
    - `A*B*C*v`
    - `A*(B*(C*v))`
- Order of arrays matters
  - Fortran ordering: Operators with equal precedence evaluated left to right
- Vectorization
  - MATLAB highly optimized for array operations
  - Whenever possible, loops should be re-written using arrays
- Memory management
  - Preallocation of arrays
  - Delayed copy
  - Contiguous memory

Order of Arrays

- Due to Fortran ordering, indexing column-wise is much faster than indexing row-wise
  - Contiguous memory

Order of Arrays

- Due to Fortran ordering, indexing column-wise is much faster than indexing row-wise
  - Contiguous memory

```
mat = ones(1000, 1000); n = 1e6;

tic();
for i=1:n, vec = mat(1,:); end
toc()

tic();
for i=1:n, vec = mat(:,1); end
toc()
```

Vectorization

Toy Example

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

Vectorization

Toy Example

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

Vectorized

```
y = sin(0:.01:10);
```

Vectorization

Toy Example

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

Vectorized

```
y = sin(0:.01:10);
```

Slightly less toy example

```
n = 100;
M = magic(n);
v = M(:,1);
for i = 1:n
    M(:,i) = ...
        M(:,i) - v
end
```

Vectorization

Toy Example

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

Vectorized

```
y = sin(0:.01:10);
```

Slightly less toy example

```
n = 100;
M = magic(n);
v = M(:,1);
for i = 1:n
    M(:,i) = ...
        M(:,i) - v
end
```

Vectorized

```
n = 100;
M = magic(n);
v = M(:,1);
M=bsxfun(@minus,M,v);
```

Memory Management Functions

Command	Description
`clear`	Remove items from workspace
`pack`	Consolidate workspace memory
`save`	Save workspace variables to file
`load`	Load variables from file into workspace
`inmem`	Names of funcs, MEX-files, classes in memory
`memory`	Display memory information
`whos`	List variables in workspace, sizes and types

pack

`pack` frees up needed space by reorganizing information so that it only uses the minimum memory required. All variables from your base and global workspaces are preserved. Any persistent variables that are defined at the time are set to their default value (the empty matrix, []).

- Useful if you have a large numeric array that you *know* you have enough memory to store, but can't find enough *contiguous* memory
- Not useful if your array is too large to fit in memory

Delayed Copy

- When MATLAB arrays passed to a function, only copied to local workspace when it is *modified*
- Otherwise, entries accessed based on original location in memory

Delayed Copy

- When MATLAB arrays passed to a function, only copied to local workspace when it is *modified*
- Otherwise, entries accessed based on original location in memory

```
1 function  b = delayed_copy_ex1(A)
2 b = 10*A(1,1);
3 end
```

```
1 function  b = delayed_copy_ex2(A)
2 A(1,1) = 5; b = 10*A(1,1);
3 end
```

```
>> A = rand(10000);
>> tic; b=delayed_copy_ex1(A); toc
Elapsed time is 0.000083 seconds.
>> tic; b=delayed_copy_ex2(A); toc
Elapsed time is 0.794531 seconds.
```

Delayed Copy

```
1 function  b = delayed_copy_ex3(A)
2 b = 10*A(1,1); disp(A); A(1,1) = 5; disp(A);
3 end
```

```
>> format debug
>> A = rand(2);
>> disp(A) % Output pruned for brevity

pr = 39cd3220

>> delayed_copy_ex3(A);  % Output pruned for brevity

pr = 39cd3220

pr = 3af96320
```

Contiguous Memory and Preallocation

- Contiguous memory
  - Numeric arrays are *always* stored in a contiguous block of memory
  - Cell arrays and structure arrays are not necessarily stored contiguously
    - The contents of a given cell or structure *are* stored contiguously

Contiguous Memory and Preallocation

- Contiguous memory
  - Numeric arrays are *always* stored in a contiguous block of memory
  - Cell arrays and structure arrays are not necessarily stored contiguously
    - The contents of a given cell or structure *are* stored contiguously
- Preallocation of contiguous data structures
  - Data structures stored as contiguous blocks of data should be preallocated instead of incrementally grown (i.e. in a loop)
  - Each size increment of such a data type requires:
    - Location of *new* contiguous block of memory able to store new object
    - Copying original object to new memory location
    - Writing new data to new memory location