Lecture 1
MATLAB Fundamentals: Features, Syntaxes, Concepts

Matthew J. Zahr

CME 292
Advanced MATLAB for Scientific Computing
Stanford University

2nd April 2015

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

Outline

Basic Information

- Grading: Satisfactory/No credit
  - Satisfactory completion of assignments
- Lectures
  - Interactive demos
  - Bring laptop
- Assignments
  - Assigned each Tuesday, due following Tuesday
  - Problem sets will be rather lengthy
    - Only required to complete a *subset* of problems on each
    - Meant for you to pick problems relevant to you
  - Submit files via *Dropbox* on Coursework
    - Create zip file containing all code
    - Additional details given with problem sets

Basic Information

- Very quick survey after first few classes
  - Keep class interesting and inline with your expectations
- Office Hours:
  - Tue/Thurs 5p - 7p (after class) Durand 028
  - Additional office hours, if requested
  - Drop-in/by appointment - Durand 028
- Course Homepage: `http://web.stanford.edu/~mzahr/cme292.html`
- Accessing MATLAB
  - See document on website (Pawin)
    - I recommend using MATLAB instead of an alternative
    - MATLAB isn't paying me to say this, I promise
- MATLAB Help
  - Very useful documentation: Use it!
  - `doc`, `help`
  - `http://www.mathworks.com/help/`

Syllabus

Lecture 1

- Fundamental MATLAB features, syntaxes, concepts
  - Data types
  - Functions/scripts, publishing
  - Debugger, profiler
  - Memory management

Lecture 2

- Graphics
  - Advanced Plotting Functions
  - Graphics objects and handles
  - Publication-quality graphics
    - MATLAB File Exchange (`http://www.mathworks.com/matlabcentral/fileexchange/`)
  - Animation
    - `VideoWriter`

Syllabus

Lecture 3

- Numerical linear algebra
  - Dense vs. sparse matrices
  - Direct vs. iterative linear system solvers
  - Matrix decompositions
    - LU, Cholesky, QR, EVD, SVD

Lecture 4

- Numerical Optimization
  - Optimization Toolbox
- Nonlinear Systems of Equations

Syllabus

Lecture 5

- Object-oriented programming
  - User-defined classes

Lecture 6

- File manipulation and system interaction
  - Text file manipulation
  - Binary file manipulation
  - System calls
  - Interfacing with spreadsheets (Excel)

Syllabus

Lecture 7

- Compiled MATLAB
  - Interface to low-level programming languages (C/C++/Fortran)
    - MEX Files
  - Standalone C/C++ code from MATLAB code
    - MATLAB Coder

Lecture 8

- Symbolic Math Toolbox
- Parallel Computing Toolbox
- Numerical solution of ODEs and PDEs
  - Partial Differential Equation Toolbox

Introduction

- High-level language for technical computing
  - Integrates computation, visualization, and programming
  - Sophisticated data structures, editing and debugging tools, object-oriented programming
- MATrix LABoratory (MATLAB)
  - Highly optimized for matrix operations
  - Originally written to provide easy access to matrix software: LINPACK (linear system package) and EISPACK (eigen system package)
  - Basic element is array that does not require dimensioning
- Highly interactive, interpreted programming language
  - Development time usually significantly reduced compared to compiled languages
- ***Very*** useful graphical debugger

Outline

Overview

- Numeric Data Types
  - `single`, `double`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `NaN`, `Inf`
- Characters and strings
- Tables
- Structures
- Cell Arrays
- Function Handles
- Map Containers

Overview

- Fortran ordering (column-wise)
- Array creation
  - `blkdiag`, `diag`, `eye`, `true`/`false`, `linspace`/`logspace`, `ones`, `rand`, `zeros`
- Array concatenation
  - `vertcat` ([· ; ·]), `horzcat` ([· , ·])
- Indexing/Slicing
  - Linear indexing
  - Indexing with arrays
  - Logical indexing
  - Colon operator, `end` keyword
- Reshaping/sorting
  - `fliplr`, `flipud`, `repmat`, `reshape`, `squeeze`, `sort`, `sortrows`
- Matrix vs. Elementwise Operations

Fortran Ordering

- MATLAB uses Fortran (column-wise) ordering of data
  - First dimension is fastest varying dimension

Fortran Ordering

- MATLAB uses Fortran (column-wise) ordering of data
  - First dimension is fastest varying dimension

```
>> M = reshape(linspace(11,18,8),[2,2,2])

M(:,:,1) =
      11      13
      12      14

M(:,:,2) =
      15      17
      16      18
```

11
12
13
14
15
16
17
18

Linear Indexing

- Linear storage and Fortran ordering can be used to index into array with *single* index

Linear Indexing

- Linear storage and Fortran ordering can be used to index into array with *single* index

```
>> M(1)
ans =
     11
>> M(8)
ans =
     18
>> M(5:8)
ans =
     15     16     17     18
>> M([1,3,4,8])
ans =
     11     13     14     18
```

Indexing with Arrays

- Arrays can be used to index/slice into arrays
  - Result is an array of the same size as the index array
  - Works with linear indexing or component-wise indexing
  - Component-wise indexing with matrices is equivalent to component-wise indexing with vectorization of matrix

Indexing with Arrays

- Arrays can be used to index/slice into arrays
  - Result is an array of the same size as the index array
  - Works with linear indexing or component-wise indexing
  - Component-wise indexing with matrices is equivalent to component-wise indexing with vectorization of matrix

```
>> M([1,3,4,8]) % Linear indexing (array)
ans =
    11    13    14    18

>> M([1,5,2;8,3,2;7,4,6]) % Linear indexing (matrix)
ans =
    11    15    12
    18    13    12
    17    14    16
```

Indexing with Arrays (continued)

```
>> M([1,2],[2,1],[2,1]) % Component indexing (array)

ans(:,:,1) =
     17     15
     18     16
ans(:,:,2) =
     13     11
     14     12

% Component-wise matrix indexing equivalent to
% component-wise indexing with vectorized matrix
>> isequal(M([2,2;2,1],[2,1],1),...
                       M(vec([2,2;2,1]),[2,1],1))
ans =
      1
```

Logical Indexing

- Index into array based on some *boolean* array
  - Match element in boolean array with those in original array one-to-one
  - If ith entry of boolean array `true`, ith entry of original array extracted
  - Useful in extracting information from an array conditional on the content of the array
- "Linear" and component-wise available
- Much quicker than using `find` and then vector indexing

```
>> P = rand(5000);
>> tic; for i = 1:10, P(P<0.5); end; toc
Elapsed time is 6.071476 seconds.
>> tic; for i = 1:10, P(find(P<0.5)); end; toc
Elapsed time is 9.003642 seconds.
```

Logical Indexing (continued)

- Example

```
>> R = rand(5)
R =
    0.8147    0.0975    0.1576    0.1419    0.6557
    0.9058    0.2785    0.9706    0.4218    0.0357
    0.1270    0.5469    0.9572    0.9157    0.8491
    0.9134    0.9575    0.4854    0.7922    0.9340
    0.6324    0.9649    0.8003    0.9595    0.6787

>> R(R < 0.15)'
ans =
    0.1270    0.0975    0.1419    0.0357

>> isequal(R(R < 0.15),R(find(R<0.15)))
ans =
     1
```

Logical Indexing (Exercise)

```
% logical array assignment
x = linspace(0,2*pi,1000);
y = sin(2*x);

plot(x,y,'k-','linew',2); hold on;
```

- Run the above code in your MATLAB command window (or use `logarray_assign.m`)
- Plot only the values of `y = sin(2*x)` in the interval $[0, \pi/2]$ in 1 additional line of code
  - Use `plot( . , ., 'r--','linew',2);`
- Plot only the values of `sin(2*x)` in the set $\{x \in [0, 2\pi] | -0.5 < \sin(2x) < 0.5\}$ in 1 additional line of code
  - Use `plot( . , ., 'b:','linew',2);`

Reshaping Arrays

Command	Description
`reshape(X,[m n p ..])`	Returns N-D matrix, size $m \times n \times p \times \cdots$
`repmat(X,[m n p ..])`	Tiles X along N dimensional specified number of times
`fliplr(X)`	Flip matrix in left/right direction
`flipud(X)`	Flip matrix in up/down direction
`squeeze(X)`	Remove singleton dimensions

- squeeze_ex.m

Matrix Operations

- MATLAB operations on numeric arrays are *matrix* operations
  - $+, -, *, \backslash, /, \hat{}$, etc
- Prepend . for element-wise operations
  - $.*, ./, .\hat{}$, etc
- Expansion of singleton dimension not automatic
  - `bsxfun(func, A, B)`

Matrix Operations

- MATLAB operations on numeric arrays are *matrix* operations
  - +, −, ∗, \, /, ^, etc
- Prepend . for element-wise operations
  - .∗, ./, .^, etc
- Expansion of singleton dimension not automatic
  - `bsxfun(func, A, B)`

```
>> A = rand(2); b = rand(2,1);
>> A-b
??? Error using ==> minus
Matrix dimensions must agree.
>> bsxfun(@minus,A,b)
ans =
    0.0990   -0.2978
    0.0013    0.1894
```

Create Cell Array and Access Data

- Collection of data of *any* MATLAB type
- Additional flexibility over numeric array
  - Price of generality is storage efficiency
- Constructed with {} or `cell`
- Cell arrays are MATLAB arrays of *cell*
- Indexing
  - Cell *containers* indexed using ()
    - `c(i)` returns ith cell of cell array `c`
  - Cell *contents* indexed using {}
    - `c{i}` returns contents of ith cell of cell array `c`

Create Cell Array and Access Data

- Collection of data of *any* MATLAB type
- Additional flexibility over numeric array
  - Price of generality is storage efficiency
- Constructed with {} or `cell`
- Cell arrays are MATLAB arrays of *cell*
- Indexing
  - Cell *containers* indexed using ()
    - `c(i)` returns ith cell of cell array `c`
  - Cell *contents* indexed using {}
    - `c{i}` returns contents of ith cell of cell array `c`

```
>> c = {14, [1,2;5,10], 'hello world!'};
>> class(c(2))
ans =
cell
>> class(c{2})
double
```

Comma-Separated Lists via Cell Arrays

- Comma-Separated List
  - List of MATLAB objects separated by commas
  - Each item displayed individually when printed
  - Useful in passing arguments to functions and assigning output variables
  - Can be generated using {:} operator in cell array

Comma-Separated Lists via Cell Arrays

- Comma-Separated List
  - List of MATLAB objects separated by commas
  - Each item displayed individually when printed
  - Useful in passing arguments to functions and assigning output variables
  - Can be generated using {:} operator in cell array

```
>> pstr={'bo-','linewidth',2,'markerfacecolor','r'};
>> plot(1:10,pstr{:}) % Pass comma-sep list to func
```

Comma-Separated Lists via Cell Arrays

- Comma-Separated List
  - List of MATLAB objects separated by commas
  - Each item displayed individually when printed
  - Useful in passing arguments to functions and assigning output variables
  - Can be generated using {:} operator in cell array

```
>> pstr={'bo-','linewidth',2,'markerfacecolor','r'};
>> plot(1:10,pstr{:}) % Pass comma-sep list to func
```

```
>> A={[1,2;5,4],[0,3,6;1,2,6]};
>> [A{:}] % Pass comma-sep list to func
ans =
     1     2     0     3     6
     5     4     1     2     6
```

Memory Requirements

- Cell arrays require additional memory to store information describing each cell
  - Information is stored in a *header*
  - Memory required for header of single cell

```
>> c = {[]}; s=whos('c'); s.bytes
ans =
     60
```

  - Memory required for cell array
    - `(head_size x number_of_cells)+ data`

Memory Requirements

- Cell arrays require additional memory to store information describing each cell
  - Information is stored in a *header*
  - Memory required for header of single cell

```
>> c = {[]}; s=whos('c'); s.bytes
ans =
     60
```

  - Memory required for cell array
    - `(head_size x number_of_cells)+ data`
- Contents of a single cell stored contiguously
- Storage not necessarily contiguous between cells in array

Functions

Command	Description
cell2mat	Convert cell array to numeric array
cell2struct	Convert cell array to structure array
cellfun	Apply function to each cell in cell array
cellstr	Create cell array of strings from character array
iscell	Determine whether input is cell array
iscellstr	Determine whether input is cell array of strings
mat2cell	Convert array to cell array
num2cell	Convert array to cell array
struct2cell	Convert structure to cell array

Structures

- Like cell arrays, can hold arbitrary MATLAB data types
- Unlike cell arrays, each entry associated with a *field*
  - Field-Value relationship

Structures

- Like cell arrays, can hold arbitrary MATLAB data types
- Unlike cell arrays, each entry associated with a *field*
  - Field-Value relationship
- Structures can be arranged in N-D arrays: *structure arrays*
- Create structure arrays
  - `struct`
  - `<var-name>.<field-name> = <field-value>`
- Access data from structure array
  - () to access structure from array, . to access field

Structures

- Like cell arrays, can hold arbitrary MATLAB data types
- Unlike cell arrays, each entry associated with a *field*
  - Field-Value relationship
- Structures can be arranged in N-D arrays: *structure arrays*
- Create structure arrays
  - `struct`
  - `<var-name>.<field-name> = <field-value>`
- Access data from structure array
  - () to access structure from array, . to access field

```
>> classes=struct('name',{'CME192','CME292'},...
                  'units',{1,1},'grade',{'P','P'});
>> classes(2)
    name: 'CME292'
   units: 1
   grade: 'P'
```

Memory Requirements

- Structure of arrays faster and more memory efficient than array of structures
  - Contiguous memory
  - Memory overhead

Memory Requirements

- Structs require additional memory to store information
  - Information is stored in a *header*
  - Header for entire structure array
- Each field of a structure requires contiguous memory
- Storage not necessarily contiguous between fields in structure or structures in array
- Structure of arrays faster/cheaper than array of structures
  - Contiguous memory, Memory overhead

Functions

Command	Description
`fieldnames`	Field names of structure
`getfield`	Field of structure array
`isfield`	Determine whether input is structure field
`isstruct`	Determine whether input is structure array
`orderfields`	Order fields of structure array
`rmfield`	Remove fields from structure
`setfield`	Assign values to structure array field
`arrayfun`	Apply function to each element of array
`structfun`	Apply function to each field of scalar structure

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

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'.
```

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

```
function [b,varargout] = vararg_ex(a,varargin)

b = a^2;
class(varargin)
varargout = cell(length(varargin)-a,1);
[varargout{:}] = varargin{1:end-a};

end
```

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

Outline

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**: `nltruss.m`
- Other performance assessment functions
  - `tic`, `toc`, `timeit`, `bench`, `cputime`
  - `memory`

Outline

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
- 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

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

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

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

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