Matlab variable types: primitives

The three main object classes in Matlab are:

- **double**
  - multidimensional array of double precision floating point numbers
- **char**
  - multidimensional array of ascii characters
- **logical**
  - Multidimensional array of 1-bit (0/1) numbers

Eight other built-in primitive variable types are:

- **uint8, uint16, uint32, uint64**
  - multidimensional array of 8, 16, 32 or 64-bit, unsigned integers
- **int8, int16, int32, int64**
  - multidimensional array of 8, 16, 32 or 64-bit, signed integers

Two things to remember:
- Every variable in Matlab is an array
- Arrays are at least 2-dimensional (no notion of 1-d array)

Matlab variable types: derived primitives

Two other important object classes in Matlab are:

- **cell**
  - multidimensional array of "containers"
- **struct**
  - multidimensional array of a structure with fields (common across array)

In a cell, the contents of a container may be:
- **double, char, intXX, uintYY**
- Another cell array
- A struct array
- An object of other classes (listed on next slide)

In a struct, the value of a field may be:
- Same list as above

Managed arrays of pointers to Matlab variables

Matlab variable types: others

One additional class within Matlab is:

- **Function_handle**
  - "address" to an existing function (builtin or user-defined)

There are also:

- User-defined, or custom classes
  - We will learn in weeks 3-5 how to write custom classes. Each session of Matlab "learns" about them when an object is created (by the constructor function)
- Java classes
  - Matlab allows creation and manipulation of several Java classes. We will learn about these at some point in the class.

Scalars, Vectors and Arrays

1-by-1 arrays are also called scalars

1-by-N arrays are also called row vectors

N-by-1 arrays are also called column vectors

Row and Column vectors are also just called vectors

Creating special arrays

- **ones(n,m,p)**
  - a n-by-m-by-p double array, each entry is equal to 1
- **zeros(n,m,p,q)**
  - a n-by-m-by-p-by-q double array, each entry is equal to 0
- **rand(n,m)**
  - a n-by-m double array, each entry is a random number uniformly distributed between 0 and 1.
- **randn(n,m)**
  - a n-by-m double array, each entry is a random number with Gaussian ("normal") distribution, mean of 0, and variance of 1.
- **cell(n,m)**
  - a n-by-m cell array, the contents of each container is a 0-by-0 (ie., empty) double.
Number of Dimensions (\texttt{ndims})

Suppose \(A\) is 7-by-4-by-5 array. Matlab also interprets \(A\) as

\[
7\text{-by-}4\text{-by-}5\text{-by-}1\text{-by-}1\text{-by-}1\text{-by-}1\text{-by-}1\text{-by-}1\text{-by-}1\text{-by-}1
\]

\textit{Infinite number of "trailing singleton dimensions"}

\texttt{ndims(A)} is the number of dimensions of \(A\), ignoring trailing singleton dimensions.

Exception: remember, everything in Matlab is at least 2-dimensional, so 1\textsuperscript{st} and 2\textsuperscript{nd} dimensions are "special"

– Neither are ever treated as a "trailing singleton dimension"

– \texttt{ndims(X)} is always at least 2

Multidimensional Arrays

Remark: We will carefully cover the syntax for multidimensional arrays. Some mention is made in these review slides, but we will focus on multidimensional arrays in the next lecture.

The \texttt{SIZE} command

If \(A\) is an array, then \texttt{size(A)} is a \(1\text{-by-}\text{ndims(A)}\) array.

– The (1,1) entry is the number of rows of \(A\)

– The (1,2) entry is the number of columns of \(A\)

– The (1,k) entry is the length of the \(k\)th dimension of \(A\)

If \(A\) is an array, then

\[
\begin{align*}
\text{size}(A,1) & \text{ is the number of rows of } A \\
\text{size}(A,2) & \text{ is the number of columns of } A \\
\text{size}(A,k) & \text{ is the length of the } k\text{th dimension of } A
\end{align*}
\]

Examples

\[
\begin{align*}
\gg A & = \text{rand(5,6,8)}; \\
\gg B & = \text{size}(A) \\
\gg [a1,a2,a3,a4] & = \text{size}(A) \\
\gg [a1,a2] & = \text{size}(A) \\
\gg \text{size}(A,2) \\
\gg \text{size}(A,6)
\end{align*}
\]

The \texttt{CLASS} command

If \(A\) is an array, then

\texttt{class(A)}

is character string, whose value is the class of \(A\) (eg., \texttt{double}, \texttt{char}, \texttt{cell}, \texttt{struct}, \texttt{uint32}, etc).

The command \texttt{class} will also be used when we create our own user-defined objects.

Examples

\[
\begin{align*}
\text{class(rand(5,6,8))} \\
\text{class(class(rand(5,6,8)))} \\
\text{class(cell(4,5))}
\end{align*}
\]

The "; (colon) convention" is used to create row vectors, whose entries are evenly spaced.

\[
7:2:18 \text{ equals the row vector } [7\ 9\ 11\ 13\ 15\ 17]
\]

If \(F\), \(J\) and \(L\) are numbers with \(J>0\), \(F \leq L\), then \(F:J:L\) creates a row vector

\[
[F\ F+J\ F+2J\ F+3J\ \ldots\ F+NJ]
\]

where \(F+NJ \leq L\), and \(F+(N+1)J>L\)

Many times, the increment is 1. The shorthand for \(F:1:L\) is \(F:L\)

\[
F:J:L \text{ with } J<0 \text{ (decrementing)}
\]

The colon-convention can also decrement (ie., \(J<0\)), in which case \(L\) should be less than \(F\).

Therefore 6.5:-1.5:0 is the row vector

\[
[6.5\ 5.0\ 3.5\ 2.0\ 0.5]
\]
Matlab Basics

Matlab as an arithmetic calculator
- Type expressions at the >>, and press return
- Result is computed, and displayed as a variable named \texttt{ans}
- Use numbers, +, -, ^, *, /, (), sin, cos, exp, abs, round, ...

Precedence rules in expressions
- Left-to-right within a precedence group
- Precedence groups are (highest first)
  - Power (\(^\))
  - Multiplication and division (*, /)
  - Addition and subtraction (+, -)
- What is \(6+3*4^2-1\)

Examples of expressions

Legal expressions (all scalar)
- \(4\)
- \(5 + \pi\)
- \(6*\sqrt{2}^4 - 12\)
- \(\sin(\pi/3)^2 + \cos(\pi/3)^2\)
- \(1.0/0.0\)
- \(-4/\infty\)
- \(0/0\)

Illegal expressions
- \(2 4\)
- \((2,4)\)

Variables and assignment

Use names to assign result of an expression to a variable
- Variables do not need to be declared before assignment

A single "equal" sign (=) is the assignment operator,
\texttt{LHS = RHS Expression}

Read this as
- evaluate expression on the right-hand-side, and then...
- assign the result to the variable named on the left-hand-side

Therefore
- The right-hand-side needs to be a legal Matlab expression
- The left-hand-side needs to be
  - a single variable name, or
  - a valid reference into an array, examples forthcoming

A semicolon at the end of the RHS expression suppresses the display, but the assignment still takes place.

Examples of Variables, Assignment and builtin arithmetic

Legal
- \(>> \texttt{A = sqrt(13)}\)
- \(>> \texttt{B = exp(2)};\)
- \(>> \texttt{A = 2*B}\)
- \(>> \texttt{A = A + 1}\)
- \(>> \texttt{C = tan(pi/4)}\)

Illegal (all for different reasons)
- \(>> \texttt{D = sqrt(E) + 1;}\)
- \(>> 3 = E\)
- \(>> 3*A = 14\)
- \(>> F = 2 3\)

Creating Arrays

Arrays can be created simply by assignment
- \(p = 4; \ % 1\text{-by-}1\ \text{double array}\)
- \(\texttt{name} = \texttt{'UC Berkeley'}; \ % 1\text{-by-}11\ \text{char array}\)
- \(\texttt{Example1} = []; \ % 0\text{-by-}0\ \text{double array, "empty"}\)
- \(\texttt{Example2} = ''; \ % 0\text{-by-}0\ \text{char array, "empty"}\)

Similar for \texttt{struct} and \texttt{cell}
- \(\texttt{M.field1} = \texttt{'Oakland'}; \ % 1\text{-by-}1\ \text{struct, 1 field}\)
- \(\texttt{M.field2} = 102.34; \ % \text{still 1\text{-by-}1, 2 fields}\)
- \(\texttt{Acell} = \{123.7\} \ % \{\} \text{creates "container", this is}\)
  \(\% \text{a 1\text{-by-}1 array of containers}\)

Concatenating Arrays

Horizontal and Vertical Concatenation (ie., "stacking")
- Square brackets, [, and ] to define arrays
- Spaces (and/or commas) to separate columns
- Semi-colons to separate rows

If \(A\) and \(B\) are arrays with the same number of rows, then
- \([A \ B]\) is the array formed by stacking \(A\) "next to" \(B\)

If \(A\) and \(B\) are arrays with the same number of columns, then
- \([A ; B]\) is the array formed by stacking \(A\) "on top of" \(B\)

Once constructed, an array does not "know" it came from concatenation of two arrays. No partitioning information is maintained. Consequently...

Extra (but consistent) brackets have no consequence, so
- \([A B [C D]\), \([A [B C D]\), \([[[A B] C D]]\) are the same
Concatenating Arrays

Examples

\[
\begin{bmatrix}
3 & 4 & 5 \\
6 & 7 & 8
\end{bmatrix}
\] is the 2-by-3 array

\[
\begin{bmatrix}
3 & 6 \\
4 & 7 & 8
\end{bmatrix}
\] is the same thing

\[M\text{.fld1} = 2;\]
\[M\text{.fld2} = \text{'Andy'};\]
\[Q\text{.fld1} = 14;\]
\[Q\text{.fld2} = \text{'Packard'};\]
\[\{M;Q\} \% a 2-by-1 struct array, w/ fields fld1, fld2\]
\[\{M Q\} \% a 1-by-2 struct array, w/ fields fld1, fld2\]
\[\{M Q\{M,M\}\} \% 3-by-2 struct array\]

Acell = {123.8}; \% 1-by-1 cell array
Bcell = {'E177'}; \% 1-by-1 cell
C = [Acell Bcell]; \% 1-by-2 cell array
D = [C ; [Bcell Acell]]; \% 2-by-2 cell array

Concatenating Multidimensional Arrays

The command \texttt{cat} generalizes horizontal and vertical concatenation.

Example

\[A = \text{rand}(6,5) ; \% 6-by-5\]
\[B = \text{rand}(6,5); \% 6-by-5\]
\[C = \text{cat}(3,A,B) \% 6-by-5-by-2\]
\[D = \text{cat}(4,A,B) \% 6-by-5-by-1-by-2\]
\[E = \text{cat}(4,C,C) \% 6-by-5-by-2-by-2\]

Concatenate along 3rd dimension

Accessing single elements of a vector

If \(A\) is a vector (ie, a row or column vector), then
\[A(1)\] is its first element,
\[A(2)\] is its second element,...

Example

\[A = [ 3 \ 4.2 \ -7 \ 10.1 \ 0.4 \ -3.5 ];\]
\[A(3)\]
\[\text{Index} = 5;\]
\[\text{A(Index)}\]

This syntax can be used to assign an entry of \(A\). Recall assignment

\texttt{VariableName = Expression}

An entry of an array may also be assigned

\texttt{VariableName(Index) = Expression}

So, change the 4’th entry of \(A\) to the natural logarithm of 3.
\[A(4) = \log(3);\]

Accessing multiple elements of a vector

\[A(3)\] refers to the 3’rd entry of \(A\). However, the index need not be a single number.

Example: Make a 1-by-6 row vector, and access multiple elements, giving back row vectors of various dimensions.
\[A = [ 3 \ 4.2 \ -7 \ 10.1 \ 0.4 \ -3.5 ];\]
\[\text{A([1 4 6])} \% 1-by-3, 1st, 4th, 6th entry\]
\[\text{Index} = [3 2 3 5];\]
\[\text{A(Index)} \% 1-by-4\]

\texttt{Index} should contain integers. Regardless of whether \(A\) is a row or column, \texttt{Index} can be a row or a column, and Matlab will do the same thing in both cases. The expressions below are the same.

\[\text{A([2 4 3])}\]
\[\text{A([2;4;3])}\]

Assigning multiple elements of a vector

In an assignment to multiple entries of a vector

\texttt{A(Index) = Expression}

the right-hand side expression should be a scalar, or the same size as the array being referenced by \texttt{A(Index)}

Example: Make a 1-by-6 row vector, and access multiple elements, giving back row vectors of various dimensions.
\[A = [ 3 \ 4.2 \ -7 \ 10.1 \ 0.4 \ -3.5 ];\]
\[\text{A([1 4 6]) = [10 100 1000];}\]
\[\text{Index} = [3 2 3 5];\]
\[\text{A(Index) = pi;}\]

Accessing elements and parts of arrays

If \(M\) is an array, then \texttt{M(3,4)} is

\[\text{the element in the (3" row, 4" column) of } M\]

If \(M\) is an array, then \texttt{M([1 4 2],[5 6])}

\[\text{is a 3-by-2 array, consisting of the entries of } M \text{ from}\]
\[\text{rows [1, 4 and 2] and columns [5 and 6]}\]

In an assignment

\texttt{M(RIndex,CIndex) = Expression}

the right-hand side expression should be

\[\text{the same class as } M \text{ and}\]
\[\text{a scalar, or the same size as the array being referenced by } M(RIndex,CIndex)\]
Accessing arrays
Although the previous slides used DOUBLEs in the examples, everything written held for all classes of primitive arrays
- double
- char
- uintXX, intYY
- cell
- struct

The entries of a numeric array in Matlab are stored together in memory in a specific order.

\[
\begin{bmatrix}
3.4 & 4.2 & 8 \\
-6.7 & 12 & 0.75
\end{bmatrix}
\]
represents the array

Somewhere in memory, Matlab has

For example, the 5th entry of A is 8 and can be referenced that way directly. Check using `isequal`

`isequal(A(5), 8)`

RESHAPE
RESHAPE changes the size, but not the values or order of the data in memory:

\[
\begin{bmatrix}
3.4 & 4.2 & 8 \\
-6.7 & 12 & 0.75
\end{bmatrix}
\]

The result (in memory is)

\[
\begin{bmatrix}
3.4 & 4.2 & 8 \\
-6.7 & 12 & 0.75
\end{bmatrix}
\]

while B is the array

\[
\begin{bmatrix}
3.4 & 12 \\
-6.7 & 8 \\
4.2 & 0.75
\end{bmatrix}
\]

Workspaces
"Workspace" is the name for the collection of variables that are visible. The commands `who` and `whos` give information about the variables in the current workspace.

\[
\begin{align*}
\text{who} \\
\text{whos} \\
S = \text{whos}; \text{ % info in struct array}
\end{align*}
\]

Initially, 3 important workspaces
- The workspace visible from the Matlab prompt `>>` is the "Base" workspace.
- Each call to a function creates a "function instance" workspace.
  - When the function exits, the workspace disappears
  - Since functions can be called recursively, there can exist many workspace instances of the same function
- There is a "global" workspace

END
Suppose \( A \) is an \( N \times M \) array, and a reference of the form

\[
A(RIndex, CIndex)
\]

Any occurrence of the word `end` in the \( RIndex \) is changed (automatically) to \( N \)

Any occurrence of the word `end` in the \( CIndex \) is changed (automatically) to \( M \)

Example:
\[
M = \text{rand}(4,5);
M(\text{end}, \text{end})
M([\text{1 end}], [\text{end 2end}])
\]

: as a row or column index
Suppose \( A \) is an \( N \times M \) array, and a reference of the form

\[
A(RIndex, CIndex)
\]

If \( RIndex \) is a single colon, `:`; then \( RIndex \) is changed (automatically) to \( 1:N \) (every row)

If \( CIndex \) is a single colon, `:`; then \( CIndex \) is changed (automatically) to \( 1:M \) (every column)

Example:
\[
M = \text{rand}(4,5);
M(:, [1 3 5])
\]
Indexing with LOGICAL arrays

In a typical row/column reference,

\[ M(RowIndex,ColIndex) \]

both \( RowIndex \) and \( ColIndex \) are double arrays, whose positive, integer values specify which rows and columns of the array \( M \) are being referenced.

If \( RowIndex \) and \( ColIndex \) are logical arrays, the locations of the 1’s specify which rows and columns of the array \( M \) are being referenced.

\[ M = \text{rand}(4,5); \]
\[ \text{Ridx} = \text{logical}([1 0 0 1]); \]
\[ \text{Cidx} = \text{logical}([0 0 1 1 1]); \]
\[ M(\text{Ridx},\text{Cidx}) \text{ same as } M([2 4],[3 4 5]) \]

Unary Numeric Operations on double Arrays

Unary operations involve one input argument. Examples are:

- Negation, using the “minus” sign
- Trig functions: \( \sin, \cos, \tan, \text{asin}, \text{acos}, \text{atan}, \ldots \)
- General rounding functions: floor, ceil, fix, round
- Exponential and logs: \( \exp, \log, \log10, \text{sqrt} \)
- Complex Arithmetic: abs, angle, real, imag

Example: If \( A \) is an \( N \times N \times N \times \ldots \) array, then

\[ B = \sin(A); \]

is an \( N \times N \times N \times \ldots \) array. Every entry of \( B \) is the sin of the corresponding entry of \( A \).

The “for”-loop that cycles the calculation over all array entries is an example of the vectorized nature of many built-in functions.

Arithmetic Binary operations on double Arrays

Addition (and subtraction)
- If \( A \) and \( B \) are arrays of the same size, then \( A+B \) is an array of the same size whose individual entries are the sum of the corresponding entries of \( A \) and \( B \)
- If \( A \) is an array and \( B \) is a scalar, then \( A+B \) is an array of the same size as \( A \), whose individual entries are the sum of the corresponding entries of \( A \) and the scalar \( B \)
- If \( A \) is a scalar, and \( B \) is an array, use same logic as above

Scalar-Array Multiplication
- If \( A \) is an array, and \( B \) is a scalar, then \( A*\text{scalar} \) is an array of the same size as \( A \), whose individual entries are the product of the corresponding entries of \( A \) and the scalar \( B \)

Element-by-Element Multiplication
- If \( A \) and \( B \) are arrays of the same size, then \( A*B \) is an array of the same size whose individual entries are the product of the corresponding entries of \( A \) and \( B \)

Matrix multiplication
- If \( A \) and \( B \) are arrays, then \( A*B \) is the matrix multiplication of the two arrays.

Character Arrays (Chapter 8, pg 129-149)

So far, we have seen numeric arrays, called double arrays.

Another basic data type in Matlab is the character array, or sometimes just called char or referred to as strings.

Use the single quote ‘ to create basic character arrays.

Example

\[ A = \text{‘andy is the teacher’}; \]
\[ \text{size}(A) \]
\[ \text{class}(A) \]
\[ \text{whos} \]

Numeric Values of Character Arrays

A character array is really just a numeric array (integer values, in fact), along with some flag which indicates it should be interpreted as characters.

You can convert a character array into its numeric array using the command DOUBLE.

You can convert a double array with integer values into its equivalent character array using the CHAR command.

\[ \text{>> double(‘Class is half over’)} \]
\[ \text{>> v=98 101 32 112 97 116 105 101 110 116} \]
\[ \text{>> char(v)} \]
Referencing a CHAR array
Use integer indices to access specific elements of a char array.

Example
\[
A = 'andy is the teacher';
A(1:4)
A(end+1:1)
A([5 8 12])
\]

Concatenating CHAR arrays
Use square brackets to stack (horizontally and vertically) character arrays, just as you do numeric arrays.

Example
\[
str1 = 'E77 was fun, ';
str2 = 'but E177 is better!';
str3 = [str1 str2]
size(str1)
size(str2)
size(str3)
str4 = [' ' str1 ; str2];
disp(str4)
\]

Review of Cell Arrays
Model: “array of containers; contents of which are variables”
Use curly brackets { and } to wrap a variable in a container.

Example: Create a 1-by-1 container, whose contents is the 1-by-12 character array ‘Andy Packard’
\[
>> name = { 'Andy Packard' };
>> size(name)
>> class(name)
\]
Create two more 1-by-1 containers with different contents
\[
>> SID = { 12345678 };
>> scores = { [82 71 64 88 99] };
\]

Concatenating
Stack all three containers next to one another, creating a 1-by-3 array of containers.
\[
>> ClassInfo = [ name SID scores ];
\]
ClassInfo is a 1-by-3 array of containers. It is called a cell array.
\[
>> size(ClassInfo)
>> class(ClassInfo)
>> whos
\]
Contents of Container 1

<table>
<thead>
<tr>
<th>Andy Packard</th>
<th>12345678</th>
<th>[82 71 64 88 99]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

class and size for cell arrays
ClassInfo is a 1-by-3 array of containers—a cell array.
\[
size(ClassInfo)
class(ClassInfo)
whos
\]
Using parenthesis () to access parts of the array keeps the containers (recall model: array of containers, with contents).
\[
ClassInfo(1)
tmp = 'Andy Packard'
isequal(ClassInfo(1),'Andy Packard')
size(ClassInfo(1))
class(ClassInfo(1))
\]
Referencing contents of cells
Use curly brackets to access the contents of a container.

```matlab
>> ClassInfo{1}
>> ClassInfo{2}
>> ClassInfo{3}
>> isequal(ClassInfo{2}, 12345678)
>> isequal(ClassInfo{1}, tmp)
```

3 by 2 Cell Array example
Create a 3-by-2 array of containers

```matlab
>> A = [{[1 2; 3 4]} {'andy'} ;
{[pi] {rand(20,2)};} 
{[['E77';'fun']] {zeros(1,6)} }];
```

Quiz: References into a 3 by 2 Cell Array
```matlab
>> size(A)  
>> class(A)
>> B = A(1,1);
>> size(B), class(B)
>> C = A{1,1}
>> size(C), class(C)
>> A(1,1){2,2}
>> D = A(:,2)
>> size(D), class(D)
>> D{3}{3}
>> ['s' D{1} ' beach']
```

CELL creates Cell Arrays with empty contents
Create a 3-by-4 array of containers. By default, the contents of each container will be an empty array (0-by-0) double array.

```matlab
>> E = cell(3,4)
Put the array [7 8 9] in the contents of the 1,1 container
>> E{1,1} = [ 7 8 9 ];
Try to insert the array [ 20 25 ] as the 1,2 container
>> E{1,2} = [ 20 25 ];
Insert a container (whose contents is the array [ 20 25 ]) in the 1,2 element of E (recall, all elements of E are containers)
>> E{1,2} = { [ 20 25 ] };
What is different in
>> E{1,3} = { [ 20 25 ] };
```

Commands to look at on your own
- `num2cell`
- `cell2mat`
- `mat2cell`
- `cellstr`
- `celldisp`
- `iscell`
Review of Struct Arrays

Structures (Chapter 7, pg 116

Data types encountered so far
– numeric arrays, called double arrays
– character arrays, called char arrays
– container arrays, called cell arrays

Like cell arrays, structure arrays allow you to group different data types together.

Example: create a 1-by-1 structure with 3 fields

```matlab
student.Name = 'Fred Smith';
student.SID = 12345678;
student.Scores = [62 78 93 61];
```

The variable `student` is created. It is a structure containing 3 fields.

```matlab
class, size, fieldnames, isfield
Recall example

```matlab
student.Name = 'Fred Smith';
student.SID = 12345678;
student.Scores = [62 78 93 61];
```

`Student` is a 1-by-1 `struct`, with 3 fields, whose names are `Name` and `SID` and `Scores`

```matlab
size(student)
class(student)
fieldnames(student)
isfield(student,'Name')
isfield(student,'Names')
```

Accessing fields

Access the contents of the fields simply by typing

```matlab
VariableName.FieldName
```

So, in this case, we can do

```matlab
student.Name
student.SID
student.Scores
```

`student.Scores` is 1-by-4 double array. Everything works. For example, access its 1st element.

```matlab
student.Scores(1)
```

Or, access its last element.

```matlab
student.Scores(end)
```

Accessing fields (continued)

Can also assign as you would expect. Change the 3rd score to 99.

```matlab
student.Scores(3) = 99;
```

Structure Arrays

How about using a `struct` to keep track of E177 students?

There are 30+ of you.

We need a `struct array`
– with 30 elements
– each with fields `Name`, `SID` and `Scores`.

Every object in Matlab can be an array (not just scalar).

Can construct this by just declaring the the 2nd entry, and then the 3rd entry, and so on...
Growing the struct
Add the 2nd student. Just assign, using the index...
student(2).Name = 'William Hung';
student(2).SID = 10308090;
student(2).Scores = [20 40 90 80];
Analogy: This works for all Matlab arrays. You can “grow”
them simply by assigning beyond their current size.
A = 1.3;
A(2) % error, A only has 1 element
A(2) = 4.7 % A is now 1 by 2
A(5) = 26 %A is now 1 by 5
Remark: this is not good practice. Preallocate (with the zeros
or struct command, for instance) to the correct dimension.

Growing the struct (continued)
Add the 3rd student. Just assign, using the index...
student(3).Name = 'N. Coughlin';
student(3).SID = 22334455;
student(3).Scores = [90 95 90 92];
Check that student is a struct array
size(student)
whos

Accessing a field of Struct Array
Recall that student is 1-by-3 struct array. One of its fields is
named ‘Scores’.
In each case below, the expression is a 1-by-4 double array
student(1).Scores
student(2).Scores
student(3).Scores
What is student.Scores
by itself?

Comma separated Lists
By itself, student.Scores is literally expanded as
\[
\begin{bmatrix}
\text{student(1).Scores} \\
\text{student(2).Scores} \\
\text{student(3).Scores}
\end{bmatrix}
\]
Explicitly, based on the data, it is literally equal to
\[
\begin{bmatrix}
62 & 78 & 99 & 61 \\
20 & 40 & 90 & 80 \\
90 & 95 & 90 & 92
\end{bmatrix}
\]
So, a field reference of a struct array represents a comma-
separated list of the elements.
A comma-separated list is not an object class (like double, char,
cell or struct), but it can be used effectively in expressions...

Student.Scores vs [Student.Scores]
Student.Scores actually represents
\[
\begin{bmatrix}
62 & 78 & 99 & 61 \\
20 & 40 & 90 & 80 \\
90 & 95 & 90 & 92
\end{bmatrix}
\]
which is 3 arrays (all happen to be 1-by-4) separated by
commas
Recall that
\[
\begin{bmatrix}
62 & 78 & 99 & 61 \\
20 & 40 & 90 & 80 \\
90 & 95 & 90 & 92
\end{bmatrix}
\]
is a 1-by-12 array (note the extra square brackets).
Therefore, [student.Scores] is simply the 1-by-12 array
\[
\begin{bmatrix}
62 & 78 & 99 & 61 & 20 & 40 & 90 & 80 & 90 & 95 & 90 & 92
\end{bmatrix}
\]

Application of RESHAPE
[Student.Scores] is a 1-by-12 array of test scores,
\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 2 & 1 & 1 & 2 & 1 & 1 & 3 & 1
\end{bmatrix}
\]
– The first 4 elements are the test scores of student 1
– The next 4 elements are the test scores of student 2
– The next 4 elements are the test scores of student 3
It would be nice to have a 3-by-4 or 4-by-3 array of the scores.
The 3-by-4 case is “hard”, since the order in memory
is different.
The 4-by-3 case is “easier”, since the order in memory is
the same.
Tests
\[
\begin{bmatrix}
1 & 1 & 1 & 2 & 1 & 1 & 2 & 1 & 1 & 3 & 1
\end{bmatrix}
\]
Student
\[
\begin{bmatrix}
1 & 2 & 1 & 1 & 2 & 1 & 1 & 2 & 1 & 1 & 3 & 1
\end{bmatrix}
\]
RESHAPE will work

Try

\[ \text{reshape([Student.Scores],[4 3])} \]

To get the 3-by-4 case, you simply need to transpose the 4-by-3 case, so it’s not much harder (but Matlab shuffles everything in memory)

\[ \text{reshape([Student.Scores],[4 3])}' \]

Multiple {} Reference of Cell Array

If \( M \) is a cell array (say 6-by-7), then \( M([2 4],[7 5]) \) is a 2-by-2 cell array, drawn from \( M \).

However, \( M([2 4],[7 5]) \) asks for the contents of 4 cells. The interpretation of this is again a comma-separated list.

Therefore, \( M([2 4],[7 5]) \) is automatically expanded as

\[ M(2,7), M(4,7), M(2,5), M(4,5) \]

and hence can be used effectively in expressions.

Working with multidimensional arrays

Remark: We will carefully cover the syntax for multidimensional arrays in next Tuesday’s lecture, focusing on

- General, order in memory
- Referencing and assignment
- Concatenation
- Reduced indexing
- Colon convention, and

“Delayed Copy”

For today, focus on copying a primitive (double, char)

\[
\begin{array}{c|c|c}
\text{A} & \text{Memory} & \text{Contents} \\
\hline
[1 2 3;4 5 6] & \text{Address} & \text{Contents} \\
\hline
0x2F44 & 7.99451 & 7.99451 \\
0x2F45 & 1.00000 & 1.00000 \\
0x2F46 & 4.00000 & 4.00000 \\
0x2F47 & 2.00000 & 2.00000 \\
0x2F48 & 5.00000 & 5.00000 \\
0x2F49 & 3.00000 & 3.00000 \\
0x2F4A & 6.00000 & 6.00000 \\
0x2F4B & \text{---} & \text{---} \\
0x2F4C & \text{---} & \text{---} \\
0x2F4D & \text{---} & \text{---} \\
0x2F4E & 4.00000 & 4.00000 \\
0x2F4F & 2.00000 & 2.00000 \\
0x2F50 & -7.4000 & -7.4000 \\
0x2F51 & 3.00000 & 3.00000 \\
0x2F52 & 6.00000 & 6.00000 \\
\end{array}
\]

Relational Operators

Relational operators are used to compare double variables.

There are 6 comparisons

- “equal to”, using ==
- “not equal to”, using ~=
- “less than”, using <
- “less than or equal to”, using <=
- “greater than”, using >
- “greater than or equal to”, using >=

The result of a comparison is of class logical, either TRUE (1) or FALSE (0)

Array comparisons

Suppose \( A \) and \( B \) are double arrays of the same size. Let \( \text{op} \) be any of the 6 relational operators (==, ~+, <, <=, >, >=)

Then the expression

\[ A \text{ op } B \]

is a logical array of the same size. The relational operator is applied elementwise, comparing \( A(i,j) \) to \( B(i,j) \). Hence, the relational operators are vectorized.

Example

\[ A = \text{rand}(2,4); \]

\[ B = 0.5 \times \text{ones}(2,4); \]

\[ A \text{<} B \]
logical arrays
The result of a relational operation is a logical array
– A logical array contains only 0’s and 1’s.
– It cannot contain any other numerical values.
– Internal representation in MATLAB is different than for double arrays.

You can use a logical array in any numerical calculation as though it is a double array—the 0’s and 1’s behave normally.

\[
A = \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}; \\
B = \text{logical}(A); \\
\text{whos} \\
A==B \\
\text{isequal}(A,B)
\]

Scalar/Array comparisons
Suppose \(A\) is a scalar, and \(B\) is a double array. Let \(\text{op}\) be any of the 6 relational operators (==, ~=, <, <=, >, >=)

Then the expression
\[
A \text{ op } B
\]
is an array of the same size as \(B\). The relational operator is applied comparing the scalar \(A\) to each element of \(B\).

Example
\[
A = 2.5; \\
B = \begin{bmatrix} 0 & 3 & 4; & -1 & 21 & 6; 2.5 & 2.4 \end{bmatrix}; \\
C = A<=B;
\]

Array/Scalar comparisons
Suppose \(A\) is a double array, and \(B\) is a scalar. Let \(\text{op}\) be any of the 6 relational operators (==, ~=, <, <=, >, >=)

Then the expression
\[
A \text{ op } B
\]
is an array of the same size as \(A\). The relational operator is applied comparing each element of \(A\) to the scalar \(B\).

Example
\[
A = \sin(\text{linspace}(0,\pi,20)); \\
B = 0.5 \\
C = A>B;
\]

find
The command \texttt{find} returns the indices of the nonzero entries.
\[
m = \text{rand}(6,1); \\
m(\text{find}(m<0.5)) = 0;
\]

But logical indexing also work, so you can just do
\[
m = \text{rand}(6,1); \\
m(m<0.5) = 0; \\
\text{For arrays, find returns the indices in a single-index form, using the well-defined ordering for the elements in an array.}
\]
\[
m = \text{rand}(4,5); \\
\text{idx} = m<0.5; \\
m(\text{idx}) = -m(\text{idx});
\]

Care in using \(==\) on numeric data
In finite precision arithmetic (MATLAB has about 17 digits of precision), it is not true that\n\[
(a+b)+c \text{ is equal to } a+(b+c)
\]

What happens
– In computing \(a+b\), some roundoff error may occur, and then in computing the additional sum with \(c\), additional roundoff occurs.
– In computing \(b+c\), some different roundoff error may occur, and then in computing the additional sum with \(a\), additional roundoff occurs.

Imagine 2-digit, floating point, arithmetic
\[
\begin{array}{c}
1.2 + .74 + .24 \\
1.9 + .24 \\
2.1
\end{array} = \begin{array}{c}
1.2 + .74 + .24 \\
1.2 + .98 \\
2.2
\end{array}
\]

Logical Operators
Logical operators implement "and" and "or" logic

There are 3 binary operations
– "logical AND", using \&
– "logical OR", using |
– "logical exclusive OR", using \texttt{xor}

along with unary negation
– "logical NOT", using ~

Lastly, "short circuit" operators
– Short-circuit AND, using \&\&
– Short-circuit OR, using ||
Logical Operators

If \( A \) and \( B \) are scalars (double or logical), then

\[ A \land B \] is TRUE if \( A \) and \( B \) are both nonzero, otherwise it is FALSE.
\[ A \lor B \] is TRUE if either \( A \) or \( B \) are nonzero, otherwise it is FALSE.
\[ \text{xor}(A, B) \] is TRUE if one argument is 0 and the other is nonzero, otherwise it is FALSE.
\[ \neg A \] is TRUE if \( A \) is 0, and FALSE if \( A \) is nonzero.

For arrays, the operations are applied elementwise, so \( A \) and \( B \) must be the same size, or one must be a scalar.

Short-Circuit Logical Operators

If \( A \) and \( B \) are scalars (double or logical), then

\[ A \land B \] is the same as \( A \lor B \), except the expression \( B \) is not evaluated if \( A \) is FALSE.
\[ A \lor B \] is the same as \( A \land B \), except the expression \( B \) is not evaluated if \( A \) is TRUE.

Example: Suppose a condition requires expression \( A \) and \( B \) to be TRUE, but if \( A \) is FALSE, expression \( B \) may have a run-time error. Then without short-circuiting, you would need if-else flow control to reliably check \( A \land B \).

Control flow

Five forms of control flow are used in Matlab:

- for loops
  - Execute a collection of statements a prescribed number of times.
  - for \( x = \text{expression} \)
    - statements
  - end
- while loops
  - Execute a collection of statements repeatedly as long as a test condition remains true.
- if/elseif/else conditionals
- switch/case/otherwise
- try/catch

for, end (page 163–74)

Execute collection of statements a fixed number of times.

for \( x = \text{expression} \)
  - statements
end

The \( \text{expression} \) is evaluated once before the loop starts. The value is called the controlvalue. The statements are executed one time for each column in the controlvalue. In the code above, \( x \) is the loopvariable.

Before each "execution pass," the loopvariable \( x \) is assigned to the corresponding column of controlvalue:
- 1st column on first pass,
- 2nd column on second pass, and so on.

for loops, most common use(!)

The most common value for \( \text{expression} \) is a row vector of integers, starting at 1 and increasing to a limit \( n \).

for \( x = 1:n \)
  - statements
end

The controlvalue is simply the row vector \([ 1 \ 2 \ 3 \ 4 \ \ldots \ n ]\).

Hence, the statements are executed \( n \) times.
- The first time through, the value of \( x \) is set equal to 1;
- the \( k \)-th time through, the value of \( x \) is set equal to \( k \).

switch, case, otherwise end

Let \( VAL \) be value of \( \text{expression} \). Assume \( VAL \) is a scalar double.

switch \( VAL \)
  case testval1
    statements1
  case testval2
    statements2
  otherwise
    statementsOther
end

Any number of cases are allowed. There does not have to be an otherwise (and associated statements).
**switch, case, otherwise end**

```plaintext
switch expression
  case testval1
    statements1
  case testval2
    statements2
  otherwise
    statementsOther
end
```

- VAL (the value of expression) need not be a scalar double. It can also be a char array.
- Matlab uses `strcmp` (string compare) to check equality of VAL to the various `testval1`, `testval2`, etc.

**if, else, end**

```plaintext
if exp_1
  statements1
else
  statements2
end
```

- One of the sets of statements will be executed
  - If `exp_1` is TRUE, then `statements1` are executed
  - If `exp_1` is FALSE, then `statements2` are executed

**while**

```plaintext
while expression
  statements
end
```

- Executing commands an undetermined number of times.
- Evaluate expression
  - If TRUE, execute statements
  - If FALSE, jump past end

**if, elseif, end**

```plaintext
if exp_1
  statements1
elseif exp_2
  statements2
elseif exp_3
  statements3
end
```

- Could also have an `else` before the `end`
Illegal stuff

if exp_1
  statements1
elseif exp_2
  statements2
else
  statements3
elseif exp_4
  statements4
end

try/catch

try
  statements1
catch
  statements2
end

Try to execute these commands
If a run-time error occurs while executing statements1, trap error, and...
Execute these commands
But, if no error occurs executing statements1, jump beyond end
Do not execute statements2

Functions

In mathematics, a function is a rule that assigns to each value of the input, a corresponding output value.

Consider the function \( f \) defined by the rule
\[
  f(x) = x^2 \text{ for all numbers } x.
\]
Here, \( x \) is the input value, and \( f(x) \) is the output value.

Equivalently, we could have written the rule as
\[
  f(y) = y^2 \text{ for all numbers } y.
\]

Functions can have many inputs and produce (through multiple rules) many outputs, \( f_1(a,b,c) = 2a+3b, f_2(a,b,c) = bc \).

FUNCTIONS in programming

If you have a "complex" task (eg., more than two lines of Matlab program code) that you will reuse in a few (or several places), you will want to write that task as a reusable function file.

Example: trigonometric \( \text{SIN} \) function

Then, every time you need to execute that task, you will only need to "call" the function.

This modularity helps break down a huge program task into a collection of smaller tasks, which individually are easier to design, write, debug and maintain.

functions are also called subroutines, methods, etc.

A MATLAB function file

The first line is the function declaration line.

function \([dp, cp] = \text{vecop}(v, w)\)

The output variables. This function has two. The function's purpose is to compute these variables, based on the values of the input variables.

The input variables. This function has two. Within the function, the names of the input variables are \( v \) and \( w \).

The function name, this function should be saved in a file called \text{vecop.m}.

Picture of \text{VECOP} function

OutputArgument #1
InputArgument #1
OutputArgument #2
InputArgument #2
6-line function in vecop.m

```matlab
function [dp,cp] = vecop(v,w)
    dp = sum(v.*w);
    cp = zeros(3,1);
    cp(1) = v(2)*w(3) - w(2)*v(3);
    cp(2) = v(3)*w(1) - w(3)*v(1);
    cp(3) = v(1)*w(2) - w(1)*v(2);
end
```

Logically correct expressions and assignments that compute the output variables using the values of the input variables.

Comments and blank lines add readability

```matlab
function [dp,cp] = vecop2(v,w)
    % VECOP computes dot product and cross product of two 3-by-1 vectors.
    dp = sum(v.*w);
    cp = zeros(3,1);  % create 3-by-1
    % Fill cp
    cp(1) = v(2)*w(3) - w(2)*v(3);
    cp(2) = v(3)*w(1) - w(3)*v(1);
    cp(3) = v(1)*w(2) - w(1)*v(2);
end
```

Calling a function

```matlab
>> v1 = [1;-2;3];
>> v2 = [0;1;1];
>> [A,B] = vecop(v1,v2);
```

BASE WORKSPACE
- v1 3-by-1
- v2 3-by-1
- A 1-by-1
- B 3-by-1

FUNCTION INSTANCE WS
- dp 1-by-1
- cp 3-by-1

Communicating across workspaces

A few built-in Matlab functions facilitate communication across workspace boundaries. The most obvious communication across workspaces occurs with function calls.

- Input variables are copied into function-instance workspace
- Actually a delayed-copy --- no overhead in passing in large arrays
- Output variables are assigned back in caller's workspace

Specific commands which give additional functionality are:

- `assignin('caller','xyz',Value)`
  - Assign a variable named `xyz` in `caller`'s workspace, with value `Value` (from current workspace)
- `A = evalin('caller','CharExpression')`
  - Evaluate (with `eval`) the expression in the `caller`'s workspace
  - Assign the result to a variable named `A` in the current workspace

Example with `assignin`

```matlab
function [B] = EXAMP1(A)
    assignin('caller','NEWV',2*A);
    B = 3*A;
end
```

```matlab
>> v1 = [1 -2 3];
>> [B] = EXAMP1(v1);
```

BASE WORKSPACE
- v1: [1 -2 3]

FUNCTION INSTANCE WS
- NEWV: [2 -4 6]
- B: [3 -6 9]

Example with `inputname`

```matlab
function [C] = EXAMP2(A,B)
    D = inputname(1);
    E = inputname(2);
    C = A + B;
end
```

```matlab
>> fred = [2 3];
>> joe = [5 5];
>> [W] = EXAMP2(fred,joe);
```

BASE WORKSPACE
- Fred: [2 3]
- Joe: [5 5]

FUNCTION INSTANCE WS
- A: [2 3]
- B: [5 5]
- C: [7 8]
Example with `evalin`

```matlab
function [dp,cp] = vecop3(v,w)
    disp('Just inside VECOP3 function')
    disp(' FunctionWS'); whos
    disp(' CallerWS'); evalin('caller','whos');
    dp = sum(v.*w);
    disp('After DP create')
    disp(' FunctionWS'); whos
    disp(' CallerWS'); evalin('caller','whos');
    cp = zeros(3,1);
    disp('After CP create')
    disp(' FunctionWS'); whos
    disp(' CallerWS'); evalin('caller','whos');
    cp(1) = v(2)*w(3)-w(2)*v(3);
    cp(2) = v(3)*w(1)-w(3)*v(1);
    cp(3) = v(1)*w(2)-w(1)*v(2);
end
```

**Function handles**

Several Matlab functions solve your problem by repeatedly calling a function(s) that you supply

- finding zeros of a function (`fzero`)
- Integrating a function (`trapz`, `quad`)
- Integrating a differential equation (`ode45`)
- minimizing a function (`fminbnd`, `fminsearch`)

For example, in calling `fzero`, you must pass a reference-to-the-function (called a function handle) that you want `fzero` to find the zeros of. Use the `@` operator to get the reference.

```matlab
shan = @sin;
class(shan)
shan([0 pi/2 pi 3*pi/2 2*pi])
```

You can call a function using its name, or its handle

```matlab
[out1,out2,...] = myfunc(arg1,arg2,...)
fh = @myfunc;
[out1,out2,...] = fh(arg1,arg2,...)
```

Always "retrieve" the handle using the `@` operator

Arrays of function handles are not allowed (the one exception to "in Matlab, everything is an array"). Attempting to create one works,

```matlab
af = [@cos @sin]
```

but, generates a warning of future doom...