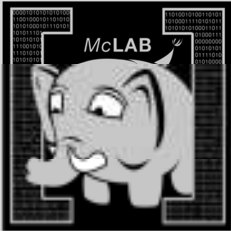


# McLab Tutorial

[www.sable.mcgill.ca/mclab](http://www.sable.mcgill.ca/mclab)



## Part 2 – Introduction to MATLAB

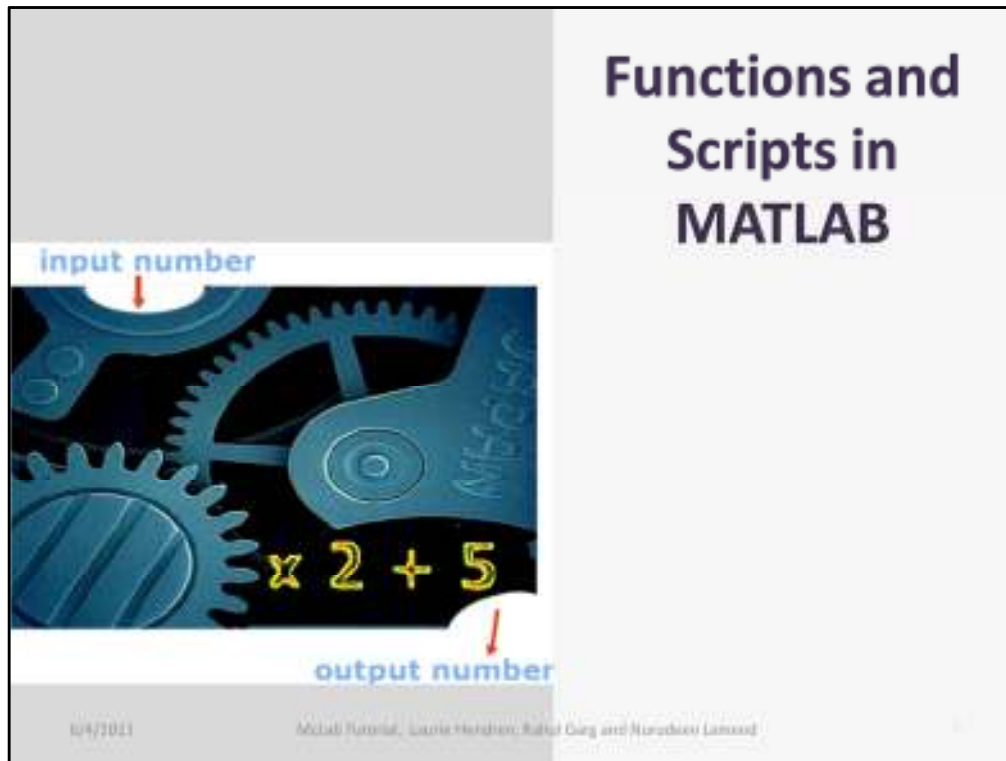
- Functions and Scripts
- Data and Variables
- Other Tricky "Features"

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab- 1

Before we can understand the tools, we first need to understand the MATLAB language. We have distilled out the important parts, and will first introduce functions and scripts, and then introduce data and variables. We then discuss some Matlab-specific tricky features.



There are two main ways of defining MATLAB computations, through functions which have input and output parameters, and through scripts which have no parameters, and are effectively just a sequence of statements. Let's look at functions first.

## Basic Structure of a MATLAB function

```

1 function [ prod, sum ] = ProdSum( a, n )
2   prod = 1;
3   sum = 0;
4   for i = 1:n
5       prod = prod * a(i);
6       sum = sum + a(i);
7   end;
8 end

```

```

>> [a,b] = ProdSum([10,20,30],3)
a = 6000
b = 60

>> ProdSum([10,20,30],2)
ans = 200

>> ProdSum('abc',3)
ans =941094

>> ProdSum([97 98 99],3)
ans = 941084

```

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 3

On first click:

Here is an example of a function definition of a function called ProdSum. It has two input parameters, "a" and "n", and two output parameters, "prod" and "sum". There is also a local variable "i".

This function should be stored in a file called ProdSum.m. If it is stored in a file of another name, say "foo.m" then the function can only be called as foo(...).

Note that there are no type declarations and no explicit declarations of variables. "i" is a variable because it is defined (i.e. it is assigned to in the header of the for loop).

There are no return statements for returning the values of the output parameters, the values returned are simply the values those variables contain when the function returns.

On second click:

Here is an example of an interaction with the MATLAB read-eval-print loop. The user types in the statement after the ">>" prompt and the statement is evaluated and the result printed. If an explicit assignment is made in the statement, the values of the lhs are printed, otherwise the result of evaluating the expression is assigned to a variable called "ans" and its value is printed.

The first call calls ProdSum with a 1x3 array [10,20,30] as the first argument, and a 1x1 array 3 as the second argument. The first return value is assigned to "a" and the second to "b".

The second call does not give a lhs, so the first of the return values is assigned to ans and the 2<sup>nd</sup> return value is thrown away. A similar thing would happen if you explicitly said "myans = ProdSum....".

## Basic Structure of a MATLAB function (2)

```

1 function [ prod, sum ] = ProdSum( a, n )
2   prod = 1;
3   sum = 0;
4   for i = 1:n
5       prod = prod * a(i);
6       sum = sum + a(i);
7   end;
8 end

```

```

>> [a,b] = ProdSum(@sin,3)
a = 0.1080
b = 1.8919

>> [a,b] = ProdSum(@(x)(x),3)
a = 6
b = 6

>> magic(3)
ans = 8 1 6
      3 5 7
      4 9 2

>>ProdSum(ans,3)
ans=96

```

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 4

1<sup>st</sup> click: On the previous slide we saw that the first argument could be 1xn arrays of different types. What else is allowed?

2<sup>nd</sup> click: The first statement shows a call using a function handle, @sin. This changes the meaning of a(i) in the body, as now it means an indirect call to sin(i). Note that the syntax of an indirect call is the same as the syntax for a direct call and an array index.

The 2<sup>nd</sup> statement illustrates an anonymous function, in this case just the identity function.

The 3<sup>rd</sup> and 4<sup>th</sup> statements illustrate what happens when you provide an array with higher dimensions than what is required by an indexing statement. In this case we create a 3x3 magic square, using the built-in function "magic" and then provide that as the 1<sup>st</sup> argument to "ProdSum". The indexing expression a(i) will still work, but it will automatically use 1 for all the missing dimensions. So, in this case it is effectively a(i,1), and the result is the product of the first column of a.

## Basic Structure of a MATLAB function (3)

```

1 function [ prod, sum ] = ProdSum( a, n )
2   prod = 1;
3   sum = 0;
4   for i = 1:n
5       prod = prod * a(i);
6       sum = sum + a(i);
7   end;
8 end

```

```

>> ProdSum([10,20,30],'a')
??? For colon operator with char operands, first and
last operands must be char.
Error in ==> ProdSum at 4
    for i = 1:n

>> ProdSum([10,20,30],i)
Warning: Colon operands must be real scalars.
> In ProdSum at 4
ans = 1

>> ProdSum([10,20,30],[3,4,5])
ans = 6000

```

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 5

1<sup>st</sup> click: although MATLAB tolerates many different inputs, not all inputs will result in an answer, some will trigger errors, others will trigger warnings.

2<sup>nd</sup> click: For example, what happens when we give a character value for n, we might expect this to work, since characters were valid for the first argument. However, this causes a run-time error, since the colon operator only works with characters if both the left and right operands are characters. This is unlike the \* and + operators, which tolerate arguments of different types.

The 2<sup>nd</sup> statement using "i" as the 2<sup>nd</sup> argument. What does this mean. In this case, it means the library function i, which returns the complex value 0+1i. In this case a warning is issued at run-time saying the colon wants real scalars, but it happily continues on assuming the real value 0.

Based on the previous error message we might expect that the colon operator only wants scalars, however if we try giving it [3,4,5], then it happily just uses the first element, 3, and produces the product of [10,20,30].

## Primary, nested and sub-functions

```

% should be in file NestedSubEx.m
function [ prod, sum ] = NestedSubEx( a, n )
    function [ z ] = MyTimes( x, y )
        z = x * y;
    end
    prod = 1;
    sum = 0;
    for i = 1:n
        prod = MyTimes(prod, a(i));
        sum = MySum(sum, a(i));
    end;
end

function [z] = MySum ( x, y )
    z = x + y;
end
  
```

6/4/2011      McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed      Matlab - 6

Although MATLAB programmers tend to put only one function in a file, there are some mechanisms for collecting together related functions in the same file.

The first function is called the primary function, and this one should have the same name as the file. This is the only function which is visible outside of the file.

Subsequent functions, defined after the primary function are called sub-functions. Sub-functions are only visible to other functions in the same file.

Function definitions can also be nested, and these follow the expected scoping rules. The only non-obvious point is determining which function "declares" a local variable. Effectively it is the innermost function which contains a parameter or definition of that variable.

## Basic Structure of a MATLAB script

```

1 % stored in file ProdSumScript.m
2 prod = 1;
3 sum = 0;
4 for i = 1:n
5     prod = prod * a(i);
6     sum = sum + a(i);
7 end;

```

```

>> clear
>> a = [10, 20, 30];
>> n = 3;
>> whos
  Name   Size   Bytes   Class
  a      1x3    24      double
  n      1x1     8      double
>> ProdSumScript()
>> whos
  Name   Size   Bytes   Class
  a      1x3    24      double
  i      1x1     8      double
  n      1x1     8      double
  prod   1x1     8      double
  sum    1x1     8      double

```

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 7

Now that we understand functions, let's look at something a little less structured, scripts ....

1<sup>st</sup> click: If a script is stored in a file foo.m, then it is called by "foo" or "foo()"

A script is neither a zero-argument function, nor a macro, but something in between.

Scripts use the workspace of their caller, which could be the read-eval-print loop, or the last-called function.

2<sup>nd</sup> click: let's look at an example of calling a script from the read-eval-print loop. First we have to ensure that the appropriate variables are defined (statements 1 through 3). Note that "whos" is a built-in function that displays the current workspace. We then call ProdSumScript, and then look in the workspace again, where we see that prod and sum have been defined.

## Directory Structure and Path

- Each directory can contain:
  - .m files (which can contain a script or functions)
  - a `private/` directory
  - a package directory of the form `+pkg/`
  - a type-specialized directory of the form `@int32/`
- At run-time:
  - current directory (implicit 1<sup>st</sup> element of path)
  - path of directories
  - both the current directory and path can be changed at runtime (`cd` and `setpath` functions)

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 8

MATLAB programmers tend to accumulate lots of functions in their current directory. However, there are mechanisms for grouping functions together.

- First, you can put them in a `/private` directory. These functions will be visible to functions in the outer directory.
- Second you can create directories starting with "+" which correspond to packages. Such functions must be called using `pkg.f()`. You can have sub-packages as well.
- Third, you can have type-specialized directories, which start with "@". These will be called when the first argument matches the type of the directory name.

At run-time a function is looked up first in the current directory, and then if not found the directories along the path are searched. Note that both the current directory and the path can be changed at run-time.



## Function/Script Lookup Order (call in the body of a function f)

- Nested function (in scope of f)
- Sub-function (in same file as f)
- Function in /private sub-directory of directory containing f.
- 1<sup>st</sup> matching function, based on function name and type of first argument, looking in type-specialized directories, looking first in current directory and then along path.
- 1<sup>st</sup> matching function/script, based on function name only, looking first in current directory and then along path.

```
function f
...
foo(a);
...
end
```

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 9

Let's look at the rules for looking up a function. They are as listed on the slide. Note that a nested, sub-function or private function takes precedence over a type-specialized function. Hence, if you want to make a function type-specialized it must be moved out a file or private directory.

In the example, for the call of foo, first look in the body of f, then in the file of f, then in the /private directory of the directory containing the file of f. If not found yet, then look along the path for a type-specialized foo that matches the run-time type of "a", and then if not found, look along the path for a function with name "foo".

## Function/Script Lookup Order (call in the body of a script s)

```
% in s.m
...
foo(a);
...
```

- Function in /private sub-directory of directory of last called function (not the /private sub-directory of the directory containing s).
- 1<sup>st</sup> matching function/script, based on function name, looking first in current directory and then along path.

dir1/	dir2/
f.m	s.m
g.m	h.m
private/	private/
foo.m	foo.m

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 10

Now, what about a looking up a call which is the body of a script? This is not equivalent to first macro-expanding the call. It is also different than the lookup call for functions.

- If f.m is a function definition containing a call to foo, foo will be found in dir1/private/foo.m.
- If s.m is a script definition containing a call to foo, foo will not necessarily be found in dir2/private/foo.m. Rather, it depends on the directory of the last function called. For example, if g.m called s, then foo will be dir1/private/foo.m. If h.m called s, then foo will be dir2/private/foo.m.

## Copy Semantics

```

1 function [ r ] = CopyEx( a, b )
2   for i=1:length(a)
3     a(i) = sin(b(i));
4     c(i) = cos(b(i));
5   end
6   r = a + c;
7 end

```

```

>> m = [10, 20, 30]
m = 10  20  30

>> n = 2 * a
n = 20  40  60

>> CopyEx(m,n)
ans = 1.3210  0.0782 -1.2572

>> m = CopyEx(m,n)
m = 1.3210  0.0782 -1.2572

```

6/4/2011      McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed      Matlab - 11


The semantics of MATLAB is call and return by value. Hence when a function call is made a copy of the input arguments are made, and then the function returns a copy of the return parameters are made. Similarly, statements of the form `a = b` mean that `a` should be a new copy.

In an implementation of MATLAB with reference counting (such as MathWorks' MATLAB and Octave), the copying is actually done lazily. At the time of parameter passing/returning or array assignments, only a reference is created, and the reference count of the pointed-to array is incremented. Then, upon any update to an array, first the reference count is checked. If the reference count is greater than 1, then a fresh copy is created at this point.

McVM uses a different approach. Since McVM supports a garbage-collected system, rather than reference counted, we use static analysis to determine when copies are needed, and the best place to insert such copies. This is reported in the paper "Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler", published in CC 2011.

Also note that fresh copies of arrays may need to be allocated if an element is assigned outside of the current range. Thus, in `CopyEx`, line 4, on each iteration of the for loop it will cause a fresh copy to be created, which is one element larger.


## Variables and Data in MATLAB



6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

12



Ok, now we understand functions.... what about variables in MATLAB.

We already know they are not explicitly declared.

## Examples of base types

```
>> clear
>> a = [10, 20, 30]
a = 10  20  30
```

```
>> b = int32(a)
b = 10  20  30
```

```
>> c = isinteger(b)
c = 1
```

```
>> d = complex(int32(4),int32(3))
d = 4 + 3i
```

```
>> whos
```

Name	Size	Bytes	Class	Attributes
a	1x3	24	double	
b	1x3	12	int32	
c	1x1	1	logical	
d	1x1	8	int32	complex

```
>> isinteger(c)
```

```
ans = 0
```

```
>> isnumeric(a)
```

```
ans = 1
```

```
>> isnumeric(c)
```

```
ans = 0
```

```
>> isreal(d)
```

```
ans = 0
```

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 13

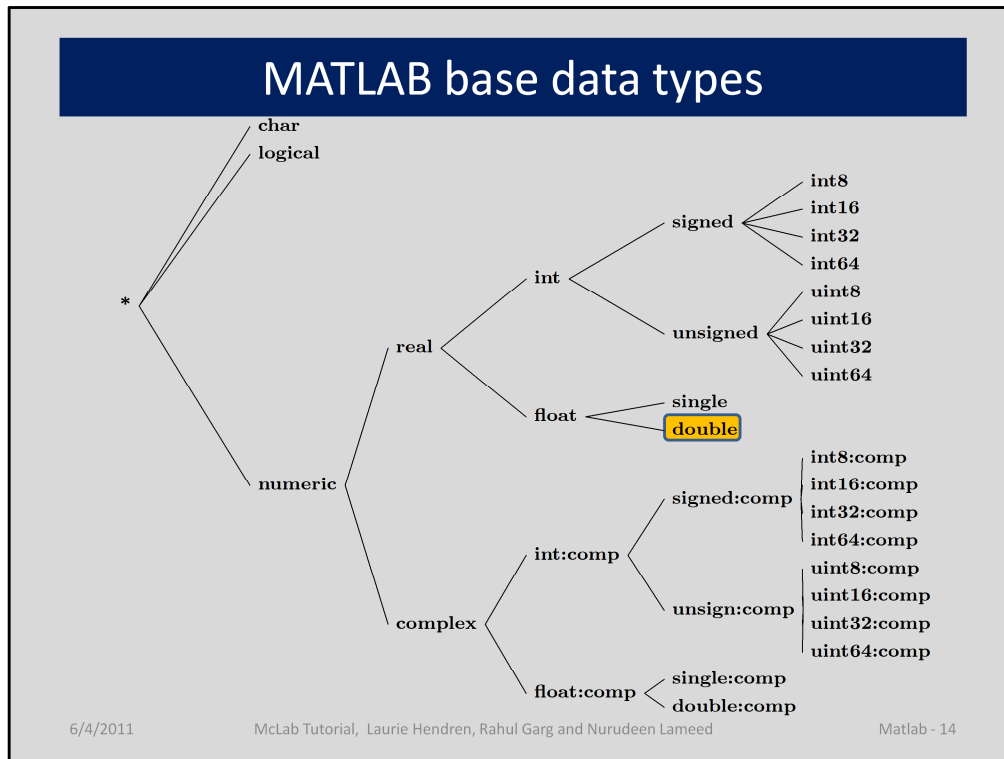
Let's first look at the base types.

If the programmer doesn't say anything different, then the base type is double – even if syntactically it looks like an integer.

To create an integer, one has to explicitly convert it, using a library function like "int32".

There are a number of built-in functions for testing the type, for example "isinteger", which return a variable with type "logical", although when printed out, they also look like integers.

There are also complex values, which have two components. These components need not be represented as doubles.



Here is our organization of the base types.

## Data Conversions

- `double + double` → `double`
- `single + double` → `double`
- `double:complex + double` → `double:complex`
- `int32 + double` → `int32`
  
- `logical + double` → error, not allowed
- `int16 + int32` → error, not allowed
- `int32:complex + int32:complex` → error, not defined

6/4/2011

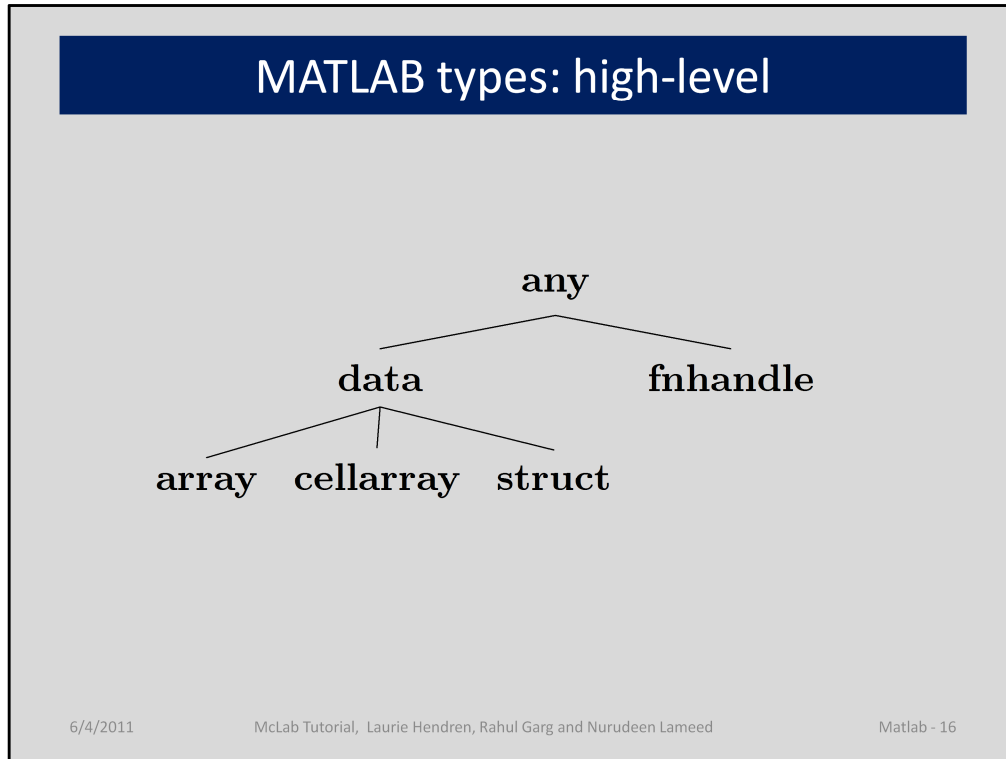
McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 15

Operations dynamically check the type of their arguments and then determine the type of their return value. Not all of these conversions are as you might expect.

The first three bullet points are as you would expect, but the fourth one shows that adding an `int32` to a `double` results in an `int32`.

Although doubles can be combined with other types quite easily, other combinations are not allowed, as illustrated by the final three bullet points.



Now let's look at the high-level data types. A variable can be either data or a function handle. If it is data, then it could be an array, which is homogenous, a cellarray which is effectively an array of cells, where each cell can contain a different type, or it can be a struct with named fields.



## Cell array and struct example

```
>> students = {'Nurudeen', 'Rahul', 'Jesse'}
students = 'Nurudeen' 'Rahul' 'Jesse'
```

```
>> cell = students(1)
cell = 'Nurudeen'
```

```
>> contents = students{1}
contents = Nurudeen
```

```
>> whos
```

Name	Size	Bytes	Class
cell	1	128	cell
contents	1x8	16	char
students	1x3	372	cell

```
>> s = struct('name', 'Laurie',
             'student', students)
s = 1x3 struct array with fields:
    name
    student
```

```
>> a = s(1)
a = name: 'Laurie'
    student: 'Nurudeen'
```

```
>> a.age = 21
a = name: 'Laurie'
    students: 'Nurudeen'
    age: 21
```

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 17

Cell arrays can contain any type of data. Cell arrays are created using the { } syntax, rather than [ ] for normal arrays. Indexing into a cell array is done using x(i) to get the i'th cell, and x{i} to get the contents of the i'th cell.

Structs are created using the struct function, as shown at the top of the 2<sup>nd</sup> column. Note that in this example, since students has shape 1x3 the created struct is a 1x3 struct array with each struct containing a name field with 'Laurie' and the i'th struct containing the i'th element of students.

If we take out the first struct, and assign to a, then we have a single struct. We can add more fields to this struct by assigning to a field which doesn't yet exist, for example a.age = 21 causes a new field to be added.

## Local variables

- Variables are not explicitly declared.
- Local variables are allocated in the current workspace.
- All input and output parameters are local.
- Local variables are allocated upon their first definition or via a load statement.
  - `x = ...`
  - `x(i) = ...`
  - `load ('f.mat', 'x')`
- Local variables can hold data with different types at different places in a function/script.

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 18

Variables are by default local. They are implicitly declared through assignments, or through load statements. Variables can hold different types at different places in the body of a function/script.

## Global and Persistent Variables

- Variables can be declared to be global.
  - `global x;`
- Persistent declarations are allowed within function bodies only (not allowed in scripts or read-eval-print loop).
  - `persistent y;`
- A persistent or global declaration of `x` should cover all defs and uses of `x` in the body of the function/script.

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 19

It is possible to explicitly declare a variable to be global or persistent.

## Variable Workspaces

- There is a workspace for global and persistent variables.
- There is a workspace associated with the read-eval-print loop.
- Each function call creates a new workspace (stack frame).
- A script uses the workspace of its caller (either a function workspace or the read-eval-print workspace).

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 20

We can think of the environment as being a collection of workspace, one for globals and persistents, one for the read-eval-print-loop and one for each function call.

## Variable Lookup

- If the variable has been declared global or persistent in the function body, look it up in the global/persistent workspace.
- Otherwise, lookup in the current workspace (either the read-eval-print workspace or the top-most function call workspace).
- For nested functions, use the standard scoping mechanisms.

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 21

How are variables looked up? If global/persistent, look in the global/persistent workspace, otherwise lookup in the current workspace. If not found, then may trigger a run-time error.

## Local/Global Example

```

1 function [ prod ] = ProdSumGlobal( a, n )
2   global sum;
3   prod = 1;
4   for i = 1:n
5     prod = prod * a(i);
6     sum = sum + a(i);
7   end;
8 end;

```

```

>> clear

>> global sum

>> sum = 0;

>> ProdSumGlobal([10,20,30],3)
ans = 6000

>> sum
sum = 60

>> whos

```

Name	Size	Bytes	Class	Attributes
ans	1x1	8	double	
sum	1x1	8	double	global

6/4/2011

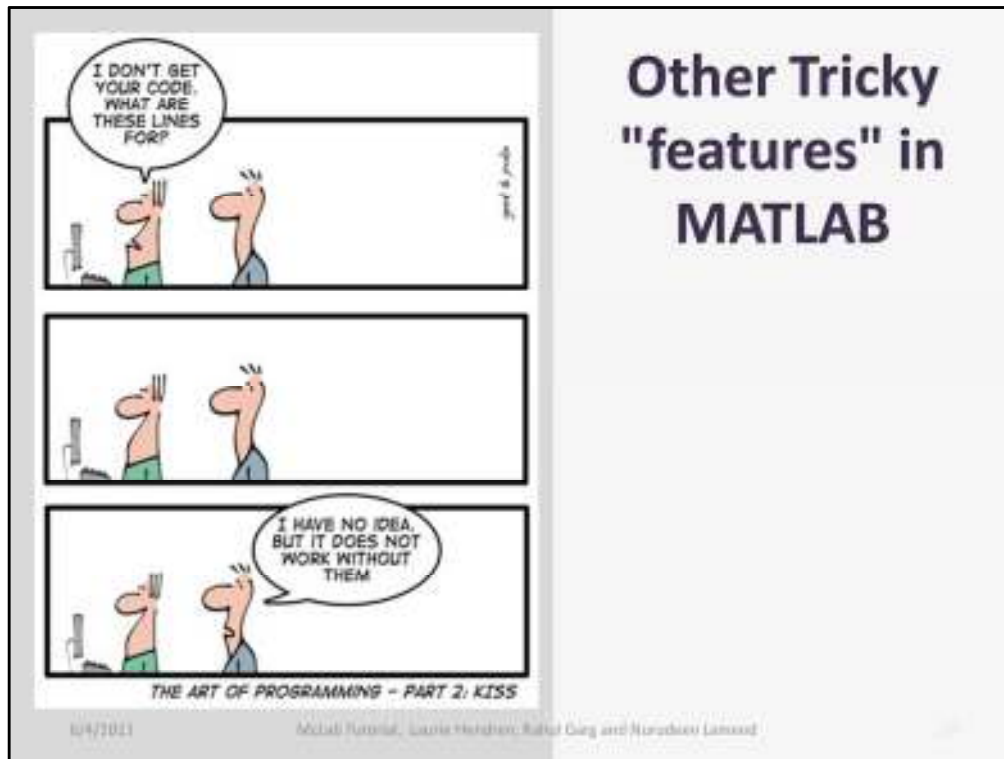
McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 22

1<sup>st</sup> click: Let's look at a variation of our example where sum is a global variable instead of an input parameter.

2<sup>nd</sup> click: In the calling context we must also declare sum to be global and give it an initial value. The default value is the empty array, so in this case we initialize it to 0.

After calling the function, the global now has accumulated the sum of a. We can see that globals have a global attribute, when displayed with "whos".



There are some tricky and non-obvious features in MATLAB.

## Looking up an identifier

### Old style general lookup - interpreter

- First lookup as a variable.
- If a variable not found, then look up as a function.

### MATLAB 7 lookup - JIT

- When function/script first loaded, assign a "kind" to each identifier. VAR – only lookup as a variable, FN – only lookup as a function, ID – use the old style general lookup.

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 24

There are two styles of looking up an identifier, the old style interpreter based lookup, and a new lookup implemented in MATLAB 7, which has a JIT. This change in lookup has effectively changed the semantics of MATLAB, and so all tools that handle modern MATLAB must implement the new semantics.

In the new semantics, at first load time each identifier is assigned a kind. This is, presumably, to allow for more efficient code generation. The kinds are VAR, FN and ID.

We cannot find any documentation on this, but have written a paper on this topic, submitted to OOPSLA.



## Kind Example

```

1 function [ r ] = KindEx( a )
2   x = a + i + sum(j)
3   f = @sin
4   eval('s = 10;')
5   r = f(x + s)
6 end

```

```

>> KindEx(3)
x = 3.0000 + 2.0000i
f = @sin
r = 1.5808 + 3.2912i
ans = 1.5808 + 3.2912

```

- VAR: r, a, x, f
- FN: i, j, sum, sin
- ID: s

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 25

Let's take a quick look at the kind assignment algorithm. Effectively it must visit all statements in the body of the function and assign a kind. The algorithm implemented by MATHWORKS is neither flow-sensitive nor flow-insensitive, but traversal-sensitive. In this example, "a" and "r" are parameters, so they are variables. "x" and "f" are assigned-to, so they are variables. "sin" has its handle taken, so it is a FN. "i", "j", "sum" are also FN because: (a) they are not VAR, and (b) they can be found when looked in the current fn/file/directory/path. Identifier "s" is not directly defined, nor can it be found upon a function lookup, so its kind is left as ID, and a fully dynamic lookup will be used at runtime. In this case when you run the program, "s" will be found in the workspace.

2<sup>nd</sup> click: trace of running, note that the statements in the body of KindEx are not terminated by ";", so the results of the statements are echoed at run-time.

## Irritating Front-end "Features"

- keyword `end` not always required at the end of a function (often missing in files with only one function).
- command syntax
  - `length('x')` or `length x`
  - `cd('mydirname')` or `cd mydirname`
- arrays can be defined with or without commas:  
`[10, 20, 30]` or `[10 20 30]`
- sometimes newlines have meaning:
  - `a = [ 10 20 30`  
    `40 50 60 ];` // defines a 2x3 matrix
  - `a = [ 10 20 30 40 50 60];` // defines a 1x6 matrix
  - `a = [ 10 20 30;`  
    `40 50 60 ];` // defines a 2x3 matrix
  - `a = [ 10 20 30; 40 50 60];` // defines a 2x3 matrix

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 26

There are quite a few irritating issues with the MATLAB syntax that are hard to handle with standard parsing tools.

## “Evil” Dynamic Features

- not all input arguments required

```
1 function [ prod, sum ] = ProdSumNargs( a, n )
2   if nargin == 1 n = 1; end;
3   ...
4 end
```

- do not need to use all output arguments
- eval, evalin, assignin
- cd, addpath
- load

6/4/2011

McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed

Matlab - 27

There are also several potentially evil dynamic features.

First, when a function is called, not all arguments need to be provided. In the body of the function one can check to see how many arguments were provided, and then assign a default value to the missing ones (as in line 2 of ProdSumNargs).

Similarly a call to a function need not capture all of the output arguments.

There are several kinds of eval, including the ordinary one, evalin – which is used to eval an expression in a different context (such as the calling functions context) and assignin which is used to assign to a variable in the calling context.

Cd and addpath can be used to dynamically change the current directory or modify the path, which causes a change in function lookup.

Load can be used to load stored variables from a file, and so may cause new variables to be created in the current workspace.