Session 2: Data types and structures

Version 1.0.1

Aaron Ponti

A data type (or datatype) is a classification identifying one of various types of data, such as floating-point, integer, or Boolean, that determines: (1) the possible values for that type; (2) the operations that can be done on values of that type; and (3) the way values of that type can be stored. Most programming languages also allow the programmer to define *additional* data types (or *classes*), usually by combining multiple elements of other types and defining the valid operations of the new data type.

Contents

1 Data types

Compared to strongly-typed programming languages, MATLAB does not force you to specify the datatype of a variable before you assign a value to it. If you tried the following in a C++ program:

```
a = 3;
```

the compiler would tell you something like:

```
error: 'a' was not declared in this scope
```
[1]

In contrast, as we saw in session 1, in MATLAB it is perfectly legitimate to type:

```
>> a = 3;
```

MATLAB creates a variable called *a* and assigns the value 3 to it. It is important to realize that the variable is **not** type-less: MATLAB knows an extensive number of data types (more to this below), but automatically decides on the variable type depending on what one tries to assign to the variable itself. The default data type for a numeric right-value (the object on the right-hand side of the = sign in an assignment) is *double*. One can test it by typing:

```
>> class( a )

ans =
double
```

A series of data types are available in MATLAB.

- logical
- char
- *numeric*
  - *integer*
    - uint8, int8
    - uint16, int16
    - uint32, int32

[1]A correct declaration (and assignment) would be: *int a = 3;*

    - uint64, int64 (on 64-bit machines only)
  - *floating point*
    - single (single precision)
    - double (double precision), contains also *complex numbers*

- struct
- cell
- function handles
- Java objects
- user-defined classes

Moreover, all these data types can be ordered into *arrays* (see section 2).

In the following we will give more details on all relevant data types.

1.1 Logical

A logical variable can take only two values: 1 (or true) and 0 (or false). You can create a logical variable like this:

```
>> a = true;
>> b = false;
>> c = logical( 1 );
>> d = logical( 3 )

d =
    1
```

By definition, 0 is converted to false when casted[2] to type logical, and everything else is casted to logical(1).

Certain MATLAB functions and operators return logical true or false to indicate whether a certain condition was found to be true or not. For example, the statement 50>40 returns a logical true value.

One can test whether a variable is logical with the following call:

```
>> islogical( a )

ans =
    1
```

1.2 Char

A variable that contains a character has class char. A string (a series of characters) is an array of chars. You can create a char variable like this:

```
>> a = 'a';
>> b = 'This is a character array (a so-called string)';
>> length( b )

ans =
    46
```

length(b) returns the length of the character arrays (the number of chars in the string).

[2]To cast a variable means to convert it from one data type to another (that is compatible).

1.3 Numeric: integers

A numeric integer, as the name says, is a positive or negative whole number (without decimal digits) or zero. Depending on the precision of the machine (e.g. 32 or 64 bit), there is a limit to the largest (*exact*) positive and negative numbers that can be stored into an integer variable. Larger numbers are possible, but will be approximated by floating point numbers and stored into a double variable.

In MATLAB, there are several different integer types: uint8, int8, uint16, int16, uint32, int32, uint64, int64 (the last two on 64-bit machines only). The *u*int (unsigned integer) data types only store positive numbers, while the int data type (signed integers) store both negative and positive numbers. The digit at the end of the type name (8, 16, 32, 64) specifies how many bits are used in memory to store the value and consequently the maximum number of different numbers that can be expressed. The maximum number of different 8-bit numbers is 2^8=256. For uint8 variables, this means all numbers between 0 and 255; for int8 variables, all numbers between -128 and 127. Although it might appear limiting (and maybe even irrelevant) to force a numeric variable to be uint8 instead of the default double, one has to note that integer (and single-precision) arrays offer more memory-efficient storage than double-precision. This can very fast become decisive when working with very large amounts of data (e.g. a 4D microscopy dataset).

The functions *intmin*('type') and *intmax*('type') return then minimum and maximum possible number that can be stored in a variable of type 'type'.

Exercise

Exercise: fill the following table.

	uint8	int8	uint16	int16	uint32	int32	uint64	int64
int min								
int max								

MATLAB stores numeric data as double by default. To store data as an integer, you need to convert from double to the desired integer type. For example:

```
>> a = int16( 1024 );
```

Be careful that MATLAB will perform mapping to the range endpoints where required.

Exercise

Exercise: Try the following and discuss what happens.

```
>> a = uint8( 150 );
>> b = uint8( 130 );
>> c = a + b
```

Exercise

Exercise: Try the following and discuss what happens.

```
>> a = int8(-120 );
>> b = uint8( 100 );
>> c = a + b
```

You can test whether a variable has a specific type by using the isa( ) function:

```
>> isa( a, 'uint8' )

ans =
    0
```

1.4 Numeric: floats

1.4.1 Real numbers

MATLAB represents floating-point numbers in either double-precision or single-precision format. The default is double precision, but you can make any number single precision with a simple conversion function (see also section 1.3):

```
>> a = single( 1.75 );
```

Just for your information, MATLAB constructs the double and single data type according to IEEE® Standard 754 for double and single precision. We don't need to know more than this.

Because MATLAB stores numbers of type single using 32 bits, they require less memory than numbers of type double, which use 64 bits. However, because they are stored with fewer bits, numbers of type single are represented to less precision than numbers of type double. Because there are only a finite number of double-precision numbers, you cannot represent all numbers in double-precision storage. On any computer, there is a small gap between each double-precision number and the next larger double-precision number. You can determine the size of this gap, which limits the precision of your results, using the *eps* function. The value of *eps*(x) is different for every value of x. To get an idea of the precision of a given data type one usually use *eps*(1).

Exercise: fill the following table. **Exercise**

	single	double
eps(1)		

Exercise: Round-Off error. The decimal number 4/3 is not exactly representable as a binary fraction. Try the following calculations. What should be the result? And what do you get? **Exercise**

```
>> 1 - 3*(4/3 - 1)
```

Moreover, the covered range of single data is much smaller than the range covered by double data. In analogy to the integers, one can obtain the minimum and maximum possible values for the single and double data types using the *realmax* function.

Exercise: fill the following table. **Exercise**

	single	double
real min		
real max		

1.4.2 Complex numbers

Complex numbers consist of two separate parts: a real part and an imaginary part. The basic imaginary unit is equal to the square root of -1. This is represented in MATLAB by either of two letters: i or j. Complex numbers are by default of class double but can be casted to single.

```
>> a = 1 + i * 3;
>> b = 2 - 5i;
>> c = single( b );
```

1.4.3 Special values

MATLAB uses the special values *inf*, *-inf*, and *NaN* to represent values that are positive and negative infinity, and not a number respectively[3].

Exercise

Exercise: Try the following.

```
>> a = 1 / 0
>> b = -1 / 0
>> c = 0 / 0
```

1.5 Structures

Structures are MATLAB arrays with named "data containers" called *fields*. In contrast to the elements of an array, the fields of a structure can contain any kind of data. For example, a structure referring to a patient's records might contain one field containing a text string representing a name, another containing a scalar representing a billing amount, a third holding a matrix of medical test results, and so on.

You can build structures in two ways:

- Using assignment statements
- Using the *struct* function

1.5.1 Building structures using assignment statements

You can build a simple structure array by assigning data to individual fields. MATLAB automatically builds the structure as you go along. For example, create the *patient* structure array introduced at the beginning of this section:

```
>> patient.name = 'John Doe';
>> patient.billing = 127.00;
>> patient.test = 79;
```

Now entering patient at the command line results in:

```
>> patient

patient =
       name: 'John Doe'
    billing: 127
       test: 79
```

1.5.2 Building structures using the *struct* function

You can preallocate an array of structures with the *struct* function. Its basic form is:

```
>> strArray = struct('field1',val1,'field2',val2, ...)
```

where the arguments are field names and their corresponding values. To reproduce the preceeding example:

[3]The Statistics Toolbox (ab)uses NaN to represent **missing measurements**.

```
>> patient = struct('name','John Doe','billing',127.00,'test',79);

patient =
       name: 'John Doe'
    billing: 127
       test: 79
```

1.5.3 Accessing structure fields

You can access the various fields in a structure with following syntax: *variable.fieldName.*

```
>> patient.name

ans =
John Doe
```

1.5.4 Nesting structures

A structure field can contain another structure, or even an array of structures. Once you have created a structure, you can use the *struct* function or direct assignment statements to nest structures within existing structure fields. For example:

```
>> name.firstName = 'John';
>> name.familyName = 'Doe';
>> patient.name = name

patient =
       name: [1x1 struct]
    billing: 127
       test: 79

>> patient.name

ans =
     firstName: 'John'
    familyName: 'Doe'
```

1.6 Cells

A cell is a container for any type of data. However, cells make sense only when used as arrays. We will postpone the treatment of cell arrays to section 3.

2 Numeric Arrays

All data types in MATLAB can be arranged into arrays. Arrays can have any number of dimensions and a maximum number of entries of $2^{16} - 1$ in MATLAB up to version 7.3 and of $2^{48} - 1$ as of MATLAB v7.4 (although *officially* only as of v7.5).

An essential characteristics of arrays is that all elements of an array must share the same data type.

Array dimensions can be changed dynamically, although this is not recommended for large arrays.

In the following we will discuss the array types that are most relevant for our purposes.

2.1 1-dimensional numeric arrays: vectors

A vector is a numeric array (containing any of the numeric data types we discussed in sections 1.1, 1.3, and 1.4) and comes in two flavours: **row** and **column vectors**.

2.1.1 Row vectors

Row vectors are lists of numbers separated by either commas or spaces. The number of entries is known as the "length" of the vector and the entries are often referred to as "elements" or "components" of the vector. The entries must be enclosed in square brackets.

```
>> v = [ 1 3 sqrt(5)]

v =
    1.0000    3.0000    2.2361

>> length(v)

ans =
    3
```

We can do certain arithmetic operations with vectors of the same length:

```
>> a = [ 1 2 3 ];
>> b = [ 3 2 1 ];
>> a + b

ans =
     4     4     4

>> a - b

ans =
     -2     0     2

>> a .^ b

ans =
     1     4     3

>> a .* b

ans =
     3     4     3

>> a ./ b

ans =
     0.3333     1.0000     3.0000

>> a * b
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

The operations **+** and − behave exactly has one would expect. The two vectors a and b are summed or subtracted element-wise. To obtain the same behavior with multiplication and division one has to use the **.*** and **./** operators, respectively. The reason for this is that MATLAB is a matrix-centric language, and the * operator is used for matrix multiplications (brush up you linear algebra books!). Indeed, the **a * b** call fails because of incompatibility of the dimensions of the two operand **a** and **b** (see below).

Arithmetic operations can also be performed between scalars (single numbers) and vectors. This is the only exception to the requirement of vectors being of the same length.

```
>> a = [ 1 2 3 ];
>> a + 1

ans =
     2     3     4

>> a - 1

ans =
     0     1     2

>> a .* 2

ans =
     2     4     6

>> a ./ 2

ans =
     0.5000     1.0000     1.5000

>> a * 2

ans =
     2     4     6
```

When multiplying a vector with a scalar, both operators **.*** and * behave the same.

2.1.2 Column vectors

These have similar constructs to row vectors. When defining them, entries are separated by **;** or "newlines":

```
>> c = [ 1; 3; sqrt(5)]

c =
    1.0000
    3.0000
    2.2361

>> c2 = [ 3
 4
 5 ]

 c2 =
    3
```

```
        4
        5
```

All rules for artithmetic operations we saw for row vectors apply to column vectors as well. The **size** of a vector can help us discern between a row vector and a column vector:

```
>> size( [ 1 2 3 ] )

ans =
    1    3

>> size( [ 1; 2; 3 ] )

 ans =
    3    1
```

Exercise

Exercise: Try the following and discuss the results.

```
>> a = [ 1 2 3 ];
>> b = [ 1; 2; 3 ];
>> a + b
>> a .*b
>> a * b
>> b * a
```

2.1.3 The colon (:) notation

This is a shortcut for producing row vectors.

```
>> 3:7

ans =
    3    4    5    6    7
```

More generally **a : b : c** produces a vector of entries starting with the value **a**, incrementing by the value **b** until it gets to **c** (it will not produce a value beyond c). To create decreasing vectors, one sets a > c and b < 0.

```
>> 0 : 0.5 : 2

ans =
    0    0.5000    1.0000    1.5000    2.0000

>> 2 : -0.5 : 0

ans =
    2.0000    1.5000    1.0000    0.5000    0
```

Exercise

Exercise: Try the following.

```
>> a = 7:3
```

2.1.4 Extracting bits of a vector (slicing)

One can extract parts of a vector by specifing the indices of the first and last elements to be extracted. In contrast to other programming language like C++, the first index of a vector is 1 (and not 0).

```
>> a = 2 : 2 : 20;
>> a( 5 : 7 )

ans =
    10    12    14
```

Exercise: Extract every third element from the vector **a** defined above. **Exercise**

2.1.5 Transposing

We can convert a row vector into a column vector (and vice versa) by a process called *transposing* denoted by ′:

```
>> a = [ 1 2 3 ];
>> a'

ans =
    1
    2
    3
```

Please notice that if **a** is a complex vector, **a′** returns the complex conjugate transpose of **a**. To get the simple transpose use **.′**.

2.1.6 Plotting vectors

In MATLAB, most if not all elementary functions are applied element-wise on vectors. In this section we will use our knowledge on vectors and functions to plot some elementary function (we will dedicate a large part of Session 4 to plotting in general).

Suppose we wish to plot a graph of $y = sin(3\pi x)$ for $0 \leq x \leq 1$. We do this by sampling the function at a sufficiently large number of points and then joining up the points (x, y) by straight lines. Suppose we take $N + 1$ points equally spaced a distance h apart:

```
>>  N = 50; h = 1/N; x = 0:h:1;
```

defines the set of points $x = 0, h, 2h, ..., 1 - h, 1$.

The corresponding y values are computed by:

```
>> y = sin(3*pi*x);
```

and finally plotted with:

```
>> plot(x,y)
```

Exercise: Try to plot again, this time setting $N = 10$. What do you notice? **Exercise**

Exercise: Add a label to the x and y axes and a title to the plot (hint: try with *help plot*). **Exercise**

2.1.7 Finding elements in an array

Often one is interested in finding the (indices of the) elements of an array that satisfy a specific condition. MATLAB offers the *find* function for this: precisely, *find* returns the indices of *non-zero* elements, but these can easily be created by means of *relational expressions*[4]. Try the following:

```
>> a = 1 : 10;
>> b = a > 5

b =
     0     0     0     0     0     1     1     1     1     1
```

The vector b is of class *logical* and contains 1 (true) at the positions where the condition is fulfilled. Now, try *find* on vector b:

```
>> find( b )

ans =
     6     7     8     9    10
```

The process can obviously be automatized:

```
>> find( a > 5 )

ans =
     6     7     8     9    10
```

Exercise

Exercise: Create an array a with elements from 1 to 100. Replace all even elements by 0 (*hint*: use the *mod* function).

Elementwise logical operators: &, |, ~ In Session 1 we saw the short-circuit logical operators &&, ||, and ~ that work on relation operations between scalars. The analogous *elementwise logical operators &, |* and ~ can be used to combine the results of relational operators applied to *all elements* of two or more arrays provided they are all of the same size:

```
>> a = 1 : 10;
>> b = a > 5 | a == 3

b =
     0     0     1     0     0     1     1     1     1     1
```

Exercise

Exercise: Find and display all numbers between 1 and 100 that are *even* and *divisible by 3* but *not by 4*.

2.2 2-dimensional numeric arrays: matrices

Row and column vectors are special cases of matrices. An m × n matrix is a rectangular array of numbers having m rows and n columns. It is usual in a mathematical setting to include the matrix in either round or square brackets; in MATLAB one must use square ones. For example, when m = 2, n = 3 we have a 2 × 3 matrix such as

[4]The *relational operators* are == (*equal to*), ~= (*not equal to*), < (*less than*), > (*greater than*), <= (*less than or equal to*), >= (*greater than or equal to*). See also Session 1.

$$M = \begin{bmatrix} 5 & 7 & 9 \\ 1 & -3 & -7 \end{bmatrix}$$

To enter such an matrix into MATLAB we type it in row by row using the same syntax as for vectors:

```
>> A = [ 5 7 9; 1 -3 -7 ]

A =
     5     7     9
     1    -3    -7
```

Exercise: Enter a 3 x 50 matrix with the first rows containing all integers from 1 to 50, the second with all integers from 51 to 100 and the third with all integers from 101 to 150 (hint: use the colon operator). **Exercise**

There are several commodity functions in MATLAB to create matrices: to create a zero-matrix of size m x n we can use the *zeros* function:

```
>> Z = zeros( 10, 20 );
```

Similarly, *ones* construct a matrix of ones.

```
>> O = ones( 10, 20 );
```

The identity matrix (a zero matrix with ones only on the diagonal) is constructed with *eye*:

```
>> I = eye( 5 );
```

A matrix of random numbers sampled from a standard normal distribution (therefore with mean = 0 and standard deviation = 1) can be obtained with *randn*:

```
>> R = randn( 1000, 1000 );
```

Exercise: Construct a 1000 x 1000 matrix with entries sampled from a normal distribution with mean = 25 and standard deviation = 5. **Exercise**

As with vectors, we can do certain arithmetic operations with matrices of the same size or between a scalar and a matrix of any size. The following operators are applied element-wise: +, −, .*, ./, .^

The matrix multiplication

$$A * B = (AB_{ij}) = \sum_{r=0}^{n} A_{i,r} B_{r,j}$$

between matrices A and B is defined only if the number of columns of A equals the number of rows of B. The size of a matrix is returned by the function *size*:

```
>> A = randn( 3, 2 );
>> B = randn( 2, 5 );
>> C = A * B;
>> size( C )
```

```
ans =
     3     5

>> a = 1 : 3;
>> b = a';
>> c = a * b, size( c )

c =
    14

ans =
    1   1
```

2.2.1 Transpose of a matrix

Transposing a vector changes it from a row to a column vector and vice versa (see 2.1.5). The extension of this idea to matrices is that transposing interchanges rows with the corresponding columns: the 1^{st} row becomes the 1^{st} column, and so on.

```
>> A = [ 1 2 3; 4 5 6 ];
>> B = A'

B =
     1     4
     2     5
     3     6
```

2.2.2 Extracting bits of a matrix (slicing)

We may extract sections from a matrix in much the same way as for a vector (see 2.1.4). Each element of a matrix is indexed according to which row and column it belongs to. The entry in the i_{th} row and j_{th} column is denoted mathematically by $A_{i,j}$ and, in MATLAB, by $A(i,j)$. So

```
>> I = [ 1 2 3; 4 5 6 ]

I =
    1     2     3
    4     5     6

>> I( 1, 1 )

ans =
     1

>> I( 2, 3 ) = I( 1, 1 ) + 2 * I( 2, 2 )

I =
    1     2     3
    4     5    11
```

As for vectors, the colon notation is useful for matrices as well:

```
>> I( 1 : 2, 2 : 3 )

I =
    2     3
    5    11
```

The colon operator alone can be used to extract a full row or column:

```
>> I( :, 1 )

I =
    1
    4

>> I( 1, : )

I =
    1    2   3
```

The following returns the whole matrix:

```
>> I( :, : )

I =
    1     2     3
    4     5    11
```

2.2.3 Iterating over arrays

As we saw in the previous session, in MATLAB a *for* statement has following syntax:

```
for i = start_value : step: end_value
  statements
end
```

which means that the counter *i* iterates over the *anonymous array* with elements ranging from *start_value* to *end_value* with step *step* and executes the following statements as many times as there are elements in the array. The *end* statement closes the for block (or for *loop*).

Most of the other programming languages also use for loops (or equivalent mechanisms) just to iterate over the elements of an array (for example to find the values larger than 0). MATLAB is a matrix language, designed for vector and matrix operations and supports *vectorization*, which means converting for loops (wherever possible) to equivalent vector or matrix operations.

Until a few releases ago there was really no alternative to vectorization, since using for loops to iterate over arrays in MATLAB was so slow to be practically impossible. In the last few years, the MATLAB interpreter has been incrementally rewritten and has sped up execution of for loops by *several orders of magnitude* (although it hasn't quite closed the gap yet). Still, for loops for array iteration are against the MATLAB design and should be avoided wherever possible.

Moreover, which is easier?

```
>> C = A + B;
```

or:

```
>> C = zeros( size( A ) );
   for i = 1 : size( A, 1 )
     for j = 1 : size( A, 2 )
       C( i, j ) = A( i, j ) + B( i, j );
     end
   end
```

Exercise: Use the functions *tic* and *toc* to compare the execution times of the two code snippets above. Set A = B = rand(5000, 5000). **Exercise**

2.3 Multidimensional numeric arrays

You can use the same techniques to create multidimensional arrays that you use for two-dimensional matrices. Since these arrays are trivial extensions of the 2D case, they won't be treated any further.

3 Structure arrays

Like simple structures, structure arrays can be built either by using assignment statements or by using the *struct* function.

3.1 Building structure arrays using assignment statements

Let's look at the patient example we introduced in section 1.5.1.

```
>> patient.name = 'John Doe';
>> patient.billing = 127.00;
>> patient.test = 79;
```

If we want to add another entry to the patient structure (thus turning it into an array), we can do it like this:

```
>> patient(2).name = 'Ann Lane';
>> patient(2).billing = 28.50;
>> patient(2).test = 68;
```

The patient structure array now has size [1 2]. Note that once a structure array contains more than a single element, MATLAB does not display individual field contents when you type the array name. Instead, it shows a summary of the kind of information the structure contains:

```
>> patient

patient =
1x2 struct array with fields:
    name
    billing
    test
```

As you expand the structure, MATLAB fills in unspecified fields with empty matrices so that:

- All structures in the array have the same number of fields;
- All fields have the same field names.

For example, entering:

```
>> patient(3).name = 'Alan Johnson';
```

expands the patient array to size [1 3]. Now both patient(3).billing and patient(3).test contain empty matrices.

Note that field sizes do not have to conform for every element in an array[5]. In the patient example, the name fields can have different lengths, the test fields can be arrays of different sizes, and so on. For example, although patient(1).test and patient(2).test contain scalars one can say set patient(3).test to a 1 x 3-vector:

```
>> patient(3).test = [ 85 123 12 ];
```

3.2 Building structure arrays using the *struct* function

You can use different methods for preallocating structure arrays. These methods differ in the way in which the structure fields are initialized. As an example, consider the allocation of a 1-by-3 structure array, *weather*, with the structure fields *temp* and *rainfall*. Three different methods for allocating such an array are shown in this table.

Method	Syntax
struct	weather(1:3)=struct('temp',72,'rainfall',0.0);
struct with repmat	weather=repmat(struct('temp',72,'rainfall',0.0),1,3);
struct with cell array syntax	weather=struct('temp',{68,80,72},'rainfall',{0.2,0.4,0.0});

Exercise: Try! **Exercise**

4 Cell arrays

Creating cell arrays in MATLAB is similar to creating arrays of other MATLAB classes like double, character, etc. The main difference is that, when constructing a cell array, you enclose the array contents or indices with curly braces { } instead of square brackets [ ]. The curly braces are cell array constructors, just as square brackets are numeric array constructors. Use commas or spaces to separate elements and semicolons to terminate each row. For example, to create a 2-by-2 cell array A, type:

```
>> A = {[1 4 3; 0 5 8; 7 2 9], 'Anne Smith'; 3+7i, -pi:pi/4:pi};
```

You also can create a cell array one cell at a time. MATLAB expands the size of the cell array with each assignment statement:

```
>> A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
>> A(1,2) = {'Anne Smith'};
>> A(2,1) = {3+7i};
>> A(2,2) = {-pi:pi/4:pi};

A =
           [3x3 double]    'Anne Smith'
    [3.0000 + 7.0000i]     [1x9 double]
```

You can also use the curly braces on the left-hand size of the assignment:

[5]To achieve this, the layout of each array element has to be explicitly maintained in memory, resulting in significant memory overhead. Compare the memory usage (using *whos*) of *a.one = zeros(1, 50000);* vs. *b(50000) = struct('one', 0);*.

```
>> B{ 1, 1 } = [ 1 2 3; 4 5 6 ];
```

Curly braces are the standard way to access the content of a given position in the cell array:

```
>> A{ 2, 1 }

ans =
   3.0000 + 7.0000i
```

If you assign data to a cell that is outside the dimensions of the current array, MATLAB automatically expands the array to include the subscripts you specify. It fills any intervening cells with empty matrices. For example, the assignment below turns the 2-by-2 cell array A into a 3-by-3 cell array.

```
>> A{3,3} = 5

A =
          [3x3 double]    'Anne Smith'      []
    [3.0000 + 7.0000i]    [1x9 double]      []
                    []              []     [5]
```

The empty square brackets [] indicate an empty cell in the array.

You slice into a cell array with this notation:

```
A( :, 1 )

ans =
    [3x3 double]
    [3.0000 + 7.0000i]
```

While using curly braces for slicing will not result in an error, you will see that it does not behave very intuitively. This is due to the fact that cells have some advanced behavior in combination with specific functions (for example, *deal()*).

5 Function handles

A **function handle** is a MATLAB value that provides a means of calling a function indirectly[6]. You can pass function handles in calls to other functions (often called *function functions*). Example: the function *fminsearch* (for multidimensional unconstrained nonlinear minimization) attempts to minimize a function *fun* that accepts an input x and returns a scalar f, the objective function evaluated at x. Since *fun* can in principle be any function, fminsearch expects a function handle as an input parameter.

```
x = fminsearch( @myfun, x0 )
```

where myfun is a function file such as:

```
function f = myfun(x)
f = ...            % Compute function value at x
```

or an **anonymous function** such as

[6]In C++ it would be a *pointer* to a funtion

```
myfun = @(x)sin(x^2)
```

Anonymous functions give you a quick means of creating simple functions without having to store your function to a file each time. You can construct an anonymous function either at the MATLAB command line or in any function or script. The syntax for creating an anonymous function from an expression is

```
fhandle = @(arglist) expr
```

Exercise: Use fminsearch to find the minimum of the function $sin(x) + cos(x)$ in the vicinity of $x = 5$. **Exercise**

6 User-defined classes

MATLAB allows the creation of *ad hoc* data types called *classes*. A class incorporates both the description of the data it encapsulates and the methods (functions) that act on it. We will dedicate most of session 3 to user-defined classes.

7 References

1. The official MATLAB documentation (html):
   http://www.mathworks.com/access/helpdesk/help/helpdesk.html.
   Each product also has a pdf version of the documentation.

2. An Introduction to Matlab, Copyright (c) David F. Griffiths 1996, University of Dundee:
   http://wiki.bc2.ch/download/attachments/5702724/MatlabNotes.pdf