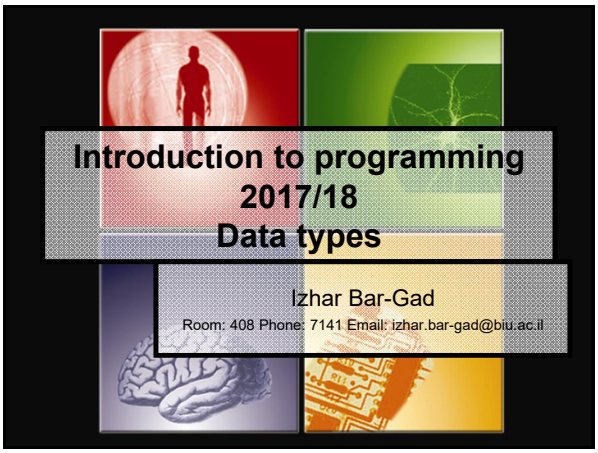
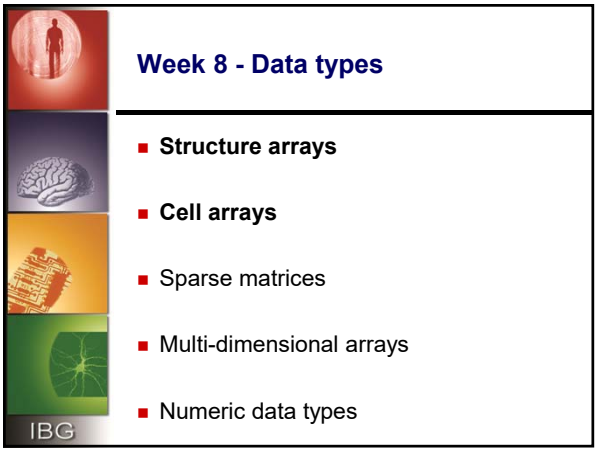
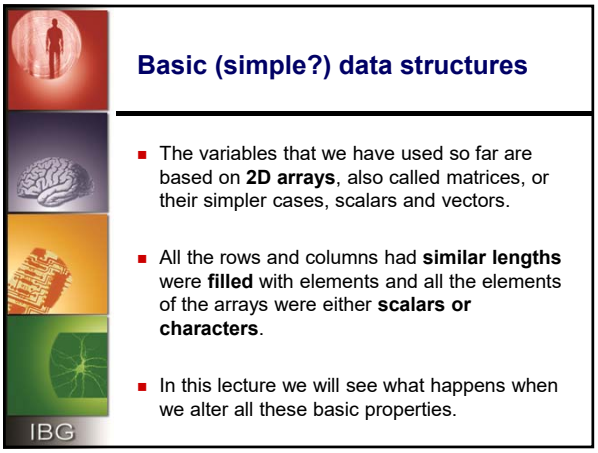

Introduction to programming
2017/18
Data types
Izhar Bar-Gad
Room: 408 Phone: 7141 Email: izhar.bar-gad@biu.ac.il


Week 8 - Data types
Structure arrays
Cell arrays
Sparse matrices
Multi-dimensional arrays
Numeric data types
IBG


Basic (simple?) data structures
The variables that we have used so far are based on 2D arrays, also called matrices, or their simpler cases, scalars and vectors.
All the rows and columns had similar lengths were filled with elements and all the elements of the arrays were either scalars or characters.
In this lecture we will see what happens when we alter all these basic properties.
IBG


## Structure arrays (structures)

- Structures are MATLAB arrays with **named** "data containers" called **fields**. The fields of a structure can contain different kinds of data.
- Multiple variables may be gathered to a single structure if they have a **common** underlying relation or denominator.
- Example: A record for a student in this course might include the fields *name* with a string value, an exGrades field which is a 12 element vector, a *projectGrade* field which is a scalar.


## Building structures

- Two basic ways of creating (or building) structures:
  - Assignment

Syntax: varName.fieldName = value;
Example:
```
myStudent.name = 'izhar';
myStudent.exGrades = [57 67 …];
myStudent.projectGrade = 99;
```

  - The function *struct*

Syntax: varName = struct('field1',val1,'field2',val2,…);
Example
```
myStudent = struct('name','izhar','exGrades',[56 67 …],'projectGrade',99)

myStudent =
        name: 'izhar'
    exGrades: [56 67 …]
 projectGrade: 99
```


## Accessing fields of a structure

- The field names are accessed by using the dot (.) separator.
- Example:

```
>> a = myStudent.name
a =
   'izhar'

>> myStudent.fairGrades=myStudent.exGrades+1;
```


## Arrays of structures

- Structures may serve as elements within an array.
- All the structures within the array must be of the same **type** (have the same number and name of fields).
- Example:

```
>> student(1).name = 'izhar';
>> student(2).name = 'yaara';
>> student(1).projectGrade = 99;
```

In this case student(2).projectGrade will exist and be []


## Adding and removing fields

- Adding a field is performed by assignment.
- Removing a field is done using *rmfield* function

Syntax: s = rmfield(s, 'fieldname');

Example:
student = rmfield(student,'projectGrade');

- Other accessory functions include:

**fieldnames setfield getfield orderfields**


## Nested structures

- A field of a structure does not have to be a simple data type. It may be a structure by itself.
- Example:

```
student(1).name.first = 'Izhar';
student(1).name.last = 'Bar-Gad';
```

- The same is true for arrays of structures of arrays of structures…


## Cell arrays

- Cell arrays are multidimensional arrays whose elements are other arrays.
- Cell arrays are created either by the **cell** function or by enclosing a miscellaneous collection of things in curly braces, {}.
- Example:

a = [1 2;3 4];  b='izhar';  c=3.14;
myCellArray = {a b c}

```
myCellArray =
    [2x2 double]    'izhar'    [3.14000000000000]
```


## Elements of a cell array

- Each element of a cell array is a different type of array. Access (both assignment and retrieval) to the elements is done using {} and not ().
- Example:  myCellArray{1} = [1 2 3;4 5 6];
- Cell arrays overcome the constraint on equal sized dimensions.
- Example:  myCellArray = {'tal' 'yossi' 'shmuel'}


## Nesting cell arrays

- Cell arrays may be nested in other regular, structure or cell arrays.
- Example:

```
firstCellArray = {'izhar' [1 2;3 4]};
secondCellArray = {1 2 'yaara'};
regularArray = [5 6 7];
bigArray = {firstCellArray regularArray 3 secondCellArray}
bigArray =
    {1x2 cell}    [1x3 double]    3    {1x3 cell}

myValue = bigArray{1}{2}(2,2)        →        4
```

## Cell arrays conversion

- *cell2mat* & *mat2cell* convert between cell arrays and regular arrays. They work only on cell arrays in which all the elements have the same size.
- Example

```
>> a=[1 2;3 4]; b=[5 6;7 8]
>> c={a b}
c =
    [2x2 double]    [2x2 double]
>> d=cell2mat(c)
d =
     1     2     5     6
     3     4     7     8
```

IBG

## Cell arrays vs. Structure arrays

- Cell and structure arrays are useful for organizing data that consists of different sizes or kinds of data.
- Structures are useful mainly when:
  - The number and type of data elements is fixed.
  - The data has a meaningful or logical organization.
- Cell arrays are useful mainly when:
  - Access is needed to multiple fields of data simultaneously.
  - There is no fixed and meaningful set of field names.
  - Fields are routinely added and removed.

IBG

## Sparse matrices I

- It is common to have matrices with a very large number of zero values.

  For example: a matrix describing a patient in each row and a disorder in ach column. In the matrix there is a numerical value for the severity. Most patients do not suffer at all from most disorders.
- The problem is that these zeros occupy space and sometimes require computing resources.
- Unlike the traditional **full** matrix, a **sparse** array holds **only** the non-zero elements.

IBG

## Sparse matrices II

- Only the non zero elements are kept in memory in the format:

(xIndex, yIndex) Value

- Sparse matrices do not exist for high dimensional array.

IBG

## Sparse matrices examples

- Assuming a 100*100 matrix of values, in which each element is the standard MATLAB variable (also called double) occupies 8 bytes.
- Assuming that only 1% of the element have non-zero values. How much memory will a full / sparse matrix use?
- Assuming that 100% of the element have non-zero values. How much memory will a full / sparse matrix use?
- Misc:
  - $2^{10}$ = 1024 bytes are also called kilobytes (KB)
  - $2^{20}$ = 1048576 bytes are also called megabytes (MB)
  - Note: K is not a thousand and M is not a million.
  - The index into the sparse matrix uses 4 bytes.
- Use sparse matrices only for low % of non-zero elements !

IBG

## Working with sparse matrices

- Creating a sparse matrix

Syntax: mat = sparse(xSize,ySize);

- Convert a full matrix → sparse matrix

Syntax: sparseMat = sparse(fullMat);

- Convert a sparse matrix → full matrix

Syntax: fullMat = full(sparseMat);

- Many functions work exactly the same for full and sparse matrices.

(Varies across MATLAB versions)

IBG

## Multi-dimensional arrays

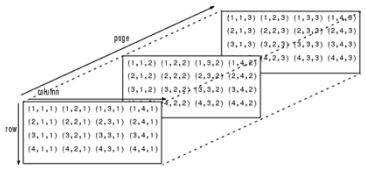
- In MATLAB, an array having more than two dimensions is called a **multidimensional** array.
- Example: Storing the EEG recording of length *k* samples of *n* patients of *m* sessions is classically performed in a (n, m, k) 3D array.
- Many of the operations that are performed on matrices (i.e., two-dimensional arrays) may also be done on multidimensional arrays. However, some functions will work correctly only on 1D (vectors) or 2D (matrices) arrays.

(Varies across MATLAB versions)

IBG

## Multi-dimensional arrays - example

- Example of a 3 dimensional array of a size (4,4,3) i.e. 4 rows, 4 columns, 3 pages.




- Higher dimensional arrays are even more problematic for visualization.

IBG

## Creating multidimensional arrays

- Generating a structured multidimensional array
  - Using: rand, randn, zeros, ones
  - Example: rand(2,3,4,5);
- Multiple 2D arrays:
  - A(:,:,1) = [1 2; 3 4; 5 6];
  - A(:,:,2) = [12 13; 14 15; 16 17];
- The “cat” command
  - B = cat( 3, [1 2; 3 4], [5 6; 7 8])

IBG

## Multi-dimensional arrays - usage

- Finding the size of the array

Syntax: arraySize = size(array); → [dim1 dim2 dim3 …]

- Finding the number of dimensions

Syntax: numDims = ndims(array);

- Reducing the dimension by removing **singleton** (size 1) dimensions.

Syntax: lowDimArray = squeeze(highDimArray);

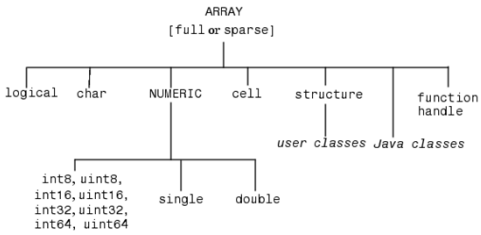
Example: getting a matrix which is the first column of a 4*4*4 3D array:

```
highDim = rand(4,4,4);
lowDim = squeeze(highDim(:,1,:));
```

IBG

## Data types in MATLAB

- There are 15 basic data types in MATLAB




IBG

## Numeric data types

- The basic data types are:
  - Logical
    - 0/1 (True/False) values.
    - Uses 1 byte per element.
  - Character
    - Elements of strings.
    - Uses 2 bytes per element.
  - Integer – whole numbers (-min, .., -1, 0, 1, …, max)
    - Signed or Unsigned
    - Uses 1-8 bytes per element.
    - Uses 2-16 for complex elements.
  - Floating point – real numbers
    - Single – Uses 4 bytes per element (8 for complex)
    - Double – Uses 8 bytes per element (16 for complex)
- The default data type is **double**.

IBG


## Functions on different data types

- Many functions and arithmetic operators work only on some of the numerical data types.
- Conversion is performed using functions with the data type name.
- The main reasons to use non-double data types is for:
  - Saving memory space (up to a factor of 8).
  - Saving time (most CPUs perform integer calculations faster than floating point calc).
  - Avoid precision issues (for example equality).


## Perspective

- There is no need to remember the various numeric data types, they are not used very often, but you need to know that they exist.
- Structures and cells are very useful and most modern MATLAB programs are based on them.
- Multidimensional and sparse arrays a less common but are still very useful for some cases of data organization.