E7: Introduction to Computer Programming for Scientists and Engineers
University of California at Berkeley, Spring 2017
Instructor: Lucas A. J. Bastien

Diary for lecture 04: Arrays
Version: release

```
% This document presents and illustrates concepts related to:
%
% - Creating arrays
% - Indexing (i.e. accessing and updating specific values in an array)
% - Arithmetic operations on arrays
% - Logical arrays and logical indexing
% - Useful functions related to arrays
%
% Notes: In this document, I do not use nor discuss arrays that have more
% than two dimensions. Most of the concepts presented below can, however,
% be extended to arrays that have three or more dimensions

%%%%%%%%%%%%%%%%%%%
% Creating arrays %
%%%%%%%%%%%%%%%%%%%

% Use square brackets to create arrays by specifying each individual value
% - use commas to separate values in a given column
% - use semi-colons to separate columns
>> a = [1, 4, 5; 7, 2, 6]

a =

     1     4     5
     7     2     6

% Spaces alone can be used to separate values in a given column, but using
% commas probably makes the code more readable
>> a = [1 4 5; 7 2 6]

a =

     1     4     5
     7     2     6

% Use semi-colons to concatenate arrays vertically (since this process is
% similar to adding rows to an existing array). The arrays being
% concatenated must have the same number of columns
>> b = [0, 7, 2; 5, 6, 8; 10, 1, 0]

b =
```

```
     0     7     2
     5     6     8
    10     1     0

>> result = [a; b]

result =

     1     4     5
     7     2     6
     0     7     2
     5     6     8
    10     1     0

% Use commas to concatenate arrays horizontally (since this process is
% similar to adding columns to an existing array). The arrays being
% concatenated must have the same number of rows
>> c = [7, 4, 9, 0, 2; 2, 10, 4, 0, 1]

c =

     7     4     9     0     2
     2    10     4     0     1

>> result = [a, c]

result =

     1     4     5     7     4     9     0     2
     7     2     6     2    10     4     0     1

% Inquire the size of an array using the function "size". for a 2-D array,
% it returns the number of rows and columns
>> size(a)

ans =

     2     3

>> size(b)

ans =

     3     3

>> size(c)

ans =

     2     5
```

```
% Note: an array that has n rows and m columns will be called an "n by m
% array"

% A 1 by n array is also called a row vector. For example:
>> row_vector = [4, 5, 1, 6, 8, 0]

row_vector =

     4     5     1     6     8     0

>> size(row_vector)

ans =

     1     6

% A n by 1 array is also called a column vector. For example:
>> column_vector = [4; 5; 1; 6; 8; 0]

column_vector =

     4
     5
     1
     6
     8
     0

>> size(column_vector)

ans =

     6     1

% The arrays that we have been defining so far are of class double
>> class(a)

ans =

double

>> class(row_vector)

ans =

double

>> class(column_vector)

ans =

double
```

```
% A scalar is also a 1 by 1 array
>> size(10)

ans =

     1     1

% The colon operator creates row vectors of equally spaced values
% -> the default spacing is one
>> a = 1:4

a =

     1     2     3     4

>> size(a)

ans =

     1     4

% -> you can change the default spacing
>> a = 1:0.5:4

a =

    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000

>> a = 4:-1:1

a =

     4     3     2     1

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Access values in arrays: indexing %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Use parentheses to access values in arrays by specifying the row index
% and column index of the value that you want to access. This process is
% called "indexing". Index counting starts at 1 (in some programming
% languages, such as C and Python, index counting starts at 0)
>> a = [1, 4, 5, 10, 9; 7, 2, 6, 11, 0]

a =

     1     4     5    10     9
     7     2     6    11     0

% -> access the value in the first row and second column
>> a(1, 2)
```

```
ans =

     4

% -> access the value in the second row and third column
>> a(2, 3)

ans =

     6

% Use the keyword "end" to index counting backward from the last element
% along a given dimension (by dimension, I mean either "along a row" or
% "along a column")
% -> access the value in the second row and last column
>> a(2, end)

ans =

     0

% -> access the value in the last row and the one-before-last column
>> a(end, end-1)

ans =

    11

% Access a range of values using the column operator
% -> access all values from the first row and second to fourth column
>> a(1, 2:4)

ans =

     4     5    10

% More generally, acces several values at once using an array as an index
% -> access values from the first row and second and fourth column
>> a(1, [2, 4])

ans =

     4    10

% Access all values along a given dimension
% -> access all values of the first row
>> a(1, :)

ans =

     1     4     5    10     9
```

```
% -> access all values of the second column
>> a(:, 2)

ans =

     4
     2

% Vectors, whether row- or colum-vectors, can be indexed using a single
% index
>> row_vector = [4, 5, 1, 6, 8, 0]

row_vector =

     4     5     1     6     8     0

>> column_vector = [4; 5; 1; 6; 8; 0]

column_vector =

     4
     5
     1
     6
     8
     0

% -> access the second value of each vector
>> row_vector(2)

ans =

     5

>> column_vector(2)

ans =

     5

% -> access the fourth value of each vector
>> row_vector(4)

ans =

     6

>> column_vector(4)

ans =
```

```
     6

% Consider an n by m array, where n > 1 and m > 1. One can still access
% values in such an array using a single index. This process is called
% "linear indexing". In this case, Matlab treats the array as a vector, by
% stacking its columns on top of each other (the left-most column first,
% the second left-most column  second, ..., the right-most column last)
>> a

a =

     1     4     5    10     9
     7     2     6    11     0

>> a(1)

ans =

     1

>> a(2)

ans =

     7

>> a(3)

ans =

     4

>> a(4)

ans =

     2

>> a(5)

ans =

     5

>> a(6)

ans =

     6

>> a(7)
```

```
ans =

    10

>> a(8)

ans =

    11

>> a(9)

ans =

     9

>> a(10)

ans =

     0

% Trying to access a value outside of an array results in Matlab throwing
% the error "Index exceeds matrix dimensions." For example:
>> a(100, 100)

Index exceeds matrix dimensions.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Replacing values in arrays %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Use indexing and the assignment operator to replace value(s) in an array
% -> replace the value in row 2 and column 4 by the value 20
>> a(2, 4) = 20

a =

     1     4     5    10     9
     7     2     6    20     0

% Setting a value outside of the array makes Matlab resize the array in
% order to accomodate the new value, setting all missing values to 0. This
% behavior is dangerous!
>> a

a =

     1     4     5    10     9
     7     2     6    20     0

>> size(a)
```

```
ans =

     2     5

>> a(4, 1) = 30

a =

     1     4     5    10     9
     7     2     6    20     0
     0     0     0     0     0
    30     0     0     0     0

>> size(a)

ans =

     4     5

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Arithmetic operations on arrays %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Use the following operators for element-wise operations:
% - addition: +
% - subtraction: -
% - multiplication: .*
% - division: ./
% - exponentiation: .^
>> a = [2, 4, 1; 5, 0, -2]

a =

     2     4     1
     5     0    -2

>> b = [10, 2, 7; 9, 1, 3]

b =

    10     2     7
     9     1     3

>> a + b

ans =

    12     6     8
    14     1     1

>> a - b
```

```
ans =

    -8     2    -6
    -4    -1    -5

>> a .* b

ans =

    20     8     7
    45     0    -6

>> a ./ b

ans =

    0.2000    2.0000    0.1429
    0.5556         0   -0.6667

>> a .^ b

ans =

        1024          16           1
     1953125           0          -8

% The two arrays must have the same size! If not, Matlab throws the error
% "Matrix dimensions must agree.". For example:
>> [2, 4; 5, 7] .* [1, 4; 3, 0; 9, 5]

Matrix dimensions must agree.

% Note: the *, /, and ^ operators between two non-scalar arrays correspond
% to matrix operations, not to element-wise operations. We will talk about
% matrix operations next week

% The element-wise operators also work on operations between a scalar and
% an array
>> a + 2

ans =

     4     6     3
     7     2     0

>> 2 + a

ans =

     4     6     3
     7     2     0
```

```
>> a - 2

ans =

     0     2    -1
     3    -2    -4

>> 2 - a

ans =

     0    -2     1
    -3     2     4

>> a .* 2

ans =

     4     8     2
    10     0    -4

>> 2 .* a

ans =

     4     8     2
    10     0    -4

>> a ./ 2

ans =

    1.0000    2.0000    0.5000
    2.5000         0   -1.0000

>> 2 ./ a


    1.0000    0.5000    2.0000
    0.4000       Inf   -1.0000

>> a .ˆ 2

ans =

     4    16     1
    25     0     4

>> 2 .ˆ a

ans =
```

```
    4.0000   16.0000    2.0000
   32.0000    1.0000    0.2500

% Note: since a scalar is also a 1 by 1 array, these operators also work
% between scalars

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Logical arrays and logical indexing %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% You can create logical arrays using a syntax similar to the syntax used
% when creating arrays of class double. For example:
>> logical_array = [true, false, false; true, true, false]

logical_array =

  2x3 logical array

   1   0   0
   1   1   0

>> class(logical_array)

ans =

logical

% The "not" operator works element-wise on the entire array
>> ~logical_array

ans =

  2x3 logical array

   0   1   1
   0   0   1

% Often, logical arrays are created by using arrays of class double and
% relational operators
% -> either using element-wise operations between two non-scalar arrays,
%    for example:
>> a = [2, 4, 5; -2, 0, 9]

a =

     2     4     5
    -2     0     9

>> b = [-10, 10, 2; 6, 4, 9]

b =
```

```
   -10    10     2
     6     4     9

>> a == b

ans =

  2x3 logical array

   0   0   0
   0   0   1

>> a ~= b

ans =

  2x3 logical array

   1   1   1
   1   1   0

>> a > b

ans =

  2x3 logical array

   1   0   1
   0   0   0

>> a >= b

ans =

  2x3 logical array

   1   0   1
   0   0   1

>> a < b

ans =

  2x3 logical array

   0   1   0
   1   1   0

>> a <= b

ans =
```

```
  2x3 logical array

   0   1   0
   1   1   1

% -> or using relational operations between an array and a scalar, for
%    example:
>> a == 2

ans =

  2x3 logical array

   1   0   0
   0   0   0

>> a ~= 2

ans =

  2x3 logical array

   0   1   1
   1   1   1

>> a > 2

ans =

  2x3 logical array

   0   1   1
   0   0   1

>> a >= 2

ans =

  2x3 logical array

   1   1   1
   0   0   1

>> a < 2

ans =

  2x3 logical array

   0   0   0
   1   1   0
```

```
>> a <= 2

ans =

  2x3 logical array

   1   0   0
   1   1   0

% "Logical indexing" is a very powerful concept. It consists of using a
% logical array as the index when indexing another array (the latter is
% often of class double). If "a" is an array and "logical_array" is an
% array of class logical, then "a(logical_array)" returns (as a column
% vector) all values of "a" located in places where logical_array is true
>> a

a =

     2     4     5
    -2     0     9

>> logical_array

logical_array =

  2x3 logical array

   1   0   0
   1   1   0

>> a(logical_array)

ans =

     2
    -2
     0

% For example, logical indexing can be used to select all strictly positive
% values in an array
>> a = [2, -4, 5; -2, 0, 9]

a =

     2    -4     5
    -2     0     9

>> logical_index = (a > 0)

logical_index =
```

```
  2x3 logical array

   1   0   1
   0   0   1

>> positive_values_of_a = a(logical_index)

positive_values_of_a =

     2
     5
     9

% This procedure can be done in one line
>> a(a > 0)

ans =

     2
     5
     9

% Logical indexing can be used to replace values in an array
% -> replace all positive values in array "a" by zero
>> a(a > 0) = 0

a =

     0    -4     0
    -2     0     0

% The logical operators & (and) and | (or) act element-wise on arrays
>> a = [2, 4, 5; -2, 0, 9]

a =

     2     4     5
    -2     0     9

>> b = [-10, 10, 2; 6, 4, 9]

b =

   -10    10     2
     6     4     9

% -> | (or)
>> a > 0 | a > b

ans =

  2x3 logical array
```

```
   1   1   1
   0   0   1

% -> & (and)
>> a > 0 & a > b

ans =

  2x3 logical array

   1   0   1
   0   0   0

% When used in arithmetic expressions, logical arrays are converted to
% class double
% -> create an array similar to "a" but with all positive values replaced by 0
>> a .* (a <= 0)

ans =

     0     0     0
    -2     0     0

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Useful functions related to arrays %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Here is a tentative list of useful functions related to arrays:
% - size: get the size of an array
% - numel: get the number of elements of an array
% - zeros: create an array full of zeros
% - ones: create an array full of ones
% - isequal: are two arrays equal?
% - isnan: check whether elements are NaN
% - isinf: check whether elements are infinite (Inf or -Inf)
% - min: get the minimum value**
% - max: get the maximum value**
% - mean: mean of the values**
% - sum: sum of the values**
% - any: true if at least one value is non-zero** (often used on logical
%        arrays)
% - all: true if all values are non-zero** (often used on logical arrays)
%
% The functions marked with ** act in a (perhaps) surprising way:
%
% If their input argument is a vector (whether row- or column-vector), the
% function acts on the entire vector. If the input argument is an n by m
% array with n > 1 and m > 1, then the function acts on each column of the
% array separately, and returns the results in a row vector. Call these
% functions twice to get the relevant information about the entire array:
% for example sum(sum(array))
```

```
%
% Below are examples of the use of these function

>> a = [5, -7, 1, 0, 10, 4; 1, -1, -5, 7, 0, 9; 7, 10, -3, 8, 20, 4]

a =

     5    -7     1     0    10     4
     1    -1    -5     7     0     9
     7    10    -3     8    20     4

>> size(a)

ans =

     3     6

% Calculate the number of elements in the array a
>> numel(a)

ans =

    18

% Create a 4 by 4 array full of zeros
>> zeros(4)

ans =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0

% or
>> zeros(4, 4)

ans =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0

% Create a 2 by 3 array full of zeros
>> zeros(2, 3)

ans =

     0     0     0
     0     0     0
```

```
% The use of the function "ones" is very similar to the use of the function
% "zeros", so only one example is shown here
% -> create a 2 by 3 array full of ones
>> ones(2, 3)

ans =

     1     1     1
     1     1     1

% Check whether two arrays are equal
>> isequal([2, 4; 5, 6], [3, 2])

ans =

  logical

   0

>> isequal([2, 4; 5, 6], [2, 4; 5, 6])

ans =

  logical

   1

% Check whether elements are NaN
>> b = [3, 5, 6, NaN; Inf, -Inf, 0, 2]

b =

     3     5     6   NaN
   Inf  -Inf     0     2

>> isnan(b)

ans =

  2x4 logical array

   0   0   0   1
   0   0   0   0

% Check whether elements are infinite (Inf or -Inf)
>> isinf(b)

ans =

  2x4 logical array

   0   0   0   0
```

```
   1   1   0   0

% The following functions act column-by-column
>> a = [5, -7, 1, 0, 10, 4; 1, -1, -5, 7, 0, 9; 7, 10, -3, 8, 20, 4]

a =

     5    -7     1     0    10     4
     1    -1    -5     7     0     9
     7    10    -3     8    20     4

>> min(a)

ans =

     1    -7    -5     0     0     4

>> max(a)

ans =

     7    10     1     8    20     9

>> sum(a)

ans =

    13     2    -7    15    30    17

>> mean(a)

ans =

    4.3333    0.6667   -2.3333    5.0000   10.0000    5.6667

>> any(a > 0)

ans =

  1x6 logical array

   1   1   1   1   1   1

>> all(a > 0)

ans =

  1x6 logical array

   1   0   0   0   0   1

% Call these functions twice to get the relevant information about the
```

```
% entire array
>> min(min(a))

ans =

    -7

>> max(max(a))

ans =

    20

>> sum(sum(a))

ans =

    70

>> mean(mean(a))

ans =

    3.8889

>> any(any(a > 0))


ans =

  logical

   1

>> all(all(a > 0))

ans =

  logical

   0

% Functions such as "sum" also work on logical arrays
>> a > 0

ans =

  3x6 logical array

   1   0   1   0   1   1
   1   0   0   1   0   1
   1   1   0   1   1   1
```

```
>> sum(a > 0)

ans =

     3     1     1     2     2     3
```