Introduction to *Matlab/Octave*

October 18, 2016

This document is designed as a quick introduction for those of you who have never used the *Matlab/Octave* language, as well as those of you who have used it before, but would like to brush up your knowledge and learn new tricks. A prerequisite is general knowledge of procedural programming languages.

Matlab is a language and an environment for numerical computation. Matlab is proprietary software, however there is an open-source alternative called Octave which in most cases mimics the behavior of Matlab and can therefore be used as its substitute.

The main focus and advantage of Matlab is the concept of matrix (as multi-dimensional arrays). Matrices can be manipulated in a way that is in other languages reserved only for primitive types. We will show more examples later. Here are some other important features that you should know about:

(a) Comments are prefixed with symbol `%`.

(b) If a function returns a result and a name of the destination variable is not given, most of the functions save the result to a variable `ans`.

(c) Unlike most of the programming languages, when addressing multi-dimensional arrays, the first index defines the row and the second index defines the column.

(d) Unlike most of the (low-level) programming languages, the addressing of an array starts with 1 and not with 0. This can cause a lot of confusion to people, who are accustomed to C or Java.

1 Integrated documentation

For informations about integrated functions in the *Matlab/Octave* environment you can use expression `help <topic>`. Example: `help image` displays information regarding function `image`, while `help +` displays information regarding operator `+`.

2 Variables, vectors, matrices

(a) A variable value is defined with an expression `a = <value>`. Variables do not have to be explicitly defined in advance.

(b) Supported primitive types are (we only list the most important): logical, int (more variants, like uint8, uint32, int32), double as well as matrices of types these types.

(c) There is no string type, instead we have a vector of characters.

(d) A vector is a matrix of size $1 \times N$

(e) Most common numerical types operators (work on matrices as well): `+`, `-`, `*`, `/`, `^`

   (a) `A + B` and `A - B` element-wise addition/subtraction with two same-size matrices

   (b) `A * B` matrix multiplication of $M \times N$ and $N \times K$ matrices, resulting matrix is of size $M \times K$

   (c) `A / B` solve a system of equations `A x = B`

   (d) `A .* B` and `A ./ B` element-wise multiplication/division with two same-size matrices

   (e) `A ^ b` power operator, `A .^ b` element-wise power operator

   (f) `A'` matrix transpose

(f) Logical operators: `<`, `>`, `<=`, `>=`, `==`, `<>`, `&`, `|`, `~` (work also with matrices in element-wise mode). Operators `&&` and `||` can only be used for scalar logical operands.

(g) Defining a matrix

   (a) `A = [ 1, 2, 3, 4 ]` - vector 1×4

   (b) `A = [ 1, 2; 3, 4 ]` - matrix 2×2

   (c) `A = zeros(2, 2)` - matrix 2×2 - all elements are 0

   (d) `A = ones(2, 2)` - matrix 2×2 - all elements are 1

   (e) `A = rand(2, 2)` - matrix 2×2 - random values between 0 and 1

(h) Element access

   Different than most other programming languages! First index defines a row, second one defines column. Indices start with 1.

   (a) `A(1, 5)` - value in first line and fifth column

   (b) `A(1:3, 3:6)` - sub-matrix between first and third line and third and sixth column.

   (c) `A(3:end, :)` - sub-matrix between the third and last line and all the columns

   (d) `A(end:-1:1, :)` - returns a matrix with a reversed order of rows

   (e) `A(1:2:10, :)` - returns a matrix with only each odd row from the first to tenth

   (f) `A([1 4 3 2], :)` - explicitly specify row indices

   (g) `A(logical([1 0 0 1]), :)` - for a matrix with four rows only return the rows where the corresponding element in a mask is 1 (true)

   (h) `A(A > 5)` - returns a vector of elements in A that are greater than 5

   (i) `A(A > 5) = 0` - sets the elements in A that are greater than 5 to 0

3 Displaying values

(a) If an assignment statement does not end with a semicolon, the result of the assignment operation will be displayed in the terminal.

```
 ?> a = 1
a = 1
```

(b) Arbitrary value can be displayed using the `disp` function:

  (a) `disp(1)` displays `1`
  (b) `disp('foo')` displays `foo`
  (c) `disp(A)` displays the content of the variable `A`

(c) For more flexible output we can also use functions `fprintf` and `sprintf` that behave similarly to their counterparts in C.

`sprintf('Value of variable a is %d\n', a)` - formates the sequence by inserting the actual value of variable `a` (if the value of variable is integer).

4 Flow-control

The language knows two types of loops (`for` and `while`) and `if` and `switch` conditional statements.

4.1 `for` loop

```
for n = <sequence>

   <code>

end
```

As a sequence we can use the sequence notation (`<start>:<step>:<finish>`) or a vector of elements.

4.2 `while` loop

```
while <condition>

   <code>

end
```

4.3 `if` statement

```
if <condition>

   <code>

end
```

We can also use `else` alternative:

```
if <condition>

   <code>

else

   <code>

end
```

4.4 `switch` statement

```
switch <expression>
   case <case1>

      <code>

   case <case2>

      <code>

   otherwise

      <code>

end
```

In contrast to the C switch statement we do not need an explicit break statement for each case. Strings can also be used as case comparison values.

5 Functions and script files

More complex programs are organized in functions that are saved in files with the same name and a suffix `.m`. A function file can contain multiple functions, however only one is publicly visible (others can only be called within the file scope).

Function definition:

```
function [<output1>, <output2>, ...] = <function name> (<input1>, <input2>, ...)

   <code>

end
```

(a) Global variables can be defined using the `global <variable name>` expression.

(b) Functions can return multiple results. In this case we have to specify multiple variables when calling such a function (e.g. `[s, i] = sort(a);`).

6 Image processing

Images are represented as matrices. Note again, that the first two dimensions in matrices are reversed. A color image is therefore represented as a three-dimensional matrix of size $height \times width \times channels$.

(a) To read/write an image you can use functions `imread` in `imwrite`.

    (a) `[IMG, MAP, ALPHA] = imread (FILENAME)`

    (b) `imwrite (IMG, [MAP], FILENAME, FMT, P1, V1, ...)`

(b) To display a matrix as an image we can use functions `image`, `imagesc`, `imshow`.

    (a) `image` displays an image in a figure

    (b) `imagesc` displays an image by adjusting the effective range to the minimum and maximum value in the data

    (c) `imshow` displays an image without coordinate axes

    (d) When displaying arbitrary data as an image we have to know the type of the data. If a matrix contains numbers of type double, then the values are displayed with with effective range 0 to 1. If a matrix contains values of type uint8 the effective range is 0 to 255.

(c) Grayscale images are displayed using color indexing. Using the `colormap` command we can set an arbitrary color mapping.

    (a) Some pre-defined color palettes are `gray`, `jet` in `bone` (e.g. `colormap jet`).

    (b) We can also define a custom color map as a matrix of size $N \times 3$.

(d) Image size can be determined with the `size` function

    (a) `[x y c] = size(A)` - size result as multiple variables

    (b) `s = size(A)` - size result as a vector

(e) **Octave-specific notice**: Octave supports multiple plotting toolkits, some of which may be unstable on your platform. To see the current toolkit use `graphics_toolkit` function, to see the list of available toolkits use `available_graphics_toolkits`. You can change the toolkit by typing e.g. `graphics_toolkit('gnuplot')`, but you will have to do it before you open any figures (otherwise you will have to restart Octave).

7 Data structures

Besides basic data types and matrices *Matlab/Octave* also knows structures and cell arrays that are more generic and can be used as general purpose containers.

Structure is a dictionary that contains named fields of arbitrary type. It is initialized with `struct` function.

```
a = struct('field1', 1, 'field2', [1, 3; 5, 8], 'field3', 'string', 'field4', struct('field', 0.5));

a.field1 = a.field1 + 5;
```

Cell array is a multidimensional array that can store arbitrary data type in each cell. Because of this, cell arrays cannot be used with arithmetic operators, however, we can use many of the the flexible addressing options that work with matrices.

```
c = cell(2, 2); % 2 x 2 cell array

c{1, 1} = 4;
c{2, 1} = [4, 5, 4, 1];
c{1, 2} = 'Cell array';


c1 = c(:, 1); % only first row
b = c{1, 1}; % when accessing elements use curly brackets (otherwise you will get a 1 x 1 cell array)
```

8 Advanced vector operations

Matlab/Octave is optimized for working with matrices, therefore all the matrix operations are very fast. As a downside of this, other explicit scripted statements can get a bit slow is used extensively (e.g. loops over large arrays) as the scripting language has to do all sorts of type checks and similar safety operations.

It is recommended that more complex operations are translated to matrix problems, even if the translation is not very intuitive. Sometimes this translations require some effort, however, as more and more problems are being ported to graphics cards and multi-core systems, that excel at matrix operations, this kind of mental model is becoming more and more important.

Matlab/Octave knows several functions that are helpful when working with large arrays efficiently. Here we only mention a few more common ones (many of those have more use cases so it is best to consult the documentation before using them):

(a) `linspace(a, b, n)` - generate a linear sequence of `n` numbers from `a` to `b`.

(b) `repmat(A, m, n)` - repeats a matrix in multiple dimensions

(c) `reshape(A, m, n)` - reshapes a matrix to new dimensions (but same number of elements)

(d) `bsxfun(fun, A, B)` - a very general function to perform various operations that involve two matrices, e.g. can be used to perform a function on subtract vector `B` from each row in `A`.

(e) `cellfun(fun, C)` - performs a function on every cell in cell array

(f) `accumarray(I, V, dims)` - accumulator array: creates an accumulator matrix of given size and uses indices in matrix `I` to aggregate values from `V` to appropriate cells in the accumulator

(g) `ind2sub(dims, I)` - convert linear indices to subscripts (multi-dimensional indices) based on the specified size in each dimension

(h) `sub2ind(dims, D1, D2, ...)` - inverse operation to `ind2sub`

(i) `squeeze(A)` - removes singleton dimensions

(j) `shiftdim(A, n)` - shifts dimensions of the matrix

9 Using MEX to speed up your program

Not all tasks can be effectively written in form of vector operations and in some cases we have to resort to native implementations. For this *Matlab/Octave* provides an easy technology called MEX, which is essentially a specialized wrapper for a native compiler that creates specialized native binaries. A similar concept can be found in Octave.

Every native function is a C/C++ file that has to contain a special entry function `mexFunction` that accepts an array of output and an array of input arguments. The simplest file will therefore look like this.

```
#include "mex.h" // include the Matlab header like this

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

... the code ...

}
```

The input arguments in the code above are as follows: `nlhs` denotes the number of left-side (output) arguments, `plhs` denotes an array of left-side arguments, `nrhs` is the number of the right-side (output) arguments and `prhs` is the array that contains them.

The obvious advantage of the MEX technology is the performance gain, however, there are some disadvantages as well. Firstly, there is quite a lot of boilerplate code as native programs are statically-typed and have to check every parameter that comes from the Matlab scripting environment. One also has to check if there are enough output arguments available. One also has to be very careful when writing the native code as any fatal mistakes such as segmentation errors will likely result in termination of the entire Matlab environment which is really inconvinient.

10 Additional resources

(a) Comprehensive introduction to *Matlab/Octave* in PDF format

(b) A list of Matlab tutorials

(c) Introduction to Octave