Bi7740: Scientific computing

MATLAB remider

Vlad Popovici
popovici@iba.muni.cz

Institute of Biostatistics and Analyses
Masaryk University, Brno

March 11, 2014

Outline

Help and paths

- `doc`, `help`, `helpdesk`, `helpwin`, `lookfor`: different ways of obtaining help
- `docopt`: (on UNIX) path where the help files reside
- `addpath`, `path`, `pathtool`, `rmpath`: manage search paths for functions
- `lasterr`, `lastwarn`: last error and warning
- `type`, `what`, `which`: list a file, list a directory, locate functions and files

Creating arrays

- `var = [val1 val2 ...]` row vector from values
- `var = [val1; val2; ...]` column vector from values; equivalent to `var = [val1 val2 ...]'` (`'` is the transposition operator)
- equally spaced data: `<low>:<step>:<high>`; is <step> is 1, omit `:<step>`. Example: `1:2:10`, `1:10`, `1.5:0.1:2.1`. last value might not be in the generated sequence
- decreasing sequences: as above, with negative <step>: `20:-5:0`
- equally spacing of an interval: `linspace(<first>,<last>,<number_of_values>)` `<first>` and `<last>` values will be in the sequence
- logarithmically equally space points: `logspace`

Creating 2D arrays - matrices

- `var = [a11, a12,...; a21, a22,...; ...]` from values
- you can use vector generators, as before, for the rows of the matrix
- `zeros(m,n)`, `ones(m,n)`, `eye(m)`: all-zeros, all-ones, or identity matrix with m rows and n columns
- *transpose* of X: `X'`

Arrays dimension

- `size(a), size(b), size(b,2)`
- `length(a)`: for vectors
- `numel(b)`: total number of elements

Addressing the elements of an array

- by index: `a = 1:3:10; a(1)` and `b = [1:4;5:8]; b(1,2)`
- by index vectors: `a(1:2)`, `b(1,:)`, `b(2,2:end)`: end gives the last index in the corresponding dimension of an array
- all elements as a row vector: `a(:)`, `b(:)`: column-by-column ordering of matrix elements (column-major)
- you can use single index for n-dimensional arrays too! Pay attention to ordering of elements: `b(1:5)`

Extending/reducing arrays

- arrays are dynamical structures, you can add a new value beyond the dimension of the array: `b(3,3)= 13`: MATLAB adds 0s to fill-in to the required new dimension
- this operation is expensive: better pre-allocate the array!
- concatenation: `[a a]`, `[a;a]`: the arrays must be consistent
- general concatenation: `cat(...)`
- delete elements: replace them by empty arrays: `c = b; c(3,:)= []`. You have to keep the shape of the array!

(Re-)shaping the array

- `reshape(b, <new_extent_specification>)`. The total number of elements must not change!
  `reshape(b, [1,2,2,3])`

Strings

- `s = 'ab cd ef'`
- `s = char('aaa', 'bb', 'c')` rows are the strings given

Basic operators

The arrays must have conforming shapes/sizes, depending on the operation.

- `+`, `-`, `*`, `^` classical matrix operators (addition, subtraction, multiplication, power)
- `/`: matrix division, related to the inverse (we discussed)
- `.*`, `.^`, `./` element-wise operations
- scalars are multicast: `[1 2 3; 4 5 6] * 2` and operation is kept element-wise

Some basic built-in functions for arrays

Usually, the summaries are column-wise - see the help for details

- `min(b),min(b, 4), [m,n] = min(b) max(...)`
- `mean(...),median(...),std(...)`
- `sum(...),prod(...)`
- `dot(...,..)`: `dot(a,a)` is the same as `a*a'`

Random arrays

- `rand(m,n)`: random uniformly distributed numbers between 0 and 1, in the form of a matrix (or vector, or scalar, depending on m and n)
- `randperm(n)`: random permutation of `1:n` vector
- `randperm(n,k)` random selection of `k` elements out of `1:n`
- `randi(imax, m, n)`: random uniformly distributed integers between 1 and `imax`, in the form of a matrix (or vector, or scalar)
- `randn(...)`: normally distributed random numbers: try `hist(randn(1000, 1))`

Relational operators

Check `help precedence` for precedence of operators!

- relational operators: `<, >, =<, ≥, ==, ~=`
- the result is either logical 1 (true) or logical 0 (false) and has the shape of the operands (after bringing them to compatible shapes)
- example: `mod(1:10,5)< 3`
- logical results can be used to address elements of arrays: `a = randn(4); a(a < 0.5)` what happend?

Logical operators and functions

- AND: `x & y`, `and(x, y)`
- OR: `x | y`, `or(x, y)`
- NOT: `~x`, `not(x)`
- XOR: `xor(x,y)`
- `all(x)`, `any(x)`, `find(x)`

Conditional branching

```
if <conditional statement 1>
    <block 1>
[elseif <conditional statement 2>
    <block 2>]
    ..........
[else
    <block 3>]
end
```

One-liner: `if <cond> instruction; end`

```
switch <expression>
  case value1
      <block 1>
  case {value2, value3,...}
      <block 2>
      ..........
  otherwise
      <block n>
end
```

Loops

```
for iterator = sequence
  <block>
end

while <condition_is_true>
    <block>
end
```

- `continue` skips the rest of the block and jumps to the next iteration (if any)
- `break`: breaks out of the loop; if it is outside a loop, it terminates the script/function

Functions

One visible function per <function_name>.m file.

```
function [<output>] = <name>(<parameters>)
% document your function here

  % process arguments
  if nargin < ...
  ...
  end

   <block>

  % prepare the return values
  if nargout...
  ...
  end
end % or return: not mandatory, but a good practice
```

- usually, the variables created in the functions are local
- use `global <variable>` to declare that a variable is from the global environment (put `global`s at the top of the file)
- the file with the function must be in the search path of MATLAB

Anonymous functions

```
<function_name> = @(arguments) <expression>

cube = @(x) x.^3
```

- `<expression>` must be a single valid MATLAB expression
- you can use also variables from the context in which the anonymous function is defined
- anonymous functions can be used as parameters for other functions

Inline functions

```
<function_name> = inline('mathematical expression ...
    as a string')

square = inline('x .^ 2')
square(1:5)
quad = inline('x .^ 2 + y .^ 2 + dot(x,y)', 'x', 'y')
```

- if you do not provide explicit parameters, they are deduced from the expression and ordered alphabetically in the argument list
- you cannot use *i* and *j* as variables
- the inline functions can be used as arguments for other functions

Functions as arguments

`@function_name` gives you a function *handle*, that can be used for calling the function itself.

```
function y = funplot(F, x0, x1)
  x = linspace(x0, x1, 100);
  y = F(x);
  plot(x, y);
return

funplot(@cos, 0, 2*pi)
funplot(square, -3, 3)
funplot(@(x) x .^ 3, 0, 5)
```

Subfunctions

In a file "main_function.m":

```
function ...= main_function(...)
end

function ...= subfunction_1(...)
...
end

function ...= subfunction_2(...)
...
end
```

Nested functions

```
function ...= A(...)
  function ...= B(...)
  ...
  end
  ...
  function ...= C(...)
  ...
  end
  ...
end
```