An introduction to MATLAB

LIACS Natural Computing Group Leiden University
What is MATLAB?

• Stands for **MATrix LABoratory**
• Developed by The Mathworks, Inc. [http://www.mathworks.com](http://www.mathworks.com)
• An interactive, integrated environment
  – for numerical computations
  – for symbolic computations
  – for scientific visualizations
• It is a programming language!
Characteristics of MATLAB

• Programming language based on matrix notation
  – Slow (compared to Fortran or C) (it is an *interpreted language*)
  – Efficient when matrices are involved!
  – Automatic memory management
  – Intuitive, easy to use, and compact code
  – Shorter program development time
  – Can be converted into C code via MATLAB compiler

• Many application-specific toolboxes available
MATLAB Toolboxes

• Statistics
• Signal Processing
• Curve-Fitting
• Bioinformatics
• Financial
• Image Processing
• Neural Networks
• Databases
• …and many others!
MATLAB at LIACS

• Installed version: 7.14.0.739 (R2012a)

• Invoke at system prompt (under LINUX)
  1. run terminal
  2. type: pushd
     /vol/share/software/matlab/liacs/r2012b/bin;
     ./matlab;
Getting started (1)

Row vector:

\[
\begin{align*}
\texttt{>> x} &= \ [3, 6, 9] \\
\texttt{x} &= \begin{bmatrix} 3 & 6 & 9 \end{bmatrix}
\end{align*}
\]

or…

\[
\begin{align*}
\texttt{>> x} &= \ [3, 6, 9] \\
\texttt{x} &= \begin{bmatrix} 3 & 6 & 9 \end{bmatrix}
\end{align*}
\]

Column vector:

\[
\begin{align*}
\texttt{>> x} &= \ [3; 6; 9] \\
\texttt{x} &= \begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix}
\end{align*}
\]
Getting started (2)

A row vector of ones:
>> x = ones(1,3)
   x =
    1   1   1

A column vector of ones:
>> x = ones(3,1)
   x =
    1
    1
    1

A row vector of zeros:
>> x = zeros(1,3)
   x =
   0   0   0

A row vector of random numbers between 0 and 1:
>> x = rand(1,3)
   x =
   0.9501  0.2311  0.6068
**Inner product / outer product**

```matlab
>> x = [3,6,9] % a row vector
    x =
         3         6         9

>> y = [1;2;3] % a column vector
    y =
         1
         2
         3

>> x*y
    ans =
          42

>> y*x
    ans =
         3         6         9
         6        12        18
         9        18        27
```
Element-wise operations

Use the ‘.’ character for element-wise operations:

Element-wise division:

```matlab
>> x = [3, 6, 9]
>> y = [3, 3, 3]
>> x ./ y
ans =
    1     2     3
```

Element-wise multiplication:

```matlab
>> x = [3, 6, 9]
>> y = [3, 3, 3]
>> x .* y
ans =
    9    18    27
```
And then matrices

Use semicolons to separate the rows:

\[
A = \begin{bmatrix}
1 & 2 & 0 \\
2 & 5 & -1 \\
4 & 10 & -1
\end{bmatrix}
\]

And the transpose of \( A \):

\[
B = A'
\]

\[
B = \begin{bmatrix}
1 & 2 & 4 \\
2 & 5 & 10 \\
0 & -1 & -1
\end{bmatrix}
\]
MATLAB and matrices

• MATLAB does not require you to deal with matrices as an array of numbers
• MATLAB knows when you are dealing with matrices and adjusts your calculations accordingly

• Matrix multiplication:
  \[ C = A \times B; \]

• Element-wise multiplication:
  \[ D = A \times B; \]

• Inverse matrix:
  \[ E = \text{inv}(A); \]

• Eigenvalues:
  \[ e = \text{eig}(A); \]
Random numbers

Efficient pseudorandom number generator:

\[
A = \text{rand}(4,4); \quad \% \quad \text{Uniform}[0,1] \ 4x4 \ matrix \\
B = 2*\text{rand}(2,2) - 1; \quad \% \quad \text{Uniform}[-1,1] \ 2x2 \ matrix \\
C = \text{randn}(1,10); \quad \% \quad \text{Normal}(0,1) \ dist \ 1x10 \ matrix
\]

*Example: generating a bitstring of length 15:*

\[
D = \text{rand}(15,1) > 0.5;
\]
Basic plots

Plot of $f(x) = \sin(3\pi x)$ using 51 points on the interval [0,1]:

```matlab
>> x=[0:1/50:1];
>> y=sin(3*pi*x);
>> plot(x,y,'k-');
```

You can fancy up the plot with a title, axis labels, a legend, etc.
3D Line Plots

>> t = 0:pi/50:10*pi;
>> plot3(sin(t),cos(t),t);
>> grid on
>> axis square
3D Graphics: \textit{mesh}

\begin{verbatim}
>> [X,Y] = meshgrid(-8:0.05:8);
>> R = sqrt(X.^2 + Y.^2) + eps;
>> Z = sin(R)./R;
>> mesh(X,Y,Z);
\end{verbatim}
3D Graphics: `surf`

```plaintext
>> [s,t] = meshgrid(0:2*pi/20:2*pi,0:2*pi/20:2*pi);
>> x = (2 + cos(s)) .* cos(t);
>> y = (2 + cos(s)) .* sin(t);
>> z = sin(s);
>> surf(x,y,z),shading('interp'),colormap(copper);
>> axis('equal'),axis('off'),view([-10,40])
```
Creating your own functions

You can create your own MATLAB functions by creating a file `function_name.m`. A simple example, traparea.m:

```
function area = traparea(a,b,h)

% traparea(a,b,h) Computes the area of a trapezoid given the dimensions a, b and h, where a and b are the lengths of the parallel sides and h is the distance between these sides

area = 0.5 * (a + b) * h;

end
```

Typing `>> help traparea` will display this section.
Conditional statements

It is just as you would expect:

```plaintext
if x > 2 && x < 5
    x = x + 5;
elseif x >= 5
    x = x - 1;
else
    x = x - 10;
end
```
Loops

For loops:
for i = 1:10
    x(i) = i^2;
end

While loops:
i = 1;
while i < 10
    x(i) = i^2;
    i = i + 1;
end
Function handles

• Pass function as argument to other functions

• Useful in programming optimizer algorithms: one optimizer for different fitness functions

• Create a function handle by adding ‘@’ symbol in front of the function name
Function handle example

function value = example(func_handle, x)
    value = func_handle(x);
end

A function to be calculated, for instance \texttt{sin}, can be passed as follows:

\begin{verbatim}
>> a = [0:0.1:10]
>> example(@sin, a)
\end{verbatim}

Or like this:

\begin{verbatim}
>> func_handle = @(x) sin(x)
>> example(func_handle, a)
\end{verbatim}
MATLAB efficiency

• User-defined MATLAB functions are *interpreted*

• For this reason MATLAB programs can be much slower than programs written in a compiled language such as Fortran or C

• Use **built-in functions and operators** whenever possible, executing compiled rather than interpreted code!

• Furthermore, use matrix operations instead of loops
  – Built-in functions operating on matrices
  – Element-wise operations
Matrix operations versus loops

Compare:

\[ \begin{align*}
&\text{>> } dx = pi/30; \\
&\text{>> } nx = 1 + 2*pi/dx; \\
&\text{>> for } i = 1:nx \\
&\phantom{\text{>> for }} x(i) = (i-1)*dx; \\
&\phantom{\text{>> for }} y(i) = \sin(3*x(i)); \\
&\text{end}
\end{align*} \]

To:

\[ \begin{align*}
&\text{>> } x = 0:pi/30:2*pi; \\
&\text{>> } y = \sin(3*x);
\end{align*} \]
Efficiency continued

- Bottom line – try to work as much as possible with matrix notation!

- Examine the time execution of your code:

  ```
  >> Tic;
  -- Operation --
  >> Toc;
  ```
MATLAB HELP!

The most important feature in MATLAB… the help command!

Great documentation, demos etc.! Use it!
Octave → the free MATLAB alternative

- GNU Octave is an open source tool that is largely compatible with MATLAB code.
- It is not as fast as MATLAB, nor does it provide the nice MATLAB look-and-feel, but it is free!
- Website: [http://www.gnu.org/software/octave/](http://www.gnu.org/software/octave/)
Optimization with MATLAB

Binary Monte Carlo Search
Binary Monte Carlo Search

Our problem:
• Search space: \( \{0,1\}^n \)
• An objective function, to be maximized: \( f(a) \rightarrow \max \)

The algorithm:

Input: objective function \( f \), bitstring length \( n \), the number of iterations \( \text{iters} \)
Output: (sub)optimal bitstring \( a_{\text{opt}} \)

1. \( a_{\text{opt}} := \) generate random bitstring of length \( n \)
2. for \( i = 2 \) to \( \text{iters} \) do
3.   \( a := \) generate random bitstring of length \( n \)
4.   if \( f(a) \geq f(a_{\text{opt}}) \)
5.     \( a_{\text{opt}} := a \)
6. end
7. end
8. return \( a_{\text{opt}} \)
From pseudo-code to MATLAB

Input: objective function $f$, bitstring length $n$, the number of iterations $iters$

Output: (sub)optimal bitstring $a_{opt}$

1. $a_{opt}$ := generate random bitstring of length $n$
2. for $i = 2$ to $iters$ do
3.   $a$ := generate random bitstring of length $n$
4.   if $f(a)$ >= $f(a_{opt})$
5.      $a_{opt}$ := $a$
6. end
7. end
8. return $a_{opt}$

```matlab
function aopt = binary_monte_carlo_optimization(f, n, iters)
aopt = rand(n,1) > 0.5;
for i = 2:iters
    a = rand(n,1) > 0.5;
    if (f(a) >= f(aopt))
        aopt = a;
    end
end
end
```
function aopt = binary_monte_carlo_optimization(f, n, iters)
%
% Performs a binary Monte Carlo search. Given objective f, bitstring length n, and number of iterations, this algorithm will try to find the bitstring that maximizes f.
%
% Author: Johannes Kruisselbrink
% Last modified: September 6, 2010

    aopt = rand(n,1) > 0.5;
    for i = 2:iters
        a = rand(n,1) > 0.5;
        if (f(a) >= f(aopt))
            aopt = a;
        end
    end
end

Displayed upon calling
help binary_monte_carlo_optimization
Store fitness function value

- Function evaluations are costly; do not waste them!

```matlab
function aopt = binary_monte_carlo_optimization(f, n, iters)
aopt = rand(n,1) > 0.5;
fopt = f(aopt);
for i = 2:iters
    a = rand(n,1) > 0.5;
    fa = f(a);
    if (fa >= fopt)
        aopt = a;
        fopt = fa;
    end
end
end
```

And so we save half of our objective function evaluations!!!
You may want to see more than just the optimal bitstring

```matlab
function [aopt, histf] = binary_monte_carlo_optimization(f, n, iters)
aopt = rand(n,1) > 0.5;
fopt = f(aopt);
histf(1) = fopt;
for i = 2:iters
    a = rand(n,1) > 0.5;
    fa = f(a);
    if (fa >= fopt)
        aopt = a;
        fopt = fa;
    end
    histf(end+1) = fopt;
end
end
```
function [aopt, histf] = binary_monte_carlo_optimization(f, n, iters)
    aopt = rand(n,1) > 0.5;
    fopt = f(aopt);
    histf(1) = fopt;
    for i = 2:iters
        a = rand(n,1) > 0.5;
        fa = f(a);
        if (fa >= fopt)
            aopt = a;
            fopt = fa;
        end
        histf(end+1) = fopt;
    end
    plot(histf)
    drawnow()
end
Initialize vectors beforehand

```matlab
function [aopt, histf] = binary_monte_carlo_optimization(f, n, iters)
    histf = zeros(1,iters);
    aopt = rand(n,1) > 0.5;
    fopt = f(aopt);
    histf(1) = fopt;
    for i = 2:iters
        a = rand(n,1) > 0.5;
        fa = f(a);
        if (fa >= fopt)
            aopt = a;
            fopt = fa;
        end
        histf(i) = fopt;
        plot(histf)
        drawnow()
    end
end
```

Initialize vectors beforehand if possible. It can save tremendous amounts of time!
(Otherwise, MATLAB will re-allocate space every time you increase their size)