Computer Vision

Matlab

A good choice for vision program development because

- Easy to do very rapid prototyping
- Quick to learn, and good documentation
- A good library of image processing functions
- Excellent display capabilities
- Widely used for teaching and research in universities and industry

Has some drawbacks

- Slow for some kinds of processes
- Not geared to the web
- Not designed for large-scale system development

Used extensively in

- aerospace and defence
- astronomy
- remote sensing
- medical imaging
- biomedical engineering
- manufacturing automation
- materials science
- genetics research

What does Matlab code look like?

A simple example:

```
a = 1
while length(a) < 10
   a = [0 a] + [a 0]
end
```

which prints out Pascal's triangle:

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
1   6  15  20  15   6   1
1   7  21  35  35  21   7   1
1   8  28  56  70  56  28   8   1
1   9  36  84 126 126  84  36   9   1
```

(with "`a=`" before each line).

Note that the example is the *whole* program, not a fragment. It has:

- no packaging
- no variable declarations
- automatic storage allocation
- automatic printing of results
- some operations which are like other languages: `while`, `=`, `<`
- some which are rather different: `[], +`

How to run a Matlab program

Start Matlab

- Type `matlab` to the system prompt in a terminal window

or

- Select `matlab` from the applications menu

The matlab prompt is >>

Recent versions of Matlab may give you an elaborate window. Learn how to use it gradually — start with the basics in the command window.

Type in the code

Type the program directly into matlab window against the prompt. The first line will execute immediately. The rest will execute as soon as you type `end`.

- This is excellent for quickly trying out ideas.

OR

Put the program in a file using your favourite text editor (not a word processor). Call the file `pascal.m` (the `.m` is significant). The file can be in

- your current working directory
- a directory called `matlab` in your home directory
- in any directory on your "Matlab path" — see the documentation.

The program is called a **script**, in an **m-file**. At the Matlab prompt, type

```
pascal
```

The script will run and you will see the results.

Matrices

Data in Matlab is always held as a **matrix**. Most are 2-D, in which individual elements are referenced by *row* and *column*. This is a 6×10 matrix:.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	12	15	12	18	19	20	19	17	16	15
2	13	14	61	19	11	9	10	12	14	19
3	13	15	18	17	19	20	23	11	10	12
4	14	15	14	14	17	20	21	10	9	12
5	12	13	13	19	30	35	36	15	19	15
6	15	16	17	18	45	40	38	16	15	12

If this matrix is given the name `m`, then `m(5,4) = 19`, for example.

Remember: ***row*** then ***column***.

In Matlab, rows and columns are always numbered starting at 1.

Matlab matrices are of various types to hold different kinds of data. We will mainly use those that hold numbers (floats and integers).

A single number is really a 1×1 matrix in Matlab!

Variables

Matlab variables are not given a type, and do not need to be declared. Any matrix can be assigned to any variable.

Building matrices with [ ]

We can assemble matrices from smaller matrices. Square brackets are used to do this.

Space or comma means “put the matrices side by side”:

```
m = [1 2 3]
```

prints

```
m =
   1 2 3
```

(`m = [1, 2, 3]` would do the same.) The separate 1×1 matrices have been assembled into a 1×3 matrix.

Semicolon means “put the matrices above each other”:

```
m = [1; 2; 3]
```

prints

```
m =
   1
   2
   3
```

Variable names inside square brackets get interpreted. So what does

```
x = [m m m]
```

print?

Square brackets do not get nested, so ordinary matrices do not have other matrices as elements. Thus `x = [[m] [m] [m]]` is the same as `x = [m m m]`.

Operating on whole matrices

In most languages, you need to write loops to do things to all the elements of a data structure such as an array. In Matlab, many loops can be avoided.

```
a = [4 5 1; 3 6 8] + 1
```

prints

```
a =
   5 6 2
   4 7 9
```

That is, 1 has been added to every element of the matrix. If + is applied to two matrices of the same dimensions, corresponding elements are added:

```
b = a + a
```

prints

```
b=
   10  12   4
    8  14  18
```

But element-by-element multiplication requires the special `.*` operator

```
c = a .* b
```

prints

```
c =
   50   72    8
   32   98  162
```

Many other operators that you are probably familiar with work on whole matrices in this way, e.g. `-` (subtraction), `./` (division), `>` (greater than), `<` (less than). Logical operators like > return a **binary** matrix which contains only 0s and 1s, where 0 means “false” and 1 means “true”.

The powerful “:” operator

The colon operator is extremely important. It produces a sequence of numbers in a matrix:

```
i = 5:9
```

prints

```
i =
   5     6     7     8     9
```

The code `i=[5:9]` does exactly the same.

Getting submatrices

A matrix can be indexed using another matrix, to produce a subset of its elements:

```
a = [100 200 300 400 500 600 700];
b = [3 5 6];
c = a(b)
```

The semicolons suppress printing of the first 2 lines, so this prints

```
c =
   300     500     600
```

To get a subsection of a matrix, we can produce the index matrix with the colon operator:

```
a(2:5)
```

prints

```
ans =
   200     300     400     500
```

This works in 2-D as well, e.g. `c(2:3, 1:2)` produces a 2×2 submatrix. *The rows and columns of the submatrix are renumbered.*

“for” loops and the colon operator

Like other languages, Matlab has loops, though these are not needed as often as you might expect. `for` loops iterate over the elements of an matrix:

```
for i = [10 9 55 65]
   i
end
```

prints (on successive lines)

```
i = 10     i = 9     i = 55     i = 65
```

These loops are very often used with the colon operator to iterate over a continuous range of integers:

```
for i = 3:7
   i
end
```

prints (on successive lines)

```
i = 3   i = 4   i = 5   i = 6   i = 7
```

The colon operator tries to be sensible

If it is used as part of an matrix index expression, the : operator will use sensible beginning and end points if these are omitted:

```
c = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

(prints)

```
c =
   1     2     3     4
   5     6     7     8
   9    10    11    12
```

```
c(2, :)
```

(prints)

```
ans =
   5     6     7     8
```

because the column numbers run from 1 to 4, the : translates into `1:4`. This is known as “selecting a row”.

Treating matrices as images

We can create a large matrix using the local vision material (see the web pages for details) like this:

```
image = teachimage('edin_lib.bmp');
```

The matrix elements can now be interpreted as grey levels and the row and column indices as position on the screen. We can produce a picture from the data using an image processing toolkit function like this:

```
imshow(image);
```

We can find out how many rows and columns this has with

```
size(image)
```

which prints

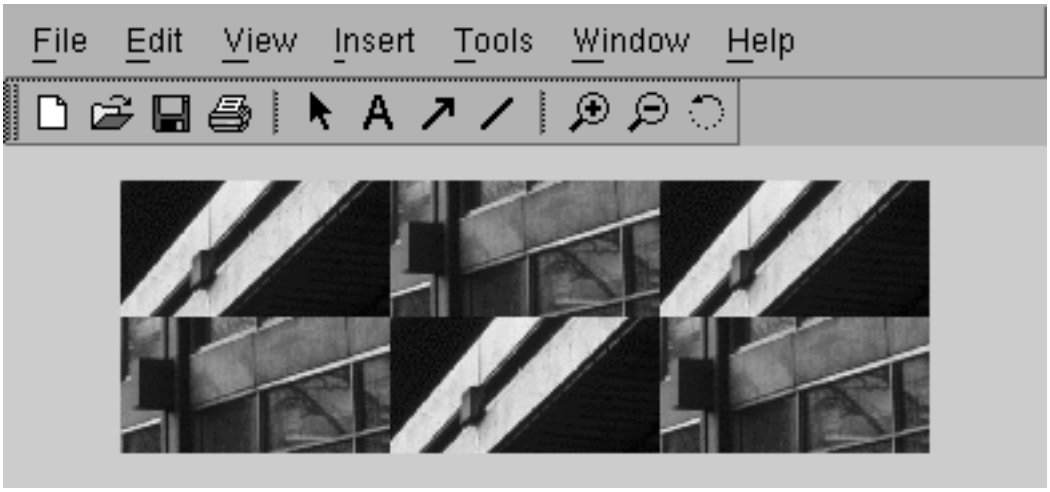
```
ans =
   314   469
```

Now it is possible to manipulate chunks of this using the operators we know about. (We really need the semicolons at the end of lines now!)

For example

```
part1 = image(100:150, 200:300);
part2 = image(250:300, 250:350);
collage=[part1 part2 part1; part2 part1 part2];
imshow(collage);
```

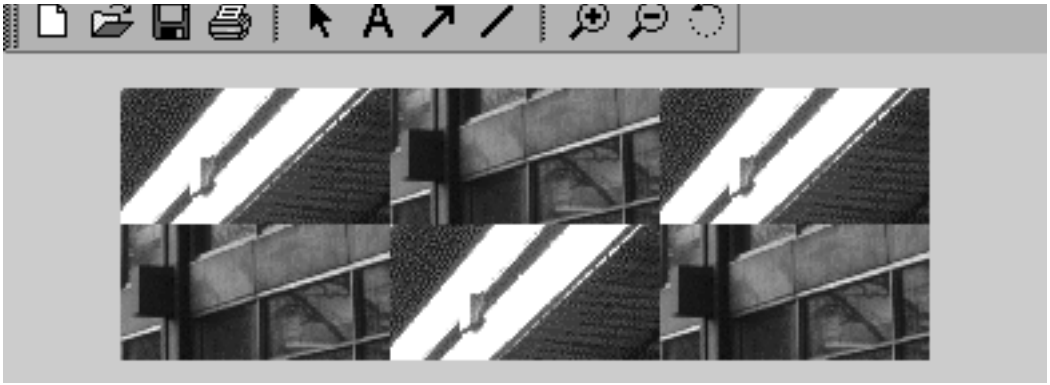
Makes a new window which shows



The normal operators can be applied to the image matrices as well:

```
part1 = 3 * part1;
collage=[part1 part2 part1; part2 part1 part2];
imshow(collage);
```

Modifies the display to make part 1 brighter:



Functions in Matlab

Direct typing and scripts are not suitable for building real programs. In Matlab, the key to this is to use **functions**.

We have already called two functions — `teachimage` and `imshow` — to carry out some image operations.

Functions take **arguments** as in other languages. They are rather like methods in Java.

Creating functions is like creating a script, except that you start with a declaration of how the function will be called. For example, here is a function that removes a border from a matrix, making a smaller matrix with a given number of elements removed all the way round:

```
function y = removeborder(x, n)
   y = x(1+n:end-n, 1+n:end-n);
```

- The word `end` just means the number of rows or columns of the matrix. It's only used this way in colon expressions which are making matrix indices. (This isn't special to functions.)
- The variables `x`, `y` and `n` are private to the function. So would be any other variables used inside the function. They can't be affected by, or affect, other parts of the program, except through a call to the function.
- `x` and `n` are input variables. Values are assigned to them when you call the function. `y` is an output variable. The value it has when the function returns is assigned to something in the calling program.
- The first line of the function shows how you call it.
- The function must go in an m-file with the same name as the function. In this case you would store it in `removeborder.m`.

Calling a function

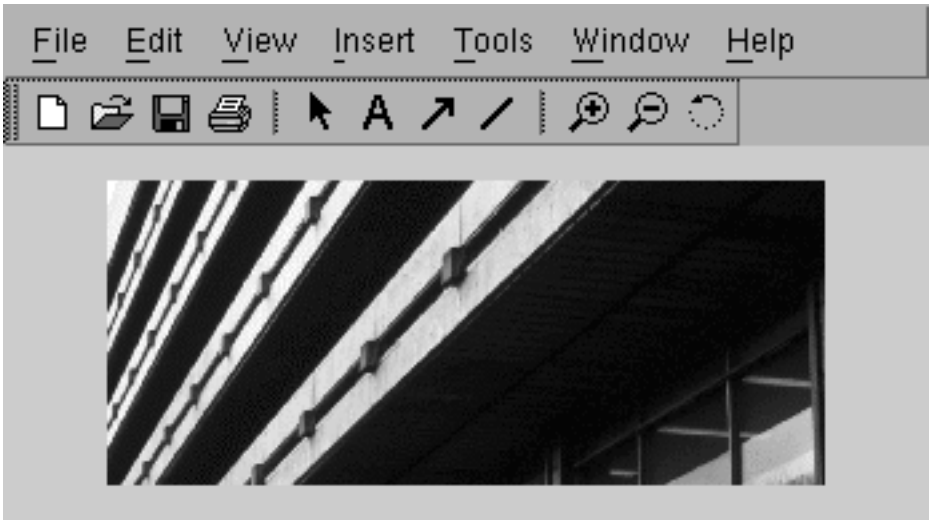
Suppose you have written this function and put it into an m-file in the current directory (or one on your Matlab path). Here's the function again:

```
function y = removeborder(x, n)
   y = x(1+n:end-n, 1+n:end-n);
```

We will call it to remove a border of 100 pixels from round our image:

```
smallimage = removeborder(image, 100);
```

If you use `imshow(smallimage)` to look at the result, we see this:



The calling code can be at top-level — or it could be in another function or a script. This is what happens, in effect (note that `%` introduces a comment):

```
x = image;      % set up the function by
n = 100;        % assigning to the arguments
% —— execute the function code ——
smallimage = y;   % return the result
```

though of course `x`, `n` and `y` are not actually visible outside the function.

When you write serious code, most or all of your program should be in the form of functions.

Going further

Look at the course documentation. In particular, the link for this lecture labelled “Matlab Introduction” points to a web page called *Introducing Matlab for Image Processing* which takes up some of these ideas.

The most important thing in that document is what it tells you about Matlab’s help system and documentation. If you use that effectively, you should be able to find out everything you need to know to produce powerful image processing programs.

Finally: look back at the Pascal’s triangle program on the first slide and make sure you understand each step. Then straight away try to write a little program of your own — it doesn’t matter what — and start experimenting.