Matlab

Carlo Tomasi

MATLAB is a simple and useful high-level language for matrix manipulation. It is often convenient to use MATLAB even for programs for which this language is not the ideal choice in terms of data structures and constructs. This is because MATLAB is an interpreted language, which makes program development very easy, and includes extensive tools for displaying matrices and functions, printing them into several different formats like Postscript, debugging, and creating graphical user interfaces. In addition, the MATLAB package provides a huge amount of predefined functions, from linear algebra to PDE solvers to image processing and much more. The majority of these functions are available as MATLAB source code, which can be read by typing `type f` to the MATLAB prompt, where `f` is the desired function.

This note is a quick primer for MATLAB version 5. This version of MATLAB differs from previous ones most importantly because of the introduction of multidimensional arrays and record-type data structures. By far the best way to learn about the various features of Matlab is to run the program by typing `matlab5`. At the prompt, type `helpdesk`, which starts Netscape and initializes it with the top-level MATLAB documentation page. Under **MATLAB topics**, click on "Getting started" and read on. It is a good idea to keep Netscape running while working with MATLAB, so you have an online manual handy.

1   Example Application: Images

Since images are matrices of numbers, many vision algorithms are naturally implemented in MATLAB. Because images must be read and written from and to files, and color images are three-dimensional arrays, images are good examples of many of the features of MATLAB. In this note, we develop a sample program for reading images, thereby introducing you to MATLAB by example. Because of the simplicity of the basic MATLAB constructs, playing with an example is an

1

effective way to learn the language. It is a good idea to read these notes in front of a terminal, so you can try out the examples.

It is useful to see images as functions $\mathbf{I}(\mathbf{x})$ from $\mathbf{R}^m$ to $\mathbf{R}^n$. For instance, with $m = 2$ and $n = 1$ we have a regular black-and-white image, while color images have $m = 2$ and $n = 3$. Thus, for these images, the vector $\mathbf{x}$ is two-dimensional and represents image position. For each value of $\mathbf{x}$, a color image has three values that identify the color of the image at that point, so for a fixed $\mathbf{x}$ the quantity $\mathbf{I}(\mathbf{x})$ is a vector with three components, say, the red, green, and blue components of the desired color. A color image *sequence* is a function from $\mathbf{R}^3$ to $\mathbf{R}^3$; the only difference with respect to a single color image is the presence of a third component in $\mathbf{x}$, which denotes time.

Of course, for digital images, these functions are represented by discrete values, and become functions that are typically from integers to integers. For instance, a color frame grabber may output an array of 480 rows by 640 columns of three-byte pixels, so the color values are in $[0 \ 255]^3$. The pixel at position $(120, 215)$ may contain, say, color $(35, 201, 96)$, which represents a bright, bluish green.

As soon as we start working on images, however, they become functions from integers to reals. For instance, if we average two images, the result is not an integer image any more, since for instance $(105 + 110)/2 = 107.5$. For values, MATLAB makes no distinction between integers and reals: numbers are numbers[1], and are represented as double-precision floating-point numbers. For subscripts (our $\mathbf{x}$), on the other hand, MATLAB wants positive integers. We stress that subscripts must be *positive* integers. If you are used to C, this is a novelty, since in C a subscript of zero is ok, but for MATLAB it is not. This convention is consistent with standard conventions in linear algebra, where the first entry of a matrix A is a_{11}, not a_{00}.

How should we represent an image? A color image sequence, as we saw, is a function from $\mathbf{N}^3$ to $\mathbf{R}^3$ (here, $\mathbf{N}$ represents natural numbers), but this does not by itself determine how it should be stored and represented. For instance, to store a 10-frame color image sequence where each image has 480 rows and 640 columns with three real-valued color bands we could use

- ten 480×640 arrays, where each entry contains a 3D vector;

- one $480 \times 640 \times 10$ array, where each entry contains a 3D vector;

- thirty 480×640 arrays, each entry corresponding to one color band (red, green, or blue) for each pixel of each frame;

---

[1] Internally, however, MATLAB is smart about the distinction between reals and integers.

- one $480 \times 640 \times 10 \times 3$ array, each entry corresponding to one color band (red, green, or blue) for each pixel of each frame.

For generality and simplicity, we use the last convention, where all subscripts, both in the domain (x, y, t) and in the range (red, green, blue) are treated uniformly. The user must know what the subscripts mean, and how many are needed.

1.1 Variables and Arrays

In MATLAB, with this convention, a 480×640 color image with all zero entries can be created by the command (here and elsewhere, '>>' is the MATLAB interpreter prompt)

```
>> image = zeros(480, 640, 3);
```

If you terminate an instruction with a semicolon the instruction is executed, but the result is not displayed. Omitting the semicolon causes the result to be displayed. When calling a function that returns an image, it is important to type the semicolon to avoid hundreds of thousands of numbers to scroll on your screen[2]. A sequence with ten color images, all zero, is created as follows:

```
>> sequence = zeros(480, 640, 10, 3);
```

Generally, MATLAB variables need not be initialized. The command

```
>> a(2, 3) = 4

a =

     0     0     0
     0     0     4

>>
```

creates the smallest possible array for which `a(2, 3)` makes sense, puts the number 4 in the proper place, and fills the rest with zeros. If we then type

---

[2]However, `Ctrl-D` will harmlessly abort any command.

3

```
>> a(4, 2) = 1

a =

      0      0      0
      0      0      4
      0      0      0
      0      1      0

>>
```

the array `a` is resized so that `a(4, 2)` addresses a valid location. Every time an array is resized, however, the space for it is reallocated, causing a call to the operating system through the C function `realloc`. This *dynamic allocation* takes time, so for large arrays it is a good idea to preallocate the array by a call to the builtin function `zeros` as in the examples above.

The variable `sequence` defined above is rather large, because it refers to $480 \times 640 \times 10 \times 3 = 9,216,000$ double-precision floating point numbers, for a total of about 74 MBytes of storage. To obtain 9,216,000 *bytes* instead, we can use the conversion function `uint8` (8-bit unsigned-integer:

```
sequence = uint8(zeros(480, 640, 10, 3));
```

The builtin function `size` returns a vector with all the dimensions of an array:

```
>> size(sequence)

ans =

    [480 640 10 3]

>>
```

The function `whos` shows the sizes and storage requirements of all the variables in the *workspace*, which is the space of variables known by the MATLAB interpreter.

MATLAB has a rather rich, though simple, mechanism for referring to parts of an array. Consider for instance the following array:

```
>> a = 10*(1:5)' * ones(1, 4) + ones(5,1) * (1:4)
```

```
a =

    11    12    13    14
    21    22    23    24
    31    32    33    34
    41    42    43    44
    51    52    53    54


>>
```

We created this array by taking the vector `1:5`, which is a row vector of the integers between 1 and 5, that is, `[1 2 3 4 5]`, transposing it with the prime, and multiplying it by ten; this yields the column vector

$$\begin{bmatrix} 10 \\ 20 \\ 30 \\ 40 \\ 50 \end{bmatrix}$$

or in MATLAB notation `[10; 20; 30; 40; 50]`, where the semicolon starts a new row. The call `ones(1, 4)` to the builtin function `ones` creates a 1×4 matrix of ones, so `10*(1:5)' * ones(1, 4)` is

$$\begin{bmatrix} 10 & 10 & 10 & 10 \\ 20 & 20 & 20 & 20 \\ 30 & 30 & 30 & 30 \\ 40 & 40 & 40 & 40 \\ 50 & 50 & 50 & 50 \end{bmatrix}$$

and similarly `ones(5,1) * (1:4)` is

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

and adding the two together yields the 5×4 matrix a shown in the example. The following commands refer to parts of a. This should be self-explanatory. If not, type `help colon` to find out.

5

```
>> a(3,2)

ans =

    32

>> a(:, 3)

ans =

    13
    23
    33
    43
    53

>> a(1, :)

ans =

    11    12    13    14

>> a(2:4, 1:3)

ans =

    21    22    23
    31    32    33
    41    42    43

>> a(1:2:5, 1:3)

ans =

    11    12    13
    31    32    33
    51    52    53
```

```
>>
```

The `ans` variable is where MATLAB puts results from an expression unless we specify some other destination. This is a variable like any other. However, its value is redefined by every command that returns a value, so be careful in its use.

In MATLAB, variables are not declared. If you say `a = 5` or `a = [1 2]`, then `a` is an array of numbers (a scalar is a 1×1 array). If you say `a = 'quaternion'`, then `a` is a string, that is, an array of characters, in this case of length 10, and `a(6)` is the character `'r'`. Variables can also refer to more complex objects called *cell lists* and *structures*. These data types are described in the online manual.

Functions and Scripts

The following program reads an image from a file in either `pgm` ("portable gray map") or `ppm` ("portable pixel map") format. Gray maps are black-and-white images, pixel maps are color images. Here is a description of these formats:

- A "magic number" for identifying the file type. The two characters 'P5' are used for `pgm` and 'P6' for `pgm`.

- Whitespace (blanks, TABs, carriage returns, line feeds).

- The image width in pixels, formatted as ASCII characters in decimal.

- Whitespace.

- The image height in pixels, again in ASCII decimal.

- Whitespace.

- The maximum value for each component, again in ASCII decimal. The only value allowed here is 255.

- Whitespace.

- Width $\times$ height gray values for `pgm`, or width $\times$ height $\times$ 3 color values for `ppm`. These values are raw bytes, with no separators. For `ppm` images, the three values for each pixel are adjacent to each other, and correspond to red, green, and blue values. Values (or triples of values for `ppm`) start at the top-left corner of the image, proceeding in normal English reading order.

7

- Characters from a '#' to the next end-of-line are ignored (comments) if they appear *before* the pixel values.

A possible source of program errors is that these image formats specify image width first and image height second. MATLAB, on the other hand, specifies matrix dimensions by giving the number of rows first, and the number of columns second.

To write a function to read an image in either format, we create a file called `pnmread.m`. The second character, n, stands for "any," meaning that this function will read either `pgm` or `ppm` (this is a rather narrow notion of "any"). All MATLAB functions are files with the same name as the function and extension `.m`. Here is most of `pnmread`:

```
function i = pnmread(filename)

% open file
[fid, msg] = fopen(filename, 'r');
if fid == -1,
  error(msg)
end

% read magic number
magic = readstring(fid);
if length(magic) ~= 2,
  error('Unknown image format')
end
if any(magic ~= 'P5') & any(magic ~= 'P6'),
  error('Unknown image format')
else
  w = readnumber(fid);
  h = readnumber(fid);
  maxvalue = readnumber(fid);
  fgetl(fid);
  if all(magic == 'P5'),
    % read pgm image; we will complete this later
  else % must be P6
    % read ppm image; we will complete this later
  end
end
```

```
% close file
fclose(fid);
```

The functions `readstring` and `readnumber` are not predefined, so we will need to write those as well. They essentially skip comment lines, which start with a '#', and return the next string or number in the file.

The first thing to notice in the function definition above is a little redundancy: the base of the file name, `pnmread`, is repeated in the function declaration, the line starting with the word `function`. What counts for MATLAB is the file name. You could replace `pnmread` in the declaration with any other name, and this function would still be called `pnmread`, because this is the name of the file in which the function resides.

After the word `function` there is either no variable, a single variable, or a comma-separated bracketed list of variables, as in `function [a,b,c] = f(d)`. Thus, functions in MATLAB can return any number of arguments, including zero. When more than one argument is returned, not all arguments need be read by the caller. Inside the function, a builtin variable `nargout` specifies how many arguments are actually being requested by the caller for the particular function invocation. So if the `function [a,b,c] = f(d)` is being called as follows:

```
[q,r] = f(3)
```

then `nargout` will have a value of 2 within that invocation of `f`. A variable `nargin` similarly holds the number of arguments actually passed to the function, which cannot exceed the number of formal parameters in the `function` declaration.

Notice that the return arguments are simply listed in the `function` declaration, and no explicit `return` statement is needed in the function body. When the function returns to the caller, the values of the return variables are the last values these variables were assigned within the function body. For early returns from the middle of a function body, MATLAB provides a `return` statement, which takes no arguments.

If the function declaration is omitted, the file becomes a *script*. This has two important consequences: first, no input or output arguments can be passed to and from a script. Second, all variables defined inside the function are also visible in the workspace. Consider for instance the rather silly function

```
function a = add(b, c)
a = b+c;
```

defined in a file `add.m`. Here is what happens when we call this function and we
try to examine `b`.

```
>> n = add(2,3)
n =

     5
>> b
??? Undefined function or variable 'b'
```

because `b` is not known in the workspace. If we now comment out or remove
the line `function a = add(b, c)` from the file `add.m`, this file becomes
a script and can be used as follows:

```
>> b = 2;
>> c = 3;
>> n = add

n =

     5

>> b

b =

     2
```

Because `add.m` is a script, the variables `b` and `c` are global, and can be
examined from the interpreter: typing `b` at the prompt, as shown above, displays
2.

Whenever you find yourself doing the same thing over and over again in
MATLAB, it is a good idea to write a script. Another use for scripts is when you are
debugging a function. Although MATLAB provides a complete set of debugging
constructs (type `help debug` to find out), it is often easier to comment out the
`function` declaration of the broken function, define values for the arguments
by hand, and run the headerless function, which is now a script. This causes all
the intermediate variables to be visible to the interpreter, and you just need to type
their names to inspect their values.

1.2 File I/O

MATLAB has extensive input/output constructs, including `fopen`, `fclose`, `fscanf`, `fprintf`, `sscanf`, `read`, `write`, `fread`, `fwrite`, `input`. Some of these behave similarly to the homonymous C functions, but with small and important differences in how matrix arguments are handled. Use the `help` command or the online documentation to see the details. In our function `pnmread`, we use a rather minimal subset. The first line in

```
[fid, msg] = fopen(filename, 'r');
if fid == -1,
  error(msg)
end
```

opens the filename whose name is in the argument string `filename`. The result is a file identifier `fid`, just like in C, and an error message stored in the string `msg`. This string is empty (`' '`) if no error has occurred. On error, `fid` is set to `-1`. The command `error(msg)` displays the message in `msg` and aborts the function, returning control to the interpreter. If this call to `error` is deep within a call stack, the entire stack is aborted.

Assuming that things went well with `fopen`, we can now read from the file through its integer identifier `fid`. Let us define the two simple functions `readstring` and `readnumber`, which we write in files `readstring.m` and `readnumber.m`[3]. Here are the contents of `readstring.m`:

```
function s = readstring(fid)

s = fscanf(fid,'%s',1);
while s(1) == '#',
  fgetl(fid);
  s = fscanf(fid,'%s',1);
end
```

This function assumes that we are starting to read from a new line, and reads one blank-space separated string from `fid` via the `fscanf` command, whose syntax is similar to the equivalent C function. The result is placed in the variable

---

[3]It is a little annoying that every function must have its own file. The advantage of this, however, is that MATLAB keeps checking if function definitions have changed, and if so reloads the newest definition. This is very convenient during program development.

11

s, which is a MATLAB vector of characters. Thus, the expression s(1) refers to the first character in s. If this character is a pound sign #, it means that the current line is a comment; the command fgetl(fid) then gets an entire line (the comment) and puts it nowhere: the comment is ignored. The next string is read by the fscanf in the while loop. This continues until some non-comment string is found. Since the variable s is in the function declaration as a return value, the last string found is passed back to the caller when readstring returns.

The function readnumber does something similar to readstring, but looks for a number rather than a generic string. Rather than repeating most of the body of readstring in readnumber, we observe that a number is a string when it is in the image file. Thus, readnumber can simply call readstring and do a string-to-number conversion:

```
function n = readnumber(fid)

s = readstring(fid);
n = sscanf(s,'%d');
```

Rather than using sscanf for the conversion, we could have used the builtin function str2num with the same effect.

Conditional Constructs

Going back to our function pnmread, the command

```
magic = readstring(fid);
```

reads a string from fid and assigns it to the variable magic. This string is expected to contain the magic number P5 for pgm images, or P6 for ppm. We check if this is the case with an if statement

```
if length(magic) ~= 2,
  error('Unknown image format')
end
if any(magic ~= 'P5') & any(magic ~= 'P6'),
  error('Unknown image format')
else
  ...
end
```

12

The comma at the end of the `if` is optional. It is required when the statement after the condition is written on the same line. Let us consider the second `if` first. Notice that the logical 'and' operator in MATLAB is a single ampersand, `&`, unlike C. Similarly, 'or' is a single vertical bar, `|`. Negation is expressed by a tilde, `~`, so `magic ~= 'P5'` means "`magic` is not equal to `'P5'`". To understand the expression `any(magic ~= 'P5')`, notice that in MATLAB equality (==) or inequality (~=) can be applied to arrays, which must be of equal sizes. This is the reason for the first `if` above,

```
if length(magic) ~= 2,
  error('Unknown image format')
end
```

Without this check, the comparison `magic ~= 'P5'` could generate a MATLAB error if `magic` has a length different from 2. The builtin function `length` returns the length of a vector, or the maximum dimension of an array.

When applied to vectors, the equality or inequality operator returns a vector of zeros and ones, corresponding to an element-by-element comparison between the two vectors. Thus, `magic ~= 'P5'` returns `[0 0]` if and only if the two strings are equal. The builtin function `any` returns 1 if any of the elements of the vector are non-zero. Otherwise it returns 0. Type `help any` to see what `any` does with arrays. Thus, `any` corresponds to the existential quantifier. The MATLAB function `all` corresponds to the universal quantifier.

The rest of our sketch of `pnmread` is obvious:

```
w = readnumber(fid);
h = readnumber(fid);
maxvalue = readnumber(fid);
fgetl(fid);
if all(magic == 'P5'),
  % read pgm image
else % must be P6
  % read ppm image
end
```

We read the image width `w`, the image height `h`, the maximum pixel value `maxvalue`, and go the the beginning of a new line with `fgetl(fid)`. Without this, the ASCII code of the newline character itself would be interpreted as the first pixel value.

Reading and Reshaping Matrices

We are now ready to do the real work of reading a `pgm` or a `ppm` image. Here MATLAB has a small inconsistency, which is caused by the fact that old versions of MATLAB only allowed vectors and matrices, and no arrays with more dimensions. The low-level `fread` function, even in MATLAB 5, reads into either a vector or a matrix, but not into an array with more dimensions. Thus, for the case of a `ppm` image, we first read into a matrix, and then convert this matrix into an array with three dimensions. Here is the complete code:

```
function i = pnmread(filename)

% open file
[fid, msg] = fopen(filename, 'r');
if fid == -1,
  error(msg)
end

% read magic number
magic = readstring(fid);
if length(magic) ~= 2,
  error('Unknown image format')
end
if any(magic ~= 'P5') & any(magic ~= 'P6'),
  error('Unknown image format')
else
  w = readnumber(fid);
  h = readnumber(fid);
  maxvalue = readnumber(fid);
  fgetl(fid);
  if all(magic == 'P5'),
    % read pgm image
    i = fread(fid, [w h], 'uint8')';
  else % must be P6
    % read ppm image
    pixels = uint8(fread(fid, [3 w*h], 'uint8'));
    i = uint8(zeros(h, w, 3));
    i(:, :, 1) = reshape(pixels(1,:), w, h)'; % red
    i(:, :, 2) = reshape(pixels(2,:), w, h)'; % green
```

```
    i(:, :, 3) = reshape(pixels(3,:), w, h)'; % blue
  end
end

% close file
fclose(fid);
```

When the image is a `pgm` (magic code P5), the instruction

```
i = fread(fid, [w h], 'uint8')';
```

reads the image into an array of size $w \times h$, rather than $h \times w$. In fact, `fread` reads data into matrices in column-major order, while `pgm` images are stored in row-major order. The prime at the end of the instruction above then transposes the result to obtain the correct image orientation.

If MATLAB 5 were completely consistent, the statements in the `else` part of the `if` statement, where the `ppm` image is being read, could be replaced by a single statement that instructs MATLAB to read $w \times h \times 3$ 8-bit unsigned integers (`uint8`s) from file `fid`, and arrange them into an array with dimensions [`h w 3`] (modulo proper transpositions to account for the different ordering conventions in MATLAB and in the `ppm` format).

Since this is not (yet) allowed, we first read the image into a $3 \times wh$ array called `pixels`. In this way, each row of `pixels` is devoted to a different color band: byte 1 in the image goes to `pixels(1,1)`; byte 2 goes to `pixels(2,1)`; byte 3 goes to `pixels(3,1)`. We have now read the first pixel (red, green, blue), and we go back to row 1, entry `pixel(1, 2)`, for the red entry of pixel 2, and so forth. We then use the builtin function `reshape` to reshape the three rows of `pixels` into arrays of size $w \times h$, transpose the results, place these into the $h \times w \times 3$ array `i`, which is preallocated both for efficiency and to obtain the proper data type `uint8` (MATLAB would make `i` double floating-point by default). The function `reshape` reads the input array in lexicographic order (column-major order for matrices) and returns an array with the specified dimensions. The number of elements in the input array must be equal to the product of the specified dimensions.

Finally, closing the file with `fclose(fid)` makes the file descriptor `fid` available for other files.

15

Writing an Image

At this point you should know almost enough about MATLAB to write a function pnmwrite that writes a black-and-white image to a file in pgm format, and a color image to a file in ppm format. On one hand, writing an image is easier than reading it because writing to a file requires no parsing. On the other hand, writing is slightly trickier than reading in that the values in the input array need to be normalized to $0 \div 255$. Also, the image must be converted to double by something like

```
i = double(i)
```

both because subtraction make no sense for unsigned integer values and because the MATLAB function fwrite only works on arrays of double-precision floating-point numbers. For the normalization, it is useful to know the MATLAB functions min and max (type help min...).

Try to write pnmwrite yourself before looking at the following code.

```
% linearly maps the values in the given array to
% [0 255], quantizes to integers, and stores the
% result in the specified file as a raw pgm or ppm
% image; returns the number of bytes written; the
% input array must be either of size [h w] or of
% size [h w 3]

function count = pnmwrite(i, filename)

fid=fopen(filename,'w');
h = size(i, 1);
w = size(i, 2);

if size(i, 3) == 1,
  bands = 1;
  magic = 'P5';
  pgm = 1;
elseif size(i, 3) == 3,
  bands = 3;
  magic = 'P6';
  pgm = 0;
```

```
else
  error('Third array dimension must be either 1 or 3')
end

% convert input to double if necessary, so arithmetic
% operations make sense; also, fwrite only works on
% doubles
i = double(i);

% scale pixel values; grays should not change to
% blacks, hence the outermost min
minvalue = min(0, min(min(min(i))));
maxvalue = max(max(max(i)));
i = uint8(round((i - minvalue) * 255 / maxvalue));

% put pixels into a 3 x (w*h) or a 1 x (w*h) array of
% 8-bit integers, one row per color band
a = zeros(bands, w*h);
for b = 1:bands,
  a(b, :) = reshape(i(:, :, b)', 1, w*h);
end

% write header
fprintf(fid,'%s\n%d %d\n%d\n', magic, w, h, 255);

% write pixels (a is read in column-major order')
count = fwrite(fid, a, 'uint8');

fclose(fid);
```

Notice that because of the grey-level normalization in `pnmwrite`, typing

```
i = pnmread('a.pgm');
pnmwrite(i, 'b.pgm');
j = pnmread('b.pgm');
pnmwrite(j, 'c.pgm');
```

may result in an image b.pgm different from a.pgm. However, the images in
b.pgm and c.pgm should be equal to each other. The same holds for color
images.

17

Displaying and Printing Images

One of the most useful features of MATLAB is its extensive set of display functions. To display an image, type

```
img = pnmread(filename);
imagesc(img);
axis('square')
```

The call `axis('square')` adjusts the aspect ratio of the display appropriately. The image appears with axis tickmarks that are often useful to identify rows and columns. Both tickmarks and the frame around the picture can be turned off with the command

```
axis('off')
```

There is also a function `image`, which is simpler than `imagesc`. However, `image` does not scale the image values to use the colormap of the display appropriately, so using `image` is not recommended. If the image is black-and-white, it will still be displayed with a color colormap. To obtain a proper display, type

```
colormap(gray)
```

which installs a gray-value colormap. Colormaps can be manipulated in many ways in MATLAB. Type `help colormap` to find out how to use them.

To print an image to a postscript file named `f.eps`, type

```
print -deps f.eps
```

This will print a black-and-white copy of the picture in the current figure, which is either the last figure you worked on or the one you select with the `figure` command. A color postscript file can be generated with

```
print -depsc f.eps
```

Many other options exist. Type `help print` for details. The `print` command prints anything in the current figure, including plots.

A single scanline of a black-and-white image can be plotted by a command like

```
plot(i(100, :))
```

and a patch of the image can be displayed by

```
mesh(i(100:150, 80:120))
```

which draws a surface as a mesh. The mesh can be filled, shaded, lit in very many different ways by using the surf command (for "surface") instead of mesh. Again, the help command or the online documentation gives all the details.

2 Going Ahead

MATLAB has a very rich set of builtin functions. Some are written as MATLAB code, and can be examined by using the type command, which displays the code itself. The help command or the online documentation give details.

While MATLAB is perhaps too inefficient to be used for production code for computer vision or graphics, once you start using it you will realize that the very modest learning time it requires pays off handsomely in terms of increased code productivity and ease of use when developing a program prototype or tinkering with an idea that is not yet fully developed.