Chapter 2
Fundamentals

The images used here are provided by the authors.

Objectives:

- Digital Image Representation
- Image as a Matrix
- Reading and Displaying Images
- Writing Images
- Storage Classes and Data Types
- Image Coordinate System
- Summary of on MATLAB

Rahman Tashakkori, Ph.D., CS – Appalachian State University, Boone, NC 28608

Chapter 2
Fundamentals

Rahman Tashakkori, Ph.D., CS – Appalachian State University, Boone, NC 28608

2.7 Converting between Data Classes and Image Types

2.7.1 Converting between Data Classes

2.7.2 Converting between Image Classes and Types

2.8 Array Indexing

2.8.1 Vector Indexing

2.8.2 Matrix Indexing

2.8.3 Selecting Array Dimensions

2.9 Some Important Standard Arrays

2.10 Introduction to M-Function Programming

2.10.1 M-Files

2.10.2 Operators

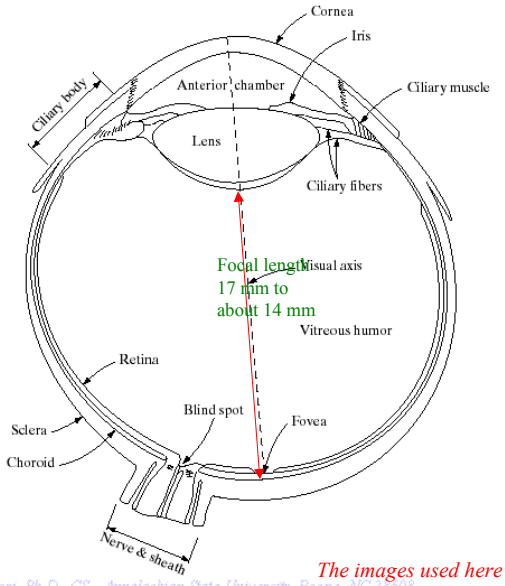
2.10.3 Flow Control

2.10.4 Code Optimization

2.10.5 Interactive I/O

2.10.6 A Brief Introduction to Cell Arrays and Structures 62

Summary



The images used here are provided by the authors.

There are 6 to 7 million cones in each eye. Cones are located at the central portion of the retina. They are highly color sensitive. We use them to resolve fine details. Cone vision is called photopic or bright-light vision.

There are 75-150 million rods distributed over the retinal surface. Larger area of distribution and the fact that several of rods are connected to a single nerve, make them less effective for resolving details. Rod vision is called scotopic or dim-light vision.

Lens is made of concentric layers of fibrous cells and is suspended by fibers that attach to the ciliary body. It contains 60% -70% water, 6% fat, and more protein than any other tissue in the eye.

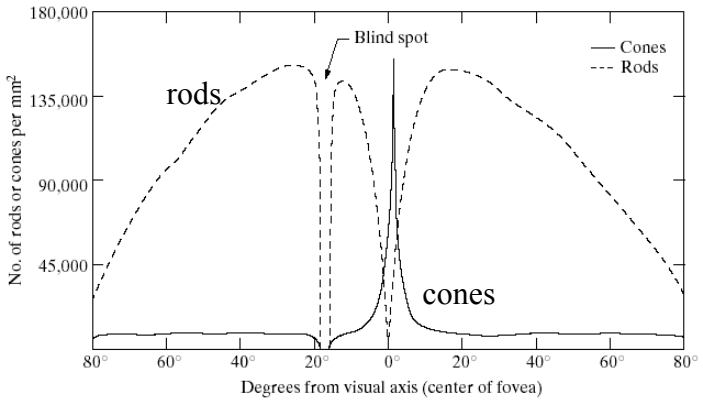


FIGURE 2.2 Distribution of rods and cones in the retina.

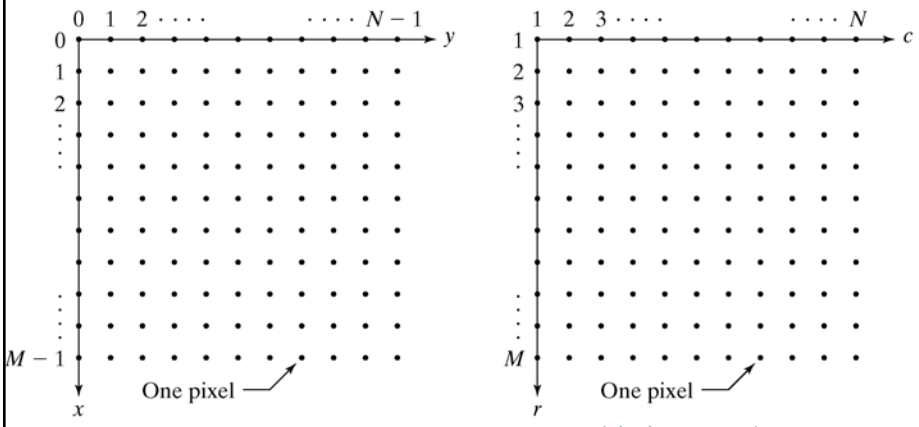
The images used here are provided by the authors.

Chapter 2
Fundamentals

Digital Image Representation

An image can be seen as a two-dimensional function, *f(x,y)*, where *x* and *y* are *spatial* (plane) *coordinates*, and the amplitude of *f* at any pair of coordinates (*x,y*) is called the *intensity* of the image at that point.

The images used here are provided by the authors.



This is an M-by-N image with indices starting from 0

This is an M-by-N image with indices starting from 1 (MATLAB Notation)

Image as Matrices

The image data can be represented as a matrix in the following form:

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & .. & .. & f(0,N-1) \\ f(1,0) & f(1,1) & .. & .. & f(1,N-1) \\ . & . & . & . & . \\ . & . & . & . & . \\ f(M-1,0) & f(M-1,1) & & & f(M-1,N-1) \end{bmatrix}$$

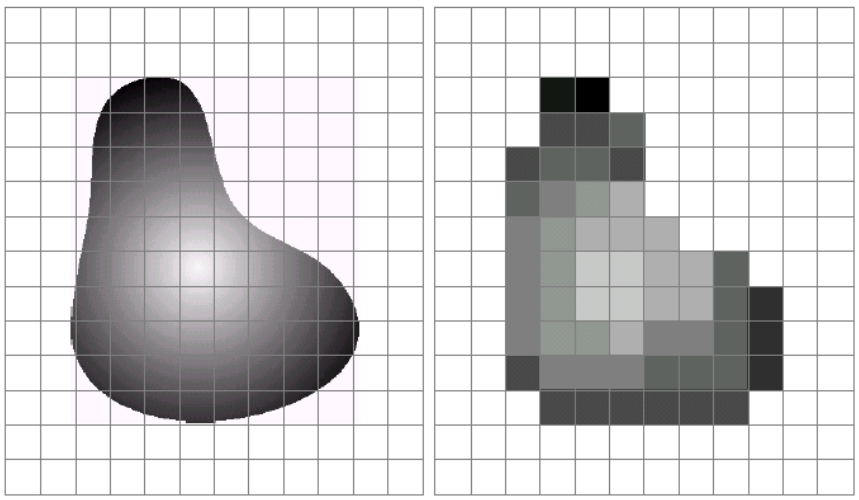
Each element (building block) of an image is called a **pixel**. In MATLAB, the above matrix is represented as:

$$f = \begin{bmatrix} f(1,1) & f(1,2) & .. & .. & f(1,N) \\ f(2,1) & f(2,2) & .. & .. & f(2,N) \\ . & . & . & . & . \\ . & . & . & . & . \\ f(M,1) & f(M,2) & & & f(M,N) \end{bmatrix}$$

This is how you store a 4-by-4 matrix as A in MATLAB:

$$A = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

```
>> A = [1 5 9 13;
       2 6 10 14;
       3 7 11 15;
       4 8 12 16]
```

a b

FIGURE 2.17 (a) Continuos image projected onto a sensor array. (b) Result of image sampling and quantization.

The images used here are provided by the authors.

Chapter 2
Fundamentals

Reading and Displaying Images

In MATLAB images are read using: **imread('filename').** This reads the image that is stored in the current directory. We can include the path to a directory if that is different from the current directory:

f = imread('C:\MATLAB7\toolbox\images\imdemos\football.jpg')

This reads the image *football.jpg* from the hard drive and stores its data in matrix *f*. It is possible to read images of different formats as indicated in Table 2.1.

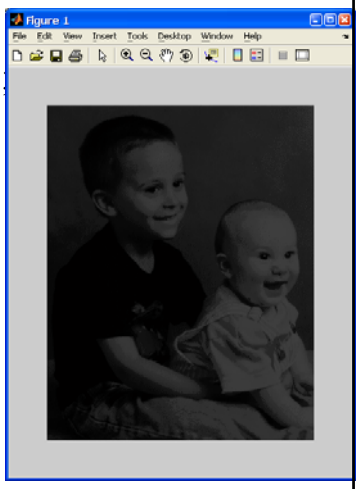
Format Name	Description	Recognized Extensions
TIFF	Tagged Image File Format	`.tif, .tiff`
JPEG	Joint Photographic Experts Group	`.jpg, .jpeg`
GIF	Graphics Interchange Format†	`.gif`
BMP	Windows Bitmap	`.bmp`
PNG	Portable Network Graphics	`.png`
XWD	X Window Dump	`.xwd`

† `GIF` is supported by `imread`, but not by `imwrite`.

The images used here are provided by the authors.

Reading and Displaying Images

Reading: >> f = imread('kids.tif');
Displaying: >> imshow(f)



The **whos** function displays additional information:

```
>> whos f
  Name      Size     Bytes  Class

  f       400x318    127200  uint8 array
Grand total is 127200 elements using 127200 bytes
```


To keep the first image and output a second image, we will use:

>> g = imread('trees.tif');

>> figure, imshow(g)

Now, since you had previously displayed the kids.tif, both the kids.tif and the trees.tif images will appear on the screen.

Writing Images

Assuming we have read an image already using:
f = imread('greens.jpg').
This is an image in jpg format.
Images are written to disk using function **imwrite**, which is used as:
imwrite(f, 'filename')
Using this format, the string for filename MUST include a recognizable file format extension. We can also specify the desired format using:

>> imwrite(f, 'greens', 'tif') Or
>> imwrite(f, 'greens.tif')

We can also write the image with a different quality defined as q%:

imwrite(f, 'greens_25.jpg', 'quality', 25), where 25 is 25% quality

Note: Quality only applies to images written in JPEG format because there is a compression associated with this format. We will discuss this later.

Writing Images

I used the **imfinfo** command on the greens images of 100% and 25% qualities. Here are the result. Everything seems to be the same, so where does the size difference come from?


```
K = imfinfo('greens.jpg')
K = Filename: 'greens.jpg'
FileModDate: '01-Mar-2001
09:52:40'
FileSize: 74948
Format: 'jpg'
FormatVersion: ''
Width: 500
Height: 300
BitDepth: 24
ColorType: 'truecolor'
FormatSignature: ''
NumberOfSamples: 3
CodingMethod: 'Huffman'
CodingProcess: 'Sequential'
Comment: {}
```

```
K = imfinfo('greens_25.jpg')
K = Filename: 'greens_25.jpg'
FileModDate: '01-Mar-2001
09:52:40'
FileSize: 22497
Format: 'jpg'
FormatVersion: ''
Width: 500
Height: 300
BitDepth: 24
ColorType: 'truecolor'
FormatSignature: ''
NumberOfSamples: 3
CodingMethod: 'Huffman'
CodingProcess: 'Sequential'
Comment: {}
```

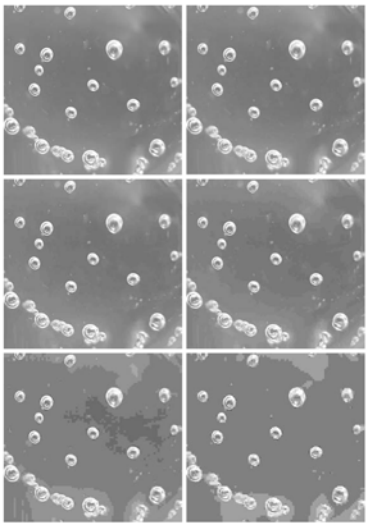
Writing Images – how much we gained how much we lost?

```
>> K=imfinfo('greens.jpg');
>> img_bytes = K.Width*K.Height*K.BitDepth/8;
>> compressed_bytes = K.FileSize;
>> compression_ratio = img_bytes/compressed_bytes

compression_ratio =
    6.0042
```



```
>> K=imfinfo('greens_25.jpg');
>> img_bytes = K.Width*K.Height*K.BitDepth/8;
>> compressed_bytes = K.FileSize;
>> compression_ratio = img_bytes/compressed_bytes

compression_ratio =
   20.0027
```

What is the size of a 16X16 gray scale image in bits? This image contains 64 gray levels.

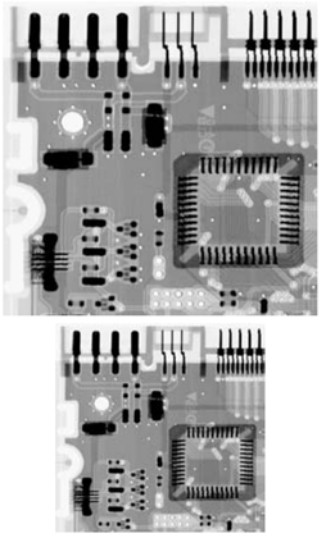


a b
c d
e f

FIGURE 2.4
(a) Original image. (b) through (f) Results of using jpg quality values q = 50, 25, 15, 5, and 0, respectively. False contouring begins to be barely noticeable for q = 15 [image (d)] but is quite visible for q = 5 and q = 0.

The images used here are provided by the authors.

Writing Images



a
b

FIGURE 2.5 Effects of changing the dpi resolution while keeping the number of pixels constant. (a) A 450 × 450 image at 200 dpi (size = 2.25 × 2.25 inches). (b) The same 450 × 450 image, but at 300 dpi (size = 1.5 × 1.5 inches). (Original image courtesy of Lixi, Inc.)

The images used here are provided by the authors.

Storage Classes

By default, MATLAB stores most data in arrays of class double. The data in these arrays is stored as double-precision (64-bit) floating-point numbers. All MATLAB functions work with these arrays.

For image processing, however, this data representation is not always ideal. The number of pixels in an image can be very large; for example, a 1000-by-1000 image has a million pixels. Since each pixel is represented by at least one array element, this image would require about 8 megabytes of memory.

To reduce memory requirements, MATLAB supports storing image data in arrays as 8-bit or 16-bit unsigned integers, class uint8 and uint16. These arrays require one eighth or one fourth as much memory as double arrays.

Data Classes

Name	Description
`double`	Double-precision, floating-point numbers in the approximate range -10^{308} to 10^{308} (8 bytes per element).
`uint8`	Unsigned 8-bit integers in the range [0, 255] (1 byte per element).
`uint16`	Unsigned 16-bit integers in the range [0, 65535] (2 bytes per element).
`uint32`	Unsigned 32-bit integers in the range [0, 4294967295] (4 bytes per element).
`int8`	Signed 8-bit integers in the range [−128, 127] (1 byte per element).
`int16`	Signed 16-bit integers in the range [−32768, 32767] (2 bytes per element).
`int32`	Signed 32-bit integers in the range [−2147483648, 2147483647] (4 bytes per element).
`single`	Single-precision floating-point numbers with values in the approximate range -10^{38} to 10^{38} (4 bytes per element).
`char`	Characters (2 bytes per element).
`logical`	Values are 0 or 1 (1 byte per element).

The images used here are provided by the authors.

Converting between Image Classes and types

Name	Converts Input to:	Valid Input Image Data Classes
`im2uint8`	`uint8`	`logical`, `uint8`, `uint16`, and `double`
`im2uint16`	`uint16`	`logical`, `uint8`, `uint16`, and `double`
`mat2gray`	`double` (in range [0, 1])	`double`
`im2double`	`double`	`logical`, `uint8`, `uint16`, and `double`
`im2bw`	`logical`	`uint8`, `uint16`, and `double`

```
>> f = [-0.5  0.5; 0.75 1.5]
f =
-0.5 0.5
0.75 1.5
>> g = uint8(f)
g =
0  1
1  2
```

Chapter 2
Fundamentals

The Image Processing Toolbox supports four basic types of images:

- Indexed images
- Intensity images
- Binary images
- RGB images

Reading a Graphics Image
Writing a Graphics Image
Querying a Graphics File
Converting Image Storage Classes
Converting Graphics File Formats
Reading and Writing DICOM Files

Indexed Images

An indexed image consists of an array, called X in this documentation, and a colormap matrix, called map. The pixel values in the array are direct indices into a colormap.
The colormap matrix is an m-by-3 array of class double containing floating-point values in the range [0,1]. Each row of map specifies the red, green, and blue components of a single color. An indexed image uses direct mapping of pixel values to colormap values. The color of each image pixel is determined by using the corresponding value of X as an index into map.

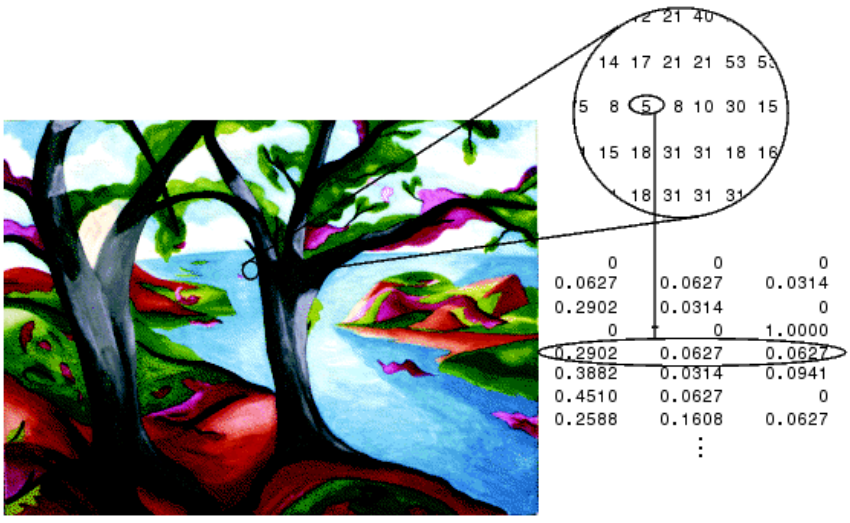
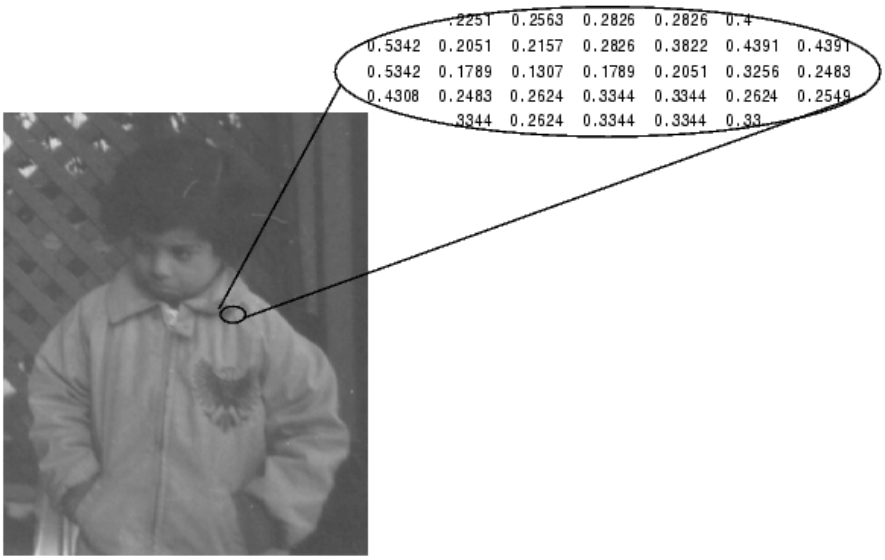


Image Courtesy of Susan Cohen

The images used here are provided by the authors.

Intensity images

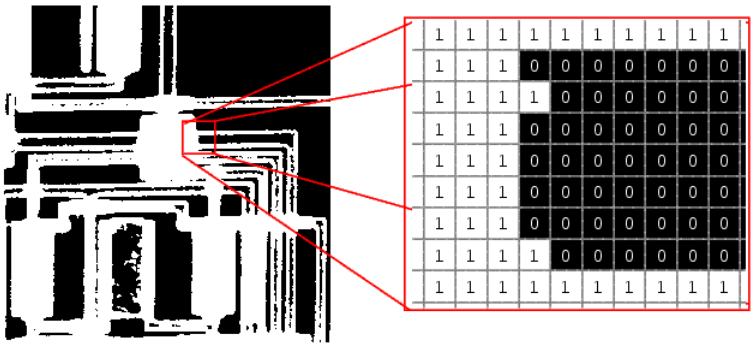
An intensity image, also known as a grayscale image, is a data matrix, I, whose values represent intensities within some range. MATLAB stores an intensity image as a individual matrix, with each element of the matrix corresponding to one image pixel. The matrix can be of class uint8, uint16, int16, single, or double.While intensity images are rarely saved with a colormap, MATLAB uses a colormap to display them. For a matrix of class single or double, using the default grayscale colormap, the intensity 0 represents black and the intensity 1 represents white. For a matrix of type uint8, uint16, or int16, the intensity intmin(class(I)) represents black and the intensity intmax(class(I)) represents white.



The images used here are provided by the authors.

Binary images

In a binary image, also known as a bilevel image, each pixel assumes one of only two discrete values: 1 or 0. A binary image is stored as a logical array.

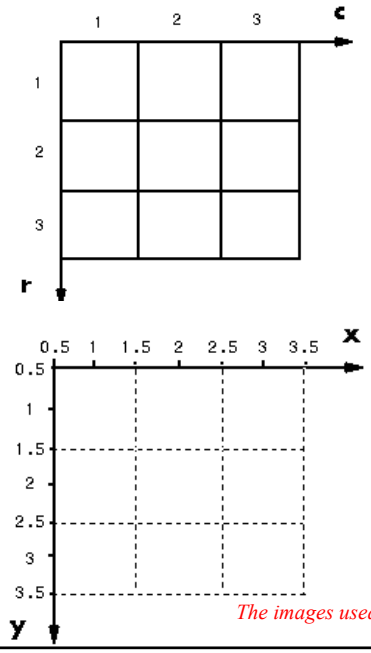


The images used here are provided by the authors.

RGB images

A truecolor image, also known as an RGB image, is stored in MATLAB as an m-by-n-by-3 data array that defines red, green, and blue color components for each individual pixel. Truecolor images do not use a colormap. The color of each pixel is determined by the combination of the red, green, and blue intensities stored in each color plane at the pixel's location. Graphics file formats store truecolor images as 24-bit images, where the red, green, and blue components are 8 bits each. This yields a potential of 16 million colors. The precision with which a real-life image can be replicated has led to the commonly used term truecolor image.

The images used here are provided by the authors.



Pixel Coordinates

Coordinate Systems

Spatial Coordinates

The images used here are provided by the authors.

A sample PGM image:

```
P2
# Created by Paint Shop Pro 6
4 4
16
15 3 2  13
5 10 11  8
9 6  7 12
4 15 14  15
```

Accessing Matrix Elements

- Subscripts
  - Uses parentheses to indicate subscripts
  - A(1,4) + A(2,4) + A(3,4) + A(4,4) returns 34
- Out of range indices
  - Trying to read an element out of range produces an error message
  - Trying to assign a value to an element out of range expands the matrix !!

```
16  3  2  13  0
 5 10 11   8  0
 9  6  7  12  0
 4 15 14   1 17
```

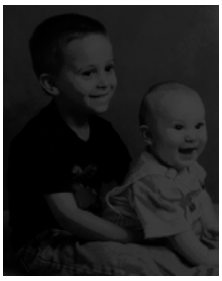
A(5,4) = 17 produces

The Colon Operator

- Examples
  - 1:5 produces 1 2 3 4 5
  - 20:-3:0 produces 20 17 14 11 8 5 2
  - 0:pi/4:pi produces 0 0.7854 1.5708 2.3562 3.1416
- Accessing portions of a matrix
  - A(1:k,j) references the first k elements in the j column
  - A(:,end) references all elements in the last column
  - How could you reference all elements in the last row?

Example:

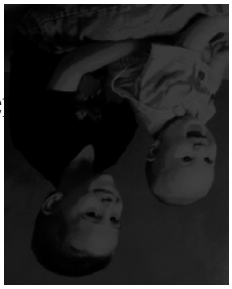
```
>> f = imread('kids.tif');
>> imshow(f)
```

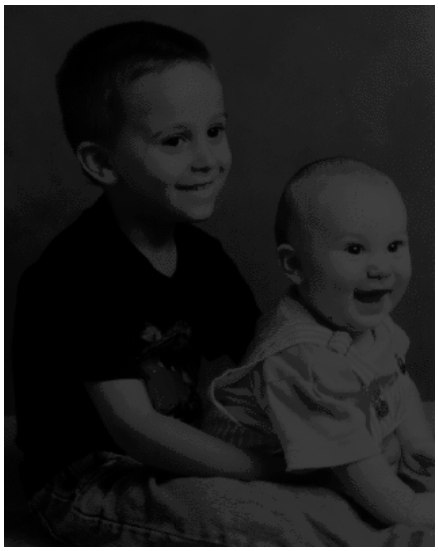


```
>> fp = f(end:-1:1, :);
```

This command, flips the image ve

```
>> imshow(fp)
```

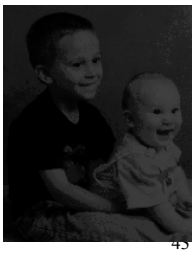


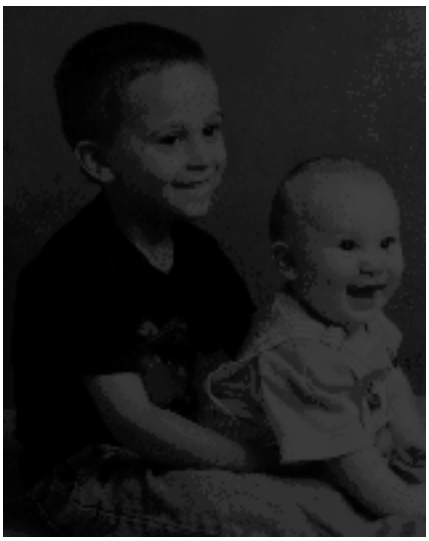


Example:

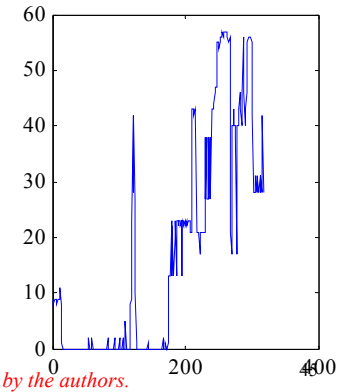
>> fc = f(100:300, 100:300);
This cuts the pixels from 100 to 300 out from the original image, f.
>> imshow(fc)

>> fs = f(1:2:end, 1:2:end);
This command create a subsampled image shown below.
>> imshow(fs)





>> plot(f(200,:))
This plots a horizontal scan line through row 200, almost the middle.



Chapter 2
Fundamentals

Expressions and Functions

- Expressions and functions obey algebraic rules
  z = sqrt(besselk(4/3,rho-i))
  z = 0.3730+ 0.3214i
- Some important constant functions

`pi`	3.14159265...
`i`	Imaginary unit, $\sqrt{-1}$
`j`	Same as `i`
`eps`	Floating-point relative precision, $\varepsilon = 2^{-52}$
`realmin`	Smallest floating-point number, 2^{-1022}
`realmax`	Largest floating-point number, $(2-\varepsilon)2^{1023}$
`Inf`	Infinity
`NaN`	Not-a-number

Generating Matrices

```
>> Z=zeros(2,4)

Z =

     0     0     0     0
     0     0     0     0

>> F = 5 * ones(3,3)

F =

     5     5     5
     5     5     5
     5     5     5
```

```
>> N = fix(10*rand(1,10))

N =

    9    2    6    4    8    7    4    0    8    4

>> R = rand(4,4)

R =

    0.6154    0.1763    0.4103    0.8132
    0.7919    0.4057    0.8936    0.0099
    0.9218    0.9355    0.0579    0.1389
    0.7382    0.9169    0.3529    0.2028
```


- **Concatenation, using our magic square A**

$$A = \begin{bmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 5 & 15 & 14 & 1 \end{bmatrix}$$

```
B = [A  A+32; A+48  A+16]
```

The result is an 8-by-8 matrix, obtained by joining the four submatrices.

```
B =

    16     3     2    13    48    35    34    45
     5    10    11     8    37    42    43    40
     9     6     7    12    41    38    39    44
     4    15    14     1    36    47    46    33
    64    51    50    61    32    19    18    29
    53    58    59    56    21    26    27    24
    57    54    55    60    25    22    23    28
    52    63    62    49    20    31    30    17
```

- **This isn't a magic square but the columns add up to the same value**

```
sum(B)

ans =
   260    260    260    260    260    260    260    260
```


Various Matrix Operations - 2

$$A = \begin{bmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 5 & 15 & 14 & 1 \end{bmatrix}$$

Deleting rows and columns

- A(: , 2) = [ ] removes the second column
- How would you remove the last row?
- Single elements can only be removed from vectors

- Some operations with transpose

•If you tried to apply the determinant operation, you would find det(A) = 0, so this matrix is not invertible; if you tried inv(A) you would get an error

```
A + A'

ans =
    32     8    11    17
     8    20    17    23
    11    17    14    26
    17    23    26     2
```

```
A'*A

ans =
   378   212   206   360
   212   370   368   206
   206   368   370   212
   360   206   212   378
```


Chapter 2
Fundamentals

Various Matrix Operations - 3

- The eigenvalue contains a 0, indicating singularity

```
e = eig(A)

e =
   34.0000
    8.0000
    0.0000
   -8.0000
```

```
P =
    0.4706    0.0882    0.0588    0.3824
    0.1471    0.2941    0.3235    0.2353
    0.2647    0.1765    0.2059    0.3529
    0.1176    0.4412    0.4118    0.0294
```

- P = A/34 is doubly stochastic, as shown above
- P^5 (raised to the fifth power) converges towards ¼, as k in p^k gets larger the values approach ¼

0.2507	0.2495	0.2494	0.2504
0.2497	0.2501	0.2502	0.2500
0.2500	0.2498	0.2499	0.2503
0.2496	0.2506	0.2505	0.2493

Array Operations

+	Addition
-	Subtraction
.*	Element-by-element multiplication
./	Element-by-element division
.\	Element-by-element left division
.^	Element-by-element power
.'	Unconjugated array transpose

- **For example, to square the elements of A, enter the expression A.*A**

256	9	4	169
25	100	121	64
81	36	49	144
16	225	196	1

Chapter 2
Fundamentals

Building Tables

- An example

  Let n = (0:9)'

  Let pows = [n n.^2 2.^n]

```
pows =
     0     0     1
     1     1     2
     2     4     4
     3     9     8
     4    16    16
     5    25    32
     6    36    64
     7    49   128
     8    64   256
     9    81   512
```

```
format short g
x = (1:0.1:2)';
logs = [x log10(x)]
```

Another example

```
logs =
    1.0          0
    1.1    0.04139
    1.2    0.07918
    1.3    0.11394
    1.4    0.14613
    1.5    0.17609
    1.6    0.20412
    1.7    0.23045
    1.8    0.25527
    1.9    0.27875
    2.0    0.30103
```


Chapter 2
Fundamentals

Programming in MATLAB

- Control structures
  - Selection (if and switch)
  - Repetition (for, while, break, continue)
  - Other (try … catch, return)
- Dynamic Structures
- Scripts and M files
- User defined functions
- Two examples
  - Finding the periodicity of sunspots
  - Multiplying polynomials using FFT

The if command

```
if rem(n,2) ~= 0
   M = odd_magic(n)
elseif rem(n,4) ~= 0
   M = single_even_magic(n)
else
   M = double_even_magic(n)
end
```


- An example

```
if A > B
   'greater'
elseif A < B
   'less'
elseif A == B
   'equal'
else
   error('Unexpected situation')
end
```

Useful Boolean tests for matrices

```
isequal
isempty
all
any
```

The switch command

```
switch (rem(n,4)==0) + (rem(n,2)==0)
   case 0
      M = odd_magic(n)
   case 1
      M = single_even_magic(n)
   case 2
      M = double_even_magic(n)
   otherwise
      error('This is impossible')
end
```

- Note: the break command in C++ is not required in MATLAB

Commands for repetition

- **The for command (notice the required 'end')**

```
for i = 1:m
   for j = 1:n
      H(i,j) = 1/(i+j);
   end
end
```

- **The while command (also requires 'end')**

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
    x = (a+b)/2;
    fx = x^3-2*x-5;
    if sign(fx) == sign(fa)
        a = x; fa = fx;
    else
        b = x; fb = fx;
    end
end
```

Does anyone recognize what this code fragment does?

Continue and Break

- The continue command

What does this code fragment do?

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) | strncmp(line,'%',1)
        continue
    end
    count = count + 1;
end
disp(sprintf('%d lines',count));
```

- The break command

Here is the finding the solution of a polynomial using bisection; why is the 'break' command an improvement?

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
   x = (a+b)/2;
   fx = x^3-2*x-5;
   if fx == 0
      break
   elseif sign(fx) == sign(fa)
      a = x; fa = fx;
   else
      b = x; fb = fx;
   end
end
```


try … catch and return

```
try
   statement
   ...
   statement
catch
   statement
   ...
   statement
end
```

You can examine the error using lasterr

An error in the exception handler causes the program to terminate

- The return command
  - Terminates execution
  - If inside a user defined function, returns to the calling environment
  - Otherwise returns to keyboard input