MATLAB practice

IPCV 2006, Budapest

Szabolcs Sergyán, László Csink

sergyan.szabolcs@nik.bmf.hu, csink.laszlo@nik.bmf.hu

Budapest Tech

Contents

Chapter 1

Fundamentals

Content

- Digital Image Representation
- Reading Images
- Displaying Images
- Writing Images
- Data Classes
- Image Types
- Converting between Data Classes and Image Types
- Array Indexing
- Some Important Standard Arrays
- Introduction to M-Function Programming

Digital Image Representation

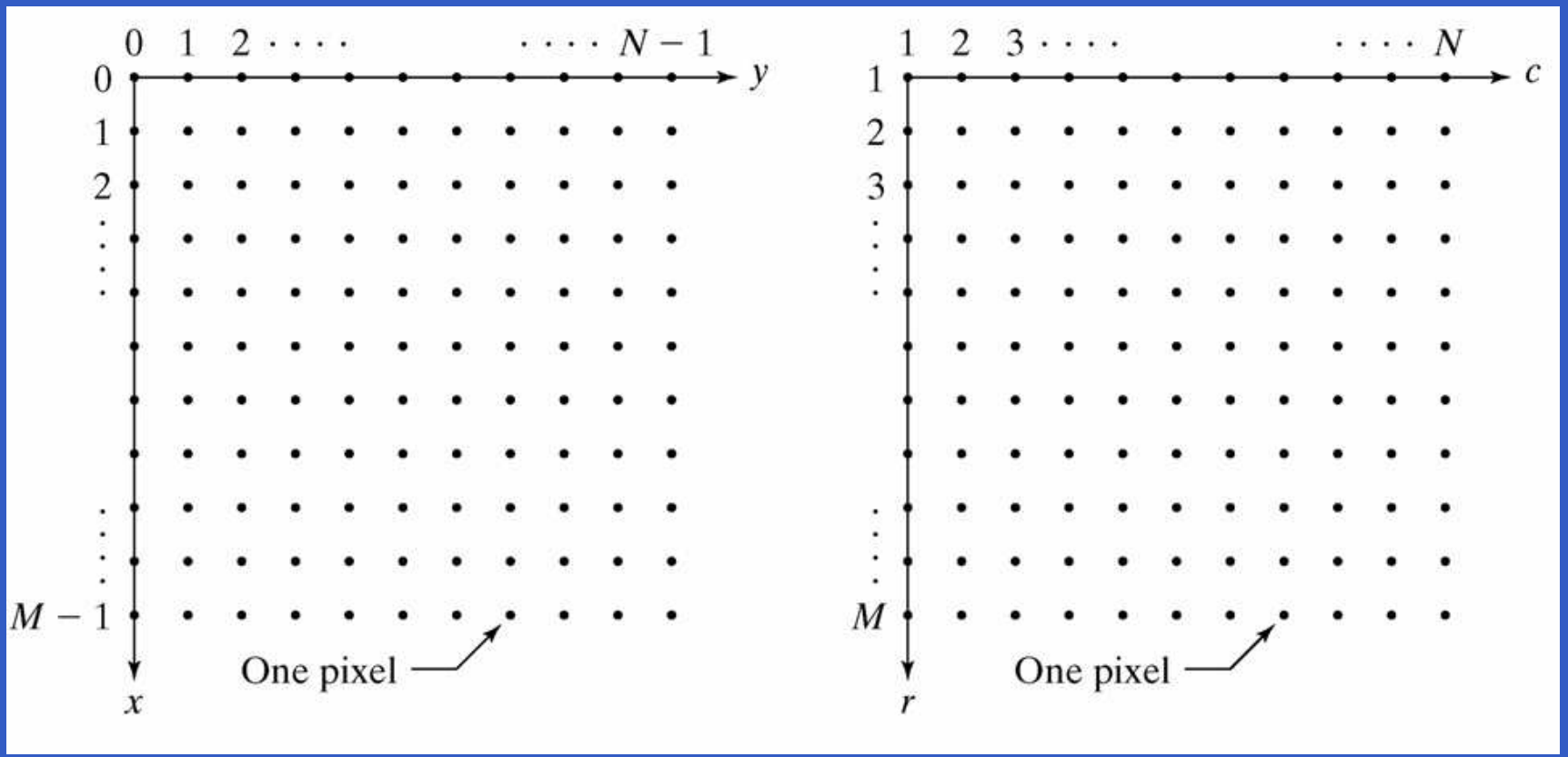
An image may be defined as a two-dimensional function,

$$f(x, y),$$

where x and y are spatial coordinates, and
f is the intensity of the image at (x, y) point.

When x, y, and the amplitude values of f are all finite, discrete quantities, we call the image a *digital image*.

Coordinate Conventions



in many image processing books

in the Image Processing Toolbox

Images as Matrices

A digital image can be represented as a MATLAB matrix:

$$\mathrm{f} = \begin{bmatrix} \mathrm{f}(1,1) & \mathrm{f}(1,2) & \cdots & \mathrm{f}(1,\mathrm{N}) \\ \mathrm{f}(2,1) & \mathrm{f}(2,2) & \cdots & \mathrm{f}(2,\mathrm{N}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{f}(\mathrm{M},1) & \mathrm{f}(\mathrm{M},2) & \cdots & \mathrm{f}(\mathrm{M},\mathrm{N}) \end{bmatrix}$$

Reading Images

```
imread('filename')
```

Some examples:

- `f=imread('chestxray.jpg');`
- `f=imread('D:\myimages\chestxray.jpg');`
- `f=imread('.\myimages\chestxray.jpg');`

Supported Image Formats

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`

`size` function

```
size(imagematrix)
```

```
>> size(f)

ans =
   494   600

>> [M,N]=size(f);

>> whos f
  Name      Size            Bytes  Class
  f         494x600        296400  uint8 array
Grand total is 296400 elements using 296400 bytes
```

Displaying Images

`imshow(f,G)`

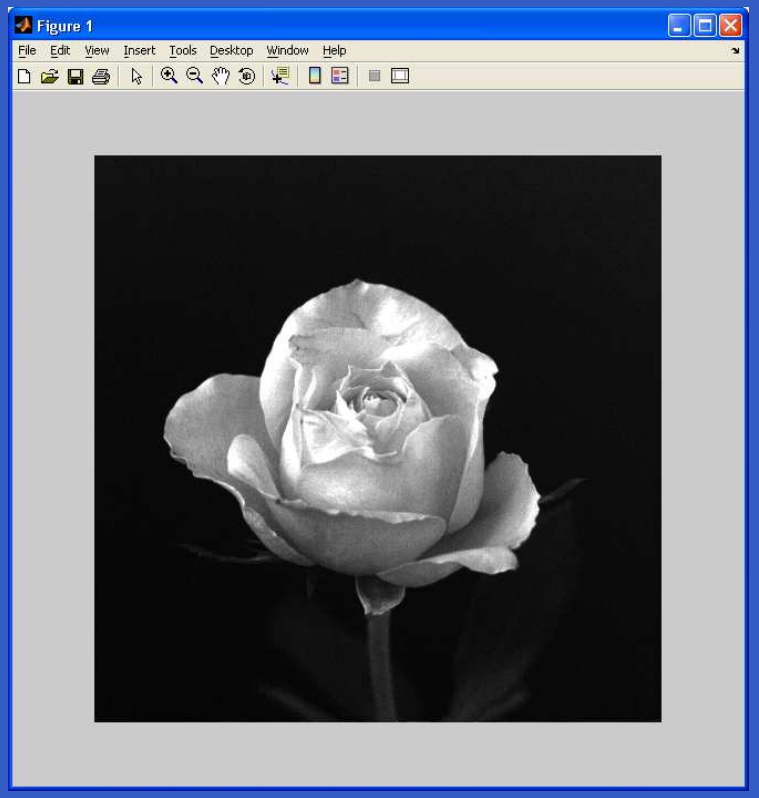
- `imshow(f, [low,high])` displays as black all values less than or equal to `low`, and as white all values greater than or equal to `high`.
- `imshow(f, [])` sets variable `low` to the minimum value of array `f` and `high` to its maximum value.

Displaying Images

An image with low dinamic range using by `imshow(f)`, and the result of scaling by using `imshow(f, [])`.

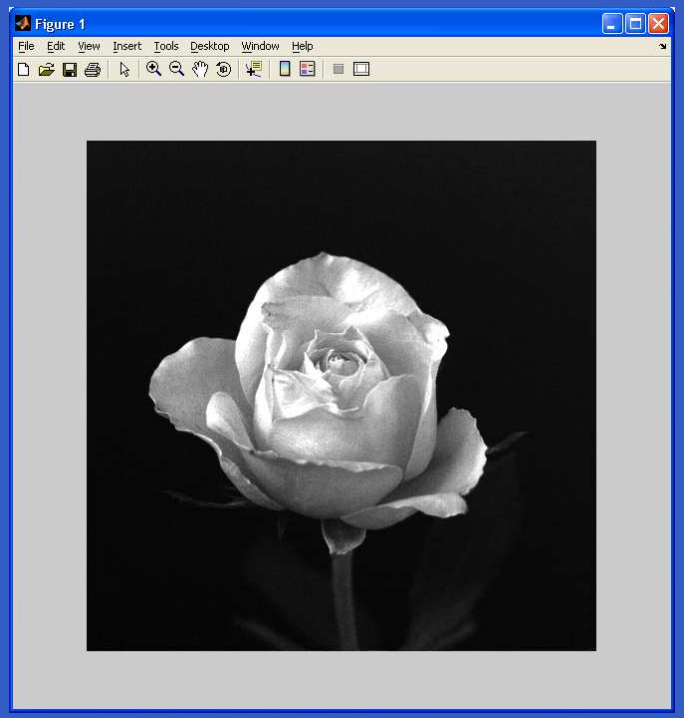
Displaying Images

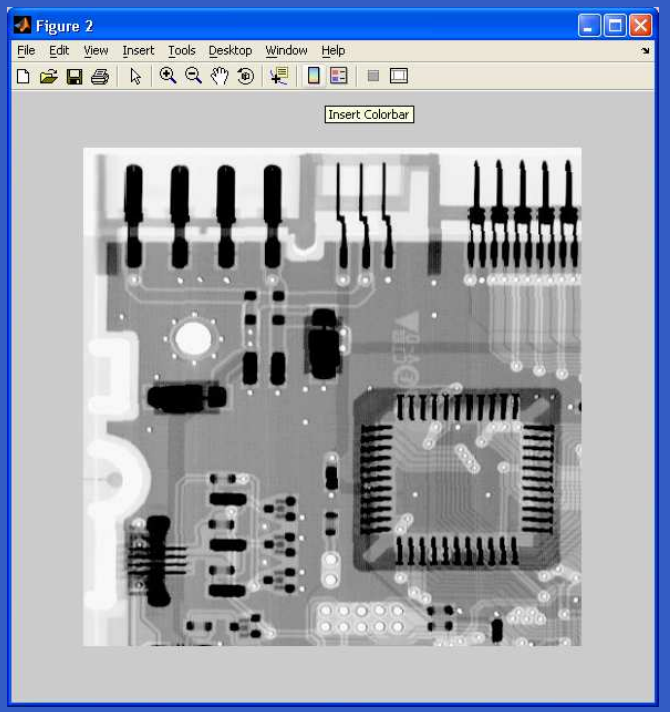
```
>> f=imread('rose_512.tif');
>> imshow(f)
```



Displaying Images

```
>> f=imread('rose_512.tif');
>> g=imread('cktboard.tif');
>> imshow(f), figure, imshow(g)
```





Writing Images

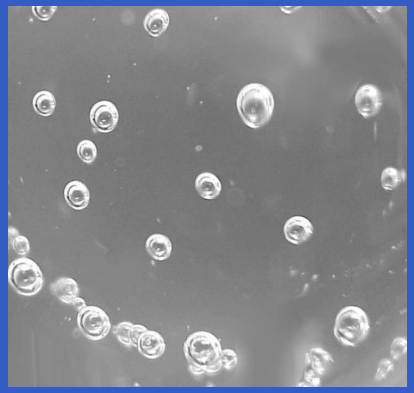
```
imwrite(f, 'filename')
```

- `imwrite(f, 'patient10_run1', 'tif')`
- `imwrite(f, 'patient10_run1.tif')`

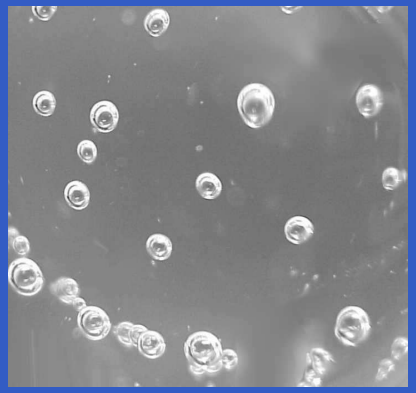
```
imwrite(f, 'filename.jpg', 'quality', q)
```

The lower the number q the higher the degradation due to JPEG compression.

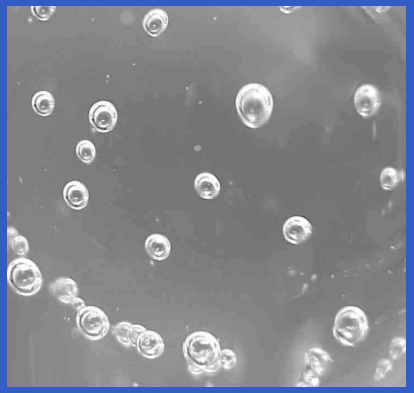
Writing Images



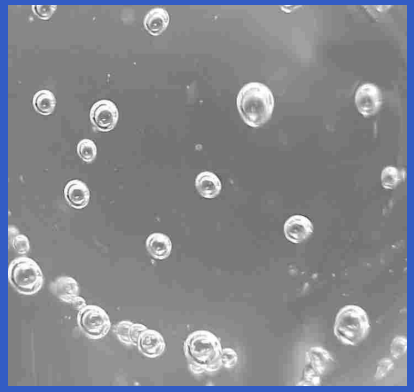
$q = 100$



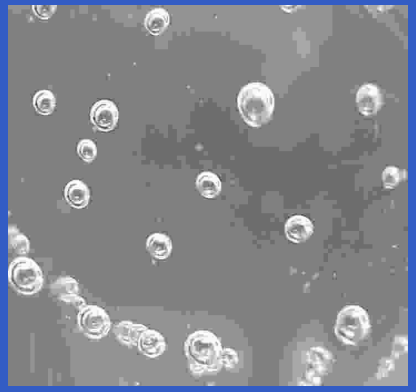
$q = 50$



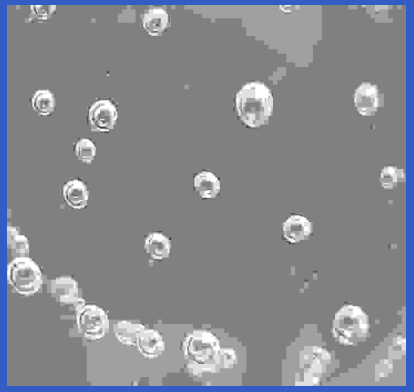
$q = 25$



$q = 15$



$q = 5$



$q = 0$

Writing Images

`imfinfo filename`

```
>> imfinfo bubbles25.jpg
ans =
           Filename: 'bubbles25.jpg'
        FileModDate: '02-Feb-2005 09:34:50'
           FileSize: 13354
             Format: 'jpg'
      FormatVersion: ''
              Width: 720
             Height: 688
           BitDepth: 8
          ColorType: 'grayscale'
    FormatSignature: ''
    NumberOfSamples: 1
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {}
```

Writing Images

```
>> K=imfinfo('bubbles25.jpg');
>> image_bytes=K.Width*K.Height*K.BitDepth/8;
>> compressed_bytes=K.FileSize;
>> compression_ratio=image_bytes/compressed_bytes

compression_ratio =

   37.0945
```

Writing Images

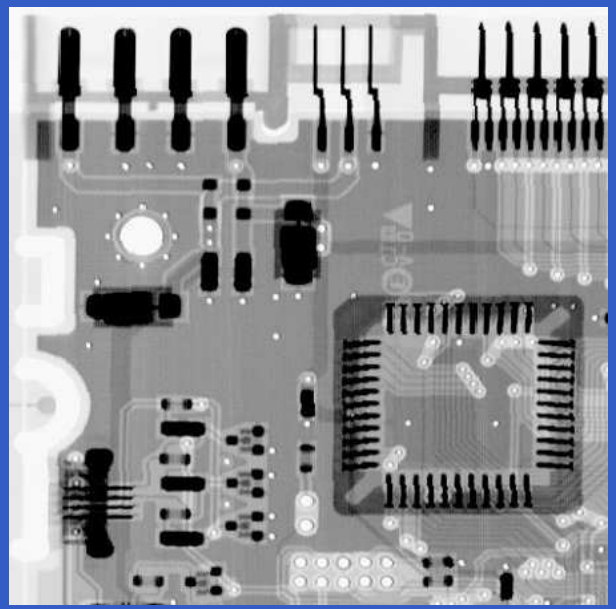
```
imwrite(g, 'filename.tif', ...
        'compression', 'parameter', ...
        'resolution', [colres rowres])
```

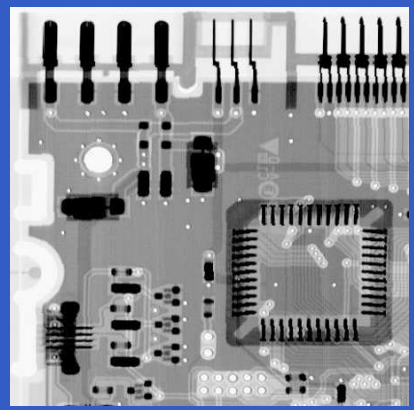
`'parameter'`:	`'none'`	no compression
	`'packbits'`	packbits compression
	`'ccitt'`	ccitt compression

`[colres rowres]` contains two integers that give the column and row resolution in dots-per-unit (the default values are [72 72]).

Writing Images

```
>> f=imread('cktboard.tif');
>> res=round(200*2.25/1.5);
>> imwrite(f, 'sf.tif', 'compression', ...
           'none', 'resolution', res)
```





Writing Images

```
print -fno -dfileformat -rresno filename
```

no	figure number in figure window
fileformat	file format (in the earlier table)
resno	resolution in dpi

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).

Image Types

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

Intensity Images

An *intensity image* is a data matrix whose values have been scaled to represent intensities. When the elements of an intensity image are of class `uint8`, or class `uint16`, they have integer values in the range [0,255] and [0,65535], respectively. If the image is of class `double`, the values are floating-point numbers. Values of scaled, class `double` intensity images are in the range [0,1] by convention.

Binary Images

A *binary image* is a *logical* array of 0s and 1s.

A numaric array is converted to binary using function `logical`.

```
B=logical(A)
```

To test if an array is logical we use the `islogical` function:

```
islogical(C)
```

If `C` is a logical array, this function returns a `1`. Otherwise it returns a `0`.

Converting between Data Classes

```
B=data_class_name(A)
```

If `C` is an array of class `double` in which all values are in the range [0,255], it can be converted to an `uint8` array with the command `D=uint8(C)`.

If an array of class `double` has any values outside the range [0,255] and it is converted to class `uint8`, MATLAB converts to 0 all values that are less than 0, and converts to 255 all values that are greater than 255.

Converting between Image Classes and Types

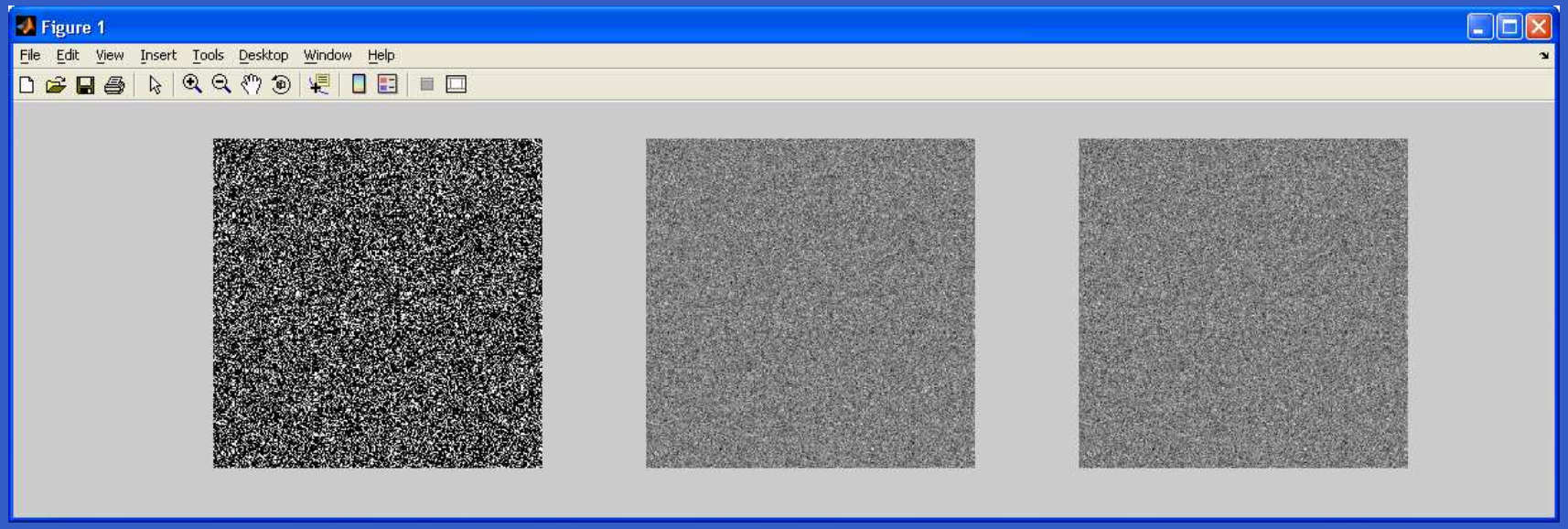
Name	Converts Input to:	Valid Input Image Data Classes
`im2uint8`	`uint8`	`logical`, `uint8`, `uint16`, and `double`
`im2uint16`	`uint16`	`logical`, `uint8`, `unit16`, and `double`
`mat2gray`	`double` (in range [0,1])	`double`
`im2double`	`double`	`logical`, `uint8`, `uint16`, and `double`
`im2bw`	`logical`	`uint8`, `uint16`, and `double`

Converting between Image Classes and Types

```
>> f=[-0.5 0.5;0.75 1.5]

f =

   -0.5000    0.5000
    0.7500    1.5000

>> g=im2uint8(f)

g =

     0  128
   191  255
```

Converting between Image Classes and Types

```
>> A=randn(252);
>> B=mat2gray(A);
>> subplot(1,3,1), imshow(A), ...
   subplot(1,3,2), imshow(A, []), ...
   subplot(1,3,3), imshow(B)
```



Converting between Image Classes and Types

```
>> h=uint8([25 50; 128 200]);
>> g=im2double(h)

g =

    0.0980    0.1961
    0.5020    0.7843
```

Converting between Image Classes and Types

```
>> f=[1 2; 3 4];
>> g=mat2gray(f)

g =

         0    0.3333
    0.6667    1.0000

>> gb=im2bw(g, 0.6)

gb =

     0     0
     1     1
```

Converting between Image Classes and Types

```
>> gb=f>2

gb =

     0     0
     1     1

>> gbv=islogical(gb)

gbv =

     1
```

Array Indexing

- Vector Indexing
- Matrix Indexing
- Selecting Array Dimensions

Vector Indexing

```
>> v=[1 3 5 7 9]

v =
     1     3     5     7     9

>> v(2)

ans =
     3

>> w=v.'

w =
     1
     3
     5
     7
     9
```

Vector Indexing

```
>> v(1:3)

ans =
     1     3     5

>> v(2:4)

ans =
     3     5     7

>> v(3:end)

ans =
     5     7     9
```

Vector Indexing

```
>> v(:)

ans =
     1
     3
     5
     7
     9

>> v(1:2:end)

ans =
     1     5     9

>> v(end:-2:1)

ans =
     9     5     1
```

Vector Indexing

`linspace(a, b, n)`

```
>> x=linspace(1,5,3)

x =
     1     3     5

>> v(x)

ans =
     1     5     9

>> v([1 4 5])

ans =
     1     7     9
```

Matrix Indexing

```
>> A=[1 2 3; 4 5 6; 7 8 9]

A =
     1     2     3
     4     5     6
     7     8     9

>> A(2,3)

ans =
     6
```

Matrix Indexing

```
>> C3=A(:,3)

C3 =
     3
     6
     9

>> R2=A(2,:)

R2 =
     4     5     6

>> T2=A(1:2,1:3)

T2 =
     1     2     3
     4     5     6
```

Matrix Indexing

```
>> B=A;
>> B(:,3)=0

B =
     1     2     0
     4     5     0
     7     8     0
```

Matrix Indexing

```
>> A(end,end)
ans =
     9

>> A(end,end-2)
ans =
     7

>> A(2:end,end:-2:1)
ans =
     6     4
     9     7

>> E=A([1 3],[2 3])
E =
     2     3
     8     9
```

Matrix Indexing

```
>> D=logical([1 0 0; 0 0 1; 0 0 0])

D =

     1     0     0
     0     0     1
     0     0     0

>> A(D)

ans =

     1
     6
```

Matrix Indexing

```
>> v=T2(:)

v =

     1
     4
     2
     5
     3
     6
```

Matrix Indexing

```
>> s=sum(A(:))

s =
     45

>> s1=sum(A)

s1 =
     12     15     18

>> s2=sum(sum(A))

s2 =
     45
```

Matrix Indexing

```
>> f=imread('rose.tif');
>> fp=f(end:-1:1,:);
```

Matrix Indexing

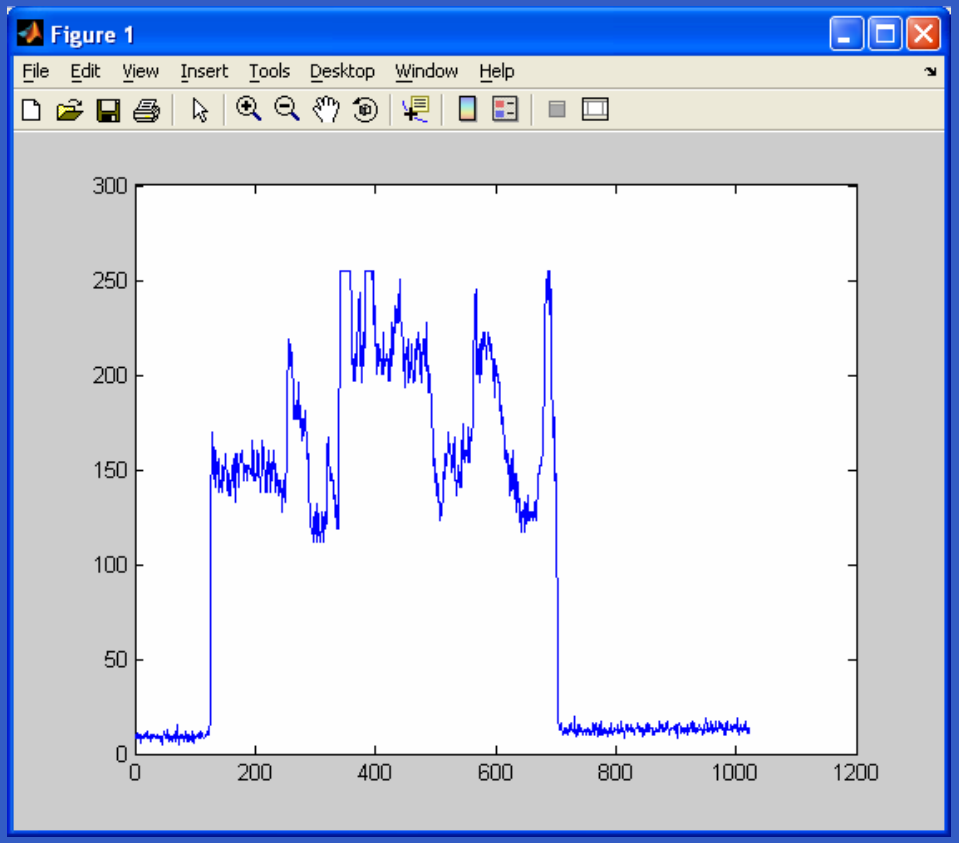
```
>> fc=f(257:768,257:768);
```

Matrix Indexing

```
>> fs=f(1:8:end,1:8:end);
```

Matrix Indexing

```
>> plot(f(512,:))
```



Selecting Array Dimensions

```
operation(A, dim)
```

where `operation` denotes an applicable MATLAB operation, `A` is an array and `dim` is a scalar.

```
>> k=size(A,1);
```

gives the size of `A` along its first dimension.

In the previous example we could have written the command as

```
>> plot(f(size(f,1)/2,:))
```

Function `ndims`, with syntax `d=ndims(A)` gives the number of dimensions of array `A`.

Some Important Standard Arrays

`zeros(M,N)`	generates an `M`×`N` matrix of 0s of class `double`.
`ones(M,N)`	generates an `M`×`N` matrix of 1s of class `double`.
`true(M,N)`	generates an `M`×`N` `logical` matrix of 1s.
`false(M,N)`	generates an `M`×`N` `logical` matrix of 0s.
`magic(M)`	generates an `M`×`M` "magic square".
`rand(M,N)`	generates an `M`×`N` matrix whose entries are uniformly distributed random numbers in the interval [0,1].
`randn(M,N)`	generates an `M`×`N` matrix whose numbers are normally distributed random numbers with mean 0 and variance 1.

Some Important Standard Arrays

```
>> A=5*ones(3)
A =
     5     5     5
     5     5     5
     5     5     5

>> magic(3)
ans =
     8     1     6
     3     5     7
     4     9     2

>> B=rand(2,4)
B =
    0.9501    0.6068    0.8913    0.4565
    0.2311    0.4860    0.7621    0.0185
```

M-Function Programming

- M-Files
- Operators
- Flow Control
- Code Optimization
- Interactive I/O
- Cell Arrays and Structures

M-Files

M-Files in MATLAB can be

scripts that simply execute a series of MATLAB statements, or they can be

functions that can accept argumens and can produce one or more outputs.

M-Files

The components of a function M-file are

- The function definition line
- The H1 line
- Help text
- The function body
- Comments

M-Files

```
function [G,x] = planerot(x)
%PLANEROT Givens plane rotation.
%   [G,Y] = PLANEROT(X), where X is a 2-component column vector,
%   returns a 2-by-2 orthogonal matrix G so that Y=G*X has Y(2)=0.
%
%   Class support for input X:
%      float: double, single

%   Copyright 1984-2004 The MathWorks, Inc.
%   $Revision: 5.10.4.1 $  $Date: 2004/04/10 23:30:05 $

if x(2) ~= 0
   r = norm(x);
   G = [x'; -x(2) x(1)]/r;
   x = [r; 0];
else
   G = eye(2,class(x));
end
```

Operators

- Arithmetic Operators
- Relational Operators
- Logical Operators and Functions

Arithmetic Operators

Operator	Name	MATLAB Function	Comments and Examples
`+`	Array and matrix addition	`plus(A,B)`	`a+b`, `A+B`, or `a+A`.
`-`	Array and matrix subtraction	`minus(A,B)`	`a-b`, `A-B`, `A-a`.
`.*`	Array multiplication	`times(A,B)`	`C=A.*B`, `C(I,J)=A(I,J)*B(I,J)`.
`*`	Matrix multiplication	`mtimes(A,B)`	`A*B`, standard matrix multiplication, or `a*A`, multiplication of a scalar times all elements of `A`.

Arithmetic Operators

Operator	Name	MATLAB Function	Comments and Examples
`./`	Array right division	`rdivide(A,B)`	`C=A./B`, `C(I,J)=A(I,J)/B(I,J)`.
`.\`	Array left division	`ldivide(A,B)`	`C=A.\B`, `C(I,J)=B(I,J)/A(I,J)`.
`/`	Matrix right division	`mrdivide(A,B)`	`A/B` is roughly the same as `A*inv(B)`, depending on computational accuracy.
`\`	Matrix left division	`mldivide(A,B)`	`A\B` is roughly the same as `inv(A)*B`, depending on computational accuracy.

Arithmetic Operators

Operator	Name	MATLAB Function	Comments and Examples
`.^`	Array power	`power(A,B)`	If `C=A.^B`, then `C(I,J)=A(I,J)^B(I,J)`.
`^`	Matrix power	`mpower(A,B)`	Square matrix to the scalar power, or scalar to the square matrix power.
`.'`	Vector and matrix transpose	`transpose(A)`	`A.'`. Standard vector and matrix transpose.
`'`	Vector and matrix complex conjugate transpose	`ctranspose(A)`	`A'`. Standard vector and matrix conjugate transpose.

Arithmetic Operators

Operator	Name	MATLAB Function	Comments and Examples
`+`	Unary plus	`uplus(A)`	`+A` is the same as `0+A`.
`-`	Unary minus	`uminus(A)`	`-A` is the same as `0-A` or `-1*A`.
`:`	Colon		Discussed earlier.

Image Arithmetic Functions

Function	Description
`imadd`	Adds two images; or adds a constant to an image.
`imsubtract`	Subtracts two images; or subtracts a constant from an image.
`immultiply`	Multiplies two image, where the multiplication is carried out between pairs of corresponding image elements; or multiplies a constant times an image.
`imdivide`	Divides two images, where the division is carried out between pairs of corresponding image elements; or divides an image by a constant.
`imabsdiff`	Computes the absolute difference between two images.
`imcomplement`	Complements an image.
`imlincomb`	Computes a linear combination of two or more images.

An Example

```
function [p,pmax,pmin,pn]=improd(f,g)
%IMPROD Computes the product of two images.
%  [P,PMAX,PMIN,PN]=IMPROD(F,G) outputs the
%  element-by-element product of two images,
%  F and G, the product maximum and minimum
%  values, and a normalized product array with
%  values in the range [0,1]. The input images
%  must be of the same size. They can be of
%  class uint8, uint 16, or double. The outputs
%  are of class double.
```

An Example

```
fd=double(f);
gd=double(g);
p=fd.*gd;
pmax=max(p(:));
pmin=min(p(:));
pn=mat2gray(p);
```

An Example

```
>> f=[1 2;3 4]; g=[1 2;2 1];
>> [p,pmax,pmin,pn]=improd(f,g)

p =
     1     4
     6     4

pmax =
     6

pmin =
     1

pn =
          0    0.6000
    1.0000    0.6000
```

An Example

```
>> help improd
 IMPROD Computes the product of two images.
   [P,PMAX,PMIN,PN]=IMPROD(F,G) outputs the
   element-by-element product of two images,
   F and G, the product maximum and minimum
   values, and a normalized product array with
   values in the range [0,1]. The input images
   must be of the same size. They can be of
   class uint8, uint 16, or double. The outputs
   are of class double.

>> help DIPUM
IMPROD Computes the product of two images.
```

Some Words about `max`

`C=max(A)`	If `A` is a vector, `max(A)` returns its largest element; if `A` is a matrix, then `max(A)` treats the columns of `A` as vectors and returns a row vector containing the maximum element from each column.
`C=max(A,B)`	Returns an array the same size as `A` and `B` with the largest elements taken from A or B.
`C=max(A,[ ],dim)`	Returns the largest elements along the dimension of `A` specified by `dim`.
`[C,I]=max(...)`	Finds the indices of the maximum values of A, and returns them in output vector `I`. If there are several identical maximum values, the index of the first one found is returned. The dots indicate the syntax used on the right of any of the previous three forms.

Relational Operations

Operator	Name
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than of equal to
==	Equal to
~=	Not equal to

Relational Operators

```
>> A=[1 2 3;4 5 6;7 8 9];
>> B=[0 2 4;3 5 6;3 4 9];
>> A==B

ans =
     0     1     0
     0     1     1
     0     0     1

>> A>=B

ans =
     1     1     0
     1     1     1
     1     1     1
```

Logical Operators

Operator	Name
&	AND
|	OR
~	NOT

Logical Operators

```
>> A=[1 2 0;0 4 5];
>> B=[1 -2 3;0 1 1];
>> A&B

ans =

     1     1     0
     0     1     1
```

Logical Functions

Function	Comments
`xor`	The `xor` function returns a `1` only if both operands are logically different; otherwise `xor` returns a `0`.
`all`	The `all` function returns a `1` if all the elements in a vector are nonzero; otherwise `all` returns a `0`. This function operates columnwise on matrices.
`any`	The `any` function returns a `1` if any of the elements in a vector is nonzero; otherwise `any` returns a `0`. This function operates columnwise on matrices.

Logical Functions

```
>> A=[1 2 3;4 5 6];
>> B=[0 -1 1;0 0 1];
>> xor(A,B)
ans =
     1     0     0
     1     1     0
>> all(A)
ans =
     1     1     1
>> any(A)
ans =
     1     1     1
>> all(B)
ans =
     0     0     1
>> any(B)
ans =
     0     1     1
```

Logical Functions

Function	Description
`iscell(C)`	True if `C` is a cell array.
`iscellstr(s)`	True if `s` is a cell array of strings.
`ischar(s)`	True if `s` is a character string.
`isempty(A)`	True if `A` is the empty array, `[]`.
`isequal(A,B)`	True if `A` and `B` have identical elements and dimensions.
`isfield(S,'name')`	True if `'name'` is a field of structure `S`.
`isfinite(A)`	True in the locations of array `A` that are finite.
`isinf(A)`	True in the locations of array `A` that are infinite.
`isletter(A)`	True in the locations of `A` that are letters of the alphabet.

Logical Functions

Function	Description
`islogical(A)`	True if `A` is a logical array.
`ismember(A,B)`	True in locations where elements of `A` are also in `B`.
`isnan(A)`	True in the locations of `A` that are `NaNs`.
`isnumeric(A)`	True if `A` is a numeric array.
`isprime(A)`	True in locations of `A` that are prime numbers.
`isreal(A)`	True if the elements of `A` have no imaginary parts.
`isspace(A)`	True at locations where the elements of `A` are whitespace characters.
`issparse(A)`	True if `A` is a sparse matrix.
`isstruct(A)`	True if `S` is a structure.

Some Important Variables and Constants

Function	Value Returned
`ans`	Most recent answer (variable). If no output variable is assigned to an expression, MATLAB automatically stores the result in `ans`.
`eps`	Floating-point relative accuracy. This is the distance between 1.0 and the next largest number representable using double-precision floating point.
`i` (or `j`)	Imaginary unit, as in `1+2i`.
`NaN` or `nan`	Stands for Not-a-Number (e.g., 0/0).
`pi`	3.14159265358979
`realmax`	The largest floating-point number that your computer can represent.
`realmin`	The smallest floating-point number that your computer can represent.
`computer`	Your computer type.
`version`	MATLAB version string.

Flow Control

Statement	Description
`if`	`if`, together with `else` and `elseif`, executes a group of statements based on a specified logical condition.
`for`	Executes a group of statements a fixed (specified) number of times.
`while`	Executes a group of statements an indefinite number of times, based on a specified logical condition.
`break`	Terminates execution of a `for` or `while` loop.
`continue`	Passes control to the next iteration of a `for` or `while` loop, skipping any remaining statements in the body of the loop.
`switch`	`switch`, together with `case` and `otherwise`, executes different groups of statements, depending on a specified value or string.
`return`	Causes execution to return to the invoking function.
`try...catch`	Changes flow control if an error is detected during execution.

`if`, `else`, and `elseif`

```
if expression
   statements
end

if expression1
   statements1
elseif expression2
   statements2
else
   statements3
end
```

if, else, and elseif

```
function av=average(A)
%AVERAGE Computes the average value of an array.
%  AV=AVERAGE(A) computes the average value of
%  input array, A, which must be a 1-D or 2-D
%  array.

% Check the validity of the input. (Keep in mind
% that a 1-D array is a special case of a 2-D
% array.)
if ndims(A)>2
    error('The dimensions of the input cannot exceed 2.')
end

%Compute the average
av=sum(A(:))/length(A(:));
%or av=sum(A(:))/numel(A);
```

for

```
for index=start:increment:end
    statements
end
```

for

```
count=0;
for k=0:0.1:1
    count=count+1;
end
```

for

```
for q=0:5:100
    filename=sprintf('series_%3d.jpg',q);
    imwrite(f,filename,'quality',q);
end
```

for

```
function s=subim(f,m,n,rx,cy)
%SUBIM Extracts a subimage, s, from a given image, f.
%  The subimage is of size m-by-n, and the coordinates
%  of its top, left corner are (rx,cy).

s=zeros(m,n);
rowhigh=rx+m-1;
colhigh=cy+n-1;
xcount=0;
for r=rx:rowhigh
    xcount=xcount+1;
    ycount=0;
    for c=cy:colhigh
        ycount=ycount+1;
        s(xcount,ycount)=f(r,c);
    end
end
```

while

```
while expression
      statements
end
```

while

```
a=10;
b=5;
while a
    a=a-1;
    while b
        b=b-1;
    end
end
```

break

```
fid = fopen('fft.m','r');
s = '';
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line)
        break
    end
    s = strvcat(s,line);
end
disp(s)
```

continue

```
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));
```

switch

```
switch switch_expression
        case case_expression
              statement(s)
        case {case_expression1, case_expression2,...}
              statement(s)
        otherwise
              statement(s)
end
```

switch

```
switch newclass
    case 'uint8'
        g=im2uint8(f);
    case 'uint16'
        g=im2uint16(f);
    case 'double'
        g=im2double(f);
    otherwise
        error('Unknown or improper image class.')
end
```

return

```
function d = det(A)
%DET det(A) is the determinant of A.
if isempty(A)
    d = 1;
    return
else
    ...
end
```

try...catch

```
function matrix_multiply(A, B)
try
    A * B
catch
    errmsg = lasterr;
    if(strfind(errmsg, 'Inner matrix dimensions'))
        disp('** Wrong dimensions for matrix
              multiplication')
    elseif(strfind(errmsg, 'not defined for variables
                            of class'))
        disp('** Both arguments must be double matrices')
    end
end
```

Code Optimization

- Vectorizing Loops
- Preallocating Arrays

Vectorizing Loops

Vectorizing simply means converting `for` and `while` loops to equivalent vector or matrix operations.

A Simple Example

Suppose that we want to generate a 1-D function of the form

$$f(x) = A\sin(x/2\pi)$$

for $x = 0, 1, 2, \ldots, M-1$.

A `for` loop to implement this computation is

```
for x=1:M %Array indices in MATLAB cannot be 0.
    f(x)=A*sin((x-1)/(2*pi));
end
```

The vectorized code:

```
x=0:M-1;
f=A*sin(x/(2*pi));
```

2-D indexing

`[C,R]=meshgrid(c,r)`

```
>> c=[0 1];
>> r=[0 1 2];
>> [C,R]=meshgrid(c,r)
C =
     0     1
     0     1
     0     1
R =
     0     0
     1     1
     2     2
>> h=R.^2+C.^2
h =
     0     1
     1     2
     4     5
```

Comparison `for` loops vs. vectorization

```
function [rt,f,g]=twodsin(A,u0,v0,M,N)
%TWODSIN Compares for loops vs. vectorization.
%  The comparison is based on implementing the function
%  f(x,y)=Asin(u0x+v0y) for x=0,1,2,...,M-1 and
%  y=0,1,2,...,N-1. The inputs to the function are
%  M and N and the constants in the function.
```

Comparison for loops vs. vectorization

```
% First implement using for loops.

tic  %Start timing.

for r=1:M
    u0x=u0*(r-1);
    for c=1:N
        v0y=v0*(c-1);
        f(r,c)=A*sin(u0x+v0y);
    end
end

t1=toc;  %End timing.
```

Comparison for loops vs. vectorization

```
%Now implement using vectorization. Call the image g.

tic  %Start timing;

r=0:M-1;
c=0:N-1;
[C,R]=meshgrid(c,r);
g=A*sin(u0*R+v0*C);

t2=toc;  %End timing

% Compute the ratio of the two times.

rt=t1/(t2+eps);  % Use eps in case t2 is close to 0.
```

Comparison `for` loops vs. vectorization

```
>> [rt,f,g]=twodsin(1,1/(4*pi),1/(4*pi),512,512);
>> rt

rt =

   19.5833

>> g=mat2gray(g);
>> imshow(g)
```

Preallocating Arrays

```
tic
for i=1:1024
    for j=1:1024
        f(i,j)=i+2*j;
    end
end
toc
Elapsed time is 30.484000 seconds.

tic
g=zeros(1024);  %Preallocation
for i=1:1024
    for j=1:1024
        g(i,j)=i+2*j;
    end
end
toc
Elapsed time is 0.221000 seconds.
```

Interactive I/O

`disp(argument)`

```
>> A=[1 2;3 4];
>> disp(A)
     1     2
     3     4

>> sc='Digital Image Processing.';
>> disp(sc)
Digital Image Processing.

>> disp('This is another way to display text.')
This is another way to display text.
```

Interactive I/O

```
t=input('message')
```

```
t=input('messages','s')
```

```
>> t=input('Enter your data: ','s')
Enter your data: 1, 2, 4
t =
1, 2, 4
>> class(t)
ans =
char
>> size(t)
ans =
     1     7
```

Interactive I/O

```
>> n=str2num(t)

n =
     1     2     4

>> size(n)

ans =
     1     3

>> class(n)

ans =
double
```

Interactive I/O

```
[a,b,c,...]=strread(cstr,'format,...
                        'param','value')
```

```
>> t='12.6, x2y, z';
>> [a,b,c]=strread(t,'%f%q%q','delimiter',',')
a =
   12.6000
b =
    'x2y'
c =
    'z'
>> d=char(b)
d =
x2y
```

Save variables on disk

```
save('filename', 'var1', 'var2', ...)
```

saves the specified variables in `filename.mat`.

```
save('filename', '-struct', 's')
```

saves all fields of the scalar structure `s` as individual variables within the file `filename.mat`.

Load variables from disk

```
load('filename')
```

loads all the variables from `filename.mat`.

```
load('filename', 'X', 'Y', 'Z')
```

loads just the specified variables from the MAT-file.

```
S=load(...)
```

returns the contents of a MAT-file in the variable `S`. `S` is a struct containing fields that match the variables retrieved.

Display directory listing

```
files=dir('match')
```

returns the list of files with name `match` in the current directory to an m-by-1 structure with the fields

- `name`: Filename
- `date`: Modification date
- `bytes`: Number of bytes allocated to the file
- `isdir`: `1` if `name` is a directory; `0` if not

Cell Arrays

Cell array is a multidimensional array whose elements are copies of other arrays.

```
>> C={'gauss',[1 0;1 0],3}
C =
    'gauss'    [2x2 double]    [3]
>> C{1}
ans =
gauss
>> C{2}
ans =
     1     0
     1     0
>> C{3}
ans =
     3
```

Pass or return variable numbers of arguments

```
function varargout = foo(n)
```

returns a variable number of arguments from function `foo.m`.

```
function y = bar(varargin)
```

accepts a variable number of arguments into function `bar.m`.

The `varargin` and `varargout` statements are used only inside a function M-file to contain the optional arguments to the function. Each must be declared as the last argument to a function, collecting all the inputs or outputs from that point onwards. In the declaration, `varargin` and `varargout` must be lowercase.

Structures

Structures allow grouping of a collection of dissimilar data into a single variable. The elements of structures are addressed by names called *fields*.

```
>> S.char_string='gauss';
>> S.matrix=[1 0;1 0];
>> S.scalar=3;
>> S
S =
    char_string: 'gauss'
         matrix: [2x2 double]
         scalar: 3
>> S.matrix
ans =
     1     0
     1     0
```

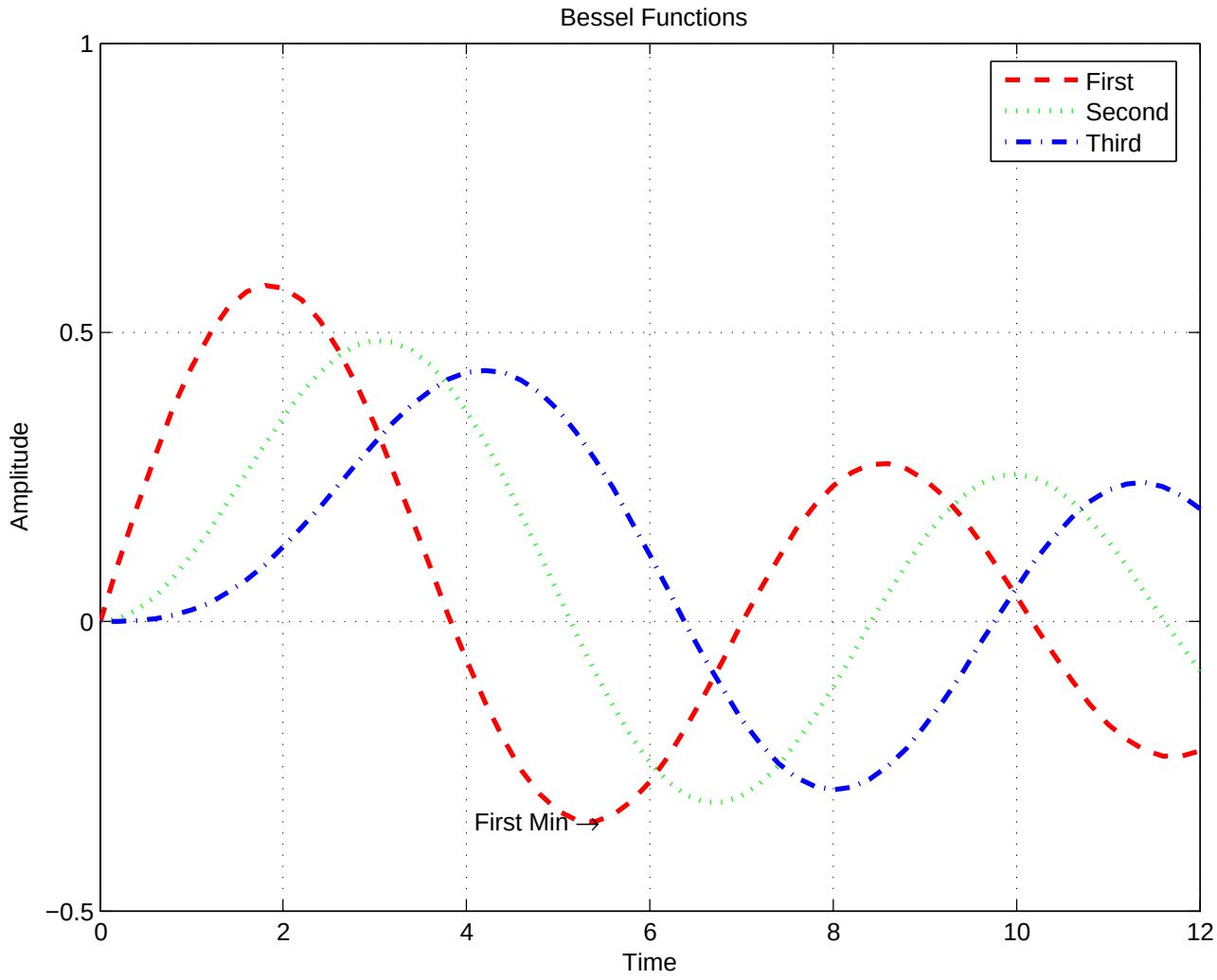
Chapter 2

MATLAB Graphics

Plotting Your Data

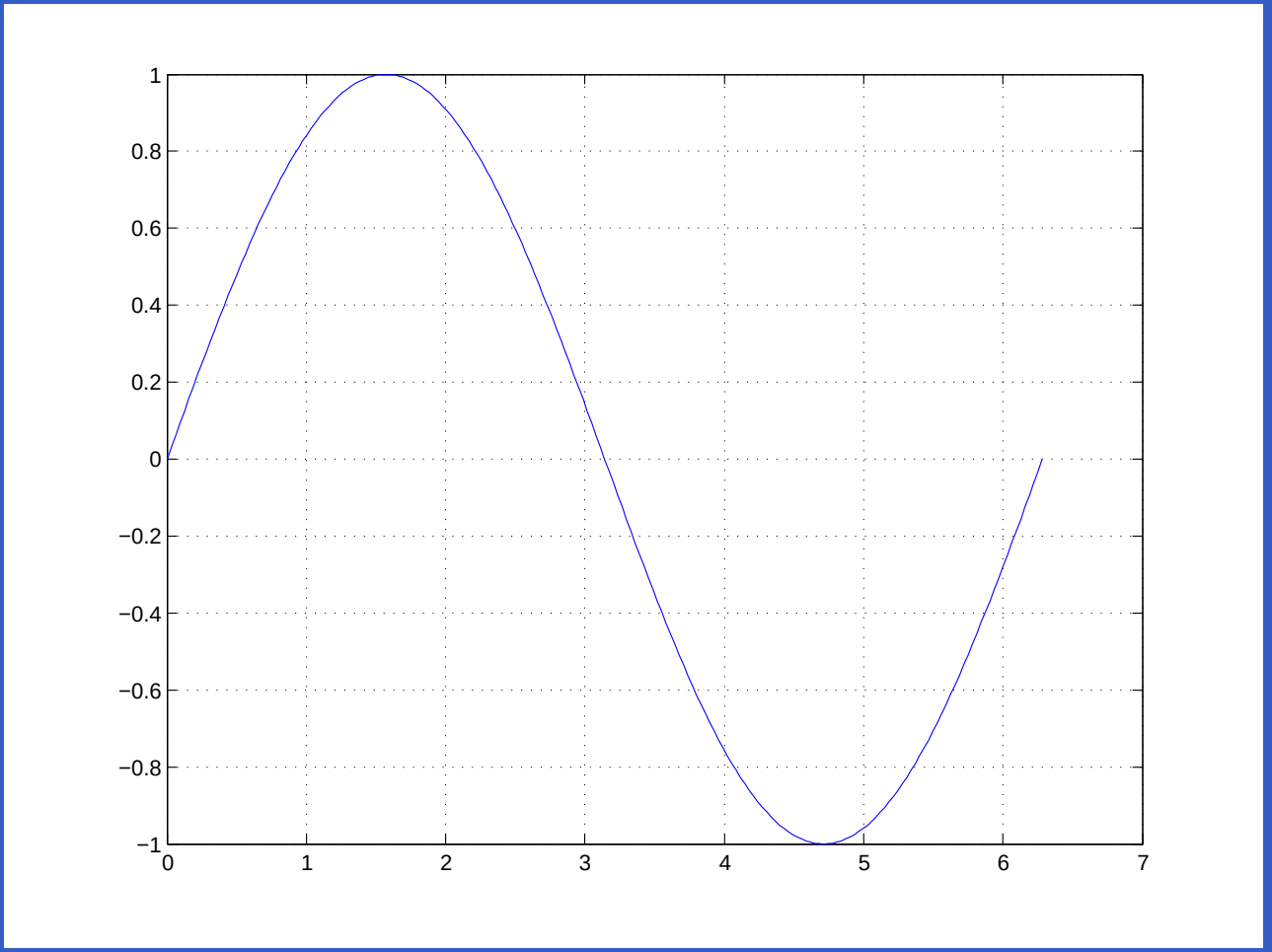
```
>> x=0:0.2:12;
>> y1=bessel(1,x);
>> y2=bessel(2,x);
>> y3=bessel(3,x);
>> h=plot(x,y1,x,y2,x,y3);
>> set(h,'LineWidth',2,{'LineStyle'},{'--';':';'-.'})
>> set(h,{'Color'},{'r';'g';'b'})
>> axis([0 12 -0.5 1])
>> grid on
>> xlabel('Time')
>> ylabel('Amplitude')
>> legend(h,'First','Second','Third')
>> title('Bessel Functions')
>> [y,ix]=min(y1);
>> text(x(ix),y,'First Min \rightarrow',...
                'HorizontalAlignment','right')
>> print -depsc -tiff -r200 myplot
```

Plotting Your Data



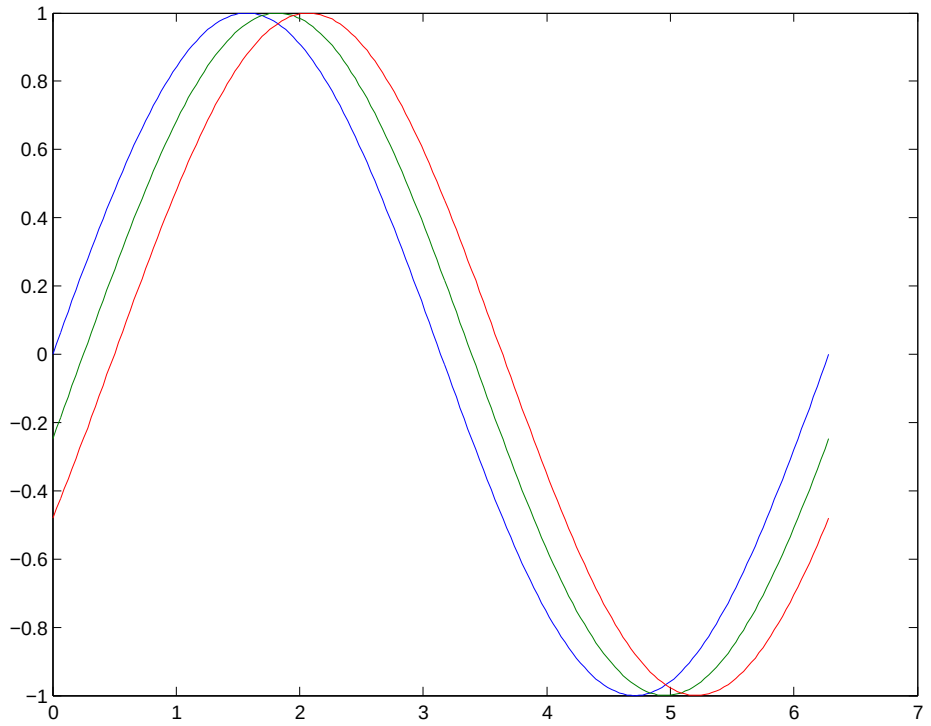
Creating Plots

```
>> t=0:pi/100:2*pi;
>> y=sin(t);
>> plot(t,y)
>> grid on
```



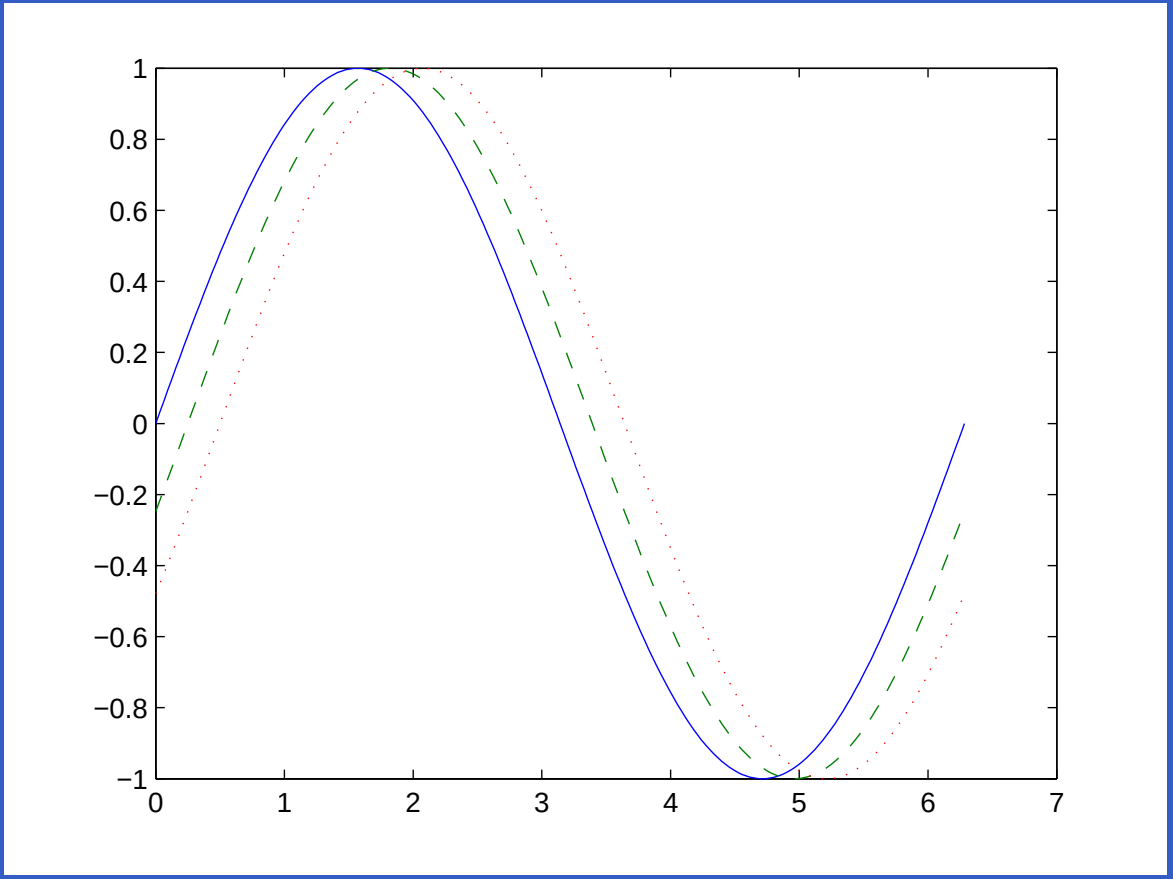
Creating Plots

```
>> y2=sin(t-0.25);
>> y3=sin(t-0.5);
>> plot(t,y,t,y2,t,y3)
```



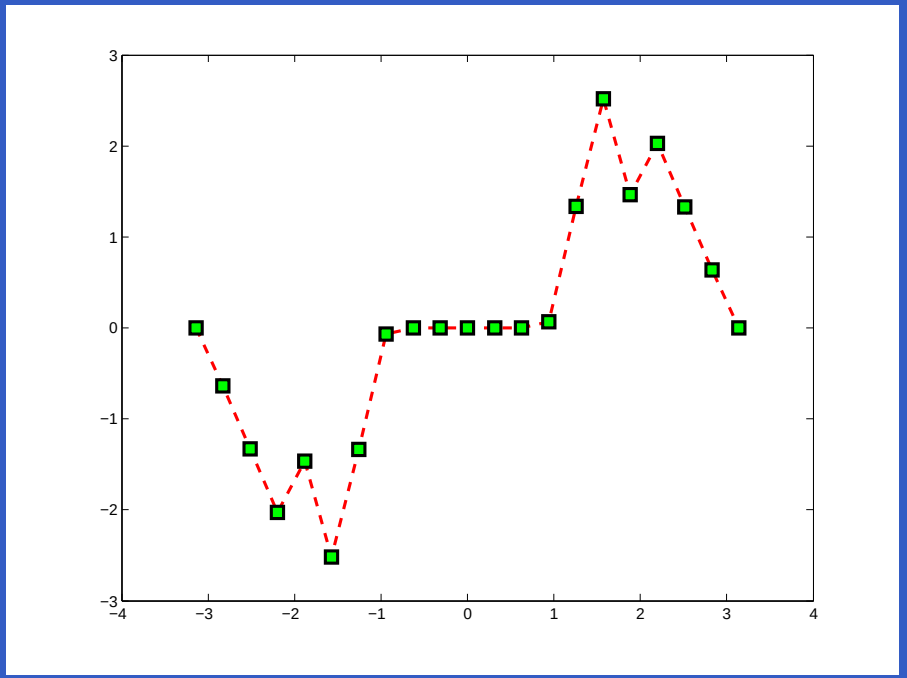
Specifying Line Style

```
>> plot(t,y,'-',t,y2,'--',t,y3,':')
```



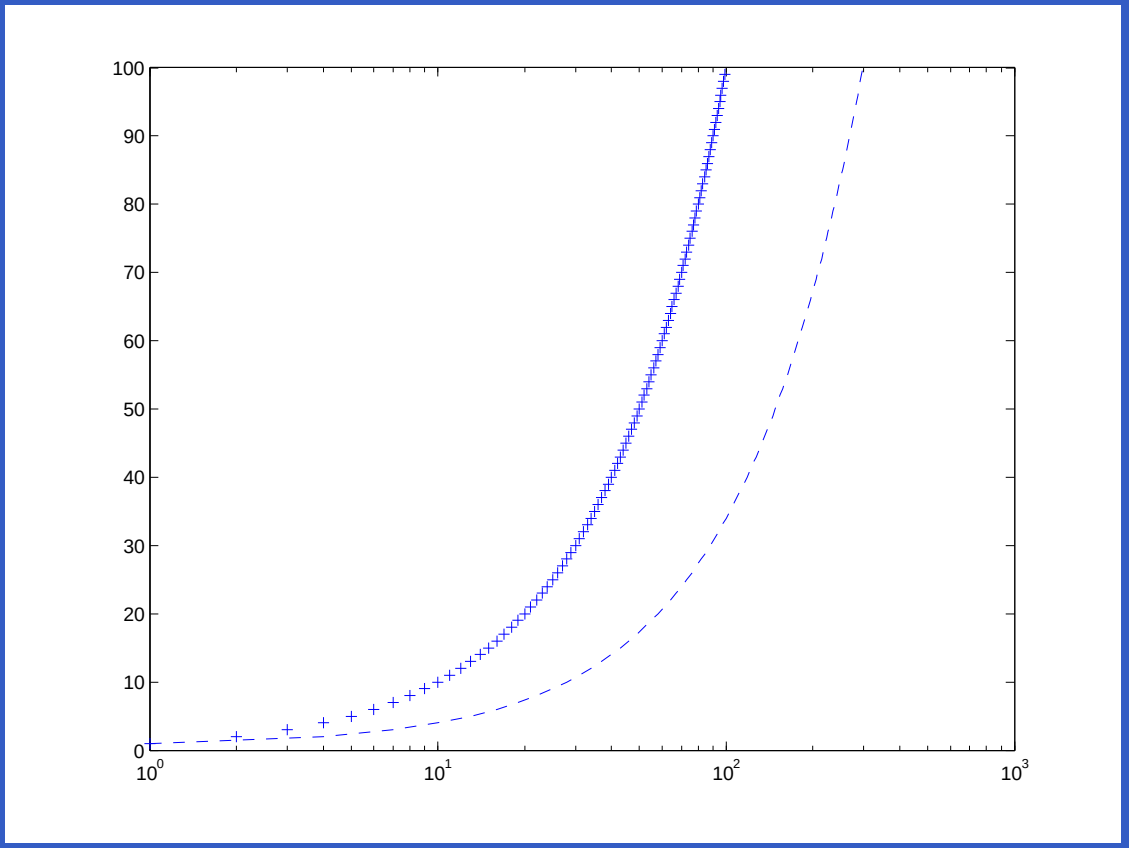
Specifying the Color and Size of Lines

```
>> x=-pi:pi/10:pi;
>> y=tan(sin(x))-sin(tan(x));
>> plot(x,y,'--rs','LineWidth',2,...
                  'MarkerEdgeColor','k',...
                  'MarkerFaceColor','g',...
                  'MarkerSize',10)
```



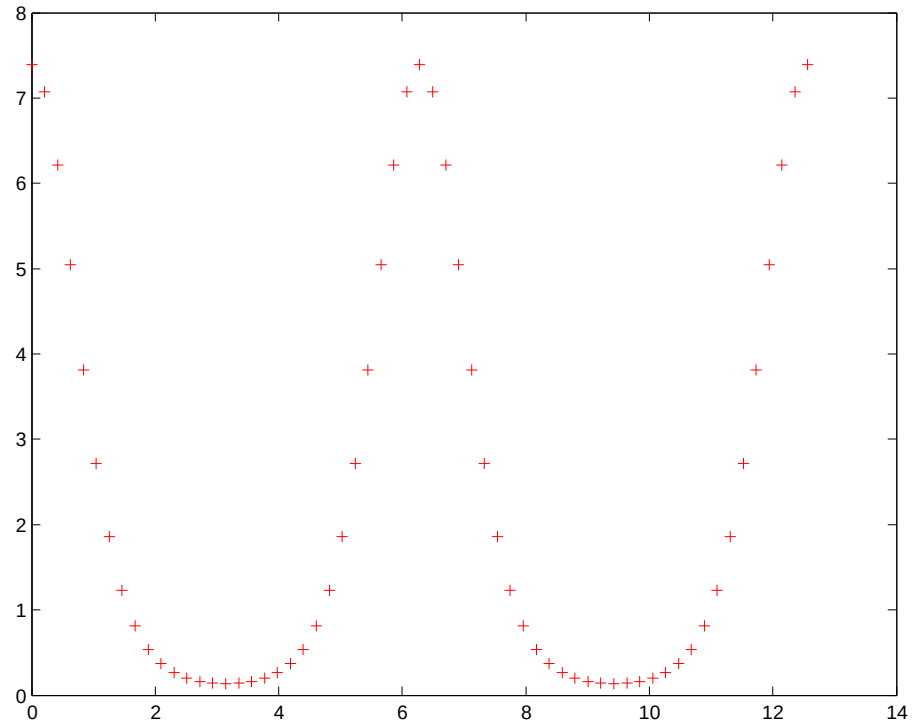
Adding Plots to an Existing Graph

```
>> semilogx(1:100,'+')
>> hold on
>> plot(1:3:300,1:100,'--')
>> hold off
```



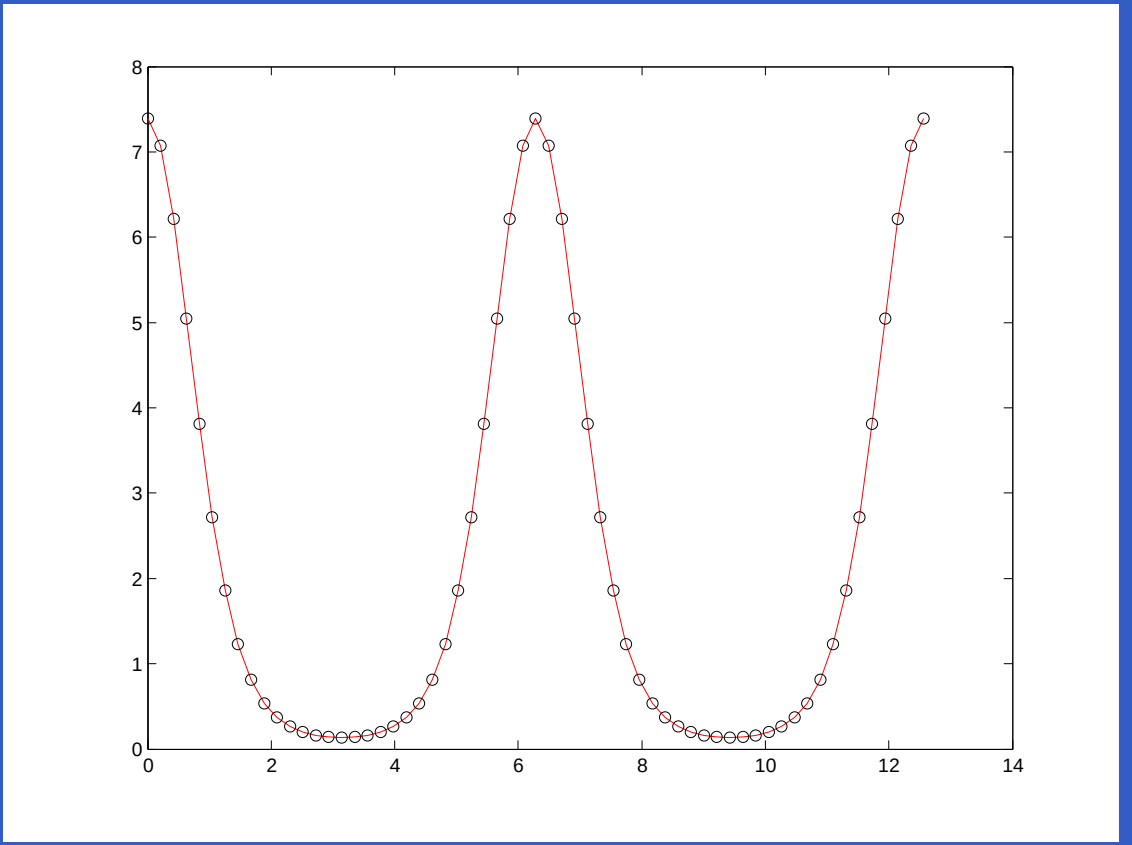
Plotting Only the Data Points

```
>> x=0:pi/15:4*pi;
>> y=exp(2*cos(x));
>> plot(x,y,'r+')
```



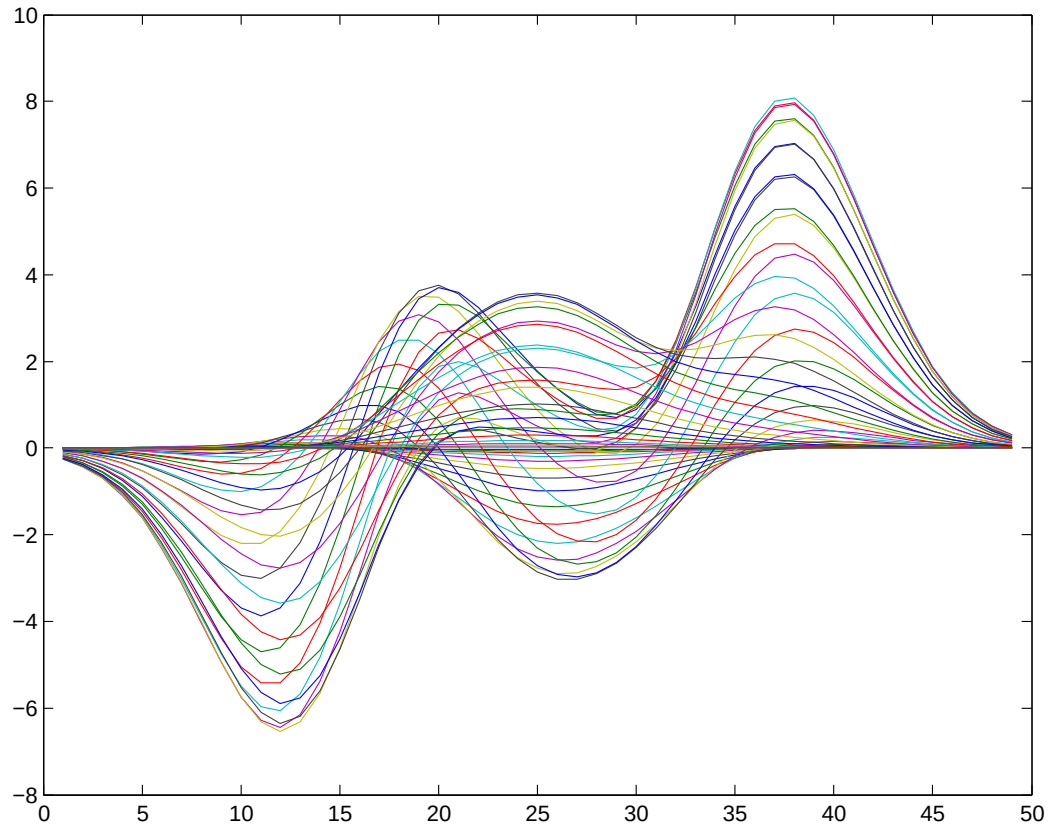
Plotting Markers and Lines

```
>> x=0:pi/15:4*pi;
>> y=exp(2*cos(x));
>> plot(x,y,'-r',x,y,'ok')
```



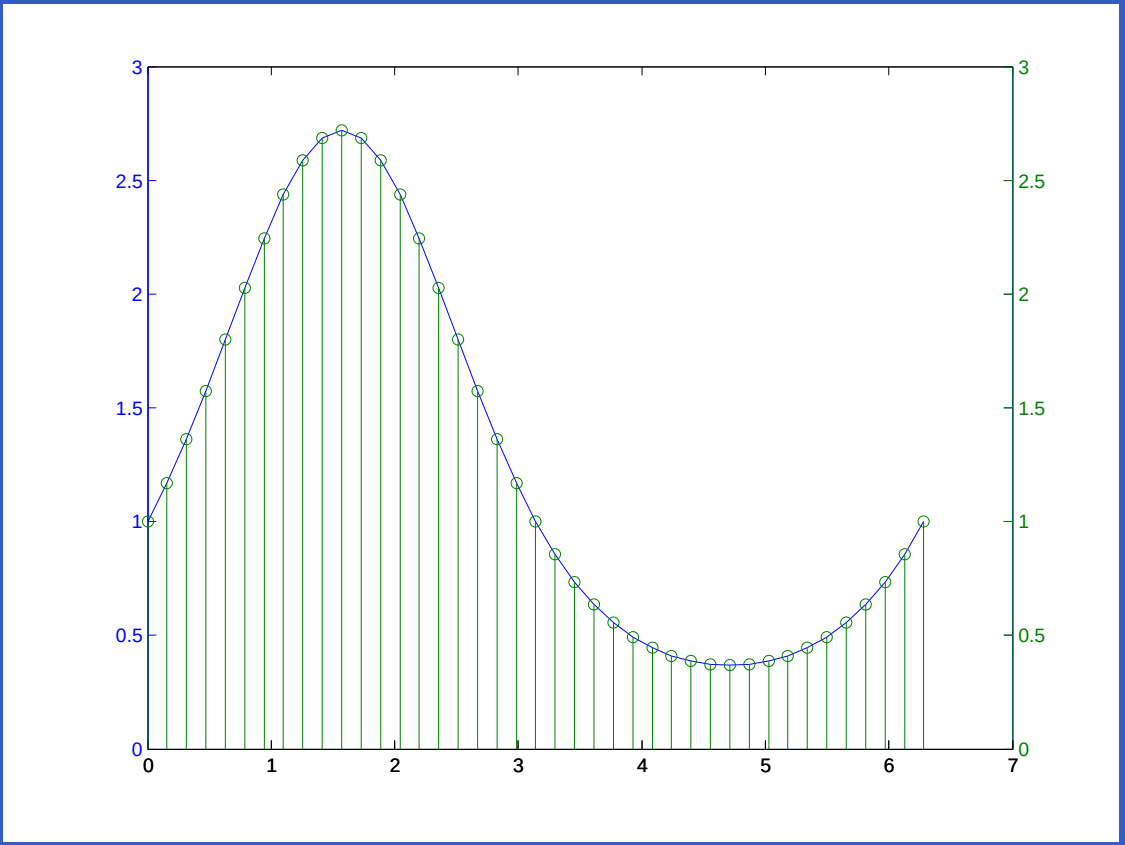
Line Plots of Matrix Data

```
>> Z=peaks;
>> plot(Z)
```



Plotting with Two Y-Axes

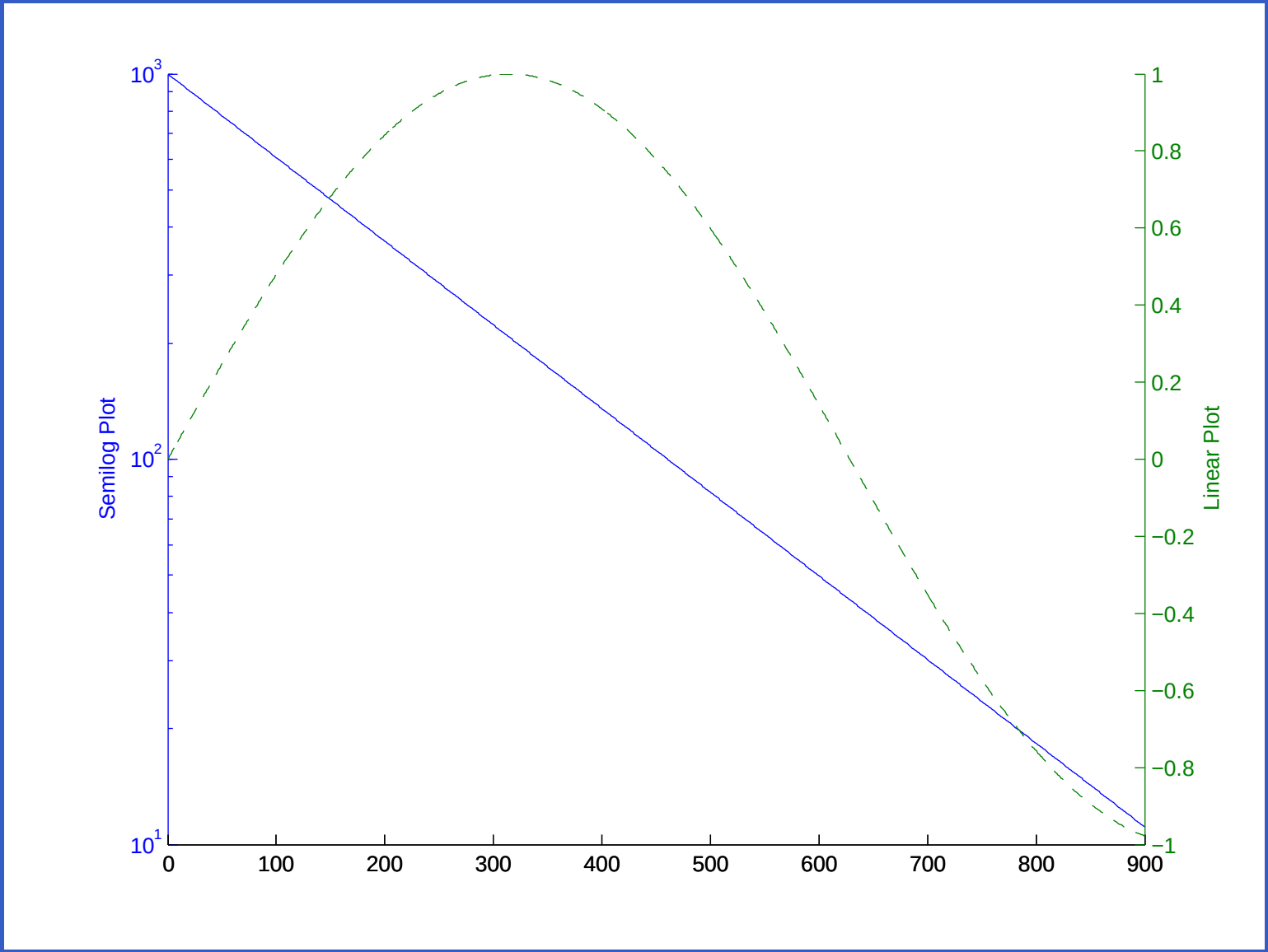
```
>> t=0:pi/20:2*pi;
>> y=exp(sin(t));
>> plotyy(t,y,t,y,'plot','stem')
```



Combining Linear and Logarithmic Axes

```
>> t=0:900;
>> A=1000;
>> a=0.005;
>> b=0.005;
>> z1=A*exp(-a*t);
>> z2=sin(b*t);
>> [haxes,hline1,hline2]=plotyy(t,z1,t,z2,'semilogy','plot');
>> axes(haxes(1))
>> ylabel('Semilog Plot')
>> axes(haxes(2))
>> ylabel('Linear Plot')
>> set(hline2,'LineStyle','--')
```

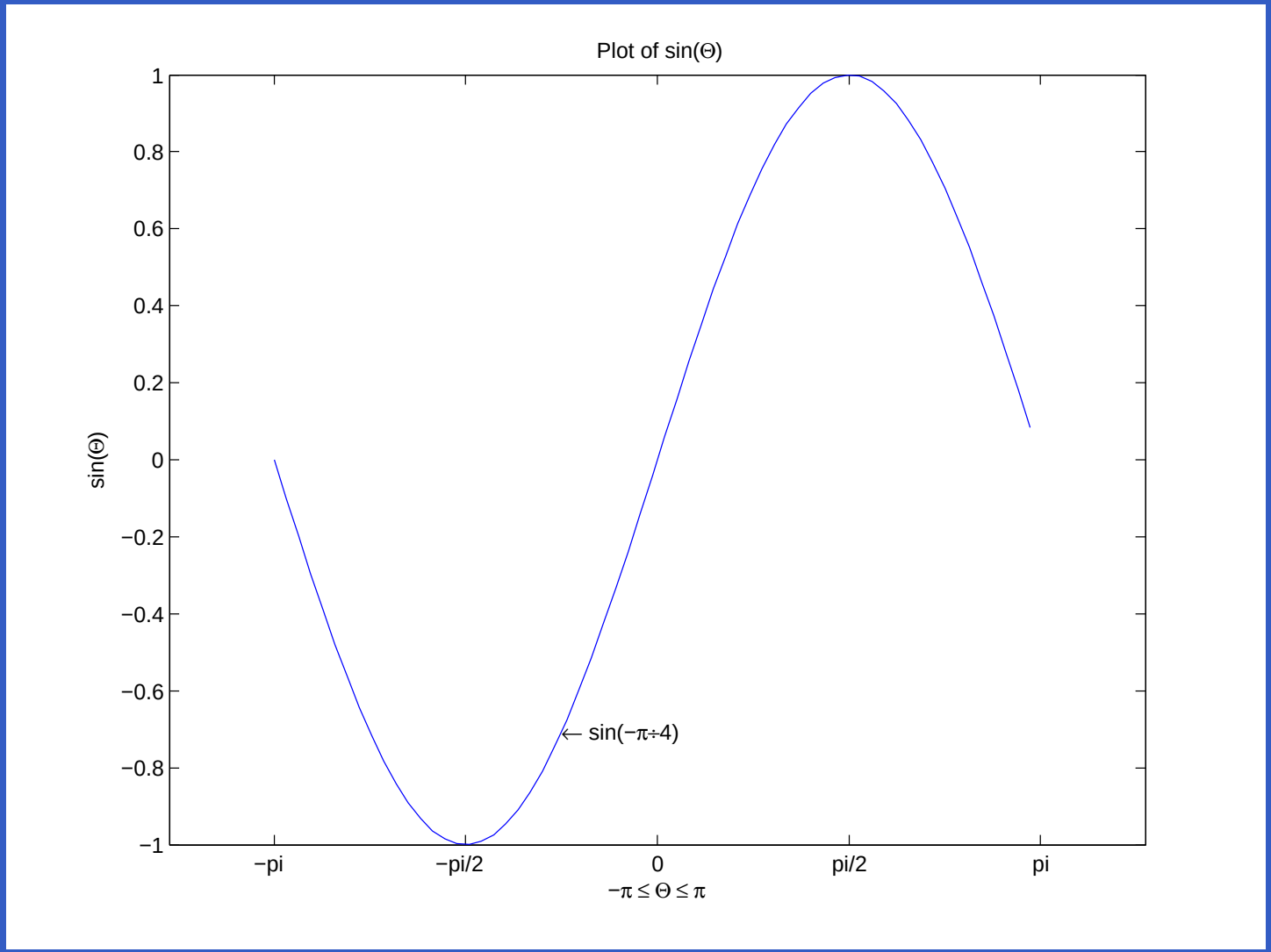
Combining Linear and Logarithmic Axes



Specifying Ticks and Tick Labels

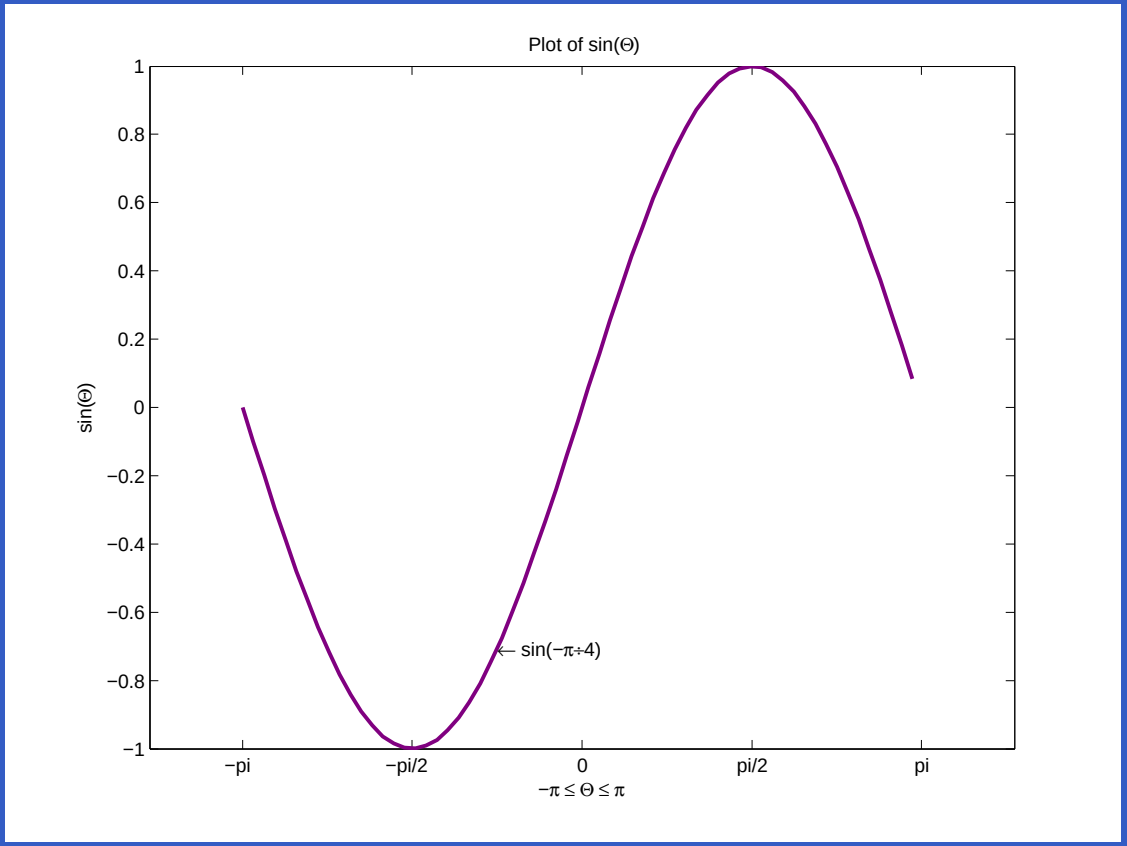
```
>> x=-pi:.1:pi;
>> y=sin(x);
>> plot(x,y)
>> set(gca,'XTick',-pi:pi/2:pi)
>> set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
>> xlabel('-\pi \leq \Theta \leq \pi')
>> ylabel('sin(\Theta)')
>> title('Plot of sin(\Theta)')
>> text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi\div4)',...
        'HorizontalAlignment','left')
```

Specifying Ticks and Tick Labels



Setting Line Properties on an Existing Plot

```
>> set(findobj(gca,'Type','line','Color',[0 0 1]),...
       'Color',[0.5 0 0.5],'LineWidth',2)
```



Chapter 3

Intensity Transformations and Spatial Filtering

Content

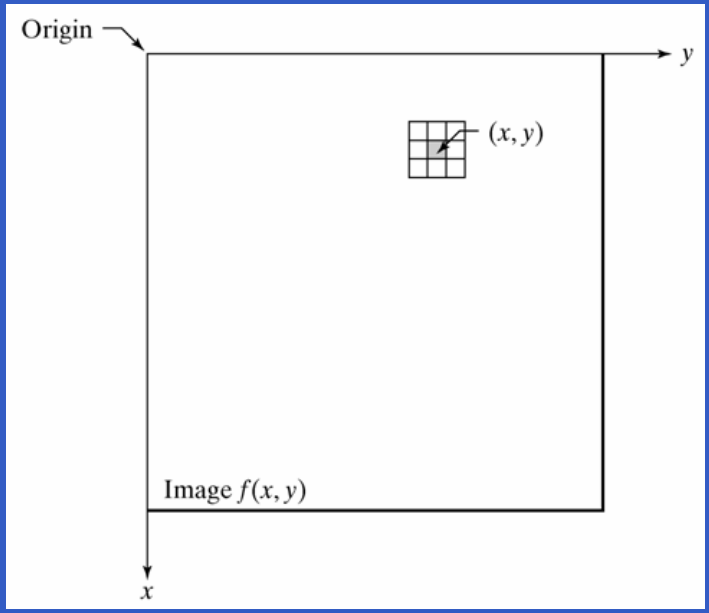
- Background
- Intensity Transformation Functions
- Histogram Processing and Function Plotting
- Spatial Filtering
- Image Processing Toolbox Standard Spatial Filters

Background

The *spatial domain* processes are denoted by the expression

$$g(x, y) = T\left[f(x, y)\right]$$

where $f(x, y)$ is the input image, $g(x, y)$ is the output (processed) image, and T is an operator on f, defined over a specified neighborhood about point (x, y).

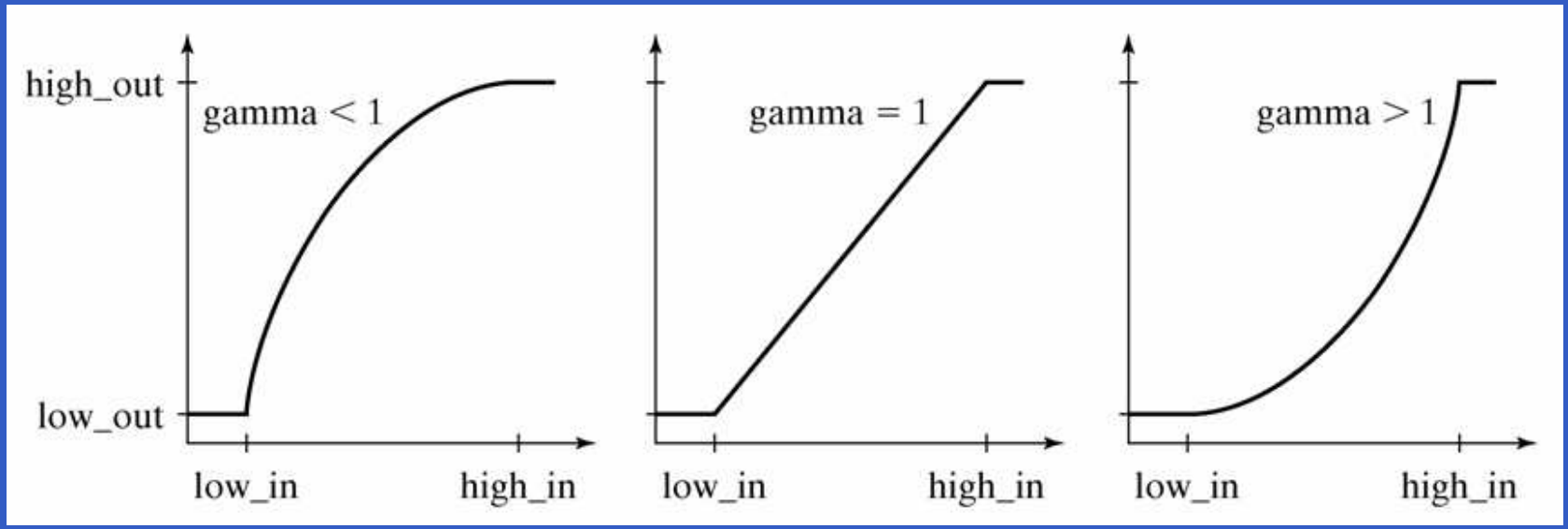


Intensity Transformation Functions

The simplest form of the transformation T is when the neighborhood is of size 1×1 (a single pixel). In this case, the value of g at (x, y) depends only on the intensity of f at that point, and T becomes an *intensity* or *gray-level* transformation function.

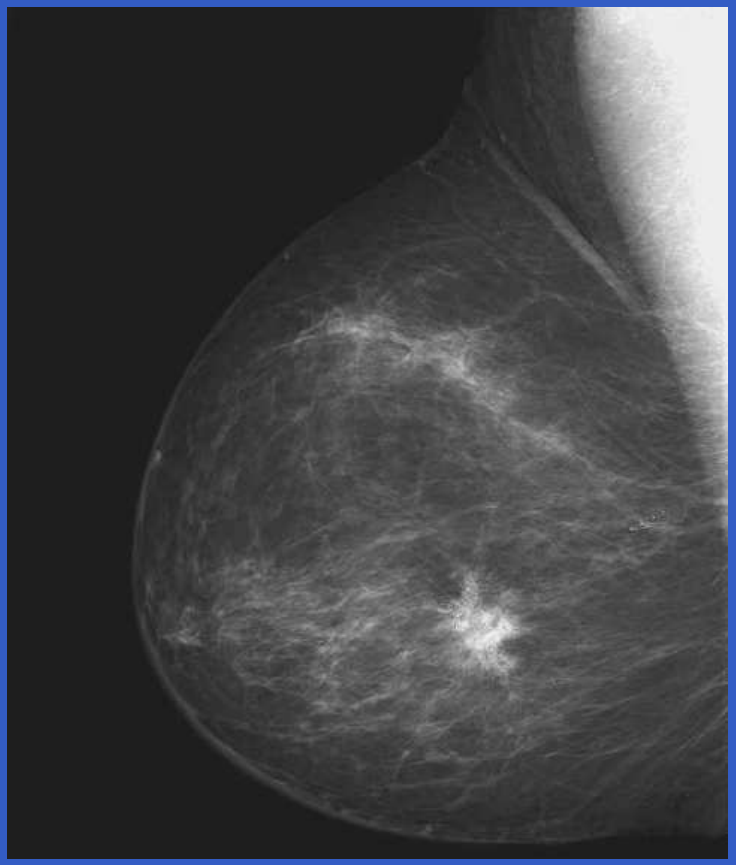
Function imadjust

```
g=imadjust(f,[low_in high_in],...
           [low_out high_out],gamma)
```



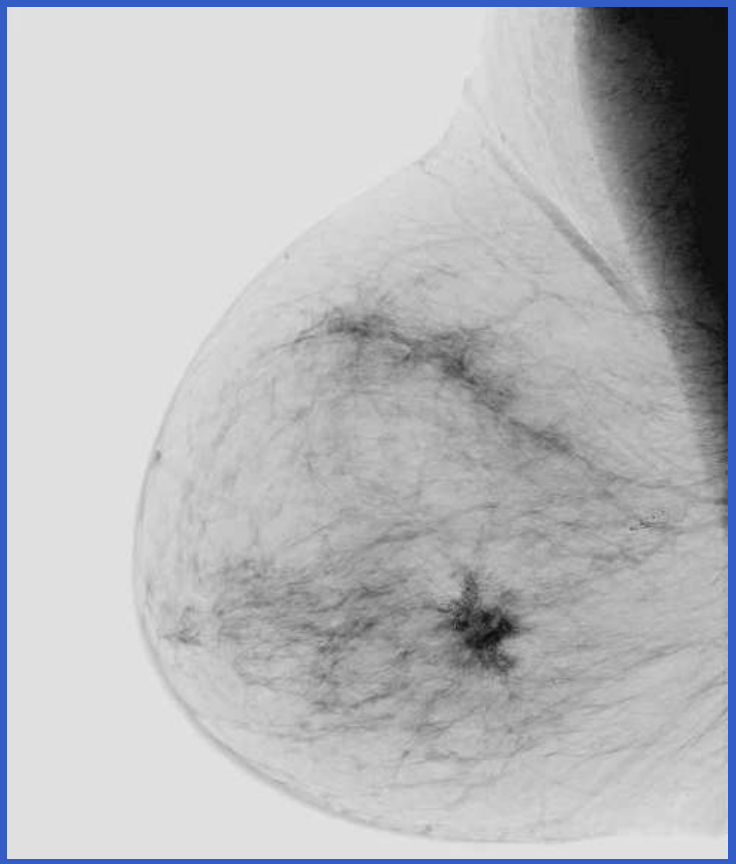
Function imadjust

```
>> f=imread('breast.tif');
```



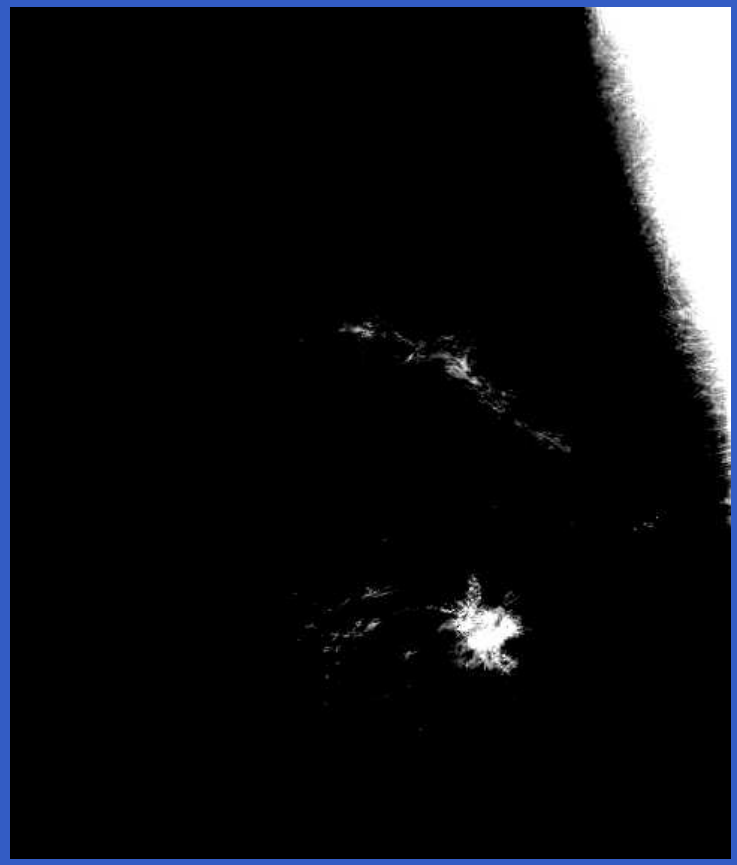
Function imadjust

```
>> g1=imadjust(f,[0 1],[1 0]);
```



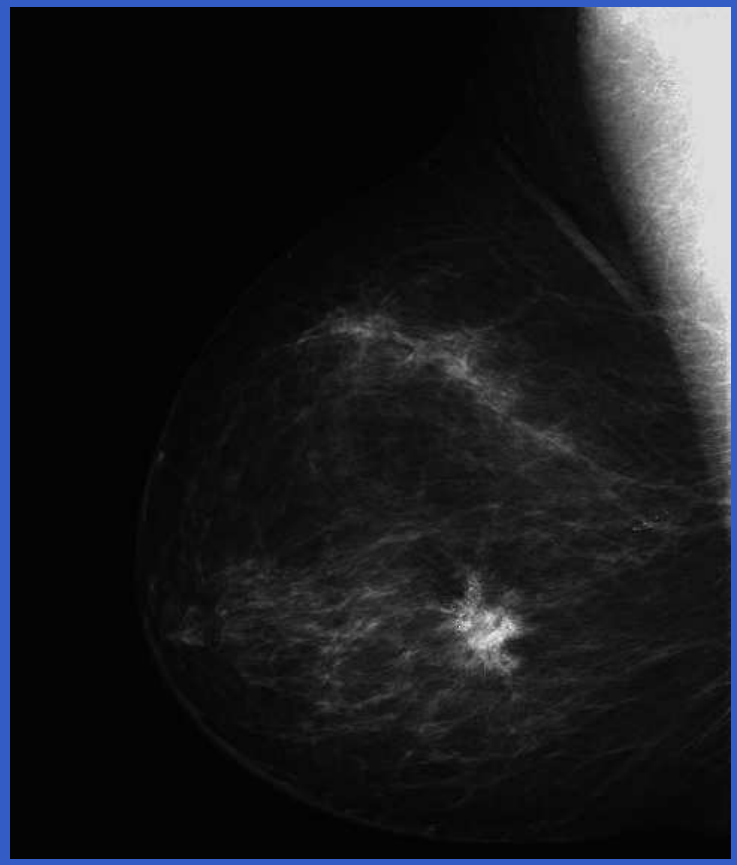
Function imadjust

```
>> g2=imadjust(f,[0.5 0.75],[0 1]);
```



Function imadjust

```
>> g3=imadjust(f,[],[],2);
```



Histogram Processing and Function Plotting

- Generating and Plotting Image Histograms
- Histogram Equalization
- Histogram Matching (Specification)

Generating and Plotting Image Histograms

The histogram of a digital image with L total possible intensity levels in the range $[0, G]$ is defined as the discrete function

$$h(r_k) = n_k$$

where r_k is the kth intensity level in the interval $[0, G]$ and n_k is the number of pixels in the image whose intensity level is r_k. The value of G is 255 for images of class `uint8`, 65535 for images of class `uint16`, and 1.0 for images of class `double`. Keep in mind that indices in MATLAB cannot be 0, so r_1 corresponds to intensity level 0, r_2 corresponds to intensity level 1, and so on, with r_L corresponding to level G. Note also that $G = L - 1$ for images of class `uint8` and `uint16`.

Generating and Plotting Image Histograms

Often, it is useful to work with *normalized* histograms, obtained simply by dividing all elements of $h(r_k)$ by the total number of pixels in the image, which we denote by n:

$$p(r_k) = \frac{h(r_k)}{n} = \frac{n_k}{n}$$

for $k = 1, 2, \ldots, L$.

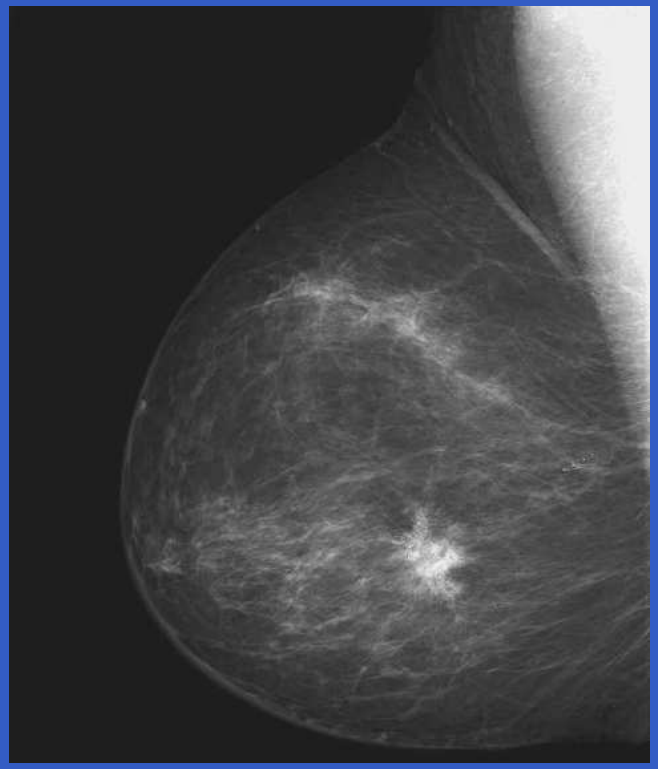
Generating and Plotting Image Histograms

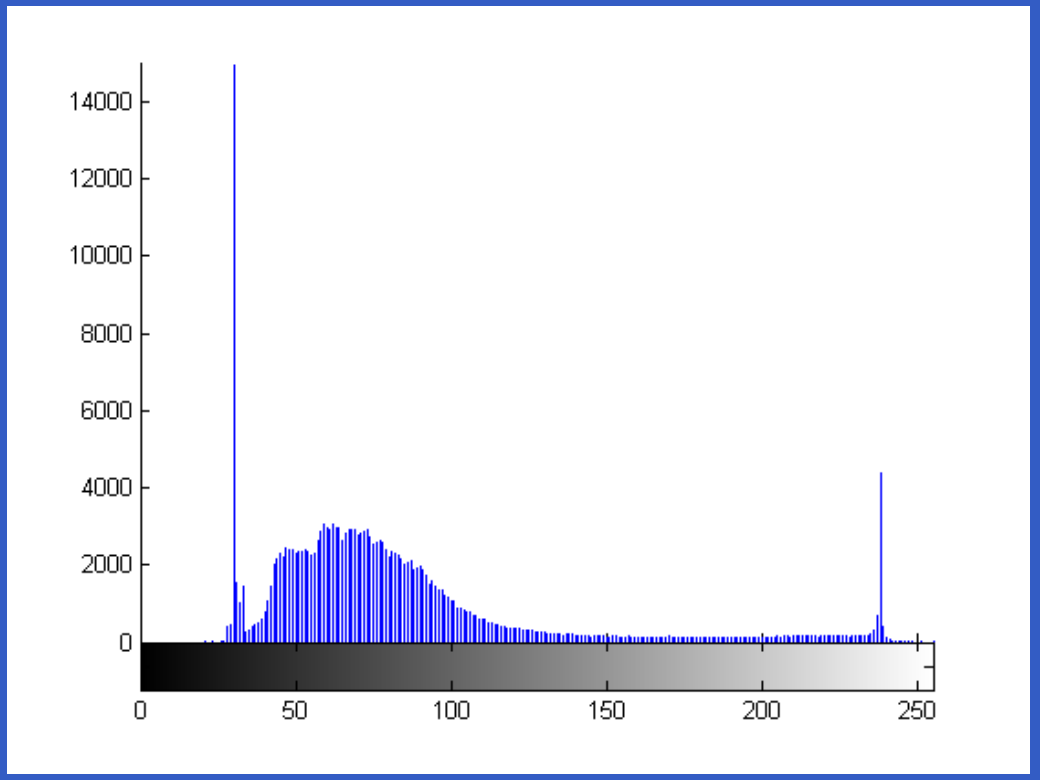
`h=imhist(f,b)`

where `f` is the input image, h is its histogram, $h(r_k)$, and b is the number of bins used in forming the histogram (if b is not included in the argument, `b=256` is used by default). A bin is simply a subdivision of the intensity scale. For example, if we are working with `uint8` images and we let b=2, then the intensity scale is subdivided into two ranges: 0 to 127 and 128 to 255. The resulting histogram will have two values: `h(1)` equal to the number of pixels in the image with values in the interval $[0, 127]$, and `h(2)` equal to the number of pixels with values in the interval $[128, 255]$.

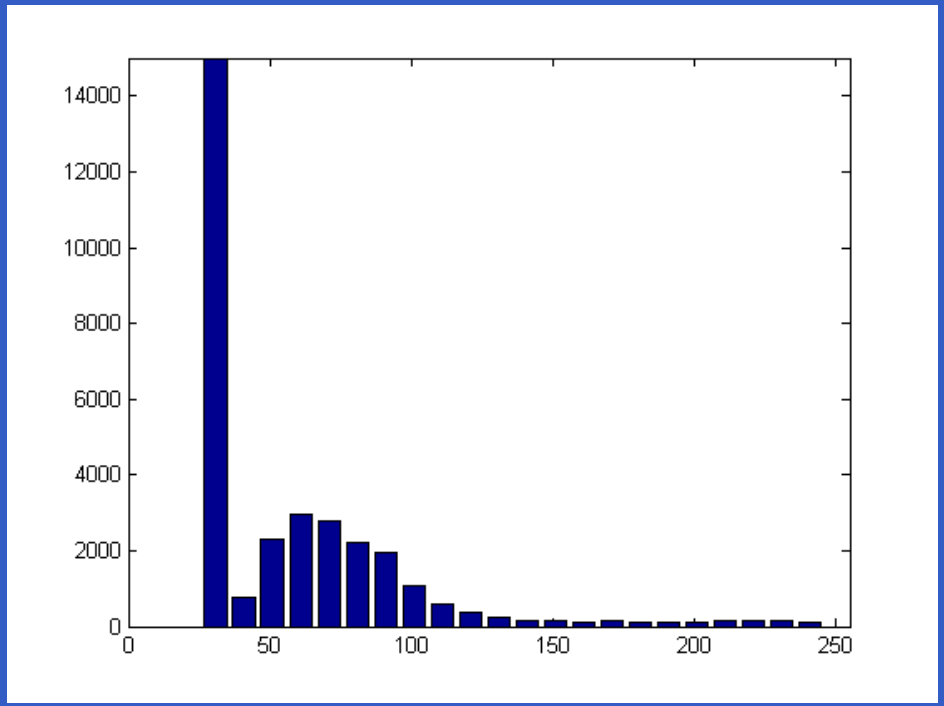
Generating and Plotting Image Histograms

```
>> f=imread('breast.tif');
>> imshow(f), imhist(f)
```





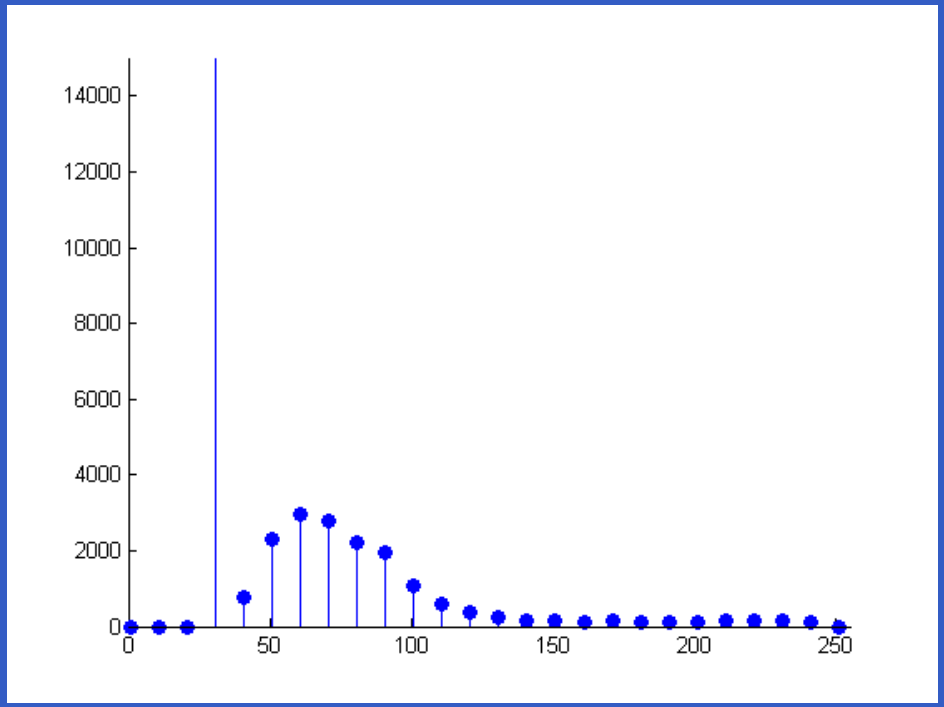
Generating and Plotting Image Histograms




```
>> h=imhist(f);
>> h1=h(1:10:256);
>> horz=1:10:256;
>> bar(horz,h1)
>> axis([0 255 0 15000])
>> set(gca,'xtick',0:50:255)
>> set(gca,'ytick',0:2000:15000)
```

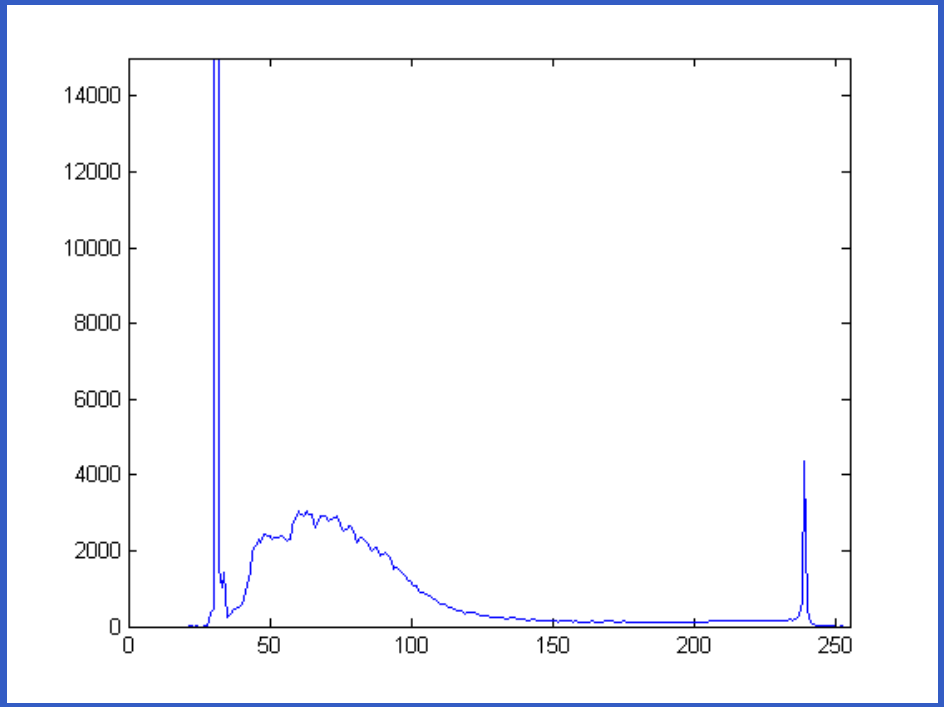
Generating and Plotting Image Histograms

```
>> h=imhist(f);
>> h1=h(1:10:256);
>> horz=1:10:256;
>> stem(horz,h1,'fill')
>> axis([0 255 0 15000])
>> set(gca,'xtick',0:50:255)
>> set(gca,'ytick',0:2000:15000)
```



Generating and Plotting Image Histograms

```
>> h=imhist(f);
>> plot(h)
>> axis([0 255 0 15000])
>> set(gca,'xtick',0:50:255)
>> set(gca,'ytick',0:2000:15000)
```



Some Useful Plotting Function

- `plot(horz,v,'color_linestyle_marker')`
- `bar(horz,v,width)`
- `stem(horz,v,'color_linestyle_marker','fill')`
- `axis([horzmin horzmax vertmin vertmax])`
- `xlabel('text string','fontsize',size)`
- `ylabel('text string','fontsize',size)`
- `text(xloc,yloc,'text string','fontsize',size)`
- `title('titlestring')`

Some Useful Plotting Function

Symbol	Color	Symbol	Line Style	Symbol	Marker
k	Black	-	Solid	+	Plus sign
w	White	--	Dashed	o	Circle
r	Red	:	Dotted	*	Asterisk
g	Green	-.	Dash-dot	.	Point
b	Blue	none	No line	x	Cross
c	Cyan			s	Square
y	Yellow			d	Diamond
m	Magenta			none	No marker

Histogram Equalization

$$s_k = \sum_{j=0}^{k} \frac{n_j}{n} \quad k = 0, 1, 2, \ldots, L-1$$

where n is the total number of pixels in the image, n_k is the number of pixels that have gray level r_k, and L is the total number of possible gray levels in the image. A processed image is obtained by mapping each pixel with level r_k in the input image into a corresponding pixel with level s_k in the output image.

Histogram Equalization

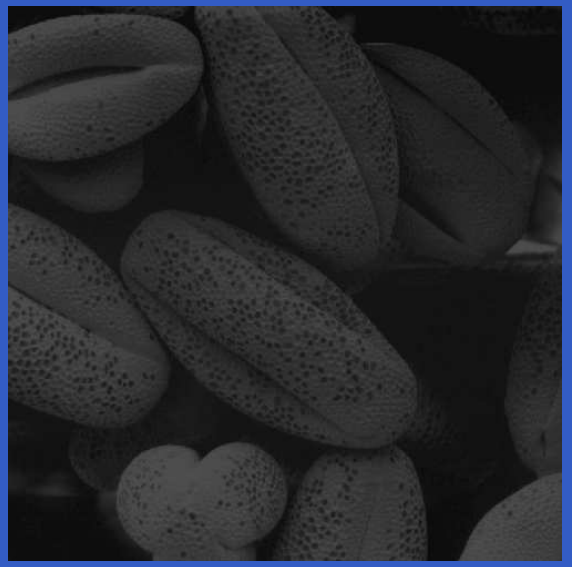
```
g=histeq(f,nlev)
```

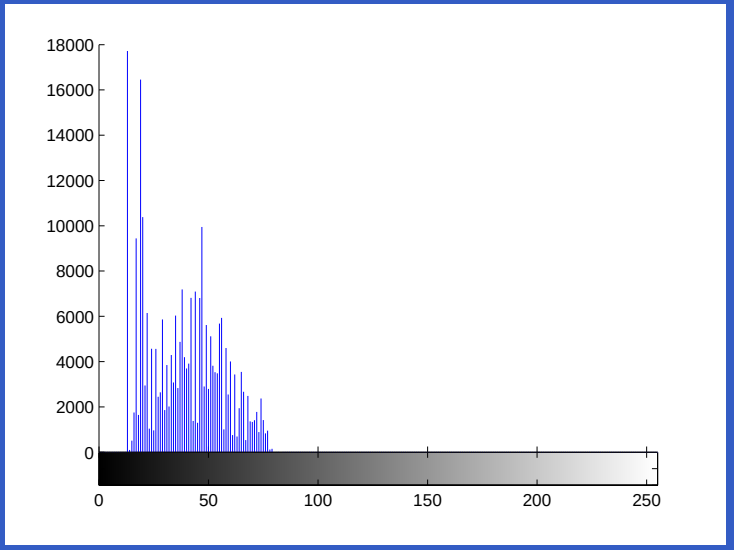
where `f` is the input image and `nlev` is the number of intensity levels specified for the output image. If `nlev` is equal to L (the total number of possible levels in the input image), then `histeq` implements the transformation function (described on the previous slide), directly. If `nlev` is less than L, then `histeq` attempts to distribute the levels so that they will approximate a flat histogram. Unlike `imhist`, the default value in `histeq` is `nlev=64`.

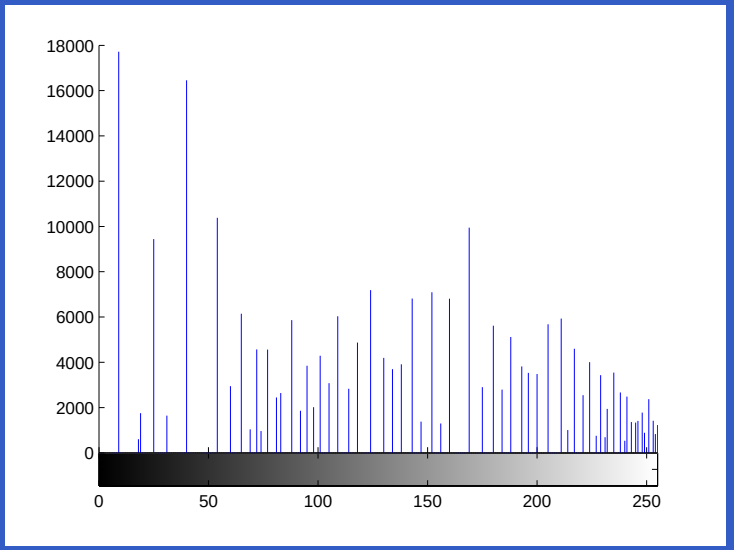
Histogram Equalization

```
>> f=imread('pollen.tif');
>> imshow(f)
>> figure, imhist(f)
>> ylim('auto')
>> g=histeq(f,256);
>> figure, imshow(g)
>> figure, imhist(g)
>> ylim('auto')
```

Histogram Equalization



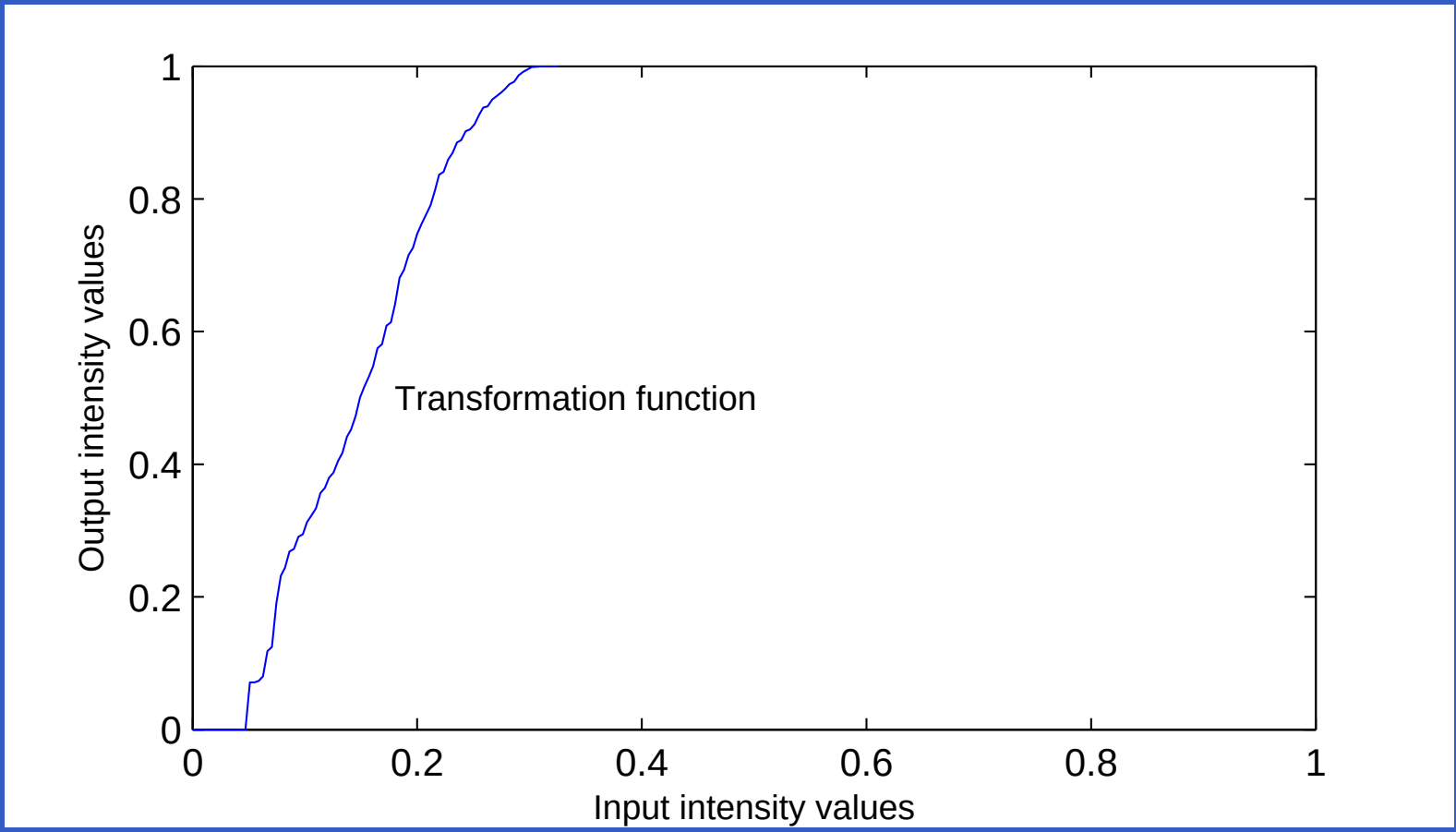




Histogram Equalization

```
>> hnorm=imhist(f)./numel(f);
>> %Cummulative distribution function:
>> cdf=cumsum(hnorm);
>> x=linspace(0,1,256);
>> plot(x,cdf)
>> axis([0 1 0 1])
>> set(gca,'xtick',0:.2:1)
>> set(gca,'ytick',0:.2:1)
>> xlabel('Input intensity values','fontsize',9)
>> ylabel('Output intensity values','fontsize',9)
>> %Specify text in the body of the graph:
>> text(0.18,0.5,'Transformation function',...
>>      'fontsize',9)
```

Histogram Equalization



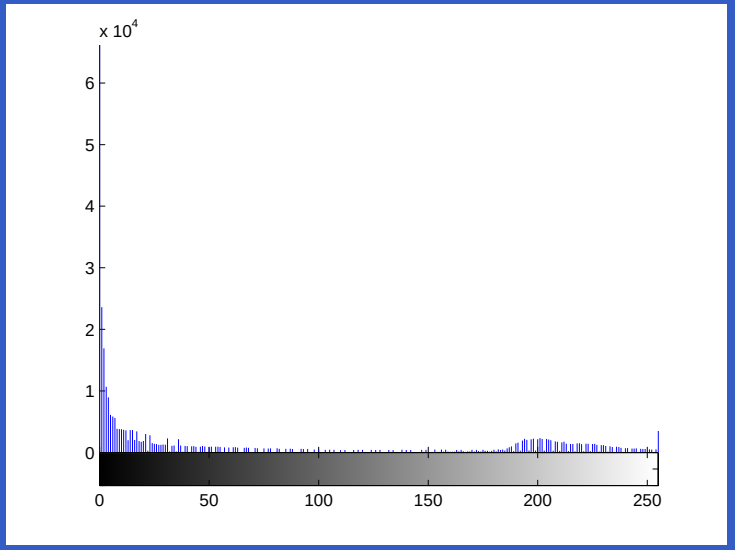
Histogram Matching

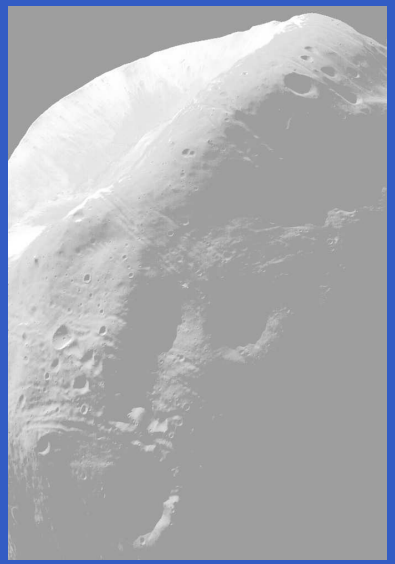
It is useful in some applications to be able to specify the shape of the histogram that we wish the processed image to have. The method used to generate a processed image that has a specified histogram is called *histogram matching*.

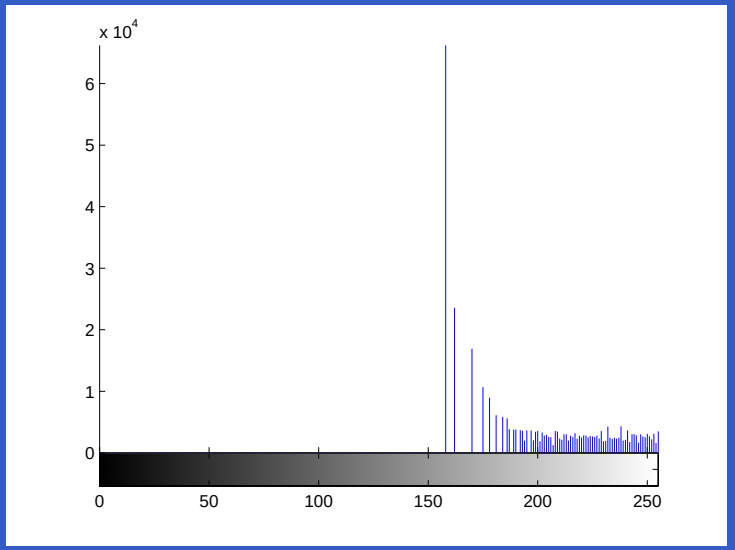
```
g=histeq(f,hspec)
```

where `f` is the input image, `hspec` is the specified histogram (a row vector of specified values), and `g` is the input image, whose histogram approximates the specified histogram, `hspec`.

Histogram Matching







Histogram Matching

```
function p=twomodegauss(m1,sig1,m2,sig2,A1,A2,k)
%TWOMODEGAUSS Generates a bimodal Gaussian function.
%   P=TWOMODEGAUSS(M1,SIG1,M2,SIG2,A1,A2,K) generates a bimodal,
%   Gaussian-like function in the interval [0,1]. P is a
%   256-element vector normalized so that SUM(P) equals 1. The
%   mean and standard deviation of the modes are (M1,SIG1) and
%   (M2,SIG2), respectively. A1 and A2 are the amplitude values
%   of the two modes. Since the output is normalized, only the
%   relative values of A1 and A2 are important. K is an offset
%   values that raises the "floor" of the function. A good set
%   of values to try is M1=0.15, SIG1=0.05, M2=0.75, SIG2=0.05,
%   A1=1, A2=0.07, and K=0.002.
```

Histogram Matching

```
c1=A1*(1/((2*pi)^0.5)*sig1);
k1=2*(sig1^2);
c2=A2*(1/((2*pi)^0.5)*sig2);
k2=2*(sig2^2);
z=linspace(0,1,256);

p=k+c1*exp(-((z-m1).^2)./k1)+...
    c2*exp(-((z-m2).^2)./k2);
p=p./sum(p(:));
```

Histogram Matching

```
function p=manualhist
%MANUALHIST Generates a bimodal histogram interactively.
%   P=MANUALHIST generates a bimodal histogram using
%   TWOMODEGAUSS(m1,sig1,m2,sig2,A1,A2,k). m1 and m2 are the
%   means of the two modes and must be in the range [0,1]. sig1
%   and sig2 are the standard deviations of the two modes. A1
%   and A2 are amplitude values, and k is an offset value that
%   raises the "floor" of histogram. The number of elements in
%   the histogram vector P is 256 and sum(P) is normalized to 1.
%   MANUALHIST repeatedly prompts for the parameters and plots
%   the resulting histogram until the user types an 'x' to quit,
%   and then it returns the last histogram computed.
%
%   A good set of starting values is: (0.15, 0.05, 0.75, 0.05, 1,
%   0.07, 0.002).
```

Histogram Matching

```
%Initialize.
repeats=true;
quitnow='x';

%Compute a default histogram in case the user quits before
%estimating at least one histogram.
p=twomodegauss(0.15,0.05,0.75,0.05,1,0.07,0.002);

%Cycle until x is input.
while repeats
    s=input('Enter m1, sig1, m2, sig2, A1, A2, k OR x to quit:','s');
    if s==quitnow
        break
    end
```

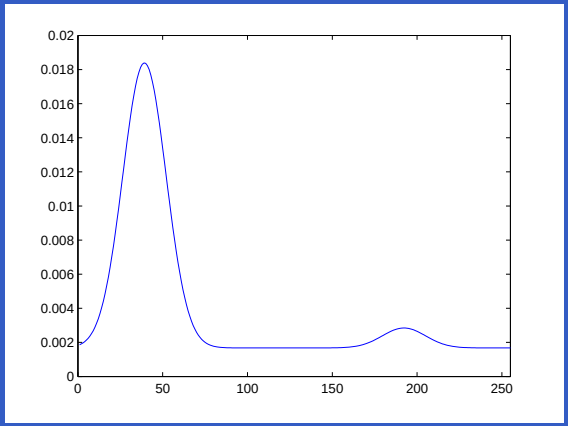
Histogram Matching

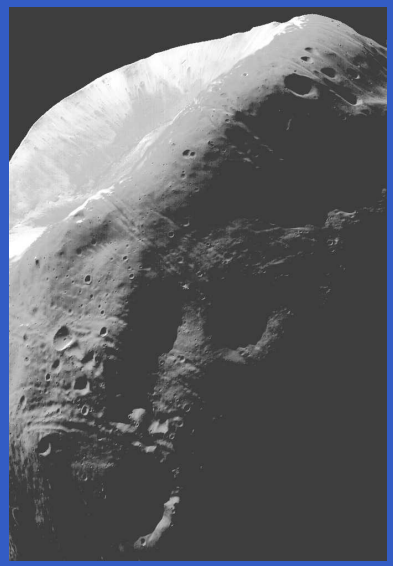
```
    %Convert the input string to a vector of numerical values and
    %verify the number of inputs.
    v=str2num(s);
    if numel(v)~=7
        disp('Incorrect number of inputs.')
        continue
    end

    p=twomodegauss(v(1),v(2),v(3),v(4),v(5),v(6),v(7));
    %Start a new figure and scale the axes. Specifying only xlim
    %leaves ylim on auto.
    figure, plot(p)
    xlim([0 255])
end
```

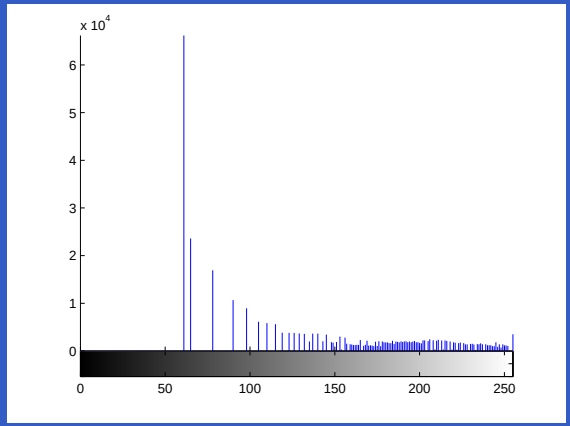
Histogram Matching

```
>> f=imread('moon_phobos.tif');
>> p=manualhist;
Enter m1, sig1, m2, sig2, A1, A2, k OR x to quit:x
>> g=histeq(f,p);
>> imshow(g)
>> figure, imhist(g)
```

Histogram Matching







Spatial Filtering

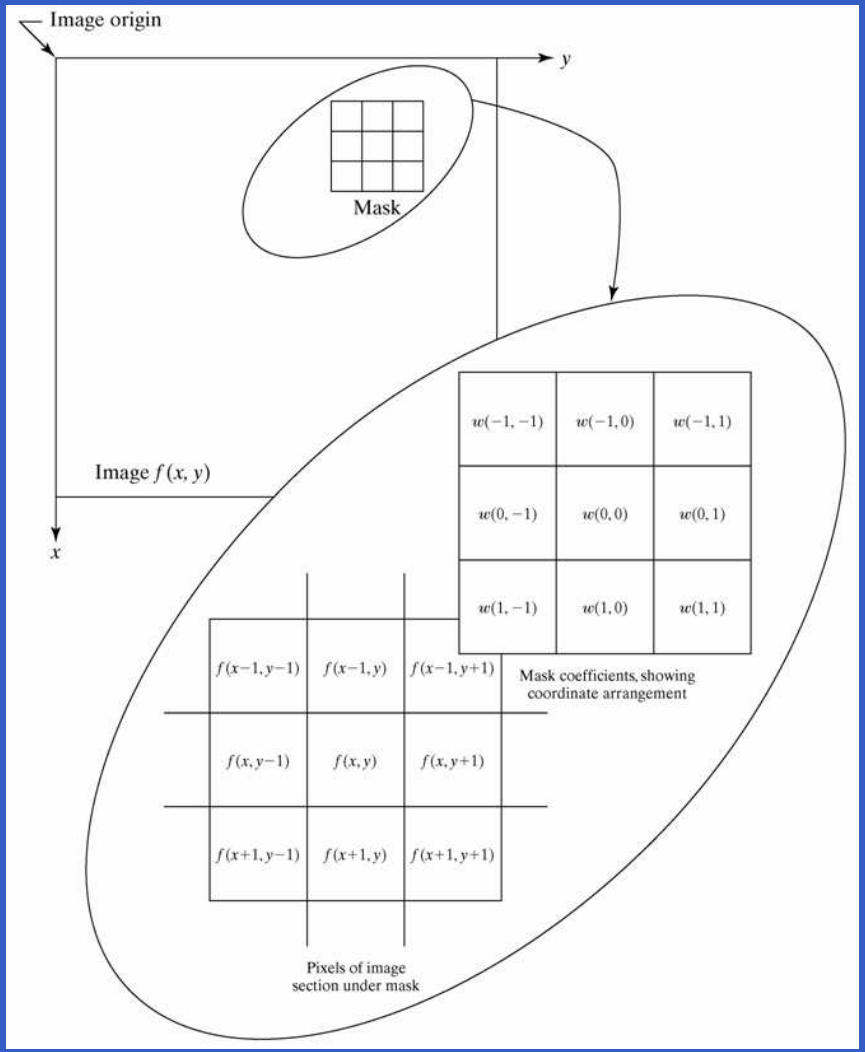
Neighborhood processing consists of

- defining a center point, (x, y);
- performing an operation that involves only the pixels in a predefined neighborhood about that center point;
- letting the result of that operation be the "response" of the process at *that* point; and
- repeating the process for every point in the image.

If the computations performed on the pixels of the neighborhoods are linear, the operation is called *linear spatial filtering*; otherwise it is called *nonlinear spatial filtering*.

Linear Spatial Filtering

The mechanics of linear spatial filtering:

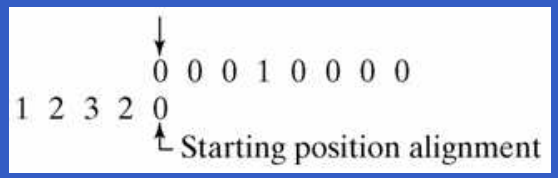


Linear Spatial Filtering

The process consists simply of moving the center of the filter mask w from point to point in an image f. At each point (x, y), the response of the filter at that point is the sum of products of the filter coefficients and the corresponding neighborhood pixels in the area spanned by the filter mask. For a mask of size $m \times n$, we assume typically that $m = 2a + 1$ and $n = 2b + 1$, where a and b are nonnegative integers.

There are two closely related concepts that must be understood clearly when performing linear spatial filtering. *Correlation* is the process of passing the mask w by the image array f in the manner described earlier. Mechanically, *convolution* is the same process, except that w is rotated by $180°$ prior to passing it by f.

Linear Spatial Filtering



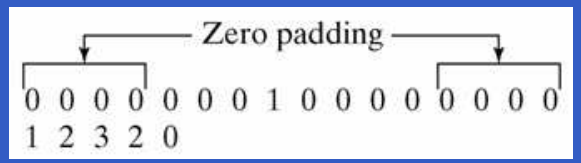
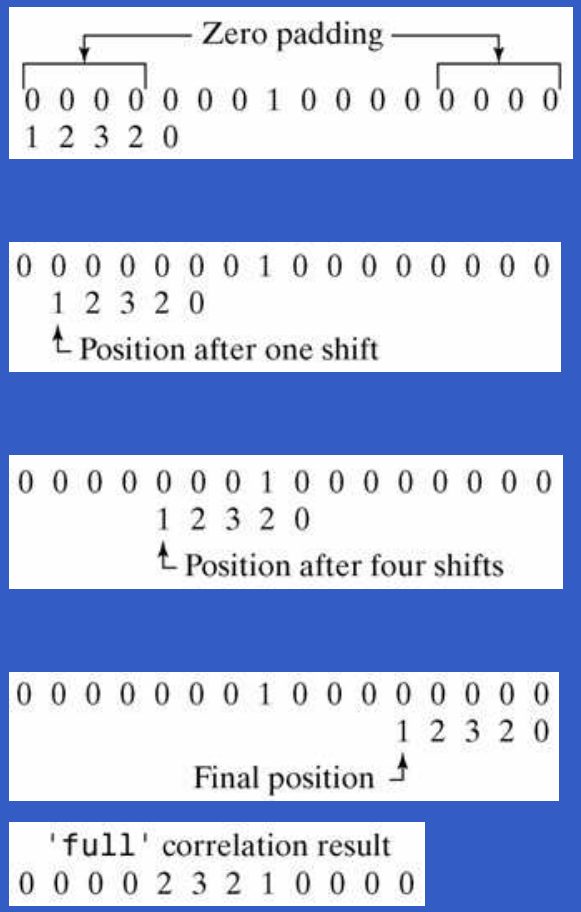


Figure shows a one-dimensional function, f, and a mask w.

To perform the correlation of the two functions, we move w so that its rightmost point coincides with the origin of f.

There are points between the two functions that do not overlap. The most common way to handle this problem is to pad f with as many 0s as are necessary to guarantee that there will always be corresponding points for the full excursion of w past f.

Linear Spatial Filtering



The first value of correlation is the sum of products of the two functions in the position shown in the figure.

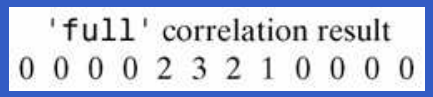
Next, we move w one location to the right and repeat the process.

After four shifts, we encounter the first nonzero value of the correlation, which is $2 \cdot 1 = 2$.

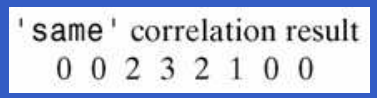
The ending geometry is shown in this figure.

If we proceed in this manner until w moves completely past f we would get this result.

Linear Spatial Filtering



The label `'full'` is a flag used by the IPT[a] to indicate correlation using a padded image and computed in the manner just described.

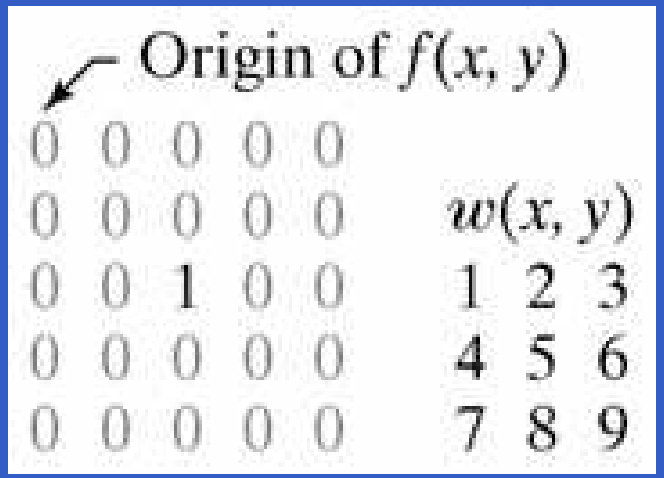


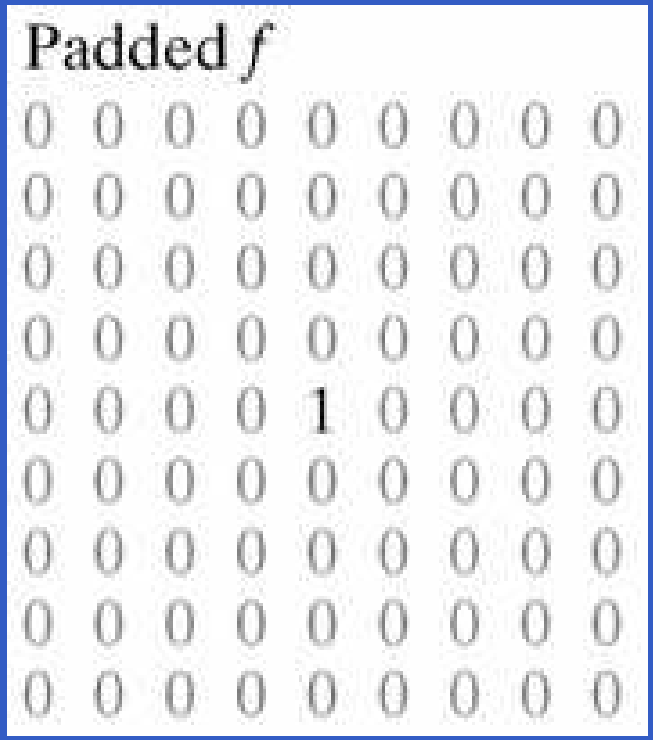
The IPT provides another option, denoted by `'same'` that produces a correlation that is the same size as f. This computation also uses zero padding, but the starting position is with the center point of the mask aligned with the origin of f. The last computation is with the center point of the mask aligned with the last point in f.

[a] Image Processing Toolbox of MATLAB

Linear Spatial Filtering

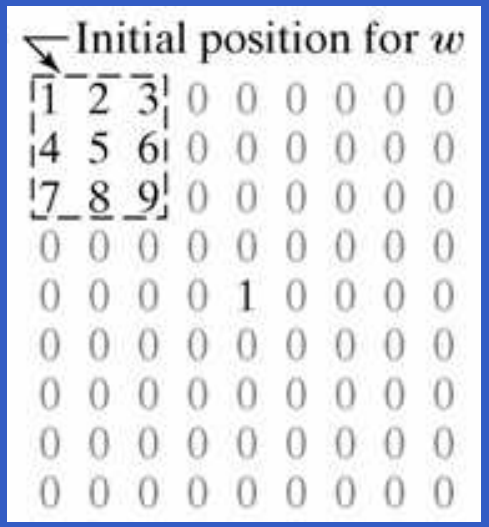
The preceding concepts extend easily to images, as illustrated in the following figures.





Linear Spatial Filtering

Correlation



'full' correlation result

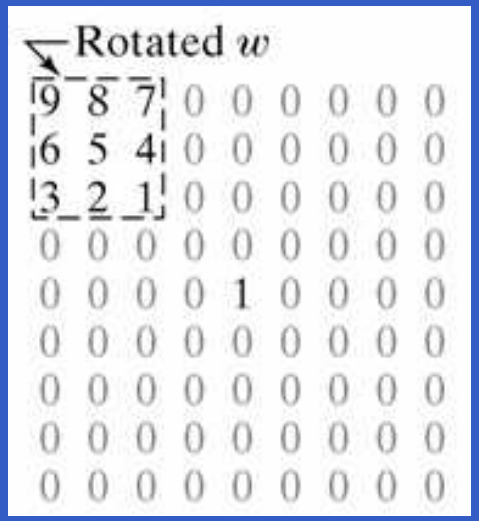
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	9	8	7	0	0	0
0	0	0	6	5	4	0	0	0
0	0	0	3	2	1	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

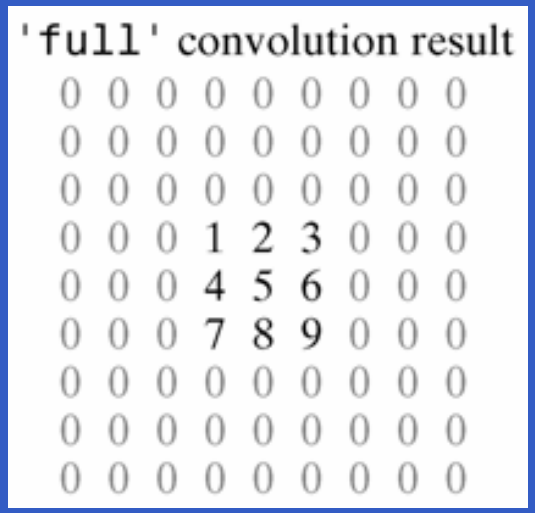
'same' correlation result

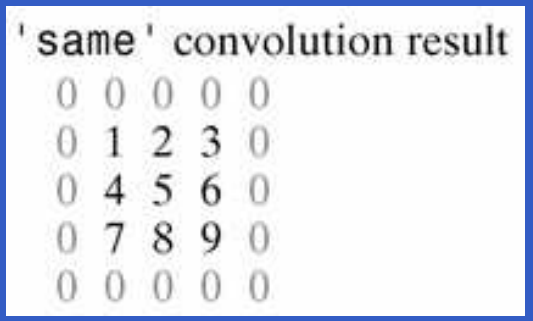
0	0	0	0	0
0	9	8	7	0
0	6	5	4	0
0	3	2	1	0
0	0	0	0	0

Linear Spatial Filtering

Convolution







Linear Spatial Filtering

```
g=imfilter(f,w,filtering_mode,...
           boundary_options,size_options)
```

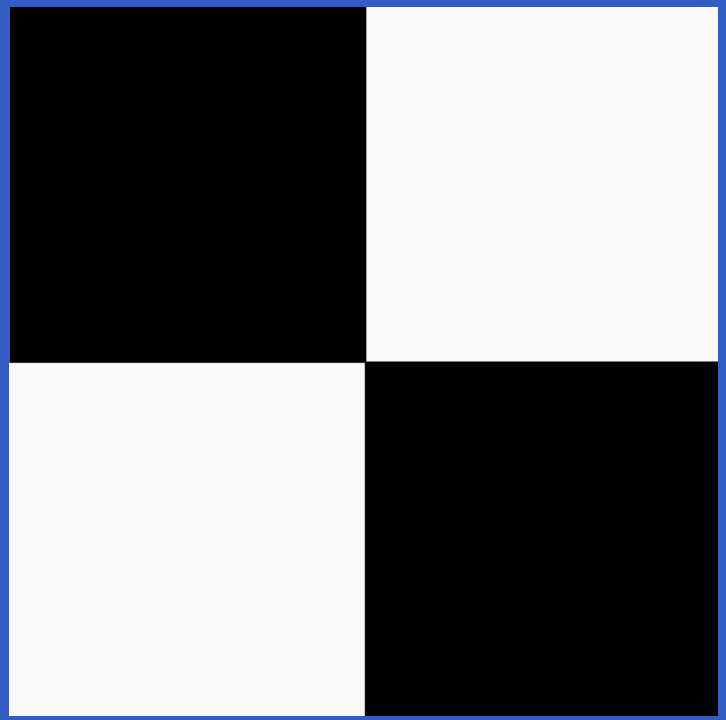
where `f` is the input image, `w` is the filter mask, `g` is the filtered result, and the other parameters are summarized in the following table.

Linear Spatial Filtering

Options	Description
Filtering Mode	
`'corr'`	Filtering is done using correlation. This is the default.
`'conv'`	Filtering is done using convolution.
Boundary Options	
`P`	The boundaries of the input image are extended by padding with a value, `P`. This is the default, with value 0.
`'replicate'`	The size of the image is extended by replicating the values in its outer border.
`'symmetric'`	The size of the image is extended by mirror-reflecting it across its border.
`'circular'`	The size of the image is extended by treating the image as one period a 2-D periodic function.
Size Options	
`'full'`	The output is of the same size as the extended (padded) image.
`'same'`	The output is of the same size as the input. This is the default.

Linear Spatial Filtering

```
>> f=imread('original_test_pattern.tif');
>> f=double(f);
>> w=ones(31);
```

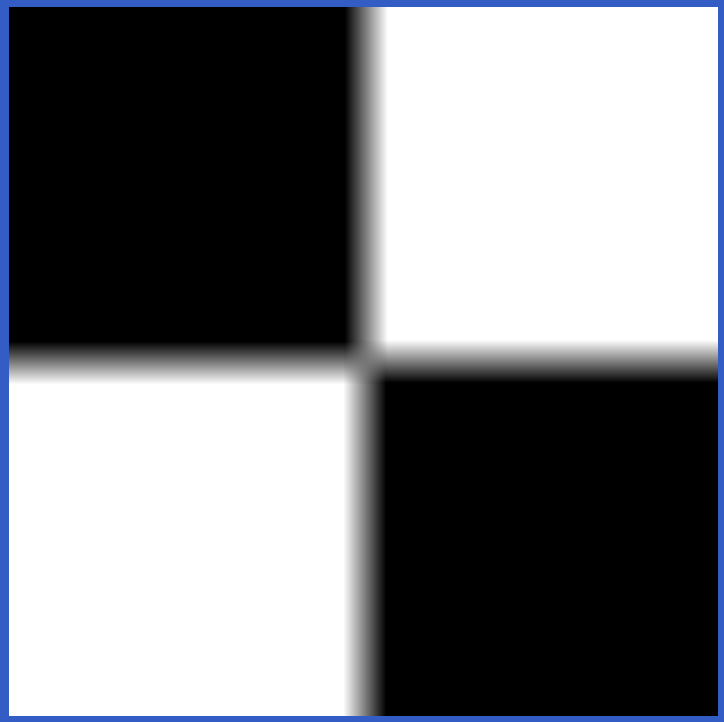


Linear Spatial Filtering

```
>> gd=imfilter(f,w);
>> imshow(gd,[])
```

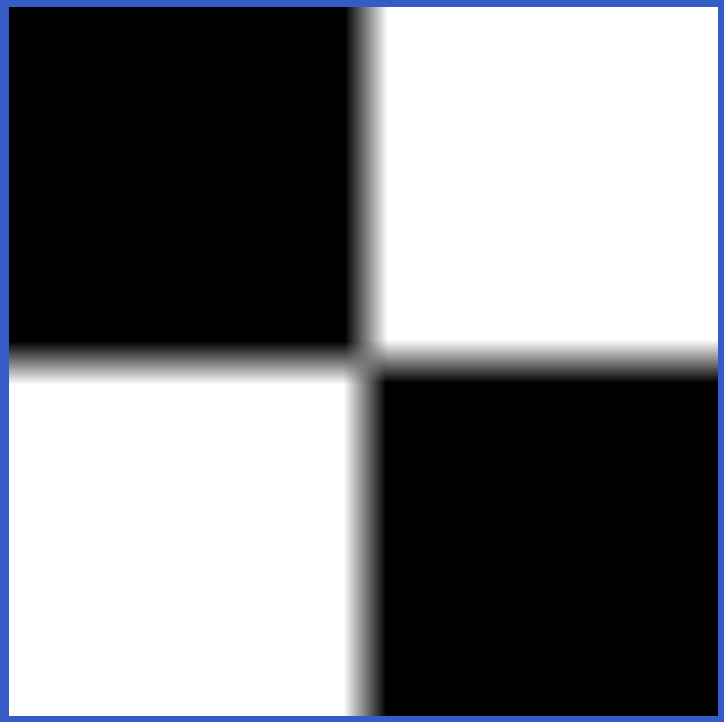
Linear Spatial Filtering

```
gr=imfilter(f,w,'replicate');
imshow(gr,[])
```



Linear Spatial Filtering

```
>> gs=imfilter(f,w,'symmetric');
>> imshow(gs,[])
```

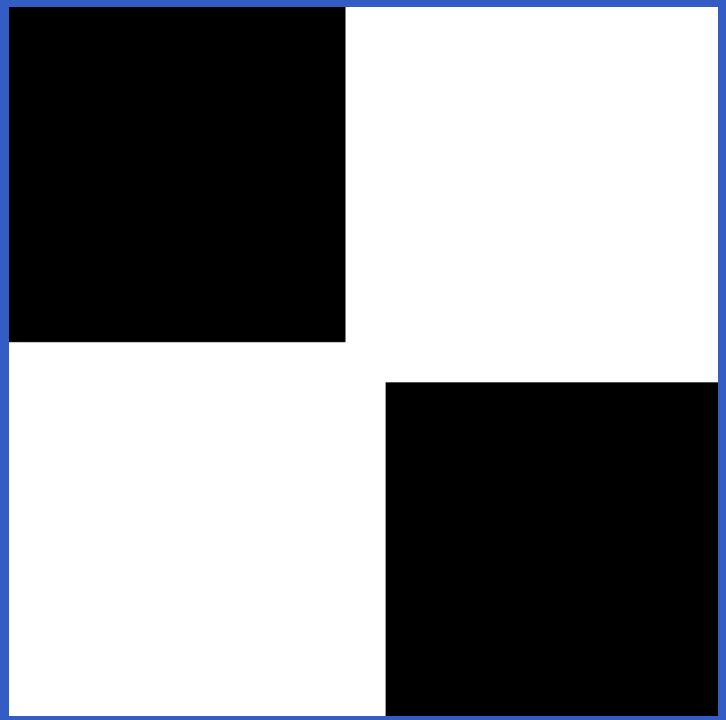


Linear Spatial Filtering

```
>> gc=imfilter(f,w,'circular');
>> imshow(gc,[])
```

Linear Spatial Filtering

```
>> f8=im2uint8(f);
>> g8r=imfilter(f8,w,'replicate');
>> imshow(g8r,[])
```



Nonlinear Spatial Filtering

Nonlinear spatial filtering is based on neighborhood operations also, and the mechanics of defining $m \times n$ neighborhoods by sliding the center point through an image are the same as discussed in linear spatial filtering. Nonlinear spatial filtering is based on nonlinear operations involving the pixels of a neighborhood. For example, letting the response at each center point be equal to the maximum pixel value in its neighborhood is a nonlinear filtering operation. Another basic difference is that the concept of a mask is not as prevalent in nonlinear processing. The ides of filtering carries over, but the "filter" should be visualized as a nonlinear function that operates on the pixels of a neighborhood, and whose response constitutes the response of the operation at the center pixel of the neighborhood.

Nonlinear Spatial Filtering

The IPT provides two functions for performing general nonlinear filtering: `nlfilter` and `colfilt`. The former performs operations directly in 2-D, while `colfilt` organizes the data in the form of columns. Altough `colfilt` requires more memory, it generally executes significantly faster than `nlfilter`. In most image processing applications speed is an overriding factor, so `colfilt` is preferred over `nlfilter` for implementing generalized nonlinear spatial filtering.

Nonlinear Spatial Filtering

Given an input image, `f`, of size $M \times N$, and a neighborhood of size $m \times n$, function `colfilt` generates a matrix, call it `A`, of maximum size $mn \times MN$, in which each column corresponds to the pixels encompassed by the neighborhood centered at a location in the image. For example, the first column corresponds to the pixels encompassed by the neighborhood when its center is located at the top, leftmost point in `f`. All required padding is handled transparently by `colfilt`.

Nonlinear Spatial Filtering

```
g=colfilt(f,[m n],'sliding',@fun,parameters)
```

where `m` and `n` are the dimensions of the filter region, `'sliding'` indicates that the process is one of sliding the $m \times n$ region from pixel to pixel in the input image `f`, `@fun` references a function, which we denote arbitrarily as `fun`, and `parameters` indicates parameters (separated by commas) that may be required by function `fun`. The symbol @ is called a *function handle*, a MATLAB data type that contains information used in referencing a function.

Nonlinear Spatial Filtering

```
fp=padarray(f,[r c],method,direction)
```

where `f` is the input image, `fp` is the padded image, `[r c]` gives the number of rows and columns, by which to pad `f`, and `method` and `direction` are as explained in the next table.

Nonlinear Spatial Filtering

Options	Description
Method	
`'symmetric'`	The size of the image is extended by mirror-reflecting it across its border.
`'replicate'`	The size of the image is extended by replicating the values in its outer border.
`'circular'`	The size of the image is extended by treating the image as one period of a 2-D periodic function.
Direction	
`'pre'`	Pad before the first element of each dimension.
`'post'`	Pad after the last element of each dimension.
`'both'`	Pad before the first element and after the last element of each dimension. This is the default.

Nonlinear Spatial Filtering

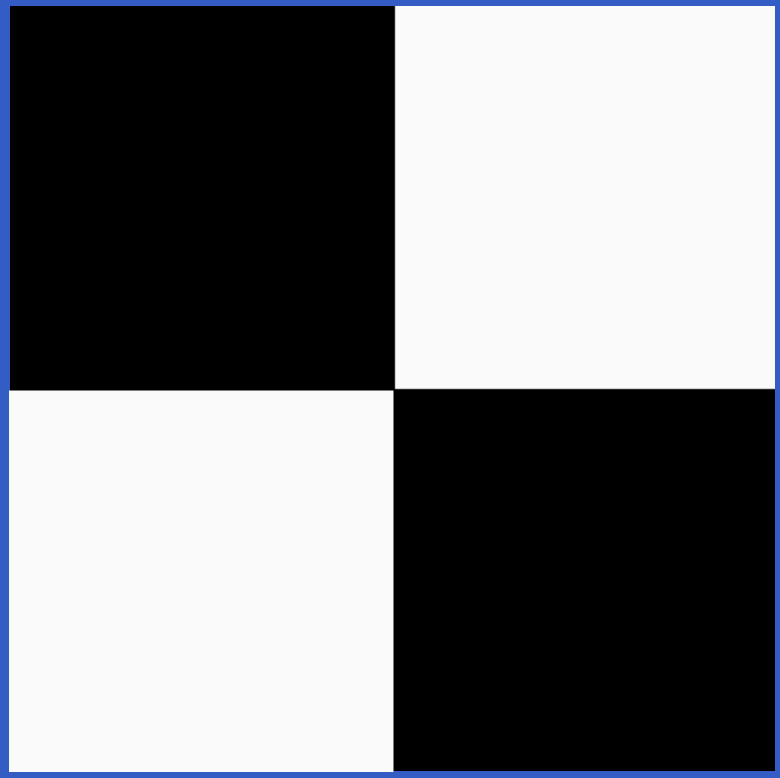
```
>> f=[1 2;3 4];
>> fp=padarray(f,[3 2],'replicate','post')

fp =

     1     2     2     2
     3     4     4     4
     3     4     4     4
     3     4     4     4
     3     4     4     4
```

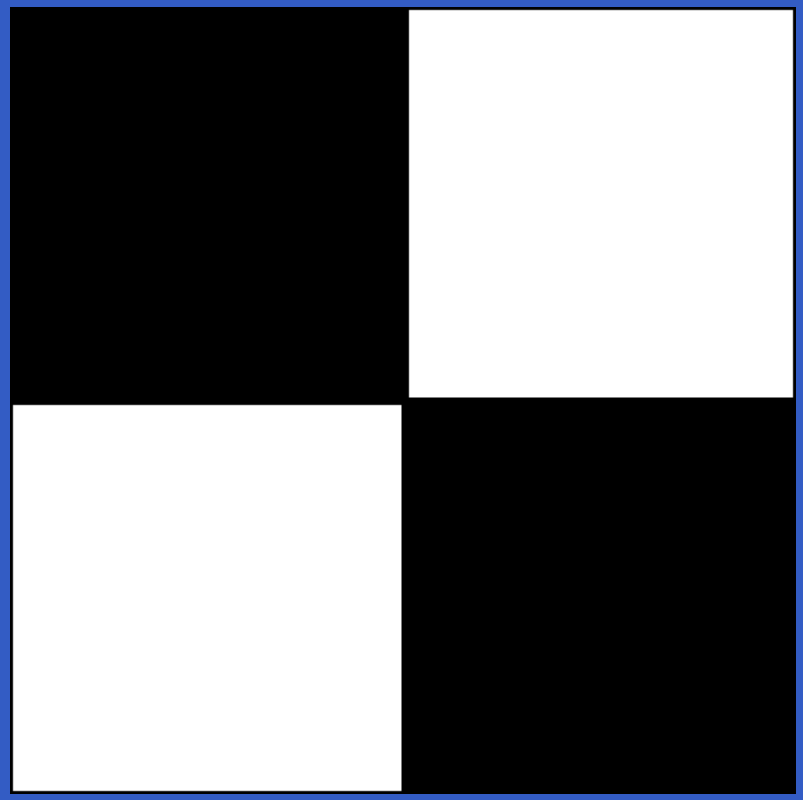
Nonlinear Spatial Filtering

```
function v=gmean(A)

%The length of the columns of A is always mn.
mn=size(A,1);
v=prod(A,1).^(1/mn);
```

```
>> f=padarray(f,[5 5],'replicate');
>> g=colfilt(f,[5 5],'sliding',@gmean);
```

Nonlinear Spatial Filtering





IPT Standard Spatial Filters

- Linear Spatial Filters
- Nonlinear Spatial Filters

Linear Spatial Filters

```
w=fspecial('type',parameters)
```

where `'type'` specifies the filter type, and `parameters` further define the specified filter. The spatial filters supported by `fspecial` are summarized in the following table, including applicable parameters for each filter.

Linear Spatial Filters

Type	Syntax and Parameters
`'average'`	`fspecial('average',[r c])`. A rectangular averaging filter of size `r`$\times$`c`. The default is 3×3. A single number instead of `[r c]` specifies a square filter.
`'disk'`	`fspecial('disk',r)`. A circular averaging filter (within a square of size 2r+1) with radius `r`. The default radius is 5.
`'gaussian'`	`fspecial('gaussian',[r c],sig)`. A Gaussian lowpass filter of size `r`$\times$`c` and standard deviation `sig` (positive). The defaults are 3×3 and 0.5. A single number instead of `[r c]` specifies a square filter.
`'laplacian'`	`fspecial('laplacian',alpha)`. A 3×3 Laplacian filter whose shape is specified by `alpha`, a number in the range $[0, 1]$. The default value for `alpha` is 0.5.
`'log'`	`fspecial('log',[r c],sig)`. Laplacian of a Gaussian (LoG) filter of size `r`$times$`c` and standard deviation `sig` (positive). The defaults are 5×5 and 0.5. A single number instead of `[r c]` specifies a square filter.

Linear Spatial Filters

Type	Syntax and Parameters
`'motion'`	`fspecial('motion',len,theta)`. Outputs a filter that, when convolved with an image, approximates linear motion (of a camera with respect to the image) of `len` pixels. The direction of motion is `theta`, mesaured in degrees, counterclockwise from the horizontal. The defaults are 9 and 0, which represents a motion of 9 pixels in the horizontal direction.
`'prewitt'`	`fspecial('prewitt')`. Outputs a 3×3 Prewitt mask, `wv`, that approximates a vertical gradient. A mask for the horizontal gradient is obtained by transposing the result: `wh=wv'`.
`'sobel'`	`fspecial('sobel')`. Outputs a 3×3 Sobel mask, `sv`, that approximates a vertical gradient. A mask for the horizontal gradient is obtained by transposing the result: `sh=sv'`.
`'unsharp'`	`fspecial('unsharp',alpha)`. Outputs a 3×3 unsharp filter. Parameter `alpha` controls the shape; it must be greater than or equal to 0 and less than or equal to 1.0; the default is 0.2.

Linear Spatial Filters

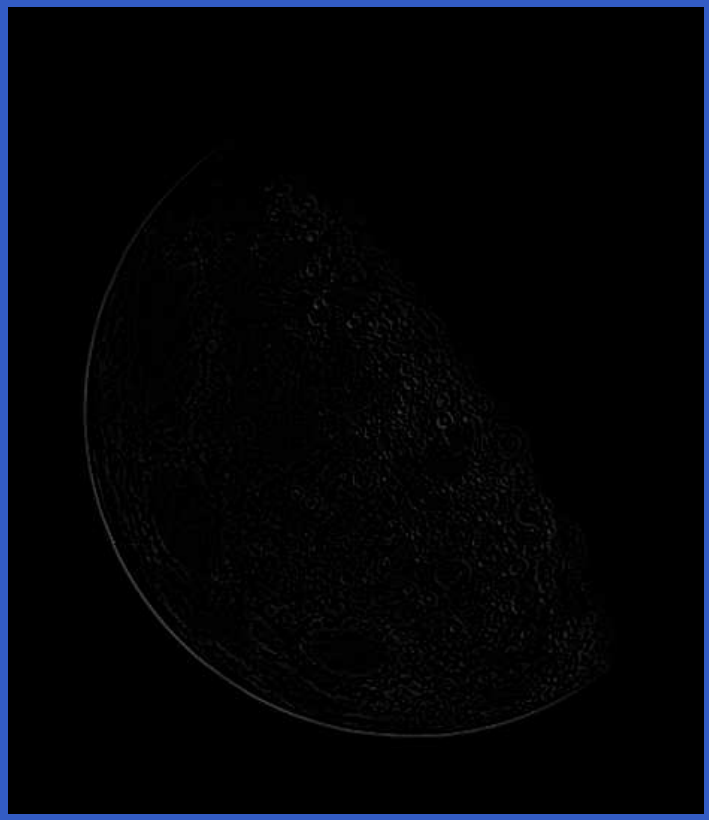
```
>> w=fspecial('laplacian',0)

w =

     0     1     0
     1    -4     1
     0     1     0
```

Linear Spatial Filters

```
>> f=imread('moon.tif');
```

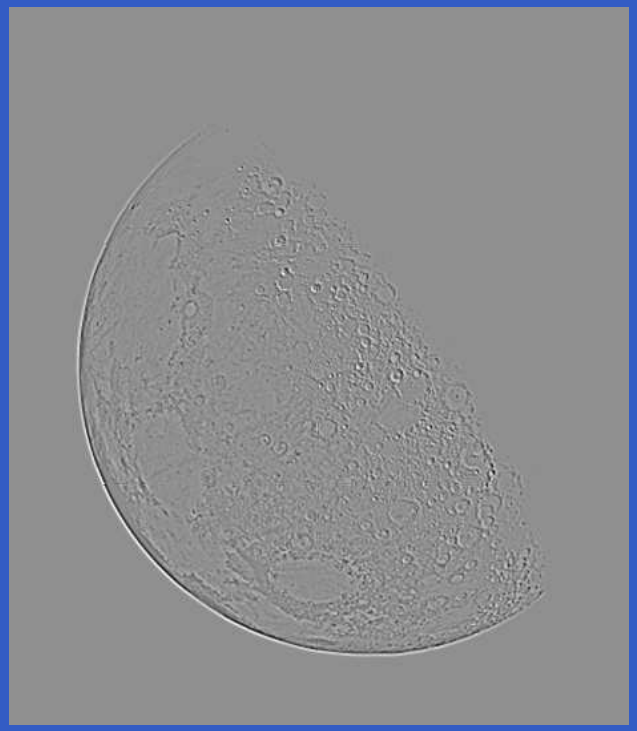
Linear Spatial Filters

```
>> g1=imfilter(f,w,'replicate');
>> imshow(g1,[])
```



Linear Spatial Filters

```
>> f2=im2double(f);
>> g2=imfilter(f2,w,'replicate');
>> imshow(g2,[])
```



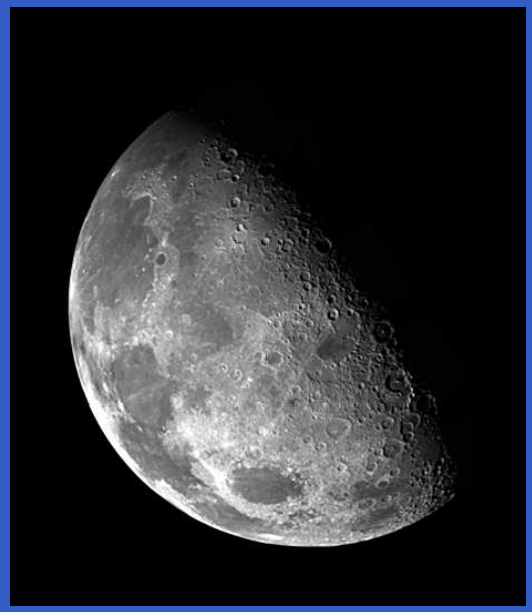
Linear Spatial Filters

```
>> g=f2-g2;
>> imshow(g)
```

Linear Spatial Filters

```
>> f=imread('moon.tif');
>> w4=fspecial('laplacian',0);
>> w8=[1 1 1;1 -8 1;1 1 1];
>> f=im2double(f);
>> g4=f-imfilter(f,w4,'replicate');
>> g8=f-imfilter(f,w8,'replicate');
>> imshow(f)
>> figure, imshow(g4)
>> figure, imshow(g8)
```

Linear Spatial Filters


Nonlinear Spatial Filters

```
g=ordfilt2(f,order,domain
```

This function creates the output image `g` by replacing each element of `f` by the `order`-th element in the sorted set of neighbors specified by the nonzero elements in `domain`. Here, `domain` is an $m \times n$ matrix of 1s and 0s that specify the pixel locations in the neighborhood that are to be used in the computation. In this sense, `domain` acts like a mask. The pixels in the neighborhood that corresponds to 0 in the `domain` matrix are not used in the computation.

Nonlinear Spatial Filters

Min filter of size $m \times n$:

```
g=ordfilt2(f,1,ones(m,n))
```

Max filter of size $m \times n$:

```
g=ordfilt2(f,m*n,ones(m,n))
```

Median filter of size $m \times n$:

```
g=ordfilt2(f,median(1:m*n),ones(m,n))
```

Nonlinear Spatial Filters

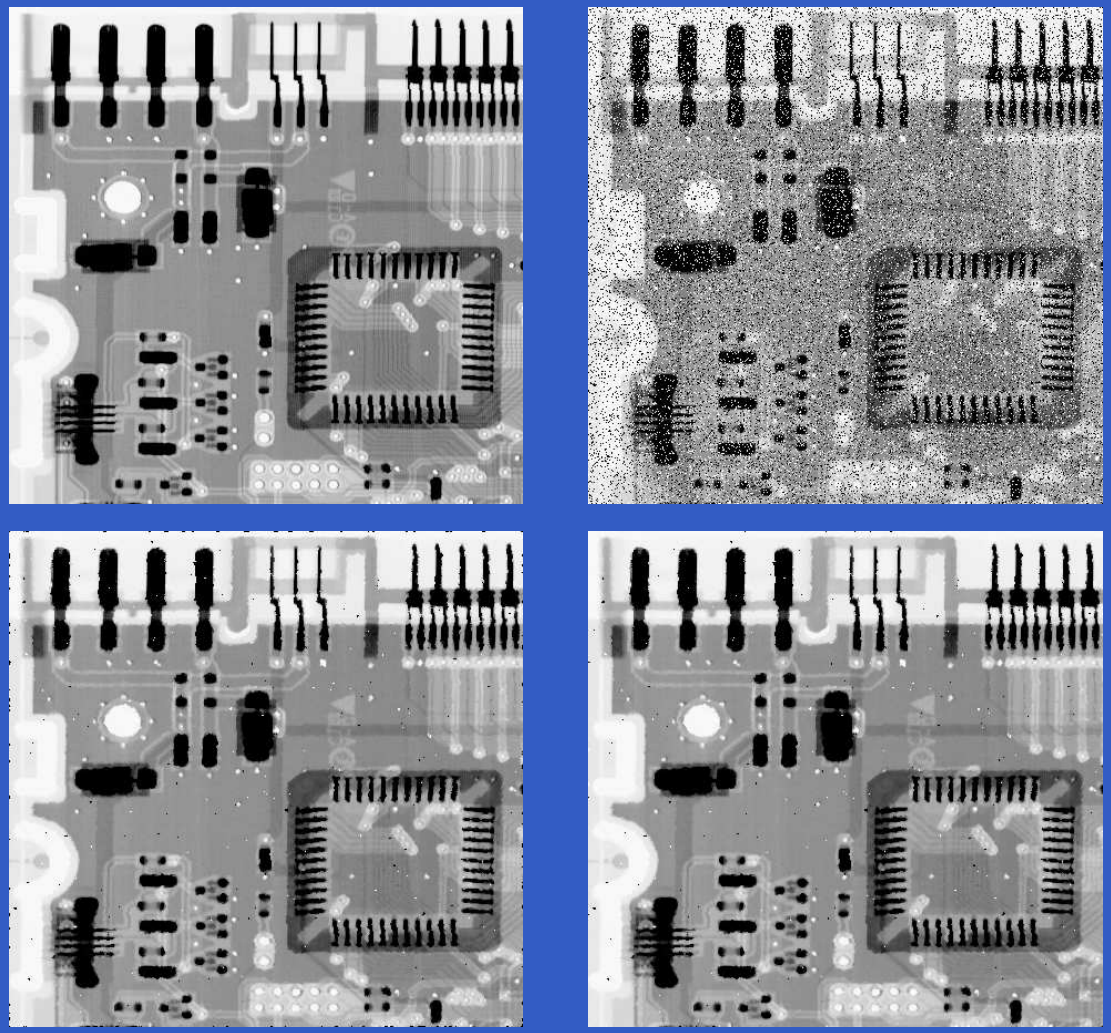
```
g=medfilt2(f,[m n],padopt
```

where the tuple `[m n]` defines a neighborhood of size $m \times n$ over which the median is computed, and `padopt` specifies one of three possible border padding options: `'zeros` (the default), `'symmetric'` in which `f` is extended symmetrically by mirror-reflecting it across its border, and `'indexed'`, in which `f` is padded with `1`s if it is of class `double` and with `0`s otherwise. The default form of this function is `g=medfilt2(f)` which uses a 3×3 neighborhood to compute the median, and pads the border of the input with `0`s.

Nonlinear Spatial Filters

```
>> f=imread('ckt-board.tif');
>> fn=imnoise(f,'salt & pepper',0.2);
>> gm=medfilt2(fn);
>> gms=medfilt2(fn,'symmetric');
>> subplot(2,2,1), imshow(f)
>> subplot(2,2,2), imshow(fn)
>> subplot(2,2,3), imshow(gm)
>> subplot(2,2,4), imshow(gms)
```

Nonlinear Spatial Filters



Chapter 4

Frequency Domain Processing

Content

- The 2-D Discrete Fourier Transform
- Computing and Visualizing the 2-D DFT in MATLAB
- Filtering in the Frequency Domain

The 2-D Discrete Fourier Transform

Let $f(x, y)$, for $x = 0, 1, 2, \ldots, M - 1$ and $y = 0, 1, 2, \ldots, N - 1$, denote an $M \times N$ image. The 2-D, *discrete Fourier transform* (DFT) of f, denoted by $F(u, v)$, is given by the equation

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-j2\pi(ux/M + vy/N)}$$

for $u = 0, 1, 2, \ldots, M - 1$ and $v = 0, 1, 2, \ldots, N - 1$.

The 2-D Discrete Fourier Transform

The *frequency domain* is simply the coordinate system spanned by $F(u, v)$ with u and v as (frequency) variables. This is analogous to the *spatial domain* studied in the previous lecture, which is the coordinate systam spanned by $f(x, y)$, with x and y as (spatial) variables. The $M \times N$ rectangular region defined by $u = 0, 1, 2, \ldots, M - 1$ and $v = 0, 1, 2, \ldots, N - 1$ is often referred to as the *frequency rectangle*.

The 2-D Discrete Fourier Transform

The inverse, discrete Fourier transform is given by

$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{j2\pi(ux/M+vy/N)}$$

for $x = 0, 1, 2, \ldots, M-1$ and $y = 0, 1, 2, \ldots, N-1$. Thus, given $F(u,v)$, we can obtain $f(x,y)$ back by means of the inverse DFT. The values of $F(u,v)$ in this equation sometimes are referred to as the *Fourier coefficients* of the expansion.

The 2-D Discrete Fourier Transform

Because array indices in MATLAB start at 1, rather than 0, `F(1,1)` and `f(1,1)` in MATLAB corresponds to the mathematical quantities $F(0,0)$ and $f(0,0)$ in the transform and its inverse.

The 2-D Discrete Fourier Transform

Even if $f(x, y)$ is real, its transform in general is complex. The principal method of visually analyzing a transform is to compute its *spectrum* and display it as an image. Letting $R(u, v)$ and $I(u, v)$ represent the real and imaginary components of $F(u, v)$, the Fourier spectrum is defined as

$$|F(u, v)| = \sqrt{R^2(u, v) + I^2(u, v)}$$

The *phase angle* of the transform is defined as

$$\phi(u, v) = \tan^{-1}\left[\frac{I(u, v)}{R(u, v)}\right]$$

The 2-D Discrete Fourier Transform

The *power spectrum* is defined as the square of the magnitude:

$$P(u,v) = |F(u,v)|^2 = R^2(u,v) + I^2(u,v)$$

For purposes of visualization it typically is immaterial whether we view $|F(u,v)|$ or $P(u,v)$.

The 2-D Discrete Fourier Transform

If $f(x, y)$ is real, its Fourier transform is conjugate symmetric about the origin; that is,

$$F(u, v) = F^*(-u, -v)$$

which implies that the Fourier spectrum also is symmetric about the origin:

$$|F(u, v)| = |F(-u, -v)|$$

The 2-D Discrete Fourier Transform

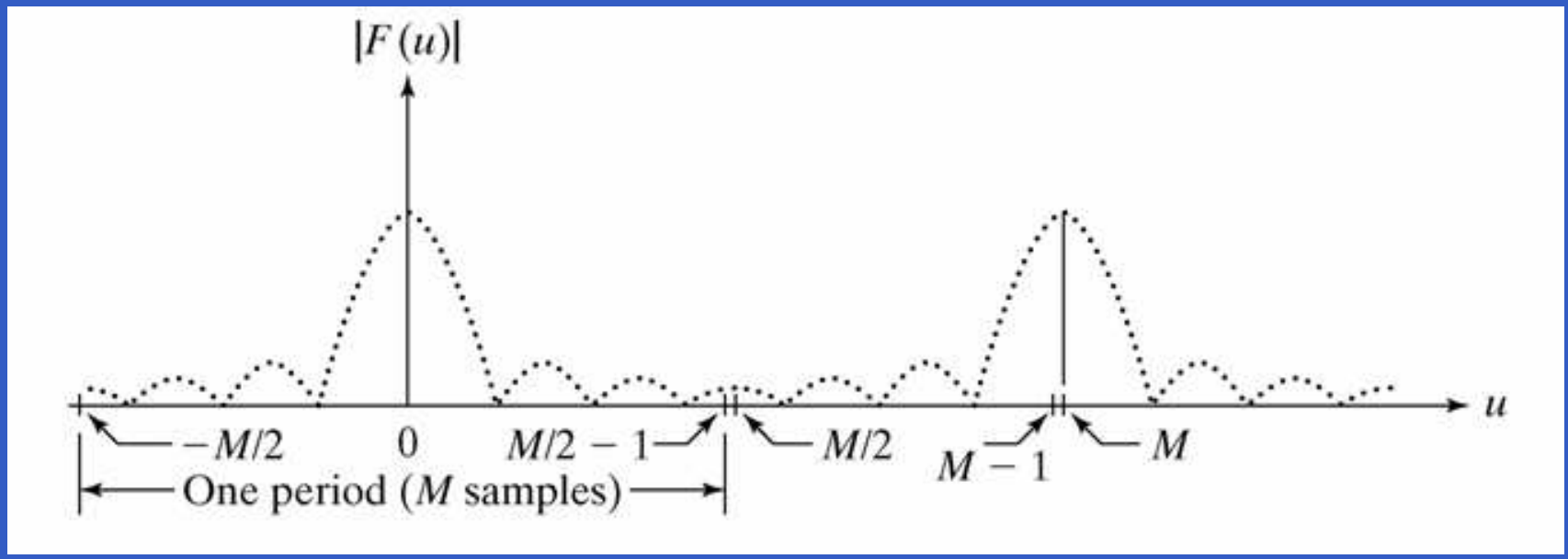
It can be shown by direct substitution into the equation for $F(u, v)$ that

$$F(u, v) = F(u + M, v) = F(u, v + N) = F(u + M, v + N)$$

In other words, the DFT is infinitely periodic in both the u and v directions, with the periodicity determined by M and N. Periodicity is also a property of the inverse DFT:

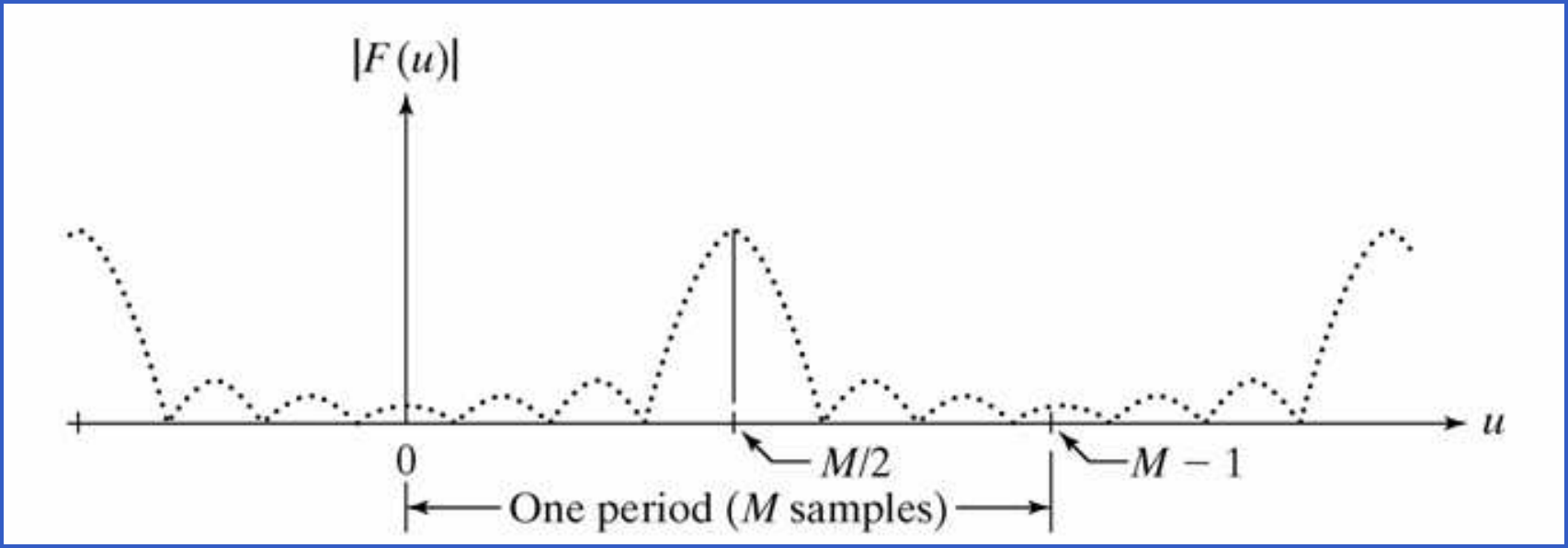
$$f(x, y) = f(x + M, y) = f(x, y + N) = f(x + M, y + N)$$

The 2-D Discrete Fourier Transform



Fourier spectrum showing back-to-back half periods in the interval $[0, M-1]$.

The 2-D Discrete Fourier Transform



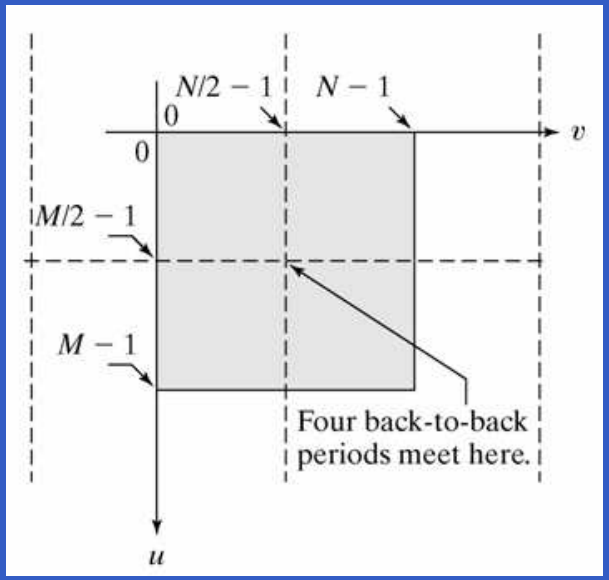
Centered spectrum in the interval $[0, M-1]$ obtained by multiplying $f(x)$ by $(-1)^x$ prior to computing the Fourier transform.

Computing and Visualizing the 2-D DFT in MATLAB

The DFT and its inverse are obtained in practice using a fast Fourier transform (FFT) algorithm. The FFT of an $M \times N$ image array `f` is obtained in the toolbox with function `fft2`, which has the simple syntax:

```
F=fft2(f)
```

This function returns a Fourier transform that is also of size $M \times N$, with the data arranged in the form shown in figure; that is, with the origin of the data at the top left, and with four quarter periods meeting at the center of the frequency rectangle.



Computing and Visualizing the 2-D DFT in MATLAB

As explained later, it is necessary to pad the input image with zeros when the Fourier transform is used for filtering. In this case, the syntax becomes

```
F=fft2(f,P,Q)
```

With this syntax, `fft2` pads the input with the required number of zeros so that the resulting function is of size $P \times Q$.

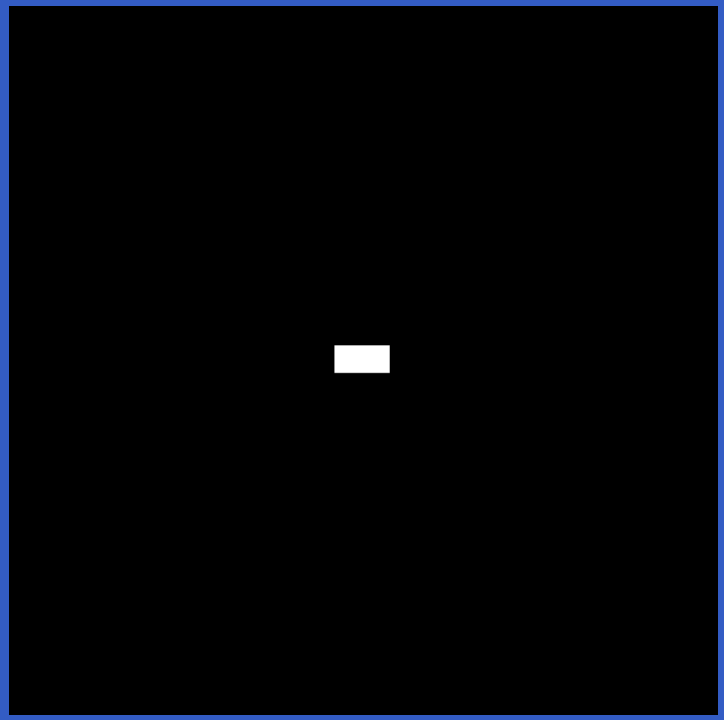
The Fourier spectrum is obtained by using function `abs`:

```
S=abs(F)
```

which computes the magnitude (square root of the sum of the squares of the real and imaginary parts) of each element of the array.

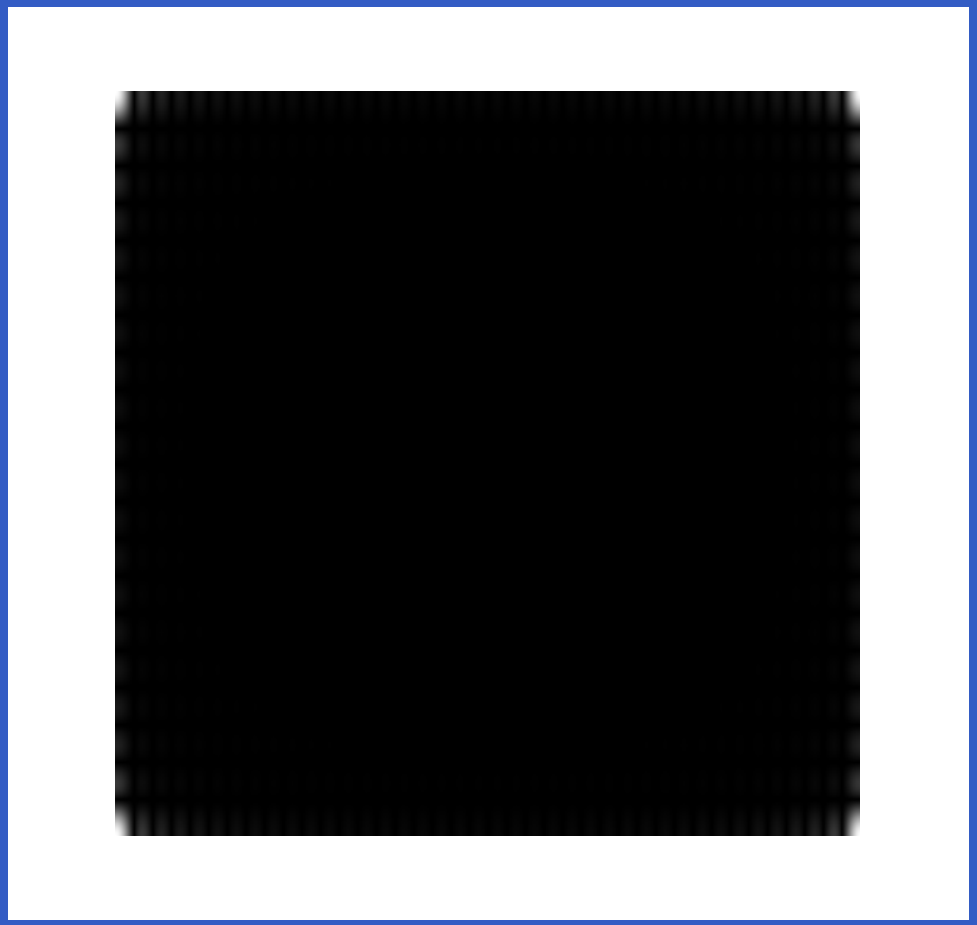
Computing and Visualizing the 2-D DFT in MATLAB

Visual analysis of the spectrum by displaying it as an image is an important aspect of working in the frequency domain. As an illustration, consider the simple image, `f`, in the figure.



Computing and Visualizing the 2-D DFT in MATLAB

```
>> F=fft2(f);
>> S=abs(F);
>> imshow(S,[])
```



Computing and Visualizing the 2-D DFT in MATLAB

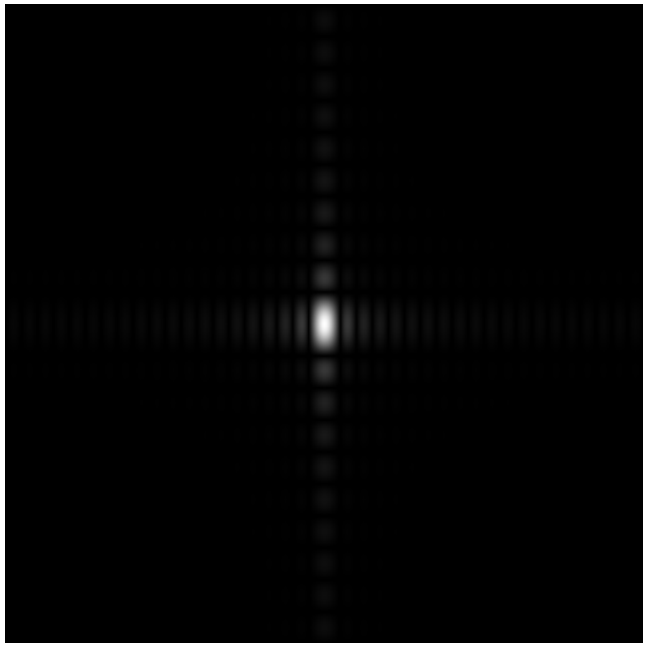
IPT function `fftshift` can be used to move the origin of the transform to the center of the frequency rectangle. The syntax is

```
Fc=fftshift(F)
```

where `F` is the transform computed using `fft2` and `Fc` is the centered transform. Function `fftshift` operates by swapping quadrants of `F`. For example if `a=[1 2;3 4]`, `fftshift(a)=[4 3;2 1]`.

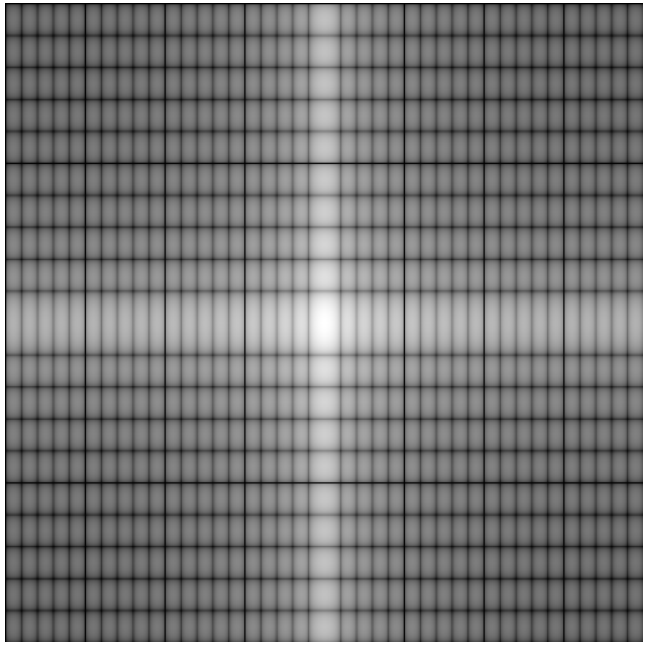
Computing and Visualizing the 2-D DFT in MATLAB

```
>> Fc=fftshift(F);
>> imshow(abs(Fc),[])
```



Computing and Visualizing the 2-D DFT in MATLAB

```
>> S2=log(1+abs(Fc));
>> imshow(S2,[])
```



Computing and Visualizing the 2-D DFT in MATLAB

Function `ifftshift` reverses the centering. Its syntax is

```
F=ifftshift(Fc)
```

This function can be used to convert a function that is initially centered on a rectangle to a function whose center is at the top, left corner of the rectangle.

Computing and Visualizing the 2-D DFT in MATLAB

While on the subject of centering, keep in mind that the center of the frequency rectangle is at $(M/2, N/2)$ if the variables u and v run from 0 to $M-1$ and $N-1$, respectively. For example, the center of an 8×8 frequency square is at point $(4, 4)$, which is the 5th point along each axis, counting up from $(0, 0)$. If, as in MATLAB, the variables run from 1 to M and 1 to N, respectively, then the center of the square is at $(M/2 + 1, N/2 + 1)$. In the case of our 8×8 example, the center would be at point $(5, 5)$, counting up from $(1, 1)$. Obviously, the two centers are the same point, but this can be a source of confusion when deciding how to specify the location of DFT centers in MATLAB computations.

Computing and Visualizing the 2-D DFT in MATLAB

If M and N are odd, the center for MATLAB computations is obtained by rounding $M/2$ and $N/2$ down to the closest integer. The rest of the analysis is as in the previous slide. For example, the center of a 7×7 region is at $(3, 3)$ if we count up from $(0, 0)$ and at $(4, 4)$ if we count up from $(1, 1)$. Using MATLAB's function `floor`, and keeping in mind that the origin is at $(1, 1)$, the center of the frequency rectangle for MATLAB computations is at

```
[floor(M/2)+1,floor(N/2)+1]
```

The center given by this expression is valid both for odd and even values of M and N.

Computing and Visualizing the 2-D DFT in MATLAB

We point out that the inverse Fourier transform is computed using function `ifft2`, which has the basic syntax

```
f=ifft2(F)
```

where `F` is the Fourier transform and `f` is the resulting image.

Computing and Visualizing the 2-D DFT in MATLAB

If the input used to compute `F` is real, the inverse in theory should be real. In practice, however output of `ifft2` often has very small imaginary components resulting from round-off errors that are characteristic of floating point computations. Thus, it is good practice to extract the real part of the result after computing the inverse to obtain an image consisting only of real values. The tow operations can be combined:

```
>> f=real(ifft2(F));
```

As in the forward case, this function has the alternate format `ifft2(F,P,Q)`, which pads `F` with zeros so that its size is $P \times Q$ before computing the inverse.

Filtering in the Frequency Domain

- Fundamental Concepts
- Basic Steps in DFT Filtering
- An M-function for Filtering in the Frequency Domain

Fundamental Concepts

Formally, the discrete convolution of two function $f(x, y)$ and $h(x, y)$ of size $M \times N$ is denoted by $f(x, y) * h(x, y)$ and is defined by the expression

$$f(x, y) * h(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) h(x - m, y - n).$$

The minus sign simply means that function h is mirrored about the origin.

Fundamental Concepts

$$
\begin{aligned}
f(x,y) &= \mathcal{F}^{-1}\left[F(u,v)\right](x,y) = \\
&= \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)e^{j2\pi(ux/M+vy/N)}
\end{aligned}
$$

$$
\begin{aligned}
g(x,y) &= \mathcal{F}^{-1}\left[G(u,v)\right](x,y) = \\
&= \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} G(u,v)e^{j2\pi(ux/M+vy/N)}
\end{aligned}
$$

Fundamental Concepts

$$
\begin{aligned}
f * g &= \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m,n)g(x-m,y-n) = \\
&= \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m,n)\left(\frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} G(u,v)e^{j2\pi(u(x-m)/M+v(y-n)/N)}\right) = \\
&= \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} G(u,v)e^{j2\pi(ux/M+vy/N)} \cdot \\
&\cdot \left(\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m,n)e^{-j2\pi(um/M+vn/N)}\right) = \\
&= \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} G(u,v)e^{j2\pi(ux/M+vy/N)} \cdot F(u,v) = \\
&= (F)^{-1}\left[F(u,v)G(u,v)\right]
\end{aligned}
$$

Fundamental Concepts

The foundation for linear filtering in both spatial and frequency domains is the convolution theorem, which may be written as

$$f(x,y) * h(x,y) \Leftrightarrow H(u,v)F(u,v)$$

and, conversely,

$$f(x,y)h(x,y) \Leftrightarrow H(u,v) * H(u,v)$$

Here, the symbol "$*$" indicates convolution of the two functions, and the expressions on the sides of the double arrow constitute a Fourier transform pair.

Fundamental Concepts

The previous equation is really nothing more than an implementation for

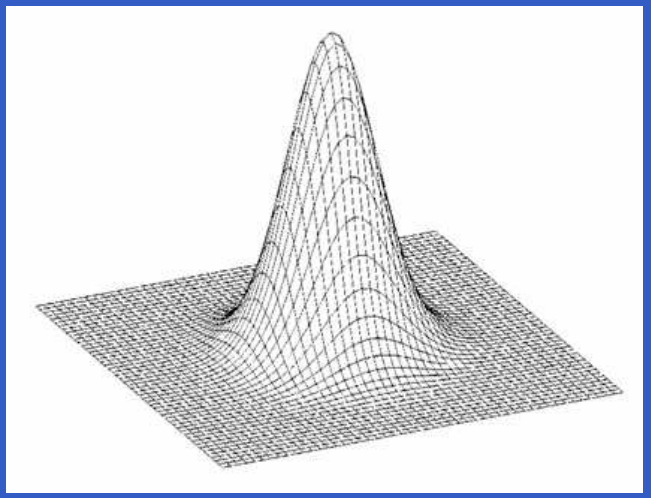
1. flipping one function about the origin;
2. shifting that function with respect to the other by changing the values of (x, y); and
3. computing a sum of products over all values of m and n, for *each* displacement (x, y).

Fundamental Concepts

Filtering in the spatial domain consists of convolving an image $f(x, y)$ with a filter mask, $h(x, y)$. According to the convolution theorem, we can obtain the same result in the frequency domain by multiplying $F(u, v)$ by $H(u, v)$, the Fourier transform of the spatial filter. It is customary to refer to $H(u, v)$ as the *filter transfer function*.

Fundamental Concepts

Basically, the idea in frequency domain filtering is to select a filter transfer function that modifies $F(u, v)$ in a specified manner. For example, the filter in the figure has a transfer function that, when multiplied by a centered $F(u, v)$, attenuates the high-frequency components of $F(u, v)$, while leaving the low frequencies relatively unchanged. Filters with this characteristic are called *lowpass* filters.



Fundamental Concepts

Based on the convolution theorem, we know that to obtain the corresponding filtered image in the spatial domain we simply compute the inverse Fourier transform of the product $H(u,v)F(u,v)$. It is important to keep in mind that the process just described is identical to what we would obtain by using convolution in the spatial domain, as long as the filter mask, $h(x,y)$, is the inverse Fourier transform of $H(u,v)$. In practice, spatial convolution generally is simplified by using small masks that attempt to capture the salient features of their frequency domain counterparts.

Fundamental Concepts

As noted earlier images and their transforms are automatically considered periodic if we elect to work with DFTs to implement filtering. It is not difficult to visualize that convolving periodic functions can cause interference between adjacent periodics if the periods are close with respect to the duration of the nonzero parts of the functions. This interference, called *wraparound error*, can be avoided by padding the functions with zeros, in the followin manner.

Fundamental Concepts

Assume that functions $f(x, y)$ and $h(x, y)$ are of size $A \times B$ and $C \times D$, respectively. We form two *expanded (padded)* functions, both of size $P \times Q$ by appending zeros to f and g. It can be shown that wraparound error is avoided by choosing

$$P \geq A + C - 1$$

and

$$Y \geq B + D - 1$$

Most of the work in this chapter deals with functions of the same size, $M \times N$, in which case we use the following padding values: $P \geq 2M - 1$ and $Q \geq 2N - 1$.

Fundamental Concepts

```
function PQ=paddedsize(AB,CD,PARAM)
%PADDEDSIZE Computes padded sizes useful for FFT-based filtering.
%   PQ=PADDEDSIZE(AB), where AB is a two-element size vector,
%   computes the two-element size vector PQ=2*AB.
%
%   PQ=PADDEDSIZE(AB,'PWR2') computes the vector PQ such that
%   PQ(1)=PQ(2)=2^nextpow2(2*m), where m is MAX(AB).
%
%   PQ=PADDEDSIZE(AB,CD), where AB and CD are two-element size
%   vectors, computes the two-element size vector PQ. The elements
%   of PQ are the smallest even integers greater than or equal to
%   AB+CD-1.
%
%   PQ=PADDEDSIZE(AB,CD,'PWR2') computes the vector PQ such that
%   PQ(1)=PQ(2)=2^nextpow2(2*m), where m is MAX([AB CD]).
```

Fundamental Concepts

```
if nargin==1
    PQ=2*AB;
elseif nargin==2 & ~ischar(CD)
    PQ=AB+CD-1;
    PQ=2*ceil(PQ/2);
elseif nargin==2
    m=max(AB); %Maximum dimension.

    % Find power-of-2 at least twice m.
    P=2^nextpow2(2*m);
    PQ=[P,P];
elseif nargin==3
    m=max([AB CD]); %Maximum dimension.
    P=2^nextpow2(2*m);
    PQ=[P,P];
else
    error('Wrong number of inputs.')
end
```

Fundamental Concepts

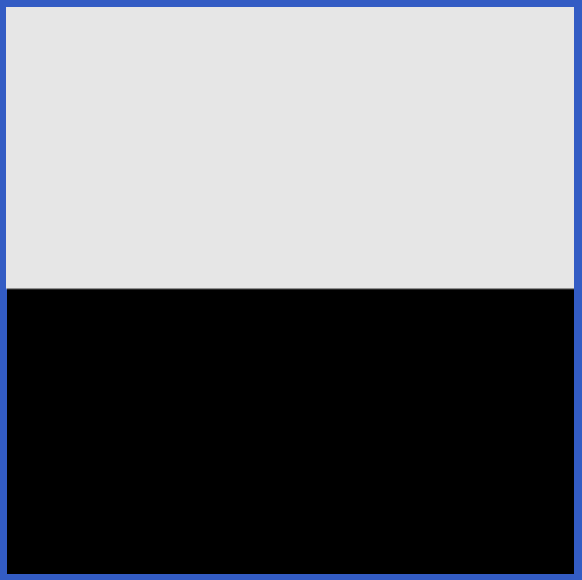
With `PQ` thus computed using function paddedsize, we use the following syntax for `fft2` to compute the FFT using zero padding:

```
F=fft2(f,PQ(1),PQ(2))
```

This syntax simply appends enough zeros to `f` such that the resulting image is of size `PQ(1)`×`PQ(2)`, and then computes the FFT as previously described. Note that when using padding the filter function in the frequency domain must be of size `PQ(1)`×`PQ(2)` also.

Fundamental Concepts

The image, `f`, in the figure is used to illustrate the difference between filtering with and without padding. In the following discussion we use function `lpfilter` to generate a Gaussian lowpass filter with a specified value of sigma (`sig`).



Fundamental Concepts

```
>> f=imread('square_original.tif');
>> [M,N]=size(f);
>> F=fft2(f);
>> sig=10;
>> H=lpfilter('gaussian',M,N,sig);
>> G=H.*F;
>> g=real(ifft2(G));
>> imshow(g,[])
```

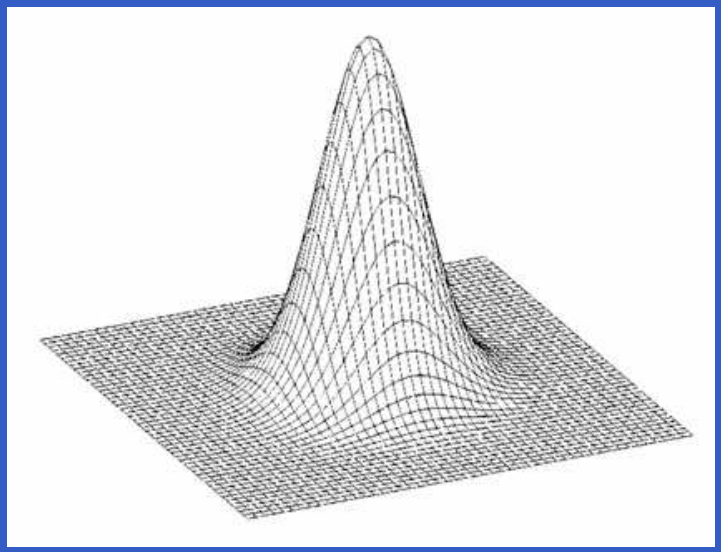
Fundamental Concepts

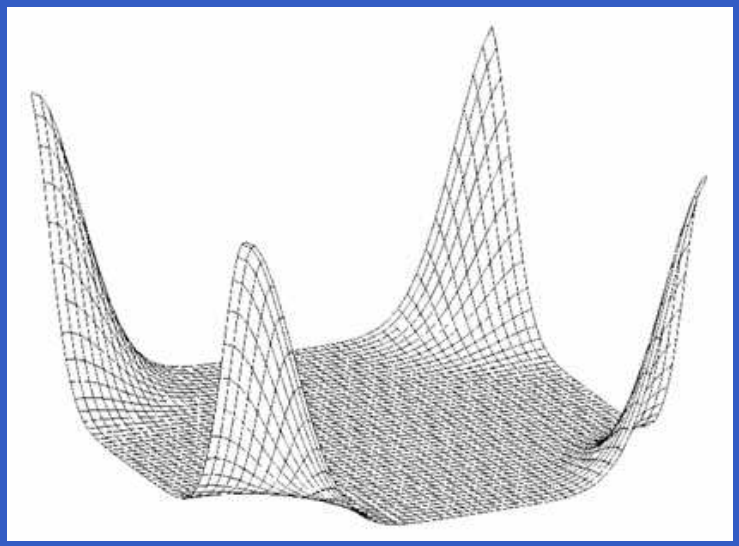
```
>> PQ=paddedsize(size(f));
   %Compute the FFT with padding.
>> Fp=fft2(f,PQ(1),PQ(2));
>> Hp=lpfilter('gaussian',PQ(1),PQ(2),2*sig);
>> Gp=Hp.*Fp;
>> gp=real(ifft2(Gp));
>> gpc=gp(1:size(f,1),1:size(f,2));
>> imshow(gp,[])
>> imshow(gpc,[])
```

Basic Steps in DFT Filtering

1. Obtain the padding parameters using function `paddedsize`:
   `PQ=paddedsize(size(f));`
2. Obtain the Fourier transform with padding:
   `F=fft2(f,PQ(1),PQ(2));`
3. Generate a filter function, `H`, of size `PQ(1)`$\times$`PQ(2)` using any of the methods discussed later. The filter must be in the format shown in the left side figure on the next slide. If it is centered instead, as in the right side figure on the next slide, let `H=fftshift(H)` before using the filter.

Basic Steps in DFT Filtering

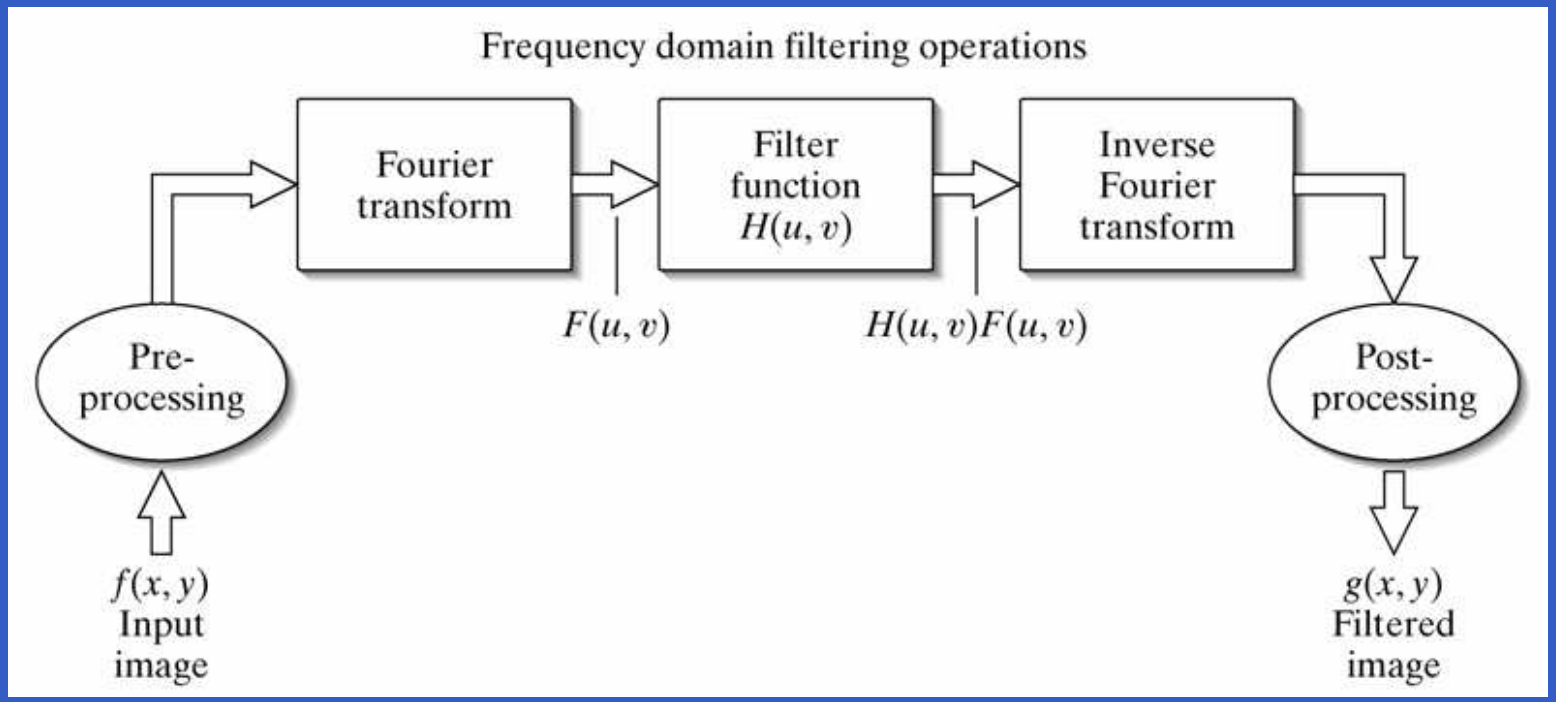




Basic Steps in DFT Filtering

4. Multiply the transform by the filter:
   `G=H.*F`
5. Obtain the real part of the inverse FFT of `G`:
   `g=real(ifft2(G));`
6. Crop the top, left rectangle to the original size:
   `g=g(1:size(f,1),1:size(f,2));`

Basic Steps in DFT Filtering



Basic Steps in DFT Filtering

It is well known from linear system theory that, under certain mild conditions, inputting an impulse into a linear system completely characterizes the system. When working with finite, discrete data, as we do, the response of a linear system, including the response to an impulse, also is finite. If the linear system is just a spatial filter, then we can completely determine the filter simply by observing its response to an impulse. A filter determined in this manner is called a *finite-impulse-response* (FIR) *filter*.

An M-function for Filtering in the Frequency Domain

```
function g=dftfilt(f,H)
%DFTFILT Performs frequency domain filtering.
%   G=DFTFILT(F,H) filters F in the frequency domain using the
%   filter transfer function H. The output, G, is the filtered
%   image, which has the same size as F. DFTFILT automatically pads
%   F to be the same size as H. Function PADDEDSIZE can be used
%   to determine an appropriate size for H.
%
%   DFTFILT assumes that F is real and that H is a real, uncentered,
%   circularly-symmetric filter function.

%Obtain the FFT of the padded input.
F=fft2(f,size(H,1),size(H,2));

%Perform filtering.
g=real(ifft2(H.*F));

%Crop to original size.
g=g(1:size(f,1),1:size(f,2));
```

Obtaining Frequency Domain Filters from Spatial Filters

Function `freqz2` computes the frequency response of FIR filters. The result is the desired filter in the frequency domain. Th syntax of interest in the present discussion is

```
H=freqz2(h,R,C)
```

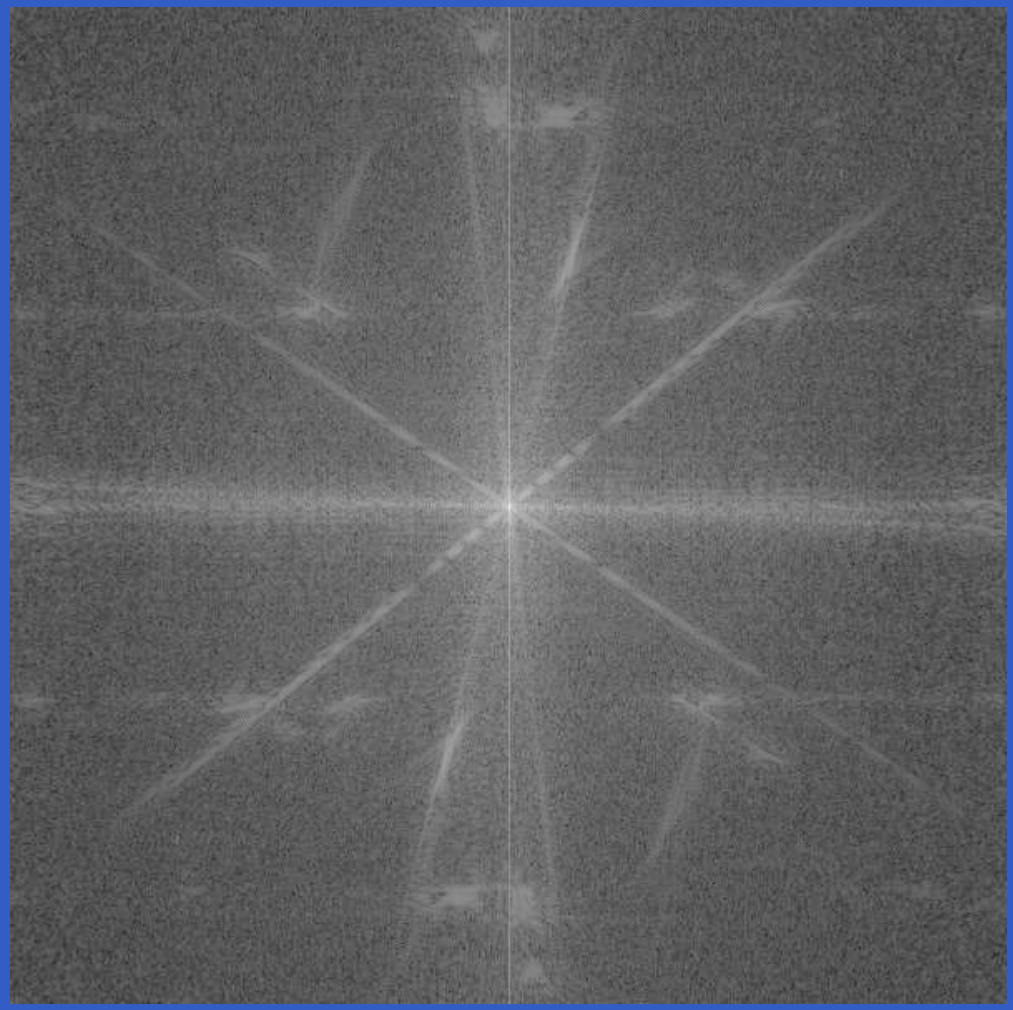
where `h` is a 2-D spatial filter and `H` is the corresponding 2-D frequency domain filter. Here, `R` is the number of rows, and `C` the number of columns that we wish filter `H` to have. Generally, we let `R=PQ(1)` and `C=PQ(2)`. If `freqz2` is written without an output argument, the absolute value of `H` is displayed on the MATLAB desktop as a 3-D perspective plot.

Obtaining Frequency Domain Filters from Spatial Filters

```
>> f=imread('bld.tif');
```

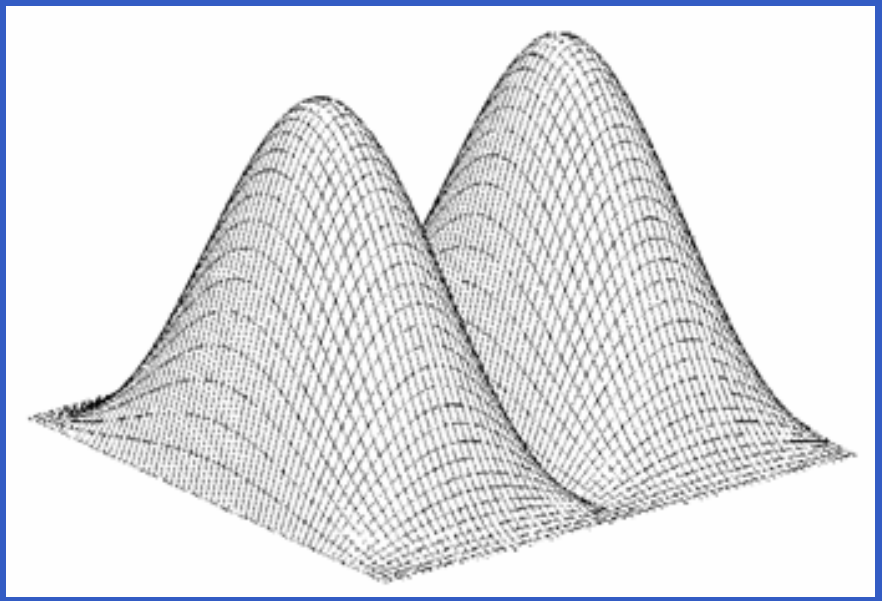
Obtaining Frequency Domain Filters from Spatial Filters

```
>> F=fft2(f);
>> S=fftshift(log(1+abs(F)));
>> S=gscale(S);
>> imshow(S)
```



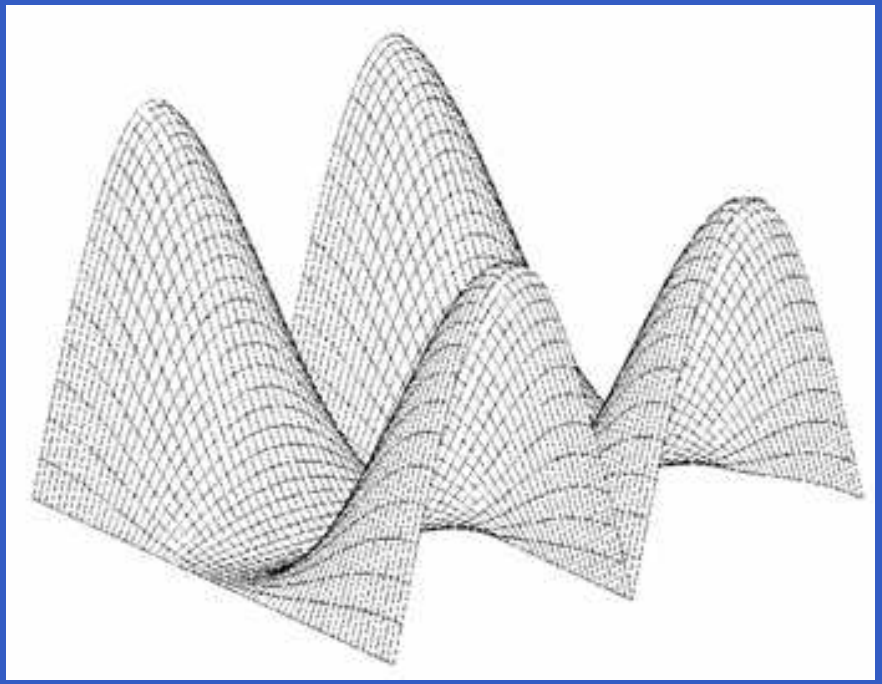
Obtaining Frequency Domain Filters from Spatial Filters

```
>> h=fspecial('sobel')'

h =

     1     0    -1
     2     0    -2
     1     0    -1

>> freqz2(h)
```



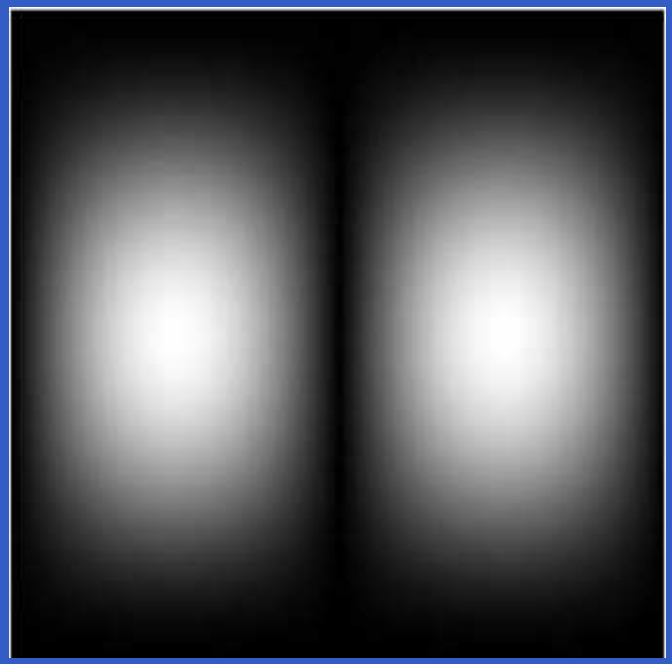
Obtaining Frequency Domain Filters from Spatial Filters

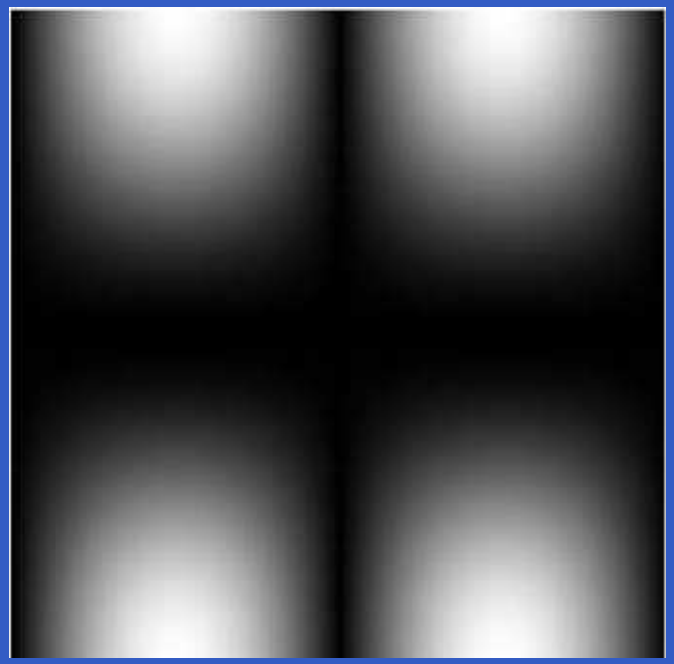
```
>> PQ=paddedsize(size(f));
>> H=freqz2(h,PQ(1),PQ(2));
>> H1=ifftshift(H);
```



Obtaining Frequency Domain Filters from Spatial Filters

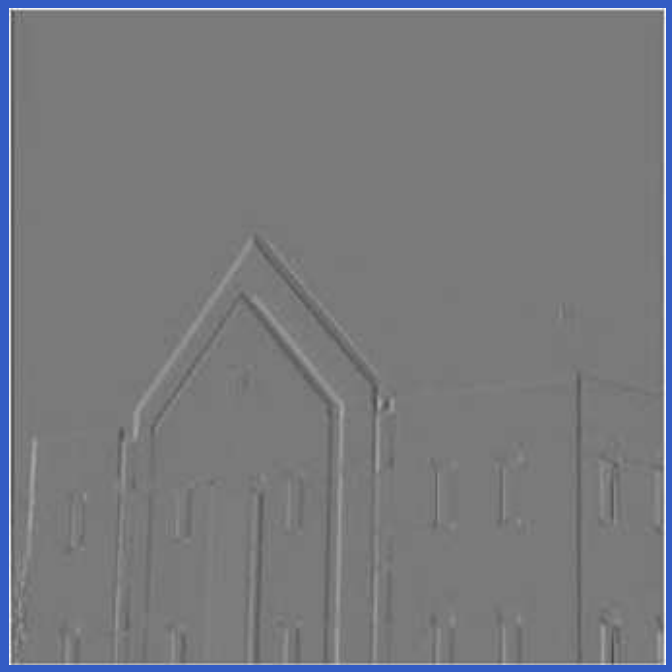
```
>> imshow(abs(H),[])
>> figure, imshow(abs(H1),[])
```

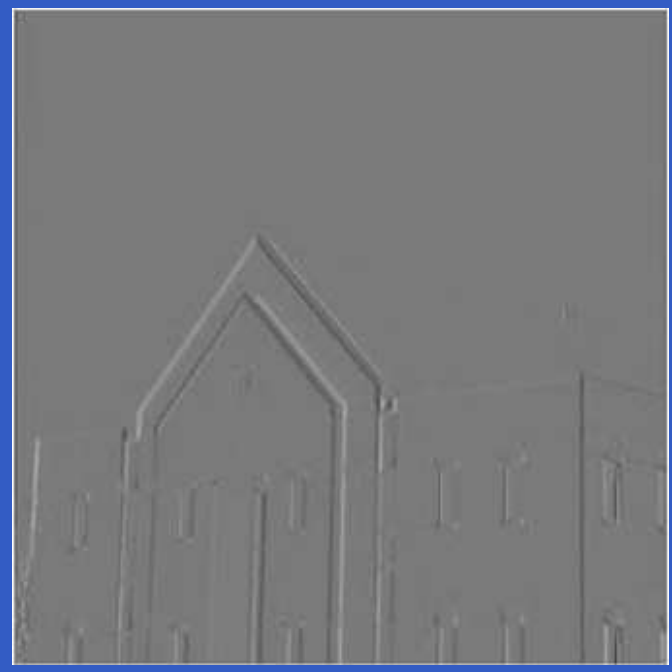




Obtaining Frequency Domain Filters from Spatial Filters

```
>> gs=imfilter(double(f),h);
>> gf=dftfilt(f,H1);
>> imshow(gs,[])
>> figure,imshow(gf,[])
```



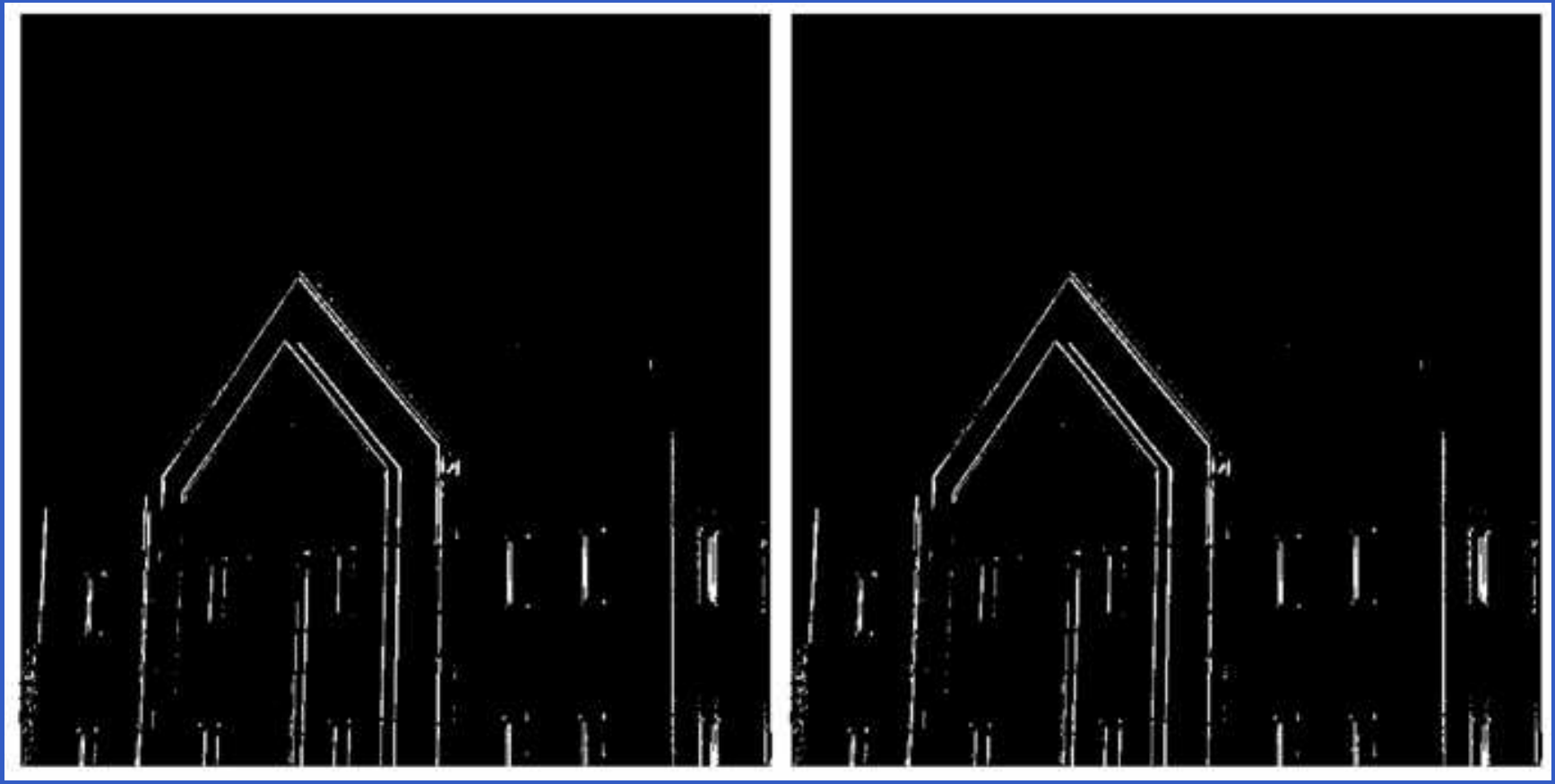


Obtaining Frequency Domain Filters from Spatial Filters

```
>> figure, imshow(abs(gs),[])
>> figure, imshow(abs(gf),[])
```

Obtaining Frequency Domain Filters from Spatial Filters

```
>> figure, imshow(abs(gs)>0.2*abs(max(gs(:))))
>> figure, imshow(abs(gf)>0.2*abs(max(gf(:))))
```



Obtaining Frequency Domain Filters from Spatial Filters

```
>> d=abs(gs-gf);
>> max(d(:))

ans =

   5.5156e-013

>> min(d(:))

ans =

        0
```

Chapter 5

Edge Detection

Edge detection

- Edges can be found in an image, where sudden intesity changing is sensed.
- The changing can be determined from the derivatives of the intensity function.
- In an image we should use gradient instead of derivates.
  - Gradient vector: $\begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$
  - Length of the gradient vector: $\sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$

edge function

MATLAB function:

```
[g,t]=edge(f,'method',parameters)
```

Possible `method` values:

- `'prewitt'`
- `'sobel'`
- `'roberts'`
- `'log'`
- `'zerocross'`
- `'canny'`

Prewitt detector

Masks: $\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$ $\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$

MATLAB function:

```
[g,t]=edge(f,'prewitt',T,dir)
```

Sobel detector

Masks: $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$ $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$

MATLAB function:

```
[g,t]=edge(f,'sobel',T,dir)
```

Roberts detector

Masks: $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$ $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$

MATLAB function:

```
[g,t]=edge(f,'roberts',T,dir)
```

Laplacion of Gaussian detector

Mask equation: $-\left[\frac{r^2-\sigma^2}{\sigma^4}\right] e^{-\frac{r^2}{2\sigma^2}}$

This mask smoothes the image, then makes the second derivative. In this filtered image the edge detector searches 0-crossings.

MATLAB function:

```
[g,t]=edge(f,'log',T,sigma)
```

Zero-crossing detector

It is very similar with the previous one, but the filter mask (H) can be determined by the user.

MATLAB function:

```
[g,t]=edge(f,'zerocross',T,H)
```

Canny detector

1. The image is smoothed using a Gaussian filter with a specified standard deviation, σ, to reduce noise.
2. The local gradient and edge direction are computed at each point.
3. The computed edges are thined by nonmaximal suppression.
4. The ridge pixels are then thresholded using two thresholds, $T1$ and $T2$, with $T1 < T2$. Ridge pixels with values greater than $T2$ are said to be "strong" edge pixels. Ridge pixels with values between $T1$ and $T2$ are said to be "weak" edge pixels.

Canny detector

5. Finally, the algorithm performs edge linking by incorporation the weak pixels that are 8-connected to the strong pixels.

MATLAB function:

```
[g,t]=edge(f,'canny',T,sigma)
```

Chapter 6

Morphological Image Processing

Dilation

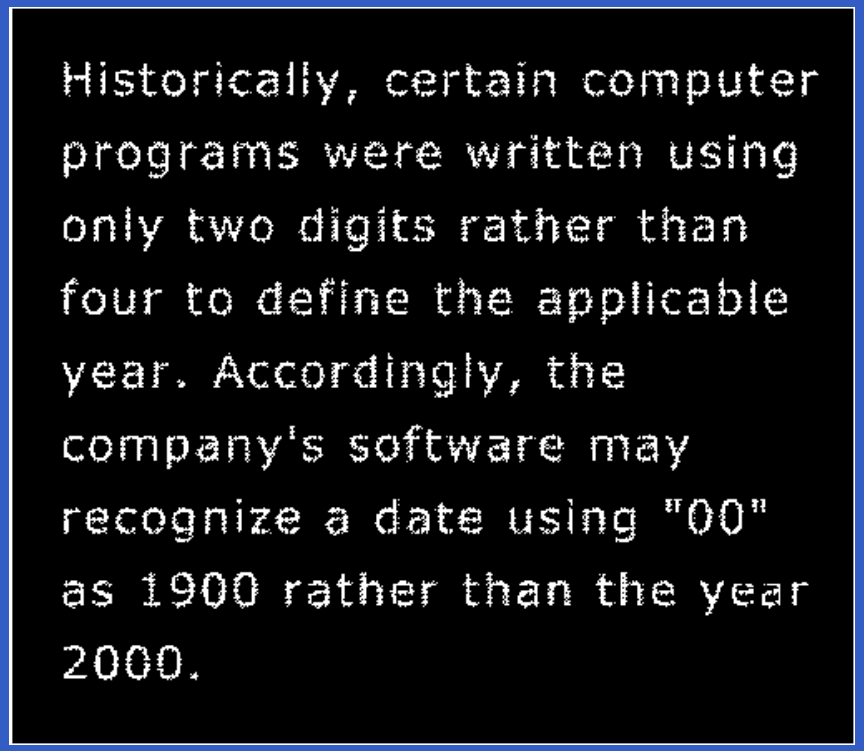
IPT function `imdilate` performs dilation. Its basic calling syntax is

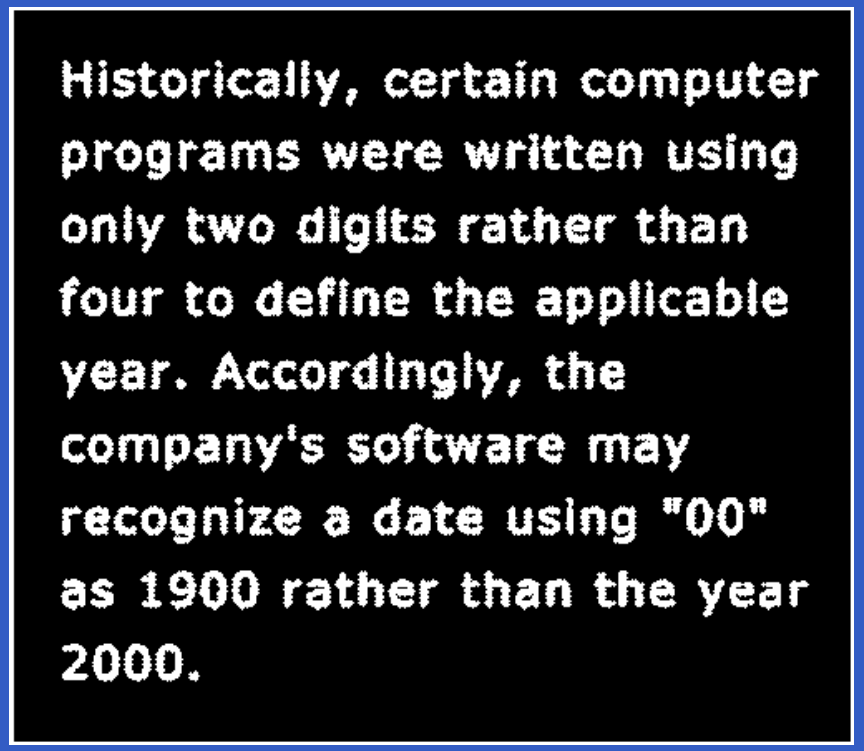
```
A2=imdilate(A,B)
```

where `A` and `A2` are binary images, and `B` is a matrix of 0s and 1s that specifies the structuring element.

Dilation

```
>> A=imread('broken-text.tif');
>> B=[0 1 0;1 1 1;0 1 0];
>> A2=imdilate(A,B);
```





Structuring Element

IPT function `strel` constructs structuring elements with a variety of shapes and sizes. Its basic syntax is

```
se=strel(shape,parameters)
```

where `shape` is a string specifying the desired shape, and `parameters` is a list of parameters that specify information about the shape, such as its size.

Structuring Element

Syntax Forms	Description
`strel('diamond',R)`	Creates a flat, diamond-shaped structuring element, where `R` specifies the distance from the structuring element origin to the extreme points of the diamond.
`strel('disk',R)`	Creates a flat, disk-shaped structuring element with radius `R`.
`strel('line',LEN,DEG)`	Creates a flat, linear structuring element, where `LEN` specifies the length, and `DEG` specifies the angle (in degrees) of the line, as measured in a counterclockwise direction from the horizontal axes.
`strel('octagon',R)`	Creates a flat, octagonal structuring element, where `R` specifies the distance from the structuring element origin to the sides of the octagon, as measured along the horizontal and vertical axes. `R` must be a nonnegative multiple of 3.

Structuring Element

Syntax Forms	Description
`strel('pair',OFFSET)`	Creates a flat structuring element containing two members. One member is located at the origin. The second member's location is specified by the vector `OFFSET`, which must be a two-element vector of integers.
`strel('periodicline',P,V)`	Creates a flat structuring element containing `2*P+1` members. `V` is a two-element vector containing integer-valued row and column offsets. One structuring element member is located at the origin. The other members are located at `1*V`, `-1*V`, `2*V`, `-2*V`, `...`, `P*V`, and `-P*V`.

Structuring Element

Syntax Forms	Description
`strel('rectangle',MN)`	Creates a flat, rectangle-shaped structuring element, where `MN` specifies the size. `MN` must be a two-element vector of nonnegative integers. The first element of `MN` is the number of rows in the structuring element; the second element is the number of columns.
`strel('square',W)`	Creates a square structuring element whose width is `W` pixels. `W` must be a nonnegative integer scalar.
`strel(NHOOD)`	Creates a structuring element of arbitrary shape. `NHOOD` is a matrix of 0s and 1s that specifies the shape.

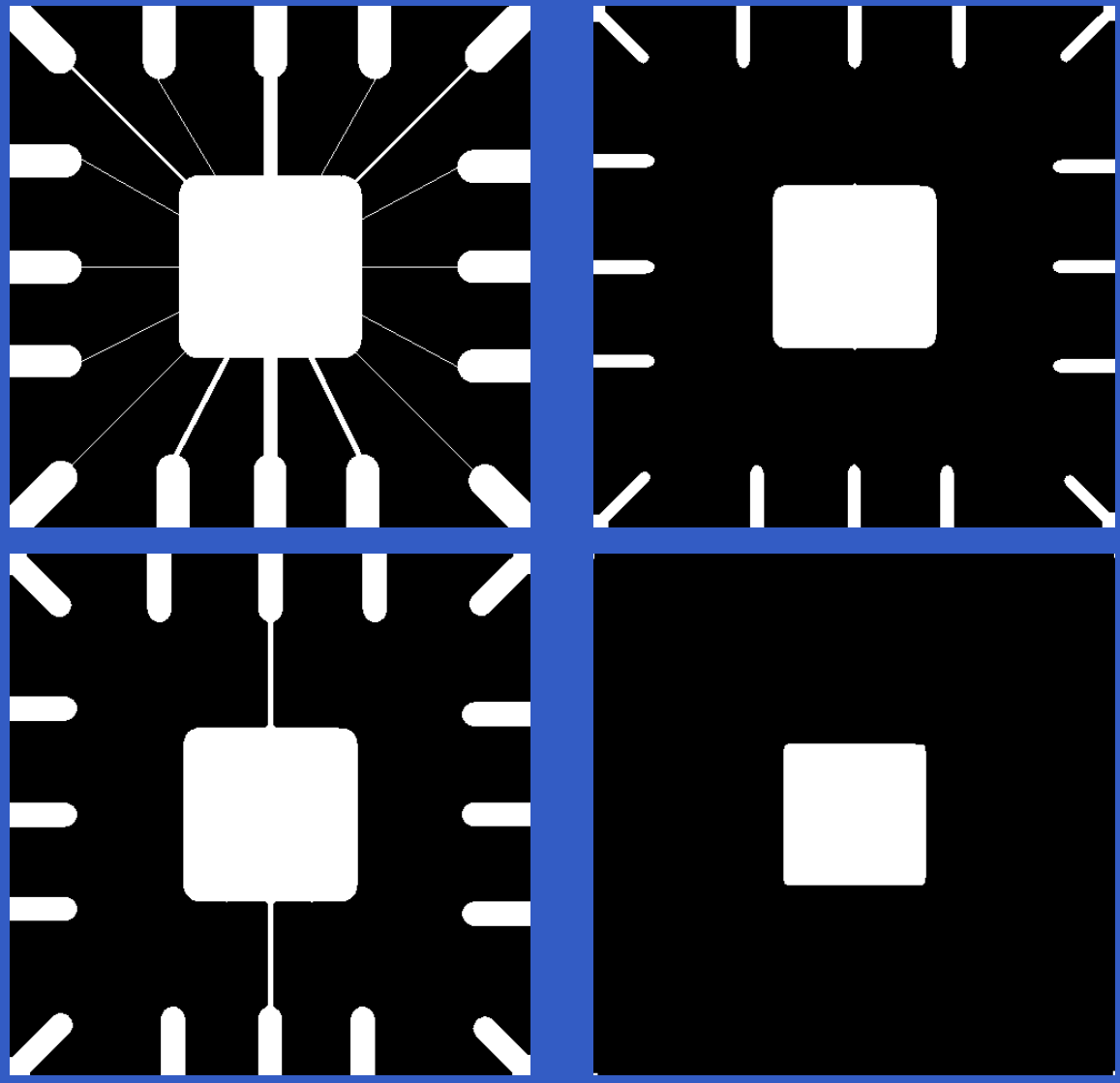
Dilation

```
>> originalI=imread('cameraman.tif');
>> se=strel('disk',2);
>> dilatedI=imdilate(originalI,se);
```

Erosion

```
>> A=imread('wirebond-mask.tif');
>> se=strel('disk',10);
>> A2=imerode(A,se);
>> se=strel('disk',5);
>> A3=imerode(A,se);
>> A4=imerode(A,strel('disk',20));
>> subplot(2,2,1), imshow(A),...
subplot(2,2,2), imshow(A2),...
subplot(2,2,3), imshow(A3),...
subplot(2,2,4), imshow(A4)
```

Erosion



Labeling Connected Components

IPT function `bwlabel` computes all the connected components in a binary image. The calling syntax is

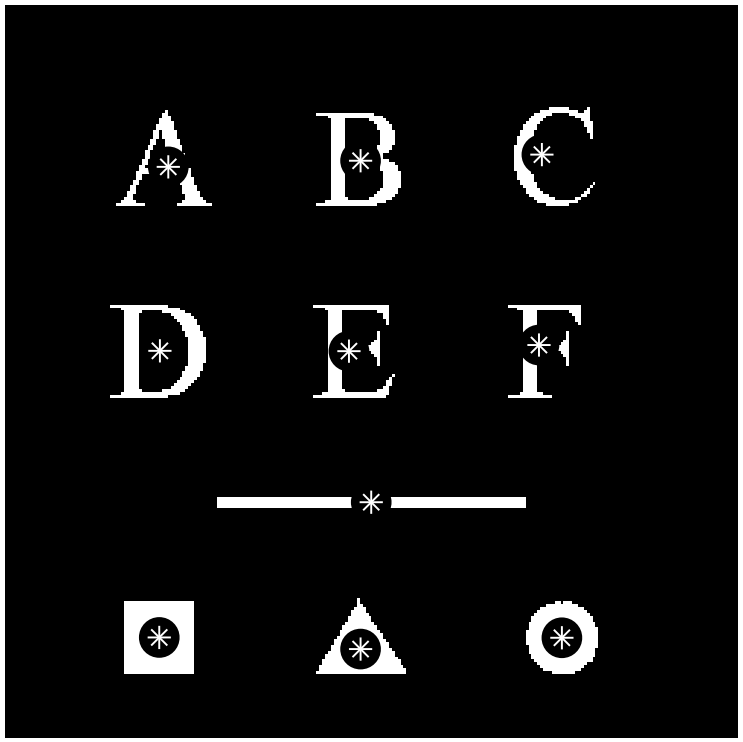
```
[L,num]=bwlabel(f,conn)
```

where `f` is an input binary image and `conn` specifies the desired connectivity (either 4 or 8). Output `L` is called a *label matrix*, and `num` (optional) gives the total number of connected components found. If parameter `conn` is omitted, its value defaults to 8.

Labeling Connected Components

```
>> f=imread('ten-objects.tif');
>> [L,n]=bwlabel(f);
>> [r,c]=find(L==3);
>> rbar=mean(r);
>> cbar=mean(c);
>> imshow(f)
>> hold on
>> for k=1:n
     [r,c]=find(L==k);
     rbar=mean(r);
     cbar=mean(c);
     plot(cbar,rbar,'Marker','o', 'MarkerEdgeColor','k',...
     'MarkerFaceColor','k', 'MarkerSize',10)
     plot(cbar,rbar,'Marker','*', 'MarkerEdgeColor','w')
   end
```

Labeling Connected Components



Chapter 7

Color Image Processing

Content

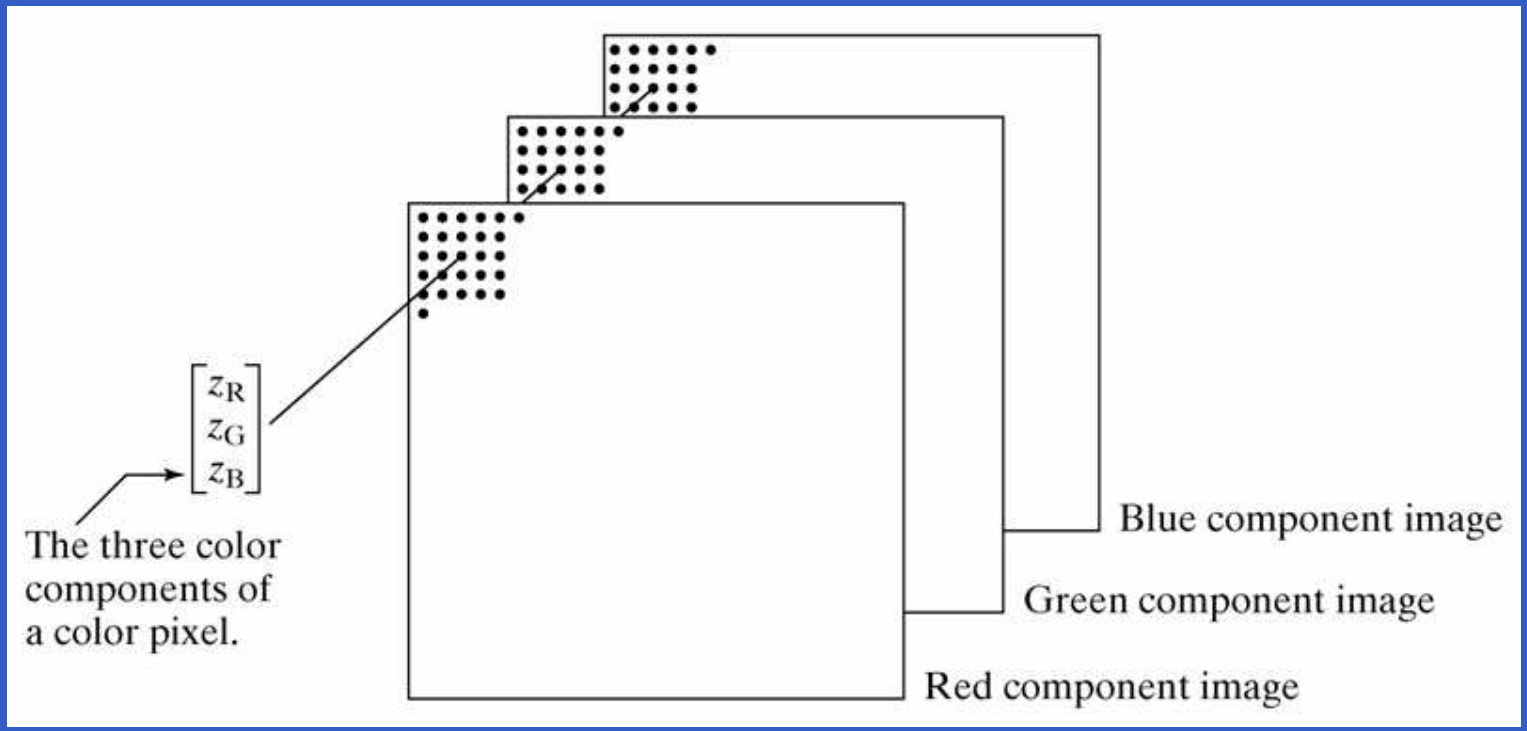
- Color Image Representation in MATLAB
- Converting to Other Color Spaces

Color Image Representation in MATLAB

- RGB Images
- Indexed Images
- IPT Functions for Manipulating RGB and Indexed Images

RGB Images

An RGB *color image* is an $M \times N \times 3$ array of *color pixels*, where each color pixel is a triplet corresponding to the red, green, and blue components of an RGB image at a specific spatial location.



RGB Images

The data class of the component images determines their range of values. If an RGB images is of class `double`, the range of values is $[0, 1]$. Similarly, the range of values is $[0, 255]$ of $[0, 65535]$ for RGB images of class `uint8` or `uint16`, respectively. The number of bits used to represent the pixel values of the component images determines the *bit depth* of an RGB image.

RGB Images

Let `fR`, `fG`, and `fB` represent three RGB component images. An RGB image is formed from these images by using the `cat` (concatenate) operator to stack the images:

```
rgb_image=cat(3,fR,fG,fB)
```

The order in which images are placed in the operand matters. In general, `cat(dim,A1,A2,...)` concatenates the arrays along the dimension specified by `dim`. For example, if `dim=1`, the arrays are arranged vertically, if `dim=2`, they are arranged horizontally, and, if `dim=3`, they are stacked in the third dimension.

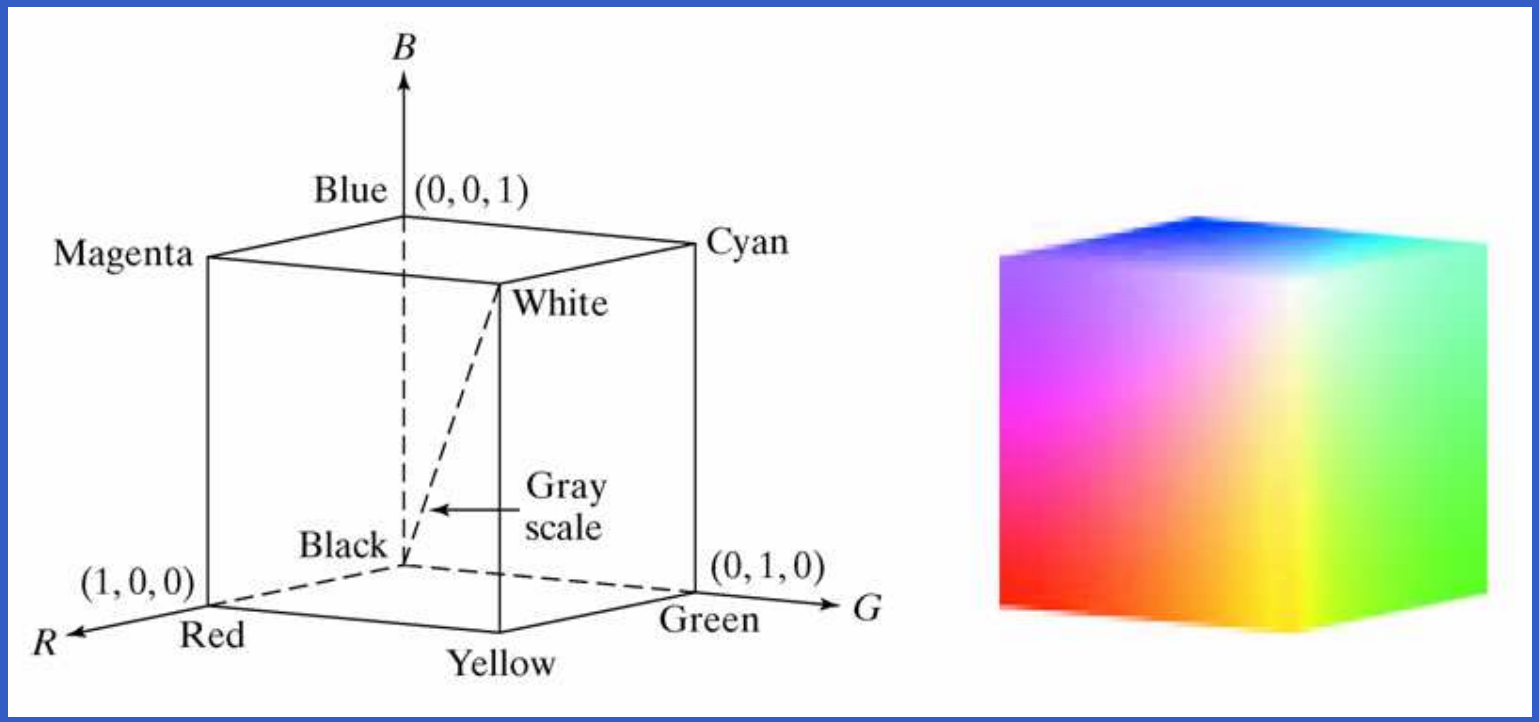
RGB Images

If all component images are identical, the result is a gray-scale image. Let `rgb_image` denote an RGB image. The following commands extract the three component images:

```
>> fR=rgb_image(:,:,1);
>> fG=rgb_image(:,:,2);
>> fB=rgb_image(:,:,3);
```

RGB Images

The RGB *color space* usually is shown graphically as an RGB color cube, as depicted in the figure. The vertices of the cube are the *primary* (red, green, and blue) and *secondary* (cyan, magenta, and yellow) colors of light.



RGB Images

```
function rgbcube(vx,vy,vz)
%RGBCUBE Displays an RGB cube on the MATLAB desktop.
%   RGBCUBE(VX,VY,VZ) displays an RGB color cube, viewed from point
%   (VX,VY,VZ). With no input arguments, RGBCUBE uses (10,10,4)
%   as the default viewing coordinates. To view individual color
%   planes, use the following viewing coordinates, where the first
%   color in the sequence is the closest to the viewing axis, and the
%   other colors are as seen from that axis, proceeding to the right
%   (ob above), and then moving clockwise.
%
%     ------------------------------------------
%          COLOR PLANE                ( VX, VY, VZ)
%     ------------------------------------------
%      Blue-Magenta-White-Cyan        (  0,  0, 10)
%      Red-Yellow-White-Magenta       ( 10,  0,  0)
%      Green-Cyan-White-Yellow        (  0, 10,  0)
%      Black-Red-Magenta-Blue         (  0,-10,  0)
%      Black-Blue-Cyan-Green          (-10,  0,  0)
%      Black-Red-Yellow-Green         (  0,  0,-10)
```

RGB Images

```
%Set up paramteres for function patch.
vertices_matrix=[0 0 0;0 0 1;0 1 0;0 1 1;1 0 0;1 0 1;1 1 0;1 1 1];
faces_matrix=[1 5 6 2;1 3 7 5;1 2 4 3;2 4 8 6;3 7 8 4;5 6 8 7];
colors=vertices_matrix;
%The order of the cube vertices was selected to be the same as
%the order of the (R,G,B) colors (e.g., (0,0,0) corresponds to
%black, (1,1,1) corresponds to white, and so on.)

%Generate RGB cube using function patch.
patch('Vertices',vertices_matrix,'Faces',faces_matrix,...
    'FaceVertexCData',colors,'FaceColor','interp',...
    'EdgeAlpha',0)
```

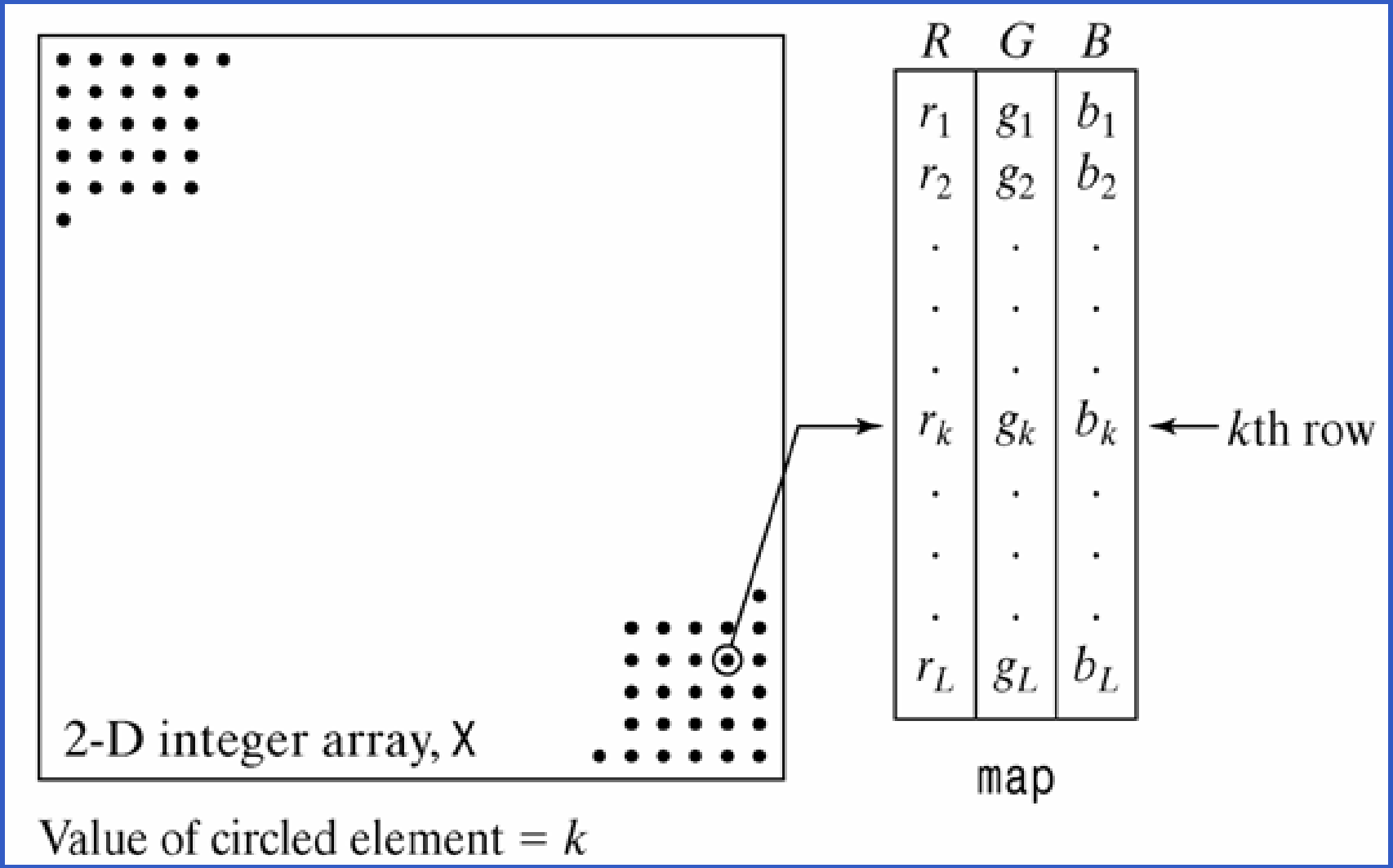
RGB Images

```
%Set up viewing point.
if nargin==0
    vx=10; vy=10; vz=4;
elseif nargin ~=3
    error('Wrong number of inputs.')
end
axis off
view([vx, vy, vz])
axis square
```

Indexed Images

An *indexed image* has two components: a *data matrix* of integers, `X`, and a `colormap matrix`, `map`. Matrix `map` is an $m \times 3$ array of class `double` containing floating-point values in the range $[0, 1]$. The length, m, of the `map` is equal to the number of colors it defines. Each row of `map` specifies the red, green, and blue components of a single color. An indexed images uses "direct mapping" of pixel intensity values to colormap values. The color of each pixel is determined by using the corresponding value of integer matrix `X` as a pointer into `map`. If `X` is of class `double`, then all of its components with value 2 point to the second row, and so on. If `X` is of class `uint8` or `uint16`, then all components with value 0 point to the first row in `map`, all components with value 1 point to the second row, and so on.

Indexed Images



Indexed Images

To display an indexed image we write

```
>> imshow(X,map)
```

or, alternatively,

```
>> image(X)
>> colormap(map)
```

A colormap is stored with an indexed image and is automatically loaded with the image when function `imread` is used to load the image.

```
>>[X,map]=imread(...)
```

Indexed Image

Sometimes it is necessary to approximate an indexed image by one with fewer colors. For this we use function `imapprox`, whose syntax is

```
[Y,newmap]=imapprox(X,map,n)
```

This function returns an array `Y` with colormap `newmap`, which has at most `n` colors. The input array `X` can be of class `uint8`, `uint16`, or `double`. The output `Y` is of class `uint8` if `n` is less than or equal to 256. If `n` is greater than 256, `Y` is of class `double`.

Indexed Images

MATLAB provides several predefined color maps, accessed using the command

```
>> colormap(map_name)
```

which sets the colormap to the matrix `map_name`; an example is

```
>> colormap(copper)
```

where `copper` is one of the predefined MATLAB colormaps. If the last image displayed was an indexed image, this command changes its colormap to `copper`. Alternatively, the image can be displayed directly with the desired colormap:

```
>> imshow(X,copper)
```

Indexed Images

Name	Description
`autumn`	Varies smoothly from red, through orange, to yellow.
`bone`	A gray-scale colormap with a higher value for the blue component. This colormap is useful for adding an "electronic" look to gray-scale images.
`colorcube`	Contains as many regularly spaced colors in RGB color space as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
`cool`	Consists of colors that are shades of cyan and magenta. It varies smoothly from cyan to magenta.
`copper`	Varies smoothly from black to bright copper.
`flag`	Consists of the colors red, white, blue, and black. This colormap completely changes color with each index increment.
`gray`	Returns a linear gray-scale colormap.
`hot`	Varies smoothly from black, through shades of red, orange, and yellow, to white.

Indexed Images

Name	Description
`hsv`	Varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. The colormap is particularly appropriate for displaying periodic functions.
`jet`	Ranges from blue to red, and passes through the colors cyan, yellow, and orange.
`lines`	Produces a colormap of colors specified by the `ColorOrder` property and a shade of gray. Consult online help regarding function `ColorOrder`.
`pink`	Contains pastel shades of pink. The pink colormap provides sepia tone colorization of grayscale photographs.
`prism`	Repeats the six colors red, orange, yellow, green, blue, and violet.
`spring`	Consists of colors that are shades of magenta and yellow.
`summer`	Consists of colors that are shades of green and yellow.
`white`	This is an all white monochrome colormap.
`winter`	Consists of colors that are shades of blue and green.

Manipulating RGB and Indexed Images

Function	Purpose
`dither`	Creates an indexed image from an RGB image by dithering.
`grayslice`	Creates an indexed image from a gray-scale intensity image by multilevel thresholding.
`gray2ind`	Creates an indexed image from a gray-scale intensity image.
`ind2gray`	Creates a gray-scale intensity image from an indexed image.
`rgb2ind`	Creates an indexed image from an RGB image.
`ind2rgb`	Creates an RGB image from an indexed image.
`rgb2gray`	Creates a gray-scale image from an RGB image.

Manipulating RGB and Indexed Images

```
>> f=imread('iris.tif');
```

Manipulating RGB and Indexed Images

```
>> [X1,map1]=rgb2ind(f,8,'nodither');
>> imshow(X1,map1)
```

Manipulating RGB and Indexed Images

```
>> [X2,map2]=rgb2ind(f,8,'dither');
>> imshow(X2,map2)
```

Manipulating RGB and Indexed Images

```
>> g=rgb2gray(f);
>> g1=dither(g);
>> figure, imshow(g); figure, imshow(g1)
```

Converting to Other Color Spaces

- NTSC Color Space
- The YCbCr Color Space
- The HSV Color Space
- The CMY and CMYK Color Spaces
- The HSI Color Space

NTSC Color Space

The NTSC Color System is used in television in the United States. One of the main advantages of this format is that gray-scale information is separate from color data. In the NTSC format, image data consists of three components:

- luminance (Y)
- hue (I)
- saturation (Q)

The luminance component represents gray-scale information, and the other two components carry the color information of a TV signal.

NTSC Color Space

The YIQ components are obtained from the RGB components of an image using the transfromation

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Note that the elements of the first row sum to 1 and the elements of the next two rows sum to 0. This is as expected because for a gray-scale image all the RGB components are equal, so the I and Q components should be 0 for such an image.

NTSC Color Space

Function `rgb2ntsc` performs the transformation:

```
yiq_image=rgb2ntsc(rgb_image)
```

where the input RGB image can be of class `uint8`, `uint16`, or `double`. The output image is an $M \times N \times 3$ array of class `double`. Component image `yiq_image(:,:,1)` is the luminance, `yiq_image(:,:,2)` is the hue, and `yiq_image(:,:,3)` is the saturation image.

NTSC Color Space

Similarly, the RGB components are obtained from the YIQ components using the transformation:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.106 & 1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

IPT function `ntsc2rgb` implements this equation:

```
rgb_image=ntsc2rgb(yiq_image)
```

Both the input and output images are of class `double`.

The YCbCr Color Space

The YCbCr color space is used widely in digital video. In this format, luminance information is represented by a single component, Y, and color information is stored as two color-difference components, Cb and Cr. Component Cb is the difference between the blue component and a reference value, and component Cr is the difference between the red component and a reference value.

The YCbCr Color Space

The transformation used by IPT to convert from RGB to YCbCr is

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112.000 \\ 112.000 & -93.786 & -18.214 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The YCbCr Color Space

The conversion function is

```
ycbcr_image=rgb2ycbcr(rgb_image)
```

The input RGB image can be of class `uint8`, `uint16`, or `double`. The output image is of the same class as the input. A similar transformation converts from YCbCr back to RGB:

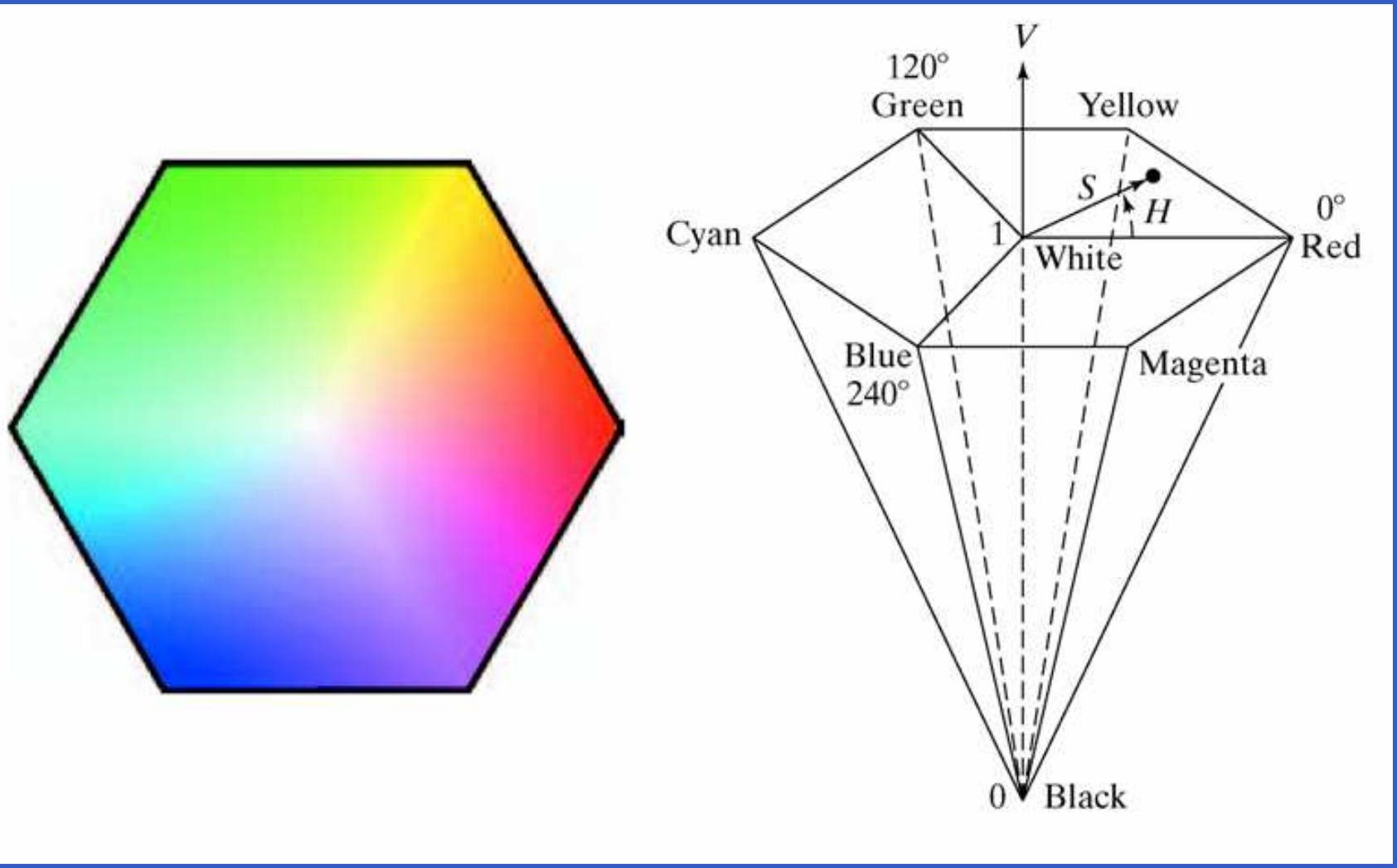
```
rgb_image=ycbcr2rgb(ycbcr_image)
```

The input YCbCr image can be of class `uint8`, `uint16`, or `double`. The output image is of the same class as the input.

The HSV Color Space

HSV (hue, saturation, value) is one of several color systems used by people to select colors from a color wheel or palette. This color system is considerably closer than the RGB system to the way in which humans experience and describe color sensations. In artist's terminology, hue, saturation, and value refer approximately to tint, shade, and tone.

The HSV Color Space



The HSV Color Space

The MATLAB function for converting from RGB to HSV is `rgb2hsv`, whose syntax is

```
hsv_image=rgb2hsv(rgb_image)
```

The input RGB image can be of class `uint8`, `uint16`, or `double`; the output image is of class `double`. The function for converting from HSV back to RGB is `hsv2rgb`:

```
rgb_image=hsv2rgb(hsv_image)
```

The input image must be of class `double`. The output also is of class `double`.

The CMY Color Space

The conversion is performed using the simple equation

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

where the assumption is that all color values have been normaized to the range $[0, 1]$.

The CMY Color Space

Function `imcomplement` can be used to convert from RGB to CMY:

```
cmy_image=imcomplement(rgb_image)
```

We use this function also to convert a CMY image to RGB:

```
rgb_image=imcomplement(cmy_image)
```

The HSI Color Space

When humans view a color object, we tend to describe it by its hue, saturation, and brightness. *Hue* is an attribute that describes a pure color, whereas *saturation* gives a mesaure of the degree to which a pure color is diluted by white light. *Brightness* is a subjective descriptor that is practically impossible to measure. It embodies the achromatic description of *intensity* and is a key factor in describing color sensation. We do know that intensity (gray level) is a most useful descriptor of monochromatic images. This quantity definitely is measurable and easily interpretable.

The color space we are about to present, called the *HSI* (hue, saturation, intensity) *color space*.

Converting Colors from RGB to HSI

Given an image in RGB color format, the H component of each RGB pixel is obtained usint the equation

$$H = \begin{cases} \theta & \text{if} B \leq G \\ 360^\circ - \theta & \text{if} B > G \end{cases}$$

with

$$\theta = \cos^{-1}\left\{\frac{\frac{1}{2}\left[(R-G)+(R-B)\right]}{\sqrt{(R-G)^2+(R-B)(G-B)}}\right\}$$

Converting Colors from RGB to HSI

The saturation component is given by

$$S = 1 - \frac{3}{(R+G+B)}\left[\min(R,G,B)\right]$$

Finally, the intensity component is given by

$$I = \frac{1}{3}(R+G+B)$$

Converting Colors from RGB to HSI

It is assumed that the RGB values have been normalized to the range $[0, 1]$, and that angle θ is measured with respect to the red axis of the HSI space. Hue can be normalized to the range $[0, 1]$ by dividing by 360° all values resulting from the equation for H. The other two HSI components already are in this range if the given RGB values are in the interval $[0, 1]$.

Converting Colors from RGB to HSI

```
function hsi=rgb2hsi(rgb)
%RGB2HSI Converts an RGB image to HSI.
%   HSI=RGB2HSI(RGB) converts an RGB image to HSI. The input image
%   is assumed to be of size M-by-N-by-3, where the third dimension
%   accounts for three image planes: red, green, and blue, in that
%   order. If all RGB component images are equal, the HSI conversion
%   is undefined. The input image can be of class double (with values
%   in the range [0,1]), uint8, or uint16.
%
%   The output image, HSI, is of class double, where:
%      hsi(:,:,1)=hue image normalized to the range [0,1] by
%                 dividing all angle values by 2*pi.
%      hsi(:,:,2)=saturation image, in the range [0,1].
%      hsi(:,:,3)=intensity image, in the range [0,1].
```

Converting Colors from RGB to HSI

```
% Extract the individual component images.
rgb=im2double(rgb);
r=rgb(:,:,1);
g=rgb(:,:,2);
b=rgb(:,:,3);

% Implement the conversion equations.
num=0.5*((r-g)+(r-b));
den=sqrt((r-g).^2+(r-b).*(g-b));
theta=acos(num./(den+eps));

H=theta;
H(b>g)=2*pi-H(b>g);
H=H/(2*pi);
```

Converting Colors from RGB to HSI

```
num=min(min(r,g),b);
den=r+g+b;
den(den==0)=eps;
S=1-3.*num./den;

H(S==0)=0;

I=(r+g+b)/3;

% Combine all three results into an hsi image.
hsi=cat(3,H,S,I);
```

Converting Color from HSI to RGB

Given values of HSI in the interval $[0, 1]$, we now find the corresponding RGB values in the same range. The applicable equations depend on the values of H. There are three sectors of interest, corresponding to the $120°$ intervals in the separation of primaries. We begin by multiplying H by $360°$, which returns the hue to its original range of $[0°, 360°]$.

Converting Color from HSI to RGB

RG sector ($0° \leq H < 120°$): When H is in this sector, the RGB components are given by the equations

$$B = I\,(1 - S)$$

$$R = I\left[1 + \frac{S\cos H}{\cos(60° - H)}\right]$$

and

$$G = 3I - (R + B)$$

Converting Color from HSI to RGB

GB sector ($120° \leq H < 240°$): If the given value of H is in this sector, we first subtract $120°$ from it:

$$H = H - 120°$$

Then the RGB components are

$$G = I\,(1 - S)$$

$$B = I\left[1 + \frac{S\cos H}{\cos(60° - H)}\right]$$

$$R = 3I - (G + B)$$

Converting Color from HSI to RGB

BR sector ($240^\circ \leq H \leq 360^\circ$): Finally, if H is in this range, we subtract 240° from it:

$$H = H - 240^\circ$$

Then the RGB components are

$$R = I\,(1 - S)$$

$$G = I\left[1 + \frac{S\cos H}{\cos(60^\circ - H)}\right]$$

$$B = 3I - (R + G)$$

Converting Color from HSI to RGB

```
function rgb=hsi2rgb(hsi)
%HSI2RGB Converts an HSI image to RGB.
%   RGB=HSI2RGB(HSI) converts an HSI image to RGB, where HSI
%   is assumed to be of class double with:
%      hsi(:,:,1)=hue image, assumed to be in the range
%                 [0,1] by having been divided by 2*pi.
%      hsi(:,:,2)=saturation image, in the range [0,1].
%      hsi(:,:,3)=intensity image, in the range [0,1].
%
%   The components of the output image are:
%      rgb(:,:,1)=red.
%      rgb(:,:,2)=green.
%      rgb(:,:,3)=blue.
```

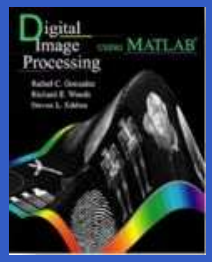
Converting Color from HSI to RGB

```
% Extract the individual HSI component images.
H=hsi(:,:,1)*2*pi;
S=hsi(:,:,2);
I=hsi(:,:,3);

% Implement the conversion equations.
R=zeros(size(hsi,1),size(hsi,2));
G=zeros(size(hsi,1),size(hsi,2));
B=zeros(size(hsi,1),size(hsi,2));

% RG sector (0<=H<2*pi/3).
idx=find((0<=H)&(H<2*pi/3));
B(idx)=I(idx).*(1-S(idx));
R(idx)=I(idx).*(1+S(idx).*cos(H(idx))./...
    cos(pi/3-H(idx)));
G(idx)=3*I(idx)-(R(idx)+B(idx));
```

Converting Color from HSI to RGB

```
% BG sector (2*pi/3<=H<4*pi/3).
idx=find((2*pi/3<=H)&(H<4*pi/3));
R(idx)=I(idx).*(1-S(idx));
G(idx)=I(idx).*(1+S(idx).*cos(H(idx)-2*pi/3)./...
    cos(pi-H(idx)));
B(idx)=3*I(idx)-(R(idx)+G(idx));

% BR sector.
idx=find((4*pi/3<=H)&(h<=2*pi));
G(idx)=I(idx).*(1-S(idx));
B(idx)=I(idx).*(1+S(idx).*cos(H(idx)-4*pi/3)./...
    cos(5*pi/3-H(idx)));
R(idx)=3*I(idx)-(G(idx)+B(idx));

% Combine all three results into an RGB image. Clip to [0,1] to
% compensate for floating-point arithmetic rounding effects.
rgb=cat(3,R,G,B);
rgb=max(min(rgb,1),0);
```

References



- R. C. Gonzalez, R. E. Woods, S. L. Eddins: Digital Image Processing Using MATLAB. Pearson Prentice Hall, 2004
- R. C. Gonzalez, R. E. Woods: Digital Image Processing. Prentice Hall, 2002
- `http://www.imageprocessingplace.com`
- MATLAB Documentation (`http://www.mathworks.com`)