Image Operations

KOM4520 Fundamentals of Robotic Vision

Most of the contents are adopted from
Robotics, Vision and Control: Fundamental Algorithms in MATLAB, Peter Corke, 2nd ed., Springer Verlag. 2017
R. C. Gonzalez, R. E. Woods, *Digital Image Processing*, 2nd edition, Prentice Hall, 2008
Digital Image Processing: An Algorithmic Introduction using Java, W Burger, Mark J. Burge, Springer Verlag, 2008

Digital Images

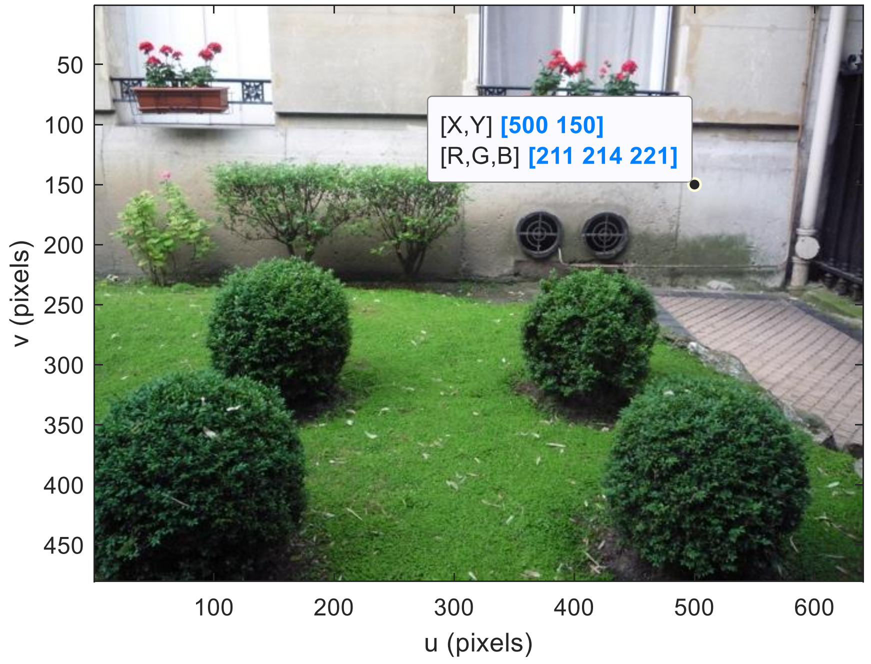
```
inim=iread('garden-r.jpg');
about(inim)
inim [uint8] : 480x640x3 (921.6 kB)

idisp(inim);
inim(150, 500, :)

%    211
%    214
%    221
```

class uint8 – the elements of the matrix are unsigned 8-bit integers in the interval [0,255]

portable network graphics (PNG) format – a lossless compression format

The image can be converted to double precision values in the range [0, 1] using the 'double' option
to iread or by applying the function idouble to the integer color image, just as for a greyscale image.

```
inim=iread('garden-r.jpg', 'double');
inim(150, 500, :)
%     0.8275
%     0.8392
 %    0.8667
```

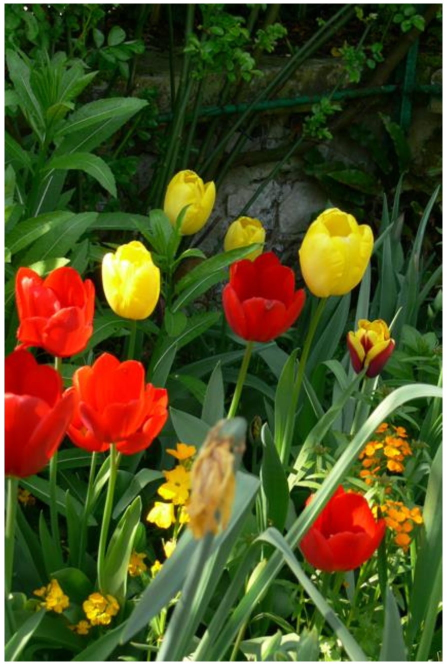


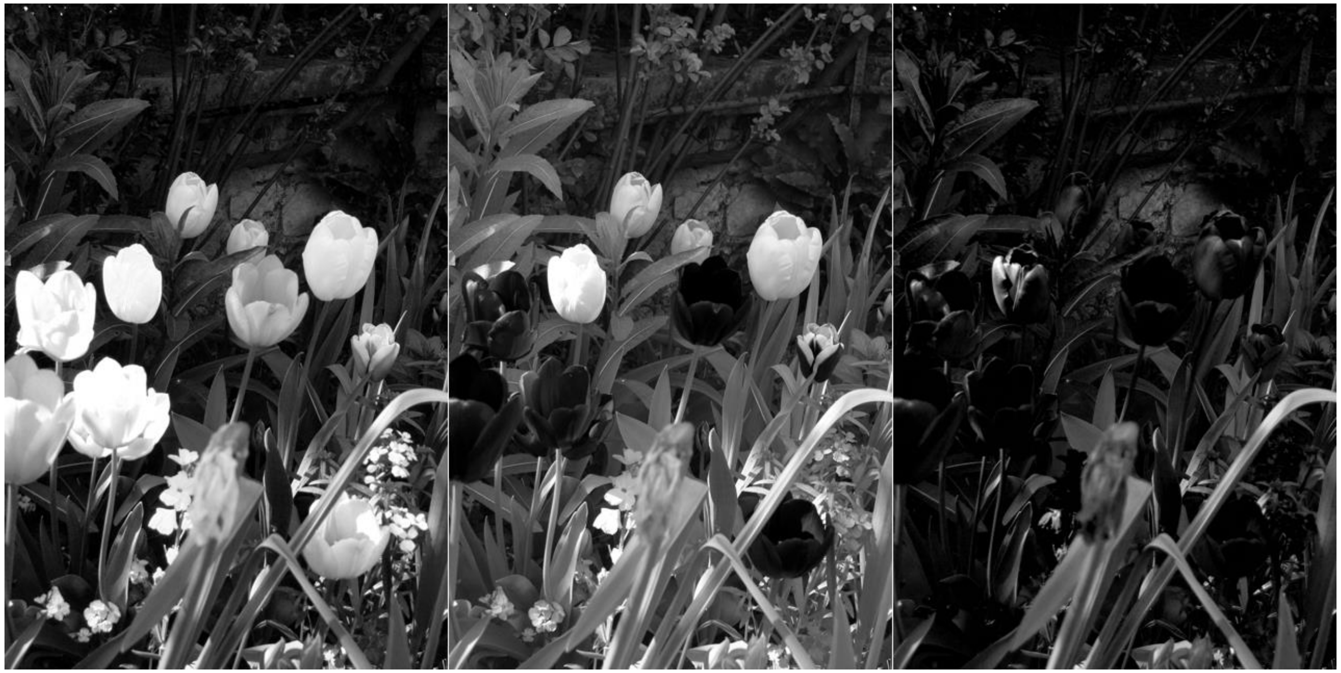
Digital Images

```
[inim,cs]=iread('garden-r.jpg');
cs
…
%            FileSize: 135003
%              Format: 'jpg'
%       FormatVersion: ''
%               Width: 640
%              Height: 480
%            BitDepth: 24
         …
%                Make: 'Panasonic'
%               Model: 'DMC-FS5'
%         Orientation: 1
%         XResolution: 180
%         YResolution: 180
%      ResolutionUnit: 'Inch'
%            Software: 'Ver.1.0  '
%            DateTime: '2009:06:15
05:33:39'
%       DigitalCamera: [1×1 struct]
```

```
cs.DigitalCamera


%
%                ExposureTime: 0.0667
%                     FNumber: 3.3000
%             ExposureProgram: 'Normal
program'
%             ISOSpeedRatings: 400
         …
%                 FocalLength: 5.2000
%             FlashpixVersion: [48 49 48
48]
%                  ColorSpace: 'sRGB'
%            CPixelXDimension: 640
%            CPixelYDimension: 480
%               SensingMethod: 'One-chip
color area sensor'
 …
```

Digital Images





Color image shown in RGB color planes

Digital Images

```
cam = Movie('traffic_sequence.mpg');
```

The size of each frame within the movie is

```
cam.size()
% 720 576
```

and the next frame is read from the movie file by

```
im = cam.grab();
about(im)
% im [uint8] : 576x720x3 (1244160 bytes)
```

which is a 720 × 576 color image. With these few primitives we can write a very simple movie player

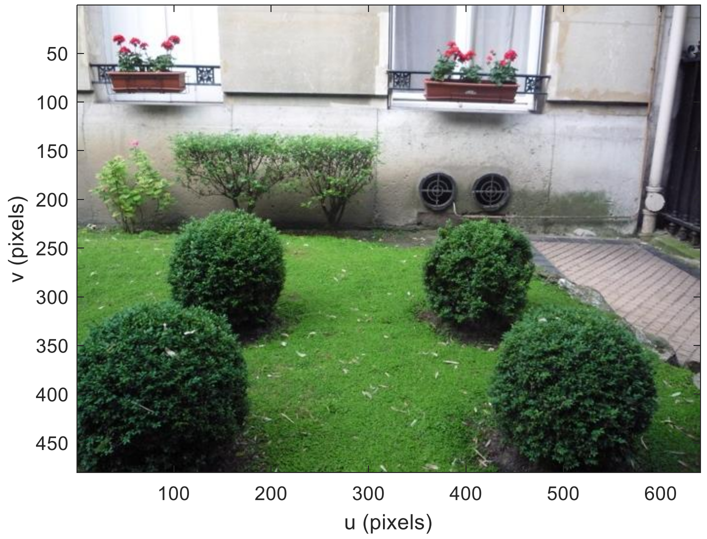
```
while 1
    im = cam.grab;
    if isempty(im)
        break;
    end
    image(im);
    drawnow
end
```

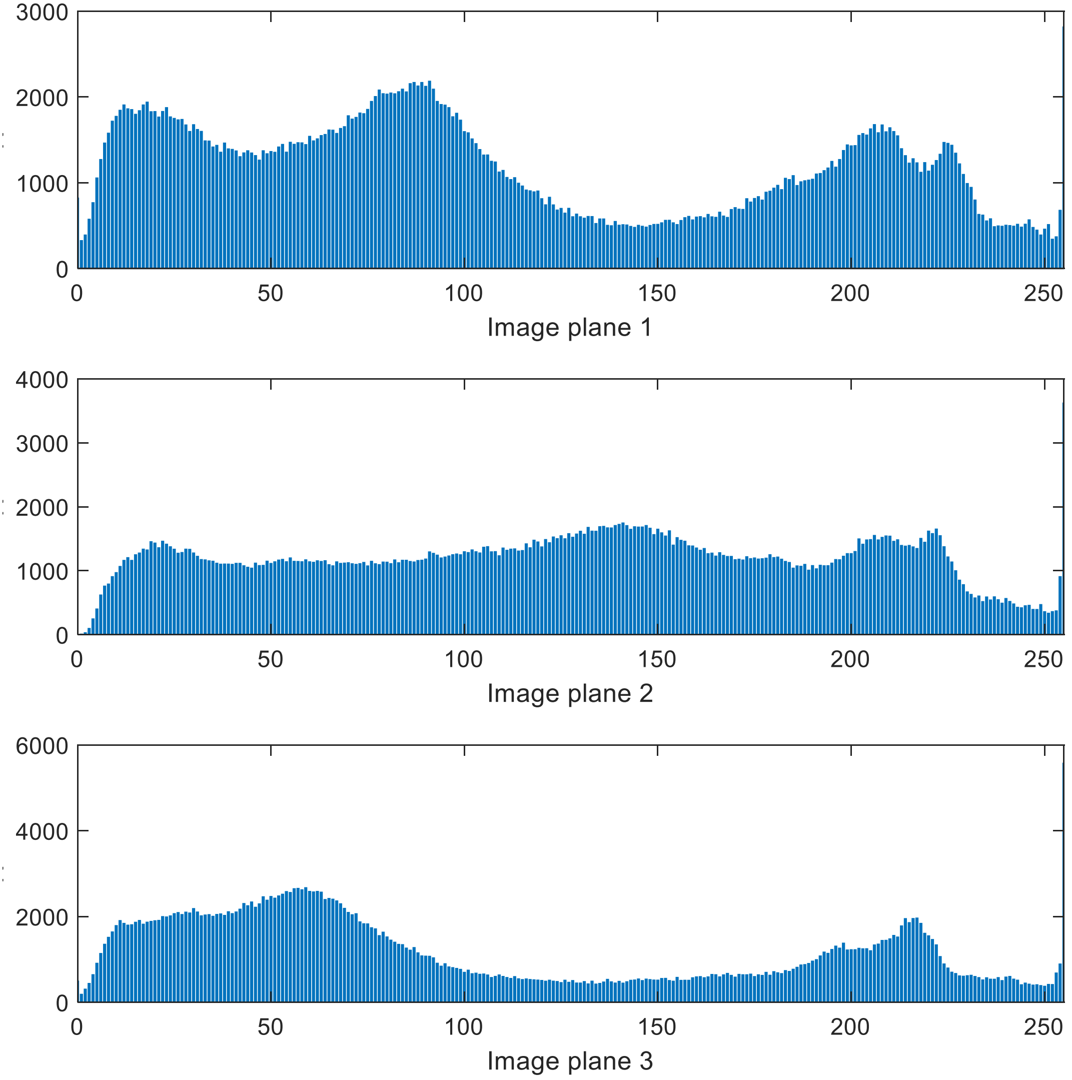
The methods nframes and framerate provide the total number of frames and the number of frames per second.

Digital Images

```
ihist(inim);
```

We obtain the distribution by computing the histogram of the image which indicates the number of times each pixel value occurs

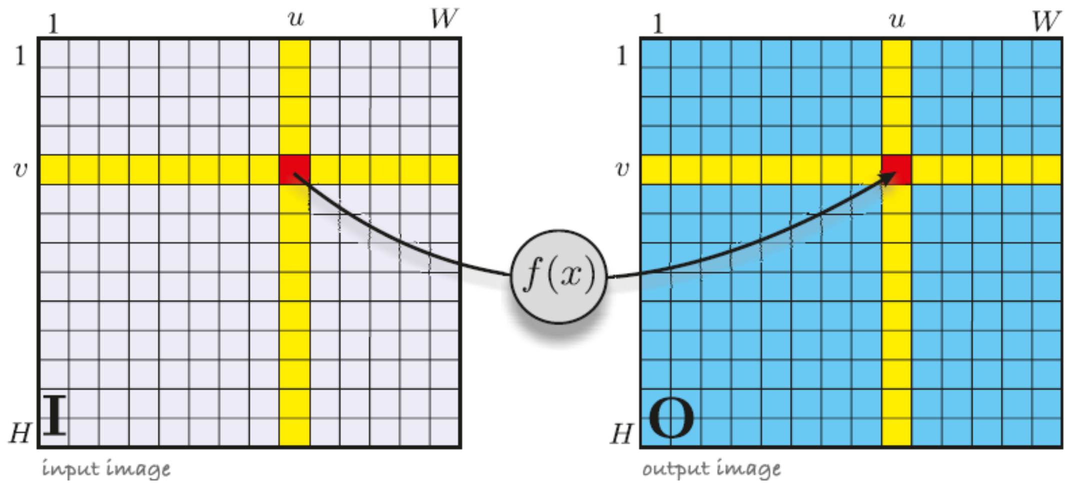




Monadic Operations

- Pixel based operations.
- The result is an image of the same size *W*× *H* as the input image, and each output pixel is a function of the corresponding input pixel.

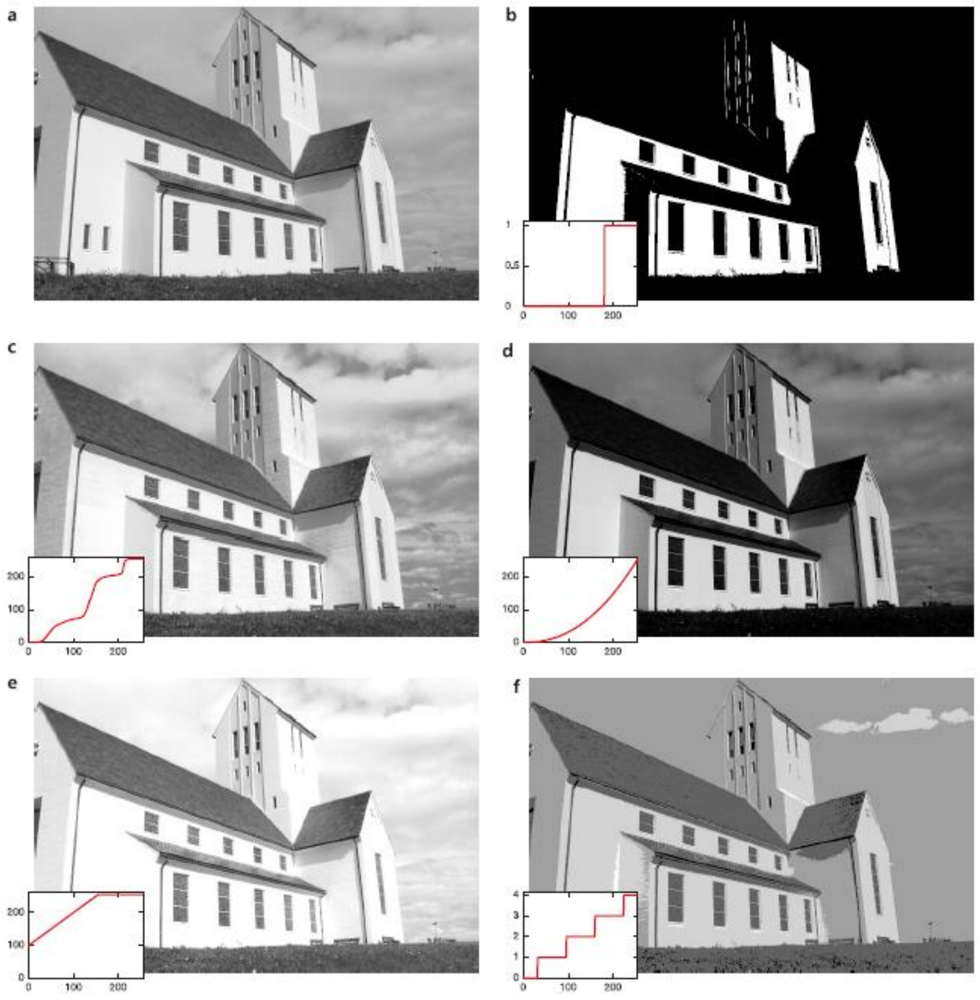
$$\boldsymbol{O}[u,v] = f(\boldsymbol{I}[u,v]), \ \forall (u,v) \in \boldsymbol{I}$$



Monadic Operations

Some monadic image operations:
a original,
b thresholding,
c histogram equalization,
d gamma correction,
e brightness increase,
f posterization.

Inset in each figure is a graph showing the mapping from image grey level on the horizontal axis to the output value on the vertical axis



Spatial Operations

- In spatial operations each pixel in the output image is a function of all pixels in a *region* surrounding the corresponding pixel in the input image.

$$\boldsymbol{O}[u,v] = f\left(\boldsymbol{I}[u+i, v+j]\right), \forall(u,v) \in \boldsymbol{I}, \forall(i,j) \in W$$

- where W is known as the window, typically a $w \times w$ square region with odd side length $w = 2h + 1$ where h is the half-width.
- Spatial operations are powerful because of the variety of possible functions $f(\cdot)$, linear or nonlinear, that can be applied.

- Linear spatial operators such as **smoothing** and **edge detection**, some nonlinear functions such as **rank filtering** and **template matching** will be given next.
- The further discussions cover a large and important class of nonlinear spatial operators known
- as **mathematical morphology**.

Spatial Operations

- A very important linear spatial operator is correlation

$$\boldsymbol{O}[u,v] = \sum_{(i,j)\in W} \boldsymbol{I}[u+i, v+j]K[i,j]\,, \forall (u,v) \in \boldsymbol{I}$$

- where $\boldsymbol{K} \in R^{w\times w}$ is the kernel and the elements are referred to as the filter coefficients.

- For every output pixel the corresponding window of pixels from the input image W is multiplied element-wise with the kernel $\boldsymbol{K}$. The center of the window and kernel is considered to be coordinate (0, 0) and $i,j \in [-h,h] \subset Z \times Z$.
- This can be considered as the weighted sum of pixels within the window where the weights are defined by the kernel $\boldsymbol{K}$.
- Correlation is often written in operator form as A closely related operation is convolution

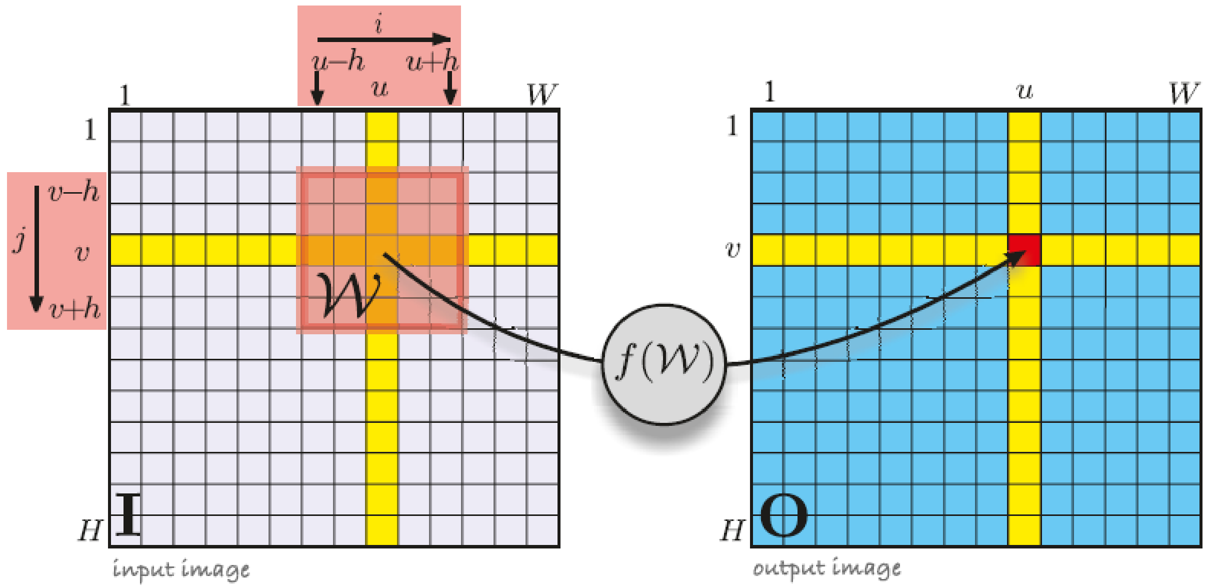
$$\boldsymbol{O}[u,v] = \sum_{(i,j)\in W} \boldsymbol{I}[u-i, v-j]K[i,j]\,, \forall (u,v) \in \boldsymbol{I}$$

- Convolution is often written in operator form as

$$\boldsymbol{O} = \boldsymbol{I} * \boldsymbol{K}$$

- Many kernels are symmetric in which case correlation and convolution yield the same result.
- However edge detection is often based on nonsymmetric kernels so we must take care to apply convolution.

Spatial Operations



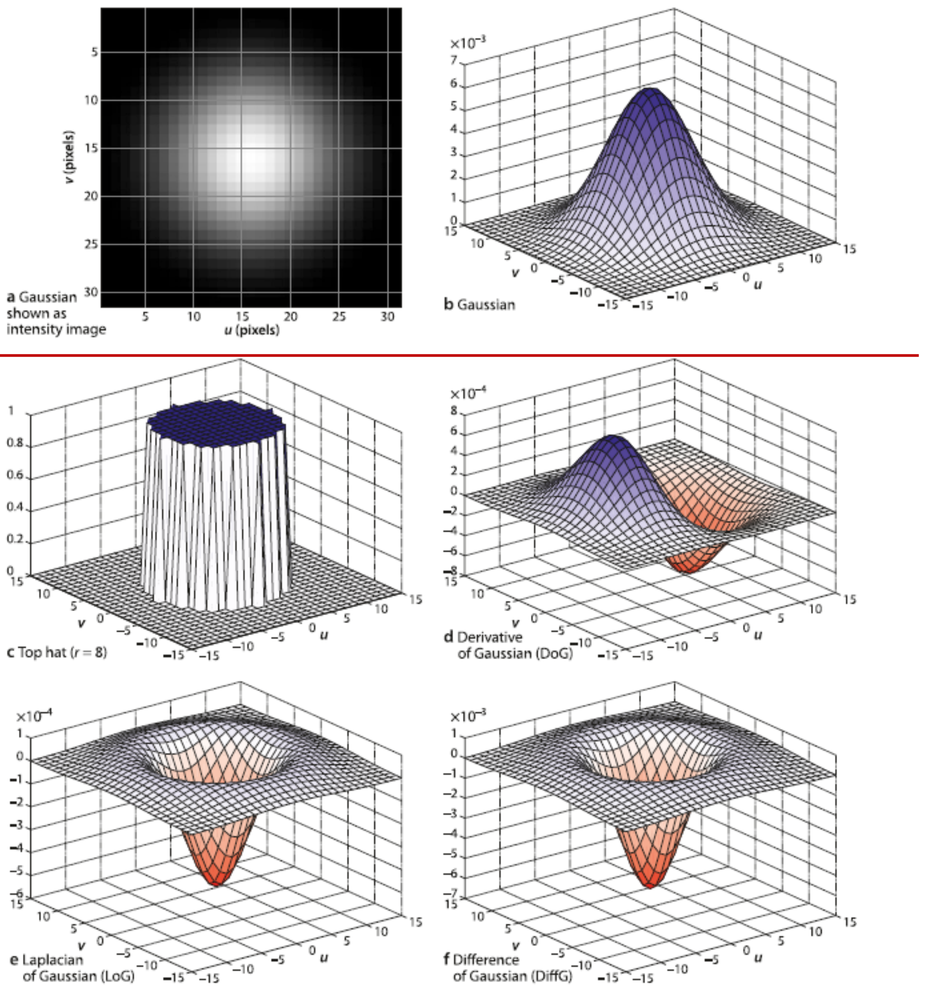
- Spatial image processing operations.
- Convolution is computationally expensive – an $N \times N$ input image with a $w \times w$ kernel requires w^2N^2 multiplications and additions.
- In the Toolbox convolution is performed using the function iconvolve

```
O = iconvolve(K, I);
```

- If I has multiple color planes then so will the output image – each output color plane is the convolution of the corresponding input plane with the kernel K.

Spatial Operations

Commonly used convolution kernels.

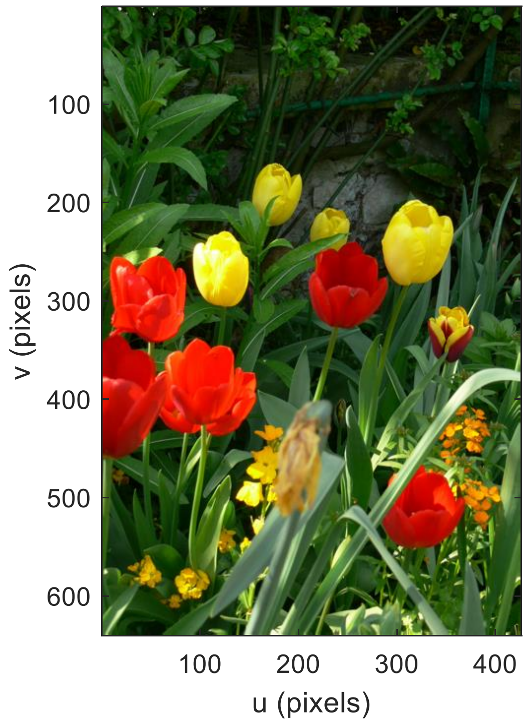


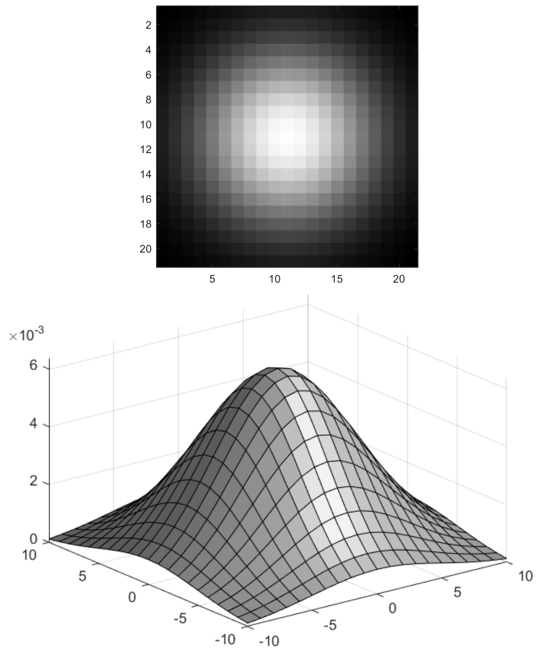
Smoothing

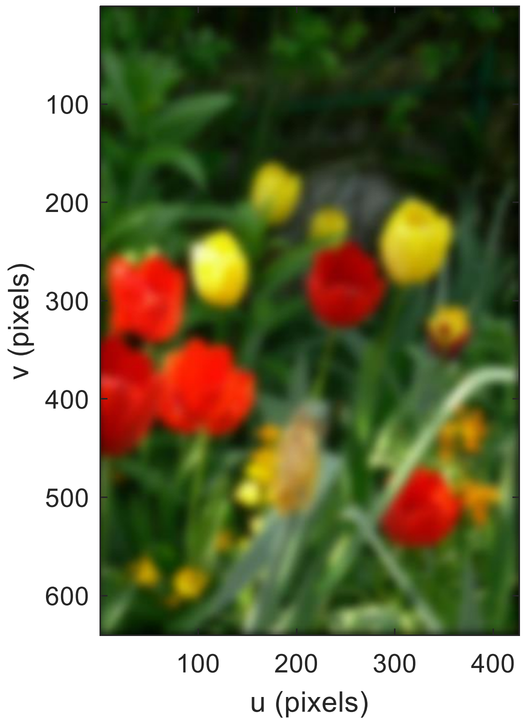
$$G(u,v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{2\sigma^2}}$$

2-dimensional Gaussian function which is symmetric about the origin and the volume under the curve is unit

```
inim=iread('flowers9.png','double');
figure;
K = kgauss(5);
idisp( iconvolve(inim, K) );
```



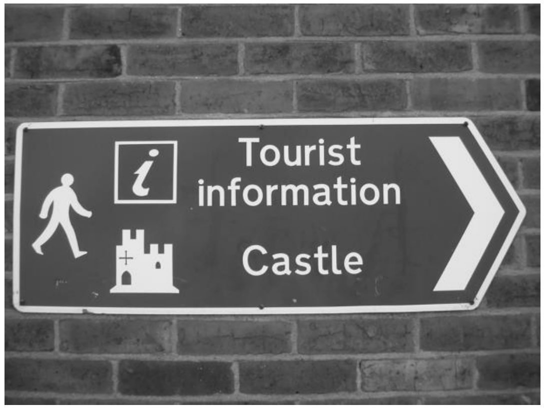


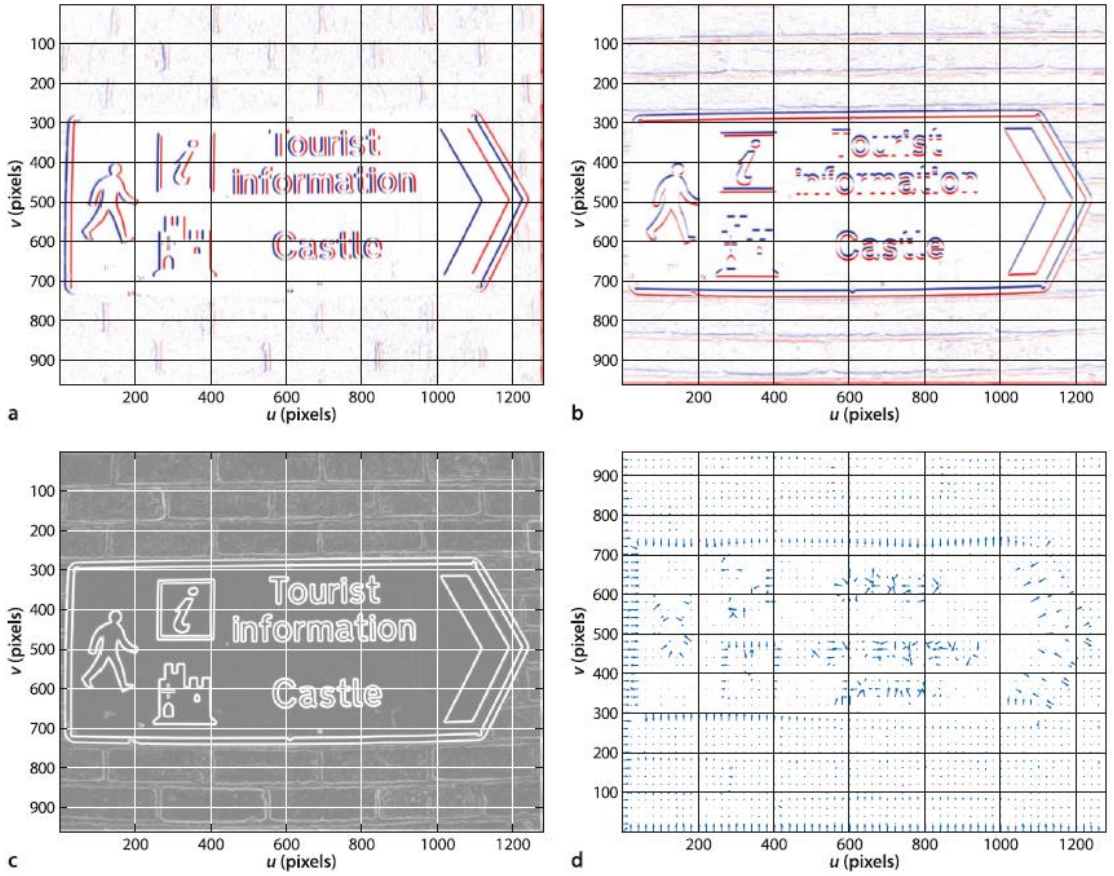


Edge Detection

- Frequently we are interested in finding the edges of objects in a scene

Edge gradient.
a *u*-direction gradient;
b *v*-direction gradient;
c gradient magnitude;
d gradient direction.
Gradients shown with *blue* as positive, *red* as negative and *white* as zero





a
```
K = [0.5 0 -0.5];
idisp( iconvolve(inim, K),
'invsigned')
```

Spatial Operations

- `Du = ksobel`

```
0.1250 0 -0.1250
0.2500 0 -0.2500
0.1250 0 -0.1250
```

$$\mathrm{I}_u = \mathrm{D} * \mathrm{I}$$
$$\mathrm{I}_v = \mathrm{D}^T * \mathrm{I}$$

- In operator form this is written where **D** is a derivative kernel such as Sobel.

$$\boldsymbol{I}_u = \boldsymbol{D} * \boldsymbol{I}$$
$$\boldsymbol{I}_v = \boldsymbol{D}^T * \boldsymbol{I}$$

- Taking the derivative of a signal accentuates high-frequency noise, and all images have noise.
- At the pixel level noise is a stationary random process – the values are not correlated between pixels.
- However the features that we are interested in such as edges have correlated changes in pixel value over a larger spatial scale.
- We can reduce the effect of noise by smoothing the image before taking the derivative.

$$\boldsymbol{I}_u = \boldsymbol{D} * (\boldsymbol{G} * \boldsymbol{I})$$

- Instead of convolving the image with the Gaussian and *then* the derivative, we exploit the associative property of convolution to write

$$\boldsymbol{I}_u = \boldsymbol{D} * (\boldsymbol{G} * \boldsymbol{I}) = \underbrace{(\boldsymbol{D} * \boldsymbol{G})}_{\text{DoG}} * \boldsymbol{I}$$

Spatial Operations

- We convolve the image with the *derivative of the Gaussian* (DoG) which can be obtained numerically by

```
Gu = iconvolve( Du, kgauss(sigma) , 'full');
```

- or analytically by taking the derivative, in the *u*-direction, of the Gaussian yielding

$$G_u(u,v) = -\frac{u}{2\pi\sigma^4} e^{-\frac{u^2+v^2}{2\sigma^2}}$$

Computing the horizontal and vertical components of gradient at each pixel

```
Iu = iconvolve( castle, kdgauss(2) );
Iv = iconvolve( castle, kdgauss(2)' );
```

allows us to compute the magnitude of the gradient at each pixel

```
m = sqrt( Iu.^2 + Iv.^2 );
```

This *edge-strength* image reveals the edges very distinctly.

The direction of the gradient at each pixel is

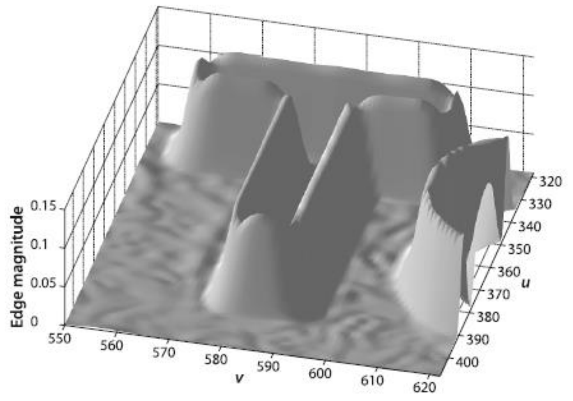
```
th = atan2(Iv, Iu);
```

- A well known and very effective edge detector is the Canny edge operator. It uses the edge magnitude and direction that we have just computed and performs two additional steps.

```
edges = icanny(castle, 2);
```

Spatial Operations

- Closeup of gradient magnitude around the letter T shown as a 3-dimensional surface

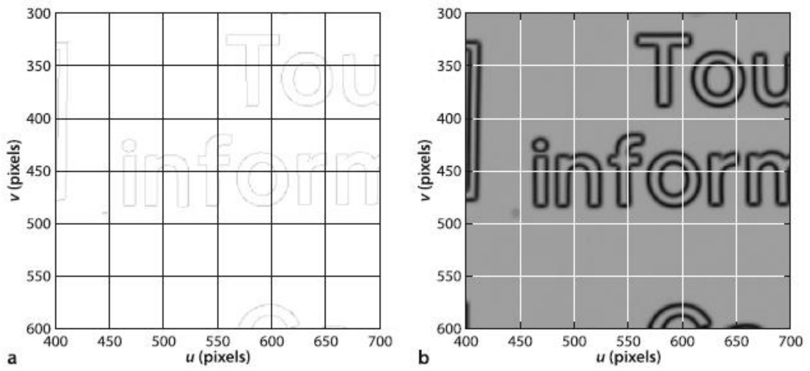


- Comparison of two edge operators:

a Canny operator with default parameters;

b Magnitude of derivative of Gaussian kernel (σ =2).

The |DoG| operator requires less computation than Canny but generates thicker edges. For both cases results are shown *inverted*, *white* is zero



- So far we have considered an edge as a point of high gradient, and nonlocal maxima suppression has been used to *search* for the maximum value in local neighborhoods.
- An alternative means to fi nd the point of maximum gradient is to compute the second derivative and determine where this is zero. The Laplacian operator

$$\nabla^2 \boldsymbol{I} = \frac{\partial^2 \boldsymbol{I}}{\partial u^2} + \frac{\partial^2 \boldsymbol{I}}{\partial v^2} = \boldsymbol{I}_{uu} + \boldsymbol{I}_{vv}$$

- is the sum of the second spatial derivative in the horizontal and vertical directions. For a discrete image this can be computed by convolution with the Laplacian kernel

```
L = klaplace()
0  1  0
1 -4  1
0  1  0
```

- which is isotropic – it responds equally to edges in any direction. The second derivative is even more sensitive to noise than the first derivative and is again commonly used in conjunction with a Gaussian smoothed image
- which we combine into the Laplacian of Gaussian kernel (LoG), and **L** is the Laplacian kernel given above. This can be written analytically as

$$\nabla^2 \boldsymbol{I} = \boldsymbol{L} * (\boldsymbol{G} * \boldsymbol{I}) = \underbrace{(\boldsymbol{L} * \boldsymbol{G})}_{\text{LoG}} * \boldsymbol{I}$$

Spatial Operations

- Laplacian of Gaussian.

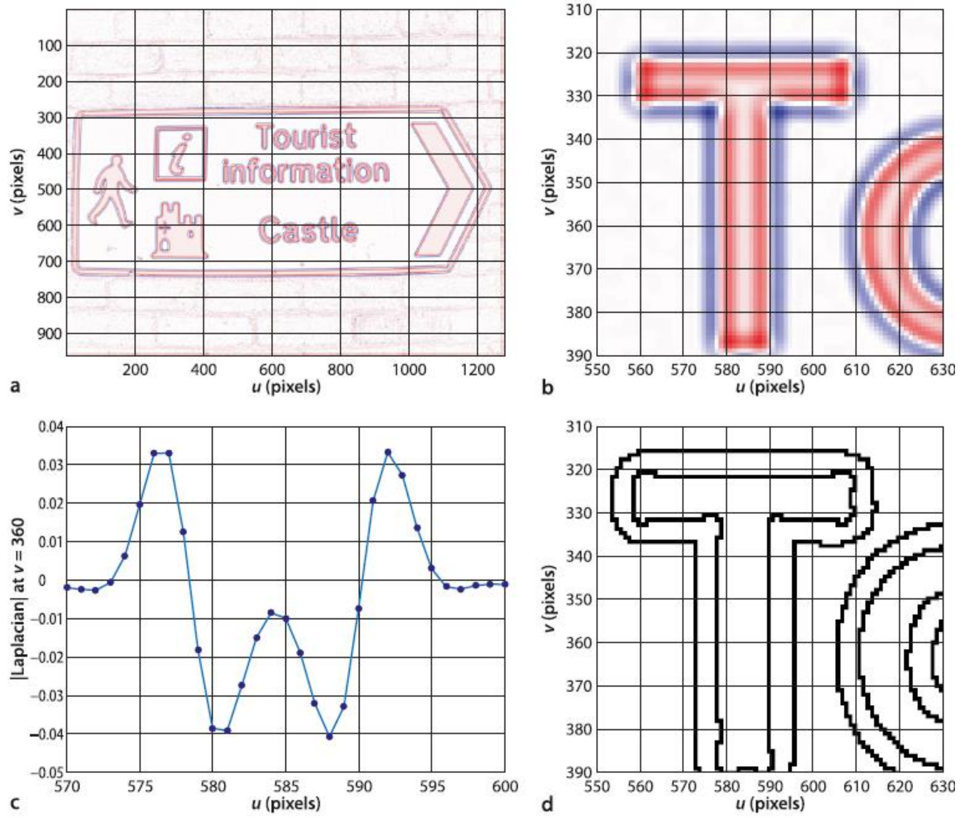
a Laplacian of Gaussian;

b closeup of **a** around the letter T where *blue* and *red* colors indicate positive and negative values respectively;

c a horizontal cross-section

of the LoG through the stem of the T;

d closeup of the zero-crossing detector output at the letter T.

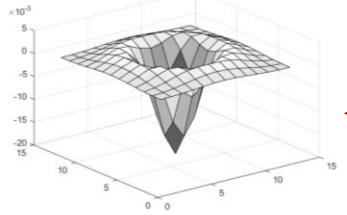


- we can combine obtain the Laplacian of Gaussian kernel (LoG) analytically as

$$LoG(u,v) = \frac{\partial^2 \boldsymbol{G}}{\partial u^2} + \frac{\partial^2 \boldsymbol{G}}{\partial v^2}$$
$$= \frac{1}{\pi\sigma^4}\left(\frac{u^2+v^2}{2\sigma^2} - 1\right)e^{-\frac{u^2+v^2}{2\sigma^2}}$$

- which is known as the Marr-Hildreth operator or the *Mexican hat kernel*
- We apply this kernel to our image by

```
lap = iconvolve( inim, klog(2) );
```



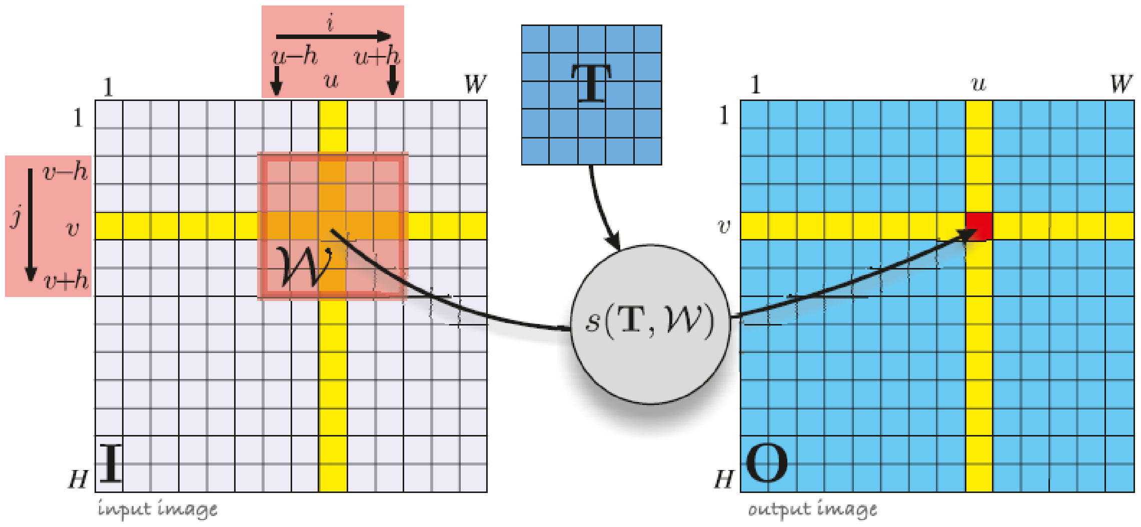
$\leftarrow$ klog(2)

Template Matching

- In our discussion so far we have used kernels that represent mathematical functions such as the Gaussian and its derivative and its Laplacian.
- We will now consider that the kernel *is an image* or a part of an image which we refer to as a template. In template matching we wish to find which parts of the input image are most similar to the temp
- Each pixel in the output image is given by

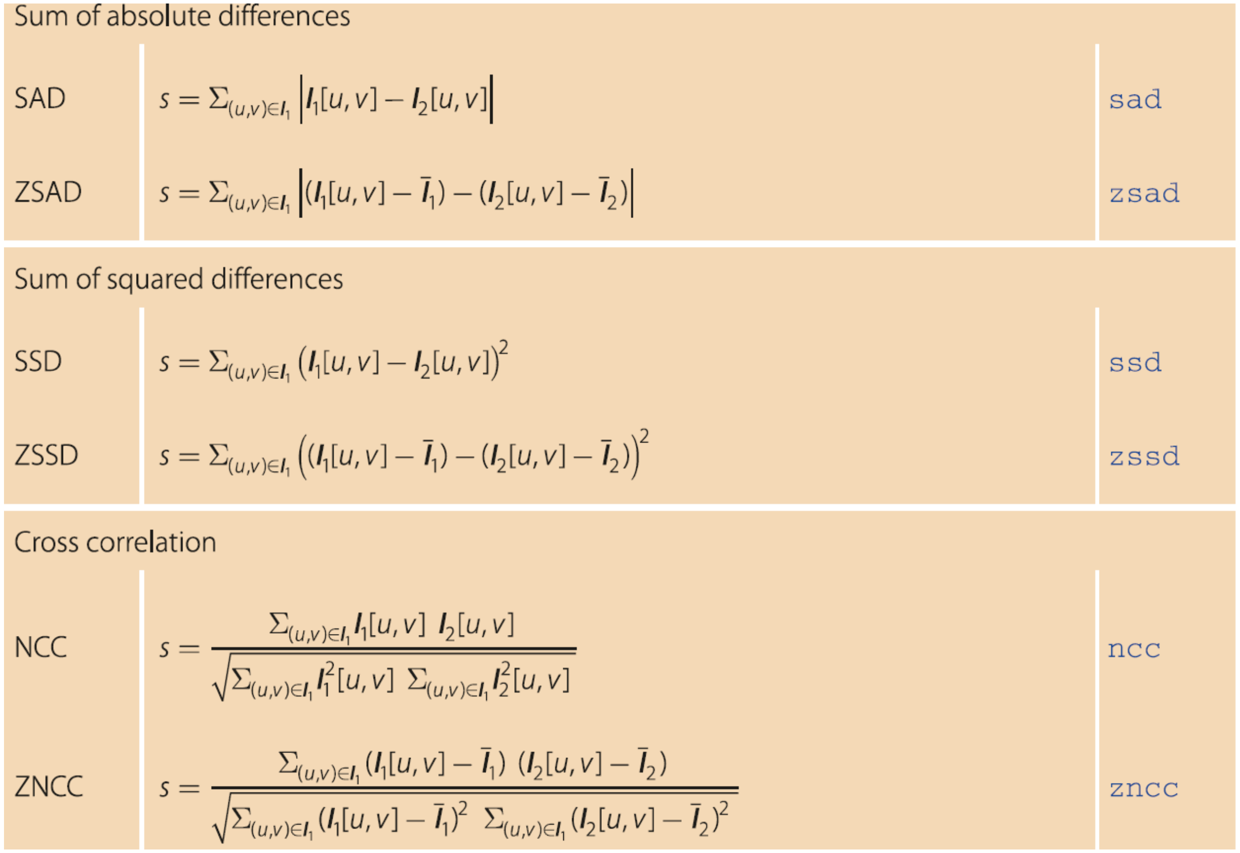
$$\boldsymbol{O}[u,v] = s(\boldsymbol{T}, W)\ , \forall (u,v) \in \boldsymbol{I}$$

- where $\boldsymbol{T}$ is the $w \times w$ template, the pattern of pixels we are looking for, with odd side length $w = 2h + 1$, and W is the $w \times w$ window centered at (u,v) in the input image.
- The function $s(\boldsymbol{I}_1, \boldsymbol{I}_2)$ is a scalar measure that describes the *similarity* of two equally sized images $\boldsymbol{I}_1$ and $\boldsymbol{I}_2$



Template Matching

- A number of common similarity measures



Sum of absolute differences		
SAD	$s = \Sigma_{(u,v)\in I_1} \left\| I_1[u,v] - I_2[u,v] \right\|$	sad
ZSAD	$s = \Sigma_{(u,v)\in I_1} \left\| (I_1[u,v] - \bar{I}_1) - (I_2[u,v] - \bar{I}_2) \right\|$	zsad
Sum of squared differences		
SSD	$s = \Sigma_{(u,v)\in I_1} \left(I_1[u,v] - I_2[u,v] \right)^2$	ssd
ZSSD	$s = \Sigma_{(u,v)\in I_1} \left((I_1[u,v] - \bar{I}_1) - (I_2[u,v] - \bar{I}_2) \right)^2$	zssd
Cross correlation		
NCC	$s = \dfrac{\Sigma_{(u,v)\in I_1} I_1[u,v]\ I_2[u,v]}{\sqrt{\Sigma_{(u,v)\in I_1} I_1^2[u,v]\ \Sigma_{(u,v)\in I_1} I_2^2[u,v]}}$	ncc
ZNCC	$s = \dfrac{\Sigma_{(u,v)\in I_1} (I_1[u,v] - \bar{I}_1)\ (I_2[u,v] - \bar{I}_2)}{\sqrt{\Sigma_{(u,v)\in I_1} (I_1[u,v] - \bar{I}_1)^2\ \Sigma_{(u,v)\in I_1} (I_2[u,v] - \bar{I}_2)^2}}$	zncc

Template Matching

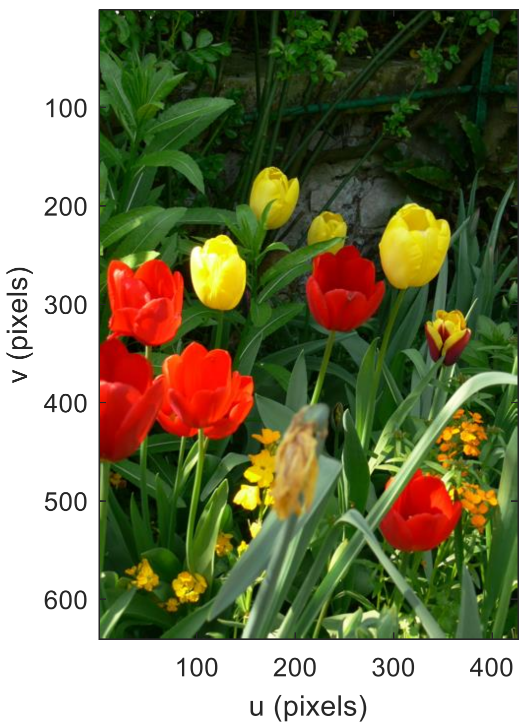
```
large = iread('flowers9.png', 'double');
temp = iread('flowers9_temp.png', 'double');
S = isimilarity(rgb2gray(temp), rgb2gray(large), @zncc);
idisp(S, 'colormap', 'jet', 'bar')

[mx,p] = peak2(S, 1, 'npeaks', 5);
idisp(large);
plot_circle(p, 30, 'edgecolor', 'g')
plot_point(p, 'sequence', 'bold', 'textsize', 24, 'textcolor', 'y')
```

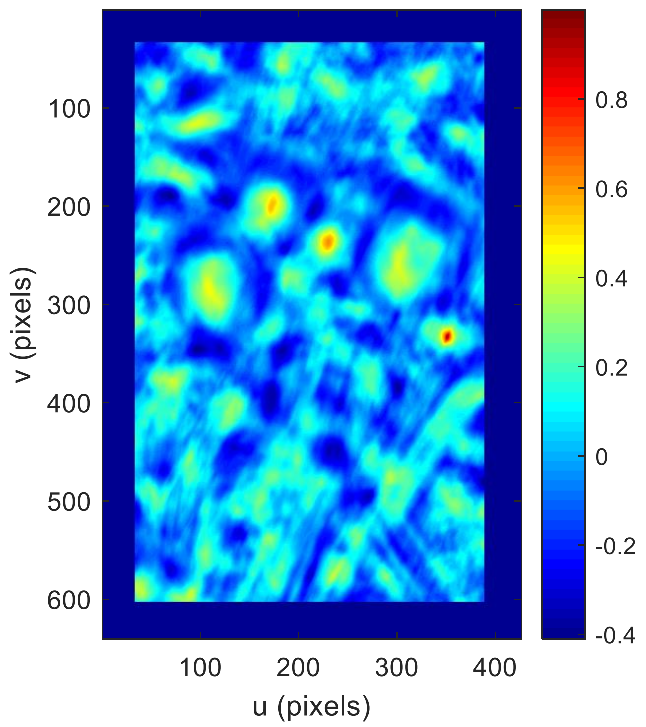
- There are some important points to note from this example. perhaps a 10–20% change in scale between T and W can be tolerated. For this example the template is synthetically created through cropping from the larger image.

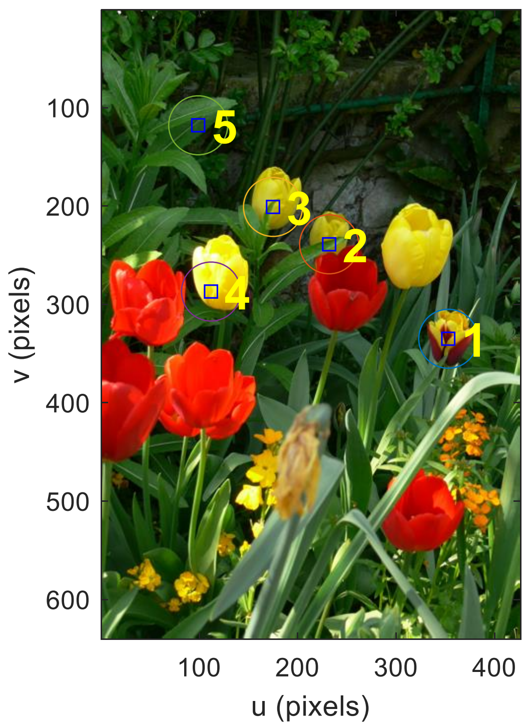
- Another problem is that the square template typically includes pixels from the background as well as the object of interest. As the object moves the background pixels may change, leading to a lower similarity score.

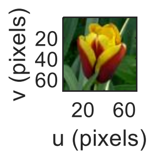
- Ideally the template should bound the object of interest as tightly as possible. In practice another problem arises due to perspective distortion. A square pattern of pixels in the center of the image will appear keystone shaped at the edge of the image and thus will match less well with the square template.

Template Matching







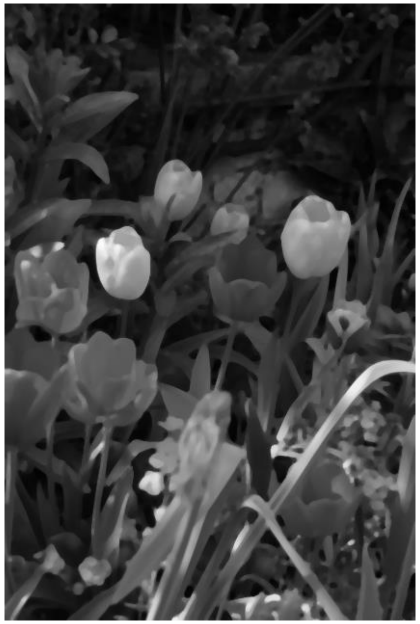


Nonlinear Operations

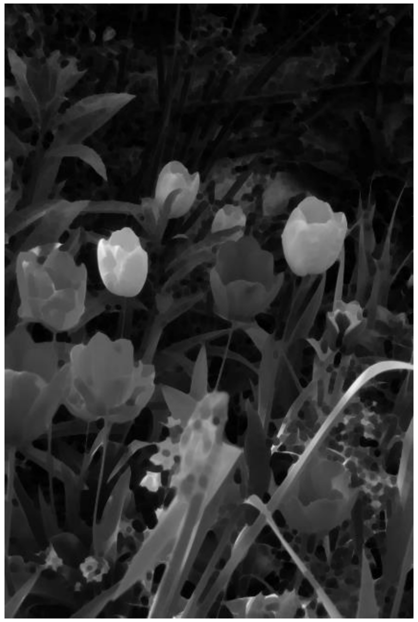
- Another class of spatial operations is based on nonlinear functions of pixels within the window

```
med = irank(inim, 13, 2);
min  = irank(inim, 25, 2);
max = irank(inim, 1, 2);
```

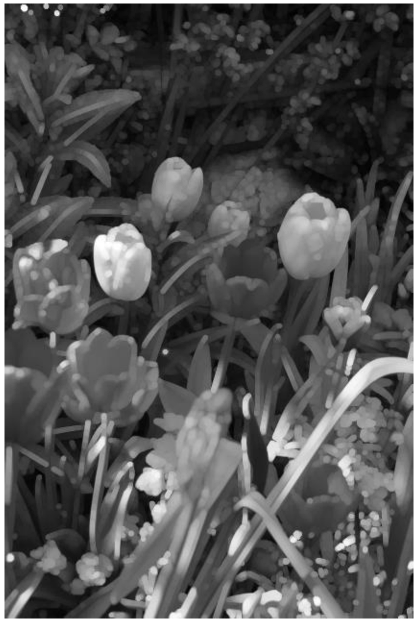
med



min



max



Mathematical Morphology

- Mathematical morphology is a class of nonlinear spatial operators
- Each pixel in the output matrix is a function of a *subset* of pixels in a region surrounding the corresponding pixel in the input image.

$$\boldsymbol{O}[u,v] = f\left(\boldsymbol{I}[u+i, v+j]\right), \forall (u,v) \in \boldsymbol{I}, \forall (i,j) \in \mathcal{S}$$

- where $\mathcal{S}$ is the structuring element, an arbitrary small binary image.
- For implementation purposes this is embedded in a rectangular window with odd side lengths.
- The structuring element is similar to the convolution kernel

Essential operations

- Dilation
- Erosion

Structuring elements

Small sets or sub-images used to probe an image under study for properties of interest

- Structuring element somewhat similar to a filter
- Contains only 0 and 1 values
- **Hot spot** marks origin of coordinate system of ***H***
- **Example of structuring element:** 1-elements marked with dark, 0-elements are empty.

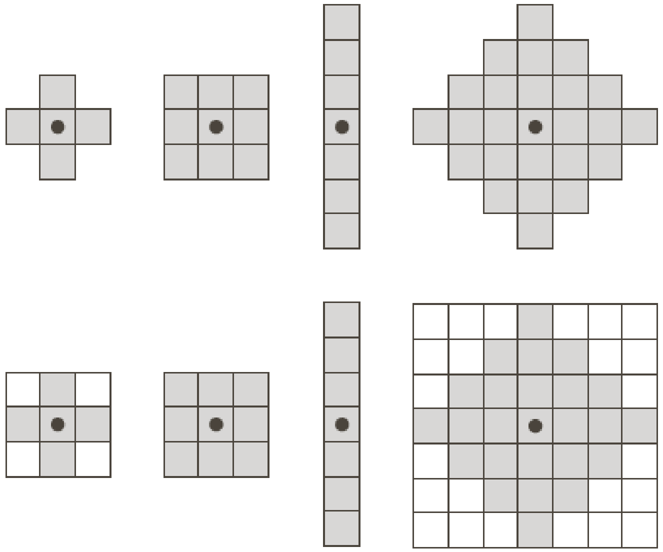


FIGURE 9.2 First row: Examples of structuring elements. Second row: Structuring elements converted to rectangular arrays. The dots denote the centers of the SEs.

Mathematical Morphology

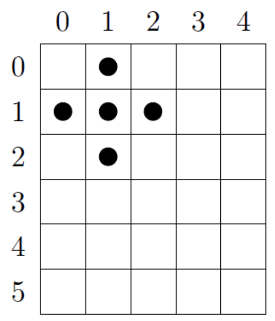
Translation:

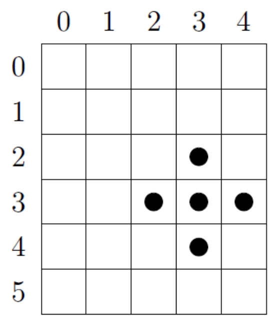
If A is set of pixels in binary image

$(A)_z$ is the set: A “translated” by $z = (z_1, z_2)$

$$(A)_z = \{c | c = a + z, for\ a \in A\}$$

- Example: If A is the cross-shaped set, and $z = (2,2)$



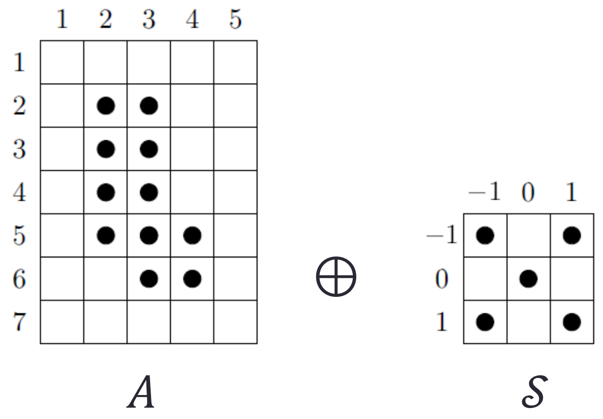


$(A)_z$

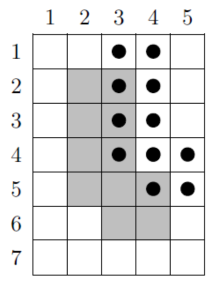
1- Dilation:

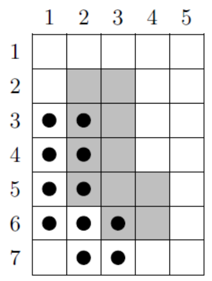
growth of the object

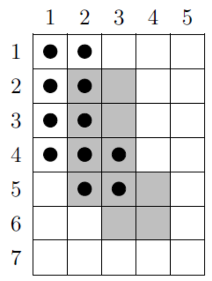
$$A \oplus \mathcal{S} = \bigcup_{s \in \mathcal{S}} (A)_s$$



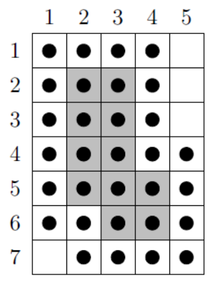
Mathematical Morphology


$A_{(-1,1)}$


$A_{(1,-1)}$


$A_{(-1,-1)}$

...


$A \oplus B$

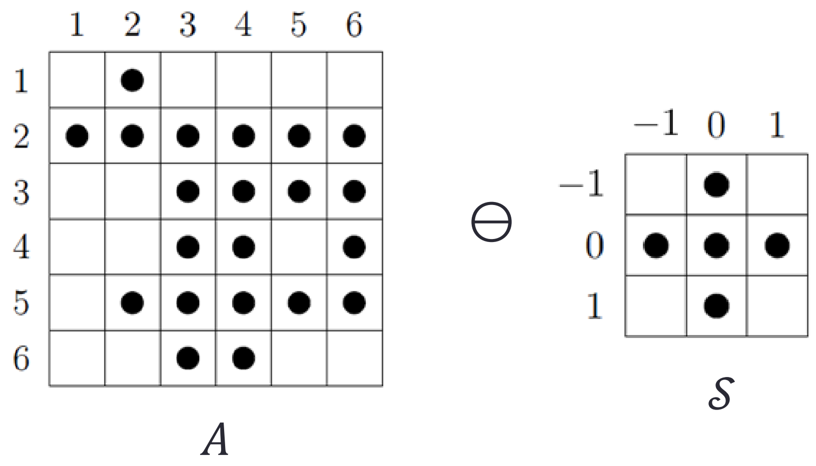
Dilation is actually union of all translations

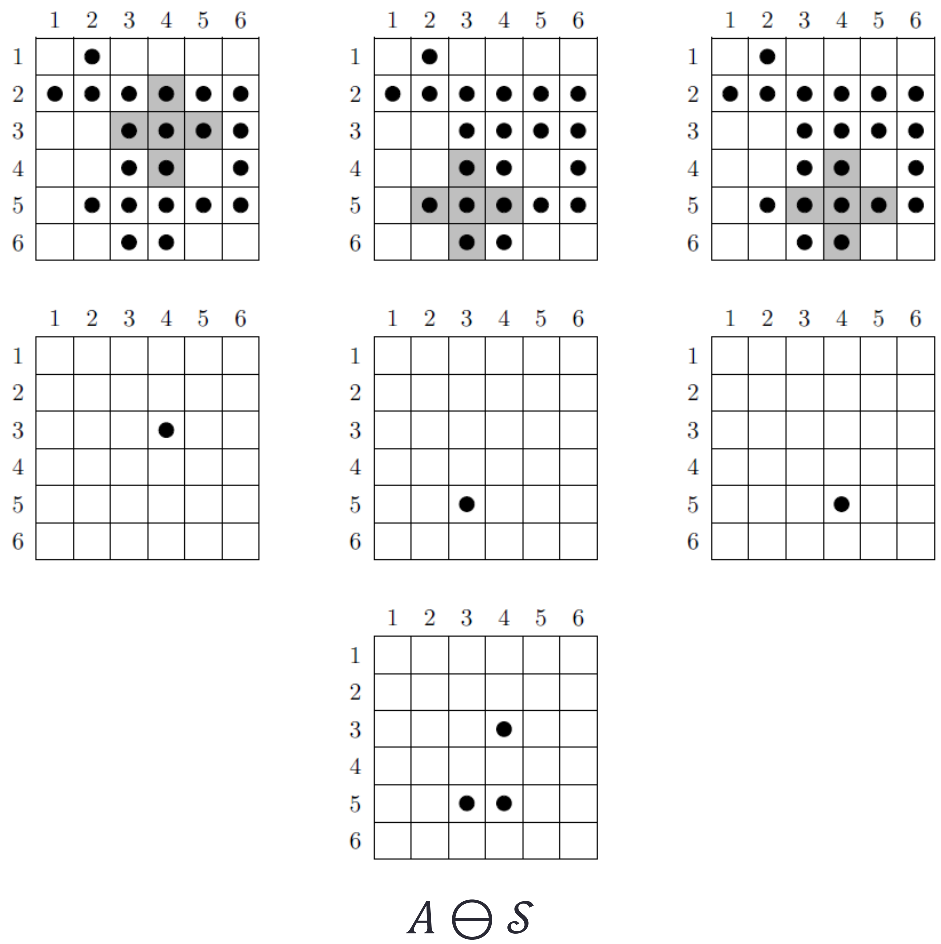
2 – Erosion

- Does the structuring element **fit the set?**
- Erosion of a set A by structuring element B: $A \ominus \mathcal{S} = \{a | (\mathcal{S})_a \subseteq A\}$ **shrink the region**

All points a such that $\mathcal{S}$ translated by a is contained in A

Mathematical Morphology





Mathematical Morphology

$A \circ \mathcal{S} = (A \ominus \mathcal{S}) \oplus \mathcal{S}$ **3- Opening** : erosion followed by dilation

$A \bullet \mathcal{S} = (A \oplus \mathcal{S}) \ominus \mathcal{S}$ **4- Closing** : dilation followed by erosion

```
S = kcircle(3)

S=
0  0  0  1  0  0  0
0  1  1  1  1  1  0
0  1  1  1  1  1  0
1  1  1  1  1  1  1
0  1  1  1  1  1  0
0  1  1  1  1  1  0
0  0  0  1  0  0  0
```

We can apply a closing operation to fill the holes in the objects

```
>> closed = iclose(objects, S);
```

The holes have been filled, but the noise pixels have grown to be small circles and some have agglomerated. We can eliminate these by an opening operation

```
>> clean = iopen(closed, S);
```

and the result shown is a considerably cleaned up image. If we apply the operations in the inverse order, opening then closing

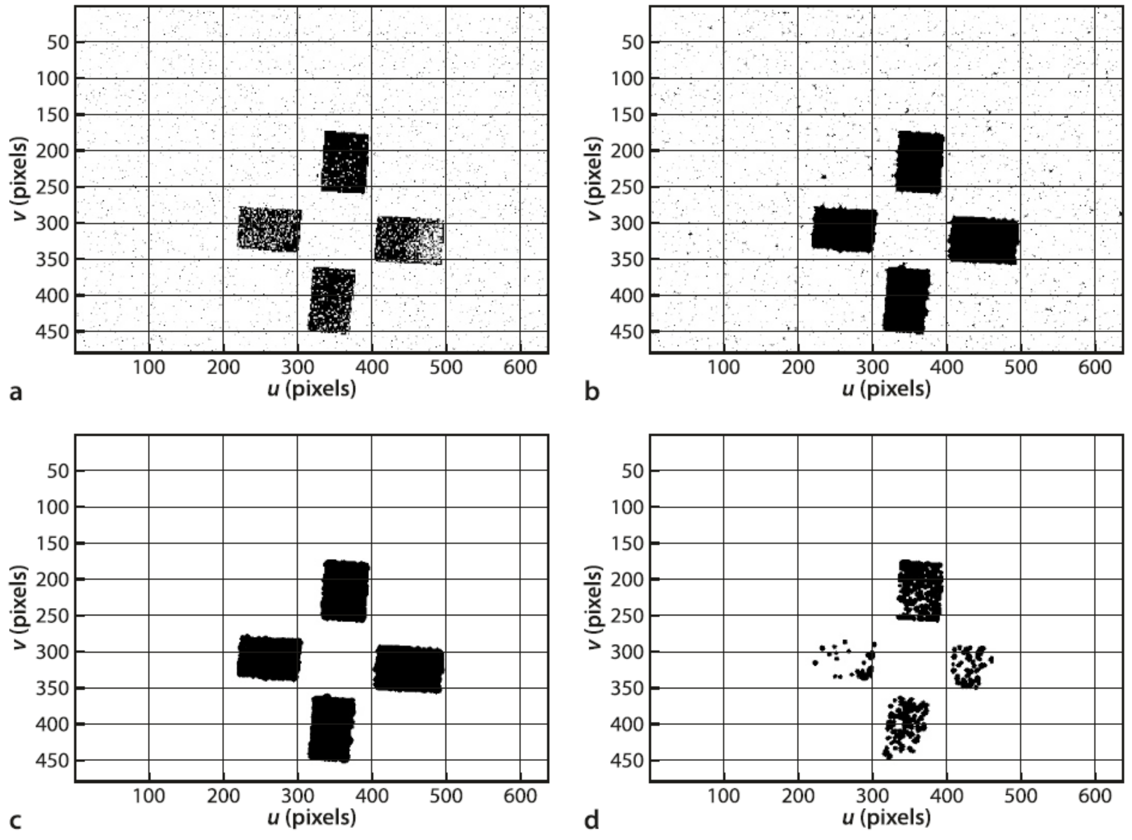
```
>> opened = iopen(objects, S);
>> closed = iclose(opened, S);
```

the results are much poorer.

Mathematical Morphology

Morphological cleanup.
a Original image,
b original after opening,
c opening then closing,
d closing then opening.

Structuring element is a *circle* of radius 3. Color map is inverted, set pixels are shown as *black*


v (pixels)
u (pixels)
a
b
c
d