Quick primer on Matlab for IACV

MATlab : Matrix Laboratory can be used as a scripting language any variable is a matrix (a scalar is a special case of a multidimensional array)

Credits:

- Alessandro Giusti alessandrog@idsia.ch
- Giacomo Boracchi giacomo.boracchi@polimi.it - September 28th, 2020

Edits: Luca Magri - September 19th, 2023

Clear the workspace and close all the figure and clear the command window

```
close all % close all the figures
clear % clear the workspace
clc % clear the command window
```

Print to the command window

```
disp('For enquiry, please send an email to luca.magri@polimi.it');
```

```
For enquiry, please send an email to luca.magri@polimi.it
```

```
fprintf('there is also a C-like printing function\n%s %dth, %d', 'September', 20, 2018
```

```
there is also a C-like printing function
September 20th, 2018
```

Dealing with variables:

all the variables are stored in matrix with proper dimensions. Scalar are treated as 1x1 matrix.

Variables are created by assignements, the size of a variable is the dimension of the matrix representing it.

```
v = 12;
c = 'c';
size(v)
```

```
ans = 1×2
     1     1
```

```
size(c)
```

```
ans = 1×2
     1     1
```

```
% use ; at the end of instruction not to display results
% in command line
v = 7;
```

Data types are automatically defined depending on the assigned value and can be checked using whos

```
whos v
```

```
  Name      Size            Bytes  Class     Attributes

  v         1x1                 8  double
```

Variables have datatypes (usually you can forget, but not when dealing with images).

Most common types we will consider are:

- *double*,(i.e., double-precision floating point),
- *uint8* (i.e., unsigned integers with 8 bits, [0 255] range),
- *logical* (i.e., boolean)

We can explicilty cast a double to a uint8 type

```
v = uint8(v)
```

```
v = uint8

    7
```

```
whos v
```

```
  Name      Size            Bytes  Class    Attributes

  v         1x1                 1  uint8
```

Arrays

can be row or column vector.

```
% a row vector (commas can be omitted)
r = [1, 2, 3, 4]
```

```
r = 1×4
     1     2     3     4
```

```
size(r)
```

```
ans = 1×2
     1     4
```

It is important to remember the index start from 1.

```
r(1)
```

```
ans = 1
```

```
%r(0) %raises an error! Array indices must be positive integers or logical values.
r(2) % returns the first element of the array r
```

```
ans = 2
```

```
% a column vector
c = [1; 2; 3; 4]
```

```
c = 4×1
     1
     2
     3
     4
```

```
size(c)
```

```
ans = 1×2
     4     1
```

You can define vectors by regular increment operator: [start : step : end]

```
a = [1 : 2 : 10]
```

```
a = 1×5
     1     3     5     7     9
```

when omitted, step is equal to 1

```
a2 = [1 : 10]
```

```
a2 = 1×10
     1     2     3     4     5     6     7     8     9    10
```

Matrices

```
v=[1 2; 3 4]
```

```
v = 2×2
     1     2
     3     4
```

```
size(v)
```

```
ans = 1×2
     2     2
```

Array concatenation

you can concatenate matrices or vectors as far as their sizes are consistent. ' denotes the transpose operation

```
B = [v', v']
```

```
B = 2×4
     1     3     1     3
     2     4     2     4
```

```
C = [v ; v]
```

```
C = 4×2
     1     2
     3     4
     1     2
     3     4
```

```
% the dimension of the data, visible also typing whos
dim = size(C)
```

```
dim = 1×2
     4     2
```

```
num_rows = size(C,1)
```

```
num_rows = 4
```

```
num_cols = size(C,2)
```

```
num_cols = 2
```

```
% this is not allowed
disp('K = [r,c]: this is not allowed')
```

```
K = [r,c]: this is not allowed
```

```
%K = [r,c]
```

Other examples of array concatenation:

```
my_vec1 = [1 2 3];
my_vec2 = 4:6;
```

```
my_matrix = [my_vec1; my_vec2]
```

```
my_matrix = 2×3
     1     2     3
     4     5     6
```

```
size(my_matrix)
```

```
ans = 1×2
     2     3
```

```
my_long_vector = [my_vec1 my_vec2]
```

```
my_long_vector = 1×6
     1     2     3     4     5     6
```

```
size(my_long_vector)
```

```
ans = 1×2
     1     6
```

C = cat(dim,A,B) concatenates B to the end of A along dimension dim when A and B have compatible sizes (the lengths of the dimensions match except for the operating dimension dim).

```
my_matrix = cat(1,my_vec1,my_vec2)
```

```
my_matrix = 2×3
     1     2     3
     4     5     6
```

```
my_long_vector = cat(2,my_vec1,my_vec2)
```

```
my_long_vector = 1×6
     1     2     3     4     5     6
```

```
my_3d_matrix = cat(3,my_vec1,my_vec2)
```

```
my_3d_matrix =
my_3d_matrix(:,:,1) =

     1     2     3


my_3d_matrix(:,:,2) =

     4     5     6
```

```
size(my_3d_matrix)
```

```
ans = 1×3
     1     3     2
```

```
%% indexing (starts from 1)
my_vec = (1:10)';
my_vec(1)
```

```
ans = 1
```

```
my_matrix = [1:4;5:8;9:12]
```

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

```
my_matrix(3,2)
```

```
ans = 10
```

```
% coumn-wise linear indexing for matrices
my_matrix(6)
```

```
ans = 10
```

Subarray

v(indexes) returns a vector of all the elements of v at locations in array indexes

```
% you can reference single or multiple values in an array:
my_matrix(2,3) % first row and second column (row and column indices are 1-based)
```

```
ans = 7
```

```
my_matrix(:,2) % the second column of my_matrix
```

```
ans = 3×1
     2
     6
    10
```

```
my_matrix(:,2) % the first row of my_matrix
```

```
ans = 3×1
     2
     6
    10
```

```
B = my_matrix(:,[1,3]) % some of the columns
```

```
B = 3×2
     1     3
     5     7
     9    11
```

You can extend vectors / matrices by assigning some element out of the range

```
my_matrix(:,5)=1
```

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

You can use end to specify the last row/columns

```
my_matrix(1:3,[2 4])
```

```
ans = 3×2
     2     4
     6     8
    10    12
```

```
my_matrix(1:end,[2 4])
```

```
ans = 3×2
     2     4
     6     8
    10    12
```

```
my_matrix(:,[2 4])
```

```
ans = 3×2
     2     4
     6     8
    10    12
```

```
my_matrix(1:2,[2 4])
```

```
ans = 2×2
     2     4
     6     8
```

```
my_matrix(1:end-1,[2 4])
```

```
ans = 2×2
     2     4
     6     8
```

You can unfold a matrix columnwise

```
my_vector = my_matrix(:)
```

```
my_vector = 15×1
     1
     5
     9
     2
     6
    10
     3
     7
    11
     4
     ⋮
```

Operations

on vectors are from linear algebra common operations work on matrices (careful about multiplication and division)

```
v = [1 2 3]
```

```
v = 1×3
     1     2     3
```

```
v*v'
```

```
ans = 14
```

```
v'*v
```

```
ans = 3×3
     1     2     3
     2     4     6
     3     6     9
```

There are also element-wise operators

```
[1 2 3].*[4 5 6]
```

```
ans = 1×3
     4    10    18
```

```
[1 2 3] + 5
```

```
ans = 1×3
     6     7     8
```

```
[1 2 3] * 2 % no need to use element-wise operator with scalars
```

```
ans = 1×3
     2     4     6
```

```
[1 2 3] .* 2 %explicit element-wise multiplication
```

```
ans = 1×3
     2     4     6
```

```
[1 2 3] / 2
```

```
ans = 1×3
    0.5000    1.0000    1.5000
```

```
[1 2 3] ./ 2 %explicit element-wise division
```

```
ans = 1×3
    0.5000    1.0000    1.5000
```

```
[1 2 3] .^ 2 %explicit element-wise power
```

```
ans = 1×3
     1     4     9
```

```
[1 2 3] * [4 5 6]' % this matrix product, returns a scalar (it is the inner product)
```

```
ans = 32
```

```
[1 2 3]' * [4 5 6] % this is the matrix product, returns a 3 x 3 matrix
```

```
ans = 3×3
     4     5     6
     8    10    12
    12    15    18
```

```
% rounding functions
ceil(10.56)
```

```
ans = 11
```

```
floor(10.56)
```

```
ans = 10
```

```
round(10.56)
```

```
ans = 11
```

```
ceil(0:0.1:1)
```

```
ans = 1×11
     0     1     1     1     1     1     1     1     1     1     1
```

```
% arithmetic functions
sum([1 2 3 4])
```

```
ans = 10
```

```
sum([1:4;5:8])
```

```
ans = 1×4
     6     8    10    12
```

```
sum([1:4;5:8],2)
```

```
ans = 2×1
    10
    26
```

```
sum(sum([1:4;5:8]))
```

```
ans = 36
```

```
my_matrix = [1:4;5:8];
sum(my_matrix(:))
```

```
ans = 36
```

Printing

```
disp('Hello World!');
```

```
Hello World!
```

```
disp(['Hello ', 'World!']);
```

```
Hello World!
```

```
% string concatenation in printf.
disp(['number of columns in the matrix: ', num2str(size(my_matrix))]);
```

```
number of columns in the matrix: 2  4
```

```
fprintf('number of columns in the matrix: %f\n',size(my_matrix,2));
```

```
number of columns in the matrix: 4.000000
```

Logicals

you can compare a vector via the customary relational operators and obtain a vector of logicals (i.e. b)

```
b = v>2
```

```
b = 1×3 logical array
   0   0   1
```

```
whos b
```

```
  Name      Size            Bytes  Class     Attributes

  b         1x3                 3  logical
```

Quick primer on image processing in Matlab

Spatial domain processing

how images are treated in MATLAB... as matrices... depending on images you deal with → take care

Credits: Giacomo Boracchi October 1st, 2018

Edits: Luca Magri, September 2023, for comments and suggestions write to luca.magri@polimi.it

```
close all
clear
```

Images are matrices

with 3 channels: RGB image is made of 3 matrix, one for each channel

```
im = imread('E1_data/image_Lena512.png');
whos im
```

Name	Size	Bytes	Class	Attributes
im	512x512	262144	uint8	

in each matrix you have the pixel R, G, B in each channel (different convention in OpenCV)

0 ÷ 2055 which tells you how much light of that color

images can be displayed:

- imshow preserve aspect ratio and uses grayscale colormap for 2D images

```
imshow(im); % show image
```

2055 ~ light
0 ~ dark

load image im MATLAB by imread

Grey ~ 1027, 1027, 1027 all same

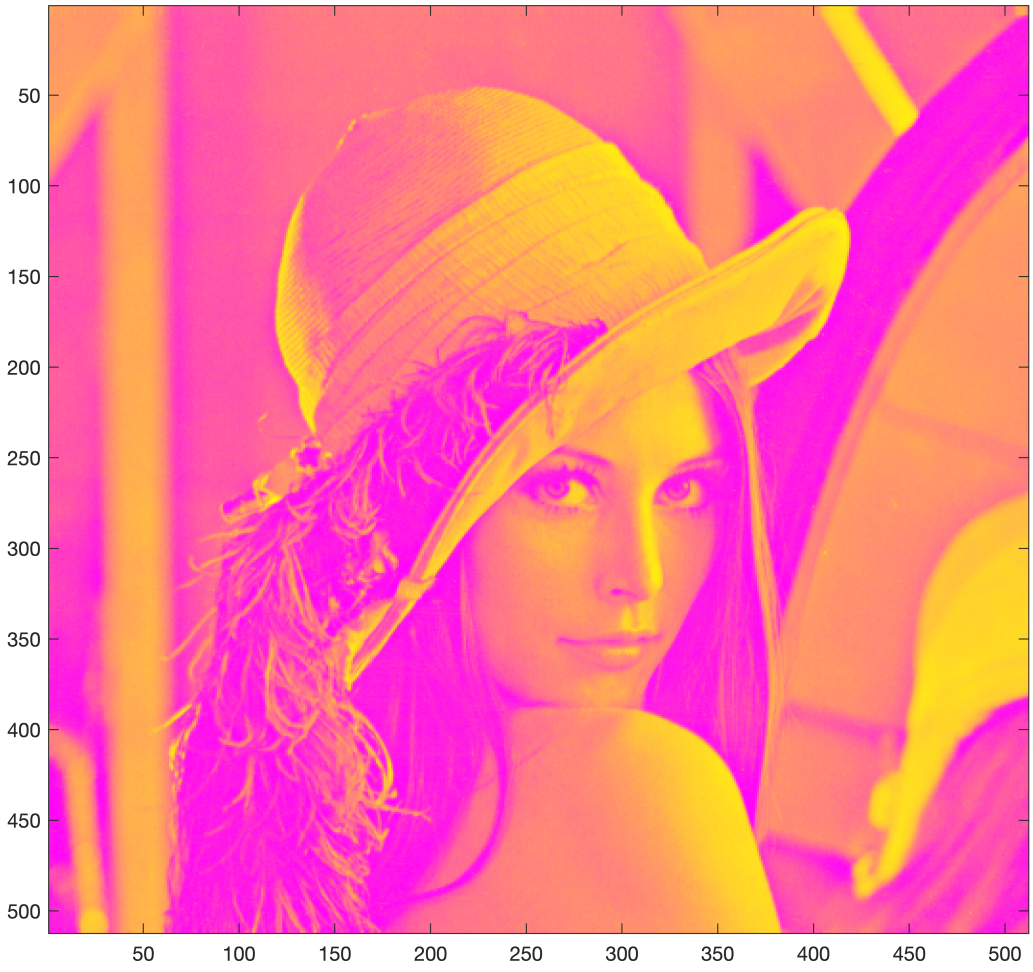
store it in a matrix of 3 dimension usually 512 x 512 x 512 if colored (RGB)
512 x 512 if greyscaler (light/dark)

- imagesc does not preser aspect ratio and can use arbitrary colormaps

```
imagesc(im), colormap spring
```

specify different color map

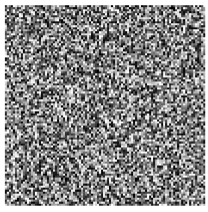
im different colormap!



Matrices are images

```
M = rand(100,100);
imshow(M);
```

you can define random MATRIX and plot it using imshow



range of intensity 0 ÷ 2055

↑ unsigned integer on 8 bit

images represented as uint8 or using double 0 ÷ 1

... but datatype matters! Careful about data range: [0 1] for double and logical, [0 255] for uint8

↳

double with range 0÷1 is OK BUT

```
imshow(double(im)); %An image of double in [0,255]
```

you have to rescale image depending on how you represent it!

if image with values uint8 then casting to double → all is White!

from uint8 everything is set to 1, 1, 1 WHITE! saturated

rescaling is necessary when casting to double images

```
imshow(double(im)/255); % An image of double in [0,1]
```

proper rescaling!

```
imshow(uint8(double(im)/255)); % An image of uint8 in [0,1]
```

cast uint8 (0÷1) ↗ cale is to very low value!

remember to use correct range of intensity!

plot the image histogram

compute histogram of image intensity values

histogram of pixels intensity

```
h = hist(im(:), [0: 255]);
```

Plot the histogram

```
figure(2)
stairs([0: 255], h, 'b', 'LineWidth', 3)
title('intensity histogram')
axis tight
```

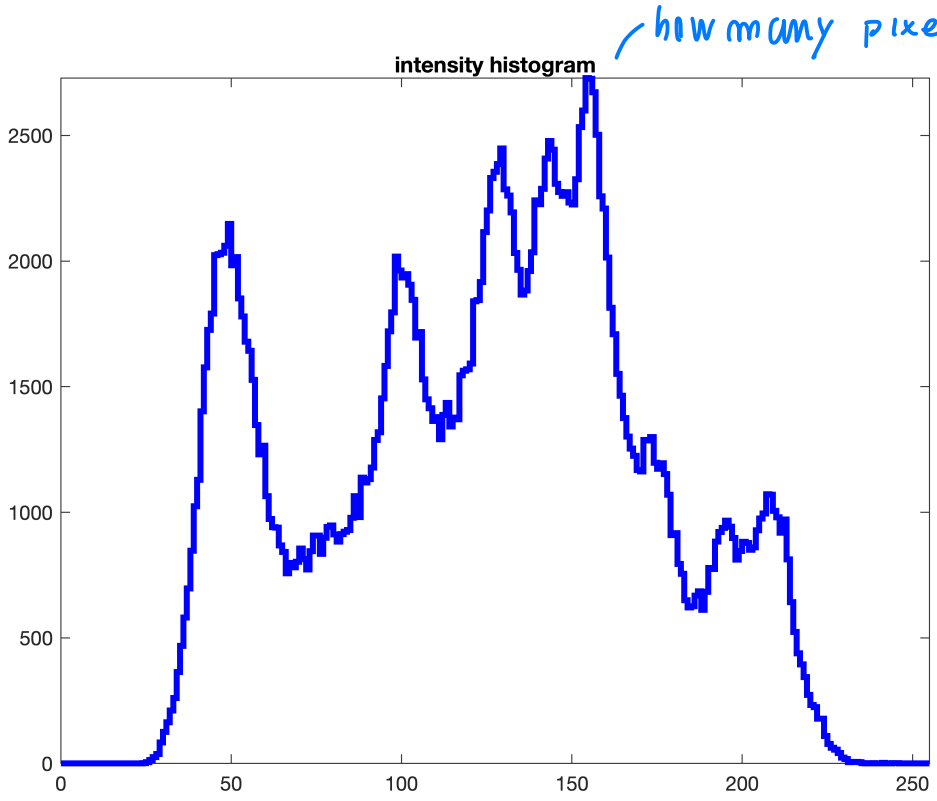


Image brigthness can be modified by adding an offset to each pixel

```
figure(1), imshow([im, im + 50]), title('brigthness increases of 50 graylevels');
```

add intensity!


```
% few pixels are saturated
figure(1), imshow(im + 100), title('brigthness increases of 100 graylevels');
```

brigthness increases of 100 graylevels

↳ +100 on each pixel ⇒ brighter

```
% contrast can be modified by streching the histogram, image has to be
% converted in double format
eq = double(im - min(im(:)))/double(max(im(:)) - min(im(:))) * 255;

figure(1), imshow(uint8(eq), []), title('Image now covers the whole range');
```

↑ stretch the intensity

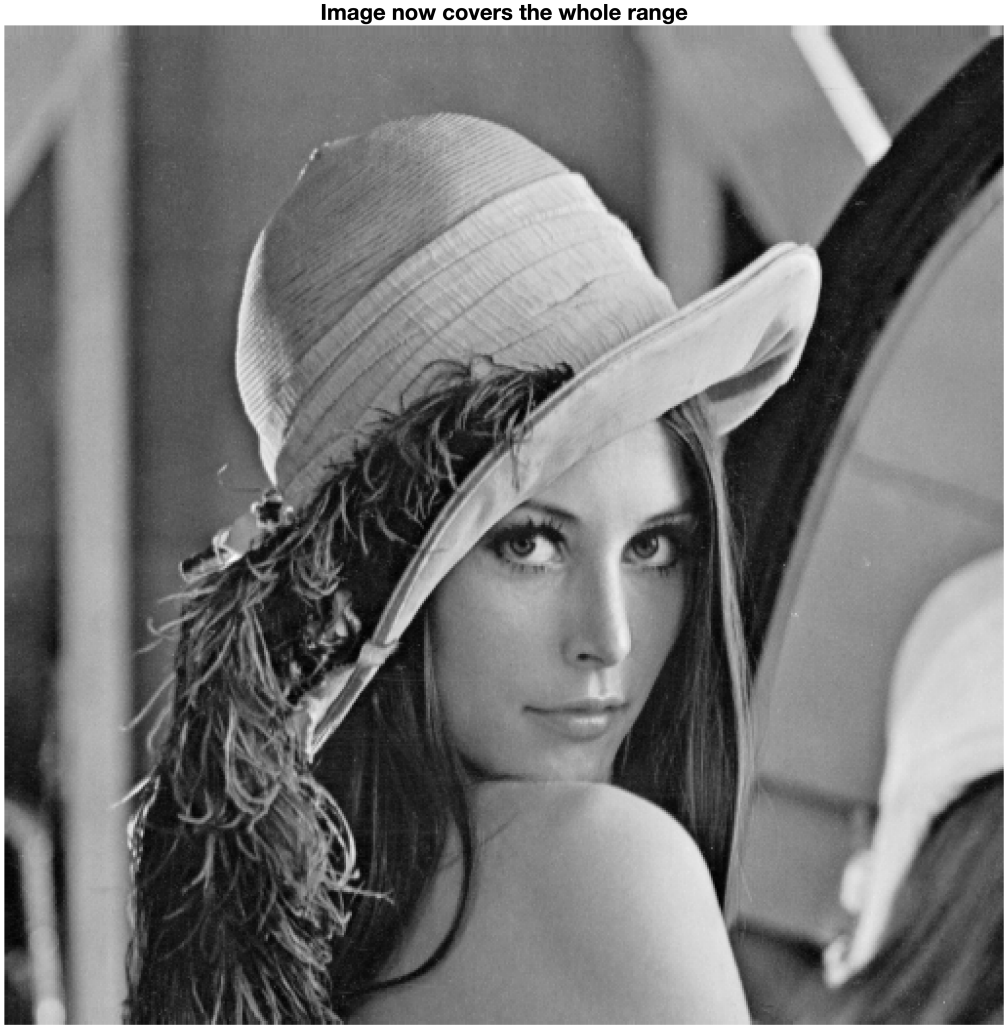
better contrast

$$\left(\frac{im - min}{range}\right) * 255$$

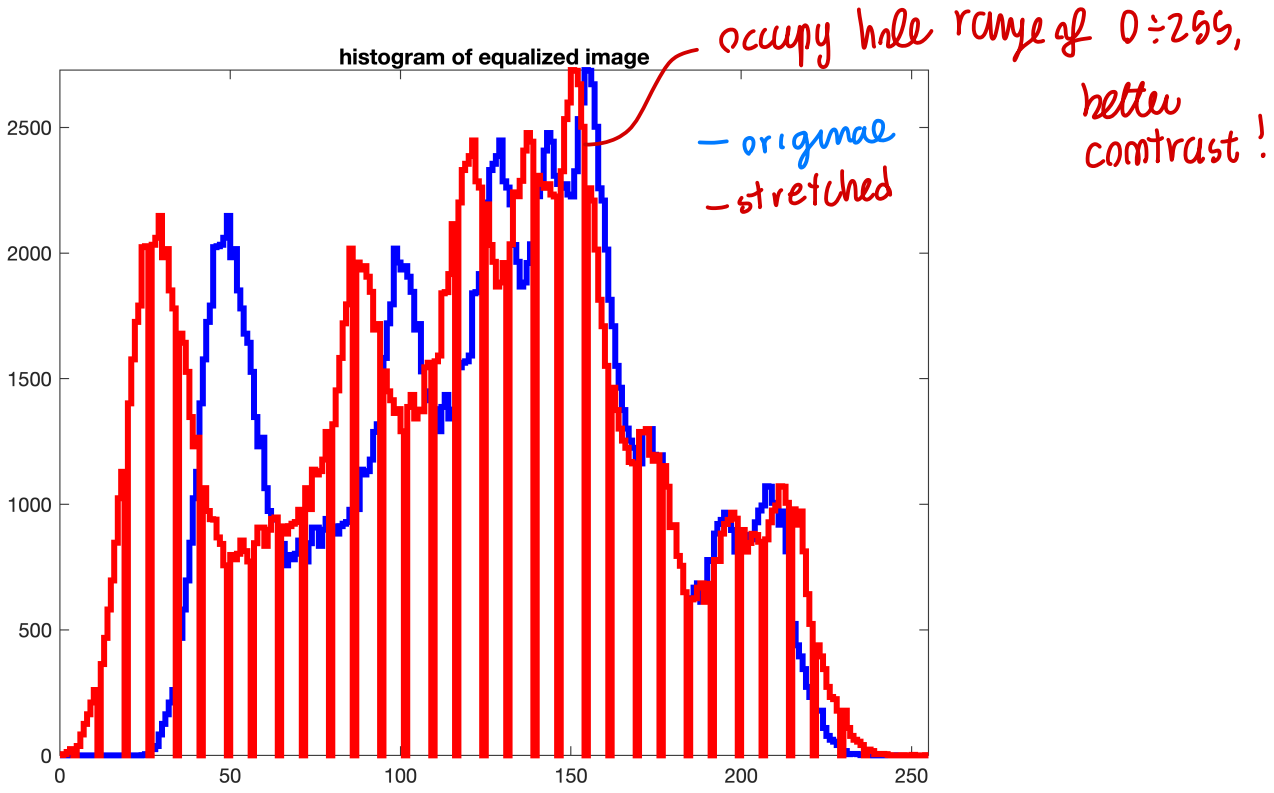
$[0 \div 1]$

so I have all range

Image now covers the whole range



```
heq = hist(eq(:), [0: 255]);
figure(2),
stairs([0: 255], h, 'b-', 'LineWidth', 3), title('histogram of original image')
hold on;
stairs([0: 255], heq, 'r-', 'LineWidth', 3), title('histogram of equalized image')
axis tight
```



The equalized image (in red) occupy the whole range [0,255]

Image ranges (in the visualization) can be also controlled by the second argument to imshow

```
figure;
subplot(2,2,1)
imshow(im,[-100 156]);
title('increased brighness same contrast');

subplot(2,2,2)
imshow(im,[0 156]);
title('increased brighness and contrast');

subplot(2,2,3)
imshow(im,[ 100 256]);
title('decreased brighness, increased contrast');

subplot(2,2,4)
imshow(im,[ 100 356]);
title('decreased brighness, same contrast');
```

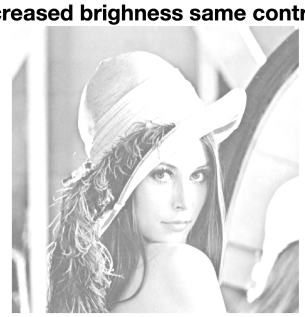
specify range you want to look at!

-100 := dark

156 := white

↵ rescale intensity of image

increased brighness same contrast



increased brighness and contrast

decreased brighness, increased contrast

decreased brighness, same contrast

Consider that brightness and contrast can be also changed by

- adding an offset

- scaling the image

Color is represented by 3 channels (RGB)

You get a 3d matrix

```
im=imread('E1_data/bluecube.jpg');
whos im
```

```
  Name         Size                Bytes  Class    Attributes

  im        1417x1417x3           6023667  uint8
```

```
figure(45)
imshow(im);
```

you can display each channel independently

this is a 3D matrix, because it has 3 dimensions (not because the 3rd dimension has only 3 elements).

```
test=im(:,:,1:2);
whos test
```

```
  Name         Size                 Bytes  Class    Attributes

  test      1417x1417x2           4015778  uint8
```

We can instead see each single channel as a grayscale image

```
imr=im(:,:,1);
whos imr % note: now it's 2D, so it will be displayed as grayscale
```

```
  Name         Size                 Bytes  Class    Attributes

  imr       1417x1417            2007889  uint8
```

```
imshow(imr), title('red channel')
```


```
img=im(:,:,2);
imshow(img), title('green channel')
```

green channel

```
imb=im(:,:,3);
imshow(imb), title('blue channel')
```

blue channel

Logicals

We can plot the result of logical operations

use logical operations on the image to threshold image for example

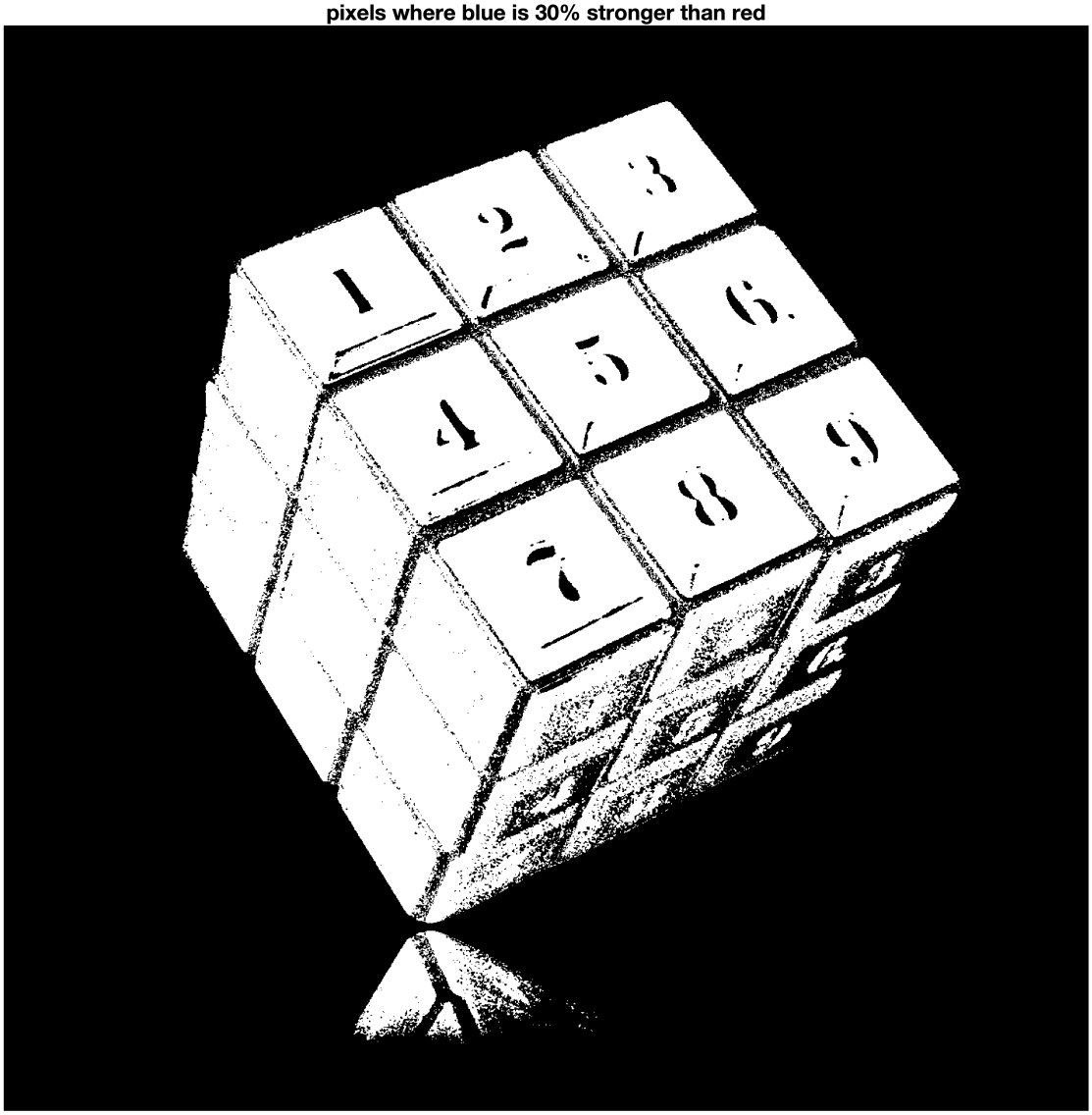
```
l = imb > (1.3 * imr);
whos l
```

```
  Name         Size                Bytes  Class      Attributes

  l         1417x1417            2007889  logical
```

```
imshow(l); title('pixels where blue is 30% stronger than red');
```

logical image 0 or 1 of blue channel such that > 1.3 red channel pixel pixel more blue than red → this extract blue pixels

pixels where blue is 30% stronger than red



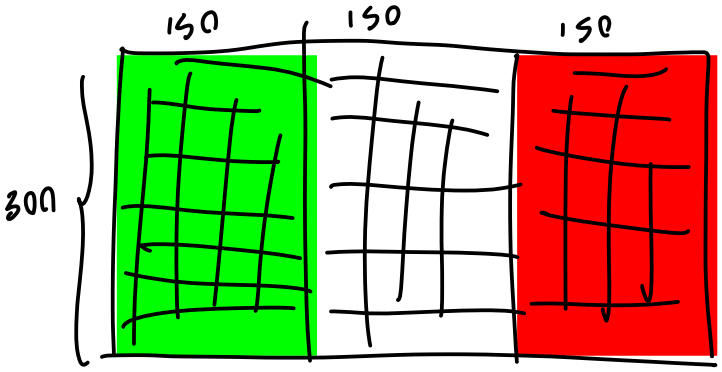
↑ very usefull to define masks on an image and other operations...

TODO: Display the italian flag

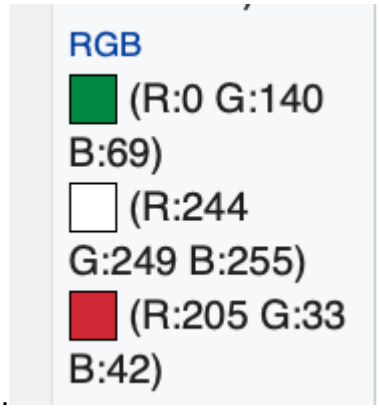
Using MATRIX

```
C = 300;
R = 150;

flag = zeros(R, C, 3);

% GREEN CHANNEL
g  = zeros(R, C);
g(:, 1 : C/3) = 255;
g(:, C/3 +1 : 2 * C /3) = 255;
g(:, 2 * C /3 + 1: C) = 0;

% BLUE CHANNEL
b  = zeros(R, C);
b(:, C/3 +1 : 2 * C/3) = 255;
```

```
% RED CHANNEL
r  = zeros(R, C);
r(:, 1 : C/3) = 0;
r(:, C/3 +1 : end) = 255;


flag(:,:,1) = r;
flag(:,:,2) = g;
flag(:,:,3) = b;

figure, imshow(flag)
```



Note: these are not exactly the official colors of the italian flag. If you want a better result, you can check for the official palette.

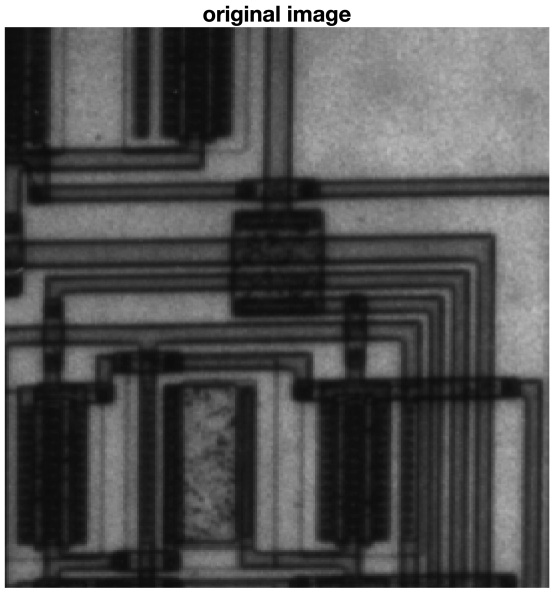


different way to manipulate image based on histogram

Histogram transformation

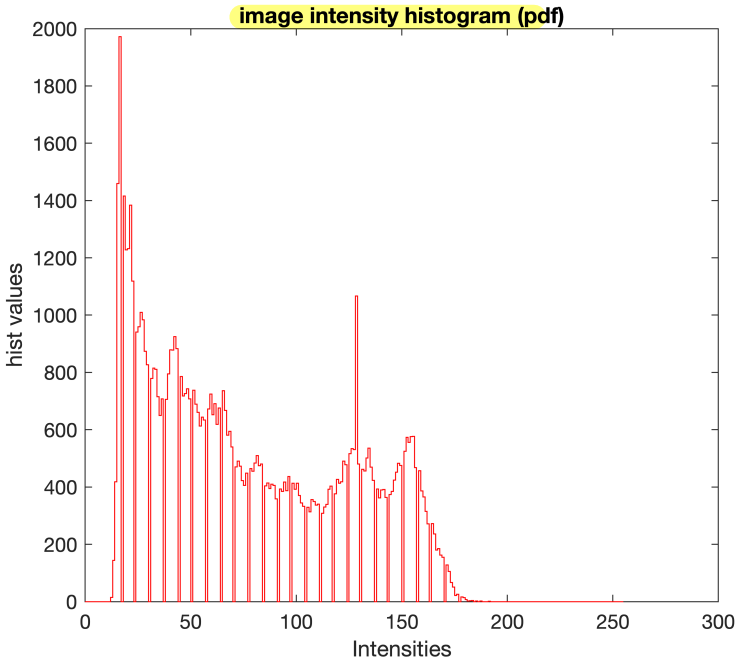
```
im = imread('circuit.tif');
figure, imshow(im), title('original image');
```

as we can take an image an pixel wise transformation, we can look whole content of an image → to get best looking image



usefull for over/under exposed image you need to fix...
↓
for better looking image...
equalize the histogram

```
% compute the histogram of all the intensities
hist_im = hist(im(:), [0 : 255]);
figure(5),
stairs([0 : 255], hist_im, 'r'), title('image intensity histogram (pdf)')
xlabel('Intensities');
ylabel('hist values');
```

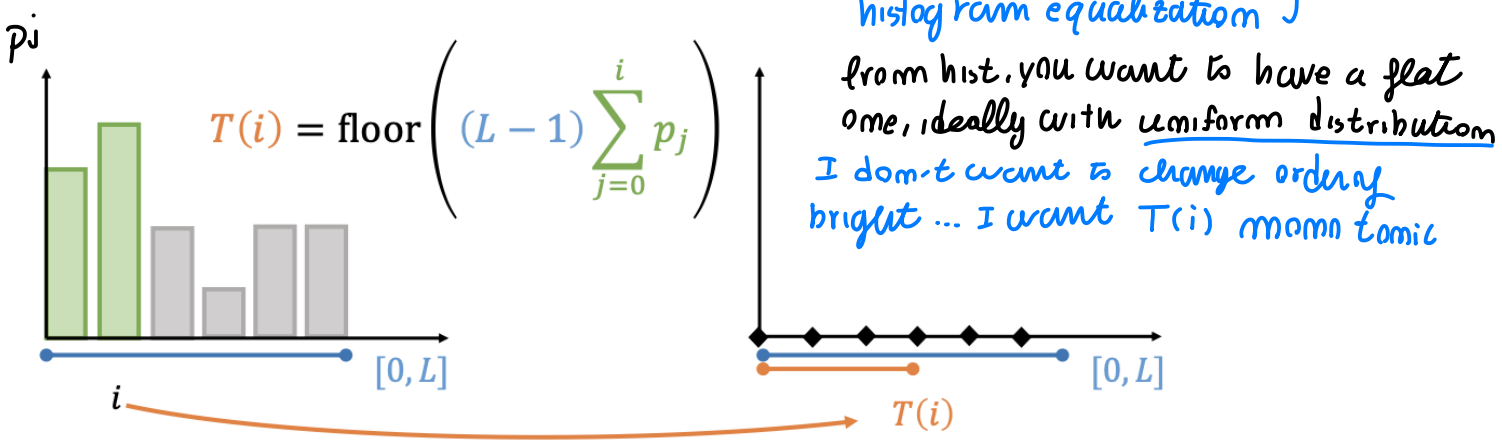


Histogram equalization

```
% the histogram can be computed also as
% imhist(im);
```

The histogram can be seen as the pdf of a RV corresponding to pixel realization histogram equalization consists in applying a monotonic transformation that brings this pdf towards a uniform distribution.

The trasformation is $T : i \mapsto \text{floor}((L-1)\text{CDF}(\text{hist}, i))$ where L are the available intensity levels and CDF is the cumulative density function of the histogram x is the intensity of a pixel: $\text{CDF}(\text{hist}, i) = \sum_{j=0}^{i} p_j$



histogram equalization ↑
from hist. you want to have a flat one, ideally with uniform distribution
I don't want to change ordering bright ... I want T(i) monotonic

```
cdf_im = cumsum(hist_im);
cdf_im = cdf_im/cdf_im(end)
```

```
cdf_im = 1×256
         0         0         0         0         0         0         0         0 ···
```

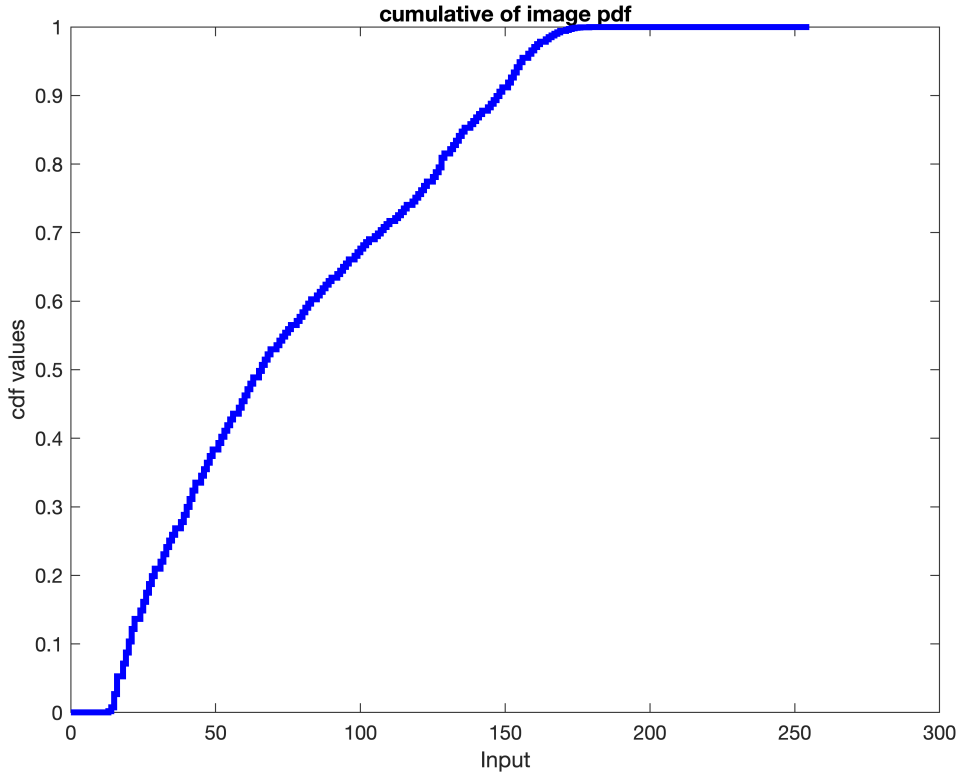
```
figure
stairs([0 : 255], cdf_im, 'b','LineWidth',3), title('cumulative of image pdf')
xlabel('Input');
ylabel('cdf values');
```

to distribute uniformly...
I look at how many pixels are brighter/darker than it...
I count, spread the mass on range, target uniform

$i \longrightarrow T(i) \rightarrow$ value as amount of pixels brighter/less than i
i in 0÷255, so you scale up (0÷1) * 255

and because intensity discrete → floor operation
mapping pixel to integer value

(transform pixel intensity by looking at histogram of image s.t better equalized)

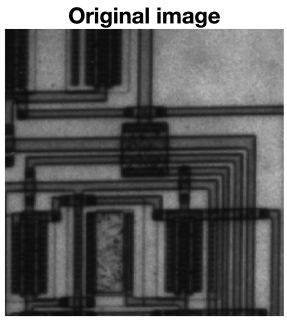


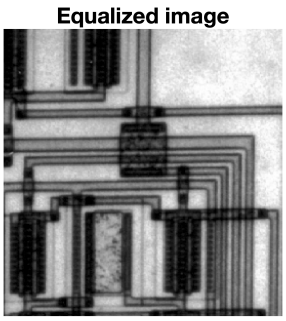
Mapping pixel intensities

```
map = floor(255 * cdf_im);
im_eq = map(im);

hist_im_eq = hist(im_eq(:), 0 : 255);


figure(8);
subplot(2,2,1);
imshow(im);
title('Original image');

subplot(2,2,2);
imshow(im_eq/255);
title('Equalized image');

subplot(2,2,3);
bar(hist_im);
title('Original image histogram');

subplot(2,2,4);
bar(hist_im_eq);
title('Equalized image histogram');
```

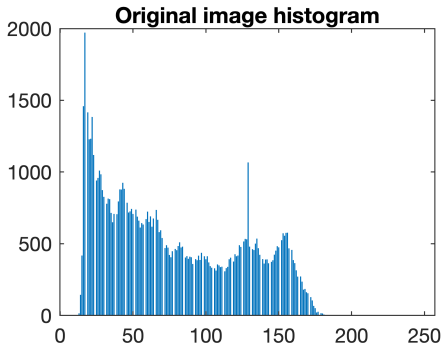
T(i)

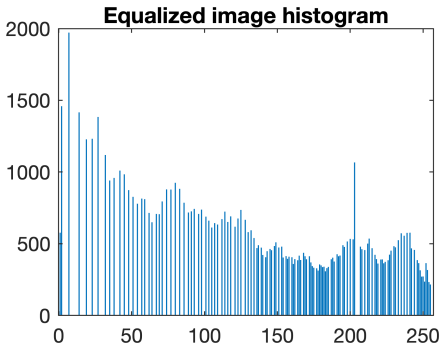
operation on image, by looking on histogram





DARK image ⇝ brighter figure, better details





histogram spread... equalized image, NOT really uniform because integer values (cdf of any is uniform)

you map pixels intensity

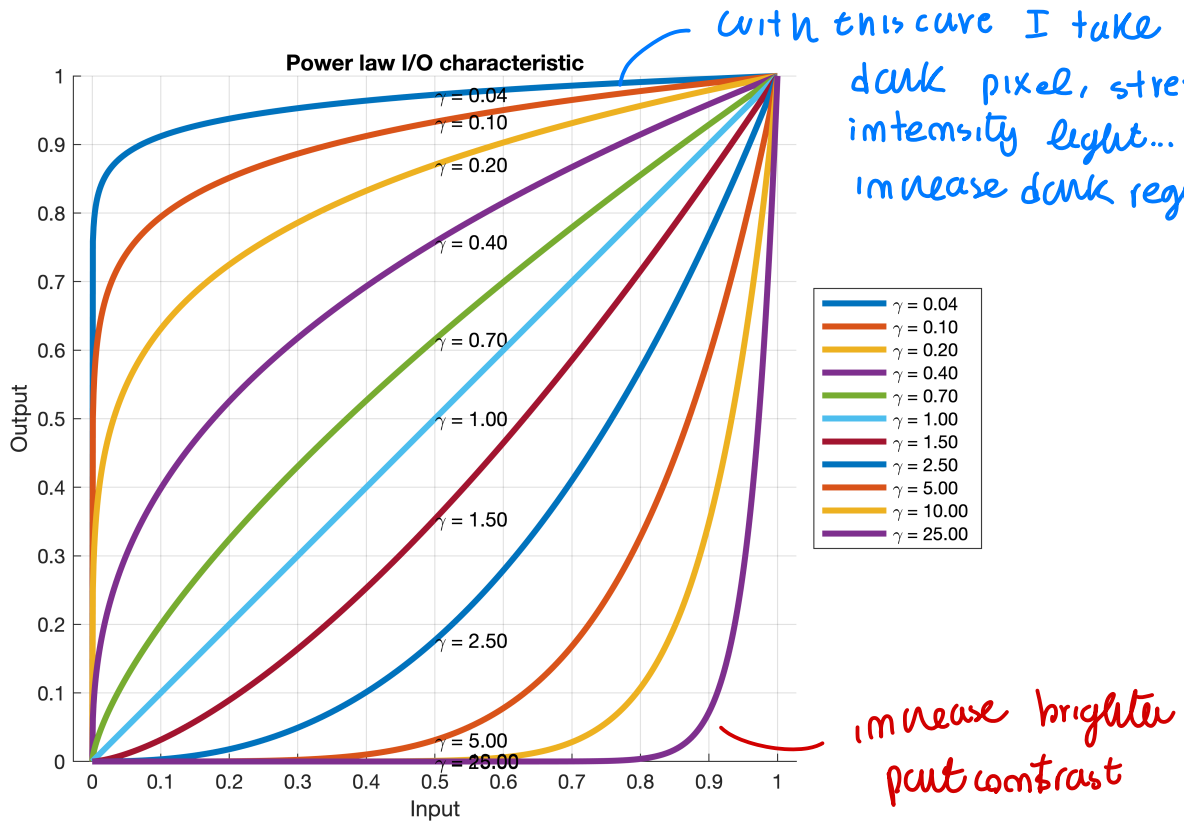
(NOT viceversa)

Gamma correction

```
im = imread('E1_data/image_Lena512.png');
figure;
cla
hold on;
x = 0:0.001:1;
for gamma = [.04 .1 .2 .4 .7 1 1.5 2.5 5 10 25]
    y = x.^gamma;
    plot(x,y,'DisplayName',sprintf('\\gamma = %.2f',gamma),'Linewidth',3);
    % display the text
    text(x(round(end/2)),y(round(end / 2)), sprintf('\\gamma = %.2f',gamma));
end
xlabel('Input');
ylabel('Output');
title('Power law I/O characteristic');
leg = legend('Location','eastoutside');
axis equal
grid on
hold off
```

⇝ power a certain image...

another way to transform image



with this curve I take dark pixel, stretchy intensity light... increase dark region contrast

increase brighter part contrast

simple image processing technique!

play with gammaVal and set a different value for the gamma Correction.

check which part of the image are being mostly affected by this transformation

```
imd = im2double(im);
gammaVal = 0.97
```

```
gammaVal = 0.9700
```

```
figure
subplot(1,2,1)
imshow(imd, [])
title('original image')
subplot(1,2,2)
imshow(imd.^gammaVal, [])
title(sprintf('Gamma corrected image using \\gamma = %.2f', gammaVal));
```

original image

Gamma corrected image using γ = 0.97

GEOMETRY : introduction of projectum geometry ...

Homogenous coordinates: lines, angles and vanishing points

Credits: Giacomo Boracchi October 1st, 2018

Edits: Luca Magri, September 2024, for comments and suggestions write to luca.magri@polimi.it

Goal: familiarize with homogeneous coordinates

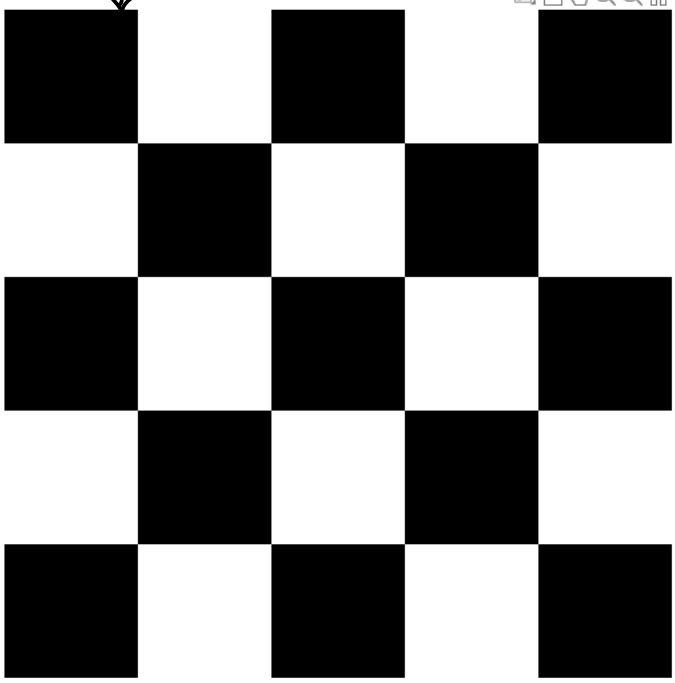
```
clear
close all
clc
%font size
FNT_SZ = 28;
```

```
% load a checkerboard image
I = imread('E1_data/checkerboard.png');
```

manually select points

```
figure(1), imshow(I);
hold on;
[x, y] = getpts();
```

allow to manually select on the image to define points and save it as $a = (x, y, w)$, with $w = 1$



↑

```
% save points in the homogeneous cooridnate. It is enough to set to 1 the third component
a = [x(1); y(1); 1];
b = [x(2); y(2); 1];
c = [x(3); y(3); 1];
d = [x(4); y(4); 1];
```

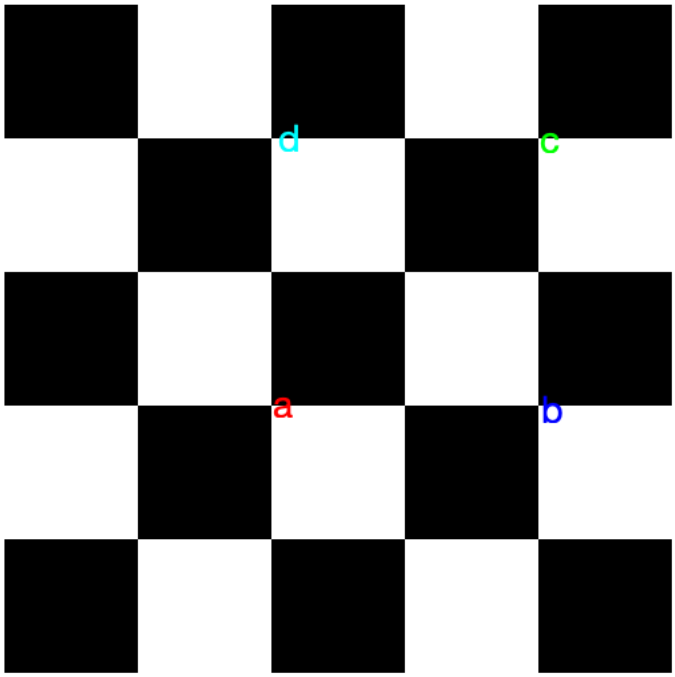
homogeneous) NOT 00 point! so w = 1 is OK!

Columnwise

Draw the points over the image

```
text(a(1), a(2), 'a', 'FontSize', FNT_SZ, 'Color', 'r')
text(b(1), b(2), 'b', 'FontSize', FNT_SZ, 'Color', 'b')
text(c(1), c(2), 'c', 'FontSize', FNT_SZ, 'Color', 'g')
text(d(1), d(2), 'd', 'FontSize', FNT_SZ, 'Color', 'c')
```

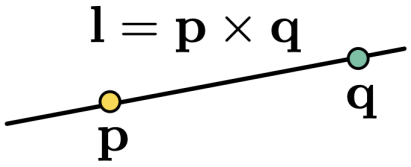
↳ display the points on image ...



given 4 points... I can extract lines by using cross product

Compute the parameters of a few lines equations

lines are represented as 3D homogeneous vector collecting the coefficients of the line equation. Lines are points in the "dual plane".

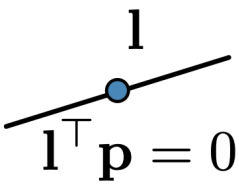


cross product define line! of two point (duality)

```
lab = cross(a, b); % this is the reference
lad = cross(a, d); % orthogonal to lab
lac = cross(a, c); % 45 degrees with lac
lcd = cross(c, d); % parallel to lab
```

{ a, b, c, p, q : points
{ l, m, n : lines (convention)

Check some incidence relations



both points, line are 3D vectors, represented in same way! depends on context

POINTS, LINES ~ same representation

they should be zero as $a \in \ell_{ab}$ and $c \in \ell_{cd}$

the incidence relation correspond to lcd(1)*c(1) + lcd(2)*c(2) + lcd(3)*c(3) = 0

```
a' * lab % this should be zero when a \in lab
```

because $a \in l_{ab}$ should be $a^T l_{ab} \simeq 0$

```
ans = -7.2760e-12
```

```
c' * lcd %
```

```
ans = 3.6948e-12
```

intersect the lines with the image borders

intersect using border equation... x = 1, first column

The general equation of a line is $\ell : ax + by + c = 0$. The equation of the "first column" $c1 : x = 1 \Rightarrow a = 1, b = 0, c = -1$ (remember that the first coordinate in matlab is the row, the second the column.

```
c1 = [1; 0; -1]; % parameters of left-most column
c500 = [1; 0; -500]; % parameters of right-most column

r1 = [0; 1; -1]; % parameters of top-most row
r500 = [0; 1; -500]; % parameters of bottom row
```

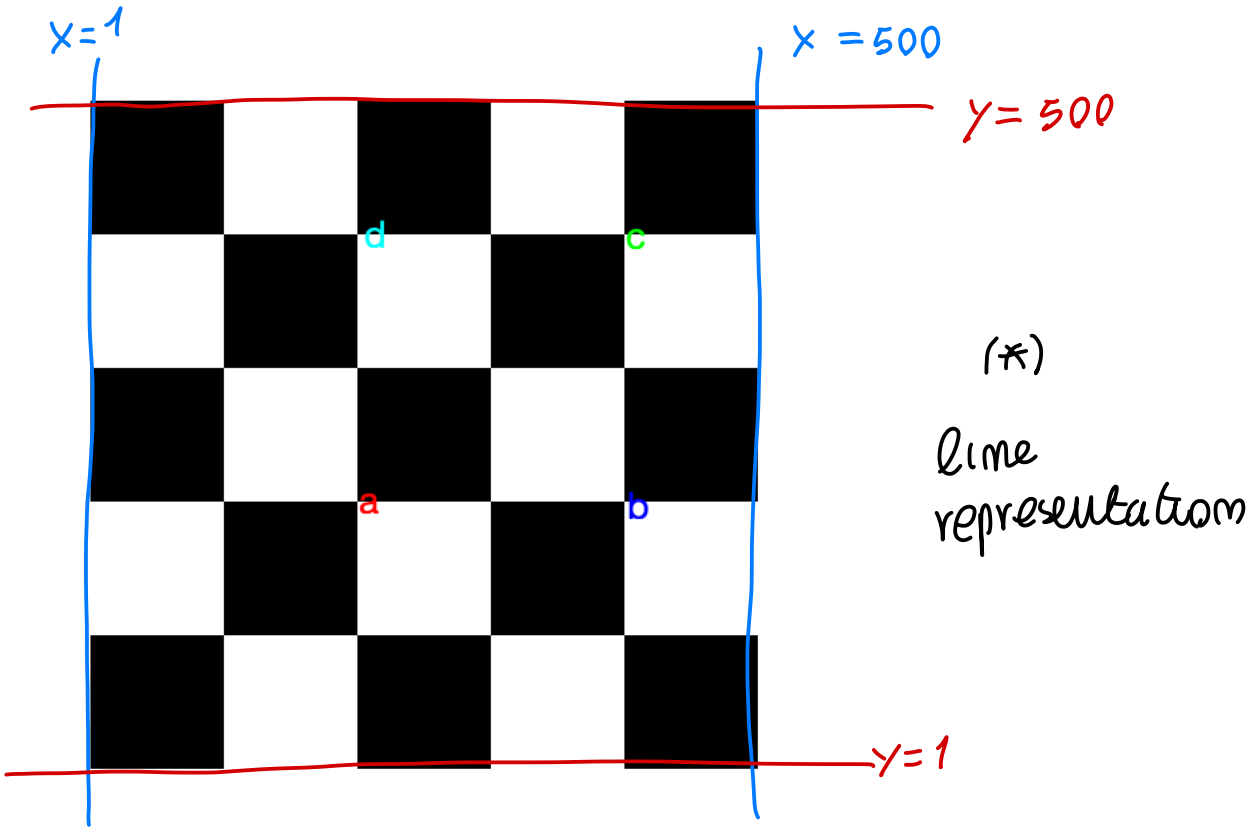
as 3 vector of coordinates, $ax + by + c = 0 \leftarrow x = \alpha$ take $1x + 0y + (-\alpha) = 0$

$a = 1, b = 0, c = -\alpha$

homogeneous represent.

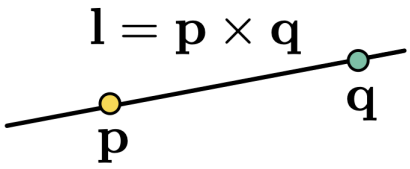
The intersection can be easily computed as the dot product between the representations of lines

$x = \alpha$ this is a vertical line! all line in homogeneous coord



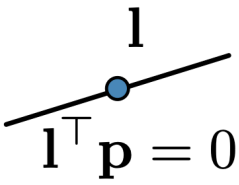
Compute the parameters of a few lines equations

lines are represented as 3D homogeneous vector collecting the coefficients of the line equation. Lines are points in the "dual plane".




```
lab = cross(a, b); % this is the reference
lad = cross(a, d); % orthogonal to lab
lac = cross(a, c); % 45 degrees with lac
lcd = cross(c, d); % parallel to lab
```

Check some incidence relations



they should be zero as $a \in \ell_{ab}$ and $c \in \ell_{cd}$

the incidence relation correspond to lcd(1)*c(1) + lcd(2)*c(2) + lcd(3)*c(3) = 0

```
a' * lab % this should be zero when a \in lab
```

```
ans = -7.2760e-12
```

```
c' * lcd %
```

```
ans = 3.6948e-12
```

intersect the lines with the image borders

The general equation of a line is $\ell : ax + by + c = 0$. The equation of the "first column" $c1 : x = 1 \Rightarrow a = 1, b = 0, c = -1$ (remember that the first coordinate in matlab is the row, the second the column.

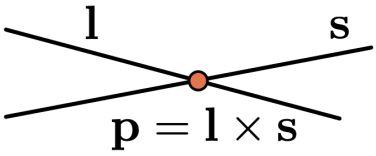
(*)

```
c1 = [1; 0; -1]; % parameters of left-most column
c500 = [1; 0; -500]; % parameters of right-most column

r1 = [0; 1; -1]; % parameters of top-most row
r500 = [0; 1; -500]; % parameters of bottom row
```

x = a

y = a

The intersection can be easily computed as the dot product between the representations of lines



take simply intersection now that I have lines...

```
% compute the intersection between lab and the first column
x1 = cross(c1, lab) % point in homogeneous coordinates
```

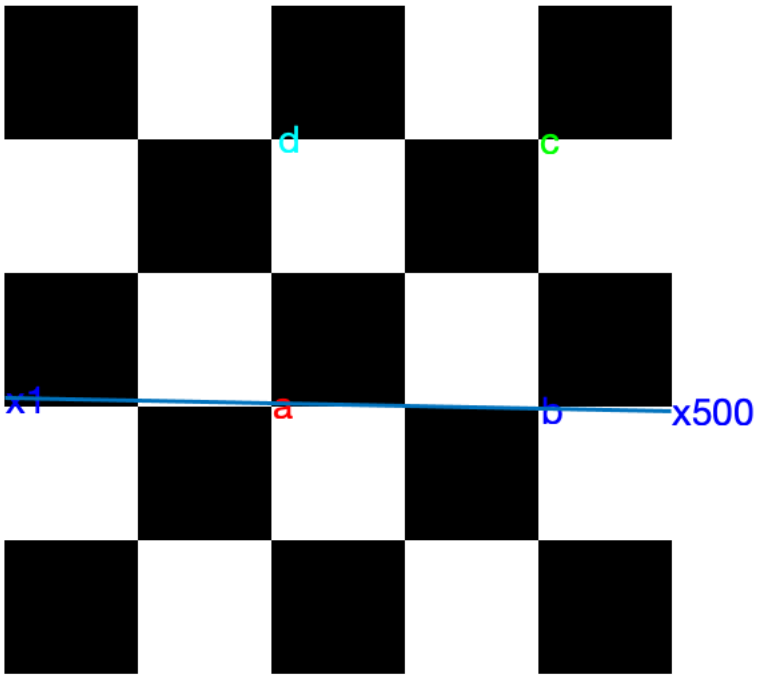
```
x1 = 3×1

10^4 ×
    0.0201
    5.9098
    0.0201
```

In order to plot x1 it is necessary to get to non homogenous (cartesian) coordinates.

Bear in mind that x_1 and λx_1 are equivalent in P^n for all $\lambda \neq 0$.

To draw the point in the image plane, we need to take the point corresponding to x1 having third component equal to 1.

```
x1 = x1/x1(3);
text(x1(1), x1(2), 'x1', 'FontSize', FNT_SZ, 'Color', 'b')

% do the same with the right most column
x500 = cross(c500, lab);
x500 = x500 / x500(3);
text(x500(1), x500(2), 'x500', 'FontSize', FNT_SZ, 'Color', 'b')
plot([x1(1), x500(1)],[x1(2), x500(2)], 'LineWidth', 3)
```

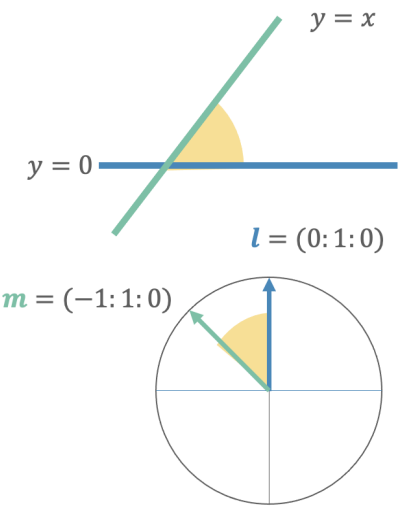


from cross produc I get this (a,b) are NOT completely horizontal

Compute the angles on these images

that is given by the formula $\cos(\theta) = \frac{l_1 m_1 + l_2 m_2}{\sqrt{||l_{1:2}||^2 * ||m_{1:2}||^2}}$

Just taking normal and compute angle of normal to line




```
l = lab;
m = lac
```

```
m = 3×1
10^4 ×
    0.0198
    0.0200
   -9.9398
```

```
cosTheta = (l(1) * m(1) + l(2) * m(2))/sqrt(sum(l(1:2).^2)*sum(m(1:2).^2));
theta = acosd(cosTheta)
```

```
theta = 45.8521
```

almost orthogonal...

Verify the property that any linear combination of a,b belongs to lab

we can verify property!

```
lambda = rand(1);
mu = 1- lambda; % this is a convex combination. So points will be in between the two. The result holds for any combination

p = lambda * a + mu * b;
p' * lab
```

given a,b any $p' = \lambda a + \mu b \in lab$

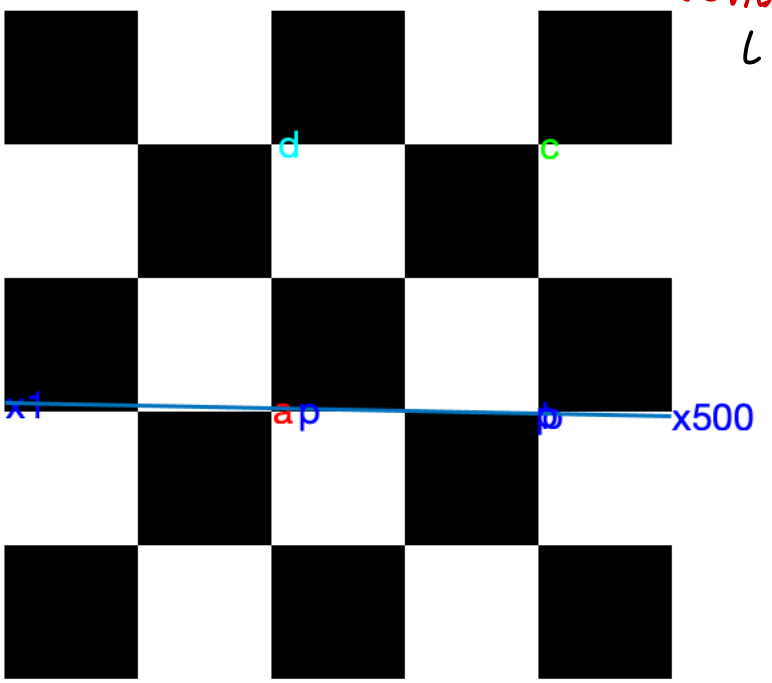
combination of a,b

```
ans = -1.2847e-11
```

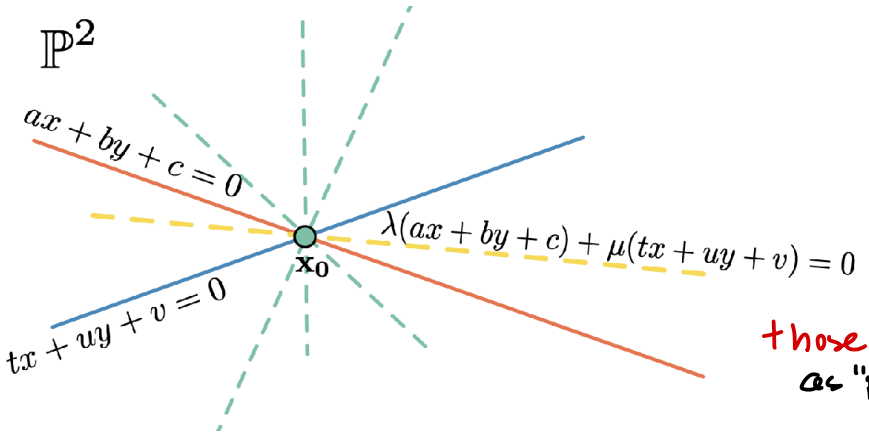
divide by p(3)=w to draw point in the CARTESIAN PLANE!

```
p = p/p(3);
text(p(1), p(2), 'p', 'FontSize', FNT_SZ, 'Color', 'b')
```

dehomogeneity



Dual property: any linear combination of two lines l1 and l2 passes through their intersection



also for DUALITY
points ↔ lines

those lines can be represented as "points in the dual plane"

```
lambda = rand(1);
mu = 1- lambda; % this is a convex combination. So points will be in between the two. The result holds for any combination

x0 = cross(lab, lad)
```

and any linear combination pass through same point! (dual of before!)

```
x0 = 3×1
10^7 ×
   -0.8043
   -1.1924
   -0.0040
```

```
% linear combination of lines
l = lambda * lab + mu * lad
```

```
l = 3×1
10^4 ×
    0.0123
    0.0078
   -4.7904
```

```
% check aInt belongs to l
x0'*l
```

```
ans = -3.5763e-07
```

NOT exactly 0 due to image noise

yes! belongs to pencil

What you get if you intersect parallel lines?

```
vac = cross(lab, lcd)
```

cross product of parallel lines is expected to have $w \simeq 0$ because of point to infinity!

```
vac = 3×1
10^6 ×
   -7.7234
    0.0177
    0.0006
```

```
vac = vac / vac(3) % look at this point, isn't it strange? It is a point at the infinity!
```

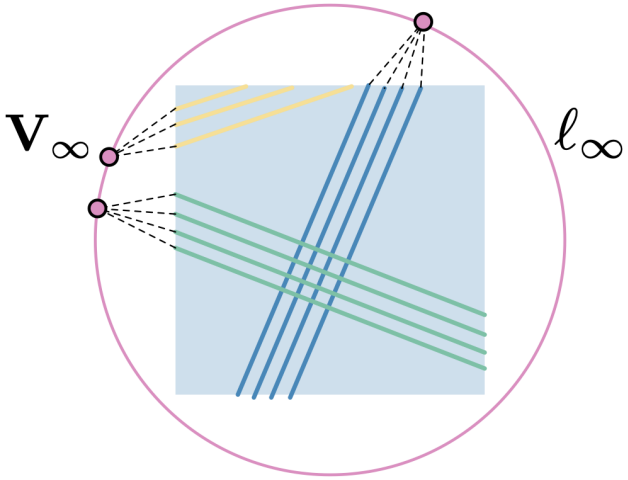
```
vac = 3×1
10^4 ×
   -1.3248
    0.0030
    0.0001
```

```
vab = cross(r1, r500)
```

```
vab = 3×1
  -499
     0
     0
```

```
vad = cross(c1, c500)
```

```
vad = 3×1
     0
   499
     0
```



$$\mathbb{P}^2 = \mathbb{E}^2 \cup \ell_\infty$$

A simple geometrical construction with 2D homogeneous coordinates

Credits: Giacomo Boracchi October 1st, 2018

Edits: Luca Magri, September 2023, for comments and suggestions write to luca.magri@polimi.it

Our goal is to complete the drawing of the **isometric projection** http://en.wikipedia.org/wiki/Isometric_projection of a cuboid when the points corresponding to a vertex and to three vertices adjacent to it are given as input.

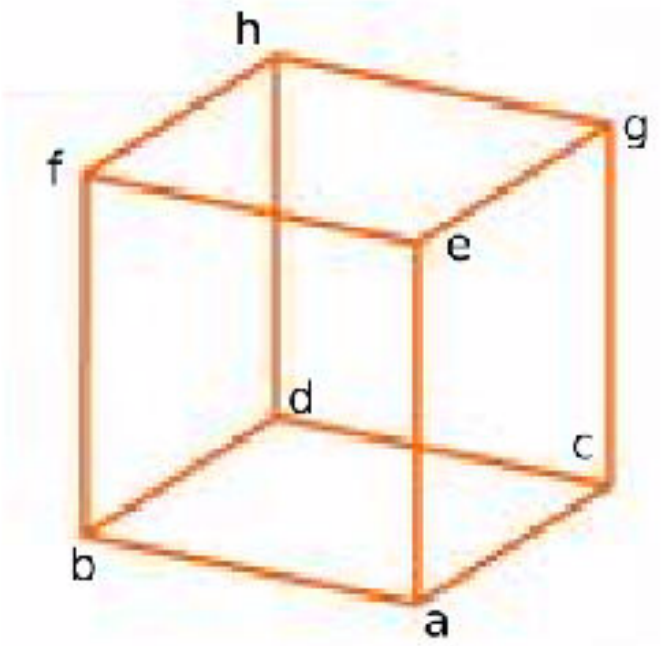
The geometrical construction only requires to find lines parallel to other lines, find lines passing through two points, and intersect lines, which is easily achieved in homogeneous coordinates.

```
clear
close all
clc
FNT_SZ = 28;
```

Naming

We will name the vertices as in the following image

```
figure(1), imshow(imread('E1_data/simplecube-letters.png'));
```



given 4 points in the image, draw the wall with all edges

Data entering

the user should click on a vertex of the cuboid, then on the three adjacent vertices

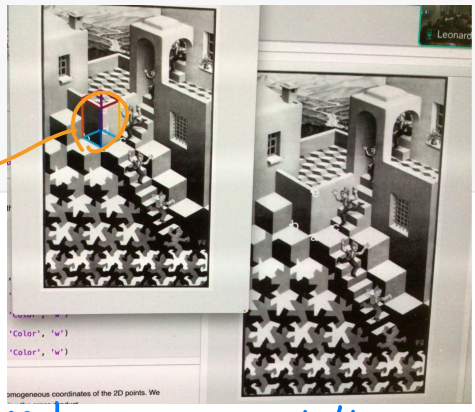
```
figure(2),
imshow(imread('E1_data/buildingSmall.png'))

hold on;
[x y]=getpts
```

select the points on image

```
x = 4×1
   443.0000
   356.0000
   512.0000
   447.0000
y = 4×1
   566.0000
   543.0000
   444.0000
   316.0000
```

very simple to do, just need to compute some parallel lines and intersection

draw all the cube from the 4 vertices



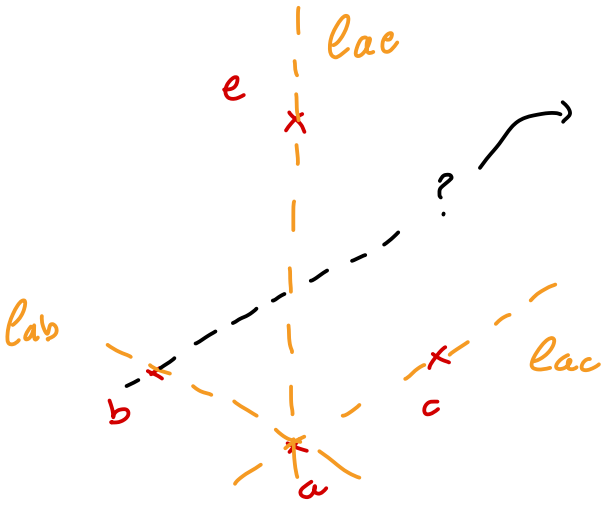
we are just using incident relationship (also parallelism is an incidence relation)

you just need one vector and three neighbouring

```
plot(x,y,'.w','MarkerSize',12, 'LineWidth', 3); % plots points clicked by user with re
a=[x(1) y(1) 1]';
text(a(1), a(2), 'a', 'FontSize', FNT_SZ, 'Color', 'w')
b=[x(2) y(2) 1]';
text(b(1), b(2), 'b', 'FontSize', FNT_SZ, 'Color', 'w')
c=[x(3) y(3) 1]';
text(c(1), c(2), 'c', 'FontSize', FNT_SZ, 'Color', 'w')
e=[x(4) y(4) 1]';
text(e(1), e(2), 'e', 'FontSize', FNT_SZ, 'Color', 'w')
```

given the 4 points selected...

draw lines easily by points cross product



how to find this line?

I need the intersection between lac and line @ infinity

All points @ infinity belongs to the line at infinity

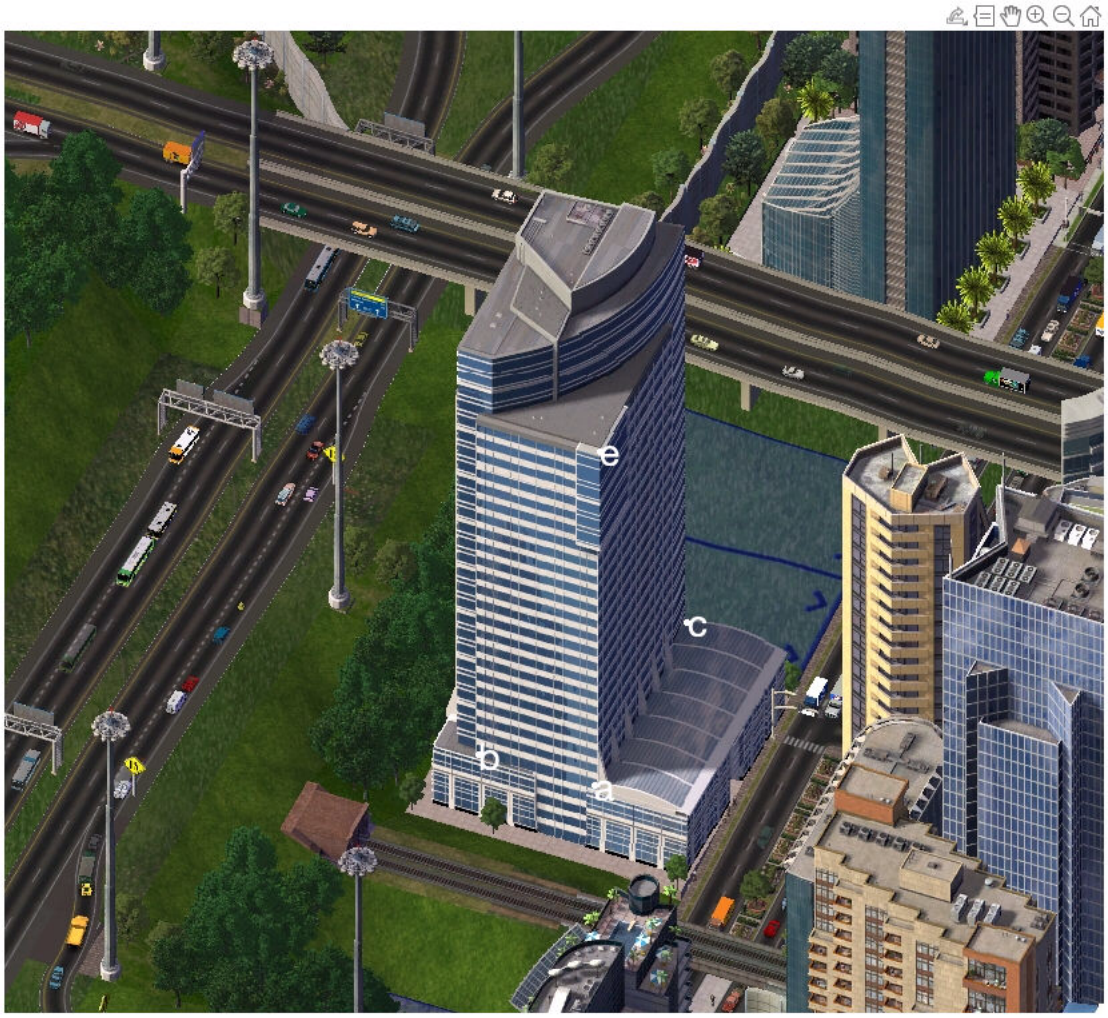
$ax+by+cw=0$

$cw=0$ all points (x,y) belongs to this line

$(0\ 0\ 1)'$

$\Downarrow$

line at infinity



Finding some lines

Now variables a, b, c and d are 3-vectors containing the homogeneous coordinates of the 2D points. We need to find the lines passing through couples of points, using the cross product

```
lab = cross(a,b)
```

```
lab = 3×1
10^4 ×
    0.0023
   -0.0087
    3.9053
```

```
lac = cross(a,c)
```

```
lac = 3×1
10^4 ×
    0.0122
    0.0069
   -9.3100
```

```
lae = cross(a,e)
```

```
lae = 3×1
10^5 ×
    0.0025
    0.0000
   -1.1301
```

Check the incidence equation

`lab`, `lac` and `lae` now contain the homogeneous represenation of the three lines ab, ac and ae. We can easily check that the lines actually contain the points by using the incidence relation:

```
lab'*a
```

```
ans = -1.8190e-11
```

```
lab'*b
```

```
ans = -1.9099e-11
```

```
lac'*a
```

```
ans = 1.4552e-11
```

```
lac'*c
```

```
ans = 1.4552e-11
```

```
lae'*a
```

```
ans = -1.4552e-11
```

```
lae'*e
```

```
ans = -1.4552e-11
```

Parallelism

We now need to compute the line parallel to ab and passing through c. In order to do this, we create the line at infinity:

```
linf=[0 0 1]';
```

then, we find the directions of segments by ab, ac and ae intersecting their lines with the line at infinity

```
dab=cross(lab,linf)
```

```
dab = 3×1
  -87.0000
  -23.0000
         0
```

```
dac=cross(lac,linf)
```

```
dac = 3×1
   69.0000
 -122.0000
```

$[0\ 0\ 1]'$ is the line at infinity, any point is part of it!

↑ line in projective plane representing cell (x,y) points as line at infinity

w ~ third projective coordinate

this represent line at infinity!

↑

transform projective line as linear equation in projective space

$0x + 0y + 1w = 0 \Rightarrow [0\ 0\ 1]' \equiv$ LIVE AT ∞ in homogeneous coordinate

to take the intersection of a line l_{ab} with $l = [0\ 0\ 1]$ i just need to take the CROSS-PRODUCT between $l_{ab} = [a\ b\ c]$ and $[0\ 0\ 1]$

$d_{ab} = \text{cross}(l_{ab}, l_{inf})$ ~ intersection with line @ infinity

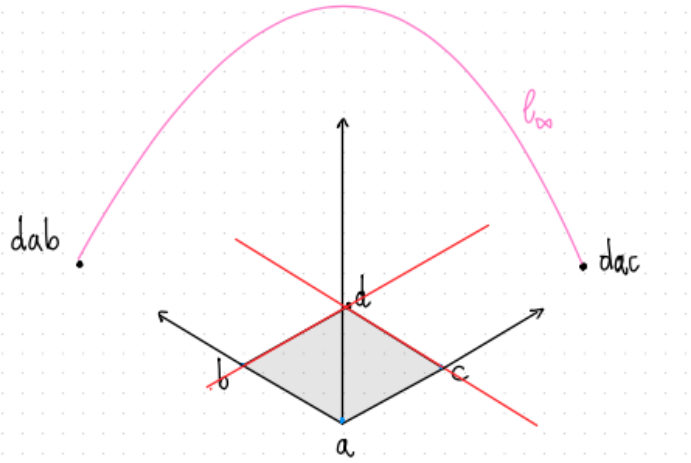
then I can get the intersection d of bd, dc from those lines

```
     0
```

```
dae=cross(lae,linf)
```

```
dae = 3×1
    4.0000
 -250.0000
         0
```

dab, dac and dae are **points at the infinity**, which represent **directions**. All lines with a given direction pass through the corresponding point at the infinity. We can then find the lines containing segments bd and cd.




```
lbd=cross(b,dac);
lcd=cross(c,dab);
```

point d is now found by just intersecting `lbd` and `lcd`

```
d=cross(lbd,lcd);
```

we normalize d's coordinates so that we can read its cartesian coordinates in `d(1)` and `d(2)`. We plot the point with a blue circle.

```
d=d/d(3);
plot(d(1),d(2),'.b','MarkerSize',30);
text(d(1), d(2), 'd', 'FontSize', FNT_SZ, 'Color', 'w')
```

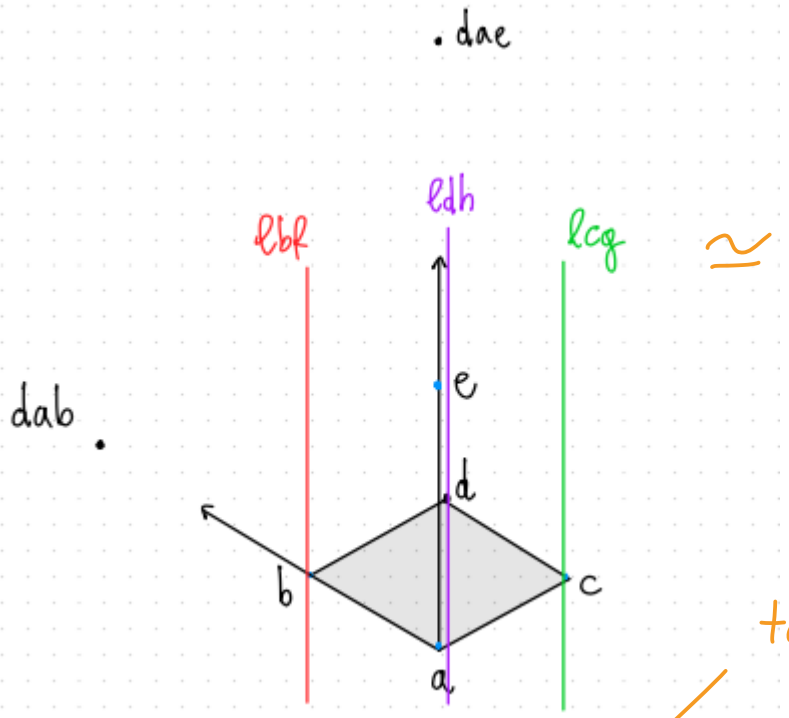
to get d, remember to divide by w to take cartesian coordinate

Finding remaining points

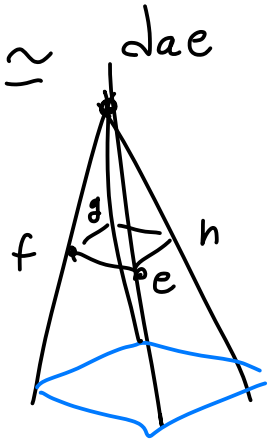
The rest of the procedure is straightforward, and follows exactly the same technique.

We first find vertical lines, that pass trough dae

↳ by taking the intersection by ones parallel to ae toward others ... again with l_∞



≃

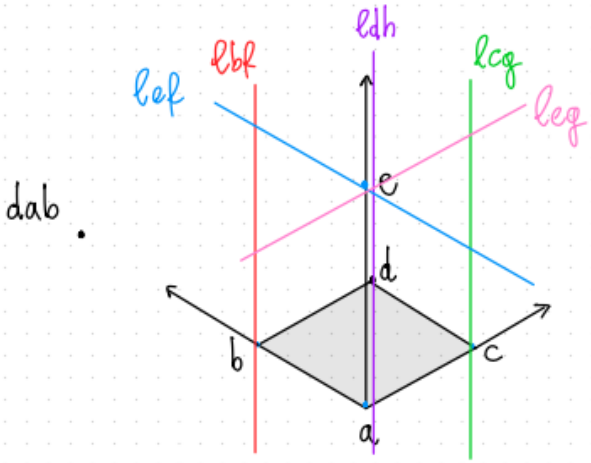


IN PERSPECTIVE
parallel lines meet in a point!

taking intersection of ae with l_∞ to get dae, point at ∞ of parallel lines, then make all others pass to it!

```
lbf=cross(b,dae);
lcg=cross(c,dae);
ldh=cross(d,dae);
```

We then find the lines incident to e parallel to lab and lac. Being parallel means having the same intersection at l_∞



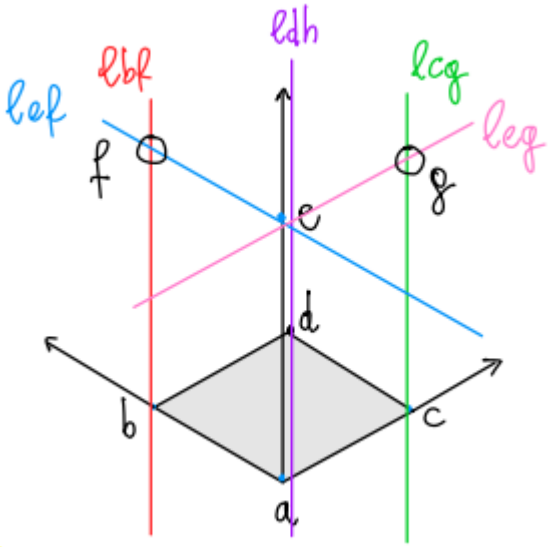
dac ≃ //ac line

and point/line duality holds! do same procedure
{ point → line
line → point

I need point at ∞ of ab to get lef as // lab

```
lef=cross(e,dab);
leg=cross(e,dac);
```

dab ~ represents the direction of ab : point at ∞ representing as direction

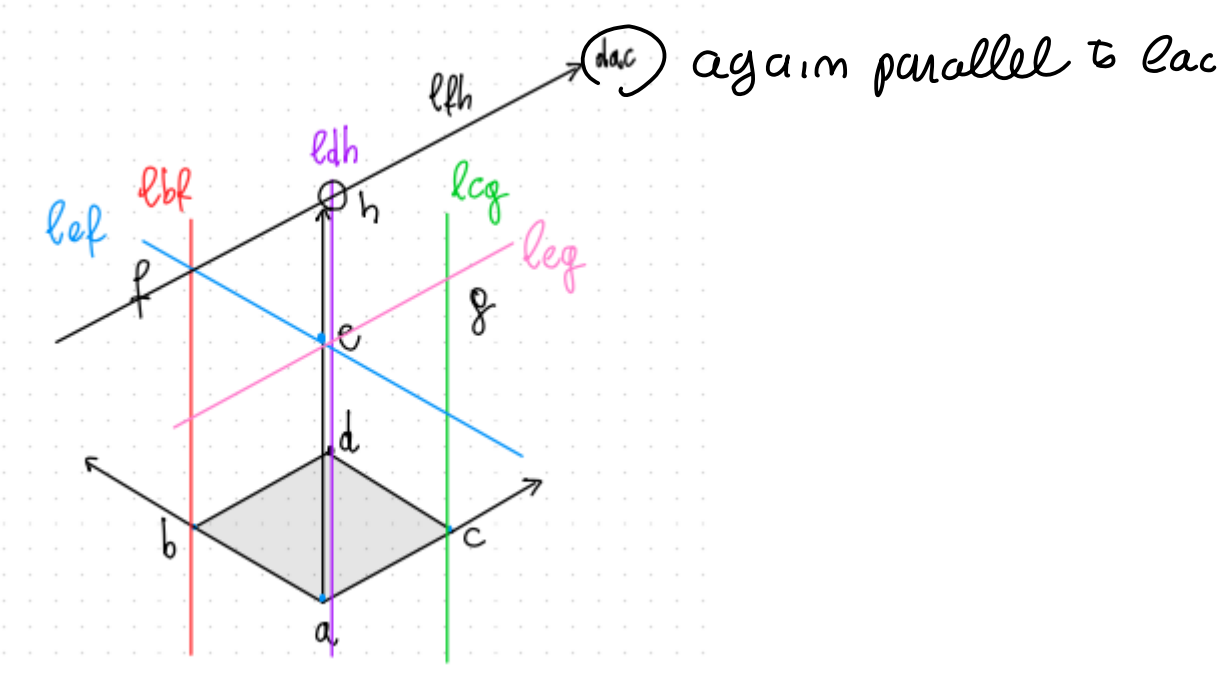


Now we can find the intersections

by crossing those lines!

```
f=cross(lbf,lef);
g=cross(lcg,leg);
```

Now the line passing through f and parallel to lac. Hence we can get h.



use lac direction "dac" to compute h as intersection

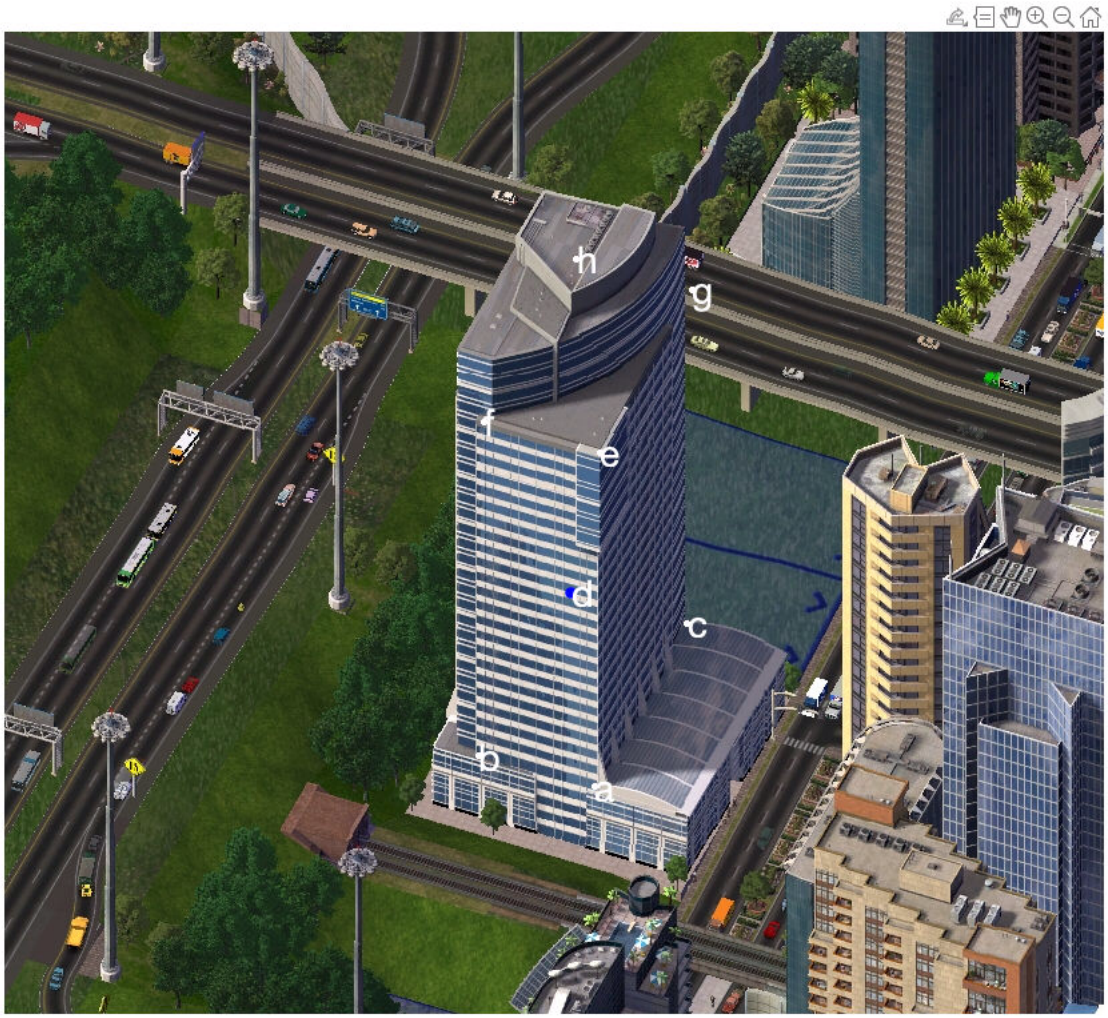
```
lfh=cross(f,dac);
h=cross(lfh,ldh);
```

Remember that points are expressed in homogenous coordinates, in order to plot them we have to convert them to cartesian.

```
f=f/f(3);
```

```
g=g/g(3);
h=h/h(3);
```
} properly divide by last coordinates... } then PLOT it!

```
plot(f(1), f(2),'.w','MarkerSize',12, 'LineWidth', 3); % plots points clicked by user w
text(f(1), f(2), 'f', 'FontSize', FNT_SZ, 'Color', 'w')
plot(g(1), g(2),'.w','MarkerSize',12, 'LineWidth', 3); % plots points clicked by user w
text(g(1), g(2), 'g', 'FontSize', FNT_SZ, 'Color', 'w')
plot(h(1), h(2),'.w','MarkerSize',12, 'LineWidth', 3); % plots points clicked by user w
text(h(1), h(2), 'h', 'FontSize', FNT_SZ, 'Color', 'w')
```



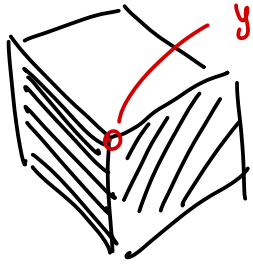
using f∞ and cross product we can derive MANY properties!

Drawing

we can now finally draw the cube.

```
myline=[a';b';d';c';a'];
line(myline(:,1),myline(:,2),'LineWidth',5);
myline=[e';f';h';g';e'];
line(myline(:,1),myline(:,2),'LineWidth',5);
myline=[a';e'];
line(myline(:,1),myline(:,2),'LineWidth',5);
myline=[b';f'];
```

If I had only 3 vertex, to obtain the CUBE can be derived using light in the image to derive the last vertex... DIFFERENT problem...



you find edges, using FILTER

⇓

imagine an image as a surface... as GRAY SCALE image, each pixel of surface you can define gradient and accordyly to how fast gradient changes, you can find edges as region of DISCONTINUITY to get vertex

(different exercise...) ⟵

⟵ procedure valid in ISOMETRIC image

↓

In PERSPECTIVE images, NO more parallel lines, this procedure does NOT work anymore

image of l_∞ is NO more $(0\ 0\ 1)'$ when perspective image.... it is moved somewhere else!

l_∞ projected in some position

IF ORTHOGRAPHIC image. l_∞ projected into l_∞ (holds before)

OTHERWISE l_∞ projected somewhere else!

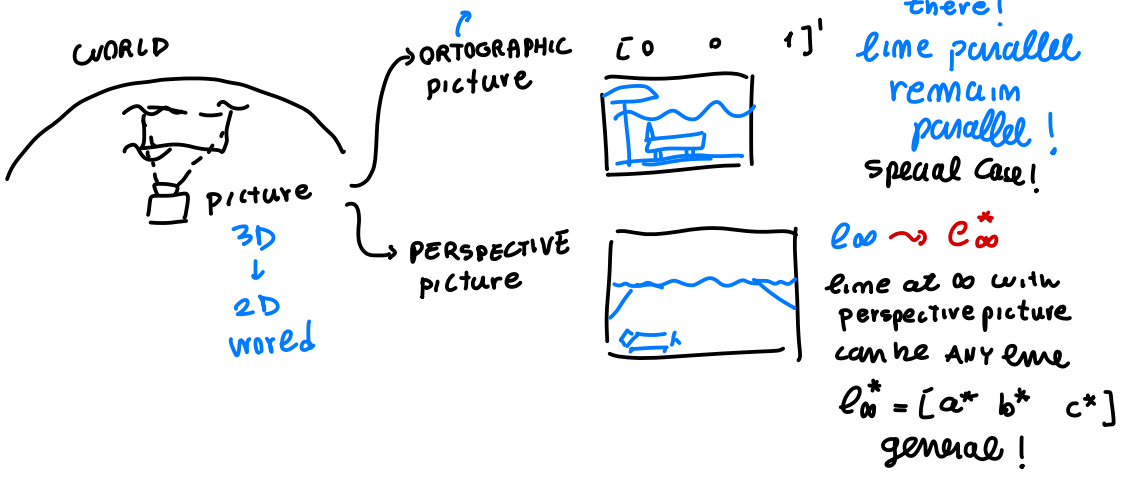
{ like when you take a picture of the sea you see the horizon even if at ∞, you project it in the middle!

⇒ taking a picture of a PLANE, you take all points of projective plane ↳ plane has $l_\infty = [0 \ 0 \ 1]'$

BUT if ORTHOGRAPHIC picture you get an image as image plane → l_∞' → $[0 \ 0 \ 1]$ again!

In general, a PERSPECTIVE picture map $l_\infty \to l_\infty^*$ new line@ ∞ ...

ex: as when we take picture of the sea, in which you map l_∞ in the middle, NOT in $[0 \ 0 \ 1]'$ any more



on the BEACH..

image plane is the sea...

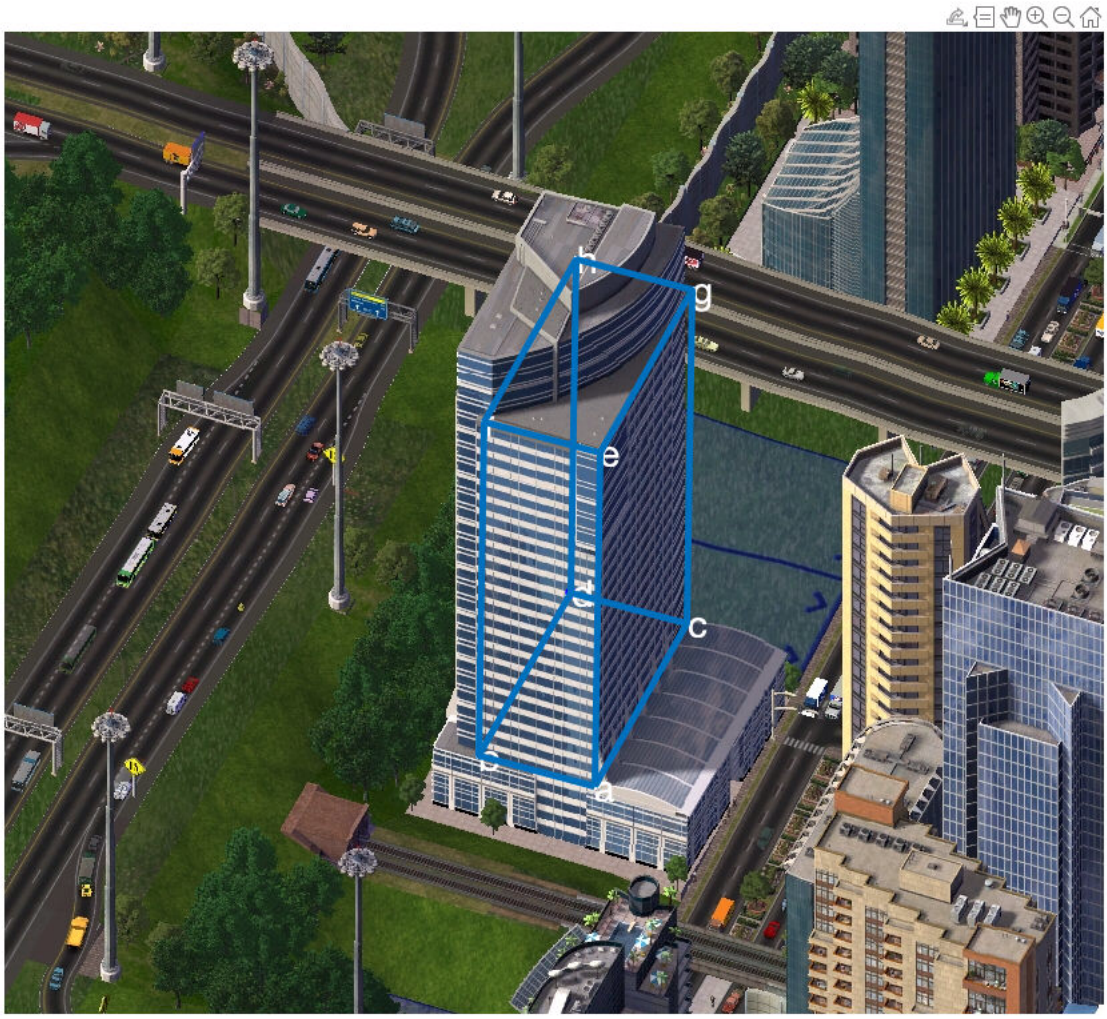
in the picture the horizon line is in the middle!
(horizontal vared plane)

in certain condition perspective camera ~ orthographic

Good approx under some conditions

always distorsion introduced... neither exactly perspective camera!

```
line(myline(:,1),myline(:,2),'LineWidth',5);
myline=[c';g'];
line(myline(:,1),myline(:,2),'LineWidth',5);
myline=[d';h'];
line(myline(:,1),myline(:,2),'LineWidth',5);
hold off
```



Notes

which vertices you click is not important: the only requirement is that the first vertex is adjacent to the other three. also, note that the algorithm works, more generally, on parallelepipeds.