# *Machine Vision* **TOOLBOX**

Release 2 for use with MATLAB

**Peter I. Corke**

Peter I. Corke

# Preface

## 1 Introduction

The Machine Vision Toolbox (MVT) provides many functions that are useful in machine vision and vision-based control. It is a somewhat eclectic collection reflecting the author's personal interest in areas of photometry, photogrammetry, colorimetry. It includes over 90 functions spanning operations such as image file reading and writing, acquisition, display, filtering, blob, point and line feature extraction, mathematical morphology, homographies, visual Jacobians, camera calibration and color space conversion. The Toolbox, combined with Matlab and a modern workstation computer, is a useful and convenient environment for investigation of machine vision algorithms. For modest image sizes the processing rate can be sufficiently "real-time" to allow for closed-loop control. Focus of attention methods such as dynamic windowing (not provided) can be used to increase the processing rate. With input from a firewire or web camera (support provided) and output to a robot (not provided) it would be possible to implement a visual servo system entirely in Matlab.

An image is usually treated as a rectangular array of scalar values representing intensity or perhaps range. The matrix is the natural datatype for Matlab and thus makes the manipulation of images easily expressible in terms of arithmetic statements in Matlab language. Many image operations such as thresholding, filtering and statistics can be achieved with existing Matlab functions. The Toolbox extends this core functionality with M-files that implement functions and classes, and mex-files for some compute intensive operations. It is possible to use mex-files to interface with image acquisition hardware ranging from simple framegrabbers to robots. Examples for firewire cameras under Linux are provided.

The routines are written in a straightforward manner which allows for easy understanding. Matlab vectorization has been used as much as possible to improve efficiency, however some algorithms are not amenable to vectorization. If you have the Matlab compiler available then this can be used to compile bottleneck functions. Some particularly compute intensive functions are provided as mex-files and may need to be compiled for the particular platform. This toolbox considers images generally as arrays of double precision numbers. This is extravagant on storage, though this is much less significant today than it was in the past.

This toolbox is not a clone of the Mathwork's own Image Processing Toolbox (IPT) although there are many functions in common. This toolbox predates IPT by many years, is open-source, contains many functions that are useful for image feature extraction and control. It was developed under Unix and Linux systems and some functions rely on tools and utilities that exist only in that environment.

### 1.1 How to obtain the Toolbox

The Machine Vision Toolbox is available subject to the License Agreement from the Toolbox home page at

http://www.petercorke.com

The files are available in either gzipped tar format (.gz) or zip format (.zip). The web page requests some information from you regarding such as your country, type of organization and application. This is just a means for me to gauge interest and to help convince myself that this is a worthwhile activity.

## 2 Support

No support is provided. The author is happy to correspond with people who have found genuine bugs or deficiencies, and to accept contributions for inclusion in future versions of the toolbox, and you will be suitably acknowledged.

I can't guarantee that I respond to your email and I will junk any requests asking for help with assignments or homework.

## 3 Right to use

Use of the Toolbox is subject to the License Agreement. Many people are using the Toolbox for teaching and this is something that the author encourages. If you plan to duplicate the documentation for class use then every copy must include the front page.

If you want to cite the Toolbox please use

```
@article{Corke05f,
    Author = {P.I. Corke},
    Journal = {IEEE Robotics and Automation Magazine},
    Title = {Machine Vision Toolbox},
    Month = nov,
    Volume = {12},
    Number = {4},
    Year = {2005},
    Pages = {16-25}
}
```

or

> "*Machine Vision Toolbox*", P.I. Corke, IEEE Robotics and Automation Magazine, 12(4), pp 16–25, November 2005.

which is also given in electronic form in the CITATION file.

### 3.1 Acknowledgments

This release includes functions for computing image plane homographies and the fundamental matrix, contributed by Nuno Alexandre Cid Martins of I.S.R., Coimbra.

## 4 MATLAB version issues

The Toolbox works with MATLAB version 6 and later. It has been developed and tested under Suse Linux and Mac OS 10.3. It has not been tested under Windows.

# 2 Reference

| Camera modeling and calibration | |
|---|---|
| camcald | Camera calibration from non-coplanar 3D point data |
| camcalp | Camera calibration from intrinsic and extrinsic parameters |
| camcalp_c | Camera calibration from intrinsic and extrinsic parameters for central projection imaging model |
| camcalt | Camera calibration Tsai's method |
| camera | Camera imaging model |
| gcamera | Graphical camera imaging model |
| invcamcal | Inverse camera calibration by Ganapathy's method |
| pulnix | Calibration data form Pulnix TN6 camera |

| Image plane points and motion | |
|---|---|
| examples/fmtest | Example of `fmatrix()` |
| examples/homtest | Example of `homography()` |
| epidist | Distance of points from epipolar lines |
| epiline | Display epipolar lines |
| fmatrix‡ | Estimate the fundamental matrix |
| frefine‡ | Refine fundamental matrix |
| homography‡ | Estimate homography between 2 sets of points |
| homtrans | Transform points by an homography |
| invhomog | Invert an homography |
| visjac_p | Image Jacobian matrix from points |

| Image filtering | |
|---|---|
| `ismooth` | Gaussian smoothing |
| `ilaplace` | Laplace filtering |
| `isobel` | Sobel edge detector |
| `ipyramid` | Pyramid decomposition |
| `ishrink` | Image smoothing and shrinking |

| Monadic filtering | |
|---|---|
| `igamma` | gamma correction |
| `imono` | convert color to greyscale |
| `inormhist` | histogram normalization |
| `istretch` | linear normalization |

| Non-linear filtering | |
|---|---|
| `iclose` | greyscale morphological closing |
| `imorph`† | greyscale morphological operations |
| `iopen` | greyscale morphological opening |
| `irank`† | neighbourhood rank filter |
| `ivar`† | neighbourhood statistics |
| `iwindow`† | generalized neighbourhood operations |
| `pnmfilt` | Pipe image through Unix utility |
| `zcross` | zero-crossing detector |

| Image kernels and structuring elements | |
|---|---|
| `kdgauss` | Derivative of 2D Gaussian kernel |
| `kgauss` | 2D Gaussian kernel |
| `kdog` | Difference of Gaussians |
| `klaplace` | Laplacian kernel |
| `klog` | Laplacian of 2D Gaussian |
| `kcircle` | Circular mask |

| Image segmentation | |
|---|---|
| `trainseg` | Return blob features |
| `colorseg` | Display histogram |
| `ilabel`† | Label an image |
| `colordistance` | Distance in rg-colorspace |
| `kmeans` | k-means clustering |

| Image feature extraction | |
|---|---|
| `iblobs` | Return blob features |
| `ihist` | Display histogram |
| `ilabel`† | Label an image |
| `imoments` | Compute image moments |
| `iharris` | Harris interest point operator |
| `ihough` | Hough transform (image) |
| `ihough_xy` | Hough transform (list of edge points) |
| `houghoverlay` | overlay Hough line segments |
| `houghpeaks` | find peaks in Hough accumulator |
| `houghshow` | show Hough accumulator |
| `max2d` | find largest element in image |
| `mpq` | compute moments of polygon |
| `npq` | compute normalized central moments of polygon |
| `markcorners` | show corner points |
| `upq` | compute central moments of polygon |

| Feature tracking | |
|---|---|
| `imatch` | Image template search |
| `isimilarity` | Image window similarity |
| `subpixel` | Subpixel interpolation of peak |
| `zncc` | Region similarity |

| Image utilities | |
|---|---|
| `idisp` | Interactive image browser |
| `idisp2` | Non-interactive image browser |
| `iroi` | Extract region of interest |
| `xv` | Display image using the XV tool |

| Color space/photometry | |
|---|---|
| `blackbody` | Blackbody radiation |
| `ccdresponse` | CCD spectral response |
| `ccxyz` | CIE XYZ chromaticity coordinate |
| `cmfxyz` | CIE XYZ color matching function |
| `rgb2xyz` | RGB color space to CIE XYZ |
| `rluminos` | relative photopic luminosity (human eye response) |
| `solar` | solar irradiance spectra |

| Image file input/output | |
|---|---|
| `firewire`† | read an image from a firewire camera |
| `loadpgm` | read a PGM format file |
| `loadppm` | read a PPM format file |
| `savepnm` | write a PNM format file |
| `loadinr` | read INRIA INRIMAGE format file |
| `saveinr` | write INRIA INRIMAGE format file |
| `webcam` | read an image from a webcamera |
| `yuvopen` | open a yuv4mpeg image stream |
| `yuvread` | read a frame from a yuv4mpeg stream |
| `yuvr2rgb` | convert a YUV frame to RGB |

| Test patterns | |
|---|---|
| `lena.pgm` | a famous test image |
| `mkcube` | return vertices of a cube |
| `mkcube2` | return edges of a cube |
| `testpattern` | create range of testpatterns |

Functions marked with ‡ are written by others, and their support of the toolbox is gratefully acknowledged. Functions marked with † are mex-files and are currently only distributed in binary form for Linux x86 architecture and as source.

# blackbody

## Purpose

Compute emission spectrum for blackbody radiator

## Synopsis

```
qdd = blackbody(lambda, T)
```

## Description

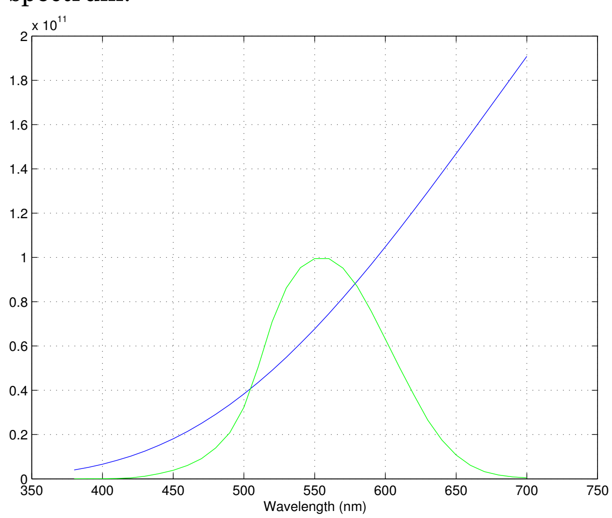
Returns the blackbody radiation in $(\mathrm{W}/\mathrm{m}^3)$ given lambda in (m) and temperature in (K). If `lambda` is a column vector, then `E` is a column vector whose elements correspond to to those in `lambda`.

## Examples

To compute the spectrum of a tungsten lamp at 2500K and compare that with human photopic response.

```
>> l = [380:10:700]'*1e-9;          % visible spectrum
>> e = blackbody(l, 2500);
>> r = rluminos(l);
>> plot(l*1e9, [e r*1e11])
>> xlabel('Wavelength (nm)')
```

which has the energy concentrated at the red-end (longer wavelength) of the spectrum.




## See Also

solar

# camcald

**Purpose** Camera calibration matrix from calibration data

**Synopsis**

```
C = camcald(D)
[C,resid] = camcald(D);
```

**Description** `camcald` returns a $3 \times 4$ camera calibration matrix derived from a least squares fit of the data in the matrix `D`. Each row of `D` is of the form `[x y z u v]` where $(x, y, z)$ is the world coordinate of a world point and $(u, v)$ is the image plane coordinate of the corresponding point. An optional residual, obtained by back substitution of the calibration data, can give an indication of the calibration quality.

At least 6 points are required and the points must not be coplanar.

**See Also** camcalp, camcalt, camera, invcamcal

**References**

I. E. Sutherland, "Three-dimensional data input by tablet," *Proc. IEEE*, vol. 62, pp. 453–461, Apr. 1974.

# camcalp

## Purpose

Camera calibration matrix from camera parameters

## Synopsis

```
C = camcalp(cp, Tcam)
C = camcalp(cp, pC, x, z)
```

```
C = camcalp_c(cp, Tcam)
C = camcalp_c(cp, pC, x, z)
```

## Description

Returns a $3 \times 4$ camera calibration matrix derived from the given camera parameters. The camera parameter object `cp` has elements:

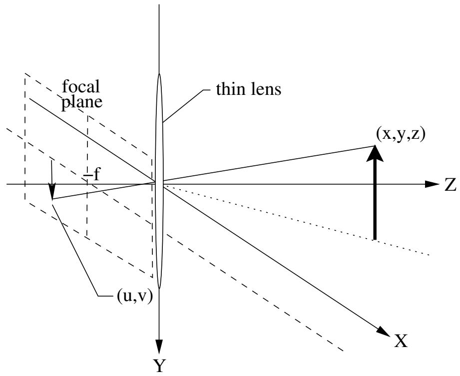
| | |
|---|---|
| `cp.f` | focal length (m) |
| `cp.u0` | principal point u-coordinate (pix) |
| `cp.v0` | principal point v-coordinate (pix) |
| `cp.px` | horizontal pixel pitch (pix/m) |
| `cp.py` | vertical pixel pitch (pix/m) |

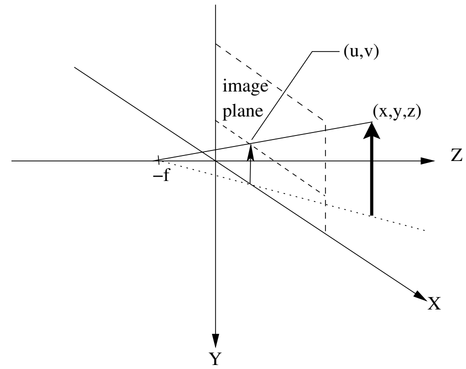
The pose of the camera (extrinsic calibration) can be specified by the homogeneous transformation `Tcam` or by specifying the coordinates of the center, `pC`, and unit vectors for the camera's x-axis and z-axis (optical axis).

This camera model assumes that the focal point is at $z = 0$ and the image plane is at $z = -f$. This means that the image is inverted on the image plane. Now in a real camera some of these inversions are undone by the manner in which pixels are rasterized so that generally increasing X in the world is increasing X on the image plane and increasing Y in the world (down) is increasing Y on the image plane. This has to be handled by setting the sign on the pixel scale factors to be negative.

`camcalp_c` is a variant for the central projection imaging model, as opposed to the thin lens model (which includes image inversion). Such a model is commonly used in computer vision literature where the focal point is at $z = 0$, and rays pass through the image plane at $z = f$. This model has no image inversion.




Lens projection model | Central projection model

**See Also** camcald, camcalt, camera, pulnix, invcamcal

# camcalt

**Purpose** Camera calibration matrix by Tsai's method

**Synopsis** `[Tcam,f,k1] = camcalt(D, PAR)`

**Description** Returns a $3 \times 4$ camera calibration matrix derived from from planar calibration data using Tsai's method. Each row of `D` is of the form `[x y z u v]` where $(x, y, z)$ is the world coordinate of a world point and $(u, v)$ is the image plane coordinate of the corresponding point. `PAR` is a vector of apriori known camera parameters `[Ncx Nfx dx dy Cx Cy]` where `Ncx` is the number of sensor elements in camera's x direction (in sels), `Nfx` is the number of pixels in frame grabber's x direction (in pixels), and `(Cx, Cy)` is the image plane coordinate of the principal point.

The output is an estimate of the camera's pose, `Tcam`, the focal length, `f`, and a lens radial distortion coefficient `k1`.

**Cautionary** I've never had much luck getting this method to work. It could be me, the type of images I take (oblique images are good), or the implementation. The Camera Calibration Toolbox `http://www.vision.caltech.edu/bouguetj/calib_doc/` gives nice results.

**See Also** camcalp, camcald, camera, invcamcal

**References** R. Tsai, "An efficient and accurate camera calibration technique for 3D machine vision," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pp. 364–374, 1986.

# camera

## Purpose

Camera projection model

## Synopsis

```
uv = CAMERA(C, p)
uv = CAMERA(C, p, Tobj)
uv = CAMERA(C, p, Tobj, Tcam)
```

## Description

This function computes the transformation from 3D object coordinates to image plane coordinates. `C` is a $3 \times 4$ camera calibration matrix, `p` is a matrix of 3D points, one point per row in X, Y, Z order. The points are optionally transformed by `Tobj`, and the camera is optionally transformed by `Tcam`, prior to imaging. The return is a matrix of image plane coordinates, where each row corresponds to the the row of `p`.

## Examples

Compute the image plane coordinates of a point at $(10, 5, 30)$ with respect to the standard camera located at the origin.

```
    >> C = camcalp(pulnix)
% create camera calibration matrix
    C =
      1.0e+05 *

      -0.7920         0   -0.3513    0.0027
            0   -1.2050   -0.2692    0.0021
            0         0   -0.0013    0.0000
    >> camera(C, [10 5 30])
    ans =
      479.9736  366.6907

    >>
```

**See Also** gcamera, camcalp, camcald

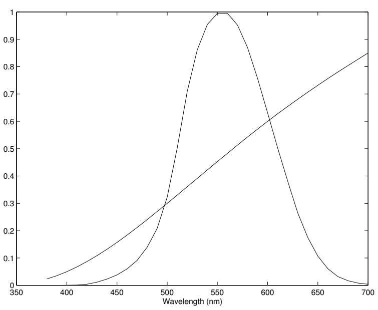
 

# ccdresponse

**Purpose** CCD spectral response

**Synopsis** `r = ccdresponse(lambda)`

**Description** Return a vector of relative response (0 to 1) for a CCD sensor for the specified wavelength `lambda`. `lambda` may be a vector.




**Examples** Compare the spectral response of a CCD sensor and the human eye. We can see that the CCD sensor is much more sensitive in the red and infra-red region than the eye.

```
>> l = [380:10:700]'*1e-9;
>> eye = rluminos(l);
>> ccd = ccdresponse(l);
>> plot(l*1e9, [eye ccd])
>> xlabel('Wavelength (nm)')
```

**Limitations** Data is taken from an old Fairchild CCD data book but is somewhat characteristic of silicon CCD sensors in general.

**See Also** rluminos

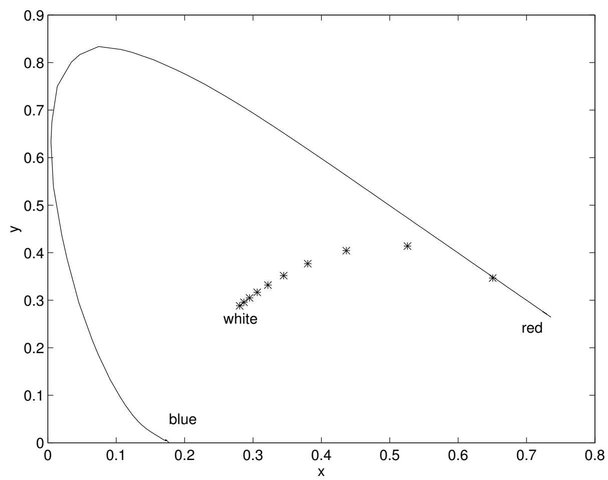
# ccxyz

**Purpose** Compute the CIE XYZ chromaticity coordinates

**Synopsis**

```
cc = ccxyz(lambda)
cc = ccxyz(lambda, e)
```

**Description** `ccxyz` computes the CIE 1931 XYZ chromaticity coordinates for the wavelength `lambda`. Chromaticity can be computed for an arbitrary spectrum given by the equal length vectors `lambda` and amplitude `e`.




**Examples** The chromaticity coordinates of peak green (550 nm) is

```
>> ccxyz(550e-9)
ans =
    0.3016    0.6924    0.0061
```

and the chromaticity coordinates of a standard tungsten illuminant (color temperature of 2856 K) is

```
>> lambda = [380:2:700]'*1e-9;          % visible spectrum
>> e = blackbody(lambda, 2856);
>> ccxyz(lambda, e)
ans =
```

```
0.4472    0.4077    0.1451
```

The spectral locus can be drawn by plotting the chromaticity y-coordinate against the x-coordinate

```
>> xyz = ccxyz(lambda);
>> plot(xyz(:,1), xyz(:,2));
>> xlabel('x'); ylabel('y')
```

The blackbody locus can be superimposed by

```
>> for T=1000:1000:10000,% from 1,000K to 10,000K
>>   e = blackbody(lambda, T);
>>   xyz = ccxyz(lambda, e);
>>   plot(xyz(1), xyz(2), '*')
>> end
```

which shows points moving from red to white hot (center of the locus) as temperature increases.

**See Also** cmfxyz, blackbody

# cmfxyz

## Purpose

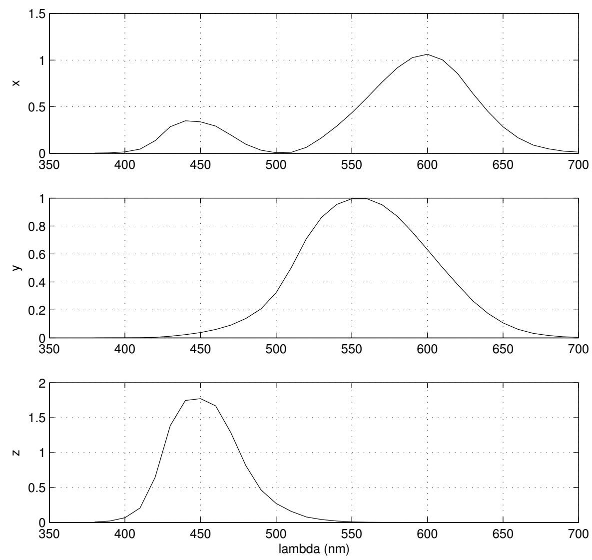
Color matching function

## Synopsis

```
xyz = cmfxyz(lambda)
```

## Description

`ccxyz` computes the CIE 1931 color matching functions for the wavelength `lambda` which is returned as a row vector. If `lambda` is a vector then the rows of `xyz` contains the color matching function for the corresponding row of `lambda`.




## Examples

Plot the X, Y and Z color matching functions as a function of wavelength.

```
>> lambda = [350:10:700]'*1e-9;
>> xyz = cmfxyz(lambda);
>> for i=1:3,
>>       subplot(310+i); plot(lambda, xyz(:,i));
>>   end
```

## See Also

ccxyz

# colordistance

**Purpose** Distance in rg-colorspace

**Synopsis**

```
r = colordistance(rgb, rg)
```

**Description** Each pixel of the input color image `rgb` is converted to normalized $(r,g)$ coordinates

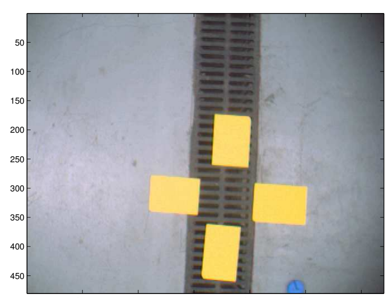
$$r = \frac{R}{R+G+B} \quad (1)$$

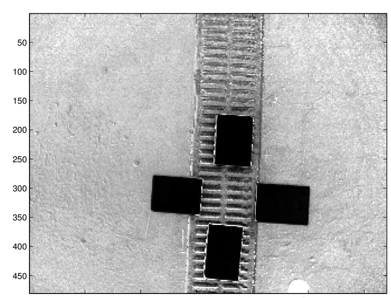
$$g = \frac{G}{R+G+B} \quad (2)$$

The Euclidean distance of each pixel from the specificed coordinage `rg` is computed and returned as the corresponding pixel value.

The output is an image with the same number of rows and columns as `rgb` where each pixel represents the correspoding color space distance.

This output image could be thresholded to determine color similarity.




**Examples** Show color distance of all targets with respect to a point, $(200, 350)$ on one of the yellow targets

```
>> targ = loadppm('target.ppm');
>> pix = squeeze( targ(200,350,:) );
>> rg = pix / sum(pix);
>> idisp( colordistance(targ, rg(1:2)), 0.02 )
```

We use the clipping option of `idisp()` to highlight small variations, since the blue object has a very large color distance.

**See Also** colorseg, trainseg

# colorseg

## Purpose

Perform rg-space color segmentation

## Synopsis

```
imseg = colorseg(rgb, map)
```

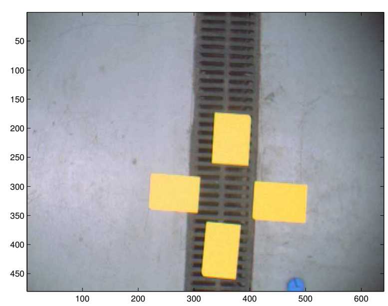
## Description

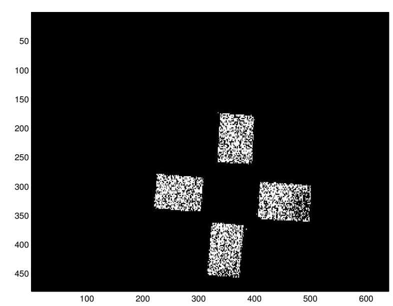
Each pixel of the input color image `rgb` is converted to normalized $(r, g)$ coordinates

$$r = \frac{R}{R+G+B} \qquad (3)$$

$$g = \frac{G}{R+G+B} \qquad (4)$$

and these pixels are mapped through the segmentation table `map` to determine whether or not they belong to the desired pixel class. The map values can be crisp (0 or 1) or fuzzy, though the `trainseg()` creates crisp values.




## Examples

Use a pre-trained color segmentation table to segment out the yellow targets

```
>> cs = colorseg(targ, map);
>> idisp(cs);
```

The segmentation is spotty because the segmentation map is not solid. We could apply morphological closing to fill the black spots in either the segmentation map or the resulting segmentation.

## See Also

trainseg

# epidist

**Purpose** Distance from point to epipolar line

**Synopsis**

```
d = epidist(F, Pa, Pb)
```

**Description** Given two sets of points `Pa` ($n \times 2$) and `Pb` ( $m \times 2$ matrix) compute the epipolar line corresponding to each point in `Pa` and the distance of each point in `Pb` from each line. The result, $d(i, j)$, is a $n \times m$ matrix of distance between the epipolar line corresponding to $Pa_i$ and the point $Pb_j$.

Can be used to determine point correspondance in a stereo image pair.

**See Also** fmatrix

# epiline

## Purpose

Display epipolar lines

## Synopsis

```
h = epiline(F, Pa)
h = epiline(F, Pa, ls)
```

## Description

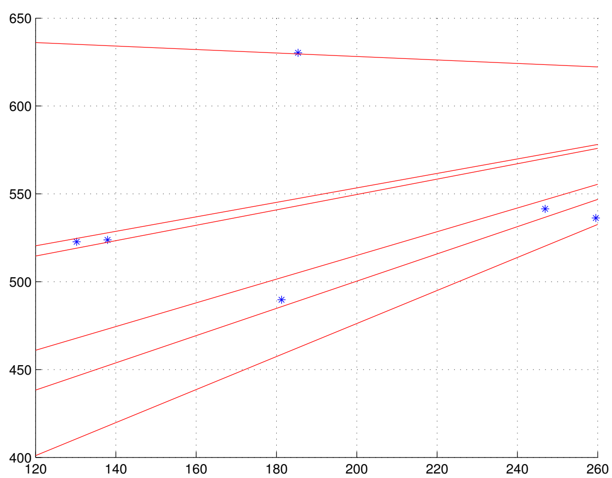
Draw epipolar lines in current figure based on points specified rowwise in `Pa` and on the fundamental matrix `F`. Optionally specify the line style `ls`.

Adds the lines to the current plot.

## Examples

Display epipolar lines for the example (`examples/fmtest`).

```
>> fmtest
  .
  .
>> Fr = frefine(F, uv0, uvf);
>> markfeatures(uvf, 0, '*')
>> epiline(Fr, uv0)
>> grid
```




## See Also

fmatrix, epidist

# firewire

**Purpose** Load an image from a firewire camera

**Synopsis**

```
h = firewire(device, color, framerate)
im = firewire(h)
```

**Description** The first form opens the interface and returns a handle or [] on error. Color is one of 'mono', 'rgb' or 'yuv'. `framerate` is one of the standard DC1394 rates: 1.875, 3.75, 7.5, 15, 30 or 60 fps. The highest rate less than or equal to rate is chosen.

The second form reads an image. For mono a 2-d matrix is returned, for rgb a 3-d matrix is returned. For yuv a structure is returned with elements `.y`, `.u` and `.v`.

Subsequent calls with the second call format return the next image from the camera in either grayscale or color format.

**Examples** Open a firewire camera in rgb mode

.

```
>> h = firewire( 0, 'rgb', 7.5);
CAMERA INFO
===============
Node: 0
CCR_Offset: 15728640x
UID: 0x0814436100003da9
Vendor: Unibrain Model: Fire-i 1.2

>> im = firewire(h);
>> whos im
  Name      Size                       Bytes  Class
```

```
  im      480x640x3              7372800  double array

Grand total is 921600 elements using 7372800 bytes

>>
```

**Limitations** Only `FORMAT_VGA_NONCOMPRESSED` $640 \times 480$ images are supported, and the camera's capabilities are not checked against the requested mode, for example older Point Grey Dragonflies give weird output when `'mono'` is requested which they don't support.

The achievable frame rate depends on your computer. The function waits for the next frame to become available from the camera. If the function is called too late you may miss the next frame and have to wait for the one after that.

**Limitations** Operates only under Linux and is a mex-file. Requires the `libdc1394` and `libraw1394` libraries to be installed.

**See Also** webcam

# fmatrix

## Purpose

Estimate the fundamental matrix

## Synopsis

```
F = fmatrix(Pa, Pb)
F = fmatrix(Pa, Pb, how)
```

## Description

Given two sets of corresponding points `Pa` and `Pb` (each a $n \times 2$ matrix) return the fundamental matrix relating the two sets of observations.

The argument `'how'` is used to specify the method and is one of `'eig'`, `'svd'`, `'pinv'`, `'lsq'` (default) or `'ransac'`.

RANSAC provides a very robust method of dealing with incorrect point correspondances through outlier rejection. It repeatedly uses one of the underlying methods above in order to find inconsistant matches which it then eliminates from the process. RANSAC mode requires extra arguments:

| | |
|---|---|
| `iter` | maximum number of iterations |
| `thresh` | a threshold |
| `how` | the underlying method to use, as above, except for ransac (optional). |

Note that the results of RANSAC may vary from run to run due to the random subsampling performed.

All methods require at least 4 points except `'eig'` which requires at least 5. The fundamental matrix is rank 2, ie. `det(F)` = 0.

## Examples

In the following example (`examples/fmtest`) we will set up a Pulnix camera and a set of random point features (within a 1m cube) 4m in front of the camera. Then we will translate and rotate the camera to get another set of image plane points. From the two sets of points we compute the fundamental matrix.

```
>> C=camcalp(pulnix);
>> points = rand(6,3);
>> C = camcalp(pulnix);
>> uv0 = camera(C, points, transl(0,0,4))
```

```
uv0 =
  310.7595  293.7777
  367.1380  317.2675
  342.7822  387.1914
  281.4286  323.2531
  285.9791  277.3937
  315.6825  321.7783

>> uvf = camera(C, points, transl(0,0,4), transl(1,1,0)*rotx
uvf =
  169.8081  577.5535
  214.7405  579.9254
  207.0585  701.6012
  145.8901  629.0040
  144.5274  559.0554
  154.8456  576.6023

>> F = fmatrix(uv0, uvf)
maximum residual 0.000000 pix

F =
    0.0000   -0.0000    0.0031
    0.0000    0.0000   -0.0027
   -0.0025   -0.0009    1.0000

>> det(F)
ans =
   1.1616e-12
```

We can see that the matrix is close to singular, theoretically it should be of rank 2.

**Author** Nuno Alexandre Cid Martins, I.S.R., Coimbra

**References**

M. A. Fischler and R. C. Bolles, “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography,” *Communications of the ACM*, vol. 24, pp. 381–395, June 1981.

O. Faugeras, *Three-dimensional computer vision.* MIT Press, 1993.

**See Also** homography, epidist, examples/fmtest

# frefine

**Purpose** Refine fundamental matrix estimate

**Synopsis** `Fr = frefine(F, Pa, Pb)`

**Description** Given two sets of corresponding points `Pa` and `Pb` (each a $n \times 2$ matrix) and an estimate of the fundamental matrix `F`, refine the estimate using non-linear optimization to enforce the rank 2 constraint.

**Examples** In the following example (`examples/fmtest`) we will set up a Pulnix camera and a set of random point features (within a 1m cube) 4m in front of the camera. Then we will translate and rotate the camera to get another set of image plane points. From the two sets of points we compute the fundamental matrix.

```
>> C=camcalp(pulnix);
>> points = rand(6,3);
>> C = camcalp(pulnix);
>> uv0 = camera(C, points, transl(0,0,4));
>> uvf = camera(C, points, transl(0,0,4), transl(1,1,0)*rotx
>> F = fmatrix(uv0, uvf);
maximum residual 0.000000 pix
F =

    0.0000    0.0000    0.0011
   -0.0000    0.0000   -0.0009
   -0.0007   -0.0016    1.0000

>> det(F)
ans =
```

```
   1.1616e-12
>> Fr = frefine(F, uv0, uvf)
 .
 .
Fr =

   -0.0000    0.0000   -0.0098
   -0.0000    0.0000    0.0033
    0.0106   -0.0267    1.3938
>> det(Fr)
ans =
   7.8939e-19
```

We can see that the determinant is much closer to zero.

**See Also** homography, epidist, examples/fmtest

# gcamera

## Purpose

Graphical camera projection model

## Synopsis

```
hcam = gcamera(name, C ,dims)
```

```
uv = gcamera(hcam, p)
uv = gcamera(hcam, p, Tobj)
uv = gcamera(hcam, p, Tobj, Tcam)
```

## Description

This function creates and graphically displays the image plane of a virtual camera.

The first function creates a camera display with given name and camera calibration matrix. The size, in pixels of the image plane is given by `dims` and is of the form `[umin umax vmin vmax]`. The function returns a camera handle for subsequent function calls.

The second form is used to display a list of 3D points `p` in the image plane of a previously created camera whose handle is `hcam`. The points are optionally transformed by `Tobj`, and the camera is optionally transformed by `Tcam` prior to imaging. A single Matlab line object (with point marker style) joins those points. Successive calls redraw this line providing an animation.

If `p` has 6 columns rather than 3, then it is considered to represent world line segments, rather than points. The first three elements of each row are the coordinates of the segment start, and the last three elements the coordinates of the end. Successive calls redraw the line segments providing an animation.

## Limitations

Mixing calls in point and line mode give unpredicable results.

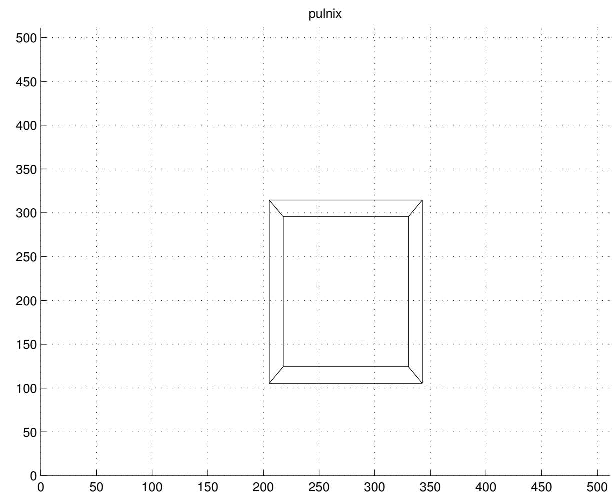
## Examples

Create a virtual Pulnix camera situated at the origin with view axis along the world Zaxis. Create a cube of unit side and view it after translating it's centre to $(0, 0, 5)$. Note that `transl` is a function from the Robotics Toolbox.

```
>> C = camcalp(pulnix);
```

```
>> h = gcamera('Pulnix', C, [0 511 0 511]);
>> c = mkcube2;
>> gcamera(h, c, transl(0, 0, 5));
```




**See Also** `camera, camcalp, pulnix, mkcube2`

# homography

## Purpose

Estimate an homography

## Synopsis

```
H = homography(Pa, Pb)
H = homography(Pa, Pb, how)
```

## Description

Given two sets of corresponding points Pa and Pb (each an nx2 matrix) return the homography relating the two sets of observations. The homography is simply a linear transformation of the initial set of points to the final set of points.

The argument `'how'` is used to specify the method and is one of `'eig'`, `'svd'`, `'pinv'`, `'lsq'` (default) or `'ransac'`.

RANSAC provides a very robust method of dealing with incorrect point correspondances through outlier rejection. It repeatedly uses one of the underlying methods above in order to find inconsistant matches which it then eliminates from the process. RANSAC mode requires extra arguments:

| | |
|---|---|
| `iter` | maximum number of iterations |
| `thresh` | a threshold |
| `how` | the underlying method to use, as above, except for ransac (optional). |

Note that the results of RANSAC may vary from run to run due to the random subsampling performed.

All methods require at least 4 points except `'eig'` which requires at least 5. The homography is only defined for points that are coplanar.

## Examples

In the following example (`examples/homtest`) we will set up a Pulnix camera and a set of planar features 8m in front of the camera. Then we will translate and rotate the camera to get another set of image plane points. From the two sets of points we compute the homography, and then check it by back subsitution.

```
>> C=camcalp(pulnix);
>> points = [0 0.3 0; -1 -1 0; -1 1 0; 1 -1 0; 1 1 0];
>> C = camcalp(pulnix);
```

```
>> uv0 = camera(C, points, transl(0,0,8))
uv0 =
  274.0000  245.2806
  196.7046   92.3978
  196.7046  327.6022
  351.2954   92.3978
  351.2954  327.6022

>> uvf = camera(C, points, transl(0,0,8), transl(2,1,0)*rotx
uvf =
  105.8668  621.9923
   41.5179  455.2694
    9.7312  724.0408
  196.5060  455.2694
  185.9104  724.0408

>> H = homography(uv0, uvf)
H =
    0.9573   -0.1338 -136.3047
   -0.0000    0.7376  366.5758
   -0.0000   -0.0005    1.0000

>> homtrans(H, uv0)-uvf
ans =
   1.0e-09 *

   -0.0876    0.0441
   -0.0473    0.2508
    0.1402   -0.3031
   -0.0290   -0.0356
```

```
0.0715    0.2944
```

**Author** Nuno Alexandre Cid Martins, I.S.R., Coimbra

**References**

M. A. Fischler and R. C. Bolles, “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography,” *Communications of the ACM*, vol. 24, pp. 381–395, June 1981.

O. Faugeras, *Three-dimensional computer vision*. MIT Press, 1993.

**See Also** homtrans, examples/homtest, fmatrix

# homtrans

**Purpose** Transform points by an homography

**Synopsis** `ph = homtrans(H, p)`

**Description** Apply the homography H to the image-plane points `p`. `p` is an $n \times 2$ or $n \times 3$ matrix whose rows correspond to individual points non-homogeneous or homogeneous form.

Returns points as `ph`, an $n \times 3$ matrix where each row is the point coordinate in homogeneous form.

**Examples** See the example for `homography()`.

**See Also** homography, examples/homtest

# iblobs

**Purpose** Compute image blob features

**Synopsis**

```
F = iblobs(image)
F = iblobs(image, options, ...)
```

**Description** Returns a vector of structures containing feature data and moments upto second order for each connected (4 or 8 way) region in the image `image`. The image is first labelled and then features are computed for each region.

The feature structure is an augmented version of that returned by `imoments` and contains in addition `F(i).minx`, `F(i).maxx`, `F(i).miny`, `F(i).maxy` and `F(i).touch` which is true if the region touches the edge of the image. `F(i).shape` is the ratio of the ellipse axes in the range 0 to 1.

The second form allows various options and blob filters to be applied by specifying name and value pairs.

| | |
|---|---|
| `'aspect', ratio` | specify the pixel aspect ratio (default 1) |
| `'connect', connectivity` | specficy connectivt (default 4) |
| `'touch', flag` | only return regions whose `touch` status matches |
| `'area', [amin amax]` | only return regions whose area lies within the specified bounds |
| `'shape', [smin smax]` | only return regions whose shape measures lies within the specified bounds |

Note that to turn one element from a vector of structures into a vector use the syntax `[F.x]`.

**Examples** Compute the blob features for a test pattern with a grid of $5 \times 5$ dots. 26 blobs are found, each of the dots (blobs 2–26), and the background (blob 1).

```
>> im = testpattern('dots', 256, 50, 10);
>> F = iblobs(im)
```

```
26 blobs in image, 26 after filtering
F =
1x26 struct array with fields:
  area
  x
  y
  a
  b
  theta
  m00
  m01
  m10
  m02
  m20
  m11
  minx
  maxx
  miny
  maxy
  touch
  shape
>> F(1)
ans =
   area: 63511
      x: 128.6116
      y: 128.6116
      a: 147.9966
      b: 147.9857
  theta: -0.7854
    m00: 63511
```

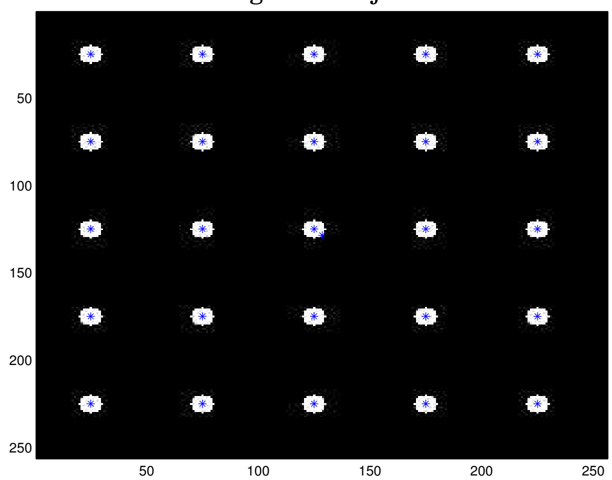
 

```
     m01: 8168251
     m10: 8168251
     m02: 1.3983e+09
     m20: 1.3983e+09
     m11: 1.0505e+09
    minx: 1
    maxx: 256
    miny: 1
    maxy: 256
   touch: 1
   shape: 0.9999

>> F(2)
ans =
    area: 81
       x: 25
       y: 25
       a: 5.0966
       b: 5.0966
   theta: 0
     m00: 81
     m01: 2025
     m10: 2025
     m02: 51151
     m20: 51151
     m11: 50625
    minx: 20
    maxx: 30
    miny: 20
    maxy: 30
```

```
  touch: 0
  shape: 1
>>
>> idisp(im)
>> markfeatures(F, 0, 'b*')
```

The last two lines overlay the centroids onto the original image. Note the centroid of the background object close to the middle dot.



**See Also** imoments, markfeatures, ilabel

# icanny

**Purpose** Canny edge operator

**Synopsis**

```
e = canny(im)
e = canny(im, sigma)
e = canny(im, sigma, th1)
e = canny(im, sigma, th1, th0)
```

**Description** Finds the edges in a gray scaled image `im` using the Canny method, and returns an image `e` where the edges of `im` are marked by non-zero intensity values. This is a more sophisticated edge operator than the Sobel.

The optional argument `sigma` is the standard deviation for the Gaussian filtering phase. Default is 1 pixel.

`th1` is the higher hysteresis threshold. Default is 0.5 times the strongest edge. Setting `th1` to zero will avoid the (sometimes time consuming) hysteresis. `th0` is the lower hysteresis threshold and defaults to 0.1 times the strongest edge.


```
>> lena = loadpgm('lena');
>> ic = icanny(lena);
```

**Author** Oded Comay, Tel Aviv University

**References** J. Canny, "A computational approach to edge detection" IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 8(6), November 1986, pp 679 - 698.

**See Also** isobel, ilaplace

# iclose

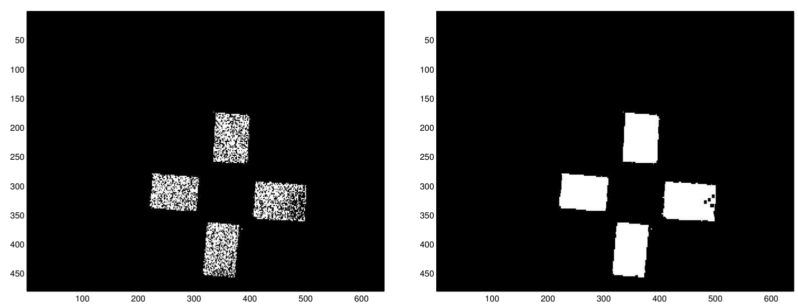
**Purpose** Grey scale morphological opening

**Synopsis**

```
im2 = iclose(im)
im2 = iclose(im, se)
im2 = iclose(im, se, N)
```

**Description** Perfoms a greyscale morphological closing on the image `im` using structuring element `se` which defaults to `ones(3,3)`. The operation comprises `N` (default 1) consecutive dilations followed by `N` consectutive erosions.

Square structuring elements can be created conveniently using `ones(N,N)` and circular structuring elements using `kcircle(N)`.




**Examples** We can use morphological closing to fill in the gaps in an initial segmentation.

```
>> idisp(cs)
>> idisp( iclose(cs, ones(5,5) ) );
```

**See Also** imorph, iopen, kcircle

# idisp

## Purpose

Interactive image display utility

## Synopsis

```
idisp(im)
idisp(im, clip)
idisp(im, clip, n)

idisp2(im)
```

## Description

Displays an image browser in a new figure window and allows interactive investigation of pixel values, see Figure.

Buttons are created along the top of the window:

**line** Prompt for two points in the image and display a cross-section in a new figure. This shows intensity along the line between the two points selected.

**zoom** Prompt for two points and rescale the image so that this region fills the figure. The zoomed image may itself be zoomed.

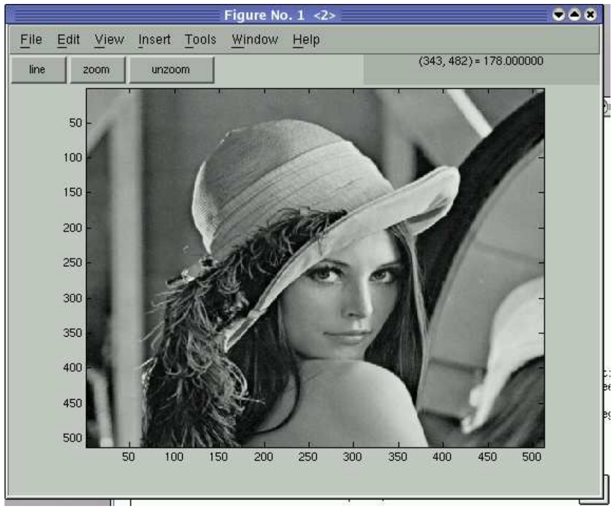
**unzoom** Return image scaling to original settings.

Clicking on a pixel displays its value and coordinate in the top row. Color images are supported.

The second form will limit the displayed greylevels. If `clip` is a scalar pixels greater than this value are set to `clip`. If `clip` is a 2-vector then pixels less than `clip(1)` are set to `clip(1)` and those greater than `clip(2)` are set to `clip(2)`. `clip` can be set to [] for no clipping. This option is useful to visualize image content when there is a very high dynamic range. The `n` argument sets the length of the greyscale color map (default 64).

`idisp2` is a non-interactive version, that provides the same display functionality but has no GUI elements.




**See Also** iroi, xv

# igamma

**Purpose** Image gamma correction

**Synopsis**

```
hn = igamma(image, gamma)
hn = igamma(image, gamma, maxval)
```

**Description** Returns a gamma corrected version of `image`, in which all pixels are raised to the power `gamma`. Assumes pixels are in the range 0 to `maxval`, default `maxval` = 1.

**See Also** inormhist

# iharris

**Purpose** Harris interest point detector

**Synopsis**

```
P = iharris

F = iharris(im)
F = iharris(im, P)
[F,rawC] = iharris(im, P)
```

**Description** Returns a vector of structures describing the corner features detected in the image `im`. This is a computationally cheap and robust corner feature detector. The Harris corner strength measure is

$$C = \hat{I^2}_x \hat{I^2}_y - \hat{I_{xy}}^2 - k(\hat{I^2}_x + \hat{I^2}_y)^2$$

and the Noble corner detector is

$$C = \frac{\hat{I^2}_x \hat{I^2}_y - \hat{I_{xy}}^2}{\hat{I^2}_x + \hat{I^2}_y}$$

Where $\hat{I^2}_x$ and $\hat{I^2}_y$ are the smoothed, squared, directional gradients, and $\hat{I_{xy}}^2$ is the smoothed gradient product. For a color image the squared gradients are computed for each plane and then summed.

The feature vector contains structures with elements:

| | |
|---|---|
| `F.x` | x-coordinate of the feature |
| `F.y` | y-coordinate of the feature |
| `F.c` | corner strength of the feature |
| `F.grad` | 3-element vector comprising $[\hat{I^2}_x, \hat{I^2}_y, \hat{I_{xy}}^2]$ the smoothed gradients at the corner. |

The gradients can be used as a simple signature of the corner to help match corners between different views. A more robust method to match corners is with cross-correlation of a small surrounding region.

There are many parameters available to control this detector, given by the second argument `P`. The default value of `P` can be obtained by the first call format.

| | |
|---|---|
| `P.k` | $k$ parameter (default 0.04) |
| `P.cmin` | minimum corner strength (default 0) |
| `P.cMinThresh` | minimum corner strength as a fraction of maximum detected corner strength (default 0.01) |
| `P.deriv` | x-derivative kernel (default is $\begin{bmatrix} -1/3 & 0 & 1/3 \\ -1/3 & 0 & 1/3 \\ -1/3 & 0 & 1/3 \end{bmatrix}$) |
| `P.sigma` | $\sigma$ of Gaussian for smoothing step (default 1) |
| `P.edgegap` | width of region around edge where features cannot be detected (default 2) |
| `P.nfeat` | maximum number of features to detect (default all) |
| `P.harris` | Harris (1) or Noble corner detector (default 1) |
| `P.tiling` | determine strongest features in a `P.tiling` $\times$ `P.tiling` tiling of the image. Allows more even feature distribution (default 1). |
| `P.distance` | enforce a separation between features (default 0). |

Optionally returns the raw corner strength image as `rawC`.

**Examples**

Find the corners in the Lena image. Display a white diamond at the location of the 20 strongest corners and label them. Enforce a separation of 20 pixels between features.

```
>> lena = loadpgm('lena');
>> P = iharris;
>> P.distance = 20;
>> F = iharris(lena, P);
12250 minima found (4.7%)
```

```
break after 629 minimas
>> markfeatures(F, 20, 'wd', {10, 'w'})
>>
>> P.tiling = 2;
>> P.nfeat = 10;
>> F = iharris(lena, P);
tile (1,1): 1399 minima found (4.8%), break after 17 minim
tile (1,2): 1356 minima found (4.7%),  10 added
tile (1,3): 1292 minima found (4.4%),  10 added
tile (2,1): 1378 minima found (4.7%),  10 added
tile (2,2): 1307 minima found (4.5%),  10 added
tile (2,3): 1380 minima found (4.7%),  10 added
tile (3,1): 1230 minima found (4.2%),  10 added
tile (3,2): 1467 minima found (5.0%), break after 34 minim
tile (3,3): 1344 minima found (4.6%),  10 added

>> F
F =
1x84 struct array with fields:
  x
  y
  c
  grad

>> markfeatures(F, 0, 'wd', {10, 'w'})
```

Note that in two of the tiles not enough corners could be found that met the criteria of inter-corner separation and corner strength. The process yielded only 84 corners, not the 90 requested, however the coverage of the scene is greatly improved.

**See Also** markfeatures, zncc, isimilarity

**References**

C. G. Harris and M. J. Stephens, “A Combined Corner and Edge Detector,” in *Proceedings of the Fourth Alvey Vision Conference, Manchester*, pp. 147–151, 1988.

# ihist

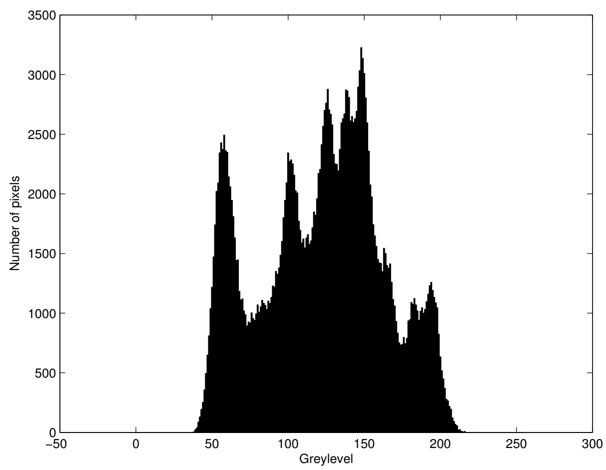
**Purpose** Compute intensity histogram (fast)

**Synopsis**

```
ihist(im)
N = ihist(im)
[N,X] = ihist(im)
```

**Description** This function computes the intensity histogram of an image.

The first form plots a graph of the histogram, while the last two forms simply return the histogram and bin values: `N` is the bin count and `X` is the bin number.




**Examples** Display the histogram of the Lena image.

```
>> lena = loadpgm('lena');
>> ihist(lena)
```

**Limitations** Assumes that the pixels are in the range 0-255 and always computes 256 bins. Some functions to interpret the histogram to find extrema or fit Gaussians would be useful, see `fit_ML_normal` from Matlab file exchange.

**See Also** hist, kmeans

# ihough

## Purpose

Linear Hough transform

## Synopsis

```
hp0 = ihough
H = ihough(edge)
H = ihough(edge, hp)
H = ihough_xy(xyz, drange, ntheta)

houghshow(H)
houghpeaks(H, n)
h = houghoverlay(p, ls)
```

## Description

Computes the linear Hough transform of the image `image`. Non-zero pixels in the input edge image `edges` increment all pixels in the accumulator that lie on the line

$$d = y\cos(\theta) + x\sin(\theta) \tag{5}$$

where $\theta$ is the angle the line makes to horizontal axis, and $d$ is the perpendicular distance between (0,0) and the line. A horizontal line has $\theta = 0$, a vertical line has $\theta = \pi/2$ or $-\pi/2$. The accumulator array has theta across the columns and offset down the rows. The Hough accumulator cell is incremented by the absolute value of the pixel value if it exceeds `params.edgeThresh` times the maximum value found in `edges`. Clipping is applied so that only those points lying within the Hough accumulator bounds are updated.

An alternative form `ihough_xy()` takes a list of coordinates rather than an image. `xyz` is either an $n \times 2$ matrix of $(x, y)$ coordinates, each of which is incremented by 1, or an $n \times 3$ matrix where the third column is the amount to incrmement each cell by.

The returned Hough object `H` has the elements:

| | |
|---|---|
| `H.h` | Hough accumulator |
| `H.theta` | vector of theta values corresponding to accumulator columns |
| `H.d` | vector of offset values corresponding to accumulator rows |

Operation can be controlled by means of the parameter object `hp` which has elements:

| | |
|---|---|
| `hp.Nd` | number of bins in the offset direction (default 64) |
| `hp.Nth` | number of bins in the theta direction (default 64) |
| `hp.edgeThresh` | edge threshold (default 0.10) |
| `hp.border` | edge threshold (default 8) |
| `hp.houghThresh` | threshold on relative peak strength (default 0.40) |
| `hp.radius` | radius of accumulator cells cleared around peak after detection (default 5) |
| `hp.interpWidth` | width of region used for peak interpolation (default 5) |

Pixels within `hp.border` of the edge will not increment, useful to eliminate spurious edge pixels near image border.

Theta spans the range $-\pi/2$ to $\pi/2$ in `hp.Nth` increments. Offset is in the range 1 to the number of rows of `edges` with `hp.Nd` steps. For the `ihough_xy` form the number of theta steps is given by `ntheta` and the offset is given by a vector `drange = [dmin dmax]` or `drange = [dmin dmax Nd]`.

The default parameter values can be obtained by calling `ihough` with no arguments.

`houghshow` displays the Hough accumulator as an image.

`houghpeaks` returns the coordinates of `n` peaks from the Hough accumulator. The highest peak is found, refined to subpixel precision, then `hp.radius` radius around that point is zeroed so as to eliminate multiple close minima. The process is repeated for all `n` peaks. `p` is an $n \times 3$ matrix where each row is the offset, theta and relative peak strength (range 0 to 1). The peak detection loop breaks early if the remaining peak has a relative strength less than `hp.houghThresh`. The peak is refined by a weighted mean over a $w \times w$ region around the peak where $w =$ `hp.interpWidth`.

`houghoverlay` draws the lines corresponding to the rows of `p` onto the current figure using the linestyle `ls`. Optionally returns a vector of handles `h` to the lines drawn.

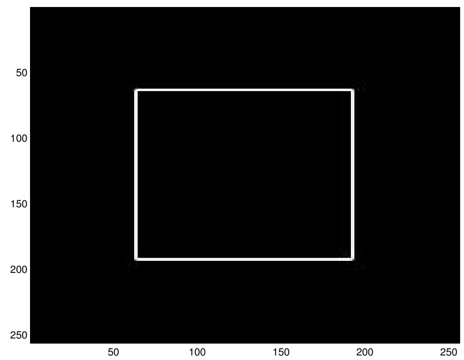
## Examples

Find the Hough transform of the edges of a large square, created using `mksq` and a Laplacian edge operator. The accumulator can be displayed as an image which shows four bright spots, each corresponding to an edge. As a surface these appear as high, but quite ragged, peaks.

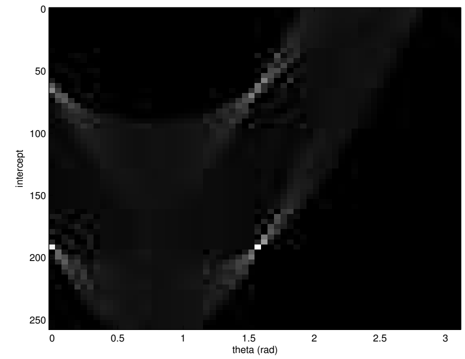
 

```
>> im=testpattern('squares', 256, 256, 128);
>> edges = isobel(im);
>> idisp(im);
>> H = ihough(edges)
H =

      h: [64x64 double]
  theta: [64x1 double]
      d: [64x1 double]

>> houghshow(H);
>> p=houghpeaks(H, 4)
p =
191.2381         0    1.0000
190.9003    1.5647    1.0000
 69.8095    0.0491    0.6455
 70.1650    1.5239    0.6455

>> idisp(im);
>> houghoverlay(p, 'g')
theta = 0.000000, d = 191.238095
theta = 1.564731, d = 190.900293
theta = 0.049087, d = 69.809524
theta = 1.523942, d = 70.164994
```

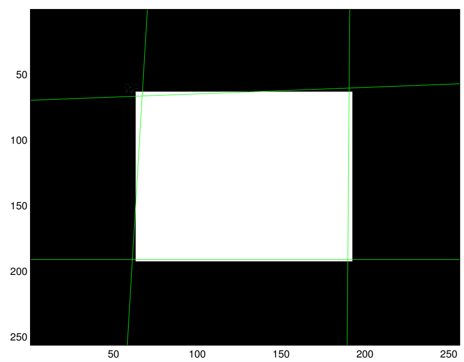
 




Edge image




Hough accumulator




Fitted line segments

# ilabel

**Purpose** Image labelling (segmentation)

**Synopsis**

```
L = ilabel(I)
[L, maxlabel] = ilabel(I)
[L, maxlabel, parents] = ilabel(I)
```

**Description** Returns an equivalent sized image, `L`, in which each pixel is the label of the region of the corresponding pixel in `I`. A region is a spatially contiguous region of pixels of the same value. The particular label assigned has no significance, it is an arbitrary label.

Optionally the largest label can be returned. All labels lie between 1 and `maxlabel`, and there are no missing values. Connectivity is 4-way by default, but 8-way can be selected.

The third form returns an array of region hierarchy information. The value of `parents(i)` is the label of the region that fully encloses region `i`. The outermost blob(s) will have a parent value of 0.

**Examples** Consider the simple binary image

```
>> labeltest
 .
 .
>> a
a =
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     1     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     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     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
>> [L,lm,p]=ilabel(a)
L =
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     2     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     3     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
lm =
     3
p =
     0
     1
     1
```

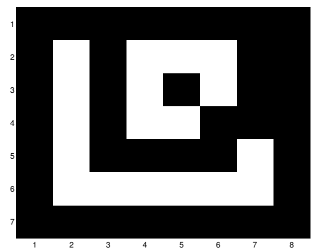
which indicates that there are 3 labels or regions. Region 1, the background has a parent of 0 (ie. it has no enclosing region). Regions 2 and 3 are fully enclosed by region 1.

To obtain a binary image of all the pixels in region 2, for example,

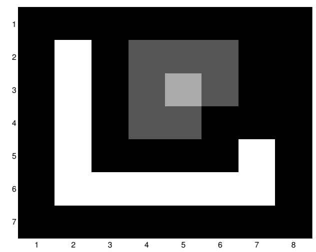
```
>> L==2
>> L==2
ans =
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 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 |
| 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 | 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 |




Binary image




Labelled image

**See Also** imoments, iblobs

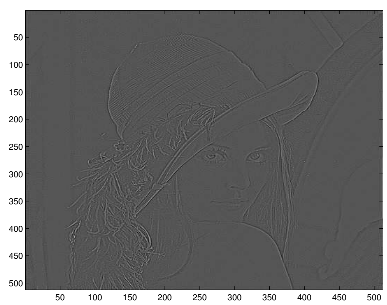
# ilaplace

**Purpose** Laplacian filter

**Synopsis** `G = ilaplace(image)`

**Description** Convolves all planes of the input image with the Laplacian filter

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$




**Examples** Laplace filter the Lena image.

```
>> lena = loadpgm('lena');
>> idisp( ilaplace( lena) )
```

**See Also** conv2, klog, ismooth, klaplace

# imatch

## Purpose

Search for matching region

## Synopsis

```
[xm, s] = imatch(im1, im2, x, y, w2, search)
```

## Description

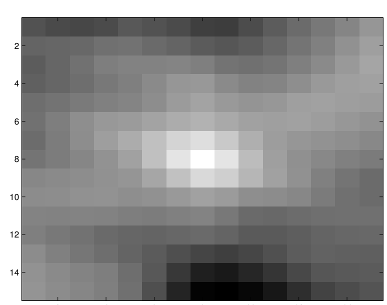
Find the best matchin in `im2` for the square region in image `im1` centered at `(x, y)` of half-width `w2`. The search is conducted over the region in `im2` centered at `(x, y)` with bounds `search`. `search = [xmin xmax ymin ymax]` relative to the `(x, y)`. If `search` is scalar it searches `[-s s -s s]`.

Similarity is computed using the zero-mean normalized cross-correlation similarity measure

$$ZNCC(A,B) = \frac{\sum(A_{ij} - \overline{A})(B_{ij} - \overline{B})}{\sqrt{\sum(A_{ij} - \overline{A}) \sum(B_{ij} - \overline{B})}}$$

where $\overline{A}$ and $\overline{B}$ are the mean over the region being matched. This measure is invariant to illumination offset. While computationally complex it yields a well bounded result, simplifying the decision process. Result is in the range -1 to 1, with 1 indicating identical pixel patterns.

Returns the best fit `xm = [dx dy cc]` where `(dx, dy)` are the coordinates of the best fit with respect to `(x, y)` and `cc` is the corresponding cross-correlation score. Optionally it can return the cross-correlation score at every point in the search space. This correlation surface can be used to interpolate the coordinate of the peak.




## Examples

Search for matching region in the Lena test image.

```
>> lena = loadpgm('lena');
>> [xm,s] = imatch(lena, lena, 200, 200, 7, 7);
>> xm
xm =
         0         0    1.0000
>> idisp(s)
```

The best match occurs as expected at coordinate $(0, 0)$ since the two images are identical. The correlation surface is shown above.

**See Also** zncc, subpixel

# imoments

**Purpose** Compute image moments

**Synopsis**

```
F = imoments(image)
F = imoments(rows, cols)
F = imoments(rows, cols)
```

**Description** Returns a structure array containing moments upto second order for the non-zero pixels in the binary image `image`. The non-zero pixels are considered as a single 'blob' but no connectivity analysis is performed. The actual pixel values are used as pixel weights. In the second form the row and column coordinates of the region's pixels can be given instead of an image.

For a binary image the return structre `F` contains simple 'blob' features `F.area`, `F.x`, `F.y`, `F.a`, `F.b` and `F.theta` where `(xc, yc)` is the centroid coordinate, `a` and `b` are axis lengths of the "equivalent ellipse" and `theta` is the angle of the major ellipse axis to the horizontal axis.

For a greyscale image `area` is actually the sum of the pixel values, and the centroid is weighted by the pixel values. This can be useful for sub-pixel estimation of the centroid of a blob taking into account the edge pixels which contain components of both foreground and background object.

The structure also contains the raw moments `F.m00`, `F.m10`, `F.m01`, `F.m20`, `F.m02`, and `F.m11`.

**Examples** An example is to compute the moments of a particular region label. First we create a test pattern of an array of large dots.

```
>> image = testpattern('dots', 256, 50, 10);
>> l = ilabel(image);
>> binimage = (l == 3); % look for region 3
>> imoments(binimage)
ans =
```

```
     area: 81
        x: 75
        y: 25
        a: 5.0966
        b: 5.0966
    theta: 0
      m00: 81
      m01: 2025
      m10: 6075
      m02: 51151
      m20: 456151
      m11: 151875
```

```
or

>> [r,c] = find(binimage);
>> imoments(r,c)
     .
     .
```

**See Also** markfeatures, ilabel, mpq

# imono

**Purpose** Convert color image to greyscale

**Synopsis**

```
im = imono(rgb)
```

**Description** Returns the greyscale information from the 3-plane RGB image `rgb`.

**See Also** rgb2hsv

# imorph

**Purpose** Grey scale morphology

**Synopsis**

```
Im = imorph(I, se, op)
Im = imorph(I, se, op, edge)
```

**Description** Perform greyscale morphological filtering on `I` with the structuring element defined by the non-zero elements of `se`. The supported operations are minimum, maximum or difference (maximum - minimum) specified by `op` values of `'min'`, `'max'` and `'diff'` respectively.

Square structuring elements can be created conveniently using `ones(N,N)` and circular structuring elements using `kcircle(N)`.

Edge handling flags control what happens when the processing window extends beyond the edge of the image. `edge` is either:

`'border'` (default) the border value is replicated

`'none'` pixels beyond the border are not included in the window

`'trim'` output is not computed for pixels whose window crosses the border, hence the output image is reduced all around by half the window size.

`'wrap'` the image is assumed to wrap around, left to right, top to bottom.

**See Also** iopen, iclose, kcircle

# inormhist

**Purpose** Histogram normalization

**Synopsis** `hn = inormhist(image)`

**Description** Returns the histogram normalized version of `image`. The grey levels of the output image are spread equally over the range 0 to 255. This transform is commonly used to enhance contrast in a dark image.

**Examples** Compare raw and histogram normalized images of Lena.

```
>> lena = loadpgm('lena');
>> idisp(lena);
>> idisp( inormhist(lena) );
```

**See Also** ihist

# invcamcal

**Purpose** Inverse camera calibration

**Synopsis**

```
[P R K delta] = invcamcal(C)
```

**Description** `invcamcal` estimates the camera extrinsic and intrinsic parameters from a $3 \times 4$ camera calibration matrix. `P` is a vector of estimated camera location, and `R` is the estimated rotation matrix. `K` is the estimated scale factors [alphax*f alphay*f] where f is camera focal length and alphax and alphay are the pixel pitch in the X and Y directions. `delta` is an estimate of the 'goodness' of the calibration matrix and is interpretted as the cosine of the angle between the X and Y axes, and is ideally 0.

**See Also** camcalp, camcald, camcalt

**References**

S. Ganapathy, "Camera location determination problem," Technical Memorandum 11358-841102-20-TM, AT&T Bell Laboratories, Nov. 1984.

S. Ganapathy, "Decomposition of transformation matrices for robot vision," in *Proc. IEEE Int. Conf. Robotics and Automation*, pp. 130–139, 1984.

# invhomog

**Purpose** Inverse homography

**Synopsis** `s = invhomog(H)`

**Description** Estimates the rotation and translation (upto scale) of the Cartesian motion corresponding to the given homography H of points in a plane.

There are in general multiple solutions, so the return is a structure array. Disambiguating the solutions is up to the user!

The elements of the structure are:

| | |
|---|---|
| R | $3 \times 3$ orthonormal rotation matrix |
| t | vector translation direction |
| n | vector normal of the plane |
| d | distance from the plane (not to scale). |

$(R, t)$ are the Cartesian transformation from the first camera position to the second.

**Limitations** Doesn't seem to work well for cases involving rotation.

**Cautionary** Not entirely sure this is correct. Use with caution.

**See Also** homography

**References**

O. Faugeras and F. Lustman, "Motion and structure from motion in a piecewise planar environment," *Int. J. Pattern Recognition and Artificial Intelligence*, no. 3, pp. 485–508, 1988.

# iopen

## Purpose

Grey scale morphological opening

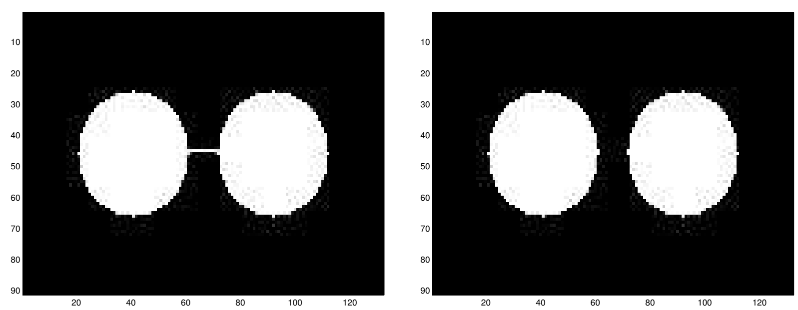
## Synopsis

```
im2 = iopen(im)
im2 = iopen(im, se)
im2 = iopen(im, se, N)
```

## Description

Perfoms a greyscale morphological opening on the image `im` using structuring element `se` which defaults to `ones(3,3)`. The operation comprises `N` (default 1) consecutive erosions followed by `N` consectutive dilations.

Square structuring elements can be created conveniently using `ones(N,N)` and circular structuring elements using `kcircle(N)`.



## Examples

Using morphological opening to separate two blobs without changing their size or shape.

```
>> idisp(im);
>> idisp( iopen( im, kcircle(3) ) );
```

## See Also

imorph, iclose, kcircle

# ipyramid

**Purpose** Pyramid decomposition

**Synopsis**

```
Ip = ipyramid(I)
Ip = ipyramid(I, sigma)
Ip = ipyramid(I, sigma, N)
```

**Description** `pyramid` returns a pyramidal decomposition of the input image `I`. Gaussian smoothing with $\sigma =$ `sigma` (default is 1) is applied prior to each decimation step. If `N` is specified then only `N` steps of the pyramid are computed, else decomposition continues down to a $1 \times 1$ image.

The result is a cell array of images in reducing size order.


**Examples** Let's place each of the images horizontally adjacent and view the resulting image.

```
>> lena = loadpgm('lena');
>> p = ipyramid(lena, 5);
>> pi = zeros(512, 992);
>> w = 1;
>> for i=1:5,
>>   [nr,nc] = size(p{i});
>>   pi(1:nr,w:w+nc-1) = p{i};
>>   w = w + nc;
>> end
```

```
>> image(pi)
```

**See Also** ishrink, kgauss

# irank

## Purpose

Fast neightbourhood rank filter

## Synopsis

```
Ir = irank(I, order, se)
Ir = irank(I, order, se, nbins)
Ir = irank(I, order, se, edge)
```

## Description

`irank()` performs a rank filter over the neighbourhood specified by `se`. The `order`'th value in rank (1 is lowest) becomes the corresponding output pixel value. A histogram method is used with `nbins` (default 256).

Square neighbourhoods can be specified conveniently using `ones(N,N)` and circular neighbourhoods using `kcircle(N)`.

Edge handling flags control what happens when the processing window extends beyond the edge of the image. `edge` is either:

`'border'` (default) the border value is replicated

`'none'` pixels beyond the border are not included in the window

`'trim'` output is not computed for pixels whose window crosses the border, hence the output image is reduced all around by half the window size.

`'wrap'` the image is assumed to wrap around, left to right, top to bottom.

## Examples

To find the median over a $5 \times 5$ square window. After sorting the 25 pixels in the neighbourhood the median will be given by the 12th in rank.

```
>> ri = irank(lena, 12, ones(5,5));
image pixel values: 37.000000 to 226.000000
>> idisp(ri);
```

## See Also

kcircle

# iroi

## Purpose

Select region of interest

## Synopsis

```
subimage = iroi(image)
[subimage,corners] = iroi(image)
subimage = iroi(image, corners)
```

## Description

The first two forms display the image and a rubber band box to allow selection of the region of interest. Click on the top-left corner then stretch the box while holding the mouse down. The selected `subimage` is output and optionally the coordinates, `corners` of the region selected which is of the form [top left; bottom right].

The last form uses a previously created region matrix and outputs the corresponding subimage. Useful for chopping the same region out of a different image. Cropping is applied to all planes of a multiplane image.

Works with color images.

## See Also

idisp

# ishrink

**Purpose** Smooth and decmimate an image

**Synopsis**

```
Is = ishrink(I)
Is = ishrink(I, sigma)
Is = ishrink(I, sigma, N)
```

**Description** Return a lower resolution representation of the image `I`. The image is first smoothed by a Gaussian with $\sigma =$ `sigma` and then subsampled by a factor `N`. Default values are `sigma = 2` and `N = 2`.



**Examples**

```
>> lena = loadpgm('lena');
>> size(lena)
ans =
   512   512

>> s = ishrink(lena, 2, 4);
>> size(s)
ans =
   128   128

>> idisp(s)
```

**See Also** kgauss, ipyramid

# isimilarity

**Purpose** Zero-mean normalized cross-correlation

**Synopsis**

```
m = isimilarity(im1, im2, c1, c2, w)
```

**Description** Compute the similarity between two equally sized image patches $(2w+1) \times (2w+1)$ centered at coordinate `c1` in image `im1`, and coordinate `c2` in image `im2`.

Similarity is computed using the zero-mean normalized cross-correlation similarity measure

$$ZNCC(A,B) = \frac{\sum(A_{ij}-\overline{A})(B_{ij}-\overline{B})}{\sqrt{\sum(A_{ij}-\overline{A})\sum(B_{ij}-\overline{B})}}$$

where $\overline{A}$ and $\overline{B}$ are the mean over the region being matched. This measure is invariant to illumination offset. While computationally complex it yields a well bounded result, simplifying the decision process. Result is in the range -1 to 1, with 1 indicating identical pixel patterns.

**See Also** zncc

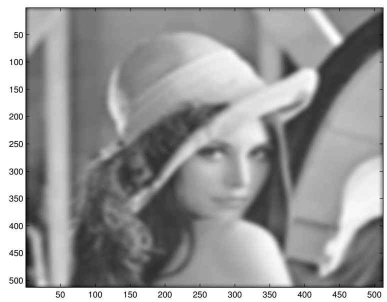
 

# ismooth

**Purpose** Gaussian filter

**Synopsis** `G = ismooth(image, sigma)`

**Description** Convolves all planes of the image with a Gaussian kernel of specified `sigma`.




**Examples** Smooth the Lena image.

```
>> lena = loadpgm('lena');
>> idisp( ismooth( lena, 4) )
```

**See Also** conv2, kgauss

# isobel

## Purpose

Sobel filter

## Synopsis

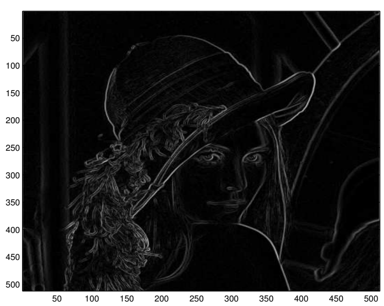
```
Is = isobel(I)
Is = isobel(I, Dx)
[ih,iv] = isobel(I)
[ih,iv] = isobel(I, Dx)
```

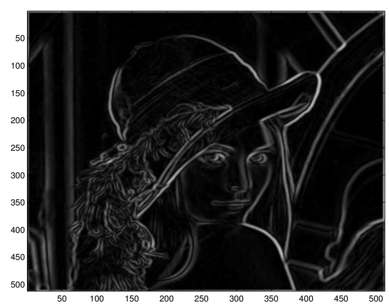
## Description

Returns a Sobel filtered version of image `I` which is the norm of the vertical and horizontal gradients. If `Dx` is specified this x-derivative kernel is used instead of the default:

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

With two output arguments specified the function will return the vertical and horizontal gradient images.




## Examples

```
>> lena = loadpgm('lena');
>> im = isobel(lena);
>> idisp(im)
>> im2 = isobel(lena, kdgauss(2));
>> idisp(im2)
```

## Cautionary

The Sobel operator is a simple edge detector and has the disadvantage of giving fat double edges.

**See Also** kdgauss

# istretch

**Purpose** Image linear normalization

**Synopsis**

```
hn = istretch(image)
hn = istretch(image, newmax)
```

**Description** Returns a normalized image in which all pixels lie in the range 0 to 1, or 0 to `newmax`.

**See Also** inormhist

# ivar

## Purpose

Fast neighbourhood variance/kurtosis/skewness

## Synopsis

```
Im = ivar(I, se, op)
Im = ivar(I, se, op, edge)
```

## Description

Computes the specified statistic over the pixel neighbourhood specified by `se` and this becomes the corresponding output pixel value. The statistic is specified by `op` which is either `'var'`, `'kurt'`, or `'skew'`.

Square neighbourhoods can be specified conveniently using `ones(N,N)` and circular neighbourhoods using `kcircle(N)`.

Edge handling flags control what happens when the processing window extends beyond the edge of the image. `edge` is either:

`'border'` (default) the border value is replicated

`'none'` pixels beyond the border are not included in the window

`'trim'` output is not computed for pixels whose window crosses the border, hence the output image is reduced all around by half the window size.

`'wrap'` the image is assumed to wrap around, left to right, top to bottom.

## Limitations

This is a very powerful and general facility but it requires that the MATLAB interpretter is invoked on every pixel, which impacts speed.

## See Also

kcircle

# iwindow

**Purpose** General function of a neighbourhood

**Synopsis**

```
Im = iwindow(I, se, func)
Im = iwindow(I, se, func, edge)
```

**Description** For every pixel in the input image it takes all neighbours for which the corresponding element in `se` are non-zero. These are packed into a vector (in raster order from top left) and passed to the specified Matlab function. The return value becomes the corresponding output pixel value.

Square neighbourhoods can be specified conveniently using `ones(N,N)` and circular neighbourhoods using `kcircle(N)`.

Edge handling flags control what happens when the processing window extends beyond the edge of the image. `edge` is either:

`'border'` (default) the border value is replicated

`'none'` pixels beyond the border are not included in the window

`'trim'` output is not computed for pixels whose window crosses the border, hence the output image is reduced all around by half the window size.

`'wrap'` the image is assumed to wrap around, left to right, top to bottom.

**Examples** To compute the mean of an image over an annular window at each point.

```
>> se = kcircle([5 10]);
>> out = iwindow(image, se, 'mean');
```

**Limitations** This is a very powerful and general facility but it requires that the MATLAB interpretter is invoked on every pixel, which impacts speed.

**See Also** iopen, iclose, kcircle

# klaplace

**Purpose** Laplacian kernel

**Synopsis** `k = klaplace`

**Description** Returns the Laplacian kernel

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

**Examples**

```
>> klaplace
ans =
     0    -1     0
    -1     4    -1
     0    -1     0
```

**See Also** conv2, klog, kgauss, ilap

# kcircle

## Purpose

Create a circular mask

## Synopsis

```
C = kcircle(r)
C = kcircle(r, w)
```

## Description

Returns a circular mask of radius `r`. `C` is a $(2r+1) \times (2r+1)$ matrix, or in second case a $w \times w$ matrix. Elements are one if on or inside the circle, else zero.

If `r` is a 2-element vector then it returns an annulus of ones, and the two numbers are interpretted as inner and outer radii.

Useful as a circular structuring element for morphological filtering.

## Examples

To create a circular mask of radius 3

```
>> kcircle(3)
ans =

     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
```

## See Also

imorph, iopen, iclose

# kdgauss

## Purpose

Create a 2D derivative of Gaussian filter

## Synopsis

```
G = kgauss(sigma)
G = kgauss(sigma, w)
```

## Description

Returns a $(2w+1) \times (2w+1)$ matrix containing the x-derivative of the 2D Gaussian function

$$g(x,y) = -\frac{x}{2\pi\sigma^2} e^{\frac{x^2+y^2}{2\sigma^2}}$$

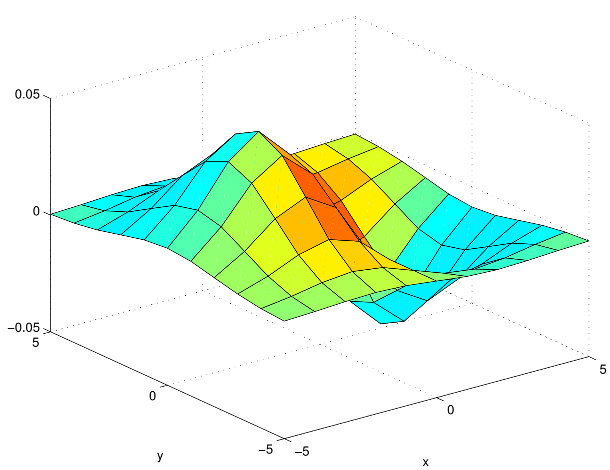
symmetric about the center pixel of the matrix. This kernel is useful for computing smoothed deriviatives. The y-derivative of the Gaussian is simply the transform of this function.

Standard deviation is `sigma`. If `w` is not specified it defaults to $2\sigma$.

This kernel can be used as an edge detector and is sensitive to edges in the x-direction.

## Examples

```
>> g = kdgauss(2, 5);
>> surfl([-5:5], [-5:5], g);
```




## See Also

conv2

# kdog

## Purpose

Create a 2D difference of Gaussian filter

## Synopsis

```
LG = kdog(sigma1, sigma2)
LG = kdog(sigma1, sigma2, w)
```

## Description

Returns a $(2w+1) \times (2w+1)$ matrix containing the difference of two 2-D Gaussian functions.

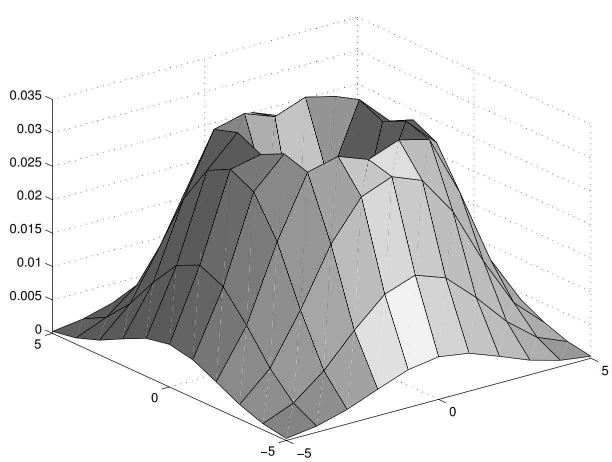
$$DoG(x,y) = \frac{1}{2\pi}\left(e^{-\frac{x^2+y^2}{2\sigma_1^2}} - e^{-\frac{x^2+y^2}{2\sigma_2^2}}\right)$$

The kernel is symmetric about the center pixel of the matrix. If `w` is not specified it defaults to twice the largest $\sigma$.

This kernel can be used as an edge detector and is sensitive to edges in any direction.

## Examples

```
>> dog = kdog(2, 1.5, 5);
>> surfl([-5:5], [-5:5], dog);
```




## See Also

conv2, kgauss, klaplace, klog

# kgauss

## Purpose

Create a 2D Gaussian filter

## Synopsis

```
G = kgauss(sigma)
G = kgauss(sigma, w)
```

## Description

Returns a $(2w+1) \times (2w+1)$ matrix containing a 2-D Gaussian function

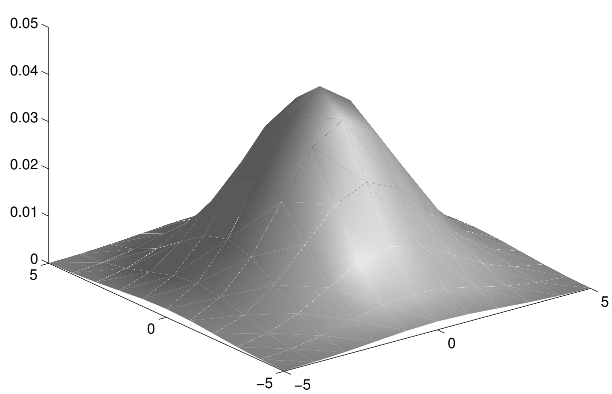
$$G(x,y) = \frac{1}{2\pi} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

symmetric about the center pixel of the matrix. The volume under the curve is unity.

Standard deviation is `sigma`. If `w` is not specified it defaults to $2\sigma$.

## Examples

```
>> g = kgauss(2, 5);
>> surfl([-5:5], [-5:5], g);
```




## See Also

conv2

# klog

**Purpose** Create a 2D Laplacian of Gaussian filter

**Synopsis**

```
LG = klog(sigma)
```

**Description** Returns a $(2w+1) \times (2w+1)$ matrix containing the Laplacian of the 2-D Gaussian function.

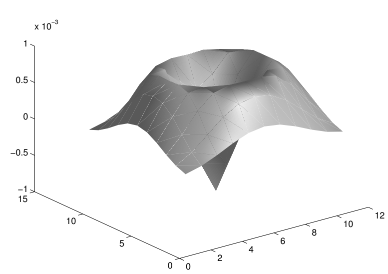
$$LoG(x,y) = \frac{-1}{2\pi\sigma^4}(2 - \frac{x^2+y^2}{\sigma^2})e^{-\frac{x^2+y^2}{2\sigma^2}}$$

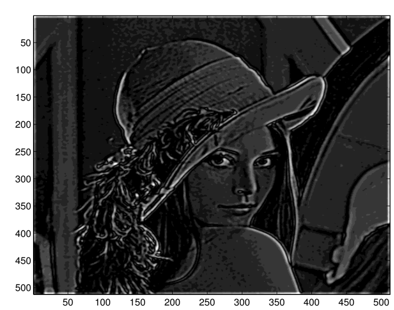
The kernel is symmetric about the center pixel of the matrix. Standard deviation is `sigma`. If `w` is not specified it defaults to $2\sigma$.

This kernel can be used as an edge detector and is sensitive to edges in any direction.

**Examples**

```
>> lg = klog(2, 5);
>> surfl([-5:5], [-5:5], lg);
>> lena = loadpgm('lena');
>> loglena = conv2(lena, klog(2), 'same');
>> image(abs(zl))
>> colormap(gray(15))
```




**See Also** conv2, kgauss, klaplace, ilap

# kmeans

**Purpose** k-means clustering

**Synopsis**

```
[c,s] = kmeans(x, N)
[c,s] = kmeans(x, N, x0)
```

**Description** Find `N` clusters for the data `x`. Returns `c` the centers of each cluster as well as `s` which contains the cluster index for each corresponding element of `x`.

The initial cluster centers are uniformly spread over the range of `x` but can be specified by an optional N-element vector `x0`.

The clustering is performed only with respect to data values, not spatially.

**Examples** Can be used for image segmentation, to find pixels with similar greyscale or hue values. Segment the Lena image into 4 greyscale bands:

```
>> [c,s] = kmeans(lena, 4);
>> c
c =
   63.7419  108.5058  143.6484  183.0839
>> idisp(s);
```


The pixels have been clustered into 4 groups with the center values shown.

**Limitations** This is an iterative algorithm which is very slow as an m-file.

**References** Tou and Gonzalez, Pattern Recognition Principles, pp 94

# loadinr

**Purpose** Load INRIMAGE format image

**Synopsis**

```
I = loadinr(fname)
```

**Description** Returns a matrix containing a gray scale image read from an INRIMAGE format file with the specified name. If no extension is provided an extension of `.inr` is appended. This is a binary floating point file format developed at INRIA. Returns `[]` if the file cannot be opened.

**Limitations** Only simple 2D images are supported in this implementation.

**See Also** saveinr

# loadpgm

**Purpose** Load PGM format image (P2 or P5)

**Synopsis** `I = loadpgm(fname)`

**Description** Returns a matrix containing the gray scale image read from the specified file. If no extension is provided an extension of `.pgm` is appended.

The given `fname` is globbed and if it matches more than 1 file then the files are read sequentially and a 3-dimensional array is returned where the last index is the frame number.

If no file is given then a GUI file browser is popped up.

The header parsing is fairly complete and allows for embedded comments which complicate what would otherwise be a simple header to read. Returns `[]` if the file cannot be opened.

**Examples** To compute the mean of an image over an annular window at each point.

```
>> lena = loadpgm('lena');
>> idisp(lena);
```

**Limitations** Currently does not return the comment field from the file header.

**See Also** savepnm

# loadppm

## Purpose

Load PPM format image (P3 or P6)

## Synopsis

```
rgb = loadppm(fname)
```

## Description

Returns a 3-dimensional matrix containing the red, green, and blue planes of the image read from the specified file. If no extension is provided an extension of `.ppm` is appended.

The given `fname` is globbed and if it matches more than 1 file then the files are read sequentially and a 4-dimensional array is returned where the last index is the frame number.

If no file is given then a GUI file browser is popped up.

The header parsing is fairly complete and allows for embedded comments which complicate what would otherwise be a simple header to read. Returns `[]` if the file cannot be opened.

## Limitations

Currently does not return the comment field from the file header.

## See Also

savepnm, loadpgm, loadinr

# markfeatures

## Purpose

Mark features

## Synopsis

```
markfeatures(xy)
markfeatures(xy, N)
markfeatures(xy, N, marker)
markfeatures(xy, N, marker, label)
```

## Description

Mark features on the current figure. The features are specified by `xy` which can be an $n \times 2$ matrix, with one row per feature, or a structure vector where each element has a `x` and `y` element. The second form limits the display to at most `N` features, if `N` is zero, then all features are displayed.

The third form allows the marker to be specified with standard Matlab linestyle specifiers to indicate shape and color.

The fourth form causes the features to be numbered. `label` is a 2-element cell array where `label1` is the font size and `label2` is the color.

## Limitations

The feature labelling should better position the label.

## Examples

See the example for `iharris`.

## See Also

iharris

# max2d

**Purpose** Find maximum point in image

**Synopsis**

```
[r,c] = max2d(image)
```

**Description** Return the interpolated coordinates (r,c) of the greatest peak in image. Useful for finding peaks in a Hough transform accumulator.

**See Also** ihough

# mkcube

## Purpose

Create a cube

## Synopsis

```
c = mkcube
c = mkcube(s)
c = mkcube(s, center)
```

```
c = mkcube2
c = mkcube2(s)
c = mkcube2(s, center)
```

## Description

`mkcube` returns an $8 \times 3$ matrix where each row is the coordinates of a vertex of the cube.

`mkcube2` returns a $12 \times 6$ matrix where each row corresponds to one edge of the cube. The first three elements of each row are the start coordinate of the edge and the last three are the end coordinate.

The cube has a side length `s` (default 1) and is centered at `center` (default $[0\,0\,0]$).

## See Also

camera

# mpq, upq, npq

**Purpose** Compute moments of a polygon

**Synopsis**

```
m = mpq(iv, p, q)
m = upq(iv, p, q)
m = npq(iv, p, q)
```

**Description** `mpq` computes the pq'th moment of the polygon whose vertices are iv.

`upq` and `npq` compute the central and normalized-central moments respectively.

**Cautionary** Note that the points must be sorted such that they follow the perimeter in sequence (either clockwise or anti-clockwise).

**See Also** imoments

**References**

J. Wilf and R. Cunningham, "Computing region moments from boundary representations," JPL 79-45, NASA JPL, Nov. 1979.

# pnmfilt

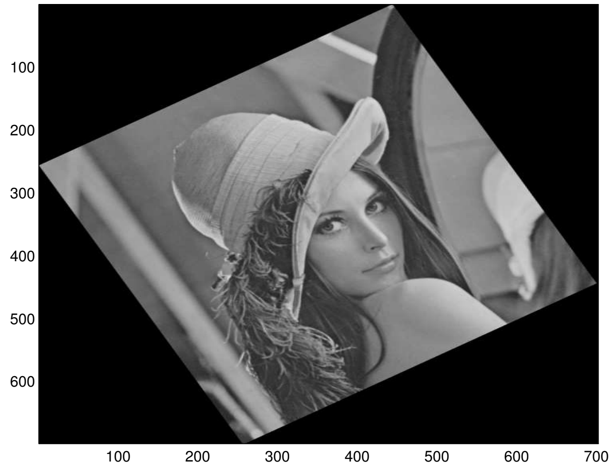
**Purpose** Pipe an image through Unix filter

**Synopsis** `im2 = pnmfilt(image, cmd)`

**Description** Pipes the image through a Unix filter program. The image is written in PGM (P5) format or PPM (P6) format to stdin of the specified command, and its output on stdout (assumed to be PNM format) is returned by this function. Provides access to many preexisting image program functions that are part of the PBMplus, ImageMagick and Khoros suites.

**Examples** To rotate an image we can make use of the `pnmrotate` utility

```
>> lena = loadpgm('lena');
>> rlena = pnmfilt(lena, 'pnmrotate 30');  % rotate by 30 de
>> image(rlena);
>> colormap(gray(256))
```




**Limitations** The mechanism is not quick, but it is convenient. Unfortunately MATLAB doesn't support proper pipes (could be done with a mex-file...) so temporary files are used.

**See Also** idisp, xv, savepnm

# pulnix

**Purpose** Model for Pulnix camera and Digimax digitizer

**Synopsis**

```
cp = pulnix
```

**Description** Returns the camera calibration matrix for a Pulnix TN-6 camera with an 8mm lense and a Datacube Digimax digitizer.

The camera parameter object `cp` has elements:

| | |
|---|---|
| `cp.f` | focal length (m) |
| `cp.u0` | principal point u-coordinate (pix) |
| `cp.v0` | principal point v-coordinate (pix) |
| `cp.px` | horizontal pixel pitch (pix/m) |
| `cp.py` | vertical pixel pitch (pix/m) |

**Examples** (

)

```
>> pulnix
ans =

   f: 0.0078
  px: -79200
  py: -120500
  u0: 274
  v0: 210
```

**References** P. I. Corke, *Visual Control of Robots: High-Performance visual servoing*. Mechatronics, Research Studies Press (John Wiley), 1996.

**See Also** camcalp

# rgb2xyz

**Purpose** RGB color space to CIE XYZ

**Synopsis**

```
xyz = rgb2xyz(r, g, b)
xyz = rgb2xyz(rgb)
```

**Description** Returns a row vector of the CIE 1931 XYZ values corresponding to the color components `r`, `g`, and `b` which can also be given as a 3-element row vector `rgb`. If the components have more than one row, the result will be a matrix with one row corresponding to each input row.

**See Also** rgb2hsv

**References** An excellent introduction to color spaces can be found at `http://www.faqs.org/faqs/graphics/colorspace-faq`

# rluminos

## Purpose

Relative photopic luminosity

## Synopsis

```
rluminos(lambda)
```

## Description

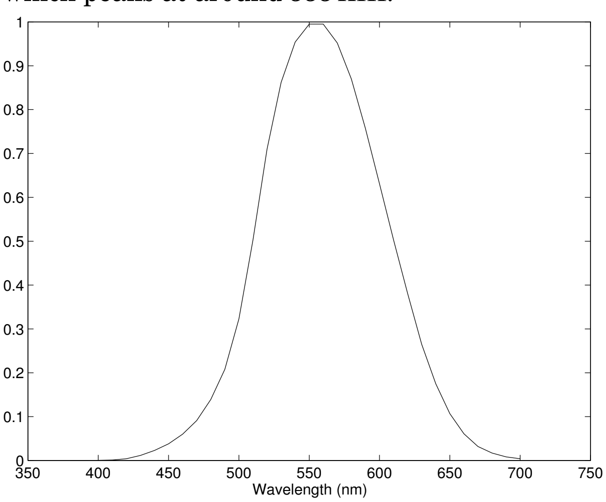
`rluminos` returns the relative photopic (light adjusted cone response) luminosity response of the human eye. CIE luminosity is obtained by multiplying by $673\,\mathrm{lumens/W}$.

## Examples

To show this response over visible wavelengths

```
>> l = [380:10:700]'*1e-9;         % visible spectrum
>> r = rluminos(l);
>> plot(l*1e9, r)
>> xlabel('Wavelength (nm)')
```

which peaks at around $555\,\mathrm{nm}$.




## Algorithm

Evaluated using the Y component of the CIE XYZ color matching function.

## See Also

cmfxyz

# saveinr

**Purpose** Save INRIMAGE format image

**Synopsis** `saveinr(fname, I)`

**Description** `saveinr` saves a matrix containing a gray scale image in an INRIMAGE format file with the specified name. If no extension is provided an extension of `.inr` is appended. This is a binary floating point file format developed at INRIA.

**Limitations** Only simple 2D images are supported in this implemenation.

**See Also** loadinr

# savepnm

**Purpose** Save PNM format image

**Synopsis**

```
savepnm(fname, I)
savepnm(fname, I, comment)
```

**Description** `savepnm` saves a matrix containing an image in binary greyscale (P5) or RGB color (P6) format to the file with the specified name. The optional comment will be embedded in the image header consistant with the PBM file format.

**See Also** loadpgm, loadppm

# solar

**Purpose** Solar irrandiance spectrum

**Synopsis**

```
p = solar(lambda)
```

**Description** Return solar irradiance in $\mathrm{W/m^2/nm}$ for wavelength `lambda`. `lambda` maybe a vector.

**References** `http://www.asdi.com/apps/arm.html`, figure 1.

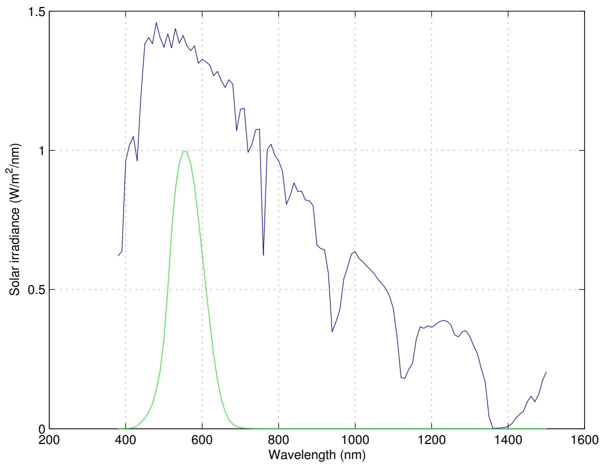
**Limitations** Solar irrandiance depends on many things: cloud, time, location etc. and this should be taken as a rough guide only.

**Examples** To show solar irradiance response over visible and infra-red wavelengths.

```
>> l = [380:10:1500]'*1e-9;        % visible and IR spectrum
>> s = solar(l);
>> r = rluminos(l);
>> plot(l*1e9, [s r])
>> xlabel('Wavelength (nm)')
>> ylabel('Solar irradiance (W/m^2/nm)')
```

along with the human visible (photopic) response.




**See Also** blackbody

# subpixel

**Purpose** Subpixel interpolation of peak

**Synopsis**

```
[dxr, dyr] = subpixel(surf)
[dxr, dyr] = subpixel(surf, dx, dy)
```

**Description** Given a 2-d surface `surf` refine the estimate of the peak to subpixel precision using first-order differences. The peak may be given by `(dx, dy)` or searched for.

To find a minimum, call the function with `-surf`.

Useful to find peaks in correlation surfaces or Hough accumulator peaks.

**See Also** max2d, imatch, ihough

# testpattern

**Purpose** Create a variety of useful test patterns

**Synopsis**

```
im = testpattern('rampx', w, ncycles)
im = testpattern('rampy', w, ncycles)

im = testpattern('sinx', w, ncycles)
im = testpattern('siny', w, ncycles)

im = testpattern('dots', w, pitch, diam)

im = testpattern('squares', w, pitch, s)

im = testpattern('line', w, theta, c)
```

**Description** Returns an image of size $w \times w$ containing a testpattern. If `w` is 2-dimensional it specifies the number of rows and colums of `im`.

With no output arguments the testpattern is displayed using `idisp()`.

The first four forms create greyscale images with triangular or sinusoidal patterns. For the ramp values are in the range $[0, 1]$ and for the sinuoids in the range $[-1, 1]$. If not specified `ncycles` corresponds to 1.

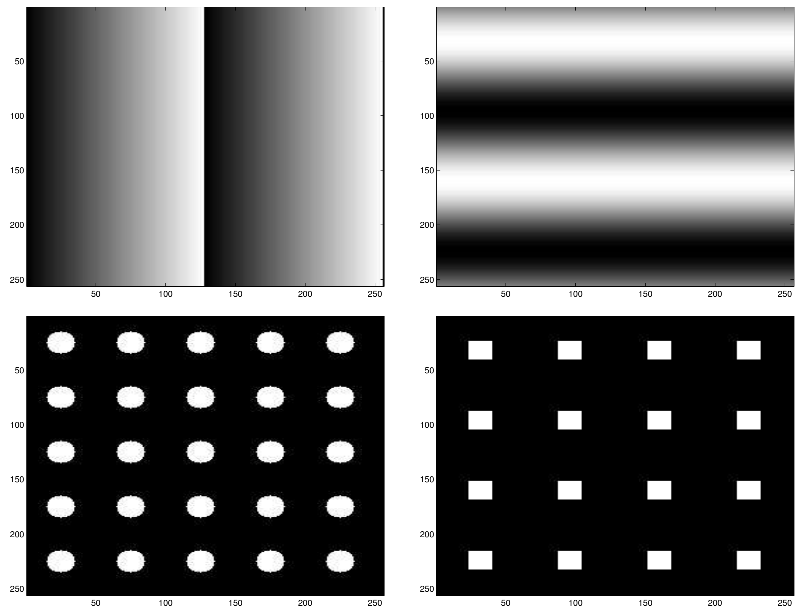
The dot and square test patterns are binary images with pixels either 0 or 1. They are specified in terms of pitch, distance between centres, and diameter `diam` or side length `s`.

The line is described by

$$v = \tan\theta u + c$$

where $v$ and $u$ are row and column respectively, and `theta` is specified in radians. Pixels on the line are set to one, elsewhere to zero.




## Examples

```
>> testpattern('rampx', 256, 2)
>> testpattern('siny', 256, 2)
>> testpattern('dots', 256, 50, 20)
>> testpattern('squares', 256, 64, 16)
```

# trainseg

**Purpose** Train an rg-space color segmentation table

**Synopsis**

```
map = trainseg(rgb)
```

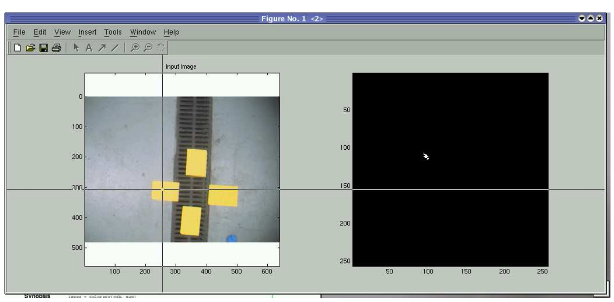
**Description** Each pixel of the input color image `rgb` is converted to normalized $(r,g)$ coordinates

$$r = \frac{R}{R+G+B} \tag{6}$$

$$g = \frac{G}{R+G+B} \tag{7}$$

The function displays a new figure with two windows, the left-hand is the original color image and the right-hand is the color segmentation map. The user clicks on pixels in the left-hand window that belong to the target set and the corresponding values in rg-space are set in the right-hand image. The right-hand image is used subsequently for segmentation.

The output is a $256 \times 256$ image with pixel values that are either 0 or 1. Typically the output image would be further processed with morphological closing to create a solid region in rg-space that represents the range of target colors.



**Examples** Train a color segmentation table of the yellow targets

```
>> targ = loadppm('target.ppm');
>> map = trainseg(targ);
```

Every mouse click in the left-hand window adds a point to the right-hand window. By clicking on many points within the target regions we can build up a generalization of its color, as shown by the finite sized region in the right-hand window.

**See Also** colorseg,imorph

# visjac_p

**Purpose** Visual Jacobian matrix

**Synopsis**

```
J = visjac_p(uv, z)
J = visjac_p(uv, z, f)
J = visjac_p(uv, z, cp)
```

**Description** Returns a $2 \times 6$ visual motion Jacobian that maps relative camera motion

$$\begin{bmatrix} \dot{u} \\ \dot{v} \end{bmatrix} = \mathbf{J} \begin{bmatrix} T_x \\ T_y \\ T_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

to image plane velocity for the point `uv` = $(u, v)$, where the image Jacobian is

$$\mathbf{J} = \begin{bmatrix} \frac{\lambda}{z} & 0 & \frac{-u}{z} & \frac{-uv}{\lambda} & \frac{\lambda^2 + u^2}{\lambda} & -v \\ 0 & \frac{\lambda}{z} & \frac{-v}{z} & \frac{-\lambda^2 - v^2}{\lambda} & \frac{uv}{\lambda} & u \end{bmatrix} \tag{8}$$

For 3 or more points the Jacobians can be stacked and used to solve for relative motion given observed image plane motion.

The Jacobian is a function of the camera parameters which can be given as just a focal length in pixels `f`, or as a full camera parameter object `cp`:

| | |
|---|---|
| `cp.f` | focal length (m) |
| `cp.u0` | principal point u-coordinate (pix) |
| `cp.v0` | principal point v-coordinate (pix) |
| `cp.px` | horizontal pixel pitch (pix/m) |
| `cp.py` | vertical pixel pitch (pix/m) |

**Examples**

```
>> visjac_p([0 0], 2, pulnix)

  ans =

   1.0e+11 *

   -0.0000         0   -0.0001    5.8425   -7.6231   -0.0002
         0   -0.0000   -0.0001    6.8129   -8.8892    0.0003
```

Which indicates that visual motion will be dominated by $\omega_x$ and $\omega_y$ camera motion.

**See Also** camera, pulnix

**References**

S. Hutchinson, G. Hager, and P. Corke, "A tutorial on visual servo control," *IEEE Transactions on Robotics and Automation*, vol. 12, pp. 651–670, Oct. 1996.

# xv

**Purpose** Display image using XV

**Synopsis** `xv(image)`

**Description** `xv` ships the image off to a background XV process. XV is a great shareware X program for image viewing, manipulation and format conversion. This script can be easily edited to use your favourite image browser, such as `display`, `eog`, `kview` etc.

**References** XV is available from `http://www.trilon.com/xv/`

**See Also** pnmfilt

# webcam

**Purpose** Load an image from a web camera

**Synopsis**

```
im = webcam(url)
```

**Description** Returns an image from the web camera with the specified URL. Note that web cameras vary widely in the way they are communicated with. Some allow for control of many camera parameters such as pan, tilt and zoom by extra arguments in the URL.

**Examples** Read an image from a Canon web camera

```
>> im = webcam('http://10.0.0.80/-wvhttp-01-/GetStillI
>>
>> im = webcam('http://www.thesurfclub.com.au/Webcam/i
>> idisp(im);
```

The first example also sets the pan angle to 5. The second example loads an image from a webcam at a beach 100km from my lab! Not a good day for the beach today...


**See Also** firewire

# yuvopen

## Purpose

Open a YUV4MPEG format file

## Synopsis

```
h = yuvopen(file)
```

## Description

Opens a file organized in YUV4MPEG format. This is a raw uncompressed file in 4:2:0 format with YUV color encoding. Returns a handle to the stream that is used for subsequent read operations.

This file format is used as a precursor to mpeg encoding and can be played by `mplayer` and transcoded by `ffmpeg`. The stream header is saved in `h.hdr`.

See the Berkeley mpeg tools manual for more details.

## Limitations

Assumes the file is in yuv420 format.

## See Also

yuvread, yuv2rgb

# yuvread

## Purpose

Read frame from a YUV4MPEG format file

## Synopsis

```
[y,u,v] = yuvread(h)
[y,u,v] = yuvread(h, skip)
[y,u,v,hdr] = yuvread(h, skip)
```

## Description

Reads the next frame from the YUV4MPEG format file opened on the handle `h`. The frame is returned as a luminance plane, `y`, and two half resolution planes `u` and `v`. If `skip` is provided then `skip` (default 0) frames will be skipped before the next frame is returned. The function can optionally return the header string, `hdr`, which contains information specific to the tool used to create the file.

Returns `y = []` on end of file.

## Limitations

Assumes the file is in yuv420 format.

## See Also

yuvopen, yuv2rgb

# yuv2rgb

**Purpose** Convert an image from YUV to RGB format

**Synopsis**

```
rgb = yuv2rgb(y, u, v)
[r,g,b] = yuvread(y, u, v)
rgb = yuv2rgb2(y, u, v)
[r,g,b] = yuvread2(y, u, v)
```

**Description** Converts the YUV image to an RGB image with all planes of the same size. The first two calls halve the resolution of luminance, `y`, to match `u` and `v`. The second two double `u` and `v` using simple pixel replication.

**See Also** yuvopen, yuvread

# zcross

## Purpose

Find zero crossings

## Synopsis

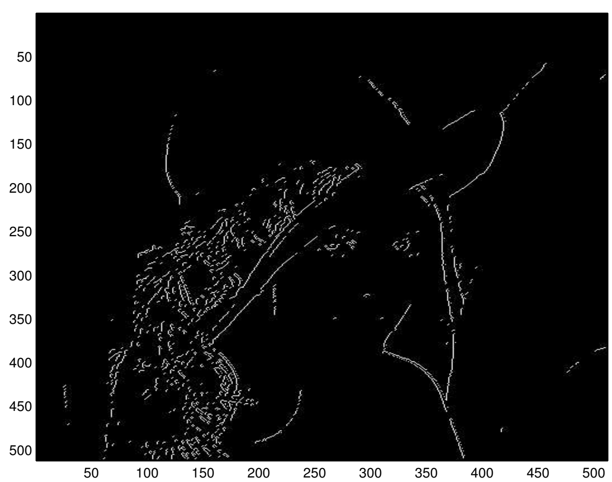
```
zc = zcross(image)
```

## Description

`zcross` returns a binary image where set pixels correspond to transitions from negative to positive in the input image. Often used in conjunction with a Laplacian of Gaussian operator which is the basis of the Marr-Poggio edge finder.

## Examples

```
>> lena = loadpgm('lena');
>> LoGlena = conv2(lena, klog(1));
>> idisp(zcross(LoGlena))
```




## Limitations

The method is quite crude, at each pixel the result is the logical or of a transition to the left or below.

## See Also

klog

# zncc

## Purpose

Zero-mean normalized cross-correlation

## Synopsis

```
m = zncc(A, B)
```

## Description

Compute the zero-mean normalized cross-correlation similarity measure between the two equally sized image patches `A` and `B`.

$$ZNCC(A,B) = \frac{\sum (A_{ij} - \overline{A})(B_{ij} - \overline{B})}{\sqrt{\sum (A_{ij} - \overline{A}) \sum (B_{ij} - \overline{B})}}$$

where $\overline{A}$ and $\overline{B}$ are the mean over the region being matched. This measure is invariant to illumination offset. While computationally complex it yields a well bounded result, simplifying the decision process. Result is in the range -1 to 1, with 1 indicating identical pixel patterns.

## See Also

similarity

## References


[1] I. E. Sutherland, "Three-dimensional data input by tablet," *Proc. IEEE*, vol. 62, pp. 453–461, Apr. 1974.

[2] R. Tsai, "An efficient and accurate camera calibration technique for 3D machine vision," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pp. 364–374, 1986.

[3] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, pp. 381–395, June 1981.

[4] O. Faugeras, *Three-dimensional computer vision*. MIT Press, 1993.

[5] C. G. Harris and M. J. Stephens, "A Combined Corner and Edge Detector," in *Proceedings of the Fourth Alvey Vision Conference, Manchester*, pp. 147–151, 1988.

[6] S. Ganapathy, "Camera location determination problem," Technical Memorandum 11358-841102-20-TM, AT&T Bell Laboratories, Nov. 1984.

[7] S. Ganapathy, "Decomposition of transformation matrices for robot vision," in *Proc. IEEE Int. Conf. Robotics and Automation*, pp. 130–139, 1984.

[8] O. Faugeras and F. Lustman, "Motion and structure from motion in a piecewise planar environment," *Int. J. Pattern Recognition and Artificial Intelligence*, no. 3, pp. 485–508, 1988.

[9] J. Wilf and R. Cunningham, "Computing region moments from boundary representations," JPL 79-45, NASA JPL, Nov. 1979.

[10] P. I. Corke, *Visual Control of Robots: High-Performance visual servoing*. Mechatronics, Research Studies Press (John Wiley), 1996.

[11] S. Hutchinson, G. Hager, and P. Corke, "A tutorial on visual servo control," *IEEE Transactions on Robotics and Automation*, vol. 12, pp. 651–670, Oct. 1996.