Part 1.

1. The process of Marr-Hildreth Detector (LoG) is:

(1) Smooth the image by applying a Gaussian filter on the image, parametrized by the window size n. Since 2nd order derivatives are quite sensitive to noise.

(2) Use 2nd order derivative to detect the edges. So calculate the Laplacian of the output Gaussian from step 1.

The output would be:

(3) Since the Laplacian output (2nd order derivative) generates negative signs of the neighbors for every edge, so we need to find the zero crossing in the output from step 2 to locate every edge.

The size of the Gaussian window n is typically chosen as an odd number that is >= 6 \* σ.

The process of a Canny Edge Detector is:

(1) Smooth the image by applying a Gaussian filter, in order to reduce the impact of noise.

(2) Compute the partial derivatives as gradient, and use approximations to compute the gradient magnitude.

, edge strength

(3) Apply non-maxima suppression on the gradient magnitude to thin the edges to single edge point. The basic idea here is to explore in the direction of gradients, and preserve the largest edge strength value as edge, and suppress all other values.

(4) Detect and refine edges by conducting ***double thresholding***, steps are:

a) plot the edge values of each pixel number along gradient maxima generated from step 3.

b) set 2 thresholds: high threshold t1 and low threshold t2.

c) edge values higher than t1 are determined as edge.

d) edges connected to strong edge pixels are determined as edge.

e) all other pixels are not considered as edge.

A close up of a map

Description automatically generated

2. The polar representation of a line:

in a plane of an image, the range of θ and ρ: , , where the image is in the size of .

The process of Hough Transform to locate lines:

1) Cut and partition the image plane into accumulator cells A[ρ, θ], so that the cell at (i, j) represents the square associated with parameter values (θi, ρi).

2) Initialize all cells with value 0.

3) For each point in (xk, yk) in the image:

3.1) calculate all ρq based on ρ = xcosθ + ysinθj, where θj are all possible values partitioned in step 1.

3.2) round these ρq to its closest cell value

3.3) increment the cell value of A(q, j)

4) Now, every cell in matrix A contains the number of points in the (x, y) space that lie on the same line.

5) find the line candidates where A(i, j) is beyond a certain threshold.

3.

1) Load an image:

I = imread('ku.jpeg');

imshow(I);

A large brick building

Description automatically generated

2) Detect edges using Canny detector with thresholds of [0.2, 0.5]:

grey = rgb2gray(I);

BW = edge(grey, 'canny', [0.2 0.5]);

imshow(BW);

A close up of a logo

Description automatically generated

3) Try with different threshold combinations:

i = 0;

for a=[0.1, 0.3, 0.4, 0.6]

bw = edge(grey, 'canny', [a a+0.3]);

i = i + 1;

subplot(2,2,i),imshow(bw); title(strcat('thresholds=[', num2str(a), ', ', num2str(a+0.3), ']'));

end

A screenshot of a cell phone

Description automatically generated

With the value of thresholds going up, it tends to detect edges that are more essential, and ignoring more subtle edges.

4) construct Hough peaks and compute Hough lines based on it.

P = houghpeaks(H, 5, 'threshold', ceil(0.3\*max(H(:))));

lines = houghlines(BW, theta, rho, P, 'FillGap', 5, 'MinLength', 7);

A close up of a logo

Description automatically generated

Traverse through all lines detected by houghlines, plot 2 primary lines: the first one, and the longest one.

figure, imshow(grey), hold on

max\_len = 0;

for k = 1:length(lines)

xy = [lines(k).point1; lines(k).point2];

if k == 1

plot(xy(:,1),xy(:,2),'LineWidth',2,'Color','green');

% Plot beginnings and ends of lines

plot(xy(1,1),xy(1,2),'x','LineWidth',2,'Color','yellow');

plot(xy(2,1),xy(2,2),'x','LineWidth',2,'Color','red');

end

% Determine the endpoints of the longest line segment

len = norm(lines(k).point1 - lines(k).point2);

if ( len > max\_len)

max\_len = len;

xy\_long = xy;

end

end

% highlight the longest line segment

plot(xy\_long(1,1),xy\_long(1,2),'x','LineWidth',2,'Color','yellow');

plot(xy\_long(2,1),xy\_long(2,2),'x','LineWidth',2,'Color','red');

plot(xy\_long(:,1),xy\_long(:,2),'LineWidth',2,'Color','red');

A large building

Description automatically generated

4. The process of finding the least-square solution of a linear system Ac = y:

1) define the residual error between the predicted value and real value:

2) Formulate the cost function:

3) Take the partial derivative on c and make it equal to 0 to find minimum:

4) compute c according to the formula, and the final solution would be y = c1x + c2

For the below data points:

![A picture containing clock, object

Description automatically generated]()

y = [1 1.5 1 2 2.5]';

A = [1 1; 2 1; 3 1; 4 1; 5 1 ];

c = inv(A'\*A)\*A'\*y

c =

0.3500

0.5500

y = 0.35x + 0.55

5.

1) To compute the first order derivative for gradient, we calculate the partial derivative of the cost function on our target c: , and find the minimum point by checking when it equals to zero.

For edge strength, we calculate the gradient of pixel values in both x axis and y axis, by computing the partial derivative on x and on y: . The strength of the edge can be represented by an approximation in magnitude: .

For edge direction, the direction angle could be represented using arctan:

2) The masks for Prewitt, Sobel and Laplacian detectors:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | |  | |  | |
| normal | diagonal | vertical | diagonal | vertical | diagonal |
| Laplacian | | Prewitt | | Sobel | |

Laplacian: use second order derivative for edge detection, with filter.

lab=[0 1 0;1 -4 1; 0 1 0];

rez=uint8(filter2(lab grey, 'same'));

imshow(rez);

title('laplacian');

A close up of a building

Description automatically generated

Prewitt: use first order derivative for edge detection, with filter.

E2 = edge(grey, 'prewitt');

figure; imshow(E2); title('prewitt');

A close up of text on a black background

Description automatically generated

Sobel: use first order derivative for edge detection, with smoothing added.

E3 = edge(grey, 'sobel');

figure; imshow(E3); title('sobel');

A close up of text on a black background

Description automatically generated

Part 2.

1. A brief technical description of the blind deconvolution algorithm

Blind deconvolution is a technique to recover a sharp version of an image from a blurred input, denoted by , where k is an unknown blur kernel.

The paper explains about the failure of the MAPx,k (maximum-a-posteriori) estimation approach, which seeks a (x, k) pair that maximizes . A possible reason of failure could be the choice of a bad prior in natural image scenarios, which fails to recover the true blur kernel.

To overcome the limitations of MAPx,k approach, the paper discussed about reconsidering the choice of estimator instead of the choice of prior, which is the MAPK estimation approach. Since the size of the kernel k is fixed and small compared to x, thus to exploit asymmetry between x and k would be wise, and this is why it selects the k that maximizes denoted by , and x is marginalized, whose full possible values are evaluated using integral.

The paper elaborates two MAPK estimations: Gaussian prior and sparse prior. The Gaussian prior is in the form of where and denote the Fourier transform of derivatives gx, gy. In this way, the solution for MAPK would result from the prior variance and looks like a sample from the prior, instead of a solution as MAPx,k did.

For a sparse prior, several approximation strategies could be applied for computation simplicity to replace a closed form solution, such as the histogram approximation that sums over all gradient histogram bins instead of all image pixels, and variational Bayes approximation that assumes follows a Bayesian distribution and tries to solve x using a picked blur kernel from the distribution.

2. Description on each step.

(1) load image and transform to grayscale.

I = imread('ku.jpeg');

grey = rgb2gray(I);

imshow(grey);

A large white building

Description automatically generated

(2) blur the image using a Gaussian filter.

PSF = fspecial('gaussian', 7, 10);

Blurred = imfilter(grey,PSF, 'symmetric', 'conv');

imshow(Blurred)

title('Blurred Image')

A sign on the side of a building

Description automatically generated

(3) Try to restore the blurred image using point-spread function. First try UNDERPSF in a uniform array with 4 pixels shorter as an initial guess of the PSF.

UNDERPSF = ones(size(PSF) - 4);

[J1,P1] = deconvblind(Blurred, UNDERPSF);

imshow(J1)

|  |  |
| --- | --- |
| A sign on the side of a building  Description automatically generated | A sign on the side of a building  Description automatically generated |
| UNDERPSF | blurred |

(4) Try OVERPSF, where the uniform array is 4 pixels longer than the true PSF.

OVERPSF = padarray(UNDERPSF, [4 4], 'replicate', 'both');

[J2,P2] = deconvblind(Blurred, OVERPSF);

imshow(J2)

|  |  |
| --- | --- |
| A picture containing cage  Description automatically generated | A sign on the side of a building  Description automatically generated |
| OVERPSF | blurred |

(5) Try the INITPSF whose size is the same as true PSF.

INITPSF = padarray(UNDERPSF,[2 2], 'replicate', 'both');

[J3,P3] = deconvblind(Blurred, INITPSF);

imshow(J3)

|  |  |
| --- | --- |
| A large white building  Description automatically generated | A sign on the side of a building  Description automatically generated |
| PSF | blurred |

It could be observed that PSF has the best solution here compared with UNDERPSF and OVERPSF.

(6) Compared the PSF produced by all above methods.

figure;

subplot(2, 2, 1)

imshow(PSF, [], 'InitialMagnification', 'fit')

title('True PSF')

subplot(222)

imshow(P1, [], 'InitialMagnification', 'fit')

title('Reconstructed Undersized PSF')

subplot(2,2,3)

imshow(P2, [], 'InitialMagnification', 'fit')

title('Reconstructed Oversized PSF')

subplot(2,2,4)

imshow(P3, [], 'InitialMagnification', 'fit')

title('Reconstructed true PSF')

A close up of a white wall

Description automatically generated

From the image above, we could see that for UNDERPSF, there are fewer value bins generated and thus displaying a far more coarse granularity. In contrary, the OVERPSF guesses a finer granularity which brings in some weird sharpness in its solution. The INITPSF is just a proper approximation of the true PSF, so it deblurs the best.

(7)