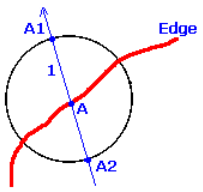
1. Canny edge detection

1.1 Goals

The goal of this exercise is to understand and implement an edge detector algorithm similar to the Canny edge detector. This popular edge detection algorithm is based on the following steps.

1. **Smoothing**: the raw unprocessed image is smoothed to reduce the effect of the noise.
2. **Gradient Computation**: the next step is to detect edges, which can be performed by convolving your filtered image with an edge detection operator (e.g. Sobel). Then, the gradient magnitude G and phase ϕ have to be computed.
3. **Non-Maxima Suppression**: for a given point $A = (x_A, y_A)$ on the image, we first compute the position of the points $A1 = (x_{A1}, y_{A1}) = A + u$ and $A2 = (x_{A2}, y_{A2}) = A - u$, where u is a unit vector in the direction of the gradient. The magnitude value $G(A)$ is kept only if it is greater or equal to $G(A1)$ and $G(A2)$, otherwise it is set to 0.



4. **Hysteresis Threshold**: Hysteresis thresholding requires two thresholds – high (H) and low (L). Pixels that have gradient values greater than H are considered as strong edges and included in the solution. Pixels that have gradient values smaller than L are discarded and the remaining set of pixels (those with gradient magnitudes between L and H) are considered as edges if they are connected to a strong edge.

Exercise 1.1

- Implement the steps of edge detector described above:
  1. **Smoothing**: load the provided image `in1.jpg`, convert it to grayscale, `double` format and smooth it with a Gaussian filter. For this and the next step, you can re-use the code of the previous exercise session.
  2. **Gradient Computation**: convolve the smoothed image with a derivative filter (e.g. Sobel), and compute the gradient magnitude G and phase ϕ as seen in the previous sessions.
  3. **Non-Maxima Suppression**: for all pixels $A = (x_A, y_A)$ on the image, compute the position of the points $A1$ and $A2$. Notice that their coordinates may not be integer: for computing $G(A1)$ and $G(A1)$, you can use the provided function `BilinearPixels( Im, u, v)`, that returns the intensity of an image at non integer coordinates by mean of bi-linear interpolation. After that, compute the non-maxima suppression as described above.
  4. **Hysteresis Threshold**: For this step you can use the provided function `HysteresisThresholding`. This is implemented in three steps:

(a) each image pixel is classified into one of the three classes (i.e., FOREGROUND = 1, CANDIDATE = 0.5 and BACKGROUND = 0) based on its gradient magnitude (after non-maxima suppression) and two thresholds.
(b) Pixels that are labeled as FOREGROUND are then grown in order to include all connected pixels with CANDIDATE labels. Such candidate pixels are then relabeled as FOREGROUND = 255. For each FOREGROUND pixel, you can check its 8 neighbours.
(c) Finally, all pixels with FOREGROUND labels are considered as edges and the rest is labeled as BACKGROUND.

Apply your edge detector to the images below. Answer the following questions:

- Why is not enough to detect edges simply taking pixels whose gradient magnitude is above a certain threshold? what are the main advantages of the proposed approach? Do you see any limitations?
- Discuss the influence of the 3 parameters of the algorithm: the smoothing variance σ, the high threshold H and the low threshold L.
- How would you get 2 edge images for the fractal below, one with the tiny details and one with only the main shape?
- Cite at least 2 practical applications where you think an edge detector like the one you implemented may be used. (*Hint: see next exercise.*)

Figure 1.1: Some images for testing Canny edge detection.

2. Hough transform

2.1 Goals

The goal of this exercise is to

- Compute edge images from the real-world images
- Understand and apply a Hough Transform for detecting objects of a specific shape

2.2 Linear Hough Transform

Assume we are given a pair of an original gray-scale image $I[x, y]$ and a corresponding edge image `edge_I`$[x, y]$, of size $W \times H$, such that $1 \leq x \leq W$, and $1 \leq y \leq H$. Every pixel of the edge image has logical value and is equal to 1 if the corresponding pixel of the original image belongs to an edge, and 0 otherwise.

We are interested in detecting the presence of a specific shape in the original image. This is usually a first step in various Computer Vision tasks (detection of road signs, eye detection, etc.). A well known approach to detecting shapes such as lines, circles and ellipses is the Hough transform.

In this exercise you are asked to implement a version of Hough transform for detecting lines. A line can be represented in the following way:

$$xcos(\theta) + ysin(\theta) = r, \qquad 0 \leq \theta < \pi \tag{2.1}$$

Here (x, y) are pixel coordinates in the edge image `edge_I`, and (θ, r) are the parameters of the model.

The Hough transform consists of checking every non-zero element of the edge image, computing its vote and storing it in an accumulator array. This array in our case is a 2-dimensional matrix, whose rows and columns correspond to different values of parameters r and θ respectively.

Exercise 2.1 Implement a function `accum = LinearHoughAccum(edge_I)` that takes the edge image as an input and returns the accumulator array as an output. For now you can assume that the image is square, that is, $W = H$.

- To compute the edge image from the gray-scale image you can use the results of the previous exercise or the MATLAB function `edge`.
- Implement a loop that for every non-zero element of the edge image and every value of parameter θ calculates the value of parameter r and increases the corresponding position of the accumulator array by 1.
  - Create an array of $\theta_k : \theta_0 = 0, \;\; \theta_k = \theta_{k-1} + 0.01, \;\; \theta_k, \forall k < \pi$
  - Allocate memory for the accumulator array with sizes: $(\lceil \sqrt{2}W \rceil, N)$, where W is the horizontal dimension of the image I, N is the number of elements in the θ_i and operator $\lceil \cdot \rceil : \mathbb{R} \to \mathbb{Z} : \lceil x \rceil - 1 < x \leq \lceil x \rceil, \forall x \in \mathbb{R}$
  - Fill the accumulator array with zeros
  - Implement a loop that does the following:

$\forall i, j, k$
$r = j * cos(\theta_k) + i * sin(\theta_k)$
$accum(\lceil r \rceil, k) = accum(\lceil r \rceil, k) + 1$

- Visualize the accumulator array using `imagesc` function
- How the dimensions of the accumulator array change if I is not square ($W \neq H$)?
- How can this approach be extended to detecting circles? How many parameters will there be for such model?

■

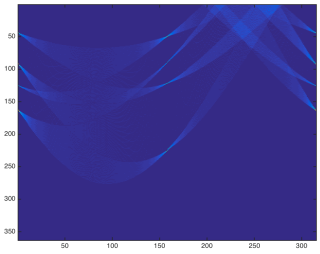
To evaluate the approach we want to visualize the results. We need to plot the line that got the higher number of votes in the accumulator array.

Exercise 2.2 Implement a function `show_lines(I,accum)` that takes the original image and the accumulator array as an input, and displays the line that has the maximum number of votes in the accumulator array. This consists of the following operations:

- Find the maximum value of the accumulator array
  Hint: you can use MATLAB function `max`
- Plot the line on top of the original image using Eq.2.1.

■

Original image



Sample output of the accumulator array from Exercise 2.1

Sample output of the Exercise 2.2