Assignment 2
Neighborhood Processing & Filters

Computer Vision 1
University of Amsterdam

Due 11:59pm, September 21, 2020 (Amsterdam time)

General guidelines

Your source code and report must be handed in together in a zip file (`ID1_ID2_ID3.zip`) before the deadline by sending it to Canvas Lab 2 Assignment. For full credit, make sure your report follows these guidelines:

- Include an introduction and a conclusion to your report.
- The maximum number of pages is 10 (single-column, including tables and figures). Please express your thoughts concisely. The number of words does not necessarily correlate with how well you understand the concepts.
- Answer all given questions (in green boxes). Briefly describe what you implemented. Blue boxes are there to give you hints to answer questions.
- Try to understand the problem as much as you can. When answering a question, give evidences (qualitative and/or quantitative results, references to papers, etc.) to support your arguments.
- Analyze your results and discuss them, e.g. why algorithm A works better than algorithm B in a certain problem.
- Tables and figures must be accompanied by a brief description. Do not forget to add a number, a title, and if applicable name and unit of variables in a table, name and unit of axes and legends in a figure.

Late submissions are not allowed. Assignments that are submitted after the strict deadline will not be graded. In case of submission conflicts, TAs' system clock is taken as reference. We strongly recommend submitting well in advance, to avoid last minute system failure issues.
Plagiarism note: Keep in mind that plagiarism (submitted materials which are not your work) is a serious crime and any misconduct shall be punished with the university regulations.

Contents

1 Introduction

In this assignment, you will get familiar with fundamentals of neighborhood processing for image processing. These techniques allow for low-level image understanding via extraction of structural patterns such as edges and blobs. Similarly, they find an extensive use in image denoising and higher level image reasoning such as shape recognition. Moreover, neighborhood or block processing is one of the key components of *Convolutional Neural Networks*. Therefore, a good understanding of these procedures will be a stepping stone towards understanding more complex machinery used in computer vision and machine learning.

In subsequent sections of this assignment, we will first explain neighborhood processing and introduce low-level filters commonly used to analyze images. After that, we will see how these mathematical concepts relate to practice by working through fundamental tasks such as denoising and segmentation. By the end of this assignment, you will have an overall understanding of the following:

- Gaussian and Gabor filters
- Edge detection and image denoising
- Texture-based image segmentation

2 Neighborhood Processing

Neighborhood processing is simply about looking around a point $\mathbf{I}(x, y)$ (i.e. pixel) in the image, $\mathbf{I}$, and applying a function, $\mathbf{h}(k, l)$, which measures certain properties or relationships between the pixels in that localized window. The function, $\mathbf{h}(k, l)$, is generally referred to as the *neighborhood* operator or *local* operator. One of the most common forms of a neighborhood operator is a *linear filter*. Linear filters simply compute the weighted sum of neighboring pixel intensities and assign it to the pixel of interest (output $\mathbf{I}_{out}(i, j)$). The filters in which we are interested here are usually represented as a square matrix.

Hint

Filters, kernels, weight matrices or masks are interchangeably used in the literature. A kernel is a matrix with which we describe a neighborhood operation. This operation can, for example, be edge detection or smoothing.

Linear filters are shifted over the entire image plane via operators such as correlation ($\otimes$) and convolution ($*$). Both of these operators are *linear shift-invariant* (LSI) implying that the filters behave the same way over the entire image. Discrete forms of these operators are given in the following:

Correlation and Convolution (Discrete form)

$$\mathbf{I}_{out} = \mathbf{I} \otimes \mathbf{h}, \quad \mathbf{I}_{out}(i,j) = \sum_{k,l} \mathbf{I}(i+k, j+l)\mathbf{h}(k,l), \quad \text{Correlation} \quad (1)$$

$$\mathbf{I}_{out} = \mathbf{I} * \mathbf{h}, \quad \mathbf{I}_{out}(i,j) = \sum_{k,l} \mathbf{I}(i-k, j-l)\mathbf{h}(k,l), \quad \text{Convolution} \quad (2)$$

Question-1 (10*pts*)

1. What is the difference between correlation and convolution operators? How do they treat the signals **I** and **h**?

2. Correlation and convolution operators are equivalent when we make an assumption on the form of the mask **h**. Can you identify the case?

We illustrate the overall idea of neighborhood processing in Figure 1.

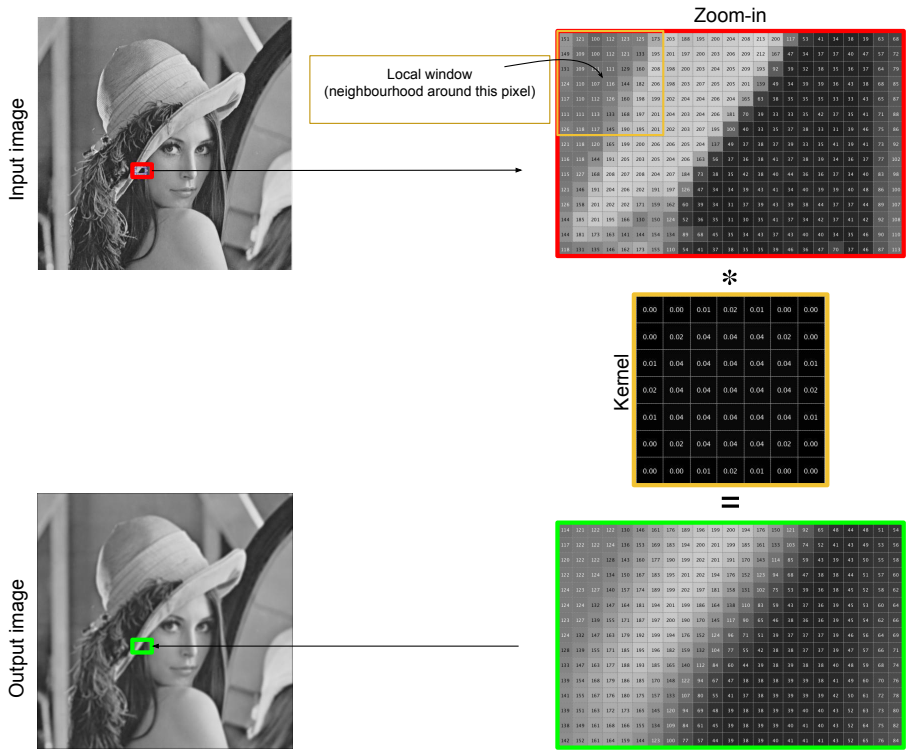

Figure 1: The kernel or the mask convolves over the input image. In the case of linear filters, this is simply multiplying each pixel intensity with the corresponding weight in the kernel (see the yellowish 7x7 window where the kernel is placed). In the figure, the kernel is 7x7 averaging mask. You can see its effect by comparing the red (before filtering) and the green (after filtering) frames.

3 Low-level filters

In this section, you will design common linear filters used in neighborhood processing. We will focus in particular on Gaussian and Gabor filters.

3.1 Gaussian Filters

3.1.1 1D Gaussian Filter

The 1D Gaussian filter is defined as follows:

$$G_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{x^2}{2\sigma^2}), \tag{3}$$

where σ is the variance of the Gaussian. However, such formulation creates an infinitely large convolution kernel. In practice, the kernel is truncated with a `kernel_size` parameter such that $-\lfloor\frac{kernel\_size}{2}\rfloor \leq x \leq \lfloor\frac{kernel\_size}{2}\rfloor$, where $\lfloor.\rfloor$ is the floor operator. As an example, if `kernel_size` equals 3, $x \in \{-1, 0, 1\}$.

Implement `gauss1D`

```
def gauss1D(sigma,kernel_size)
    ...
    return G
```

Note: You are not allowed to use a Python built-in function provided by *SciPy* or other libraries to compute the kernel.

Hints

- Do not forget to normalize your filter.
- `gauss1D(2, 5)` should give you:
  `[0.1525, 0.2218, 0.2514, 0.2218, 0.1525]`.

3.1.2 2D Gaussian Filter

One of the most important properties of 2D Gaussian kernels is separability. Therefore, convolving an image with a 2D Gaussian is equivalent to convolving the image twice with a 1D Gaussian filter, once along the x-axis and once along the y-axis **separately**. A 2D Gaussian kernel can then be defined as a product of two 1D Gaussian kernels:

$$G_\sigma(x, y) = G_\sigma(x) \times G_\sigma(y) \tag{4}$$

$$= \frac{1}{\sigma^2 2\pi} \exp(-\frac{x^2 + y^2}{2\sigma^2}) \tag{5}$$

Implement `gauss2D`

Your **gauss2D** implementation corresponds to Eq.4 (not Eq.5) and should make use of **gauss1D**.

```
def gauss2D(sigma,kernel_size)
    ...
    return G
```

Note: You are not allowed to use a Python built-in function provided by *SciPy* or other libraries to compute the kernel.

Question-2 (5*pts*)

What is the difference between convolving an image with (1) a 2D Gaussian kernel and (2) a 1D Gaussian kernel in the x- and y-direction? Will the result be the same? What is their computational complexity?

3.1.3 Gaussian Derivatives

So far the Gaussian kernels that we computed are mainly targeted to image enhancement algorithms (e.g. denoising an image). These kernels can also be used for detecting changes in the image intensity pixels. These low-level features can then further be used as building blocks for more complicated tasks like object detection or segmentation.

Concretely, the first order derivative of the 1D Gaussian kernel is given by:

$$\begin{aligned} \frac{d}{dx}G_\sigma(x) &= \frac{d}{dx}\left(\frac{1}{\sigma\sqrt{2\pi}}\exp(\frac{x^2}{2\sigma^2})\right) \\ &= -\frac{x}{\sigma^3\sqrt{2\pi}}\exp(-\frac{x^2}{2\sigma^2}) \\ &= -\frac{x}{\sigma^2}G_\sigma(x) \end{aligned} \tag{6}$$

Similarly, the first order derivative of the 2D Gaussian kernel can be obtained by computing $\frac{d}{dx}G_\sigma(x,y)$ and $\frac{d}{dy}G_\sigma(x,y)$.

Question-3 (5*pts*)

A second order derivative of the Gaussian kernel can also be computed. Why is it interesting to design a second order kernel?

3.2 Gabor filters

Gabor filters fall into the category of linear filters and are widely used for *texture analysis*. The reason why they are a good choice for texture analysis is that they localize well in the frequency spectrum (*optimally* bandlimited) and therefore work as flexible *band-pass* filters. See Figures 2 and 3.

3.2.1 1D Gabor Filters

For the sake of simplicity, we start by studying what a Gabor function is using 1D signals (e.g. speech). The idea will later be generalized to the 2D case, which is suited for our primary interest, images.

A Gabor function is a Gaussian function modulated with a complex sinusoidal carrier signal. Let us denote the Gaussian with $x(t)$ and complex sinusoidal with $m(t)$. Then, a Gabor function $g(t)$ can be formulated by

$$g(t) = x(t)m(t) \tag{7}$$

where $x(t) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{t^2}{2\sigma^2}}$ and $m(t) = e^{-j2\pi f_c t} = e^{-jw_c t}$. σ is the parameter determining the spread of the Gaussian and w_c is the central frequency of the carrier signal.

Hint: *Euler's formula*

A complex sinusoidal can be represented as follows using the Euler's formula:

$$e^{jwt} = \cos(wt) + j\sin(wt)$$

Using Euler's formula, we get the following:

$$g(t) = x(t)m(t) \tag{8}$$

$$g(t) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{t^2}{2\sigma^2}} e^{-jw_c t} \tag{9}$$

$$g(t) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{t^2}{2\sigma^2}} [\cos(w_c t) + j\sin(w_c t)] \tag{10}$$

We can further arrange the terms and arrive at the following form

$$g(t) = g_e(t) + jg_o(t) \tag{11}$$

where $g_e(t)$ and $g_o(t)$ are the even and odd parts arranged orthogonally on the complex plane $\mathbf{Z}^2$. In practice, one can use either the even or the odd part for filtering purposes (or one can use the complex form).

3.2.2 2D Gabor Filters

The Gabor filters can also be defined in 2D as well. The main difference lies in the dimensionality of the signals (i.e. carrier and gaussian). A sine wave in 2D is described by two orthogonal spatial frequencies u_0 and v_0 such that it is given as $s(x, y) = sin(2\pi(u_0 x + v_0 y))$ where a 2D gaussian is simply $Ce^{-(\frac{(x-x_0)^2}{2\sigma_x^2}) + \frac{(y-y_0)^2}{2\sigma_y^2})}$ with C being a normalizing constant. 2D Gabor function then takes the following forms in the real and complex parts:

$$g_{real}(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \cos\left(2\pi\frac{x'}{\lambda} + \psi\right) \tag{12}$$

$$g_{im}(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \sin\left(2\pi\frac{x'}{\lambda} + \psi\right), \tag{13}$$

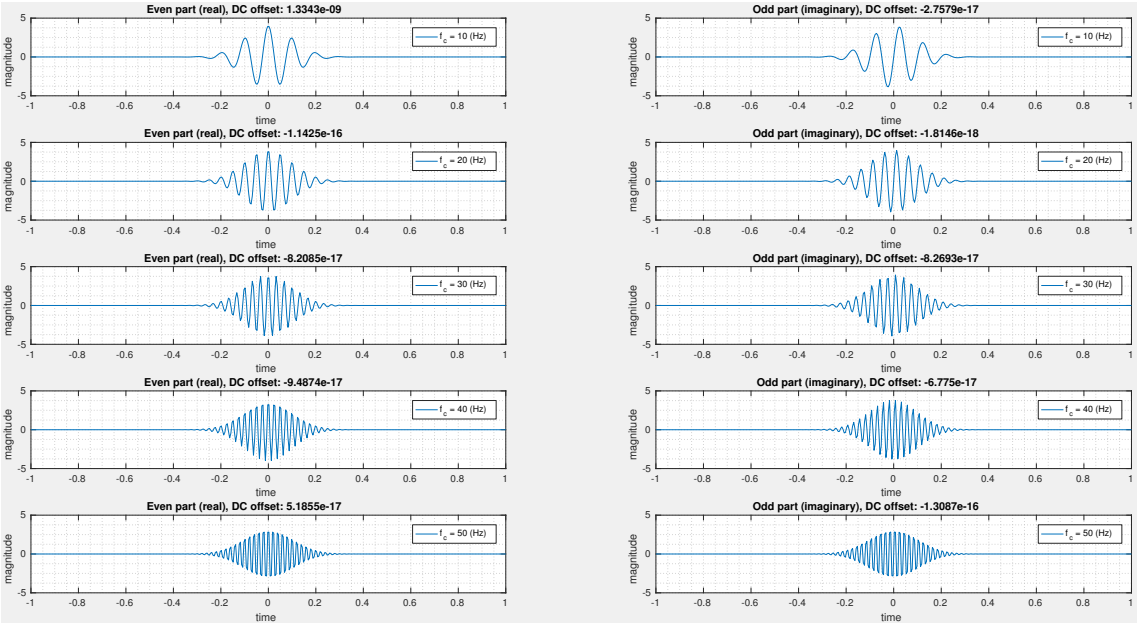

Figure 2: Even (cosine-modulated) and odd parts (sine-modulated) of Gabor filters with fixed-σ Gaussian. We illustrate the time-domain filters for the modulating sinusoidals of central frequencies, 10, 20, 30, 40 and 50 Hz, respectively.

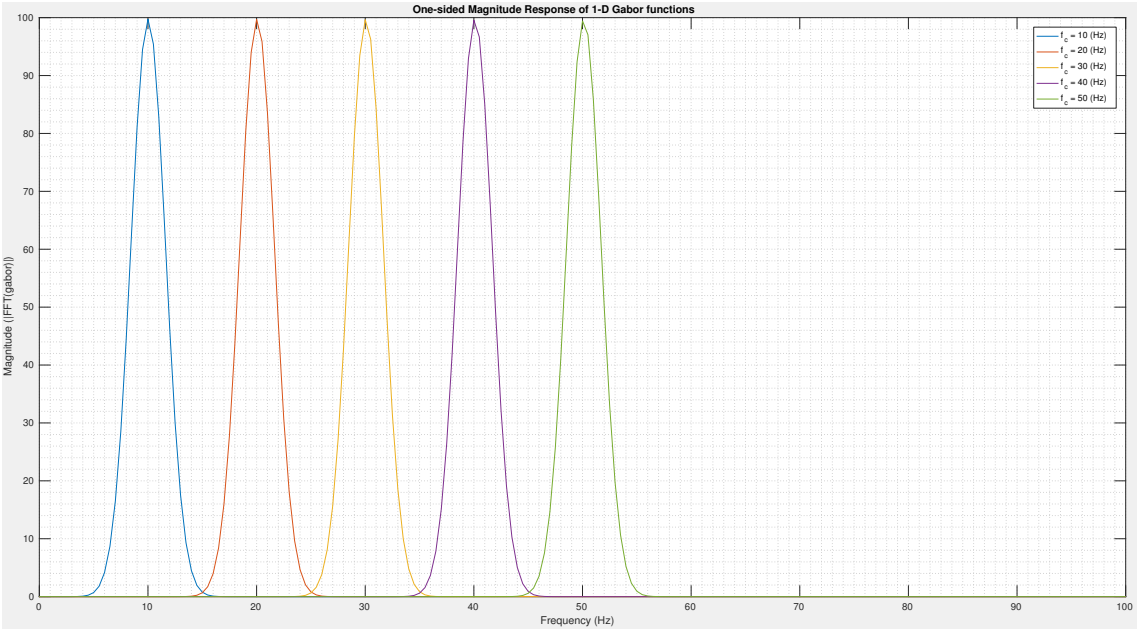

Figure 3: Gabor filters with varying center frequencies are sensitive to different frequency bands. Notice that the neighboring (in the frequency spectrum) filters minimally interfere with each other.

where

$$x' = x\cos\theta + y\sin\theta \tag{14}$$

$$y' = -x\sin\theta + y\cos\theta \tag{15}$$

Question-4 (5*pts*)

Conduct a self-study on the Gabor filters. Explain shortly what the parameters $\lambda, \theta, \psi, \sigma, \gamma$ control.

Implement `createGabor`

Implement the function **createGabor** using (12) and (13).

Question-5 (5*pts*)

Visualize how the parameters θ, σ and γ affect the filter in spatial domain.

4 Applications in image processing

4.1 Noise in digital images

The quality of digital images can be affected in different ways. For example, the acquisition process can be very noisy and with a low-resolution (e.g. some medical imaging modalities only generate a 128x128 image). Noise can also come from the user who set wrong parameters on the digital camera. Consequently, different computer vision algorithms are required to enhance noisy or corrupted images. With the growing amount of photos taken every day, image enhancement has then become a very active area of research.

In this section, we only focus on simple algorithms to correct noise coming typically from the sensor of your camera. Many other types of noise or corruption can happen but are out of the scope of this assignment.

4.1.1 Salt-and-pepper noise

Noise can also occur with over-exposition causing a "hot" pixel or with a defective sensor causing a "dead" pixel. This is called salt-and-pepper noise. Pixels in the image are randomly replaced by either a white or black pixel.

4.1.2 Additive Gaussian noise

Noise also occurs frequently when the camera heats up. This is called thermal noise and this can be modeled as an additive Gaussian noise. Every pixel in the image has a noise component that corresponds to a random value chosen independently from the same Gaussian probability distribution. The Gaussian distribution has a mean of 0 and its standard deviation corresponds to a parameter.

$$\mathbf{I}'(x) = \mathbf{I}(x) + \epsilon, \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{16}$$

where $\mathbf{I}'$ is the noisy image and $\mathbf{I}$ is the original image without any noise ϵ.

4.2 Image denoising

4.2.1 Quantitative evaluation

The peak signal-to-noise ratio (PSNR) is a commonly used metric to quantitatively evaluate the performance of image enhancement algorithms. It is derived from the mean squared error (MSE):

$$MSE = \frac{1}{m \cdot n} \sum_{x,y} \left[\mathbf{I}(x,y) - \hat{\mathbf{I}}(x,y) \right]^2 \tag{17}$$

where $\mathbf{I}$ is the original image of size $m \times n$ and $\hat{\mathbf{I}}$ its approximation (i.e. in our case an enhanced corrupted image). The PSNR corresponds to:

$$\begin{aligned} PSNR &= 10 \cdot \log_{10} \left(\frac{\mathbf{I}_{max}^2}{MSE} \right) \\ &= 20 \cdot \log_{10} \left(\frac{\mathbf{I}_{max}}{\sqrt{MSE}} \right) \\ &= 20 \cdot \log_{10} \left(\frac{\mathbf{I}_{max}}{RMSE} \right) \end{aligned} \tag{18}$$

where $\mathbf{I}_{max}$ is the maximum pixel value of $\mathbf{I}$ and RMSE is the root of the MSE.

Implement `myPSNR`

```
def myPSNR(orig_image, approx_image):
    ...
    return PSNR
```

Note: You are not allowed to use the Python built-in functions provided in *PIL* and *Skimage*.

Question-6 (10*pts*)

1. Explain briefly in your own words what the PSNR is (without any equations). When comparing different methods with the PSNR metric, is a higher value the better or the opposite?

2. Using your implemented function **myPSNR**, compute the PSNR between image1_saltpepper.jpg and image1.jpg. Report the PSNR.
   Hint: You should have a PSNR around 16 dB.

3. Using your implemented function **myPSNR**, compute the PSNR between image1_gaussian.jpg and image1.jpg. Report the PSNR.

4.2.2 Neighborhood processing for image denoising

We will now design filters to remove these two types of noise. The function will denoise the image by either applying:

1. *box filtering*: You can use `cv2.blur` function.
2. *median filtering*: You can use `cv2.medianBlur` function.
3. *Gaussian filtering*: You must use your `cv2.GaussianBlur` function.

Implement `denoise`

```
def denoise(image, kernel_type, **kwargs):
    ...
    return imOut
```

Hints

1. `kernel_type` is just a string to specify the kernel type.
2. `**kwargs` allows to have an undefined key-value pairs in a Python function. For example, you can have `sigma` and `kernel_size` as argument when using a Gaussian kernel but only `kernel_size` when using a box kernel. For more information about how **kwargs works, take a look at usage of kwargs.

Question-7 (20*pts*)

1. Using your implemented function **denoise**, try denoising image1_saltpepper.jpg and image1_gaussian.jpg by applying the following filters:
   (a) Box filtering of size: 3x3, 5x5, and 7x7.
   (b) Median filtering with size: 3x3, 5x5 and 7x7.

   Show the denoised images in your report. Use tables to present your quantitative results.
2. Using your implemented function **myPSNR**, compute the PSNR for every denoised image (12 in total). What is the effect of the filter size on the PSNR? Report the results in a table and discuss.
3. Which is better for the salt-and-pepper noise, box or median filters? Why? What about the Gaussian noise?
4. Try denoising image1_gaussian.jpg using a Gaussian filtering. Choose an appropriate window size and standard deviation and justify your choice. Show the denoised images in your report.
5. What is the effect of the standard deviation on the PSNR? Report the results in a table and discuss.

6. What is the difference among median filtering, box filtering and Gaussian filtering? Briefly explain how they are different at a conceptual level. If two filtering methods give a PSNR in the same ballpark, can you see a qualitative difference?

4.3 Edge detection

Edges appear when there is a sharp change in brightness. In an image this usually corresponds to the boundaries of an object. Edge detection is a fundamental task used in many computer vision applications. One of them is road detection in autonomous driving, which is used for determining the vehicle trajectory.

Many different techniques exist for computing the edges. In this section, we will focus on filters that extract the gradient of the image. We will try to detect the road in an still image.

4.3.1 First-order derivative filters

Sobel kernels approximate the first derivative of a Gaussian filter. Below are the Sobel kernels used in the x and y directions.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{I} \tag{19}$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{I} \tag{20}$$

The gradient magnitude is defined as the square root of the sum of the squares of the horizontal (G_x) and the vertical (G_y) components of the gradient of an image, such that:

$$G = \sqrt{{G_x}^2 + {G_y}^2} \tag{21}$$

The gradient direction is calculated as follows:

$$\theta = \tan^{-1} \frac{G_y}{G_x} \tag{22}$$

Implement `compute_gradient`

```
def compute_gradient(image):
    ...
    end Gx,Gy,im_mag,im_dir
```

Note: You are not allowed to use the Python built-in functions for computing gradient. But for doing 2D convolution, you can benefit from *scipy.signal.convolve2d* function.

Question-8 (10*pts*)

Using your implemented function **compute_gradient** on image2.jpg, display the following figures:

1. The gradient of the image in the x-direction.
2. The gradient of the image in the y-direction.
3. The gradient magnitude of each pixel.
4. The gradient direction of each pixel.

Discuss what kind of information every image conveys.

4.3.2 Second-order derivative filters

Compared to the Sobel filter, a Laplacian of Gaussian (LoG) relies on the second derivative of a Gaussian filter. Hence, it will focus on large gradients in the image. A LoG can be computed by the following three methods:

- method 1: Smoothing the image with a Gaussian kernel (kernel size of 5 and standard deviation of 0.5), then taking the Laplacian of the smoothed image (i.e. second derivative).
- method 2: Convolving the image directly with a LoG kernel (kernel size of 5 and standard deviation of 0.5).
- method 3: Taking the Difference of two Gaussians (DoG) computed at different scales σ_1 and σ_2.

Implement `compute_LoG`

The function should be able to apply any of the above mentioned methods depending on the value passed to the parameter *LOG_type*.

```
def compute_LoG(image, LOG_type):
    ...
    return imOut
```

Question-9 (10*pts*)

1. Test your function using image2.jpg and visualize your results using the three methods.
2. Discuss the difference between applying the three methods.
3. In the first method, why is it important to convolve an image with a Gaussian before convolving with a Laplacian?
4. In the third method, what is the best ratio between σ_1 and σ_2 to achieve the best approximation of the LoG? What is the purpose of having 2 standard deviations?
5. What else is needed to improve the performance and isolate the road, i.e. what else should be done? You don't have to provide any specific parameter or specific algorithm. Try to propose a direction which would be interesting to explore and how you would approach it.

4.4 Foreground-background separation

Foreground-background separation is an important task in the field of computer vision (see Figure 4). In this exercise, you will implement a simple unsupervised algorithm that leverages the variations in texture to segment the foreground object from the background. We will assume the foreground object has a distinct combination of textures compared to background. As mentioned earlier, Gabor filters are well-suited for texture analysis thanks to their frequency domain characteristics. Therefore, we will use a collection of Gabor filters with varying scale and orientations which we call a *filter bank*. The outline of the algorithm is as follows:

Figure 4: **(Left)** Input image, **(Middle)** Foreground mask, **(Right)** Masked object. Foreground-Background separation aims at masking out the salient object pixels from the background pixels.

We provide you with additional instructions in the **gabor_segmentation.py** file.

Algorithm 1 Foreground-Background Segmentation Algorithm

Input: x - input image
Output: y - pixelwise labels

1. Convert to grayscale if necessary.
  if x is RGB **then**
    $x \leftarrow \text{rgb2gray}(x)$
  end if
2. Create Gabor filterbank, $\mathcal{F}_{gabor}$, with varying σ, λ and θ.
3. Filter x with the filterbank. Store each output in $fmaps$.
  $fmaps \leftarrow \mathcal{F}_{gabor}(x)$
4. Compute the magnitude of the complex $fmaps$. Store the results in $fmags$.

  $fmags \leftarrow |fmaps|$
5. Smooth $fmags$.
  $fmags \leftarrow \text{smooth}(fmags)$
6. Convert $fmags$ into data matrix, f.
  $f \leftarrow \text{reshape}(fmags)$
7. Cluster f using kmeans into two sets.
  $y \leftarrow \text{kmeans}(f, 2)$

Implement `gabor_segmentation`

Please get yourself familiar with provided skeleton code **gabor_segmentation.py**. Keep in mind that you will need your implementation of the **createGabor** function.

Implement all code sections where you see a comment in the form:

```
# \\TODO: xxx
```

When you succesfully implement it all, it should run without problems and produce a reasonable segmentation with the default parameters on *kobi.png*.

Question-10 (20*pts*)

1. Run the algorithm on all test images with the provided parameter settings. What do you observe? Explain shortly in the report.

2. Experiment with different λ, σ and θ settings until you get reasonable outputs. Report what parameter settings work better for each input image and try to explain why.

   Hint: Don't change multiple variables at once. You might not need to change some at all.

3. After you achieve good separation on all test images, run the script again with corresponding parameters but this time with

   ```
   smoothingFlag = False
   ```

   Describe what you observe at the output when smoothing is not applied on the magnitude images. Explain why it happens and try to reason about the motivation behind this step.