István Sárándi <sarandi@vision.rwth-aachen.de>
Dan Jia <jia@vision.rwth-aachen.de>

Exercise 1: Filtering, Derivatives, Edges and Hough Transform

due **before** 2019-05-06

Important information regarding the exercises:

- The goal of exercise is to provide you with hands-on experience on the algorithms covered in class.
- Each exercise question is a self-contained Jupyter notebook (.ipynb), with some missing implementations to be filled by you (you can modify the other parts, too). (What is a Jupyter notebook? See below).
- Solving the exercise is not mandatory and does not count towards your grade.
- When the deadline has passed, we will release an example implementation.
- We encourage you to work on the exercises in teams of up to 4 students.

Software Setup

Python will be the language we use for the exercises. Primarily because Python has an extensive and lively ecosystem of libraries for data science, machine learning and computer vision. Specifically, we will use extensively these libraries:

- `NumPy`, and `SciPy` for matrix manipulation
- `OpenCV` for image processing
- `TensorFlow` for deep learning (later)

Do not worry if you are not familiar with these libraries (or not with Python in general). Working through the exercises will help you to master these tools. If you do not know what a function does, simply search online for its documentation. There are also plenty of good tutorials. For example, here is a short introduction to `NumPy`: `http://cs231n.github.io/python-numpy-tutorial/`.
We recommend using Anaconda to manage Python libraries. Download it here and follow the instructions: `https://www.anaconda.com/distribution/`. Jupyter Notebook is already included in Anaconda distribution. If you choose not to use Anaconda, follow the instruction here to install Jupyter Notebook `https://jupyter.org/install`.
If you have not already installed OpenCV, we recommend you use the pre-built package for Python, instead of compiling it from source. There are several versions available. With Anaconda you can use:

```
conda install opencv3 -c menpo
```

It can be a good idea to collaborate with your fellow group members to make sure that everyone manages to set up their environment.

Questions

This first exercise contains five questions:

1. Gaussian Filtering
2. Fourier Transform
3. Image Derivatives
4. Edge Detection
5. Hough Transform

We recommend completing these questions in the order listed above. The corresponding Jupyter notebooks contain the actual questions. All images used should be put under `./data` directory.

For your convenience, the following pages include a PDF rendering of the Jupyter notebooks. Note, however, that several images are still incorrect at this point and will only be correct when you solve the exercise.

gaussian_filtering

April 23, 2019

1 Gaussian Filtering

In the following, you will implement a method which generates and applies a Gaussian filter for a given variance and number of samples.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        import cv2
```

1.1 Some convenience functions

```
In [2]: def imread_rgb(filename):
            """Read a color image from our data directory."""
            im = cv2.imread(f'../../data/{filename}')
            im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
            return im


        def plot_multiple(images, titles, colormap='gray', max_columns=np.inf, share_axes=True
            """Plot multiple images as subplots on a grid."""
            assert len(images) == len(titles)
            n_images = len(images)
            n_cols = min(max_columns, n_images)
            n_rows = int(np.ceil(n_images / n_cols))
            fig, axes = plt.subplots(
                n_rows, n_cols, figsize=(n_cols * 4, n_rows * 4),
                squeeze=False, sharex=share_axes, sharey=share_axes)

            axes = axes.flat
            # Hide subplots without content
            for ax in axes[n_images:]:
                ax.axis('off')

            if not isinstance(colormap, (list,tuple)):
                colormaps = [colormap]*n_images
```

```
    else:
        colormaps = colormap

    for ax, image, title, cmap in zip(axes, images, titles, colormaps):
        ax.imshow(image, cmap=cmap)
        ax.set_title(title)

    fig.tight_layout()
```

1.2 Part a

Start by writing a function gauss which creates a 1D Gaussian from a given vector of integer indices $x = [-w, \ldots, w]$:

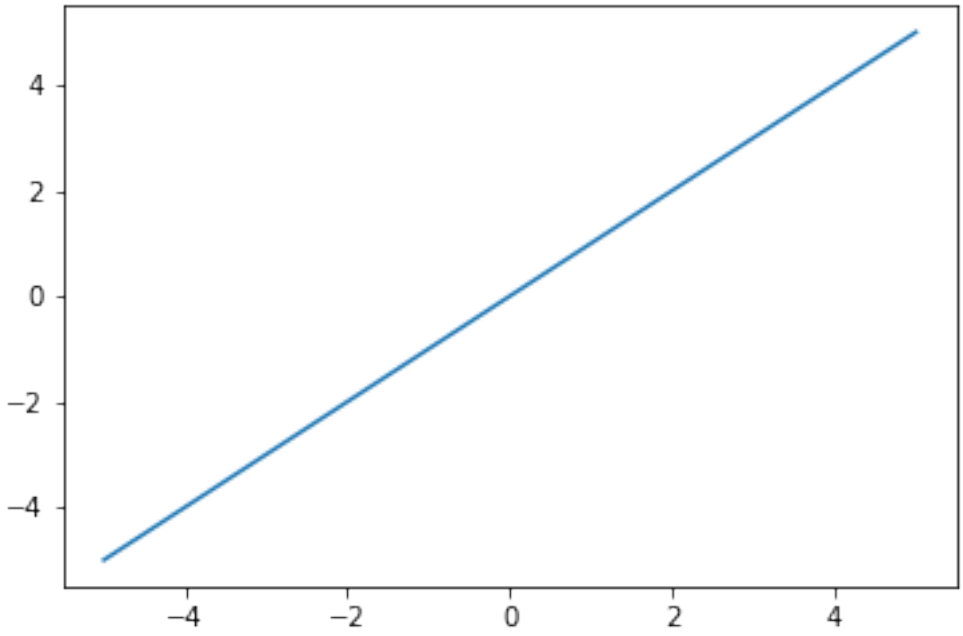
$$G[i] = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x[i]^2}{2\sigma^2}\right)$$

where σ is the standard deviation.

```
In [3]: def gauss(x, sigma):
            # Your code here
            return x
```

```
In [4]: x = np.linspace(-5, 5, 100)
        y = gauss(x, 1.0)
        fig, ax = plt.subplots()
        ax.plot(x, y)
```

```
Out[4]: [<matplotlib.lines.Line2D at 0x7f23b729e9b0>]
```



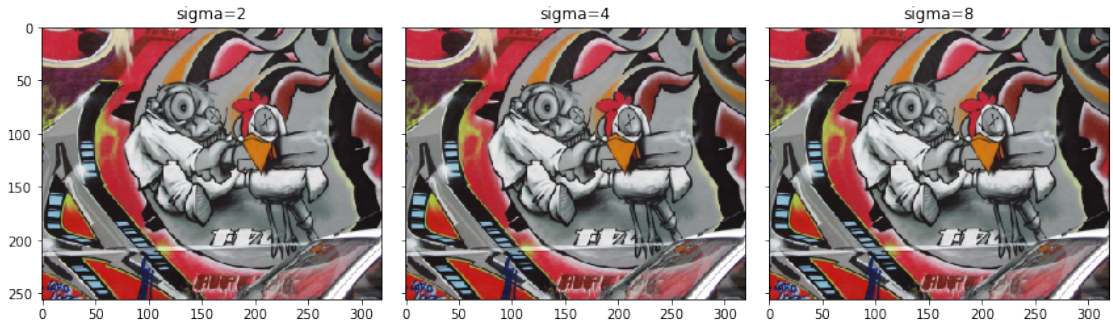
1.3 Part b

Use the above function to implement a function `gaussian_filter(image, sigma, padding)`, which first generates a Gaussian filter and then applies it to the `image`. The size of the filter should be $2 \cdot \lceil 3\sigma \rceil + 1$. Remember that the Gaussian is separable, *i.e.* that an equivalent 2D result can be obtained through a sequence of two 1D filtering operations. Do not use any existing implementation for convolution in this part (e.g. `scipy.ndimage.convolve`). However, you are allowed to use these implementations in the following questions.

```
In [5]: def gaussian_filter(image, sigma, padding=True):
            # Your code here
            return image
```

Read the image `graf_small.png` and apply the filters with `sigma = 2`, `4`, and `8`. Again, choose the kernel size as $2 \cdot \lceil 3\sigma \rceil + 1$. What do you observe?

```
In [6]: image = imread_rgb('graf_small.png')
        sigmas = [2, 4, 8]
        blurred_images = [gaussian_filter(image, s) for s in sigmas]
        titles = [f'sigma={s}' for s in sigmas]

        plot_multiple(blurred_images, titles)
```



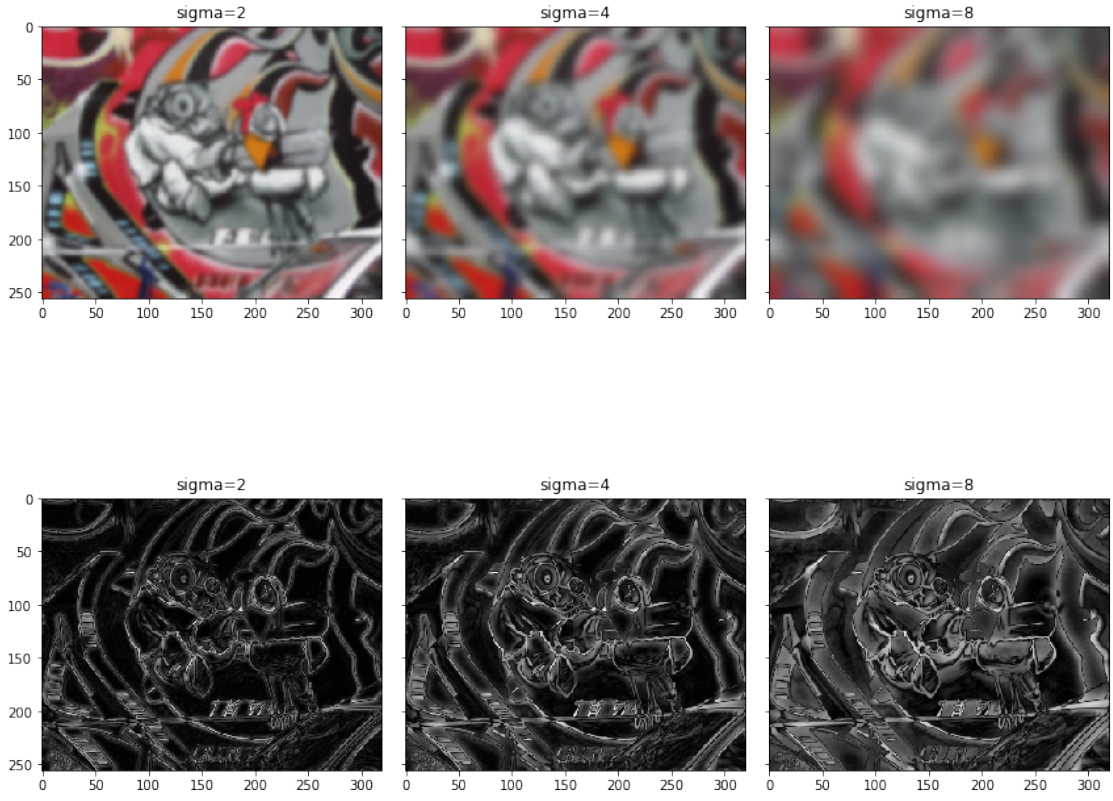
What do you observe? Type your answer here:

OpenCV has many built-in function for image smoothing. Check out this page: https://docs.opencv.org/3.1.0/d4/d13/tutorial_py_filtering.html

Compare the result of `cv2.GaussianBlur` with your own implementation by computing the difference image. Was your implementation correct?

```
In [7]: def gauss_cv(image, sigma):
            ks = 2 * int(3 * sigma) + 1
            return cv2.GaussianBlur(image, (ks, ks), sigma, cv2.BORDER_DEFAULT)

        def abs_diff(image1, image2):
```

```
    image1 = image1.astype(np.float32)
    image2 = image2.astype(np.float32)
    return np.mean(np.abs(image1-image2), axis=-1)

blurred_images_cv = [gauss_cv(image, s) for s in sigmas]
differences = [abs_diff(x,y) for x, y in zip(blurred_images, blurred_images_cv)]

plot_multiple(blurred_images_cv, titles)
plot_multiple(differences, titles)
```



Was your implementation correct? What do you observe? Type your answer here:

In [8]:

fourier

April 23, 2019

1 Fourier transform

In this part, we look at the effect of filtering in Fourier space.

This gives us a different way of looking at images and yields deeper insights to what is going on when we apply a filter or downsample an image.

```
In [1]: %matplotlib inline
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        import scipy.signal
        import numpy as np
        from scipy import ndimage
        import cv2
```

1.1 Some convenience functions

```
In [2]: def imread_gray(filename):
            """Read grayscale image from our data directory."""
            return cv2.imread(f'../../data/{filename}',
                              cv2.IMREAD_GRAYSCALE).astype(np.float32)

        def convolve_with_two(image, kernel1, kernel2):
            """Apply two filters, one after the other."""
            image = ndimage.convolve(image, kernel1, mode='wrap')
            image = ndimage.convolve(image, kernel2, mode='wrap')
            return image

        def fourier_spectrum(im):
            normalized_im = im / np.sum(im)
            f = np.fft.fft2(normalized_im)
            return np.fft.fftshift(f)

        def log_magnitude_spectrum(im):
            return np.log(np.abs(fourier_spectrum(im))+1e-8)

        def plot_with_spectra(images, titles):
            """Plots a list of images in the first column and the logarithm of their
```

```
        magnitude spectrum in the second column."""

        assert len(images) == len(titles)
        n_cols = 2
        n_rows = len(images)
        fig, axes = plt.subplots(
            n_rows, 2, figsize=(n_cols * 4, n_rows * 4),
            squeeze=False)

        spectra = [log_magnitude_spectrum(im) for im in images]

        lower = min(np.percentile(s, 0.1) for s in spectra)
        upper = min(np.percentile(s, 99.999) for s in spectra)
        normalizer = mpl.colors.Normalize(vmin=lower, vmax=upper)

        for ax, image, spectrum, title in zip(axes, images, spectra, titles):
            ax[0].imshow(image, cmap='gray')
            ax[0].set_title(title)
            ax[0].set_axis_off()
            c = ax[1].imshow(spectrum, norm=normalizer, cmap='viridis')
            ax[1].set_title('Log magnitude spectrum')
            ax[1].set_axis_off()

        fig.tight_layout()

    def generate_pattern():
        x = np.linspace(0, 1, 256, endpoint=False)
        y = np.sin(x**2 * 16 * np.pi)
        return np.outer(y,y)/2+0.5

    im_grass = imread_gray('grass.jpg')
    im_zebras = imread_gray('zebras.jpg')
    im_pattern = generate_pattern()
```

1.2 Plotting demo

This is how you can load example images and plot the logarithm of its magnitude spectrum.

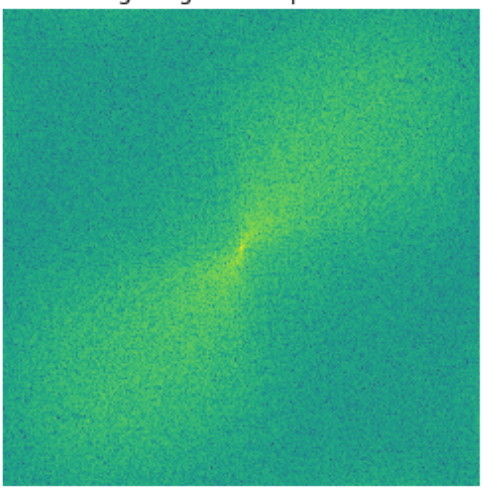
Low frequencies appear near the center and higher frequencies towards the outside.

Greener (lighter) colors mean higher values. The color scale is consistent within the subplots of a single plot, but may differ in different plots.

```
In [3]: plot_with_spectra([im_grass, im_zebras, im_pattern],
                          ['Grass image', 'Zebra image', 'Sine pattern'])
```

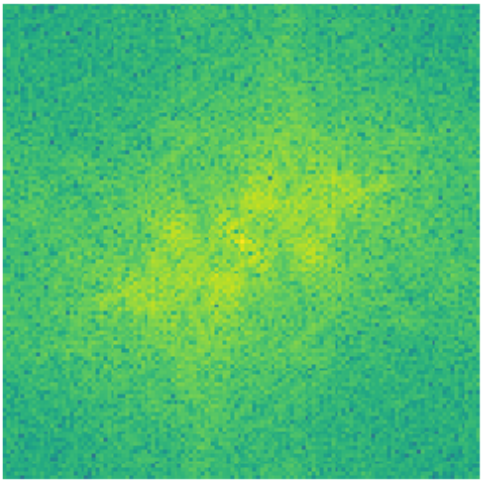
Grass image

Log magnitude spectrum

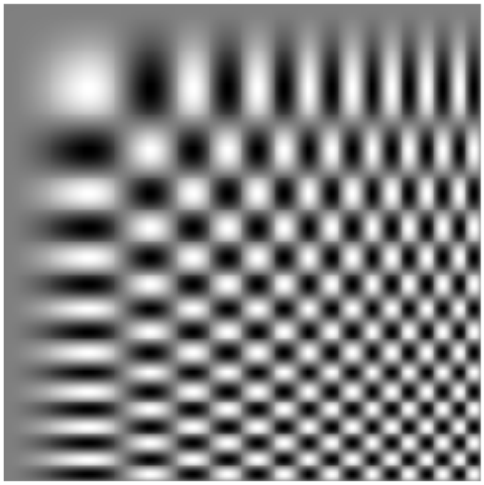


Zebra image

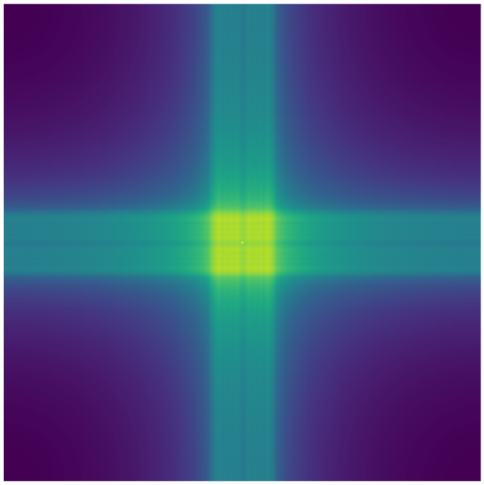
Log magnitude spectrum



Sine pattern



Log magnitude spectrum



1.3 (a) Blurring

Consider one of the images (`im_grass` is a good choice).

1.3.1 i)

Implement `filter_box(image, size)` that outputs the box-filtered version of `image`, using `convolve_with_two` (since the box filter is separable). The parameter `size` refers to the side length of the box filter.

1.3.2 ii)

Implement `filter_gauss(image, kernel_factor, sigma)` using `convolve_with_two`. The parameter `kernel_factor` defines the half size of the kernel relative to sigma (our rule of thumb from the lecture was to set this as 3).

Plot the image and its blurred versions (with the box and the Gauss filter) along with their spectra using `plot_with_spectra()`.

Vary the size of the box filter. What do you observe? For the Gaussian, change sigma. What happens if you increase or decrease the `kernel_factor` compared to our rule-of-thumb value 3?

```
In [4]: def filter_gauss(image, kernel_factor, sigma):
            pass # Change this

        def filter_box(image, size):
            pass # Change this

        im = im_grass
        box_filtered = im  # Change this
        gauss_filtered = im # Change this

        plot_with_spectra(
            [im, box_filtered, gauss_filtered],
            ['Image', 'Box filtered', 'Gauss filtered'])
```

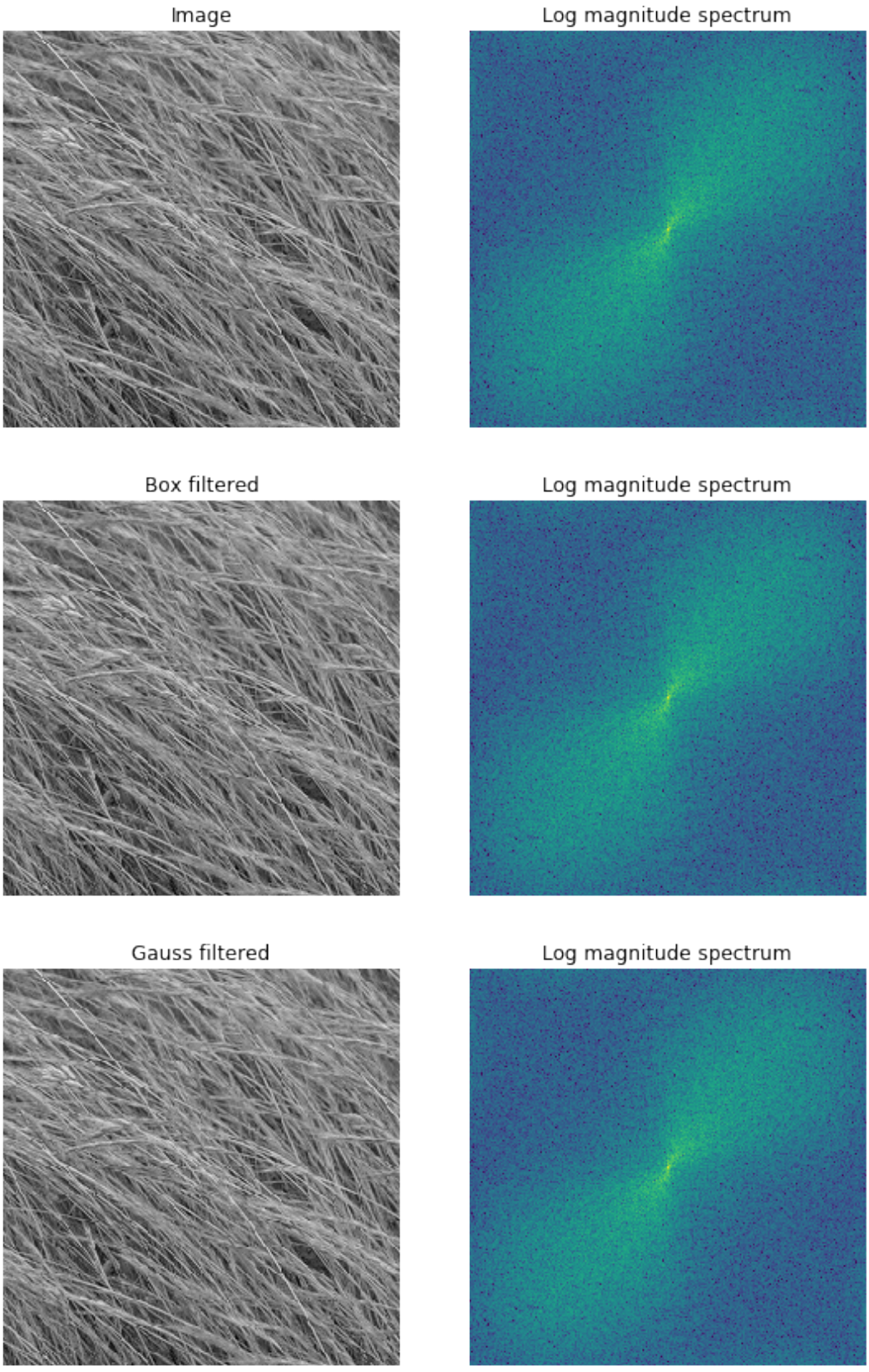


Image
Log magnitude spectrum
Box filtered
Log magnitude spectrum
Gauss filtered
Log magnitude spectrum

1.4 (b) Sampling and aliasing

1.4.1 i)

Implement a function `sample_with_gaps(image, period)`, where `period` is an integer and defines the distance between two sampled points in pixels. The output should have the same size as `image` but have zeros everywhere except at the sampled points, where it should be equal to `image`. For example if `period=2` then every second row and every second column of the image should be zero in the output.

Using `sample_with_gaps`, sample the `im_zebras` image with a period of 2 pixels and plot the original and sampled images along with their spectra.

1.4.2 ii)

Consider the image `im_pattern` and sample it with a period of 4, 8 and 16 and plot the resulting spectra. What happens as you increase the sampling period? Now apply **Gaussian blurring** before sampling, with different sigma values. Approximately what sigma do you need for avoiding artifacts when `period=16`?

1.4.3 iii)

Implement `sample_without_gaps(image, period)`, which is like `sample_with_gaps` but the output does not contain gaps (rows and columns of zeros) and therefore the output size is smaller. What effect do you see in the resulting magnitude spectrum, compared to `sample_with_gaps`?

```
In [5]: def sample_with_gaps(image, period):
            pass

        def sample_without_gaps(image, period):
            pass
```

image_derivatives

April 23, 2019

1 Image Derivatives

This exercise introduces image derivative operators.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        import cv2
        from scipy import ndimage
```

1.1 Some Convenience Functions.

```
In [2]: def convolve_with_two(image, kernel1, kernel2):
            """Apply two filters, one after the other."""
            image = ndimage.convolve(image, kernel1)
            image = ndimage.convolve(image, kernel2)
            return image

        def imread_gray(filename):
            """Read grayscale image from our data directory."""
            return cv2.imread(f'../../data/{filename}',
                              cv2.IMREAD_GRAYSCALE).astype(np.float32)

        def plot_multiple(images, titles, colormap='gray',
                          max_columns=np.inf, share_axes=True):
            """Plot multiple images as subplots on a grid."""
            assert len(images) == len(titles)
            n_images = len(images)
            n_cols = min(max_columns, n_images)
            n_rows = int(np.ceil(n_images / n_cols))
            fig, axes = plt.subplots(
                n_rows, n_cols, figsize=(n_cols * 4, n_rows * 4),
                squeeze=False, sharex=share_axes, sharey=share_axes)

            axes = axes.flat
            # Hide subplots without content
```

```
        for ax in axes[n_images:]:
            ax.axis('off')

        if not isinstance(colormap, (list,tuple)):
            colormaps = [colormap]*n_images
        else:
            colormaps = colormap

        for ax, image, title, cmap in zip(axes, images, titles, colormaps):
            ax.imshow(image, cmap=cmap)
            ax.set_title(title)

        fig.tight_layout()
```

```
In [3]: # From Question 1: Gaussian Filtering
        def gauss(x, sigma):
            return 1.0 / np.sqrt(2.0 * np.pi) / sigma * np.exp(- x**2 / 2.0 / sigma**2)
```

1.2 Part a

Implement a function for creating a Gaussian derivative filter in 1D according to the following equation

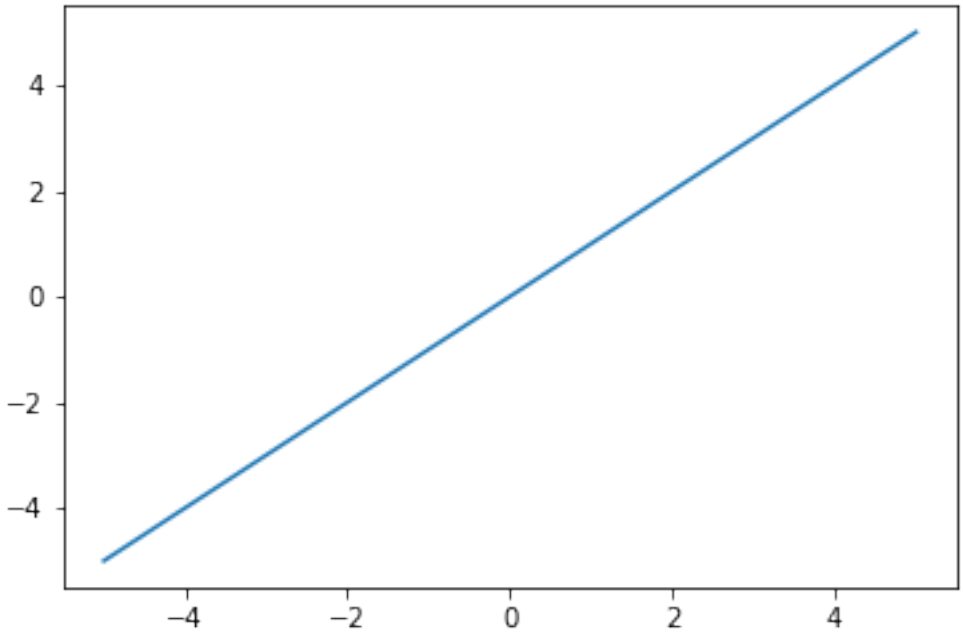
$$\frac{d}{dx}G = \frac{d}{dx}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{x^2}{2\sigma^2}\right) = -\frac{1}{\sqrt{2\pi}\sigma^3}x\exp\left(-\frac{x^2}{2\sigma^2}\right)$$

Your function should take a vector of integer values x and the standard deviation `sigma` as arguments.

```
In [4]: def gaussdx(x, sigma):
            # Your code here
            return x
```

```
In [5]: x = np.linspace(-5, 5, 100)
        y = gaussdx(x, sigma=1.0)
        fig, ax = plt.subplots()
        ax.plot(x, y)
```

```
Out[5]: [<matplotlib.lines.Line2D at 0x7fa6990034a8>]
```



The effect of applying a filter can be studied by observing its so-called *impulse response*. For this, create a test image in which only the central pixel has a non-zero value (called an *impulse*):

```
In [6]: impulse = np.zeros((25, 25), dtype=np.float32)
        impulse[12, 12] = 255
```

Now, create the following 1D filter kernels `gaussian` and `derivative`.

```
In [7]: sigma = 6.0
        kernel_radius = int(3.0 * sigma)
        x = np.arange(-kernel_radius, kernel_radius + 1)[np.newaxis]
        G = gauss(x, sigma)
        D = gaussdx(x, sigma)
```

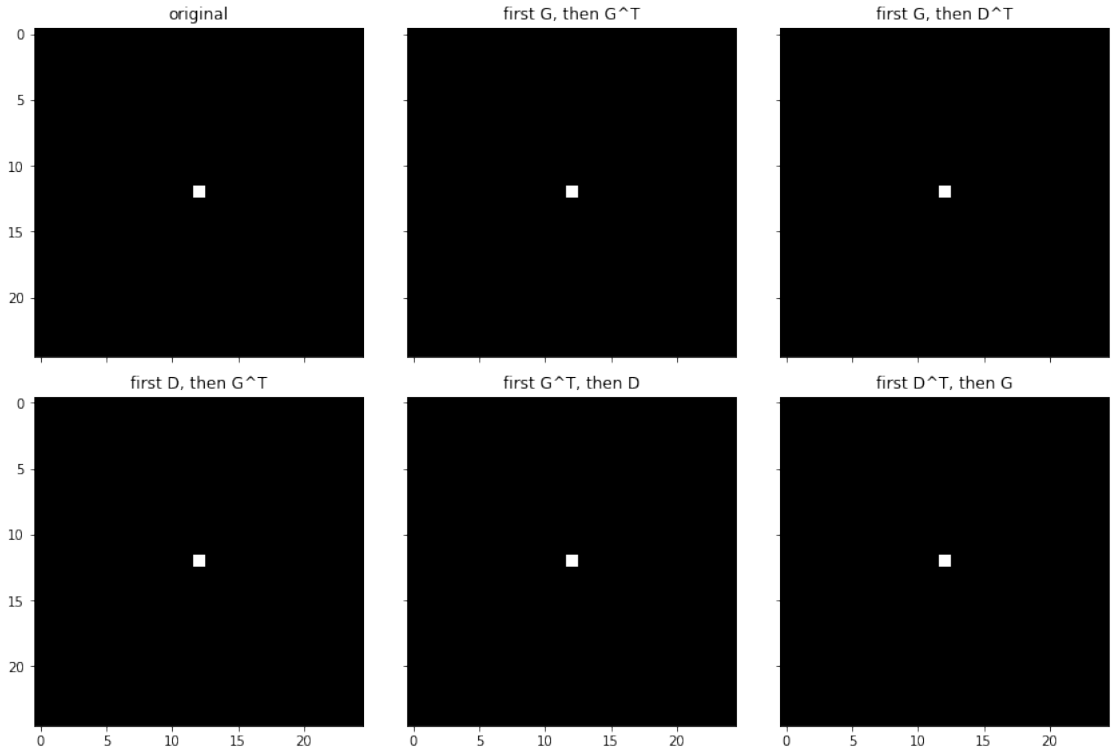
What happens when you apply the following filter combinations? (Hint: use `convolve_with_two` function at the beginning of this notebook.)

- first `gaussian`, then `gaussian^T`.
- first `gaussian`, then `derivative^T`.
- first `derivative`, then `gaussian^T`.
- first `gaussian^T`, then `derivative`.
- first `derivative^T`, then `gaussian`.

Display the result images with the `plot_multiple` function. Describe your result.

```
In [8]: # Modify this
        images = [
```

```
        impulse,
        impulse,
        impulse,
        impulse,
        impulse,
        impulse]

titles = [
    'original',
    'first G, then G^T',
    'first G, then D^T',
    'first D, then G^T',
    'first G^T, then D',
    'first D^T, then G']

plot_multiple(images, titles, max_columns=3)
```



1.3 Part b

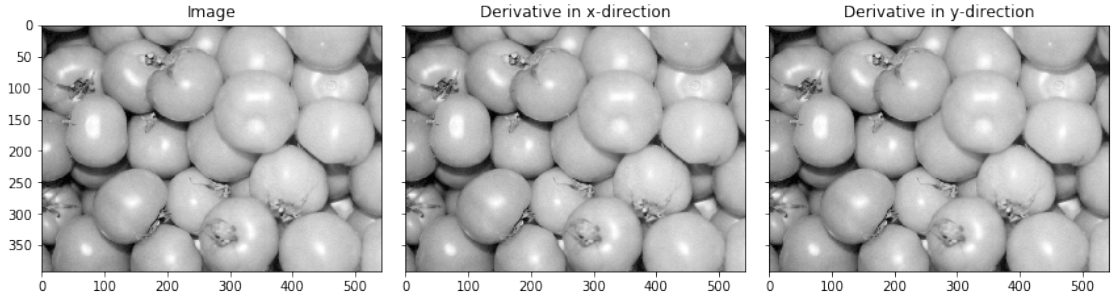
Use the functions `gauss` and `gaussdx` directly in order to create a new function `gauss_deriv` that returns the 2D Gaussian derivatives of an input image in x and y direction.

```
In [9]: def gauss_derivs(image, sigma):
            # Your code here.
```

```
    image_dx, image_dy = image, image
    return image_dx, image_dy
```

Try the function on the given example images and describe your results.

```
In [10]: image = imread_gray('tomatoes.png')
         grad_dx, grad_dy = gauss_derivs(image, sigma=5.0)
         plot_multiple([image, grad_dx, grad_dy],
                       ['Image', 'Derivative in x-direction', 'Derivative in y-direction'])
```

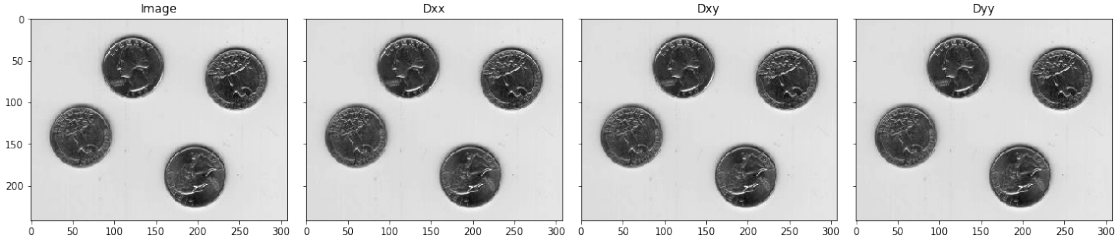


In a similar manner, create a new function `gauss_second_derivs` that returns the 2D second Gaussian derivatives $\frac{d^2}{dx^2}$, $\frac{d^2}{dxdy}$ and $\frac{d^2}{dy^2}$ of an input image.

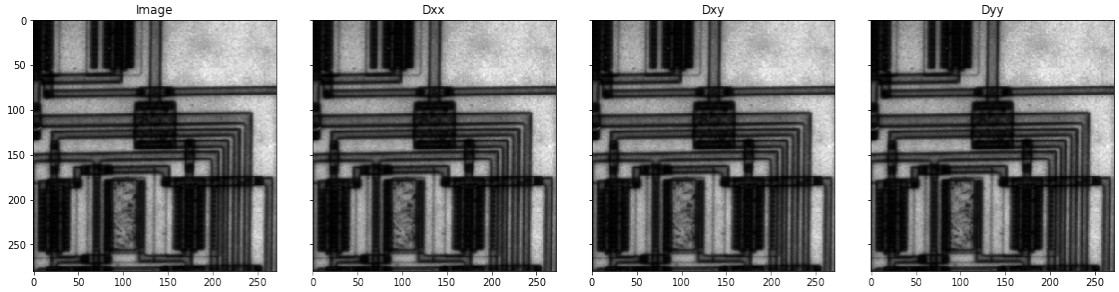
```
In [11]: def gauss_second_derivs(image, sigma):
             # Your code here
             image_dxx, image_dxy, image_dyy = image, image, image
             return image_dxx, image_dxy, image_dyy
```

Try the function on the given example images and describe your results.

```
In [12]: image = imread_gray('coins1.jpg')
         grad_dxx, grad_dxy, grad_dyy = gauss_second_derivs(image, sigma=2.0)
         plot_multiple([image, grad_dxx, grad_dxy, grad_dyy],
                       ['Image', 'Dxx', 'Dxy','Dyy'])
```



```
In [13]: image = imread_gray('circuit.png')
         grad_dxx, grad_dxy, grad_dyy = gauss_second_derivs(image, sigma=2.0)
         plot_multiple([image, grad_dxx, grad_dxy, grad_dyy],
                       ['Image', 'Dxx', 'Dxy','Dyy'])
```



Briefly describe the results here:

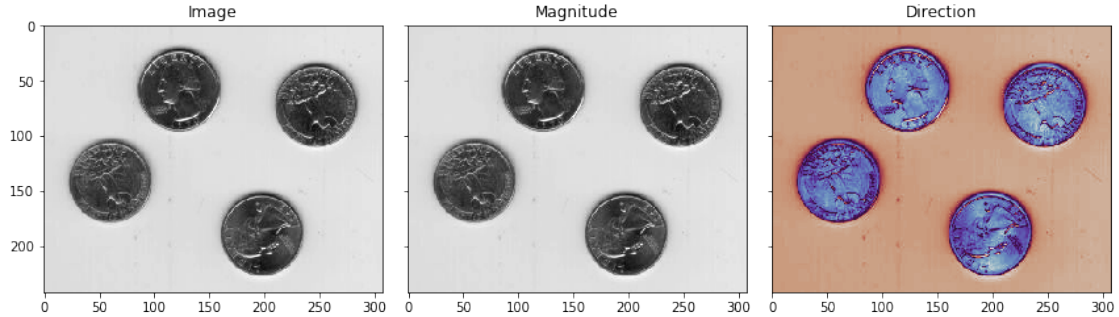
1.4 Part c

Create a new function `image_gradients_polar` that returns two images with the magnitude and orientation of the gradient for each pixel of the input image.

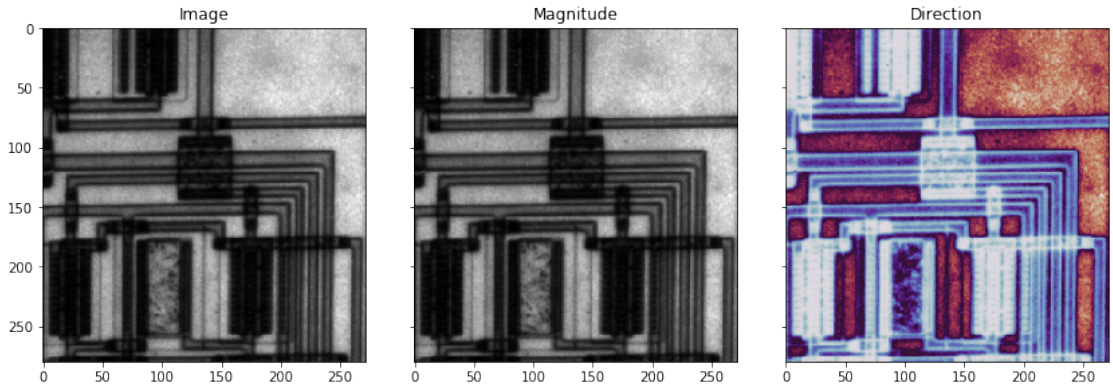
```
In [14]: def image_gradients_polar(image, sigma):
             # Your code here
             magnitude, direction = image, image
             return magnitude, direction
```

Try the function on the given example images and describe your results.

```
In [15]: image = imread_gray('coins1.jpg')
         grad_mag, grad_dir = image_gradients_polar(image, sigma=2.0)

         # Note: the twilight colormap only works since Matplotlib 3.0, use 'gray' in earlier
         plot_multiple([image, grad_mag, grad_dir],
                       ['Image', 'Magnitude', 'Direction'],
                       colormap=['gray', 'gray', 'twilight'])
```



```
In [16]: image = imread_gray('circuit.png')
         grad_mag, grad_theta = image_gradients_polar(image, sigma=2.0)
         plot_multiple([image, grad_mag, grad_theta],
                       ['Image', 'Magnitude', 'Direction'],
                       colormap=['gray', 'gray', 'twilight'])
```



Briefly describe your results here:

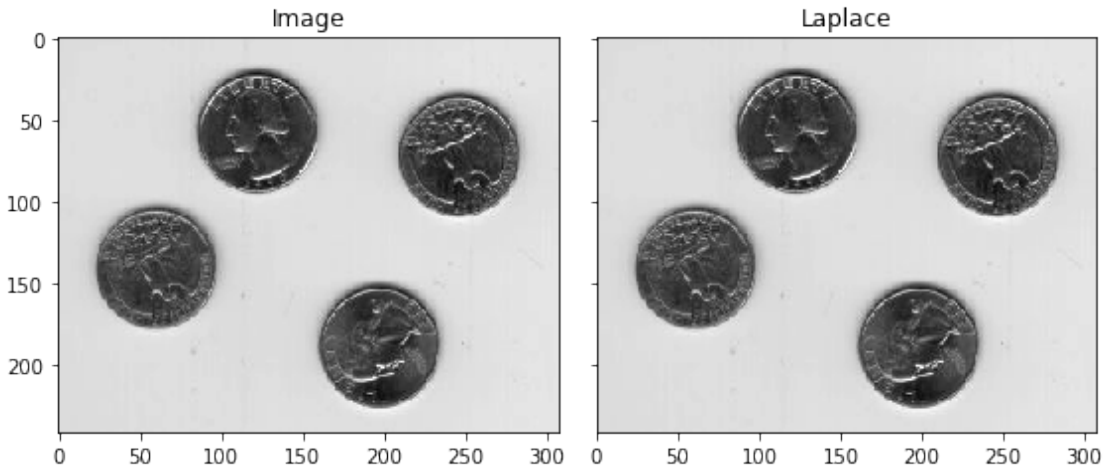
1.5 Part d

Create a new function `laplace` that returns an image with the Laplacian-of-Gaussian for each pixel of the input image.

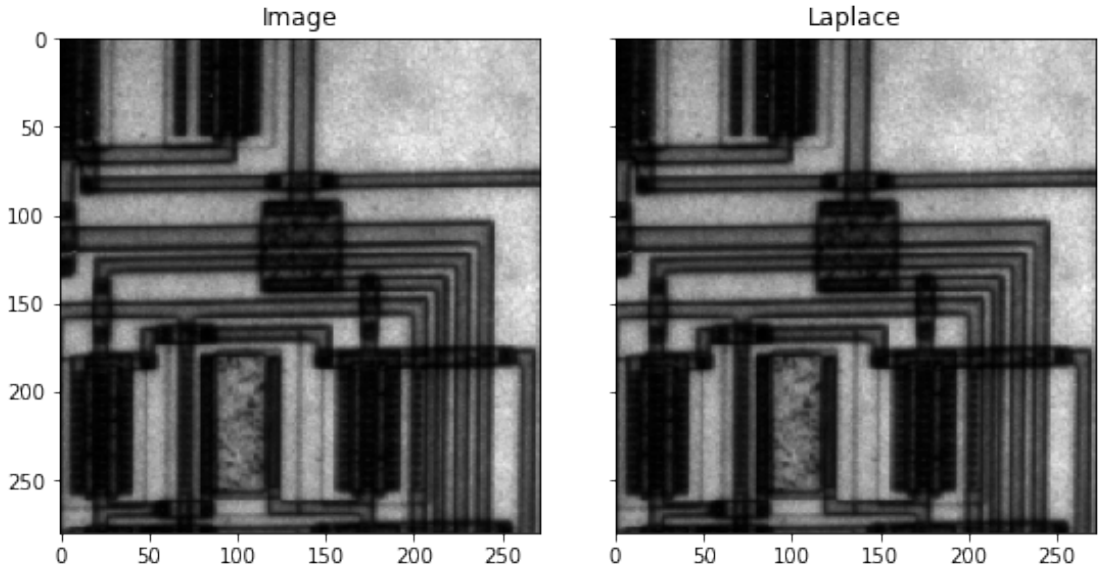
```
In [17]: def laplace(image, sigma):
             # Your code here
             lap = image
             return lap
```

Try the function on the given example images and describe your results.

```
In [18]: image = imread_gray('coins1.jpg')
         lap = laplace(image, sigma=2.0)
         plot_multiple([image, lap], ['Image', 'Laplace'])
```




```
In [19]: image = imread_gray('circuit.png')
         lap = laplace(image, sigma=2.0)
         plot_multiple([image, lap], ['Image', 'Laplace'])
```



Briefly describe your results here:

edge_detection

April 23, 2019

1 Edge Detection

In this exercise we will create a simple edge detector.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        import cv2
        from scipy import ndimage
```

1.1 Some convenience functions.

```
In [2]: def convolve_with_two(image, kernel1, kernel2):
            """Apply two filters, one after the other."""
            image = ndimage.convolve(image, kernel1)
            image = ndimage.convolve(image, kernel2)
            return image

        def imread_gray(filename):
            """Read grayscale image from our data directory."""
            return cv2.imread(f'../../data/{filename}',
                              cv2.IMREAD_GRAYSCALE).astype(np.float32)

        def plot_multiple(images, titles, colormap='gray',
                          max_columns=np.inf, imsize=4, share_axes=True):
            """Plot multiple images as subplots on a grid."""
            assert len(images) == len(titles)
            n_images = len(images)
            n_cols = min(max_columns, n_images)
            n_rows = int(np.ceil(n_images / n_cols))
            fig, axes = plt.subplots(
                n_rows, n_cols, figsize=(n_cols * imsize, n_rows * imsize),
                squeeze=False, sharex=share_axes, sharey=share_axes)

            axes = axes.flat
            # Hide subplots without content
```

```
    for ax in axes[n_images:]:
        ax.axis('off')

    if not isinstance(colormap, (list,tuple)):
        colormaps = [colormap]*n_images
    else:
        colormaps = colormap

    for ax, image, title, cmap in zip(axes, images, titles, colormaps):
        ax.imshow(image, cmap=cmap)
        ax.set_title(title)

    fig.tight_layout()
```

```
In [3]: # From Question 2: Image Derivatives
        def gauss_derivs(image, sigma):
            kernel_radius = int(3.0 * sigma)
            x = np.arange(-kernel_radius, kernel_radius + 1)[np.newaxis]

            # Compute 1D Gaussian kernel
            gaussderiv_kernel1d = (
                -1 / np.sqrt(2 * np.pi) / sigma**3 * x * np.exp(-x**2.0 / 2 / sigma**2))
            # Compute 1D Derivative-of-Gaussian kernel
            gauss_kernel1d = (
                1 / np.sqrt(2 * np.pi) / sigma * np.exp(-x**2 / 2 / sigma**2))

            image_dx = convolve_with_two(image, gaussderiv_kernel1d, gauss_kernel1d.T)
            image_dy = convolve_with_two(image, gauss_kernel1d, gaussderiv_kernel1d.T)
            return image_dx, image_dy

        def image_gradients_polar(image, sigma):
            dx, dy = gauss_derivs(image, sigma)
            magnitude = np.sqrt(dx**2 + dy**2)
            direction = np.arctan2(dy, dx)  # between -pi and +pi
            return magnitude, direction
```

1.2 Part a

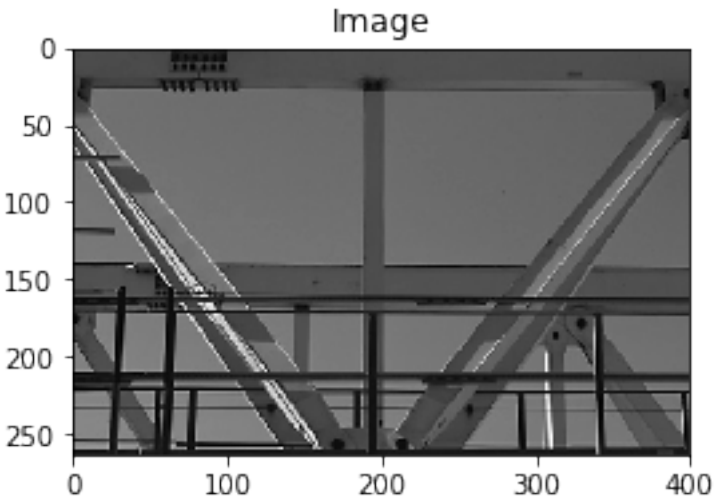
Write a function `get_edges` that returns a binary image `edge` from an input image where the color of each pixel p is selected as follows (for a given threshold `theta`):

$$p = \begin{cases} 1, \text{ if } |\text{grad}(\texttt{img})|\,(p) \geq \theta \\ 0, \text{ else} \end{cases}$$

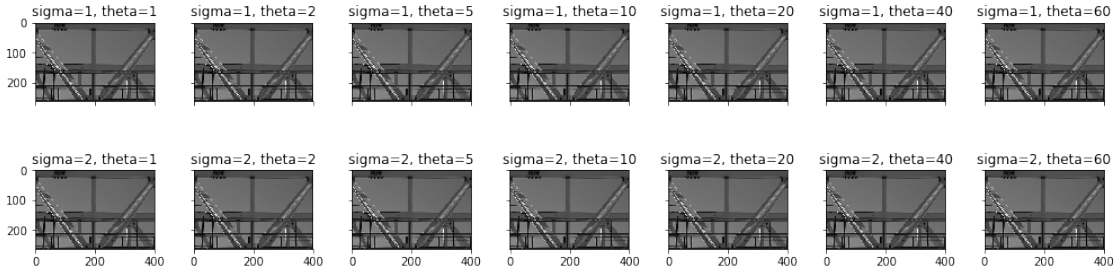
```
In [4]: def get_edges(image, sigma, theta):
            # Your code here
            edge = image
            return edge
```

Experiment with the function `get_edges` on the example images. Try to get good edge images for different values of `sigma`. What difficulties do you observe? (Note: it may pay off to look at the magnitude of the image gradient in order to get a feeling for suitable values of `theta`).

```
In [5]: image = imread_gray('gantrycrane.png')
        plot_multiple([image], ['Image'])
```




```
In [6]: # Try to play these values
        sigmas = [1, 2]
        thetas = [1, 2, 5, 10, 20, 40, 60]

        images = []
        titles = []
        for sigma in sigmas:
            for theta in thetas:
                edges = get_edges(image, sigma, theta)
                images.append(edges)
                titles.append(f'sigma={sigma}, theta={theta}')

        plot_multiple(images, titles, max_columns=7, imsize=2)
```



What difficulties do you observe? Type your answer here:

1.3 Part b

Using the above function, returned edges are still several pixels wide. In practice, this is often not desired. Create a function `get_edges_with_nms` that extends `get_edges` by using the following function to suppress non-maximum points along the gradient direction.

```
In [7]: def nms_for_canny(grad_mag, grad_dir):
            result = np.zeros_like(grad_mag)

            # Pre-define pixel index offset along different orientation
            offsets_x = [-1, -1, 0, 1, 1, 1, 0, -1, -1]
            offsets_y = [0, -1, -1, -1, 0, 1, 1, 1, 0]
            height, width = grad_mag.shape
            for y in range(1, height-1):
                for x in range(1, width-1):
                    d = grad_dir[y, x]
                    # Find neighboring pixels in direction of the gradient
                    # `d` is in radians, ranging from -pi to +pi
                    # make `idx` range from 0 to 8, quantized
                    idx = int(round((d + np.pi) / (2*np.pi) * 8))
                    ox, oy = offsets_x[idx], offsets_y[idx]

                    # Suppress all non-maximum points
                    # Note: this simplified code does not
                    # interpolate between neighboring pixels!
                    if ((grad_mag[y, x] > grad_mag[y + oy, x + ox]) and
                        (grad_mag[y, x] > grad_mag[y - oy, x - ox])):
                        result[y, x] = grad_mag[y, x]

            return result
```

Note that this simplified code does not interpolate between the neighboring pixel values in order to look up the real magnitude samples along the gradient direction. This interpolation is crucial to obtain the necessary robustness for an actual implementation. Here it was left out for better readability, since the interpolation involves some extra effort in order to deal with all special cases (e.g. exactly horizontal or vertical gradients). If you feel motivated, you can try to add this step to make the function more robust.

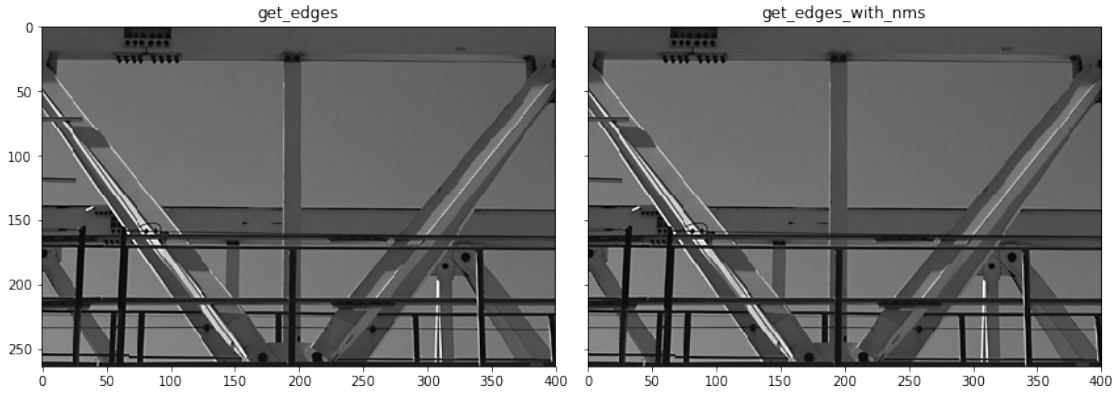
Another problem is that suitable values for `theta` may vary substantially between images. Extend the function `get_edges_with_nms` such that the threshold $\theta \in [0, 1]$ is defined relative to the maximal gradient magnitude value in the image.

```
In [8]: def get_edges_with_nms(image, sigma, theta):
            # Your code here
```

```
    edge = image
    return edge
```

Try your function on the given example images and describe your results.

```
In [9]: edges1 = get_edges(image, sigma=2, theta=5)
        edges2 = get_edges_with_nms(image, sigma=2, theta=0.17) # 0.17 corresponds to an absol

        plot_multiple([edges1, edges2],
                      ['get_edges', 'get_edges_with_nms'], imsize=6)
```



Describe your results here:

1.4 Part c

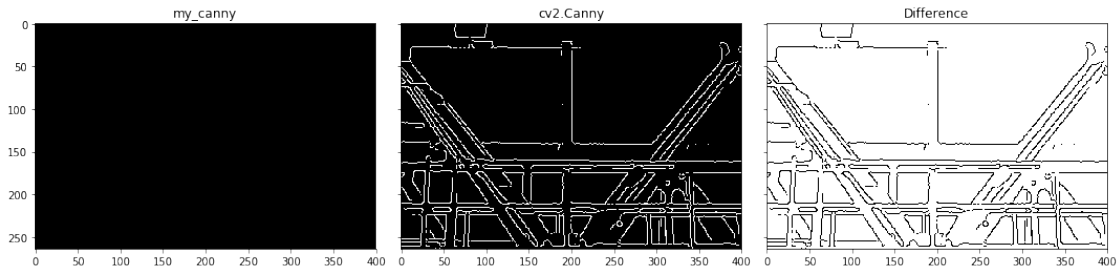
The function `get_edges` you implemented is a simplified version of the Canny edge detection pipeline. The main step that is still missing is the edge following with hysteresis thresholding. The idea here is that instead of applying a single threshold over the entire image, the edge detector works with two different thresholds `theta_high` and `theta_low`. It starts with an edge pixel with a value above `theta_high` and then follows the contour in the direction orthogonal to the gradient until the pixel value falls below `theta_low`. Each pixel visited along the way is labeled as an edge. The procedure is repeated until no further pixel above `theta_high` remains.

Try writing a function `my_canny` that implements this procedure. Don't worry about efficiency for the moment. The following hints may help you: - You can create a boolean array `visited` for already visited and yet-to-visit image pixels. Since we are not interested in pixels below the low threshold you can mark them as visited. In another boolean array you can flag the pixels that serve as starting points for line following. - You can also avoid having to deal with special cases along the image borders by creating a 1-pixel boundary where the `visited` flag is set to `true`. - The actual edge following part is most easily implemented as a recursive procedure. In most cases, you will have the option to choose between several possible continuation points. Again, the easiest way is to try all of them in sequence (or even all 8 neighbors) and let the recursive procedure (together with the `visited` flags) do the rest.

```
In [10]: def my_canny(image, sigma, theta_low, theta_high):
             # Output image
             image_out = np.zeros_like(image)

             def follow_edge(x, y):
                 visited[y, x] = True
                 image_out[y, x] = 255

                 # Pre-define pixel index offset along different orientation
                 offsets_x = [-1, -1, 0, 1, 1, 1, 0, -1]
                 offsets_y = [0, -1, -1, -1, 0, 1, 1, 1]

                 for ox, oy in zip(offsets_x, offsets_y):
                     # Note: `visited` is already False for points
                     # below the low threshold.
                     if not visited[y + oy, x + ox]:
                         follow_edge(x + ox, y + oy)

             # Your code here

             return image_out
```

OpenCV already provides built-in function that implements the Canny edge detector. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html Try `cv2.Canny` on the provided example images and compare the results to those of your implementation. Note: the implementation of `cv2.Canny` does not include bluring the image as we did. So apply `cv2.GaussianBlur` on the image, before passing to `cv2.Canny`. What do you observe?

```
In [11]: edge_canny = my_canny(image, sigma=2, theta_low=0.1, theta_high=0.3)

         blurred_cv = cv2.GaussianBlur(image, ksize=(7,7), sigmaX=2)
         edge_canny_cv = cv2.Canny(
             blurred_cv.astype(np.uint8),
             39, 72, L2gradient=True).astype(np.float32)

         plot_multiple([edge_canny, edge_canny_cv, edge_canny-edge_canny_cv],
                       ['my_canny', 'cv2.Canny', 'Difference'], imsize=5)
```



What do you observe?

1.5 Pard d (bonus)

This solution gives better results, but its results still depend strongly on the maximal gradient magnitude value in the image. For a cleaner solution, we want to adapt the threshold to the distribution of all gradient magnitude values, such that we can directly control the number of edge pixels we get. Extend the function `get_edges` by the following steps in order to do this:

- Perform non-maximum suppression on the gradient magnitude image as shown above.
- Transform the result image into a vector.
- Build a histogram of the remaining gradient magnitude values.
- Compute the cumulative sum over the histogram (except for the first cell).
- The last cell of the cumulative histogram now contains the total number of edge pixels in the image, `num_total_edge_pixels`. Compute the desired number of edge pixels `num_desired_edge_pixels` as the percentage `theta` of `num_total_edge_pixels`.
- Find the threshold for which the cumulative histogram contains the value `num_desired_edge_pixels`.

In [12]:

hough_transform

April 23, 2019

1 2D Structure Extraction (Hough Transform)

In this exercise, we will implement a Hough transform in order to detect parametric curves, such as lines or circles. In the following, we shortly review the motivation for this technique.

Consider the point $p = (\mathrm{x}, \mathrm{y})$ and the equation for a line $y = mx + c$. What are the lines that could pass through p? The answer is simple: all the lines for which m and c satisfy $\mathrm{y} = m\mathrm{x} + c$. Regarding (x, y) as fixed, the last equation is that of a line in (m, c)-space. Repeating this reasoning, a second point $p' = (\mathrm{x}', \mathrm{y}')$ will also have an associated line in parameter space, and the two lines will intersect at the point $(\tilde{m}, \tilde{c})$, which corresponds to the line connecting p and p'.

In order to find lines in the input image, we can thus pursue the following approach. We start with an empty accumulator array quantizing the parameter space for m and c. For each edge pixel in the input image, we then draw a line in the accumulator array and increment the corresponding cells. Edge pixels on the same line in the input image will produce intersecting lines in (m, c)-space and will thus reinforce the intersection point. Maxima in this array thus correspond to lines in the input image that many edge pixels agree on.

In practice, the parametrization in terms of m and c is problematic, since the slope m may become infinite. Instead, we use the following parametrization in polar coordinates:

$$\mathrm{x} \cos\theta + \mathrm{y} \sin\theta = \rho \tag{1}$$

This produces a sinusoidal curve in (ρ, θ)-space, but otherwise the procedure is unchanged.

The following sub-questions will guide you through the steps of building a Hough transform.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        import cv2
```

1.1 Some convenience functions

```
In [2]: def imread_gray(filename):
            """Read grayscale image from our data directory."""
            return cv2.imread(f'../../data/{filename}', cv2.IMREAD_GRAYSCALE)

        def imread_rgb(filename):
            """Read a color image from our data directory."""
```

```
    im = cv2.imread(f'../../data/{filename}', cv2.IMREAD_COLOR)
    return cv2.cvtColor(im, cv2.COLOR_BGR2RGB)

def plot_hough(image, edges, hough_space):
    fig, axes = plt.subplots(1, 3, figsize=(3 * 4, 4))
    axes = axes.flat

    axes[0].imshow(image, cmap='gray')
    axes[0].set_title('Image')

    axes[1].imshow(edges, cmap='gray')
    axes[1].set_title('Edges')

    axes[2].imshow(hough_space, cmap='hot')
    axes[2].set_title('Hough space')
    axes[2].set_xlabel('theta (index)')
    axes[2].set_ylabel('rho (index)')
    fig.tight_layout()

def plot_with_hough_lines(image_rgb, rhos, thetas):
    """Plot an image with lines drawn over it.
    The lines are specified as an array of rho values and
    and array of theta values.
    """

    rhos = np.asarray(rhos)
    thetas = np.asarray(thetas)

    # compute start and ending point of the line x*cos(theta)+y*sin(theta)=rho
    x0, x1 = 0, image.shape[1] - 1  # Draw line across image
    y0 = rhos / np.sin(thetas)
    y1 = (rhos - x1 * np.cos(thetas)) / np.sin(thetas)

    # Check out this page for more drawing function in OpenCV:
    # https://docs.opencv.org/3.1.0/dc/da5/tutorial_py_drawing_functions.html
    for yy0, yy1 in zip(y0, y1):
        cv2.line(image_rgb, (x0, int(yy0)), (x1, int(yy1)), color=(255, 0, 0))

    return image_rgb
```

1.2 Part a

Build up an accumulator array `votes_acc` for votes in the parameter space (ρ, θ). θ ranges from $-\pi/2$ to $\pi/2$, and ρ ranges from $-D$ to D, where D denotes the length of the image diagonal. Use `n_bins_rho` and `n_bins_theta` as the number of bins in each direction. Initially, the array should be filled with zeros.

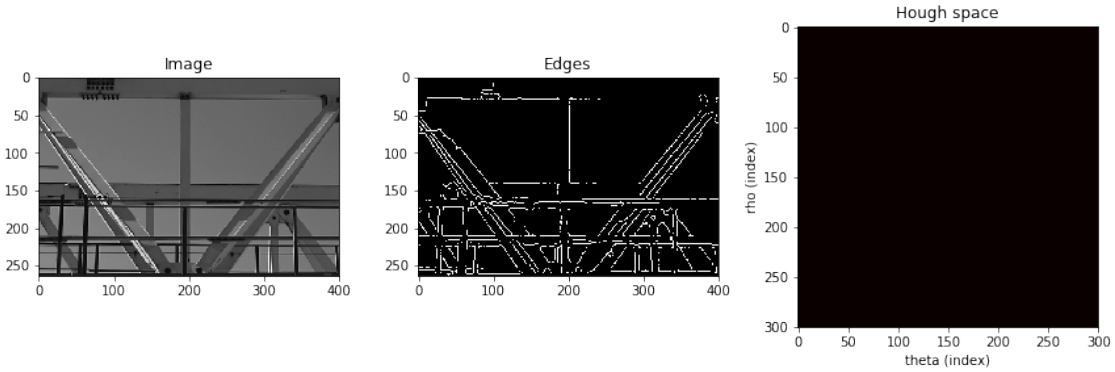
For each edge pixel in the input image, create the corresponding curve in (ρ, θ) space by evaluating above line equation for all values of θ and increment the corresponding cells of the accu-

mulator array.

```
In [3]: def hough_transform(edge_image, n_bins_rho, n_bins_theta):
            # Vote accumulator
            votes_acc = np.zeros((n_bins_rho, n_bins_theta), dtype=np.int)

            # Create bins
            diag = np.linalg.norm(edge_image.shape)  # Length of image diagonal
            theta_bins = np.linspace(-np.pi / 2, np.pi / 2, n_bins_theta)
            rho_bins = np.linspace(-diag, diag, n_bins_rho)

            # Implement Hough transform here

            return votes_acc, rho_bins, theta_bins
```

Test the implementation on an example image. Visualize the resulting Hough space by displaying it as a 2D image.

```
In [4]: image = imread_gray('gantrycrane.png')

        # Get edges using Canny
        sigma = 2
        kernel_size = 2 * int(3 * sigma) + 1
        blurred = cv2.GaussianBlur(image, (kernel_size, kernel_size), sigma)
        edges = cv2.Canny(blurred, threshold1=30, threshold2=90)  # 30, 90 are manually tuned

        n_bins_rho, n_bins_theta = 300, 300
        hough_space, rho_bins, theta_bins = hough_transform(edges, n_bins_rho, n_bins_theta)
        plot_hough(image, edges, hough_space)
```



1.3 Part b

Write a function `nms2d` which suppresses all points in the Hough space that are not local maxima. This can be achieved by looking at the 8 direct neighbors of each pixel and keeping only pixels

whose value is greater than all its neighbors. This function is simpler than the non-maximum suppression from the Canny Edge Detector since it does not take into account local gradients.

```
In [5]: def nms2d(image):
            image_out = np.zeros_like(image)
            # Your code here
            return image_out
```

Write a function `find_hough_peaks` that takes the result of `hough_transform` as an argument, finds the extrema in Hough space using `nms2d` and returns the index of all points (ρ_i, θ_i) for which the corresponding Hough value is greater than `threshold`.

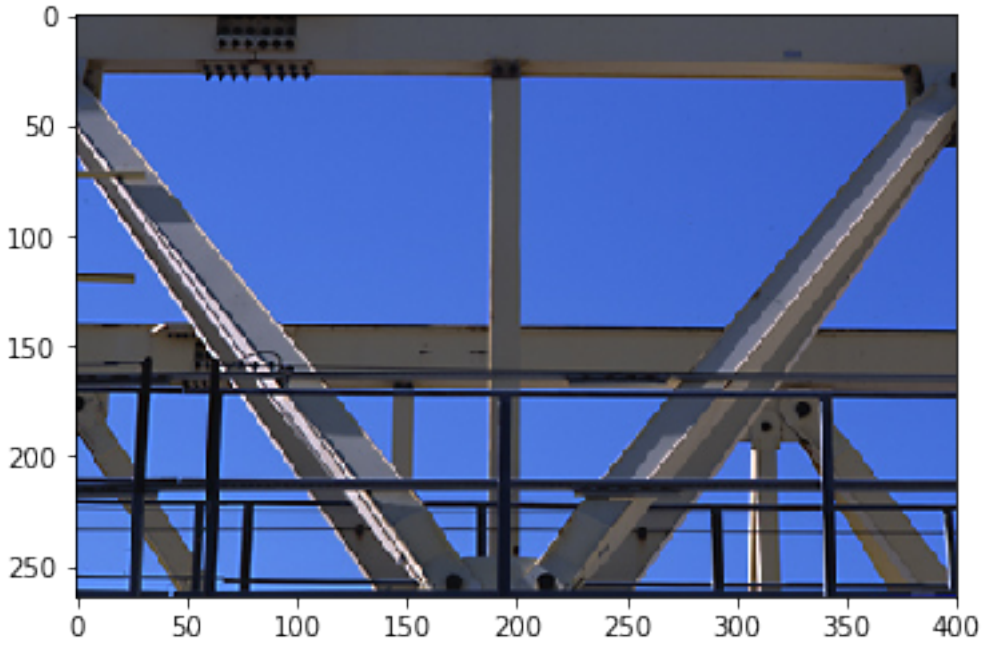
```
In [6]: def find_hough_peaks(hough_space, threshold):
            rho_max_index = [0, 1, 2]  # Change this
            theta_max_index = [0, 1, 2]  # Change this
            return rho_max_index, theta_max_index
```

Try your implementation on the images `gantrycrane.png` and `circuit.png`. Do you find all the lines?

```
In [7]: # Find maximum
        rho_max_idx, theta_max_idx = find_hough_peaks(hough_space, 250)
        print(f'gantrycrane.png: found {len(rho_max_idx)} lines in the image.')
        rho_max, theta_max = rho_bins[rho_max_idx], theta_bins[theta_max_idx]

        color_image = imread_rgb('gantrycrane.png')
        image_with_lines = plot_with_hough_lines(color_image, rho_max, theta_max)

        # Plot
        fig, ax = plt.subplots(figsize=(8, 4))
        ax.imshow(image_with_lines)
```

```
gantrycrane.png: found 3 lines in the image.
```

```
Out[7]: <matplotlib.image.AxesImage at 0x7f5ad2682828>
```



In [8]:
```
# Try another image
image = imread_gray('circuit.png')

sigma = 2
kernel_size = 2 * int(3 * sigma) + 1
blurred = cv2.GaussianBlur(image, (kernel_size, kernel_size), sigma)
edge = cv2.Canny(blurred, threshold1=30, threshold2=90)
hough_space, rho_bins, theta_bins = hough_transform(edge, n_bins_rho, n_bins_theta)

# Find maximum
rho_max_idx, theta_max_idx = find_hough_peaks(hough_space, 130)
print('circuit.png: found {} lines in the image.'.format(len(rho_max_idx)))
rho_max, theta_max = rho_bins[rho_max_idx], theta_bins[theta_max_idx]
color_image = cv2.cvtColor(image, cv2.COLOR_GRAY2RGB)
image_with_lines = plot_with_hough_lines(color_image, rho_max, theta_max)

# Plot
fig, ax = plt.subplots(figsize=(8, 4))
ax.imshow(image_with_lines)
```

```
circuit.png: found 3 lines in the image.
```

Out[8]:
```
<matplotlib.image.AxesImage at 0x7f5ad2671fd0>
```