Homework 1

This homework will cover convolutions and Canny edge detectors.

This notebook includes both coding and written questions. Please hand in this notebook file with all the outputs and your answers to the written questions.

Setup

```
import os
#get current directory
os.chdir('../..')
print(os.getcwd())

if not os.path.exists("CS131_release"):
  #Clone the repository if it doesn't already exist
  !git clone https://github.com/StanfordVL/CS131_release.git
```

```
/Users/jasonsun/Desktop/CS131/CS131_release/CS131_release
Cloning into 'CS131_release'...
remote: Enumerating objects: 2519, done.ote: Counting objects: 100%
(217/217), done.ote: Compressing objects: 100% (143/143), done.ote:
Total 2519 (delta 137), reused 77 (delta 73), pack-reused 2302 (from
2)
```

```
%cd CS131_release/winter_2025/hw1_release/
```

```
/Users/jasonsun/Desktop/CS131/CS131_release/CS131_release/
CS131_release/winter_2025/hw1_release
```

```
# Install the necessary dependencies
# (restart your runtime session if prompted to, and then re-run this
cell)

!pip install -r requirements.txt
```

```
Requirement already satisfied: numpy in
/opt/anaconda3/lib/python3.12/site-packages (from -r requirements.txt
(line 1)) (1.26.4)
Requirement already satisfied: scikit-image in
/opt/anaconda3/lib/python3.12/site-packages (from -r requirements.txt
(line 2)) (0.23.2)
Requirement already satisfied: scipy in
/opt/anaconda3/lib/python3.12/site-packages (from -r requirements.txt
(line 3)) (1.13.1)
Requirement already satisfied: matplotlib in
/opt/anaconda3/lib/python3.12/site-packages (from -r requirements.txt
(line 4)) (3.8.4)
Requirement already satisfied: networkx>=2.8 in
```

```
/opt/anaconda3/lib/python3.12/site-packages (from scikit-image->-r
requirements.txt (line 2)) (3.2.1)
Requirement already satisfied: pillow>=9.1 in
/opt/anaconda3/lib/python3.12/site-packages (from scikit-image->-r
requirements.txt (line 2)) (10.3.0)
Requirement already satisfied: imageio>=2.33 in
/opt/anaconda3/lib/python3.12/site-packages (from scikit-image->-r
requirements.txt (line 2)) (2.33.1)
Requirement already satisfied: tifffile>=2022.8.12 in
/opt/anaconda3/lib/python3.12/site-packages (from scikit-image->-r
requirements.txt (line 2)) (2023.4.12)
Requirement already satisfied: packaging>=21 in
/opt/anaconda3/lib/python3.12/site-packages (from scikit-image->-r
requirements.txt (line 2)) (23.2)
Requirement already satisfied: lazy-loader>=0.4 in
/opt/anaconda3/lib/python3.12/site-packages (from scikit-image->-r
requirements.txt (line 2)) (0.4)
Requirement already satisfied: contourpy>=1.0.1 in
/opt/anaconda3/lib/python3.12/site-packages (from matplotlib->-r
requirements.txt (line 4)) (1.2.0)
Requirement already satisfied: cycler>=0.10 in
/opt/anaconda3/lib/python3.12/site-packages (from matplotlib->-r
requirements.txt (line 4)) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/opt/anaconda3/lib/python3.12/site-packages (from matplotlib->-r
requirements.txt (line 4)) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in
/opt/anaconda3/lib/python3.12/site-packages (from matplotlib->-r
requirements.txt (line 4)) (1.4.4)
Requirement already satisfied: pyparsing>=2.3.1 in
/opt/anaconda3/lib/python3.12/site-packages (from matplotlib->-r
requirements.txt (line 4)) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in
/opt/anaconda3/lib/python3.12/site-packages (from matplotlib->-r
requirements.txt (line 4)) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in
/opt/anaconda3/lib/python3.12/site-packages (from python-
dateutil>=2.7->matplotlib->-r requirements.txt (line 4)) (1.16.0)
```

```
# Setup
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from time import time
from skimage import io


%matplotlib inline
plt.rcParams['figure.figsize'] = (15.0, 12.0) # set default size of
```

```
plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Part 1: Convolutions (50 points)

1.1 Commutative Property (5 points)

Recall that the convolution of an image $f: R^2 \rightarrow R$ and a kernel $h: R^2 \rightarrow R$ is defined as follows:

$$(f*h)[m,n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i,j] \cdot h[m-i, n-j]$$

Or equivalently, \begin{align} (f*h)[m,n] &= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h[i,j] \cdot f[m-i,n-j] \\ &= (h*f)[m,n] \end{align}

Show that this is true (i.e. prove that the convolution operator is commutative: $f*h = h*f$).

Your Answer: *Write your solution in this markdown cell. Please write your equations in LaTex equations.*

1. Let us substitute:

\begin{align} u &= m - i \\ v &= n - j \\ i &= m - u \\ j &= n - v \\ \end{align}

1. Therefore substituting i and j in the original equation

$$(f*h)[m,n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i,j] \cdot h[m-i, n-j]$$

2. it becomes:

$$(f*h)[m,n] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[m-u, n-v] \cdot h[u,v]$$

3. This form is the same as the alternate equation:

$$(f*h)[m,n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h[i,j] \cdot f[m-i, n-j]$$

4. Therefore $f*h = h*f$ is true.

1.2 Shift Invariance (5 points)

Let f be a function $R^2 \rightarrow R$. Consider a system $f \xrightarrow{S} g$, where $g=(f*h)$ with some kernel $h: R^2 \rightarrow R$. Also consider functions $f'[m,n]=f[m-m_0, n-n_0]$ and $g'[m,n]=g[m-m_0, n-n_0]$.

Show that S defined by any kernel h is a shift invariant system by showing that $g'=(f'*h)$.

Your Answer: *Write your solution in this markdown cell. Please write your equations in LaTex equations.*

1. Let the convolution be:

$$(f*h)[m,n]=\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} f[i,j]\cdot h[m-i,n-j]=g[m,n]$$

2. Shifted f and g are respectively:

$$f'[m,n]=f[m-m_0, n-n_0]$$

$$g'[m,n]=g[m-m_0, n-n_0]$$

3. Therefore the convolution on the shifted f, f' is:

$$(f'*h)[m,n]=\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} f[i-m_0, j-n_0]\cdot h[m-i, n-j]$$

4. And g' is:

$$g'[m,n]=\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} f[i,j]\cdot h[m-m_0-i, n-n_0-j]$$

5. Introduce a substitution where $a=i-m_0$ and $b=j-n_0$ to f'

$$(f'*h)[m,n]=\sum_{a=-\infty}^{\infty}\sum_{b=-\infty}^{\infty} f[a,b]\cdot h[m-m_0-a, n-n_0-b]$$

6. By inspection, the formulas for $f'*h$ and $g'[m,n]$ are identical (i.e. $g'=(f'*h)$), therefore S by kernal h is shift invariant

1.3 Linearity (10 points)

Recall that a system S is considered a linear system if and only if it satisfies the superposition property. In mathematical terms, a (function) S is a linear invariant system iff it satisfies:

$$S\{\alpha f_1[n,m]+\beta f_2[n,m]\}=\alpha S\{f_1[n,m]\}+\beta S\{f_2[n,m]\}$$

Let f_1 and f_2 be functions $R^2 \rightarrow R$. Consider a system $f \xrightarrow{S} g$, where $g=(f*h)$ with some kernel $h: R^2 \rightarrow R$.

Prove that S defined by any kernel h is linear by showing that the superposition property holds.

Your Answer: *Write your solution in this markdown cell. Please write your equations in LaTex equations.*

1. Let f_1 and f_2 be expanded in convolution form with kernal h:

$$(f_1 * h)[m,n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f_1[i,j] \cdot h[m-i, n-j] = g_1[m,n]$$

$$(f_2 * h)[m,n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f_2[i,j] \cdot h[m-i, n-j] = g_2[m,n]$$

2. Apply LHS combination of f_1 and f_2 by ratios α and β respectively:

$$S\{\alpha f_1[n,m] + \beta f_2[n,m]\} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} (\alpha f_1[n,m] + \beta f_2[n,m]) \cdot h[m-i, n-j]$$

3. Expand the formula out:

$$S\{\alpha f_1[n,m] + \beta f_2[n,m]\} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \alpha f_1[n,m] \cdot h[m-i, n-j] + \beta f_2[n,m] \cdot h[m-i, n-j]$$

4. By inspection, the two terms on the RHS can be separated into two formulas outside of the summation:

$$S\{\alpha f_1[n,m] + \beta f_2[n,m]\} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \alpha f_1[n,m] \cdot h[m-i, n-j] + \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \beta f_2[n,m] \cdot h[m-i, n-j]$$

5. Extract constant terms to be outside of the summations:

$$S\{\alpha f_1[n,m] + \beta f_2[n,m]\} = \alpha \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f_1[n,m] \cdot h[m-i, n-j] + \beta \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f_2[n,m] \cdot h[m-i, n-j]$$

6. Using definition in 1., revert to original transformation equations

$$S\{\alpha f_1[n,m] + \beta f_2[n,m]\} = \alpha (f_1 * h)[m,n] + \beta (f_2 * h)[m,n]$$

7. Convert to original S equation form and the condition holds, proving system S with kernal h is linear. QED

$$S\{\alpha f_1[n,m] + \beta f_2[n,m]\} = \alpha S\{f_1[n,m]\} + \beta S\{f_2[n,m]\}$$

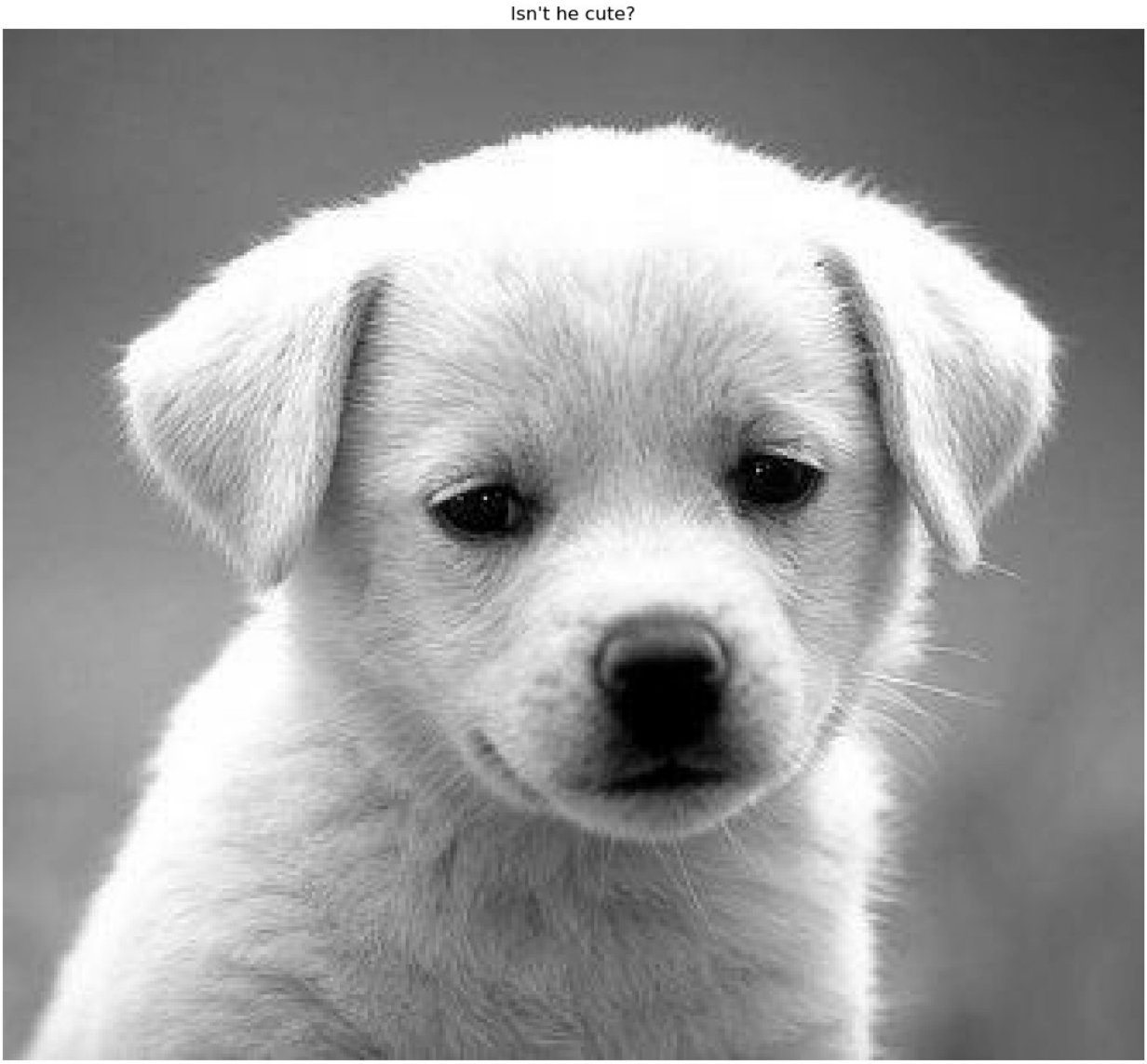
1.4 Implementation (30 points)

In this section, you will implement two versions of convolution:

- `conv_nested`
- `conv_fast`

First, run the code cell below to load the image to work with.

```
# Open image as grayscale
img = io.imread('dog.jpg', as_gray=True)

# Show image
plt.imshow(img)
plt.axis('off')
plt.title("Isn't he cute?")
plt.show()
```



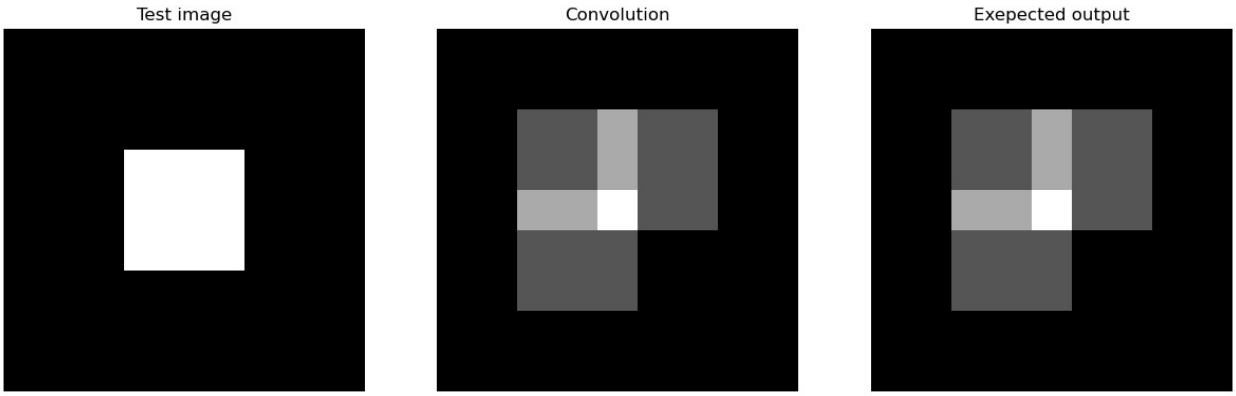
Now, implement the function **`conv_nested`** below. This is a naive implementation of convolution which uses 4 nested for-loops. It takes an image f and a kernel h as inputs and outputs the convolved image $(f*h)$ that has the same shape as the input image. This implementation should take a few seconds to run.

- Hint: It may be easier to implement $(h*f)$

We'll first test your `conv_nested` function on a simple input.

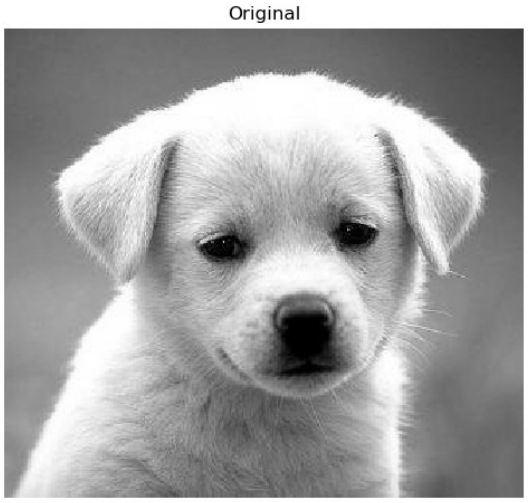
```
def conv_nested(image, kernel):
    """A naive implementation of convolution filter.

    This is a naive implementation of convolution using 4 nested for-
loops.
    This function computes convolution of an image with a kernel and
outputs
    the result that has the same shape as the input image.

    Args:
        image: numpy array of shape (Hi, Wi).
        kernel: numpy array of shape (Hk, Wk). Dimensions will be odd.

    Returns:
        out: numpy array of shape (Hi, Wi).
    """
    Hi, Wi = image.shape
    Hk, Wk = kernel.shape
    out = np.zeros((Hi, Wi))

    ### YOUR CODE HERE



    padded_image = np.pad(image, ((Hk//2, Hk//2), (Wk//2, Wk//2)),
mode='constant')


    for i in range(Hi):
        for j in range(Wi):
            for m in range(Hk):
                for n in range(Wk):
                    out[i, j] += kernel[Hk - 1 - m, Wk - 1 - n] *
padded_image[i + m, j + n]

    ### END YOUR CODE

    return out


# Simple convolution kernel.
kernel = np.array(
[
    [1,0,1],
    [0,0,0],
    [1,0,0]
```

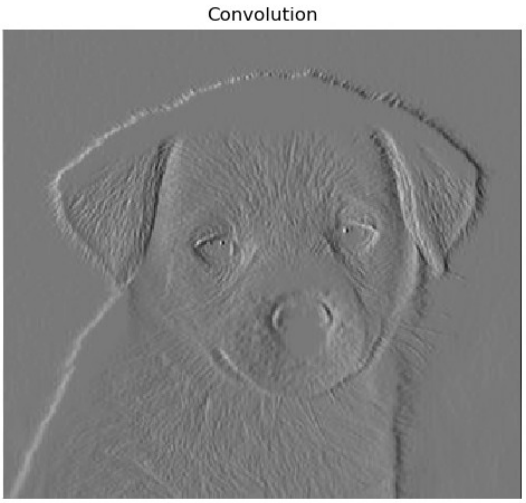
```
])

# Create a test image: a white square in the middle
test_img = np.zeros((9, 9))
test_img[3:6, 3:6] = 1

# Run your conv_nested function on the test image
test_output = conv_nested(test_img, kernel)

# Build the expected output
expected_output = np.zeros((9, 9))
expected_output[2:7, 2:7] = 1
expected_output[5:, 5:] = 0
expected_output[4, 2:5] = 2
expected_output[2:5, 4] = 2
expected_output[4, 4] = 3

# Plot the test image
plt.subplot(1,3,1)
plt.imshow(test_img)
plt.title('Test image')
plt.axis('off')

# Plot your convolved image
plt.subplot(1,3,2)
plt.imshow(test_output)
plt.title('Convolution')
plt.axis('off')

# Plot the exepected output
plt.subplot(1,3,3)
plt.imshow(expected_output)
plt.title('Exepected output')
plt.axis('off')
plt.show()

# Test if the output matches expected output
assert np.max(test_output - expected_output) < 1e-10, "Your solution
is not correct."
```



Now let's test your `conv_nested` function on a real image.

```
# Simple convolution kernel.
# Feel free to change the kernel to see different outputs.
kernel = np.array(
[
    [1,0,-1],
    [2,0,-2],
    [1,0,-1]
])

out = conv_nested(img, kernel)

# Plot original image
plt.subplot(2,2,1)
plt.imshow(img)
plt.title('Original')
plt.axis('off')

# Plot your convolved image
plt.subplot(2,2,3)
plt.imshow(out)
plt.title('Convolution')
plt.axis('off')

# Plot what you should get
solution_img = io.imread('convolved_dog.png', as_gray=True)
plt.subplot(2,2,4)
plt.imshow(solution_img)
plt.title('What you should get')
plt.axis('off')

plt.show()
```



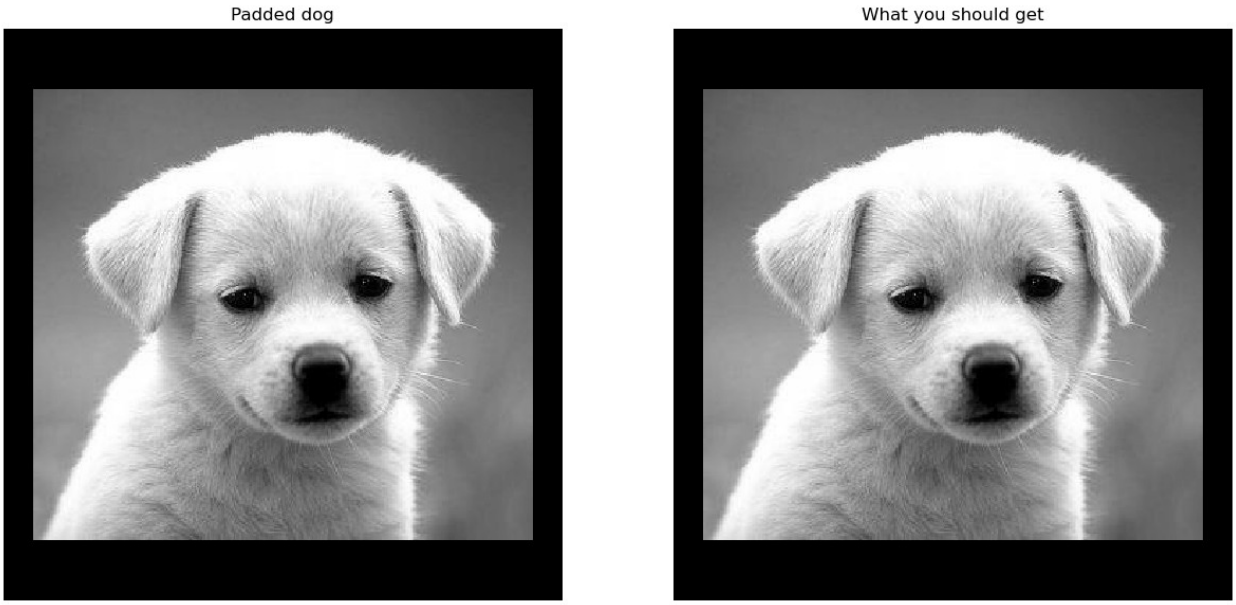


Let us implement a more efficient version of convolution using array operations in numpy. As shown in the lecture, a convolution can be considered as a sliding window that computes sum of the pixel values weighted by the flipped kernel. The faster version will i) zero-pad an image, ii) flip the kernel horizontally and vertically, and iii) compute weighted sum of the neighborhood at each pixel.

First, implement the function **`zero_pad`** below.

```
def zero_pad(image, pad_height, pad_width):
    """ Zero-pad an image.

    Ex: a 1x1 image [[1]] with pad_height = 1, pad_width = 2 becomes:

        [[0, 0, 0, 0, 0],
         [0, 0, 1, 0, 0],
         [0, 0, 0, 0, 0]]         of shape (3, 5)

    Args:
```

```
        image: numpy array of shape (H, W).
        pad_width: width of the zero padding (left and right padding).
        pad_height: height of the zero padding (bottom and top
padding).

    Returns:
        out: numpy array of shape (H+2*pad_height, W+2*pad_width).
    """

    H, W = image.shape
    out = None

    ### YOUR CODE HERE
    out = np.pad(image, ((pad_height,pad_height),
(pad_width,pad_width)), mode='constant', constant_values=0)
    ### END YOUR CODE
    return out



pad_width = 20 # width of the padding on the left and right
pad_height = 40 # height of the padding on the top and bottom

padded_img = zero_pad(img, pad_height, pad_width)

# Plot your padded dog
plt.subplot(1,2,1)
plt.imshow(padded_img)
plt.title('Padded dog')
plt.axis('off')

# Plot what you should get
solution_img = io.imread('padded_dog.jpg', as_gray=True)
plt.subplot(1,2,2)
plt.imshow(solution_img)
plt.title('What you should get')
plt.axis('off')

plt.show()
```

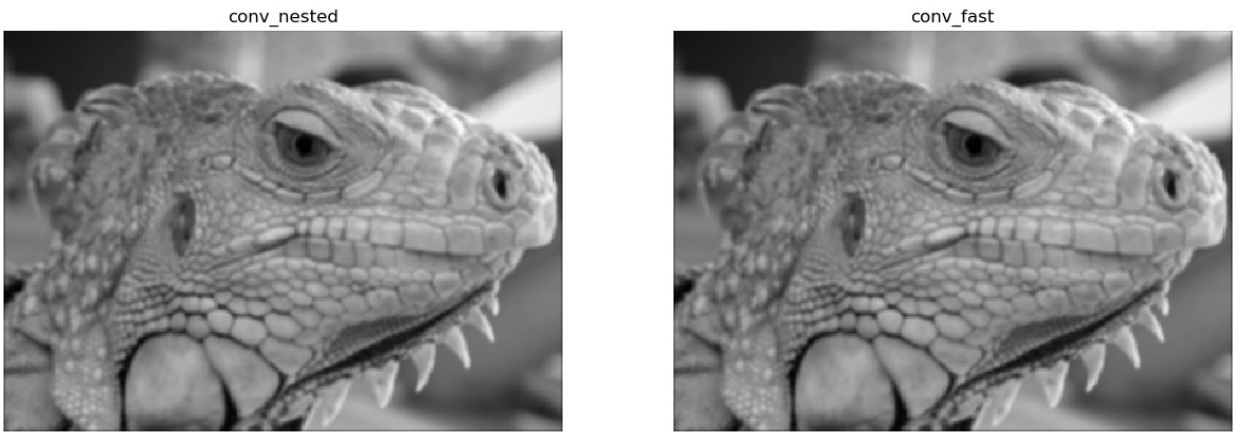


Next, complete the function **`conv_fast`** below using `zero_pad`. Run the code below to compare the outputs by the two implementations. `conv_fast` should run noticeably faster than `conv_nested`.

```
def conv_fast(image, kernel):
    """ An efficient implementation of convolution filter.

    This function uses element-wise multiplication and np.sum()
    to efficiently compute weighted sum of neighborhood at each
    pixel.

    Hints:
        - Use the zero_pad function you implemented above
        - There should be two nested for-loops
        - You may find np.flip() and np.sum() useful

    Args:
        image: numpy array of shape (Hi, Wi).
        kernel: numpy array of shape (Hk, Wk). Dimensions will be odd.

    Returns:
        out: numpy array of shape (Hi, Wi).
    """
    Hi, Wi = image.shape
    Hk, Wk = kernel.shape
    out = np.zeros((Hi, Wi))
    flipped_kernel = np.flip(kernel)
    padded_image = zero_pad(image, Hk//2, Wk//2)
    ### YOUR CODE HERE
    for i in range(Hi):
        for j in range(Wi):
```

```
            region = padded_image[i:i+Hk,j:j+Wk]
            out[i, j] = np.sum(flipped_kernel * region)
    ### END YOUR CODE

    return out


t0 = time()
out_fast = conv_fast(img, kernel)
t1 = time()
out_nested = conv_nested(img, kernel)
t2 = time()

# Compare the running time of the two implementations
print("conv_nested: took %f seconds." % (t2 - t1))
print("conv_fast: took %f seconds." % (t1 - t0))

# Plot conv_nested output
plt.subplot(1,2,1)
plt.imshow(out_nested)
plt.title('conv_nested')
plt.axis('off')

# Plot conv_fast output
plt.subplot(1,2,2)
plt.imshow(out_fast)
plt.title('conv_fast')
plt.axis('off')

# Make sure that the two outputs are the same
if not (np.max(out_fast - out_nested) < 1e-10):
    print("Different outputs! Check your implementation.")
```

```
conv_nested: took 0.825820 seconds.
conv_fast: took 0.247771 seconds.
```



Part 2: Cross-correlation (30 points)

Cross-correlation of an image f with a template g is defined as follows:

$$(g * f)[m,n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} g[i,j] \cdot f[m+i, n+j]$$

2.1 Template Matching with Cross-correlation (12 points)

Suppose that you are a clerk at a grocery store. One of your responsibilites is to check the shelves periodically and stock them up whenever there are sold-out items. You got tired of this laborious task and decided to build a computer vision system that keeps track of the items on the shelf.

Luckily, you have learned in CS131 that cross-correlation can be used for template matching: a template g is multiplied with regions of a larger image f to measure how similar each region is to the template.

The template of a product (`template.jpg`) and the image of shelf (`shelf.jpg`) is provided. We will use cross-correlation to find the product in the shelf.

Implement **`cross_correlation`** function below and run the code below.

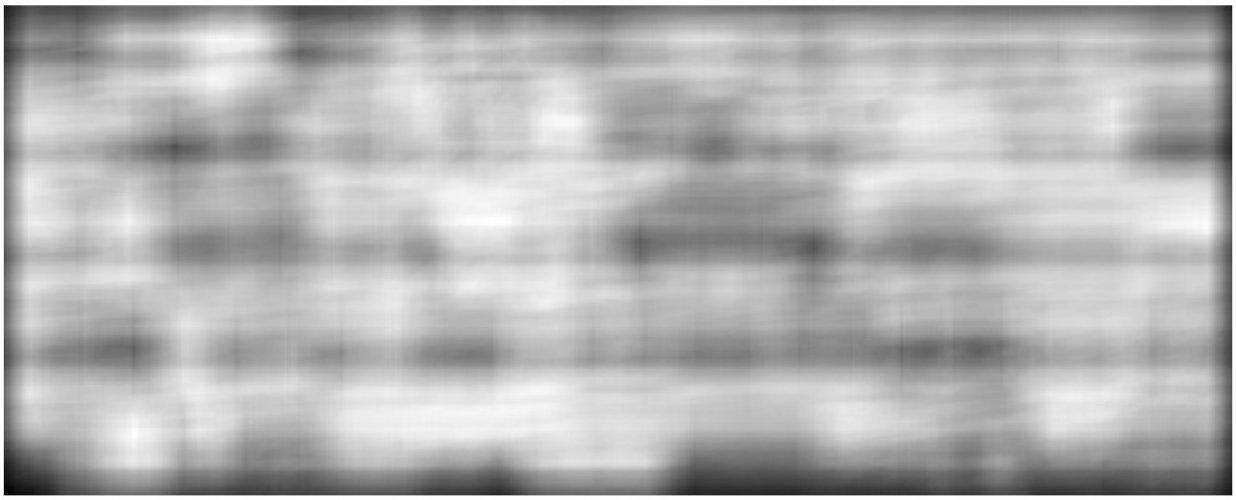
- Hint: you may use the `conv_fast` function you implemented in the previous question.

```
def cross_correlation(f, g):
    """ Cross-correlation of image f and template g.

    Hint: use the conv_fast function defined above.

    Args:
        f: numpy array of shape (Hf, Wf).
        g: numpy array of shape (Hg, Wg).

    Returns:
        out: numpy array of shape (Hf, Wf).
    """

    out = None
    ### YOUR CODE HERE
    Hi, Wi = f.shape
    Hk, Wk = g.shape
    out = np.zeros((Hi, Wi))
    padded_f = zero_pad(f, Hk//2, Wk//2)
    for i in range(Hi):
        for j in range(Wi):
            region = padded_f[i:i+Hk,j:j+Wk]
            out[i, j] = np.sum(g * region)
    ### END YOUR CODE
```

```
    return out


# Load template and image in grayscale
img = io.imread('shelf.jpg')
img_gray = io.imread('shelf.jpg', as_gray=True)
temp = io.imread('template.jpg')
temp_gray = io.imread('template.jpg', as_gray=True)

# Perform cross-correlation between the image and the template
out = cross_correlation(img_gray, temp_gray)

# Find the location with maximum similarity
y,x = (np.unravel_index(out.argmax(), out.shape))

# Display product template
plt.figure(figsize=(25,20))
plt.subplot(3, 1, 1)
plt.imshow(temp)
plt.title('Template')
plt.axis('off')

# Display cross-correlation output
plt.subplot(3, 1, 2)
plt.imshow(out)
plt.title('Cross-correlation (white means more correlated)')
plt.axis('off')

# Display image
plt.subplot(3, 1, 3)
plt.imshow(img)
plt.title('Result (blue marker on the detected location)')
plt.axis('off')

# Draw marker at detected location
plt.plot(x, y, 'bx', ms=40, mew=10)
plt.show()
```

Template

Cross-correlation (white means more correlated)



Result (blue marker on the detected location)

Interpretation

How does the output of cross-correlation filter look? Explain what problems there might be with using a raw template as a filter.

Your Answer: *Write your solution in this markdown cell.*

The cross-correlation filter looks inconclusive/noisey. You might want to use a different high gain function to exaggerate higher correlation values to create separation between signal and noise. The raw filter may be low resolution enough that false positives may show up by chance. The filter also seemed to miss the actual object? Maybe its a size issue? the kernel was too big?

2.2 Zero-mean cross-correlation (6 points)

A solution to this problem is to subtract the mean value of the template so that it has zero mean.

Implement **`zero_mean_cross_correlation`** function and run the code below.

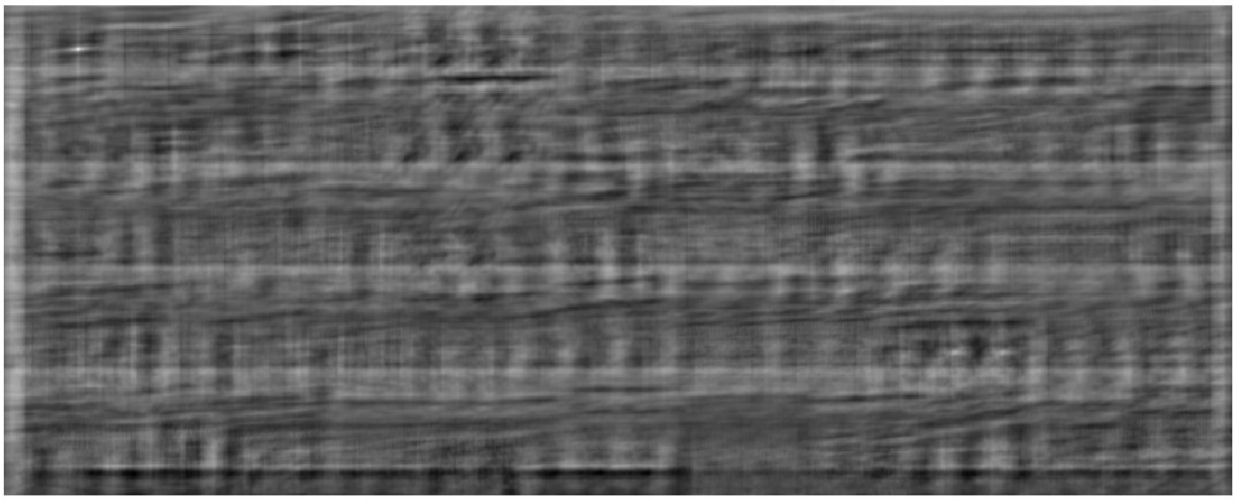
If your implementation is correct, you should see the blue cross centered over the correct cereal box.

```
def zero_mean_cross_correlation(f, g):
    """ Zero-mean cross-correlation of image f and template g.

    Subtract the mean of g from g so that its mean becomes zero.

    Hint: you should look up useful numpy functions online for
calculating the mean.

    Args:
        f: numpy array of shape (Hf, Wf).
        g: numpy array of shape (Hg, Wg).

    Returns:
        out: numpy array of shape (Hf, Wf).
    """

    out = None
    ### YOUR CODE HERE
    Hi, Wi = f.shape
    Hk, Wk = g.shape
    mean = np.mean(g)
    g_0mean = g - np.full((Hk,Wk),mean)
    out = np.zeros((Hi, Wi))
    padded_f = zero_pad(f, Hk//2, Wk//2)
    for i in range(Hi):
        for j in range(Wi):
            region = padded_f[i:i+Hk,j:j+Wk]
            out[i, j] = np.sum(g_0mean * region)
```

```
    ### END YOUR CODE

    return out

# Perform cross-correlation between the image and the template
out = zero_mean_cross_correlation(img_gray, temp_gray)

# Find the location with maximum similarity
y,x = np.unravel_index(out.argmax(), out.shape)

# Display product template
plt.figure(figsize=(30,20))
plt.subplot(3, 1, 1)
plt.imshow(temp)
plt.title('Template')
plt.axis('off')

# Display cross-correlation output
plt.subplot(3, 1, 2)
plt.imshow(out)
plt.title('Cross-correlation (white means more correlated)')
plt.axis('off')

# Display image
plt.subplot(3, 1, 3)
plt.imshow(img)
plt.title('Result (blue marker on the detected location)')
plt.axis('off')

# Draw marker at detected location
plt.plot(x, y, 'bx', ms=40, mew=10)
plt.show()
```

Template

Cross-correlation (white means more correlated)



Result (blue marker on the detected location)

You can also determine whether the product is present with appropriate scaling and thresholding.

```
def check_product_on_shelf(shelf, product):
    out = zero_mean_cross_correlation(shelf, product)

    # Scale output by the size of the template
    out = out / float(product.shape[0]*product.shape[1])

    # Threshold output (this is arbitrary, you would need to tune the
threshold for a real application)
    out = out > 0.025

    if np.sum(out) > 0:
        print('The product is on the shelf')
    else:
        print('The product is not on the shelf')

# Load image of the shelf without the product
img2 = io.imread('shelf_soldout.jpg')
img2_gray = io.imread('shelf_soldout.jpg', as_gray=True)

plt.imshow(img)
plt.axis('off')
plt.show()
check_product_on_shelf(img_gray, temp_gray)

plt.imshow(img2)
plt.axis('off')
plt.show()
check_product_on_shelf(img2_gray, temp_gray)
```


```
The product is on the shelf
```

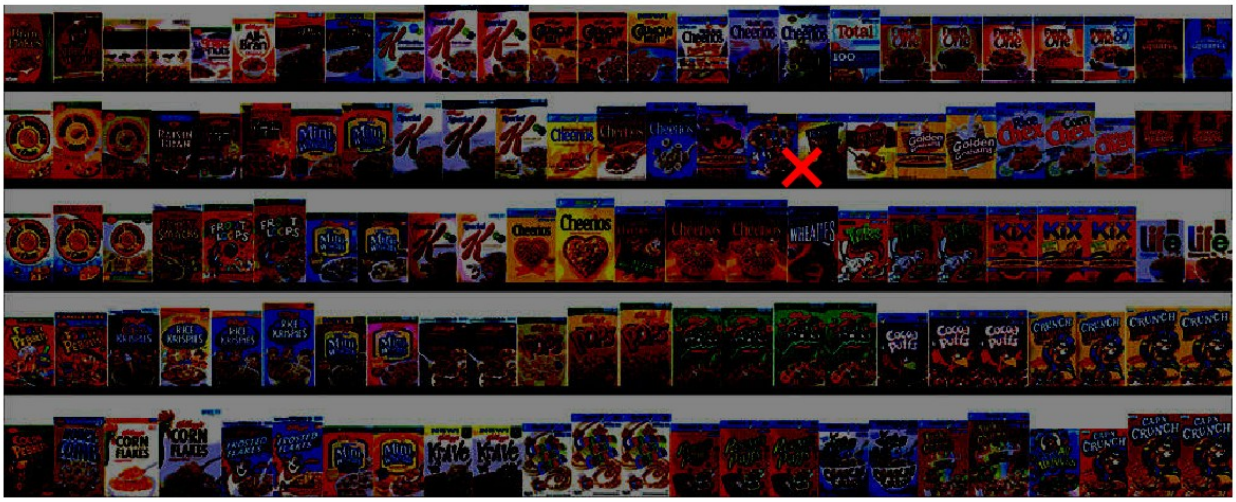
```
The product is not on the shelf
```

2.3 Normalized Cross-correlation (12 points)

One day the light near the shelf goes out and the product tracker starts to malfunction. The `zero_mean_cross_correlation` is not robust to change in lighting condition. The code below demonstrates this.

```
# Load image
img = io.imread('shelf_dark.jpg')
img_gray = io.imread('shelf_dark.jpg', as_gray=True)

# Perform cross-correlation between the image and the template
out = zero_mean_cross_correlation(img_gray, temp_gray)

# Find the location with maximum similarity
y, x = np.unravel_index(out.argmax(), out.shape)

# Display image
plt.imshow(img)
plt.title('Result (red marker on the detected location)')
plt.axis('off')

# Draw marker at detcted location
plt.plot(x, y, 'rx', ms=25, mew=5)
plt.show()
```

Result (red marker on the detected location)



A solution is to normalize the pixels of the image and template at every step before comparing them. This is called **normalized cross-correlation**.

The mathematical definition for normalized cross-correlation of f and template g is:

$$(g \star f)[m,n] = \sum_{i,j} \frac{g[i,j] - \overline{g}}{\sigma_g} \cdot \frac{f[m+i, n+j] - \overline{f_{m,n}}}{\sigma_{f_{m,n}}}$$

where:

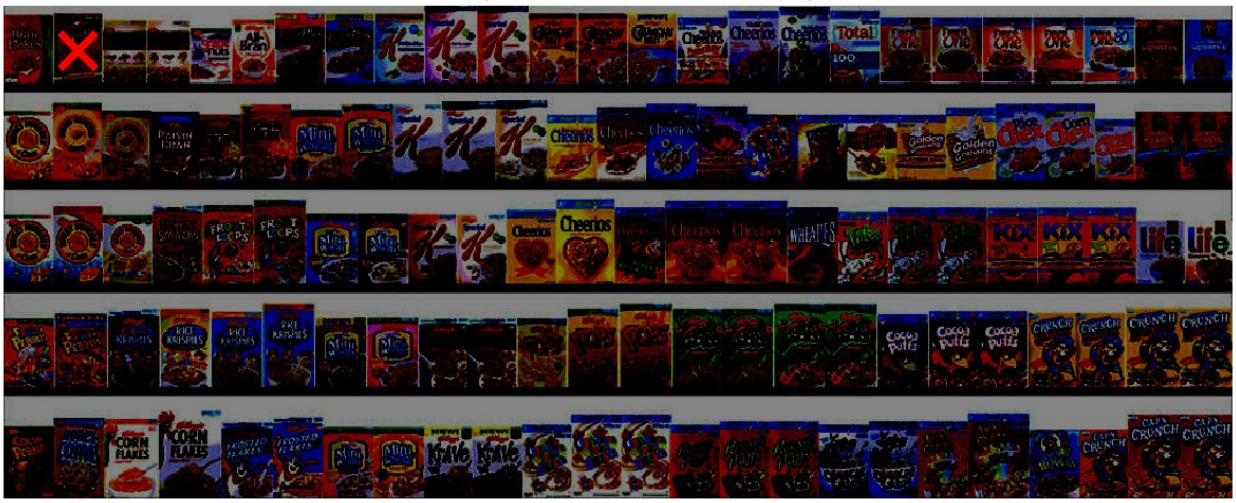
- $f_{m,n}$ is the patch image at position (m, n)
- $\overline{f_{m,n}}$ is the mean of the patch image $f_{m,n}$
- $\sigma_{f_{m,n}}$ is the standard deviation of the patch image $f_{m,n}$
- $\overline{g}$ is the mean of the template g
- σ_g is the standard deviation of the template g

Implement the **`normalized_cross_correlation`** function and run the code below.

```
def normalized_cross_correlation(f, g):
    """ Normalized cross-correlation of image f and template g.

    Normalize the subimage of f and the template g at each step
    before computing the weighted sum of the two.

    Hint: you should look up useful numpy functions online for
calculating
          the mean and standard deviation.

    Args:
        f: numpy array of shape (Hf, Wf).
        g: numpy array of shape (Hg, Wg).

    Returns:
```

```
        out: numpy array of shape (Hf, Wf).
    """

    out = None
    ### YOUR CODE HERE
    Hi, Wi = f.shape
    Hk, Wk = g.shape
    u_g = np.mean(g)
    g_0mean = g - np.full((Hk,Wk),u_g)
    std_g = np.std(g)

    out = np.zeros((Hi, Wi))
    padded_f = zero_pad(f, Hk//2, Wk//2)
    for i in range(Hi):
        for j in range(Wi):
            region = padded_f[i:i+Hk,j:j+Wk]
            u_f = np.mean(region)
            std_f = np.std(region)

            region_0mean = region - np.full((Hk,Wk),u_f)
            out[i, j] = np.sum((g_0mean/std_g) * (region_0mean/std_f))
    ### END YOUR CODE

    return out


# Perform normalized cross-correlation between the image and the
template
out = normalized_cross_correlation(img_gray, temp_gray)

# Find the location with maximum similarity
y, x = np.unravel_index(out.argmax(), out.shape)

# Display image
plt.imshow(img)
plt.title('Result (red marker on the detected location)')
plt.axis('off')

# Draw marker at detcted location
plt.plot(x, y, 'rx', ms=25, mew=5)
plt.show()
```

Result (red marker on the detected location)



Part 3: Canny Edge Detector (85 points)

In this part, you are going to implement a Canny edge detector. The Canny edge detection algorithm can be broken down in to five steps:

1. Smoothing
2. Finding gradients
3. Non-maximum suppression
4. Double thresholding
5. Edge tracking by hysterisis

3.1 Smoothing (10 points)

Implementation (5 points)

We first smooth the input image by convolving it with a Gaussian kernel. The equation for a Gaussian kernel of size $(2k+1) \times (2k+1)$ is given by:

$$h_{ij}=\frac{1}{2\pi\sigma^2}\exp{\Bigl(-\frac{(i-k)^2+(j-k)^2}{2\sigma^2}\Bigr)}, 0\leq i,j < 2k+1$$

Implement **`gaussian_kernel`** and run the code below.

```
def gaussian_kernel(size, sigma):
    """ Implementation of Gaussian Kernel.

    This function follows the gaussian kernel formula,
    and creates a kernel matrix.

    Hints:
    - Use np.pi and np.exp to compute pi and exp.
```

```
    Args:
        size: int of the size of output matrix.
        sigma: float of sigma to calculate kernel.

    Returns:
        kernel: numpy array of shape (size, size).
    """

    kernel = np.zeros((size, size))

    ### YOUR CODE HERE
    k = (size-1)//2

    for i in range(size):
        for j in range(size):
            kernel[i,j] = (1/(2*np.pi*sigma**2)) * np.exp(-((i-k)**2 +
(j-k)**2)/(2*sigma**2))
    ### END YOUR CODE

    return kernel

# Define 3x3 Gaussian kernel with std = 1
kernel = gaussian_kernel(3, 1)
kernel_test = np.array(
    [[ 0.05854983, 0.09653235, 0.05854983],
     [ 0.09653235, 0.15915494, 0.09653235],
     [ 0.05854983, 0.09653235, 0.05854983]]
)

# Test Gaussian kernel
if not np.allclose(kernel, kernel_test):
    print('Incorrect values! Please check your implementation.')
```

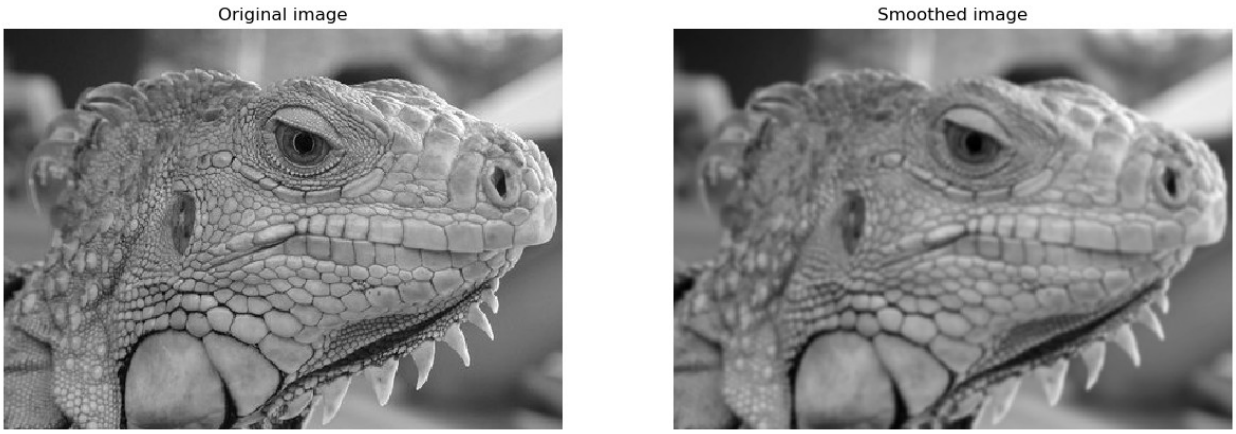
Implement **`conv`** and run the code below. This time, ensure that you're using **edge** padding (as opposed to zero-padding, as done in `conv_fast`).

Hint: Check out `np.pad`, and the various `mode`s that it takes.

```
def conv(image, kernel):
    """ An implementation of convolution filter.

    This function uses element-wise multiplication and np.sum()
    to efficiently compute weighted sum of neighborhood at each
    pixel.

    Args:
        image: numpy array of shape (Hi, Wi).
        kernel: numpy array of shape (Hk, Wk).
```

```
    Returns:
        out: numpy array of shape (Hi, Wi).
    """
    Hi, Wi = image.shape
    Hk, Wk = kernel.shape
    out = np.zeros((Hi, Wi))

    # For this assignment, we will use edge values to pad the images.
    # Zero padding will make derivatives at the image boundary very
big,
    # whereas we want to ignore the edges at the boundary.
    pad_width0 = Hk // 2
    pad_width1 = Wk // 2
    pad_width = ((pad_width0,pad_width0),(pad_width1,pad_width1))
    padded = np.pad(image, pad_width, mode='edge')

    ### YOUR CODE HERE
    flipped_kernel = np.flip(kernel)
    ### YOUR CODE HERE
    for i in range(Hi):
        for j in range(Wi):
            region = padded[i:i+Hk,j:j+Wk]
            out[i, j] = np.sum(flipped_kernel * region)
    ### END YOUR CODE

    return out

# Test with different kernel_size and sigma
kernel_size = 5
sigma = 1.4

# Load image
img = io.imread('iguana.png', as_gray=True)

# Define 5x5 Gaussian kernel with std = sigma
kernel = gaussian_kernel(kernel_size, sigma)

# Convolve image with kernel to achieve smoothed effect
smoothed = conv(img, kernel)

plt.subplot(1,2,1)
plt.imshow(img)
plt.title('Original image')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(smoothed)
plt.title('Smoothed image')
plt.axis('off')
```

```
plt.show()
```



Question (5 points)

What is the effect of changing kernel_size and sigma?

Your Answer: Write your solution in this markdown cell.

Increasing sigma increases blur effect. This fattens the tail of the distribution and makes the weights in the pixels far from the center higher.

Increasing kernel size also increases blur effect but at a diminishing rate as you increase kernel size. This makes sense as the weights of pixels in the kernel far from the center have reduced weights. Therefore holding u,sigma constant, the larger the kernel gets only increases the blur effect so much.

3.2 Finding gradients (15 points)

The gradient of a 2D scalar function $I: R^2 \rightarrow R$ in Cartesian coordinate is defined by:

$$\nabla I(x, y)=\left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right],$$

where

$$ \frac{\partial{I(x,y)}}{\partial{x}}=\lim_{\Delta{x}\to{0}}\frac{I(x+\Delta{x},y)-I(x,y)}{\Delta{x}} \\ \frac{\partial{I(x,y)}}{\partial{y}}=\lim_{\Delta{y}\to{0}}\frac{I(x,y+\Delta{y})-I(x,y)}{\Delta{y}}. $$

In case of images, we can approximate the partial derivatives by taking differences at one pixel intervals:

$$ \frac{\partial{I(x,y)}}{\partial{x}}\approx{\frac{I(x+1,y)-I(x-1,y)}{2}} \\ \frac{\partial{I(x,y)}}{\partial{y}}\approx{\frac{I(x,y+1)-I(x,y-1)}{2}} $$

Note that the partial derivatives can be computed by convolving the image I with some appropriate kernels D_x and D_y:

$$ \frac{\partial{I}}{\partial{x}}\approx{I*D_x}=G_x \\ \frac{\partial{I}}{\partial{y}}\approx{I*D_y}=G_y $$

Implementation (5 points)

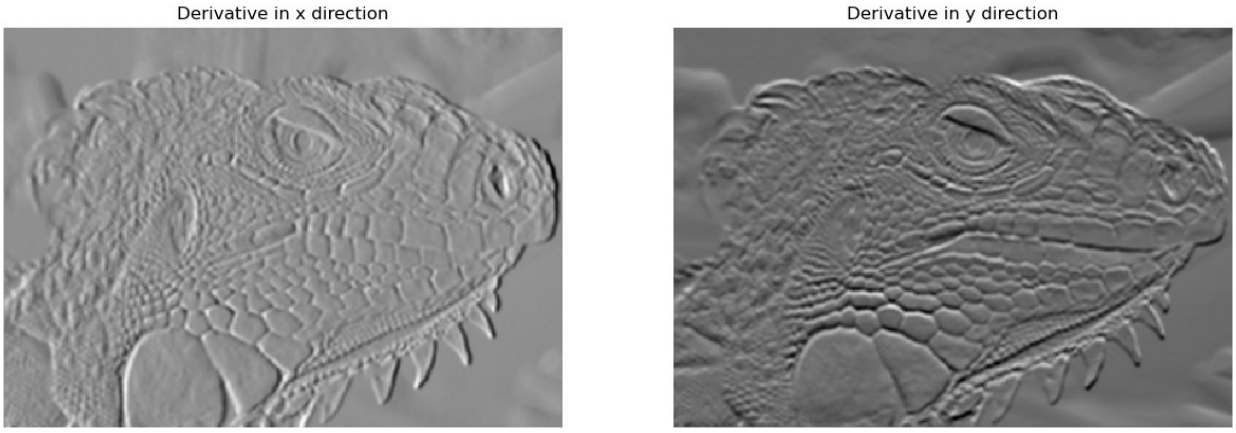
Find the kernels D_x and D_y and implement **`partial_x`** and **`partial_y`** using `conv` defined below.

-Hint: Remeber that convolution flips the kernel.

```
def partial_x(img):
    """ Computes partial x-derivative of input img.

    Hints:
        - You may use the conv function in defined in this file.

    Args:
        img: numpy array of shape (H, W).
    Returns:
        out: x-derivative image.
    """

    out = None

    ### YOUR CODE HERE
    dx_kernel = np.array([[0,0,0],
                [0.5,0,-0.5],
                [0,0,0]]
    )
    out = conv(img, dx_kernel)
    ### END YOUR CODE

    return out

def partial_y(img):
    """ Computes partial y-derivative of input img.

    Hints:
        - You may use the conv function in defined in this file.

    Args:
        img: numpy array of shape (H, W).
    Returns:
        out: y-derivative image.
    """

    out = None
```

```
    ### YOUR CODE HERE
    dy_kernel = np.array([[0,0.5,0],
                  [0,0,0],
                  [0,-0.5,0]]
    )
    out = conv(img, dy_kernel)
    ### END YOUR CODE

    return out


# Test input
I = np.array(
    [[0, 0, 0],
     [0, 1, 0],
     [0, 0, 0]]
)

# Expected outputs
I_x_test = np.array(
    [[ 0, 0, 0],
     [ 0.5, 0, -0.5],
     [ 0, 0, 0]]
)

I_y_test = np.array(
    [[ 0, 0.5, 0],
     [ 0, 0, 0],
     [ 0, -0.5, 0]]
)

# Compute partial derivatives
I_x = partial_x(I)
I_y = partial_y(I)

# Test correctness of partial_x and partial_y
if not np.all(I_x == I_x_test):
    print('partial_x incorrect')

if not np.all(I_y == I_y_test):
    print('partial_y incorrect')

# Compute partial derivatives of smoothed image
Gx = partial_x(smoothed)
Gy = partial_y(smoothed)

plt.subplot(1,2,1)
plt.imshow(Gx)
plt.title('Derivative in x direction')
```

```
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(Gy)
plt.title('Derivative in y direction')
plt.axis('off')

plt.show()
```



Question (5 points)

What is the reason for performing smoothing prior to computing the gradients?

Your Answer: Write your solution in this markdown cell.

Images are naturally noisy which look like individual cells with very different values comparing to neighbors. As a result, this noise is amplified when you take a derivative (by definition noise is a discontinuity that disrupts smoothness). Smoothing away the noise will remove this in the gradient image and leave the signal that we care about. Essentially acting as a low pass filter.

Implementation (5 points)

Now, we can compute the magnitude and direction of gradient with the two partial derivatives:

$$ G = \sqrt{G_{x}^{2}+G_{y}^{2}} \\ \Theta = arctan\bigl(\frac{G_{y}}{G_{x}}\bigr) $$

Implement **`gradient`** below which takes in an image and outputs G and Θ.

```
def gradient(img):
    """ Returns gradient magnitude and direction of input img.

    Args:
        img: Grayscale image. Numpy array of shape (H, W).

    Returns:
        G: Magnitude of gradient at each pixel in img.
```

```
            Numpy array of shape (H, W).
        theta: Direction(in degrees, 0 <= theta < 360) of gradient
            at each pixel in img. Numpy array of shape (H, W).

    Hints:
        - Use np.sqrt and np.arctan2 to calculate square root and
arctan
    """
    G = np.zeros(img.shape)
    theta = np.zeros(img.shape)

    ### YOUR CODE HERE
    g_dx = partial_x(img)
    g_dy = partial_y(img)

    G = np.sqrt(g_dx**2 + g_dy**2)
    theta = np.degrees(np.arctan2(g_dy, g_dx)) % 360 #modulo to ensure
between [0,360) degrees, also use arctan2 for better divide-by-zero
behavior

    ### END YOUR CODE

    return G, theta


G, theta = gradient(smoothed)
if not np.all(G >= 0):
    print('Magnitude of gradients should be non-negative.')

if not np.all((theta >= 0) * (theta < 360)):
    print('Direction of gradients should be in range 0 <= theta <
360')

plt.imshow(G)
plt.title('Gradient magnitude')
plt.axis('off')
plt.show()
```

Gradient magnitude

3.3 Non-maximum suppression (15 points)

You should be able to see that the edges extracted from the gradient of the smoothed image are quite thick and blurry. The purpose of this step is to convert the "blurred" edges into "sharp" edges. Basically, this is done by preserving all local maxima in the gradient image and discarding everything else. The algorithm is for each pixel (x,y) in the gradient image:

1. Round the gradient direction $\Theta[y, x)$ to the nearest 45 degrees, corresponding to the use of an 8-connected neighbourhood.

2. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions. For example, if the gradient direction is south (theta=90), compare with the pixels to the north and south.

3. If the edge strength of the current pixel is the largest; preserve the value of the edge strength. If not, suppress (i.e. remove) the value.

Implement **`non_maximum_suppression`** below.

We provide the correct output and the difference between it and your result for debugging purposes. If you see white spots in the Difference image, you should check your implementation.

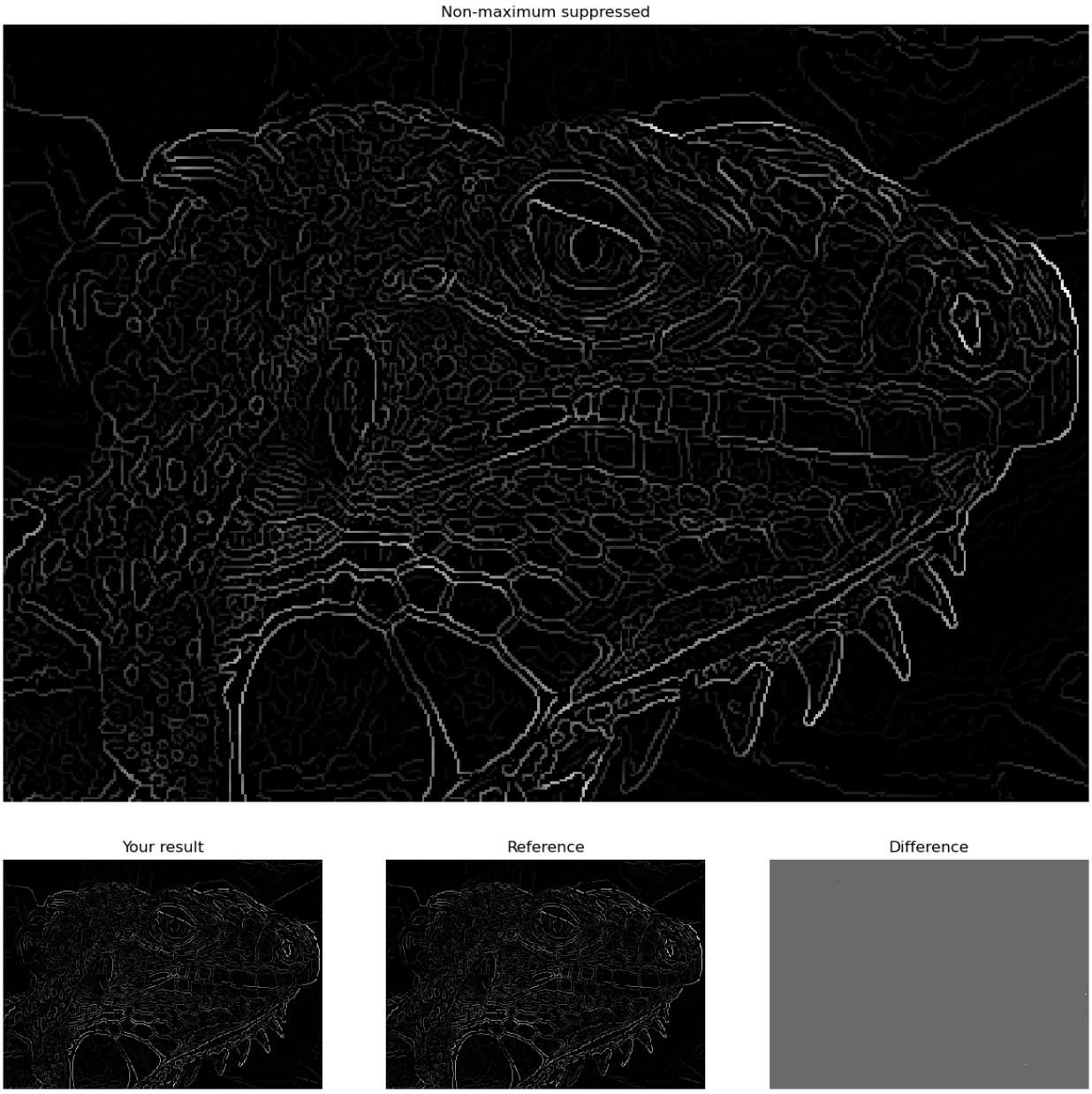
```
def non_maximum_suppression(G, theta):
    """ Performs non-maximum suppression.

    This function performs non-maximum suppression along the direction
    of gradient (theta) on the gradient magnitude image (G).

    Args:
        G: gradient magnitude image with shape of (H, W).
        theta: direction of gradients with shape of (H, W).

    Returns:
        out: non-maxima suppressed image.
    """
    H, W = G.shape
    out = np.zeros((H, W))

    # Round the gradient direction to the nearest 45 degrees
    theta = np.floor((theta + 22.5) / 45) * 45

    ### BEGIN YOUR CODE

    theta = np.mod(theta, 360)

    # Pad arrays with zeros
    G_pad = np.pad(G, pad_width=1, mode='constant', constant_values=0)
    theta_pad = np.pad(theta, pad_width=1, mode='constant',
constant_values=0)

    for i in range(H):
        for j in range(W):
            angle = theta_pad[i+1,j+1]

            if angle == 0 or angle == 180:
                if G_pad[i+1,j+1] >= G_pad[i+1,j+2] and G_pad[i+1,j+1]
>= G_pad[i+1,j]:
                    out[i,j] = G_pad[i+1,j+1]
            elif angle == 45 or angle == 225:
                if G_pad[i+1,j+1] >= G_pad[i,j] and G_pad[i+1,j+1] >=
G_pad[i+2,j+2]:
                    out[i,j] = G_pad[i+1,j+1]
            elif angle == 90 or angle == 270:
                if G_pad[i+1,j+1] >= G_pad[i,j+1] and G_pad[i+1,j+1]
>= G_pad[i+2,j+1]:
                    out[i,j] = G_pad[i+1,j+1]
            elif angle == 135 or angle == 315:
                if G_pad[i+1,j+1] >= G_pad[i,j+2] and G_pad[i+1,j+1]
>= G_pad[i+2,j]:
                    out[i,j] = G_pad[i+1,j+1]
    ### END YOUR CODE
```

```
    return out


# Test input
g = np.array(
    [[0.4, 0.5, 0.6],
     [0.3, 0.5, 0.7],
     [0.4, 0.5, 0.6]]
)

# Print out non-maximum suppressed output
# varying theta
for angle in range(0, 180, 45):
    print('Thetas:', angle)
    t = np.ones((3, 3)) * angle # Initialize theta
    print(non_maximum_suppression(g, t))
```

```
Thetas: 0
[[0.  0.  0.6]
 [0.  0.  0.7]
 [0.  0.  0.6]]
Thetas: 45
[[0.  0.  0.6]
 [0.  0.  0.7]
 [0.4 0.5 0.6]]
Thetas: 90
[[0.4 0.5 0. ]
 [0.  0.5 0.7]
 [0.4 0.5 0. ]]
Thetas: 135
[[0.4 0.5 0.6]
 [0.  0.  0.7]
 [0.  0.  0.6]]
```

```
nms = non_maximum_suppression(G, theta)
plt.imshow(nms)
plt.title('Non-maximum suppressed')
plt.axis('off')
plt.show()

plt.subplot(1, 3, 1)
plt.imshow(nms)
plt.axis('off')
plt.title('Your result')

plt.subplot(1, 3, 2)
reference = np.load('references/iguana_non_max_suppressed.npy')
plt.imshow(reference)
plt.axis('off')
```

```
plt.title('Reference')

plt.subplot(1, 3, 3)
plt.imshow(nms - reference)
plt.title('Difference')
plt.axis('off')
plt.show()

np.amax(nms-reference)
```




```
0.004954806714090541
```

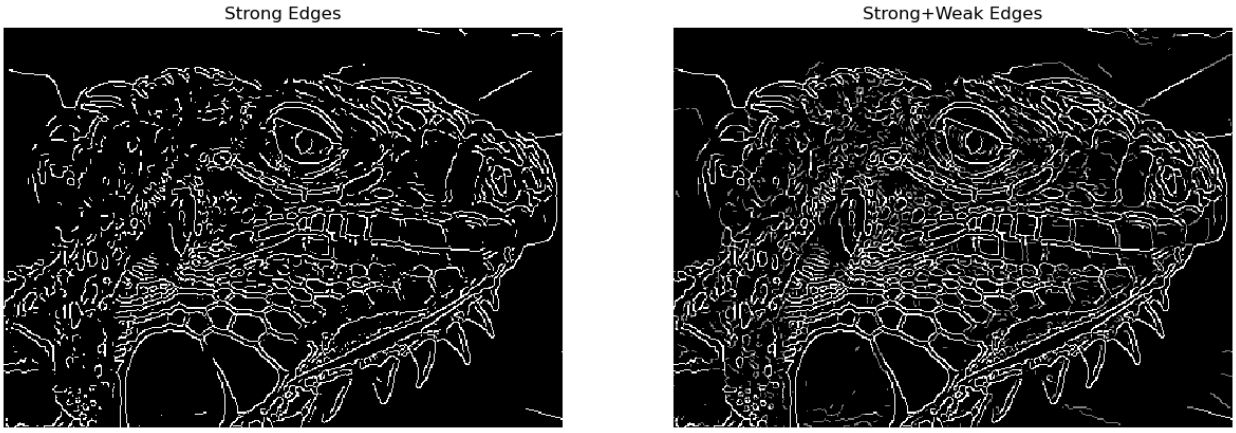
3.4 Double Thresholding (20 points)

The edge-pixels remaining after the non-maximum suppression step are (still) marked with their strength pixel-by-pixel. Many of these will probably be true edges in the image, but some may be caused by noise or color variations, for instance, due to rough surfaces. The simplest way to discern between these would be to use a threshold, so that only edges stronger that a certain value would be preserved. The Canny edge detection algorithm uses double thresholding. Edge pixels stronger than the high threshold are marked as strong; edge pixels weaker than the low threshold are suppressed and edge pixels between the two thresholds are marked as weak.

Implement **`double_thresholding`** below.

```
def double_thresholding(img, high, low):
    """
    Args:
        img: numpy array of shape (H, W) representing NMS edge
response.
        high: high threshold(float) for strong edges.
        low: low threshold(float) for weak edges.

    Returns:
        strong_edges: Boolean array representing strong edges.
            Strong edeges are the pixels with the values greater than
            the higher threshold.
        weak_edges: Boolean array representing weak edges.
            Weak edges are the pixels with the values smaller or equal
to the
            higher threshold and greater than the lower threshold.
    """

    strong_edges = np.zeros(img.shape, dtype=bool)
    weak_edges = np.zeros(img.shape, dtype=bool)

    ### YOUR CODE HERE
    H, W = img.shape

    for i in range(H):
        for j in range(W):
            if img[i,j] >= high:
                strong_edges[i,j] = True
            elif img[i,j] >= low and img[i,j] < high:
                weak_edges[i,j] = True

    ### END YOUR CODE

    return strong_edges, weak_edges


low_threshold = 0.02
high_threshold = 0.03
```

```
strong_edges, weak_edges = double_thresholding(nms, high_threshold,
low_threshold)
assert(np.sum(strong_edges & weak_edges) == 0)

edges=strong_edges * 1.0 + weak_edges * 0.5

plt.subplot(1,2,1)
plt.imshow(strong_edges)
plt.title('Strong Edges')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(edges)
plt.title('Strong+Weak Edges')
plt.axis('off')

plt.show()
```



3.5 Edge tracking (15 points)

Strong edges are interpreted as “certain edges”, and can immediately be included in the final edge image. Consider its neighbors iteratively then declare it an 'edge pixel' if it is connected to a 'strong edge pixel' directly or via pixels between Low and High. The logic is of course that noise and other small variations are unlikely to result in a strong edge (with proper adjustment of the threshold levels). Thus strong edges will (almost) only be due to true edges in the original image. The weak edges can either be due to true edges or noise/color variations. The latter type will probably be distributed independently of edges on the entire image, and thus only a small amount will be located adjacent to strong edges. Weak edges due to true edges are much more likely to be connected directly to strong edges.

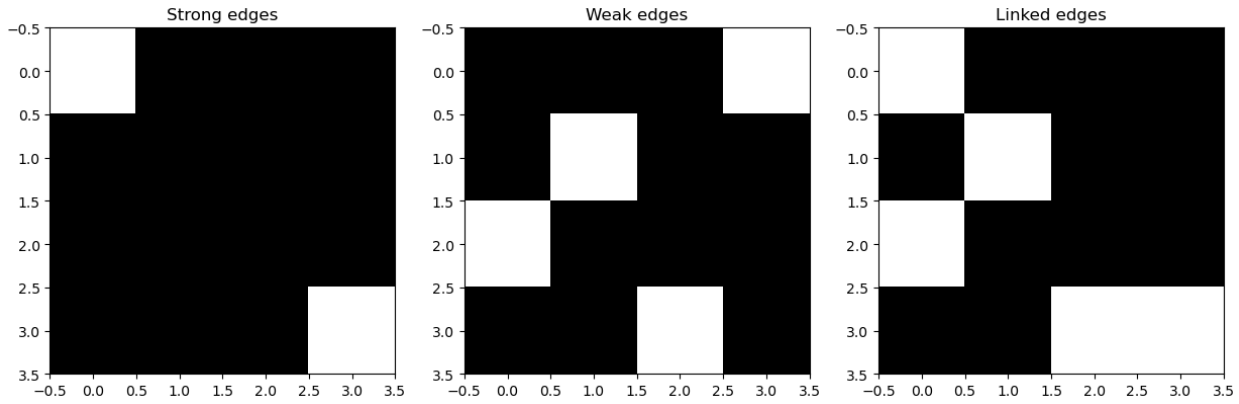
Implement **`link_edges`** below.

We provide the correct output and the difference between it and your result for debugging purposes. If you see white spots in the Difference image, you should check your implementation.

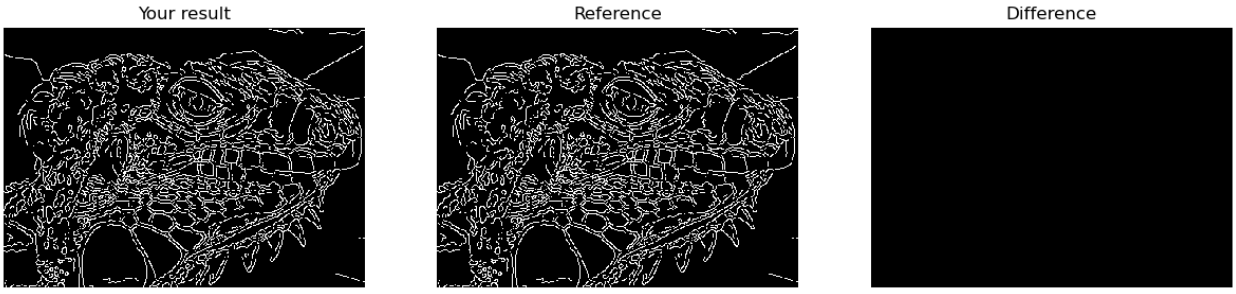
```python
# a helper function for you to use:
def get_neighbors(y, x, H, W):
    """ Return indices of valid neighbors of (y, x).

    Return indices of all the valid neighbors of (y, x) in an array of
    shape (H, W). An index (i, j) of a valid neighbor should satisfy
    the following:
        1. i >= 0 and i < H
        2. j >= 0 and j < W
        3. (i, j) != (y, x)

    Args:
        y, x: location of the pixel.
        H, W: size of the image.
    Returns:
        neighbors: list of indices of neighboring pixels [(i, j)].
    """
    neighbors = []

    for i in (y-1, y, y+1):
        for j in (x-1, x, x+1):
            if i >= 0 and i < H and j >= 0 and j < W:
                if (i == y and j == x):
                    continue
                neighbors.append((i, j))

    return neighbors


def link_edges(strong_edges, weak_edges):
    """ Find weak edges connected to strong edges and link them.

    Iterate over each pixel in strong_edges and perform breadth first
    search across the connected pixels in weak_edges to link them.
    Here we consider a pixel (a, b) is connected to a pixel (c, d)
    if (a, b) is one of the eight neighboring pixels of (c, d).

    Args:
        strong_edges: binary image of shape (H, W).
        weak_edges: binary image of shape (H, W).

    Returns:
        edges: numpy boolean array of shape(H, W).
    """

    H, W = strong_edges.shape
    indices = np.stack(np.nonzero(strong_edges)).T
    edges = np.zeros((H, W), dtype=bool)

    # Make new instances of arguments to leave the original
```

```
    # references intact
    weak_edges = np.copy(weak_edges)
    edges = np.copy(strong_edges)

    ### YOUR CODE HERE
    from collections import deque

    for y, x in indices:
        queue = deque([(y, x)])
        while queue:
            i, j = queue.popleft()
            neighbors = get_neighbors(i, j, H, W)
            for ni, nj in neighbors:
                if weak_edges[ni, nj] and not edges[ni, nj]:
                    edges[ni, nj] = True
                    queue.append((ni, nj))
    ### END YOUR CODE

    return edges


test_strong = np.array(
    [[1, 0, 0, 0],
     [0, 0, 0, 0],
     [0, 0, 0, 0],
     [0, 0, 0, 1]],
    dtype=bool
)

test_weak = np.array(
    [[0, 0, 0, 1],
     [0, 1, 0, 0],
     [1, 0, 0, 0],
     [0, 0, 1, 0]],
    dtype=bool
)

test_linked = link_edges(test_strong, test_weak)

plt.subplot(1, 3, 1)
plt.imshow(test_strong)
plt.title('Strong edges')

plt.subplot(1, 3, 2)
plt.imshow(test_weak)
plt.title('Weak edges')

plt.subplot(1, 3, 3)
plt.imshow(test_linked)
```

```
plt.title('Linked edges')
plt.show()
```




```
edges = link_edges(strong_edges, weak_edges)

plt.imshow(edges)
plt.axis('off')
plt.show()

plt.subplot(1, 3, 1)
plt.imshow(edges)
plt.axis('off')
plt.title('Your result')

plt.subplot(1, 3, 2)
reference = np.load('references/iguana_edge_tracking.npy')
plt.imshow(reference)
plt.axis('off')
plt.title('Reference')

plt.subplot(1, 3, 3)
plt.imshow(edges ^ reference)
plt.title('Difference')
plt.axis('off')
plt.show()
```



3.6 Canny edge detector

Implement **`canny`** below using the functions you have implemented so far. Test edge detector with different parameters.

Here is an example of the output:

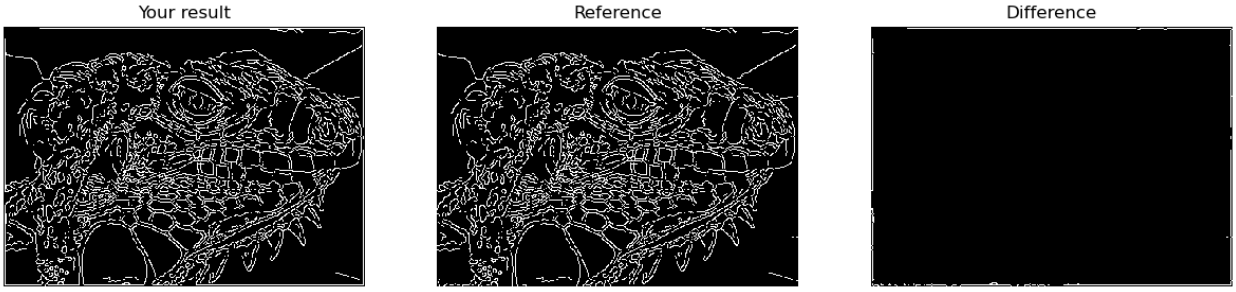
iguana_edges.png

We provide the correct output and the difference between it and your result for debugging purposes. If you see white spots in the Difference image, you should check your implementation.

```
def canny(img, kernel_size=5, sigma=1.4, high=20, low=15):
    """ Implement canny edge detector by calling functions above.
```

```
    Args:
        img: binary image of shape (H, W).
        kernel_size: int of size for kernel matrix.
        sigma: float for calculating kernel.
        high: high threshold for strong edges.
        low: low threashold for weak edges.
    Returns:
        edge: numpy array of shape(H, W).
    """
    ### YOUR CODE HERE
    kernel = gaussian_kernel(kernel_size, sigma)

    smoothed = conv_fast(img, kernel)

    G, theta = gradient(smoothed)

    nms = non_maximum_suppression(G, theta)

    strong_edges, weak_edges = double_thresholding(nms, high, low)

    edge = link_edges(strong_edges, weak_edges)

    ### END YOUR CODE

    return edge


# Load image
img = io.imread('iguana.png', as_gray=True)

# Run Canny edge detector
edges = canny(img, kernel_size=5, sigma=1.4, high=0.03, low=0.02)
print (edges.shape)

plt.subplot(1, 3, 1)
plt.imshow(edges)
plt.axis('off')
plt.title('Your result')

plt.subplot(1, 3, 2)
reference = np.load('references/iguana_canny.npy')
plt.imshow(reference)
plt.axis('off')
plt.title('Reference')

plt.subplot(1, 3, 3)
plt.imshow(edges ^ reference)
plt.title('Difference')
plt.axis('off')
plt.show()
```

```
(310, 433)
```



3.7 Question (10 points)

1.7a.png

(a) Suppose that the Canny edge detector successfully detects an edge in an image. The edge (see the figure above) is then rotated by θ, where the relationship between a point on the original edge (x, y) and a point on the rotated edge (x', y') is defined as

$$ x'=x\cos{\theta}\\ y'=x\sin{\theta} $$

Will the rotated edge be detected using the same Canny edge detector? Provide either a mathematical proof or a counter example.

-Hint 1: The detection of an edge by the Canny edge detector depends only on the magnitude of its derivative. The derivative at point (x, y) is determined by its components along the x and y directions. Think about how these magnitudes have changed because of the rotation. -Hint 2: You can assume that (x,y) lies on the x-axis, i.e., y = 0. -Hint 3: You can also assume that G_x(x, y) = 0. In other words, the gradient which is perpendicular to the direction of the unrotated edge at (x, y) only has a vertical component and thus only consists of G_y(x, y).

Your Answer: Write your solution in this markdown cell.

Yes it should not matter if its rotated or not. The Canny edge detector identify edges based on magnitude and should identify it either way because rotation does not affect the magnitude of the gradient.

If original gradient is:

$$\nabla I = \begin{pmatrix} 0 \\ G_y \end{pmatrix}$$

Then rotated becomes:

$$\begin{pmatrix} 0 \\ G_y \end{pmatrix} \overset{\text{rotate by } \theta}{\rightarrow} \begin{pmatrix} G_y \sin\theta \\ G_y \cos\theta \end{pmatrix}$$

Check the magnitude to confirm its identical:

$$\sqrt{0^2 + G_y^2} = \left| G_y \right)$$

$$\sqrt{\left(G_y \sin\theta\right)^2 + \left(G_y \cos\theta\right)^2} = \left|G_y\right)$$

Magnitude is identical. Canny edge detector should detect it all the same.QED

(b) After running the Canny edge detector on an image, you notice that long edges are broken into short segments separated by gaps. In addition, some spurious edges appear. For each of the two thresholds (low and high) used in hysteresis thresholding, explain how you would adjust the threshold (up or down) to address both problems. Assume that a setting exists for the two thresholds that produces the desired result. Briefly explain your answer.

Your Answer: Write your solution in this markdown cell.

For long edges broken into short segments with gaps, I would lower the low threshold to increase the likelihood of weak edges that can be used to connect the strong edges already present.

For spurious edges, I can increase the high threshold to reduce the amount of random noise counting as a strong edge point.