ICC/Programming — Week 10 Exercises

27 April 2026

Goal. In these exercises you will learn how convolution works and how to implement it using NumPy. Convolution is the mathematical foundation of image filtering: blurring and smoothing are both special cases. You will build everything from scratch, then discover how NumPy does it for you.

These exercises are self-contained. You only need NumPy and Pillow.

Setup — create a fresh virtual environment. A virtual environment is an isolated Python installation that keeps your project's libraries separate from the rest of your system. This avoids version conflicts and means you can always reproduce your exact setup on another machine.

Open Visual Studio Code, open a terminal, navigate to your project folder, and run:

```
python3 -m venv .venv
```

This creates a hidden folder called `.venv` that holds a private copy of Python and any libraries you install. Nothing is installed yet.

Activate the virtual environment. You must activate it every time you open a new terminal. The command differs by operating system:

- **macOS / Linux:** `source .venv/bin/activate`
- **Windows (PowerShell):** `.venv\Scripts\Activate.ps1`
- **Windows (Command Prompt):** `.venv\Scripts\activate.bat`

Once active, your terminal prompt will show `(.venv)` at the start. **If you do not see this prefix, the environment is not active and `import numpy` will fail.**

Install the required libraries. Do this once, immediately after the first activation:

```
python3 -m pip install numpy Pillow
```

You do not need to reinstall them next time — activating the environment is enough.

Create your working file. Create a new file `convolution.py` and put the following at the top. Every code snippet in these exercises assumes these lines are present:

```python
import numpy as np
from PIL import Image

def save_image(arr, filename):
    """Save a numpy array as an image file."""
    Image.fromarray(arr.astype("uint8")).save(filename)
```

Run your file at any point by typing the following command in the terminal:

```
python3 convolution.py
```

Exercise 1. What Is a Kernel?

A **kernel** (also called a filter or mask) is a small 2D array of weights. During convolution, the kernel slides over an image and, at each position, computes a weighted sum of the pixels it covers. The result replaces the centre pixel.

(a) Create the 3×3 **box-blur** kernel. All weights are equal and sum to 1, so bright pixels stay bright on average:

```python
box = np.ones((3, 3)) / 9
print(box)
print("Sum of weights:", box.sum())
```

Why do the weights sum to 1? What would happen to image brightness if they summed to 2?

(b) Create the **identity** kernel — it leaves the image unchanged because it simply reads the centre pixel with weight 1:

```
identity = np.array([[0, 0, 0],
                     [0, 1, 0],
                     [0, 0, 0]], dtype=np.float64)
print(identity)
```

If you apply this kernel to any image, what should the output look like?

(c) Create a 5×5 box kernel with all weights equal to $1/25$. Print it and verify the weights sum to 1:

```
box5 = np.ones((5, 5)) / 25
print(box5)
print("Sum:", box5.sum())
```

(d) **Question.** A $k \times k$ kernel examines a $k \times k$ neighbourhood of each pixel. How many pixels does a 3×3 kernel examine? If a 5×5 blur kernel is applied to a 100×100 image, how many multiplications are performed in total?

Exercise 2. Manual Convolution on a Tiny Array

Before writing a general function, work through the mechanics by hand.

(a) Consider this 5×5 signal and the 3×3 box kernel. The convolution output at position **`[2,2]`** (the centre) is the weighted sum of the 3×3 neighbourhood centred there:

```
signal = np.array([[  0,   0,   0,   0,   0],
                   [  0, 100, 100, 100,   0],
                   [  0, 100, 200, 100,   0],
                   [  0, 100, 100, 100,   0],
                   [  0,   0,   0,   0,   0]], dtype=np.float64)

kernel = np.ones((3, 3)) / 9

region = signal[1:4, 1:4]        # 3x3 neighbourhood of the centre
result = np.sum(region * kernel)  # element-wise multiply, then sum
print("Output at [2,2]:", result)
```

Compute the expected value by hand first, then run the code to check.

(b) Compute the output value at position **`[1,1]`** the same way. Which nine pixels does the neighbourhood cover?

```
region = signal[0:3, 0:3]
result = np.sum(region * kernel)
print("Output at [1,1]:", result)
```

(c) When the kernel is centred on a border pixel, part of it falls outside the image. Two common strategies exist to handle this:

- **Zero-padding:** treat every out-of-bounds position as 0. This is what **`np.pad(..., constant_values=0)`** does.
- **Clamp (replicate):** repeat the value of the nearest border pixel for any position outside the image.

We use zero-padding in this session. You will see the effect at the image edges in Exercise 4.

Exercise 3. Implementing a General Convolution Function

Now write the function that automates sliding the kernel over the entire image.

(a) Study the skeleton below, then fill in the one missing line marked **`...`**:

```python
def convolve2d(image, kernel):
    """
    Convolve a 2D grayscale image with a 2D kernel.
    Edges are handled by zero-padding.
    Returns a float64 array of the same shape as image.
    """
    kh, kw = kernel.shape
    pad_h = kh // 2    # e.g. 1 for a 3x3 kernel
    pad_w = kw // 2

    # Zero-pad so the kernel can be centred on every pixel,
    # including those on the border.
    padded = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)),
                    mode='constant', constant_values=0)

    h, w = image.shape
    output = np.zeros((h, w), dtype=np.float64)

    for row in range(h):
        for col in range(w):
            region = padded[row : row + kh, col : col + kw]
            output[row, col] = ...   # fill this in

    return output
```

Hint: use **`np.sum`** and the * operator, exactly as in Exercise 2.

(b) Test with the identity kernel — the output must match the input:

```python
test = np.array([[10, 20, 30],
                 [40, 50, 60],
                 [70, 80, 90]], dtype=np.float64)

identity = np.array([[0, 0, 0],
                     [0, 1, 0],
                     [0, 0, 0]], dtype=np.float64)

result = convolve2d(test, identity)
print(result)
print("Matches original:", np.allclose(result, test))
```

If **`np.allclose`** prints **`False`**, re-read your loop body.

(c) Apply the 3×3 box kernel to **`signal`** from Exercise 2 and print the result. Compare the centre value with your hand calculation from 2(a):

```python
box = np.ones((3, 3)) / 9
blurred = convolve2d(signal, box)
print(np.round(blurred, 1))
```

Do the edge values (where zero-padding was used) make sense?

Exercise 4. Blur Kernels

A blur kernel replaces each pixel with a weighted average of its neighbours. The wider the kernel, the stronger the blur.

(a) Load the provided image and apply the 3×3 box blur. Make sure **`original.jpg`** is in the same folder as your script:

```python
img = np.array(Image.open("original.jpg").convert("L"), dtype=np.float64)
print("Image shape:", img.shape)

box3 = np.ones((3, 3)) / 9
blurred3 = convolve2d(img, box3)
save_image(blurred3, "blur_3x3.png")
```

(b) Apply a 5×5 and a 9×9 box kernel and save the results:

```python
box5 = np.ones((5, 5)) / 25
blurred5 = convolve2d(img, box5)
save_image(blurred5, "blur_5x5.png")
```

```
box9 = np.ones((9, 9)) / 81
blurred9 = convolve2d(img, box9)
save_image(blurred9, "blur_9x9.png")
```

Open **`original.jpg`**, **`blur_3x3.png`**, **`blur_5x5.png`**, and **`blur_9x9.png`** side by side. What do you notice as the kernel grows larger?

(c) **Question.** Your **`convolve2d`** function uses two nested Python loops. For a 1000×1000 image with a 3×3 kernel, how many iterations does the inner loop perform in total?

Exercise 5. Convolution with NumPy

You have built **`convolve2d`** from scratch to understand how it works. NumPy provides **`np.convolve`** for 1D signals — a fast, compiled implementation of exactly the same operation. In this exercise you will explore it and then apply it row by row to an image.

(a) **np.convolve on a 1D signal.** Run the following and observe the output:

```
signal_1d = np.array([0, 0, 100, 200, 100, 0, 0], dtype=np.float64)
kernel_1d = np.array([1/3, 1/3, 1/3])   # 3-point moving average

result_same = np.convolve(signal_1d, kernel_1d, mode='same')

print("signal :", signal_1d)
print("result :", np.round(result_same, 1))
```

`mode='same'` keeps the output the same length as the input, padding with zeros at the edges — exactly what your **`convolve2d`** does.

(b) **Apply np.convolve row by row.** **`np.convolve`** only works on 1D arrays, but you can apply it to each row of a 2D image in a loop. Complete the function below by filling in the missing argument:

```
def convolve_rows(image, kernel_1d):
    """Apply a 1D kernel to every row of a 2D image."""
    h, w = image.shape
    output = np.zeros_like(image, dtype=np.float64)
    for row in range(h):
        output[row, :] = np.convolve(image[row, :], kernel_1d, mode=...)
    return output
```

(c) **Separable filtering.** A 2D box blur can be decomposed into two 1D passes: first blur every row, then blur every column of the result. This is called a separable filter and is much faster for large kernels. Apply it to your image and save the result:

```
k = np.array([1/5, 1/5, 1/5, 1/5, 1/5])   # 5-point moving average

# Pass 1: blur every row
after_rows = convolve_rows(img, k)

# Pass 2: blur every column by transposing, blurring rows, then transposing back
after_cols = convolve_rows(after_rows.T, k).T

save_image(after_cols, "blur_separable.png")
```

Compare **`blur_separable.png`** with **`blur_5x5.png`**. Do they look the same?

(d) **Verify the results match.** Check that your separable filter gives the same result as **`convolve2d`** with the 5×5 box kernel:

```
result_2d  = convolve2d(img, box5)
result_sep = after_cols

print("Max difference:", np.max(np.abs(result_2d - result_sep)))
print("Results match:", np.allclose(result_2d, result_sep))
```