Essential 2D Transforms & Filters

Fourier, Radon, and convolution
+ a "practical" application

Gia Jadick

October 27, 2025

1 Introduction

Welcome to the second seminar in the GPMP practical skills seminar series! The materials necessary for this week can be found in Jupyter Notebooks for `week2_transforms`. We'll be taking a closer look at the 2D transforms you have learned thus far in the math course. In this seminar, you will learn what these transforms look like in actual images and how they can be implemented using Python. This will hopefully aid you in your future reserch and practicum courses. We'll wrap things up with an independent application, which we'll each share with each other!

Learning Objectives:

- Enhance conceptual understanding of 2D transforms.
- Develop boilerplate code for future research implementations of these transforms (useful for work in CT, MRI, and more)
- Independently apply these skills to develop your own application utilizing one or more transforms.

2 Independent Exercise: Vision Simulator

Is your vision 20/20? A variety of "blurry vision simulators" have been developed and shared online to demonstrate how various uncorrected vision levels appear. This is one of the best I've found. Take a second to check it out and experiment with the inputs. Today, you will be building a vision simulator of your own using what we've learned about convolution! We will start with a simple implementation and build upon it depending on time. Figure 1 shows a nice schematic of the 2D convolution operation.

Start by navigating to the environment you set up in last week's seminar. Additional packages you'll want to have installed for this week are `scipy` and `imageio` if you don't already have them. Then, boot up Jupyter Lab.

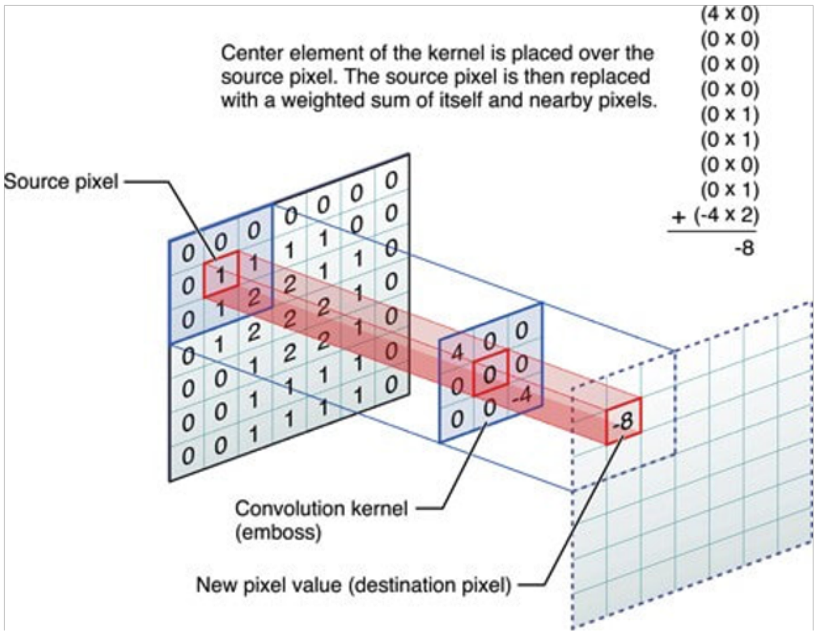
Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

Figure 1: Conceptual schematic of the 2D convolution.

2.1 Choose your image

Decide what you want the subject of your simulator to be. The link above has a few options, like classrooms and a gymnasium. Be creative but strategic—there should be a reasonable degree of depth to your input image (10–20 feet is good to start).

Find the image online and save it in the `week2` folder, then load it into your Jupyter Lab and show it. You can start with grayscale for a simpler task or launch right into RGB if you are comfortable working with multiple channels. If you forgot how to load images, you can always reference part 4 from the `coding-in-science` repository.

2.2 Choose your kernel

What is the most realistic kernel for blurry vision? Does it make a difference for near- or far-sightedness? Do a bit of research on what might be most faithful, and feel free to make approximations and experiment with different kernels. You have several options here.

2.3 Implement the 2D convolution

Now convolve the image and kernel to get the final blurred result. You can reference the code from the first half of the seminar. Make a plot with three subpanels showing the image, kernel, and convolved output. Here you can test your different kernel ideas.

If you have non-20/20 vision, which kernel do you think produces images most similar to what you see without glasses? How do you simulate different degrees of bad vision?

2.4 Make the vision simulator function

Define a function that takes as input a single float representing the degree of near- or far-sightedness. (Negative = near-sighted, most common, perhaps easiest to just start with that). Within your function, generate the kernel for that degree of vision. Return a single output: the blurred image. You can make the original background image either a global variable or optional input to the function.

How does it look? Critically assess your outputs, compare to the website from earlier, and explore alternative vision simulator websites available online. What makes one simulator look "better" than another? Think of ways you might improve your simulator.

2.5 Advanced refinements: depth (optional)

Now that you've looked at the other simulators, ideally you have some ideas on what makes them look more realistic. A key issue is the idea of *depth*. A raw convolution blurs everything in the image equally, whether near or far. In reality, different distances should have various degrees of blur; for example, with near sightedness, objects closer to the viewer may be in focus while the background is very blurry.

In your simulator, you could solve this by either choosing an image with everything roughly a single distance from the camera, or you could take an advanced approach of *layering images* with transparency at different depths so you can blur each depth to a different degree.

Dig around online for potential multi-depth layer images you could use for this. You could also try manually segmenting an image into depth layers using Photoshop or a free online alternative. Save this to your directory, load them into your code, and rewrite your function to account for the different depths.

How does your simulator look now? Do you have more ideas for improvements?

3 Conclusion

And that's it, great work! I hope this has been equal parts helpful and entertaining.

— *Gia*