

## Problem Set 1: Image processing

**Posted:** Tuesday, August 26, 2025

**Due:** Monday, September 15, 2025

Please submit your written solution to [Gradescope](#) as a .pdf file. Please convert your Colab notebooks to PDF. For your convenience, we have included the PDF conversion script at the end of the notebook.

Starter code:

- Submit your written solution to 1.1 as a PDF file (either written or typeset).
- The notebook for problems 1.2 can be found at:  
<https://colab.research.google.com/drive/1YML0Vd8HCb1W6EsZt-QWELWUlawYBzue?usp=sharing>
- The notebook for problems 1.3 and 1.4 can be found at:  
[https://colab.research.google.com/drive/1U7eq7NfelJgNMI3Q1tbI0\\_uYXtpm84bC?usp=sharing](https://colab.research.google.com/drive/1U7eq7NfelJgNMI3Q1tbI0_uYXtpm84bC?usp=sharing)

We recommend editing and running your code in Google Colab, although you are welcome to use your local machine instead. Please note that problems marked optional will not be graded.

### Problem 1.0 *numpy review (optional)*

We have provided a Colab notebook containing a brief introduction to `numpy`. If you are new to `numpy` and numerical computing, we encourage you to work through these examples. They should cover the background that you need to complete this problem set. The Colab notebook can be found at:

<https://colab.research.google.com/drive/1ysPA0NyV7gyySrb8QN5PWs1YgbfJRL1H?usp=sharing>

### Problem 1.1 *Properties of convolution*

Recall that 1D convolution between two signals  $f, g \in \mathbb{R}^N$  is defined:

$$(f * g)[n] = \sum_{k=0}^{N-1} f[n-k]g[k]. \quad (1)$$

(a) Construct a matrix multiplication that produces the same result as a 1D convolution with a given filter. In other words, given a filter  $f$ , construct a matrix  $H$  such that  $f * g = Hg$  for

any input  $g$ . Here  $Hg$  denotes matrix multiplication between the matrix  $H$  and vector  $g$  (**2 points**).

*Note:* One option is to define  $H$  in “bracket” notation using “...” symbols, e.g.,  $H = \begin{bmatrix} 0 & 1 & \dots & N-1 \\ N-1 & N-2 & \dots & 1 \end{bmatrix}$ . Another option is to explicitly define the entries  $H_{ij}$ .

(b) In class, we showed that convolution with a 2D Gaussian filter can be performed efficiently as a sequence of convolutions with 1D Gaussian filters. This idea also works with other kinds of filters. We say that a 2D filter  $F \in \mathbb{R}^{N \times N}$  is separable if  $F = uv^T$  for some  $u, v \in \mathbb{R}^N$ , i.e.,  $F$  is the *outer product* of  $u$  and  $v$ . Show that if  $F$  is separable, then the 2D convolution  $G * F$  can be computed as a sequence of two one-dimensional convolutions (**2 points**).

(c) (*optional*) Show that convolution is commutative, i.e.,  $f * g = g * f$ . Assume circular padding (e.g.,  $f[-1] = f[N-1]$ ), zero padding, or whatever is convenient. *Note: we will not grade this problem, and you will **not** get bonus points for completing it.*

**Hint:** Write the equation for  $f * g$ , and figure out how to “rename” the variable used in the summation to arrive at  $g * f$ .

(d) (*optional*) Show that convolution is associative, i.e.  $(f * g) * h = f * (g * h)$ .

**Hint:** Start by writing down the expression for  $G * F$  and then, inside the double summation, write  $F$  in terms of  $u$  and  $v$ .

(e) (*optional*) Show that cross-correlation,

$$h[n] = \sum_{k=0}^{N-1} f[n+k]g[k], \quad (2)$$

is *not* commutative.

### Problem 1.2 *Sponsored problem: pet edge detection*

As you know, this course is largely funded by grants from Petsco, Inc.<sup>™</sup>. Unfortunately, one of the strings attached to this funding is that we must occasionally assign *sponsored problems* that cover topics with significant business implications for our sponsor<sup>1</sup>. Our partners at Petsco see an opportunity to use computer vision to pull ahead of their bitter rivals, Petmart<sup>™</sup>. To avoid painstakingly marking the edges of dogs and cats in pictures by hand, the current industry best practice, they have turned to us for an automated solution.

(a) Apply the horizontal and vertical gradient filters  $[1 \ -1]$  and  $[1 \ -1]^T$  to the picture of the provided pet<sup>2</sup> producing filter responses  $I_x$  and  $I_y$ . Write a function `convolve(im, h)` that takes a grayscale image and a 2D filter as input, and returns the result of the convolution.

<sup>1</sup>The class (unfortunately) is not actually funded by Petsco.

<sup>2</sup>These convolution filters are  $1 \times 2$  and  $2 \times 1$  matrices respectively. Even though these filters are not square, your convolution function should have no problem using them, since it should be able to handle filters of any dimension.



(a) An input image



(b) “Fluffy coat” failure case

Figure 1: (a) A pet photo helpfully delivered by our sponsor. (b) A failure case for a simple edge detector. These images are provided in the starter code. Photo credits can be found [here](#).

Please do not use any “black box” filtering functions for this, such as the ones in `scipy`<sup>3</sup>. You may use `numpy.dot`, but it is not necessary. Instead, implement the convolution as a series of nested `for` loops. Compute the *edge strength* as  $I_x^2 + I_y^2$ . After that, create visualizations of  $I_x$ ,  $I_y$ , and the edge strength, following the sample code.

The filter response that your function returns should be the same dimensions as the input image. Please use *zero padding*, i.e. assume that out-of-bounds pixels in the image are zero. Also please make sure to implement convolution, *not* cross-correlation. Note that this simple filtering method will have a fairly high error rate — there will be true object boundaries it misses and spurious edges that it erroneously detects. The team at Petsco, thankfully, has volunteered to fix these errors by hand (**4 points**).

(b) (*optional*) This method detects edges fairly well on one of the dogs, but not the other. Our sponsor would like us to investigate why this is happening. First, convert your gradient outputs to edge detections using a hand-chosen threshold  $\tau$  (i.e., set strength values at most  $\tau$  to 0 and those above to 1). Point out 2 errors in the resulting edge map — that is, edge detections that do not correspond to the boundary of an object — and explain what causes these errors.

(c) While the edge detector you submitted works well on some pets, engineers are reporting a large number of failures, and the Petsco execs are *not* happy. Worrisomely, it has been failing on dogs with fluffy coats, such as “doodle” mixes — a lucrative market for our sponsor (Figure 1b). The source of this error, Petsco’s team has concluded, is that the gradient filter is often firing on small, spurious edges in highly textured regions.

Please kindly address this problem by creating an edge detector that only responds to edges at larger spatial scales. To do this, blur the image with a Gaussian filter prior to computing gradients. Implement your Gaussian filter on your own<sup>4</sup>. Please do not use any “black box” Gaussian filtering functions for this, such as `scipy.ndimage.gaussian_filter`. Apply both the Gaussian filter and the gradient filter using a black box convolution function

<sup>3</sup>Our sponsor has been prohibited from using these functions as part of the terms of an lawsuit that has been rumored to involve a dog bite and a key member of the `scipy` development team.

<sup>4</sup>Hint: Make sure that the kernel is properly centered.

`scipy.ndimage.convolve` on the image, rather than your hand-crafted solution.

- (i) Compute the edges without blurring, so we can look at the before-and-after results.
- (ii) Compute the blurred image using  $\sigma = 2$  and an  $11 \times 11$  Gaussian filter.
- (iii) Instead of blurring the image with a Gaussian filter, use a box filter (i.e., set each of the  $11 \times 11$  filter values to  $1/11^2$ ).
- (iv) Compute edges on the two blurred images.
- (v) (*optional*) Do you see artifacts in the box-filtered result? Describe how the two results differ. Include your written response in the notebook.

Visualize the edges in the same manner as Problem 1.2(a). Your notebook should contain the edges that were computed: (i) using no blurring, (ii) using Gaussian blurring, and (iii) box filtering. Please also show the (iv) Gaussian blurred image, (v) the box-filtered image (**2 points**).

(d) Instead of convolving the image with two filters to compute  $I_x$  (i.e., a Gaussian blur filter followed by a gradient filter), create a *single* filter  $Gx$  that yields the same response. You can reuse the function in part (c). Visualize this filter using the provided code (**2 points**).

(e) Good news. Petsco execs are pleased with your work and have renewed our funding for several more problem sets. At our last meeting, however, they made an additional request. Their competitors at Petmart are now computing *oriented* edges of their pets. Rather than estimating the only horizontal or the vertical gradients, they now can provide gradients for an arbitrary angle,  $\theta$ . Petmart's method for doing this, however, is computationally expensive: they construct a new filter for each angle, and filter the image with it. Petsco sees an opportunity to pull ahead of their rivals, using the class's knowledge of steerable filters.

Write a function `oriented_grad( $I_x, I_y, \theta$ )` that returns the gradient steered in the direction  $\theta$ , given the horizontal and vertical gradients  $I_x$  and  $I_y$ . Use this function to compute your gradients on a blurred version of the input image at  $\theta \in \{\frac{1}{4}\pi, \frac{1}{2}\pi, \frac{3}{4}\pi\}$ , using the same Gaussian blur kernel as (c). Visualize these results in the same manner as the gradients in Problem 1.2 (a) (**2 points**).

(f) Petsco would also like to offer the ability to enhance low quality pet photos. Given a noisy image of a pet, please remove the noise by applying the box and Gaussian filters that you have developed in (c). Then, implement a function `median(im, k)` that performs *median filtering* using a  $k \times k$  window. More specifically, for each pixel of the noisy image, you will compute the median intensity value of the other pixels within a  $k \times k$  window. Your code should be very similar to your code for `convolve`, but instead of computing a weighted average in the inner loop, you will compute the median. You may use a built-in function for computing the median of a list or array, such as `np.median`. Include the results for  $k = 3$  and  $k = 7$ .

Your solution should display the following 5 images: (i) the noisy image itself, (ii) the box filtered noisy image, (iii) the Gaussian filtered noisy image, (iv) median filtered noisy image with  $k = 3$ , (v) median filtered image with  $k = 7$ . (**3 points**)

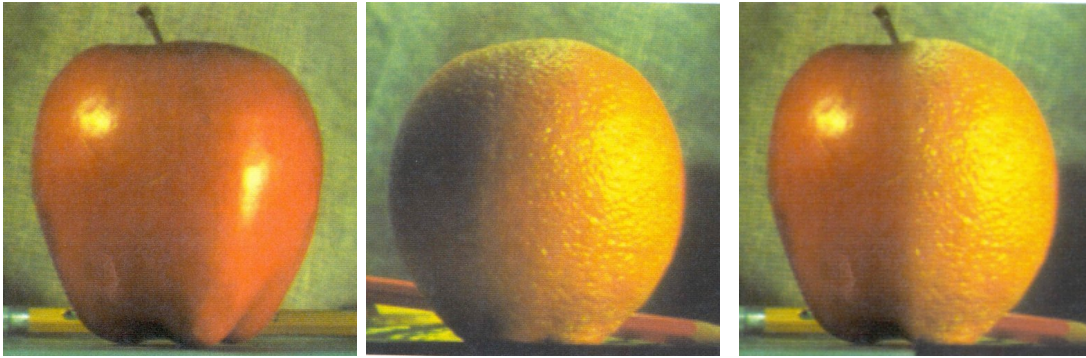


Figure 2: Blending with a Laplacian pyramid of 6 levels. Note that your result may look different from ours.

### Problem 1.3 *Image blending*

We will use the Laplacian pyramids to blend two images.<sup>5</sup>

(a) **(5 points)** First, we will implement the following functions, which will be used to create a Laplacian pyramid from an image, and to reconstruct an image from a Laplacian pyramid.

You'll recall that we'll need to downsample in both the Gaussian and Laplacian pyramids. In your implementation, use Gaussian kernels for `pyramid_upsample` and `pyramid_downsample`. The kernel for `pyramid_upsample` should be the same as the one for `pyramid_downsample`, except that the kernel itself will be multiplied by 4. (Hint: `np.insert` may come in handy when implementing `pyramid_upsample`. Also, `scipy.ndimage.gaussian_filter` may come in handy when implementing `pyramid_upsample` and `pyramid_downsample`. Make sure to read about their "radius" argument in order to set the correct blur strength). Set the standard deviation of the Gaussian kernel to be  $\sigma = 1$ .

- `pyramid_upsample`
- `pyramid_downsample`

Now that you can downsample and upsample your images, you can implement the Gaussian and Laplacian pyramids. (Hint: remember that you need the highest level of the Gaussian pyramid to start the Laplacian pyramid.)

- `gaussian_pyramid`
- `laplacian_pyramid`

Now that you can generate a Laplacian pyramid, you can reconstruct the original image with it. Use a Laplacian pyramid with 4 levels. Recall from lecture that, to reconstruct the original image, you repeatedly upsample the Laplacian (starting with the highest level of the Gaussian pyramid), and then add back the Laplacian from the next level of the pyramid. Please plot the original image, the Laplacian pyramid, and the reconstructed image. (Also, `numpy` and `cv2` libraries perform a clipping when subtracting images. Try to use a different method for image subtraction to make sure clipping doesn't happen!)

- `reconstruct_image`

---

<sup>5</sup>This problem was originally based on a problem set by William Freeman and Antonio Torralba.

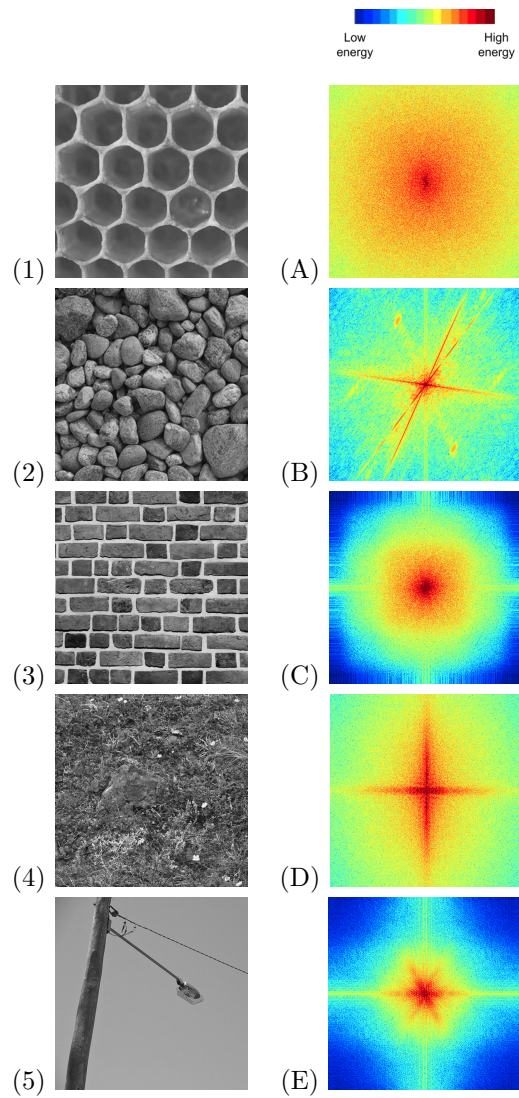


Figure 3: Match each image with its corresponding Fourier transform magnitude.

(b) **(2 points)** Implement the function `pyramid_blend(im1, im2, mask, num_levels)`. Its inputs are two images and a binary mask (indicating which pixels to use from each image). The function produces a Laplacian pyramid with `num_levels` levels that will blend the two image inputs. Use your function to blend the images of an orange and an apple that we provided in the Colab notebook. Plot the blended images with `num_levels`  $\in \{1, 2, 3, 4, 5, 6\}$ . Please describe the difference between the blended images as the number of levels in the Laplacian pyramid varies: how does the result change as we increase the number of levels? Please include this in the cell in the `.ipynb` file, rather than uploading a separate document with a written answer.

To obtain color images, you can apply the blending to each color channel independently. In our implementation, this did not require any extra code (the same code worked on single-channel and multi-channel images due to `numpy broadcasting`). However, your implementation may differ.

(c) **(optional)** Use your code to blend your own images. If you would *not* like your blending

results shown in class, please let us know!

**Problem 1.4** *Fourier Transform*

(a) **(1 point)** Please match each image in the left column of Fig. 3 to its corresponding Fourier spectrum visualization in the right column. Write your answers in a colon separated format (i.e. 1:A, 2:E, ...) in the notebook.

(b) **(2 point)** Convolve the provided image with a Gaussian filter in two ways: i) using direct convolution in the spatial domain, and ii) product in the frequency domain (via the convolution theorem). To perform DFT and inverse DFT, use `fft2` and `ifft2` from `scipy.fft`. For 2D convolution, we use `scipy.signal.convolve2d`. Note that doing this in the spatial domain versus the frequency domain might result in slightly different results near the image boundaries (which is not an issue for this problem).