Assignment 06

Please refer to MyCourses/Gradescope for assignment deadlines.

The goal of this assignment is to give you hands-on practice with the following concepts:

- The 2D Discrete Fourier Transform
- Reasoning about high- and low-frequency components in images
- Working with complex numbers
- The Convolution Theorem

Part 1 of 3: Hybrid low- and high-pass images

In this part, you will implement a "cute" demo of the 2D DFT. You will be creating what are known as "hybrid images". These are images that look like one thing when viewed from far away, and another thing when viewed up close. You will be implementing the missing code in `hybrid.py`, namely the `high_pass` and `hybrid` functions. The source images `elephant.png` and `cheetah.png` have been provided as some example inputs.

For a deep dive into what Hybrid Images are and how they work, you can read the original 2006 paper by Oliva et al., titled simply "Hybrid images". Or, we can just give you the gist here:

We start with two images:


And we will combine the **low frequency** parts of the first image with the **high frequency** parts of the second image to produce a **hybrid image** which looks something like this:

Figure 1: Hybrid image

Note that when viewed from far away, you see the low-frequency components of the image (the elephant). When viewed up close, you see the high-frequency components of the image (the cheetah).

The **low frequency** part of the first image looks like this:

Figure 2: Elephant low-pass

…and is computed by multiplying the 2D DFT of the image by a low-pass filter which looks like this (after `fftshifting` to put the DC component in the center):

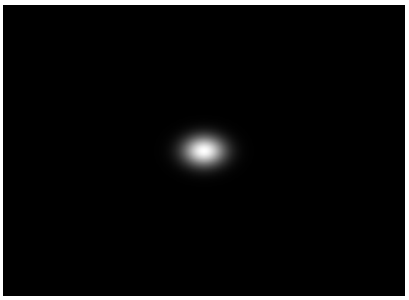
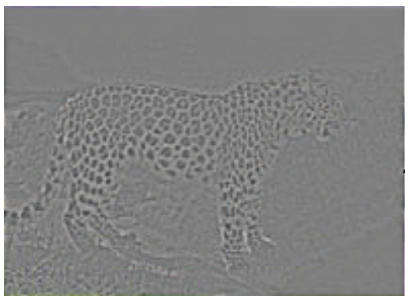
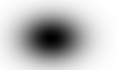


Figure 3: Low-pass filter

Likewise, the high-frequency part of the second image and its corresponding high-pass filter look like this:





You must do three things:

1. Implement the `high_pass()` function
2. Implement the `hybrid()` function
3. Specify good values for the cutoff frequencies at the top of the file. Note that the units are in "cycles per pixel", so the smallest possible frequency is 0.0 and the largest is 0.5 (the Nyquist frequency).

Part 2 of 3: Implementing the Convolution Theorem

In this part, you will once again implement a convolution function yourself, like you did in assignment 3. However, this time you will use the Convolution Theorem to do so in the frequency domain rather than the spatial domain. You'll be implementing the missing code in `convolution_theorem.py`, namely the `convolution_theorem()` function.You may use any of the images in `src/images` and any of the filters in `src/filters` for testing.

Recall from class that the convolution theorem states that if

$$g = f * h$$

, i.e. g is the result of convolving the image f with the kernel h, then

$$G = F \cdot H$$

, where G, F, and H are the 2D DFTs of g, f, and h, respectively, and $\cdot$ denotes element-wise multiplication. (Note that in numpy, you'll use $\star$ for element-wise multiplication, and the provided `conv2D` function for convolution.) Thus, the steps involved to convolve f with h are:

1. Compute the 2D DFT of f and h
2. Multiply the results element-wise
3. Compute the inverse 2D DFT of the result

Note that `fftshift` and `ifftshift` are not required; they are just useful for visualization, not for actual computation in the frequency domain.

As in assignment 3, the tricky part is handling borders. The Fourier transform essentially assumes that your image is periodic, so if you do nothing to handle the borders, your result will be *as if* you used `border='cv.BORDER_WRAP'` in `conv2D`. This can in principle be handled by zero-padding the image first. If you do this, you will get partial but not full credit. The full solution requires you to handle borders *as if* you are zero padding, but using only the features built in to `fft2`. (Hint: using the `s` argument, you will tell `fft2` that the image is bigger than it actually is, and under the hood, it will treat the image as being zero-padded.)

In this part of the assignment, you may *not* assume grayscale (single-channel) images. Hint: you will also need the `axes=` argument to `np.fft.fft2` and `np.fft.ifft2` to handle multi-channel images.

Part 3 of 3: Low-pass filtering and camera focus

In this part of the assignment, you will make use of the *power spectrum* of an image to determine whether it is in focus. Essentially, you will be emulating the 'autofocus' feature present on many modern cameras. You need to fill in the missing code in `autofocus.py`.

The images in `src/images/focus*.JPG` have been provided for you. They are images of a scene that contains objects at varying depths from the camera. Each image was taken with a slightly different manual focus setting (distance from the lens to the sensor). The setup looked like this:

Figure 4: Rig used to create the focus images

(Providing you this set of 6 images is the best way we can *simulate* the effect of changing focus without resorting to running things on a real camera or using a fancy renderer).

You are creating a mini point-and-click application. The idea is that the user will click on a point in the scene, and your program will determine which of the 6 images is in focus at that point. There is also a `radius` parameter that determines how big of a circle around the clicked point to consider. Here are some example screenshots from the answer key (clicking on 3 different points – pay attention to the green circle moving around to where the user clicked):

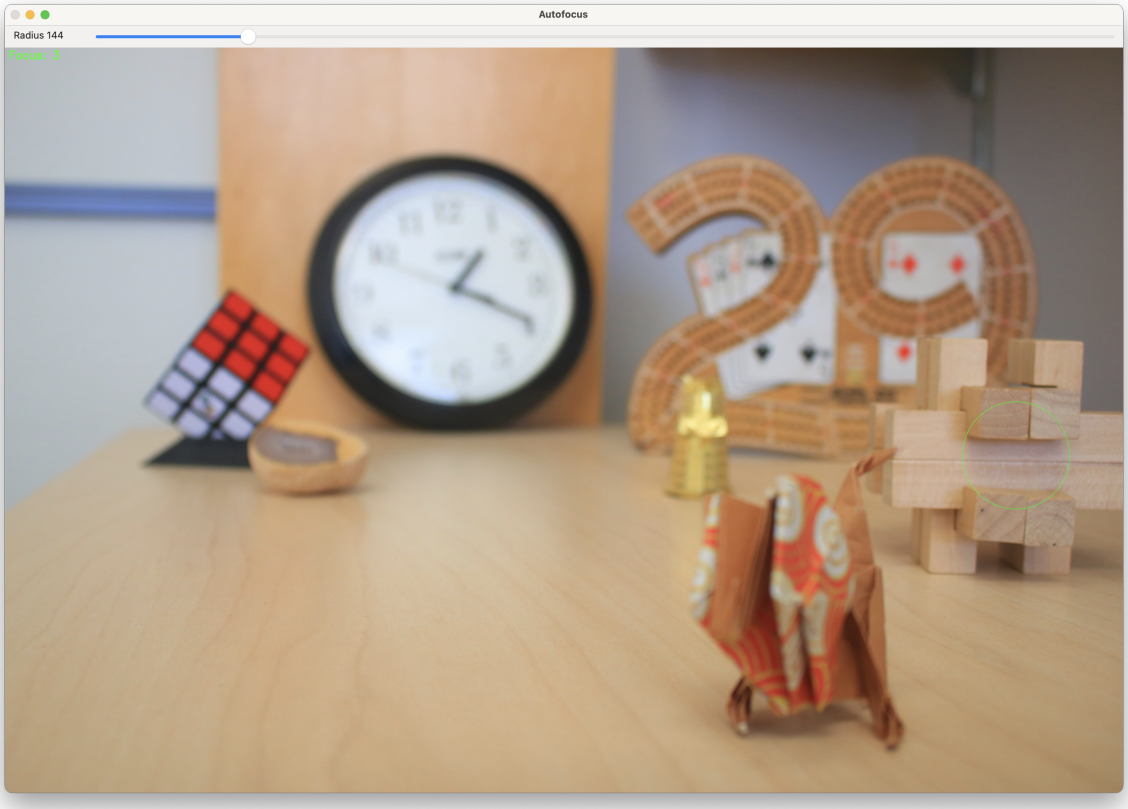


Figure 5: Example 1

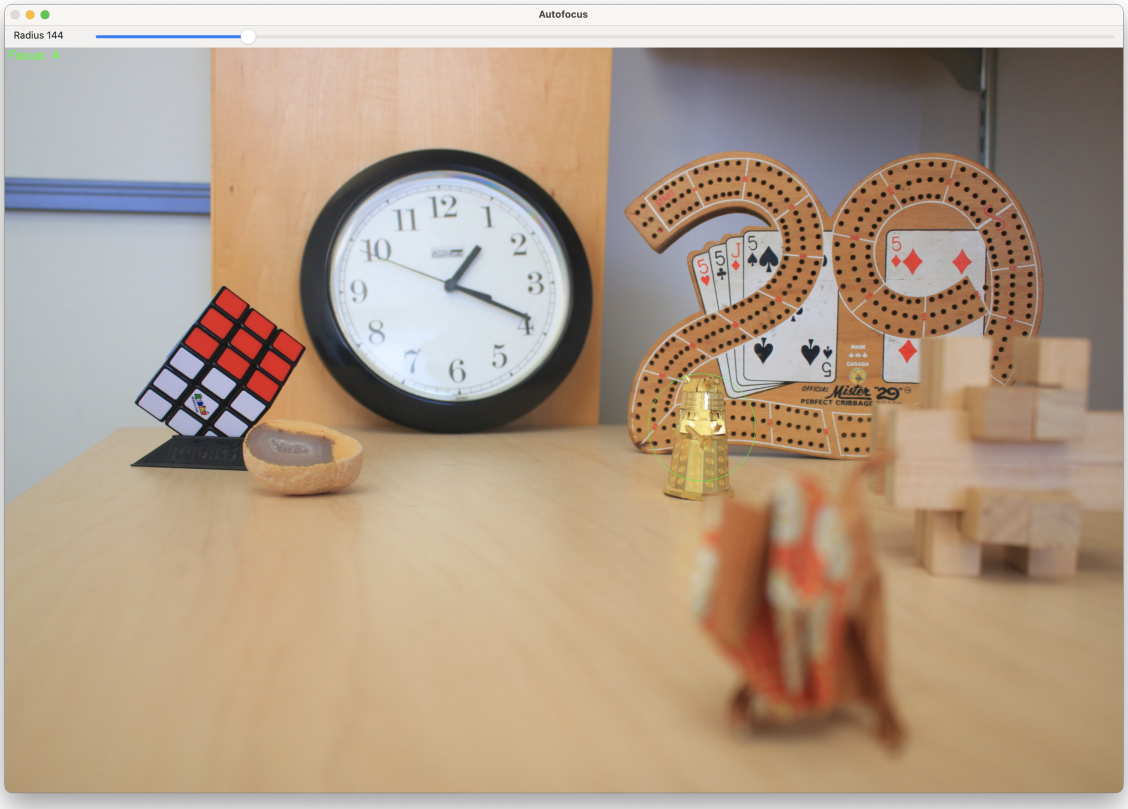


Figure 6: Example 2

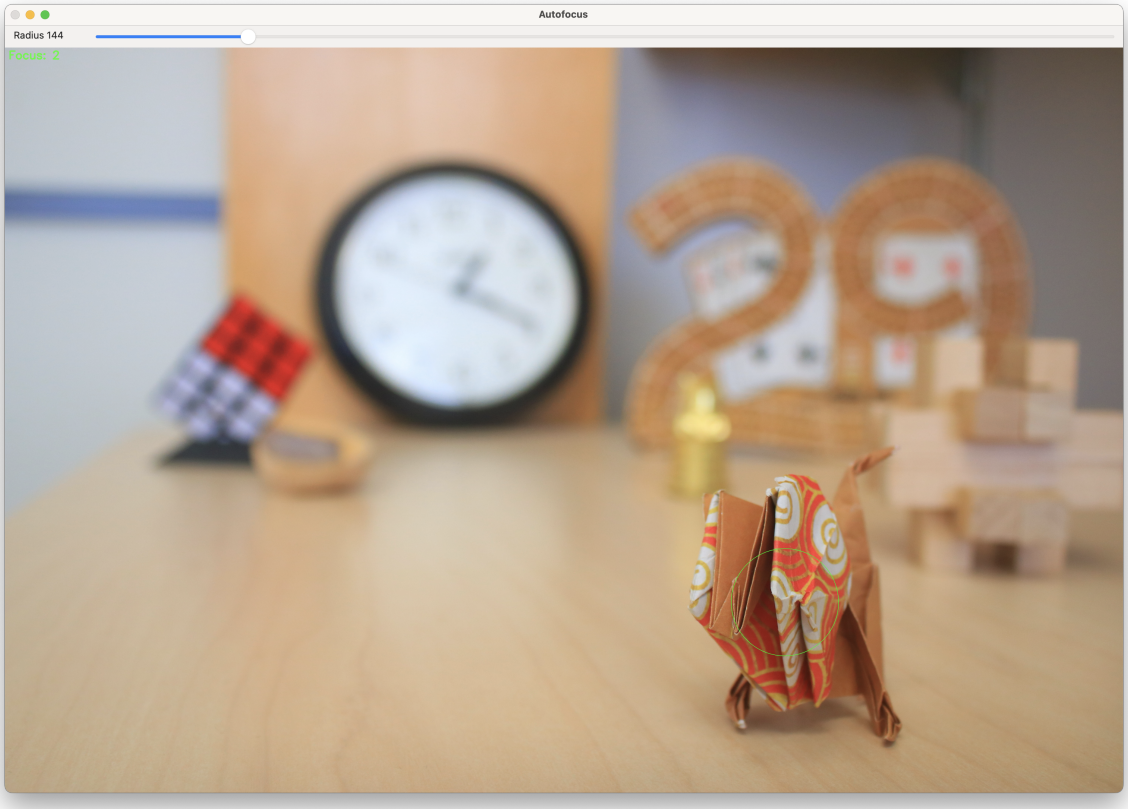


Figure 7: Example 3

You need to implement three functions in `autofocus.py`:

- `get_mask()` to create the Gaussian mask
- `average_power_in_band()` to select frequencies between the lower and upper bounds and average the power for those frequencies
- `select_best_focus()` the main logic for selecting the best focus

The main entrypoint is the `select_best_focus()` function. It takes in a list of images, a point (x, y), and a radius. It returns the index of the image that is in focus at the given point. A `plot_power_spectrum()` function has been provided for you which you can use to visualize the power spectrum of a given image. It's not currently used in the file, but provided for you to help you debug and visualize things (it's very useful to help you figure out the 'band' below).

Here's how it works:

- Images are converted to grayscale for the purposes of calculating the power spectrum.

- A “masked image” for each focus setting is created by multiplying the grayscale image with a Gaussian mask centered at the clicked point. The `radius` parameter controls the standard deviation of the Gaussian mask (specifically, `radius = 3 * sigma`). It looks like this:

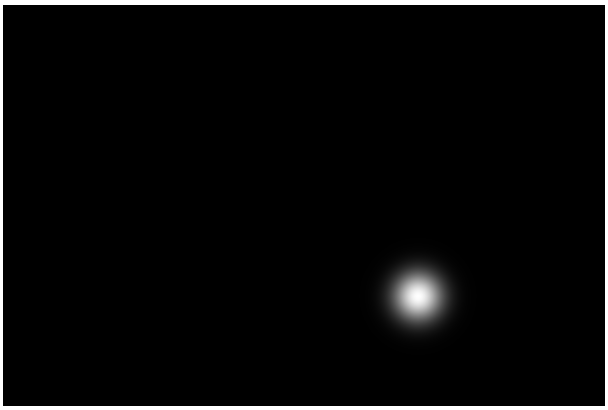


Figure 8: Gaussian mask

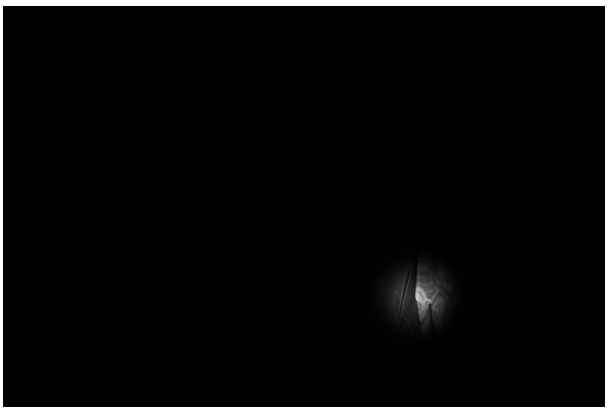


Figure 9: Masked grayscale image

- The power spectrum of each masked image is calculated. “Power” refers to the squared magnitude of the 2D DFT. Each ‘pixel’ in the power spectrum tells us the amount of power in the image at a particular frequency. A `calculate_power_spectrum()` function has been provided for you. It returns a tuple of `freqs, power` such that `power[i,j]` is the power at spatial frequency `freqs[i,j]`.

  Here, spatial frequency is measured in units of “cycles per pixel”. Thus, the smallest frequency is 0 (DC component), and the largest frequency is 0.5 (because, as we saw in class, one cycle every 2 pixels is the theoretical max, AKA the Nyquist frequency).

- The power spectrum is averaged over all frequencies with a “band” specified by some lower

and upper bounds on the frequency. You must set the `_DEFAULT_FREQ_BAND` variable at the top of the file to tell us what values you found worked well. We recommend using the `plot_power_spectrum()` function while debugging to help you figure out what the band should be. You need to implement the `average_power_in_band()` function to do this.

- Since depth blur is a *low pass filter*, and since low-pass filtering dampens the amplitude of high-frequency ("crisp") parts of an image, the image with the least blur will be the one with the greatest average power. Thus, the output of `select_best_focus()` should be the index of the image with the most average power (in the band of frequencies you selected).

Your job is to fill in the missing code in `autofocus.py`. You should run it on the provided images. Remember also to set the `_DEFAULT_FREQ_BAND` variable at the top of the file. This variable is important for getting the autofocus to work well.

P.S. You may notice that the result is somewhat slow (on the order of 0.7 seconds per click on the instructor's laptop). This is because the implementation is not optimized – the steps above are intended to show the process, not to be the fastest way to do it. If you're looking for a challenge, here's some guidelines for how to make it actually fast:

- Everything can be done in the Fourier domain directly if you precompute the Fourier transform of each image.
- The tricky part is implementing `mask * image` in the Fourier domain. However, the convolution theorem works both ways! Element-wise multiplication in the image domain is equivalent to convolution in the Fourier domain. Thus, `mask * image` is equivalent (up to border effects, etc) to `ifft2(conv2D(fft2(image), fft2(mask)))`.
- Recall that the Fourier transform of a Gaussian mask is a Gaussian, and the Fourier transform of a shifted input is a phase shift in the Fourier domain. Thus, you could in principle create a function that directly computes `fft2(mask)` from `(x, y)` and `radius`.
- The convolution in the frequency domain is separable, since `fft2(mask)` is itself Gaussian in the frequency domain.
- Thus, you can in principle compute the convolution of `fft2(image)` with `fft2(mask)` using (i) a precomputed `fft2(image)`, (ii) directly computing `fft2(mask)` from `(x, y)` and `radius` in linear time, and (iii) fast separable convolution in the Fourier domain.

Consider this a challenge for those who are interested. It is by no means a required part of the assignment.

Collaboration and Generative AI disclosure and Assignment Reflection

Did you collaborate with anyone? Did you use any Generative AI tools? How did the assignment go? Briefly explain what you did in the `collaboration-disclosure.txt` file. **Completing the collaboration disclosure for each assignment is required.** Completing the reflection part of the file is optional, but it is a good way to give feedback to the instructor and grader about how the course is going.

Submitting

As described above, use `make submission.zip` to generate a zip file that you can upload to Gradescope. You can upload as many times as you like before the deadline. To account for late penalties, contact the instructor or grader directly if you plan on uploading any further revisions after the deadline.