EECS 442 Computer Vision: Homework 0 – Numbers and Images

Instructions

- This homework is **due at 11:59:59 p.m. on Wednesday February 3, 2021**.
- The submission includes two parts:
  1. **To Gradescope:** a `pdf` file as your write-up, including your answers to all the questions. *Please do not handwrite and please mark the location of each answer in gradescope.*
  2. **To Canvas:** a `zip` file including all your code.

1 Overview

In this assignment, you'll work through three tasks that help set you up for success in the class. The document seems big, but most of it is information, advice, and hand-holding. The assignment has two goals.

First, the assignment will show you bugs in a low-stakes setting. You'll encounter a lot of programming mistakes in the course. If you have buggy code, it could be that the *concept* is incorrect (or incorrectly understood) or it could be that the *implementation* in code is incorrect. You'll have to sort out the difference. It's a lot easier if you've seen the bugs in a controlled environment where you know what the answer is. Here, the programming problems are deliberately easy and we even provide the solution for one!

Second, the assignment incentivizes you to learn how to write reasonably good python and numpy code. You should learn to do this anyway, so this gives you credit for doing it and incentivizes you to learn things in advance.

The assignment has three parts and corresponding folders in the starter code:

- Numpy (Section 2 – folder `numpy/`)
- Image dithering (Section 3 – folder `dither/`)
- Data visualization (Section 4 – folder `visualize/`)

Here's my recommendation for how to approach this homework:

- If you have not had any experience with numpy, read this tutorial. Numpy is like a lot of other high-level numerical programming languages. Once you get the hang of it, it makes a lot of things easy. However, you need to get the hang of it and it won't happen overnight!
- You should then do Section 2.
- You should then read our description about images in Section A. Some will make sense; some may not. That's OK! This is a bit like learning to ride a bike, swim, cook a new recipe, or play a new game

by being told by someone. A little teaching in advance helps, but actually doing it yourself is crucial. Then, once you've tried yourself, you can revisit the instructions (which might make more sense).

If you haven't recently thought much about the difference between an integer and a floating point number, or thought about multidimensional arrays, it might be worth brushing up on both.

- You should then do Section 3 and then Section 4. Both are specifically designed to produce common bugs, issues, and challenges that you will likely run into the course. As you run into issues (both now and later in the course), you may want to go back to Section A.

What to submit:

- **Canvas**: submit a zip file of all of your code. This should contain a single directory which has the same name as your uniqname. If I (David, uniqname `fouhey`) were submitting my code, the zip file should contain a single folder `fouhey/` containing the folders from the homework. So, for instance, my zip file should contain `fouhey/numpy/run.py` and `fouhey/dither/dither.py`. We provide a script that validates the submission format here.

- **Gradescope:** submit a pdf file with the answers to the questions from each section.

  The write-up must be electronic. *No handwriting!* You can use Word, Open Office, Notepad, Google Docs, LaTeX(try overleaf!), or any other form.

  You might like to combine several files. Here is an example online link for combining multiple PDF files: https://combinepdf.com/. Try also looking at this stack overflow post.

Python Environment

We are using Python 3.7. We will make use of the following packages extensively in this course:

- Numpy (https://docs.scipy.org/doc/numpy-dev/user/quickstart.html).
- Matplotlib (http://matplotlib.org/users/pyplot_tutorial.html).
- OpenCV (https://opencv.org/).

Changelog:

23 Jan: correction to dithering pseudocode; clarification about hint for visualization question.

27 Jan: misc additional clarifications and hints; correction in hint for visualizing IQ; additional info about images in image section.

29 Jan: correction to dithering pseudocode (seems to have been clobbered?).

2 Numpy Intro

All the code/data for this is located in the folder `numpy/`. Each assignment requires you to fill in the blank in a function (in `tests.py` and `warmup.py`) and return the value described in the comment for the function. There's a driver code you do not need to read in `run.py` and `common.py`.

When you open one of these two files, you will see starter code that looks like this:

```
def sample1(xs):
    """
    Inputs:
    - xs: A list of values

    Returns:
    The first entry of the list
    """
    return None
```

You should fill in the implementation of the function, like this:

```
def sample1(xs):
    """
    Inputs:
    - xs: A list of values

    Returns:
    The first entry of the list
    """
    return xs[0]
```

You can test your implementation by running the test script:

```
python run.py --test w1          # Check warmup problem w1 from warmups.py
python run.py --allwarmups       # Check all the warmup problems
python run.py --test t1          # Check the test problem t1 from tests.py
python run.py --alltests         # Check all the test problems

# Check all the test problems; if any of them fail, then launch the pdb
# debugger so you can find the difference
python run.py --alltests --pdb
```

This will show:

```
python run.py --allwarmups
Running w1
Running w2
...
Running w20
Ran warmup tests
20/20 = 100.0
```

2.1 Warmup Problems

You need to solve all 20 of the warmup problems in `warmups.py`. They are all solvable with one line of code.

2.2 Test Problems

You need to solve all 20 problems in `tests.py`. Many are not solvable in one line. You may not use a loop to solve any of the problems, with the exception of t10 (but this one can also be solved without loops).

Here is one example:

```
def t4(R, X):
    """
    Inputs:
    - R: A numpy array of shape (3, 3) giving a rotation matrix
    - X: A numpy array of shape (N, 3) giving a set of 3-dimensional vectors

    Returns:
    A numpy array Y of shape (N, 3) where Y[i] is X[i] rotated by R

    Par: 3 lines
    Instructor: 1 line

    Hint:
    1) If v is a vector, then the matrix-vector product Rv rotates the vector
       by the matrix R.
    2) .T gives the transpose of a matrix
    """
    return None
```

2.3 What We Provide

For each problem, we provide:

Inputs: The arguments that are provided to the function

Returns: What you are supposed to return from the function

Par: How many lines of code it should take. If it takes more than this, there is probably a better way to solve it. **Except for t10, you should not use any explicit loops.**

Instructor: How many lines our solution takes

Hints: Functions and other tips you might find useful for this problem

Question 1.1 Put the terminal output in your pdf from:
`python run.py --allwarmups`
`python run.py --alltests`.

Input Naive Rounding Dithered

Figure 1: Results with 1 bits of brightness (two levels – off or on) per channel. With 3 channels, this leads to 2^3 possible colors. Naively rounding to the nearest value produces weird results. You'll produce the result on the right. Both use the same values (look carefully!) but use them differently.

3 Lights on a Budget – Quantizing and Dithering

The code and data for this are located in `dither/`. This contains starter code `dither.py`, an image gallery `gallery/`. Some of these images are very high resolution, so we are providing a copied that has been downsampled to be $\leq$ 200 pixels in `gallery200/`. We're also providing sample outputs for all algorithms in `bbb/`.

While modern computer screens are typically capable of showing 256 different intensities per color (leading to $256^3 = 16.7$ million possible color combinations!) this wasn't always the case. Many types of displays are only capable of showing a smaller number of light intensities. Similarly, some image formats cannot represent all 256^3 colors: GIFs, for instance, can only store 256 colors.

You'll start out with a full byte of data to work with and will be asked to represent the image with a smaller number of bits (typically 1 or 2 bits per pixel).

Input: As input the algorithm will get as input: (1) a $H \times W$ **floating point** image with brightness ranging from 0 to 1, and (2) a palette consisting of all the K allowed brightness settings (each floats in the range 0 to 1). For the curious, the word palette comes from painting.

Output: As output, each algorithm produce a $H \times W$ **uint8** image with brightness ranging from 0 to K-1. We'll call this a *quantized* image. You can take the palette and the quantized image and make a new image via ReconstructedImage[y,x] = Palette[QuantizedImage[y,x]]. Note that the array you get from numpy will get be indexed by y (or row) first and then by x (or column). The goal of the algorithm is to find a quantized image such that it is close to the reconstructed image. While this doesn't technically save us space if implemented naively, we could further gain savings by using $\log_2(K)$ bits rather than the 8 bits.

The Rest of the Homework: You'll build up to Floyd-Steinberg Dithering. You'll: (1) start with a really naive version; (2) do Floyd-Steinberg; (3) add a resize feature to the starter code; (4) handle color; (5) (optionally) handle funny stuff about how monitors display outputs.

You'll be able to call each implementation via a starter script `dither.py` that takes as arguments a source folder with images, a target image to put the results in, and the function to apply to each. For instance if there's a function `quantizeImageNaive`, you can call:

```
python dither.py gallery/ results/ quantizeImageNaive
```

and the folder results will contain the outputs of running quantizeImageNaive on each. There will also be a file `view.htm` that will show all the results in a table. The starter code contains a bunch of helper functions for you to use.

Important: Web browsers mess with images. Set your zoom to 100%. The images are outputted so that each pixel in the original image corresponds to a few pixels in the output image (i.e., they're upsampled to deliberately be blocky). Try opening `bbb/view.htm`. The outputs quantizeFloyd and quantizeFloydGamma should look like BBB. The quantizeNaive outputs should look bad.

3.1 The Naive Approach (3 questions)

The simplest way to make an image that's close is to just pick the closest value in the palette for each pixel.

First, fill in `quantize(v,palette)` in the starter code. This should return the **index** of the nearest value in the palette to the single value v. Note that we're making this general by letting it take a palette. For speed this would normally done by pre-selecting a palette where the nearest entry could be calculated fast. You can do this without a for-loop. Look at `np.argmin`. Indeed, the beauty of the Internet is that if you search for "numpy find index of smallest value", you'll likely find this on your own. In general, you should feel free to search for numpy documentation or for whether there are functions that will make your life easier.

Second, fill in `quantizeNaive(IF,palette)` in the starter code. This takes a floating point image of size HxW and a palette of values. Create a new uint8 matrix and use quantize() to find the index of the nearest pixel. Return the HxW uint8 image.

Finally, hit run. Call `python dither.py gallery200 results quantizeNaive` to see the results. Open up `view.htm`. You can sort of recognize the images, but this is not an aesthetically pleasing result.

Question 2.1 (1 sentence) If you apply this to the folder `gallery`, why might your code (that calls quantize) take a very long time?

Question 2.2 (1 sentence) Pause the program right after `algoFn` (the function for your dithering algorithm) gets called. Visualize the values in the image with `plt.imsave` or `plt.imshow`. Try the gray or gist_gray colormap in matplotlib. Do you notice anything odd compared to the original input?

Question 2.3 (3 pictures) Put a three results of inputs and outputs in your answer document. Use `aep.jpg` plus any two others you like.

3.2 Floyd-Steinberg (2 questions)

Naively quantizing the image to a palette doesn't work. The key is to spread out the error to neighboring pixels. So if pixel i,j is quantized to a value that's a little lower, you can have pixel (i,j+1) and (i+1,j) be brighter. Your job is next to implement `quantizeFloyd`, or Floyd-Steinberg Dithering. The pseudocode from Wikipedia (more or less) is below (updated to handle the fact that we're returning the index):

```
output = new HxW that indexes into the palette
for y in range(H): for x in range(W):
    oldValue = pixel[x][y]
    colorIndex = quantize(oldValue,palette)
    output[x][y] = colorIndex
    newValue = palette[colorIndex]
    error = oldValue-newValue
    pixel[x+1][y]   += error*7/16
    pixel[x-1][y+1] += error*3/16
    pixel[x][y+1]   += error*5/16
    pixel[x+1][y+1] += error*1/16
return Output
```

Beware 1! In general, you should be careful and think carefully with indices when working with images. Different programs, libraries, notations will make different assumptions about whether x or y come first, and whether the image is height x width or width x height. Sometimes the system won't even say which it expects! Here, the person who wrote up the code on the Wikipedia article says that you should access pixels as pixel[x][y]. In numpy, we'll refer to the pixel at a given row y and column x as pixel[y,x]. When you're not sure, you can often tell by giving the code it a *non-square* image and seeing where it breaks!

Beware 2! This algorithm has (literal) edge cases. Most image processing algorithms do! You typically won't be told what to do because typically these cases aren't defined. I usually take the laziest functional solution that preserves the intent, but does not try something fancy. When you try to be oversmart, you open yourself up to oversmart bugs.

Beware 3! The algorithm requires modifying the array you're given. When you get `IF` as an argument, you are getting a **reference/address/pointer**! If you modify that variables, the underlying data changes. Make a copy in a new variable via `IF.copy()`. It's generally not nice to tamper with data you're passed unless you've been explicitly told you can modify it or asked to. Try the algorithm with and without first making a copy.

Question 3.1 (1-2 sentences) Why does dithering work (in your own words)? Try stepping back from your computer screen or, if you wear glasses, take them off.

Question 3.2 (3 pictures) Run the results on `gallery200`. Put another three results in your document.

3.3 Resizing Images (0 questions)

We provided you with two folders of images, `gallery/` and `gallery200/`. The images in `gallery200` are way too small; the images in `gallery` are way too big! Fill in `resizeToSquare(I,maxDim)`. Use the opencv function `cv2.resize`.

You can now resize to your hearts content using the `--resizeto` flag.

3.4 Handling Color

You've written a version of dithering that handles grayscale images. Now you'll write one that handles color.

First, rewrite `quantize(v,palette)` so that it can handle both scalar v and vector v. If v is a n-dimensional vector it should return a set of n vector indices (i.e., for each element, what is the closest value in the palette). You can use a for loop, but remember that: (a) broadcasting can take a M-dimensional vector and N-dimensional vector and produce a MxN dimensional matrix; and (b) many functions have an axis argument.

Second, make sure that your version of `quantizeFloyd(IF,palette)` can handle images with multiple channels. You likely will not have to do anything. If IF is a $H \times W \times 3$ array, `IF[i,j]` refers to the 3D vector at the i,jth pixel (i.e., `[IF[i,j,0],IF[i,j,1],IF[i,j,2]]`. You can add and subtract that vector however you want.

Beware! When you get `v = IF[i,j]`, you are getting a **reference/address/pointer**! If you modify that variable, the underlying data changes! This can lead to hard to track down bugs. You may want to `.copy()` the pixel if you're going to modify it.

Question 4.1 (3 pictures) Generate any three results of your choosing or on some other image you'd like. Put them in your document.

Question 4.2 (optional) Pick your favorite result and put it in a folder by itself named `dither/mychoice/` with the result named `dither/mychoiceresult/` in your zip file. We'll have a vote among the class.

3.5 (Optional) Gamma Correction – Worth Reading If You Have Time But Optional to Do

If you look at your outputs from a distance (unless you crank up the number of bits added), you'll notice they're quite a bit brighter! This is a bit puzzling. As a sanity check, you can check the average light via `np.mean(reconstructed)` and `np.mean(original)`. They should be about the same.

The amount of light your monitor sends out isn't linearly related to the value of the image. In reality, if the image has a value $v \in [0, 1]$, the monitor actually shows something like v^γ for $\gamma = 2.4$ (for most values, except for some minus some technicalities near 0 – see sRGB on Wikipedia). This is because human perception isn't linearly related to light intensity and storing the data pre-exponent makes better use of a set of equally-spaced values. However, this messes with the algorithm's assumptions: suppose the algorithm reconstructs two pixels which are both 0.5 as a pixel with a 0 and one with a 1. The total amount of light that comes off the screen is $2 * 0.5^{2.4} \approx 0.379$ and $0 + 1^{2.4} = 1$. They're not the same! Oof.

The solution is to operate in linear space. You can convert between linear and sRGB via `linearToSRGB` and `SRGBToLinear`. First convert the image from sRGB to linear; whenever you want to quantize, convert the linear value back to SRGB and find the nearest sRGB value in the palette; when you compute the error, make sure to convert the new value back to linear.

You should feel free to use the outputs from this implementation for Question 3.2.

4 Data Interpretation and Making Your Own Visualization

All the code/data is located in `visualize/`.

We've come across a bunch of numpy files in a folder called `mysterydata/` and some starter code in `mystery_visualize.py`. We've been told it's four datasets from a satellite and a plasma colormap to show the data in. Your job is to figure out what this data is, in part by filling in a code stub that visualizes an arbitrary matrix using false colors. While we are giving everybody the same data they have to interpret, this is meant to model other debugging you will do later on in the course with your own debugging (plus demystifying one of the tools for looking at data).

First, open up the matrices using `np.load`. Poke around. Guess what the matrices mean. See if there are surprises. Try some of the techniques in Section A.4. You'll be implementing your own version of plt.imwrite, so maybe use that.

Second, fill in `colormapArray(X,colors)`. The goal is to make a false color image using the given $N \times 3$ matrix colors where the columns of colors are Red/Green/Blue. Given an input value v, the corresponding output pixel's color is the color at row $\approx (N-1) * (v - vmin)/(vmax - vmin)$ where vmin and vmax are the minimum and maximum non-NaN values in the array. Can you do this without looping over pixels? Try doing it first for one channel of output only. Then loop over channels and use `np.dstack`.

Beware! There are a bunch of edge cases in the equation for the color: it won't always return an integer between 0 and $N-1$. It will also definitely blow up under certain input conditions (also, watch the type).

Problem Solving Tip: You will often have to rewrite various components that exist in other libraries. It's often a good idea to start with something you know works and make sure that the data you are passing in is correct. Try some of the options in Sec A.4.

Question 5.1 (a few words): What do you think the images show? We're looking for the average person answer, not a dissertation.

Question 5.2 (a few words): What shape is the data (i.e., what is the height and width and number of channels)?

Question 5.3 (1 sentence or so): Mystery data 2 is hard to see. Can you adjust it so that you can see the details across the full image? Can you think of something?

Question 5.4 (some pictures): Put the visualization of one of the datasets in your document.

A What's an Image?

What's an image? It is a representation that encodes how much light exists at each place on a regular grid.

For better or for worse, images will come in all sorts of forms throughout this course and the code you will use will assume different formats or conventions. This will be a source of **great annoyance** throughout the course. I've been doing computer vision for over a decade and it's still an annoyance.

A.1 Shapes of Images

Images come in a variety of shapes. Depending on who's in charge, either the rows come first or the columns come first in terms of how it's laid out in memory, referenced in terms of array access, and in terms of how "size" is denoted. Once you add in colors, the number of options gets even worse! We'll usually refer to a grayscale image as a $h \times w$ matrix/array and a color (RGB) image as a $h \times w \times 3$ matrix/array.

A.2 Types of Images

The meaning of data depends on context and convention. The four bytes `0xF30F58C1` mean:

- $-1.13570952431 \times 10^{31}$ if you interpret it as a floating point number
- 4077869249 if you interpret it as an unsigned integer
- -217098047 if you interpret it as a signed integer (using two's complement)
- `addss xmm0,xmm1` if you interpret it as an instruction to an Intel x86-64 processor.
- ó<*Shift-in control code*>XA if you treat it as an a string and decode it with the Latin-1 encoding

But really, what does it actually mean? It fundamentally depends on who wrote it and why.

Images are no different and this is a source of lots of bugs. Sometimes your bugs will be that two pieces of code are completely correct, but they expect different formats. Let's look at a few common values/representations each pixel v can take and what you need to look out for when working with each:

- **An unsigned 8-bit integer** $\in \{0, 1, \ldots, 255\}$ where 0 and 255 are the minimum and maximum amounts of light. This is how things are commonly stored for images – humans can't spot small differences in light and so there's no value in storing past 8-bits of precision (although of course there do exist 16-bit integer images, for instance in medicine and astrophysics).

  Danger: Beware that math is integer math! Integers are annoying since they lose precision (e.g., $2/3 \rightarrow 0$) and overflow ($9 \times 40 \rightarrow 104$). Doing all the math in uint8 can be *much* faster than using floating point numbers; however, it's easy to write code that doesn't work. When you start out, convert *everything* to floating point (preferably double precision/`np.float64`) where there are fewer gotchas.

  Note: It may be helpful to think of this as just a representation where you're storing things that range from 0 to 1, and to get that value you need to divide by 255. It's often common for instance for people to store probabilities as integers from [0,255]. To get the correct value, you just divide by 255 (after converting to a float).

- **A floating point number** $\in [0, 255]$ where 0 and 255 are the minimum and maximum amounts of light. This is quite common when you want to do stuff like take the average of things without worrying about overflow. You can immediately convert back to the nearest uint8 integer and things will be fine.

  Danger: Beware that the maximum light is 255, not 1! This leads to two common issues: (1) some systems may think that the maximum value of the image is 1 and will clip any value bigger than 1 to be equal to 1. So you may have a pixel that has a value 42, and it'll be forced to 1. This results in an image that's almost all white. (2) some equations/formulas assume values fall between 0 and 1 and will screw up outside this range – e.g., squaring a positive number x reduces its value if $x \in [0, 1]$ but increases it if $x > 1$.

- **A floating point number** $\in [0, 1]$ where 0 and 1 are the minimum and maximum amounts of light. This is the most common way floating point images are represented since 255 is a totally arbitrary convention – there's no reason why you need 8 bits of precision – and a lot of things are better expressed in some sort of common scale. $[0, 1]$ is about as inoffensive as it gets.

  Danger: Beware that the maximum light is 1, not 255! If you convert this directly from a float to an unsigned integer, you'll end up with pixels that are 0 and 1 (which are the darkest and second-darkest for `uint8`). You'll end up with an image that's completely black!

- **An unsigned integer** $[0, 1, \ldots, K]$. This is less common, but can indicate:

  1. regions in an image (e.g., from a system that divides the image into a set of regions corresponding to objects) where all pixels that have the same value are part of the same region.
  2. categories in an image (e.g., from an object detector) where $v = 2$ might indicate that there's a horse at that pixel or $v = 10$ might indicate that there's a dog at that pixel.
  3. the index into a palette (like Paint by numbers) where $v = i$ might indicate that the pixel is colored with color i (e.g., $0 =$ blue, $1 =$ red).

  Danger: Beware that this normally isn't a real image! You can look at it, but think of it like a map that's been colored. Tanzania being colored purple on a globe doesn't mean that Tanzania is purple.

- **A floating point number** $\in [-\infty, \infty]$ that represents some physical quantity. This could be the distance from the camera plane, the velocity at a location, or anything else.

  Danger: keep track of units. You probably won't encounter this in the course as an input. But you will encounter this as an intermediate step of a calculation. It's important to keep track of what the image means. Otherwise you have something that looks interesting but doesn't have meaning.

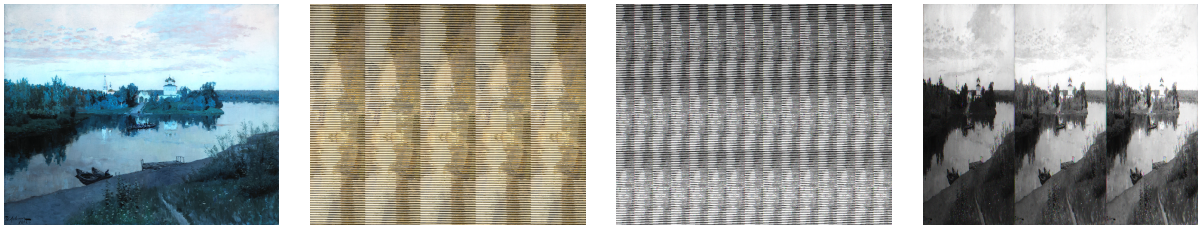
A.3 Visually Identifying Bugs

Many bugs can be identified quickly entirely visually in the end product. Fortunately, we have a lot of experience writing code with bugs. We've made a montage of common image bugs in Fig 2. It's rare for you to have exactly one of these bugs. But the type of behavior suggests different bugs.

1. **I can't see anything:** If you expect an image with something, but it's just black or just white, this is often a type issue, a NaN somewhere, an image that has no range (i.e., everything is indeed the same value), or an image with too big of a range (i.e., most values are $\sim 10^1$ and one value is $\sim 10^{20}$).

Original Image I | Grayscaled (by averaging colors) | Sum(I) in uint8 | (I*2) with [0,1] as limits

(I*3)/3 in uint8 | (I/64)*64 in uint8 | I/2 in uint8 | I*2 in uint8



BGR and RGB flipped | Dimensions 0/1 flipped | Dimensions 0/2 flipped | Dimensions 1/2 flipped

Figure 2: **Visually identifying bugs.** Particular types of bugs have particular visual appearances. Here are a few common issues. Some are due to doing operations on uint8 that don't act like normal numbers (indicated by a colors being off). Some are do to indexing the image incorrectly (e.g., treating rows and columns incorrectly).

2. **Colors gone wrong:** If general shape looks right but the colors are wrong, it's likely that your math is doing something that doesn't quite work out correctly. Common causes include things like:

   (a) UINT8 math: Remember that integers on computers are modular. For instance UINT8 192 (light gray) + 128 (medium gray) = 64 (dark gray).

   This has issues especially in colors – [192,64,64] plus [32,128,192] = [224,192,0] or red plus blue equals not purple! You might do something like take the average of a blue and red (i.e., ([192,64,64]+[32,128,192])/2) and end up with a dark yellow like this instead of something vaguely magenta-y like this.

   You prevent this by having enough precision to do the math you need. I recommend using a float (preferably double-precision) for all your math for when you start out. You can also use 16-bit uints. But the speedup compared to doubles (under an order of magnitude) is not worth the increased chance of edge cases for when you start out. For the curious: when your computer does these averages (e.g., to blend two images) it may be doing it via a CPU instruction (pavgb) that temporarily uses 9 bits!

   (b) Some formula goes from 0 to 2 **instead of** from 0 to 1. If you multiply the image by 2, anything that is really intensity 127.5/0.5 will really be 255/1!

   (c) A formula that's missing an important constant. For instance, if you compute an average by summing but forget to divide by a constant, you'll end up with something funny.

3. **Right type of color, but shapes gone wrong or points in the wrong position:** If the colors on average look OK, but the shape looks wrong (e.g., zig-zag patterns like an old TV, the image duplicated but squished), the shape of your image is being interpreted incorrectly. You can have this same issue if you're trying to point to locations on the image (e.g., where eyes, locations of interest, or corners of a box are).

If you don't have an image that you can pass in, generate fake data where you can identify what the answer should be. A common trick is to use `meshgrid` to generate images where the columns ascend and rows descend. Before you throw it in, identify – what things do you expect?

A.4 Visual Debugging

You'll probably write some buggy code! Don't panic. Here are a few things you should do:

1. **Design defensively beforehand.** Write small functions that do exactly one thing. Save results along the way in a format that is easy to understand. Load your results to make sure they load correctly.

2. **Assume every line of code could have a bug.** The bug is almost always something boring. In fact, because you're likely to look at the places for an *interesting* bug, any long-lasting annoying bug will probably not be there. It's often useful to write a separate script from scratch that loads in the data. Writing a second time is faster a little slower but prevents re-introduction of typos that are correct code but which are wrong.

3. **Use a debugger.** If you use an IDE, use breakpoints; if you don't, try pdb. If you print debug, you can't ask follow-up questions without re-running the program. This increases the amount of time you spend waiting for that code to finish.

4. **Check what's in the data!**

   OK, so you've got the program stopped and you have a variable named `X` that you think is suspect. What should you do?

   (a) Check the data type (`X.dtype`)! You might have set the data type to one type, but you might have gotten the data from somewhere else. A lot of stuff will just screw up due to this. If it's not the type you expect, be skeptical and try to find out why.

   (b) Check for not a numbers (NaNs) (`np.sum(np.isnan(X))` or `np.isnan(X).sum()`). NaNs emerge out of things like division by zero in floating point arithmetic, square roots of negative numbers. They get *everywhere* because NaN $\circ$ x = NaN for any x and (basically) operator $\circ$.

   (c) Check the range: `X.min()` or `np.nanmin(X)` and `X.max()` or `np.nanmax(X)`. Do they make sense?

   (d) Check the average value: `X.mean()`. Does this makes sense?

   (e) Check the fraction of items above/below a threshold. If the value should almost always be positive, try `np.mean(X>0)`.

5. **Plot!** Your visual cortex is large and quite well-developed. You should visualize things even if we don't ask you to. This maximizes the chance you catch the bug earlier rather than later. Here are a few options:

   (a) Matplotlib imsave: `plt.imsave(filename,matrix)`
   This makes a false color image and saves it. This automatically scales things so that one end of the color scheme is the minimum and the other end is the maximum. Set `vmin` and `vmax` if you want different behavior. You'll implement a version of this later in the homework. You can select a colormap from the documentation on colormaps. The default colormap in matplotlib is called *viridis* and is pretty good.

   (b) Matplotlib imshow: `plt.imshow(X)`
   This shows the result as a figure.

   (c) Rescaling things to [0,255] to make an image that you can save.
   `((X-X.min()) * 255 / (X.max()-X.min())).astype(np.uint8)`
   This is grayscale so considerably less fun but also works.

   (d) Re-normalizing. If you can't see anything, play with the contrast. Do `X**0.5` to make larger values smaller (for $X > 1$).

6. **Develop an intuition for things to be skeptical of.** This takes a while. I'll give a few:

   (a) Values are rare at the min/max value for natural signals. It is rare for a normal image to have cyan in the image. It's rare for lots of the values to be close to 1.0 for floats or 255 for uint8.

   (b) NaNs are danger. Treat an *unexplained* NaN as a smoke detector beeping in your code.

   (c) Contrastless visualization. Common causes: You have a NaN; your range is 0; you have a single large value in one pixel (e.g., all pixels are in the range 0-3; one pixel is 9001).

   (d) Stuff that doesn't line up. If you have two pieces of data and they *should* line up but don't, there's probably an indexing thing that will get you later on.