CS 376 Computer Vision:
Homework 1 – Numbers and Images

Instructions

- This homework is **due at 23:59 on Wednesday September 10, 2025**.
- The submission includes two parts to Gradescope:
  1. 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.
     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.
     We have marked things you should provide as a write-up in blue.
  2. A `zip` file including all your code.
     This should contain a single directory which has the same name as your EID. If your EID is ab12345, then the zip file should contain a single folder:
     `ab12345/code/numpy/run.py` and `ab12345/code/dither/dither.py`.
     We have marked things that involve submitting code in red.
     Please check the `README.md` file of the `auto_grade` folder for a more comprehensive list of files expected for submission.

1 Overview

In this assignment, you'll work through three tasks that help set you up for success in the class as well as a short assignment involving playing with color. 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!

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.

1.1 Collaboration on HW1

As a one time, only for HW1 policy, we allow any sort of collaboration on the numpy exercises. Specifically, you can collaborate in any way, shape or form on Section 2

(Numpy). This will not be the case for the rest of the homework as well as the other homeworks. However, the point of Section 2 is to learn numpy and we would like to encourage collaboration.

1.2 The Actual Assignment

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

- Numpy (Section 2 – folder `numpy/`)
- Data visualization (Section 3 – folder `visualize/`)
- Image dithering (Section 4 – folder `dither/`)
- Looking at color (Section 5)

Here's our 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 designed to produce common bugs, issues, and challenges that you will likely run into the course.
- Finally, conclude with Section 5.

Python Environment

We are using Python 3.12. We will make use of the following packages extensively in this course:

- Numpy (https://numpy.org/doc/stable/user/quickstart.html)
- Matplotlib (https://matplotlib.org/2.0.2/users/pyplot_tutorial.html)
- OpenCV (https://opencv.org/)

If you are using the CS machines to run your code, you can access a working python environment:

```
source /lusr/opt/miniconda/bin/activate cs376-cpu
```

This will make sure you are using the right version of each dependency, so that you don't need to worry about using different package versions. We will also run your code using this environment.

2 Numpy Intro (40 points)

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 `warmups.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`.

Note: All the `python` below refer to `python3`.

Report 1.1 (the only question): Fill in the code stubs in `tests.py` and `warmups.py`. Put the terminal output in your pdf from:

```
python run.py --allwarmups
python run.py --alltests
```

Do I have to get every question right? We give partial credit: each warmup exercise is worth 2% of the total grade for this question and each test is worth 3% of the total grade for this question.

2.1 How the Tests Work

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

If you are checking all the warmup problems (or test problems), the perfect result will be:

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

2.2 Warmup Problems

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

2.3 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, although you may want to first figure out a slow for-loop solution to make sure you know what the right computation is, before changing the for-loop solution to a non for-loop solution. The one exception to the no-loop rule is t10 (although this 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
```

2.4 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. We don't grade on this, but if it takes more lines 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. Can you do better?
- *Hints:* Functions and other tips you might find useful for this problem.

2.5 Walkthroughs and Hints

Test 8: If you get the axes wrong, numpy will do its best to make sure that the computations go through but the answer will be wrong. If your mean variable is 1×1, you may find yourself with a matrix where the full matrix has mean zero. If your standard deviation variable is $1 \times M$, you may find each column has standard deviation one.

Test 9: This sort of functional form appears throughout data-processing. This is primarily an exercise in writing out code with multiple nested levels of calculations. Write each part of the expression one line at a time, checking the sizes at each step.

Test 10: This is an exercise in handling weird data formats. You may want to do this with for loops first.

1. First, make a loop that calculates the vector `C[i]` that is the centroid of the data in `Xs[i]`. To figure out the meaning of things, there is no shame in trying operations and seeing which produce the right shape. Here we have to specify that the centroid is M-dimensions to point out how we want `Xs[i]` interpreted. The centroid (or average of the vectors) has to be calculated with the average going down the columns (i.e., rows are individual vectors and columns are dimensions).
2. Allocate a matrix that can store all the pairwise distances. Then, double for loop over the centroids i and j and produce the distances.

Test 11: You are given a set of vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ where each vector $\mathbf{x}_i \in \mathbb{R}^M$. These vectors are stacked together to create a matrix $\mathbf{X} \in \mathbb{R}^{N \times M}$. Your goal is to create a matrix $\mathbf{D} \in \mathbb{R}^{N \times N}$ such that $\mathbf{D}_{i,j} = \|\mathbf{x}_i - \mathbf{x}_j\|$. Note that $\|\mathbf{x}_i - \mathbf{x}_j\|$ is the L2-norm or the Euclidean length of the vector. The useful identity you are given is that:

$$\|\mathbf{x} - \mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\mathbf{x}^T\mathbf{y}.$$

This is true for any pair of vectors $\mathbf{x}$ and $\mathbf{y}$ and can be used to calculate the distance quickly.

At each step, your goal is to replace slow but correct code with fast and correct code. If the code breaks at a particular step, you know where the bug is.

1. First, write a correct but slow solution that uses two for loops. In the inner body, you should plug in the given identity to make $\mathbf{D}_{i,j} = \|\mathbf{x}_i\|^2 + \|\mathbf{x}_j\|^2 - 2\mathbf{x}_i^T\mathbf{x}_j$. Do this in three separate terms.

2. Next, write a version that first computes a matrix that contains all the dot products, or $\mathbf{P} \in \mathbb{R}^{N \times N}$ such that $\mathbf{P}_{i,j} = \mathbf{x}_i^T \mathbf{x}_j$. This can be done in a single matrix-matrix operation. You can then calculate the distance by doing:

$$\mathbf{D}_{i,j} = \|\mathbf{x}_i\|^2 + \|\mathbf{x}_j\|^2 - 2\mathbf{P}_{i,j}.$$

3. Finally, calculate a matrix containing the norms, or a $\mathbf{N} \in \mathbb{R}^{N \times N}$ such that $\mathbf{N}_{i,j} = \|\mathbf{x}_i\|^2 + \|\mathbf{x}_j\|^2$. You can do this in two stages: first, calculate the squared norm of each row of $\mathbf{X}$ by summing the squared values across the row. Suppose $\mathbf{S}$ is this array (i.e., $\mathbf{S}_i = \|\mathbf{x}_i\|^2$ and $\mathbf{S}_i \in \mathbb{R}^N$), but be careful that you look at the *shape* that you get as output. If you compute $\mathbf{S} + \mathbf{S}^T$, you should get $\mathbf{N}$. Now you can calculate the distance inside the double for loop as $\mathbf{D}_{i,j} = \mathbf{N}_{i,j} - 2\mathbf{P}_{i,j}$.

4. The double for loop is now not very useful. You can just add/scale the two arrays together element-wise.

Test 18: Here you draw a circle by finding all entries in a matrix that are within r cells of a row y and column x. This is not a particularly intellectually stimulating exercise, but it is practice in writing (and debugging) code that reasons about rows and columns.

1. First, write a correct but slow solution that uses two for loops. In the inner body, plug in the correct test for the given i, j. Make sure the test passes; be careful about rows and columns.

2. Look at the documentation for `np.meshgrid` briefly. Then call it with `np.arange(3)` and `np.arange(5)` as arguments. See if you can create two arrays such that `IndexI[i,j]` $= i$ and `IndexJ[i,j]` $= j$.

3. Replace your test that uses two for loops with something that just uses `IndexI` and `IndexJ`.

3 Data Interpretation and Making Your Own Visualization (10 points total)

Throughout the course, a lot of the data you have access to will be in the form of an image. These won't be stored and saved in the same format that you're used to when interacting with ordinary images, such as off your cell phone: sometimes they'll have negative values, really really big values, or invalid values. If you can look at images quickly, then you'll find bugs quicker. If you **only** print debug, you'll have a bad time. To teach you about interpreting things, we've got a bunch of mystery data[1] that we'll analyze together. You'll write a brief version of the important `imsave` function for visualizing.

Let's load some of this mysterious data.

```
>>> import numpy as np
>>> X = np.load("mysterydata/mysterydata.npy")
>>> X
array([[[0, 0, 0, ..., 0, 0, 1],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       ... (some more zeros) ...

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 1]]], shape=(512, 512, 9), dtype=uint32)
```

Looks like it's a bunch of **zeros**. Nothing to see here folks! For better or worse: Python only shows the sides of an array when printing it and the sides of images that have gone through some processing tend to not be representative.

After you print it out, you should always look at the **shape** of the array and the **data type**.

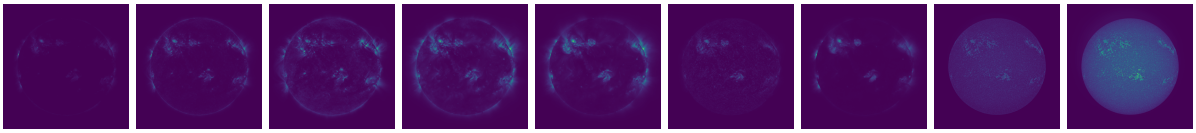
```
>>> X.shape
(512, 512, 9)
>>> X.dtype
dtype('uint32')
```

The shape and datatype are really important. If something is an unexpected shape, that's **really bad**. This is similar to doing a physics problem where you're trying to measure the speed of light (which should be in m/s), and getting a result that is in gauss. The computer may happily chug along and produce a number, but if the units or shape are wrong, the answer is almost certainly

[1] For the curious, this is data that originates from Solar Dynamics Observatory, and in particular the AIA instrument. These are, to a first approximation, photos of the sun in wavelengths that in the ultraviolet and extreme UV, and which primarily form in the upper part of the Sun's atmosphere.

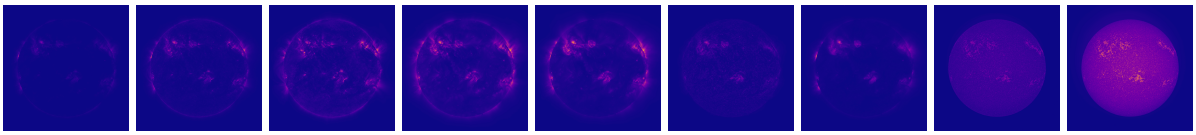
wrong. The datatype is also really important, because data type conversion can be lossy: if you have values between 0 and 1, and you convert to an integer format, you'll end up with 0s and 1s.

In this particular case, the data has height 512 (the first dimension), width 512 (the second), and 9 channels (the last). These order of dimensions here is a *convention*, meaning that there are multiple equivalent options. For instance, in other conventions, the channel is the first dimension. Generally, you'll be given data that is HxWxC and we'll try to be clear about how data is stored. If later on, you're given some other piece of data, figuring out the convention is a bit like rotating a picture until it looks right: figure out what you expect, and flip things until it looks like you'd expect.



vis_0.png vis_1.png vis_2.png vis_3.png vis_4.png vis_5.png vis_6.png vis_7.png vis_8.png

Figure 1: The Mystery Data



vis_0.png vis_1.png vis_2.png vis_3.png vis_4.png vis_5.png vis_6.png vis_7.png vis_8.png

Figure 2: The Mystery Data, Visualized with the Plasma Colormap

If you've got an image, after you print it, you probably want to *visualize* the output.

We don't see in 9 color channels, and so you can't look at them all at once. If you've got a bug, one of the most important things to do is to look at the image. You can look at either the first channel or all of the channels as individual images.

```
>>> import matplotlib.pyplot as plt
>>> plt.imsave("vis.png", X[:,:,0])
```

You can see what the output looks like in Figure 1. This is a false color image. You can read about these in Section A.3, but the short version is that the given image, you find the minimum value (`vmin`) and the maximum value (`vmax`) and assign colors based on where each pixel's value falls between those. These colors look like: Low  High. `plt.imsave` finds `vmin` and `vmax` for you on its own by computing the minimum and maximum of the array.

If you'd like to look at all 9 channels, save 9 images:

```
>>> for i in range(9):
...     plt.imsave("vis_%d.png" % i, X[:,:,i])
```

If you're inside a normal python interpreter, you can do a for loop. If you're inside a debugger, it's sometimes hard to do a for loop. You can use side effects to do a for loop to save the data.

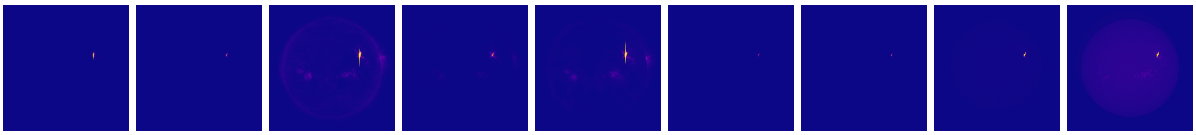
```
(pdb) [plt.imsave("vis_%d.png" % i, X[:,:,i]) for i in range(9)]
[None, None, None, None, None, None, None, None, None]
```

The list of Nones is just the return value of `plt.imsave`, which saves stuff to disk and returns a None.

If you'd like to change the colormap, you can specify this with `cmap`. For instance:

```
>>> for i in range(9):
...     plt.imsave("vis_%d.png" % i, X[:,:,i], cmap='plasma')
```

produces the outputs in Figure 2. These use the plasma colormap which looks like: Low  High.



vis2_0.png vis2_1.png vis2_2.png vis2_3.png vis2_4.png vis2_5.png vis2_6.png vis2_7.png vis2_8.png

Figure 3: The Mystery Data #2.

3.1 Images with wide ranges of values

Report 2.1 (pictures, 2 points) Try loading `mysterydata2.npy` and visualizing it. You should get something like Figure 3. It's hard to see stuff because one spot is *really* bright. In this case, it's because there's a solar flare that's producing immense amounts of light. A common trick for making things easier to see is to apply a nonlinear correction. Here are a few options:

$$p^\gamma \text{ with } \gamma \in [0,1] \quad \text{or} \quad \log(p) \quad \text{or} \quad \log(1+p) \tag{1}$$

where the last one can be done with `np.log1p`. Apply a nonlinear correction to the second mystery data and visualize it. **Put two of the channels (i.e., `X[:,:,i]`) as images into the report. You can stick them side-by-side.**

3.2 Images that Screw up and Knowing Your Limits

vis3_0.png vis3_1.png vis3_2.png vis3_3.png vis3_4.png vis3_5.png vis3_6.png vis3_7.png vis3_8.png

Figure 4: The Mystery Data #3, Visualized!

Let's try this again, using one of the other data.

```
>>> X = np.load("mysterydata/mysterydata3.npy")
>>> for i in range(9):
...     plt.imsave("vis3_%d.png" % i, X[:,:,i])
```

The results are shown in Figure 4. They're all white. What's going on?

If you've got an uncooperative piece of data that won't visualize or produces buggy results, it's worth checking to see if all the values are reasonable. One option that'll cover all your cases is `np.isfinite`, which is False for values that are NaN (not a number) or $\pm\infty$ and True otherwise. If you then take the mean over the array, you get the fraction of entries that are normal-ish. If the mean value is *anything* other than 1, you may be in trouble. Here:

```
>>> np.mean(np.isfinite(X))
0.682482825851997
```

Alternatively, this also works:

```
>>> np.sum(~np.isfinite(X))
749117
```

Other things to check are `np.isnan` (which returns True for NaNs) and `np.isinf` (which returns True for infinite values). Even a single NaN in your data is a problem: any number that touches another NaN turns into a NaN. The totally white values happen because `plt.imsave` tries to find `vmin/vmax`, and the minimum of a value and a NaN is a NaN. The resulting color is a NaN as well. If you've got NaNs in your data, many functions you may want to use (e.g., mean, sum, min, max, median, quantile) have a nan version.

Report 2.2 (some pictures, 2 points) Fix the images by determining the right range (use `np.nanmin`, `np.nanmax`) and then pass arguments into `plt.imsave` to set the range for the visualization. To figure out what arguments to set, look at the documentation for `plt.imsave`. Put two images from mystery data 3 in the report.

3.3 Rolling Your Own `plt.imsave`

You'll make your own `plt.imsave` by filling in `colormapArray`. Here's how the false color image works: You're given a $H \times W$ matrix X and a colormap matrix C that is $N \times 3$ where each row is a color red/green/blue. You produce a $H \times W \times 3$ matrix O. Each scalar-valued pixel $X[i, j]$ gets converted into red/green/blue values for $O[i, j, :]$ following this rule: if the pixel has value v, the corresponding output color comes from a row determined by (approximately)

$$(N-1)\frac{(v - v_{\min})}{(v_{\max} - v_{\min})}. \tag{2}$$

However, you'll have to be very careful – this precise equation won't always work. As an exercise – can you spot something that might happen if the values v are not in a nice range?

Coding 2.1 (3 points): Fill in `colormapArray`. To test, you'll have to write some calling code in the main part. You can use either `plt.imsave` or `cv2.imwrite` to save the image to a file.

Report 2.3 (3 points): Visualize `mysterydata4.npy` using your system without it crashing and put all nine images into your report. You may have to make a design decision about what to do about results that are undefined. If the results are undefined, then any option that seems reasonable is fine. Your colormap should look similar to Figure 2. If the colors look inverted, see Beware 3!

Beware 1! 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).

Beware 2! You're asked by the code to return a $H \times W$ `uint8` image. There are a lot of shortcuts/implied sizes and shapes in computer vision notation – since this is a $H \times W$ *color image*, it should have 3 channels (i.e., be $H \times W \times 3$). Since it's `uint8`, you should make the image go from 0 to 255 before returning it (otherwise everything gets clipped to 0 and 1, which correspond to the two lowest brightness settings). Like all other jargon, this is annoying until it is learned; after it is learned, it is then useful.

Beware 3! If you choose to save the results using `opencv`, you may have blue and red flipped – opencv assumes blue is first and the rest of the world assumes red is first. You can identify this by the fact that the columns of the are defined as Red/Green/Blue and there is a lot of blue and not much red in the lowest entry.

4 Lights on a Budget – Quantizing and Dithering (30 points)

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 copy that has been downsampled to be ≤ 200 pixels in `gallery200/`. We're also providing sample outputs for all algorithms in `gdc/`.

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: 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 produces 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 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 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: There are two caveats for verifying your results.

First, 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 `gdc/view.htm`. The outputs `quantizeFloyd` and `quantizeFloydGamma` should look like GDC. The outputs `quantizeNaive` outputs should look bad.

Second, due to slight variations in implementations, your outputs may not look bit-wise identical to these outputs. You can check your results with what I call the "look away from the screen" test.

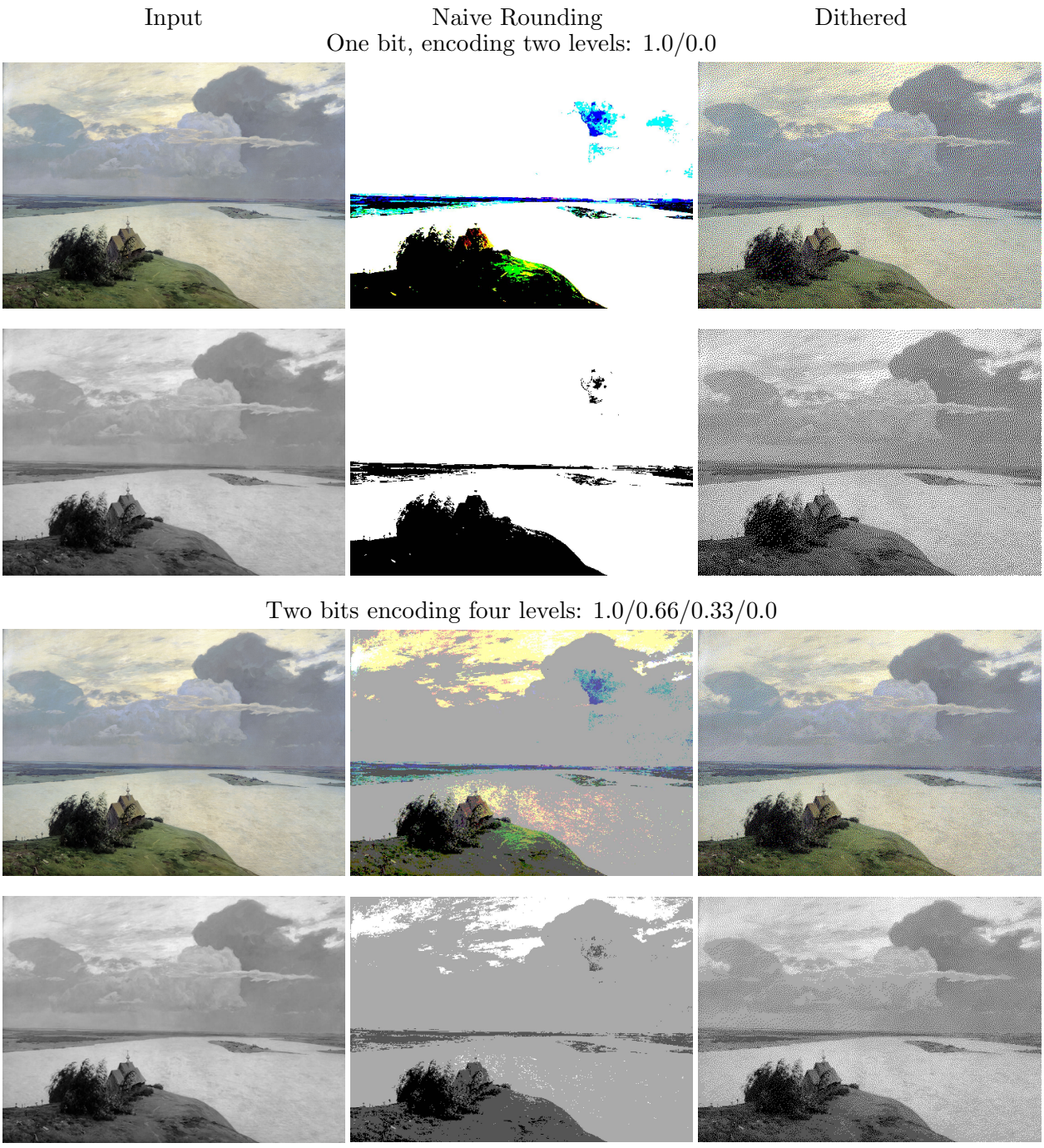


Figure 5: (top two rows) Results with one bit of brightness (two levels – off or on) per channel. With 3 channels, this leads to 2^3 possible colors. (bottom two rows) Results with two bits of brightness per channel. With 3 channels, this leads to 4^3 possible colors. In both cases, naively rounding 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.

To do this, load up your output and our output side by side. Cover your output with your hand; and look at our output. Now, look at something else and move your hand over your output. Look

at your output for under a second. Does it look the same? If it's hard to tell and they seem to be the same, you're fine.

4.1 The Naive Approach (2 Coding, 3 Report Questions, 8 points)

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

Coding 3.1 (your code will be evaluated via its results; we need it to give partial credit): 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.

Coding 3.2: Second, fill in `quantizeNaive(IF, palette)` in the starter code.

This takes a floating point image of size $H \times W$ and a palette of values. Create a new `uint8` matrix and use `quantize()` to find the index of the nearest pixel. Return the $H \times W$ `uint8` image containing the palette indices (not values!). Once you've done this, you can 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. If you change `--numbits` you can control the size of the palette.

Beware 1! At this stage, many people naturally want to do something like `output = IF`, which gives you an array that's as big as the input. Keep in mind that when you get `IF` as an argument, you are getting a *reference/address/pointer*! If you modify that variable, the underlying data changes. Allocate a new matrix via `output = np.zeros(IF.shape, dtype=np.uint8)`. Otherwise, this is like asking to copy your friends' notes, and then returning them with doodles all over them.

Report 3.1 (1 sentence, 2 points): If you apply this to the folder `gallery`, why might your code (that calls `quantize`) take a very long time?

Report 3.2 (1 sentence, 2 points): 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`. These produce false color images (see Section A.3). The default value colormap goes from Low  High. Do low intensity values correspond to low palette values? Explain what's going on. You may have to look through the code you're given (a good habit to get into).

Report 3.3 (2 pictures, 4 points): Put two results of inputs and outputs in your answer document. Use `aep.jpg` plus any other one that you like. While you can play with `--num_bits` to get a sense of how things work, you should have `--num_bits` set to 1 for the output.

4.2 Floyd-Steinberg (1 Coding, 2 Report Questions, 15 points)

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 darker, 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):

```
#... some calling code up here ..
#you'll have to set up pixel
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)
        #See Beware 1! re: rows/columns
        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
```

This is pseudocode; it will be provided to you throughout the course (and indeed your career as a programmer), so you'll have to figure out how to translate pseudocode.

Coding 3.3 (your code will be evaluated by its results; we need it to give partial credit): Implement Floyd-Steinberg Dithering in `quantizeFloyd()`.

Beware 1! In general, you should be careful with indices when working with images. Different programs, libraries, and notations will make different assumptions about whether x or y come first, and whether the image is height $\times$ width or width $\times$ 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, like most image processing algorithms, has literal edge cases. 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. Again, when you get `IF` as an argument, you are getting a *reference/address/pointer*! Modifying this data modifies the original data. 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. See what happens when the starter code tries to save the image it gave you.

Report 3.4 (1-2 sentences, 3 points): In your own words, why does dithering (the general concept) work? Try stepping back from your computer screen or, if you wear glasses, take them off.

Report 3.5 (3 pictures, 12 points): Run the results on `gallery200`. Put three results in your document, including `aep.jpg`. Don't adjust `--num_bits` and use the defaults.

4.3 Resizing Images (1 Coding, 2 points)

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! Giving you the images in all sizes would be too big, so it would be ideal if we could resize images to some size we decide when we run the program.

Coding 3.4 (2 points): Fill in `resizeToSquare(I, maxDim)`. If the input image is smaller than `maxDim` on both sides, leave it alone; if it is bigger, resize the image to fit inside a `maxDim` by `maxDim` square **while keeping the aspect ratio the same** (i.e., the image should not stretch). Use the opencv function `cv2.resize`. As is always the case in the course, you can look up any documentation you'd like for this function. You can now resize to your hearts content using the `--resizeto` flag.

4.4 Handling Color (2 Coding, 1 Report Questions, 5 points)

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

Coding 3.5: 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 $M \times N$ dimensional matrix; and (b) many functions have an axis argument. If you are given a vector, don't overwrite individual elements of it either!

Coding 3.6: Second, make sure that your version of `quantizeFloyd(IF, palette)` can handle images with multiple channels. You may 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.

Report 3.6 (3 pictures, 5 points): Generate any three results of your choosing. This can be on the images we provide or on some other image you'd like. Put them in your document.

Report 3.7 (optional – 2 extra points): Pick your favorite result and put the image saved as `code/dither/mychoice.jpg` in your zip file. We'll have a vote among the class for their favorite.

4.5 (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 used), 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 your chosen result.

5 Colorspaces (20 points)

The same color may look different under different lighting conditions. Images `rubik/indoor.png` and `rubik/outdoor.png` are two photos of a same Rubik's cube under different illuminances.[2]

Coding 4.1: (three pictures, 2.5 pts) Load the images and plot their R, G, B channels separately as grayscale images using `plt.imshow()`.

Coding 4.2: (three pictures, 2.5 pts) Then convert them into LAB color space using `cv2.cvtColor` and plot the three channels again.

Report 4.1: (5 pts) Include the LAB color space plots in your report. Which color space (RGB vs. LAB) better separates the illuminance (i.e., total amount of light) change from other factors such as hue? Why?

DIY/Coding 4.3: (10 pts) Choose two different lighting conditions and take two photos of a non-specular object. Try to make the same color look as different as possible (a large distance on AB plane in LAB space). Below in Figure 3 is an example of two photos of the same piece of paper under two different lighting conditions.

Figure 6: The same piece of paper under different lighting conditions.

Submit the following in the same folder:

- The two images with names `im1.jpg` and `im2.jpg`, both cropped and scaled to 256×256. You can use python and opencv to resize; you can also crop. On many systems, you can use a program called imagemagick.
- Also submit a file `info.txt` that contains only two lines: Line 1 contains four integers x_1, y_1, x_2, y_2 where we will take a 32×32 patch around the coordinate on each image and compare colors. (You can use `plt.imshow()` and `plt.show()` to bring up a window where you can select pixel with coordinates.) This coordinate should be somewhere on your object in the image. Line 2 is a description of the lighting conditions that you choose.

[2]The images are taken from this blog post: https://www.learnopencv.com/color-spaces-in-opencv-cpp-python/

Your `info.txt` should look something like:

```
92 101 82 110
The first image was taken in the basement under a light. The second was
    taken by a window with natural light.
```

Since the sense of color difference is subjective, we will display all images and patches on a webpage. Every student can vote for their favorite pair of images that illustrates color difference on Ed. The winner will get **Extra Credits** (2 pts).

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.

Here are three fairly common issues with image shapes:

- In a lot of material, sizes will be *implied*: a $H \times W$ color image is actually a $H \times W \times 3$ array.
- The **order** of colors in a color image isn't always standardized. Most libraries assume that the colors are stored in red, green, and blue (i.e., $I[:, :, 0]$ is red) but for historical reasons, OpenCV stores things in blue, green, red order.
- Some images have a fourth channel, called *alpha*. This represents opacity. An alpha of 0 corresponds to transparent, and an alpha of 1 (or maximum value) corresponds to opaque. Some image formats store this and some libraries return it. If you don't want it, you can usually just ignore the channel.

A.2 Values 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.
- *6<Shift-in control code>XA* if you treat it as 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. Numpy helps a lot by trying to nudge you to do floats, but depending on what precise operations you do, you can find yourself still in integer land.

  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** $\in [0, 1]$. 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.

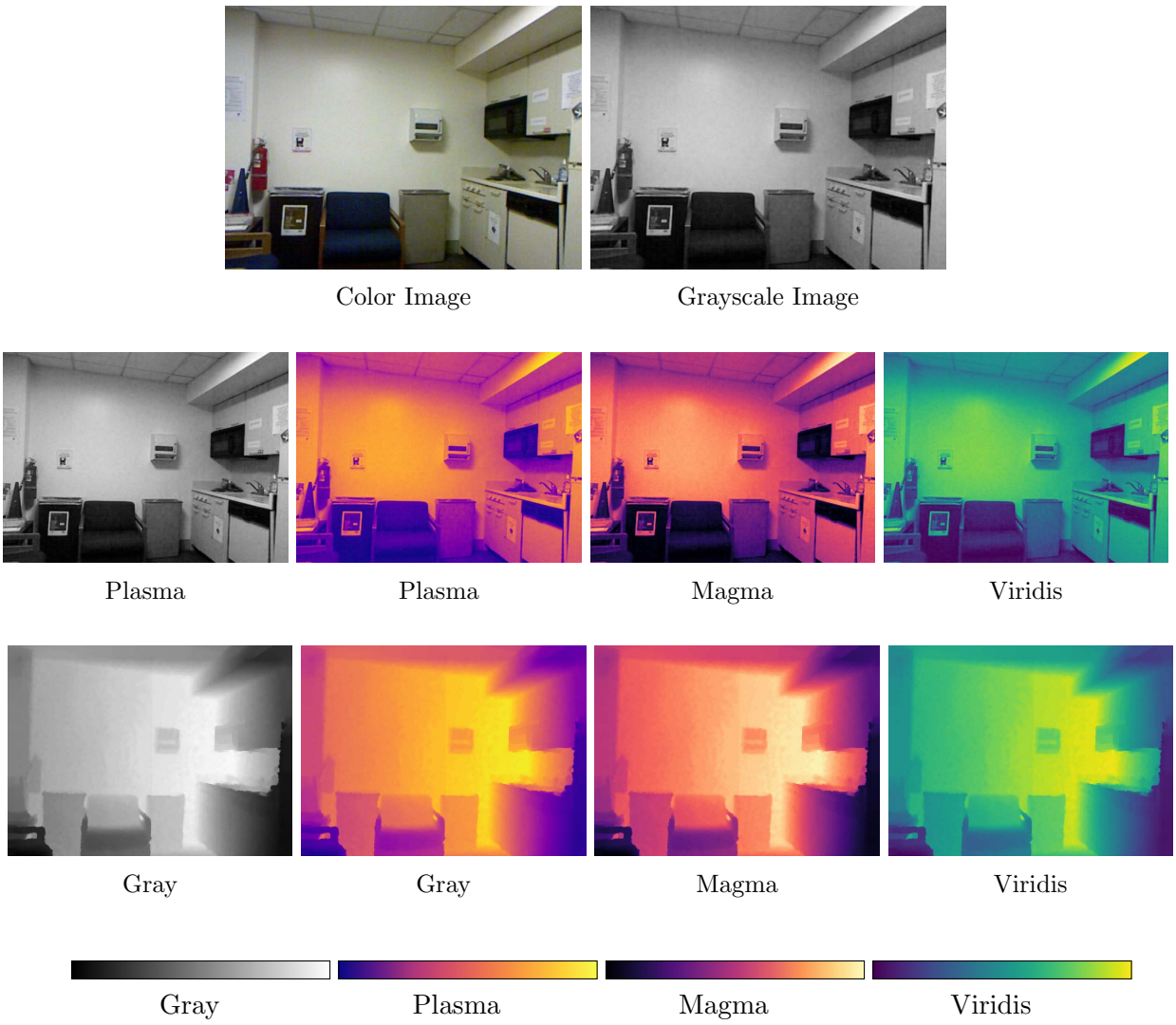


Figure 7: Colormapping a grayscale image and its corresponding depth (distance to the camera along the viewing axis) as measured by a ranging sensor. Both are shown with four colormaps that matplotlib provides (viridis is the default option).

A.3 False Color/Colormapped Images

Many of the images you'll deal with in the course are *false color* images. These are ways of showing some data that is like an image but isn't an image for which there's a natural way of showing things

(e.g., the amount of brightness). You can think of it as a way of converting some arbitrary image with arbitrary type into a standard RGB image.

You've seen many of these without thinking about it. When people visualize population density in the US, you can't actually "see" 40 people per square kilometer. These are converted into visualizations where different values are colored differently: maybe areas with few people are drawn as white, and areas with many people are drawn as red. The same idea applies to looking at "images" from Magnetic Resonance Imaging (MRI) machines, depth from a camera, or magnetic fields on the Sun. You may encounter this while trying to visually debug a program since many variables inside the code you write are actually images.

The two ingredients to a false color image are: a piece of imaging data to visualize, and a colormap. The image can be anything so long as there is a single value. The colormap is a mapping from every number.

between 0 and 1 to a color. In Figure 7, for instance the plasma map slowly changes from purple to yellow. The number 0 corresponds to purple; the number 1 corresponds to yellow, and 0.68 is some shade of orange.

Images don't naturally range from 0 to 1 – the depths of objects in Figure 7 range from 1.83 to 3.62m – so there is one more step needed – rescaling the image values. Let's assume the image data is all real-valued and ranges from v_{min} to v_{max}. When the image is colormapped, every single pixel v in the original image is replaced by the corresponding color for $(v - v_{\text{min}})/(v_{\text{max}} - v_{\text{min}})$. The halfway point in the original data is $(v_{\text{max}} - v_{\text{min}}) \times 0.5 + v_{\text{min}}$ (i.e., half the range plus the minimum) and if you stare at this, this value will correspond to 0.5, and so therefore to the color halfway through[3].

While v_{min} and v_{max} are often automatically set via the data, it is sometimes important to manually set them. This is crucial if you are comparing two colormapped images. If the values of v_{min} and v_{max} are different between images, then colors mean totally different things! Imagine if we colormapped weather maps from July and from January. If we decide on what "yellow" and "blue" mean independently, one map's yellow might correspond to a lower value than another's blue.

B Debugging Hints

B.1 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 8. It's rare for you to have exactly one of these bugs. But the type of behavior suggests different bugs.

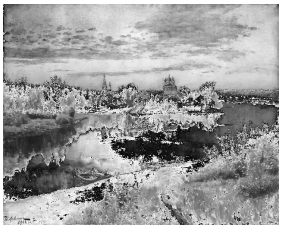
1. **I can't see anything:** If you expect an image with something, but it's just black or just white, this is often a type issue, a NaN somewhere, an image that has no range (i.e., everything is indeed the same value), or an image with too big of a range (i.e., most values are $\sim 10^1$ and one value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

2. **Colors gone wrong:** If general shape looks right but the colors are wrong, it's likely that

[3] A final technical nuance is that colormaps are instead frequently stored as tables. While many colormaps have an analytical form, this is not always the case. Thus, it is common practice to instead store a "look-up" table which contains colors 0 to $N-1$. One therefore does a final remapping from 0 to 1 to 0 to $N-1$.

Original
Image I

Grayscaled
(by averaging colors)



Sum(I)
in uint8

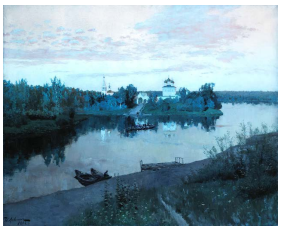
$(I * 2)$ with
$[0, 1]$ as limits

$(I * 3)/3$ in uint8

$(I/64) * 64$ in uint8

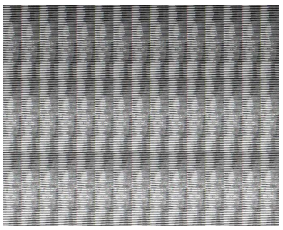
$I/2$ in uint8

$I * 2$ in uint8



BGR and RGB flipped

Dimensions 0/1 flipped



Dimensions 0/2 flipped

Dimensions 1/2 flipped

Figure 8: Visually identifying bugs. Particular types of bugs have particular visual appearances. Some are due to doing operations on uint8 that don't act like normal numbers (indicated by colors being off). Some are due to indexing the image incorrectly (e.g., treating rows and columns incorrectly).

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 ints. 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?

B.2 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 but prevents re-introduction of typos.

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 time waiting for the program 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`):

   ```
   1 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 $\text{NaN} \circ x = \text{NaN}$ for any x and (basically) operator $\circ$.

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

(d) Check the average value: `X.mean()`. Does this make 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 time. A few tips:

   (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.

References and Credits

Part of this homework is taken from UMich EECS 442 by David Fouhey.