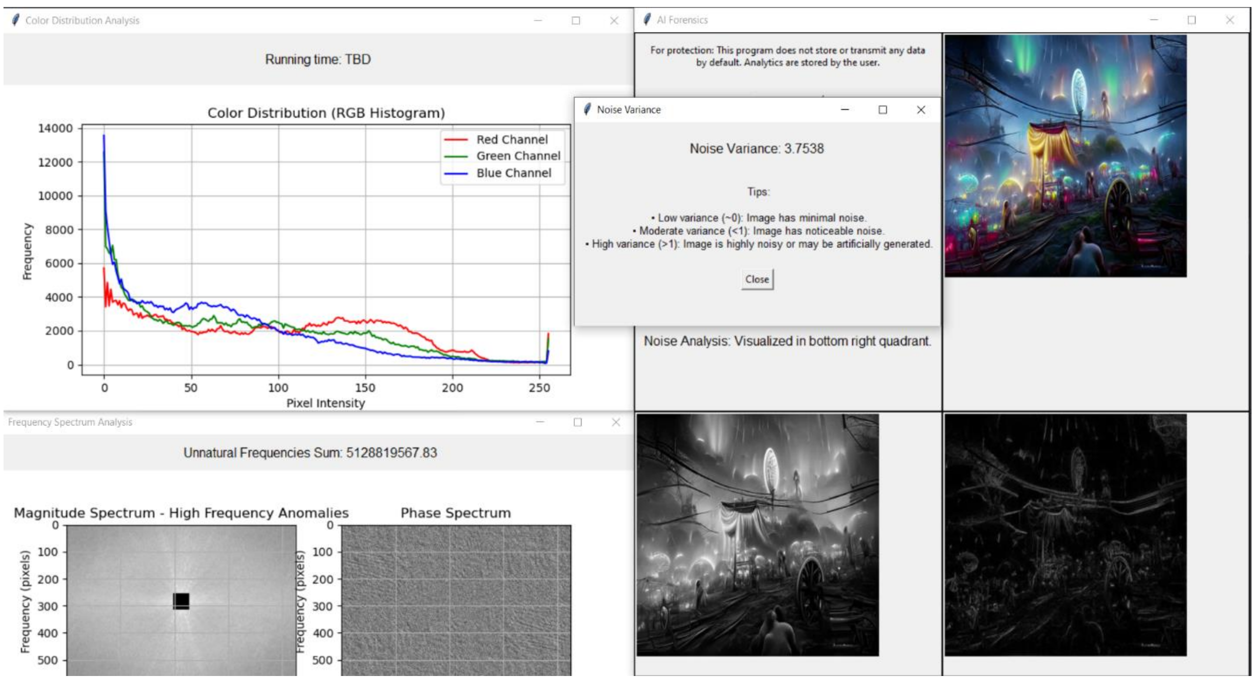
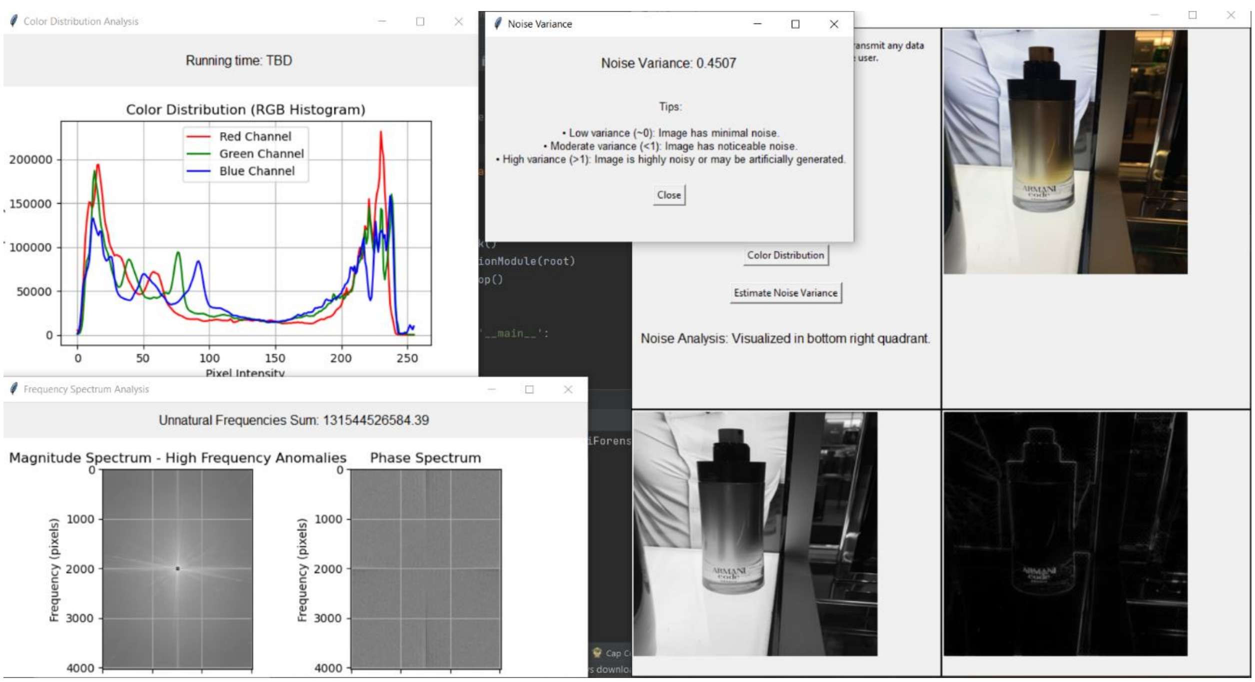
When it comes to real-world applications, artificial intelligence is taking over the world and people may have some issues with being able to detect if a certain image is artificially generated. We as a team, have decided to grab various sources of code from various places and compile an application that allows you to determine if an image is artificially generated.

To start off, we'd like to give a demonstration of how everything works and what these various graphs may correlate to.



- This above is an artificially generated image. One thing that may set off the "artificial" alarms the noise variance. Usually when an image is artificially generated, the noise variance will be above 1.0, however this is not perfect, it usually is a great tell as to why an image may be artificially generated. We've given three images, the original, the gray version, and the noise analysis which is visualized in the bottom right.

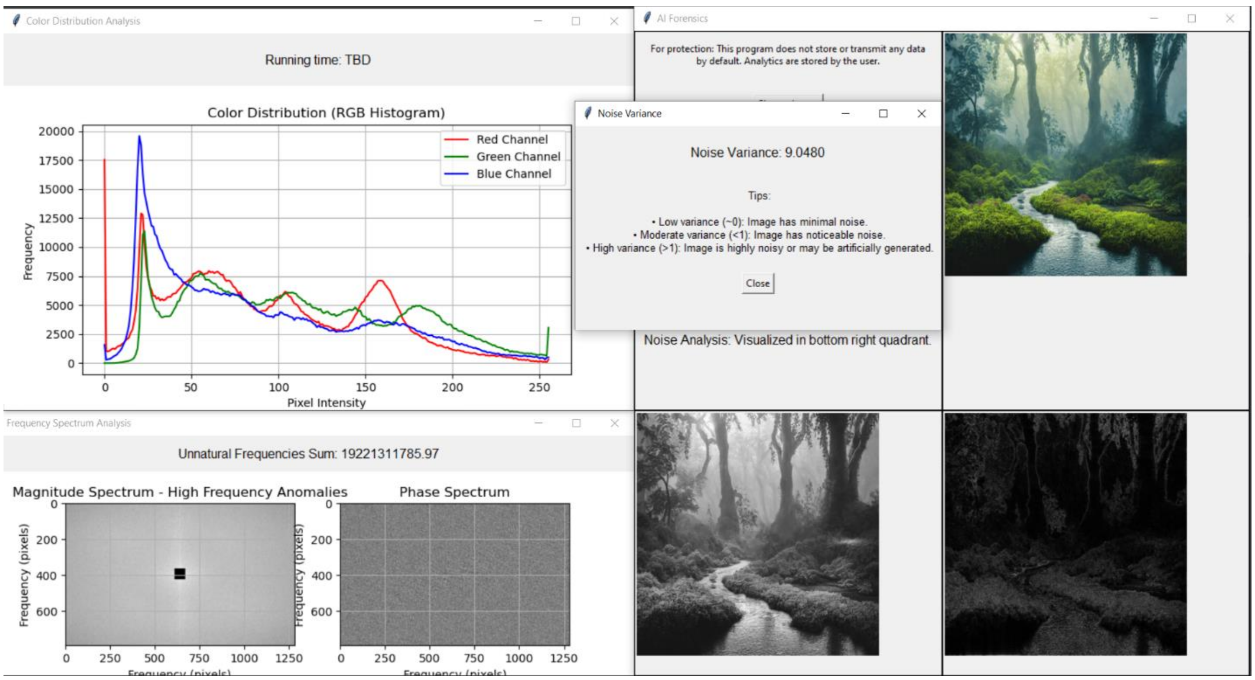
- Essentially what you're looking for, for real images, is a wild color distribution (lines going all over the place), a silky-smooth phase spectrum (meaning that the transition phases are more natural), and a smaller section of unnatural frequencies on the magnitude spectrum (meaning, a smaller black square in the middle, as that part is our highlighter so to speak)
- With the image noise, look for depth. Notice when you take selfies that there's noise behind you that gets detected. That's another metric.
- Anything that is vice versa to these measurements is probably evidence for AI. If not one method, then if other methods are proving its AI means it probably is. I used multiple methods here because it's a sort of voting process. If only 2/5 methods seem to come out as "real", then confidence is lower.

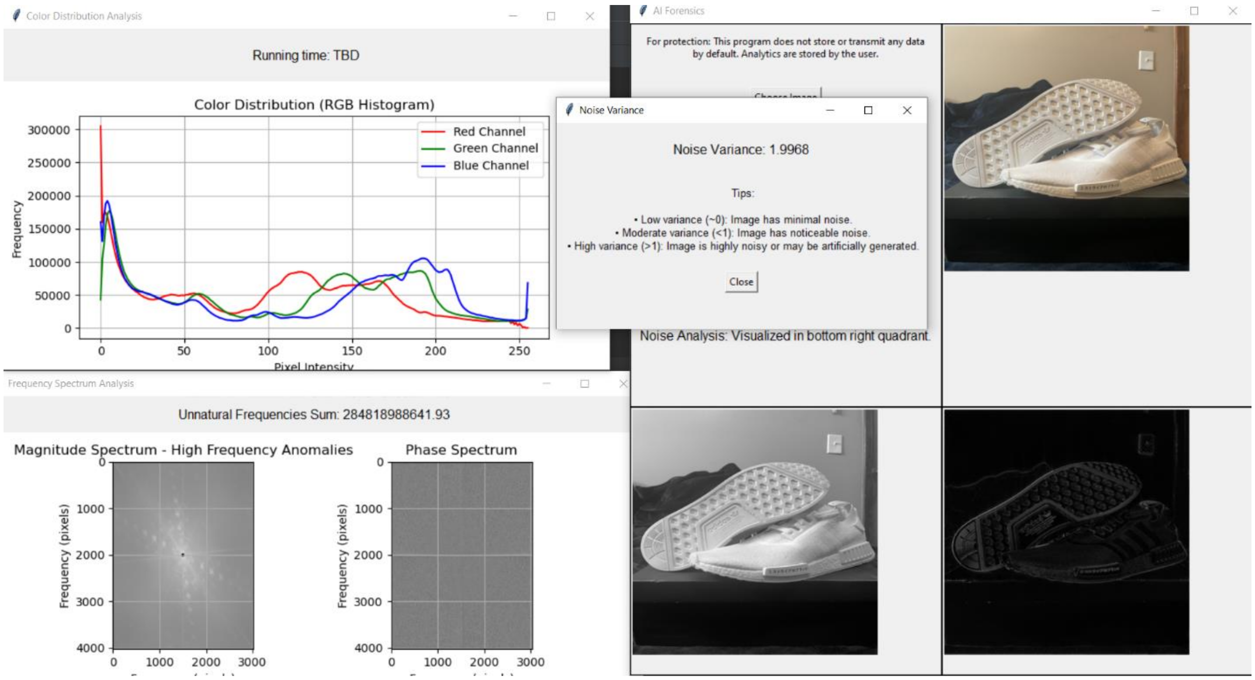


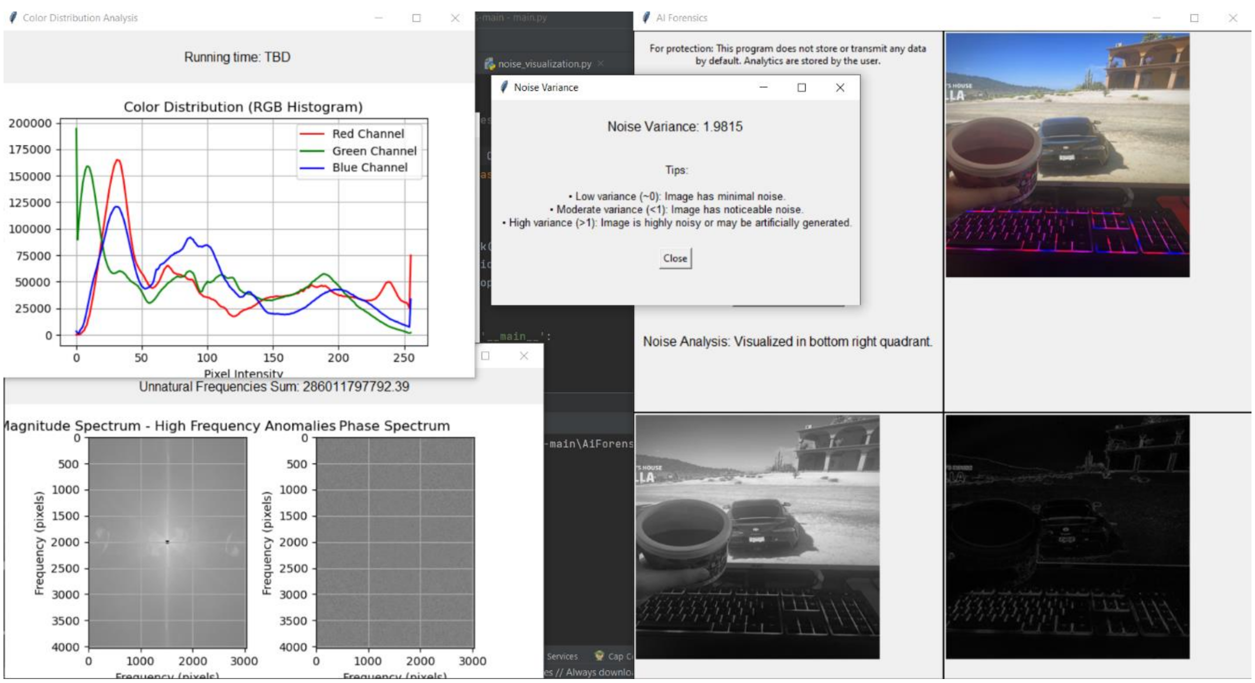
Another piece of advice that wasn't mentioned is that when we look at the magnitude spectrum, real images usually have a smaller square than the artificially generated images. With this, noise variance is also a huge thing once again and anything under 2ish is usually a real image while anything over is most likely AI.

Just a note: Combing all these things like variance, depth of noise, magnitude spectrum will help you get better confidence of if it's AI or real.

More examples:



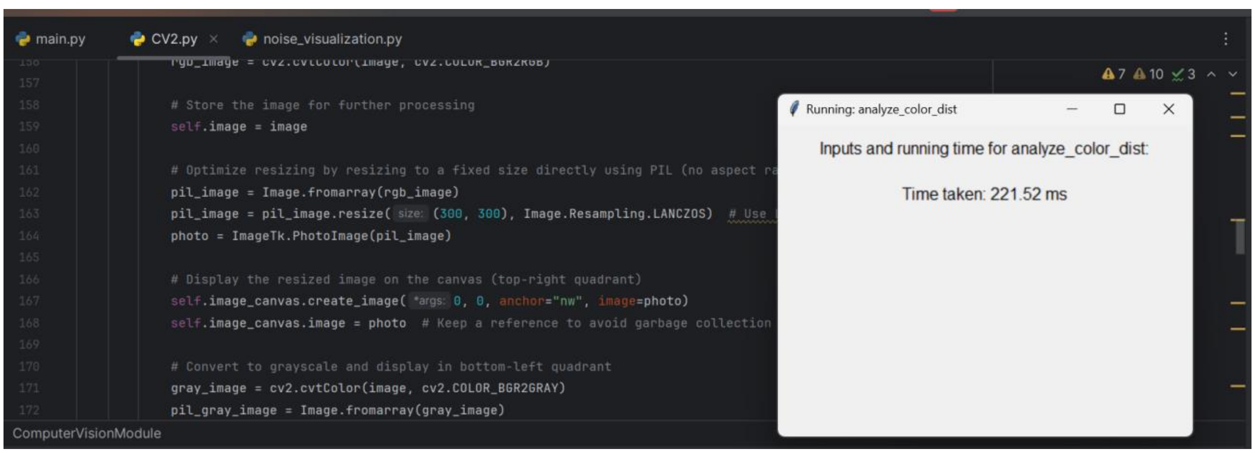

Color Distribution Analysis
Running time: TBD
Color Distribution (RGB Histogram)
Red Channel
Green Channel
Blue Channel
Frequency
Pixel Intensity
AI Forensics
For protection: This program does not store or transmit any data by default. Analytics are stored by the user.
Noise Variance
Noise Variance: 1.9968
Tips:
• Low variance (~0): Image has minimal noise.
• Moderate variance (<1): Image has noticeable noise.
• High variance (>1): Image is highly noisy or may be artificially generated.
Close
Noise Analysis: Visualized in bottom right quadrant.
Frequency Spectrum Analysis
Unnatural Frequencies Sum: 284818988641.93
Magnitude Spectrum - High Frequency Anomalies
Phase Spectrum
Frequency (pixels)


Color Distribution Analysis
Running time: TBD
Color Distribution (RGB Histogram)
Red Channel
Green Channel
Blue Channel
Pixel Intensity
AI Forensics
For protection: This program does not store or transmit any data by default. Analytics are stored by the user.
noise_visualization.py
Noise Variance
Noise Variance: 1.9815
Tips:
• Low variance (~0): Image has minimal noise.
• Moderate variance (<1): Image has noticeable noise.
• High variance (>1): Image is highly noisy or may be artificially generated.
Close
Noise Analysis: Visualized in bottom right quadrant.
Unnatural Frequencies Sum: 286011797792.39
Magnitude Spectrum - High Frequency Anomalies
Phase Spectrum
Frequency (pixels)

Now that we have demonstrated how to understand the program let’s talk about benchmarks.

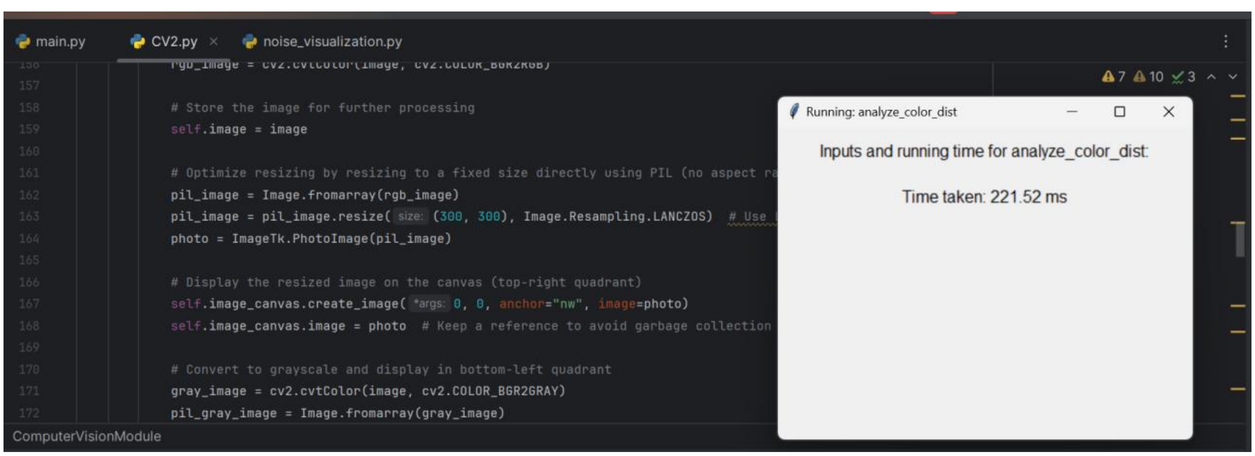
The benchmarks are as follows.

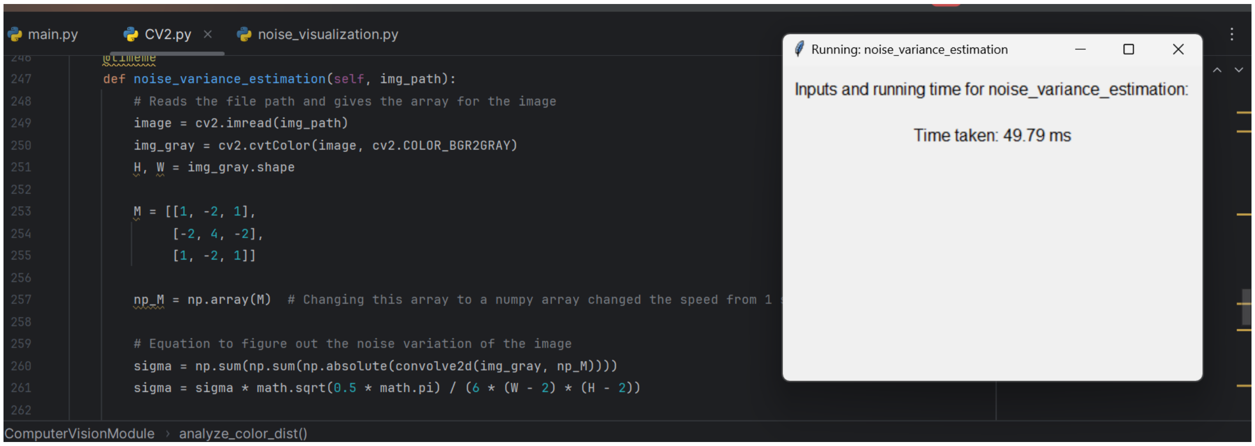
~ Original

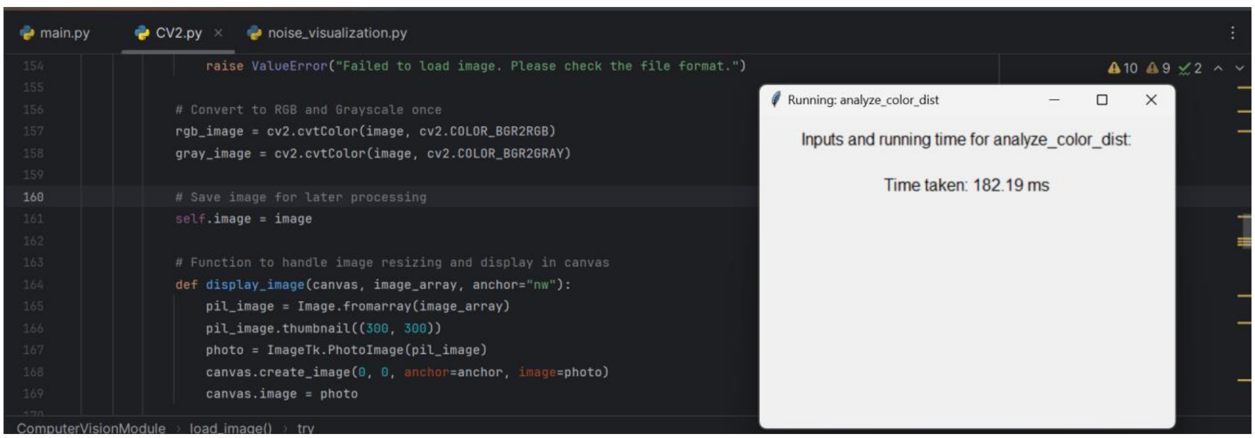


The load images had many effects on various functions in the program. We had to optimize this so that we could have our program running a little bit faster. This was benchmarks of our original code, and we had tried to benchmark our run_prediction_model function, but it seemed nothing seemed to have an effect on it so we couldn't lower the speed of it.

~ Optimized







These are our optimized times, cutting noticeable milliseconds off our program. Of course, it was quite hard to optimize these code patches because they had already used very effective libraries but there were some spots that we fixed. The main improvement was the load and using the helper function to run all the code instead of having to rerun the code repeatedly. For some reason, the helper function had drastically helped our speed. For our noise_variance_estimation function, the code was intricate and definitely very hard to optimize, but just by changing the array to a Np array and allowing for a more time-effective way to go through consolve2D, we were able to cut down 60~ milliseconds on average. We had also removed the "run_noise_variance_estimation" function and recoded it directly to the button to save some time as well. Our choice was originally profiling the code and looking at the memory, but we decided to just look at speed using a timeme decorator with a simple GUI to tell you the speed of the certain function when the buttons were clicked. As for the analyzing color function, simply adding that helper function to our load function increased the speed by a marginal amount. When measuring the time, it was pretty hard to measure because it was always fluctuating so we decided that it was best to run the functions over and over again to see what an “average” was and then take a screenshot once we ran through it to got a general basis of it.

To measure the load images time (note) you must change the manual file itself.

Replace the following lines in load_images function to benchmark:

```
try:
        # Load the image with OpenCV
        image = cv2.imread(r"C:\Users\Name\Location\pictureexample.jpg")
        if image is None:
            raise ValueError("Failed to load image. Please check the file format.")
```

The challenges were harder than we had much anticipated whether it was the noise variance, noise depth, tkinkter itself, and optimization. The application that we had chosen was so complex that they were built using libraries that are already time efficient, so it was definitely hard to shave off a marginal amount of time. It may seem like a short amount of time, but the point of this class is to make everything optimized and faster, whether it’s one minute, one second, or even one millisecond. Building the GUI and finding the necessary code to be able to even create the project was difficult and finding a way to "correctly" time the functions was another obstacle that we had to face. Another challenge was our time schedules, not being able to meet up in person and having to talk online and remotely working on it one-by-one because our schedules didn't match was one of the very difficult things that we hadn't quite prepared for. GitHub was also a thing that we hadn't used very often or at all, so it was a bit different to walk into. Overall, we learned a great amount of knowledge of how to use libraries such as NumPY, SciPY, OpenCV, and Pillow to optimize our way of image processing. This final project taught us a lot about time management and how teamwork would feel like in the real-life world. In the end, it allowed us to experience what it was like to work on a project where everyone's schedule didn't match and how to evenly disperse our work loads based on what we could manage.