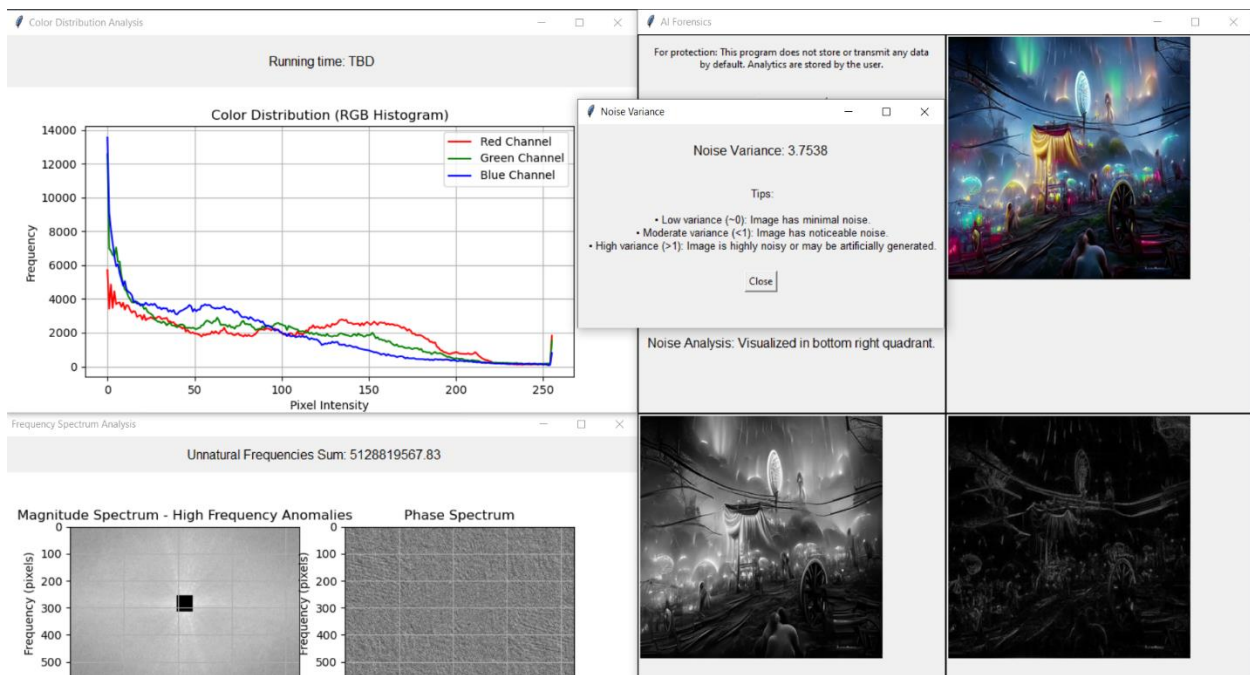


When it comes to real-world applications, artificial intelligence is taking over the world and people may have some issues with being able to detect if a certain image is artificially generated.

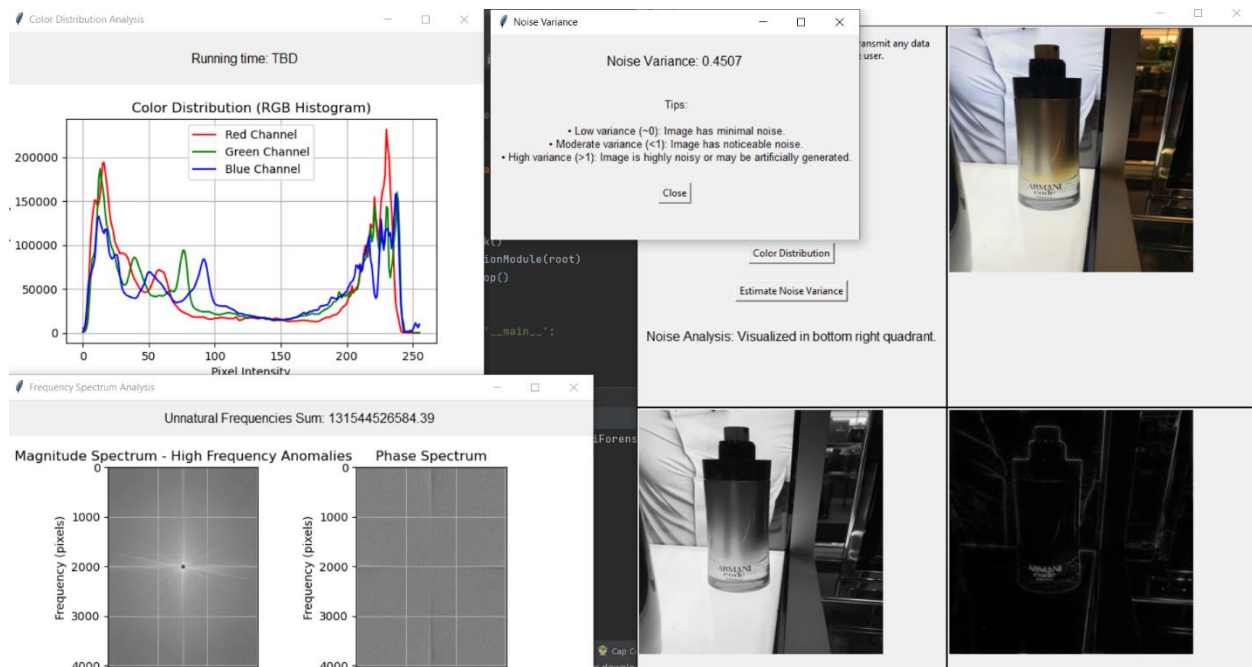
We as a team, have decided to grab various sources of code from various places and compile an application that allows you to determine if an image is artificially generated.

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.

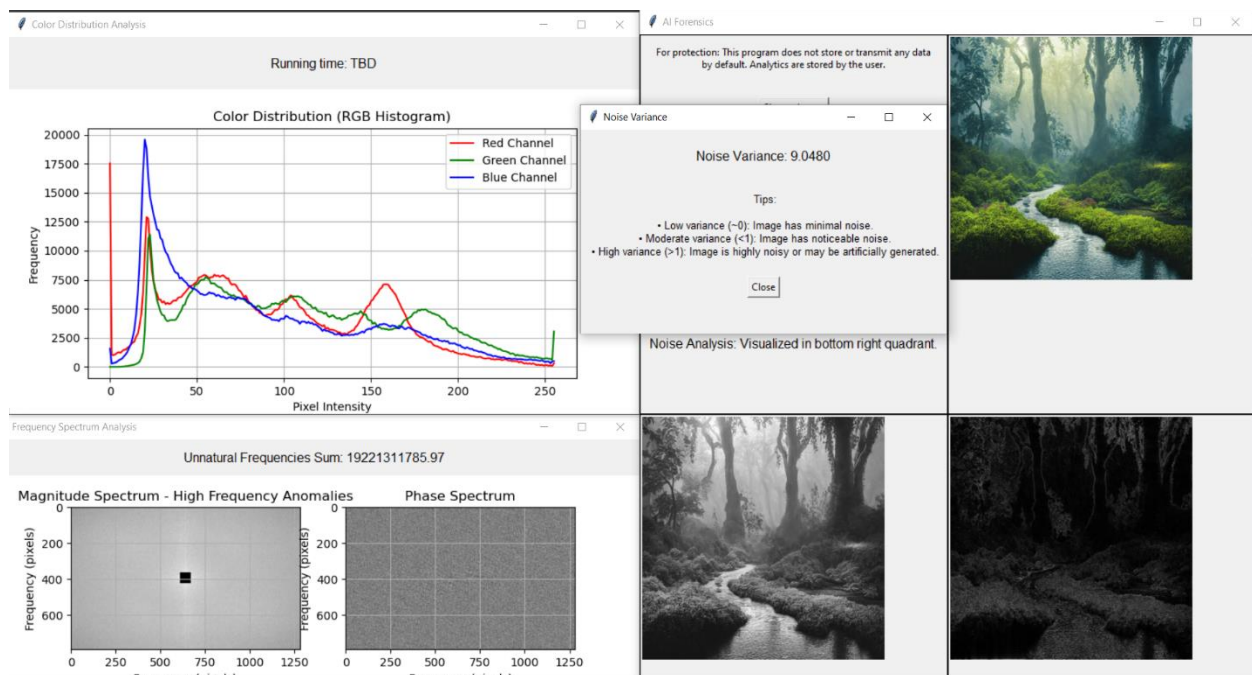


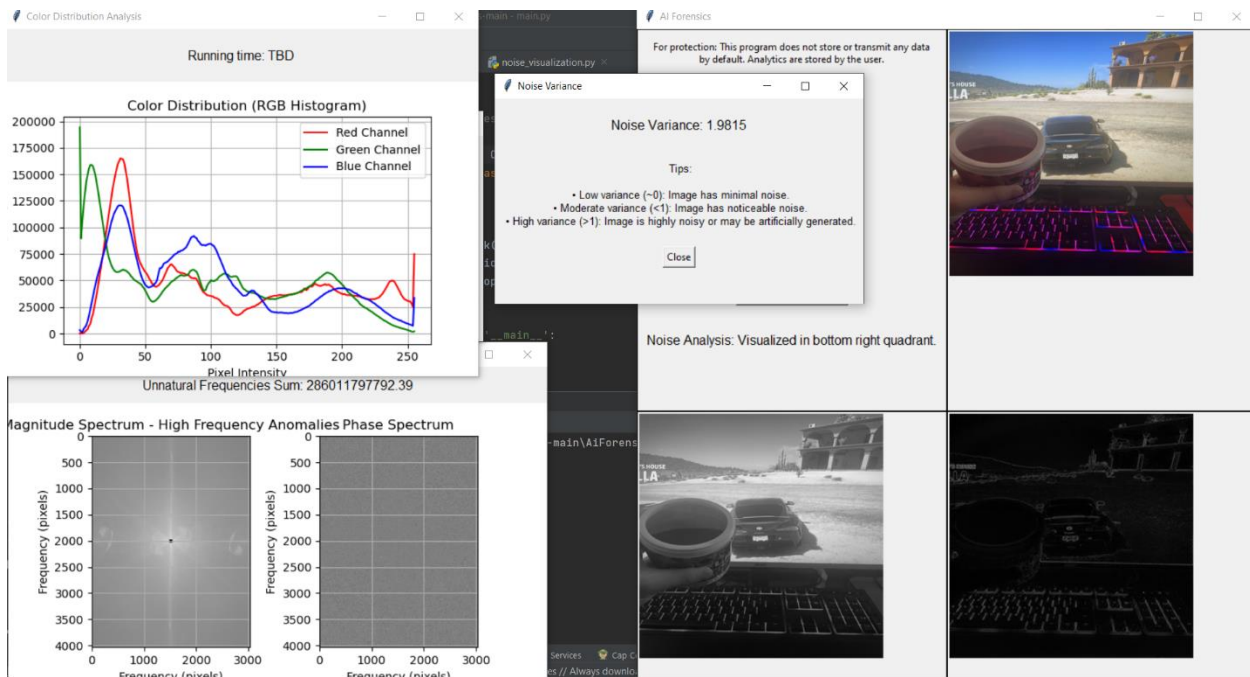
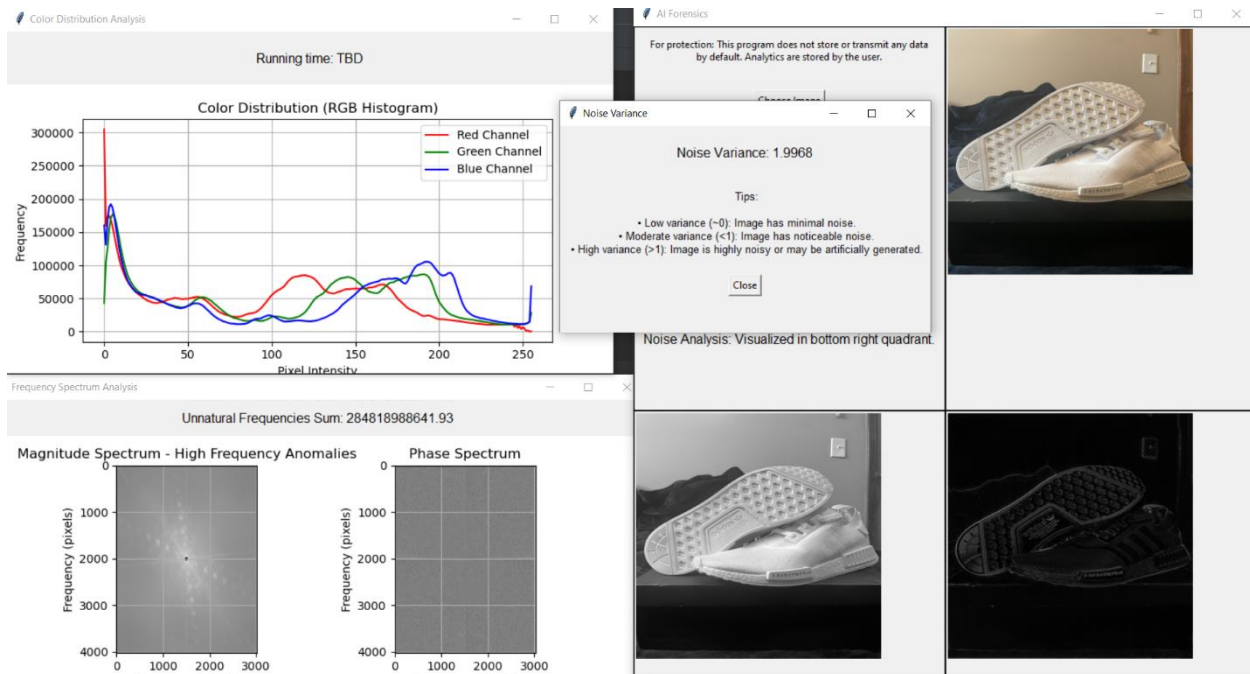
Another piece of advice that wasn't mentioned is that when we look at the magnitude spectrum, real images usually have a smaller square than the artificially generated images. With this, noise

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:

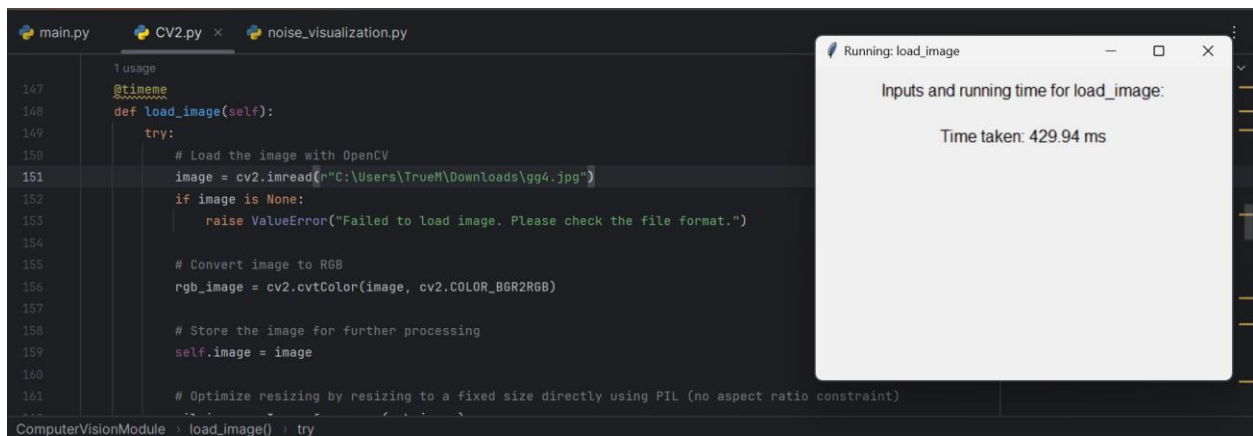




Now that we have demonstrated how to understand the program let's talk about benchmarks.

The benchmarks are as follows.

~ Original



The screenshot shows a code editor with three tabs: main.py, CV2.py, and noise_visualization.py. The CV2.py tab is active, displaying the load_image function. The function is decorated with @timed and contains a try block for loading an image from a file, converting it to RGB, and storing it. A benchmark window titled 'Running: load_image' is open, showing the time taken for the function to execute: 429.94 ms.

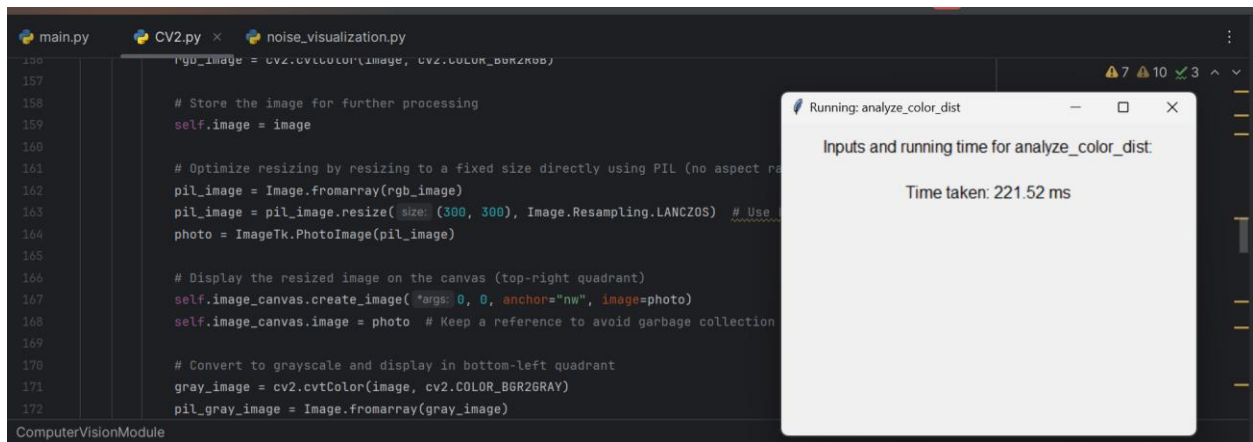
```
147 @timed
148 def load_image(self):
149     try:
150         # Load the image with OpenCV
151         image = cv2.imread(r"C:\Users\TrueH\Downloads\gg4.jpg")
152         if image is None:
153             raise ValueError("Failed to load image. Please check the file format.")
154
155         # Convert image to RGB
156         rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
157
158         # Store the image for further processing
159         self.image = image
160
161         # Optimize resizing by resizing to a fixed size directly using PIL (no aspect ratio constraint)
```

ComputerVisionModule > load_image() > try

Running: load_image

Inputs and running time for load_image:

Time taken: 429.94 ms



The screenshot shows the same code editor with the CV2.py tab active, displaying the analyze_color_dist function. The function continues from the previous one, showing the image being resized and converted to grayscale. A benchmark window titled 'Running: analyze_color_dist' is open, showing the time taken for the function to execute: 221.52 ms.

```
156 rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
157
158 # Store the image for further processing
159 self.image = image
160
161 # Optimize resizing by resizing to a fixed size directly using PIL (no aspect ratio constraint)
162 pil_image = Image.fromarray(rgb_image)
163 pil_image = pil_image.resize(size=(300, 300), Image.Resampling.LANCZOS) # Use LANCZOS for better quality
164 photo = ImageTk.PhotoImage(pil_image)
165
166 # Display the resized image on the canvas (top-right quadrant)
167 self.image_canvas.create_image(*args, 0, 0, anchor="nw", image=photo)
168 self.image_canvas.image = photo # Keep a reference to avoid garbage collection
169
170 # Convert to grayscale and display in bottom-left quadrant
171 gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
172 pil_gray_image = Image.fromarray(gray_image)
```

ComputerVisionModule

Running: analyze_color_dist

Inputs and running time for analyze_color_dist:

Time taken: 221.52 ms



```
main.py CV2.py x noise_visualization.py
1246 @time
1247 def noise_variance_estimation(self, img_path):
1248     # Reads the file path and gives the array for the image
1249     image = cv2.imread(img_path)
1250     img_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
1251     H, W = img_gray.shape
1252
1253     M = [[1, -2, 1],
1254          [-2, 4, -2],
1255          [1, -2, 1]]
1256
1257     # Equation to figure out the noise variation of the image
1258     sigma = np.sum(np.sum(np.absolute(convolve2d(img_gray, M))))
1259     sigma = sigma * math.sqrt(0.5 * math.pi) / (6 * (W - 2) * (H - 2))
1260
1261     return sigma
```

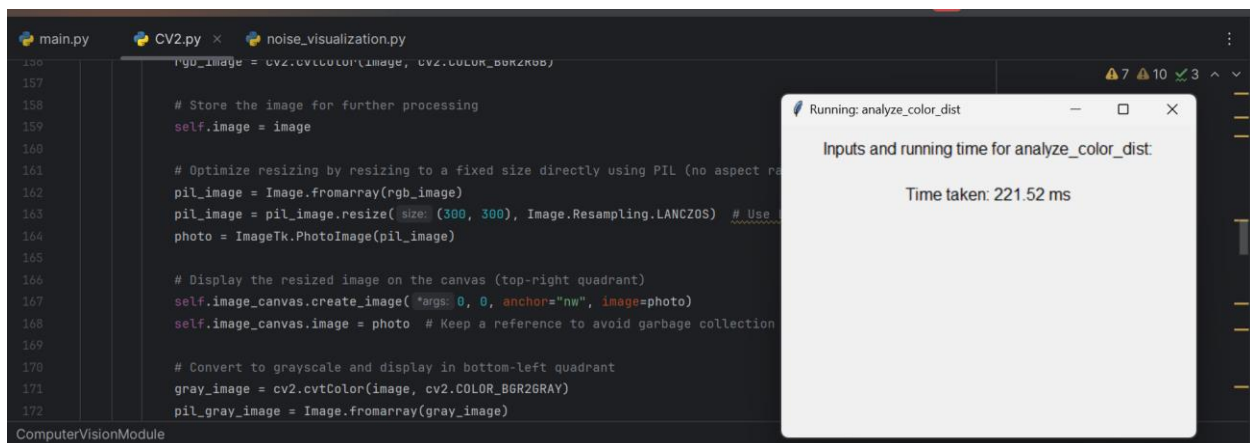
Running: noise_variance_estimation

Inputs and running time for noise_variance_estimation:

Time taken: 103.84 ms

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



```
main.py CV2.py x noise_visualization.py
130 rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
131
132 # Store the image for further processing
133 self.image = image
134
135 # Optimize resizing by resizing to a fixed size directly using PIL (no aspect ratio)
136 pil_image = Image.fromarray(rgb_image)
137 pil_image = pil_image.resize(size=(300, 300), Image.Resampling.LANCZOS) # Use LANCZOS for better quality
138 photo = ImageTk.PhotoImage(pil_image)
139
140 # Display the resized image on the canvas (top-right quadrant)
141 self.image_canvas.create_image(args=0, 0, anchor="nw", image=photo)
142 self.image_canvas.image = photo # Keep a reference to avoid garbage collection
143
144 # Convert to grayscale and display in bottom-left quadrant
145 gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
146 pil_gray_image = Image.fromarray(gray_image)
```

Running: analyze_color_dist

Inputs and running time for analyze_color_dist:

Time taken: 221.52 ms


```
main.py CV2.py noise_visualization.py
247 def noise_variance_estimation(self, img_path):
248     # Reads the file path and gives the array for the image
249     image = cv2.imread(img_path)
250     img_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
251     H, W = img_gray.shape
252
253     M = [[1, -2, 1],
254          [-2, 4, -2],
255          [1, -2, 1]]
256
257     np_M = np.array(M) # Changing this array to a numpy array changed the speed from 1
258
259     # Equation to figure out the noise variation of the image
260     sigma = np.sum(np.sum(np.absolute(convolve2d(img_gray, np_M))))
261     sigma = sigma * math.sqrt(0.5 * math.pi) / (6 * (W - 2) * (H - 2))
262
```

Running: noise_variance_estimation

Inputs and running time for noise_variance_estimation:

Time taken: 49.79 ms

ComputerVisionModule > analyze_color_dist()

```
main.py CV2.py noise_visualization.py
154 raise ValueError("Failed to load image. Please check the file format.")
155
156 # Convert to RGB and Grayscale once
157 rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
158 gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
159
160 # Save image for later processing
161 self.image = image
162
163 # Function to handle image resizing and display in canvas
164 def display_image(canvas, image_array, anchor="nw"):
165     pil_image = Image.fromarray(image_array)
166     pil_image.thumbnail((300, 300))
167     photo = ImageTk.PhotoImage(pil_image)
168     canvas.create_image(0, 0, anchor=anchor, image=photo)
169     canvas.image = photo

```

Running: analyze_color_dist

Inputs and running time for analyze_color_dist:

Time taken: 182.19 ms

ComputerVisionModule > load_image() > try

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 get 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, tinkter 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.