REPORT OF THE LAB 1

1. THEORETICAL UNDERSTANDING OF THE ALGORITHMS USED

SECTION 1. Introduction to Image Filtering

Image filtering is a fundamental concept in computer vision used to modify images. These modifications allow you to clarify an image to extract the information you want. This could involve anything from extracting edges from an image, blurring it, or removing unwanted objects. Some common applications include:

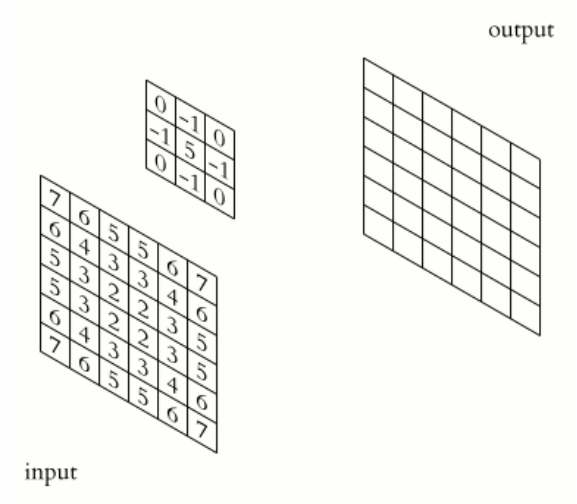
- **Denoising:** Noise Reduction. Filtering is used to reduce noise in images, making them clearer for subsequent processing. Basically, images' quality is enhanced.
- **Edge Detection:** Filters are applied to highlight edges and boundaries in images, aiding in object detection.
- **Image Enhancement:** Increase of the resolution of an image. Filters can enhance contrast, brightness, or certain features, making images more visually appealing or informative.
- **Feature Extraction:** Filters help extract specific features or patterns from images, which are crucial in tasks like object recognition.
- **Image Segmentation:** Filtering can be used to separate an image into distinct regions or segments.
- **Inpainting:** Filling in missing or corrupted parts of images.

SECTION 2. Convolution

Convolution involves taking a filter (or kernel) and sliding it over the image, computing element-wise multiplication between the filter values and corresponding pixel values in the image, and then summing these results to give a new pixel value. This process is repeated for every pixel in the image. Convolution filters are used with images to achieve blurring, sharpening, embossing, edge detection, and other effects. This is performed through the convolution of a kernel and an image. Kernels are typically 3×3 matrices, and the convolution process is formally described as follows:

$$g(x,y)=w*f(x,y)$$

Where g(x,y) represents the filtered output image, f(x,y) represents the original image, and w represents the filter kernel. The graphic below shows how the convolution works.



Changing the size of the kernel affects the level of detail captured by the convolution operation. Larger kernels capture more global information (lower frequencies), while smaller kernels capture more local information (higher frequencies).
Some significant effects on the outcome of the convolution operation due to the size of the kernel are the following:

1. **Detail Capture**: Smaller kernels can capture a lot of details, while larger kernels may lose some details. For instance, a 3x3 kernel might detect smaller features better than a 5x5 kernel. The size of the kernel represents a trade-off between smoothing and detail preservation. Choosing an appropriate kernel size is crucial to balance the removal of noise and subtle details with the enhancement of important image features.
2. **Overfitting and Underfitting**: A smaller kernel will give you a lot of details, which can lead to overfitting, while larger kernels can lead to underfitting as they might miss out on some details.
3. **Computational Efficiency**: Larger kernels can make the computation faster and use less memory, but at the cost of losing details.
4. **Receptive Field**: The size of the kernel influences the receptive field, i.e., the region of the input space that affects a particular unit of output.
5. **Output Size**: If padding is not used, larger kernels will result in smaller output dimensions.

Padding is an important aspect of convolution, involving the addition of extra pixels to the input image before applying convolution. Acting as a protective boundary around the image, these additional pixels allow the network to retain more spatial information, preventing any information loss, particularly from the edges of the image. Moreover, padding helps to prevent the image's compression after every convolutional operation. This is the main use of padding in CNN.

There are three main important types of padding: Same Padding, Casual Padding, and Valid Padding.

1. **Same Padding.** Ensures size parity between the output feature maps and the input image. It achieves this by padding the input borders in a way that allows the convolutional filter to stride evenly across the image. Allows every pixel in the input to be at the center of the filter at least once. Its application preserves spatial information and avoids border effects. Here, padding size depends directly on the size of the filter.
2. **Casual Padding.** The padding is applied asymmetrically: you add extra values only to the left side of the sequence. This is considered the "past" of the sequence. Thus, this technique maintains the causality of the data. It ensures that each output element only depends on past or current input elements. Causal padding is essential when dealing with tasks that require predicting future values based on past or present observations.
3. **Valid Padding.** Valid padding is the absence of any extra pixels added to the borders of the input feature map. Thus, Valid Padding is essentially the same as no padding. The convolution is performed only on the pixels which completely overlap with the filter. As a result, its output feature maps are smaller than the input.

Stride is another crucial parameter in convolution, determining how the filter moves across the input image or feature map. On other words, the strides parameter indicates how fast the kernel moves along the rows and columns on the input layer. If a stride is (1, 1), the kernel moves one row/column for each step; if a stride is (2, 2), the kernel moves two rows/columns for each step.

The stride affects the output in two main ways:

- It can be used to **downsample** the output feature map. A larger stride results in a smaller output size, as it means that the filter is jumping over more pixels at each step.
- It provides some level of **shift-invariance**, reducing sensitivity to exact pixel locations and offering a form of translation invariance.

Therefore, stride is an important hyperparameter in convolutional neural networks that allows for more control over the size and properties of the output feature maps.

Regarding the choice of filter kernel size, odd-sized kernels are often preferred in image filtering. This preference arises from the fact that odd-sized kernels have a clear center, which allows for symmetry around the output pixel. This reduces distortion between layers as the previous layer pixels would be symmetrical around the output pixel. This symmetry is often desirable in image processing as it helps preserve the original positioning of the image features.

However, even-sized filters can be used: For example, in some specific types of filtering where you might want to preserve certain properties of the original image. But it's important to note that using an even-sized filter can lead to an asymmetrical kernel, which might introduce a shift or bias in the filtered image.

The kernel is flipped during the convolution operation to ensure that the operation is commutative, meaning the order in which the signals are processed does not affect the result. If the kernel is not flipped, you end up computing a correlation of a signal with itself.; which means that instead of mixing or blending the image and filter information (which is what we want), we are just comparing how similar they are (which is not what we want).

Correlation is the measurement of the similarity between two signals/sequences. While convolution is the measurement of effect of one signal on the other signal.
The fundamental difference between convolution and correlation lies in the operation performed on the signals. In convolution, one of the signals (≈ the kernel) is flipped before being multiplied and summed with the other signal (≈ before applying it to the image). In correlation, no such flipping occurs. This difference leads to distinct behaviour in edge detection and feature extraction.

On the other side, separable kernels are a concept in convolution operations that can significantly optimize computational efficiency.

A kernel is said to be separable if it can be expressed as the outer product of two vectors. For instance, a 3x3 kernel is separable if it can be decomposed into a 3x1 and a 1x3 kernel. This property allows us to perform what is called Separable Convolution, that is a process in which a single convolution can be divided into two or more convolutions with smaller kernels to produce the same output. A single process is divided into two or more sub-processes to achieve the same effect.

The benefits of using separable kernels in convolution operations include:

1. **Reduced Computational Complexity**: Separable convolutions reduce the computational complexity from O(m*n) to O(m+n) per output pixel, assuming that the cost of performing a convolution is relatively cheap.
2. **Efficiency**: They are particularly useful when optimizing the model for smaller size or higher speed, compromising least with accuracy.
3. **Flexibility**: There are two types of separable convolutions - Spatially Separable Convolutions and Depth-wise Separable Convolutions. This provides flexibility in handling different types of kernels and data.

However, it’s important to note that not every kernel can be separated.

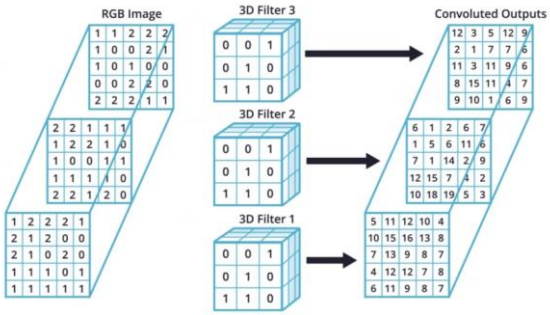
OTHER APPLICATIONS: Template matching and convolution operations to coloured images

Template matching involves sliding a template image over a larger image; it's essentially a method for detecting objects in an image.
The process of template matching using normalized cross-correlation involves the following steps:

1. Defining the Goal: The goal is to find the location of a small template image within a larger image.
2. Assumptions: The offsets we need to estimate are expressed as geometric transformations with some assumptions. The cross-correlation assumption is that only a translation change is present between source and destination.
3. Cross-Correlation: Cross-correlation is used to find certain image content in another image and determine the exact location of the content in the image.
4. Normalization: Normalization is applied to make the method more robust against variations such as changes in illumination.

The underlying principle of this method is the calculation of the cross-correlation between the template and the image for each possible position of the template within the image. The positions with highest cross-correlation values indicate potential matches.

Convolution operations can also be extended to colour images by considering the image as a 3D volume composed of width, height, and depth, where depth corresponds to the colour channels. For instance, an RGB image has three channels (Red, Green, and Blue). For instance, if the filter used would had been CMYK (Cian, Magenta, Yellow, and Black), the image would have four channels (depth = 4).



Knowing that, the convolution operation is performed on each channel separately. Each color channel is convolved with its own 2D filter, which means that it corresponds to a two-dimensional array of pixel values, and the results are summed up to form the final output feature map.

However, dealing with multiple channels introduces some complexities:

- Increased Computational Complexity: The need to perform convolutions on each channel separately increases the computational complexity.
- Increased Model Parameters: Each channel requires its own set of filters, increasing the number of parameters that the model needs to learn.
- Variable Number of Channels: Some images may have more or fewer channels than others (e.g., grayscale vs. RGB), which requires additional handling.

Despite these complexities, using multiple channels allows Convolutional Neural Networks (CNNs) to capture more complex features and leads to more robust models.

Finally, we must mention that there are several methods to apply filters to an image besides convolution. Here are a few:

1. **Gaussian Filtering**: This method applies a Gaussian filter to the image, effectively reducing noise and enhancing quality. It relies on the standard deviation (sigma) to control the blur intensity.
2. **Median Filtering**: This is a non-linear method used to remove noise from images. It replaces each pixel's value with the median value of the neighbouring pixels.
3. **Adaptive Filtering**: This method adjusts the filter parameters based on the local image variance, making it effective for images with varying noise levels.
4. **Morphological Filtering**: This method applies structuring elements to an image, enabling operations like dilation and erosion. It's useful for extracting image features.
5. **Frequency Domain Filtering**: This involves transforming the image into the frequency domain (using Fourier Transform), applying filters, and then transforming back into the spatial domain.

Comparing these methods, Gaussian and median filtering are often used for noise reduction, with Gaussian filters being particularly effective at reducing Gaussian noise and median filters at removing salt-and-pepper noise. Adaptive filtering provides a more flexible approach to noise reduction by adjusting to local noise levels.

Morphological filtering is less about noise reduction and more about feature extraction, making it useful for tasks like edge detection. Frequency domain filtering is a powerful technique that can make certain operations more computationally efficient, such as convolution becoming simple multiplication in the frequency domain.

Each of these methods has its own strengths and weaknesses, and the choice of method depends on the specific requirements of the image processing task at hand.

For instance, let's focus on the real-scenario application **Self-driving car vision system**.
The primary sensors in these systems are cameras, radar, and lidar. These sensors work together to provide the car with visuals of its surroundings and help it detect the speed and distance of nearby objects, as well as their three-dimensional shape. This results on the fact that, in self-driving cars, computer vision plays a crucial role in feature extraction and object detection.

Various types of filters are used in this context:

- Edge Detection Filters: These filters are used for lane detection. They help identify boundaries of lanes, which is critical for path planning and navigation.
- Colour Filters: These filters are used to detect traffic signs. They can isolate specific colour ranges in an image, which is useful for identifying traffic lights or road signs.
- Noise Reduction Filters: These filters are used to reduce sensor noise and improve the quality of the data. For example, radar and lidar data can be noisy and may require filtering to remove erroneous readings.

LINEAR FILTERS

Linear filters, often used in image processing, have limitations as they can distort the original signal and affect its information content. For example, a linear filter can alter the shape or width of a spectral peak, which can affect the interpretation of the data. This is especially problematic when the signal has sharp features or poles that are important for the analysis. Linear filters can also introduce artifacts such as ringing or overshooting, which can obscure the true signal.

Another limitation of linear filters is that they are not adaptive to the characteristics of the noise or the signal. For example, a linear filter may not be able to distinguish between noise and signal when they have similar frequencies or amplitudes. A linear filter may also not be able to handle non-stationary noise, which changes over time or depends on the signal itself. A linear filter may also not be able to reduce noise effectively when the signal-to-noise ratio is low.

Therefore, linear filters are not ideal for noise reduction when the signal has complex or delicate features, when the noise is variable or similar to the signal, or when the signal quality is poor. In these cases, alternative methods such as nonlinear filters, adaptive filters, or statistical methods may be more suitable.

Linear filters might not be very effective at removing “salt and pepper” noise because this type of noise consists of sharp, sudden disturbances in the image. Linear filters, such as averaging or Gaussian filters, are not typically used for this purpose because they can blur the image and reduce detail.

Instead, Median filters, which are non-linear, are often used for this purpose as they preserve boundaries while effectively reducing noise. The median filter works by replacing each pixel value with the median value of the neighbouring pixels. This method is effective because the median is less sensitive to extreme values (outliers), which in this case are the black and white pixels of the salt and pepper noise.

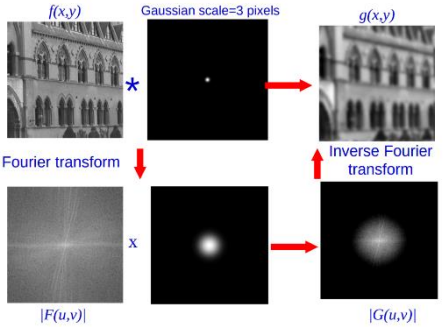
However, traditional median filtering can cause loss of texture details and blurred edges when the noise intensity is large. To overcome this, advanced methods like the adaptive median filter and edge-adaptive total variation model have been proposed. These methods aim to preserve edges and details while effectively removing salt and pepper noise.

How do we combine the traditional median filtering and the adaptative total variation model?

Firstly, the image is segmented into edge regions and non-edge regions by edge detection operators. Secondly, the salt and pepper noise of the image is processed using the median filter and adaptive total variation model, respectively. Lastly, the non-edge regions processed by the median filter and the edge regions processed by the adaptive total variation model are extracted for splicing. The experimental results show that the method cannot only effectively remove salt and pepper noise, but also effectively protect the main edge details of the image.

SECTION 3. The Fourier Transformation

In Image processing, the Fourier Transform tells us what is happening in the image in terms of the frequencies of those sinusoidal. For example, eliminating high frequencies blurs the image; and eliminating low frequencies gives edges. The Fourier Transform decomposes an image into its sine and cosine components, representing it in the frequency domain. It's significant because it allows us to analyse and manipulate images based on their frequency components.



On account of this, the Fourier transform is a way of breaking down an image into its basic components, like colours, shapes, and details. It helps us understand and manipulate the image better.

Outside of image processing, the Fourier Transform is widely used in the field of medical imaging, particularly, we found out as an example the Electrocardiogram (ECG) analysis. An ECG is a graphical representation of the electrical activity of the heart, recorded from electrodes placed on the surface of the chest. The Fourier Transform plays a crucial role in this context as it can be employed to identify and filter out "noise" from these signals. This noise reduction enhances the accuracy of heart rate monitoring and diagnosis, thereby contributing to more effective patient care and treatment. Thus, the Fourier Transform proves to be an invaluable tool in the medical field, extending its utility beyond just image processing.

On the flip side, as it comes to choosing to operate in the frequency domain using the Fourier Transform instead of the spatial domain, several advantages come into play:

1. Control Over Image Characteristics: The frequency domain gives you control over the whole image, where you can enhance (e.g., edges) and suppress (e.g., smooth shadow) different characteristics of the image very easily.
2. Efficiency: Some operations that are computationally expensive in the spatial domain become simple multiplications in the frequency domain. This is particularly beneficial when the size of the spatial filter is big (say > 3 x 3).
3. Quantitative Analysis: The frequency domain represents the rate of change of spatial pixels, which can be advantageous when dealing with problems related to the rate of change of pixels, which is very important in image processing.
4. Noise Filtering: Certain types of noise, such as periodic noise, can only be observed and removed in the frequency domain.
5. Established Processes and Tools: The frequency domain has an established suite of processes and tools that can be borrowed directly from signal processing in other domains.

However, it's important to note that the choice between frequency and spatial domains depends on the specific task and nature of the input data. Some tasks might be more efficiently or effectively performed in one domain or the other. If you're dealing with tasks that involve frequency content analysis or periodic noise removal, then the frequency domain might be more suitable. Howbeit, for tasks that involve direct manipulation of pixel values, like simple noise reduction or contrast enhancement, operating in the spatial domain might be more intuitive and straightforward.

That's why we introduce the following approach: Interconversion between spatial and frequency domains.

The interconversion between spatial and frequency domains using Fourier and other transforms is of critical importance in image processing and, for some imaging methods, the construction of images from raw scan data. A significant feature of the transforms is that we can convert back and forth between spatial and frequency domains without loss of information or introduction of noise.

On the other side, we know that there are two main approaches for the Fourier transformation: FFT and DFT.

FFT (Fast Fourier Transform) and DFT (Discrete Fourier Transform) are mathematical methods for signal analysis in digital signal processing, audio and picture processing, and communication systems. Their Fourier transform speeds differ greatly.

The simplest Fourier transform, DFT, converts a finite sequence of equally spaced function samples into a sequence of complex integers reflecting the signal's frequency components. DFT computation time is related to the square of the number of samples, making it expensive for bigger data sets.

In contrast, FFT reduces the number of computations needed to compute DFT. FFT uses the divide-and-conquer technique to break DFT computation into smaller sub-problems that can be solved individually and merged to achieve the final answer.

FFT is faster than DFT since it requires Nlog(N) operations for N samples, while DFT requires N^2 operations. Here emerges the primary reason FFT is preferred over DFT in many applications: its speed and efficiency. FFT helps in converting the time domain into frequency domain, which makes calculations easier as we often deal with various frequency bands in communication systems. Moreover, FFT can convert discrete data into a continuous data type available at various frequencies, which is advantageous for signal processing applications.

There are many other advantages to using the FFT over the DFT, including improved robustness and reliability when dealing with long sequences, better approximation quality for all sequences, reduced sensitivity to round-off errors, and improved speed for most types of problems.

2. PRACTICUM: RESULTS AND DISCUSSION

Imports: The code starts with importing necessary libraries, such as **cv2** for computer vision, **urllib3** for handling URLs (although it's not used in this code), **numpy** for numerical operations, **PIL** for image processing, **imutils** for various image processing utilities, and **matplotlib** for plotting images.
Add noise to an image. There are two types of noise that can be added: random noise and periodic noise.

- **add_noise Function**: **add_noise(image, noise_type, mean=0, std=0.01, amount=0.05, salt=0.5)** This function adds random noise to an image. The type of noise can be 'gaussian', 'salt_pepper', or 'speckle'.
    - **Gaussian Noise**: It adds Gaussian noise to the image. Gaussian noise is a type of statistical noise with a bell-shaped probability density function (PDF).
    - **Salt and Pepper Noise**: It adds salt and pepper noise to the image. Salt and pepper noise introduces random bright and dark pixels in an image.
    - **Speckle Noise**: It adds speckle noise to the image. Speckle noise is a multiplicative noise, meaning it multiplies the pixel value instead of adding to it.
- **add_periodic_noise Function**: **add_periodic_noise(image, frequency=5, amplitude=10)** This function adds periodic noise to an image. Periodic noise is a type of noise that repeats at regular intervals.

Both functions return the image with the added noise. The images are normalized to the [0, 255].

1. **Convolution Functions:**
    - **convolution(image, kernel, stride=1, padding=0)**: This function performs 2D convolution on an input image using a given kernel, with options for specifying stride and padding.
        - Parameters:
            1. **image**: The input image represented as a NumPy array.
            2. **kernel**: The convolution kernel represented as a 2D NumPy array.
            3. **stride** (default: 1): Specifies the stride for the convolution operation.
            4. **padding** (default: 0): Specifies the amount of zero-padding applied to the image.
        - The function calculates the dimensions of the output based on the input image size, kernel size, stride, and padding. It considers both even and odd-sized kernels for correct output dimensions.
        - It creates a temporary image with padding and performs convolution by sliding the kernel over the image.
        - The result is an output image after convolution.
    - **gaussian_kernel(size, sigma)**: This function generates a 2D Gaussian kernel of the specified size and standard deviation.

- Parameters:
    1. **size**: The size of the kernel (the kernel is square).
    2. **sigma**: The standard deviation of the Gaussian distribution.
- The function calculates each element of the kernel based on the Gaussian formula and normalizes the kernel to ensure the sum of all values is 1.

2. **Image Upscaling and Downscaling Functions:**
    - **upscale_image(image, scale_factor)**: This function upscales an image by a specified scale factor using nearest-neighbor interpolation and applies Gaussian smoothing to the resulting image.
    - **downscale_image(image, scale_factor)**: This function downscales an image by a specified scale factor using Gaussian smoothing.

3. **Linear Filtering Functions:**
    - **apply_linear_filter(image, kernel)**: This function applies linear filters to an input image using the **convolution** function. It includes examples like average box filter, Gaussian blur filter, Laplacian filter, Sobel operators, and Prewitt operators.

4. **Image Loading and Testing:**
    1. **Convolution with Gaussian Kernel**: The image is convolved with a Gaussian kernel. The kernel is generated by the gaussian_kernel function (not shown in the code). The convolution operation is performed by the convolution function (also not shown in the code). The output image size is determined by the scale factor.
    2. **Image Loading**: The image is loaded using the cv2.imread function. The path to the image is specified in the function argument.
    3. **Image Downscaling and Upscaling**: The grayscale image is downscaled and upscaled using the downscale_image and upscale_image functions respectively. The scale factor for these operations is specified by the downscale variable.
    4. **Image Clipping and Type Conversion**: The pixel values of the downscaled and upscaled images are clipped to the range [0, 255] using the np.clip function. The images are then converted to 8-bit unsigned integer format using the astype function.
    5. **Image Display**: The original grayscale image, the downscaled image, and the upscaled image are displayed using matplotlib.pyplot. The images are displayed in a 1x3 subplot, with each image in a separate subplot.

5. **Creating Kernels:** Kernels are created as 2D numpy arrays. Each kernel has a specific purpose:
    - average_box_filter: This is a simple smoothing filter and is used for blurring the image and reducing noise. Each pixel in the image is set to the average value of its neighborhood.
    - gaussian_blur_filter: This is a Gaussian filter which is used for blurring the image and reducing noise. It uses a Gaussian function instead of a box shape function to calculate the transformation to apply to each pixel in the image.
    - laplacian_filter: This is used for edge detection. It calculates the Laplacian of the image given by the sum of second order derivatives and seeks to find regions of rapid intensity change.

- sobel_operator_h and sobel_operator_v: These are used for edge detection. The Sobel operator calculates the gradient of the image intensity at each pixel and gives the direction of the largest possible intensity increase.
- prewitt_operator_h and prewitt_operator_v: These are used for edge detection. The Prewitt operator is similar to the Sobel operator and is used for detecting vertical and horizontal edges in images.

6. **Loading the Image:** The image is loaded using the cv2.imread function and converted to grayscale using the cv2.cvtColor function.

7. **Applying the Kernels:** To apply the kernels, you would typically use the cv2.filter2D function (not shown in the code). This function convolves the kernel with the image. This means that for each pixel in the image, a neighborhood is defined around the center pixel, and the kernel is applied on this neighborhood (i.e., the kernel is multiplied element-wise with the pixel values in the neighborhood, and the results are summed up). Then, the center pixel is replaced by the summed up value. This process is repeated for all the pixels in the image to obtain the final output image.

8. **Frequency Domain Processing:**
    - The code then performs frequency domain processing on three sample images. It includes the following steps:
        - Apply the Fast Fourier Transform (FFT) to the input image to obtain the frequency domain representation.
        - Enhance the visualization by taking the logarithm of the magnitude.
        - Apply a mask to the magnitude of the Fourier transform to isolate specific frequency components.
        - Perform an inverse FFT on the masked frequency domain to obtain an image with only the isolated frequency components.
        - Subtract these isolated frequency components from the original frequency domain representation.
        - Perform an inverse FFT to get the clean image by removing the isolated frequency components.
9. **Displaying Results:** Finally, the code displays various images and their processing results using Matplotlib.

3. PROBLEMS / DIFFICULTIES / EXPLANATIONS

1. Convolution /downscaling and upscaling. We have obtained the optimal solutions.
2. OpenCV: It shows black, and we don't know why. Until now, it was outputting correctly, but for some reason we might be passing the parameters in a wrong way or something like that.
3. OpenCV and SciPy have a mismatch.
4. Template matching. It works more or less correctly.
    - Identify templates outputs everything except the L and the square. And the triangles is 2 times (that's because the template of the circle detects the triangle and there are two boxes).
    - If we change the value of threshold, the output changes. If we make its value smaller, we shorten the strictness of the method.

4. EXPERIMENTS / ALTERNATIVES

Regarding Fourier transform:

We do FT, the FFT version. We pass to Fourier and implement a Shift (0 frequency, takes the frequencies and places the most important ones in the center).

- LFS → Is for the plot to be visualized better.

Then, we need to convert the image from the space domain to the frequency domain (the frequency domain is the second image). Once this has been done, we can't see them because they are very small, but if we would do zoom or something like that, we would see that in the center there are 3, 4, 9, etc points respectively. These Little objects are the corrupting frequencies = frequencies that are not of the amplitude or value required.
Then, we had to look visually where these corrupting frequencies were and remove them. That is, to add a mask and multiply it by the corrupted image. That mask has 1s everywhere, ans 0s where the little points are placed.
Here is where we had the problem, since we did not know how to pass the point to the function.

Exploring: We passed the frequency domain and applied the mask. What happens? That the mask (0,9), we made it of 0s i and put 1s where the corrupting frequencies were. That was a mistake, but the lines that are in each iage showed as an output. So we thought that if we did the same in the inverse, it would work. But it did not.
So, finally, we keep the lines and substract them from the original. What happens? That we don't substract with the space domain, we do with the frequency domain.

Then, we have 4 functions:

- Get frequency domain
- Apply mask
- Inverse on corrupting frequencies → We pass it to the space domain for it to plot. So its only use is to plot.
- Subtract on the original… - frequency → clean = sf, where sf is the original image passed to frequency domain. WE take this and the frequency domain of the mask. Then, when we have the clean image, that I in frequency domain, we have to pass it to space domain; so we do the inverse in the cleaning.

The last step of the original and frequency: simplify…. It is basically because the frequencies are float and we want to take the real part of the number to simplify.

So, we have done this whole process in a different way, instead of removing the frequencies, we have done the image of the lines and have subtracted them.

The third image does not show perfect at all, but the others outputs quite reduced.