Harris Corner Detection

Theory

Chris Harris & Mike Stephens in their paper **A Combined Corner and Edge Detector** in 1988, so now it is called the Harris Corner Detector. He took this simple idea to a mathematical form. It basically finds the difference in intensity for a displacement of
(u,v)
:in all directions. This is expressed as below

$$E(u,v) = \sum_{x,y} \underbrace{w(x,y)}_{\text{window function}} \underbrace{[I(x+u,y+v)}_{\text{shifted intensity}} - \underbrace{I(x,y)}_{\text{intensity}}]^2$$

The window function is either a rectangular window or a Gaussian window which gives weights to pixels underneath

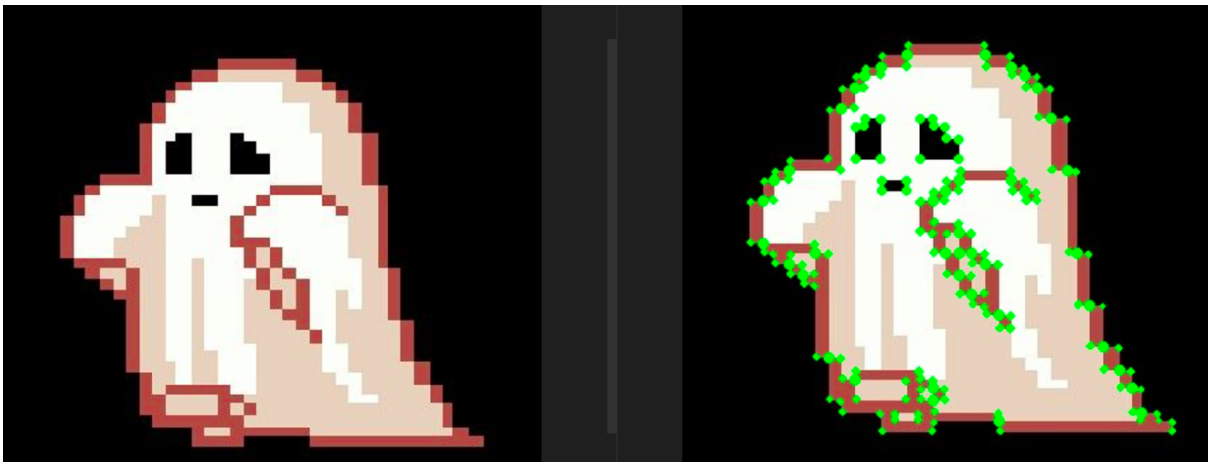
We have to maximize this function. That means we have to maximize the second term. Applying Taylor Expansion to the above equation and using some mathematical steps (please refer to any :standard text books you like for full derivation), we get the final equation as

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

$$R = \det(M) - k(\text{trace}(M))^2$$

- $\det(M) = \lambda_1 \lambda_2$
- $\text{trace}(M) = \lambda_1 + \lambda_2$
- λ_1 and λ_2 are the eigenvalues of M



Λ Operator

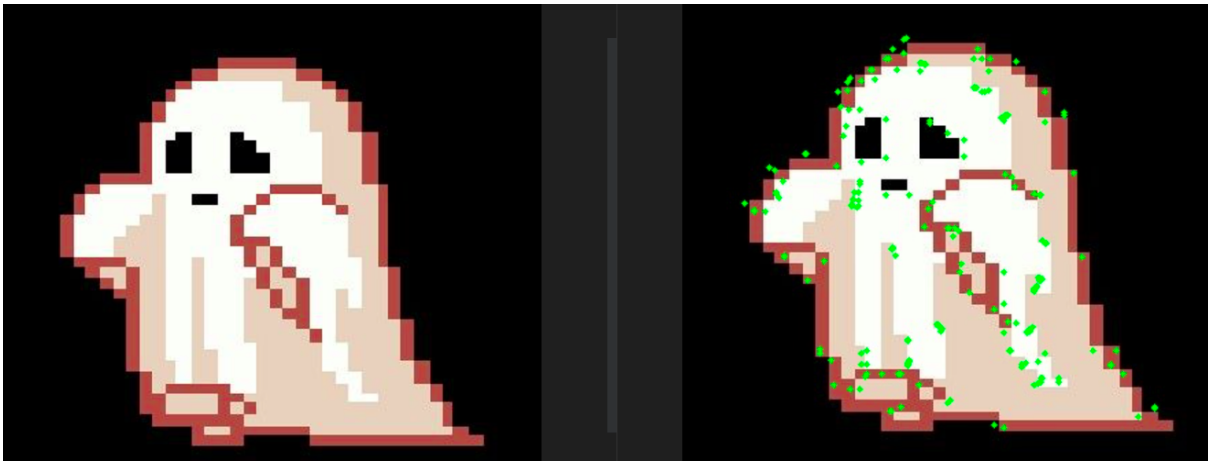
Theory

This function detects corners in an image using the Harris corner detector with the lambda_min criterion. The Harris corner detector is a method to identify corners in an image based on the changes in intensity in different directions. It calculates the eigenvalues of the structure tensor for each pixel in the image and computes the minimum eigenvalue lambda_min. This value is used as a criterion to detect corners. The function applies Gaussian smoothing to the image, computes the gradients using the Sobel derivative, computes the elements of the structure tensor, and computes lambda_min for each pixel. Non-maximum suppression is then applied to get the highest lambda values. The function returns a boolean matrix of the same size as the input image, with the points of interest (corners) masked with True.

λ_- is a variant of the “Harris operator” for feature detection

$$f = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2}$$
$$= \frac{determinant(H)}{trace(H)}$$

Returns:
A boolean matrix of the same size as the input image, with the points of interest (corners) masked with True.



SIFT (Scale-Invariant Feature Transform) algorithm

Overview

The SIFT algorithm works by detecting and describing distinctive local features in an image that are invariant to scale, rotation, and illumination changes. The algorithm consists of several steps:

- Scale-space extrema detection: The image is convolved with a Gaussian filter at different scales to create a scale-space representation of the image. Local extrema are then detected in the scale-space representation to identify potential feature locations.

- Keypoint localization: The potential feature locations are refined by fitting a 3D quadratic function to the scale-space representation around each extrema. Keypoints that are poorly localised or have low contrast are discarded.

- Orientation assignment: A dominant orientation is assigned to each keypoint based on the gradient magnitude and orientation in the local neighbourhood around the keypoint. This step ensures that the keypoint descriptor is invariant to rotation.

- Keypoint descriptor generation: A 128-dimensional descriptor is generated for each keypoint by taking the gradient magnitude and orientation in a 16x16 pixel patch around the keypoint. The descriptor is further normalized to ensure invariance to illumination changes.

The SIFT algorithm has been used in a wide range of applications, including object recognition, image stitching, and 3D reconstruction. It is known for its robustness to variations in viewpoint and lighting conditions, and its ability to handle cluttered and occluded scenes.

Coding First part

main function, `computeKeypointsAndDescriptors()`, which gives you a clear overview of the different components involved in SIFT. First,

we call `generateBaseImage()` to appropriately blur and double the input image to produce the base image of our "image pyramid", a set of successively blurred and downsampled images that form our scale space. We then call `computeNumberOfOctaves()` to compute the number of layers ("octaves") in our image pyramid.Now we can actually build the image pyramid. We start with `generateGaussianKernels()` to create a list of scales (gaussian kernel sizes) that is passed to `generateGaussianImages()`, which repeatedly blurs and downsamples the base image. Next we subtract adjacent pairs of gaussian images to form a pyramid of difference-of-Gaussian ("DoG") images. We'll use this final DoG image pyramid to identify keypoints using `findScaleSpaceExtrema()`. We'll clean up these keypoints by removing duplicates and converting them to the input image size. Finally, we'll generate descriptors for each keypoint via `generateDescriptors()`.

Our first step is `generateBaseImage()`, which simply doubles the input image in size and applies Gaussian blur. Assuming the input image has a blur of `assumed_blur = 0.5`, if we want our resulting base image to have a blur of `sigma`, we need to blur the doubled input image by `sigma_diff`. Note that blurring an input image by kernel size σ_1 and then blurring the resulting image by σ_2 is equivalent to blurring the input image just once by σ, where $\sigma^2 = \sigma_1^2 + \sigma_2^2$.

Then `computeNumberOfOctaves()` this function computes the number of times we can repeatedly halve an image until it becomes too small. Well, for starters, the final image should have a side length of at least 1 pixel. We can set up an equation for this. If y is the shorter side length of the image, then we have y / 2^x = 1, where x is the number of times we can halve the base image. We can take the logarithm of both sides and solve for x to obtain log(y) / log(2). So why does the -1 show up in the function above? At the end of the day, we have to round x down to the nearest integer (floor(x)) to have an integer number of layers in our image pyramid

Next we `generateGaussianKernels()`, which creates a list of the amount of blur for each image in a particular layer. Note that the image pyramid has `numOctaves` layers, but each layer itself has `numIntervals + 3` images. All the images in the same layer have the same width and height, but the amount of blur successively increases.

We `generateGaussianImages()` by starting with our base image and successively blurring it according to our `gaussian_kernels`. Note that we skip the first element of `gaussian_kernels` because we begin with an image that already has that blur value.

Finally, we `generateDoGImages()` by subtracting successive pairs of these Gaussian-blurred images. Careful — although ordinary subtraction will work here because we've cast the input image to `float32`, we'll use OpenCV's `subtract()` function so that the code won't break if you choose to remove this cast and pass in `uint` type images.

Then `computeKeypointsWithOrientations()`. The goal here is to create a histogram of gradients for pixels around the keypoint's neighbourhood. Note that we use a square neighbourhood here, as the OpenCV implementation does. One detail to note is the `radius_factor`, which is `3` by default. This means the neighbourhood will cover pixels within `3 * scale` of each keypoint, where `scale` is the standard deviation associated with the Gaussian weighting.

Next, we compute the magnitude and orientation of the 2D gradient at each pixel in this neighbourhood. We create a 36-bin histogram for the orientations — 10 degrees per bin. The orientation of a particular pixel tells us which histogram bin to choose, but the actual value we place in that bin is that pixel's gradient magnitude with a Gaussian weighting. This makes pixels farther from the keypoint have less of an influence on the histogram. We repeat this procedure for all pixels in the neighbourhood, accumulating our results into the same 36-bin histogram.

When we're all done, we smooth the histogram. The smoothing coefficients correspond to a 5-point Gaussian filter.

Now we look for peaks in this histogram that lie above a threshold we specify it. We localise each peak using quadratic interpolation — finite differences again.
We create a separate keypoint for each peak, and these keypoints will be identical except for their orientation attributes.
Then We sort and remove duplicates with `removeDuplicateKeypoints()`

Coding Second part

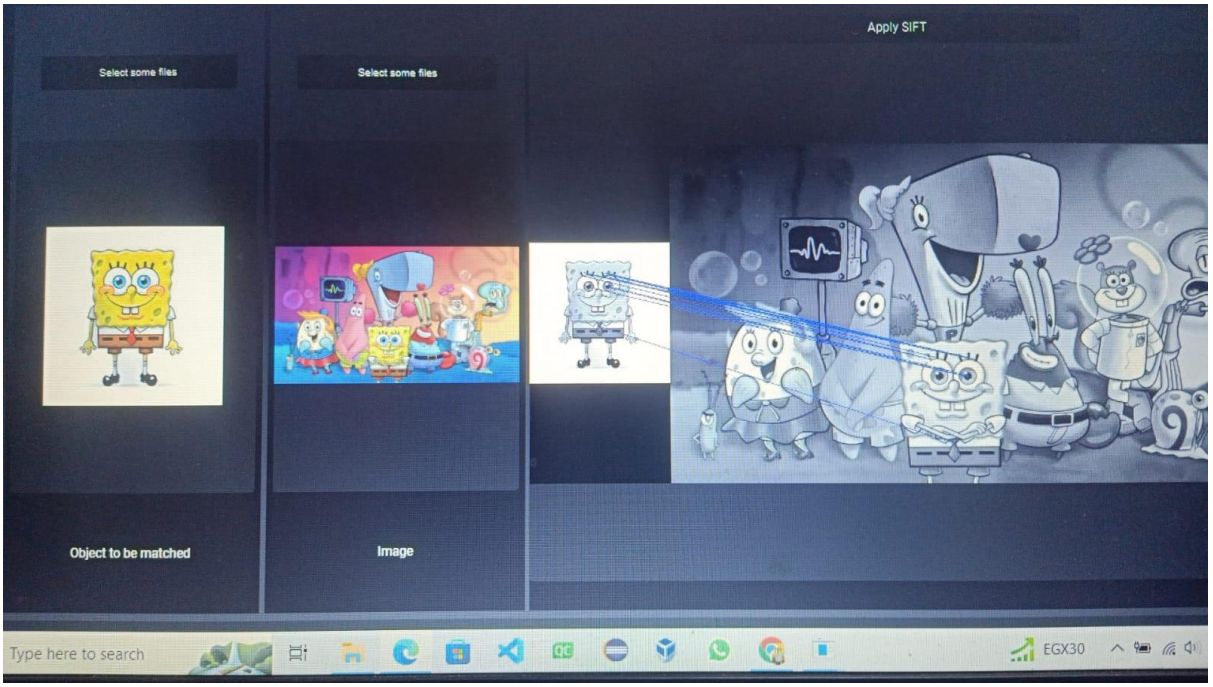
Finally, descriptor generation. Descriptors encode information about a keypoint's neighbourhood and allow comparison between keypoints. SIFT descriptors are particularly well designed, enabling robust keypoint matching.
For each keypoint, our first step is to create another histogram of gradient orientations. We consider a square neighbourhood (different side length this time) around each keypoint, but now we rotate this neighbourhood by the keypoint's angle.

Now comes the tricky part. Imagine that we take our square neighbourhood, a 2D array, and replace each pixel with a vector of length 36. The orientation bin associated with each pixel will index into its 36-length vector, and at this location we'll store the weighted gradient magnitude for this pixel.

Our last step is to flatten our smoothed 3D array into a descriptor vector of length 128. Then we'll apply a threshold and normalise. In the OpenCV implementation, the descriptor is then scaled and saturated to lie between 0 and 255 for efficiency when comparing descriptors later.

Finally Given a template image, the goal is to detect it in a scene image and compute the homography. We compute SIFT keypoints and descriptors on both the template image and the scene image, and perform an approximate nearest neighbours search on the two sets of descriptors to find similar keypoints. Keypoint pairs closer than a threshold are considered good matches.

Output example of SIFT algorithm:



SSD

Equation:-

$$\sum (I(x1) - I(x0))^2$$

What's SSD?

The Sum of Squared Differences (SSD) is a method used in computer vision and image processing to compare two images or image features. It is often used in conjunction with the Scale-Invariant Feature Transform (SIFT) algorithm.

SIFT is a feature detection algorithm that identifies and describes local features in an image that are invariant to scale, rotation, and illumination changes. These features can be used for object recognition, image matching, and other computer vision tasks.

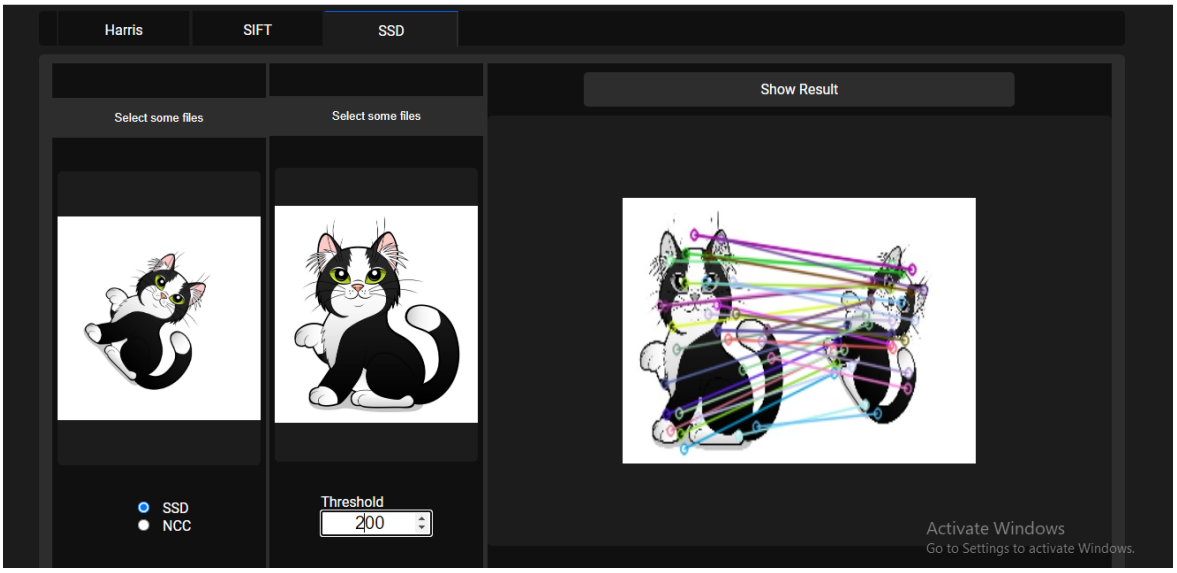
To use the SSD method with SIFT features, the algorithm first identifies a set of key points or interest points in each image. These key points are then described using a set of SIFT features, which include a descriptor vector that represents the intensity and orientation of the gradient at the key point.

The SSD method then calculates the sum of the squared differences between the descriptor vectors of the corresponding key points in the two images being compared. This measures the similarity between the two sets of SIFT features and can be used to match the two images or to identify objects in the images.

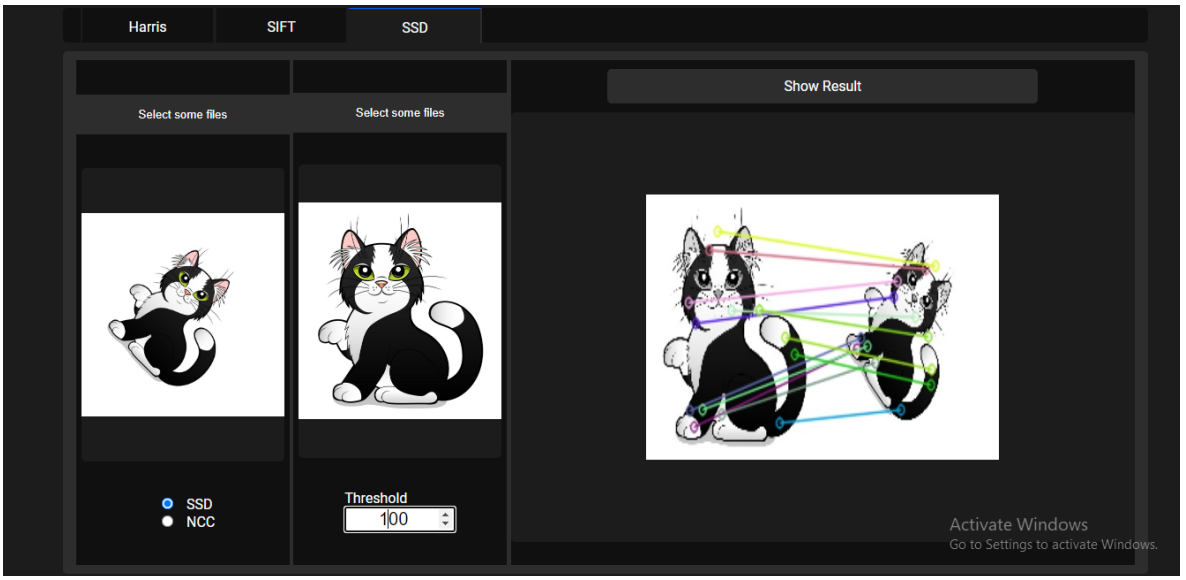
In practice, the SSD method is often used in combination with other algorithms and techniques to improve the accuracy and robustness of feature detection and matching. Other methods may include feature selection, filtering, and clustering to reduce noise and increase the discriminative power of the SIFT features.

Examples:-

Result at threshold 200:-



Result when threshold 100:-



So when threshold is smaller smaller matches appear as we take the difference less than this threshold

Time:-

```
SSD
the time taken by algorithm is 54.39532947540283
```

NCC

Equation:-

$$\frac{1}{n-1}\sum_{i}\frac{(f(x_i)-\bar{f})(g(x_i)-\bar{g})}{\sigma_f\sigma_g}$$

What's NCC?

Normalized Cross Correlation (NCC) is another technique used in computer vision and image processing to compare two images or image features, and it is also used with the Scale-Invariant Feature Transform (SIFT) algorithm.

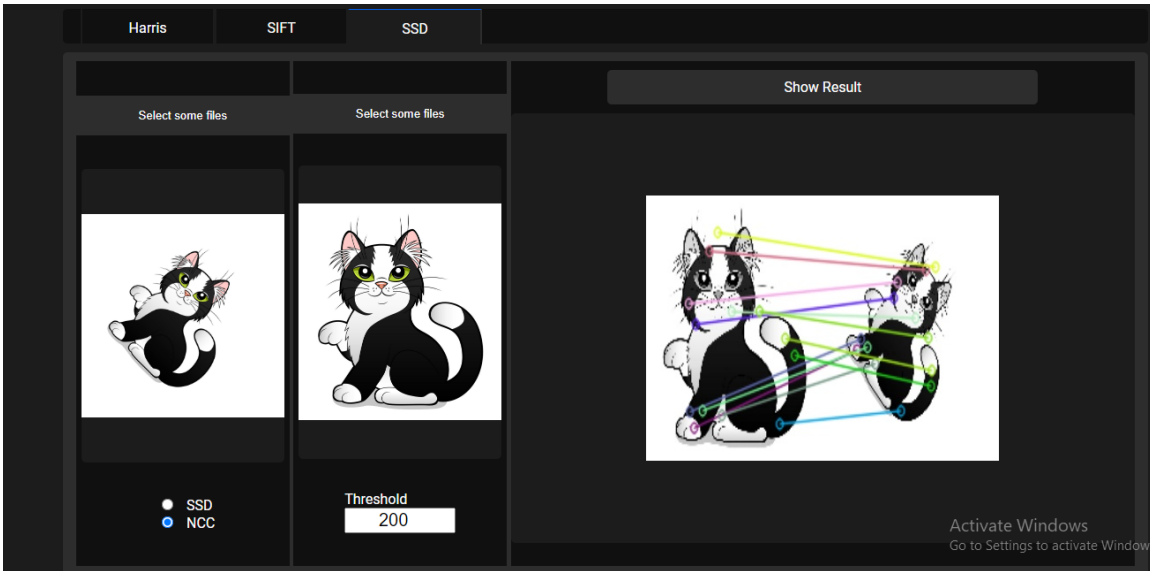
NCC is a similarity measure that compares the similarity between two feature vectors by calculating their cross-correlation after normalizing them to have zero mean and unit variance. This normalization step ensures that the NCC value is invariant to the overall brightness and contrast of the image.

To use NCC with SIFT features, the algorithm first identifies a set of key points or interest points in each image. These key points are then described using a set of SIFT features, which include a descriptor vector that represents the intensity and orientation of the gradient at the key point.

The NCC method then calculates the normalized cross-correlation between the descriptor vectors of the corresponding key points in the two images being compared. This measures the similarity between the two sets of SIFT features and can be used to match the two images or to identify objects in the images.

In practice, NCC is often used as an alternative to the SSD method, as it can be more robust to changes in illumination and contrast. However, NCC can be computationally expensive, especially when dealing with large images or a large number of features, so other techniques may be used to speed up the calculation, such as pyramid matching or approximate nearest neighbor search.

Examples:-



Time:-

```
NCC
the time taken by algorithm is 36.76842141151428
1
```