

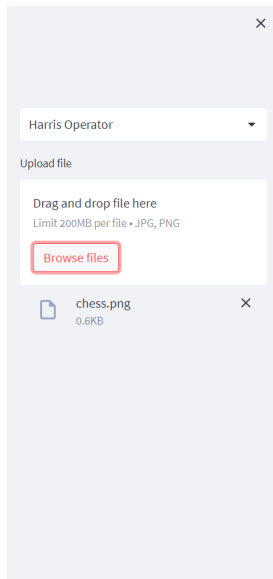
**Assignment 3 Report**  
**Team 14**

Name	Section	Bench Number
Hanya Ahmad	2	51
Nourhan Sayed	2	47
Ahmed Ashraf	1	1
Mohammed Salah	2	19
Ammar Mostafa	2	1

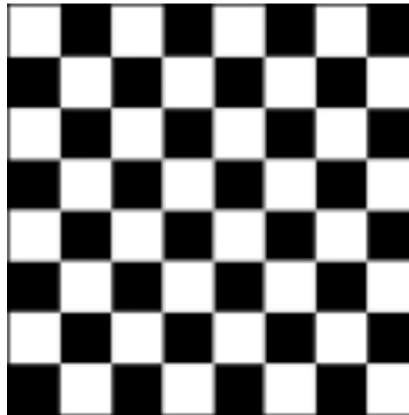
**1. Harris Operator**

- a. **Implementation:** It is a technique used in computer vision for detecting corners, edges, and extracting features in images. To implement it , we firstly convert the image to grayscale then , we compute the the derivation using sobel operator, then we compute the second order moments of the image then, compute the harris response function then threshold the response function to obtain a binary image of corner points then apply none maximum suppression to eliminate multiple responses.

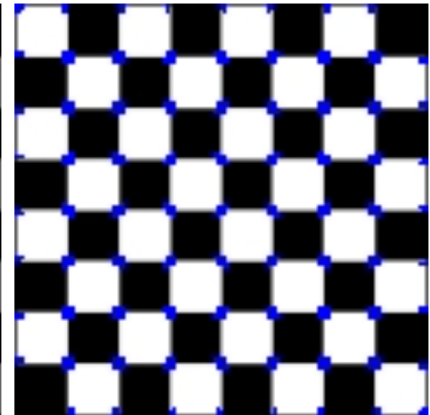
b. **output**



Input Images



Output Images



Computation time: 0.02099895477294922

### c. time

Photo size	time
640X480	5.76

### d. comments

The Harris operator involves computing the eigenvalues of a matrix for every pixel in the image, which can be computationally expensive for large images. This can limit the real-time performance of the algorithm for large-scale image processing tasks.

## 2. Lambda

### a. Implementation:

we firstly convert the image to gray scale and get the spatial derivative then we compute the H matrix from the gradient entries and then compute the eigen value of each pixel  
Then we apply non-maximum suppression to eliminate multiple suppression

### b. Output

Lambda Operator

Upload file

Drag and drop file here

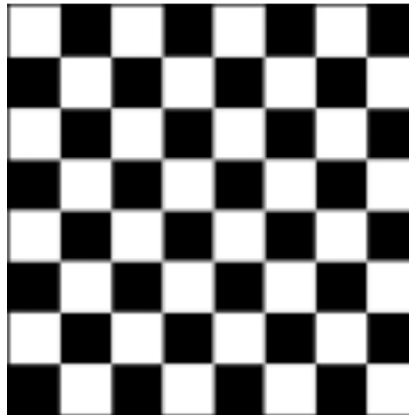
Limit 200MB per file • JPG, PNG

Browse files

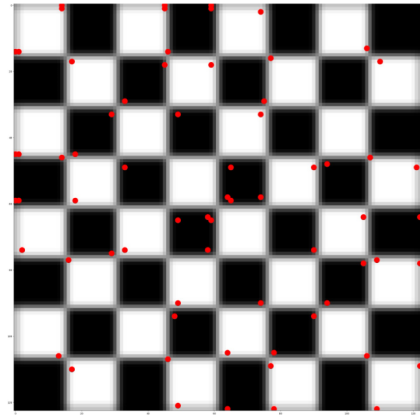
chess.png

0.6KB

Input Images



Output Images



Computation time: 1.6651644706726074

c. time

Photo size	time
640X480	6.65

d.

e. comment

it's take more time due to more square root calculation to get the eigen value so it's slow than harris operator

## 2. SIFT

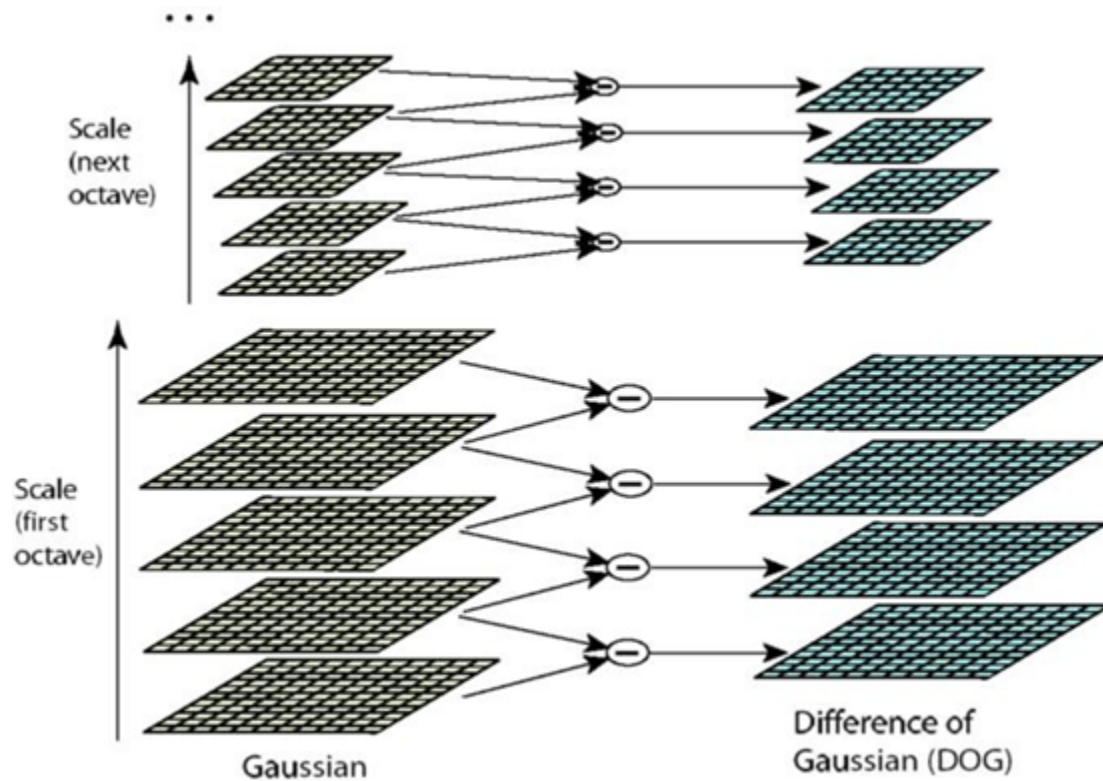
SIFT key points are useful due to their distinctiveness. This distinctiveness is achieved by assembling a high-dimensional vector representing the image gradients within a local region of the image. The key points have been shown to be invariant to image rotation and scale and robust across a substantial range of affine distortion, the addition of noise, and change in illumination.

The fact that key points are detected over a complete range of scales means that small local features are available for matching small and highly occluded objects, while large key points perform well for images subject to noise and blur.

### Scale Space Construction:

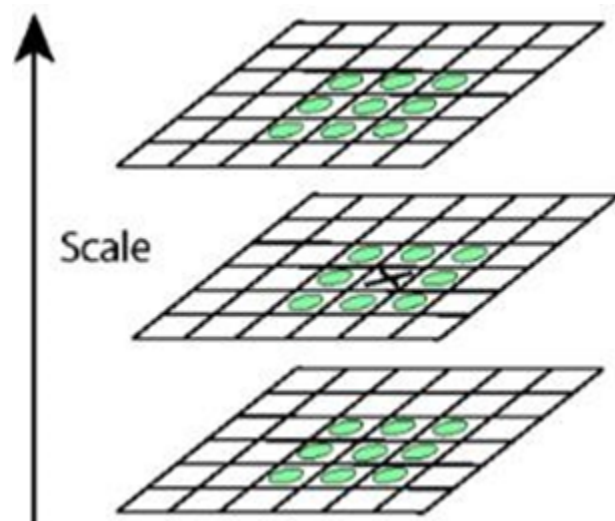
For each octave of scale space, the initial image is repeatedly convolved with Gaussians to produce the set of scale-space images shown on the left. Adjacent Gaussian images are subtracted to produce the

difference-of-Gaussian images on the right. After each octave, the Gaussian image is down-sampled by a factor of 2, and the process is repeated.



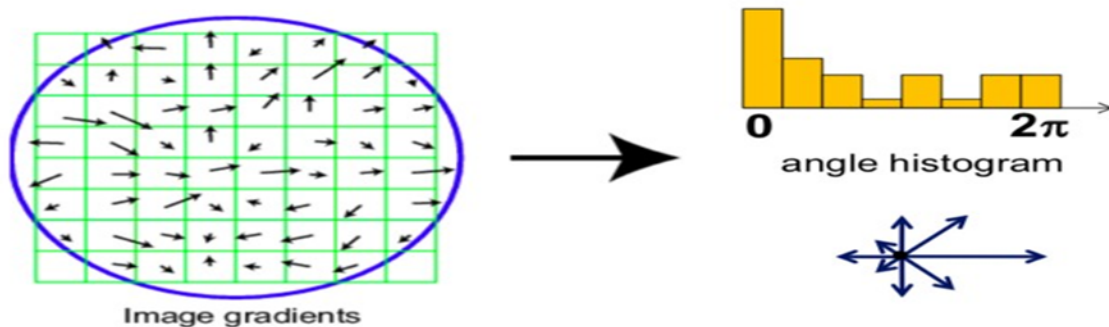
### Scale Space Extrema Detection:

Maxima and minima of the difference-of- Gaussian images are detected by comparing a pixel (marked with X) to its 26 neighbors in 3x3 regions at the current and adjacent scales (marked with circles).



### Key point descriptor:

- Take 16x16 square window around detected feature
- Compute edge orientation (angle of the gradient -  $90^\circ$ ) for each pixel
- Throw out weak edges (threshold gradient magnitude)
- Create histogram of surviving edge orientations



#### **a. Implementation :**

At first we call function called `computeKeyPointsAndDescriptors` and this function is responsible for all of the implementation of the SIFT technique it calls function called `base_generator` and first it resize the image and calculating the sigma to blur with Gaussian filter and then it returns this image and added to function `octaves_numm` and that to see how many images can be got from this image and that to know number of images with different sizes and then we call another function which is `kernel_sigmas` and get it finds number of images in each octave then then calculating different sigmas to apply them to the images in each octave and the same for each one and that to have different types of blurred images and stay the same for others then we use `generateGaussianImages` and here we add the image that we got and it loops on all the images in octave and do blurring for them to get the pyramid of the octave images and then adding the DOG function. The `DoG_images` function then generates a difference-of-Gaussians (DoG) image pyramid from the Gaussian image pyramid by subtracting adjacent Gaussian images within each octave then function find `scaleSpaceExtrema` it finds the pixel positions of all scale-space extrema in the difference-of-Gaussians

(DoG) image pyramid. the function begins by calculating a threshold and then loops over each triplet of adjacent DoG images and compares its value with its 8 neighbors and the 9 in the other two adjacent in the pyramid then the function

>>isPixelAnExtremum and this function uses bitmasking to find if it is the correct position (local min or local max ) or not then function

>>The gradient\_comp computes the gradient vector at a given pixel in the DoG image pyramid using central differences then function

>>hessian\_comp it computes the Hessian matrix at a given pixel in the DoG image pyramid and it's second order derivative matrix>>then we use function

>>quadratic\_fit function it implements the keypoint localization step of the SIFT algorithm The function first sets a flag to check if the extremum is outside the image. It then iteratively refines the pixel location by solving the system of linear equations obtained from the Taylor expansion of the DoG function and checks if the extremum has converged by comparing the size of the extremum update vector with a threshold. If the extremum update moves the pixel location outside the image, the function sets the flag to True and breaks out of the loop. Then we use

>>computeKeypointsWithOrientations it computes the dominant orientation(s) for each keypoint detected in the previous step of the SIFT algorithm and that by calculating the number of orientation bins and the histogram then it returns a list of keypoints with orientations. Then we use function

>>compare\_keypoints function takes in two keypoint objects First\_keypoint and Second\_keypoint and returns True if First\_keypoint is less than Second\_keypoint. Then we use

>>get\_unique\_keypoints it takes in a list of keypoint objects keypoints and returns a sorted list of unique keypoints. Then we use function

>>removeDuplicateKeypoints function takes in a list of keypoint objects keypoints and returns a sorted list of unique keypoints.

>>The removeDuplicateKeypoints function takes a list of keypoint objects and returns a sorted list of unique keypoints. It checks if the length of keypoints is less than 2 and returns keypoints if it is. The function sorts the keypoints and initializes a list with the first keypoint from the sorted list. It iterates over the rest of the keypoints and adds each one to the unique\_keypoints list if it has a different (x, y) location, size, or orientation compared to the previous unique keypoint. The function prints the shape of the unique\_keypoints array for debugging purposes. Overall, the

function is similar to `get_unique_keypoints` but with an additional check for the length of keypoints.

>>The `convertKeypointsToInputImageSize` function takes a list of keypoint objects and adjusts their point, size, and octave to the input image size. It initializes an empty list for the adjusted keypoints and iterates over each keypoint. For each keypoint, it halves the (x, y) location, size, and octave to adjust for the change in image size, modifies the octave to match the input image size, and adds the modified keypoint to the adjusted keypoints list. The function prints the shape of the adjusted keypoints array and returns the list of adjusted keypoints.

>>The `keypoints_orientations` function takes a keypoint object, octave index, Gaussian image, and optional parameters to return a list of keypoint objects with assigned orientations. It initializes an empty list, computes standard deviation based on the keypoint size and octave index, and creates a Gaussian-weighted circular window around the keypoint. The function computes gradient magnitude and orientation for each pixel in the window, weights the gradient magnitude, and accumulates the weighted magnitudes into an orientation histogram with `num_bins` bins. The function blurs the histogram and finds peaks greater than `peak_ratio` times the maximum value. It performs quadratic interpolation to find the orientation angle that corresponds to each peak and creates a new keypoint object with the interpolated orientation angle. The function adds this new keypoint to the `orientation_keypoints` list.

>>The `unpackOctave` function extracts the octave, layer, and scale associated with a keypoint object

>>The `generateDescriptors` function takes in a list of keypoints, a set of Gaussian images, and other parameters to compute a set of descriptors for each keypoint. The function extracts information about the octave, layer, and scale associated with each keypoint using the `unpackOctave` function. It then computes a descriptor for each keypoint using gradient magnitude and orientation within a window around the keypoint, weighted histogram of gradients, trilinear interpolation, thresholding, and normalization. The descriptor vector is a flattened version of the histogram, multiplied by 512, rounded, and saturated to convert from float32 to unsigned char. The function returns the descriptors as a numpy array.

>>The `computeKeypointsAndDescriptors` function takes an input image and various parameters, then returns a set of

keypoints and descriptors for the image. It converts the input image to grayscale, generates a base image, and creates a pyramid of Gaussian images using the `base_generator`, `kernel_sigmas`, and `generateGaussianImages` functions. It then generates a pyramid of difference-of-Gaussian images using the `DoG_images` function. Keypoints are extracted from the DoG images using the `findScaleSpaceExtrema` function and redundant keypoints are removed using the `get_unique_keypoints` function. Descriptors are then computed for each keypoint using the `generateDescriptors` function. The function can also save an output image with the keypoints overlaid on the original image if the `save_img` parameter is set to 1. Finally, the function returns the keypoints and descriptors computed for the input image.

## b. Output

×

SIFT Operator

Upload file

Drag and drop file here  
Limit 200MB per file • JPG, PNG


Browse files

Sift2.jpg


46.6KB

×

Input Images



Output Images



Computation time: 18.84829616546631

## c. Comments

### 3. Feature Matching

#### a. SSD

##### i. Implementation:

Our function takes three parameters: original image's descriptor, target image's descriptor, and method: SSD or



NCC. The descriptors are those obtained from our implemented SIFT function. For the SSD case, we loop over each feature descriptor of the first -original- image, while initiating a distance variable of  $-\infty$  and looping over all the features descriptors of the second -target- image in each iteration, where a function is called that calculates the sum of squared distance each iteration according to the following equation:

$$\text{SSD} = \sum_i (I_1(x_1) - I_0(x_0))^2$$

But our calculateSSD() function returns the negative of the SSD score; so, if the returned score is bigger than the distance, we update the distance variable to hold the score. So, when the inner loop is finished, our distance variable is equal to the highest negative score, which is equal to the lowest positive score. Afterwards, we append the index of the first image's feature and the corresponding index of the second image's feature in a list after each iteration of the outer loop. The function finally returns the list of all matches in the two images. This negative sign in the score is essential so that we can implement both distance metrics in the same function with minimal changes to reduce code repetition.

## ii. Output

Browse files

Sift.jpg  
104.0KB

Upload second file

Drag and drop file here

Limit 200MB per file • JPG, PNG

Browse files

Sift\_ori.jpg  
106.1KB

Choose Distance Metric  
☒ SSD  
☐ NCC

Choose Image Quality (decreasing quality decreases computation time)  

256

64256

Choose Number Of Matches  

80

10280

Input Images

Output Images

Computation time: 43.64423179626465

Made with Streamlit

### iii. Comments

SSD distance metric is faster to calculate but has some limitations, as it will result in a match even if there is not one, as it sees the lowest score as a match.

## b. NCC

### i. Implementation

We use the same function as the one discussed above in the case of SSD, but we call a function that calculates Normalized Cross-Correlation distance metric according to the following equation:

$$NCC = \frac{1}{n-1} \sum_i \frac{(f(x_i) - \bar{f})(g(x_i) - \bar{g})}{\sigma_f \sigma_g}$$

Then, if the returned NCC value is bigger than the distance variable's value, we update the distance variable to hold the NCC score. The rest of the function is executed as previously discussed in detail in the SSD case.

### ii. Output

Browse files

Sift.jpg

104.0KB

×

Upload second file

Drag and drop file here

Limit 200MB per file • JPG, PNG

Browse files

Sift\_ori.jpg

106.1KB

×

Choose Distance Metric

☐ SSD

☒ NCC

Choose Image Quality (decreasing quality decreases computation time)

256

64

256


Choose Number Of Matches

89

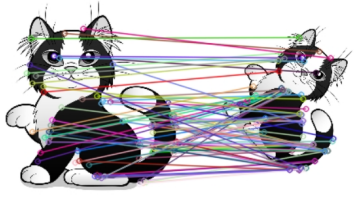
10

280

Input Images



Output Images



Computation time: 61.20479774475098

Made with Streamlit

### iii. Comments

NCC distance metric calculation takes more computational time but is robust to the linear variations in brightness due to different lighting conditions and so it provides more accurate results.