

# Scale-Invariant Feature Transform

---

## Mid-Term Progress Report

---

### Team 4: Drowning in Inefficiency

- Kannav Mehta (2019101044)
- Raj Maheshwari (2019101039)
- Triansh Sharma (2019101006)
- Aashwin Vaish (2019114014)

[Github link](#)

## Code

---

### How to run:

Although the instructions are clearly written in the [README](#), let us rewrite them here.

1. Create an empty build directory.

```
mkdir build && cd build
```

2. Call cmake inside it.

```
cmake ..
```

3. Now make.

```
make
```

4. Finally run the executable along with the path of your image as an argument.

```
./smai <path>
```

### Code structure

The code is written in **C++**.

The sift implementation consists of primarily 3 files - the main.cpp, the sift.cpp and the sift.hpp

1. **main.cpp** -- Here we read the image file from the path passed in as input argument. The image is converted to GRAYSCALE and passed to the sift handler. A window is opened to display the output image.
2. **sift.cpp** -- All the major code resides here. Everything from core logic to algorithm implementation lies here.
3. **sift.hpp** -- This is the header file which separates all the variable and function declarations from the main logic. We define all the critical constants here so tweaks can easily be made while analysing performance.

Here are some of the constants that we used:

```

SCALES = 3 // number of scales per octave
BORDER = 5 // boundary region where no keypoints will be detected
contrast_threshold = 0.04
threshold = (0.5 * contrast_threshold / SCALES * 255)
SIGMA = 1.6
assumed_blur = 0.5
IMAGES = SCALES + 3
EIGEN_VALUE_RATIO = 10.0 // To eliminate edge responses
THRESHOLD_EIGEN_RATIO = pow((EIGEN_VALUE_RATIO + 1), 2) / EIGEN_VALUE_RATIO
BINS = 36 // Number of bins in histogram 1 per 10 degrees
PEAK_RATIO = 0.8
SCALE_FACTOR = 1.5
RADIUS_FACTOR = 3

```

These hyperparameters were chosen by the discretion of the authors in the research paper by Lowe, and we trust them.

Although we are using OpenCV, we restricted its usage to only the following:

- Reading and writing images
- Mathematical classes and functions (`cv::Mat` classes, `cv::Range` and `cv::parallel_for` functions etc.)
- Gaussian Blur
- Interpolation

## sift\_handler class

The sift handler class contains the various different methods that get sequentially called in the `exec` function. As a first step, we perform some preprocessing steps in the constructor of this class.

Number of octaves is calculated:

```

octaves = (size_t)std::round(std::log2((double)std::min(sz.width, sz.height)))-1

```

We compute the log base 2 of the minimum of the dimensions to find how times the image can be halved before it is reduced to a size of atmost 3 pixels.

A base image is created by resizing the original image to twice its size using `inter linear` interpolation. Then we apply gaussian blur on it so our final image has a blur of `SIGMA`.

## sift\_handler::exec()

This is basically the **execute** function which orderly calls all the different functions on the image.

1. Generate the **Gaussian** images
2. Generate the **Difference of Gaussian** images
3. Generating the **Scale Space Extremas**
4. **Cleaning** out the keypoints

Since we care about latency, we have timed these important functions and made efforts to reduce processing time. Finally, display the image upon which are nicely plotted the keypoints captured by our program.

## **sift\_handler::gen\_gaussian\_images()**

Here we generate gaussian kernels that successively blur the image. Each octave has `IMAGES` number of images. These images differ by a constant amount of blur. The third last image of the octave gets carried forward to the next octave after squeezing it to half its size.

## **sift\_handler::gen\_dog\_images()**

The DoGs are simply the differences between successive images in a octave. When plotted, they seem like edge maps.

## **sift\_handler::gen\_scale\_space\_extrema()**

Since space scale extremum is an operation that needs to be carried out for all the pixels of each image in every octave, we decided to perform parallel processing on these pixels with the help of threading. This greatly improves the performance of our code and makes it much more efficient because of better use of computational resources.

## **sift\_handler::clean\_keypoints()**

Upon running the algorithm, lots of keypoints are generated. Many of these of keypoints lie close to each other and are unnecesassry repetitions. In order to focus on different features of an image, these need to be cleaned and filtered out. We try to club different keypoints into small subsets by removing those keypoints which lie very close to one another. First we sort them using a custom comparator function based on their coordinate, size, octave etc, and then we find unique points by considering 2 points distanced by a single pixel as nearly equal. Finally, we scale these keypoints back by half to match the original image.

## **sift\_handler::scale\_space\_extrema\_parallel class**

We are using multi-threading, using opencv's builtin `parallel_for_` to speed up finding the scale space extrema points and computing keypoints. This class contains all the functions that find keypoints of a  $3 \times 3 \times 3$  pixel cube centered on a pixel  $(i, j)$ .

## **sift\_handler::scale\_space\_extrema\_parallel::is\_pixel\_extremum()**

A pixel cube is the 1 pixel neighbourhood of an pixel in an image. This is the layer of  $3 \times 3$  pixels above and below it and  $3 + 2 + 3$  pixels around it. A pixel extremum is one which is greater or lesser than all 26 pixels around it. For such pixel extremas, we perform keypoint localization and find the orientation of this keypoint.

## **sift\_handler::scale\_space\_extrema\_parallel::localize\_extrema()**

In order to localize a keypoint, a quadratic model is fit to the  $3 \times 3 \times 3$  pixel cube centered on the keypoint pixel. `get_gradient()` and `get_hessian()` functions implement second-order central finite difference approximations in three dimensions.

The localization of the extremum is defined as:

$$\hat{x} = -\frac{\partial^2 D}{\partial x^2}^{-1} \frac{\partial D}{\partial x}$$

In code, this resolves to:

```
bool temp = cv::solve(hess, grad, res, cv::DECOMP_NORMAL);
if (!temp) return 0;
res *= -1;
```

Here, `hess` is the hessian and `grad` is the gradient while the resultant `x` gets stored in `res`. `temp` denotes whether or the equation was solved correctly.

Following updates are made in each iteration.

```
j += (int)std::round(res[0][0]);
i += (int)std::round(res[1][0]);
img += (int)std::round(res[2][0]);
```

Convergence is achieved for following condition:

```
std::abs(res[0][0]) < 0.5 &&
std::abs(res[1][0]) < 0.5 &&
std::abs(res[2][0]) < 0.5
```

The extremum must be within the search zone:

```
BORDER < i and i <= sz.width - BORDER and
BORDER < j and j <= sz.height - BORDER and
1 < img and img <= IMAGES - 2
```

Threshold is defined as:

$$\tau = \frac{Tr(H)^2}{Det(H)} < \frac{(r+1)^2}{r}$$

if  $\tau$  is less than `THRESHOLD_EIGEN_RATIO` then we compute the following keypoint:

```
double keypt_octave = oct + (1 << 8) * img + (1 << 16) * std::round((res[2][0] +
0.5) * 255);
double keypt_pt_x = (j + G(res, 0, 0)) * (1 << oct);
double keypt_pt_y = (i + G(res, 1, 0)) * (1 << oct);
double keypt_size = SIGMA * (std::pow(2, img + res[2][0]) / (1.0 * SCALES)) * (1
<< (oct + 1));
double keypt_response = std::abs(value);
kpt = cv::KeyPoint(keypt_pt_x, keypt_pt_y, keypt_size, -1, keypt_response,
keypt_octave);
```

## `sift_handler::scale_space_extrema_parallel::get_keypoint_orientations()`

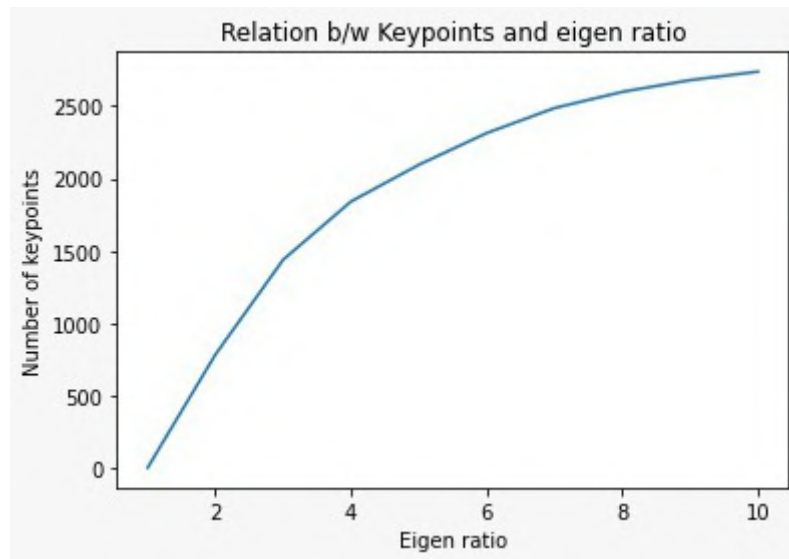
To calculate the orientation(s) of a keypoint, we use the histograms of gradients for pixels in a square neighbourhood centered on the keypoint. For each pixel in this neighbourhood, we calculate the magnitude and orientation of the 2D gradient. Using a 36-bin histogram (each  $10^\circ$ ). The value put into the bin is the gradient magnitude \* gaussian weight for that pixel, so that farther pixels have less influence than central ones.

At last we smooth out the histogram using 5 point gaussian kernel.

## Statistics

We modified the `EIGEN_VALUE_RATIO` from `[1, 2, 3, ..., 10]` and calculated the number of keypoints obtained using it for image of size  $512 \times 512$ .

The below plot describes the nature of curve.



We also calculated DoG and found the keypoints in variety of images with different dimensions. Below are the results shown for a few test images.

**Note:** The time measured is highly dependent on configurations of system. Hence, the time might be not reproducible. The system configuration used to calculate the total time taken by program to run has *i7 9th Gen processor with 12 CPU cores and 16GB RAM*.

## Image 1

**Size:**  $512 \times 512$

**Keypoints:** 2739

**Time taken:** 0.868 seconds



Input image



Keypoints

DoG pyramid

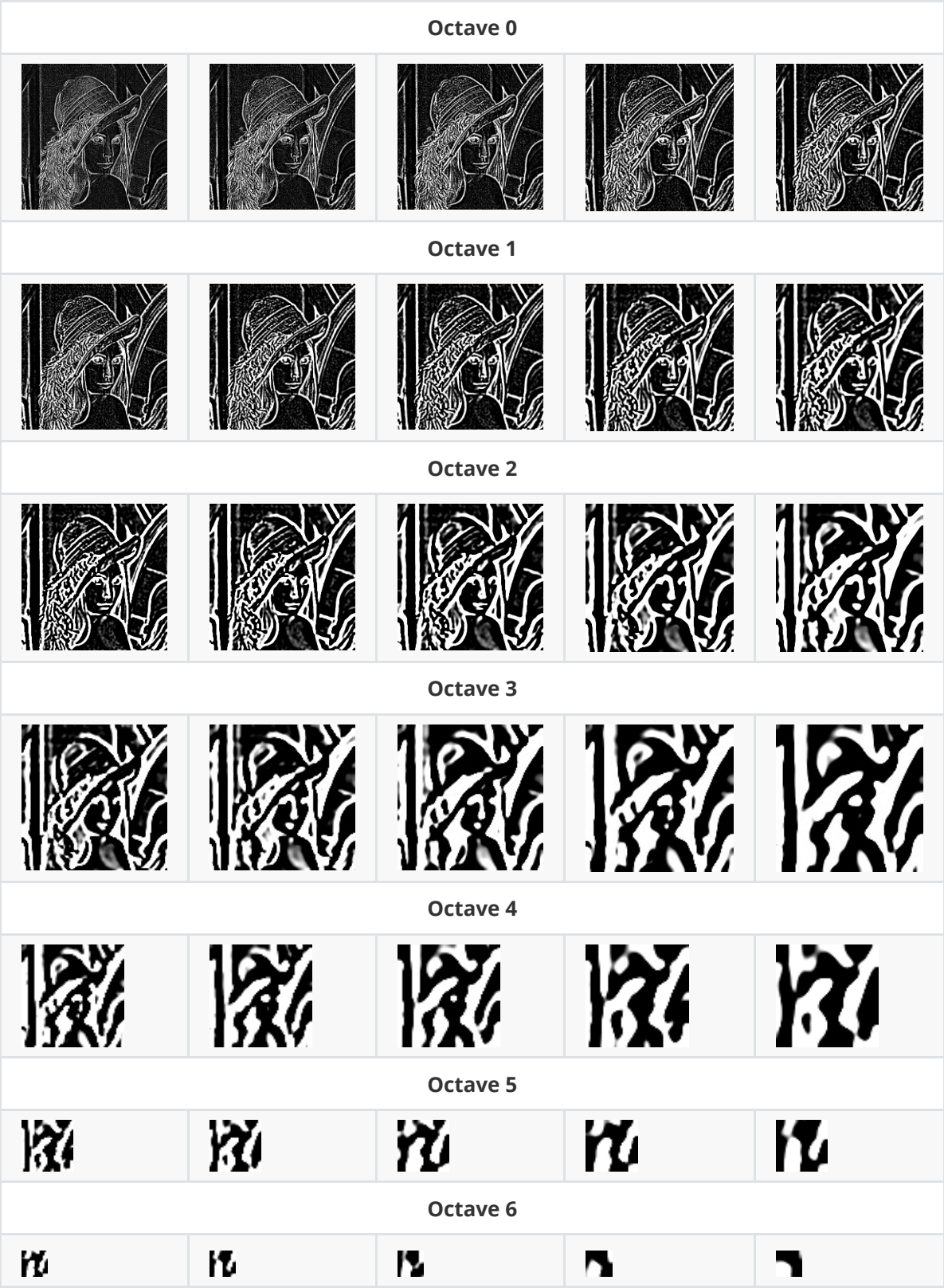


Image 2

Size: 910 × 683  
Keypoints: 6030  
Time taken: 2.175 seconds





Input image



Keypoints

**DoG pyramid**



























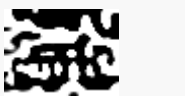



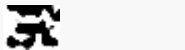
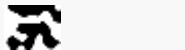
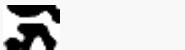
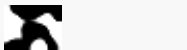
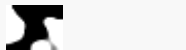
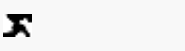
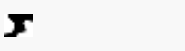
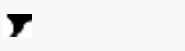
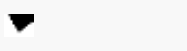
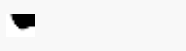
Octave 0				
				
Octave 1				
				
Octave 2				
				
Octave 3				
				
Octave 4				
				
Octave 5				
				
Octave 6				
				
Octave 7				
				

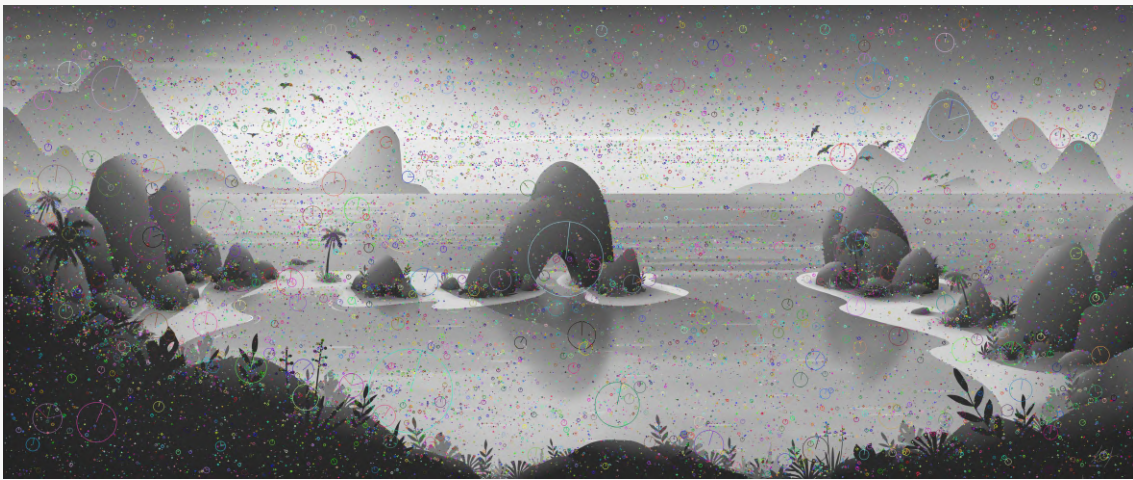
Image 3

Size: 3440 × 1440  
Keypoints: 31370  
Time taken: 17.411 seconds





Input image



Keypoints

**DoG pyramid**



## Further Plan

- We are still left with generating descriptors from the keypoints of the image. When done, we will complete the implementation of SIFT and then implement feature matching across two images using [flann](#) library to implement KNN.
- Further, we will try to implement the image stitching algorithm using the features detected by our SIFT implementation.