Scale Invariant Feature Transform

-And it is not just scale invariant

Scale-invariant feature transform (or **SIFT**) is an algorithm in computer vision to detect and describe local features in images. The algorithm was published by David Lowe in 1999.

For any object in an image, interesting points on the object can be extracted to provide a "feature description" of the object. This description, extracted from a training image, can then be used to identify the object when attempting to locate the object in a test image containing many other objects. To perform reliable recognition, it is important that the features extracted from the training image be detectable even under changes in image scale, noise and illumination.

Another important characteristic of these features is that the relative positions between them in the original scene shouldn't change from one image to another. For example, if only the four corners of a door were used as features, they would work regardless of the door's position; but if points in the frame were also used, the recognition would fail if the door is opened or closed. Similarly, features located in articulated or flexible objects would typically not work if any change in their internal geometry happens between two images in the set being processed. However, in practice SIFT detects and uses a much larger number of features from the images, which reduces the contribution of the errors caused by these local variations in the average error of all feature matching errors.

For the project undertaken, we are required to find certain keypoint descriptors of the image. In order to achieve this, we shall make use of the SIFT algorithm.SIFT isn’t just scale invariant. You can change the following, and still get good results:

- Scale (duh)
- Rotation
- Illumination
- Viewpoint

The general steps involved in doing so using the SIFT Algorithm can be described as follows:

1. **Constructing a scale space:**
   This is the initial preparation.Internal representations of the original image are created to ensure scale invariance. This is done by generating a “scale space”.

2. **LoG Approximation**:
   The Laplacian of Gaussian is great for finding interesting points (or key points) in an image. But it’s computationally expensive. So we approximate it using the representation created earlier.

3. **Finding keypoints**:
   With the above approximation, we now try to find key points. These are maxima and minima in the Difference of Gaussian image we calculate in step 2

4. **Get rid of bad key points**:
   Edges and low contrast regions are bad keypoints. Eliminating these makes the algorithm efficient and robust.

5. **Assigning an orientation to the keypoints**:
   An orientation is calculated for each key point. Any further calculations are done relative to this orientation. This effectively cancels out the effect of orientation, making it rotation invariant.

6. **Generate SIFT features**:
   Finally, with scale and rotation invariance in place, one more representation is generated. This helps uniquely identify features. Lets say you have 50,000 features. With this representation, you can easily identify the feature you’re looking for (say, a particular eye, or a sign board).

SIFT Step 1: Constructing a scale space

Suppose we wish to have a look at a tree or a leaf. If it is a tree, we get rid of some detail from the image (like the leaves, twigs, etc) intentionally.

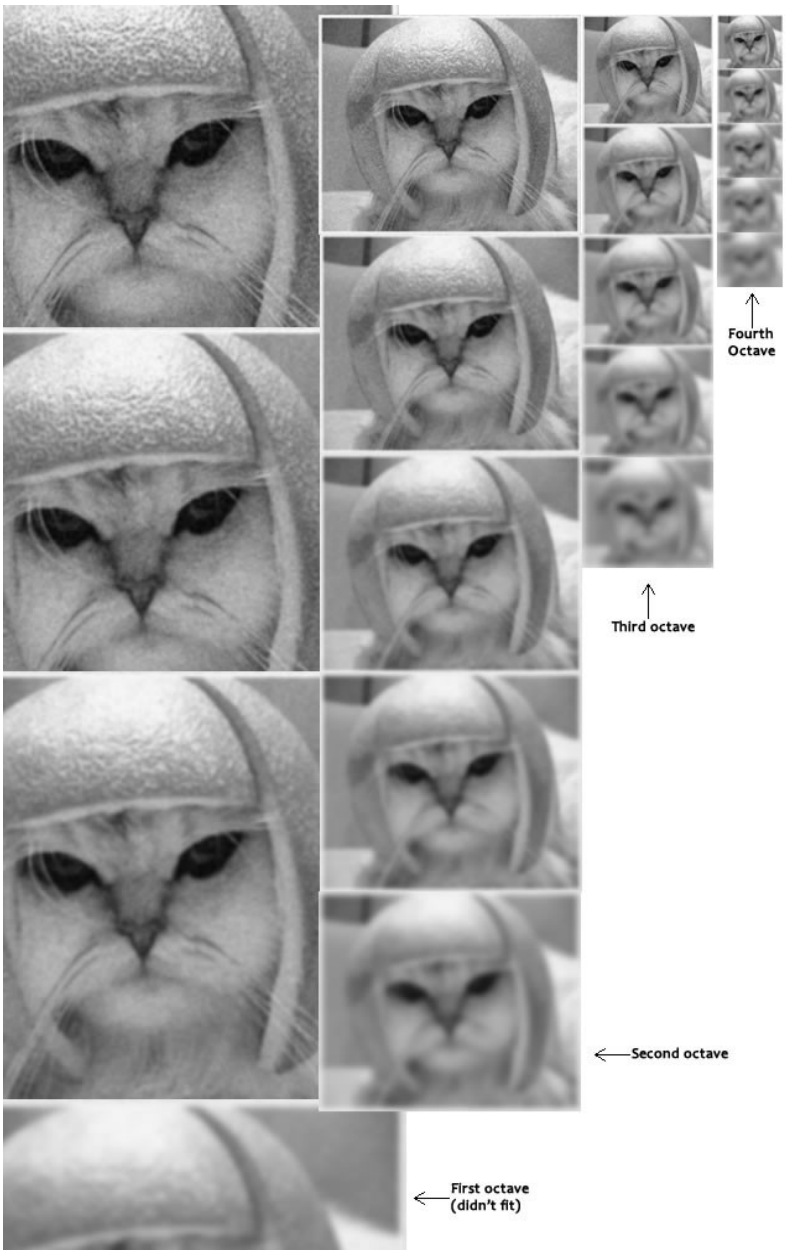
While getting rid of these details, we must ensure that we do not introduce new false details. The only way to do that is with the Gaussian Blur (it was proved mathematically, under several reasonable assumptions).

So to create a scale space, you take the original image and generate progressively blurred out images. Here's an example:

Scale spaces in SIFT :

SIFT takes scale spaces to the next level. You take the original image, and generate progressively blurred out images. Then, you resize the original image to half size. And you generate blurred out images again. And you keep repeating.

Here's what it would look like in SIFT:



Images of the same size (vertical) form an octave. Above are four octaves. Each octave has 5 images. The individual images are formed because of the increasing "scale" (the amount of blur).

- **Octaves and Scales:** The number of octaves and scale depends on the size of the original image. While programming SIFT, you'll have to decide for yourself how many octaves and scales you want. However, the creator of SIFT suggests that 4 octaves and 5 blur levels are ideal for the algorithm.
- **The first octave**: If the original image is doubled in size and antialiased a bit (by blurring it) then the algorithm produces more four times more keypoints. The more the keypoints, the better!

- **Blurring**: Mathematically, “blurring” is referred to as the convolution of the gaussian operator and the image. Gaussian blur has a particular expression or “operator” that is applied to each pixel. What results is the blurred image.

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

The symbols:

- L is a blurred image
- G is the Gaussian Blur operator
- I is an image
- x,y are the location coordinates
- σ is the “scale” parameter. Think of it as the amount of blur. Greater the value, greater the blur.
- The * is the convolution operation in x and y. It “applies” gaussian blur G onto the image I.

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}$$

SIFT Step 2: Laplacian of Gaussian Approximation

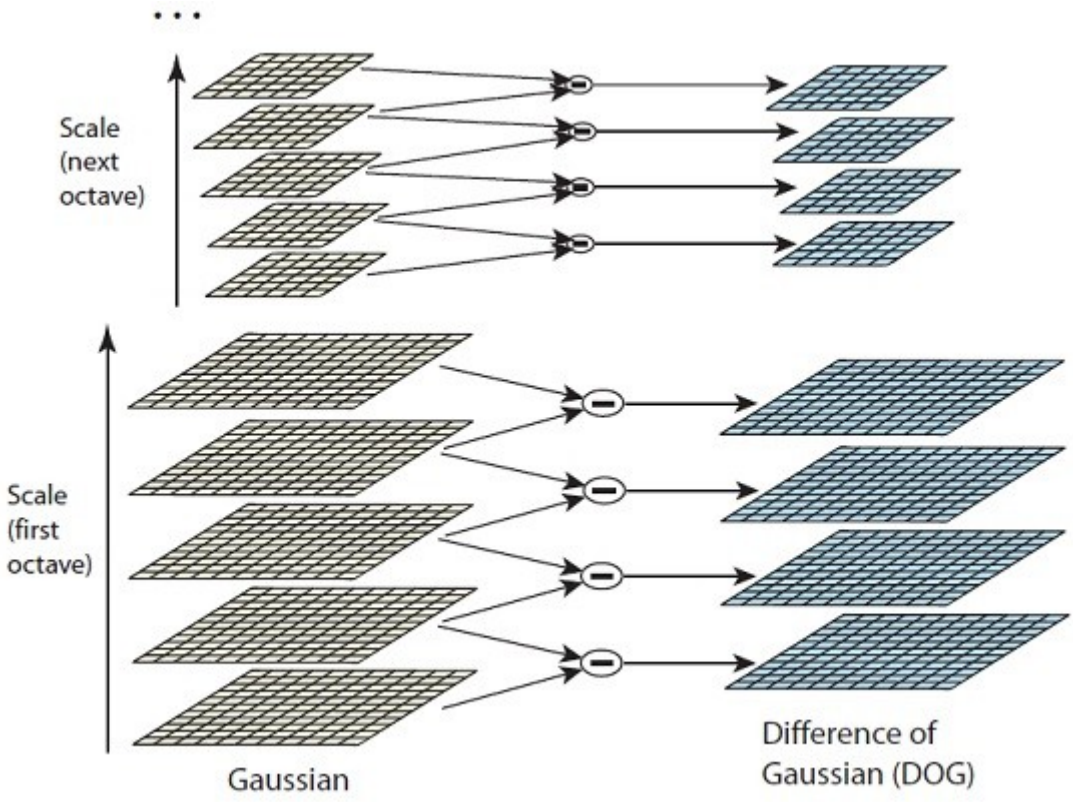
Laplacian of Gaussian

The Laplacian of Gaussian (LoG) operation goes like this. You take an image, and blur it a little. And then, you calculate second order derivatives on it (or, the “laplacian”). This locates edges and corners on the image. These edges and corners are good for finding keypoints.

But the second order derivative is extremely sensitive to noise. The blur smoothes it out the noise and stabilizes the second order derivative.

The problem is, calculating all those second order derivatives is computationally intensive. So , we need another optimization for this, which we have discussed below.

To generate Laplacian of Guassian images quickly, we use the scale space. We calculate the difference between two consecutive scales. Or, the Difference of Gaussians. Here’s how:



These Difference of Gaussian images are approximately equivalent to the Laplacian of Gaussian. And we’ve replaced a computationally intensive process with a simple subtraction (fast and efficient).

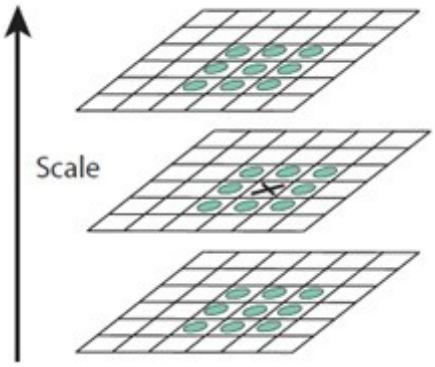
SIFT Step 3: Finding Features

Finding key points is a two part process:

1. Locate maxima/minima in DoG images
2. Find subpixel maxima/minima

Locate maxima/minima in DoG images

The first step is to coarsely locate the maxima and minima. This is simple. You iterate through each pixel and check all it's neighbours. The check is done within the current image, and also the one above and below it. Something like this:



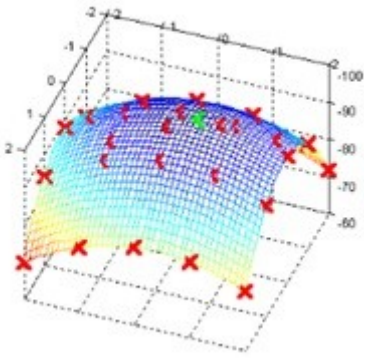
X marks the current pixel. The green circles mark the neighbours. This way, a total of 26 checks are made. **X is marked as a "key point" if it is the greatest or least of all 26 neighbours.**

Usually, a non-maxima or non-minima position won't have to go through all 26 checks. A few initial checks will usually sufficient to discard it.

Note that keypoints are not detected in the lowermost and topmost scales. There simply aren't enough neighbours to do the comparison. So simply skip them!

Once this is done, the marked points are the approximate maxima and minima. They are "approximate" because the maxima/minima almost never lies exactly on a pixel. It lies somewhere between the pixel. But we simply cannot access data "between" pixels. So, we must mathematically locate the subpixel location.

That is to say:



The red crosses mark pixels in the image. But the actual extreme point is the green one.

Find subpixel maxima/minima

Using the available pixel data, subpixel values are generated. This is done by the Taylor expansion of the image around the approximate key point.

Mathematically, it's like this:

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}} \mathbf{x} + \frac{1}{2} \mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2} \mathbf{x}$$

We can easily find the extreme points of this equation (differentiate and equate to zero). On solving, we'll get subpixel key point locations. These subpixel values increase chances of matching and stability of the algorithm.

SIFT Step 4:Eliminate edges and low contrast regions

Removing low contrast features

This is simple. If the magnitude of the intensity (i.e., without sign) at the current pixel in the DoG image (that is being checked for minima/maxima) is less than a certain value, it is rejected.

Because we have subpixel keypoints (we used the Taylor expansion to refine keypoints), we again need to use the taylor expansion to get the intensity value at subpixel locations. If it's magnitude is less than a certain value, we reject the keypoint.

Removing edges

The idea is to calculate two gradients at the keypoint. Both perpendicular to each other. Based on the image around the keypoint, three possibilities exist. The image around the keypoint can be:

A flat region
If this is the case, both gradients will be small.

An edge
Here, one gradient will be big (perpendicular to the edge) and the other will be small (along the edge)

A "corner"
Here, both gradients will be big.

Corners are great keypoints. So we want just corners. If both gradients are big enough, we let it pass as a key point. Otherwise, it is rejected.

Mathematically, this is achieved by the Hessian Matrix. Using this matrix, one can easily check if a point is a corner or not.

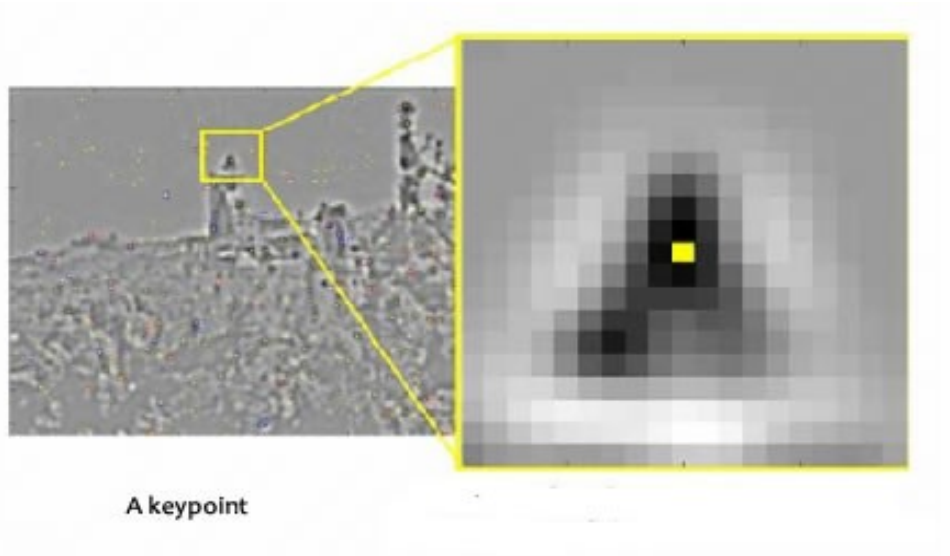
SIFT Step 5:Assigning Keypoin Orientations

After step 4, we have legitimate key points. They've been tested to be stable. We already know the scale at which the keypoint was detected (it's the same as the scale of the blurred image). So we have scale invariance. The next thing is to assign an orientation to each keypoint. This orientation provides rotation invariance. The more invariance you have the better it is.

The Idea

The idea is to collect gradient directions and magnitudes around each keypoint. Then we figure out the most prominent orientation(s) in that region. And we assign this orientation(s) to the keypoint.

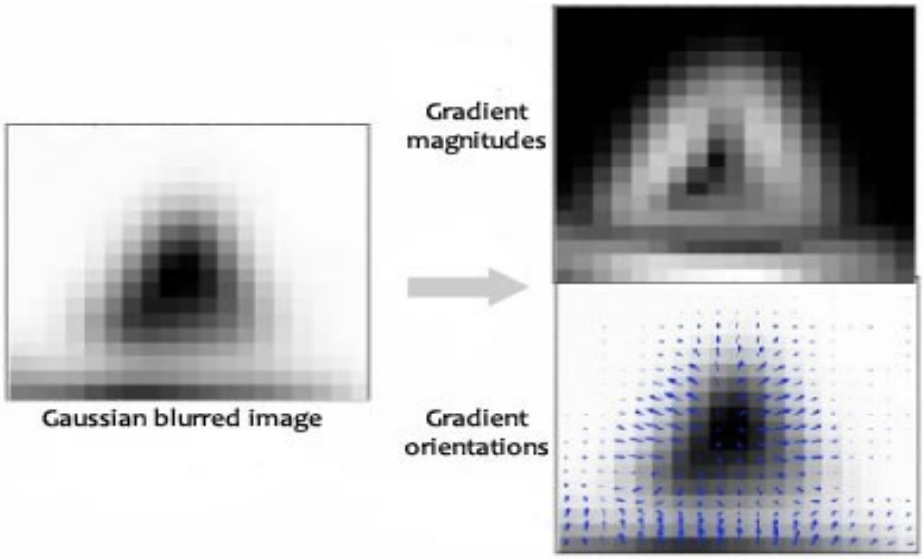
Any later calculations are done relative to this orientation. This ensures rotation invariance.



The size of the "orientation collection region" around the keypoint depends on it's scale. The bigger the scale, the bigger the collection region.

The Details:

Now for the little details about collecting orientations.



Gradient magnitudes and orientations are calculated using these formulae:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)))$$

The magnitude and orientation is calculated for all pixels around the keypoint. Then, A histogram is created for this.

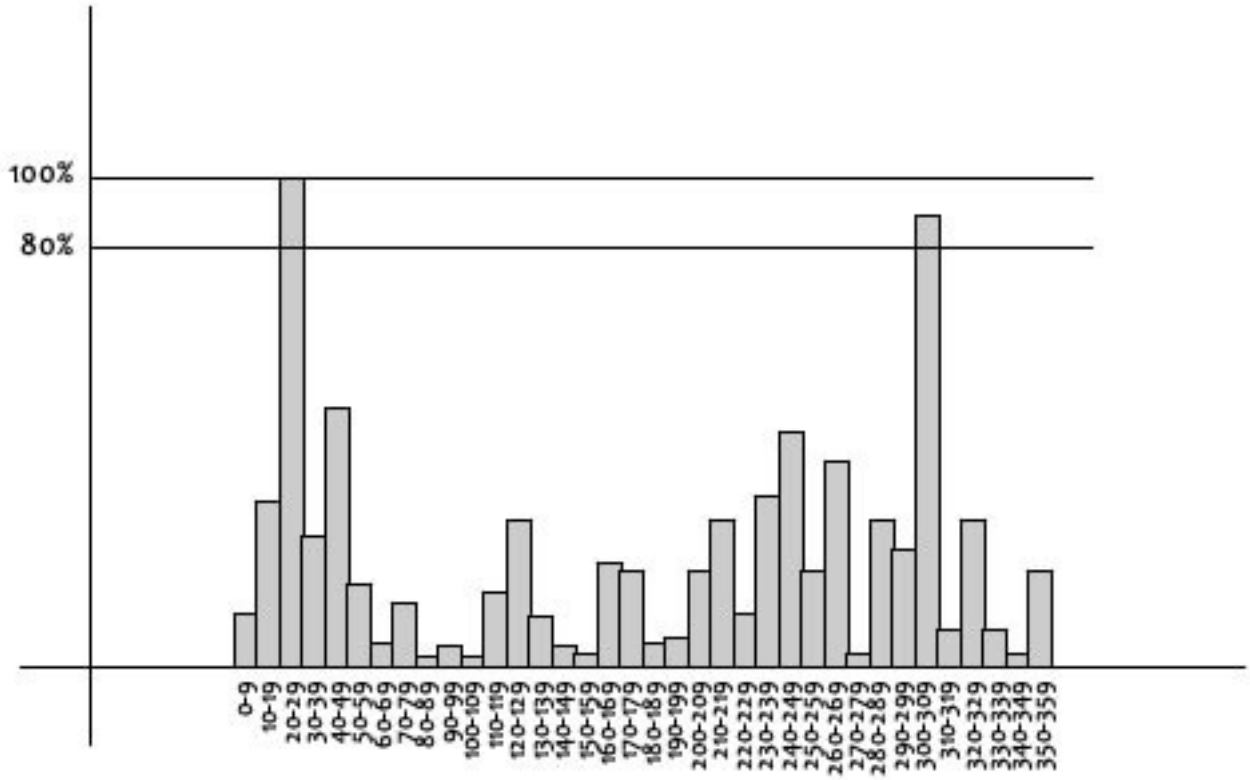
In this histogram, the 360 degrees of orientation are broken into 36 bins (each 10 degrees). Lets say the gradient direction at a certain point (in the “orientation collection region”) is 18.759 degrees, then it will go into the 10-19 degree bin. And the “amount” that is added to the bin is proportional to the magnitude of gradient at that point.

Once you’ve done this for all pixels around the keypoint, the histogram will have a peak at some point.

Above, you see the histogram peaks at 20-29 degrees. So, the keypoint is assigned orientation 3 (the third bin)

Also, any peaks above 80% of the highest peak are converted into a new keypoint. This new keypoint has the same location and scale as the original. But it’s orientation is equal to the other peak.

So, orientation can split up one keypoint into multiple keypoints.



Technical Details

Magnitudes:

Saw the gradient magnitude image above? In SIFT, you need to blur it by an amount of 1.5*sigma.

Size of the Window:

The window size, or the “orientation collection region”, is equal to the size of the kernel for Gaussian Blur of amount 1.5*sigma.

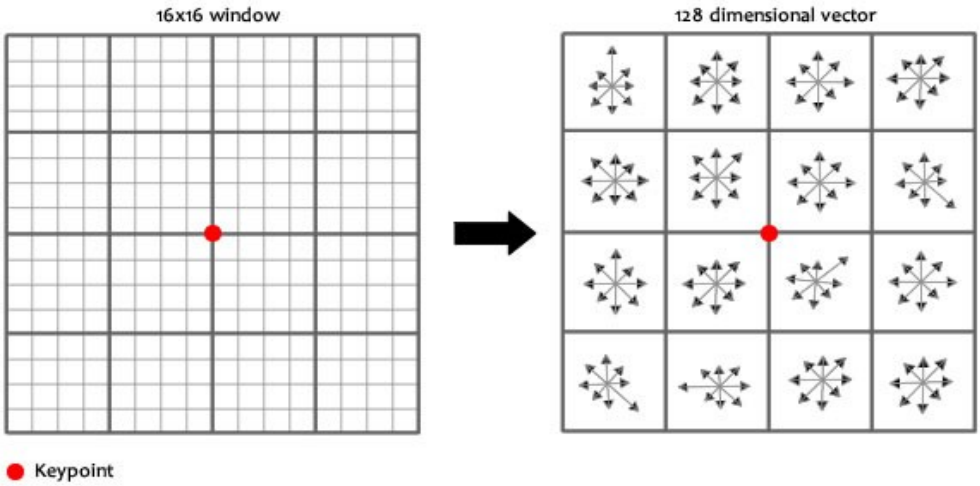
SIFT Step 6:Generate SIFT Features

Now we create a fingerprint for each keypoint. This is to identify a keypoint. If an eye is a keypoint, then using this fingerprint, we'll be able to distinguish it from other keypoints, like ears, noses, fingers, etc.

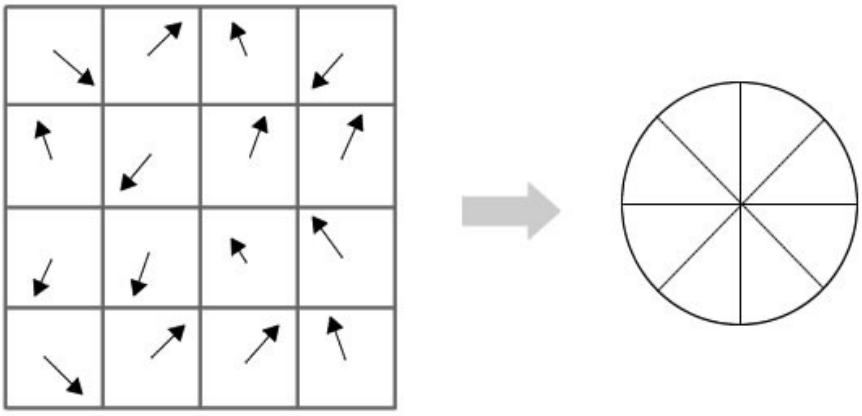
The Idea:

We want to generate a very unique fingerprint for the keypoint. It should be easy to calculate. We also want it to be relatively lenient when it is being compared against other keypoints. Things are never EXACTLY same when comparing two different images.

To do this, a 16×16 window around the keypoint. This 16×16 window is broken into sixteen 4×4 windows.



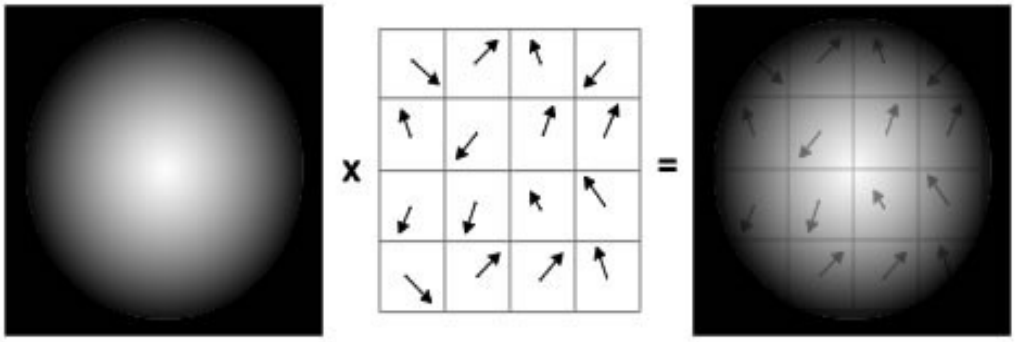
Within each 4×4 window, gradient magnitudes and orientations are calculated. These orientations are put into an 8 bin histogram.



Any gradient orientation in the range 0-44 degrees add to the first bin. 45-89 add to the next bin. And so on.And (as always) the amount added to the bin depends on the magnitude of the gradient.

Unlike the past, the amount added also depends on the distance from the keypoint. So gradients that are far away from the keypoint will add smaller values to the histogram.

This is done using a "gaussian weighting function". This function simply generates a gradient (it's like a 2D bell curve). You multiple it with the magnitude of orientations, and you get a weighted thingy. The farther away, the lesser the magnitude.



Doing this for all 16 pixels, you would've "compiled" 16 totally random orientations into 8 predetermined bins. You do this for all sixteen 4×4 regions. So you end up with 4x4x8 = 128 numbers. Once you have all 128 numbers, you normalize them (just like you would normalize a vector in school, divide by root of sum of squares). These 128 numbers form the "feature vector". This keypoint is uniquely identified by this feature vector.

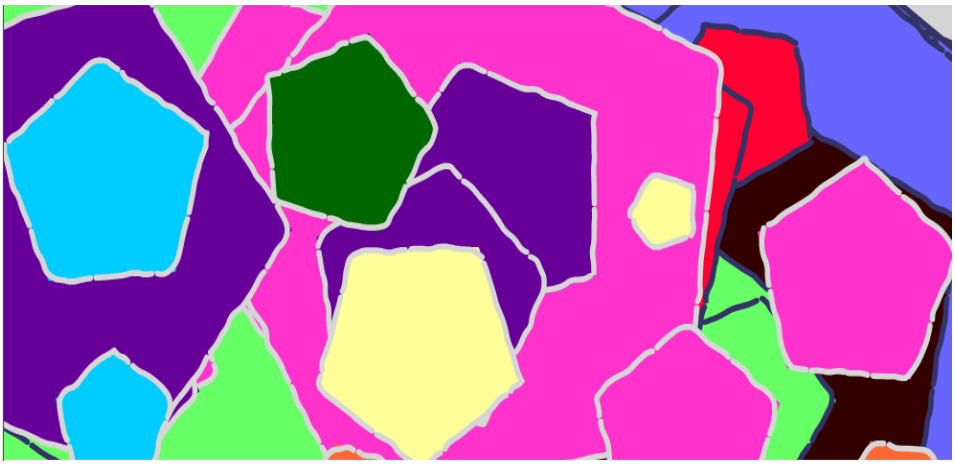
You might have seen that in the pictures above, the keypoint lies "in between". It does not lie exactly on a pixel. That's because it does not. The 16×16 window takes orientations and magnitudes of the image "in-between" pixels. So you need to interpolate the image to generate orientation and magnitude data "in between" pixels.

Our Progress:

We have currently defined all the class structures.

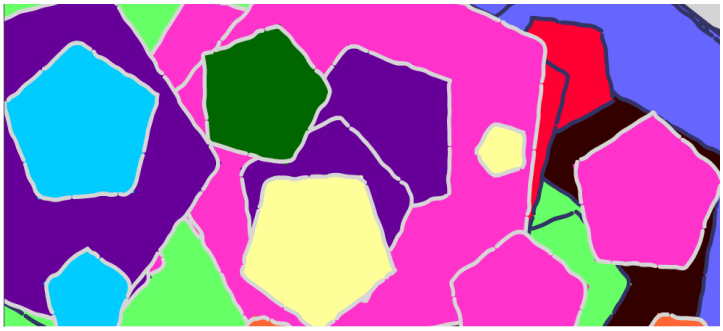
Also, we have coded the SIFT algorithm. Specifically, we have coded parts 1 and 2 of the algorithm.

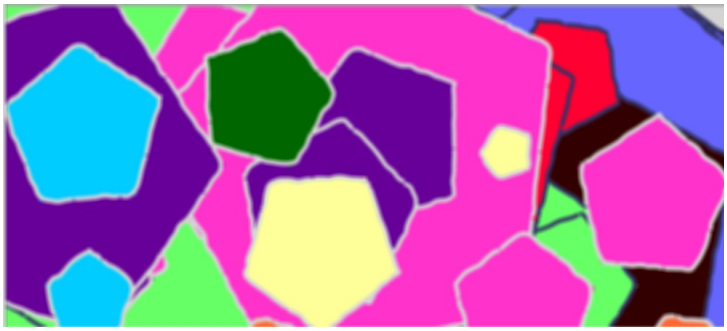
The following are some screen-shots of the outcome of the code :

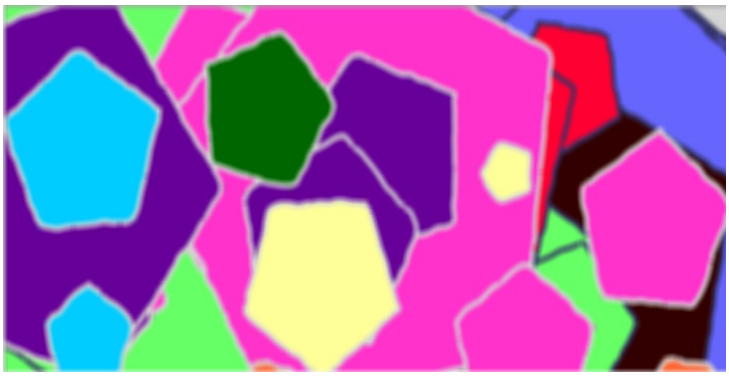


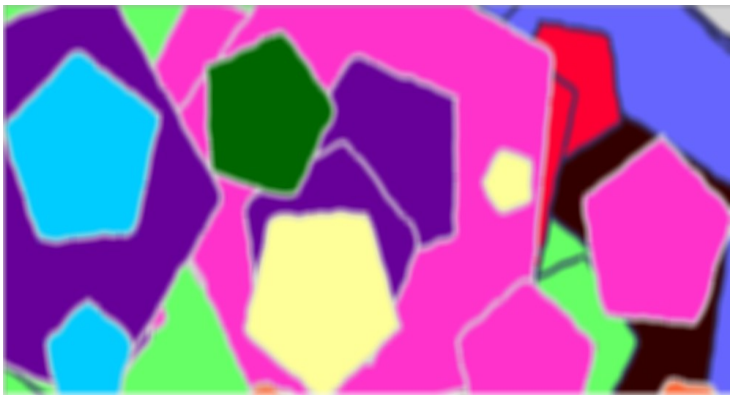
The original Image

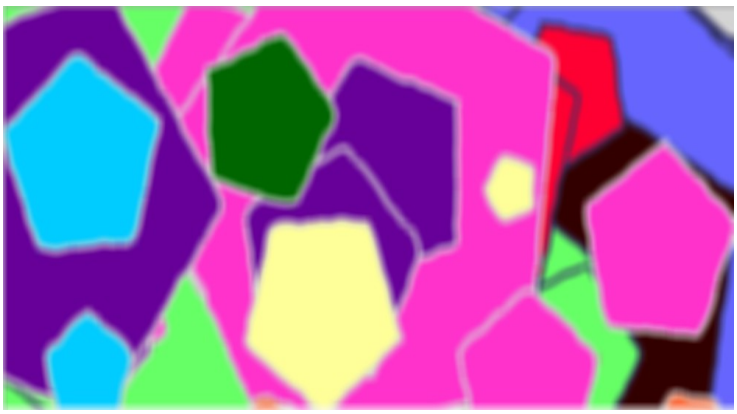
The octave:











Difference of Gaussians: