Project 2 Report: Implementation of SIFT

CMSC 426 Spring 2021

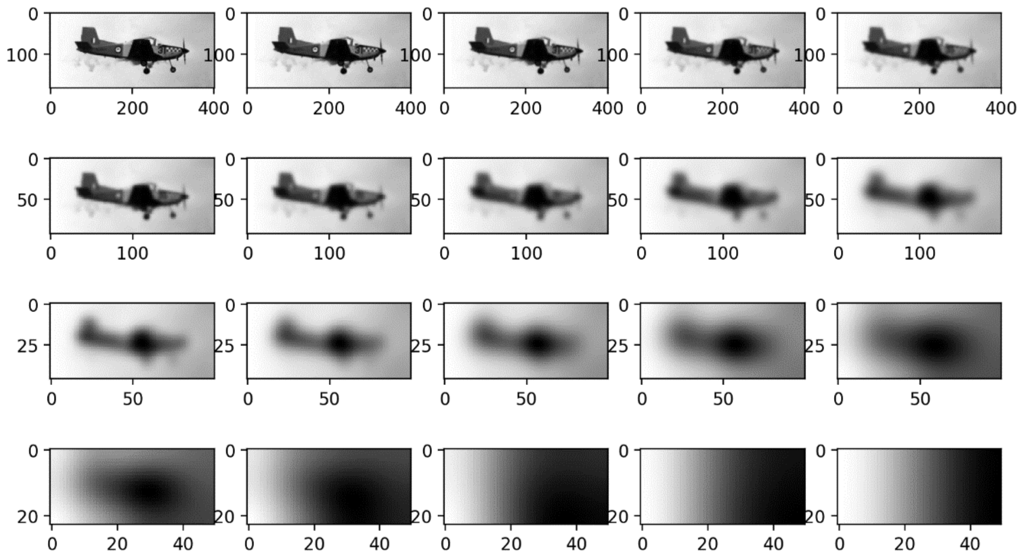
Hongyu Tu 115323568

For this project, since we are not doing matching with SIFT, I split the entire project into 2 parts: detection and description. Two sets of functions are written to implement the algorithm. For this report, I will use image_0002.jpg in dataset 101_ObjectCategories under directory airplanes.

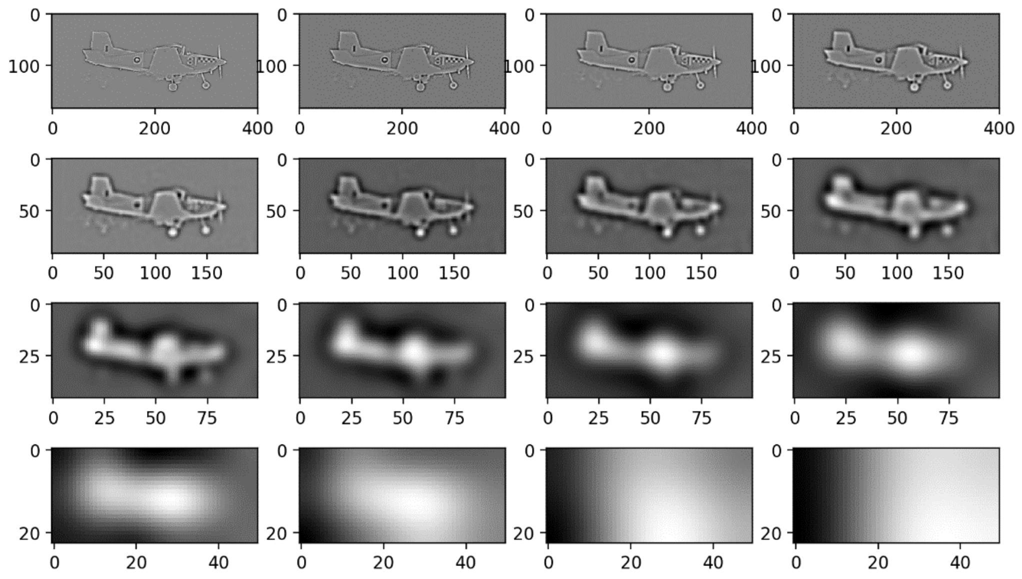
1. Detection

1.1 Keypoint Detection

To start with, I wrote two functions to first calculate the octaves from the original image, after it gets normalized to having all entries within 0 and 1, and then get the DOGs from the octaves. I decided to go with the original paper, 4 octaves each with 5 scales.

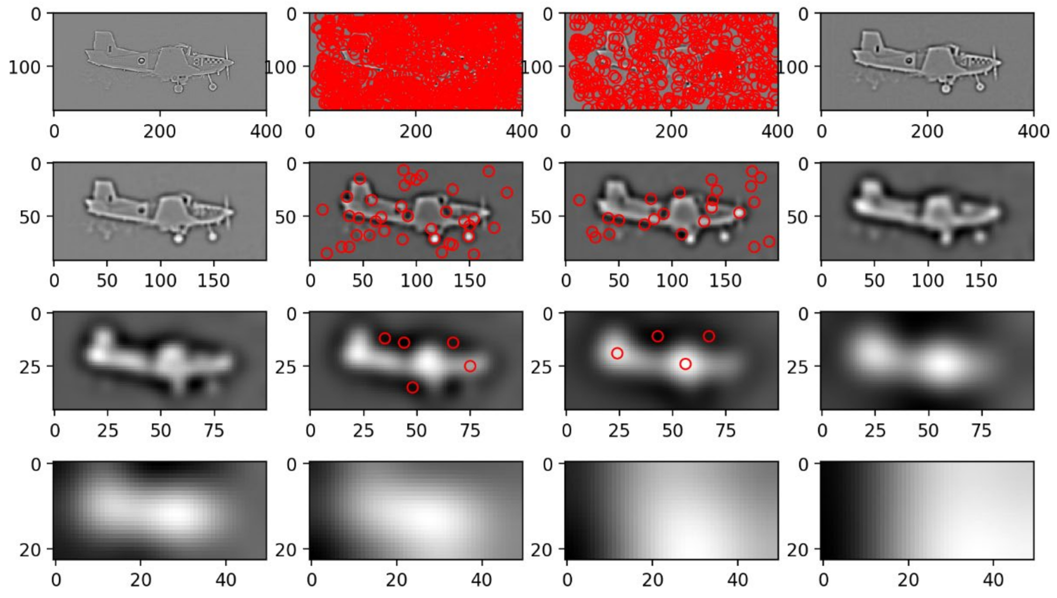


Octave



DOGs

After storing all the DOG into lists, I iterated through to find the local max by comparing each pixel with the 9 + 9 + 8 = 26 surrounding points in two nearby DOGs. With that, I obtained the list of all local keypoints that is ready to be feed into the eliminating process.

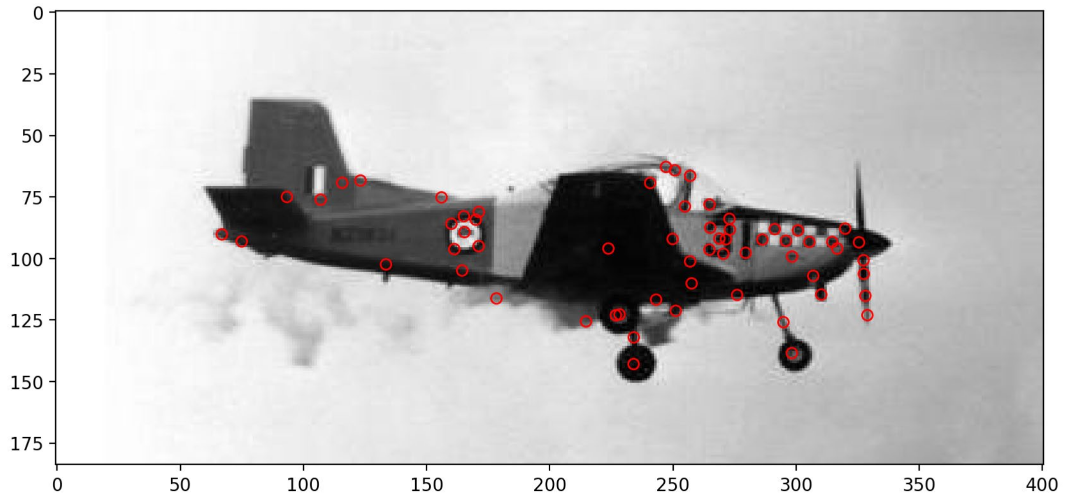


1.2 Eliminating edge & Low contrast points

To obtain the more precious location of keypoints, location of the extrema was calculated through the formula $\hat{x} = -(\partial^2 D / \partial x\hat{}2)\hat{} -1 * \partial D / \partial x$, where the 2nd derivative is obtained by calculating the 3 by 3 Hessian matrix. With that, the low contrast points whose $|D(\hat{x})|$ < 0.03 will be discarded.

To eliminate the edge points, I used the upper-left part of the 3 by 3 Hessian matrix to obtain the $H = \begin{bmatrix} Dxx & Dxy \\ Dxy & Dyy \end{bmatrix}$. And the ratio of the eigenvalues of H, λ_1, and λ_2 will allow us to do the elimination if the ratio is equal to or larger than 10.

After the elimination, I was able to reduce the number of key points from over 1407 to 61. Since this specific image is an airplane in the sky and has a lot of blank space but also not 0, I'd say removing low contrast and edge points worked well here to get rid of most of the meaningless keypoints.



Keypoints after localization and elimination

2. Description

2.1 Assign orientation to keypoints

To find the orientations for each keypoint, I used a for-loop to go over the list of key points and then used the sigma value related to each keypoint to find the size of region which will be applied with these two formulas below:

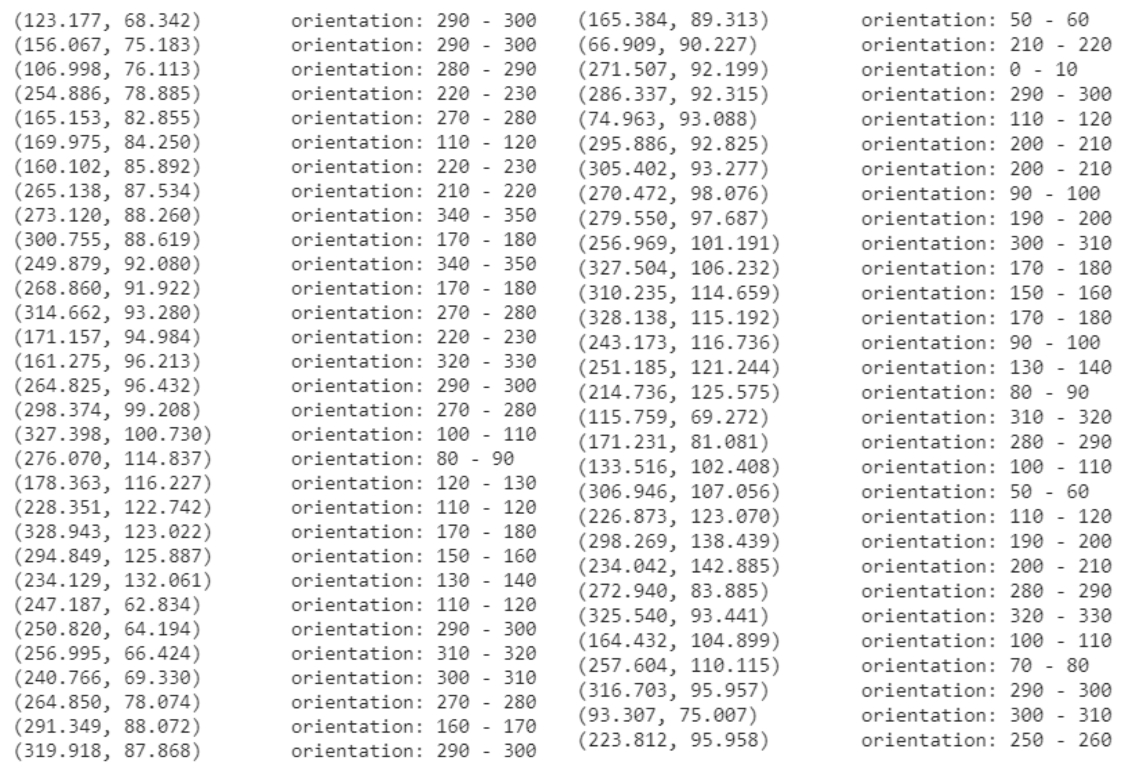
Gradient magnitude,

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

Orientation,

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

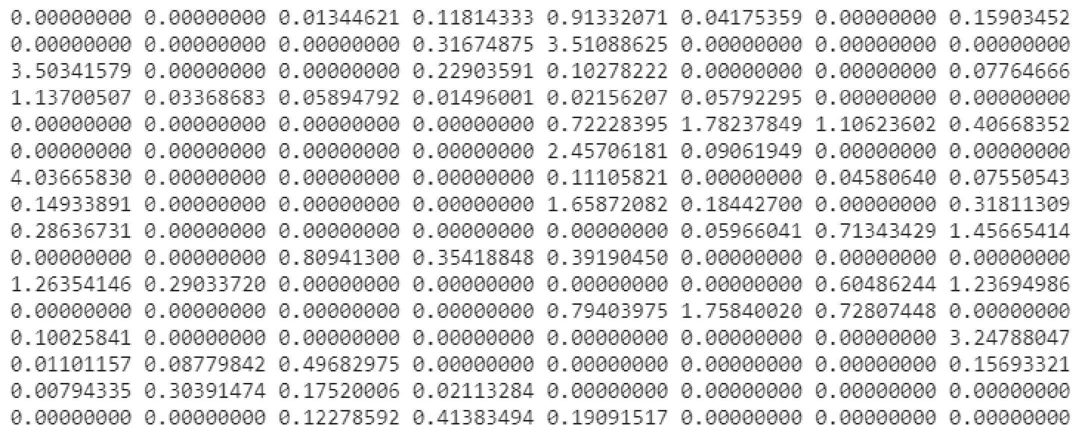
I made a temporary array called histogram of length 36 (corresponding to the 36 bins) and then while I'm in a nested for loop going pixel by pixel within the certain region, I store each pixel's magnitude into the bin that matches the orientation that I found. After I went through all the pixels in the region, the orientation of the key point will be the index of the max of the temporary array. There is the list of key points and their corresponding orientation that gets outputted from this step:



```
(123.177, 68.342)       orientation: 290 - 300   (165.384, 89.313)       orientation: 50 - 60
(156.067, 75.183)       orientation: 290 - 300   (66.909, 90.227)        orientation: 210 - 220
(106.998, 76.113)       orientation: 280 - 290   (271.507, 92.199)       orientation: 0 - 10
(254.886, 78.885)       orientation: 220 - 230   (286.337, 92.315)       orientation: 290 - 300
(165.153, 82.855)       orientation: 270 - 280   (74.963, 93.088)        orientation: 110 - 120
(169.975, 84.250)       orientation: 110 - 120   (295.886, 92.825)       orientation: 200 - 210
(160.102, 85.892)       orientation: 220 - 230   (305.402, 93.277)       orientation: 200 - 210
(265.138, 87.534)       orientation: 210 - 220   (270.472, 98.076)       orientation: 90 - 100
(273.120, 88.260)       orientation: 340 - 350   (279.550, 97.687)       orientation: 190 - 200
(300.755, 88.619)       orientation: 170 - 180   (256.969, 101.191)      orientation: 300 - 310
(249.879, 92.080)       orientation: 340 - 350   (327.504, 106.232)      orientation: 170 - 180
(268.860, 91.922)       orientation: 170 - 180   (310.235, 114.659)      orientation: 150 - 160
(314.662, 93.280)       orientation: 270 - 280   (328.138, 115.192)      orientation: 170 - 180
(171.157, 94.984)       orientation: 220 - 230   (243.173, 116.736)      orientation: 90 - 100
(161.275, 96.213)       orientation: 320 - 330   (251.185, 121.244)      orientation: 130 - 140
(264.825, 96.432)       orientation: 290 - 300   (214.736, 125.575)      orientation: 80 - 90
(298.374, 99.208)       orientation: 270 - 280   (115.759, 69.272)       orientation: 310 - 320
(327.398, 100.730)      orientation: 100 - 110   (171.231, 81.081)       orientation: 280 - 290
(276.070, 114.837)      orientation: 80 - 90     (133.516, 102.408)      orientation: 100 - 110
(178.363, 116.227)      orientation: 120 - 130   (306.946, 107.056)      orientation: 50 - 60
(228.351, 122.742)      orientation: 110 - 120   (226.873, 123.070)      orientation: 110 - 120
(328.943, 123.022)      orientation: 170 - 180   (298.269, 138.439)      orientation: 190 - 200
(294.849, 125.887)      orientation: 150 - 160   (234.042, 142.885)      orientation: 200 - 210
(234.129, 132.061)      orientation: 130 - 140   (272.940, 83.885)       orientation: 280 - 290
(247.187, 62.834)       orientation: 110 - 120   (325.540, 93.441)       orientation: 320 - 330
(250.820, 64.194)       orientation: 290 - 300   (164.432, 104.899)      orientation: 100 - 110
(256.995, 66.424)       orientation: 310 - 320   (257.604, 110.115)      orientation: 70 - 80
(240.766, 69.330)       orientation: 300 - 310   (316.703, 95.957)       orientation: 290 - 300
(264.850, 78.074)       orientation: 270 - 280   (93.307, 75.007)        orientation: 300 - 310
(291.349, 88.072)       orientation: 160 - 170   (223.812, 95.958)       orientation: 250 - 260
(319.918, 87.868)       orientation: 290 - 300
```

2.2 Construct SIFT descriptor

Lastly, it's time to obtain SIFT descriptor. Since this step is extremely similar to what we did in the previous step, I simply duplicated the code, changed the size of the region to 16 by 16 instead of 1.5 times sigma, and then did 8 bins instead of 36. Since we want the length of the descriptor to be 128 (4 x 4 x 8), I added another layer of the nested loop so that the 16 by 16 region will be processed in units of 4 by 4 smaller blocks. The function works with no problem as expected. (Since SIFT descriptor is long, I'll show just one below.)

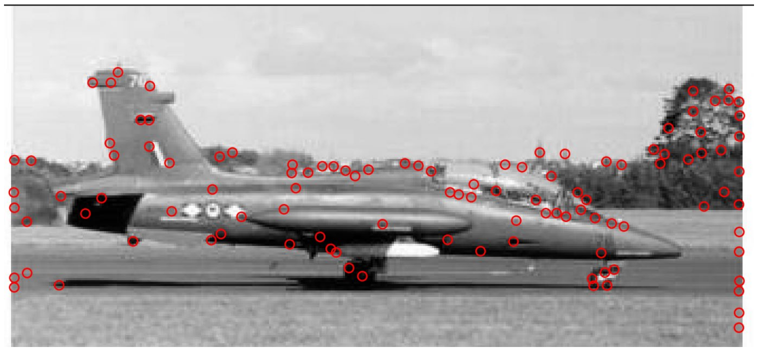


```
0.00000000 0.00000000 0.01344621 0.11814333 0.91332071 0.04175359 0.00000000 0.15903452
0.00000000 0.00000000 0.00000000 0.31674875 3.51088625 0.00000000 0.00000000 0.00000000
3.50341579 0.00000000 0.00000000 0.22903591 0.10278222 0.00000000 0.00000000 0.07764666
1.13700507 0.03368683 0.05894792 0.01496001 0.02156207 0.05792295 0.00000000 0.00000000
0.00000000 0.00000000 0.00000000 0.00000000 0.72228395 1.78237849 1.10623602 0.40668352
0.00000000 0.00000000 0.00000000 0.00000000 2.45706181 0.09061949 0.00000000 0.00000000
4.03665830 0.00000000 0.00000000 0.00000000 0.11105821 0.00000000 0.04580640 0.07550543
0.14933891 0.00000000 0.00000000 0.00000000 1.65872082 0.18442700 0.00000000 0.31811309
0.28636731 0.00000000 0.00000000 0.00000000 0.00000000 0.05966041 0.71343429 1.45665414
0.00000000 0.00000000 0.80941300 0.35418848 0.39190450 0.00000000 0.00000000 0.00000000
1.26354146 0.29033720 0.00000000 0.00000000 0.00000000 0.00000000 0.60486244 1.23694986
0.00000000 0.00000000 0.00000000 0.00000000 0.79403975 1.75840020 0.72807448 0.00000000
0.10025841 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 3.24788047
0.01101157 0.08779842 0.49682975 0.00000000 0.00000000 0.00000000 0.00000000 0.15693321
0.00794335 0.30391474 0.17520006 0.02113284 0.00000000 0.00000000 0.00000000 0.00000000
0.00000000 0.00000000 0.12278592 0.41383494 0.19091517 0.00000000 0.00000000 0.00000000
```

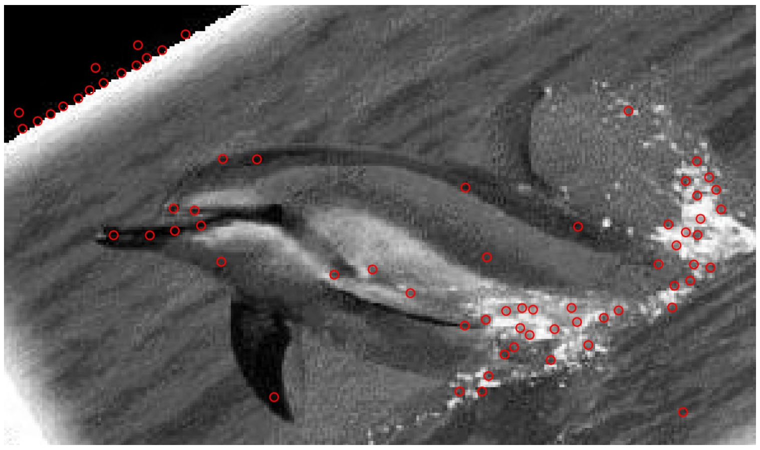
3. Observation

For this project, I specifically looked at three categories of images: airplanes, dolphins, and Leopards. Here are a few results that I got:

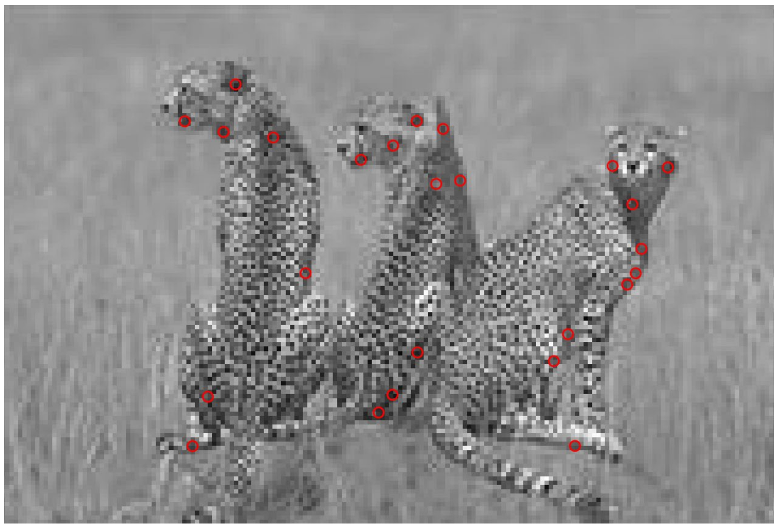
Airplanes:



Dolphins:



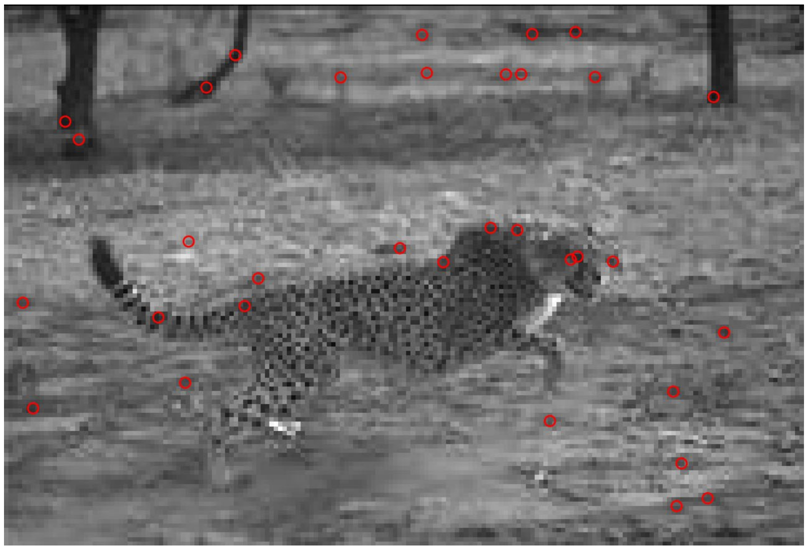
Leopards:



Out of these three, I think the airplane is the easier to find keypoints. I think the reason is that airplanes usually have color and shape that are not similar to the background and stands out. Also, the background for airplanes can be clean as they are usually in the air, the elimination process can easily get rid of undesired points and return what we expected.

For dolphins, since they are in water and their body doesn't usually contain distinguish colors, it's easy for SIFT to pick up on water more than dolphins. Although I believe with better tuning on the thresholds within my implementation, the performance can be better.

I would say leopards are the hardest out of these three to find keypoints. As a predator that is good at blending into the surroundings, the spots on leopards and the resemblance of the body color allow leopards to have every little significant spot for SIFT to pick up. The leopard images above did the job decently but here is an image that SIFT is just simply lost:



The key points are everywhere but the leopard's body. This will make it hard on matching later, as the dots on the leopard acts as a natural camouflage.