

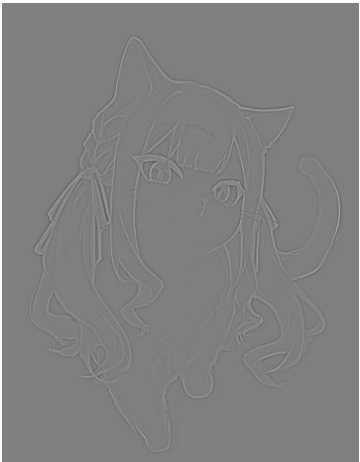





Computer Vision HW1 Report



Student ID: b11901073

Name: 林禹融


Part 1.

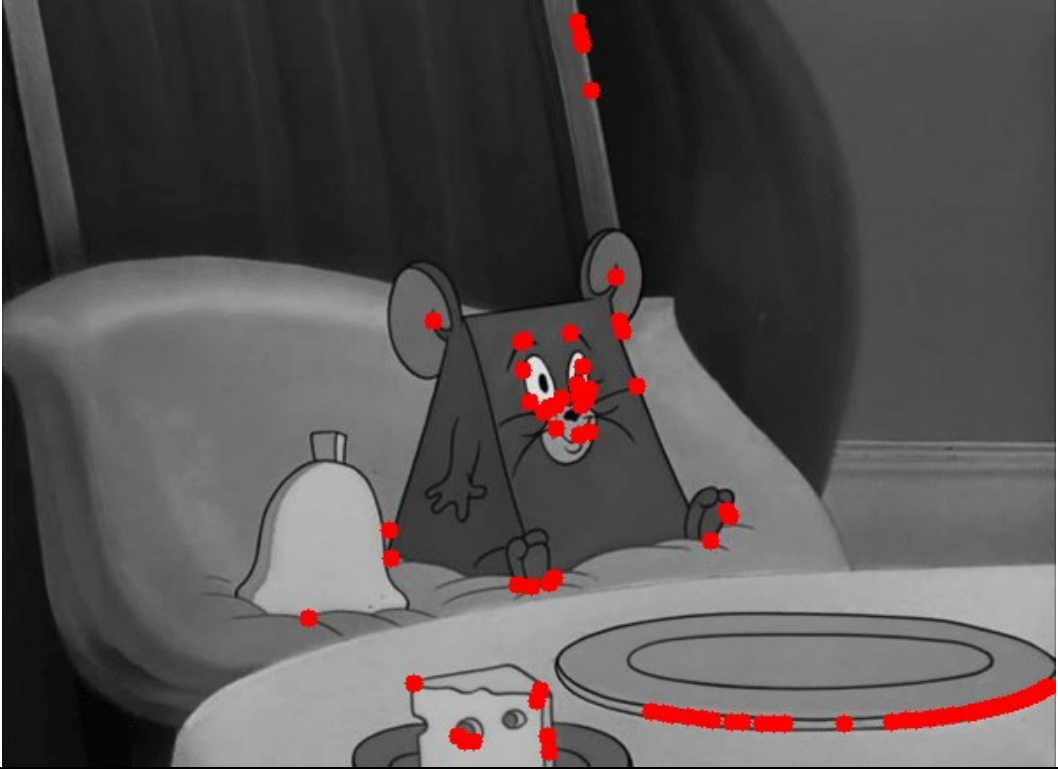

- Visualize the DoG images of 1.png.

	DoG Image (threshold = 5)		DoG Image (threshold = 5)
DoG1-1.png		DoG2-1.png	
DoG1-2.png		DoG2-2.png	
DoG1-3.png		DoG2-3.png	

DoG1-4.png		DoG2-4.png	
------------	---	------------	---

- Use three thresholds (1,2,3) on 2.png and describe the difference.

Threshold	Image with detected keypoints on 2.png
2	

5	
7	

When using a Difference of Gaussian (DoG) approach for feature detection, adjusting the threshold essentially controls how “strong” a feature (such as an edge or corner) must be to be counted. A lower threshold will detect many more points, while a higher threshold will pick up fewer but more salient points.

First Image uses the lowest threshold among the three. It detects a large number of features, including many low-contrast or noisy areas.

The detector in Second Image rejects some of the weaker edges or corners, so fewer points remain. Notice that major contours around the character’s face and the plate are still highlighted, but extraneous noise is filtered out compared to the first image.






For Third Image, only the most prominent or high-contrast features pass the threshold, yielding a small set of very strong keypoints. You can see that most tiny details have disappeared, leaving only clearly defined points (e.g., around the character’s eyes and a couple of strong corners).

Part 2.

- **Report the cost for each filtered image.**

Gray Scale Setting	Cost (1.png)	Gray Scale Setting	Cost (2.png)
cv2.COLOR_BGR2GRAY	1207799	cv2.COLOR_BGR2GRAY	183850
$R*0.0+G*0.0+B*1.0$	1439568	$R*0.1+G*0.0+B*0.9$	77884
$R*0.0+G*1.0+B*0.0$	1305961	$R*0.2+G*0.0+B*0.8$	86023
$R*0.1+G*0.0+B*0.9$	1393620	$R*0.2+G*0.8+B*0.0$	188019
$R*0.1+G*0.4+B*0.5$	1279697	$R*0.4+G*0.0+B*0.6$	128341
$R*0.8+G*0.2+B*0.0$	1127913	$R*1.0+G*0.0+B*0.0$	110862




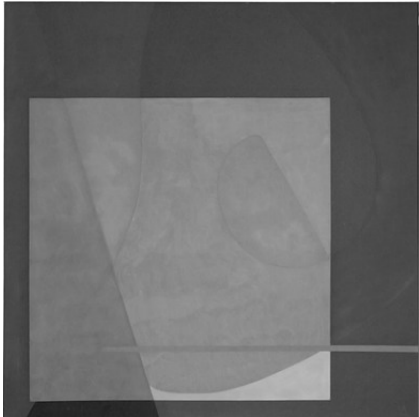

- **Show original RGB image / two filtered RGB images and two grayscale images with highest and lowest cost.**

Original RGB image (1.png)	Filtered <u>RGB image</u> and <u>Grayscale image</u> of Highest cost	Filtered <u>RGB image</u> and <u>Grayscale image</u> of Lowest cost
		
		

If two visibly distinct colors (red vs. green) end up mapped to nearly the same brightness in grayscale, the cost function flags that as a higher cost.

In the left (darker) grayscale image, the leaf does not stand out well from the grass because both are mapped to similar mid-gray tones. This represents a bigger “failure” to preserve the original contrast, so the algorithm assigns a higher cost.

In the right (lighter) grayscale image, the leaf appears clearly brighter than the grass, reflecting the strong color contrast from the original. That more faithfully preserves the visual distinction between leaf and background, which leads to a lower cost.

Original RGB image (2.png)	Filtered <u>RGB image</u> and <u>Grayscale image</u> of Highest cost	Filtered <u>RGB image</u> and <u>Grayscale image</u> of Lowest cost
		
		

If two distinctly different colors (such as the warm brown vs. cool blue or orange vs. purple) end up mapped to nearly the same brightness in grayscale, the cost function flags that as a higher cost.

In the left (darker) grayscale image, many of the colored shapes appear to blend together into similar mid-gray tones. This represents a bigger “failure” to preserve the original color contrast, so the algorithm assigns a higher cost.

By contrast, in the right (lighter) grayscale image, the shapes exhibit stronger gray-level differences that better reflect the original color separations. That more faithfully preserves the visual distinction between the overlapping shapes, leading to a lower cost.

- **Describe how to speed up the implementation of bilateral filter.**

A key reason this implementation is faster than a completely naïve approach is that it precomputes the spatial Gaussian factor once (the “Gs ” term) and reuses it for each pixel. Traditionally, if you recompute that 2D spatial kernel inside each loop over the image, you waste a lot of effort on the same exponentials again and again. Here, the filter computes that kernel once in the constructor and simply multiplies it by the “range” part on every iteration.

Additionally, the code:

1. Normalizes the guidance image in advance, so you avoid repeated floating-point conversions, and
2. Leverages NumPy's vectorized operations to compute the range kernel and final weighted sums in blocks, rather than looping in pure Python.

All of these details cut down on redundant calculations and make the filter run more efficiently.