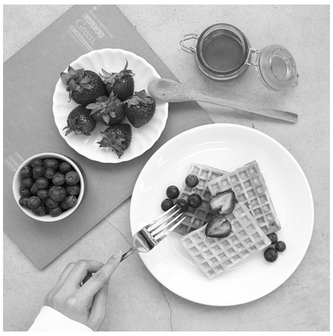
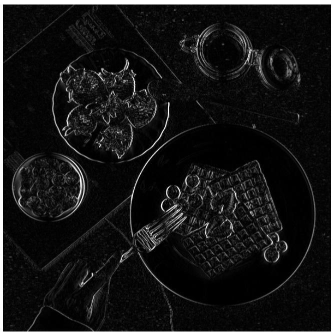
Problem 1 : EDGE DETECTION

(a) Apply Sobel edge detection to **sample1.png**. Output the gradient image as **result1.png** and its corresponding edge map as **result2.png**. Additionally, describe the threshold selection process and its impact on the result, and provide your analysis.

Result


sample1.png


result1.png

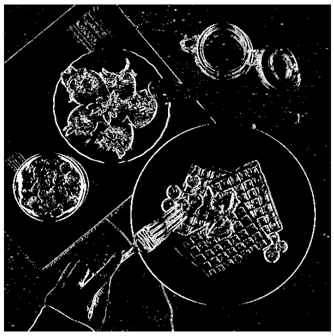

result2.png

Figure 1: Sobel Edge Detection w/ kernel size = 3, thres = 40

Approach

一開始使用了上課介紹了最基本的 3×3 Sobel Kernel

$$G_x = \frac{1}{4}\begin{bmatrix} -1, & 0, & 1 \\ -2, & 0, & 2 \\ -1, & 0, & 1 \end{bmatrix}, \quad G_y = \frac{1}{4}\begin{bmatrix} 1, & 2, & 1 \\ 0, & 0, & 0 \\ -1, & -2, & -1 \end{bmatrix}$$

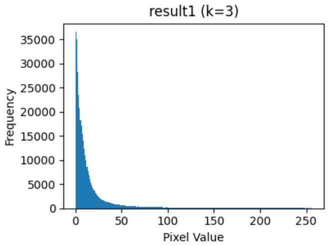
想了一下，發現他其實就是在算每一個 pixel 對 center point 所貢獻的梯度。因此我們可以推廣到 $n \times n$ 的 Sobel Kernel，推廣如下：

Proof. 對於一個 $n \times n$ 的 Sobel Kernel G，我們以 G 的中心為 $(0,0)$, 則對於 $G_{i,j}, i, j \in [-n//2, n//2]$, 我們有

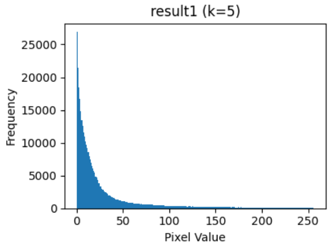
$$\begin{cases} G_x(i,j) = \frac{i}{(i^2+j^2)} \\ G_y(i,j) = \frac{j}{(i^2+j^2)} \end{cases}$$

QED

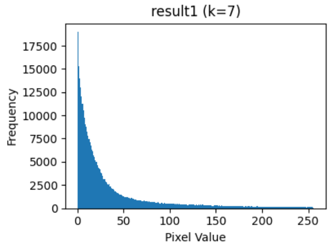
因此，我嘗試了 kernel size $= 3, 5, 7$，並畫出它們的 histogram 來決定 threshold



kernel size $= 3$



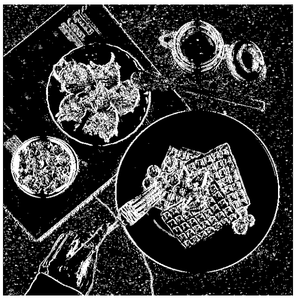
kernel size $= 5$



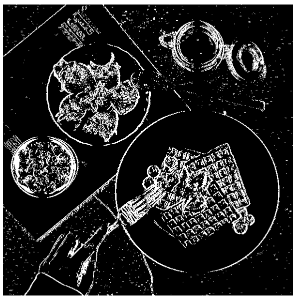
kernel size $= 7$

Figure 2: Histogram of Sobel kernel w/ different size

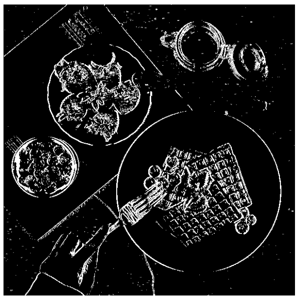
可以看到隨著 kernel 變大，gradient magnitude 也越多，因此對於 kernel size $= 3$ 我選擇 20 to 100 來當作 threshold。結果如下：



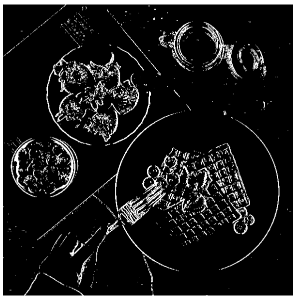
threshold $= 20$



threshold $= 30$

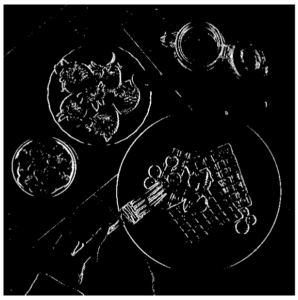


threshold $= 40$

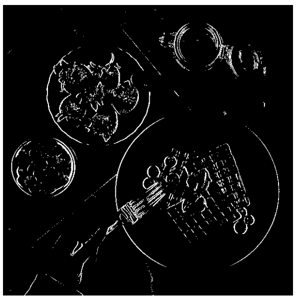


threshold $= 50$

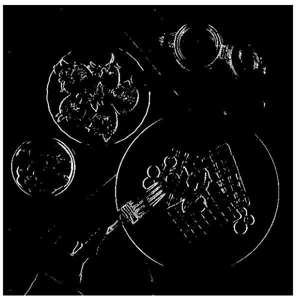
threshold $= 60$



threshold $= 70$



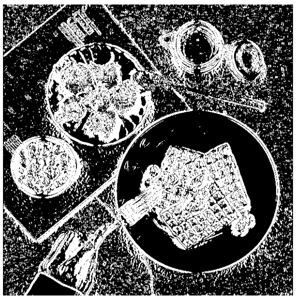
threshold $= 80$



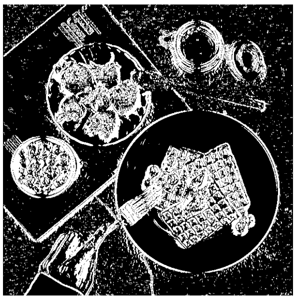
threshold $= 90$

threshold $= 100$

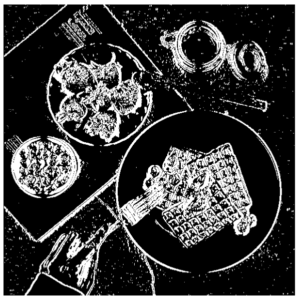
Figure 3: Edge map of Sobel filtering w/ kernel size $= 3$



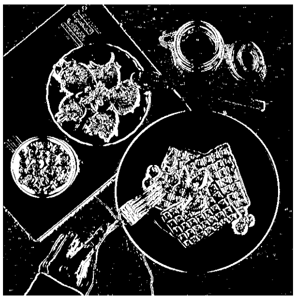
threshold = 20



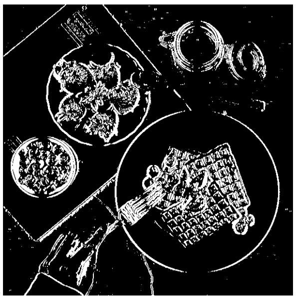
threshold = 30



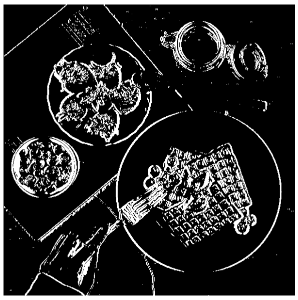
threshold = 40



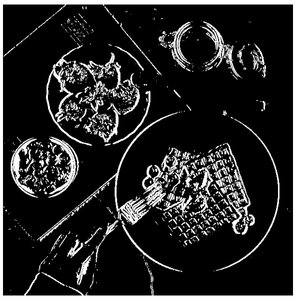
threshold = 50



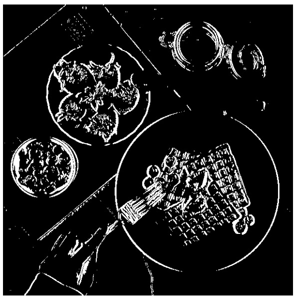
threshold = 60



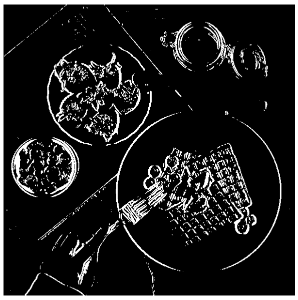
threshold = 70



threshold = 80

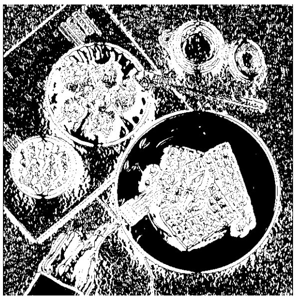


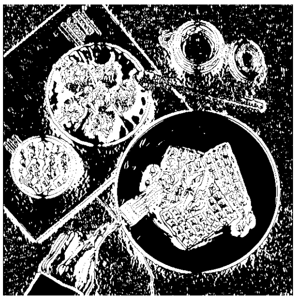
threshold = 90

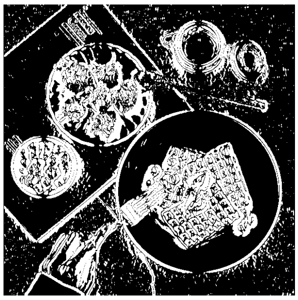


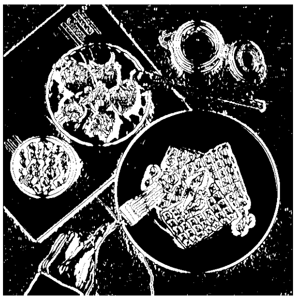
threshold = 100

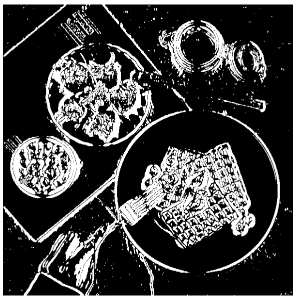
Figure 4: Edge map of Sobel filtering w/ kernel size = 5

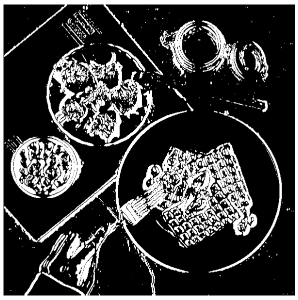

threshold = 20

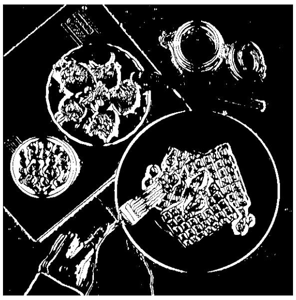

threshold = 30

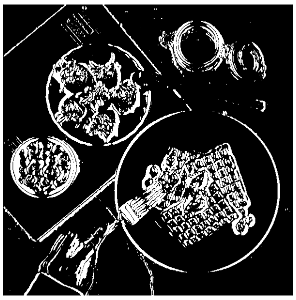

threshold = 40


threshold = 50


threshold = 60


threshold = 70


threshold = 80


threshold = 90

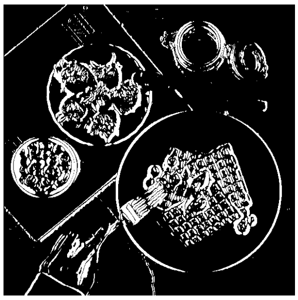

threshold = 100

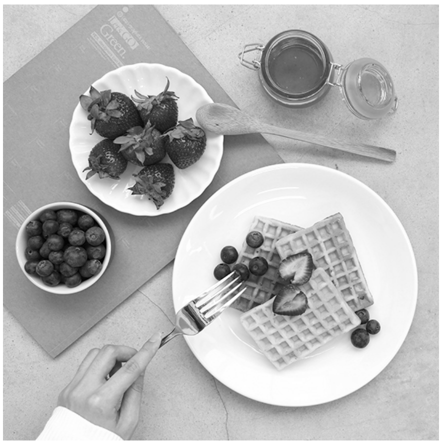
Figure 5: Edge map of Sobel filtering w/ kernel size = 7

Discussion

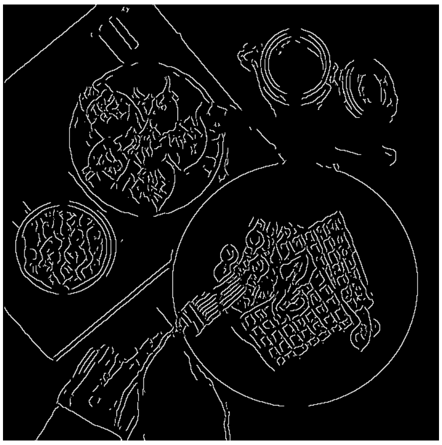
對於同一個 kernel size，可以看到 threshold 過低時，noise 會很多，因此好的 threshold 要能夠減少 noist 外，也要維持 edge 的強度。對於不同 kernel size，可以觀察 kernel size = 5, 7 時，雖然整體較清晰，但 edge 會過粗。因此這題我最後選擇 kernel size = 3, threshold = 40。

(b) Perform Canny edge detection on **sample1.png** and save the resulting edge map as **result3.png**. Also, provide an explanation of your parameter selection process and how it impacts the outcome.

Result



sample1.png



result3.png

Figure 6: Canny Edge Detection

Approach

Canny edge detection 有 5 個步驟，前面 4 個步驟比較普通，因此我簡略帶過，此題我的重點放在第 5 個步驟。

1. **Noise Reduction**
   我使用 $n \times n$ Gaussian noise filter 來降噪，其中我嘗試 kernel size $= 3, 5, 7$。

2. **Compute Gradient**
   我使用 $n \times n$ Sobel kernel 來計算 gradient magnitude and orientation，其中我嘗試 kernel size $= 3, 5, 7$。

3. **Non-Maximal Suppression**
   透過上一步驟計算的梯度方向，將其 map 到 8 個方位 (8 connectivity)，再 perform non-maximal suppresion。

4. **Hysterestic Thresholding**
   像 p1-(a) 一樣先畫出其 histogram，再參考整體分佈決定兩個 threshold，基本上決定一個我覺得可以的值後，我就暴搜他附近的值。

5. **Connected-Component Labeling**
   這裡我就有多花點心思了。最 naive 的做法會是遇到一個是 strong-edge 的 pixel，就遞迴附近的 weak-pixel 去更新他們的值，這樣的最糟時間複雜度會是 $O(w^2h^2)$，十分地沒效率。因此我實作了一個基於並查集 (Disjoint Set

Union) 的演算法。首先，每個 coordinate 會是自己一個 set，先將已經為 strong-edge 的點的 root 設為 $(-1,-1)$(一個 pseudo set 表示 strong edge 的集合)，接下來從 $(0,0)$ 遍歷一遍到 $(h-1,w-1)$，其中經過的點每次檢查他的左，左上，上，右上，四個點，如果其中有任何 edge candidate (包含 strong 跟 weak)，就跟他 union。最後去檢查每一個 coordinate 的 root 是否為 $(-1,-1)$，是的話他就是 strong edge，反之則否。這樣就可以 $O(wh)$ 來完成 Connected-Component Labeling。

Algorithm 1: Find (DSU)

```
Data: x : current coordinate, p : root list
Function find (x, p)
    if p[x] = x then
        return x
    end
    return p[x] = find(p[x], x)
end
```

Algorithm 2: Union (DSU)

```
Data: x : target1 coordinate, y : target2 coordinate, p : root list
Function union (x, y, p)
    s1 ← find(x, p);
    s2 ← find(y, p);
    if s1! = s2 then
        // Should always let (-1, -1) be root
        if s1 = (−1, −1) then
            p[s2] ← p[s1];
        else
            p[s1] ← p[s2];
        end
    end
end
```

Parameter

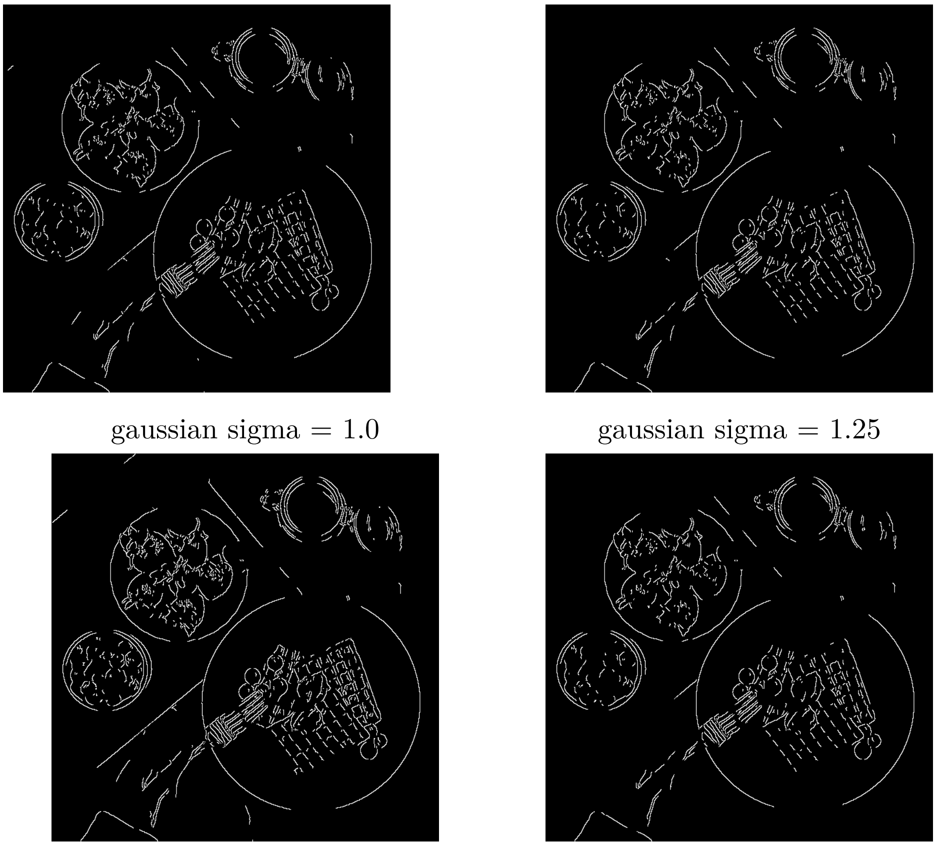
- **Gaussian Kernel Size** (Constant: gaussian kernel sigma = 1.5, sobel kernel size = 3, thres low = 30, thres high = 50)

kernel size = 3 kernel size = 5 kernel size = 7

Figure 7: Canny Edge Detection w/ different gaussian size

可以看到隨著 gaussian kernel 變大，同個 threshold 被偵測到的 edge 就越少，因此我們對於更大的 kernel，就需要更大的 threshold。

- **Gaussian Kernel sigma** (Constant: gaussian kernel size = 3, sobel kernel size = 3, thres low = 40, thres high = 60)



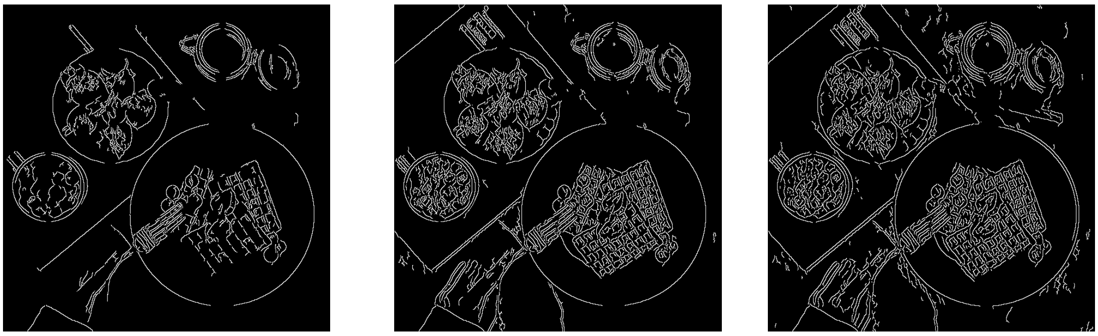
gaussian sigma = 1.0 gaussian sigma = 1.25

gaussian sigma = 1.5 gaussian sigma = 2.0

Figure 8: Canny Edge Detection w/ different gaussian sigma

可以看到隨著 gaussian sigma 變大，noise 就越少，但好像沒有很明顯。

- **Sobel Kernel size** (Constant: gaussian kernel size = 3, gaussian kernel sigma = 1.5, thres low = 40, thres high = 60)

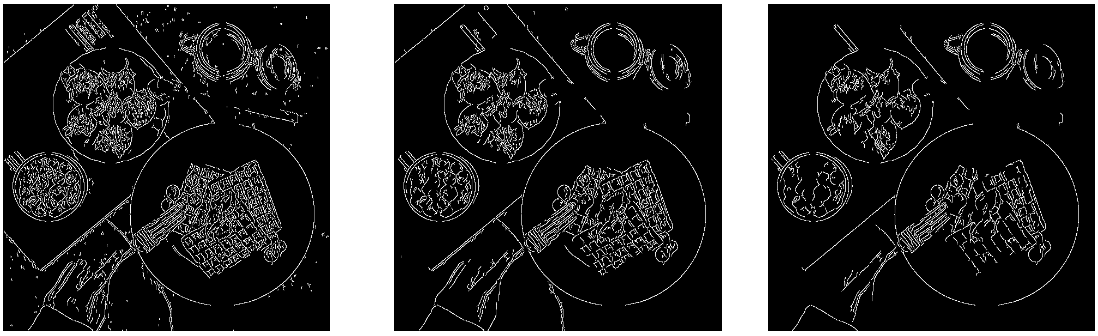


kernel size = 3 kernel size = 5 kernel size = 7

Figure 9: Canny Edge Detection w/ different gaussian size

可以看到隨著 sobel kernel 變大，效果很明顯的提升很多。

- **Threshold** (Constant: gaussian kernel size = 3, gaussian kernel sigma = 1.5, sobel kernel size = 3)



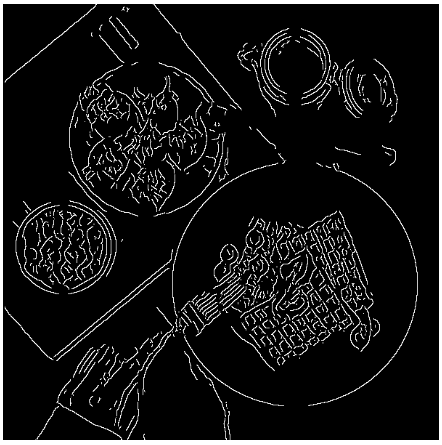
Tl=10.0, Th=20.0 Tl=10.0, Th=40.0 Tl=10.0, Th=60.0

Tl=40.0, Th=40.0 Tl=40.0, Th=60.0 Tl=40.0, Th=80.0

Figure 10: Canny Edge Detection w/ different threshold

可以看到 T_l, T_h 如果太低，則會有太多 noise，反之若太高，則會使得 edge 減少。因此我們要找出適合的 T_l 才 T_h 才能達到最好的效果

- **Final Result**
  gaussian kernel size = 7, gaussian kernel sigma = 2.0, sobel kernel size = 7, thres low = 40, thres high = 60

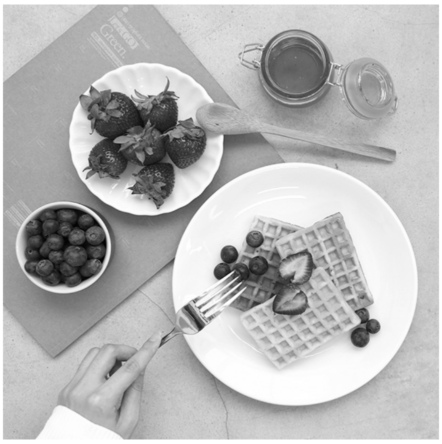


result3.png

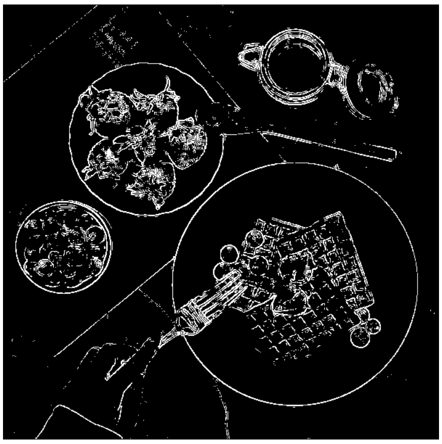
Figure 11: Canny Edge Detection

(c) Using Laplacian of Gaussian edge detection to generate the edge map of **sample1.png** and save it as **result4.png**. Compare **result2.png**, **result3.png**, and **result4.png**, and discuss the differences among these three results.

Result



sample1.png



result4.png

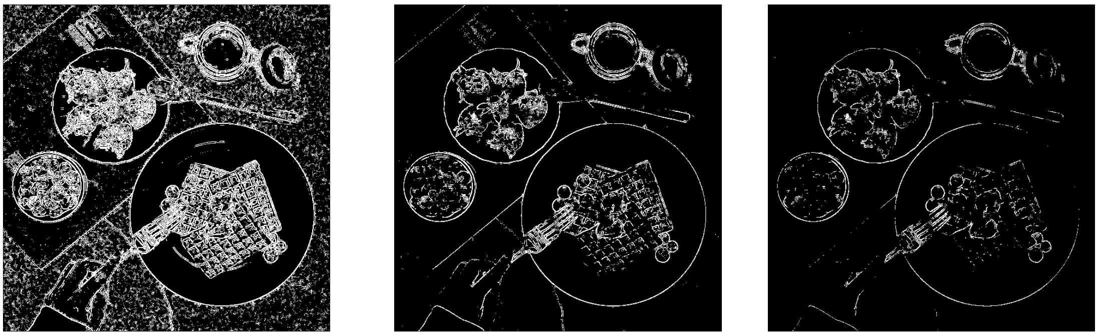
Figure 12: LoG

Approach

建立 $n \times n$ 的 gaussian kernel 和 3×3 laplacians kernel，先後對 image 做 converge，再設定 threshold，判斷是否是 edge point (zero-crossing detection)，步驟很重複，所以簡述自此。

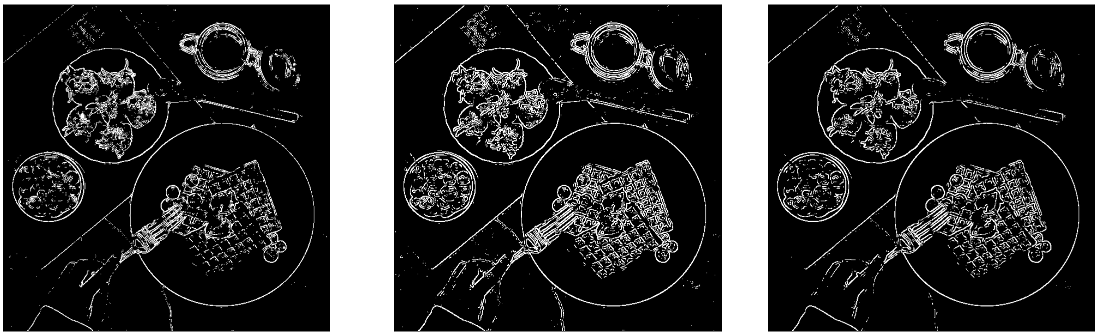
Parameter

將 histogram 畫出來後，我們可以發現 LoG 後的 pixel value 都集中在 0 5 附近，因此我去對各種 gaussian kernel size 和 sigma 去試 threshold，結果如下：



T = 1 | T = 3 | T = 5

Figure 13: LoG w/ gaussian kernel size = 3 / sigma = 1.5

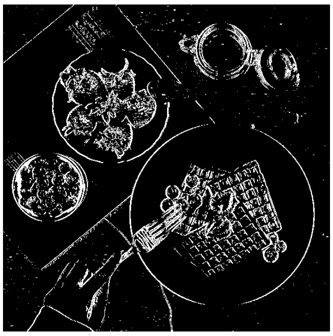


kernel size = 3 / T = 3 | kernel size = 5 / T = 1 | kernel size = 7 / T = 1

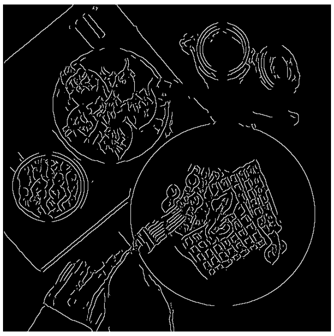
Figure 14: LoG w/ gaussian kernel sigma = 1.5 / Best T

我們可以觀察到在相同的 Gaussian kernel 作用後，Threshold 越低會有越多 noise，反之則 edge 也會消失。而在相同的 sigma 下，kernel 越大，所需要的 threshold 也越低，而且 edge 有變粗變亮的現象。

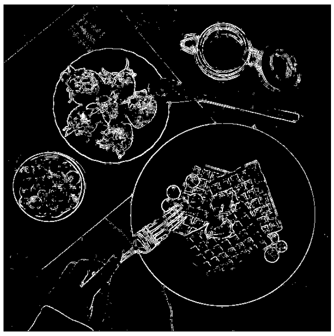
Discussion



result2 (Sobel)



result3 (Canny)



result4 (LoG)

Figure 15: Comparison between edge detection model

我們可以看出 *Sobel* 和 *LoG* 的 edge 較細緻，而 *Canny* 的 edge 因為經過 non-maxmimal surpression 所以粗度只剩一個 pixel，因此叫簡潔所以看起來有點失真。而 *Sobel* 與 *LoG* 比較起來，*LoG* 又顯得更為細緻，*Sobel* 則顯得糊糊的。

(d) Perform edge crispening on **sample2.png** and save the result as **result5.png**. Describe the differences between **sample2.png** and **result5.png**. Please also specify the parameters you used and explain how they influenced the outcome.

Result

sample1.png

result5.png

Figure 16: Edge Crispening

Approach

首先，我建立了一個 high-pass filter 來做對比：

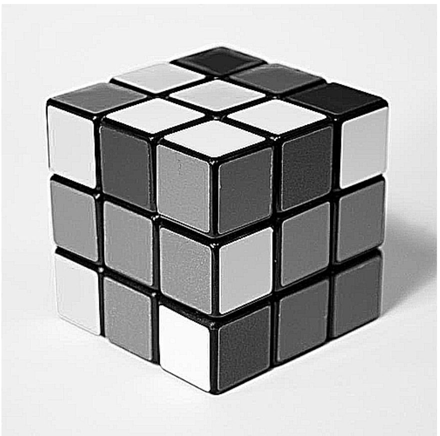
$$K = \begin{bmatrix} -1, & -1, & -1 \\ -1, & 9, & -1 \\ -1, & -1, & -1 \end{bmatrix}$$

另外再使用一個 Gaussian Kernel G 來進行 Unsharp Masking：

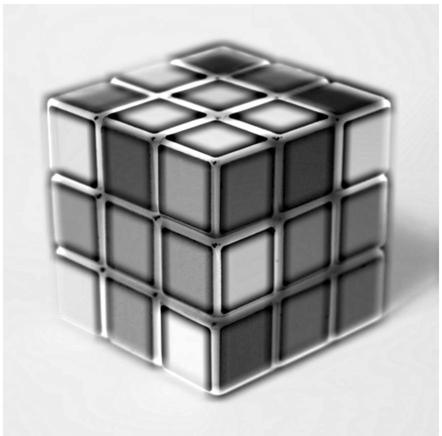
$$G(j,k) = \frac{c}{2c-1} * F(j,k) + \frac{1-c}{2c-1} * G * F(j,k), where\ c \in [\frac{3}{5}, \frac{5}{6}]$$

Parameter

我測試了 Unsharp Masking 中不同的 c，以下是結果：



High-pass only



c = 0.35

c = 0.7

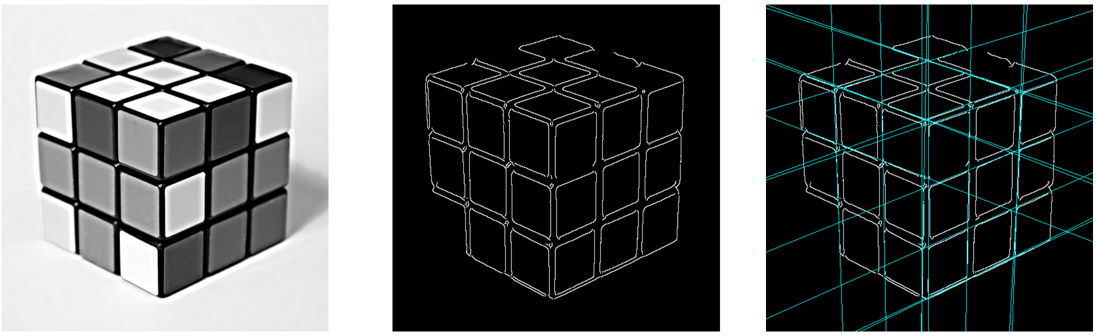
c = 1.0

Figure 17: Unsharp Masking

可以看出在沒有經過 Unsharp masking 前，銳利度十分的高，隨著 c 到了規定的範圍內，就有越柔和的感覺。另外我發現，需要很大的 gaussian kernel size 和 sigma，才會比較有效果，我最後選用 gaussian kernel size = 21, gaussian kernel sigma = 30, c = 0.7。

(e) Perform Canny edge detection on **result5.png** and save the edge map as **result6.png**. Then, apply the Hough transform to **result6.png** and save the resultant image as **result7.png**. What lines can you detect using this method?

Result



result5 (Edge Crispening) result6 (Canny) result7 (Hough)

Figure 18: Hough Transform

Discussion

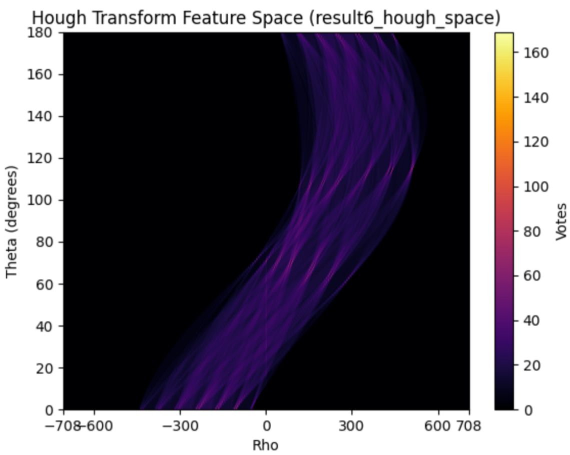
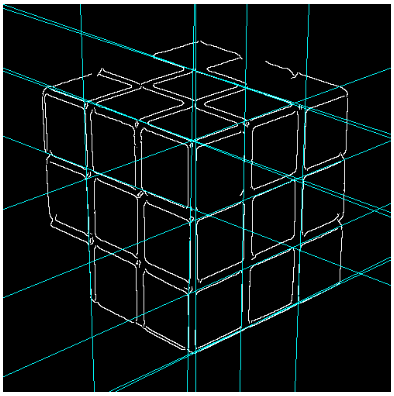
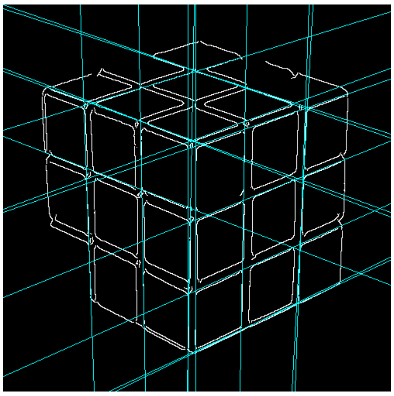


Figure 19: result6 (Hough Feature Space)

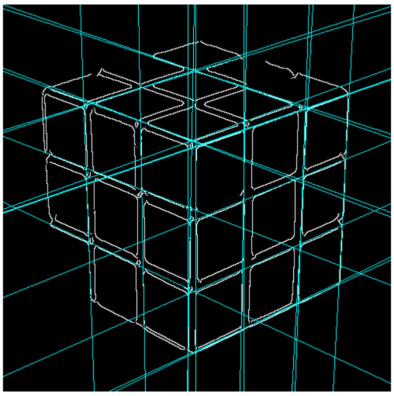
可以看到經過 Canny detect 出來的 edge，將 edge 上每一個 pixel 根據不同 θ 轉換過去 Hough Feature Space 後會是一條曲線，這些曲線重疊最多次的點就很有機會是原本的 edge，因此我們要取 Top_K 個交點，轉換回 Image Space。結果如下：



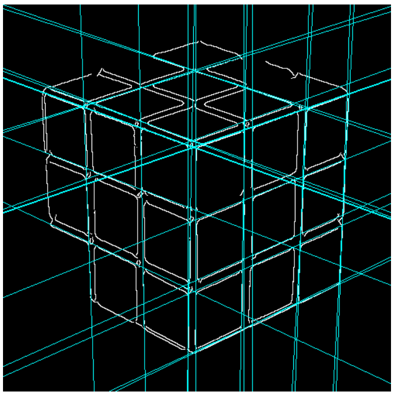
k = 15



k = 20



k = 25



k = 30

Figure 20: Hough Transform

可以看到 k 越大，邊越多，也會也重複的邊出現，因此最後我選擇 k=20，Canny 的參數則為：gaussian kernel size = 5, gaussian kernel sigma = 1.25, sobel kernel size = 3, thres low = 10, thres high = 50

Problem 2 : GEOMETRICAL MODIFICATION

在開始之前，我想先介紹我程式碼的架構，我將 ***Generalized Linear Geometrical Transformations*** 的矩陣系統刻了出來，並且實做了 ***backward treatment*** 以及 ***bilinear interpolation***，因此對於 global 的 shift, scale, rotate 只需要將相對應的矩陣與 image convolve 就好，十分輕鬆。

首先，我們需要將 Image coordinate 與 Catesian coordinate 兩者之間的轉換，為了方便起見，兩者的 origin 我都設定為 $(0, 0)$。因此，對於 Image coordinate 中的一點 (p, q) 與 Catesian coordinate 中的一點 (x, y) 轉換的公式為：

$$\begin{cases} x = q + \frac{1}{2} \\ y = P - \frac{1}{2} - p \end{cases}$$

接下來，需要將座標系擴增為 *Homogeneous Coordinate*，很簡單，多加一個全為 1 的維度即可。

$$\begin{bmatrix} x \\ y \end{bmatrix} \longrightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

處理完座標的轉換，我們就可以研究 ***backward treatment*** 的實作細節。我的思考方式是：「對於 result image 上的一點 (u, v)，就竟是原圖上的哪一點 (p, q) 經過線性映射得到的？」，所以每次我處理一張照片時，順序會是這樣的：

1. 將 result image 的所有座標組合存入一個 list: res-image-coordinates
2. 將 res-image-coordinates 轉成 Catesian 座標: res-cartesian-coordinates
3. 對 res-cartesian-coordinates 施以反向映射 (原本是平移 10 再旋轉 θ，反操作就是旋轉 $-\theta$ 再平移 -10): original-cartesian-coordinates
4. 將 original-cartesian-coordinates 轉成 Image 座標: original-images-coordinates
5. 用 ***bilinear interpolation*** 將原圖上 original-images-coordinates 的值填入 result image

各操作實做程式碼：

```python
# @return in the form of [[x1, y1, 1], [x2, y2, 1], ... , [xn, yn, 1]] (0 <= x < h, 0 <= y < w)
def generate_coords_pairs(h:int, w:int):
    assert h>0 and w>0, "h and w should be positive"
    return np.vstack((np.stack([x for x in np.ndindex(h,w)]).T, np.ones(h*w))).T.astype(np.float64)

# convert image coordinate to cartesian coordinate
def coords_img_to_cartesian(coords:np.ndarray, h:int, w:int):
    assert coords.ndim == 2 and coords.shape[1] == 3, "coords should be in (x,y,1) form"
    return coords[:, [1, 0, 2]] * [1, -1, 1] + [0.5, h-0.5, 0]

# convert cartesian coordinate to image coordinate
def coords_cartesian_to_img(coords:np.ndarray, h:int, w:int):
    assert coords.ndim == 2 and coords.shape[1] == 3, "coords should be in (x,y,1) form"
    return coords[:, [1, 0, 2]] * [-1, 1, 1] + [h-0.5, -0.5, 0]
```

Figure 21: Coordinate Transformation

```python
# The Pipeline for performing a series of transformation
# Will do the transformation in the inverse order to caculate the original coordinates
# operarions:
#   shift:  {"type":"shift", "tx":0.0, "ty":0.0}
#   scale:  {"type":"scale", "sx":0.0, "sy":0.0}
#   rotate: {"type":"rotate", "theta":0.0}
def geometrical_transformation(src:np.ndarray, series:list, h:int, w:int):
    series = series[::-1]
    img_coords = generate_coords_pairs(h, w)
    res_coords = coords_img_to_cartesian(coords=img_coords, h=h, w=w)
    for op in series:
        if op["type"] == "shift":
            res_coords = geometrical_shift(res_coords, tx=op["tx"], ty=op["ty"])
        elif op["type"] == "scale":
            res_coords = geometrical_scale(res_coords, sx=op["sx"], sy=op["sy"])
        elif op["type"] == "rotate":
            res_coords = geometrical_rotate(res_coords, theta=op["theta"])

    return warpAffine_bilinear_interpolation(src=src, coords=res_coords, res_h=h, res_w=w, fill=255)
```

Figure 22: Generalized Linear Geometrical Transformations Pipeline

```python
# coords[]  := the original coordidates on the src (cartesian coordinates form)
# h         := the height of the result image
# w         := the width of the result image
# fill      := the color for blank pixel ([0, 255])
def warpAffine_bilinear_interpolation(src:np.ndarray, coords:np.ndarray, res_h:int, res_w:int, fill:int=0):
    assert fill >= 0 and fill <= 255, "fill should be an interger between [0,255]"
    src_h, src_w = src.shape
    res = np.ones((res_h, res_w)).astype(np.float64) * fill

    img_coords = coords_cartesian_to_img(coords, res_h, res_w)
    for i in range(res_h):
        for j in range(res_w):
            idx = i*res_w+j
            # bilinear interpolation
            if img_coords[idx,0] < 0 or img_coords[idx,0] > src_h-1 or img_coords[idx,1] < 0 or img_coords[idx,1] > src_w-1:
                continue
            y, x = img_coords[idx,0], img_coords[idx,1]
            left_x, left_y = np.floor(x).astype(np.int32), np.floor(y).astype(np.int32)
            right_x, right_y = np.ceil(x).astype(np.int32), np.ceil(y).astype(np.int32)
            assert left_x >= 0 and left_x < src_w and right_x >= 0 and right_x < src_w, f"invalid boundary for x:{x} l:{left_x} r:{right_x}"
            assert left_y >= 0 and left_y < src_h and right_y >= 0 and right_y < src_h, f"invalid boundary for y:{y} l:{left_y} r:{right_y}"

            a = (y-left_y)
            b = (x-left_x)
            res[i,j] = (1-a)*(1-b)*src[left_y, left_x] + (1-a)*b*src[left_y, right_x] + a*(1-b)*src[right_y, left_x] + a*b*src[right_y, right_x]
    return res
```

Figure 23: Bilinear Interpolation

(a) Toothless wants to become stronger. Please design an algorithm to convert **sample3.png** into **sample4.png**. The output results should be saved in **result8.png**, and the output image size is required to be the same as **sample3.png**. Please describe your approach and implementation details clearly. (hint: you may perform rotation, scaling, translation, etc.)

Result

sample3.png

sample4.png

result8.png

Figure 24: Fat Toothless

Approach

觀察 sample4.png，我們可以猜出，在形狀上他一定經過類似 Barrel Distortion，在位置上，他被順時針旋轉約 45°，並且往右上角平移。並且根據他變形的趨勢，我猜測他是先經過 Distortion 才被 rotate 跟 scale。

對於 Barrel Distortion，我參考 https://blog.csdn.net/yangtrees/article/details/9095731，設定好 distortion center 後，對於其距離中心為 r 的每一點 (x, y)，其最終座標使用一個 power function 來近似 Barrel Distortion：

$$R1 = \frac{\sqrt{h^2 + w^2}}{10}, \quad dist = \sqrt{x^2 + y^2}, \quad \begin{cases} new_x = x * (\frac{dist}{R1})^{1.2} \\ new_y = y * (\frac{dist}{R1})^{1.2} \end{cases}$$

式中的 $\frac{dist}{R1}$ 會是介於 1 跟 2 的數字，對他進行 1.2 次方後更可以模擬出非線性的 shift，因此很適合拿來近似 Barrel Distortion。

處理完 Distortion 後，我們就可以很輕易的透過上述介紹的系統將 Toothless 移到正確的位置

Parameter

- **Barrel Distortion**: $r = 200$, $center_x = 270$, $center_y = 270$

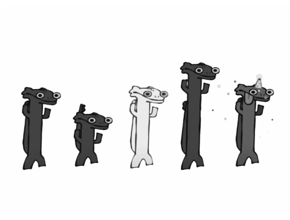
- **Linear Transformation**:
  - (Step1): x 方向平移 -270, y 方向平移 -310
  - (Step2): x 方向放大 1.4 倍, y 方向放大 1.6 倍
  - (Step3): 逆時針旋轉 50°
  - (Step4): x 方向平移 350, y 方向平移 -380

```python
def p2_a():
    sample3 = read_image("sample3.png", 0, options=cv2.IMREAD_GRAYSCALE)
    warped_sample3 = warping(sample3, center_x=270, center_y=270, radius=200)
    h, w = warped_sample3.shape
    series = [
        {"type":"shift", "tx":-270, "ty":-310},
        {"type":"scale", "sx":1.4, "sy":1.6},
        {"type":"rotate", "theta":-50.0},
        {"type":"shift", "tx":350, "ty":380},
    ]
    result8 = geometrical_transformation(src=warped_sample3, series=series, h=h, w=w)
    write_image("result8", result8)
```

Figure 25: 簡潔的程式碼

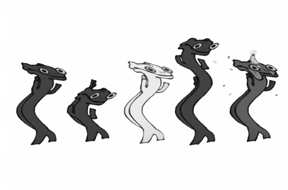
(b) Toothless and his friends are practicing their new dance moves. Please design an algorithm to convert **sample5.png** into **sample6.png**. The output results should be saved as **result9.png**, and the output image size is required to be the same as **sample5.png**. Please describe the details of your method and provide some discussion on the design approach, results, and differences between **result9.png** and **sample6.png**.

Result



sample5.png

sample6.png



result9.png

Figure 26: Fat Toothless

觀察 sample6.png，我們可以猜出他經過某種非線性的水平方向 shifting，因此這題我使用 $f(x) = \sin x$ 來模擬這種波形。給定一個最大位移量 $dist$，週期 T，相

位差 k 後，我使用的公式是：

$$new_x = x + dist * \sin(2\pi * \frac{x+k}{T})$$

我們先對 sample5 和 sample6 做一些輔助線 (線與線間隔 100 pixel)：

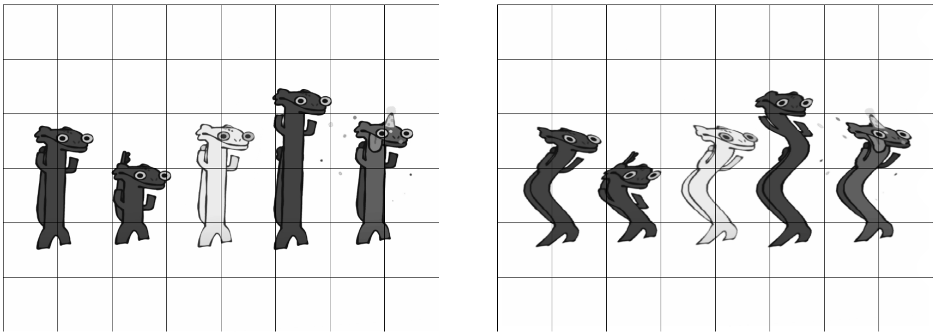


Figure 27: sample5 and sample6 with grid

這樣就很好觀察得出哪裡是波鋒，哪裡是波谷。我們可以看出大約在**頭部**會是波鋒 ($\sin(x)$ 為 1)，在**腰部**會是波谷 ($\sin(x)$ 為 -1)，因此可以判斷出週期約為 120 pixel，進而推得相位差大約為 50 pixel，至於最大位移距離則可以判斷大約為 25 pixel。將這些參數帶入上面的轉換式之後，將其原本座標位置平移過去就會是 resul9 了。

Parameter

- $dist = 25$
- $T = 150$
- $k = 60$

Bonus

上課提到的能不能讓不同隻扭出不一樣的方向，其實只需要去算當前 x 座標在哪，來當作扭哪邊的依據就可以很輕鬆地達成！

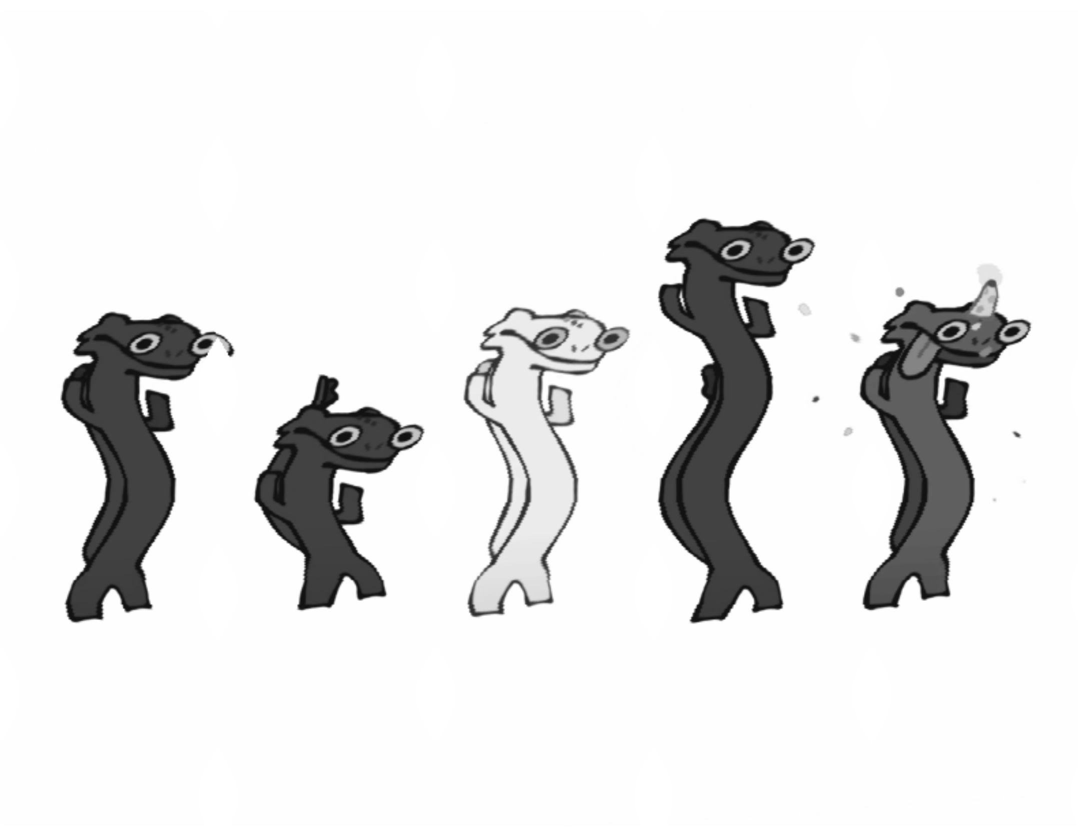


Figure 28: result10 (p2(b) bonus)