Digital Image Processing - Homework Assignment#1

B13902022 賴昱錡

Due: 10/13/2025

1 Exercise 1 - Scaling

1.1 Code

This part is corresponding to the subproblem 1 in exercise 1. I use `opencv` module and the `resize()` function in it, the function covers common methods of interpolations (e.g. `cv2.INTER_LINEAR` for bilinear interpolation, `cv2.INTER_CUBIC` for bicubic interpolation). Just fill the scaled width and height (unit: pixels) of the image in the function, we can get our result.

How to run this code (`scale.py`)? Just run `python3 scale.py` (And ensure the modules opencv-python and numpy are installed using methods like `pip install opencv-python`, `numpy`, `sudo apt-get install python3-opencv`) and enter the scaling factor the result will be saved in the current working directory with scaling factor as prefix.

```python
import cv2 as cv
import numpy as np
try:
    scale = float(input())
except:
    print('Invalid Input: Number is expected')
filename = 'me.jpg'

img = cv.imread(filename)
height, width = img.shape[0], img.shape[1]

height = int(scale * height)
width = int(scale * width)

result = cv.resize(img, (width, height), interpolation = cv.INTER_CUBIC)
# or INTER_LINEAR
cv.imwrite(f'{scale}_{filename}', result)
```

1.2 Demo

This part is corresponding to the subprolem 2 in exercise 1.

Figure 1: Bilinear interpolation with scaling factor of 0.13 (images/013_me_linear.jpg)

Figure 2: Bilinear interpolation with scaling factor of 0.13 (images/013_me_cubic.jpg)

Figure 3: Bicubic interpolation with scaling factor of 9 (images/9_me_cubic.png)

Figure 4: Bilinear interpolation with scaling factor of 9 (images/9_me_linear.png)

1.3 Comparison

When enlarging or resampling an image, bilinear interpolation estimates the value of each new pixel by taking a weighted average of the four closest pixels in the original image. This method is computationally efficient and produces smoother results than the simplest method (nearest-neighbor). However, because it only considers a small neighborhood, **bilinear interpolation tends to blur edges and lose fine details**, **making the image appear soft or slightly out of focus when scaled up significantly.**

On the other hand, bicubic interpolation uses a larger neighborhood of 16 surrounding pixels and fits cubic polynomials to estimate pixel values. This allows it to preserve edges and textures more effectively, **producing sharper and more visually pleasing images compared to bilinear interpolation**, this propery is especially evident when we shrink the image. The trade-off is that bicubic interpolation requires more computation and can sometimes introduce minor artifacts like halo effects near strong edges.

In general, bilinear interpolation is preferred when speed is more important than quality, such as in real-time applications, while bicubic interpolation is favored when higher-quality image scaling is required, such as in photo editing or printing. Thus, bicubic usually gives superior results, but at the cost of additional processing time.

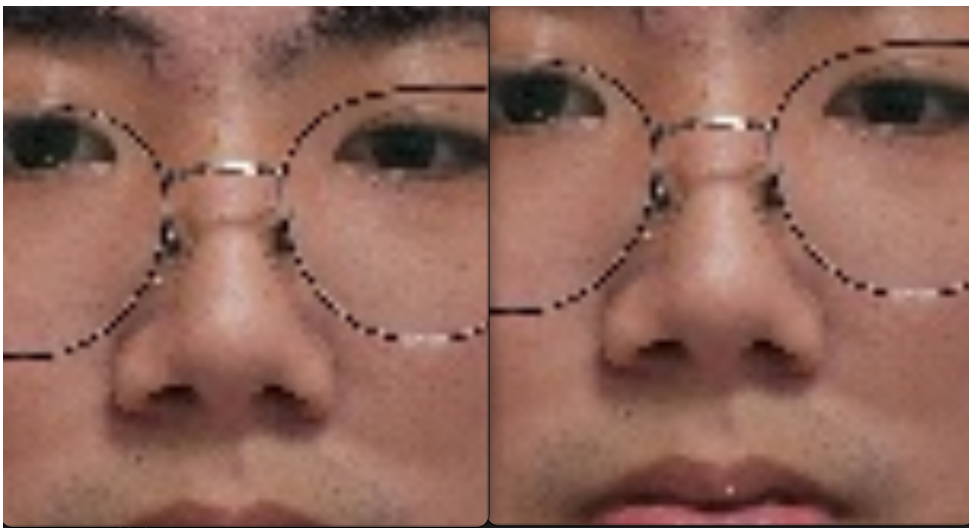


Figure 5: The comparison of details when scaling factor is 0.13

Figure 6: The comparison of details when scaling factor is 9

1.4 Bicubic Interpolation

This part is corresponding to the subprolem 4 in exercise 1.

Bicubic interpolation estimates a value at a 2D grid point using a 4×4 neighborhood (16 points) for smoother results. It extends 1D cubic splines separably:

1. For each of 4 rows, compute 1D cubic interpolation across 4 columns using offset t:

$$p_k = \sum_{m=0}^{3} c_m(t) \cdot f(i+m-1, j+k-1), \quad k = 0 \ldots 3$$

(e.g., Catmull-Rom basis: $c_0(t) = -0.5t^3 + t^2 - 0.5t$, etc.)

2. Interpolate the 4 p_k vertically using offset s:

$$p(x, y) = \sum_{k=0}^{3} c_k(s) \cdot p_k.$$

This approximates a degree-3 surface, reducing blur/artifacts in tasks like image scaling.
In bilinear interpolation, we uses 2×2 neighborhood (4 points) for linear weighting:

$$p(x, y) = (1-t)(1-s)f(i, j) + t(1-s)f(i+1, j) + (1-t)sf(i, j+1) + tsf(i+1, j+1).$$

Separable: horizontal linears, then vertical.

The comparison between the complexity of two methods can be summarized as follows, we can see bicubic trades ∼4× computations for better detail preservation; bilinear prioritizes speed.

Aspect	**Bilinear**	**Bicubic**
Pixels Used	4 (2×2)	16 (4×4)
Operations per Pixel	∼4 mult + ∼2 add	∼20 mult + ∼15 add (4× cubic horiz + 1 vert)
Speed	Very fast ($O(1)$, real-time ok)	4 5× slower ($O(1)$, but higher cost)
Quality	Basic, can blur	Sharper, smoother gradients

2 Exercise 2 - Distortion

2.1

This part is corresponding to the subprolem 1 in exercise 2.

1. Brown–Conrady Model of Radial Distortion

The Brown–Conrady model expresses the relation between the ideal (undistorted) image point (x, y) and the distorted image point (x_d, y_d) as:

$$x_d = x \cdot \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \dots\right)$$
$$y_d = y \cdot \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \dots\right)$$

where:

- (x, y) are normalized image coordinates (centered at the principal point),
- $r^2 = x^2 + y^2$ is the squared radial distance from the optical axis,
- $k_1, k_2, k_3, \dots$ are the radial distortion coefficients.

The sign and magnitude of the coefficients determine whether the lens exhibits *barrel distortion* or *pincushion distortion*.

2. Barrel Distortion

- **Definition:** Straight lines appear to bulge outwards, like the sides of a barrel.
- **Mathematical explanation:** Occurs when $k_1 < 0$ (dominant case). The scaling factor

$$1 + k_1 r^2 + k_2 r^4 + \dots$$

becomes *smaller* as r increases. Thus, points farther from the image center are mapped closer inward, compressing the edges and making straight lines look convex.

Typical example: wide-angle or fisheye lenses.

3. Pincushion Distortion

- **Definition:** Straight lines appear to bend inward, like the edges of a pincushion.
- **Mathematical explanation:** Occurs when $k_1 > 0$ (dominant case). The scaling factor

$$1 + k_1 r^2 + k_2 r^4 + \dots$$

grows with r. Thus, points farther from the image center are pushed outward, stretching the edges and making straight lines bow inward.

Typical example: telephoto lenses.

4. Visual Summary

- If the radial factor decreases with r: **Barrel distortion** ($k_1 < 0$).
- If the radial factor increases with r: **Pincushion distortion** ($k_1 > 0$).

2.2 Code

This part is corresponding to the subprolem 2 in exercise 2.

I use `cv2.remap()` to achieve the lens distortion. We can express the remap for every pixel location (x, y) as:

$$g(x, y) = f(h(x, y))$$

where $g(x, y)$ is the remapped image, the source image $f()$ and $h(x, y)$ is the mapping function that operates on (x, y). The remapping function is referenced from the last section. Since we want the pixels in original image "move", i.e., the pixels in new position should have the same value of original position, we need to negate k_1 before sending k_1 to our distortion function.

The rest of details of my implementation are all in the comments of my code. (as follows)

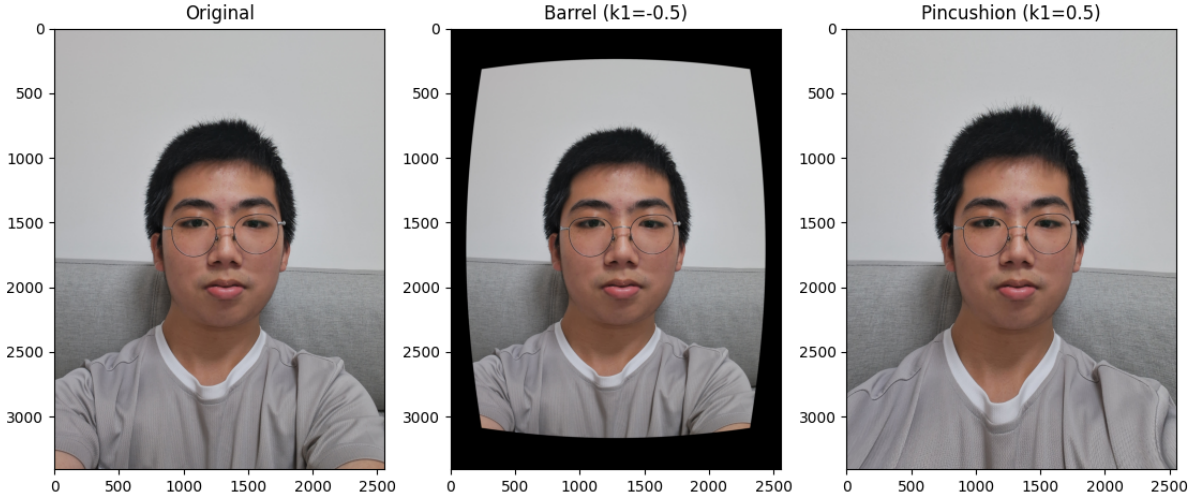
How to run this code (`distort.py`)? Just run `python3 distort.py` (And ensure the modules opencv-python, numpy and matplotlib are installed using methods like `pip install opencv-python, numpy, matplotlib`, `sudo apt-get install python3-opencv`) and change the value of variable `x` in this code, after running it, we can see three images shown, they are original images, barrel distorted image ($k_1 = -x$), pincushion distorted image ($k_1 = x$)

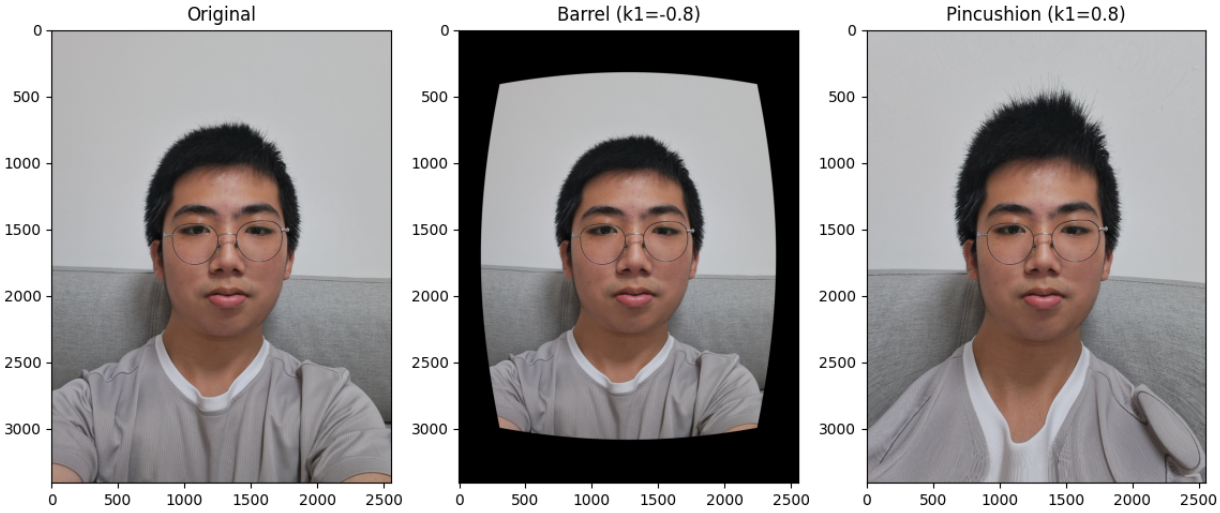
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

def apply_radial_distortion(img, k1=0.0, k2=0.0, k3=0.0):
    k1 = -k1
    k2 = -k2
    h, w = img.shape[:2]
    cx, cy = w / 2.0, h / 2.0  # principal point = image center
    fx = fy = w                # focal length (heuristic choice)

    # Build meshgrid of pixel coordinates
    x = np.linspace(0, w-1, w)
    y = np.linspace(0, h-1, h)
    xx, yy = np.meshgrid(x, y)

    # Normalize to camera coordinates
```

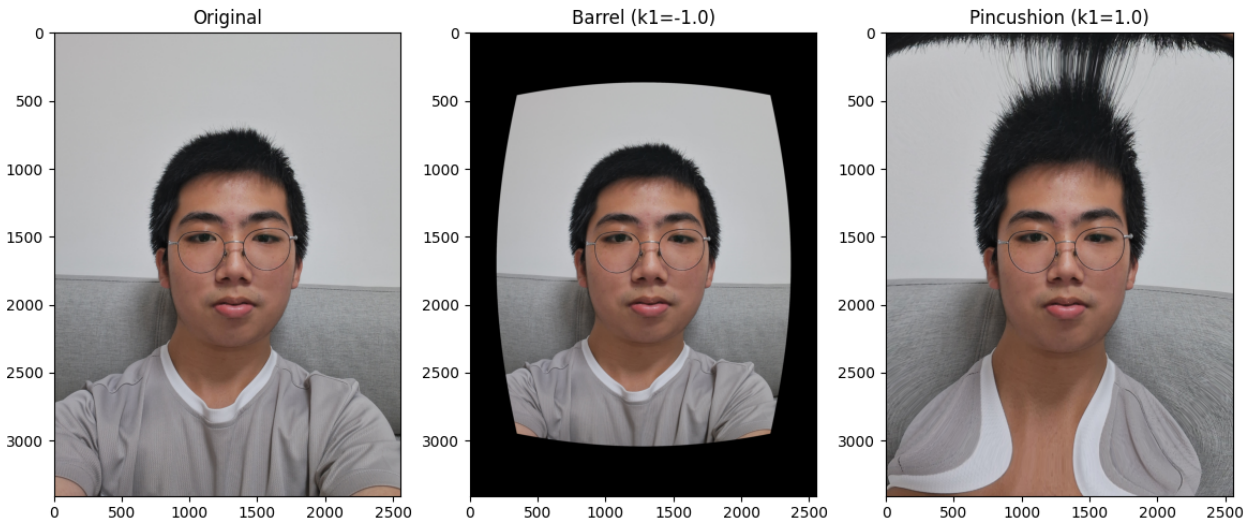
```
    x_norm = (xx - cx) / fx
    y_norm = (yy - cy) / fy
    r2 = x_norm**2 + y_norm**2

    # Apply radial distortion model
    factor = 1 + k1*r2 + k2*r2**2 + k3*r2**3
    x_distorted = x_norm * factor
    y_distorted = y_norm * factor

    # Convert back to pixel coordinates
    u = x_distorted * fx + cx
    v = y_distorted * fy + cy

    # Build remap maps (must be float32 for cv2.remap)
    map_x = u.astype(np.float32)
    map_y = v.astype(np.float32)

    # Warp the image
    distorted = cv2.remap(img, map_x, map_y, interpolation=cv2.INTER_LINEAR,
    ↪  borderMode=cv2.BORDER_CONSTANT)
    return distorted


# Load image
img = cv2.imread("me.jpg")    # Replace with your image
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

x = 0.5

# Apply barrel (k1 < 0) and pincushion (k1 > 0)
barrel = apply_radial_distortion(img, k1=-x)
pincushion = apply_radial_distortion(img, k1=x)

# Show results
plt.figure(figsize=(12,6))
plt.subplot(1,3,1), plt.imshow(img), plt.title("Original")
plt.subplot(1,3,2), plt.imshow(barrel), plt.title(f"Barrel (k1={-x})")
plt.subplot(1,3,3), plt.imshow(pincushion), plt.title(f"Pincushion (k1={x})")
plt.show()
```

2.3 Demo

This part is corresponding to the subproblem 3 and 4 in exercise 2. The images (Figure_[1,2,3].png) are put under the `images` folder.






Original
Barrel (k1=-1.0)
Pincushion (k1=1.0)
0
500
1000
1500
2000
2500
3000