Image Processing HW2

tags: `2019imageprocess` `HW`

url https://hackmd.io/4M1K7ooPSP6oxA1BxDKKqQ (https://hackmd.io/4M1K7ooPSP6oxA1BxDKKqQ)

github https://github.com/linnil1/2019ImageProcessing/tree/master/hw2 (https://github.com/linnil1/2019ImageProcessing/tree/master/hw2)

HW

2.5, 2.12, 2.18, 2.36, 3.12, 3.21

2.5

What's the resolution that 2048x2048 to fit in 5x5cm

> $2048/50 = 40.96$ (lines per mm)

What's the resolution that 2048x2048 fit in 2x2inches.

> $2048/2 = 1024$ (dpi)

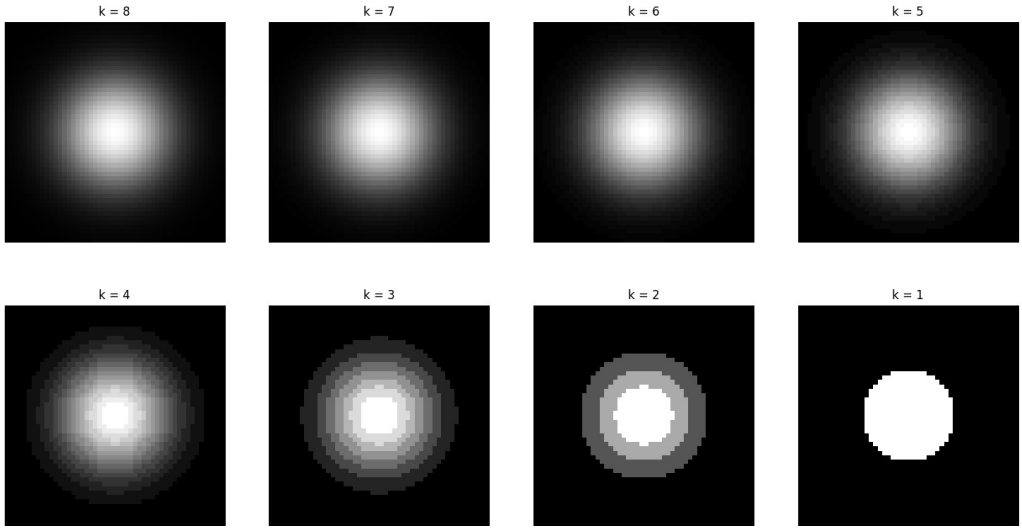
2.12

The intensity distribution: $i(x,y) = Ke^{-[(x-x_0)^2+(y-y_0)^2]}$ is quantized using k bits. What's is the highest value of k that will cause visible false contouring?

It's very easy for numpy to lower the quantized.

```
k = (2 ** i)
new_m = np.uint(m // k * k)
```

The result:



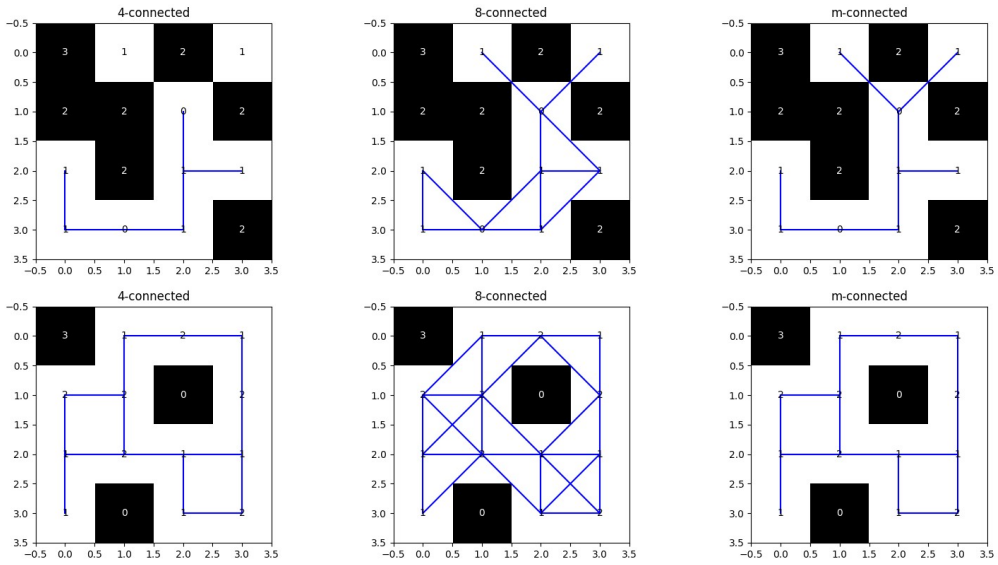
We can find the false contouring when $k = 3$, and a little in $k = 4$. The code is implemented in `hw2/questions.py` .

2.18

Find the lengths of the shortest 4, 8, m path between p and q in the following map. Try it when $V = \{0, 1\}$ or $V = \{1, 2\}$.

```
 3 1 2 1q
 2 2 0 2
 1 2 1 1
p1 0 1 2
```

Just implemented the connection rule. The result shown in below.



I didn't write the shortest path algorithm because it's not the question I need to solve in the lesson. The answer of shortest path are: ∞ , 4, 5, 6, 4, 6.

The code is implemented in `hw2/questions.py` .

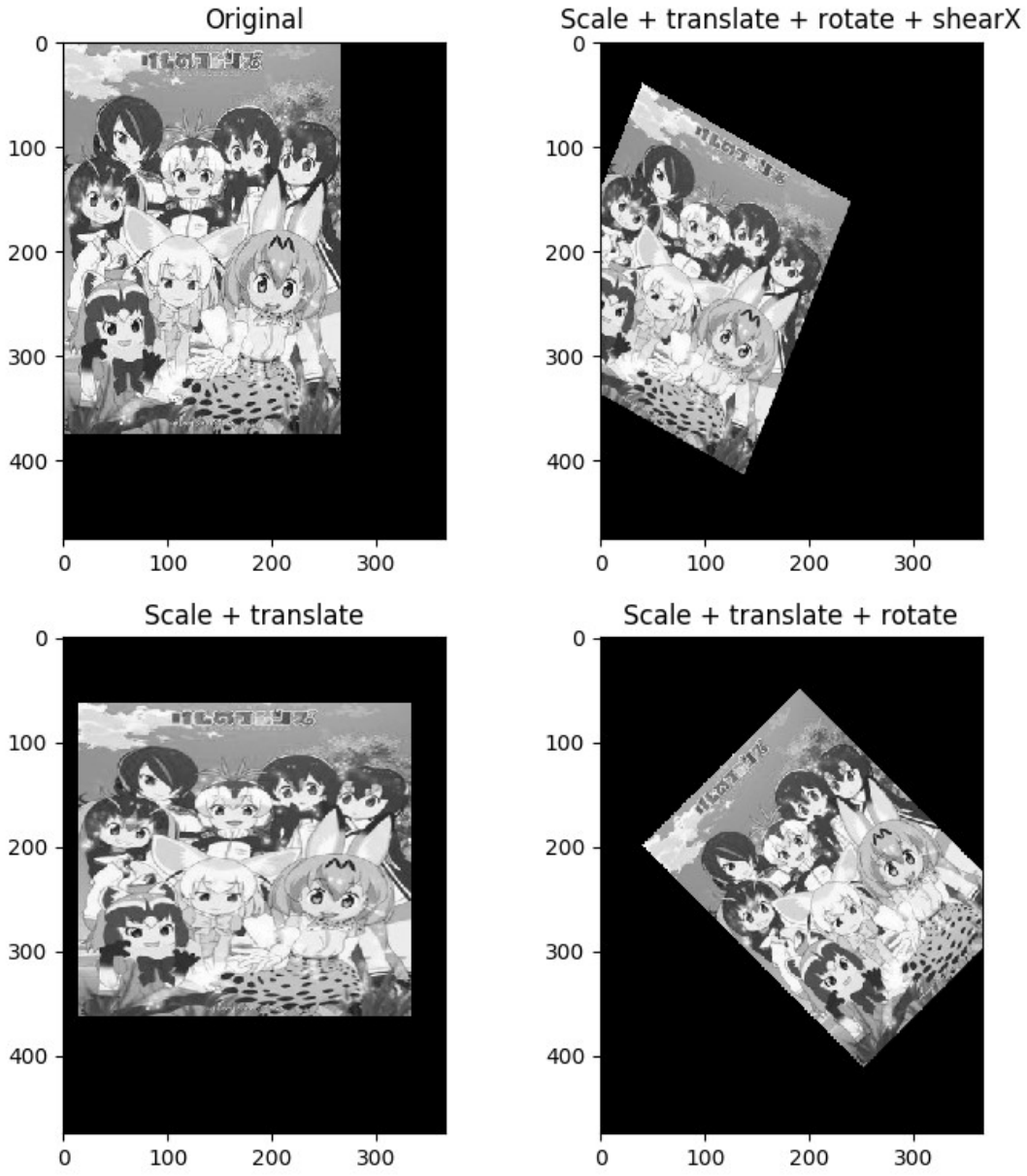
2.36

Provide single. composite transformation for performing the following operations

- scaling and translations
- Scaling, translation, and rotation
- Vertical shear, scaling, translation, and rotation.
- Does the order of multiplication of the individual matrics to produce a single transformations make a difference?

All operations and details provided in `hw2/test_affine.py` . I add 100x100 more padding to image.

```
affine0 = Base()
affine1 = Base() * scaleX(0.8) * scaleY(1.2) *
affine2 = Base() * scaleX(.8) * scaleY(.8) * translateX(250) * translateY(50) * r
affine3 = Base() * scaleX(.8) * scaleY(.8) * translateX(50) * translateY(50) * sh
```



And the order of transformations make difference

```
>>> affine4 = Base() * scaleX(10) * translateX(50)
>>> affine5 = Base() * translateX(50) * scaleX(10)
>>> print(affine4 - affine5)
[[  0.   0. 450.]
 [  0.   0.   0.]
 [  0.   0.   0.]]
```

It's same as $(10x + 50)$ vs $(x + 50) * 10$.

3.12

Find the transformation from $P_r(r) = 2 - 2r$ to $P_z(z) = 2z$

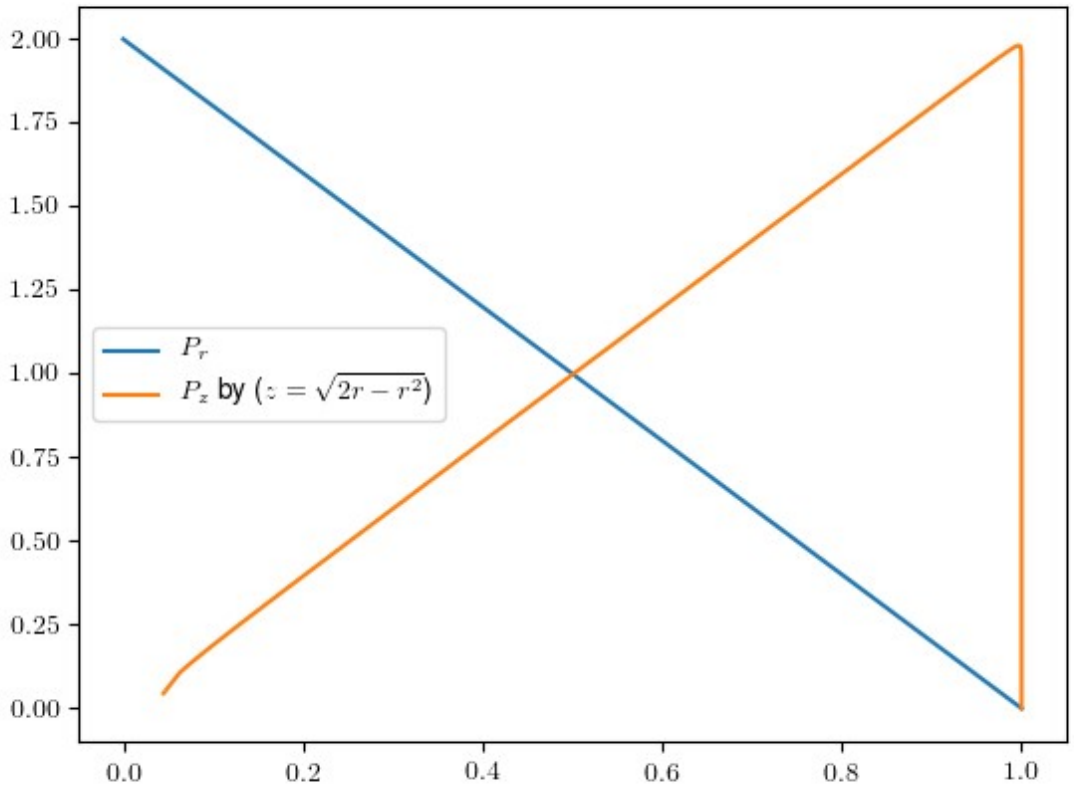
$$s = T(r) = \int_0^r P_r(w)dw = 2r - r^2$$

$$s = T(z) = \int_0^z P_z(w)dw = z^2$$

Then, the inverse of T(z) is easy.

$$z = \sqrt{2r - r^2}$$

I plot it out, the front one and last one has a little error.



3.21

Give convolution of the two.

$$w = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad f = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$Ans = \begin{bmatrix} 16 & 16 & 16 \\ 16 & 16 & 16 \\ 16 & 16 & 16 \end{bmatrix}$$

1. Read color image

Read a color BMP or JPEG image file and display it on the screen. You may use the functions provided by Qt, OpenCV, or MATLAB to read and display an image file. (10%)

> Using `cv2.imread` . Then, change the default BGR format to RGB.

2. Convert to gray scale

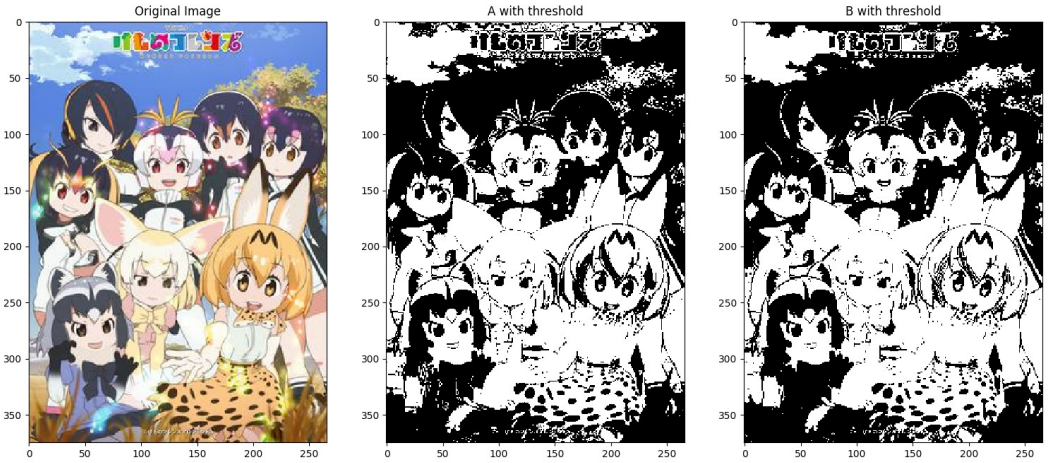
Convert a color image into a grayscale image using the following equations:

- A. GRAY = $(R + G + B)/3.0$
- B. GRAY = $0.299 * R + 0.587 * G + 0.114 * B$

> In numpy, the simplest solution of this operation is using dot: `img.dot([1/3, 1/3, 1/3])` and `img.dot([0.299, 0.587, 0.114])`

Compare the grayscale images obtained from the above equations. One way to compare the difference between two images is by image subtraction (10%)

This two gray scale method look similar, but I think latter one looks more colorful.



Then, I set the threshold on it, the difference is larger. The latter on can easily separate the sky and the clould one the top of image.

I plot the difference from A to B. There is lots of difference in the yellow and red area, which caused by the difference in weights when calculating. Gray B method is much perfer to red.

3. Histogram

Determine and display the histogram of a grayscale image. (10%)

> Deplicated in HW1

4. Binarized

Implement a manual threshold function to convert a grayscale image into a binary image. (10%)

> This still one line for numpy: `return img > threshold` . The resultes are shown in question 2.

5. Resize

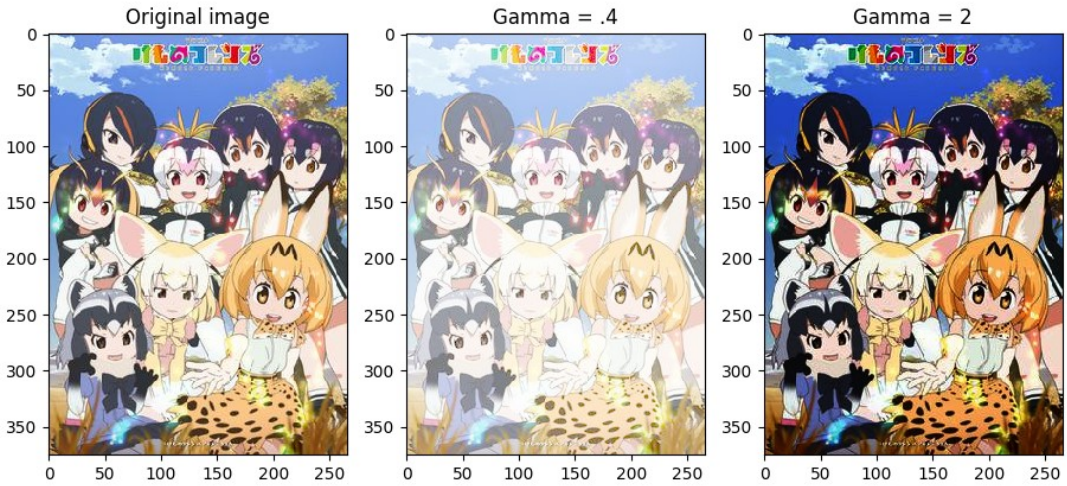
Implement a function to adjust the spatial resolution (enlarge or shrink) and grayscale levels of an image. Use interpolation on enlarging the image. (10%)

Using interpolation, the detail of the algorithm is described in the below section. I choose `Bilinear` to resize the image.

6. Adjust brightness and constrast

Implement a function to adjust the brightness and constrast of an image. (10%)

Gamma correction can adjust constrast and a little birghtness in the same time.

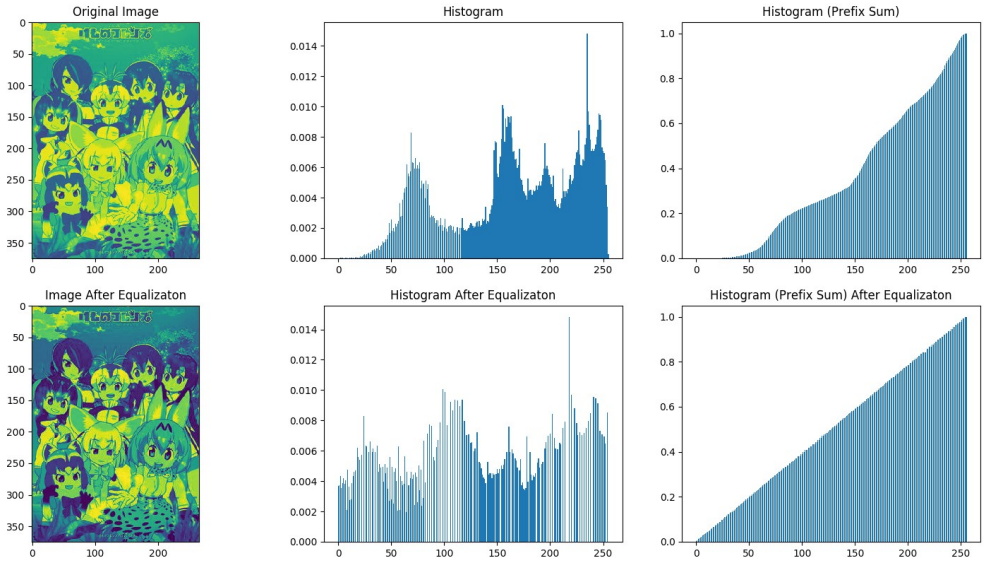


7. Histogram equalization

Implement a histogram equalization function for automatic constrast adjustment. (10%)

It’s very easy for numpy user. Only three lines you need to code.

```
count_old = getHist(img)  # The function used in hw1
map_value = np.cumsum(count_old)  # prefix sum
return map_value[img]  # mapping
```

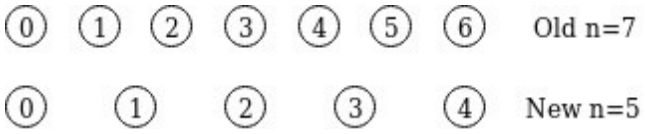


The result is pretty amazing. It’s a much clear image after histogram equalization.

Algorithm

Interpolation

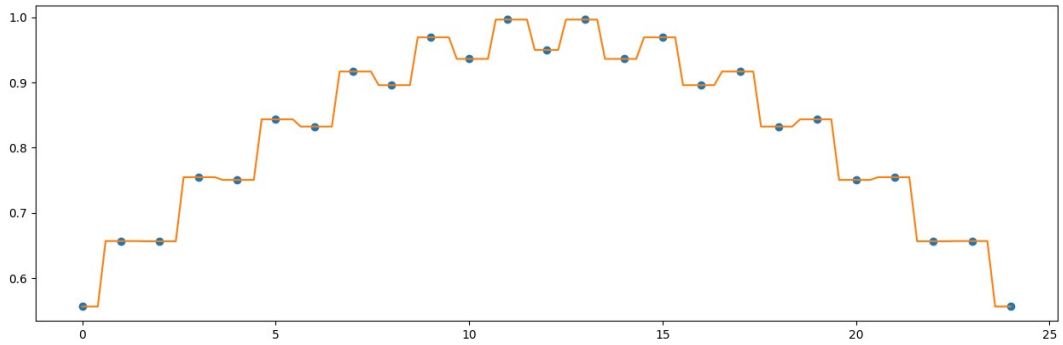
First, we draw our alignment technique and test on 1D array.



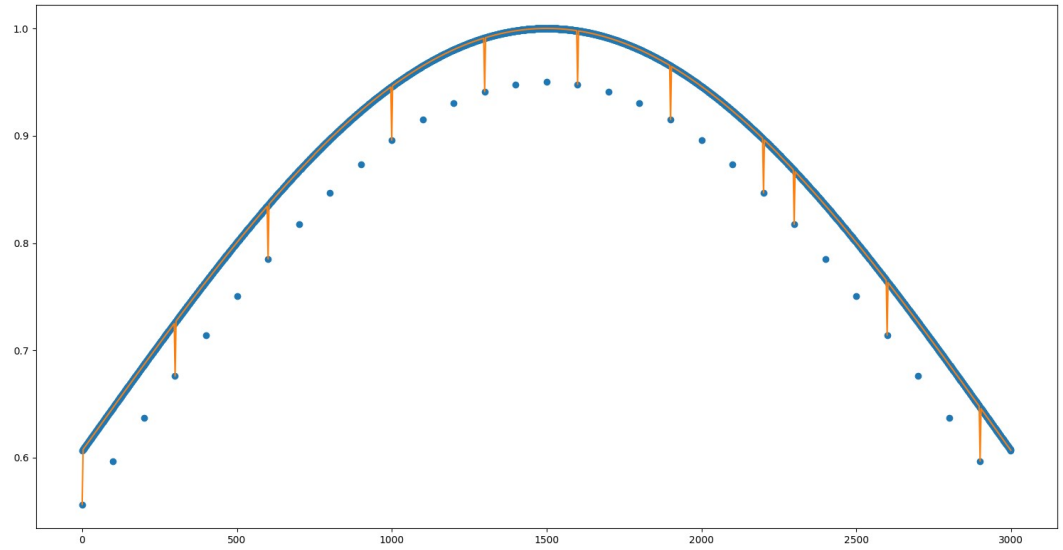
In above image, we treat pixel in the image as a point, and align the first and last one.

For example, we calculate the location mapping for index 3 of the new array. First, calculate the inverse location in old array: $(7-1)/(5-1)*3=4.5$, 4.5 is the corresponsed location, then interpolate 4.5 from it’s neightborhood (maybe the average of 4 and 5).

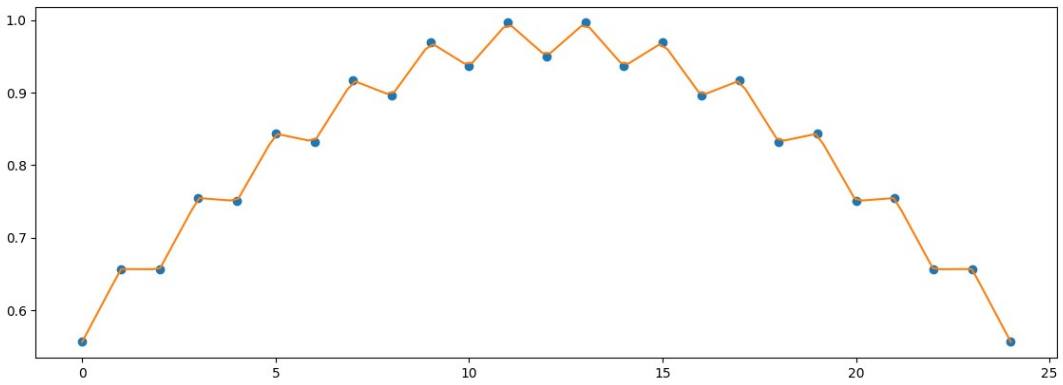
- Nearest Neighbor(Upsample)



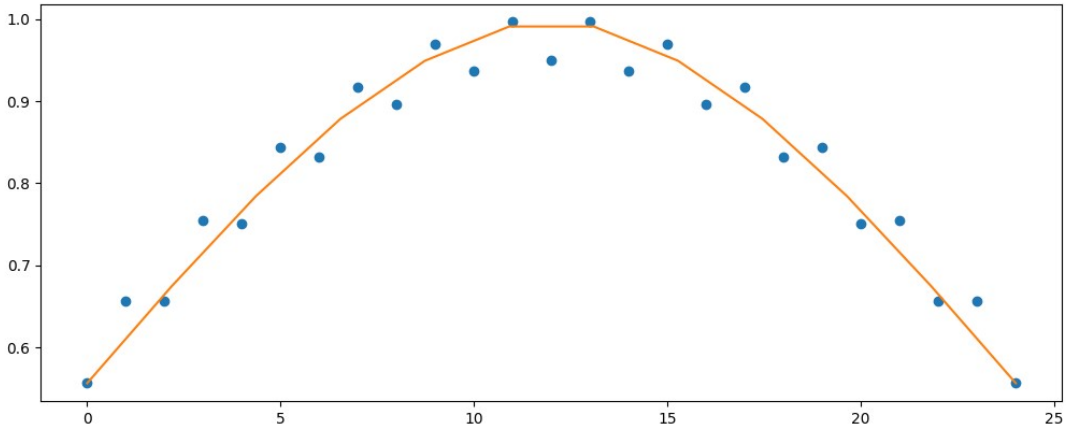
- Nearest Neighbor(Downsample)



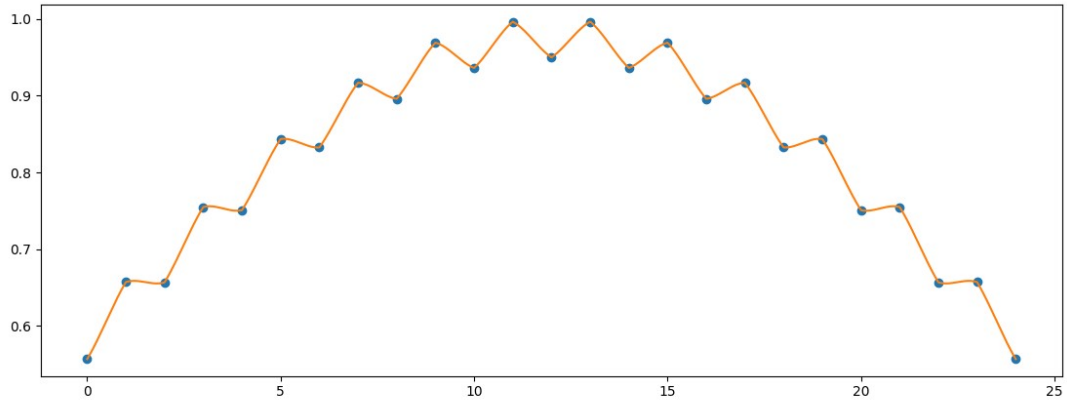
- Linear(upsample)



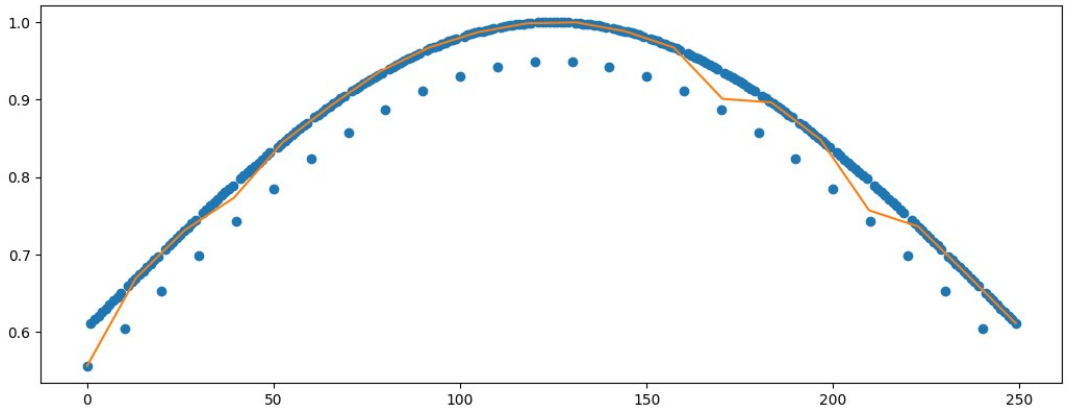
- Linear(Downsample)



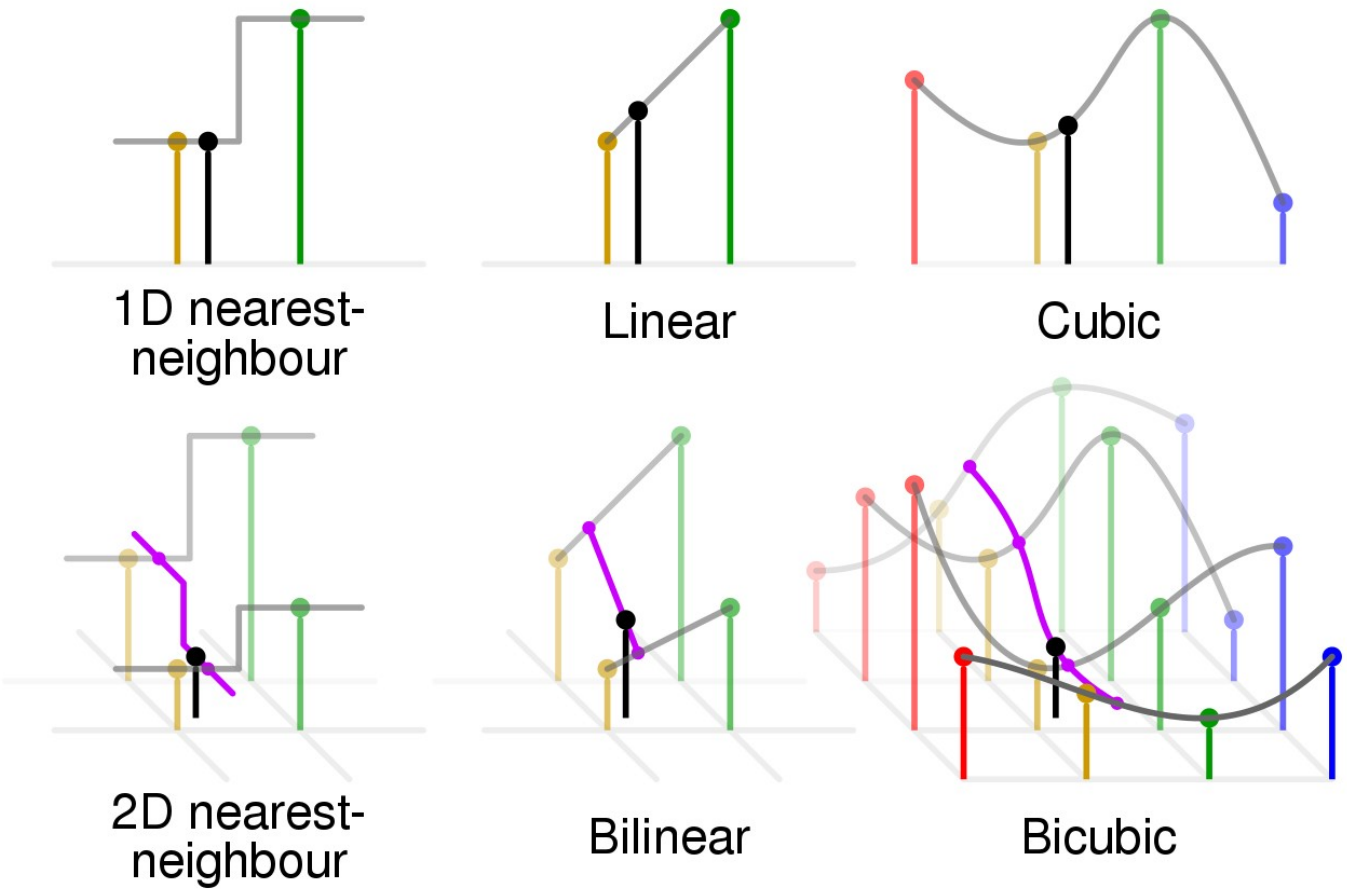
- Cubic (Upsample)



- Cubic (Downsample)

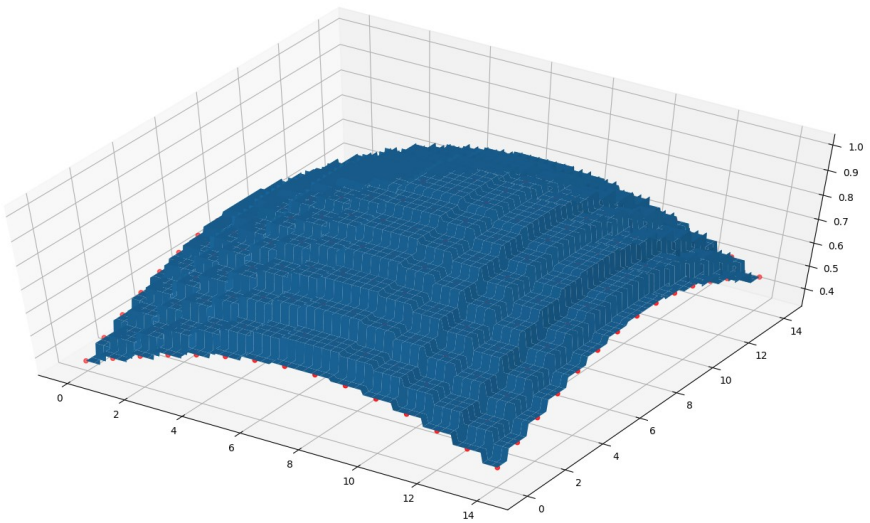


For 2D, using the same method on each dimention. The concept of multivariate intepolation shown below.

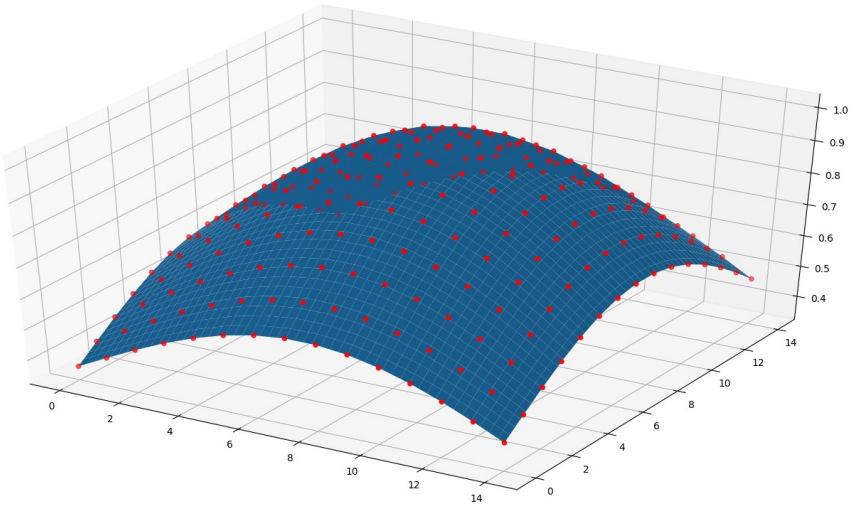


Read more in this reference (https://en.wikipedia.org/wiki/Multivariate_interpolation).

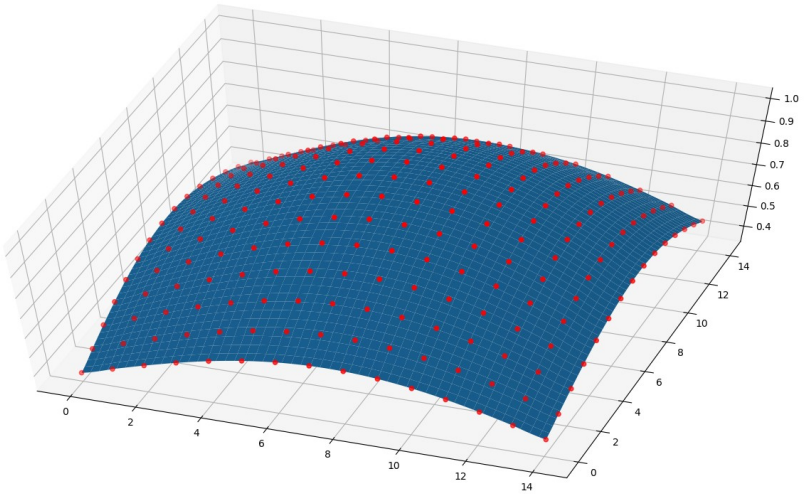
- Nearest Neighbor



- BiLinear



- Bicubic



You can find that bicubic is more smoother than bilinear, however, it takes a lot of time. Without any optimization(Hard to code), the result tested on my laptop shown as below. (Upsample to 200x200 and run 10 times)

	Bilinear	**Bicubic**
Time	1.075s	10.359s

You can find the code in `test_1D_interpolation.py` or `test_2D_interpolation.py`

Optimize interpolation

If you use numpy, you may make full used of broadcasting and indexing. Recommand to use meshgrid to get the locations.

```
def linear(q, v1, v2):
    return v1 + q * (v2 - v1)


data = np.pad(img, ((0, 1), (0, 1)), 'constant')
# prepare x,y
ksizex = img.shape[0] / new_shape[0]
ksizey = img.shape[1] / new_shape[1]
y, x = np.meshgrid(np.arange(new_shape[1]) * ksizey,
                   np.arange(new_shape[0]) * ksizex)
int_x = np.array(x, dtype=np.int32)
int_y = np.array(y, dtype=np.int32)

# bilinear
return linear(x - int_x,
              linear(y - int_y,
                     data[int_x,     int_y],
                     data[int_x,     int_y + 1]),
              linear(y - int_y,
                     data[int_x + 1, int_y],
                     data[int_x + 1, int_y + 1]))
```

I test 10 times for transforming to 500x500 image.

	Iteration	**numpy**
Time	13.860s	0.255s

The other advantage of separting the calculation of location transformaing is easliy to apply affine transformation.

```
y, x = np.meshgrid(np.arange(new_img.shape[1]),
                   np.arange(new_img.shape[0]))
xyz = np.stack([x, y, np.ones(new_img.shape)], 2)
affine = np.array(affine ** -1).T
pos = xyz.dot(affine)
x, y = pos[:, :, 0], pos[:, :, 1]
...
```

Usage

Install on local mechine

```
sudo pip3 install -U numpy matplotlib PyQtChart
sudo apt update
sudo apt install -y python3-pyqt5
```

or `./setup.sh`

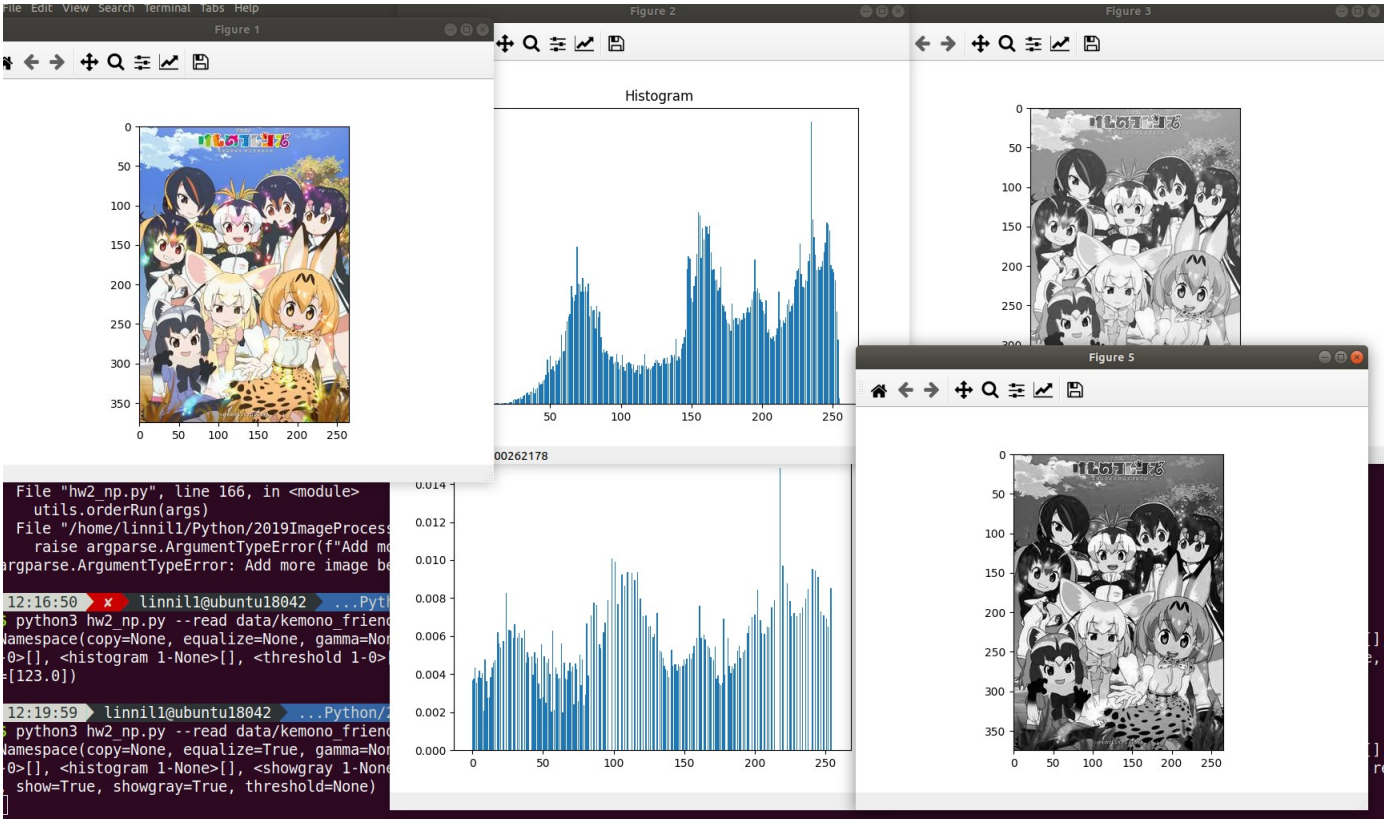
Install by Docker

```
docker build -t linnil1/2019imageprocess .
docker run -ti --rm \
    --user $(id -u) \
    -v $PWD:/app \
    -v /tmp/.X11-unix:/tmp/.X11-unix \
    -e DISPLAY=$DISPLAY \
    linnil1/2019imageprocess python3 qt.py
```

Without QT

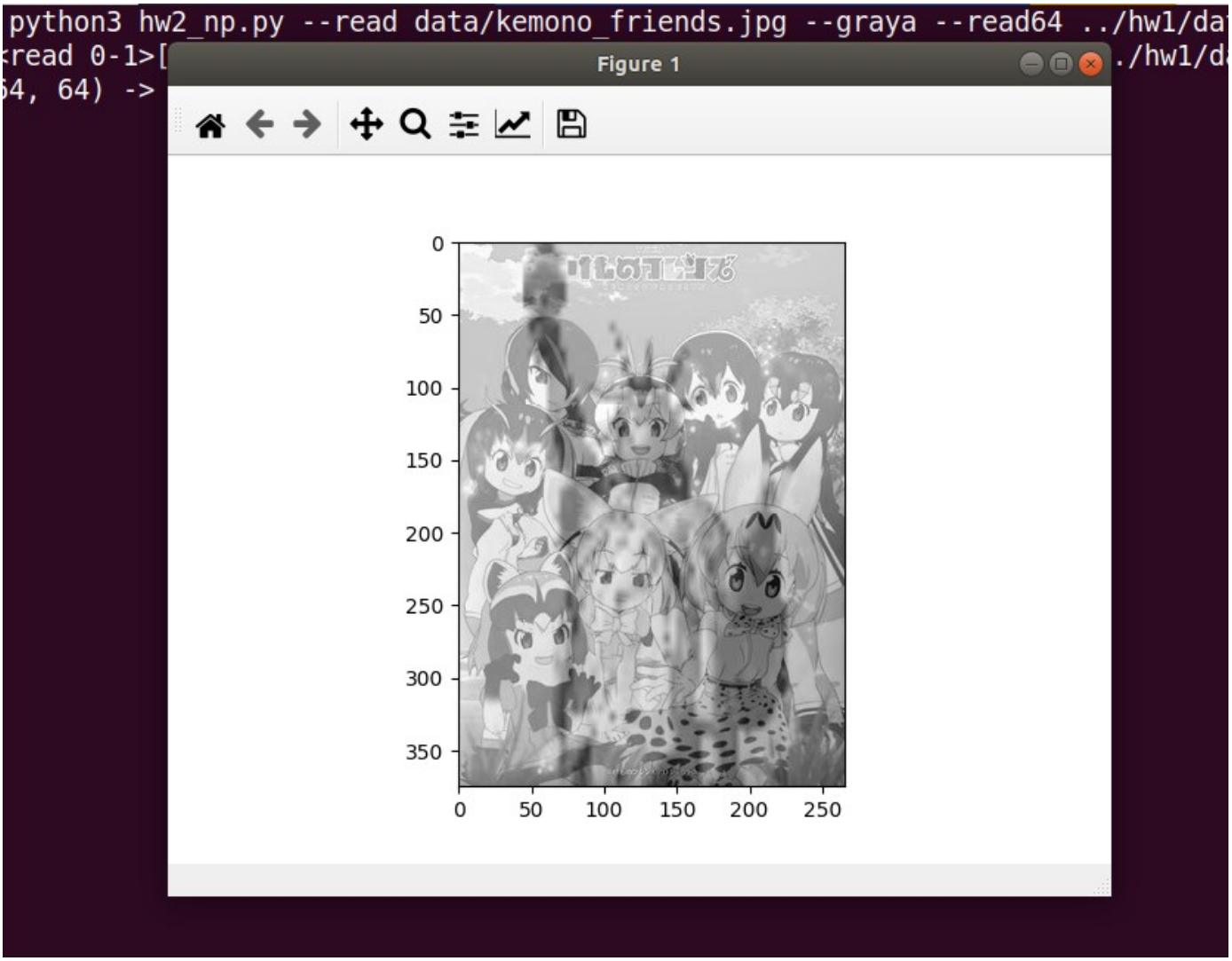
You can use command line as a postfix image processor.

For example `python3 hw2_np.py --read data/kemono_friends.jpg --show --graya --hist --showgray --equ --hist --showgray` will read image by `--read` and show it. After `--graya`, it convert the image to gray scale by method A, the histogram and image is shown by `--hist --showgrap`. Then, `--equ` to equalize the image, it show the histogram and the image of the result again.



Also, you can use module defined in hw1.

```
python3 hw2_np.py --read data/kemono_friends.jpg --graya --read64 ../hw1
/data/LIBERTY.64 --resize 375x266 --avg --showgray
```

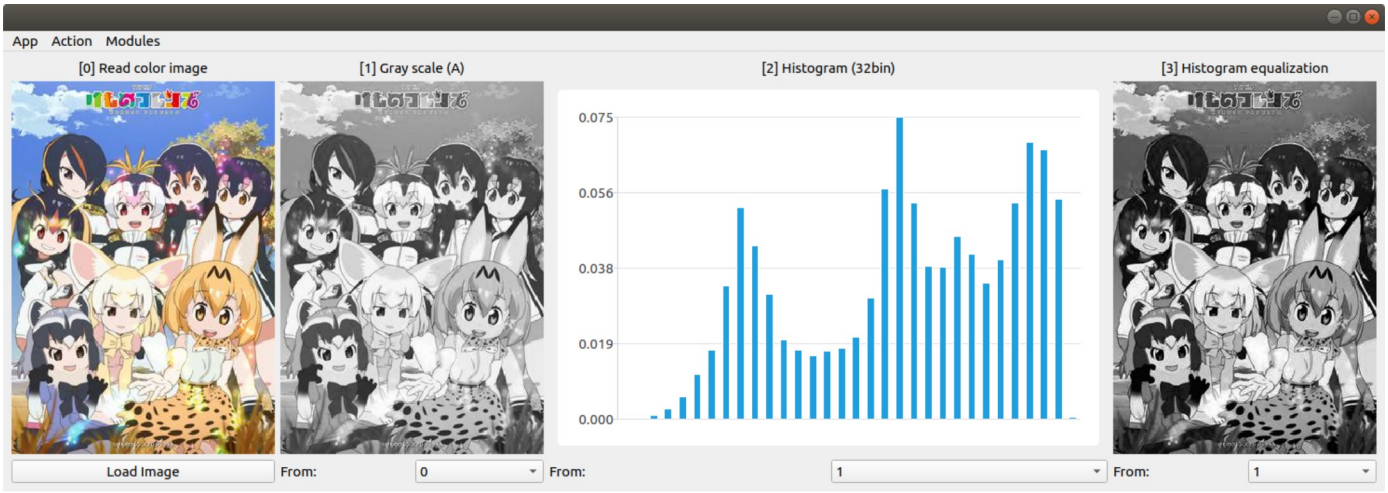


You can use `python3 hw2_np.py --help` to see more.

QT

`python3 qt.py` or `./run.sh`

The layout is more dynamic than hw1. You can add the module and delete module. What's more, all image in module are linked, the changed in parent widget will affect children widget. The source of image can be specific, all previous image can be it's parent.


App
Action
Modules
[0] Read color image
[1] Gray scale (A)
[2] Histogram (32bin)
[3] Histogram equalization
0.075
0.056
0.038
0.019
0.000
Load Image
From:
0
From:
1
From:
1