Assignment 2

Junyi Liao

June 30, 2023

1 Basic Global Thresholding

The basic global thresholding (BGT) algorithm can be based on the histogram of an image with more efficiency. We use the algorithm to find a global threshold T_{glb} for an image.

Notations $L:$ The color resolution of the image;
$P:$ An array of length L, which indicates the density histogram of the image. Here $P_i = \dfrac{n_i}{N},\ i = 0, \cdots, L-1$, n_i is the number of pixels at level i, and N is the total pixel number of the image;
$\omega_{[a,b]} = \sum_{i=a}^{b} P_i:$ The frequency of pixels in class $[a, b]$;
$\mu_{[a,b]} = \sum_{i=a}^{b} \dfrac{iP_i}{\omega_{[a,b]}}:$ The mean value of pixels in class $[a, b]$;
$\mu_{\text{glb}} = \sum_{i=0}^{L-1} iP_i:$ The (global) mean value of all pixels in the image;
$m_{[a,b]} = \sum_{i=a}^{b} iP_i = \omega_{[a,b]}\mu_{[a,b]}:$ The accumulated mean of class $[a, b]$ in terms of the total image.

Algorithm The algorithm takes in an image I as input and produces an threshold value T_{glb} as output. In the i-th loop, the algorithm compute the mean value of class $[0, T_i]$ and class $[T_i + 1, L-1]$, where T_i is the current threshold, then set $T_{i+1} = \dfrac{\mu_{[0,T_i]} + \mu_{[T_i+1,L-1]}}{2}$ as renewed threshold. To accelerate the computation,

$$T_{i+1} = \frac{\mu_{[0,T_i]} + \mu_{[T_i+1,L-1]}}{2} = \frac{1}{2}\left(\frac{m_{[0,T_i]}}{\omega_{[0,T_i]}} + \frac{m_{[T_i+1,L-1]}}{\omega_{[T_i+1,L-1]}}\right) = \frac{1}{2}\left(\frac{m_{T_i}}{\omega_{T_i}} + \frac{\mu_{\text{glb}} - m_{T_i}}{1 - \omega_{T_i}}\right);$$

We rearrange the renewal formula as: $T_{i+1} = \dfrac{m_{T_i} + (m_{L-1} - 2m_{T_i})\omega_{T_i}}{2\,\omega_{T_i}(1 - \omega_{T_i})}$, for brevity we notate $\omega_{[0,T_i]}$ as ω_{T_i}.

Pseudocode

```
BasicGlbThr($I$)
1  Compute the density histogram $P[0 : L-1]$ of input image $I$
2  Compute the frequency $\omega[0 : L-1]$ and $m[0 : L-1]$ on basis of $P$
3  Set an initial threshold estimate $T_0 = \lfloor m[L-1] \rfloor$
4  for $i = 0, 1, 2, \cdots$
5      $T_{i+1} = \lfloor (m[T_i] + (m[L-1] - 2 * m[T_i]) * \omega[T_i])/(2 * \omega[T_i] * (1 - \omega[T_i])) \rfloor$
6      if $T_{i+1} == T_i$
7          $T_{\text{glb}} = T_{i+1}$
8          break
9  return $T_{\text{glb}}$
```

In this algorithm based on histogram, since arrays $\{\omega_i\}_{i=0}^{L-1}$ and $\{m_i\}_{i=0}^{L-1}$ are prefix sums of $\{P_i\}_{i=0}^{L-1}$ and $\{iP_i\}_{i=0}^{L-1}$, they can be computed efficiently. Besides, all these arrays only need to be computed once. However they would be computed repeatedly in the loop if it was based on the image. Thus we accelerate the BGT algorithm based on histogram than on the image itself.

2 Locally Adaptive Thresholding

Algorithm We designed an algorithm of locally adaptive thresholding based on local OTSU, which selects the threshold value that maximize the criterion function

$$\eta = \frac{\delta^2_{\text{bet}}}{\delta^2_{\text{tot}}};$$

We inherit the notations from question **1**. For each pixel (x, y) in an image, consider a neighborhood S_{xy} of given size (e.g. $3 \times 3,\ 7 \times 7,\ \cdots$), the definitions of $\delta^2_{\text{bet}},\ \delta^2_{\text{tot}}$ are:

$$\mu_{\text{loc}} = \sum_{i=1}^{L-1} iP_i, \quad ,\omega_{\text{bk}} = \sum_{i=1}^{T_{xy}} P_i, \quad \mu_{\text{bk}} = \sum_{i=1}^{T_{xy}} iP_i, \quad \omega_{\text{fg}} = \sum_{i=T_{xy}+1}^{L-1} P_i, \quad \mu_{\text{fg}} = \sum_{i=T_{xy}+1}^{L-1} iP_i,$$

$$\delta^2_{\text{tot}} = \sum_{i=1}^{L-1} (i - \mu_{\text{loc}})^2 P_i, \quad \delta^2_{\text{bet}} = \omega_{\text{bk}}(\mu_{\text{bk}} - \mu_{\text{loc}})^2 + \omega_{\text{fg}}(\mu_{\text{fg}} - \mu_{\text{loc}})^2;$$

Since δ^2_{tot} is independent from the selected threshold T_{xy}, we only need to solve the optimization problem:

$$\max_{T_{xy}} \ \delta^2_{\text{bet}} \ \text{ s.t. } T_{xy} \in \{0, 1, \cdots, L-1\};$$

If the threshold T_{xy} doesn't make a real division, say all pixels belong to the same class, then $\delta^2_{\text{bet}} = 0$, else the computation can be accelerated by formula: $\delta^2_{\text{bet}} = \dfrac{(\mu_{\text{loc}}\omega_{\text{bk}} - m_{\text{bk}})^2}{\omega_{\text{bk}}(1 - \omega_{\text{bk}})},\ 0 < \omega_{\text{bk}} < 1;$
Apply this for all possible thresholds (to be specified, $0, 1, \cdots, L-1$), then choose the threshold that maximizes our objective function. If there exists multiple values that maximizes our objective function, use the mean of these values as the threshold.

Code The code is shown below. In implement we run this algorithm on the histogram of neighborhood at each pixel, which can be updated efficiently by moving along a zig-zag path pixel by pixel. The variable `loc_hist` is a `class` defined to efficiently compute and update the histogram of neighborhood.

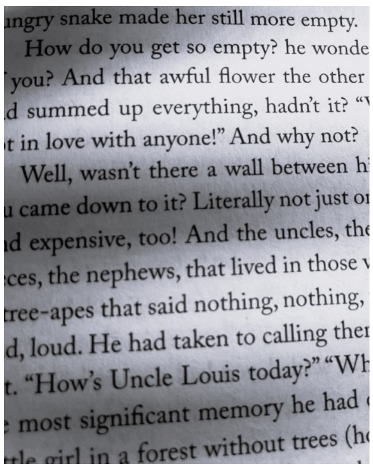
```python
def adapt_otsu(img, size, kernel=[0, 0], nbins=256):
    # img: the image need processing.
    # kernel: the pending pixel's coordinate.
    # size: yield an (2*size + 1)*(2*size + 1) neighborhood centered at kernel.
    loc_hist = local_histogram(img, size, kernel, nbins)
    img_bi = np.zeros_like(img)  # Initiate the binarized image.
    # Compute the threshold.
    while loc_hist.done == 0:   # Stop until all pixels are updated.
        cdf = np.cumsum(loc_hist.hist)  # Compute cumulated pixels.
        npixels = cdf[-1]  # number of pixels at the neighborhood.
        cdf = cdf / npixels  # Compute the cdf by normalizing to [0,1].
        acm = np.cumsum(np.array(range(nbins)) * loc_hist.hist / npixels)  # Compute the accumulated average from 0 to L-1.
        mean_glb = acm[-1]  # The global mean.
        w1w2 = cdf * (1 - cdf)
        w1w2[w1w2 <= 1e-6] = 1  # Preprocess data to avoid the expression below being divided by 0.
        dsq_bet = (mean_glb * cdf - acm)**2 / w1w2  # Compute between-classes variance divided by threshold from 0 to L-1.
        thr = np.argwhere(dsq_bet == np.max(dsq_bet)).mean()  # Determine the threshold.
        img_bi[tuple(loc_hist.kernel)] = (img[tuple(loc_hist.kernel)] >= thr)*(nbins-1)  # Thresholding.
        loc_hist.update()  # Move to next pixel.
    return img_bi
```

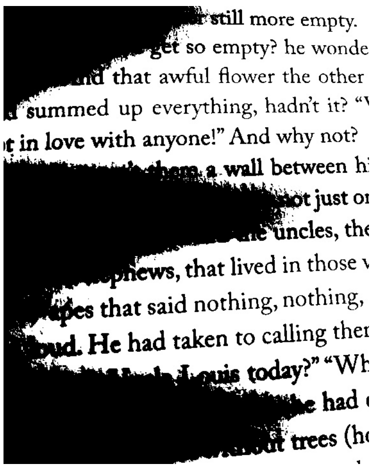
Code1. Adapt thresholding

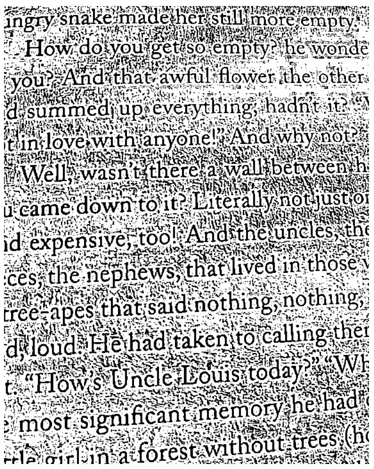
Consider computation error, we find that when the threshold level doesn't make division (ω_{bk} is 0 or 1), the formula will divide $(\mu_{\text{glb}}\omega_{\text{bk}} - m_{\text{bk}})^2$ by $\omega_{\text{bk}}(1 - \omega_{\text{bk}})$, which is zero. To avoid this error, set the divisor to 1 when it's 0. Because the dividend is also 0 in this case, the change doesn't impact the algorithm's correctness. For acceleration, the computing of between-classes variance from thereshold 0 to $L - 1$ is vectorized.

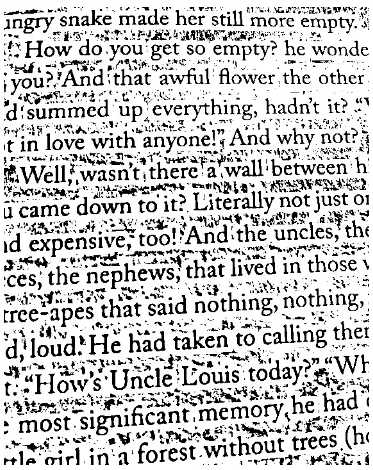
To compare the adapt thresholding with global thresholding, we implement both methods on the same image. We also set different size of neighborhood to find out how it influence the thresholded result. Results are shown below.

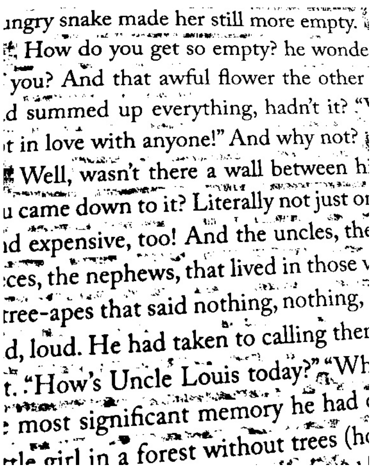
Figure. Results of global & adapt thresholding with Otsu algorithm

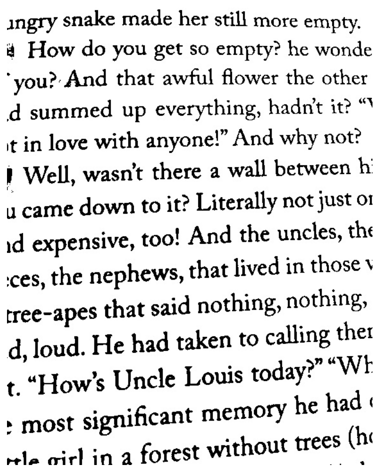

(a) far451.jpg (Original)


(b) After global Otsu thresholding


(c) After 5 × 5 thresholding (Otsu)

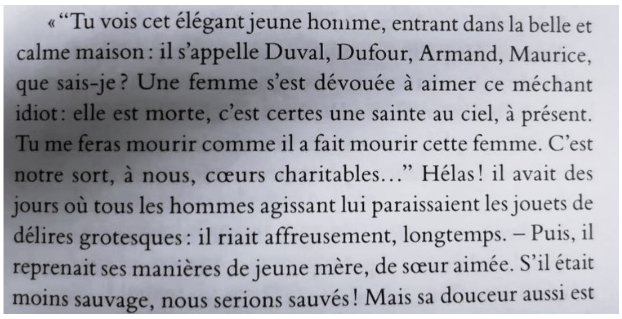

(d) After 15 × 15 thresholding (Otsu)


(e) After 25 × 25 thresholding (Otsu)

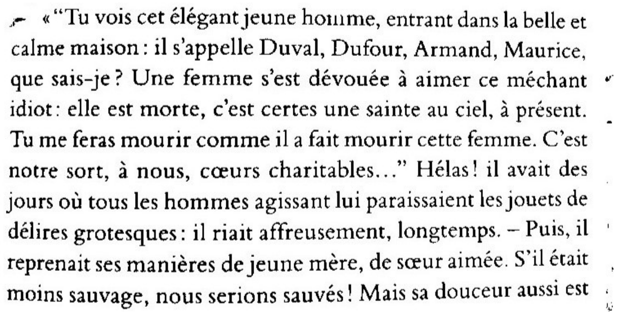

(f) After 45 × 45 thresholding (Otsu)

Analysis In the image `far451.jpg` (570 × 714), global Otsu selects a threshold which seperates the shadow from the book page, and text in the shadow is shielded after thresholding. However our goal is often to extract the text from an image, and this algorithm doesn't work.
For adapt thresholding, the threshold of a pixel only depends on its neighborhood, in that case we may extract text from a shadow. But this method is sensitive to noise, especially when the neighborhood is small. It exposes the noise in low-contrast region (for example, the blank in a page). We can reduce the impact of noise by expanding the neighborhood area, as is shown in our result. With a neighborhood of size 45 × 45, this method perfectly extracts the text in the image.
Following are some additional results yielded by adapt thresholding.

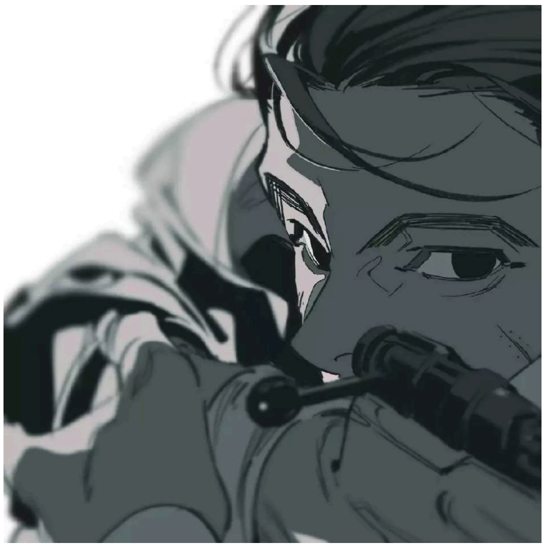
Figure. Additional Results of adapt thresholding with Otsu algorithm

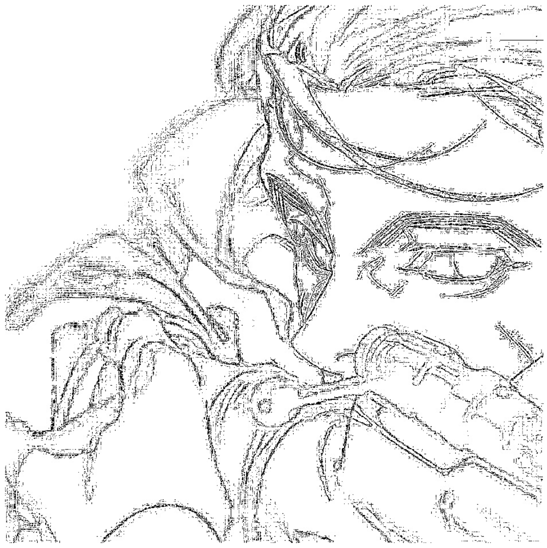

« “Tu vois cet élégant jeune homme, entrant dans la belle et calme maison : il s’appelle Duval, Dufour, Armand, Maurice, que sais-je ? Une femme s’est dévouée à aimer ce méchant idiot : elle est morte, c’est certes une sainte au ciel, à présent. Tu me feras mourir comme il a fait mourir cette femme. C’est notre sort, à nous, cœurs charitables…” Hélas ! il avait des jours où tous les hommes agissant lui paraissaient les jouets de délires grotesques : il riait affreusement, longtemps. – Puis, il reprenait ses manières de jeune mère, de sœur aimée. S’il était moins sauvage, nous serions sauvés ! Mais sa douceur aussi est

(g) rimbaud.jpg (Original)


« “Tu vois cet élégant jeune homme, entrant dans la belle et calme maison : il s’appelle Duval, Dufour, Armand, Maurice, que sais-je ? Une femme s’est dévouée à aimer ce méchant idiot : elle est morte, c’est certes une sainte au ciel, à présent. Tu me feras mourir comme il a fait mourir cette femme. C’est notre sort, à nous, cœurs charitables…” Hélas ! il avait des jours où tous les hommes agissant lui paraissaient les jouets de délires grotesques : il riait affreusement, longtemps. – Puis, il reprenait ses manières de jeune mère, de sœur aimée. S’il était moins sauvage, nous serions sauvés ! Mais sa douceur aussi est

(h) rimbaud.jpg after 45 × 45 Otsu thresholding


(i) avatar.jpg (Original)


(j) avatar.jpg after 3 × 3 Otsu thresholding

3 Linear Interpolation

Algorithm We implemented a linear interpolation algorithm to enhance the spactial resolution of an image. The algorithm refers to the upsampling method, and estimate the value of unknown (or inserted) pixels by linear interpolation. Suppose we enhance the resolution of an $h \times w$ sized image by N times, then we need to rebuild an image of size $[N(h-1)+1] \times [N(w-1)+1]$.
The coordinates of original sampled points in rebuilt image are $\{(Nx, Ny) \,|\, (x, y) \in \{0, 1, \cdots, h-1\} \times \{0, 1, \cdots, w-1\}\}$. For the inserted points, the pixel value is estimated by:

$$I(Nx+r, Ny+s) = w_{00}I_0(x, y) + w_{01}I_0(x+1, y) + w_{10}I_0(x, y+1) + w_{11}I_0(x+1, y+1)$$

$$w_{00} = (N-r)(N-s),\ w_{01} = r(N-s),\ w_{10} = (N-r)s,\ w_{11} = rs,$$

here $(x, y) \in \{0, 1, \cdots, h-2\} \times \{0, 1, \cdots, w-2\}$, and each (x, y) corresponds to an $N \times N$ block in the rebuilt image, where $(r, s) \in \{0, 1, \cdots, N-1\}^2$, and the value of each pixel in the block is estimated by the nearest 4 known pixels, say (x, y), $(x+1, y)$, $(x, y+1)$, $(x+1, y+1)$ in the original image.

Code The code is shown below.

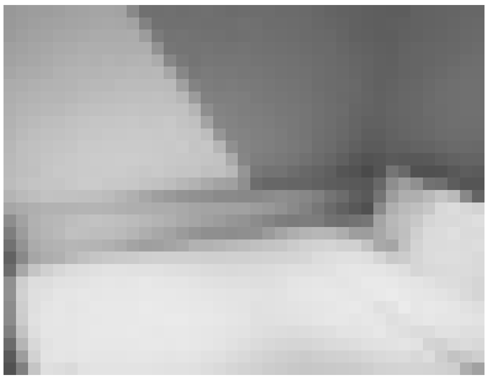
```python
def lin_interp(img, t):
    h, w = img.shape
    img_interp = np.zeros(((h - 1) * t + 1, (w - 1) * t + 1))  # Initiate the interpolated image.
    weight = [i / t for i in range(t)]  # Initiate the weight of interpolation.
    r = np.array([weight * (h - 1)])  # The weight of each row in the image.
    s = np.array([weight * (w - 1)])  # The weight of each column in the image.
    block = np.ones((t, t))  # A block in the image matrix.
    lblock = np.ones((1, t))  # A line block in the edge of the image matrix.

    # Process pixels in the image.
    # The weight of the nearest pixels:
    # upper-left: (1-r)*(1-s), upper-right: r*(1-s), lower-left: (1-r)*s, lower-right: r*s
    img_interp[:(h-1) * t, :(w-1) * t] = (
        np.kron(img[:h-1, :w-1], block) * ((1 - r).T * (1 - s))
        + np.kron(img[1:, :w-1], block) * (r.T * (1 - s))
        + np.kron(img[:h-1, 1:], block) * ((1 - r).T * s)
        + np.kron(img[1:, 1:], block) * (r.T * s)
    )
    # Supplement pixels in the edge of the image.
    img_interp[:(h-1) * t, (w-1) * t] = np.kron(img[:h-1, w-1], lblock) * (1 - r) + np.kron(img[1:, w-1], lblock) * r
    img_interp[(h-1) * t, :(w-1) * t] = np.kron(img[h-1, :w-1], lblock) * (1 - s) + np.kron(img[h-1, 1:], lblock) * s
    img_interp[(h-1) * t, (w-1) * t] = img[h-1, w-1]

    '''
    # Process by loop. This method is the same as the above in effect but runs slower.
    for x in range(h-1):
        r = np.array([[i/t for i in range(t)]])
        for y in range(w-1):
            img_interp[t*x:t*(x+1), t*y:t*(y+1)] = img[x, y]*(1-r).T*(1-r) + img[x+1, y]*r.T*(1-r) + img[x, y+1]*(1-r).T*r + img[x+1, y+1]*r.T*r
            img_interp[t*x:t*(x+1), t*(w-1)] = img[x, w-1]*(1-r) + img[x+1, w-1]*r

    for y in range(w-1):
        s = np.array([[j/t for j in range(t)]])
        img_interp[t*(h-1), t*y:t*(y+1)] = img[h-1, y]*(1-s) + img[h-1, y+1]*s

    img_interp[t*(h-1), t*(w-1)] = img[h-1, w-1]'''

    return img_interp
```

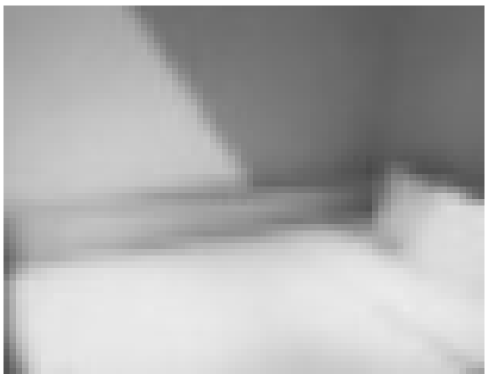
Code2. Linear interpolation

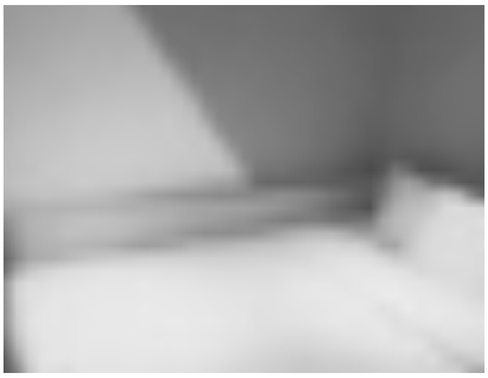
For computation efficiency, the interpolation is executed on a matrix initialized with expected height and width after processing. For each pixel, we derive the weight of four nearest pixels and store them in a matrix, in the code, `(1-r).T*(1-s)`, `r.T*(1-s)`, `(1-r).T*s`, `r.T*s`. Coordinates of nearest pixels can also be stored in matrices, in the code, `img[:(h-1)*t, :(w-1)*t]` (upper-left), `img[1:, :(w-1)*t]` (upper-right), `img[:(h-1)*t, 1:]` (lower-left), `img[1:, 1:]` (lower-right). To resize these matrices, use their Kronecker products ($\otimes$) with `np.ones((t,t))`. Then pixel values in processed image can be computed by elementwise operation on matrices. We need to supplement the right and lower edges of the image after matrix operation. The code based on loop structure is given as annotations, which is the same as matrix operation in effect, but with less efficiency.

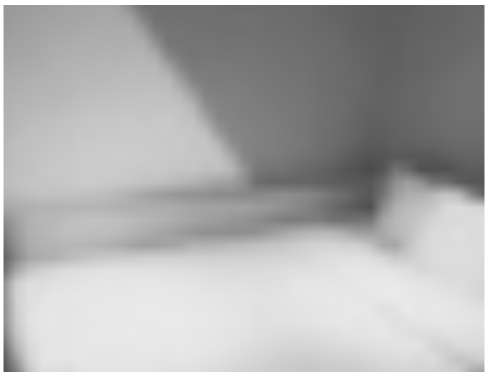
We test the code on several images with different magnification factor. Our results are shown below.

Figure. Images before & after linear interpolation


(k) hotel.jpg (Original)

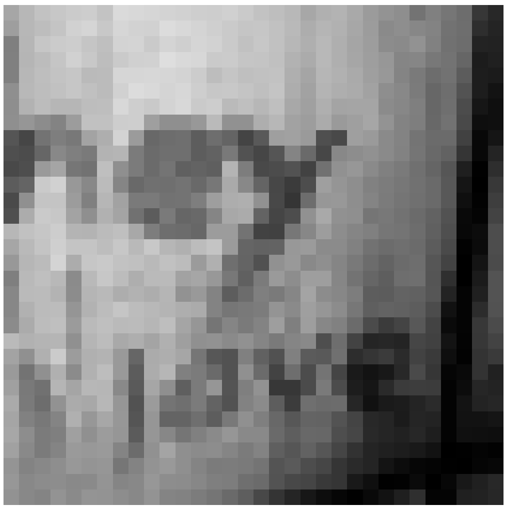

(l) After 2× interpolation

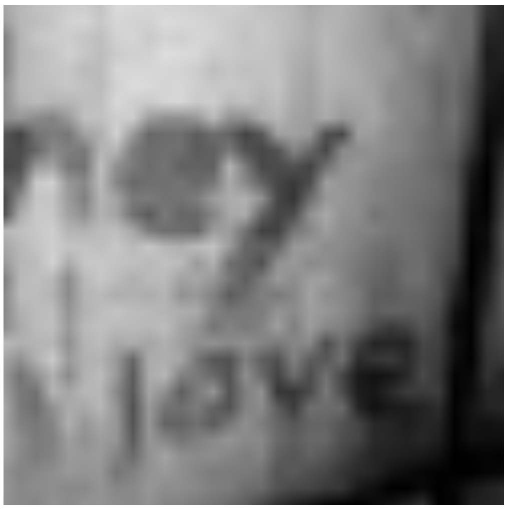

(m) After 8× interpolation

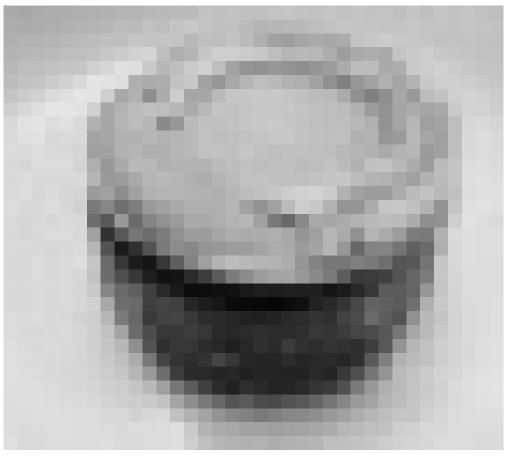

(n) After 32× linear interpolation

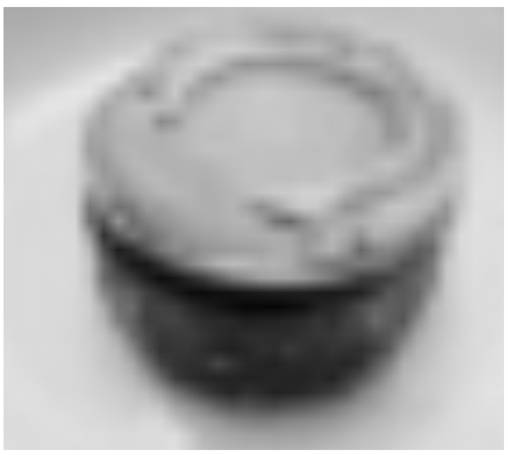
Analysis As the magnification factor increase, the spatial resolution of processed image also increase. However the edges of objects are not sharp, and the magnified images are somewhat blurred because the pixel values in the image are not necessarily linearly distributed. (For example at an edge, the pixel value rises or falls dramatically, say the gradient is large, and this detail can't be restored by linear interpolation.) Thus the linear interpolation is only a coarse estimation of unsampled points. Meanwhile, a low-resolution image has already lost much information of the original scene, thus the original scene can't be fully restored.
Some additional results are shown in next page.

Figure. Images before & after linear interpolation


(o) bathroom.jpg (Original)


(p) After 16× interpolation


(q) bathroom.jpg (Original)


(r) After 16× interpolation

4 Appendix

4.1 Libraries

We use the `numpy` library to support the large-scale computation of matrices which store the grey level information of images.
We use `opencv` library (`cv2`) to read the input images and save the output images.
We use `matplotlib` to show images before and after processing.
We also use `argparse` to read in the arguments for processing, which can be specified in the command line.

4.2 IDE

All our experiments are done on PyCharm Community Edition 2022.1.2.