Dictionary Method for Micrograph Image Analysis

Haoran Shu
The Chinese University of Hong Kong
henryshu1994@gmail.com

Hang Cheung
The Chinese University of Hong Kong
henryc9473@gmail.com

August 7, 2016

1 Introduction

Microscopy methods are now universally used in various scientific fields and the analysis of tons of thousands of micrographs is at the core of modern scientific laboratory analysis. The recent development of computer vision can facilitate many of these analyses largely so as to reduce human workload and encourage new discoveries based on the ability of efficient mass processing.

Dictionary method requires to categorize multiple atoms in one micrograph into groups according to a given (or learned) dictionary so that they correspond to items in the dictionary. In our problem, a dictionary of 125 items is given (see Figure 1 for part of them). Also given is a micrograph consisting of multiple atoms. Each atom in the micrograph corresponds to one and only one mode in the dictionary (a dictionary item). Our task is to identify these correspondences with an efficient and accurate algorithm.

An illustration of the problem setting could be found below:

We divide this task into several steps: scale-invariant locating, normalization and comparison. We first locate the individual atoms within the micrograph, then normalize possible brightness or contrast disparities between the micrograph and the dictionary, and finally compare them so as to give the best match.

In the rest of this literature, we will use 'dictionary' to refer the whole dictionary of modes, 'micrograph' the whole input image, 'dictionary items' of individual modes/items in the dictionary, and 'micrograph units' the individual atoms in the micrograph.

In part 2, we mainly explain the details of our algorithm; in part 3, the performance is displayed and part 4 focuses on possible future work directions we are

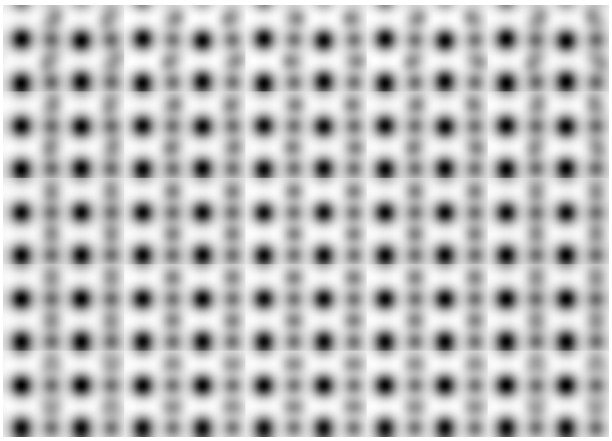


Figure 1: A sample of the given dictionary (part)

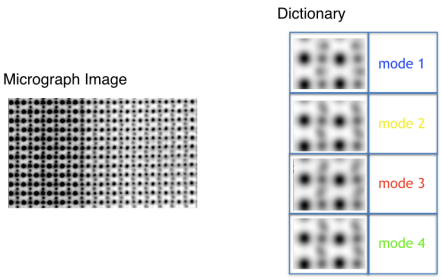


Figure 2: An illustration of the problem setting

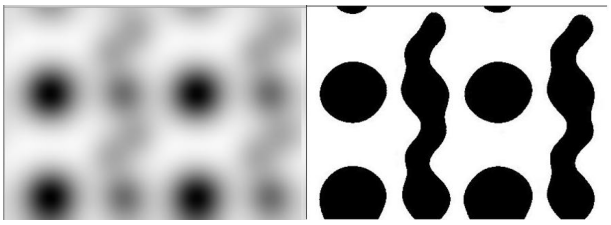


Figure 3: The left image is before the transformation while the right one is after the transformation

aware of.

2 Algorithm

2.1 Scale-Invariant Template Matching of Unit

Motivation. We are given a micrograph as our input data. Each micrograph consists of multiple units, thus our first task is to locate each unit and we propose to use a Template Matching approach. Template Matching is a technique in digital image processing for finding small parts of an image which match a template image.

From the samples given, it is possible that the dictionary item and micrograph are of different scale. However, it is unlikely that the micrograph is rotated or sheared, making our problem a Scale Invariant Template Matching.

Workflow. Given the characteristics of these micrographs, we assume periodicity and even distribution of units in the micrograph, we conquer this problem as follows:
First we will find out explicity one unit with a naive scale approximation and a brute force search; then with this located unit, we use the periodicity assumption to locate the rest of them. Here come the details :

Eating from the outside.. To find out the first unit as a starter, we will begin with a process called 'Eating From The Outside' to get an approximated scale. We normalize the micrograph according to the dictionary item, and do a simple transformation to turn the image into black and white. In grayscale images each pixel is represented as an intensity between 0 and 255, 0 for black and 255 for white. For both images, we calculate the average pixel intensity for each, and make that pixel 255 if its intensity is above its average, 0 otherwise. This will turn the atoms in the image black, kind of a naive segmentation (See Figure 3). After that we start 'eating'. The idea is that for every pixel, if any of its nearest 8 pixels is black, we also turn it black. Doing this for many rounds we can count the rounds needed for both the images to be all black (or the sum of the intensity is smaller than a tolerant constant). The ratios of these counts can serve as an estimate of the scale since the micrograph and the dictionary item are similar.

Brute Force Search. After getting the approximate scale, we perturb the scale obtained back and forth and scan in the micrograph with a sliding window sized the same as the dictionary item, from the upper left corner to the lower right corner. This enables us to match for a best scale. For example, if we get $rounds\ of\ dictionary\ item : rounds\ of\ image = 2:1$, we may try sliding window of size $\frac{1}{2}, \frac{1}{2.1}, \frac{1}{2.2}, \frac{1}{1.9}$ or $\frac{1}{1.8}$ of the size of the dictionary item. For each tested scale, choose the closest one to the dictionary. Note that we don't have to get an exact match for the dictionary item and the micrograph. Instead, we just need the right intensity distribution (*i.e.* atoms matched to atoms), so any arbitary dictionary item will work for us.

Similarity Measure. To measure how good/simiar a matching is, we use the normalized correlation as our similarity metric. The formula of the normalized correlation of dictionary item and the sliding window of the same size is

$$\frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}} \tag{1}$$

where x is the dictionary item, y is the sliding window, $\bar{x}, \bar{y}$ are the intensity averages, and i sums across all the pixels.

As shown in the formula, the computation cost of normalized correlation is large. Let N^2 be the size of the dictionary item (also the size of the sliding window), M^2 be the size of the microscopy. For the calculation of the normalized correlation, it requires $O(N^2)$ amount of times, but because of the sliding window approach, we have to perform the calculation $(M - N + 1)^2$ times, so the total cost will be $O(N^2(M - N + 1)^2)$. Therefore, to boost up performance, several techniques are used. Some will be discussed below and some will be discussed in FUTURE WORK.

Optimization. For the matching process, we will use the upper left quarter as our microgrpah image, instead of the whole image. Because of the periodicity of the micrograph, the upper left quarter will be similar to the whole image, and also our first goal is just to locate one unit, no matter where it is.

Besides, we employ the method of Image Pyramid. In brief, Image Pyramid is about building several levels of the same image in the following manner : Start with level 1, for every level N, Level N+1 is the same as Level N but with half the sizes. Keeps building until some condition satisfied. (See Figure 4). We keep building levels until one of the dimension of the dictionary item is less than 50. Then we do the matching process on the highest level of the pyramid. Say we get the rectangle $[x_1, x_2] \times [y_1, y_2]$ as our initial matching result, we project it down for one level and get $[x_1', x_2'] \times [y_1', y_2']$; then we do a matching among $[x_1' - \epsilon, x_2' + \epsilon] \times [y_1' - \epsilon, y_2' + \epsilon]$ to modify our result. Here ϵ is a small positive integer, and we choose $\epsilon = \lfloor \frac{1}{20} * max(width\ of\ level,\ length\ of\ level) \rfloor$. Similarly, we run all the way down to level 1 and use the final refinement result as our matching result.

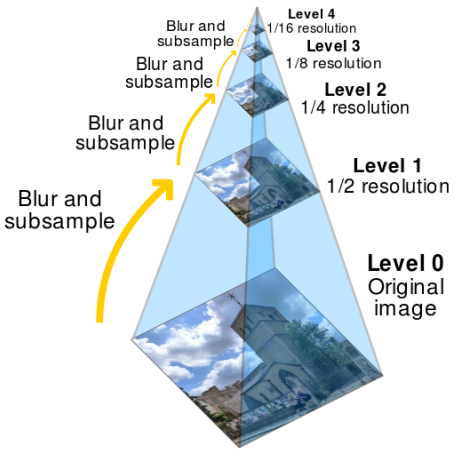


Figure 4: The idea of Image Pyramid

Also, note that in equation (1), $\bar{x}$ and $\sqrt{\sum_i (x_i - \bar{x})^2}$ will never be changed during the sliding of the window. Therefore these two can be pre-calculated to save computational time.

The usage of integral image to save computational time will be discussed in the FUTURE WORK.

Locating the other units. Say now we have got $[a_1, a_2] \times [b_1, b_2]$ as our matching result of the first unit, by the periodicity assumption, we get the next unit to the right by doing a template matching in the rectangle $[a_2 - \epsilon, a_2 + (a_2 - a_1) + \epsilon] \times [b_1 - \epsilon, b_2 + \epsilon]$. Similarly, we repeat for the left, upper, lower units, until we locate all the units that are intact in the micrograph.

Disadvantage of Eating from the outside. This approach is too naive that we will abandon this once we have done the integral image approach in the FUTURE WORK.

2.2 Normalization

Motivation. The given micrograph images, most of the time, are produced in different types of experiments and under various lab settings, which leads to differences in brightness and contrast even within one micrograph image (see Figure 5). There are some operations and measures that are immune to these differences with reasons to be explained in a minute, but some are not, for example, the Histogram of Oriented Gradients (to be explained later). Therefore, it is necessary that we normalise the micrograph image according to the dictionary.

Mathematical Interpretation. For the grayscale images that we are studying, colors are represented by integer 'intensities'. The differences in brightness and contrast of two grayscale images are essentially the differences in their intensity distributions. Thus, what we need to do is to normalize the intensity

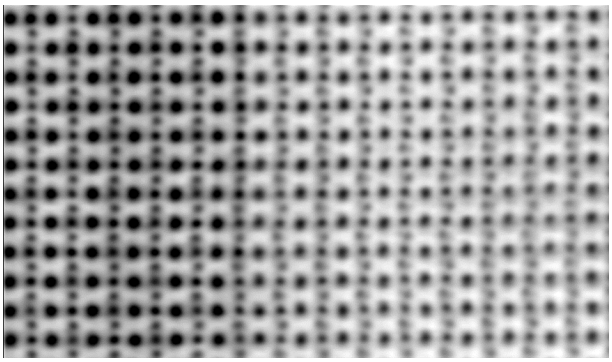


Figure 5: A Micrograph with Uneven Brightness and Contrast

distribution of the micrograph image to that of the dictionary item.

A typical way to do this is to adjust the intensity distribution of the micrograph image so that it has the same average and standard deviation with that of the dictionary item.

For an image of size $m \times n$ pixels, denote by y_i for $i = 1, 2, ..., mn$ the intensity of the micrograph image at the ith pixel ($0 \leq y_i \leq 255$). Let $\mathcal{D}(\mu_m, \sigma_m)$ and $\mathcal{D}(\mu_d, \sigma_d)$ (μ for average and σ for standard deviation) denote the intensity distributions of the micrograph and the dictionary item, respectively and $\mathcal{D}(\mu'_m, \sigma'_m)$ that of the adjusted micrograph image. (Note: we are using an arbitrary dictionary item from the dictionary here because they have the same information about brightness and contrast in their intensity distributions).

Then what we have from a distribution-wise normalization is

$$y'_i = \frac{y_i - \mu_m}{\sigma_m} \times \sigma_d + \mu_d \tag{2}$$

so that $\mu'_m = \mu_d$ and $\sigma'_m = \sigma_d$.

It should be noted that if we want to apply operations like cross correlation on the images, this normalization step can be dropped because such operation already contains normalization. However, operations like HOG (Histogram of Oriented Histograms) and MI (Mutual Information) cannot be applied directly.

Implementation. To implement this normalization, we calculate the adjusted intensity of the micrograph pixel by pixel and assign it to the original image. We also use the $floor$ function in MATLAB to round each calculated intensity to integer type.

Performance. In general, our algorithm functions well for this task, mainly because the dictionary items serve as a good reference. Figure 6 shows the result of pixel-wise intensity-normalization for the micrograph in Figure 5.

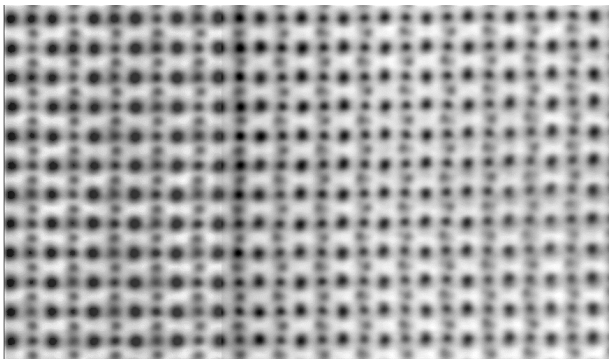


Figure 6: Normalization result for Figure 5

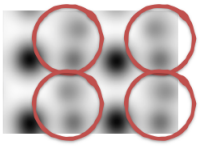


Figure 7: Interesting Parts

2.3 Comparison

Masking. When it comes to comparison, it is important that we figure out which part of the micrograph is of principal interest. In this specific problem, the material scientists are particularly interested in the periphery areas of the atoms, where reside most of the differences among atoms of different modes. The core parts should thus be discarded during comparison because differences there, if any, are of no relevance. The following mask (Figure 8.) is applied for Figure 7.

Feature Extraction. Even after masking irrelevant parts, the differences seem to be rather elusive. It is important to grasp key features in the images: using wholistic comparison methods like MI (Mutual Information) or normalised CC (Cross Correlation) may fail to grasp local information while using direct pixel-wise comparison may be too sensitive to tiny rigid shifts (to be illustrated in Figure 9.) in the images. Therefore, we use a technique called HOG (Histogram of Oriented Gradients) to extract local features of the concerned images.

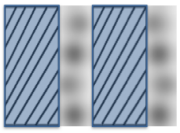


Figure 8: Mask

Figure 9: The gray part and the black part are actually the same thing, but a tiny rigid movement/shift makes it confusing for the computer to realize their high similarity

HOG essentially looks statistically at the changes in intensity within small patches of the image. It is thus immune to tiny rigid shifts and comprehensive in grasping local information. The details of HOG and illustrations can be found below:
1). Divides the input image into square cells of size $cellSize$;
2). Calculates the image gradient using central difference at each pixel;
3). Assign the gradient at each pixel to a histogram of $2 \times numOrientations$ (set by user) orientations to obtain a directed orientation histogram h_d and a histogram of undirected orientations h_u by folding h_d into two;
4). Calculate a feature vector with 36 features according to UOCTTI, and then reduce it to a 31-dimension feature vector.

Using HOG makes both the micrograph units and dictionary items much more differentiable: the normalized cross correlation of two distance dictionary items can be as high as 0.9928 (after masking). It is now reduced to 0.9262.

Similarity Measure. Now we can apply normalized CC (cross correlation) on the feature vectors we got.

$$Cross\ Correlation = \frac{\sum_{i=1}^{i=mn}(x_i - \bar{x})(y_i - \bar{y})}{mn \times \sigma_x \sigma_y} \tag{3}$$

For each micrograph image unit, calculate its cross correlation with each dictionary item and pick the item with the highest normalized CC as the identified item.

However, since the difference between micrographs of different atom modes are extremely elusive, using normalized CC directly on the feature vector might still suffer from ambiguity. Noticing that in this case, the interesting part can actually be subdivided into four parts, with each having some individual characteristics in direction and shape (see Figure 8.), we can apply the HOG-CC paradigm on each of these four parts and then combine the results to get a new similarity measure. Let's denote the new distance by $Dist$. ($distance = 1 - similarity$)

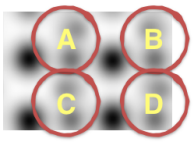


Figure 10: A, B, C, D are separated for individual comparisons

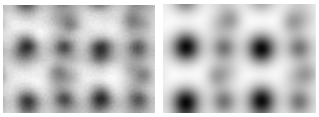


Figure 11: A micrograph unit and a very similar dictionary item (not corresponding)

$$Dist = 100 \times (1 - CC_A)(1 - CC_B)(1 - CC_C)(1 - CC_D) \\ (1 - CC_A \times CC_B \times CC_C \times CC_D) \tag{4}$$

This newly defined similarity measure largely enlarged the gap between similar and dissimilar images. For example, the normalized CC between feature vectors of one micrograph unit and an arbitrary dictionary item in Figure 11 is 0.6057 while that between the same micrograph unit and its corresponding dictionary item in Figure 12 is 0.6378, showing a really small gap in the distances. After using $Dist$, assuming each part of A, B, C, D has the same feature vector normalized CC (that is, 0.6057 and 0.6378), we have

$$Dist_{different\ modes} = 2.09182956 \tag{5}$$

$$Dist_{same\ mode} = 1.43625651296 \tag{6}$$

This is a lot more robust compared to the 0.3622 and 0.3943 distances just using normalised CC once.

2.4 Another Approach: Grouping

Motivation. After testing our locate-normalise-compare pipeline, we still find outliers and mistakes in the output we produce here or there. This mainly results from the intrinsic similarity among dictionary items. Actually, when the noises have even larger effect than different modes of atoms on the micrographs, there is hardly any algorithm that can produce 100% correct answers.

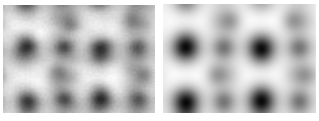


Figure 12: The same micrograph unit and its corresponding dictionary item

Nevertheless, this problem needs a really low tolerence rate on mistakes because of the scientific rigorousness required in subsequent studies based on our output. Thus, we think of compromising our goal to outputing a weaker result: instead of giving one single match, we propose to give a candidate group that is as small as possible and we wish to maintain a 100% confidence rate.

K-means Clustering. Among various clustering methods, we picked K-means algorithm because it is the easiest to implement and it turned out working just fine for our problem. We believe algorithms like Spectral Clustering and Max-Spacing will produce similar results as well. The above *Dist* measure is used as a pseudo-distance in our K-means clustering and the details of our K-means methods can be found below:
1). Randomly initialize K centroids at K input data points (in our case, dictionary items);
2). Assign each data point (dictionary item) to its nearest centroid;
3). Recalculate centroids to be the mean of all its assigned data points;
4). Repeat 2) and 3) until converge;

Sampling. K-means Clustering is a randomized algorithm, so we cannot get deterministic result for grouping. To make our grouping more stable, we use the technique of sampling :
We repeat K-means Clustering for K times, assumes the size of the dictionary is S.
Define

$$X_{ij}^{(n)} \begin{cases} 1, & \text{if dictionary item i,j are in the same group} \\ & \text{in the n-th experiment} \\ 0, & \text{otherwise} \end{cases}$$

where $1 \leq i, j \leq S$, $1 \leq n \leq K$.
Also define $X^{(n)}$ as a $S \times S$ matrix with $X^{(n)}(i,j) = X_{ij}^{(n)}$.
Therefore we get a sequence of matrix $X^{(n)}, 1 \leq n \leq K$.
Let

$$X = \frac{1}{S} \sum_{n=1}^{S} X^{(n)}$$

Then $X(i,j)$ is the percentage that *item* i and j are in the same group for these K experiment. We group *item* i and j into the same group if $X(i,j) \geq$ *threshold constant*, we choose *threshold constant* to be 0.8 in our case.

For the grouping result, please see APPENDIX.

Comparison. Now that we have divided the dictionary into K groups, we proceed to generate a signature for each group. The signature of a group is nothing special than an average of all its member items. Then it is these signatures that are used to be compared with the micrographs. With the same methods discussed in last section, we will pick the optimal group as the identified group and all its member items the identified candadites.

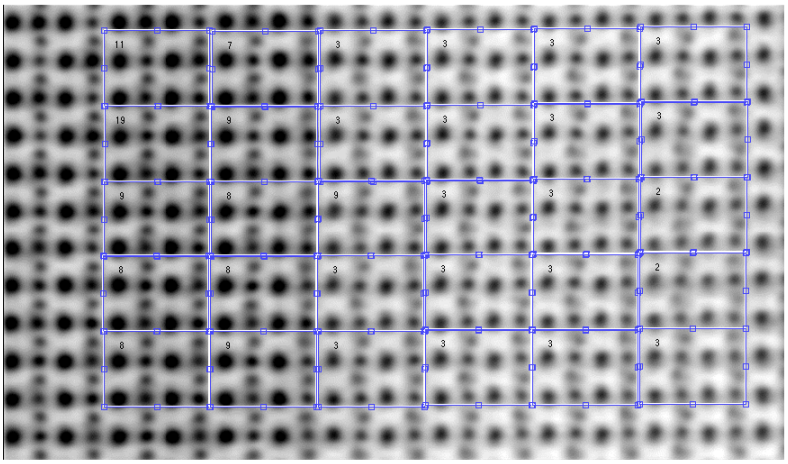


Figure 13: The output of original algorithm

3 Performance

3.1 Output of Algorithm

Figure 13 shows the output of our algorithm. Each rectangular stands for a unit we located in the input micrograph and the small number at their upper-left corners stand for the ID of dictionary item they are identified with.

On average our algorithm takes 110s to run on a MacBook Air (1.8 GHz Intel Core i5, 4 GB 1600 MHz DDR3) for a dictionary of size 25 and a microscopy of size 30 (items).

Our algorithms performs very well in locating the units even when there are large differences in the scale of the dictionary item and micrograph units. It also successfully captured the characteristics of the input micrograph which is that units in the same column belong to the same or similar dictionary items. However, it also made mistakes. The middle unit of the third column, for example, is identified with the 9th dictionary item, which is obviously wrong. We have tuned the cellSize we use in HOG extraction but that helps little. We proceed to try exchanging the order of normalization and masking.

3.2 After Adjusting Order of Normalization and Masking

As suggested before, we adjusted the order of normalization and masking because when normalizing we are assigning the brightness information of one dictionary item to any micrograph unit, including those not corresponding to it. This might introduce changes in other information of these micrograph units but they were thought to be negligible.

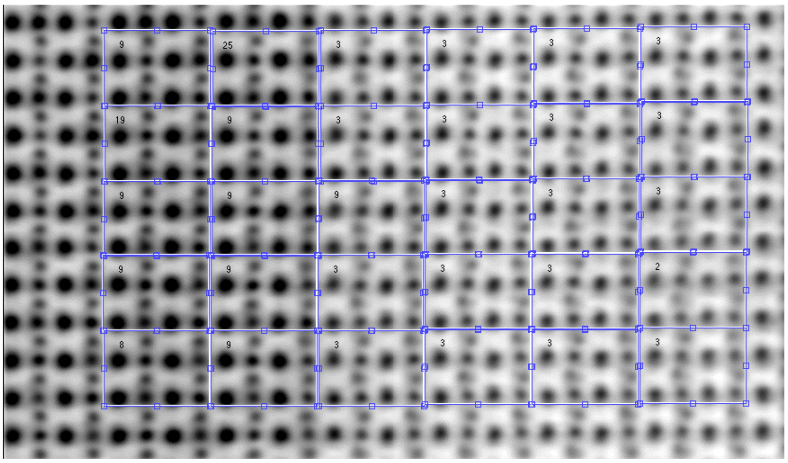


Figure 14: The output after exchanging the order of normalization and masking

Figure 14 shows the result after adjusting the order and it does not seem to fix our problem.

3.3 Output of Grouping Method

As mistakes, even tiny ones, cannot be tolerated in our task, we turn to think about compromising our goal to outputting a list of candidates instead of one final answer for each micrograph unit. We call it the Grouping Methods.

Figure 15 shows the result of the Grouping Methods (where the average of each group is used as signature). This time we do fix the outlier problem but we also introduced new problem: How can we determine the K in grouping? In principal, we want as many groups as possible so that each 'candidate list' obtained would be smaller, but how do we now to which point we can enlarge K? Especially when we don't have prior 'right answer' for the matching.

4 Future Work

4.1 Speeding Up the Matching by Integral Image

Recall (1),

$$\frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}$$

where x is the dictionary item, y is the sliding window, $\bar{x}, \bar{y}$ are the intensity averages, and i sums across all the pixels. x, $\bar{x}$ can be pre-calculated. Actually,

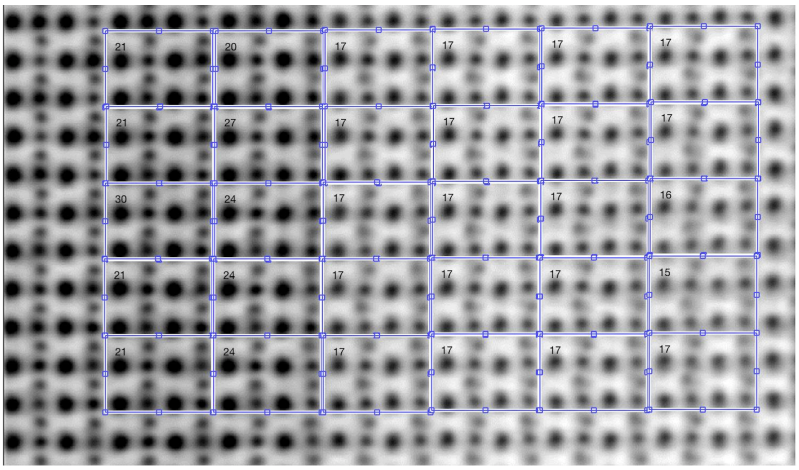


Figure 15: The output of Grouping Method

$\sqrt{\sum_i (y_i - \bar{y})^2}$ can be obtain by $O(1)$ time after we have pre-calculated the image's Integral Image.

Denote the intensity of microgrpah at (x, y) by $i(x, y)$, and define

$$I(x, y) = \sum_{a=1}^{x} \sum_{b=1}^{y} i(a, b)$$

We call I the integral image of i. Note that

$$\sum_{x=a}^{a'} \sum_{y=b}^{b'} i(x, y) = I(a', b') - I(a', b-1) - I(a-1, b') + I(a, b)$$

Therefore calculating $\sum_{x=a}^{a'} \sum_{y=b}^{b'} i(x, y)$ takes $O(1)$ time. Note that in $\sqrt{\sum_i (y_i - \bar{y})^2}$,

$$\sum_i (y_i - \bar{y})^2 = \sum_i y_i^2 - 2\bar{y} \sum_i y_i + N^2 \bar{y}$$

We obtain the integral image of y_i^2 and y_i then we can get $\sqrt{\sum_i (y_i - \bar{y})^2}$ in $O(1)$ time.

And for the calculation of the numerator, that is

$$\sum_i (x_i - \bar{x})(y_i - \bar{y})$$

we have

$$\sum_i (x_i - \bar{x})(y_i - \bar{y}) = \sum_i x_i (y_i - \bar{y}) - \bar{x} \sum_i (y_i - \bar{y})$$

Note that the latter term equals to zero. Therefore we have only the term $\sum_i x_i(y_i - \bar{y})$ left. Since we are locating the units, we don't have to have an exact match of the units. It suffices to consider the approximation of the normalized cross correlation. So we may do the approximation of the normalized cross correlation as follows : Define R_i as a rectangular block, *i.e.*, $R_i = 1$ *on some rectangular region*, $R_i = 0$ *otherwise*.
If we can approximate the Image of $y_i - \bar{y}$ by rectangular blocks, *i.e*, $Image_{y_i - \bar{y}} = \sum_{rectangular\ blocks} k_j R_j$, where k_j is some nonzero constant, then

$$\sum_i x_i(y_i - \bar{y}) = \sum_j k_j \sum_{nonzero\ region\ of\ R_j} x_i$$

then we can use the technique of Integral Image. For the rectangular blocks approximation of the $Image_{y_i - \bar{y}}$, one may uses Gradient Descent approach to get such approximation.

For details of the above method, please refer to [5], [8].

A Remark is that with the above algorithm, the calculation of normalized cross correlation will be reduced to $O(1)$ time, so the total cost of the template matching will be $O(M - N + 1)^2)$, provided we have pre-calculated the integral images of the micrograph image, square of the micrograph image.

4.2 Increasing Speed Through Parallel Computing

For the calculation of the sliding window of normalized cross correlation, $(M - N + 1)^2$ calculations of the normalized cross correlation are performed. However, these CC values are independent, therefore we may use parallel computing method increase the speed. Once are calculation are done, we may find the one with the maximum value.

4.3 LSH(Locality-Sensitive Hashing).

We also propose to use Locality-Sensitive Hashing to speed up our algorithm when the dictionary size is large. Currently we compare each micrograph unit with each dictionary item to query the best match, which consumes $O(MN) \times O(comparison)$ time if we have N micrograph units with a dictionary of size M. LSH can reduce it to $O(N) \times O(comparison)$ by pre-processing the dictionary information and hash them to multiple hashtables. Each time when a micrograph unit is given, the algorithm hashes this micrograph unit to these hash tables and only compare it when dictionary items within the same bucket. By selecting a suitable hash function we can efficiently bound the number of HOG-CC comparisons by a constant. The detailed LSH goes like:
(Assume we have micrograph units and dictionary items of size $m \times n$ pixels.)
1). Initialise r (usually $r < 10$) straight lines in the mn dimension Euclidean Space; these lines will serve as hash tables;
2). Divide these straight lines into small segments of length a; these segments will serve as different buckets;

3). The hashing process is actually projecting each dictionary item onto each straight line (hash table), and the segment it falls in is the bucket it is hashed to;
4). Cache these hashing results;
5). Each time when a micrograph unit is input, also hash it to these hash tables;
6). Adopting HOG-CC comparison between this micrograph unit and the dictionary items in the same bucket only guarantees to give the same answer.

Note here are too ways to tune LSH: **AND** and **OR** methods.

AND: we can make a relatively larger, which allows multiple dictionary items to be hashed to the same bucket; then we need to compare the micrograph unit with the dictionary items that are hashed to the same bucket **in each hash table**.

OR: we can make a small enough to ensure that no two dictioanry items are hashed to the same bucket; then we compare the micrograph with the dictionary items that appears in the same bucket in **any hash table**.

Particularly, the **OR** method guarantees that we at most compare r (constant) times and thus bound the computing time by $O(N) \times O(comparison)$.

5 Conclusion

We have established a pipeline to analyze the units in the micrograph. We proposed a Scale Invariant Template Matching approach to find out the units. For the comparison part, first we used HOG to extract features in order to capture the differences. Then we compared the extracted feature vectors to the dictionary items. Since the differences of the dictionary items are too small, we divided the items into groups and see which group the unit belonged. FUTURE WORK can be done to improve the time performance of our algorithm.

6 Acknowledgement

Thanks to University of Tennessee Knoxville, Oak Ridge National Laboratory, The Joint Institute of Computational Science and The Chinese University of Hong Kong.

Also thanks to our mentors Dr. K. Wong, Dr. J. Yin, Dr. R. Archibald, Dr. E. D'Azevedo, Dr. A. Borisevich.

Finally specially thanks to Professor Raymond Chan for his advice of the grouping approach.

References

[1] P. F. Felzenszwalb, R. B. Grishick, D. McAllester, and D. Ramanan. *Object detection with discriminatively trained part based models*, PAMI, 2009.

[2] A. Vedaldi and B. Fulkerson. *VLFeat Library.*
`http://www.vlfeat.org/`

[3] A. Ng(2016) *Machine Learning on Coursera*
`https://www.coursera.org/learn/machine-learning`

[4] Jisung Yoo, Sung Soo Hwang, Seong Dae Kim, Myung Seok Ki, Jihun Cha *Corrigendum to Scale-invariant template matching using histogram of dominant gradients*
Pattern Recognit. 47/9 (2014) 3006–3018
Pattern Recognition, Volume 47, Issue 12, December 2014, Page 3980

[5] Fatih Porikli *Integral Histogram: A Fast Way To Extract Histograms in Cartesian Spaces*
2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition 2005, pp. 829-836, doi:10.1109/CVPR.2005.188

[6] J.P. Lewis *Fast Template Matching*, Vision Interface 95, Canadian Image Processing and Pattern Recognition Society, Quebec City, Canada, May 15-19, 1995, p. 120-123.

[7] S. Korman, D. Reichman, G. Tsur, S. Avidan *FAsT-Match: Fast Affine Template Matching*, CVPR 2013, Portland

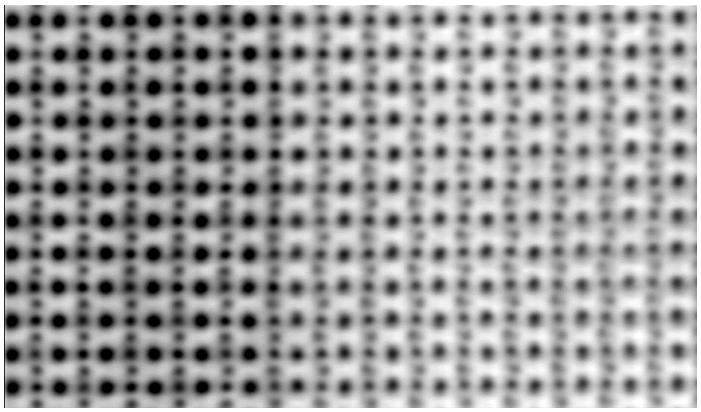
[8] K. Briechle, U. D. Hanebeck *Template matching using fast normalized cross correlation*, SPIE 4387, Optical Pattern Recognition XII, (20 March 2001); doi: 10.1117/12.421129

[9] E.H. Andelson, C.H. Anderson, J.R. Bergen, P.J. Burt, J.M. Ogden *Pyramid methods in image processing*

[10] Summed Area Table, Wikipedia, `http://www.wikibooks.org`

APPENDIX

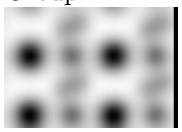
1 Sample Micrograph



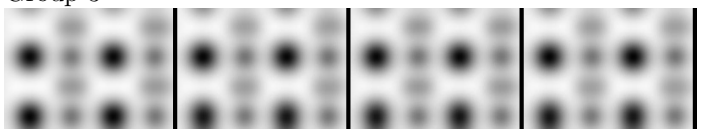
2 Grouping Result

Group 1

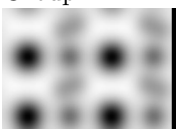
Group 2



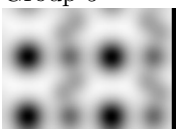
Group 3



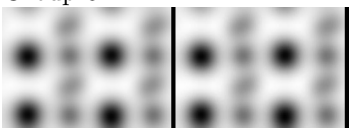
Group 4



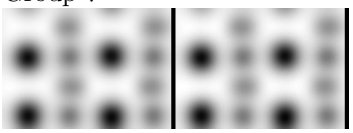
Group 5



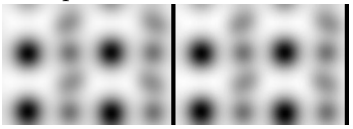
Group 6



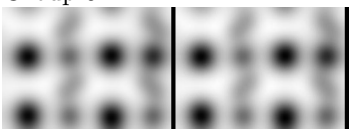
Group 7



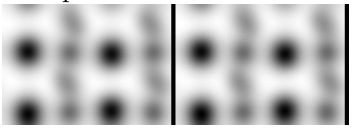
Group 8



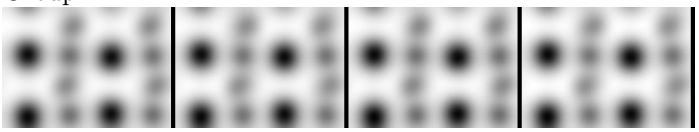
Group 9



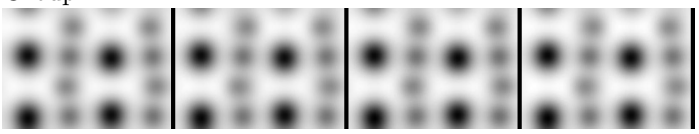
Group 10



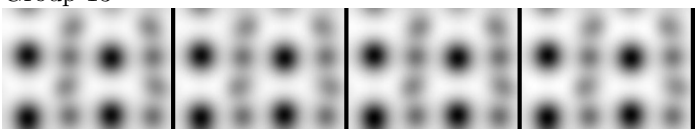
Group 11



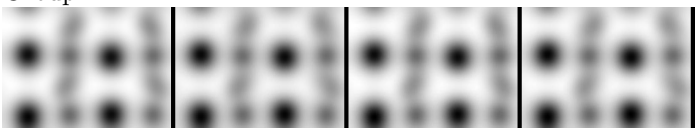
Group 12



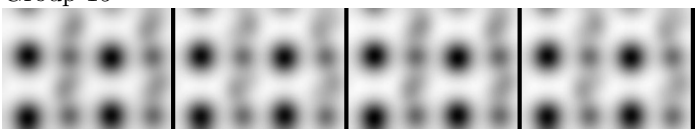
Group 13



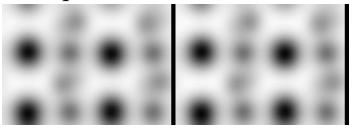
Group 14



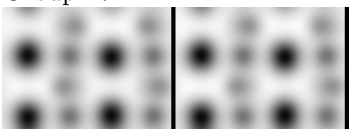
Group 15



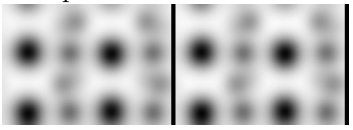
Group 16



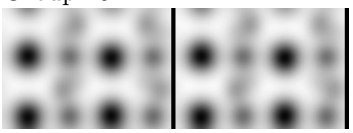
Group 17



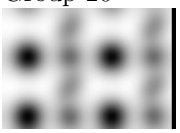
Group 18



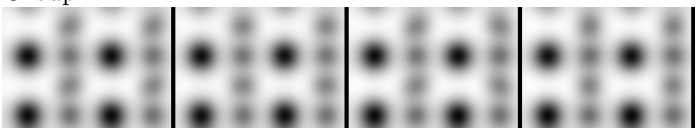
Group 19

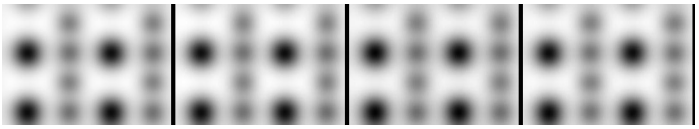


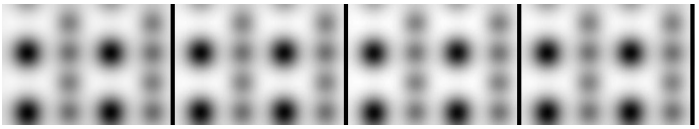
Group 20

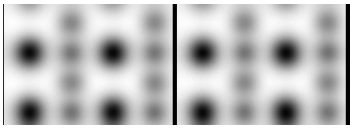


Group 21

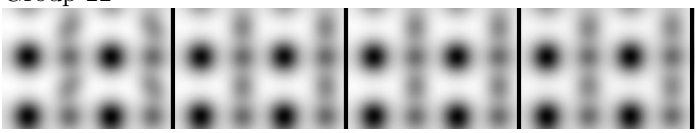


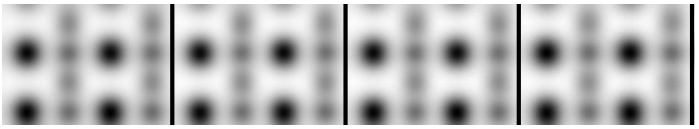


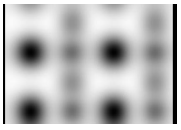




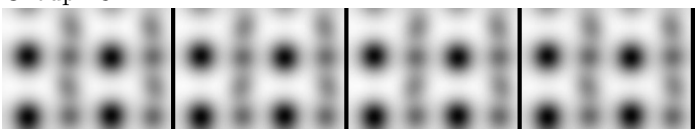
Group 22



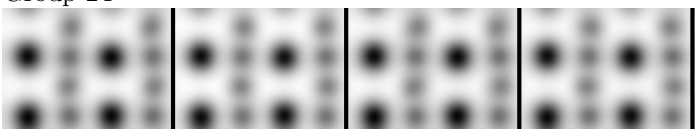


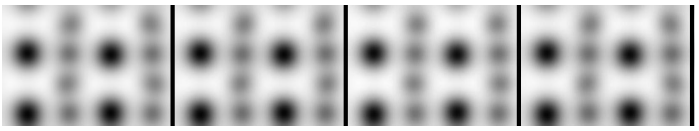


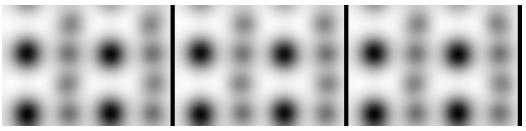
Group 23



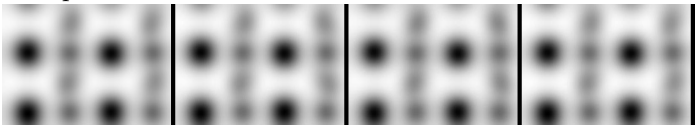
Group 24



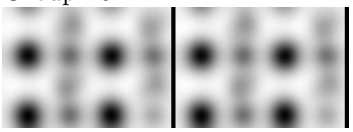




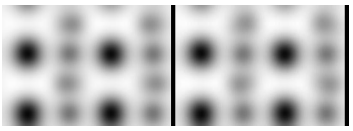
Group 25



Group 26

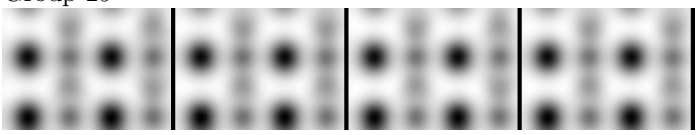


Group 27



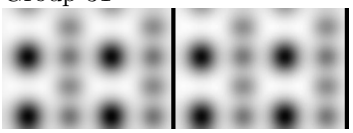
Group 28

Group 29



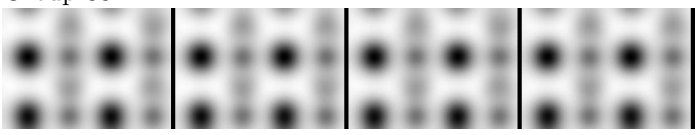
Group 30

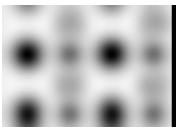
Group 31



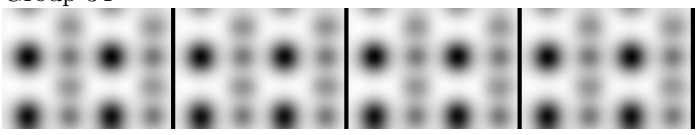
Group 32

Group 33

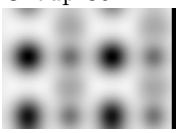




Group 34



Group 35



3 MATLAB Code

The code may contain bugs, please contact us if you found any.

3.1 crosco

```
function output = crosco( image1, image2 )
    %image1 = imread(im1);
    %image2 = imread(im2);
    %image1 = im2double(image1);
    %image2 = im2double(image2);
    imsize = size(image1);
    %assuming that image1 and image2 have the same size

    if length(imsize) > 3
        return
    end

    %calculate sd of image1 and image2
    %did not divide by n for error minimizing (n will be cancelled)
    avg1 = sum(sum(sum(image1)))/numel(image1);
    avg2 = sum(sum(sum(image2)))/numel(image2);
    sd1 = sqrt(sum(sum(sum((image1-avg1).^2))));
    sd2 = sqrt(sum(sum(sum((image2-avg2).^2))));

    %calculating crosco
    output = sum(sum(sum((image1-avg1).*(image2-avg2))))/(sd1*sd2);
end
```

3.2 compareAndPick

```
function bestMatch = compareAndPick(dictionary, item)
    unitLength = size(item,1);
    totalLength = size(dictionary,1);

    if mod(totalLength, unitLength) ~= 0
        return
    end

    bestMatch = 0;
    cc = 0;

    k = totalLength/unitLength;
    for i = 1:k
```

```
        %cc_new = crosco(mask(item),mask(dictionary(((i-1)*unitLength+1):i*
            unitLength,:,:)));
        cc_new = crosco(item,dictionary(((i-1)*unitLength+1):i*unitLength,:,:));
        if cc_new > cc
            bestMatch = i;
            cc = cc_new;
        end
    end
end
```

3.3 levelAdjust

```
%search for x, calculate percentile, search percentile, calculate
function image = levelAdjust(image1, image2)
    size1 = numel(image1);
    size2 = numel(image2);

    rank = 1;
    image2rank = zeros(size(image2));
    for j = 0:255
        for k = 1:size2
            if image2(k) == j
                image2rank(k) = rank;
                rank = rank+1;
            end
        end
    end

    done = 0;
    for i = 0:255
        sum1 = sum(image1(:)==i);
        ratio2 = round(sum1*size2/size1);
        done1 = done;
        done2 = done + ratio2;
        for k = 1:size2
            if (image2rank(k)>done1) && (image2rank(k)<=done2)
                image2(k) = i;
                done = done+1;
            end
        end
    end

    image = image2;
end
```

3.4 hogFeatures

```
function hog = hogFeatures(cellSize, im)
    run('./vlfeat/toolbox/vl_setup');
    %im = imread(image);
    im = single(im);
    hog = vl_hog(im, cellSize);
    %imhog = vl_hog('render', hog, 'verbose');
    %clf; imagesc(imhog); colormap gray;
end
```

3.5 mask

```
function masked = mask(mat)
    sizeMat = size(mat);
    cutLength = round(sizeMat(2)/4);
    masked = [mat(:,(cutLength+1):(cutLength*2)),mat(:,cutLength*3+1:end)];
end
```

3.6 dicMatch

```
function match = dicMatch(dictionary, microscopy, cellSize)
    %dictionary: N*1 cell of strings
    %microscopy: an image
    numDic = size(dictionary, 1);
    dicSample = imread(char(dictionary(1)));
    dicSample2 = im2double(dicSample);
    microscopy = imread(microscopy);
    microscopy = im2double(microscopy);

    [items, itemLength, itemWidth, positions] = FullSILocate(microscopy,
         dicSample2);

    length = size(items,1);
    if mod(length, itemLength) ~= 0
        return
    end

    dic = []; dicFeatures = [];
    for j = 1: numDic
        dictmp = imread(char(dictionary(j)));
        dictmp = double(dictmp);
        dic = [dic; dictmp];
        dictmp = mask(dictmp);
        hogtmp = hogFeatures(cellSize, dictmp);
        dicFeatures = [dicFeatures;hogtmp];
    end

    %hogLength = size(dicFeatures,1)/size(hogtmp,1);

    match = [];
    numItem = length/itemLength;
    figure;
    hAx = axes;
    imshow(microscopy, 'Parent',hAx);
    hold on;
    for i = 1:numItem
        item = items(((i-1)*itemLength+1):i*itemLength,:);
        item = uint8(item*255);
        item = levelAdjust(dicSample,item);
        %item2 = [item2; item];
        item = imresize(item, size(dicSample));
        itemComp = mask(item);
        %itemComp = levelAdjust(mask(dicSample), itemComp);
        hogItem = hogFeatures(cellSize, itemComp);
```

```
        match = [match;compareAndPick(dicFeatures, hogItem)];
        imrect(hAx, [positions(i,2),positions(i,1),itemWidth, itemLength]);
        txt = num2str(match(i));
        text(positions(i,2)+round(itemWidth/5),positions(i,1)+round(itemLength
            /5),txt,'HorizontalAlignment', 'right');
    end
end
```

3.7 crosco more output

```
%this is the function to calculate cross correlation
%the latter two argument is the standard deviation, mean of the image2 (the
    dictionary item)
function output = crosco_more_input(image1, image2,sd2,avg2)
    %image1 = imread(im1);
    %image2 = imread(im2);
    %image1 = im2double(image1);
    %image2 = im2double(image2);
    %imsize = size(image1);
    %assuming that image1 and image2 have the same size

    %if length(imsize) > 3
     % return
    %end

    %calculate sd of image1 and image2
    %did not divide by n for error minimizing (n will be cancelled)
    avg1 = sum(sum(image1))/numel(image1);
    sd1 = sqrt(sum(sum((image1-avg1).^2)));

    %calculating crosco
    output = sum(sum((image1-avg1).*(image2-avg2)))/(sd1*sd2*sqrt(numel(
        image2)));
end
```

3.8 new locate level 1

```
%this is the function to locate where the best sliding window fits
%it assumes no image pyramid used
function [output_coor,cc] = new_locate_level1(input_image,input_template,
    input_template_std,input_template_avg);
%output format is as follows
% p1(row,col) ---- p2(row,col)
% | |
% | |
% p3(row,col) ---- p4(row,col)
imsize = size(input_image);
templatesize = size(input_template);
output_coor = zeros(4,2);
cc = 0;
%counter = 0;
for row = 1:imsize(1)-templatesize(1)+1
    for col = 1:imsize(2)-templatesize(2)+1
```

```
        cc_new = crosco_more_input(input_template,input_image(row:row+
            templatesize(1)-1,col:col+templatesize(2)-1),input_template_std,
            input_template_avg);
        if cc_new > cc
            cc = cc_new;
            output_coor = [row,col;row,col+templatesize(2)-1;row+templatesize
                (1)-1,col;row+templatesize(1)-1,col+templatesize(2)-1];
        end
        %counter = counter+1;
    end
end
%counter;

end
```

3.9 ConvertCoorPyramid

```
%This function converts the coordinate between levels of pyramid.
function output = ConvertCoorPyramid(input_coor,direction,level_size)
%to reduce or expand the coordinate between levels of pyramid
%the default level size is 2
%input is a 4*2 matrix indicates the 4 pts of the rectangle
% p1(row,col) ---- p2(row,col)
% | |
% | |
% p3(row,col) ---- p4(row,col)
%so is output
if exist('level_size') == false
    level_size = 2;
end

output = zeros(4,2);
switch direction
    case 'expand'
        output = floor(level_size*(input_coor-[1,1;1,1;1,1;1,1])+[1,1;1,1;1,1;1,1]);
    case 'reduce'
        output = floor((1/level_size)*(input_coor-[1,1;1,1;1,1;1,1])
            +[1,1;1,1;1,1;1,1]);
end

end
```

3.10 new locate level up

```
%this function implements the image pyramid
function [output_coor,cc] = new_locate_levelup(input_image,input_template,
    iteration,input_template_std,input_template_avg)
%output format is as follows
% p1(row,col) ---- p2(row,col)
% | |
% | |
% p3(row,col) ---- p4(row,col)
output_coor = zeros(4,2);
fluct = 5;
```

```
imsize = size(input_image);
py_image = impyramid(input_image,'reduce');
py_template = impyramid(input_template,'reduce');

if iteration == 1
    output_coor = new_locate_level1(py_image,py_template,input_template_std,
        input_template_avg);
else
    output_coor = new_locate_levelup(py_image,py_template,iteration-1,
        input_template_std,input_template_avg);
end
%imshow(py_image);pause
%imshow(py_template);pause
output_coor = ConvertCoorPyramid(output_coor,'expand',2);
%imshow(input_image(max(output_coor(1,1)-fluct,1):min(output_coor(3,1)+fluct,
    imsize(1)),max(output_coor(1,2)-fluct,1):min(output_coor(2,2)+fluct,imsize
    (2))));pause
coor_add = [max(output_coor(1,1)-fluct,1),max(output_coor(1,2)-fluct,1);max(
    output_coor(1,1)-fluct,1),max(output_coor(1,2)-fluct,1);max(output_coor
    (1,1)-fluct,1),max(output_coor(1,2)-fluct,1);max(output_coor(1,1)-fluct,1),
    max(output_coor(1,2)-fluct,1);];
[output_coor,cc] = new_locate_level1(input_image(max(output_coor(1,1)-fluct,1):
    min(output_coor(3,1)+fluct,imsize(1)),max(output_coor(1,2)-fluct,1):min(
    output_coor(2,2)+fluct,imsize(2))),input_template,input_template_std,
    input_template_avg);
output_coor = coor_add + output_coor;
end
```

3.11 scale

```
%this function implements the Eating from Outside
function output = scale(temp_in,ref_in);
%aims to scale the temp and ref
%works for dim2 only
%should be used with chan vese
%template,reference

%initialization
%temp = imread(temp_in);
%ref = imread(ref_in);
temp = temp_in;
ref = ref_in;
size_temp = size(temp);
size_ref = size(ref);
prescale_temp = min(size_temp/500);
prescale = min(size_ref/250);
temp = imresize(temp,1/prescale_temp);
ref = imresize(ref,1/prescale);
size_temp = size(temp);
size_ref = size(ref);
%check whether RGB or not, somehow rgb2gray doesn't work
if length(size_temp) == 3
    temp = temp(:,:,1);
end
```

```
if length(size_ref) == 3
    ref = ref(:,:,1);
end

%create mask
mask_temp = zeros(size_temp);
mask_temp(temp<mean(mean(temp))) = 0;
mask_temp(temp>=mean(mean(temp))) = 1;

mask_ref = zeros(size_ref);
mask_ref(ref<mean(mean(temp))) = 0;
mask_ref(ref>=mean(mean(temp))) = 1;

temp = im2double(mask_temp);
ref = im2double(mask_ref);

%Since the performance is the best when the ref size is about 200*200, we
%rescale it
%ref = imresize(ref,200*((1/size_ref(1))*size_ref));


%the function of 'eating' as a time parameter
    function output_scale_test = scale_test(input_scale_test,tol)
        output_scale_test = tol;
        tolerance = 10;
        while sum(sum(input_scale_test)) > tolerance
            input_scale_test = conv2(input_scale_test
                ,[1/8,1/8,1/8;1/8,0,1/8;1/8,1/8,1/8],'same');
            input_scale_test(input_scale_test<1) = 0;
            output_scale_test = output_scale_test + 1;
            %imshow(input_scale_test);
            %pause(0.01);
        end
    end

%rescale the picture
noise_mean = 0;
scale_temp = scale_test(imresize(temp(1:min(size_temp(1),2*size_ref(1)),1:min(
    size_temp(1),2*size_ref(2))),3),noise_mean);
scale_ref = scale_test(imresize(ref,3),noise_mean);
output = (scale_ref/scale_temp)*prescale/prescale_temp

%ref_in = imread(ref_in);
%output = imresize(ref_in,scale_temp/scale_ref);
end
```

3.12 ScaleSearch

```
function [output,output_scale] = ScaleSearch(input_image,input_template,fluct);
%fluct defines the ranges
%e.g. [1 2]
%then the program searches from fluct(1),fluct(2),....fluct(end) to find
%the suitable scale
size_image = size(input_image);
```

```
std_template = std2(input_template);
mean_template = mean2(input_template);
compare_template = imresize(input_template,fluct(1));
size_template = size(compare_template);
input_size = min(size_image,max(2*size_template,floor(size_image/2)));
iteration = min(floor(min(log(input_size)/log(2)-6)),floor(min(log(input_size)/log
    (2)-5)));
output_scale = fluct(1);
[best_fit_coor,best_fit_cc] = new_locate_levelup(input_image(1:input_size(1),1:
    input_size(2)),compare_template,iteration,std_template,mean_template);
for trial = 2:length(fluct)
    ratio = fluct(trial);
    compare_template = imresize(input_template,ratio);
    size_template = size(compare_template);
    input_size = min(size_image,max(2*size_template,floor(size_image/2)));
    iteration = min(floor(min(log(input_size)/log(2)-6)),floor(min(log(input_size)/
        log(2)-5)));
    [test_fit_coor,test_fit_cc] = new_locate_levelup(input_image(1:input_size(1),1:
        input_size(2)),compare_template,iteration,std_template,mean_template);
    if test_fit_cc > best_fit_cc
        best_fit_cc = test_fit_cc;
        best_fit_coor = test_fit_coor;
        output_scale = fluct(trial);
    end
end
output = best_fit_coor;
end
```

3.13 SILocate

```
%this function finds the first unit
function [output,scaling] = SILocate(input_image,input_template);
input_size = min(size(input_image),max(2*size(input_template),floor(size(
    input_image))/2));
estimated = scale(input_image(1:input_size(1),1:input_size(2)),input_template);
estimated_scale = floor(10*estimated)/10;
estimated_min = estimated_scale - 0.2*(round(estimated_scale)+1);
estimated_max = estimated_scale + 0.3*(round(estimated_scale)+1);
[output,scaling] = ScaleSearch(input_image,input_template,1./[estimated_min:0.1:
    estimated_max]);
scaling = 1/scaling
end
```

3.14 FullSILocate

```
%This function finds units in the micrograph
function [output_matrix,length,width,output_position,output_coor] = FullSILocate(
    input_image,input_template)
size_image = size(input_image);
size_template = size(input_template);
%locate the first one
temp_coor = SILocate(input_image,input_template);

%change template size according to the first result
```

```
size_template(1) = temp_coor(3,1) - temp_coor(1,1)+1;
size_template(2) = temp_coor(2,2) - temp_coor(1,2)+1;
input_template = imresize(input_template,size_template);

no_col = floor(size_image(2)/size_template(2));
no_row = floor(size_image(1)/size_template(1));
output_coor = zeros(4,2,no_col*no_row);

function output_position = position(coor,image_size,template_size)
    output_position = floor((coor(1,2)-1)/template_size(2))+1 + floor((coor(1,1)
        -1)/template_size(1))*floor(image_size(2)/template_size(2));
end

first_position = position(temp_coor,size_image,size_template);
output_coor(:,:,first_position) = temp_coor;

%input the scale-modified template and the original image
function output_direction_coor = direction_coor(temp_coor,input_image,
    input_template,direction,fluct)
    size_image = size(input_image);
    size_template = size(input_template);
    input_template_std = std2(input_template);
    input_template_avg = mean2(input_template);
    switch direction
        case 'left'
            row_min = max(1,temp_coor(1,1)-fluct);
            row_max = min(size_image(1),temp_coor(3,1)+fluct);
            col_min = max(1,temp_coor(1,2)-size_template(2)-fluct);
            col_max = min(size_image(2),temp_coor(2,2)-size_template(2)+fluct);
            temp_subimage = input_image(row_min:row_max,col_min:col_max);
        case 'right'
            row_min = max(1,temp_coor(1,1)-fluct);
            row_max = min(size_image(1),temp_coor(3,1)+fluct);
            col_min = max(1,temp_coor(1,2)+size_template(2)-fluct);
            col_max = min(size_image(2),temp_coor(2,2)+size_template(2)+fluct);
            temp_subimage = input_image(row_min:row_max,col_min:col_max);
        case 'up'
            col_min = max(1,temp_coor(1,2)-fluct);
            col_max = min(size_image(2),temp_coor(2,2)+fluct);
            row_min = max(1,temp_coor(1,1)-size_template(1)-fluct);
            row_max = min(size_image(1),temp_coor(3,1)-size_template(1)+fluct);
            temp_subimage = input_image(row_min:row_max,col_min:col_max);
        case 'down'
            col_min = max(1,temp_coor(1,2)-fluct);
            col_max = min(size_image(2),temp_coor(2,2)+fluct);
            row_min = max(1,temp_coor(1,1)+size_template(1)-fluct);
            row_max = min(size_image(1),temp_coor(3,1)+size_template(1)+fluct);
            temp_subimage = input_image(row_min:row_max,col_min:col_max);
    end
            coor_add = [row_min-1,col_min-1;row_min-1,col_min-1;row_min-1,
                col_min-1;row_min-1,col_min-1];
            output_direction_coor = coor_add + new_locate_level1(temp_subimage,
                input_template,input_template_std,input_template_avg);
end
```

```
%debug
%output_coor = direction_coor(temp_coor,input_image,input_template,'down',10);
fluct = 10;
col_coor = temp_coor;
for col = 1: floor((temp_coor(1,2)-1)/size_template(2))
    col_coor = direction_coor(col_coor,input_image,input_template,'left',fluct);
    output_coor(:,:,position(col_coor,size_image,size_template)) = col_coor;
end

col_coor = temp_coor;
for col = 1: floor((size_image(2)-temp_coor(2,2))/size_template(2))
    col_coor = direction_coor(col_coor,input_image,input_template,'right',fluct);
    output_coor(:,:,position(col_coor,size_image,size_template)) = col_coor;
end

row_matrix = [];
for i = 1:no_col*no_row
    if output_coor(:,:,i) ~= [0,0;0,0;0,0;0,0]
    row_matrix = cat(3,row_matrix, output_coor(:,:,i));
    end
end

for row_count = 1:size(row_matrix,3)
    row_coor = row_matrix(:,:,row_count);
    for row = 1:floor((row_matrix(1,1,row_count)-1)/size_template(1))
        row_coor = direction_coor(row_coor,input_image,input_template,'up',fluct);
        output_coor(:,:,position(row_coor,size_image,size_template)) = row_coor;
    end

    row_coor = row_matrix(:,:,row_count);
    for row = 1:floor((size_image(1)-row_matrix(3,1,row_count))/size_template(1))
        row_coor = direction_coor(row_coor,input_image,input_template,'down',
            fluct);
        output_coor(:,:,position(row_coor,size_image,size_template)) = row_coor;
    end
end

output_matrix = [];
for i = 1:no_col*no_row
    if output_coor(:,:,i) ~= [0,0;0,0;0,0;0,0]
    output_matrix = cat(1,output_matrix, input_image(output_coor(1,1,i):
        output_coor(3,1,i),output_coor(1,2,i):output_coor(2,2,i)));
    end
end

length = output_coor(3,1,1) - output_coor(1,1,1) + 1;
width = output_coor(2,2,1) - output_coor(1,2,1) + 1;

output_position = [];
for i = 1:no_col*no_row
    if output_coor(:,:,i) ~= [0,0;0,0;0,0;0,0]
    output_position = cat(1,output_position,[output_coor(1,1,i),output_coor(1,2,i)])
        ;
    end
```

```
end

end
```

3.15 kmean

```
function [grouping] = kmeans(images, k, nIter, sizeSet, showresult)
%UNTITLED Summary of this function goes here
% Detailed explanation goes here

images = images(:,:,1);
sizeInput = size(images);
%if ~exist(showresult)
% showresult = 0;
%end
%randome initialization

sizePoint = sizeInput;
sizePoint(1) = sizePoint(1)/sizeSet;

random_pick = randperm(125,k);

centers = zeros(k*sizePoint(1),sizePoint(2));
for i = 1:k
    centers((i-1)*sizePoint(1)+1:i*sizePoint(1),:) = images((random_pick(i)-1)*
        sizePoint(1)+1:random_pick(i)*sizePoint(1),:);
end

grouping = zeros(1,sizeSet);
for i = 1:nIter
    %calculate grouping
    for j = 1:sizeSet
        dist = 100;
        for n = 1:k
            testing_number = (1-crosco(centers((n-1)*sizePoint(1)+floor(
                sizePoint(1)/2)+1:n*sizePoint(1),:),images((j-1)*sizePoint(1)+
                floor(sizePoint(1)/2)+1:j*sizePoint(1),:)))*(1-crosco(centers((n
                -1)*sizePoint(1)+1:(n-1)*sizePoint(1)+floor(sizePoint(1)/2),:),
                images((j-1)*sizePoint(1)+1:(j-1)*sizePoint(1)+floor(sizePoint
                (1)/2),:)))*(1-crosco(centers((n-1)*sizePoint(1)+1:(n-1)*
                sizePoint(1)+floor(sizePoint(1)/2),:),images((j-1)*sizePoint(1)
                +1:(j-1)*sizePoint(1)+floor(sizePoint(1)/2),:))*crosco(centers((n
                -1)*sizePoint(1)+floor(sizePoint(1)/2)+1:n*sizePoint(1),:),images
                ((j-1)*sizePoint(1)+floor(sizePoint(1)/2)+1:j*sizePoint(1),:)));
            if (testing_number < dist)
                dist = testing_number;
                group = n;
            end
        end
        grouping(1,j) = group;
    end
    %reallocate centers
    for n = 1:k
        totalMat = zeros(sizePoint);
```

```
        numPoints = 0;
        for j = 1:sizeSet
            if (grouping(j) == n)
                totalMat = totalMat + images((j-1)*sizePoint(1)+1:j*sizePoint
                    (1),:);
                numPoints = numPoints + 1;
            end
        end
        centers((n-1)*sizePoint(1)+1:n*sizePoint(1),:) = totalMat./numPoints;
    end
    grouping = zeros(1,sizeSet);
end

for j = 1:sizeSet
    dist = 100;
    for n = 1:k
            testing_number = (1-crosco(centers((n-1)*sizePoint(1)+floor(
                sizePoint(1)/2)+1:n*sizePoint(1),:),images((j-1)*sizePoint(1)+
                floor(sizePoint(1)/2)+1:j*sizePoint(1),:)))*(1-crosco(centers((n
                -1)*sizePoint(1)+1:(n-1)*sizePoint(1)+floor(sizePoint(1)/2),:),
                images((j-1)*sizePoint(1)+1:(j-1)*sizePoint(1)+floor(sizePoint
                (1)/2),:)))*(1-crosco(centers((n-1)*sizePoint(1)+1:(n-1)*
                sizePoint(1)+floor(sizePoint(1)/2),:),images((j-1)*sizePoint(1)
                +1:(j-1)*sizePoint(1)+floor(sizePoint(1)/2),:))*crosco(centers((n
                -1)*sizePoint(1)+floor(sizePoint(1)/2)+1:n*sizePoint(1),:),images
                ((j-1)*sizePoint(1)+floor(sizePoint(1)/2)+1:j*sizePoint(1),:)));
        if (testing_number < dist)
            dist = testing_number;
            group = n;
        end
    end
    grouping(1,j) = group;
end

if (showresult == 1)
    figure;
    for n = 1:k
        temp = [];
        for j = 1:sizeSet
            if (grouping(j) == n)
                temp = [temp, images((j-1)*sizePoint(1)+1:j*sizePoint(1),:)/255,
                    zeros(sizePoint(1),10)];
            end
        end
    subplot(k,1,n);
    imshow(temp);
    end
end
end
```

3.16 kmean sampling

```
function [grouping_out] = sampling_kmean(images, k, nIter, sizeSet, noSampling,
    threshold,showresult)
```

```
%intialized the matrix to store the grouping
matrix_record = zeros(sizeSet,sizeSet,noSampling);

%now is the main dish
for times = 1:noSampling
    tic
    [grouping] = kmeans(images,k,nIter,sizeSet,0);
    for compare = 1:sizeSet
        for being_compare = compare:sizeSet
            if (grouping(compare) == grouping(being_compare))
                matrix_record(compare,being_compare,times) = 1;
            end
        end
    end
    times
    toc
end
matrix_record = sum(matrix_record,3)/noSampling;
matrix_record(matrix_record >= threshold) = 1;
matrix_record(matrix_record < threshold) = 0;

track = zeros(1,sizeSet);
grouping_out = zeros(1,sizeSet);
for current = 1:sizeSet
    if (track(current) == 0)
        %haven't been checked
        for being_compare = current:sizeSet
            if matrix_record(current,being_compare ) == 1
                grouping_out(being_compare) = current;
                track(being_compare) = 1;
            end
        end
    end
end

end
```