Progress Report of Intra-image Super-resolution by using self-similar pattern

Hao Zhang
Graduate student,
School of Electrical Engineering and Computer Science
University of Ottawa

Zichao Zhang
Undergraduate student,
School of Electrical Engineering and Computer Science
University of Ottawa

Abstract—

I. Introduction

Super resolution (SR) stands for a class of algorithms enhancing image resolution. Recently, as the machine learning and deep learning gaining great popularity, more people tends to conduct research on super resolution based on machine learning. But learning-based schemes have inner shortcomings, such as requiring large amount of training images to achieve simple algorithms, must retrain the model every time the scaling factor has to be increased and so on.

Another group of studies focused on the repeating feature of an image, like self-similarity driven algorithms. Certain patches of a nartual image occur multiple times, which is also true in more urban images. A great number of these algorithms try to find this kind of patch pairs, one of which is high resolution (HR) and the other is low resolution (LR). However, they only focus on patch pairs correspond after translation, which is often impossible because of the shape of the surface.

In this paper, a more appropriate method is proposed. By allowing geometrical transformation, the size of internal dictionary gets to increase, and a model for this algorithm is composed into perspective distortion for handling structured scenes and additional affine transformation for modeling local shape deformation. [4]

II. Basic concept

In the current development of super-resolution(SR) algorithm, there mainly exists 2 different approaches: Using external database and using data within the origin image.

We propose to work on the an Intra-image Super-resolution by self-similar pattern. This SR algorithm is proposed by [4]. This method only uses data inside the original image to construct a high resolution of the image.

The core of this concept is to reconstruct the high-resolution(HR) image from the infomation within the source image. In this concept, to upscale a part of the image, this algorithm targets to find the best matching patch within the source image, then using a transformation to transport this part into the reconstructed upscaled image.

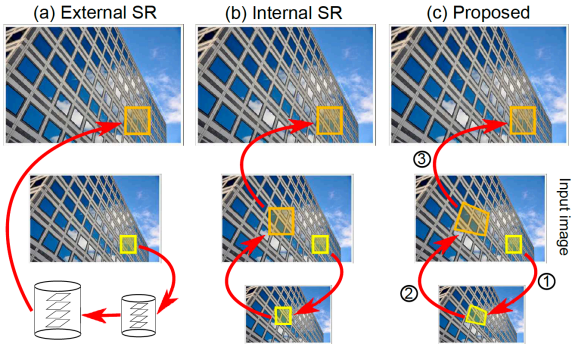


Fig. 1: Different approaches on SR methods by [4].

Comparing with traditional intra-image approaches, this approaches allows warping of the original patch, thus could lead to better quality and less distortion comparing with previous approaches.

The basic algorithm of construct a HR image I_D using source image I in [4] contains following steps:

1) Create a transformation matrix T that transform original patch P to the target best-result patch Q in image I.

2) Take Q from image I and use the inverse tranformation of T to apply Q into the corresponding patch in I_D

3) Repeat previous action till all available patches is transformed.

4) Verify the generated result to make sure the generated I_D is correctly corresponding with original image I.

To find two corresponding patch for upscaling, this method is based on the work from the PatchMatch algorithm from [1]. Patchmatch searches the nearest-neighbor filed (NNF) function between different patches to find two similar patches. This method delivers improvement upon original Patchmatch, which can only search relationships between small square regions. In this algorithm, warped similar patches can also be compared to provide better result for super-resolution.

Test result in [4] shows that this algorithm can deliver a reasonable performance in constructing the texture

of a self-similar structure. Comparing with other SR approaches, this algorithm can create similar or better HR image in typical urban image.

III. Patchmatch algorithm

PatchMatch [1] is a randomized correspondence algorithm used in structural image editing, it saves time for at least an order than ordinary algorithms. In scenarios like image retargeting, image completion and image reshuffling, they are based on densely sampled patches and are able to generate new image that resemble original image, with texture and struct reserved. However, Their computing time is beyond acceptable. In [1], Barnes et al. proposed this method that saves substantial time for applications of structural image editing.

The core of this algorithm is divided into three steps.

1) Initialization: First we need to define a nearest neighbor field(NNF) by randomly generating or using priors. The NNF is a function f taking an image, generating an offset, which is, a coordinate. In practical programming, we can use the source image to generate a matrix with entries the offsets, all randomly generated, of patches in original image corresponded to those in target image. In order to prevent the algorithm stuck in some local minima, we can perform a few iterations first.

2) Iteration: The iteration part is composed of two steps:

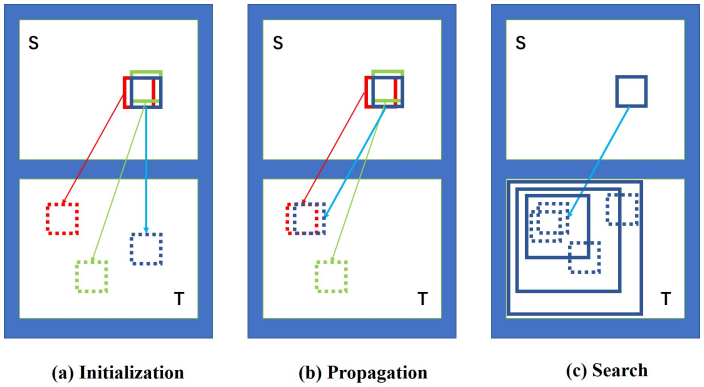
Propagation. For every entry of NNF matrix, we can use the known mapping to help the mapping to be decided. For example, we want to find a good mapping for entry (x, y), we just need to check known mappings at $(x-1, y)$ and $(x, y-1)$. Let $D(\mathrm{v})$ to denote distance metric between two patches, v is just offset, which is the output of function f, also the entries of NFF matrix. To find the best mapping among them, we take $f(x,y) = \arg\min_f \{D(f(x-1,y)), D(f(x,y)), D(f(x,y-1))\}$ The good thing is if $f(x,y)$ is in a coherent region R, we just need to use the corresponding patch of $f(x-1,y)$ or $f(x,y-1)$ right shift or down shift one pixel.

Random Search. After propagation, it's far from enough to find the best mapping, so we still need to improve the current result. So we do random search among a certain area. The search procedure is confined inside a shrinking region, the whole action could be modeled with

$$\mathrm{u}_i = \mathrm{v}_0 + w\alpha^i R_i \tag{1}$$

Where v_0 is the point we want to refine, i represents the i_{th} search sequence, α is a ratio between searching radius, w is the maximum search radius and R is a uniformly distributed random variable in $[-1,1]\times[-1,1]$. We can notice that the radius of searching area shrinks exponentially, we stop the searching process when radius $w\alpha^i < 1$.

We can represent the steps by a figure



(a) Initialization (b) Propagation (c) Search

Fig. 2: (a) Patches randomly assigned to the target image B (b) Propagation progress, the blue patch checks its adjacent patch distance, and assign the shifted patch to itself(probably not assign) (c) The patch searches randomly in the shrinking searching region

The complete process does not only include three steps, after initialization, we will have to iterate several times. Denote Propagation by P, and denote Search by S, they interrleave during the iteration: P, S, P, S, $\cdots$

IV. Projective transformation

Conventional self-exemplar super-resolution schemes only contain translation and rotation, but under circumstances that contain modern artifacts like sktscrapers and bridges, it's better to have projective transformation like affine transformation and projective transformation.

According to [2], in 2D projective plane, points and lines can be represented by homogeneous coordinates. For example, we augment the planar coordinate of a point $\mathbf{x} = (x_1, x_2)^T$ with a third coordinate $\tilde{\mathbf{x}} = (x_1, x_2, x_3)$, every homogeneous coordinate, except $(0,0,0)^T$, corresponds a point on 2D plane. We can know $\tilde{\mathbf{x}} \in \mathbb{R}^3 - (0,0,0)^T$, and $\mathbb{R}^3 - (0,0,0)^T = \mathbb{P}^2$, it is called projective plane. A easy way to visualize this, as in Fig 3 .

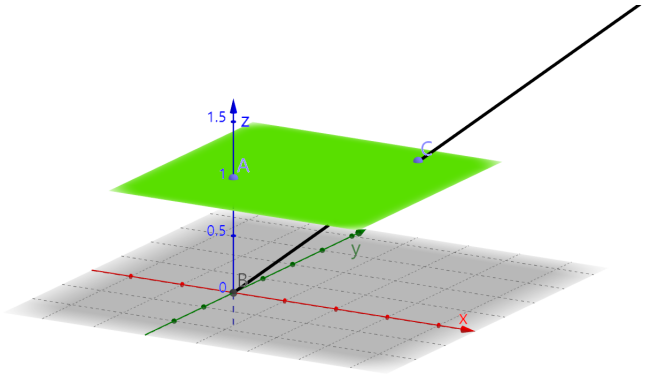


Fig. 3: A homogeneous coordinate can be regarded as a vector in $\mathbb{R}^3$, the corresponding point on 2D plane is the intersection of its direction and $z = 1$

The joint of two lines can be determined by the product of the homogeneous coordinate of these two lines, and a line passing two points can be determined by the cross product of the homogeneous coordinate of these two points.

1) Ideal points and the line at infinity: We know in standard Euclidean geometry $\mathbb{R}^2$, parallel lines won't intersect, but it's different in projective space. Two parallel lines can intersect at ideal points, which is a point with $x_3 = 0$. Imagine the vector showed in Fig 3 to fall on the xy plane, it intersects the $z = 1$ plane at infinity, which corresponds the ideal point. And it's proved that all the ideal points are collinear, they are all on the line at infinity, which is $\mathbf{l}_\infty = (0, 0, 1)^T$

Under projective transformation or affine transformation, parallel lines are still mapped to lines intersect at infinity, but this time this infinity line is a little different, it becomes visible on graph. Like in Fig 4, BA and CD are parallel lines, but in perspective view, they intersect at point F, and same with CB and DA, E and F is called vanishing point, they are transformed from infinity to here. Two vanishing points can determine the line at infinity, EF. If we can find the representation of this line, we can transform it back to infinity, and inversely, if we get the transformation matrix, we can transform the rectangle to this quadrilateral, which is what we are going to do in PatchMatch

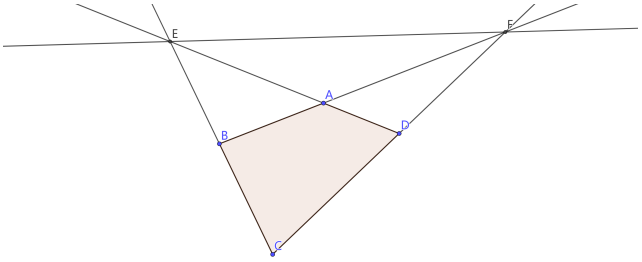


Fig. 4: The vanishing points determine an infinity line

2) Transformation matrix: As described in Fig 1, we need a transformation matrix to "wrap" the patch to its corresponding patch in downsampled image, and then use the inverse to "unwrap" the high resolution patch to paste on the constructed image. The projective matrix $\mathbf{T}$ can be composed into a chain of transformations.

$$\mathbf{T} = \mathbf{T}_S \mathbf{T}_A \mathbf{T}_P \tag{2}$$

We parameterize $\mathbf{T}$ by $\theta = (\mathbf{s}, m)$, where $\mathbf{s} = (s^x, s^y, s^s, s^\theta, s^\alpha, s^\beta)$, so

$$\mathbf{T}(\theta) = \mathbf{T}(\mathbf{t}, s^x, s^y, m)\mathbf{S}(s^s, s^\theta)\mathbf{A}(s^\alpha, s^\beta) \tag{3}$$

The matrix $\mathbf{S}$ captures the similarity transformation

$$\mathbf{S}(s^s, s^\theta) = \begin{bmatrix} s^s\mathbf{R}(s^\theta) & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4}$$

where s^s is a scaling parameter and $\mathbf{R}(s^\theta)$ is a rotation matrix. Matrix

$$\mathbf{A}(s^\alpha, s^\beta) = \begin{bmatrix} 1 & s^\alpha & 0 \\ s^\beta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

represents shearing mapping in affine transformation.

V. Detecting Planar Surface and Regularity

A great number of photos containing modern artifacts are taken from a perspective view, so if we want to find a self-exemplar patch, we would need to perform projective transformation. A basic problem is to detect the plane for the artifact, in order to determine the matrix $\mathbf{T}$ [3]. First we extract lines, or line segments, to find the corresponding vanishing point. We only need to find up to three vanishing points, this means we assume there are at most three planes in an image, which is reasonable for manmade structures.

We can represent the plane m by a vanishing line $\mathbf{l}_\infty^m$, the perspective view of this image can be affine rectified using a pure perspective transformation matrix

$$\mathbf{H}_m = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1^m & l_2^m & l_3^m \end{bmatrix} \tag{6}$$

VI. Specific Implementation

1) Parameter setting: The size of our testing image (HR) is 384×512, patch size is set to 16, which may vary. For simplicity concerning, we determine a patch by its upper-left pixel, instead of the center point. We set down sampling rate to 2 (LR), and because we need to reconstruct a super-resolution image, we set the resolution of super-resolution (SR) image to 768×1024.

2) Initializing: First of all, we build a matrix f of size 377×505, which stores the corresponding coordinate in LR of every patch in HR. The fastest way is to simply initialize the patches with its coordinate divided by two, after all we are matching patches between two same images, with only difference in size.

3) Propagation: Then we do as said in section III, for every chosen patch in HR, we find the corresponding patch of size 16×16 in LR with matrix f, every entry of f, $f(i, j)$ represents the coordinate in LR of corresponding pixel $H(i, j)$. Then we compare by

$$\text{Loss} = \sum_i \sum_j (p_H(i, j) - p_L(i, j))^2 \tag{7}$$

where p_H represents the patch in HR and p_L represents the patch in LR. We ignore the patches that exceed the boundary of LR.

4) Searching: It's nearly impossible that we find the ideal patch in LR all at once, so this process provides us with another chance to find the satisfying patch.

We set the initial searching radius to half of the image parameter, which is the maximum of image height and image width. After that, crop the searching area with LR image boundary. Randomly pick one patch from the searching area, then compare the loss between this patch and the patch in f. Then keep shrinking the searching radius by half, until it's smaller than one.

5) Iteration: Iterate the above propagation and searching process over and over again, and particularly, we change the direction of propagation in even iterations, which means from bottom-right to upper-left.

The general time of one iteration is around 10 seconds, which is acceptable with the image size and huge amount of computing.

6) Reconstruction: In order to obtain an upsampled image, I used the method of upsampling from ELG 5378, generate one empty image of size 768×1024 and stuff which with pixels of HR with one zero between two pixels. Then filter this image with a low-pass filter, we get the upsampled image.

In this part, my method is simple, divide the SR by 32×32 patches, then each patch corresponds one 16×16 block in f(the missing part of f is stuffed with replicated value from the border). The basic idea was to find the best patch from this 16×16 block. So I compute the loss of every patch in this block and pick the minimum one, paste the corresponding patch in HR to SR. Beyond that, we set a threshold to decide whether substitute the original patch or not.

The result of this method is rather miserable, If we set a high threshold, only a few patches are substituted, but we still got some successful cases. After lowering the threshold, image becomes uglier

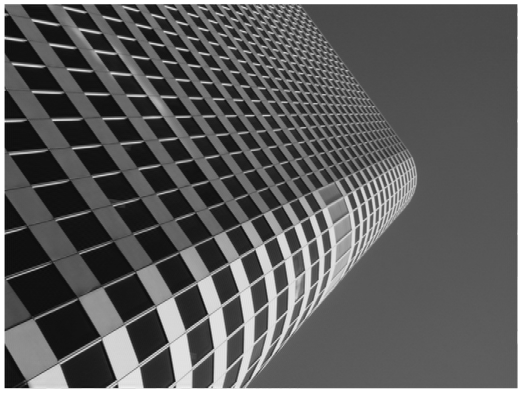


Fig. 5: Reconstructed image with high threshold

Fig. 6: Reconstructed image with high threshold

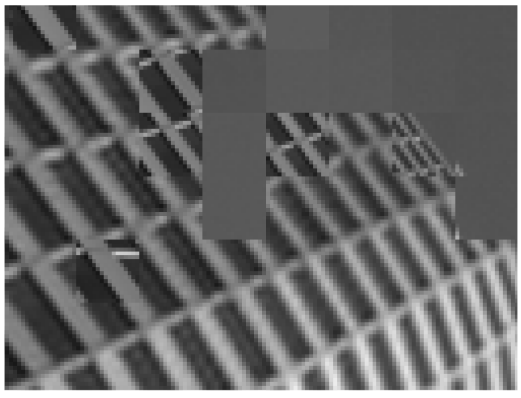


Fig. 7: Some successful cases

A. Detecting Planar Surface and Regularity

A great number of photos containing modern artifacts are taken from a perspective view, so if we want to find a self-exemplar patch, we would need to perform projective transformation. A basic problem is to detect the plane for the artifact, in order to determine the matrix $\mathbf{T}$ [3]. First we extract lines, or line segments, to find the corresponding vanishing point. We only need to find up to three vanishing points, this means we assume there are at most three planes in an image, which is reasonable for manmade structures.

We can represent the plane m by a vanishing line $\mathbf{l}_\infty^m$, the perspective view of this image can be affine rectified using a pure perspective transformation matrix

$$\mathbf{H}_m = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1^m & l_2^m & l_3^m \end{bmatrix} \tag{8}$$

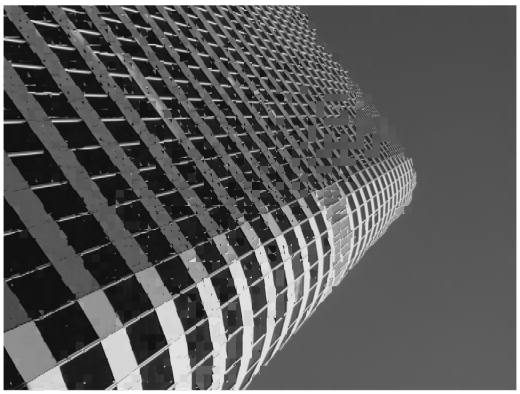


Fig. 8: Reconstructed image with lower threshold

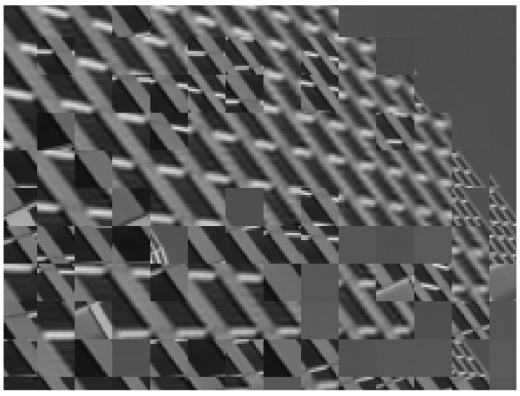


Fig. 9: Zoomed in image

VII. Plane and Vanishing Point Detection

There are many methods proposed to identify planes or rectify planes, but most of them are complex and computational requiring. Here we extract plane information by analysing the known image region. According to [3], they used a relatively standard method to extract planes. Based on the proposed method, we made some change to it, with the price of increasing computation time. Here we used a combined process including line segment extraction [?], line information computation [?], [?], grouping, vanishing point estimation [?] and plane support region computation [3]. The result of vanishing point estimation can be used for determining transformation matrix as said in section VI-A. We are using the images provided by [4], and I provide the link as follows: [1]. In the following procedure I will mainly use this image:

Fig. 10: The image we are using

A. Edge detection

Modern research has developed many powerful and high efficiency methods for edge detection, for example, Sobel method, Canny method or Fuzzy Logic method. In this project I used Sobel method to detect edges first. The Sobel method applies a filter, or a kernel, to an image and different structure of Sobel operator will produce different results. If we use κ to represent our kernel, there are two kinds of Sobel kernels:

$$\kappa_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \tag{9}$$

This is the kernel for extracting horizontal edges. Once the kernel is applied to horizontal edge, the difference on both sides of the edge will be magnified by this kernel, while flatten area will only produce zero, because the nine pixels corresponding to the kernel will be the same. Fig. 11 shows the result of applying horizontal Sobel operator to our image.

And we also heve the kernel for vertical edges:

$$\kappa_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{10}$$

And the result is shown in Fig. 11

After we apply this to our image, we can consider this as a differential operation. When we apply horizontal Sobel operator, we will get $\mathbf{I}(x+1, y) - \mathbf{I}(x, y)$, in other words, $\frac{\partial \mathbf{I}}{\partial x}$. Similarly, we can get $\frac{\partial \mathbf{I}}{\partial y}$ after we apply κ_y. In the end we get two matrix with the same size our image, the entries of them will be $\mathbf{I}_x(x, y)$ or $\mathbf{I}_y(x, y)$. Thus, as proposed by [?], we can compute gradient magnitude m by adding these two matrices $m = |\mathbf{I}_x| + |\mathbf{I}_y|$. By adding them up, we get a magnitude graph (Fig. 14).

[1] Huang et al. 2015
https://github.com/jbhuang0604/StructCompletion/tree/master/data

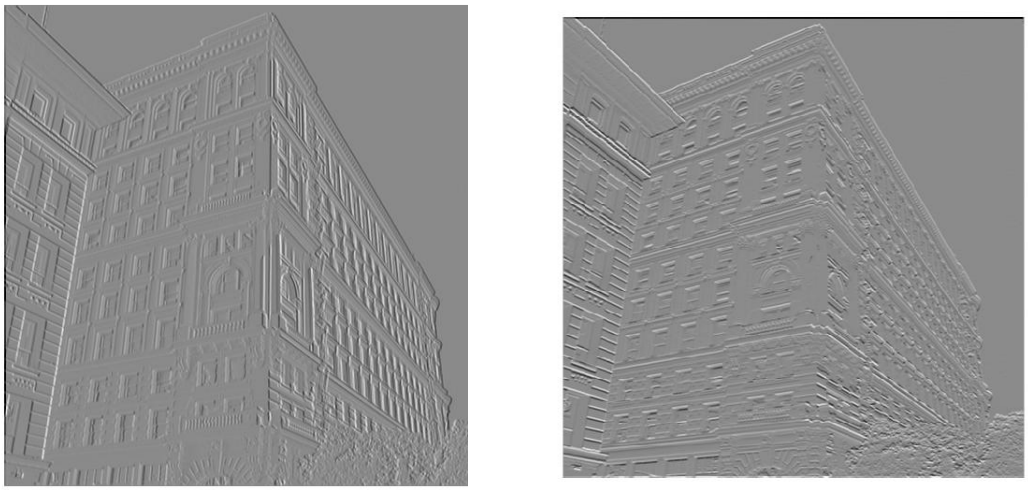


Fig. 11: Horizontal edges (left) and vertical edges (right)

Fig. 12: Magnitude graph

As you can see, Most of the light parts of this image are edges. If we apply Sobel operator to image, pixels corresponding to edges will usually have high values, those corresponding to smooth area will have zero value or small value. In order to filter these pixels with small value, in other words, to exclude those components don't look like edges, we set a threshold to cut those pixels with small values. The matrix we get will be a map indicating those parts 'look like' edges, which will be used later to get orientation map.

Then we do elementwise division between $\mathbf{I}_y$ and $\mathbf{I}_x$ to get gradient orientation. If we regard $\mathbf{I}$ as a two dimensional signal or function, its gradient will be

$$\nabla \mathbf{I} = \begin{bmatrix} \mathbf{I}_x \\ \mathbf{I}_y \end{bmatrix} \tag{11}$$

which is a vector on a 2D plane, if we compute $\tan^{-1} \frac{\mathbf{I}_y}{\mathbf{I}_x}$, we can get its orientation. An example of gradient orientation map Θ is shown in 13

According to amplitude map, those parts that have small value in amplitude map will be also erased in the gradient orientation map, we call this process 'filtering'. According to 13, we can find that the whole figure becomes more cleaner than before. In gradient orientation map, the value of every entry varies from $-\pi$ to $+\pi$.

B. Grouping

Once the local gradient orientation has been estimated, The remaining problem will be segmenting or classifying the line segments. Here we classified them using fixed partitions as proposed by Burns et al. The range of orientation is 360 degrees, divide this into 8 sections we can get a pie chart

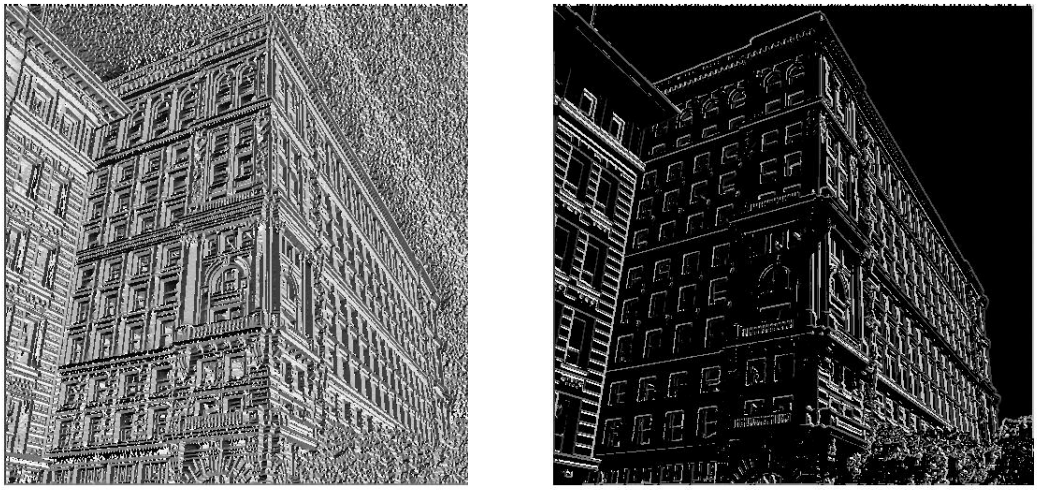


Fig. 13: Gradient orientation map (left) and the map after filtering (right)

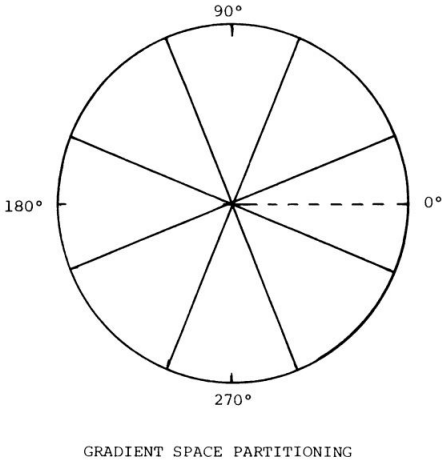


Fig. 14: Orientation sectors (the buckets)

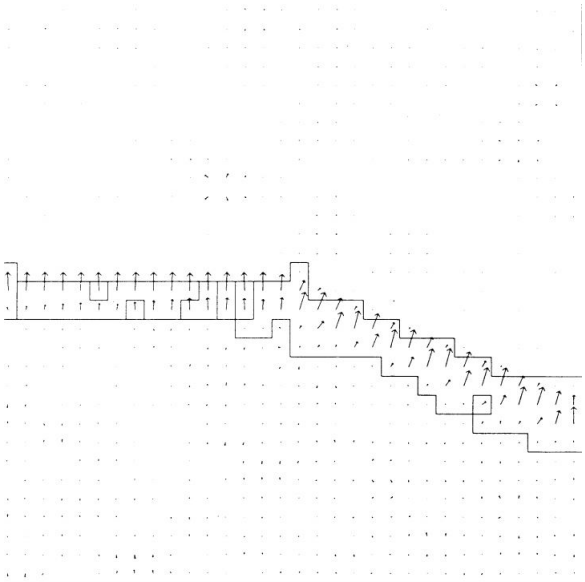


Fig. 15: Two different lines are labeled with the same bucket

Each sector is called a 'bucket', in practice, each pixel will fall into a bucket according to its gradient orientation, which means, it is labeled with the corresponding bucket. Notice that we are classifying our line segments, so when we apply this classifying scheme to line segment, lines labeled with opposite bucket will have the same orientation, possibly they are the same line.

However, there are problems with this simple approach. visually distinct lines could fall into the same bucket because they have similiar orientation. Fig. 15 shows one case of this situation.

In order to fix this problem, we apply another biased bucket, namely the first sector centers at 22.5° as shown in Fig. 16. Now each line support region will have at most 2 different labels, we didn't use the voting method mentioned in the paper, instead I erased all the line support region with different labels. After that we can apply Connected Components Algorithm (CCA) to the result and we get thousands of connected region.

However, after experiment, we are having many region that is not even straight line, they are mostly scattered points or tiny spots. So I applied filtering to it, regions have connected pixels that have small number are removed

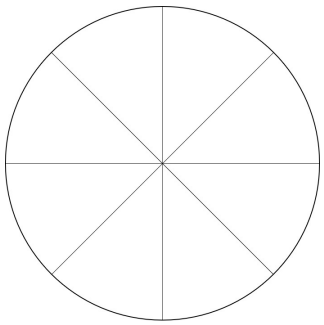


Fig. 16: Biased bucket

for further process. The final result is shown in Fig. 17

Fig. 17: Filtered result after CCA

Now there are three orientations among our lines, the reason for this will be explained later. Next step will reasonably be separating them, in other words, grouping them. In order to specify which bucket they are distributed (now that there are four buckets in total), histogram of the figure is used.

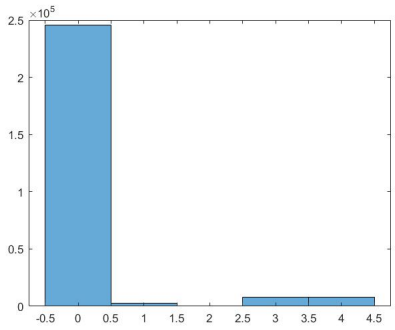


Fig. 18: Histogram of buckets in which lines are distributed

It is clear that except those pixels labeled 0 (they are filtered before), our lines are distrubuted in bucket 1, 3 and 4, and they are shown in Fig 19.

C. Line information computation

After extracting line segments of each bucket, we still need to obtain their orientation and position so we can get the functions of them. I used PCA (Principal Component Analysis) method to get the direction vector of line segments.

1) Brief introduction to PCA method: PCA is one of the classical dimensionality reduction algorithms, also used for lossy data compression, feature extraction and data visualization. [?]. Many dimension reduction and feeature extraction algorithms like PCA also reduce noise and relieve overftting, so they will improve performance on supervised learning. PCA can be considered as projecting the data onto a lower dimension subspace while maximizing the variance of projected data. Like the projection shown below

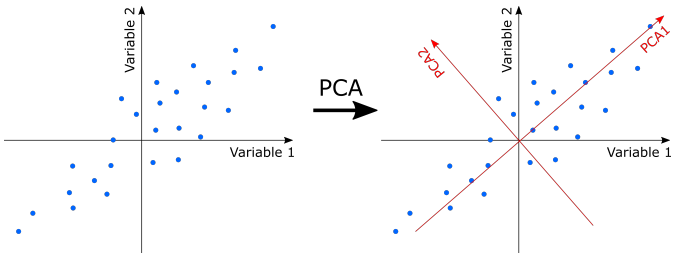


Fig. 20: Project data onto lower dimensions

Data points are projected onto a vector indicating their distributed direction on which they have the biggest variance. The direction is the eigenvector of the covariance matrix, and variance will be eigenvalue of the covariance matrix.

2) Computing eigenvectors: If we want to compute eigenvectors, first we need the covariance matrix. Covariance matrix is defined as follows

$$\mathbf{C}_{X_iX_j} = cov[X_i, X_j] = E[(X_i - E[X_i])(X_j - E[X_j])] \quad (12)$$

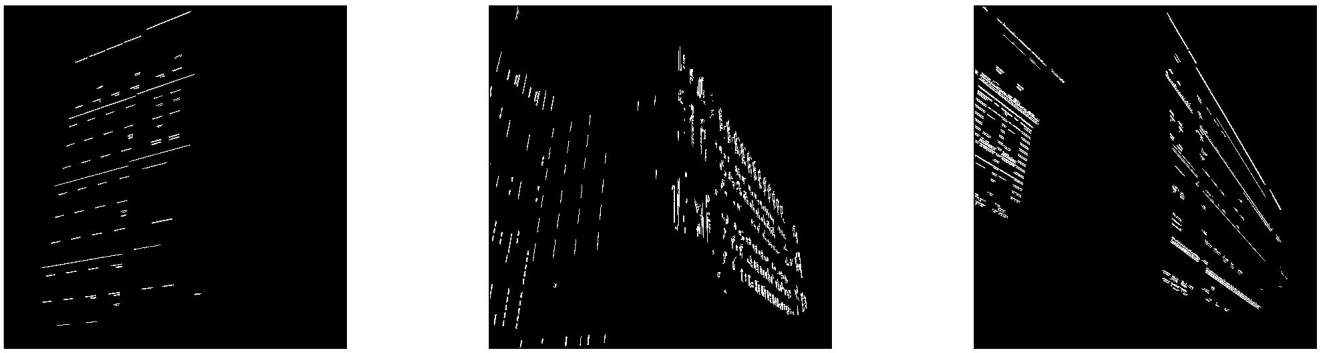
where $\mathbf{C}$ is the covariance matrix and $\mathbf{X} = [X_1, X_2, \cdots, X_d]^T$. It can be arranged as the form

$$\begin{aligned} \mathbf{C} =& cov[\mathbf{X}, \mathbf{X}] = E[(\mathbf{X} - \mu_{\mathbf{X}})(\mathbf{X} - \mu_{\mathbf{X}})^T] & (13) \\ =& E[\mathbf{X}\mathbf{X}^T] - \mu_{\mathbf{X}}\mu_{\mathbf{X}}^T & (14) \end{aligned}$$

where $\mu_{\mathbf{X}} = E[\mathbf{X}]$.

First thing to do is to construct the covariance matrix, according to (14), the true expectation can be computed by empirical expectation, which is, the mean value. If we represent our data point as $X_i = (x_i, y_i)$, then subtract every data point with mean value $X_i - \bar{X}$, the covariance matrix will be

$$\mathbf{C} = \frac{1}{N}\sum_{i=1}^{N} X_i^T X_i = \frac{1}{N}\mathbf{X}^T\mathbf{X} \quad (15)$$



(a) (b) (c)

Fig. 19: (a). Line segments fell into bucket 1. (b). Line segments fell into bucket 2. (c). Line segments fell into bucket 3.

Compute its eigenvector corresponsing to the biggest eigenvalue, we can get the expected eigenvector, which indicates the orientation of line segment. Beyond that, the eigenvalues of covariance matrix show the variance of projection on that direction, so we can compute the ratio of two eigenvalue, if the ratio is bigger than a threshold, we can consider it as an ellipsoid or something other than straight line. Note that after compute the eigencalues, we still have to normalize them.

As for line position, we just need to compute the mean value of the line segment, in this case, it will be the middle point of that line segment.

D. Vanishing point estimation

In projective geometry, if we represent straight lines by homogeneous coordinate, we can determine the intersection of two lines by computing the cross product between them: $x = l_1 \times l_2$. We applied a voting scheme called RANSAC to compute the vanishing point of one group of line segments. The basic idea is to first randomly pick two lines and compute their intersection, then compute the loss by including all the line segments. The optimal combination would be the one that has minimum loss. If the vanishing point is on one line, the product between this point (homogeneous coordinate) and the line would be zero

$$x^T l = 0 \tag{16}$$

On the other hand, if it's not on the line, the product between them will not be zero, so the loss function can look like the following:

$$L = \sum_{i=1}^{N} (x^T l_i)^2 \tag{17}$$

After trying every possible combination, we can find the optimized one pair of lines. One vanishing point is shown in Fig. 21

E. Plane support region computation

A plane typically consists of two sets of parallel 3D lines, there are usually two sets of the line segments with two distinct VPs that should reside within the same image region. We identify the support of each plane by locating positions where the two sets of line segments corresponding to the two VPs overlap with each other. Choose one set of line segments and filter which with Gaussian kernel, then make element-wise multiplication. We can fix one plane with two sets of line segments, here we assume there's only three plane orientations, which is always the case for artificial structures, that's the reason why there's only three groups of line orientations.

A Gaussian kernel is also a Gaussian filter, applying it to line segments expands the support region of lines, by multiplying the vertival and horizontal Gaussian kernel, we get results in Fig. 22

And the result is used as plane prior, for locating planes, those region of prior probability that has bigger value will more likely to be a plane. Here shows two planes extracted from the image.

By computation, three groups of line segments are shown in different color, indicating different orientation or plane.

Acknowledgment

I have read the content about plagiarism, "PLAGIARISM is taking another person's words, ideas or statistics and passing them off as your own. The complete or partial translation of a text written by someone else also constitutes plagiarism if you do not acknowledge your source."[2] I've fully understanded the consequence of plagiarism and would restrictly follow the rules.

[2]How to avoid plagiarism
https://www.uottawa.ca
/about/sites/www.uottawa.ca.about/files/plagiarism.pdf

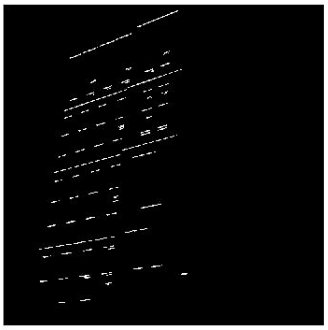
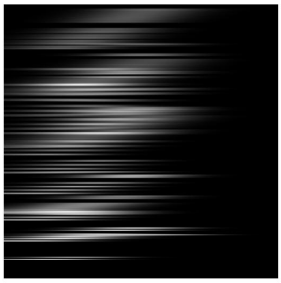
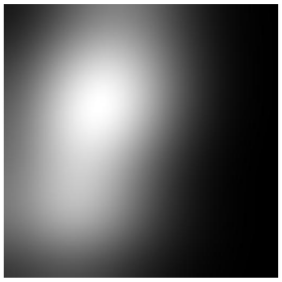


Fig. 21: Vanishing point of one group of lines



(a)



(b)

Fig. 22: (a). Result after applying horizontal Gaussian kernel 7 times (b). Result after applying vertical Gaussian kernel 7 times

Fig. 23: Planes extracted from the image

References

[1] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. In ACM Transactions on Graphics (ToG), volume 28, page 24. ACM, 2009.

[2] R. I. Hartley and A. Zisserman. Multiple View Geometry in Computer Vision. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[3] Jia-Bin Huang, Sing Bing Kang, Narendra Ahuja, and Johannes Kopf. Image completion using planar structure guidance. ACM Trans. Graph., 33(4), July 2014.

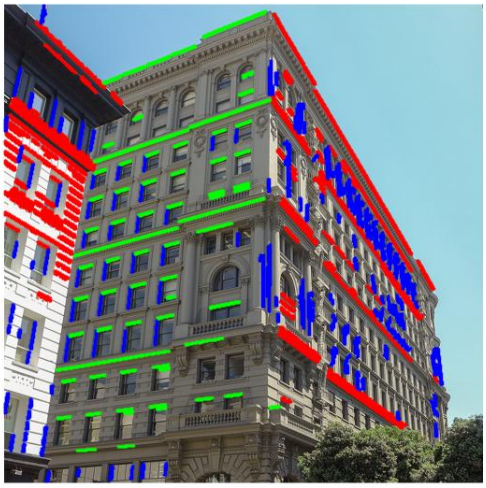


Fig. 24: Line segments in the figure

[4] Jia-Bin Huang, Abhishek Singh, and Narendra Ahuja. Single image super-resolution from transformed self-exemplars. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 5197–5206, 2015.