# Numerical Analysis Project: Image Stitching

December 17, 2023

## 1 Introduction

Image stitching is a computer vision technique that involves combining multiple images into a seamless panoramic or wide-angle image. The process aims to create a comprehensive view by aligning and blending individual images, thereby overcoming the limitations of a single photograph's field of view.

This technology finds applications in various domains, such as photography, virtual reality, and surveillance. In photography, image stitching enables the creation of stunning panoramic photographs, capturing expansive landscapes or crowded cityscapes. In virtual reality, stitched images contribute to immersive 360-degree experiences, providing users with a sense of being present in a particular environment.

## 2 Some challenges in stitching images

First of all, to start image stitching, the first question is how to find the similar points between two images. To solve this problem, we use SIFT or SURF to find the feature points in the images, then use their orientation assignment to match them together. After finding the similar points between them, we have to calculate the transformation of these matching points. In order to achieve it, we found that the transformation between the real things and the image in camera is called homography or projective. The next problem is that it might be match wrong in SIFT or SURF, so we use an algorithm called RANSAC to exclude the point we don't want to. Last, after we find the best matching matrix, there is still a problem. The absorbance of the light in two image might not be totally same. That would cause the area of overlapping part become blurry, and the edge of the image would be obvious. Thus, we use the weighted RGB between two image to blend it.

## 3 SIFT [1] [2]

**Introduction**

We use SIFT to find feature points of a image because feature points obtained by SIFT are invariant to image scaling and rotation, and partially invariant to change in illumination. First, we use Gaussian blur to get the sequence of DOG in order to find the candidate feature points. Second, we delete some feature points which have low contrast and locate on the edge. Finally, we use feature point descriptor to assign each feature point attributes so as to define the feature point we find.

**Gaussian Blur**

Let $f$ be a function whose domain is the size of an image $a \times b$, then $f : (0, a) \times (0, b) \subset \mathbb{N}^2 \to \mathbb{N}^3$ defined by

$f(x,y) = (f_1(x,y), f_2(x,y), f_3(x,y))$, where $0 \leq f_i(x,y) \leq 255$ for $i = 1,2,3$, each $(x,y) \in dom(f)$ is called a pixel, and $f_1, f_2, f_3$ decide 'red', 'green', 'blue' respectively. Gaussian Blur uses PDF of $N(0,\sigma^2)$ to assign weight of each pixel in order to get a weighted average $u$ to each pixel, then multiply the value of each pixel by $u$ to reduce the difference between other pixels and then to blur the image. Because an image is defined on 2-dimension space, we use the 2-dimensional PDF $g(x,y)$ of $N(0,\sigma^2)$(ie : $g(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$) to perform the method mentioned above.

Next problem is how to perform the method mentioned above. We use 'convolution' to do it. Setting $f$ and $g$ as above, and denoting the convolution of $f$ and $g$ by $f * g$, then we can obtain a new image $l(x,y) = (f*g)(x,y)$. By the definition of convolution, we have

$$l(x,y) = \sum_{s=1}^{k} \sum_{t=1}^{k} f(x,y) g(x-s, y-t) \tag{1}$$

, for $k$ is odd which is closest to $2\pi\sigma$. We get the weighted function

$$L(x,y) = \frac{l(x,y)}{\sum\limits_{s=-\frac{k}{2}}^{\frac{k}{2}} \sum\limits_{t=-\frac{k}{2}}^{\frac{k}{2}} g(x,y)} \tag{2}$$

Repeating convolution n times, we get a sequence $\{L_1, L_2, ..., L_n\}$. Let $L'_i = L_{i+1} - L_i$ for $i = 1, 2..., n-1$, then we have a sequence$\{L'_1, L'_2, ..., L'_{n-1}\}$ and each $L'_i$ is called a difference of Gaussian(DOG)

Figure 1: Gaussian Blur

**Detection of feature point**

We do Gaussian blur to get a group of images called an interval, and a octave consists of these images.

Figure 2: DOG

Repeatedly Gaussian blur with the number of pixels multiplied by 1/2 and then we can get many octaves. We first consider one of these octaves. By finding maximum and minimum, we can detect rough feature point. In one of octaves, we find maximum and minimum by comparing a pixel to its 26 neighbors in 3x3x3 regions at the current and adjacent interval. Although we can find maximum and minimum by the way mentioned above, it's possible that the maximum or minimum isn't the true maximum or minimum because we find maximum and minimum on the discrete space. We use the following way to avoid this situation.

**Locate true maximum and minimum**

Suppose we find a extreme value $(x, y)$ in one of interval of an octave. we select points near $(x, y)$ to give a approximate function by Taylor polynomial, saying $f$. We translate $(x, y)$ to $(0, 0)$ and then do Taylor polynomial of degree 2 about $(0, 0)$ and we can get

$$f(x,y) \approx f(0,0) + (\frac{\partial f}{\partial x}x + \frac{\partial f}{\partial y}y) + \frac{1}{2}(\frac{\partial^2 f}{\partial x^2}x^2 + 2\frac{\partial^2 f}{\partial x \partial y}xy + \frac{\partial^2 f}{\partial y^2}y^2) \tag{3}$$

If we change equation(3) into the form of matrix, we can get

$$f(\begin{bmatrix} x & y \end{bmatrix} \approx f(\begin{bmatrix} 0 & 0 \end{bmatrix}) + \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix} \begin{bmatrix} x & y \end{bmatrix}^{\mathrm{T}} + \frac{1}{2} \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} \begin{bmatrix} x & y \end{bmatrix}^{\mathrm{T}} \tag{4}$$

Rewrite equation(4) in the form of vector, then we have

$$f(a) \approx f(0) + \frac{\partial (f)^{\mathrm{T}}}{\partial a} a^{\mathrm{T}} + \frac{1}{2} a \frac{\partial^2 f}{\partial a^2} a^{\mathrm{T}} \tag{5}$$

,where $a = \begin{bmatrix} x & y \end{bmatrix}$. Finding the derivative of $f$, we have

$$\frac{\partial f}{\partial a} = \frac{\partial (f)^{\mathrm{T}}}{\partial a} + \frac{\partial^2 f}{\partial a^2} a \tag{6}$$

and let equation(6)=0, then we have

$$a = - \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}^{-1} \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{7}$$

Because $a$ means the offset from $(x, y)$, if the offset along x-axis or y-axis exceeds 0.5, then we change another maximum or minimum to perform the same way to get another $a$. If a pixel keeps on oscillating then we discard it and we say it is unstable.

**Eliminate low contrast points and edge responses**

Let $(x, y)$ be a stable maximum or minimum. By the method above, the offset $a$ can be calculated. If $f(a)$ is less than 0.03 then we discard $(x, y)$ and saying it's a low contrast point(we assume the value of pixel in [0,1]).Next we eliminate the edge responses. Because the maximum or minimum $(x, y)$ having severe change along only one direction isn't the feature point we want, we use the following method to delete such point. Recall that eigenvalue of Hessian matrix means the rate of change along corresponding eigenvector. We use $f$ mentioned above to get a Hessian matrix $H$

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} \tag{8}$$

Let $\lambda_1$ and $\lambda_2$ be two eigenvalues of $H$, then we have Det(H)=$\lambda_1$+$\lambda_2$ and trace(H)=$\lambda_1 \times \lambda_2$. Observing

$$\frac{trace(H)^2}{Det(H)} = \frac{(r+1)^2}{r} \tag{9}$$

where $r = \frac{\lambda_1}{\lambda_2}$ and the quantity $\frac{(r+1)^2}{r}$ is at a minimum when the two eigenvalues are equal and it increases with r. If

$$\frac{trace(H)^2}{Det(H)} > \frac{(r+1)^2}{r} \tag{10}$$

where $r$=10, then we discard this point $(x, y)$.

**The code associated with finding feature points in SIFT algorithm**

```
\begin{verbatim}
points = detectSIFTFeatures(Image)
checkImage(I);

Iu8 = im2uint8(I);

if isSimMode()
    params = parseInputs(varargin{:});
    PtsStruct=ocvDetectSIFT(Iu8, params);
else
    params = parseInputs_cg(varargin{:});

    PtsStruct = vision.internal.buildable.detectSIFTBuildable.detectSIFT(...
        Iu8, params.ContrastThreshold, params.EdgeThreshold, ...
        params.NumLayersInOctave, params.Sigma);
end
```

```
Pts = SIFTPoints(PtsStruct.Location, PtsStruct);
```

**Orientation assignment**

The scale of the feature point is used to select the Gaussian smoothed image, $L$, with the closest scale, so that all computations are performed in a scale-invariant manner. For each pixel $(x, y)$, we can get the gradient magnitude $m(x, y)$ and orientation $\theta(x, , y)$ as following

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1)^2)} \tag{11}$$

$$\theta(x,y) = tan^{-1}(((L(x+1,y) - L(x-1,y))^2)/((L(x,y+1) - L(x,y-1))^2)) \tag{12}$$

We find points in the neighborhood of $(x, y)$ to calculate $m(x, y)$ and $\theta(x, y)$ and then use a Gaussian-weighted circular window with $\sigma = 1.5\times$the scale of the feature point to obtain the weighted gradient magnitude. Dividing 360° into 36 partitions averagely and creating the histogram having 36 bins with the height in each bin determined by the sum of weighted $m(x, y)$ in the corresponding direction. the direction of the peak of the histogram is the direction of the feature point. Besides, any other local peak that is within 80% of the highest peak is used to also create a feature point with that orientation and same location.

**Descriptor representation**

Figure 3 shows the computation of the feature point descriptor. First, we compute gradient magnitudes and

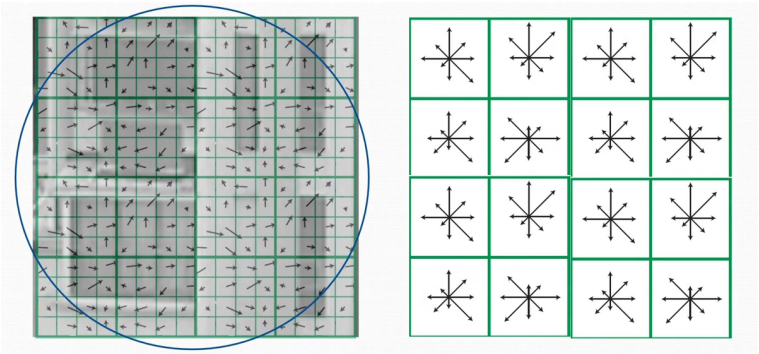

Figure 3: feature point descriptor

[3]

orientations of points which is near to feature point, as the left of Figure 3, using the scale of feature point to determine the level of the Gaussian blur for the image. Second, to achieve orientation invariance, we rotate the coordinates of the descriptor and the gradient orientations relatively to the orientation of feature point. A Gaussian weighting function with $\sigma = 1.5$xthe width of the descriptor window is used to give the weight to the magnitude of points around the feature point. We use this way to avoid the sudden changes in the descriptor with small changes in the position of the window and to focus on points which is near the feature point because the effect of points near to the feature point is greater than points far from the feature point. These points are then accumulated into orientation histograms summarizing the contents over 4x4 subregions, as the right of Figure 3, with the length of each arrow corresponding to the sum of the gradient magnitudes near that direction within the region.

**The code associated with orientation assignment and descriptor representation in SIFT algorithm**

```
        function [features, valid_points] = extractSIFTFeatures(I, points)

[Iu8,ptsStruct] = parseSIFTInputs(I,points);

if isSimMode()
    [vPts, features] = ocvExtractSift(Iu8, ptsStruct);
else
    [features, vPts] = ...
        vision.internal.buildable.extractSIFTBuildable.extractSIFT_uint8(...
        Iu8, ptsStruct);
end

% modify the orientation so that it is measured counter-clockwise from
% horizontal the x-axis
vPts.Orientation = single(2*pi) - vPts.Orientation;

if isBRISKPointsObj(points)
    scale = vPts.Scale * 4.5;
    scale(scale <12) = 12;
    valid_points = BRISKPoints(vPts.Location, 'Metric', vPts.Metric,...
        'Scale', scale, 'Orientation', vPts.Orientation);
elseif isSURFPointObj(points)
    scale = vPts.Scale * 0.75;
    scale(scale <1.6) = 1.6;
    valid_points = SURFPoints(vPts.Location, 'Metric', vPts.Metric,...
        'Scale', scale, 'Orientation', vPts.Orientation);
elseif isKAZEPointObj(points)
    scale = vPts.Scale * 0.75;
    scale(scale <1.6) = 1.6;
    valid_points = KAZEPoints(vPts.Location, 'Metric', vPts.Metric,...
        'Scale', scale, 'Orientation', vPts.Orientation);
else
    valid_points = SIFTPoints(vPts.Location, vPts);
end
```

# 4 SURF [4] [5] [6] [7] [8] [9]

The concept of the SURF algorithm is based on the SIFT algorithm,but there are some differences in detail. The algorithm has three major steps, including feature points detection, scale-space representation and location of feature points, and feature description.

**Hessian matrix**

In SIFT algorithm, image pyramid is constructed by the difference of Gaussian(DOG), however, it is time-consuming. In SURF algorithm, image pyramid is constructed by the determinant of Hessian matrix. The determinant of Hessian matrix represents the amount of change around the pixel point. Therefore, feature points will be selected at the maximum and minimum values of the Hessian's determinant. Given the the point (x,y) in image I, Hessian's matrix H(x,y,$\sigma$) at point (x,y) at the scale $\sigma$ is defined as the following:

$$H(x,y,\sigma) = \begin{bmatrix} \frac{\partial^2 f((x,y,\sigma))}{\partial x^2} & \frac{\partial^2 f((x,y,\sigma))}{\partial x \partial y} \\ \frac{\partial^2 f((x,y,\sigma))}{\partial x \partial y} & \frac{\partial^2 f((x,y,\sigma))}{\partial y^2} \end{bmatrix} \tag{13}$$

where selecting points near $(x, y)$ to give a approximate function by Taylor polynomial, saying $f$. Hence, to find the determinant of H, we need to apply convolution with Gaussian filter, then second-order derivative. However, the amount of calculation is large. Therefore, SURF uses box filter instead of Gaussian filter as an approximation of Gaussian smoothing. This box filter can use integral image to accelerate the calculating speed by just requiring evaluations at the rectangle's four corners. So the determinant of H can be approximately calculated as

$$Det(H_{approx}) = D_{xx} * D_{yy} - (0.9 * D_{xy})^2 \tag{14}$$

, where D is a function obtained by box filter. In this equation, 0.9 is the deviation with box filter comparing to Gaussian filter.

**The algorithm of integral image**

In the algorithm of integral image, the positive x-axis points to the right and the positive y-axis points below. The value at any point(x,y) in the integral image is defined as the sum of all the pixels above and to the left of (x, y). Let I be the mapping from (x,y) to the value in the integral image at (x,y). Then we can have the following equation:

$$I(x,y) = \sum_{\substack{x' \le x \\ y' \le y}} i(x',y') \tag{15}$$

,where i(x,y) is the value of the pixel at (x,y) We can compute integral image efficiently at (x,y) because of the following equation derived from the definition of the functions I and i:

$$I(x,y) = i(x,y) + I(x,y-1) + I(x-1,y) - I(x-1,y-1) \tag{16}$$

As long as the integral image has been computed, evaluating the sum of value over any rectangular area requires exactly four array references regardless of the area size. That is, the notation in the figure below, having A = (a,c), B = (b,c), C = (a,d), D = (b,d),the sum of i(x,y) over the rectangle spanned by A, B, C, and D is:

$$\sum_{\substack{a<x\le b \\ c<y\le d}} i(x,y) = I(D) + I(A) - I(B) - I(C) \tag{17}$$

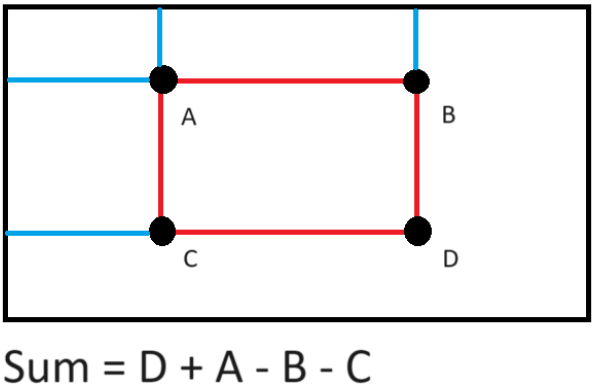


Figure 4: integral image

[4]

**Scale-space representation**

In SURF algorithm, when constructing the scale space, the size of images remain same in different levels. The changing factor is the size of box filter. In SURF, the lowest level of the scale space is obtained from the output of the 9×9 box filters. The upper level of the scale space, the size of the box filter becomes larger. For example, 15x15,21×21,27×27..... In general, the formula of the scale is:

$$\sigma_{approx} = current\ filter\ size * \frac{base\ filter\ scale}{base\ filter\ size} \tag{18}$$

By finding the maximum and minimum values of the Hessian's determinant, we can detect rough feature point. The method is similar to the SIFT algorithm. In one of levels, we find maximum and minimum by comparing a pixel to its 26 neighbors in 3x3x3 regions at the current and adjacent levels. The method of eliminating low contrast points and edge responses in surf algorithm is similar to that in sift algorithm.

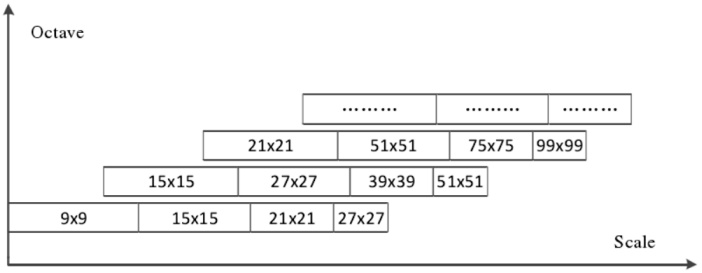


Figure 5: scale representation

[5]

**The code associated with finding feature points in SURF algorithm**

```
function Pts = detectSURFFeatures(I, varargin)
checkImage(I);
```

```
Iu8 = im2uint8(I);

if isSimMode()
    [Iu8, params] = parseInputs(Iu8,varargin{:});
    %ocvFastHessianDetector is a built-in function to calculate the
    %approximate determinant of Hessian matrix and construct the scale space
    %and find feature points
    PtsStruct=ocvFastHessianDetector(Iu8, params);

else
    [I_u8, params] = parseInputs_cg(Iu8,varargin{:});

    % get original image size
    nRows = size(I_u8, 1);
    nCols = size(I_u8, 2);
    numInDims = 2;

    if coder.isColumnMajor
        % column-major (matlab) to row-major (opencv)
        Iu8 = I_u8';
    else
        Iu8 = I_u8;
    end
    [PtsStruct_Location, PtsStruct_Scale, PtsStruct_Metric, PtsStruct_SignOfLaplacian] = ...
        vision.internal.buildable.fastHessianDetectorBuildable.fastHessianDetector_uint8(Iu8, ...
        int32(nRows), int32(nCols), int32(numInDims), ...
        int32(params.nOctaveLayers), int32(params.nOctaves), int32(params.hessianThreshold));

    PtsStruct.Location        = PtsStruct_Location;
    PtsStruct.Scale           = PtsStruct_Scale;
    PtsStruct.Metric          = PtsStruct_Metric;
    PtsStruct.SignOfLaplacian = PtsStruct_SignOfLaplacian;
end

PtsStruct.Location
= vision.internal.detector.addOffsetForROI(PtsStruct.Location,params.ROI,params.usingROI);

% Object for storing SURF feature points and provides the ability to pass data between
% the detectSURFFeatures and extractFeatures functions.
Pts = SURFPoints(PtsStruct.Location, PtsStruct);
```

**Orientation assignment**

1. In SURF algorithm, we first calculate the Haar-wavelet responses in x and y-direction, and this in a circular neighborhood of radius 6s around the feature point, with s the scale at which the feature point was detected. Also, the sampling step is scale dependent and chosen to be s, and the wavelet responses are computed at that current scale s. Accordingly, at high scales the size of the wavelets is big. Therefore integral images are used again for fast filtering.

2. Then we calculate the wavelet responses and weighted with a Gaussian ( $\sigma = 2.5$s) centered at the feature point, the responses are represented as vectors in a space with the horizontal response strength along the abscissa and the vertical response strength along the ordinate. The dominant orientation is estimated by calculating the sum of all responses within a sliding orientation region covering an angle of $\frac{\pi}{3}$ . The horizontal and vertical responses within the region are summed. The two summed responses then yield a new vector. The longest such vector lends its main orientation to the feature point.

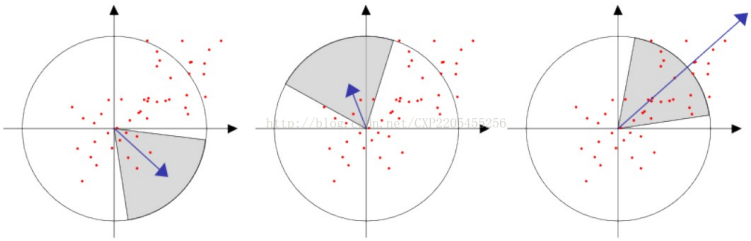

Figure 6: orientation assignment

[8]

**Descriptor representation**

1. Constructing a square region centered around the feature point and oriented along the orientation we already got above. The size of this region is 20s.

2. Then the region is split up regularly into smaller $4 \times 4$ square sub-regions. For each sub-region, we calculate a few simple Haar-wavelet features at 5×5 regularly spaced sample points. For reasons of simplicity, we call $d_x$ the Haar wavelet response in the horizontal direction and $d_y$ the Haar wavelet response in the vertical direction (filter size 2s). To increase the robustness towards geometric deformations and localization errors, the responses $d_x$ and $d_y$ are first weighted with a Gaussian ( $\sigma$= 3.3s) centered at the feature point.

3. Then, the wavelet responses $d_x$ and $d_y$ are summed up over each sub-region and form a first set of entries to the feature vector. In order to bring in information about the polarity of the intensity changes, we also calculate the sum of the absolute values of the responses, $|d_x|$ and $|d_y|$. Hence, each sub-region has a four-dimensional descriptor vector $v$ which can be represented as

$$v = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|) \tag{19}$$

. This results in a descriptor vector for all 4×4 sub-regions of length 64.

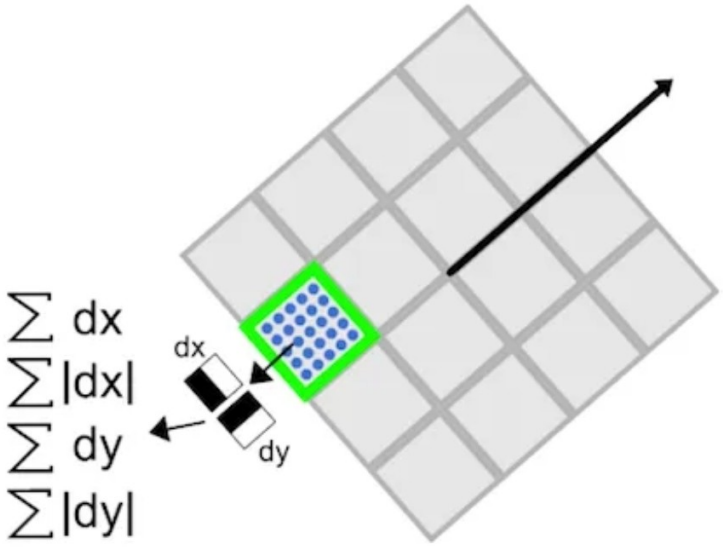


Figure 7: descriptor representation

[8]

**The code associated with orientation assignment and descriptor representation in SURF algorithm**

```
function [features, valid_points] = extractSURFFeatures(I, points, SURFSize,upright)
[Iu8,ptsStruct] = parseSURFInputs(I,points);

params.extended = SURFSize == 128;
params.upright  = upright;

if isSimMode()
    [vPts, features] = ocvExtractSurf(Iu8, ptsStruct, params);
end

% modify the orientation so that it is measured counter-clockwise from
% horizontal the x-axis
vPts.Orientation = single(2*pi) - vPts.Orientation;

if isCornerPointObj(points)
% For cornerPoints input, valid_points is a cornerPoints object
valid_points = cornerPoints(vPts.Location, 'Metric', vPts.Metric);

elseif isBRISKPointsObj(points)
    valid_points = BRISKPoints(vPts.Location, 'Metric', vPts.Metric,...
```

```
            'Scale', 6 * vPts.Scale, 'Orientation', vPts.Orientation);
    elseif isSIFTPointObj(points)
        valid_points = SIFTPoints(vPts.Location, 'Metric', vPts.Metric,...
            'Scale', (4/3) * vPts.Scale, 'Orientation', vPts.Orientation);
        valid_points = computeSIFTOctavefromSIFTScale(Iu8, valid_points);
    else
        % For other inputs, valid_points is a SURFPoints object
        valid_points = SURFPoints(vPts.Location, vPts);
    end
```

# 5 Key Point Matching

The Key Point Matching is designed to find a 1-1 relation between key points in two images. After finding n feature points with key point descriptor of size (16,8), we flatten the data to the size (n,128). The best candidate match for each key point is found by identifying its nearest neighbor in the database of key points from another image. The nearest neighborhood is defined as the key point with the minimum Euclidean distance for the vector.

$$d(x,y) = \sqrt{\sum_{j=1}^{m}(a_j - b_j)^2} \tag{20}$$

For boosting calculations, we use the following distance where m is the dimension of key point descriptor, which is 128 in this article.

$$d(x,y) = \sum_{j=1}^{m}|a_{ij} - b_{ij}| \tag{21}$$

The idea of pairing key points is simply choose one key point in the first picture, and iterate all key points in second picture to find the point minimizing the distance(p1,p2). However, many features from an image would not have a correct match due to background clutter or were not detected in the training images. Therefore, there should be a way to discard feature points that do not have any good match. We define ratio of distances by closest distance divided by second-closet distance, then obtain the relation between ratio of distances and probability of correct matching as Figure 9. We applying the method by discarding those feature points with ratio of distances higher than 0.8. A result from [2] shows that it eliminates 90% of false matches while discarding less than 5% of the correct matches. A sample code having a similar effect is provided below.

```
def keypoint_matching(keypoint1, keypoint2):
    pairs = []
    for point in keypoint1:
        distance = sort([dist(point1,point2) for point2 in keypoint2])
        closest_distance = distance[0]
        second-closest distance = distance[1]
        if closest distance / second-closest distance < 0.8:
            point2 = find_distance_equals(closest_distance,point1,keypoint2);
            pairs.append(point,point2)
    return pairs
```

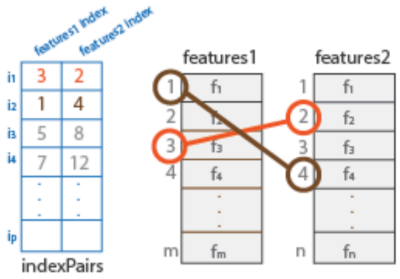


Figure 8: a simple illustration of keypoint matching [2]

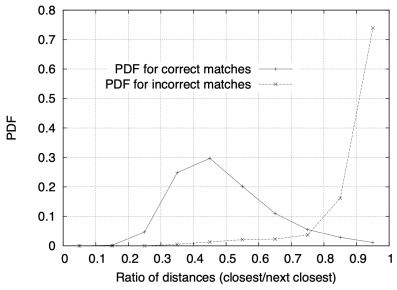


Figure 9: pdf of correct and incorrect matches [2]

# 6 RANSAC(Random Sample Consensus) [10] [11]

**Introduction**

The most important function of RANSAC is to exclude the outliers. For instance, in Figure, some features don't match(as Figure 10). Hence, we need to use the method of RANSAC to exclude features which don't match successfully.

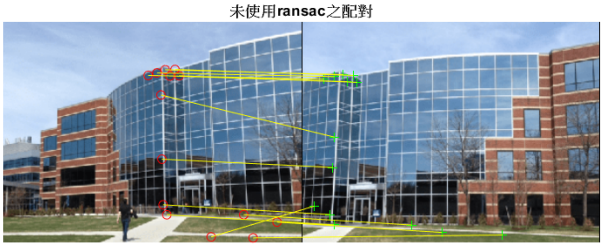


Figure 10:

**Definition of inliers and outliers**

Inliers:The data points that are considered consistent with the suitable model being estimated.

Outliers:The data points that deviate significantly from the suitable model.

**Why the method of RANSAC is pivotal in the homography?**

When we compute the homography matrix, given by all pairs of matching points. By these data, we can compute this homography matrix directly. However, we know that this homography matrix is not the most awesome since not all pairs of matching features are necessary. Hence, we need to use the method of RANSAC to exclude some pairs of matching features that we think it is not necessary.

**Algorithm**

The RANSAC algorithm is composed of these steps:

(1)Choose s samples "randomly". s is the minimum samples to fit a model.s is usually a small number, such as 2 or 3 to form our initial model hypothesis.

(2)Find the suitable model to the randomly chosen samples. For instance, when s=2, the suitable model is a

line; when s=3, the suitable model is a plane.

(3)Count the number of data points (inliers) that fit the model within a measure of error e.(As figure11, s=2, we can define error e as the distance, if the distance of a feature point and fitting line > e, then this feature will be regarded as the outlier.)

(4)Repeat step(1) to step(3) N times.(N can be defined by ourselves)

(5)Choose the model generated by this N times. Find the model that has the largest number of inliers.

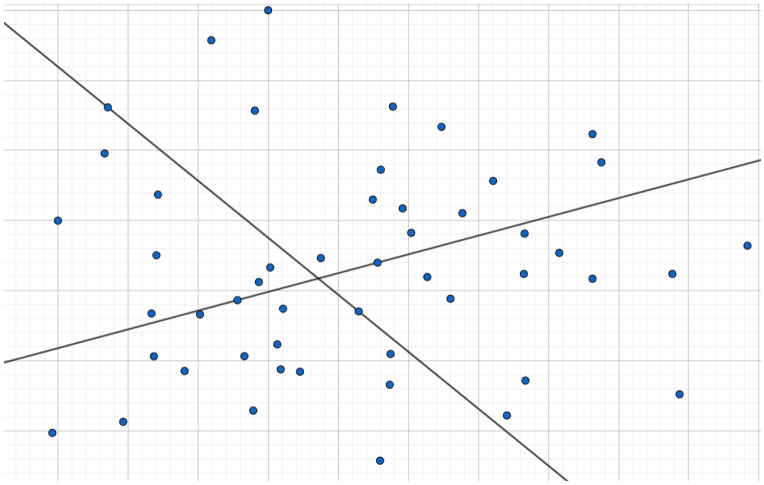

Figure 11: Choose the line that has the most inliers

**A sample code of RANSAC**

```
[tform,inlierIdx] = estgeotform2d(matchedimg1Points,matchedimg2Points, "projective" );
inlierimg1Points = matchedimg1Points(inlierIdx,:);
inlierimg2Points = matchedimg2Points(inlierIdx,:);
```

# 7 Mathematical formulations of image transformations and images wrapping [12] [13]

1.

To find the linear transformation matrix from $(x_1, y_1)$ to $(x_2, y_2) = (x_1 + t_1, y_1 + t_2)$, because there is no way to incorporate these constants $t_1, t_2$ into a $2 \times 2$ transformation matrix. We have to solve it by bumping it up to a representation by one dimension,

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_1 \\ 0 & 1 & t_2 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \tag{22}$$

and we call it affine transformation.

2.

To get every kinds of image transformations, we first try the $2 \times 2$ matrix. Then, we can get the stretching image,

$$\begin{cases} x_2 = kx_1 \\ y_2 = y_1 \end{cases} \tag{23}$$

by the matrix,

$$T = \begin{bmatrix} k & 0 \\ 0 & 1 \end{bmatrix} \tag{24}$$

we can get the rotation image,

$$\begin{cases} x_2 = x_1 \cos\theta - y_1 \sin\theta \\ y_2 = y_1 \cos\theta + x_1 \sin\theta \end{cases} \tag{25}$$

by the matrix,

$$T = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{26}$$

and we can also get the shearing image,

$$\begin{cases} x_2 = x_1 + ky_1 \\ y_2 = y_1 \end{cases} \tag{27}$$

by the matrix,

$$T = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix} \tag{28}$$

but we can't use a $2 \times 2$ matrix to represent a translation image,

$$\begin{cases} x_2 = x_1 + t_1 \\ y_2 = y_1 + t_2 \end{cases} \tag{29}$$

Thus, we have to use the matrix in 7-1 which is called affine transformation. Affine transformation is a linear 2-dimensions transformation combined by translation, stretching, rotation, and shearing.

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{30}$$

To represent affine transformations with matrices, we can use homogeneous coordinates. This means representing a 2-vector $(x, y)$ as a 3-vector $(x\mathbb{Z}, y\mathbb{Z}, \mathbb{Z})$. Using this system, translation can be expressed with matrix multiplication, and it becomes:

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \tag{31}$$

which is called affine transformation.

What if we do not fix $(0, 0, 1)$ in the last row, and let it become

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ z_1' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \tag{32}$$

We say the matrix H maps from $\mathbb{R}^2$ to $\mathbb{R}P^2$ (called real projective plane). Real projective plane is a space which collects all the lines through the origin in $\mathbb{R}^3$. Then we use homogeneous coordinate we talked before to represent the point in the space. Thus, the vector $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$ equals to $\begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix}$ for $z \neq 0$ and the other will be defined as infinity point in $\mathbb{R}P^2$.

And the matrix H called homography transformation or projective transformation.

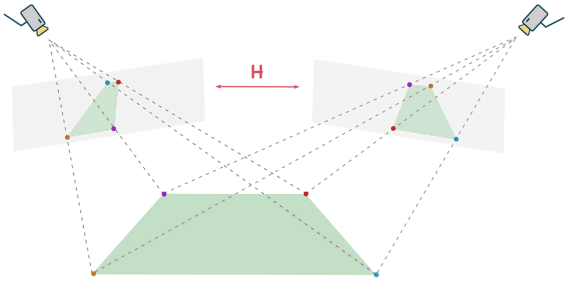


Figure 12: An example taken from [12]

Why we need to use the homography transformation? We can see in the figure 9. A homography transformation can transform a point on a plane to another plane like what camera do, and that is what we want to do for image stitching.

We can multiply a constant k on matrix H, and we can still get the same $(x_2, y_2)$ after we standardized with z. Thus, we fixed the matrix H with $\|H\|^2 = 1$.

By arranging the formula(32), we get

$$\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_2x_1 & -x_2y_1 & -x_2 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y_2x_1 & -y_2y_1 & y_2 \end{pmatrix} \times \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{33}$$

Then solve the $h_{ij}$ by using the characteristic points pairs,

$$\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_2x_1 & -x_2y_1 & -x_2 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y_2x_1 & -y_2y_1 & y_2 \\ x_1^{(2)} & y_1^{(2)} & 1 & 0 & 0 & 0 & -x_2^{(2)}x_1^{(2)} & -x_2^{(2)}y_1^{(2)} & -x_2^{(2)} \\ 0 & 0 & 0 & x_1^{(2)} & y_1^{(2)} & 1 & -y_2^{(2)}x_1^{(2)} & -y_2^{(2)}y_1^{(2)} & y_2^{(2)} \\ x_1^{(3)} & y_1^{(3)} & 1 & 0 & 0 & 0 & -x_2^{(3)}x_1^{(3)} & -x_2^{(3)}y_1^{(3)} & -x_2^{(3)} \\ 0 & 0 & 0 & x_1^{(3)} & y_1^{(3)} & 1 & -y_2^{(3)}x_1^{(3)} & -y_2^{(3)}y_1^{(3)} & y_2^{(3)} \\ ... & & & & & & & & \end{pmatrix} \times \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ ... \end{pmatrix} \tag{34}$$

Now, we simplify it as $Ah = 0$ with $\|h\|^2 = 1$. To solve this question, we use the method called constrained least square. Change the question into solving

$$\min_h \|Ah\|^2 = 0 \tag{35}$$

with $\|h\|^2 = 1$.

And we can rewrite it as

$$\min_h h^T A^T Ah = 0 \tag{36}$$

with $h^T h = 1$.

Define Loss function

$$L(h, \lambda) = h^T A^T Ah - \lambda(h^T h - 1) \tag{37}$$

and we want to find h that minimized L.

Thus, we compute the differential of $L$ w.r.t. $h$, and it becomes

$$2A^T Ah - 2\lambda h = 0 \tag{38}$$

Rewrite the equation as

$$A^T Ah = \lambda h \tag{39}$$

It is a eigenvalue problem of $A^T A$. Thus, we find the eigenvector $h$ which has the smallest eigenvalue $\lambda$. The $h$ is what we looking for.

Finally, we get the homography transformation of these matching points.

Actually, we could use at least 4 pairs to get the matrix H. However, we might pair the wrong characteristic points. Thus, to minimize the error caused by choosing the wrong pair of points. We used RANSAC method we talked before to find the matrix H which can match more pairs of points.

# 8 RANSAC in image stitching

To decrease the error in image stitching, we use RANSAC to avoid using the wrong pairs of characteristic points generated by SIFT. First, choose 4 pairs of points we match before, then we can construct a homogeneous matrix $H_1$. Next, put all the characteristic points in image1 into the transformation and get the new points. Last, calculate the difference between the new points and the points we pair to the original characteristic points. If it is smaller

than a constant $\epsilon$ we choose, we called this pair of points "inliers". Repeated these steps n times(In our code, we run 1000 times), then we can get $H_1, H_2, ..., H_n$ and the number of inliers every transformation has. In the end, we choose the matrix which have the most inliers to be the transformation between image1 and image2.

# 9 Image blending

### Introduction

When we do image blending, if we don't use special method to do image blending, we can clearly to see the ghost in output images. Besides, when the lightness is greatly different in images we want to blend, the image we blend also has a clear boundary on the place where images blend. Thus, we use the following methods to deal with above problems.

Figure 13:

Figure 14:

### Weighted

The ghost happens because the transformation matrix is derived by four pairs of feature points we find in distinct images. This results in that the blending image can't match completely and thus ghost occurs. To fix this problem, we will do weighted to two images. The weighted we defined in overlapping area for pixel $P$ is:
$w_1 = \min\{\text{distance of } P \text{ to photo 1 boundary}\}$, $w_2 = \min\{\text{distance of } P \text{ to photo 2 boundary}\}$.
Which $w_1$ and $w_2$ are weighted to photo 1 and photo 2.

Figure 15:

**Lightness**

For the difference of lightness in two images, we will adjust the lightness of images. First, we let images be gray and calculate $m_i$ the mean of value store in matrix for image i. Second, calculate m the mean of $m_i$, and the ratio $m/m_i$ will multiply to image i to modify the lightness.

Figure 16: Before weighted

Figure 17: After weighted

# References


[1] 張庭榮. Sift 演算法於立體對影像匹配與影像檢索應用之研究, 2008.

[2] David G. Lowe. Distinctive image features from scale-invariant keypoints, 2004.

[3] Pratik Jain. feature point descriptor. https://drive.google.com/file/d/1VP9rAgWhbddkKyYKlK1JbxmPkgbO9ZGN/view.

[4] Summed-area table. https://en.wikipedia.org/wiki/Summed-area_table.

[5] Surf 算法详解. https://reurl.cc/MyeYWW.

[6] Luc Van Gool12 Herbert Bay, Tinne Tuytelaars2. Surf: Speeded up robust features. https://people.ee.ethz.ch/~surf/eccv06.pdf.

[7] Vidhu Chaudhary. Speeded up robust features (surf). https://www.codingninjas.com/studio/library/speeded-up-robust-features-surf.

[8] Deepanshu Tyagi. Introduction to surf (speeded-up robust features). https://medium.com/@deepanshut041/introduction-to-surf-speeded-up-robust-features-c7396d6e7c4e.

[9] tongle.Wang. surf 算法原理-包你明白 surf 过程. https://blog.csdn.net/ecnu18918079120/article/details/78195792.

[10] First Principles of Computer Vision. Dealing with outliers: Ransac. https://youtu.be/J1DwQzab6Jg?list=PL2zRqk16wsdp8KbDfHKvPYNGF2L-zQASc.

[11] Wikipedia. Random sample consensus. https://en.wikipedia.org/wiki/Random_sample_consensus.

[12] Yalda Shankar. homography transformation. https://towardsdatascience.com/estimating-a-homography-matrix-522c70ec4b2c.

[13] First Principles of Computer Vision. Image stitching. https://youtu.be/J1DwQzab6Jg?list=PL2zRqk16wsdp8KbDfHKvPYNGF2L-zQASc.