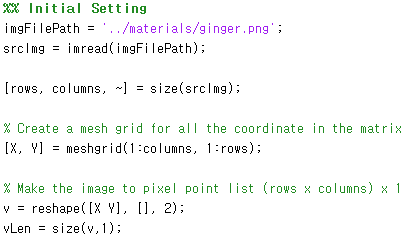
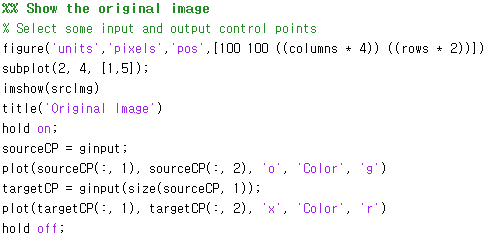
# EXERCISE 2: MOVING LEAST SQUARES FOR IMAGE MANIPULATION

GCT722 MATHEMATICAL METHODS FOR VISUAL COMPUTING  
20183151 Chaelin Kim

## PART 1: iMAGE DEFORMATION USING MOVING LEAST SQUARES

### **Description of implementation**

There are 1 script file & 8 function files for the exercise image deformation.

* **Script file**
* **main.m**  
  : This is the script for doing image deformation. It calls functions for making parameters (calWeight, calStar, calHat), doing several deformations (doAffineDeform, doSimilarityDeform, doRigidDeform) and making deformed image (makeDefImg, makeDefImgBack). Details of these functions are explained below. We can set the initial settings in this script.  
    
    
   To select points in the image, it shows the original image. You can select source control points in the image and press the ‘enter’ key to end it. Then, you have to select target points to match the number of source points.  
    
   After that, functions that used to set parameters (weight, star, hat) and do deformation are called in order. When process of deformations is over, it shows the result images shown next to the original image.
* **Function files**
* **calWeight.m**  
  : This function is used to calculate weight values. The weights have the form  
     
   The denominator of weight is the square value of distance between source control points p and points v in the image. Because the weights in this least squares problem are dependent on the point of evaluation , we call this a *Moving Least Squares* minimization. Therefore, we obtain a different transformation for each .  
  
  + Input Value
    - *v*: The array of data points (x, y) in the original image.  
       (size: (rows \* columns) x 2)
    - *sourceCP*: The array of source control points (x, y)
    - *alpha*: The alpha value used in calculating
  + Output
    - *weight*: The array of weights for each points of original image.
* **calStar.m**  
  : This function is used to calculate weighted centroids of source and target control points.  
  In main.m, this function is called for and .  
  
  + Input Value
    - *weight*: The array of weights for each points of original image
    - *vLength*: The length of data points (x, y) array in the original image
    - *controlPoint*: The array of control points.
  + Output
    - *resultStar*: The array of computed centroid data for each points of original image. (size: (rows \* columns) x 2)
* **calHat.m**  
  : This function is used to calculate hat values of source and target control points. The hat value means the difference of control point and weighted centroid.  
  In main.m, this function is called for and .  
  
  + Input Value
    - *vLength*: The length of data points (x, y) array in the original image
    - *controlPoint*: The array of control points
    - *cpStar*: The array of weighted centroids
  + Output
    - *resultHat*: The array of computed hat values for each points of original image. (size: (rows \* columns) x 2 x (length of control points))
* **doAffineDeform.m**  
  : This function is used to deform the image with affine transformation. Affine transformations contain shear and non-uniform scaling. In the paper [Schaefer et al. 2006], the deformation function can be expressed like:  
   It returns the coordinates array deformed with affine transformation.  
   For backward warping, I implemented the code for it using inverse matrix but it is not used because of accordance with similarity and rigid. In main.m, this function is called same as forward warping except exchanging source and target.  
  
  + Input Value
    - *weight*: The array of weights for each points of original image
    - *v*: The array of data points (x, y) in the original image.  
       (size: (rows \* columns) x 2)
    - *sourceCP*: The array of source control points (x, y)
    - *targetCP*: The array of target control points (x, y)
    - *pstar*: The array of weighted centroids of source control points
    - *phat*: The array of difference of source control points and pstar
    - *qstar*: The array of weighted centroids of target control points
    - *qhat*: The array of difference of target control points and qstar
  + Output
    - *affineDef*: The array of result coordinates (x, y) for deformed image with affine transformation. (size: (rows \* columns) x 2)
* **doSimilarityDeform.m**  
  : This function is used to deform the image with similarity transformation. Similarity transformations are a subset of affine transformations and contain translation, rotation and uniform scaling. In the paper [Schaefer et al. 2006], the deformation function can be expressed like:  
   It returns the coordinates array deformed with similarity transformation.  
   For backward warping, it is called same as forward warping except exchanging source and target in main.m. 
  + Input Value
    - *weight*: The array of weights for each points of original image
    - *v*: The array of data points (x, y) in the original image.  
       (size: (rows \* columns) x 2)
    - *sourceCP*: The array of source control points (x, y)
    - *targetCP*: The array of target control points (x, y)
    - *pstar*: The array of weighted centroids of source control points
    - *phat*: The array of difference of source control points and pstar
    - *qstar*: The array of weighted centroids of target control points
    - *qhat*: The array of difference of target control points and qstar
  + Output
    - *similarityDef*: The array of result coordinates (x, y) for deformed image with similarity transformation. (size: (rows \* columns) x 2)
* **doRigidDeform.m**  
  : This function is used to deform the image with rigid transformation. Rigid transformations are related to similarity transformations and contain translation and rotation. In the paper [Schaefer et al. 2006], the deformation function can be expressed like:  
   It returns the coordinates array deformed with rigid transformation.  
   For backward warping, it is called same as forward warping except exchanging source and target in main.m  
  
  + Input Value
    - *weight*: The array of weights for each points of original image
    - *v*: The array of data points (x, y) in the original image.  
       (size: (rows \* columns) x 2)
    - *sourceCP*: The array of source control points (x, y)
    - *targetCP*: The array of target control points (x, y)
    - *pstar*: The array of weighted centroids of source control points
    - *phat*: The array of difference of source control points and pstar
    - *qstar*: The array of weighted centroids of target control points
    - *qhat*: The array of difference of target control points and qstar
  + Output
    - *rigidDef*: The array of result coordinates (x, y) for deformed image with rigid transformation. (size: (rows \* columns) x 2)
* **makeDefImg.m**  
  : This function makes the deformed image with deformed coordinates for forward warping. Color values at each points in the original image are assigned to the result points in the result image.  
  
  + Input Value
    - *originImg*: The origin image called by imread  
       (size: rows x columns x (number of color channels))
    - *defCoord*: The array of computed coordinates (x, y) for target image. It is resulted from forward warping (affine, similarity or rigid transformation)  
       (size: (rows \* columns) x 2)
  + Output
    - *defImg*: The array of computed centroid data for each points of original image.  
       (size: rows x columns x (number of color channels))
* **makeDefImgBack.m**  
  : This function makes the deformed image with deformed coordinates for backward warping. It is divided into three parts depending on whether or not interpolation is performed and using *griddata* function or not. If the parameter *interpM* is ‘T’, the interpolation is performed using implementation of bilinear interpolation. Bilinear interpolation is a linear interpolation in 2-dimension. It interpolates four points in each of the height and width, and then interpolates two results to obtain the result point. If the result coordinates are out of range, color value in that coordinate is set to white.  
  If the parameter *interpM* is ‘F’, color values at computed points in the original image are assigned to the points in the result image. If *interpM* is ‘G’, the interpolation is performed using *griddata*.  
  
  + Input Value
    - *originImg*: The origin image called by imread.  
       (size: rows x columns x (number of color channels))
    - *bDefCoord*: The array of computed coordinates (x, y) for source image. It is resulted from backward warping (affine, similarity or rigid transformation)  
       (size: (rows \* columns) x 2)
    - *interpM*: The variable that determines whether or not to interpolate. It can have three values, ‘T’, ‘F’ and ‘G’ depending on whether interpolation is used and using griddata or not.  
      (‘T’: use interpolation, ‘F’: not use interpolation, ‘G’: use interpolation with griddata)
  + Output
    - *defImg*: The array of computed centroid data for each points of original image.  
       (size: rows x columns x (number of color channels))

### **Instructions for running**

1. Open the file “main.m” in Matlab.
2. Execute that file.
   1. The window that shows the original image is opened.
3. Select source control points in the original image and if you are done, press the enter key.
4. Select target control points in the original image. You have to select them to match the order of selecting source control points and the number of them.
5. The result images of affine, similarity and rigid deformation is shown next to the original image. Result images are composed of the result of forward warping and backward warping for each deformation method.
   1. Result images are saved in ‘./materials/resultImage’.

### **Screenshots**

* **Forward & Backward warping (with interpolation) (Ginger man)**
* **Forward & Backward warping with another image (make the Pooh happy :D)** ** (Original)  
      
   (Affine) (Similarity) (Rigid)**

### **Discuss the results**

I have implemented the affine, similarity and rigid transformation for forward and backward warping according to the reference paper “Image Deformation Using Moving Least Squares [Schaefer et al. 2006]” with Matlab.

In this code, I made synthesized model with outlier ratios 5% 20%, 30% and 70%, did RANSAC and exhausted search with synthesized data, and draw the plots that show the results.

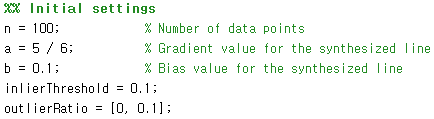
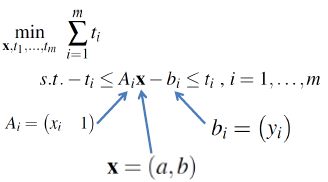
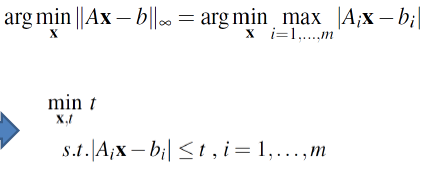
In the histogram, considering with outlier ratios, RANSAC finds the number of inliers similar to real number of inliers. In the plots, RANSAC with re-applying 1000 times finds quite fitted circle model in case of all outlier ratios.

Exhaustive searching finds the best fitted model because it considers all combinations of data points. In that point, if the number of data increases, the computational cost exponentially increases. On the contrary, RANSAC is not affected with the number of data. So, if other conditions are fixed and the number of data increases, RANSAC finds a fitted model faster than exhaustive search. But if iteration for re-applying is lower and outlier ratio is higher, RANSAC would be hard to find well-fitted model.

## PART 2: IRLS AND NORMS FOR LINE FITTING

### **Description of implementation**

There are 1 script file & 4 function files for the exercise RANSAC.

* **Script file**
* **main\_LineFitting.m**  
  : This is the script for doing line fitting using norm (employing IRLS and LP) and  norm (employing LP). It calls functions for line fitting (genLineData, doIRLS, doLP, drawLineFittingPlot). Details of these functions are explained below. We can set the initial settings in this script.  
  
* **Function files**
* **genLineData.m**  
  : This function generates data points on a synthesized line with inliers and outliers. The number of inliers and outliers are decided by the given outlier ratio. The inlier data is made with random noise (between -0.1 and 0.1). In the part of outlier generation, *while* loop is executed until the number of made outlier data meets the given number of outliers.  
    
  
  + Input Value
    - *n*: The number of data points
    - *a*: The gradient value of the synthesized line
    - *b*: The bias value of the synthesized line
    - *inlierThreshold*: The inlier distance threshold
    - *outlierRatio*: The ratio of outliers in data
  + Output
    - *data*: The data in the form of line with inliers and outliers according to the outlier ratio.
* **doIRLS.m**  
  : This function runs IRLS with for the given data. For IRLS, we change norm to weight \* norm for easy computation. norm is computed with setting the gradient to 0. The weight is computed with the value of gradient and bias calculated by norm. With this weight, the result is recomputed. This process is iterated until the absolute value of difference between previous value and current value is smaller than 0.0001.  
    
  
  + Input Value
    - *data*: The data made from *genLineData* function (type: matrix)
  + Output
    - *result\_IRLS*: The result data [a, b] computed by IRLS
* **doLP.m**  
  : This function runs Linear Programming(LP) with  and  for the given data. Linear programming solver finds the minimum of a problem. The purpose of is minimize the sum of the absolute value of all errors and the purpose of is minimize the maximum error among the absolute value of errors. In order to solve this problem with LP, I made the matrix A and b that consist of linear equations with given data. Using *linprog()* function, the result is easily computed.  
  `   
  🡪 for LP(left) and for LP(right)

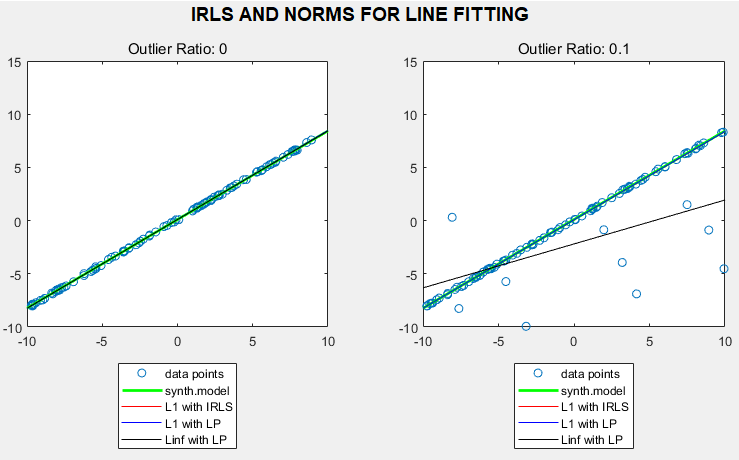


* + Input Value
    - *data*: The data made from *genLineData* function (type: matrix)
    - *Lnorm*: The kind of Lp norm (type: String)
  + Output
    - result\_LP. The result data computed by Linear programming
* **drawLineFittingPlot.m**  
  : This function draws plots of line fitting results(IRLS with and LP with and ) with different outlier ratios. The result is shown in category *Screenshots* below.  
    
  
  + Input Value
    - *a*: The gradient value of the synthesized line
    - *b*: The bias value of the synthesized line
    - *data*: All data made from *genLineData* function (type: cell)
    - *result*: The results of IRLS with made from *doIRLS* function and LP with and made from *doLP* function (type: cell)
    - *outlierRatio*: The ratio of outliers in data for draw outlier data
  + Output
    - No output. It shows result plots.

### **Instructions for running**

1. Open the file “main\_LineFitting.m” in Matlab.
2. Execute that file
   1. The another window that shows plots of IRLS with and LP with  and  norms results is opened.

### **Screenshots**



### **Discuss the results**

In this code, I made synthesized model with outlier ratios 0% and 10%, doing line fitting using  norm (employing IRLS and LP) and  norm (employing LP) with synthesized data, and draw the plots that show the results.

In case of outlier ratio 0%, all methods show similar results each other and also almost match with the synthesized model. In case of outlier ratio 10%, however, the results of norm still similar to the synthesized model but there is quite difference between the result of norm and synthesized model. The purpose of norm is minimize the maximum error among all errors, so it is not robust to outliers. The results of norm with IRLS and LP are almost same but I think LP is more intuitive than IRLS