ECE661 Computer Vision : HW5

Dae Woo Kim

October 28, 2010

1 Problem

In the last homework, the DLT algorithm was used to find the homography which minimized an algebraic cost function. In this homework, we wish to find the homography H which minimizes the geometric cost function given by Equation 4.6 in the textbook (pg. 94)

$$C(\mathrm{H}) = \sum_i d(\mathbf{x}_i', \mathrm{H}\mathbf{x}_i)^2 \tag{1}$$

where $d()$ is Euclidean distance.

Equation (1) can be rewritten as a nonlinear least squares problem with the form

$$C(\mathbf{p}) = \|\mathbf{X} - F(\mathbf{p})\|^2 \tag{2}$$

where $\mathbf{X}$ is a set of data points and $F(\mathbf{p})$ is a nonlinear function with parameters $\mathbf{p}$. Note that the minimization is carried out only over the parameters $\mathbf{p}$; the data points and nonlinear function are fixed. The first part of the problem is to determine what $\mathbf{X}$, $F(\mathbf{p})$, and $\mathbf{p}$ should be in Equation (2) in order to give the same cost as Equation (1). Once the cost function is in the form of Equation (2), there are several iterative methods available to minimize it. In this homework you will try 2 standard methods, gradient descent and Gauss-Newton, as well as two hybrid methods, Levenberg-Marquardt and dog leg.

2 Nonlinear least problem

$d(\mathbf{x}_i^{'}, \mathrm{H}\mathbf{x}_i)^2$is the Euclidean image distance in the second image between the measured point $\mathbf{x}_i^{'} = (x_i^{'}, y_i^{'})^{\mathrm{T}}$ and the point $\mathrm{H}\mathbf{x}_i$ at which the the corresponding point $\mathbf{x}_i = (x_i, y_i)^{\mathrm{T}}$ is mapped from the first image. So if we let homography $\mathrm{H} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix}$, then

$$\mathrm{H}\mathbf{x}_i = \left(\frac{h_1 x_i + h_2 y_i + h_3}{h_7 x_i + h_8 y_i + h_9}, \frac{h_4 x_i + h_5 y_i + h_6}{h_7 x_i + h_8 y_i + h_9} \right)^{\mathrm{T}} \tag{3}$$

and

$$d(\mathbf{x}_i^{'}, \mathrm{H}\mathbf{x}_i)^2 = \left(x_i^{'} - \frac{h_1 x_i + h_2 y_i + h_3}{h_7 x_i + h_8 y_i + h_9} \right)^2 + \left(y_i^{'} - \frac{h_4 x_i + h_5 y_i + h_6}{h_7 x_i + h_8 y_i + h_9} \right)^2 \tag{4}$$

For all corresponding points $(\mathbf{x}_i^{'}, \mathbf{x}_i)$,where $i = 1, 2, \cdots, n$ and n is the number of corresponding points, let $\mathbf{X}^{'} = \begin{bmatrix} \mathbf{x}_1^{'} \\ \mathbf{x}_2^{'} \\ \vdots \\ \mathbf{x}_n^{'} \end{bmatrix}$ and $\mathrm{H}\mathbf{X} = \begin{bmatrix} \mathrm{H}\mathbf{x}_1 \\ \mathrm{H}\mathbf{x}_2 \\ \vdots \\ \mathrm{H}\mathbf{x}_n \end{bmatrix}$, then cost function can be expressed like Equation (5)

$$C(\mathrm{H}) = \sum_i d(\mathbf{x}_i^{'}, \mathrm{H}\mathbf{x}_i)^2 = \left\| \mathbf{X}^{'} - \mathrm{H}\mathbf{X} \right\|^2 \tag{5}$$

We can determine homography H by minimizing this nonlinear cost function with parameter H. To solve this minimization problem I will use and compare 4 kinds of method: gradient descent, Gauss-Newton, Levenberg-Marquardt and dog leg.

3 Implementation of the 4 iterative solving approaches

If we let $\mathbf{h} = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_9 \end{bmatrix}$ and $F(\mathbf{h}) = \mathrm{H}\mathbf{X}$, then the error vector $r(\mathbf{h})$is given by

$$r(\mathbf{h}) = \mathbf{X}^{'} - \mathrm{H}\mathbf{X} = \mathbf{X}^{'} - F(\mathbf{h}) =$$

$$\begin{bmatrix} x_1^{'} - \frac{h_1x_1+h_2y_1+h_3}{h_7x_1+h_8y_1+h_9} \\ y_1^{'} - \frac{h_4x_1+h_5y_1+h_6}{h_7x_1+h_8y_1+h_9} \\ x_2^{'} - \frac{h_1x_2+h_2y_2+h_3}{h_7x_2+h_8y_2+h_9} \\ y_2^{'} - \frac{h_4x_2+h_5y_2+h_6}{h_7x_2+h_8y_2+h_9} \\ \vdots \\ x_n^{'} - \frac{h_1x_n+h_2y_n+h_3}{h_7x_n+h_8y_n+h_9} \\ y_n^{'} - \frac{h_4x_n+h_5y_n+h_6}{h_7x_n+h_8y_n+h_9} \end{bmatrix} \tag{6}$$

Note that $C(\mathrm{H}) = C(\mathbf{h}) = r(\mathbf{h})^{\mathrm{T}} r(\mathbf{h})$.

The gradient of function $C(\mathbf{h})$ become

$$\nabla C(\mathbf{h}) = 2J_r(\mathbf{h})^{\mathrm{T}} r(\mathbf{h}) \tag{7}$$

where

$$J_r(\mathbf{h}) = -J_F(\mathbf{h}) =$$

$$\begin{bmatrix} \frac{1}{h_7x_1+h_8y_1+h_9} \\ \frac{1}{h_7x_1+h_8y_1+h_9} \\ \frac{1}{h_7x_2+h_8y_2+h_9} \\ \frac{1}{h_7x_2+h_8y_2+h_9} \\ \vdots \\ \frac{1}{h_7x_n+h_8y_n+h_9} \\ \frac{1}{h_7x_n+h_8y_n+h_9} \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & \frac{(h_1x_1+h_2y_1+h_3)x_1}{h_7x_1+h_8y_1+h_9} & \frac{(h_1x_1+h_2y_1+h_3)y_1}{h_7x_1+h_8y_1+h_9} & \frac{h_1x_1+h_2y_1+h_3}{h_7x_1+h_8y_1+h_9} \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & \frac{(h_4x_1+h_5y_1+h_6)x_1}{h_7x_1+h_8y_1+h_9} & \frac{(h_4x_1+h_5y_1+h_6)y_1}{h_7x_1+h_8y_1+h_9} & \frac{h_4x_1+h_5y_1+h_6}{h_7x_1+h_8y_1+h_9} \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & \frac{(h_1x_2+h_2y_2+h_3)x_2}{h_7x_2+h_8y_2+h_9} & \frac{(h_1x_2+h_2y_2+h_3)y_2}{h_7x_2+h_8y_2+h_9} & \frac{h_1x_2+h_2y_2+h_3}{h_7x_2+h_8y_2+h_9} \\ 0 & 0 & 0 & -x_x & -y_2 & -1 & \frac{(h_4x_2+h_5y_2+h_6)x_2}{h_7x_2+h_8y_2+h_9} & \frac{(h_4x_2+h_5y_2+h_6)y_2}{h_7x_2+h_8y_2+h_9} & \frac{h_4x_2+h_5y_2+h_6}{h_7x_2+h_8y_2+h_9} \\ & & & & \vdots & & & & \\ -x_n & -y_n & -1 & 0 & 0 & 0 & \frac{(h_1x_n+h_2y_n+h_3)x_n}{h_7x_n+h_8y_n+h_9} & \frac{(h_1x_n+h_2y_n+h_3)y_n}{h_7x_n+h_8y_n+h_9} & \frac{h_1x_n+h_2y_n+h_3}{h_7x_n+h_8y_n+h_9} \\ 0 & 0 & 0 & -x_n & -y_n & -1 & \frac{(h_4x_n+h_5y_n+h_6)x_n}{h_7x_n+h_8y_n+h_9} & \frac{(h_4x_n+h_5y_n+h_6)y_n}{h_7x_n+h_8y_n+h_9} & \frac{h_4x_n+h_5y_n+h_6}{h_7x_n+h_8y_n+h_9} \end{bmatrix} \tag{8}$$

If we let $\hat{x}_i = h_1x_i + h_2y_i + h_3$, $\hat{y}_i = h_4x_i + h_5y_i + h_6$ and $\hat{w}_i = h_7x_i + h_8y_i + h_9$ then

$$r(\mathbf{h}) = \begin{bmatrix} x_1^{'} - \frac{\hat{x}_1}{\hat{w}_1} \\ y_1^{'} - \frac{\hat{y}_1}{\hat{w}_1} \\ x_2^{'} - \frac{\hat{x}_2}{\hat{w}_2} \\ y_2^{'} - \frac{\hat{y}_2}{\hat{w}_2} \\ \vdots \\ x_n^{'} - \frac{\hat{x}_n}{\hat{w}_n} \\ y_n^{'} - \frac{\hat{y}_n}{\hat{w}_n} \end{bmatrix} \tag{9}$$

and

$$J_r(\mathbf{h}) = \begin{bmatrix} -\frac{x_1}{\hat{w}_1} & -\frac{y_1}{\hat{w}_1} & -\frac{1}{\hat{w}_1} & 0 & 0 & 0 & \frac{\hat{x}_1x_1}{\hat{w}_1^2} & \frac{\hat{x}_1y_1}{\hat{w}_1^2} & \frac{\hat{x}_1}{\hat{w}_1^2} \\ 0 & 0 & 0 & -\frac{x_1}{\hat{w}_1} & -\frac{y_1}{\hat{w}_1} & -\frac{1}{\hat{w}_1} & \frac{\hat{y}_1x_1}{\hat{w}_1^2} & \frac{\hat{y}_1y_1}{\hat{w}_1^2} & \frac{\hat{y}_1}{\hat{w}_1^2} \\ -\frac{x_2}{\hat{w}_2} & -\frac{y_2}{\hat{w}_2} & -\frac{1}{\hat{w}_2} & 0 & 0 & 0 & \frac{\hat{x}_2x_2}{\hat{w}_2^2} & \frac{\hat{x}_2y_2}{\hat{w}_2^2} & \frac{\hat{x}_2}{\hat{w}_2^2} \\ 0 & 0 & 0 & -\frac{x_2}{\hat{w}_2} & -\frac{y_2}{\hat{w}_2} & -\frac{1}{\hat{w}_2} & \frac{\hat{y}_2x_2}{\hat{w}_2^2} & \frac{\hat{y}_2y_2}{\hat{w}_2^2} & \frac{\hat{y}_2}{\hat{w}_2^2} \\ & & & & \vdots & & & & \\ -\frac{x_n}{\hat{w}_n} & -\frac{y_n}{\hat{w}_n} & -\frac{1}{\hat{w}_n} & 0 & 0 & 0 & \frac{\hat{x}_nx_n}{\hat{w}_n^2} & \frac{\hat{x}_ny_n}{\hat{w}_n^2} & \frac{\hat{x}_n}{\hat{w}_n^2} \\ 0 & 0 & 0 & -\frac{x_n}{\hat{w}_n} & -\frac{y_n}{\hat{w}_n} & -\frac{1}{\hat{w}_n} & \frac{\hat{y}_nx_n}{\hat{w}_n^2} & \frac{\hat{y}_ny_n}{\hat{w}_n^2} & \frac{\hat{y}_n}{\hat{w}_n^2} \end{bmatrix} \tag{10}$$

3.1 Gradient Descent (GD)

The gradient of a function indicates the direction of maximum increase (i.e. greatest positive slope). In the GD method, we iterativelymove in the opposite direction of the gradient in small steps until reaching the minimum. At the kth iteration we compute a new set of parameters as follows

$$\mathbf{h}_{k+1} = \mathbf{h}_k - \alpha_k \nabla C(\mathbf{h}_k) = \mathbf{h}_k - 2\alpha_k J_r(\mathbf{h_k})^{\mathrm{T}} r(\mathbf{h}_k) \tag{11}$$

The step size $k > 0$ is used to control the rate of convergence and can optionally be changed for every iteration. The algorithm terminates either when a maximum number of iterations has been reached or when

$$\|\mathbf{h}_{k+1} - \mathbf{h}_k\| < \epsilon \tag{12}$$

for some small ϵ. Free parameters are α_k and ϵ.

3.2 Gauss-Newton (GN)

The Gauss-Newton method takes the approximation one step farther by using an approximation of the 2nd derivative of $r(\mathbf{h})$ (which is called the Hessian) based on the Jacobian. At the kth iteration, we compute a new set of parameters as follows:

$$\mathbf{h}_{k+1} = \mathbf{h}_k - \left(J_r(\mathbf{h_k})^{\mathrm{T}} J_r(\mathbf{h_k})\right)^{-1} J_r(\mathbf{h_k})^{\mathrm{T}} r(\mathbf{h}_k) \tag{13}$$

The criteria for stopping is the same as in the GD method. Free parameter is ϵ.

3.3 Levenberg-Marquardt (LM)

The Levenberg-Marquardt algorithm is a hybrid of the GD and GN methods which attempts to combine the advantages of both approaches. At the kth iteration, we compute a new set of parameters as follows

$$\mathbf{h}_{k+1} = \mathbf{h}_k - \left(J_r(\mathbf{h_k})^{\mathrm{T}} J_r(\mathbf{h_k}) + \mu_k \mathbf{I}\right)^{-1} J_r(\mathbf{h_k})^{\mathrm{T}} r(\mathbf{h}_k) \tag{14}$$

For the rst iteration we use a heuristic to choose a value, for example

$$\mu_0 = \tau \times \max\left(\mathrm{diag}\left(J_r(\mathbf{h_0})^{\mathrm{T}} J_r(\mathbf{h_0})\right)\right) \tag{15}$$

where $0 < \tau \ll 1$. Then, after each iteration we update the value for the next iteration. We first compute the ratio of the actual reduction in the cost function to the expected reduction

$$\rho_{LM} = \frac{C(\mathbf{h}_k) - C(\mathbf{h}_{k+1})}{\delta_{\mathbf{h}}^T \left(\mu_k \delta_{\mathbf{h}} - J_r(\mathbf{h_k})^{\mathrm{T}} r(\mathbf{h}_k)\right)} \tag{16}$$

where

$$\delta_{\mathbf{h}} = \mathbf{h}_{k+1} - \mathbf{h}_k \tag{17}$$

If $\rho_{LM} > 0$

$$\mu_{k+1} = \mu_k \times \max\left(\frac{1}{3}, 1 - (2\rho_{LM} - 1)^3\right) \tag{18}$$

If $\rho_{LM} \leqq 0$ then this iteration actually increased the cost instead of decreasing it. We first undo the damage by setting $\mathbf{h}_{k+1} = \mathbf{h}_k$. We then set

$$\mu_{k+1} = 2\mu_k \tag{19}$$

The criteria for stopping is the same as in the GD method. Free parameters are τ and ϵ.

3.4 Dog Leg (DL)

Like the LM method, the dog leg method is a hybrid approach combining GD and GN. At each iteration we first compute two possible values for the parameter vector, one based on GD and the other based on GN

$$\mathbf{h}_{GD} = \mathbf{h}_k - \left(\frac{\left\| J_r(\mathbf{h_k})^{\mathrm{T}} r(\mathbf{h}_k) \right\|^2}{\left\| J_r(\mathbf{h_k}) J_r(\mathbf{h_k})^{\mathrm{T}} r(\mathbf{h}_k) \right\|^2}\right) J_r(\mathbf{h_k})^{\mathrm{T}} r(\mathbf{h}_k) \tag{20}$$

$$\mathbf{h}_{GN} = \mathbf{h}_k - \left(J_r(\mathbf{h_k})^{\mathrm{T}} J_r(\mathbf{h_k})\right)^{-1} J_r(\mathbf{h_k})^{\mathrm{T}} r(\mathbf{h}_k) \tag{21}$$

The actual value which is used for $\mathbf{h}_{k+1}$ depends on the size of the trust region. Let $\delta_{\mathbf{h},GD} = \mathbf{h}_{GD} - \mathbf{h}_k$ and $\delta_{\mathbf{h},GN} = \mathbf{h}_{GN} - \mathbf{h}_k$. Then for the given Δ_k we set $\mathbf{h}_{k+1}$ as follows:

$$\mathbf{h}_{k+1} = \mathbf{h}_k + \begin{cases} \delta_{\mathbf{h},GN} & \|\delta_{\mathbf{h},GN}\| \leqq \Delta_k \\ \delta_{\mathbf{h},GD} + \beta(\delta_{\mathbf{h},GN} - \delta_{\mathbf{h},GD}) & \|\delta_{\mathbf{h},GD}\| < \Delta_k \leq \|\delta_{\mathbf{h},GN}\| \\ \frac{\Delta_k}{\|\delta_{\mathbf{h},GD}\|}\delta_{\mathbf{h},GD} & \text{otherwise} \end{cases} \tag{22}$$

where

$$\beta = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{23}$$

where

$$\begin{aligned} a &= \|\delta_{\mathbf{h},GN} - \delta_{\mathbf{h},GD}\|^2 \\ b &= 2\delta_{\mathbf{h},GD}^{\mathrm{T}} (\delta_{\mathbf{h},GN} - \delta_{\mathbf{h},GD}) \\ c &= \|\delta_{\mathbf{h},GD}\|^2 - \Delta_k^2 \end{aligned}$$

The final issue is how to set Δ_k for each iteration. we start with an initial value for the first iteration (try $\Delta_0 = 1$ initially) and then update it after each iteration. To update Δ_k we first compute the ratio of the actual reduction in the cost function to the expected reduction

$$\rho_{DL} = \frac{C(\mathbf{h}_k) - C(\mathbf{h}_{k+1})}{-\delta_{\mathbf{h}}^T J_r(\mathbf{h_k})^{\mathrm{T}} J_r(\mathbf{h}_k)\delta_{\mathbf{h}} - 2r(\mathbf{h_k})^{\mathrm{T}} J_r(\mathbf{h}_k)\delta_{\mathbf{h}}} \tag{24}$$

where $\delta_{\mathbf{h}}$is same as Equation (17). Δ_{k+1} is then given by

$$\mathbf{\Delta}_{k+1} = \begin{cases} \frac{1}{4}\Delta_k & \rho_{DL} \leqq \frac{1}{4} \\ \Delta_k & \frac{1}{4} < \rho_{DL} \leq \frac{3}{4} \\ 2\Delta_k & \text{otherwise} \end{cases} \tag{25}$$

As with LM, if $\rho_{DL} \leqq 0$ then the cost actually increased for this iteration and we undo the damage by setting $\mathbf{h}_{k+1} = \mathbf{h}_k$

The criteria for stopping is the same as in the GD method. Free parameters are Δ_0 and ϵ.

4 Results

The performance comparison of DLT and four optimization method is in Table. 1. The error ϵ used as a stop criterion is $1e-7$ equally. DL method showed best performance in final estimation quality but convergence rate is too slow compare to other method. And LM method showed fast convergence rate and the final estimation quality was also good. In bad initial condition problem, GN and DL method showed good performance. LM method showed some error but it's quality is better than GD method. Final reconstucted image and x' and Hx plot are shown from Fig. 1 to Fig. 30.

	DLT	GD	GN	LM	DL
Iteration #	X	1	33	1	48
Execution time	614us	674us	10.58ms	705us	14ms
Correspondence error ($\|\|r(\mathbf{h})\|\|$)	8.0	101.9068	7.8103	8.1240	7.8102
Bad init condition covergence	X	Bad (3127)	Best (7.8)	Good (87.1)	Best (7.8)
Free parameter (except ϵ)	X	$\alpha_0 = 1e-11$ $\alpha_{k+1} = 0.01 \cdot \alpha_k$	NONE	$\tau = 1e-5$	$\Delta_0 = 1$
Effect of parameter	X	Very sensitive	X		

Table. 1

(1) Sample.jpg : DLT

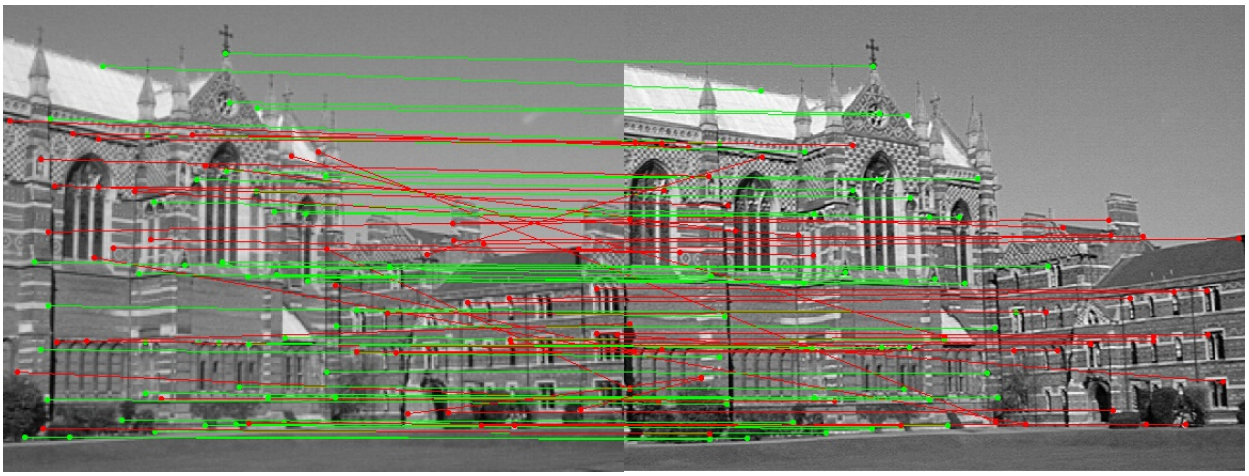

fig 1. Weighted RANSAC Result

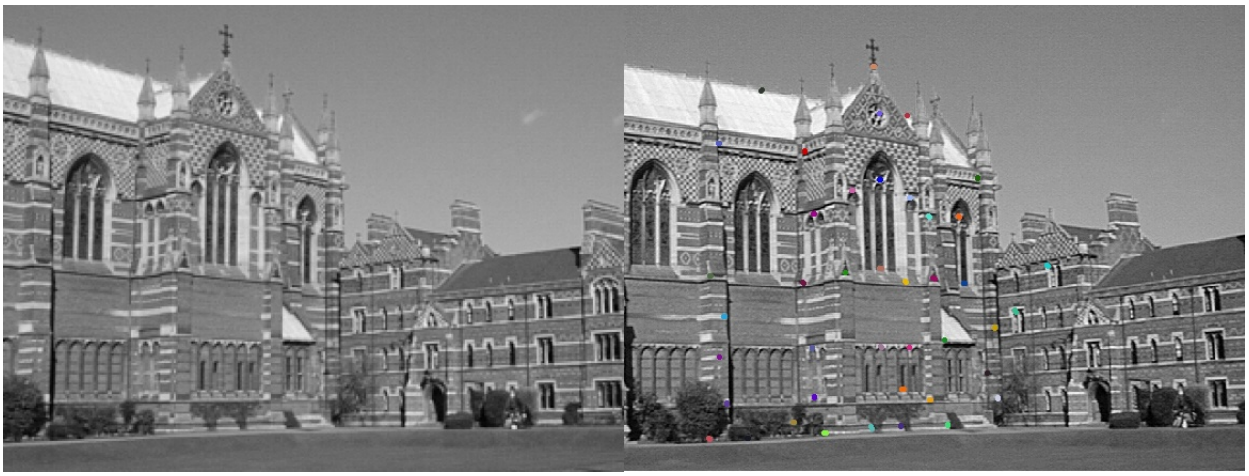

fig 2. x' vs Hx plot

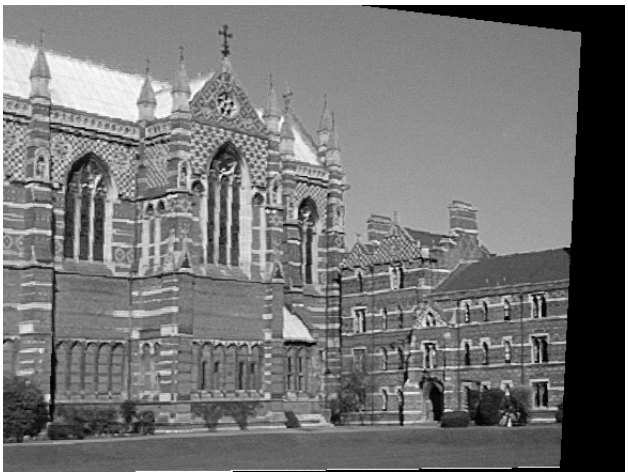

fig 3. Reconstructed Image

(2) Sample.jpg : GD Algorithm (Good Initial Condition)

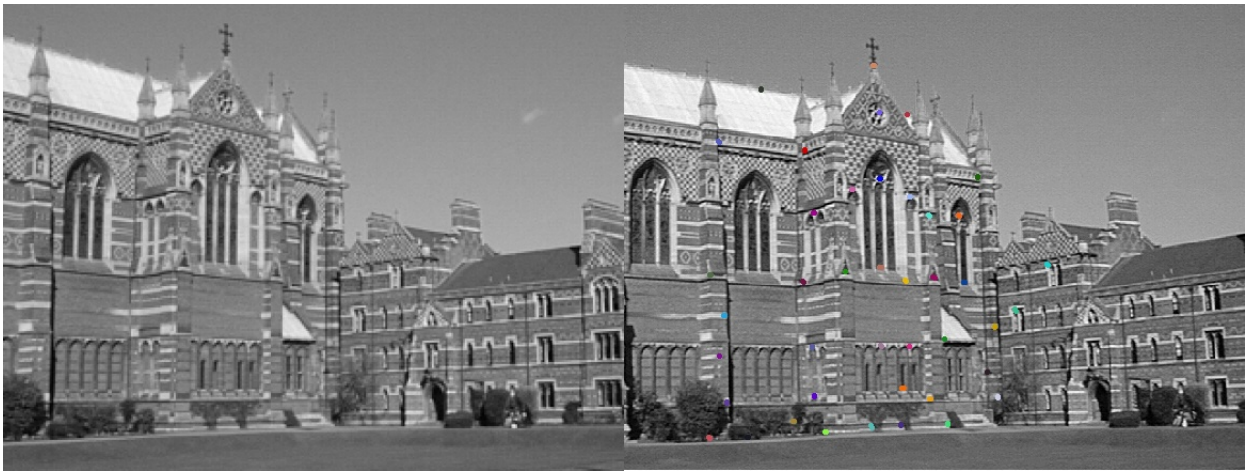

fig 4. x' vs Hx plot (Good Initial Condition)

fig 5. Reconstructed Image (Good Initial Condition)

(3) Sample.jpg : GD Algorithm (Bad Initial Condition)

fig 6. x' vs Hx plot (Bad Initial Condition)

fig 7. Reconstructed Image (Bad Initial Condition)

(4) Sample.jpg : GN Algorithm (Good Initial Condition)

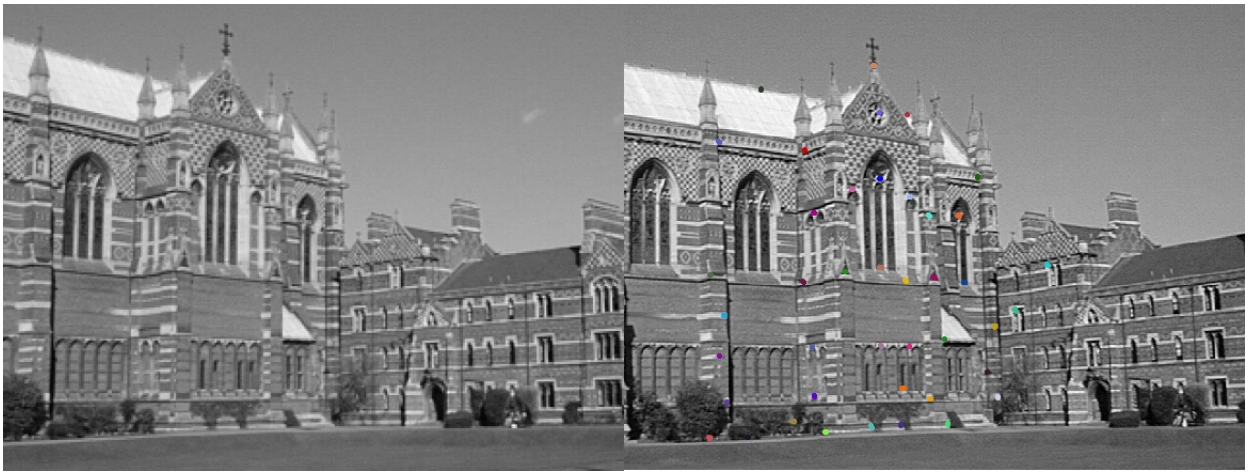

fig 8. x' vs Hx plot (Good Initial Condition)

fig 9. Reconstructed Image (Good Initial Condition)

(5) Sample.jpg : GN Algorithm (Bad Initial Condition)

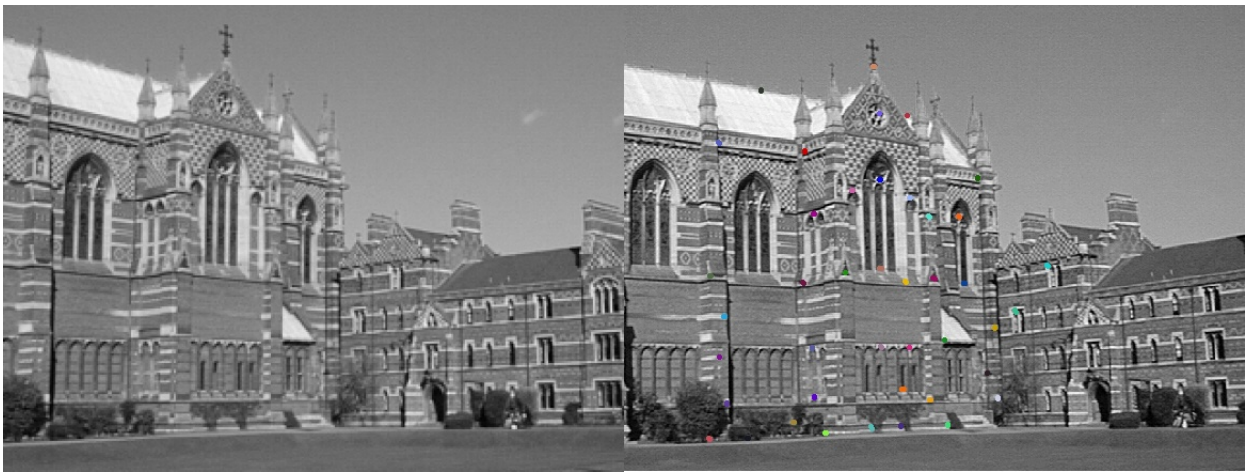

fig 10. x' vs Hx plot (Bad Initial Condition)

fig 11. Reconstructed Image (Bad Initial Condition)

(6) Sample.jpg : LM Algorithm (Good Initial Condition)

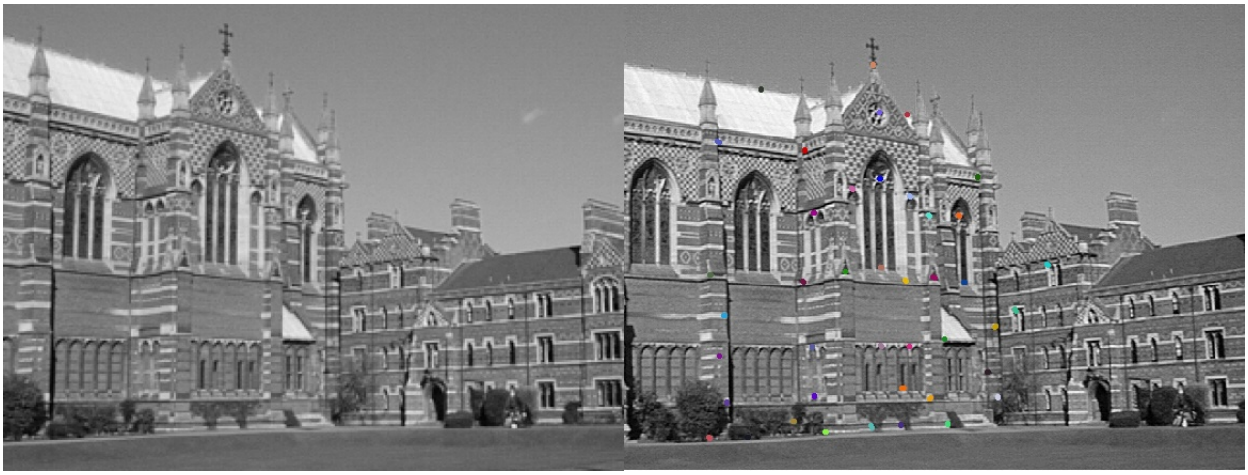

fig 12. x' vs Hx plot (Good Initial Condition)

fig 13. Reconstructed Image (Good Initial Condition)

(7) Sample.jpg : LM Algorithm (Bad Initial Condition)

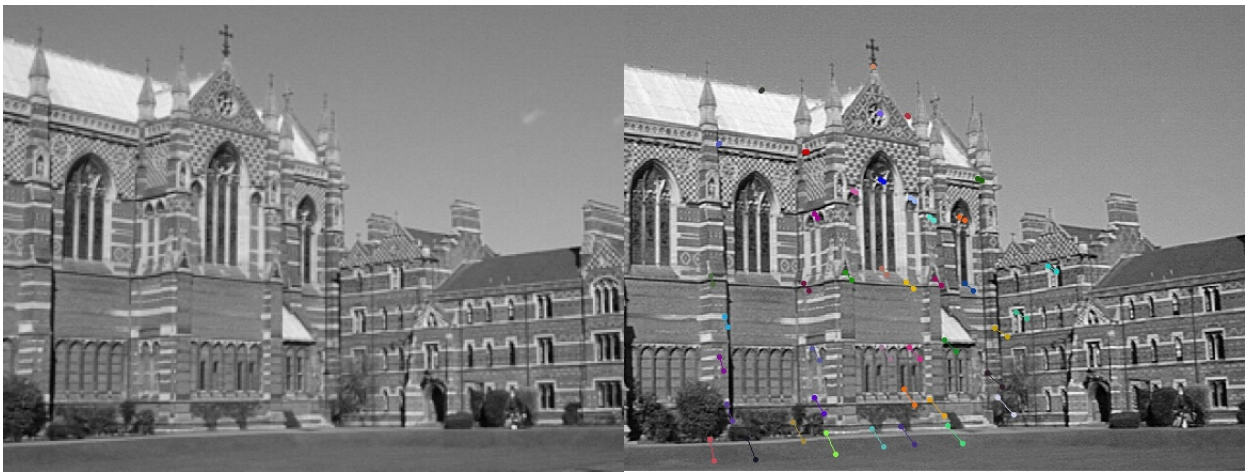

fig 14. x' vs Hx plot (Bad Initial Condition)

fig 15. Reconstructed Image (Bad Initial Condition)

(8) Sample.jpg : DL Algorithm (Good Initial Condition)

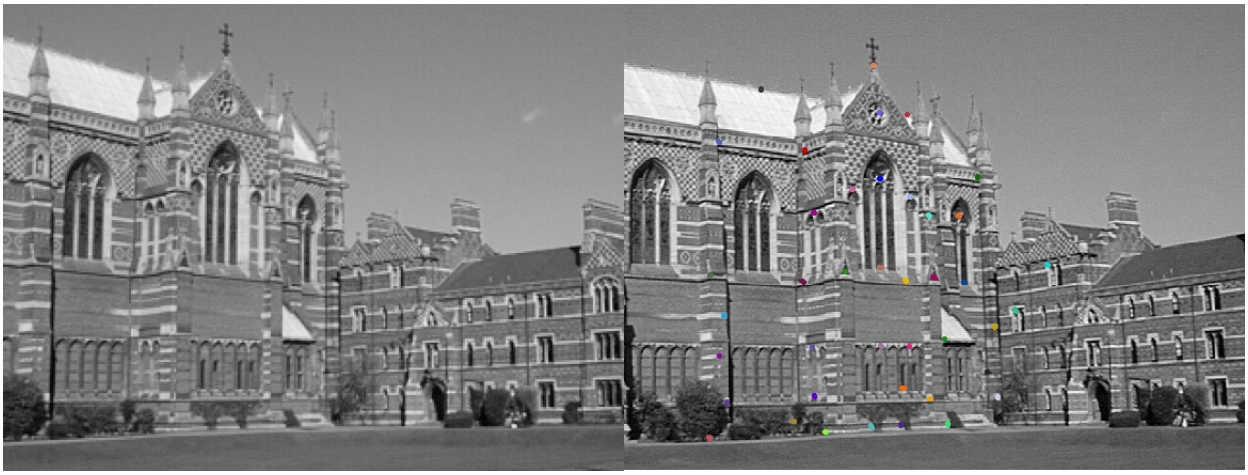

fig 16. x' vs Hx plot (Good Initial Condition)

fig 17. Reconstructed Image (Good Initial Condition)

(9) Sample.jpg : GD Algorithm (Bad Initial Condition)

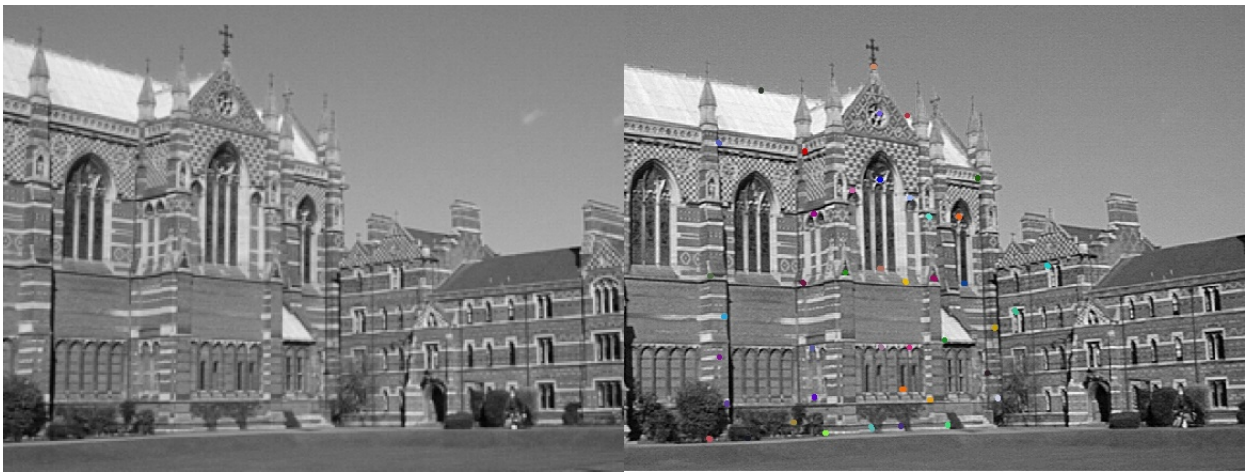

fig 18. x' vs Hx plot (Bad Initial Condition)

fig 19. Reconstructed Image (Bad Initial Condition)

(10) book.jpg : DLT

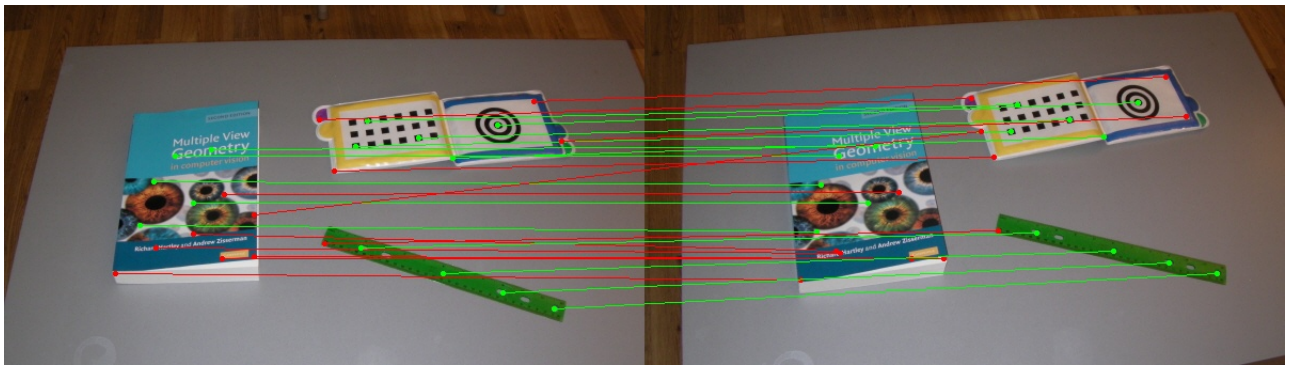

fig 20. Weighted RANSAC Result

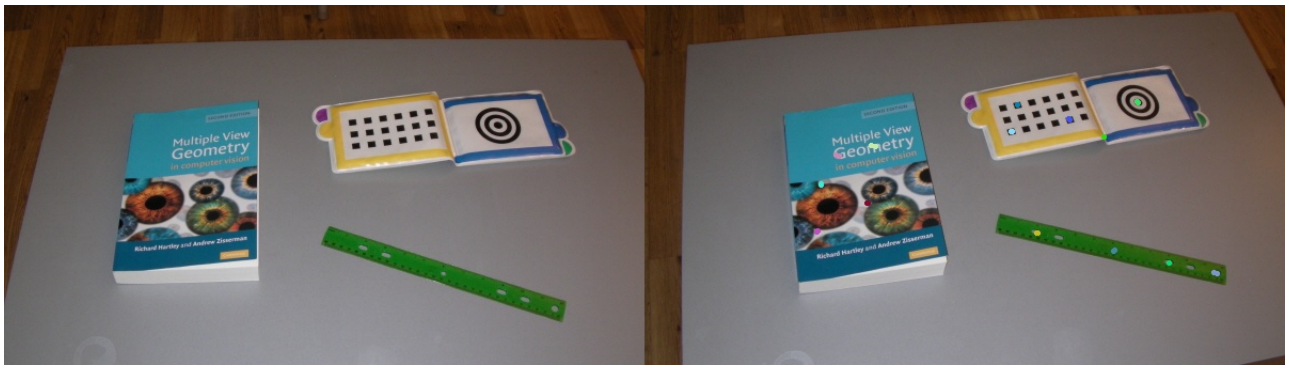

fig 21. x' vs Hx plot

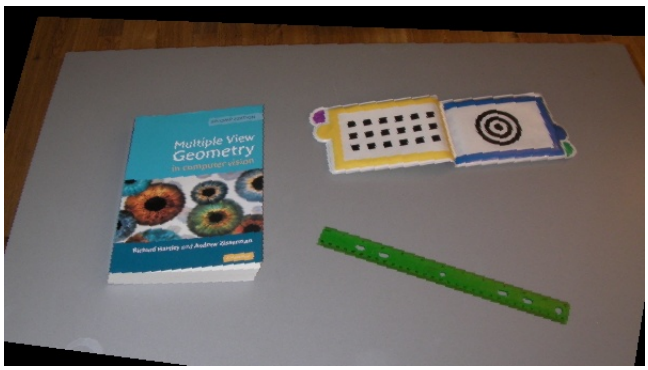

fig 22. Reconstructed Image

(11) book.jpg : GD Algorithm (Good Initial Condition)

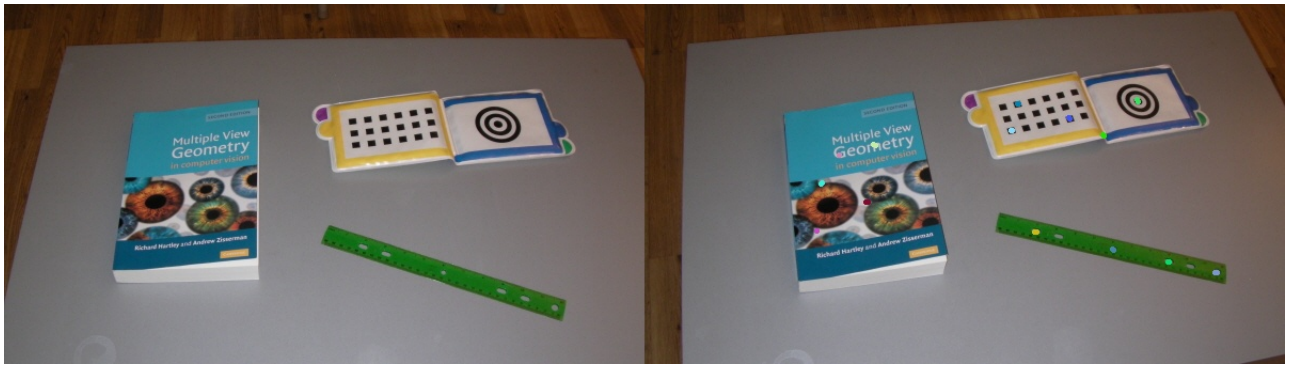

fig 23. x' vs Hx plot (Good Initial Condition)

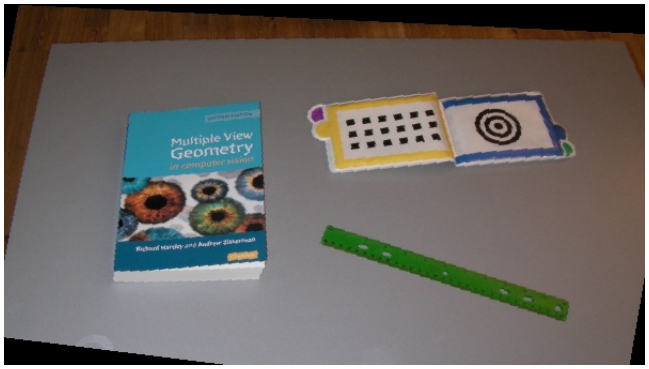

fig 24. Reconstructed Image (Good Initial Condition)

(12) book.jpg : GN Algorithm (Good Initial Condition)

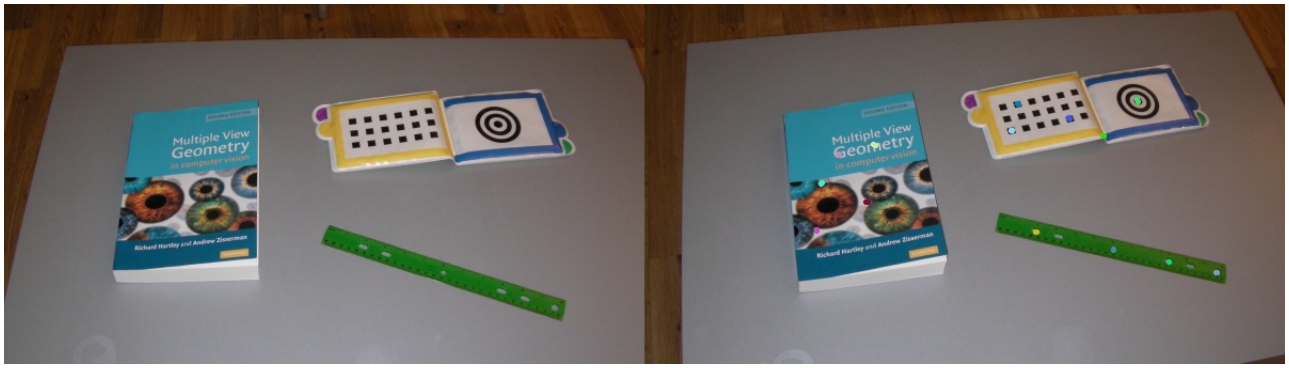

fig 25. x' vs Hx plot (Good Initial Condition)

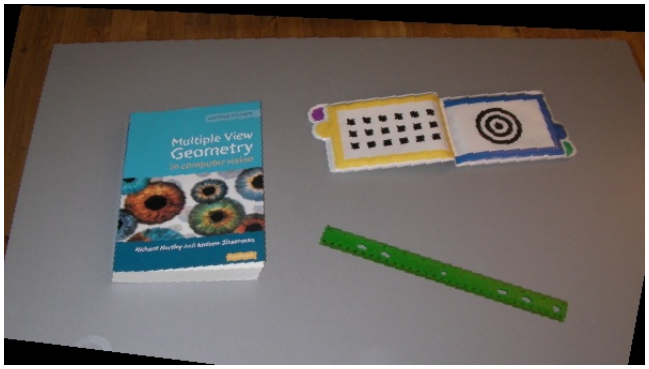

fig 26. Reconstructed Image (Good Initial Condition)

(13) book.jpg : LM Algorithm (Good Initial Condition)

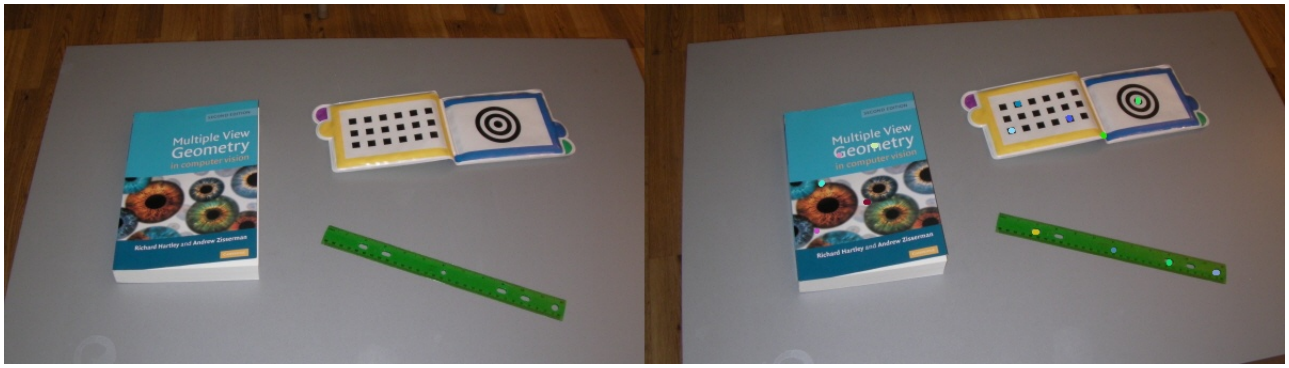

fig 27. x' vs Hx plot (Good Initial Condition)

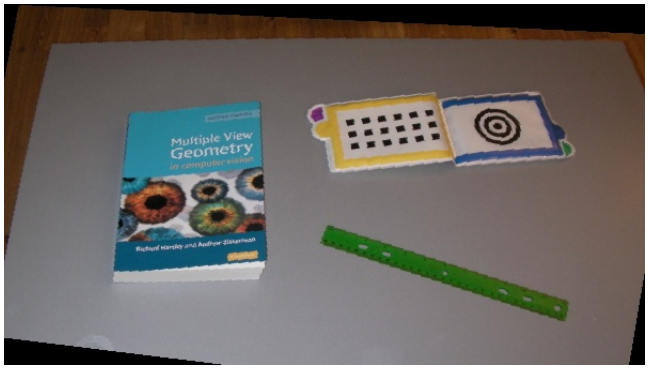

fig 28. Reconstructed Image (Good Initial Condition)

(14) book.jpg : DL Algorithm (Good Initial Condition)

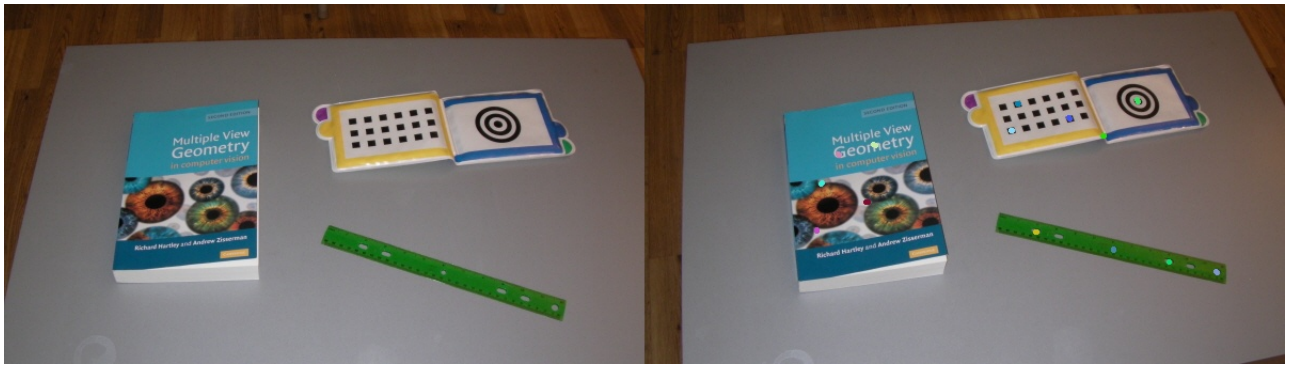

fig 29. x' vs Hx plot (Good Initial Condition)

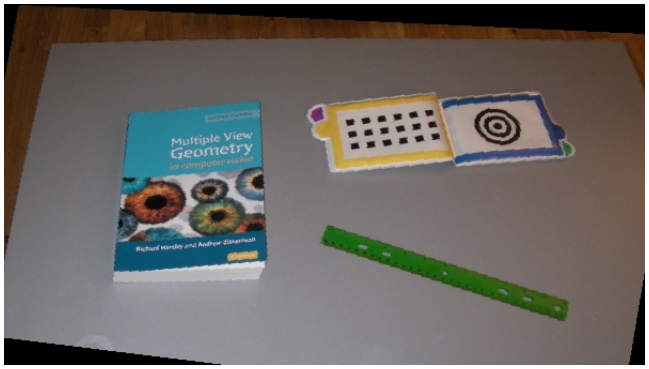

fig 30. Reconstructed Image (Good Initial Condition)

5. Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cv.h"
#include "cxcore.h"
#include "highgui.h"
// For check program execution time check
#include <windows.h>
#include <time.h>

#define CLIP2(minv, maxv, value) (min(maxv, max(minv, value)))

#define WINSIZE_CORNERFIND      3
#define TH_CORNERDIST           25
#define K_CONST                         0.04
#define TH_CORNERPOINT          100

#define TH_MATCH_NCC            0.00 //0.63(sample), 0.85(chair), 0.70(bed)
#define WINSIZE_NCC                     35
#define MAX_CORNERS                     10000

#define TH_DISTANCE                     10 // threshold of distance in weighted RANSAC // 30

#define MAX_ITER_NUM            50 // max iteration number for minimization
#define ERROR_POW2                      1e-7 // max iteration number for minimization

#define OPTIMIZE_DLT            0       // DLT
#define OPTIMIZE_GD                     1       // Gradient_Descent
#define OPTIMIZE_GN                     2       // Gauss_Newton
#define OPTIMIZE_LM                     3       // Levenberg_Marquardt
#define OPTIMIZE_DL                     4       // Dog_Leg


typedef struct _CornerData
{
        int num;
        CvPoint pt[MAX_CORNERS];
} CornerData;

typedef struct _CornerPair
{
        int num;
        CvPoint ptx[MAX_CORNERS];
        CvPoint ptxp[MAX_CORNERS];
        float sm_val[MAX_CORNERS];
```

```
} CornerPair;


IplImage *imga, *imgb;
CornerPair cnpair, incnpair;
CornerData cndata_x, cndata_xp;

//-------------------------------------------------------------//
// Error_n_Jacobian
//-------------------------------------------------------------//
void Error_n_Jacobian(CvMat *error, CvMat *jacob, CornerPair incnpair, CvMat *h)
{
        double  xhat, yhat, what, what_pow2, htmp[9];
        double ptxx, ptxy, ptxpx, ptxpy;
        int i;

        cvZero(error);
        cvZero(jacob);
        htmp[0] = cvmGet(h,0,0);
        htmp[1] = cvmGet(h,1,0);
        htmp[2] = cvmGet(h,2,0);
        htmp[3] = cvmGet(h,3,0);
        htmp[4] = cvmGet(h,4,0);
        htmp[5] = cvmGet(h,5,0);
        htmp[6] = cvmGet(h,6,0);
        htmp[7] = cvmGet(h,7,0);
        htmp[8] = cvmGet(h,8,0);
        for (i=0; i<incnpair.num;i++){
                ptxx = (double)incnpair.ptx[i].x;
                ptxy = (double)incnpair.ptx[i].y;
                ptxpx = (double)incnpair.ptxp[i].x;
                ptxpy = (double)incnpair.ptxp[i].y;
                xhat =   htmp[0]*ptxx+htmp[1]*ptxy+htmp[2];
                yhat =   htmp[3]*ptxx+htmp[4]*ptxy+htmp[5];
                what =   htmp[6]*ptxx+htmp[7]*ptxy+htmp[8];
                what_pow2 = what*what;
#ifdef DEBUG
                if(i==0){
                        printf("xhat[%d] = %f, yhat[%d] = %f, what[%d]
= %f₩n",i,xhat,i,yhat,i,what);
                        printf("incnpair.ptxp[%d].x = %f, xhat/what[%d]
= %f₩n",i,ptxpx,i,xhat/what);
                        printf("incnpair.ptxp[%d].y = %f, yhat/what[%d]
= %f₩n",i,ptxpy,i,yhat/what);
                }
#endif
                // error vector
                cvmSet(error,i*2,0,ptxpx - xhat/what);
```

```
                cvmSet(error,i*2+1,0,ptxpy - yhat/what);
#ifdef DEBUG
                if(i==0){
                        printf("error[%d] = %f₩n",i*2,cvmGet(error,i*2,0));
                        printf("error[%d] = %f₩n",i*2+1,cvmGet(error,i*2+1,0));
                }
#endif
                // Jacobian matrix
                cvmSet(jacob,i*2,0,-ptxx/what);
                cvmSet(jacob,i*2,1,-ptxy/what);
                cvmSet(jacob,i*2,2,-1./what);
                cvmSet(jacob,i*2,6,ptxx*xhat/what_pow2);
                cvmSet(jacob,i*2,7,ptxy*xhat/what_pow2);
                cvmSet(jacob,i*2,8,xhat/what_pow2);
                cvmSet(jacob,i*2+1,3,-ptxx/what);
                cvmSet(jacob,i*2+1,4,-ptxy/what);
                cvmSet(jacob,i*2+1,5,-1./what);
                cvmSet(jacob,i*2+1,6,ptxx*yhat/what_pow2);
                cvmSet(jacob,i*2+1,7,ptxy*yhat/what_pow2);
                cvmSet(jacob,i*2+1,8,yhat/what_pow2);
#ifdef DEBUG
                if(i==0){
                        printf("jacob[%d][%d] = %f₩n",i*2,0,cvmGet(jacob,i*2,0));
                        printf("jacob[%d][%d] = %f₩n",i*2,1,cvmGet(jacob,i*2,1));
                        printf("jacob[%d][%d] = %f₩n",i*2,2,cvmGet(jacob,i*2,2));
                        printf("jacob[%d][%d] = %f₩n",i*2,3,cvmGet(jacob,i*2,3));
                        printf("jacob[%d][%d] = %f₩n",i*2,4,cvmGet(jacob,i*2,4));
                        printf("jacob[%d][%d] = %f₩n",i*2,5,cvmGet(jacob,i*2,5));
                        printf("jacob[%d][%d] = %f₩n",i*2,6,cvmGet(jacob,i*2,6));
                        printf("jacob[%d][%d] = %f₩n",i*2,7,cvmGet(jacob,i*2,7));
                        printf("jacob[%d][%d] = %f₩n",i*2,8,cvmGet(jacob,i*2,8));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,0,cvmGet(jacob,i*2+1,0));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,1,cvmGet(jacob,i*2+1,1));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,2,cvmGet(jacob,i*2+1,2));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,3,cvmGet(jacob,i*2+1,3));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,4,cvmGet(jacob,i*2+1,4));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,5,cvmGet(jacob,i*2+1,5));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,6,cvmGet(jacob,i*2+1,6));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,7,cvmGet(jacob,i*2+1,7));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,8,cvmGet(jacob,i*2+1,8));
                }
#endif
        }
}

//-------------------------------------------------------------//
// Gradient_Descent
//-------------------------------------------------------------//
```

```
double Gradient_Descent(CvMat *H, CornerPair incnpair, double err_pow2)
{
        double tmp, alpha = 1e-11, ch0;
        CvMat *error = cvCreateMat(incnpair.num*2,1,CV_64FC1);
        CvMat *jacob = cvCreateMat(incnpair.num*2,9,CV_64FC1);
        CvMat *jacob_T = cvCreateMat(9,incnpair.num*2,CV_64FC1);
        CvMat *jacob_T_error = cvCreateMat(9,1,CV_64FC1);
        CvMat *h0 = cvCreateMat(9,1,CV_64FC1);
        CvMat *h1 = cvCreateMat(9,1,CV_64FC1);
        CvMat *hdiff = cvCreateMat(9,1,CV_64FC1);

        // Matrix H to Vector h0
        cvmSet(h0,0,0,cvmGet(H,0,0));
        cvmSet(h0,1,0,cvmGet(H,0,1));
        cvmSet(h0,2,0,cvmGet(H,0,2));
        cvmSet(h0,3,0,cvmGet(H,1,0));
        cvmSet(h0,4,0,cvmGet(H,1,1));
        cvmSet(h0,5,0,cvmGet(H,1,2));
        cvmSet(h0,6,0,cvmGet(H,2,0));
        cvmSet(h0,7,0,cvmGet(H,2,1));
        cvmSet(h0,8,0,cvmGet(H,2,2));

        for(int k=0;k<MAX_ITER_NUM;k++){
#ifdef DEBUG
                printf("h0 = %f₩n",cvmGet(h0,0,0));
                printf("h1 = %f₩n",cvmGet(h0,1,0));
                printf("h2 = %f₩n",cvmGet(h0,2,0));
                printf("h3 = %f₩n",cvmGet(h0,3,0));
                printf("h4 = %f₩n",cvmGet(h0,4,0));
                printf("h5 = %f₩n",cvmGet(h0,5,0));
                printf("h6 = %f₩n",cvmGet(h0,6,0));
                printf("h7 = %f₩n",cvmGet(h0,7,0));
                printf("h8 = %f₩n",cvmGet(h0,8,0));
#endif
                // Calculate Jr(h) and r(h)
                Error_n_Jacobian(error, jacob, incnpair, h0);
                // Jr(h)^T
                cvmTranspose(jacob, jacob_T);
                // Jr(h)^T*r(h)
                cvMatMul(jacob_T,error,jacob_T_error);
                // h1 = h0 -2.*alpha*jr(h)^T*r(h)
                cvScaleAdd(jacob_T_error, cvScalar(-2.*alpha), h0, h1 );
                // hdiff = h1 - h0 = -2.*alpha*jr(h)^T*r(h)
                cvSub(h1, h0, hdiff);
                // || h1- h0 ||^2 < ERROR_POW2
                tmp = cvDotProduct(hdiff,hdiff);
                printf("iter = %d, || hdiff ||^2 = %f₩n", k, tmp);
                if(tmp < err_pow2)
```

```
                        break;
                cvCopy(h1,h0,NULL);
                alpha *= 0.01;
        }
        // Vector h1 to Matrix H
        cvmSet(H,0,0,cvmGet(h1,0,0));
        cvmSet(H,0,1,cvmGet(h1,1,0));
        cvmSet(H,0,2,cvmGet(h1,2,0));
        cvmSet(H,1,0,cvmGet(h1,3,0));
        cvmSet(H,1,1,cvmGet(h1,4,0));
        cvmSet(H,1,2,cvmGet(h1,5,0));
        cvmSet(H,2,0,cvmGet(h1,6,0));
        cvmSet(H,2,1,cvmGet(h1,7,0));
        cvmSet(H,2,2,cvmGet(h1,8,0));

        ch0 = cvDotProduct(error, error);

        cvReleaseMat(&error);
        cvReleaseMat(&jacob);
        cvReleaseMat(&jacob_T);
        cvReleaseMat(&jacob_T_error);
        cvReleaseMat(&h0);
        cvReleaseMat(&h1);
        cvReleaseMat(&jacob);
        cvReleaseMat(&hdiff);

        return ch0;
}

//-------------------------------------------------------------//
// Gauss_Newton
//-------------------------------------------------------------//
double Gauss_Newton(CvMat *H, CornerPair incnpair, double err_pow2)
{
        double tmp, ch0;
        CvMat *error = cvCreateMat(incnpair.num*2,1,CV_64FC1);
        CvMat *jacob = cvCreateMat(incnpair.num*2,9,CV_64FC1);
        CvMat *jacob_T = cvCreateMat(9,incnpair.num*2,CV_64FC1);
        CvMat *jacob_T_error = cvCreateMat(9,1,CV_64FC1);
        CvMat *jacob_T_jacob = cvCreateMat(9,9,CV_64FC1);
        CvMat *inv_jacob_T_jacob = cvCreateMat(9,9,CV_64FC1);
        CvMat *invjTj_jTe = cvCreateMat(9,1,CV_64FC1);
        CvMat *h0 = cvCreateMat(9,1,CV_64FC1);
        CvMat *h1 = cvCreateMat(9,1,CV_64FC1);
        CvMat *hdiff = cvCreateMat(9,1,CV_64FC1);

        // Matrix H to Vector h0
        cvmSet(h0,0,0,cvmGet(H,0,0));
```

```
cvmSet(h0,1,0,cvmGet(H,0,1));
cvmSet(h0,2,0,cvmGet(H,0,2));
cvmSet(h0,3,0,cvmGet(H,1,0));
cvmSet(h0,4,0,cvmGet(H,1,1));
cvmSet(h0,5,0,cvmGet(H,1,2));
cvmSet(h0,6,0,cvmGet(H,2,0));
cvmSet(h0,7,0,cvmGet(H,2,1));
cvmSet(h0,8,0,cvmGet(H,2,2));

for(int k=0;k<MAX_ITER_NUM;k++){
        // Calculate Jr(h) and r(h) : 2nX9, 2nX1
        Error_n_Jacobian(error, jacob, incnpair, h0);
        // Jr(h)^T : 9X2n
        cvmTranspose(jacob, jacob_T);
        // Jr(h)^T*r(h) : 9X1
        cvMatMul(jacob_T, error, jacob_T_error);
        // Jr(h)^T*Jr(h) : 9X9
        cvMatMul(jacob_T, jacob, jacob_T_jacob);
        // (Jr(h)^T*Jr(h))^-1 : 9X9
        cvInvert(jacob_T_jacob, inv_jacob_T_jacob);
        // (Jr(h)^T*Jr(h))^-1*jr(h)^T*r(h) : 9X1
        cvMatMul(inv_jacob_T_jacob, jacob_T_error, invjTj_jTe);
        // h1 = h0 -(Jr(h)^T*Jr(h))^-1*jr(h)^T*r(h)
        cvSub(h0, invjTj_jTe, h1);
        // hdiff = h1 - h0 = -(Jr(h)^T*Jr(h))^-1*jr(h)^T*r(h)
                // cvSub(h1, h0, hdiff);
        // || h1- h0 ||^2 < ERROR_POW2
                // tmp = cvDotProduct(hdiff,hdiff);
        tmp = cvDotProduct(invjTj_jTe, invjTj_jTe);
        printf("iter = %d, || hdiff ||^2 = %f₩n", k, tmp);
        if(tmp < err_pow2)
                break;
        cvCopy(h1,h0,NULL);
}
// Vector h1 to Matrix H
cvmSet(H,0,0,cvmGet(h1,0,0));
cvmSet(H,0,1,cvmGet(h1,1,0));
cvmSet(H,0,2,cvmGet(h1,2,0));
cvmSet(H,1,0,cvmGet(h1,3,0));
cvmSet(H,1,1,cvmGet(h1,4,0));
cvmSet(H,1,2,cvmGet(h1,5,0));
cvmSet(H,2,0,cvmGet(h1,6,0));
cvmSet(H,2,1,cvmGet(h1,7,0));
cvmSet(H,2,2,cvmGet(h1,8,0));

ch0 = cvDotProduct(error, error);

cvReleaseMat(&error);
```

```
        cvReleaseMat(&jacob);
        cvReleaseMat(&jacob_T);
        cvReleaseMat(&jacob_T_error);
        cvReleaseMat(&h0);
        cvReleaseMat(&h1);
        cvReleaseMat(&jacob);
        cvReleaseMat(&hdiff);

        return ch0;
}

//-------------------------------------------------------------//
// Levenberg_Marquardt
//-------------------------------------------------------------//
double Levenberg_Marquardt(CvMat *H, CornerPair incnpair, double err_pow2)
{
        double tmp, tmpmax, mu, tau = 1e-1, rhoLM, ch0, ch1;
        CvMat *error = cvCreateMat(incnpair.num*2,1,CV_64FC1);
        CvMat *jacob = cvCreateMat(incnpair.num*2,9,CV_64FC1);
        CvMat *ident = cvCreateMat(9,9,CV_64FC1);
        CvMat *jacob_T = cvCreateMat(9,incnpair.num*2,CV_64FC1);
        CvMat *jacob_T_error = cvCreateMat(9,1,CV_64FC1);
        CvMat *jacob_T_jacob = cvCreateMat(9,9,CV_64FC1);
        CvMat *jTj_mui = cvCreateMat(9,9,CV_64FC1);
        CvMat *inv_jTj_mui = cvCreateMat(9,9,CV_64FC1);
        CvMat *diag = cvCreateMat(9,1,CV_64FC1);
        CvMat *invjTjmui_jTe = cvCreateMat(9,1,CV_64FC1);
        CvMat *h0 = cvCreateMat(9,1,CV_64FC1);
        CvMat *h1 = cvCreateMat(9,1,CV_64FC1);
        CvMat *htmp = cvCreateMat(9,1,CV_64FC1);
        CvMat *deltah = cvCreateMat(9,1,CV_64FC1);
        CvMat *jTr_mudeltah = cvCreateMat(9,1,CV_64FC1);

        cvSetIdentity(ident);

        // Matrix H to Vector h0
        cvmSet(h0,0,0,cvmGet(H,0,0));
        cvmSet(h0,1,0,cvmGet(H,0,1));
        cvmSet(h0,2,0,cvmGet(H,0,2));
        cvmSet(h0,3,0,cvmGet(H,1,0));
        cvmSet(h0,4,0,cvmGet(H,1,1));
        cvmSet(h0,5,0,cvmGet(H,1,2));
        cvmSet(h0,6,0,cvmGet(H,2,0));
        cvmSet(h0,7,0,cvmGet(H,2,1));
        cvmSet(h0,8,0,cvmGet(H,2,2));

        // To set initial mu
        Error_n_Jacobian(error, jacob, incnpair, h0);
```

```
// Jr(h)^T : 9X2n
cvmTranspose(jacob, jacob_T);
// Jr(h)^T*r(h) : 9X1
cvMatMul(jacob_T, error, jacob_T_error);
// Jr(h)^T*Jr(h) : 9X9
cvMatMul(jacob_T, jacob, jacob_T_jacob);
// diag(Jr(h)^T*Jr(h)) : 9X1
cvGetDiag(jacob_T_jacob, diag, 0);
// find max element of diag(Jr(h)^T*Jr(h))
tmpmax = cvmGet(diag,0,0);
for(int i=1;i<9;i++){
        tmpmax = max(tmpmax,cvmGet(diag,i,0));
}
// initial mu
mu = tau*tmpmax;
for(int k=0;k<MAX_ITER_NUM;k++){
        // Calculate Jr(h) and r(h) : 2nX9, 2nX1
        Error_n_Jacobian(error, jacob, incnpair, h0);
        /* ------- for calculate mu -------- */
        if(k!=0){
                // C(hk+1) = ||r(hk+1)||^2 = ||error||^2
                ch1 = cvDotProduct(error, error);
                // C(hk+1) - C(hk)
                ch0 = ch1 - ch0;
                // Jr(hk)^T*r(hk)-mu*deltahk
                cvScaleAdd(deltah, cvScalar(-1.*mu), jacob_T_error, jTr_mudeltah);
                // deltah^T*(Jr(hk)^T*r(hk)-mu*deltahk)
                tmp = cvDotProduct(deltah, jTr_mudeltah);
                // rhoLM = (C(hk+1) - C(hk))/ (deltah^T*(Jr(hk)^T*r(hk)-mu*deltahk))
                rhoLM = ch0/tmp;
                // if rhoLM > 0  then muk+1 = muk*max(1/3,1-(2*rhoLM-1)^3)
                // else muk+1 = 2*muk
                if(rhoLM >0)
                        mu = mu*max(1/3, pow(1-(2*rhoLM-1),3));
                else{
                        mu = 2.*mu;
                        cvCopy(htmp,h0,NULL); // restore previous value
                        // Calculate Jr(h) and r(h) again: 2nX9, 2nX1
                        Error_n_Jacobian(error, jacob, incnpair, h0);
                }
        }
        /* ------- for update hk -------- */
        // Jr(h)^T : 9X2n
        cvmTranspose(jacob, jacob_T);
        // Jr(h)^T*r(h) : 9X1
        cvMatMul(jacob_T, error, jacob_T_error);
        // Jr(h)^T*Jr(h) : 9X9
        cvMatMul(jacob_T, jacob, jacob_T_jacob);
```

```
        // Jr(h)^T*Jr(h)+mu*I
        cvScaleAdd(ident, cvScalar(mu), jacob_T_jacob, jTj_mui);
        // (Jr(h)^T*Jr(h)+mu*I)^-1 : 9X9
        cvInvert(jTj_mui, inv_jTj_mui);
        // (Jr(h)^T*Jr(h)+mu*I)^-1*jr(h)^T*r(h)
        cvMatMul(inv_jTj_mui, jacob_T_error, invjTjmui_jTe);
        // h1 = h0 -(Jr(h)^T*Jr(h)+mu*I)^-1*jr(h)^T*r(h)
        cvSub(h0, invjTjmui_jTe, h1);
        // hdiff = h1 - h0
        cvSub(h1, h0, deltah);
        // || h1- h0 ||^2 < ERROR_POW2
        tmp = cvDotProduct(deltah, deltah);
        printf("iter = %d, || hdiff ||^2 = %fWn", k, tmp);
        if(tmp < err_pow2)
                break;
        // C(hk) = ||r(hk)||^2 = ||error||^2
        ch0 = cvDotProduct(error, error);
        // hk backup
        cvCopy(h0,htmp,NULL);
        // hk update
        cvCopy(h1,h0,NULL);
}
// Vector h1 to Matrix H
cvmSet(H,0,0,cvmGet(h1,0,0));
cvmSet(H,0,1,cvmGet(h1,1,0));
cvmSet(H,0,2,cvmGet(h1,2,0));
cvmSet(H,1,0,cvmGet(h1,3,0));
cvmSet(H,1,1,cvmGet(h1,4,0));
cvmSet(H,1,2,cvmGet(h1,5,0));
cvmSet(H,2,0,cvmGet(h1,6,0));
cvmSet(H,2,1,cvmGet(h1,7,0));
cvmSet(H,2,2,cvmGet(h1,8,0));

ch0 = cvDotProduct(error, error);

cvReleaseMat(&error);
cvReleaseMat(&jacob);
cvReleaseMat(&jacob_T);
cvReleaseMat(&jacob_T_error);
cvReleaseMat(&jacob_T_jacob);
cvReleaseMat(&h0);
cvReleaseMat(&h1);
cvReleaseMat(&htmp);
cvReleaseMat(&deltah);
cvReleaseMat(&ident);
cvReleaseMat(&jTj_mui);
cvReleaseMat(&inv_jTj_mui);
cvReleaseMat(&diag);
```

```
        cvReleaseMat(&invjTjmui_jTe);
        cvReleaseMat(&jTr_mudeltah);

        return ch0;
}

//-------------------------------------------------------------//
// Dog_Leg
//-------------------------------------------------------------//
double Dog_Leg(CvMat *H, CornerPair incnpair, double err_pow2)
{
        int j=0;
        double tmp, deltahGD_size, deltahGN_size;
        double rhoDL, Delta = 1., ch0, ch1, jTe_size, jjTe_size;
        double beta, a, b, c;
        CvMat *error = cvCreateMat(incnpair.num*2,1,CV_64FC1);
        CvMat *jacob = cvCreateMat(incnpair.num*2,9,CV_64FC1);
        CvMat *ident = cvCreateMat(9,9,CV_64FC1);
        CvMat *jacob_T = cvCreateMat(9,incnpair.num*2,CV_64FC1);
        CvMat *jacob_T_error = cvCreateMat(9,1,CV_64FC1);
        CvMat *jacob_T_jacob = cvCreateMat(9,9,CV_64FC1);
        CvMat *h0 = cvCreateMat(9,1,CV_64FC1);
        CvMat *h1 = cvCreateMat(9,1,CV_64FC1);
        CvMat *hGN = cvCreateMat(9,1,CV_64FC1);
        CvMat *hGD = cvCreateMat(9,1,CV_64FC1);
        CvMat *deltah = cvCreateMat(9,1,CV_64FC1);
        CvMat *deltah_T = cvCreateMat(1,9,CV_64FC1);
        CvMat *deltahGD = cvCreateMat(9,1,CV_64FC1);
        CvMat *deltahGN = cvCreateMat(9,1,CV_64FC1);
        CvMat *dhGN_GD = cvCreateMat(9,1,CV_64FC1);
        CvMat *dhGD_d_dhGN_GD = cvCreateMat(9,1,CV_64FC1);
        CvMat *error_T = cvCreateMat(1,incnpair.num*2,CV_64FC1);
        CvMat *error_T_jacob = cvCreateMat(1,9,CV_64FC1);
        CvMat *dhT_jTj = cvCreateMat(1,9,CV_64FC1);
        CvMat *dhT_jTj_dh = cvCreateMat(1,1,CV_64FC1);
        CvMat *eTj_dh = cvCreateMat(1,1,CV_64FC1);
        CvMat *jTj_mui = cvCreateMat(9,9,CV_64FC1);
        CvMat *inv_jTj_mui = cvCreateMat(9,9,CV_64FC1);
        CvMat *invjTjmui_jTe = cvCreateMat(9,1,CV_64FC1);
        CvMat *jjTe = cvCreateMat(incnpair.num*2,1,CV_64FC1);
        CvMat *htmp = cvCreateMat(9,1,CV_64FC1);

        cvSetIdentity(ident);

        // Matrix H to Vector h0
        cvmSet(h0,0,0,cvmGet(H,0,0));
        cvmSet(h0,1,0,cvmGet(H,0,1));
        cvmSet(h0,2,0,cvmGet(H,0,2));
```

```
	cvmSet(h0,3,0,cvmGet(H,1,0));
	cvmSet(h0,4,0,cvmGet(H,1,1));
	cvmSet(h0,5,0,cvmGet(H,1,2));
	cvmSet(h0,6,0,cvmGet(H,2,0));
	cvmSet(h0,7,0,cvmGet(H,2,1));
	cvmSet(h0,8,0,cvmGet(H,2,2));

	for(int k=0;k<MAX_ITER_NUM;k++){
		// Calculate Jr(h) and r(h) : 2nX9, 2nX1
		Error_n_Jacobian(error, jacob, incnpair, h0);
		/* ------- for calculate Delta -------- */
		if(k!=0){
			// C(hk+1) = ||r(hk+1)||^2 = ||error||^2
			ch1 = cvDotProduct(error, error);
			// C(hk+1) - C(hk)
			ch0 = ch1 - ch0;
			// deltah^T : 1X9
			cvmTranspose(deltah, deltah_T);
			// deltah^T*Jr(h)^T*Jr(h) : 1X9
			cvMatMul(deltah_T, jacob_T_jacob, dhT_jTj);
			// deltah^T*Jr(h)^T*Jr(h)*deltah : 1X1
			cvMatMul(dhT_jTj, deltah, dhT_jTj_dh);
			// r(h)^T*Jr(h) : 1X9
			cvMatMul(error_T, jacob, error_T_jacob);
			// r(h)^T*Jr(h)*deltah : 1X1
			cvMatMul(error_T_jacob, deltah, eTj_dh);
			// rhoLM = (C(hk+1) - C(hk))/
(deltah^T*Jr(h)^T*Jr(h)*deltah+2*r(h)^T*Jr(h)*deltah))
			tmp = cvmGet(dhT_jTj_dh,0,0) + 2*cvmGet(eTj_dh,0,0);
			rhoDL = ch0/tmp;
			// Delta update
			if(rhoDL<=0){
				Delta = Delta/4.;
				cvCopy(htmp,h0,NULL); // restore previous value
				// Calculate Jr(h) and r(h) again: 2nX9, 2nX1
				Error_n_Jacobian(error, jacob, incnpair, h0);
			}
			else if(rhoDL <= 1/4){
				Delta = Delta/4.;
			}
			else if((rhoDL>1/4)&&(rhoDL<=3/4))
				Delta = Delta;
			else
				Delta = 2.*Delta;
		}
		/* ------- for update hk -------- */
		// Jr(h)^T : 9X2n
		cvmTranspose(jacob, jacob_T);
```

```
// Jr(h)^T*r(h) : 9X1
cvMatMul(jacob_T, error, jacob_T_error);
// Jr(h)^T*Jr(h) : 9X9
cvMatMul(jacob_T, jacob, jacob_T_jacob);
// r(h)^T : 1X2n
cvmTranspose(error, error_T);
// r(h)^T*Jr(h) : 1X9
cvMatMul(error_T, jacob, error_T_jacob);
        /* hGN */
// Jr(h)^T*Jr(h)
cvScaleAdd(ident, cvScalar(0), jacob_T_jacob, jTj_mui);
// (Jr(h)^T*Jr(h)+mu*I)^-1 : 9X9
cvInvert(jTj_mui, inv_jTj_mui);
// (Jr(h)^T*Jr(h)+mu*I)^-1*jr(h)^T*r(h)
cvMatMul(inv_jTj_mui, jacob_T_error, invjTjmui_jTe);
// hGN = h0 -(Jr(h)^T*Jr(h)+mu*I)^-1*jr(h)^T*r(h)
cvSub(h0, invjTjmui_jTe, hGN);
// deltahGN = hGN - h0
cvSub(hGN, h0, deltahGN);
// || hGN- h0 ||
deltahGN_size = sqrt(cvDotProduct(deltahGN, deltahGN));
        /* hGD */
// ||Jr(h)^T*r(h)||^2
jTe_size = cvDotProduct(jacob_T_error, jacob_T_error);
// Jr(h)*Jr(h)^T*r(h) : 2nX1
cvMatMul(jacob, jacob_T_error, jjTe);
// ||Jr(h)*Jr(h)^T*r(h)||^2
jjTe_size = cvDotProduct(jjTe, jjTe);
// hGD = h0-(||Jr(h)^T*r(h)||^2/||Jr(h)*Jr(h)^T*r(h)||^2)*Jr(h)^T*r(h)
cvScaleAdd(jacob_T_error, cvScalar(-1.*jTe_size/jjTe_size), h0, hGD);
// deltahGD = hGD - h0
cvSub(hGD, h0, deltahGD);
// || hGD- h0 ||
deltahGD_size = sqrt(cvDotProduct(deltahGD, deltahGD));
        /* update h1 */
if(Delta >= deltahGN_size)
        // h1 = h0 + deltahGN
        cvAdd(h0,deltahGN,h1);
else if((Delta < deltahGN_size)&&(Delta >= deltahGD_size)){
        // deltahGN - deltahGD
        cvSub(deltahGN, deltahGD, dhGN_GD);
        // beta
        a = cvDotProduct(dhGN_GD, dhGN_GD);
        b = 2*cvDotProduct(deltahGD, dhGN_GD);
        c = cvDotProduct(deltahGD, deltahGD)-Delta*Delta;
        beta = (sqrt(b*b-4*a*c)-b)/(2*a);
        // h1 = h0 + deltahGD + beta*(deltahGN-deltahGD)
        cvScaleAdd(dhGN_GD, cvScalar(beta), deltahGD, dhGD_d_dhGN_GD);
```

```
                cvAdd(h0, dhGD_d_dhGN_GD, h1);
        }
        else
                // h1 = h0 + Delta/||deltahGD||*deltahGD
                cvScaleAdd(deltahGD, cvScalar(Delta/deltahGD_size), h0, h1);

        // hdiff = h1 - h0
        cvSub(h1, h0, deltah);
        // || h1- h0 ||^2 < ERROR_POW2
        tmp = cvDotProduct(deltah, deltah);
        printf("iter = %d, || hdiff ||^2 = %f₩n", k, tmp);
        if(tmp < err_pow2)
                break;
        // C(hk) = ||r(hk)||^2 = ||error||^2
        ch0 = cvDotProduct(error, error);
        // hk backup
        cvCopy(h0,htmp,NULL);
        // hk update
        cvCopy(h1,h0,NULL);
}
// Vector h1 to Matrix H
cvmSet(H,0,0,cvmGet(h1,0,0));
cvmSet(H,0,1,cvmGet(h1,1,0));
cvmSet(H,0,2,cvmGet(h1,2,0));
cvmSet(H,1,0,cvmGet(h1,3,0));
cvmSet(H,1,1,cvmGet(h1,4,0));
cvmSet(H,1,2,cvmGet(h1,5,0));
cvmSet(H,2,0,cvmGet(h1,6,0));
cvmSet(H,2,1,cvmGet(h1,7,0));
cvmSet(H,2,2,cvmGet(h1,8,0));

ch0 = cvDotProduct(error, error);

cvReleaseMat(&error);
cvReleaseMat(&jacob);
cvReleaseMat(&jacob_T);
cvReleaseMat(&jacob_T_error);
cvReleaseMat(&jacob_T_jacob);
cvReleaseMat(&jjTe);
cvReleaseMat(&h0);
cvReleaseMat(&h1);
cvReleaseMat(&hGN);
cvReleaseMat(&hGD);
cvReleaseMat(&htmp);
cvReleaseMat(&deltahGN);
cvReleaseMat(&deltahGD);
cvReleaseMat(&ident);
cvReleaseMat(&jTj_mui);
```

```
        cvReleaseMat(&inv_jTj_mui);
        cvReleaseMat(&invjTjmui_jTe);
        cvReleaseMat(&dhGN_GD);
        cvReleaseMat(&dhGD_d_dhGN_GD);
        cvReleaseMat(&error_T);
        cvReleaseMat(&error_T_jacob);
        cvReleaseMat(&deltah);
        cvReleaseMat(&deltah_T);
        cvReleaseMat(&dhT_jTj);
        cvReleaseMat(&dhT_jTj_dh);
        cvReleaseMat(&eTj_dh);

        return ch0;
}

//-------------------------------------------------------------//
// SobelFilter
//-------------------------------------------------------------//
void SobelFilter(IplImage *pimg, CvMat* lx, CvMat* ly)
{
        int dx[9] = {-1, 0, 1,
                                -1, 0, 1,
                                -1, 0, 1};
        int dy[9] = {-1, -1, -1,
                                 0,  0,  0,
                                 1,  1,  1};
        double xm[9] = {-1, 0, 1,
                                       -2, 0, 2,
                                       -1, 0, 1};
        double ym[9] = {1, 2, 1,
                                       0, 0, 0,
                                       -1, -2, -1};
        int width, height, x, y, k, m;
        double sumx, sumy;
        CvScalar imgval;
        CvMat sobelx = cvMat(3,3,CV_64FC1,xm);
        CvMat sobely = cvMat(3,3,CV_64FC1,ym);

        width = pimg->width;
        height = pimg->height;

        for(y=1; y<height-1; y++)
        for(x=1; x<width-1; x++){
                sumx = 0; sumy = 0;
                for(k=-1; k<=1; k++)
                for(m=-1; m<=1; m++){
                        imgval = cvGet2D(pimg,y+k,x+m);
                        sumx += imgval.val[0]*cvmGet(&sobelx,k+1,m+1);
```

```
                        sumy += imgval.val[0]*cvmGet(&sobely,k+1,m+1);
                }
                cvmSet(Ix,y,x,(sumx));
                cvmSet(Iy,y,x,(sumy));
        }
}
//-------------------------------------------------------------//
// Function : CornerCheck
//-------------------------------------------------------------//
void CornerCheck(int x, int y, int *cornerNo, CvPoint *corners,
                                double *cornerVal, double cvalue)
{
        int i, exist;
        double r;

        if(cvalue > TH_CORNERPOINT )
        {
                exist=0;
                for(i=0;i<*cornerNo;i++)
                {
                        r = sqrt((double)((corners[i].x-x)*(corners[i].x-x)
                                                        +(corners[i].y-y)*(corners[i].y-y)));
                        if(r < TH_CORNERDIST)
                        {
                                if( cornerVal[i] < cvalue)
                                {
                                        corners[i].x = x;
                                        corners[i].y = y;
                                        cornerVal[i]= cvalue;
                                }
                                exist=1;
                                break;
                        }
                }
                if(exist==0)
                {
                        corners[*cornerNo].x = x;
                        corners[*cornerNo].y = y;
                        if((*cornerNo) >= MAX_CORNERS)
                        {
                                printf("Too Many Corners!!");
                        }
                        else
                        {
                                cornerVal[*cornerNo]=cvalue;
                                (*cornerNo) = (*cornerNo) + 1;
                        }
                }
```

```
        }
}


//-----------------------------------------------------------------//
// FindingCorners
//-----------------------------------------------------------------//
void FindingCorners(IplImage *pimg,     CornerData *corners)
{
        int width, height, i,j,k,m;
        int num = 0;
        double cornerVal[MAX_CORNERS];
        double lxx_sum, lyy_sum, lxy_sum;
        double cornerness;

        CvPoint curr_point;
        width = pimg->width;
        height = pimg->height;
        CvMat *lx = cvCreateMat(height,width,CV_64FC1);
        CvMat *ly = cvCreateMat(height,width,CV_64FC1);

        SobelFilter(pimg, lx, ly);
        corners->num = 0;
        for(i=WINSIZE_CORNERFIND;i<height-WINSIZE_CORNERFIND;i++)
        for(j=WINSIZE_CORNERFIND;j<width-WINSIZE_CORNERFIND;j++)
        {
                curr_point = cvPoint(j,i);
                lxx_sum = 0.;
                lyy_sum = 0.;
                lxy_sum = 0.;
                cornerness = 0.;
                for(k=-WINSIZE_CORNERFIND;k<=WINSIZE_CORNERFIND;k++)
                for(m=-WINSIZE_CORNERFIND;m<=WINSIZE_CORNERFIND;m++)
                {
                        lxx_sum += cvmGet(lx,i+k,j+m)
                                                *cvmGet(lx,i+k,j+m)/10000;
                        lyy_sum += cvmGet(ly,i+k,j+m)
                                                *cvmGet(ly,i+k,j+m)/10000;
                        lxy_sum += cvmGet(lx,i+k,j+m)
                                                *cvmGet(ly,i+k,j+m)/10000;
                }
                cornerness = (lxx_sum*lyy_sum - lxy_sum*lxy_sum)
                                        - K_CONST*(lxx_sum+lyy_sum)*(lxx_sum+lyy_sum);
                if(i> (WINSIZE_CORNERFIND+1) && i<height-(WINSIZE_CORNERFIND+1)
                        && j>(WINSIZE_CORNERFIND+1) && j<width-(WINSIZE_CORNERFIND+1))
                       CornerCheck(j,i, &(corners->num), corners->pt, cornerVal, cornerness);

        }
```

```
	cvReleaseMat(&Ix);
	cvReleaseMat(&Iy);
}

//-------------------------------------------------------------//
// NCC
//-------------------------------------------------------------//
int NCC(IplImage *img_a, IplImage *img_b, CornerData *corners1,
		CornerData *corners2, CornerPair *cnpair)
{
	int width, height,i,j, k, m, ncc_num;
	int ax, ay, bx, by, num;
	double amean, bmean, numer;
	double asum, bsum, diff_ij, max_val;
	int check;
	CvPoint max_point;
	CvScalar imgval_a, imgval_b;
	int *matchedidx = new int [corners1->num];

	width= img_a->width;
	height= img_a->height;
	cnpair->num = corners1->num;
	ncc_num = 0;
	for(i=0;i<corners1->num;i++)
	{
		ax = corners1->pt[i].x;
		ay = corners1->pt[i].y;
		max_val=-10000;
		cnpair->ptx[ncc_num].x = ax;
		cnpair->ptx[ncc_num].y = ay;
		for(j=0;j<corners2->num;j++)
		{
			bx = corners2->pt[j].x;
			by = corners2->pt[j].y;
			amean = 0.;
			bmean = 0.;
			num = 0;
			for(k=-WINSIZE_NCC; k<=WINSIZE_NCC;k++)
			for(m=-WINSIZE_NCC; m<=WINSIZE_NCC;m++)
			{
				if((ay+k)<0 || (ay+k)>=height || (ax+m)<0 || (ax+m)>=width)
					continue;
				if((by+k)<0 || (by+k)>=height || (bx+m)<0 || (bx+m)>=width)
					continue;
				imgval_a = cvGet2D(img_a, ay+k, ax+m);
				amean += imgval_a.val[0];
				imgval_b = cvGet2D(img_b, by+k, bx+m);
				bmean += imgval_b.val[0];
```

```
                num++;
        }
        amean /= num;
        bmean /= num;
        asum = 0.;
        bsum = 0.;
        numer = 0.0;
        for(k=-WINSIZE_NCC; k<=WINSIZE_NCC;k++)
        for(m=-WINSIZE_NCC; m<=WINSIZE_NCC;m++)
        {
                if((ay+k)<0 || (ay+k)>=height || (ax+m)<0 || (ax+m)>=width)
                        continue;
                if((by+k)<0 || (by+k)>=height || (bx+m)<0 || (bx+m)>=width)
                        continue;
                imgval_a = cvGet2D(img_a, ay+k, ax+m);
                imgval_b = cvGet2D(img_b, by+k, bx+m);
                numer += (imgval_a.val[0] - amean)
                                *(imgval_b.val[0] - bmean);
                asum += (imgval_a.val[0] - amean)
                                *(imgval_a.val[0] - amean);
                bsum += (imgval_b.val[0] - bmean)
                                *(imgval_b.val[0] - bmean);
        }
        diff_ij = numer/ sqrt(asum*bsum);
        if(diff_ij>max_val)
        {
                max_val=diff_ij;
                max_point.x = bx;
                max_point.y = by;
                matchedidx[ncc_num] = j;
        }
}
cnpair->sm_val[ncc_num]=(float)max_val;
cnpair->ptxp[ncc_num].x = max_point.x;
cnpair->ptxp[ncc_num].y = max_point.y;
cnpair->ptx[ncc_num].x = ax;
cnpair->ptx[ncc_num].y = ay;

check = 0;
for(j=0; j<ncc_num; j++){
        if(matchedidx[j] == matchedidx[ncc_num]){
                if(cnpair->sm_val[j] < cnpair->sm_val[ncc_num]){
                        cnpair->sm_val[j] = cnpair->sm_val[ncc_num];
                        cnpair->ptx[j].x = cnpair->ptx[ncc_num].x;
                        cnpair->ptx[j].y = cnpair->ptx[ncc_num].y;
                }
                check = 1;
                break;
```

```
                        }
                }
                if(check == 0)
                        ncc_num++;
        }
        cnpair->num = ncc_num;
        delete matchedidx;
        return ncc_num;
}

//-------------------------------------------------------------//
// ColinearCheck
//-------------------------------------------------------------//
bool ColinearCheck(int num, CvPoint2D64f *point)
{
        int i,j,k;
        bool colinear = false;
        double tmp;

        CvMat *point1 = cvCreateMat(3,1,CV_64FC1);
        CvMat *point2 = cvCreateMat(3,1,CV_64FC1);
        CvMat *point3 = cvCreateMat(3,1,CV_64FC1);
        CvMat *line = cvCreateMat(3,1,CV_64FC1);
        for(i=0; i<num-2; i++){
                cvmSet(point1,0,0,point[i].x);
                cvmSet(point1,1,0,point[i].y);
                cvmSet(point1,2,0,1);
                for(j=i+1; j<num-1; j++){
                        cvmSet(point2,0,0,point[j].x);
                        cvmSet(point2,1,0,point[j].y);
                        cvmSet(point2,2,0,1);
                        cvCrossProduct(point1, point2, line);
                        for(k=j+1; k<num; k++){
                                cvmSet(point3,0,0,point[k].x);
                                cvmSet(point3,1,0,point[k].y);
                                cvmSet(point3,2,0,1);
                                tmp = cvDotProduct(point3, line);
                                if(abs(tmp) < 0.01){
                                        colinear = true;
                                        break;
                                }
                        }
                        if(colinear == true)
                                break;
                }
                if(colinear == true)
                        break;
        }
```

```
        cvReleaseMat(&point1);
        cvReleaseMat(&point2);
        cvReleaseMat(&point3);
        cvReleaseMat(&line);

        return colinear;
}


//-------------------------------------------------------------//
// CalculateHomography
//-------------------------------------------------------------//
void CalculateHomography(int num, CvPoint2D64f *point1,
                                                CvPoint2D64f *point2, CvMat *H)
{
        CvMat *A = cvCreateMat(2*num, 9, CV_64FC1);
        CvMat *U = cvCreateMat(2*num, 2*num, CV_64FC1);
        CvMat *D = cvCreateMat(2*num, 9, CV_64FC1);
        CvMat *V = cvCreateMat(9, 9, CV_64FC1);
        cvZero(A);

        for(int i=0; i<num; i++){
                cvmSet(A,2*i,3,-point1[i].x);
                cvmSet(A,2*i,4,-point1[i].y);
                cvmSet(A,2*i,5,-1);
                cvmSet(A,2*i,6,point2[i].y*point1[i].x);
                cvmSet(A,2*i,7,point2[i].y*point1[i].y);
                cvmSet(A,2*i,8,point2[i].y);
                cvmSet(A,2*i+1,0,point1[i].x);
                cvmSet(A,2*i+1,1,point1[i].y);
                cvmSet(A,2*i+1,2,1);
                cvmSet(A,2*i+1,6,-point2[i].x*point1[i].x);
                cvmSet(A,2*i+1,7,-point2[i].x*point1[i].y);
                cvmSet(A,2*i+1,8,-point2[i].x);
        }

        cvSVD(A, D, U, V, CV_SVD_U_T|CV_SVD_V_T);       // SVD

        cvmSet(H, 0, 0, cvmGet(V, 8, 0));
        cvmSet(H, 0, 1, cvmGet(V, 8, 1));
        cvmSet(H, 0, 2, cvmGet(V, 8, 2));
        cvmSet(H, 1, 0, cvmGet(V, 8, 3));
        cvmSet(H, 1, 1, cvmGet(V, 8, 4));
        cvmSet(H, 1, 2, cvmGet(V, 8, 5));
        cvmSet(H, 2, 0, cvmGet(V, 8, 6));
        cvmSet(H, 2, 1, cvmGet(V, 8, 7));
        cvmSet(H, 2, 2, cvmGet(V, 8, 8));
```

```
        cvReleaseMat(&A);
        cvReleaseMat(&U);
        cvReleaseMat(&D);
        cvReleaseMat(&V);
}


//-------------------------------------------------------------//
// CalculateInlierNumber
//-------------------------------------------------------------//
int CalculateInlierNumber(int num, CvPoint *point1, CvPoint *point2,
                                        CvMat *H, CvMat *inlier_mask, double *std_d)
{
        double curr_d, sum_d, mean_d, stdev_d;
        CvPoint tmp;
        int i, num_inlier;

        CvMat *x = cvCreateMat(3,1,CV_64FC1);
        CvMat *xp = cvCreateMat(3,1,CV_64FC1);
        CvMat *point = cvCreateMat(3,1,CV_64FC1);
        CvMat *invH = cvCreateMat(3,3,CV_64FC1);
        CvMat *distance = cvCreateMat(num, 1, CV_64FC1);

        cvInvert(H, invH);

        cvZero(inlier_mask);
        sum_d = 0;
        num_inlier = 0;
        for(i=0; i<num; i++){
                cvmSet(x,0,0,point1[i].x);
                cvmSet(x,1,0,point1[i].y);
                cvmSet(x,2,0,1);
                cvmSet(xp,0,0,point2[i].x);
                cvmSet(xp,1,0,point2[i].y);
                cvmSet(xp,2,0,1);
                // d(Hx, xp)
                cvMatMul(H, x, point);
                tmp.x = (int)(cvmGet(point,0,0)/cvmGet(point,2,0));
                tmp.y = (int)(cvmGet(point,1,0)/cvmGet(point,2,0));
                curr_d = (tmp.x-point2[i].x)*(tmp.x-point2[i].x)
                                + (tmp.y-point2[i].y)*(tmp.y-point2[i].y);
                // d(x, invH xp)
                cvMatMul(invH, xp, point);
                tmp.x = (int)((cvmGet(point,0,0)/cvmGet(point,2,0)));
                tmp.y = (int)((cvmGet(point,1,0)/cvmGet(point,2,0)));
                curr_d += (tmp.x-point1[i].x)*(tmp.x-point1[i].x)
                                + (tmp.y-point1[i].y)*(tmp.y-point1[i].y);
```

```
                if(curr_d < TH_DISTANCE){
                        cvmSet(distance,i,0,curr_d);
                        sum_d += curr_d;
                        cvmSet(inlier_mask,i,0,1);
                        num_inlier++;
                }
        }

        // standard deviation
        mean_d = sum_d/(double)num_inlier;
        stdev_d = 0;
        for(i=0; i<num; i++){
                if(cvmGet(inlier_mask,i,0) == 1)
                        stdev_d += (cvmGet(distance,i,0)-mean_d)
                                        *(cvmGet(distance,i,0)-mean_d);
        }
        *std_d = stdev_d/(double) (num_inlier -1);

        cvReleaseMat(&x);
        cvReleaseMat(&xp);
        cvReleaseMat(&point);
        cvReleaseMat(&invH);
        cvReleaseMat(&distance);

        return num_inlier;
}


//-------------------------------------------------------------//
// Normalize
//-------------------------------------------------------------//
void Normalize(int num, CvPoint2D64f *point, CvMat *T)
{
        double scale, tx, ty, meanx, meany, tmp;
        int i;
        CvMat *x = cvCreateMat(3,1,CV_64FC1);
        CvMat *xp = cvCreateMat(3,1,CV_64FC1);

        meanx = 0;      meany = 0;
        for(i=0; i<num; i++){
                meanx += point[i].x;
                meany += point[i].y;
        }
        meanx = meanx/(double)num;
        meany = meany/(double)num;
        tmp = 0;
        for(i=0; i<num; i++){
                tmp += sqrt((point[i].x-meanx)*(point[i].x-meanx)
```

```
                                + (point[i].y-meany)*(point[i].y-meany));
        }
        tmp = tmp/(double)num;
        scale = sqrt(2.0)/tmp;
        tx = -scale * meanx;
        ty = -scale * meany;
        cvZero(T);
        cvmSet(T,0,0,scale);
        cvmSet(T,1,1,scale);
        cvmSet(T,0,2,tx);
        cvmSet(T,1,2,ty);
        cvmSet(T,2,2,1.0);

        for(i=0; i<num; i++){
                cvmSet(x,0,0,point[i].x);
                cvmSet(x,1,0,point[i].y);
                cvmSet(x,2,0,1.0);
                cvMatMul(T,x,xp);       // xp = T*x
                point[i].x = cvmGet(xp,0,0)/cvmGet(xp,2,0);
                point[i].y = cvmGet(xp,1,0)/cvmGet(xp,2,0);
        }
        cvReleaseMat(&x);
        cvReleaseMat(&xp);
}


//-------------------------------------------------------------//
// Weighted RANSAC algorithm
//-------------------------------------------------------------//
void weighted_RANSAC(CornerPair cnpair, CvMat *H, CvMat *inlier_mask)
{
        int i, j, N, sample_size, sample_count;
        int numinlier, max_numinlier;
        double curr_std_d, std_d;
        double prob, e, weight_sum, corr, uW;
        bool colinear;
        CvMat *curr_inlier_mask = cvCreateMat(cnpair.num,1,CV_64FC1);
        CvMat *curr_H = cvCreateMat(3,3,CV_64FC1);
        CvMat *Ta = cvCreateMat(3,3,CV_64FC1);
        CvMat *Tb = cvCreateMat(3,3,CV_64FC1);
        CvMat *invTb = cvCreateMat(3,3,CV_64FC1);
        CvMat *tmp_point = cvCreateMat(3,1,CV_64FC1);

        max_numinlier = -1;
        N = 500;
        sample_size = 4;
        sample_count = 0;
        prob = 0.99;
```

```
CvPoint2D64f *curr_mpointa = new CvPoint2D64f[sample_size];
CvPoint2D64f *curr_mpointb = new CvPoint2D64f[sample_size];
int *curr_idx = new int[sample_size];

weight_sum = 0;
for (i=0;i<cnpair.num;i++){
	weight_sum += cnpair.sm_val[i];
}
while(N > sample_count){
	colinear = true;
	while(colinear == true){
		colinear = false;
		for(i=0; i<sample_size; i++){
			// weighted sampling
			uW = ((double)(rand()%1000))/1000.*weight_sum;
			corr = 0;
			for(j=0; j<cnpair.num; j++){
				corr += cnpair.sm_val[j];
				if(corr>=uW){
					curr_idx[i] = j;
					break;
				}
			}
			for(j=0; j<i; j++){
				if(curr_idx[i] == curr_idx[j]){
					colinear = true;
					break;
				}
			}
			if(colinear == true) break;
			curr_mpointa[i].x = (double)cnpair.ptx[curr_idx[i]].x;
			curr_mpointa[i].y = (double)cnpair.ptx[curr_idx[i]].y;
			curr_mpointb[i].x = (double)cnpair.ptxp[curr_idx[i]].x;
			curr_mpointb[i].y = (double)cnpair.ptxp[curr_idx[i]].y;
		}
		// Colinear check
		if(colinear == false)
			colinear = ColinearCheck(sample_size, curr_mpointa);
	}

	// Nomalized DLT
	Normalize(sample_size, curr_mpointa, Ta);
	Normalize(sample_size, curr_mpointb, Tb);

	// H = invTb * curr_H * Ta
	CalculateHomography(sample_size, curr_mpointa, curr_mpointb, curr_H);
	cvInvert(Tb, invTb);
```

```
                cvMatMul(invTb, curr_H, curr_H);
                cvMatMul(curr_H, Ta, curr_H);

                // Calculate the # of inliers
                numinlier = CalculateInlierNumber(cnpair.num, cnpair.ptx, cnpair.ptxp,

        curr_H,curr_inlier_mask,&curr_std_d);
                // Update H
                if(numinlier > max_numinlier ||
                        (numinlier == max_numinlier && curr_std_d < std_d))
                {
                        max_numinlier = numinlier;
                        std_d = curr_std_d;
                        cvCopy(curr_inlier_mask, inlier_mask);
                        cvCopy(curr_H, H);
                }

                // update number N
                e = 1 - (double)numinlier / (double)cnpair.num;
                N = (int)(log(1-prob)/log(1-pow(1-e,sample_size)));
                sample_count++;
        }
        cvReleaseMat(&curr_H);
        cvReleaseMat(&curr_inlier_mask);
        cvReleaseMat(&Ta);
        cvReleaseMat(&Tb);
        cvReleaseMat(&invTb);
        cvReleaseMat(&tmp_point);
        delete curr_mpointa;
        delete curr_mpointb;
        delete curr_idx;
}
//-------------------------------------------------------------//
// Compute the homography matrix H
// solve the optimization problem min ||Ah||=0 s.t. ||h||=1
//-------------------------------------------------------------//
void ComputeH_usingSVD(CornerPair incnpair, CvMat *H)
{
        int i;
        CvMat *A = cvCreateMat(2*incnpair.num, 9, CV_64FC1);
        CvMat *U = cvCreateMat(2*incnpair.num, 2*incnpair.num, CV_64FC1);
        CvMat *D = cvCreateMat(2*incnpair.num, 9, CV_64FC1);
        CvMat *V = cvCreateMat(9, 9, CV_64FC1);
        cvZero(A);
        for(i=0; i<incnpair.num; i++){
                // 2*i row
                cvmSet(A,2*i,3,-incnpair.ptx[i].x);
                cvmSet(A,2*i,4,-incnpair.ptx[i].y);
```

```
                cvmSet(A,2*i,5,-1);
                cvmSet(A,2*i,6,incnpair.ptxp[i].y*incnpair.ptx[i].x);
                cvmSet(A,2*i,7,incnpair.ptxp[i].y*incnpair.ptx[i].y);
                cvmSet(A,2*i,8,incnpair.ptxp[i].y);
                // 2*i+1 row
                cvmSet(A,2*i+1,0,incnpair.ptx[i].x);
                cvmSet(A,2*i+1,1,incnpair.ptx[i].y);
                cvmSet(A,2*i+1,2,1);
                cvmSet(A,2*i+1,6,-incnpair.ptxp[i].x*incnpair.ptx[i].x);
                cvmSet(A,2*i+1,7,-incnpair.ptxp[i].x*incnpair.ptx[i].y);
                cvmSet(A,2*i+1,8,-incnpair.ptxp[i].x);
        }
        // SVD
        cvSVD(A, D, U, V, CV_SVD_U_T|CV_SVD_V_T);
        for(i=0; i<9; i++)
                cvmSet(H, i/3, i%3, cvmGet(V, 8, i));
        cvReleaseMat(&A);
        cvReleaseMat(&U);
        cvReleaseMat(&D);
        cvReleaseMat(&V);
}

//-------------------------------------------------------------//
// Main
//-------------------------------------------------------------//
void main(int argc, char *argv[])
{
        char input_prefix[100] = "sample";
        char input_a_name[100], input_b_name[100];
        char output_NCC_name[100],output_RANSAC_name[100];
        char output_Error_name[100], output_recon_name[100];
        int width, height, step, channels, num_ncc, num_inlier;
        IplImage *img_a, *img_b, *img_recon;
        double th_ncc, err_pow2;
        CvMat *pointxp = cvCreateMat(3,1,CV_64FC1);
        CvMat *pointx = cvCreateMat(3,1,CV_64FC1);
        int clipi, clipj, optimize_method;
        double mse = 0;
        double start_time;

        // For checking program execution time

        if( argc >= 4)
        {
                sprintf_s(input_a_name, "%s_a.jpg",argv [1]);
                sprintf_s(input_b_name, "%s_b.jpg",argv [1]);
                sprintf_s(output_NCC_name, "%s_NCC_result.jpg",argv [1]);
                sprintf_s(output_RANSAC_name, "%s_RANSAC_result.jpg",argv [1]);
```

```
        optimize_method = atoi(argv[2]);
        err_pow2 = atoi(argv[3]);
        th_ncc = TH_MATCH_NCC;
}
else if( argc == 3)
{
        sprintf_s(input_a_name, "%s_a.jpg",argv [1]);
        sprintf_s(input_b_name, "%s_b.jpg",argv [1]);
        sprintf_s(output_NCC_name, "%s_NCC_result.jpg",argv [1]);
        sprintf_s(output_RANSAC_name, "%s_RANSAC_result.jpg",argv [1]);
        optimize_method = atoi(argv[2]);
        err_pow2 = ERROR_POW2;
        th_ncc = TH_MATCH_NCC;
}
else if( argc == 2)
{
        sprintf_s(input_a_name, "%s_a.jpg",argv [1]);
        sprintf_s(input_b_name, "%s_b.jpg",argv [1]);
        sprintf_s(output_NCC_name, "%s_NCC_result.jpg",argv [1]);
        sprintf_s(output_RANSAC_name, "%s_RANSAC_result.jpg",argv [1]);
        optimize_method = OPTIMIZE_DLT;
        err_pow2 = ERROR_POW2;
        th_ncc = TH_MATCH_NCC;
}
else
{
        sprintf_s(input_a_name, "%s_a.jpg",input_prefix);
        sprintf_s(input_b_name, "%s_b.jpg",input_prefix);
        sprintf_s(output_NCC_name, "%s_NCC_result.jpg",input_prefix);
        sprintf_s(output_RANSAC_name, "%s_RANSAC_result.jpg",input_prefix);
        optimize_method = OPTIMIZE_DLT;
        err_pow2 = ERROR_POW2;
        th_ncc = TH_MATCH_NCC;
}

/* Read input image */
imga = cvLoadImage(input_a_name,-1);
imgb = cvLoadImage(input_b_name,-1);
width = imga->width;
height = imga->height;
step = imga->widthStep;
channels = imga->nChannels;

/* Create Image buffers */
img_a = cvCreateImage(cvSize(width,height),IPL_DEPTH_8U,1);
img_b = cvCreateImage(cvSize(width,height),IPL_DEPTH_8U,1);
IplImage *pImageNCC = cvCreateImage(cvSize(width*2, height), IPL_DEPTH_8U, 3);
IplImage *pImageRANSAC = cvCreateImage(cvSize(width*2, height), IPL_DEPTH_8U, 3);
```

```
        IplImage *pImageError = cvCreateImage(cvSize(width*2, height), IPL_DEPTH_8U, 3);
        IplImage *pImageRecon = cvCreateImage(cvSize(width*2, height), IPL_DEPTH_8U, 3);

        for (int i=0; i<height; i++)
                memcpy(&(pImageNCC->imageData[i * pImageNCC->widthStep]), &(imga->imageData[i
* step]), width * 3);
        for (int i=0; i<height; i++)
                memcpy(&(pImageNCC->imageData[i * pImageNCC->widthStep + width * 3]), &(imgb-
>imageData[i * step]), width * 3);
        for (int i=0; i<height; i++)
                memcpy(&(pImageRANSAC->imageData[i * pImageRANSAC->widthStep]), &(imga-
>imageData[i * step]), width * 3);
        for (int i=0; i<height; i++)
                memcpy(&(pImageRANSAC->imageData[i * pImageRANSAC->widthStep + width * 3]),
&(imgb->imageData[i * step]), width * 3);
        for (int i=0; i<height; i++)
                memcpy(&(pImageError->imageData[i * pImageError->widthStep]), &(imga-
>imageData[i * step]), width * 3);
        for (int i=0; i<height; i++)
                memcpy(&(pImageError->imageData[i * pImageError->widthStep + width * 3]),
&(imgb->imageData[i * step]), width * 3);

        /* Convert color images to gray */
        cvCvtColor(imga, img_a,CV_BGR2GRAY);
        cvCvtColor(imgb, img_b,CV_BGR2GRAY);
        cvSmooth(img_a, img_a, CV_GAUSSIAN, 3, 3, 0);
        cvSmooth(img_b, img_b, CV_GAUSSIAN, 3, 3, 0);

        /* ---------------------------------------------------------- */
        /* Finding corner points                                      */
        /* ---------------------------------------------------------- */
        FindingCorners(img_a, &cndata_x);
        FindingCorners(img_b, &cndata_xp);
        printf("num_a = %d , num_b = %d₩n", cndata_x.num, cndata_xp.num);

        /* ---------------------------------------------------------- */
        /* Corner points matching : Normalized Cross Correlation(NCC) */
        /* ---------------------------------------------------------- */
        NCC(img_a, img_b, &cndata_x, &cndata_xp, &cnpair);

        for(int i=0;i<cndata_x.num;i++)
                cvCircle(pImageNCC, cndata_x.pt[i], 2, CV_RGB(255,0,0), 2, 8, 0);
        for(int i=0 ; i<cndata_xp.num; i++){
                CvPoint pt;
                pt.x = cndata_xp.pt[i].x + width;
                pt.y = cndata_xp.pt[i].y;
                cvCircle(pImageNCC, pt, 2, CV_RGB(0,255,0), 2, 8, 0);
        }
```

```
num_ncc = 0;
for(int i=0;i<cnpair.num;i++)
{
        if( cnpair.sm_val[i]>= th_ncc)
        {
                CvPoint ptxp;
                ptxp.x = cnpair.ptxp[i].x + width;
                ptxp.y = cnpair.ptxp[i].y;
                cvLine(pImageNCC, cnpair.ptx[i], ptxp, CV_RGB(0,0,255),1,8,0);
                num_ncc++;
        }
}
cvNamedWindow("NCC", CV_WINDOW_AUTOSIZE);
cvShowImage("NCC",pImageNCC);
cvSaveImage(output_NCC_name,pImageNCC);
printf("num_NCC = %d₩n", num_ncc);

/* ------------------------------------------------------------ */
/* Weighted RANSAC                                              */
/* ------------------------------------------------------------ */
CvMat *H = cvCreateMat(3,3,CV_64FC1);
CvMat *invH = cvCreateMat(3,3,CV_64FC1);
CvMat *inlier_mask = cvCreateMat(cnpair.num,1,CV_64FC1);

weighted_RANSAC(cnpair, H, inlier_mask);

num_inlier = 0;
for(int i=0; i<cnpair.num; i++){
        CvPoint ptxp;
        ptxp.x = cnpair.ptxp[i].x + width;
        ptxp.y = cnpair.ptxp[i].y;
        if(cvmGet(inlier_mask,i,0) == 1){
                cvCircle(pImageRANSAC, cnpair.ptx[i], 1, CV_RGB(0,255,0), 2, 8, 0);
                cvCircle(pImageRANSAC, ptxp, 1, CV_RGB(0,255,0), 2, 8, 0);
                cvLine(pImageRANSAC, cnpair.ptx[i], ptxp, CV_RGB(0,255,0),1, 8, 0);
                incnpair.ptx[num_inlier].x = cnpair.ptx[i].x;
                incnpair.ptx[num_inlier].y = cnpair.ptx[i].y;
                incnpair.ptxp[num_inlier].x = cnpair.ptxp[i].x;
                incnpair.ptxp[num_inlier].y = cnpair.ptxp[i].y;
                incnpair.sm_val[num_inlier] = cnpair.sm_val[i];
                num_inlier++;
                incnpair.num = num_inlier;
        }
        else{
                cvCircle(pImageRANSAC, cnpair.ptx[i], 1, CV_RGB(255,0,0), 2, 8, 0);
                cvCircle(pImageRANSAC, ptxp, 1, CV_RGB(255,0,0), 2, 8, 0);
                cvLine(pImageRANSAC, cnpair.ptx[i], ptxp, CV_RGB(255,0,0),1, 8, 0);
        }
```

```
        }
        cvNamedWindow("Weighted RANSAC", CV_WINDOW_AUTOSIZE);
        cvShowImage("Weighted RANSAC",pImageRANSAC);
        cvSaveImage(output_RANSAC_name,pImageRANSAC);
        printf("num_inlier = %d₩n", num_inlier);
#ifdef DEBUG
        printf("h0 = %f₩n",cvmGet(H,0,0));
        printf("h1 = %f₩n",cvmGet(H,0,1));
        printf("h2 = %f₩n",cvmGet(H,0,2));
        printf("h3 = %f₩n",cvmGet(H,1,0));
        printf("h4 = %f₩n",cvmGet(H,1,1));
        printf("h5 = %f₩n",cvmGet(H,1,2));
        printf("h6 = %f₩n",cvmGet(H,2,0));
        printf("h7 = %f₩n",cvmGet(H,2,1));
        printf("h8 = %f₩n",cvmGet(H,2,2));
#endif
        /* ------------------------------------------------------------ */
        /* Find Homography                                              */
        /* ------------------------------------------------------------ */

#ifdef BAD_INIT_CONDITION
        // Bad Initial Condition
        cvmSet(H,0,0,cvmGet(H,0,0)+0.001);
        cvmSet(H,0,1,cvmGet(H,0,1)+0.002);
        cvmSet(H,0,2,cvmGet(H,0,2)+0.001);
        cvmSet(H,1,0,cvmGet(H,1,0)+0.001);
        cvmSet(H,1,1,cvmGet(H,1,1)+0.002);
        cvmSet(H,1,2,cvmGet(H,1,2)+0.001);
        cvmSet(H,2,0,cvmGet(H,2,0)+0.001);
        cvmSet(H,2,1,cvmGet(H,2,1)+0.001);
        cvmSet(H,2,2,cvmGet(H,2,2)+0.002);
#endif

        double correspond_err;
        LARGE_INTEGER lFreq;
        LARGE_INTEGER lTime1;
        LARGE_INTEGER lTime2;
        long dwEllipseTime;

        QueryPerformanceFrequency(&lFreq);// 1초당count 수를얻어옴
        QueryPerformanceCounter(&lTime1);

        switch(optimize_method)
        {
                case OPTIMIZE_GD:
                        printf("GD Algorithm₩n");
                        /* ------------------------------------------------------------ */
                        /* GD Algorithm                                                 */
```

```
        /* ------------------------------------------------------------ */
        correspond_err = Gradient_Descent(H, incnpair, err_pow2);
        if(argc >= 2){
                sprintf_s(output_Error_name, "%s_error_GD.jpg",argv [1]);
                sprintf_s(output_recon_name, "%s_recon_GD.jpg",argv [1]);
        }
        else{
                sprintf_s(output_Error_name, "%s_error_GD.jpg",input_prefix);
                sprintf_s(output_recon_name, "%s_recon_GD.jpg",input_prefix);
        }
        break;

case OPTIMIZE_GN:
        printf("GN Algorithm₩n");
        /* ------------------------------------------------------------ */
        /* GN Algorithm                                                 */
        /* ------------------------------------------------------------ */
        correspond_err = Gauss_Newton(H, incnpair, err_pow2);
        if(argc >= 2){
                sprintf_s(output_Error_name, "%s_error_GN.jpg",argv [1]);
                sprintf_s(output_recon_name, "%s_recon_GN.jpg",argv [1]);
        }
        else{
                sprintf_s(output_Error_name, "%s_error_GN.jpg",input_prefix);
                sprintf_s(output_recon_name, "%s_recon_GN.jpg",input_prefix);
        }

        break;

case OPTIMIZE_LM:
        printf("LM Algorithm₩n");
        /* ------------------------------------------------------------ */
        /* LM Algorithm                                                 */
        /* ------------------------------------------------------------ */
        correspond_err = Levenberg_Marquardt(H, incnpair, err_pow2);
        if(argc >= 2){
                sprintf_s(output_Error_name, "%s_error_LM.jpg",argv [1]);
                sprintf_s(output_recon_name, "%s_recon_LM.jpg",argv [1]);
        }
        else{
                sprintf_s(output_Error_name, "%s_error_LM.jpg",input_prefix);
                sprintf_s(output_recon_name, "%s_recon_LM.jpg",input_prefix);
        }
        break;

case OPTIMIZE_DL:
        printf("DL Algorithm₩n");
        /* ------------------------------------------------------------ */
```

```
                        /*  DL Algorithm                                                */
                        /* ------------------------------------------------------------ */
                        correspond_err = Dog_Leg(H, incnpair, err_pow2);
                        if(argc >= 2){
                                sprintf_s(output_Error_name, "%s_error_DL.jpg",argv [1]);
                                sprintf_s(output_recon_name, "%s_recon_DL.jpg",argv [1]);
                        }
                        else{
                                sprintf_s(output_Error_name, "%s_error_DL.jpg",input_prefix);
                                sprintf_s(output_recon_name, "%s_recon_DL.jpg",input_prefix);
                        }
                        break;

                case OPTIMIZE_DLT:
                default :
                        printf("DLT AlgorithmWn");
                        /* ------------------------------------------------------------ */
                        /* DLT
                                                        */
                        /* ------------------------------------------------------------ */
                        ComputeH_usingSVD(incnpair, H);
                        if(argc >= 2){
                                sprintf_s(output_Error_name, "%s_error_DLT.jpg",argv [1]);
                                sprintf_s(output_recon_name, "%s_recon_DLT.jpg",argv [1]);
                        }
                        else{
                               sprintf_s(output_Error_name,  "%s_error_DLT.jpg",input_prefix);
                               sprintf_s(output_recon_name,  "%s_recon_DLT.jpg",input_prefix);
                        }
                        break;
        }
        QueryPerformanceCounter(&lTime2);
        printf("QueryPerformance = %f msecWn",(double)(lTime2.QuadPart-
lTime1.QuadPart)/(lFreq.QuadPart/1000));

#ifdef DEBUG
        printf("h0 = %fWn",cvmGet(H,0,0));
        printf("h1 = %fWn",cvmGet(H,0,1));
        printf("h2 = %fWn",cvmGet(H,0,2));
        printf("h3 = %fWn",cvmGet(H,1,0));
        printf("h4 = %fWn",cvmGet(H,1,1));
        printf("h5 = %fWn",cvmGet(H,1,2));
        printf("h6 = %fWn",cvmGet(H,2,0));
        printf("h7 = %fWn",cvmGet(H,2,1));
        printf("h8 = %fWn",cvmGet(H,2,2));
#endif

        /* ------------------------------------------------------------ */
```

```
        /* Plot x' and Hx in 'prime' image                                  */
        /* ---------------------------------------------------------------- */
        correspond_err = 0;
        for(int i=0; i<incnpair.num; i++){
                CvPoint ptxp, ptHx;
                int randR, randG, randB;
                ptxp.x = incnpair.ptxp[i].x + width;
                ptxp.y = incnpair.ptxp[i].y;
                // set X_a
                cvmSet(pointx,0,0,incnpair.ptx[i].x);
                cvmSet(pointx,1,0,incnpair.ptx[i].y);
                cvmSet(pointx,2,0,1.0);
                // compute Xp
                cvMatMul(H, pointx, pointxp);
                clipj = (int)(cvmGet(pointxp,0,0)/cvmGet(pointxp,2,0));
                clipi = (int)(cvmGet(pointxp,1,0)/cvmGet(pointxp,2,0));
                correspond_err += pow((double)(clipj-incnpair.ptxp[i].x),2)
                                                + pow((double)(clipi-incnpair.ptxp[i].y),2);
                if((clipi >= 0)&&(clipi<=height-1)&&(clipj >= 0)&&(clipj<=width-1))
                {
//                      randR = rand()%128+127;
//                      randG = rand()%128+127;
//                      randB = rand()%128+127;
                        randR = rand()%255;
                        randG = rand()%255;
                        randB = rand()%255;
                        ptHx.x = clipj + width;
                        ptHx.y = clipi;
                        cvCircle(pImageError, ptxp, 1, CV_RGB(randR,randG,randB), 2, 8, 0);
                        cvCircle(pImageError, ptHx, 1, CV_RGB(randR,randG,randB), 2, 8, 0);
                        cvLine(pImageError, ptxp, ptHx, CV_RGB(randR,randG,randB),1, 8, 0);
                }
        }
        correspond_err = sqrt(correspond_err);
        printf("correspondence error = %f₩n",correspond_err);
        cvNamedWindow("x' Hx plot", CV_WINDOW_AUTOSIZE);
        cvShowImage("x' Hx plot",pImageError);
        cvSaveImage(output_Error_name,pImageError);

        /* ---------------------------------------------------------------- */
        /* Reconstruct Image                                                */
        /* ---------------------------------------------------------------- */
        img_recon = cvCloneImage(imgb);
        cvZero(img_recon);

        for (int i=0; i<imga->height; i++)
        for (int j=0; j<imga->width; j++){
                // set X_a
```

```
                cvmSet(pointx,0,0,(double)j);
                cvmSet(pointx,1,0,(double)i);
                cvmSet(pointx,2,0,1.0);
                // compute Xp
                cvMatMul(H, pointx, pointxp);
                clipi = (int)(cvmGet(pointxp,1,0)/cvmGet(pointxp,2,0));
                clipj = (int)(cvmGet(pointxp,0,0)/cvmGet(pointxp,2,0));
                if((clipi >= 0)&&(clipi<=height-1)&&(clipj >= 0)&&(clipj<=width-1))
                        cvSet2D(img_recon,i,j,cvGet2D(imgb,clipi,clipj));
        }

        cvNamedWindow("Reconstructed Scene", CV_WINDOW_AUTOSIZE);
        cvShowImage("Reconstructed Scene",img_recon);
        cvSaveImage(output_recon_name,img_recon);

        cvWaitKey(0);

        cvDestroyWindow("NCC");
        cvDestroyWindow("RANSAC");
        cvDestroyWindow("x' Hx plot");
        cvDestroyWindow("Reconstructed Scene");
        cvReleaseImage(&img_a);
        cvReleaseImage(&img_b);
        cvReleaseImage(&pImageNCC);
        cvReleaseImage(&pImageRANSAC);
        cvReleaseImage(&pImageError);
        cvReleaseImage(&pImageRecon);
        cvReleaseImage(&img_recon);
}
```