ECE661 HW7

Chad Aeschliman

11/06/08

1 Problem Statement

The problem is to determine the homographies between several images taken from a common camera location (but with different orientations) and then use these homographies to mosaic the images into a single wide angle image. RANSAC is to be used to determine the homographies with Levenberg-Marquardt minimization used to refine the results. Furthermore, the change in camera angle between each image should be estimated from the eigenvalues of the homography.

2 Solution

A previous homework already covered using RANSAC with LM optimization to estimate the homography between two images. To solve the problem, this previous work was used to estimate the homography between sequential pairs of images. From these homographies, it was necessary to estimate the homography from each image to the central image. The following two properties of homographies were used ($\mathrm{H}_{i\to j}$ is the homography from image i to image j):

$$\begin{aligned}\mathrm{H}_{i\to k}\mathrm{H}_{k\to j} &= \mathrm{H}_{i\to j}\\ \mathrm{H}^{-1}_{i\to j} &= \mathrm{H}_{j\to i}\end{aligned}$$

Applying these properties, the homography from image i to the center image (denoted c) is given by

$$\mathrm{H}_{i\to c} = \begin{cases}\prod_{k=i}^{c-1}\mathrm{H}_{k\to(k+1)} & i<c\\ \mathrm{I}_{3\times3} & i=c\\ \prod_{k=i}^{c+1}\mathrm{H}^{-1}{}_{(k-1)\to k} & i>c\end{cases}$$

Once the homography from every image to the center image is found, the mosaiced image can be produced. The first step is to map the four corners of each individual image to the plane of the center image using the computed homographies. A bounding box is formed around these points to in order to determine the dimensions of the output image. Also, an appropriate translation ($\vec{t} = [t_x\ t_y\ 0]^T$) between the center image plane and the output image is determined so that the lower left corner of the bounding box in the center image plane is at the 0,0 location in the output image.

The output image is built one pixel at a time by looping over all rows and columns. A pixel $\vec{x}_o$ in the output image is mapped to each image i by

$$\vec{x}_i = \mathrm{H}^{-1}_{i\to c}\left(\vec{x}_o - \vec{t}\right) \tag{1}$$

$\vec{x}_i$ is mapped to real world coordinates and if it falls within the bounds of image i then a bilinear approximation for each color channel k (denoted v_{ik})is computed using the nearest pixels. Once this process has been repeated for all of the images the final output value at pixel $\vec{x}_o$ for each color channel k is given by

$$v_k = \left(\sum_{i \in I} w_i\right)\left(\sum_{i \in I} \frac{v_{ik}}{w_i}\right)$$

where I is the set of images for which the pixel was within the bounds of the image and w_i is a weighting factor given by

$$w_i = \mathrm{d}^2\left(\vec{\tilde{x}}_i - \vec{\tilde{c}}_i\right)$$

where $\vec{\tilde{c}}_i$ is the center of image i, $\vec{\tilde{x}}_i$ is the result of equation 1 in real world coordinates, and $\mathrm{d}(\cdot)$ is euclidean distance. This weighting factor reduces the impact of the pixels in an image with the square of their distance from the center, smoothing the transition between images.

To estimate the angle difference between two images it is necessary to estimate the eigenvalues of the rotation matrix R. Because the camera center does not move significantly between images, the homography between two images is given by

$$\mathrm{H} = \mathrm{KRK}^{-1}$$

This means that (as shown in class), the eigenvalues of R are the same as the eigenvalues of H. Furthermore, it was shown in class that for rotation within a plane, the eigenvalues of R are given by

$$\begin{aligned} \lambda_1 &= 1 \\ \lambda_2 &= e^{\imath\theta} \\ \lambda_3 &= e^{-\imath\theta} \end{aligned}$$

where θ is the desired rotation angle. Using Euler's equation ($e^{\imath\theta} = \cos\theta + \imath\sin\theta$) it can be seen that

$$\theta = \tan^{-1}\left(\frac{\Im\{\lambda_2\}}{\Re\{\lambda_2\}}\right)$$

Note that this formula was used as opposed to simpler forms (e.g. $\theta = \cos^{-1}(\Re\{\lambda_2\})$) because in practice, $\lambda_2 = \alpha e^{\imath\theta}$ for some scalar $\alpha \neq 1$. This nullifies the simpler formulas which are based on only the real or imaginary part of λ_2.

Note that the eigenvalues of H were computed using the GNU Scientific Library[1].

3 Results

Five sets of example images are included at the end of this report. Figures 1-3 show three horzontal scenes constructed of nine images. Each image is fairly sharp showing that the homographies were correctly computed. Furthermore, the boundaries between images are nearly invisible showing the effectiveness of the smoothing which was used. Figure 4 shows a vertical mosaic of 11 images. Note that the image boundaries can be seen pretty clearly in the clouds because they moved between images. Figure 5 shows a row of people standing against a white wall. This example represents a difficult mosaicing problem because the only available features (the people) invariably moved slightly between images. Because of this the person on each end is blurred. Also, flash was used which resulted in dark areas near the edge of each image which can be seen in the mosaiced image. Figure 6 shows the placement of each individual image in the mosaiced images.

Table 1 (located after the code) gives the angle between images as determined from the homography for all five sets of images. From the tripod markings the estimated angle change was around 10° for the horizontal sets and 5° for the vertical set which match closely with the computed estimates.

[1] http://www.gnu.org/software/gsl/

4 Code

4.1 hw7.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <limits.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_eigen.h>
#include "opencv/cv.h"
#include "opencv/highgui.h"
#include "utilities.h"

#define NUMBER_OF_IMAGES 9
#define FIRST_IMAGE 0
#define CENTER_IMAGE 4
#define RAD2DEG 57.296

// Determines the homographies between a sequence of images
// and then mosaics the images.
int main(int argc, char** argv)
{
   char* base_filename;
   IplImage* image1;
   IplImage* image2;
   IplImage *images[NUMBER_OF_IMAGES];
   int i, image_number;
   FILE *log_file;

   CvMat *H[NUMBER_OF_IMAGES-1];
   CvMat *fullH[NUMBER_OF_IMAGES];
   CvMat *invfullH[NUMBER_OF_IMAGES];

   // read in the command line argument
   if (argc >= 2)
   {
      base_filename = argv[1];
   }
   else
   {
      printf("Usage: hw7 first_filename");
      return 1;
   }

   // load images
   char new_filename[FILENAME_MAX];
   for (image_number = 0; image_number < NUMBER_OF_IMAGES; image_number++)
   {
      strcpy(new_filename, base_filename);
      sprintf(new_filename + strlen(base_filename) - 5, "%d%s", image_number+FIRST_IMAGE,
           base_filename + strlen(base_filename) - 4);
      if ((images[image_number] = cvLoadImage(new_filename, 1)) == 0)
      {
         printf("Unable to open file %s\n", base_filename);
         return 2;
      }
   }

   // open log file
   strcpy(new_filename, base_filename);
   strcpy(new_filename + strlen(base_filename) - 5, "_log.txt");
   log_file = fopen(new_filename,"w");

   // compute sequential homographies
   for (image_number = 0; image_number < NUMBER_OF_IMAGES - 1; image_number++)
   {
      H[image_number] = cvCreateMat(3, 3, CV_64FC1);
      printf("\n%d->%d\n", image_number, image_number + 1);
      fprintf(log_file, "\n%d->%d\n", image_number, image_number + 1);
      image1 = images[image_number];
      image2 = images[image_number + 1];
      compute_homography(image2, image1, H[image_number]);
      printMatrix(H[image_number], "  H");
```

```
    // use GSL (Gnu Scientific Library) to compute the evals of H
    double raw_data[9];
    for (i=0;i<9;i++)
    {
        raw_data[i] = cvmGet(H[image_number],i/3,i%3);
    }
    gsl_matrix_view m = gsl_matrix_view_array(raw_data, 3, 3);
    gsl_vector_complex *eval = gsl_vector_complex_alloc(3);
    gsl_matrix_complex *evec = gsl_matrix_complex_alloc(3, 3);
    gsl_eigen_nonsymmv_workspace * w = gsl_eigen_nonsymmv_alloc(3);
    gsl_eigen_nonsymmv(&m.matrix, eval, evec, w);
    gsl_eigen_nonsymmv_free(w);
    gsl_eigen_nonsymmv_sort(eval, evec, GSL_EIGEN_SORT_ABS_DESC);
    double theta = 0;
    for (i=0;i<3;i++)
    {
        gsl_complex eval_i = gsl_vector_complex_get (eval, i);
        double imag_part = GSL_IMAG(eval_i);
        if (fabs(imag_part)>100.0*DBL_EPSILON)
        {
            double real_part = GSL_REAL(eval_i);
            theta = fabs(atan(imag_part/real_part));
            break;
        }
    }
    printf("   theta = %lg degrees\n",theta*RAD2DEG);
    fprintf(log_file, "   theta = %lg degrees\n",theta*RAD2DEG);
}
fclose(log_file);

// now compute the homographies relative to the center image
for (i = CENTER_IMAGE; i >= 0; i--)
{
    fullH[i] = cvCreateMat(3, 3, CV_64FC1);
    if (i == CENTER_IMAGE)
    {
        cvSetIdentity(fullH[i], cvScalarAll(1.0));
    }
    else
    {
        cvMatMul(H[i],fullH[i+1],fullH[i]);
    }
}
CvMat *invH = cvCreateMat(3, 3, CV_64FC1);
for (i = CENTER_IMAGE + 1; i < NUMBER_OF_IMAGES; i++)
{
    cvInvert(H[i - 1], invH, CV_LU);
    fullH[i] = cvCreateMat(3, 3, CV_64FC1);
    cvMatMul(invH,fullH[i-1],fullH[i]);
}

// print out the cascaded homographies
printf("\nCascaded Homographies\n");
for (i = 0; i < NUMBER_OF_IMAGES; i++)
{
    char name[256];
    sprintf(name, "H%d%d", i, CENTER_IMAGE);
    printMatrix(fullH[i], name);
    invfullH[i] = cvCreateMat(3, 3, CV_64FC1);
    cvInvert(fullH[i],invfullH[i],CV_LU);
}

// now determine what size the mosaiced image should be
// the idea is to make it large enough so that none of the
// remapped images are clipped
int min_x = INT_MAX;
int max_x = INT_MIN;
int min_y = INT_MAX;
int max_y = INT_MIN;
CvMat* input_coord = cvCreateMat(3, 1, CV_64FC1);
CvMat* output_coord = cvCreateMat(3, 1, CV_64FC1);
cvmSet(input_coord, 2, 0, 1.0);
for (i = 0; i < NUMBER_OF_IMAGES; i++)
{
    double temp_x, temp_y;
```

```
    // (0,0)
    cvmSet(input_coord, 0, 0, 0.0);
    cvmSet(input_coord, 1, 0, 0.0);
    cvMatMul(fullH[i],input_coord,output_coord);
    temp_x = (int) (cvmGet(output_coord, 0, 0) / cvmGet(output_coord, 2, 0));
    temp_y = (int) (cvmGet(output_coord, 1, 0) / cvmGet(output_coord, 2, 0));
    if (temp_x < min_x)
       min_x = temp_x;
    if (temp_y < min_y)
       min_y = temp_y;
    if (temp_x > max_x)
       max_x = temp_x;
    if (temp_y > max_y)
       max_y = temp_y;

    // (0,h)
    cvmSet(input_coord, 0, 0, 0.0);
    cvmSet(input_coord, 1, 0, images[i]->height);
    cvMatMul(fullH[i],input_coord,output_coord);
    temp_x = (int) (cvmGet(output_coord, 0, 0) / cvmGet(output_coord, 2, 0));
    temp_y = (int) (cvmGet(output_coord, 1, 0) / cvmGet(output_coord, 2, 0));
    if (temp_x < min_x)
       min_x = temp_x;
    if (temp_y < min_y)
       min_y = temp_y;
    if (temp_x > max_x)
       max_x = temp_x;
    if (temp_y > max_y)
       max_y = temp_y;

    // (w,o)
    cvmSet(input_coord, 0, 0, images[i]->width);
    cvmSet(input_coord, 1, 0, 0.0);
    cvMatMul(fullH[i],input_coord,output_coord);
    temp_x = (int) (cvmGet(output_coord, 0, 0) / cvmGet(output_coord, 2, 0));
    temp_y = (int) (cvmGet(output_coord, 1, 0) / cvmGet(output_coord, 2, 0));
    if (temp_x < min_x)
       min_x = temp_x;
    if (temp_y < min_y)
       min_y = temp_y;
    if (temp_x > max_x)
       max_x = temp_x;
    if (temp_y > max_y)
       max_y = temp_y;

    // (w,h)
    cvmSet(input_coord, 0, 0, images[i]->width);
    cvmSet(input_coord, 1, 0, images[i]->height);
    cvMatMul(fullH[i],input_coord,output_coord);
    temp_x = (int) (cvmGet(output_coord, 0, 0) / cvmGet(output_coord, 2, 0));
    temp_y = (int) (cvmGet(output_coord, 1, 0) / cvmGet(output_coord, 2, 0));
    if (temp_x < min_x)
       min_x = temp_x;
    if (temp_y < min_y)
       min_y = temp_y;
    if (temp_x > max_x)
       max_x = temp_x;
    if (temp_y > max_y)
       max_y = temp_y;
}
printf("(%d,%d)->(%d,%d)\n", min_x, min_y, max_x, max_y);

// now compute the mosaiced iamge
IplImage *corrected_image = cvCreateImage(cvSize(max_x - min_x, max_y - min_y), 8, 3);
cvZero(corrected_image);
int j, k;
double value[3];
double scale[NUMBER_OF_IMAGES];
for (i = 0; i < corrected_image->width; i++)
{
   cvmSet(input_coord, 0, 0, (double) (i + min_x));
   for (j = 0; j < corrected_image->height; j++)
   {
      double xi, yi, fx, fy, dx, dy;
      cvmSet(input_coord, 1, 0, (double) (j + min_y));
```

```
        // check which remapped images affect this pixel and compute an average
        // value to set the output pixel
        for (k = 0; k < 3; k++)
        {
           value[k] = 0;
        }
        for (image_number=0;image_number<NUMBER_OF_IMAGES;image_number++)
        {
           scale[image_number] = 0;
        }
        for (image_number = 0; image_number < NUMBER_OF_IMAGES; image_number++)
        {
           image2 = images[image_number];
           cvCopy(invfullH[image_number], invH, NULL);

           /* compute the associated image coordinate */
           cvMatMul(invH,input_coord,output_coord);
           xi = cvmGet(output_coord, 0, 0) / cvmGet(output_coord, 2, 0);
           yi = cvmGet(output_coord, 1, 0) / cvmGet(output_coord, 2, 0);

           // if outside of the image then move on
           if (xi < 0 || yi < 0 || xi >= (image2->width - 1) || yi >= (image2->height -
               1))
           {
              continue;
           }
           dx = xi - image2->width/2;
           dy = yi - image2->height/2;
           scale[image_number] = 1.0/(dx*dx + dy*dy);

           // compute the fractional component of the image coord.
           fx = xi - (int) xi;
           fy = yi - (int) yi;

           // compute the pixel value using linear interpolation
           for (k = 0; k < 3; k++)
           {
              value[k] += scale[image_number] * (1.0 - fx) * (1.0 - fy) * ((uchar*) (
                  image2->imageData + image2->widthStep * (int) yi))[((int) xi) * 3 + k];
              value[k] += scale[image_number] * (1.0 - fx) * fy * ((uchar*) (image2->
                  imageData + image2->widthStep * (int) (yi + 1)))[((int) xi) * 3 + k];
              value[k] += scale[image_number] * fx * (1.0 - fy) * ((uchar*) (image2->
                  imageData + image2->widthStep * (int) yi))[((int) (xi + 1)) * 3 + k];
              value[k] += scale[image_number] * fx * fy * ((uchar*) (image2->imageData +
                  image2->widthStep * (int) (yi + 1)))[((int) (xi + 1)) * 3 + k];
           }
        }
        double den = 0;
        for (image_number=0;image_number<NUMBER_OF_IMAGES;image_number++)
        {
           den += scale[image_number];
        }
        if (den > 0)
        {
           for (k = 0; k < 3; k++)
           {
              ((uchar*) (corrected_image->imageData + corrected_image->widthStep * j))[i
                  * 3 + k] = (uchar)(value[k]/den);
           }
        }
     }
  }
  strcpy(new_filename, base_filename);
  sprintf(new_filename + strlen(base_filename) - 5, "%s", base_filename + strlen(
       base_filename) - 4);
  cvSaveImage(new_filename,corrected_image);

  // draw in the image boundaries
  for (i = 0; i < NUMBER_OF_IMAGES; i++)
  {
     // (0,0)
     cvmSet(input_coord, 0, 0, 0.0);
     cvmSet(input_coord, 1, 0, 0.0);
     cvMatMul(fullH[i],input_coord,output_coord);
     double LL_x = -min_x + (int) (cvmGet(output_coord, 0, 0) / cvmGet(output_coord, 2,
         0));
```

```
        double LL_y = -min_y + (int) (cvmGet(output_coord, 1, 0) / cvmGet(output_coord, 2,
            0));

        // (0,h)
        cvmSet(input_coord, 0, 0, 0.0);
        cvmSet(input_coord, 1, 0, images[i]->height);
        cvMatMul(fullH[i],input_coord,output_coord);
        double UL_x = -min_x + (int) (cvmGet(output_coord, 0, 0) / cvmGet(output_coord, 2,
            0));
        double UL_y = -min_y + (int) (cvmGet(output_coord, 1, 0) / cvmGet(output_coord, 2,
            0));

        // (w,o)
        cvmSet(input_coord, 0, 0, images[i]->width);
        cvmSet(input_coord, 1, 0, 0.0);
        cvMatMul(fullH[i],input_coord,output_coord);
        double LR_x = -min_x + (int) (cvmGet(output_coord, 0, 0) / cvmGet(output_coord, 2,
            0));
        double LR_y = -min_y + (int) (cvmGet(output_coord, 1, 0) / cvmGet(output_coord, 2,
            0));

        // (w,h)
        cvmSet(input_coord, 0, 0, images[i]->width);
        cvmSet(input_coord, 1, 0, images[i]->height);
        cvMatMul(fullH[i],input_coord,output_coord);
        double UR_x = -min_x + (int) (cvmGet(output_coord, 0, 0) / cvmGet(output_coord, 2,
            0));
        double UR_y = -min_y + (int) (cvmGet(output_coord, 1, 0) / cvmGet(output_coord, 2,
            0));

        cvLine(corrected_image,cvPoint(LL_x,LL_y),cvPoint(UL_x,UL_y),CV_RGB(255,0,0),2,CV_AA
            ,0);
        cvLine(corrected_image,cvPoint(UL_x,UL_y),cvPoint(UR_x,UR_y),CV_RGB(255,0,0),2,CV_AA
            ,0);
        cvLine(corrected_image,cvPoint(UR_x,UR_y),cvPoint(LR_x,LR_y),CV_RGB(255,0,0),2,CV_AA
            ,0);
        cvLine(corrected_image,cvPoint(LR_x,LR_y),cvPoint(LL_x,LL_y),CV_RGB(255,0,0),2,CV_AA
            ,0);
    }
    strcpy(new_filename, base_filename);
    sprintf(new_filename + strlen(base_filename) - 5, "_outlines%s", base_filename + strlen
        (base_filename) - 4);
    cvSaveImage(new_filename,corrected_image);


    return 0;
}
```

4.2 utilities library (mostly code from previous homeworks)

4.2.1 utilities.h

```
#ifndef UTILITIES_H_
#define UTILITIES_H_

#define MAX_NUM_CORNERS 4000

void printMatrix(CvMat* M, const char* name);

void compute_base_correspondences(IplImage* image1, IplImage* image2,
        CvPoint corners1[MAX_NUM_CORNERS], CvPoint corners2[MAX_NUM_CORNERS],
        int *number_of_correspondences);

void compute_ransac_correspondences(CvPoint corners1[MAX_NUM_CORNERS],
        CvPoint corners2[MAX_NUM_CORNERS],
        int number_of_correspondences,
        CvPoint inlier_set1[MAX_NUM_CORNERS],
        CvPoint inlier_set2[MAX_NUM_CORNERS],
        int *number_of_inliers,
        CvMat *best_H);

void compute_homography(IplImage *image1, IplImage *image2, CvMat *H);

#endif /* UTILITIES_H_ */
```

4.2.2 print_matrix.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <float.h>
#include "opencv/cv.h"


// utility function which prints out a matrix
void printMatrix(CvMat* M, const char* name)
{
   int indent_size;
   int i, j;
   int rows = M->rows;
   int cols = M->cols;

   // print the matrix name
   indent_size = printf("%s = ", name);

   // print out the matrix
   for (i = 0; i < rows; i++)
   {
      // start of a row
      if (i==0)
         printf("[");
      for (j = 0; j < cols; j++)
      {
         if (j > 0)
         {
            printf(",");
         }
         printf("%11.5lg", cvmGet(M, i, j));
      }
      if (i==rows-1)
         printf("]\n");
      else
         printf("\n");

      // indent the next line
      for (j = 0; j < indent_size; j++)
      {
         printf(" ");
      }
   }
   printf("\n");
}
```

4.2.3 compute_correspondences.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "opencv/cv.h"
#include "opencv/highgui.h"
#include "utilities.h"

#define DIR_X 0
#define DIR_Y 1

#define W 5
#define K 0.04
#define USE_K 0
#define THRESHOLD_K 1700
#define THRESHOLD_EIG 25

#define MATCH_W 13
#define THRESHOLD_NCC 0.85

//#define THRESHOLD_EIG 12

//#define MATCH_W 7
//#define THRESHOLD_NCC 0.75
```

```
// compute the 3x3 Sobel gradient of a grayscale image
void computeSobelGradient(IplImage* input, IplImage* output, int direction)
{
    short sobel[3][3];
    int i,j,i2,j2;
    short temp;

    // fill in sobel matrix
    if (direction==DIR_X)
    {
        sobel[0][0] = -1;
        sobel[1][0] = -2;
        sobel[2][0] = -1;
        sobel[0][1] = 0;
        sobel[1][1] = 0;
        sobel[2][1] = 0;
        sobel[0][2] = 1;
        sobel[1][2] = 2;
        sobel[2][2] = 1;
    }
    else
    {
        sobel[0][0] = -1;
        sobel[0][1] = -2;
        sobel[0][2] = -1;
        sobel[1][0] = 0;
        sobel[1][1] = 0;
        sobel[1][2] = 0;
        sobel[2][0] = 1;
        sobel[2][1] = 2;
        sobel[2][2] = 1;
    }

    // now convolve
    cvZero(output);
    for (i=1;i<(input->width - 1);i++)
    {
        for (j=1;j<(input->height - 1);j++)
        {
            temp = 0;
            for (i2=-1;i2<=1;i2++)
            {
                for (j2=-1;j2<=1;j2++)
                {
                    temp += sobel[j2+1][i2+1]*(short)((uchar*)(input->imageData + input->
                        widthStep*(j+j2)))[i+i2];
                }
            }
            ((short*)(output->imageData + output->widthStep*j))[i] = temp;
        }
    }
}

// find the corners of an image using Harris method
int find_corners(IplImage* dx, IplImage* dy, short* corners_x, short* corners_y, int*
    corners_value)
{
    int i,j,i2,j2;

    // compute a Gaussian window of the appropriate size
    double sigma = (W*0.5 - 1)*0.3 + 0.8;
    double inv_sigma = 1.0/sigma;
    int kernel[W][W];
    for (i=-W/2;i<(W+1)/2;i++)
    {
        for (j=-W/2;j<(W+1)/2;j++)
        {
            kernel[j+W/2][i+W/2] = (int)(W*2*inv_sigma*exp(-0.5*inv_sigma*inv_sigma*(i*i+j*j)
                ));
        }
    }


    // sum squared gradients over neighborhoods and identify corners.
    int sum_dx2, sum_dy2, sum_dxy;
```

```
    double test_value;
    unsigned short corner_count = 0;
    for (i=W/2;i<(dx->width-W/2);i++)
    {
        for (j=W/2;j<(dx->height-W/2);j++)
        {
            // compute local squared gradient sum
            sum_dx2 = 0;
            sum_dy2 = 0;
            sum_dxy = 0;
            for (i2=-W/2;i2<(W+1)/2;i2++)
            {
                for (j2=-W/2;j2<(W+1)/2;j2++)
                {
                    short single_dx = ((short*)(dx->imageData + dx->widthStep*(j+j2)))[i+i2];
                    short single_dy = ((short*)(dy->imageData + dy->widthStep*(j+j2)))[i+i2];
                    // Note: scale by 1/(W^2) to ensure no overflow
                    sum_dx2 += kernel[j2+W/2][i2+W/2]*single_dx*single_dx/(W*W*kernel[W/2][W
                        /2]);
                    sum_dy2 += kernel[j2+W/2][i2+W/2]*single_dy*single_dy/(W*W*kernel[W/2][W
                        /2]);
                    sum_dxy += kernel[j2+W/2][i2+W/2]*single_dx*single_dy/(W*W*kernel[W/2][W
                        /2]);
                }
            }
            sum_dx2 = sum_dx2/(256);
            sum_dy2 = sum_dy2/(256);
            sum_dxy = sum_dxy/(256);

            // now compute whether or not this is a corner
            // and decide if we should keep it.
#if USE_K
            test_value = (sum_dx2*sum_dy2 - sum_dxy*sum_dxy)
                        -K*(sum_dx2+sum_dy2)*(sum_dx2+sum_dy2);
            if (test_value > THRESHOLD_K)
            {
#else
            double trace = sum_dx2+sum_dy2;
            test_value = 0.5*(trace - sqrt(trace*trace - 4*(sum_dx2*sum_dy2 - sum_dxy*sum_dxy
                )));
            if (test_value > THRESHOLD_EIG)
            {
#endif
                // meets the threshold, now check its neighbors
                char should_use = 1;
                int index = -1;
                for (i2=0;i2<corner_count;i2++)
                {
                    if (i-corners_x[i2] <= W/2 && corners_x[i2]-i <= W/2 &&
                         j-corners_y[i2] <= W/2 && corners_y[i2]-j <= W/2)
                    {
                        // the corner with index i2 is a near neighbor so compare
                        if (test_value > corners_value[i2])
                        {
                            // replace the other corner
                            should_use = 1;
                            index = i2;
                            break;
                        }
                        else
                        {
                            // don't use it
                            should_use = 0;
                            break;
                        }
                    }
                }
                if (should_use)
                {
                    // check if we are replacing a neighboring corner
                    if (index < 0)
                    {
                        // add as a new corner if there is room, otherwise we drop it
                        if (corner_count < MAX_NUM_CORNERS)
                        {
                            index = corner_count;
```

```
                        corner_count++;
                    }
                }
                // store the corner
                if (index >= 0)
                {
                    corners_x[index] = i;
                    corners_y[index] = j;
                    corners_value[index] = test_value;
                }
            }
        }
    }
}
    return corner_count;
}

// compute the normalized cross correlation of two matrices
double compare_squares_ncc(CvMat* template, CvMat* subimage, double template_norm, double
    subimage_norm)
{
    int i,j;
    int temp_sum = 0;
    int mean_template = 0;
    int mean_subimage = 0;
    for (i=0;i<template->cols;i++)
    {
        for (j=0;j<template->rows;j++)
        {
            mean_template += ((uchar*)(template->data.ptr + template->step*j))[i];
        }
    }
    mean_template = mean_template/(template->rows*template->cols);
    for (i=0;i<template->cols;i++)
    {
        for (j=0;j<template->rows;j++)
        {
            mean_subimage += ((uchar*)(subimage->data.ptr + subimage->step*j))[i];
        }
    }
    mean_subimage = mean_subimage/(subimage->rows*subimage->cols);
    for (i=0;i<template->cols;i++)
    {
        for (j=0;j<template->rows;j++)
        {
            temp_sum += (((uchar*)(template->data.ptr + template->step*j))[i]-mean_template)
                * (((uchar*)(subimage->data.ptr + subimage->step*j))[i]-mean_subimage);
        }
    }
    return ((double)temp_sum)/(template_norm*subimage_norm);
}


void compute_base_correspondences(IplImage* image1, IplImage* image2, CvPoint corners1[
    MAX_NUM_CORNERS], CvPoint corners2[MAX_NUM_CORNERS], int *number_of_correspondences)
{
    int n, i, j;
    short corners_x[2][MAX_NUM_CORNERS];
    short corners_y[2][MAX_NUM_CORNERS];
    int num_corners[2];
    int match_index_ncc[MAX_NUM_CORNERS];
    int match_count = 0;

    for (n=0;n<2;n++)
    {
        IplImage* image;
        IplImage* gray;

        // convert image to grayscale
        if (n==0)
        {
            image = image1;
        }
        else
        {
            image = image2;
```

```
    }
    gray = cvCreateImage( cvGetSize(image), IPL_DEPTH_8U, 1 );
    cvCvtColor( image, gray, CV_BGR2GRAY );

    // compute gradient in the x and y directions
    IplImage* dx = cvCreateImage( cvGetSize(gray), IPL_DEPTH_16S, 1 );
    IplImage* dy = cvCreateImage( cvGetSize(gray), IPL_DEPTH_16S, 1 );
    computeSobelGradient(gray,dx,DIR_X);
    computeSobelGradient(gray,dy,DIR_Y);
    cvReleaseImage(&gray);

    // find the corners
    int corners_value[MAX_NUM_CORNERS];
    num_corners[n] = find_corners(dx, dy, corners_x[n], corners_y[n], &corners_value[n])
        ;
    if (num_corners[n]==MAX_NUM_CORNERS)
    {
        printf("  HARRIS: Warning, truncated corner features!\n");
    }
    cvReleaseImage(&dx);
    cvReleaseImage(&dy);
}

// determine correspondences
double template_norm, best_match_ncc;
int best_match_index_ncc;
unsigned char first,second;
IplImage* temp_image1;
IplImage* temp_image2;
if (num_corners[0]<=num_corners[1])
{
    first = 0;
    second = 1;
    temp_image1 = image1;
    temp_image2 = image2;
}
else
{
    first = 1;
    second = 0;
    temp_image1 = image2;
    temp_image2 = image1;
}
printf("  HARRIS: %d -> %d\n",first,second);
CvMat* sub_image = cvCreateMat(MATCH_W,MATCH_W,CV_8UC1);
CvMat* template = cvCreateMat(MATCH_W,MATCH_W,CV_8UC1);
for (i=0;i<num_corners[first];i++)
{

    // assume no match
    match_index_ncc[i] = -1;

    // check that the square is inside the image
    if (corners_x[first][i] < MATCH_W/2 ||
        corners_y[first][i] < MATCH_W/2 ||
        corners_x[first][i]+MATCH_W/2 >= temp_image1->width ||
        corners_y[first][i]+MATCH_W/2 >= temp_image1->height)
    {
        continue;
    }

    // get the template sub image
    cvGetSubRect(temp_image1,template,cvRect(corners_x[first][i]-MATCH_W/2,corners_y[
        first][i]-MATCH_W/2,MATCH_W,MATCH_W));
    template_norm = sqrt(compare_squares_ncc(template,template,1.0,1.0));

    // loop over all possible matches
    best_match_ncc = 0.0;
    for (j=0;j<num_corners[second];j++)
    {
        double match_ncc;

        // check that the square is inside the image
        if (corners_x[second][j] < MATCH_W/2 ||
            corners_y[second][j] < MATCH_W/2 ||
            corners_x[second][j]+MATCH_W/2 >= temp_image2->width ||
```

```
                corners_y[second][j]+MATCH_W/2 >= temp_image2->height)
            {
                continue;
            }

            // get the comparison sub_image rectangle
            cvGetSubRect(temp_image2,sub_image,cvRect(corners_x[second][j]-MATCH_W/2,
                corners_y[second][j]-MATCH_W/2,MATCH_W,MATCH_W));
            double sub_image_norm = sqrt(compare_squares_ncc(sub_image,sub_image,1.0,1.0));

            // perform the matching
            match_ncc = compare_squares_ncc(template, sub_image, template_norm,
                sub_image_norm);

            // compare the results and update if necessary
            if (match_ncc > best_match_ncc)
            {
                best_match_ncc = match_ncc;
                best_match_index_ncc = j;
            }
        }

        // compare best matches to threshold and store
        if (best_match_ncc >= THRESHOLD_NCC)
        {
            match_index_ncc[i] = best_match_index_ncc;
            if (first==0)
            {
                corners1[match_count].x = corners_x[0][i];
                corners1[match_count].y = corners_y[0][i];
                corners2[match_count].x = corners_x[1][best_match_index_ncc];
                corners2[match_count].y = corners_y[1][best_match_index_ncc];
            }
            else
            {
                corners2[match_count].x = corners_x[1][i];
                corners2[match_count].y = corners_y[1][i];
                corners1[match_count].x = corners_x[0][best_match_index_ncc];
                corners1[match_count].y = corners_y[0][best_match_index_ncc];
            }
            match_count++;
        }
        if (match_count==MAX_NUM_CORNERS)
        {
            break;
        }
    }
    *number_of_correspondences = match_count;
}
```

4.2.4 ransac.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <float.h>
#include "opencv/cv.h"
#include "utilities.h"

#define INLIER_THRESHOLD 1.5
#define PROBABILITY_REQUIRED 0.99

// Using a base set of correspondences (contained in corners1 and corners2),
// computes various homographies from randomly generated data until one
// is found which produces enough inliers to have a high probability of
// being correct.  The inlier correspondences and the estimated (unrefined)
// homography are returned.
void compute_ransac_correspondences(CvPoint corners1[MAX_NUM_CORNERS],
        CvPoint corners2[MAX_NUM_CORNERS],
        int number_of_correspondences,
        CvPoint best_inlier_set1[MAX_NUM_CORNERS],
        CvPoint best_inlier_set2[MAX_NUM_CORNERS],
        int *number_of_inliers,
        CvMat *best_H)
{
```

```
int i, j;
int N, sample_count;

N = INT_MAX;
sample_count = 0;
int max_inliers = 0;
double best_variance = 0;
CvPoint inlier_set1[MAX_NUM_CORNERS];
CvPoint inlier_set2[MAX_NUM_CORNERS];
CvMat *H = cvCreateMat(3,3,CV_64FC1);
CvMat* image1_coord = cvCreateMat(3,1,CV_64FC1);
CvMat* image2_coord = cvCreateMat(3,1,CV_64FC1);
CvPoint points1[4];
CvPoint points2[4];
double sol_matrix[64];
double sol_vector[8];
CvMat A;
CvMat B;
cvInitMatHeader(&A,8,8,CV_64FC1,sol_matrix,CV_AUTOSTEP);
cvInitMatHeader(&B,8,1,CV_64FC1,sol_vector,CV_AUTOSTEP);
while (N > sample_count)
{
   // get 4 random correpondences
   i = 0;
   while (i<4)
   {
      int index = rand()%number_of_correspondences;

      // check for duplicate point
      int duplicate = 0;
      for (j=0;j<i;j++)
      {
         if (points1[j].x==corners1[index].x && points1[j].y==corners1[index].y)
         {
            duplicate = 1;
            break;
         }
      }
      if (duplicate)
      {
         continue;
      }

      // add correspondence to list
      points1[i].x = corners1[index].x;
      points1[i].y = corners1[index].y;
      points2[i].x = corners2[index].x;
      points2[i].y = corners2[index].y;
      i++;
   }

   // set up the problem as a matrix equation
   for (i=0;i<4;i++)
   {
      sol_matrix[2*i*8+0] = points2[i].x;
      sol_matrix[2*i*8+1] = points2[i].y;
      sol_matrix[2*i*8+2] = 1;
      sol_matrix[2*i*8+3] = 0;
      sol_matrix[2*i*8+4] = 0;
      sol_matrix[2*i*8+5] = 0;
      sol_matrix[2*i*8+6] = -points1[i].x*points2[i].x;
      sol_matrix[2*i*8+7] = -points1[i].x*points2[i].y;

      sol_matrix[(2*i+1)*8+0] = 0;
      sol_matrix[(2*i+1)*8+1] = 0;
      sol_matrix[(2*i+1)*8+2] = 0;
      sol_matrix[(2*i+1)*8+3] = points2[i].x;
      sol_matrix[(2*i+1)*8+4] = points2[i].y;
      sol_matrix[(2*i+1)*8+5] = 1;
      sol_matrix[(2*i+1)*8+6] = -points1[i].y*points2[i].x;
      sol_matrix[(2*i+1)*8+7] = -points1[i].y*points2[i].y;

      sol_vector[2*i] = points1[i].x;
      sol_vector[2*i+1] = points1[i].y;
   }
```

```
// solve the problem and copy the solution into H
CvMat *temp = cvCreateMat(8,1,CV_64FC1);
cvSolve(&A, &B, temp, CV_LU);
for (i=0;i<8;i++)
{
   cvmSet(H,i/3,i%3,cvmGet(temp,i,0));
}
cvmSet(H,2,2,1.0);
cvReleaseMat(&temp);

// H should map points from image 1 into image 2. The H can be
// checked by computing the backprojection error for each point
// correspondence
int num_inliers = 0;
double sum_distance = 0;
double sum_distance_squared = 0;
for (i=0;i<number_of_correspondences;i++)
{
   // first compute the distance between the original coordinate and the
       backprojected
   // corresponding coordinate
   cvmSet(image2_coord,0,0,corners2[i].x);
   cvmSet(image2_coord,1,0,corners2[i].y);
   cvmSet(image2_coord,2,0,1.0);
   cvMatMul(H,image2_coord,image1_coord);
   double dx = ((double)cvmGet(image1_coord,0,0)/(double)cvmGet(image1_coord,2,0))-
       corners1[i].x;
   double dy = ((double)cvmGet(image1_coord,1,0)/(double)cvmGet(image1_coord,2,0))-
       corners1[i].y;
   double distance = sqrt(dx*dx + dy*dy);

   // compare this distance to a threshold to determine if it is an inlier
   if (distance<INLIER_THRESHOLD)
   {
      // it is an inlier so add it to the inlier set
      inlier_set1[num_inliers].x = corners1[i].x;
      inlier_set1[num_inliers].y = corners1[i].y;
      inlier_set2[num_inliers].x = corners2[i].x;
      inlier_set2[num_inliers].y = corners2[i].y;
      num_inliers++;
      sum_distance += distance;
      sum_distance_squared += distance*distance;
   }
}

// check if this is the best H yet (most inliers, lowest variance in the event
// of a tie)
if (num_inliers >= max_inliers)
{
   // compute variance in case of a tie
   double mean_distance = sum_distance/((double)num_inliers);
   double variance = sum_distance_squared/((double)num_inliers-1.0) - mean_distance*
       mean_distance*(double)num_inliers/((double)num_inliers-1.0);
   if ((num_inliers > max_inliers) || (num_inliers==max_inliers && variance <
       best_variance))
   {
      // this is the best H so store its information
      best_variance = variance;
      cvCopy(H,best_H,NULL);
      max_inliers = num_inliers;
      for (i=0;i<num_inliers;i++)
      {
         best_inlier_set1[i].x = inlier_set1[i].x;
         best_inlier_set1[i].y = inlier_set1[i].y;
         best_inlier_set2[i].x = inlier_set2[i].x;
         best_inlier_set2[i].y = inlier_set2[i].y;
      }
   }
}

// update N and sample_count using algorithm 4.5
sample_count++;
if (num_inliers > 0)
{
   double epsilon = 1.0 - ((double)num_inliers)/((double)number_of_correspondences);
   double inv_epsilon = 1.0 - epsilon;
```

```
        double inv_epsilon2 = inv_epsilon * inv_epsilon;
        double inv_epsilon4 = inv_epsilon2 * inv_epsilon2;
        double log_den = log(1.0 - inv_epsilon4);
        int temp = (int) (log(1.0 - PROBABILITY_REQUIRED) / log_den);
        if (temp > 0 && temp < N)
        {
          N = temp;
        }
      }
      if (sample_count % 100000 == 0)
      {
        printf("    RANSAC: %d, %d of %d\n",max_inliers,sample_count,N);
      }
    }
    printf("  RANSAC: %d iterations\n",sample_count);
    *number_of_inliers = max_inliers;
}
```

4.2.5 compute_homography.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <limits.h>
#include "opencv/cv.h"
#include "lmfit/lmmin.h"
#include "utilities.h"

#define NO_LM 0

typedef struct {
    int number_of_inliers;
    CvPoint inlier_set1[MAX_NUM_CORNERS];
    CvPoint inlier_set2[MAX_NUM_CORNERS];
} optimization_data;


// This is the function which is to be minimized. In particular,
// the LM algorithm attempts to force the square of the entries
// in fvec to 0.
void lm_evaluate_custom(double *par, int m_dat, double *fvec,
          void *data, int *info)
{
    int i;
    CvMat* H = cvCreateMat(3,3,CV_64FC1);;
    CvMat* image1_coord = cvCreateMat(3,1,CV_64FC1);
    CvMat* image2_coord = cvCreateMat(3,1,CV_64FC1);
    optimization_data *opt_data = (optimization_data *)data;
    int number_of_inliers = opt_data->number_of_inliers;

    // first fill in the homography with the parameters in p
    for (i=0;i<9;i++)
    {
        cvmSet(H,i/3,i%3,par[i]);
    }

    // compute the distance between the "true" coordinates in image1
    // and image2 and the base coordinates
    cvmSet(image1_coord,2,0,1.0);
    for (i=0;i<number_of_inliers; i++)
    {
        double dx = par[9+2*i]-opt_data->inlier_set2[i].x;
        double dy = par[9+2*i+1]-opt_data->inlier_set2[i].y;
        fvec[4*i] = (dx*dx);
        fvec[4*i+1] = (dy*dy);
        cvmSet(image1_coord,0,0,par[9+2*i]);
        cvmSet(image1_coord,1,0,par[9+2*i+1]);
        cvMatMul(H,image1_coord,image2_coord);
        dx = cvmGet(image2_coord,0,0)/cvmGet(image2_coord,2,0)-opt_data->inlier_set1[i].x;
        dy = cvmGet(image2_coord,1,0)/cvmGet(image2_coord,2,0)-opt_data->inlier_set1[i].y;
        fvec[4*i+2] = (dx*dx);
        fvec[4*i+3] = (dy*dy);
    }
}
```

```c
// Dummy function
void lm_print_custom(int n_par, double *par, int m_dat, double *fvec,
            void *data, int iflag, int iter, int nfev)
{

}


void compute_homography(IplImage *image1, IplImage *image2, CvMat *H)
{
    int i;
    int number_of_correspondences;
    CvPoint corners1[MAX_NUM_CORNERS];
    CvPoint corners2[MAX_NUM_CORNERS];

    optimization_data data; //defined in hw5.h

    CvMat *ransac_H = cvCreateMat(3,3,CV_64FC1);
    CvMat *invH = cvCreateMat(3,3,CV_64FC1);

    // compute some correspondences using the Harris corner detector and
    // NCC using a simplified version of the code from hw3 (code is in
    // compute_correspondences.c).
    compute_base_correspondences(image1,image2,
                                 corners1, corners2,
                                 &number_of_correspondences);
    printf("  HARRIS: %d base correspondences:\n",number_of_correspondences);

    // run RANSAC on these base correspondences to get an inlier set
    // and an initial guess for the homography (code is in ransac.c)
    compute_ransac_correspondences(corners1, corners2, number_of_correspondences,
                                   data.inlier_set1, data.inlier_set2, &data.
                                       number_of_inliers, ransac_H);
    printf("  RANSAC: %d inlier correspondences:\n",data.number_of_inliers);

#if NO_LM
    cvCopy(ransac_H,H,NULL);
    return;
#endif

    // With an initial homography from RANSAC and a set of inliers, we now
    // need to run the LM algorithm to refine this homography.
    CvMat* input_coord = cvCreateMat(3,1,CV_64FC1);
    CvMat* output_coord = cvCreateMat(3,1,CV_64FC1);
    cvmSet(input_coord,2,0,1.0);
    int n_p = 9+2*data.number_of_inliers;
    double *p = malloc(n_p*sizeof(double));
    for (i=0;i<9;i++)
    {
        p[i] = cvmGet(ransac_H,i/3,i%3);
    }
    cvInvert(ransac_H,invH,CV_LU);
    for (i=0;i<data.number_of_inliers;i++)
    {
        cvmSet(input_coord,0,0,data.inlier_set1[i].x);
        cvmSet(input_coord,1,0,data.inlier_set1[i].y);
        cvMatMul(invH,input_coord,output_coord);
        p[9+2*i] = cvmGet(output_coord,0,0)/cvmGet(output_coord,2,0);
        p[9+2*i+1] = cvmGet(output_coord,1,0)/cvmGet(output_coord,2,0);
    }

    // auxiliary settings:
    lm_control_type control;
    lm_initialize_control(&control);
    control.maxcall = 200000;
    control.ftol = 1.0e-15;
    control.xtol = 1.0e-15;
    control.gtol = 1.0e-15;
    control.stepbound = 10.0;

    // perform the Levenberg-Marquardt fit
    lm_minimize(4*data.number_of_inliers, n_p, p, lm_evaluate_custom, lm_print_custom,
      &data, &control);
    printf("  LM: %s\n",lm_shortmsg[control.info]);
```

```
    // form the error minimizing homography
    for (i=0;i<9;i++)
    {
        cvmSet(H,i/3,i%3,p[i]);
    }
    free(p);
}
```

	pavilion (°)	fence (°)	street (°)	building (°)	people (°)
1→2	8.50885	8.97197	7.92820	5.80404	8.09517
2→3	8.42178	9.01538	9.11606	4.35361	8.46900
3→4	8.12453	8.01454	8.74617	5.03216	9.12146
4→5	7.40971	7.65930	7.79042	3.76927	8.20806
5→6	8.97455	10.0016	8.17724	5.7864	8.72024
6→7	7.91320	7.56248	7.58355	5.58176	9.50501
7→8	8.76608	8.11272	9.10167	3.59215	7.94005
8→9	8.46872	8.59136	8.52127	4.19301	9.34125
9→10				4.09441	
10→11				4.43358	

Table 1: Angle measurements between individual images.

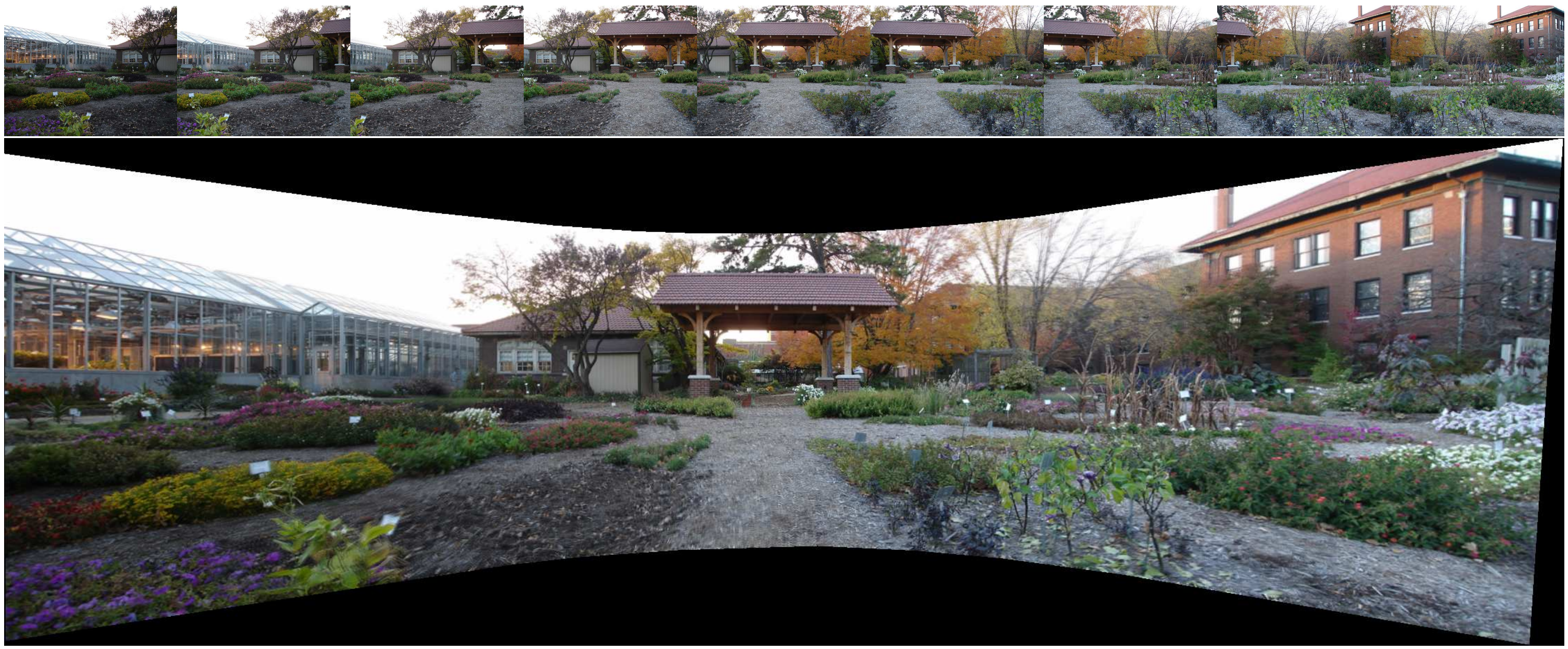

Figure 1: Individual and mosaiced images of a pavilion and surrounding garden.

Figure 2: Individual and mosaiced images of a fence/garden.

Figure 3: Individual and mosaiced images of a street with a row of buildings.

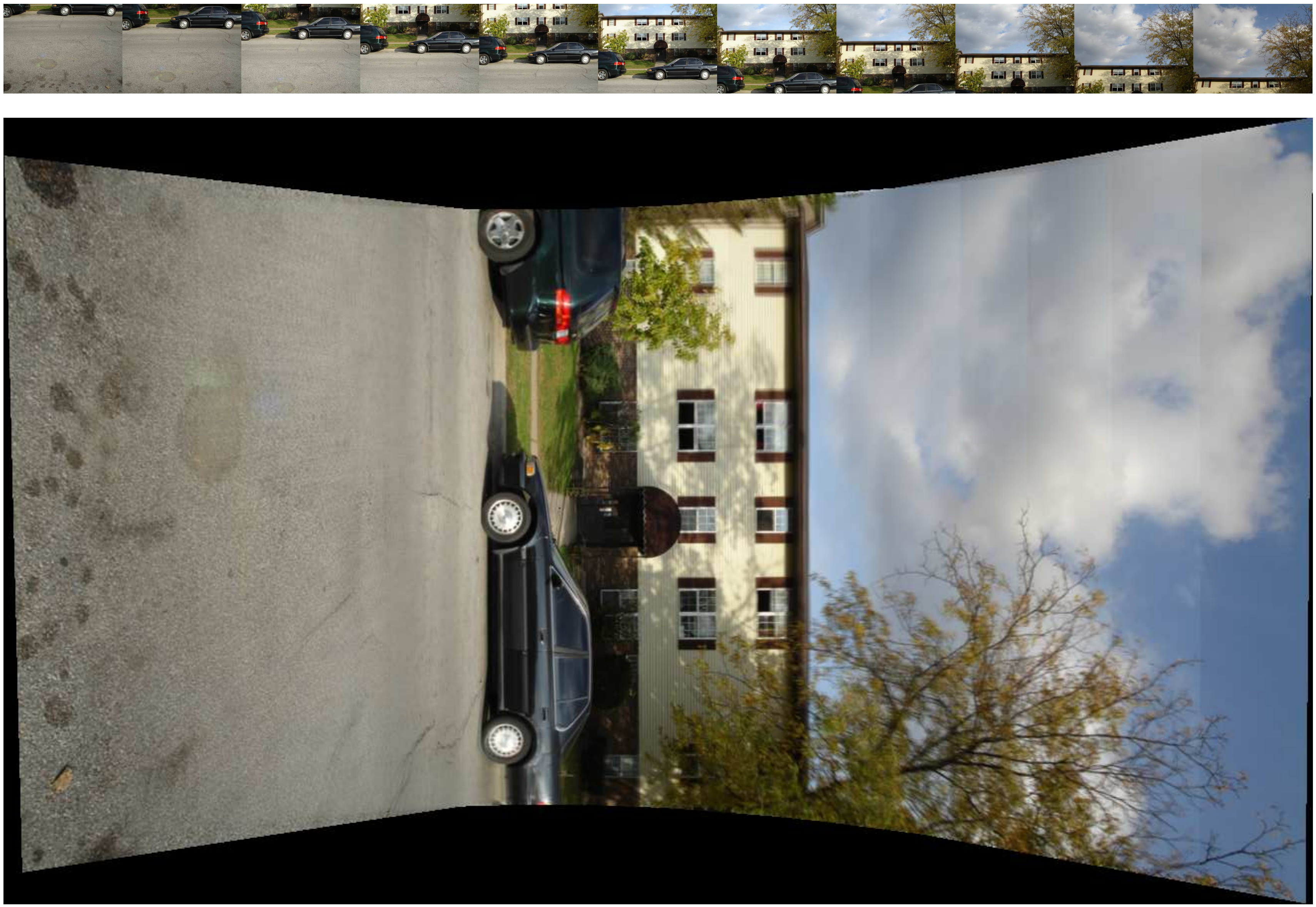

Figure 4: Individual and mosaiced images of a single building (vertical).

Figure 5: Individual and mosaiced images of a row of people.

Figure 6: The placement of the individual images in each mosaiced image.