CS 766: Computer Vision
Homework #1
Camera Projection, Calibration and Rectification

Due: September 28, 2006

General information:

- This homework is based on the readings in the textbook in Sections 3.1.1 and 3.2 plus the handout by Cipolla and Gee.
- Please type your homework.
- Please clearly identify your answer.
- Do **not** print out your MATLAB code listing. Please submit your code to the handin directory.

Problem Set:

1. CCD to Camera Transformation
   Consider a perfect perspective projection camera with focal length 24 mm and a CCD array of size 16 mm $\times$ 12 mm, containing 500 $\times$ 500 pixels.

   (a) Field of View (FOV) is defined as the angle between two points at opposite edges of the image (CCD array), either horizontally or vertically. Thus there are two FOVs, one horizontal and one vertical. Assuming the image center is the center of the image, then FOV is twice the angle between the optical axis and one edge of the image.

      i. Give a general expression for computing FOV from focal length and image width.
      ii. Compute the horizontal FOV and vertical FOV of the given camera.
      iii. Comment on how FOV affects resolution in an image.

   (b) Application

      i. Give an expression for computing the pixel coordinates of a point in a 3D scene that is given in camera-frame coordinates. Assume the upper-left corner pixel is (0,0).
      ii. Compute the pixel where a scene point at coordinates (12 m, 7 m, 103 m) is imaged.

2. Camera Projection

   (a) Prove that straight lines project to straight lines under perspective projection. You may do this by making geometric arguments using lines and planes, or else algebraically using a line parameterized as described in the section on Vanishing Points, page 8, in the Cipolla and Gee handout.

   (b) Consider a sphere of radius r with its center at camera coordinates $(x_0, 0, z_0)$, which is imaged using a perfect perspective camera with focal length f and image plane perpendicular to its optical axis. Prove, formally or informally (but convincingly), whether or not the image of the sphere is a circle.

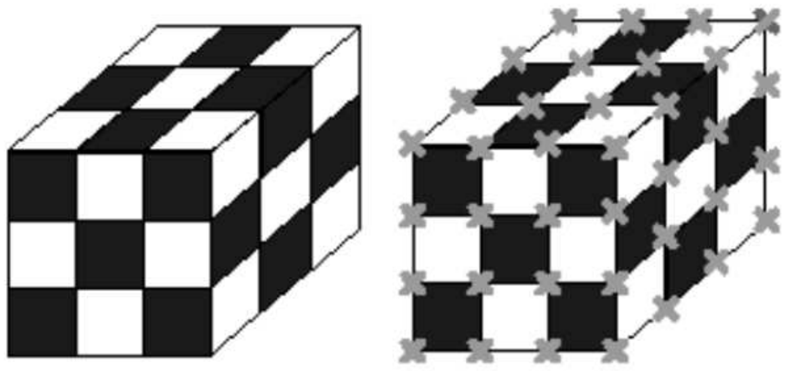

Figure 1: Input image and detected features. On the left is the original image and on the right are the detected feature points.

3. Camera Calibration
   Camera calibration is one of the most fundamental vision problems. It refers to the process of calculating the intrinsic and extrinsic parameters of the camera, which are necessary for many applications such as 3D scene reconstruction. The goal of this problem is to implement a linear calibration algorithm in MATLAB based on the method described in Section 3.2 of the textbook. Provided an input image, we want to compute the intrinsic (focal length, principal point, etc.) and extrinsic (rotation and translation) parameters of the camera used to capture this image. Assume **no radial distortion**.

   A typical way to calibrate a camera is to take a picture of a calibration object, find 2D features in the image and derive the calibration from the 2D features and their corresponding positions in 3D. In our case, we use a 2m-wide cube as a calibration object, textured with a checkerboard pattern. We detect in the image the 2D features corresponding to the corners of each tile on the checkerboard.

   Since we know its size (2m), we can find the 3D position of each 2D feature relative to the center of the cube. This process of finding correspondences is simple but time consuming, so we did this part for you. *~cs766-1/public/html/fall06/hw/hw1/Feature2D.mat* and *~cs766-1/public/html/fall06/hw/hw1/Feature3D.mat* contain the 2D corner features and the corresponding 3D positions. We also provide you readable form of the two files as *~cs766-1/public/html/fall06/hw/hw1/Feature2D.txt* and *~cs766-1/public/html/fall06/hw/hw1/Feature3D.txt*.

   (a) Linear camera calibration
   Calibrate a projective camera using a simple linear least squares approach and without taking radial distortion into account. Given a MATLAB data file that contains 3D coordinates of some points in the scene, along with their corresponding 2D projections in the image, you are to write a MATLAB function called `LinearCalib` that computes the projective camera parameters. The signature of the function should be as follows:

```
function [CamMatrix] = LinearCalib(Points3D, Points2D)

Input:
  Points2D = a (2 x N) matrix of N 2D points
  Points3D = a (4 x N) matrix of N 4D Homogeneous coordinates
Output:
  CamMatrix = 3 x 4 projective camera matrix
```

   Some implementation hints:

- One way to solve this problem is using the "direct linear transformation" algorithm in which you set up the problem as solving a linear system of the form Ap=0 where p is a 12x1 column vector of the entries for the 3 x 4 projective camera matrix and A is a 2n x 12 matrix as described in `http://www.icaen.uiowa.edu/~dip/LECTURE/3DVisionP1_2.html#knownscene` or `http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT9/node4.html` Compute the SVD of A and then the solution is the unit eigenvector with the least eigenvalue. Be sure to describe the steps of the method you used in your writeup.
- Be sure you understand how to define matrices in MATLAB.

Data: In *~cs766-1/public/html/fall06/hw/hw1*, you can find two files, *Features2D.mat* (2D projection) and *Feature3D.mat* (3D points represented in homogeneous coordinates). The 2D points in *Features2D.mat* are the projections of the 3D points in *Features3D.mat*. The following is the function to load in the data into MATLAB.

```
>> load('/p/course/cs766-dyer/public/html/fall06/hw/hw1/Features2D.mat');
>> load('/p/course/cs766-dyer/public/html/fall06/hw/hw1/Features3D.mat');
```

and the following is the method to check whether the data is loaded or not.

```
>> whos
Name      Size                     Bytes  Class

f2D       2 x 37                     592  double array
f3D       4 x 37                    1184  double array

Grand total is 222 elements using 1776 bytes
```

Handin: The values of the 3 × 4 projective camera matrix and a description of the steps you used to compute this matrix. For extra credit, decompose this matrix into $\mathbf{K}[\mathbf{R}|\mathbf{T}]$ and compute the resulting intrinsic and extrinsic parameters associated with these matrices. Section 3.2.2 in the text and page 32 of the Cipolla and Gee paper describe methods for doing this decomposition.

(b) 3D to 2D projection
You can check the accuracy of your camera calibration result by projecting the given 3D points (in homogeneous coordinates) using the camera matrix that was obtained by your linear camera calibration method. Write a function:

```
function [ProjPoints2D] = CameraProject(Points3D,CamMatrix)

Input:
  Points3D = a (4 x N) matrix of N 4D Homogeneous coordinates
  CamMatrix = 3 x 4 camera matrix
Output:
  ProjPoints2D = a (2 x N) matrix of N 2D points
```

Make sure that the points computed by `ProjPoints2D` are close to the given Points2D matrix to ensure correctness.

Handin: Transform the three 3D scene points: (0, 0, 0), (1, 1, 1), (5, 5, 5) and three other points chosen by you onto the image plane. Comment on any errors that you find.

4. Image Rectification

   Perspective images of planar surfaces are distorted in that parallel lines do not remain parallel in general, converging to a *vanishing point*. One way of removing this perspective distortion is to first compute the plane-to-plane projective transformation that relates the given view to the view that is fronto-parallel (i.e., parallel to the image plane). Then compute the inverse of this transformation, and apply it to the input image. The result will be a synthesized image in which the scene plane appears as it would if a camera had been physically moved to directly in front of the plane. This process is usually called **image rectification**. In this section, we discuss three methods to solve this problem.

   (a) One way to do rectification is to specify the true geometry of the scene using "ground truth" measurements. For example, in the directory *~cs766-1/public/images/hw1/*, there are several images showing perspectively distorted planes. In the image *hall1.pgm*, the inside door frame dimensions were physically measured to be 91 cm $\times$ 182 cm. The following is the procedure to do image rectification:

      i. Use an image viewing program such as **xv**, **ImageMagick**, **PhotoShop** to view an image and hand select corner points of the door to determine their image coordinates (which are displayed as the user interactively moves the cursor over the input image). Convert both the image coordinates and the physical coordinates to homogeneous coordinates and set up the problem in the form $\mathbf{x'_i} = \mathbf{H}\mathbf{x_i}$ as given on page 33 of the handout by Cipolla and Gee, where $\mathbf{H}$ is the 3 $\times$ 3 projective camera transformation for this homography, $x'_i = [su_i, sv_i, s]^T$ are the homogeneous coordinates of an image point at (u_i, v_i) that corresponds to scene point x_i which has homogeneous coordinates $[x_i, y_i, 1]^T$ and scene coordinates (x_i, y_i).
      Hint: the physical coordinate is set up by choosing an arbitrary origin and x, y, z coordinate. However, the length unit must correspond to the physical measurement of the door.

      ii. Convert the problem to the form $\mathbf{A}\mathbf{h}$, where $\mathbf{h}$ is a column vector $[h_{11}, h_{12}, \ldots, h_{33}]^T$ containing the nine entries of the matrix $\mathbf{H}$. Given $n \geq 4$ pairs of point correspondences, $\mathbf{A}$ is a $2n\times$ 9 matrix. Each point correspondence will result in two rows of $\mathbf{A}$. These rows are specified by rewriting the relation in terms of nonhomogeneous coordinates, giving two equations as shown on page 31 of the handout, except in our case there is no Z term because we are considering only a planar scene.
      Attention: Even though $\mathbf{H}$ is only defined up to a scale factor, **do not** set one value (usually h_{33}) to 1 because MATLAB does this for you when you call SVD.

      iii. Use MATLAB to solve for the linear least-squares solution of $\mathbf{A}\mathbf{h} = 0$, avoiding the obvious solution $\mathbf{h} = 0$, using singular value decomposition (SVD). Note, of course, that the solution will only be an estimate of the true parameter values because of noise in the measurements. From the singular value decomposition (SVD) of $\mathbf{A}$, the eigenvector corresponding to the smallest eigenvalue is the solution. To compute this in MATLAB do:

```
% solve the homogeneous system Ah = 0
[U,S,V] = svd(A);
h = V(:, size(V,2);
```

      SVD is described in the textbook on page 264. Given the projective matrix, $\mathbf{H}$, just computed, next compute the inverse of the matrix, $\mathbf{H^{-1}}$, using MATLAB's function *inv()*. Then use this matrix to compute the result image, i.e., compute $\mathbf{x} = \mathbf{H^{-1}}\mathbf{x'}$. See the bottom of page 35 in the Cipolla and Gee handout for more information on this step.

   (b) The method used in (a) depends, in general, on the coordinate frame in which points are expressed, so the result is not invariant to similarity transformations of the image. To make the result invariant to arbitrary scale and coordinate origin, and also to improve the numerical accuracy of the results, data normalization is, in general, important. The procedure is as follows:

      i. Normalize the image data and scene data as follows: Translate and scale the image points' coordinates so that the centroid of all these points is $(0, 0)$ and their average distance from the origin is approximately $\sqrt{2}$ (so that the "average" point has homogeneous coordinates of magnitude $(1, 1, 1)^T$). Similarly, normalize the planar scene points so that their centroid is $(0, 0)$ and their average distance from the origin is $\sqrt{2}$ (Note: This approach for normalizing the scene data is only suitable when the set of points is compactly distributed in 3D.)

ii. Use SVD as in (a) to compute the projection matrix **H**.

iii. "Denormalize" the values in **H** by applying the inverse point transformations used in the first step. That is, if $\widetilde{x}_i = Tx_i$ are the normalized image points and $\widetilde{X}_i = UX$ are the normalized scene points, then the denormalized projection matrix for the original coordinates is obtained from **H** by $\mathbf{H'} = \mathbf{T^{-1}HU}$.

(c) Another method for image rectification is based on the idea that the location and orientation of the *vanishing line* (aka *horizon line*) associated with a scene plane determines the true 3D orientation of the plane with respect to the camera's optical axis. That is, when the equation of the vanishing line in the image is $ax + by + c = 0$, the normal to the *scene plane*, in camera coordinates, is the *unit vector*

$\mathbf{n} = \frac{(a,b,\frac{c}{f})}{\|a,b,\frac{c}{f}\|}$

where f is the estimated focal length of the camera and $\|a, b, c\|$ denotes the Euclidean norm. In other words, $\mathbf{n}$ is the vector (a, b, c) normalized so that $a^2 + b^2 + c^2 = 1$. For a fronto-parallel scene plane, which is what we want to create an image of, the normal must be made parallel to the z-axis of the camera by rotating the scene plane. The following is the procedure:

i. Estimating the vanishing line is usually computed by

A. Computing two or more vanishing points associated with sets of parallel lines contained within this plane. (A line in one set is not parallel to a line in any other set.) The vanishing line is the line that passes through (or near) all of these vanishing points.

There are a variety of methods for automatically estimating vanishing points; for one, see the paper "Vanishing point detection" by C. Rother at
http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/ROTHER1/CVonline.html

For this assignment, you can estimate vanishing points *manually* by hand-selecting a set of points on each of a set of parallel lines in the image. Choose an image with lines that are parallel in the scene plane but have significantly different slopes in the input image meaning the view is very oblique; this will ensure that the vanishing points are easier to locate. To improve accuracy, zoom into your image so that the point coordinates are determined as accurately as possible, and choose points that are as far apart as possible.

B. Once you have the coordinates of several points on a single line, compute the equation of the line and then find an approximate intersection point of all the lines parallel to it.

C. Estimate the line passing through all of the vanishing points, determining the parameters (a, b, c). Again, to ensure that the transformation matrix is a rotation matrix, normalize the vector $(a, b, c/f)$ to be a unit vector.

ii. Now you can use the equation given earlier for computing the plane's unit normal oriented towards the image ($c \geq 0$). To bring this vector into coincidence with the positive z-axis requires a rotation by angle $acos(\mathbf{n}.(0, 0, 1))$ about the axis $n \times (0, 0, 1)$. The effects of this camera rotation on the image can be simulated by an invertible projective transformation in the image plane, specified in homogeneous coordinates by

$$\begin{bmatrix} su'_i \\ sv'_i \\ s \end{bmatrix} = \begin{bmatrix} E & F & a \\ F & G & b \\ -a & -b & c \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$E = \frac{a^2c+b^2}{a^2+b^2}, F = \frac{ab(c-1)}{a^2+b^2}, G = \frac{a^2+b^2c}{a^2+b^2}$

Compute this matrix from your input parameters, a, b, and c, invert the matrix, and then apply it to the input image. The result image should look like the camera was pointing directly at the scene plane (though expect quite a lot of error because of the crude method used for estimating the vanishing line). Instead of using raw pixel coordinates, use pixel coordinates relative to the center of the image, which will improve the numerical conditioning of the rotation matrix. To do this, say for example your image is 640 × 480 with raw image coordinates relative to the upper-left corner of the image.

Then you should first convert image coordinates (u, v) to coordinates $((u-320),(v-240))$ For more information on this method, see the paper by R. Collins and J. Beveridge, "Matching perspective views of coplanar structures using projective unwarping and similarity matching," *Proc. Computer Vision and Pattern Recognition Conf.*, 1993, pp. 240-245, available at *http://www.cs.wisc.edu/~cs766-1/readings/collins93.pdf*

(d) How many degrees of freedom are there if we used an affine transformation instead of a projectivity to describe a plane-to-plane transformation? What geometric invariants hold for an affine transformation that do not hold in general for a projective transformation? Give an example of a transformed planar square that is possible under a projective transformation that is **not** possible under an affine transformation. Under what viewing conditions will an affine transformation be an appropriate model for describing a plane-to-plane transformation?

(e) Extra Credit: For extra credit, implement vanishing point detection from a set of line segments manually selected by the user. For more information on this, see the method by Bob Collins, as described in *http://www.cs.wisc.edu/~dyer/cs766/hw/hw1/vanishing.txt* Necessary numerical code is also available in *http://www.cs.wisc.edu/~dyer/cs766/hw/hw1/jacobi.txt*

- **Test Images**

  There are a variety of test images in the directory *~cs766-1/public/images/hw1/*

  - For parts (a) and (b), you must compute results for at least the two images *hall1.pgm* and *edwardVI.pgm*, plus a third image of your choice. The third image can be one of the other ones in the same directory or else search the web to find an interesting image to use, for example of the side of a building. The second image is from the painting "King Edward VI" by William Scrots in 1546 and is an example of an artistic technique called "anamorphosis," which means a distorted image produced optically or with mirrors. (In our case we're considering only plane-to-plane transformations, which is called "plane anamorphosis.") For more information on this technique, including other paintings, see websites on anamorphosis such as *http://www.artborder.com/anamorph.html* *http://www.anamorphosis.com* and *http://www.mathsyear2000.org/explorer/anamorphic/plane-anamorphosis/* For the second image you'll have to experimentally guess at its real dimensions, which are approximately square.
  - For part (c), use at least the two images *Checkerboard.pgm* and *RailRoad.pgm*, plus a third image of your choice. The third image can be one of the other ones in the same directory or else search the web to find one.
  - **Include in your report for parts (a), (b), (c)**:
    - A description of your procedure and the parameters in each step, e.g., the **H** matrix for part (a).
    - The rectified image for each method.
    - Other information you think is critical for this method.

- Note: Most of the provided images are in *pgm(ascii)* format. To see how to read and write *pgm*, *ppm*, and *pbm* images, use the Unix command *more* to view an existing ASCII image file. For example, a *pbm* image has the magic word "P1" in the first line; the second line lists the number of columns and then the number of rows in the image (in ASCII); and the remaining lines list the image pixel values in raster scan order (left-to-right, top-to-bottom), where each value is a 0 or 1 separated by white space. A *pgm(raw)* image has the magic word "P5" in the first line; the second line contains the number of columns and number of rows; the third line contains the maximum graylevel value in the image; and the remaining lines are the successive pixel values in raster scan order. In *raw* format each pgm pixel is 1 byte and in *ascii* format each pgm pixel is 4 bytes. The magic word for *pgm(ascii)* images is "P2". For more complete information on these formats, see, for example, *http://astronomy.swin.edu.au/~pbourke/dataformats/ppm/*