B1 Information Engineering Mini-project 2016-17

Calibrating a digital camera for computer vision

Michaelmas 2016
Rev: October 26, 2016

Ron Daniel
ron.daniel@eng.ox.ac.uk

Backing up

As you may wish to work on your own machine as well as on those in the Department, it is vital that you back up your work and keep track of the latest version. We suggest treating the Departmental copy as your master copy (as is it backed up properly!).

Take special care when copying back updated material to your main repository (*e.g.* the Department). Suppose you have worked on `prog.m` on your machine and want to replace it in the Department. Don't immediately overwrite it. Instead make a reserve copy.

```
mv prog.m prog.Nov10
cp /mykey/prog.m .
```

Never rely on a memory stick for backup. Sticks are great for transporting stuff from A to B, but flash memory can die suddenly. You should anyway be able to transport material back and forth using secure ftp.

For those who wish to be really professional, there are a number of free CVS repositories (look up 'version control' on Wikipedia). An example is `SourceTree` to be found at `https://www.sourcetreeapp.com`. It is a valuable skill to learn how to use version control, particularly for your fourth year project. If you do use such a system, remember you can also maintain a history of any Latex (see `https://www.latex-project.org`) also note that Word may, or may not, be problematic because of embedded control characters in Word files. Note: All large businesses use some form of document control similar to a CVS - think of your CV!

Not everyone is comfortable in writing Matlab code. There are many texts available that will introduce you to good programming in Matlab - the one I suggest to my students is 'Essential Matlab' by B.H. Hahn and D.T. Valentine published by Academic Press. There are also many on-line tutorial guides.

1 Introduction

The aim of this project is to illustrate the use of optimization to build a real-world model of a camera suitable for use in Computer Vision.

Suppose that you wish to use a digital camera (be it a video or still camera, or even a mobile phone) to record images of objects and to measure some of the object's features reasonably accurately. You will need some way to characterise the measuring tool (your camera) so that any measurements can be used properly.

There are a number of ways that this task could be performed:

- You could use existing code. The Toolbox `http://www.vision.caltech.edu/bouguetj/calib_doc` is and example that you may wish to look at.
- Use a calibration service - Google '`camera calibration service`' to get an idea of the number of companies offering this service.

Calibration is only really suitable for fixed focus cameras, as varying the focus changes the important physical parameters that you need to capture. So we are going to assume that you have turned off the auto-focus of your camera and you are not changing the zoom setting. Your task is to write a simple calibration package for your camera.

Project specification

The (imaginary) design scenario for this project is:

- You wish to post pictures on Facebook or Ebay taken with your mobile phone (or any other **fixed focus** digital camera).
- Your recipient needs to be able to measure distances between important features on the object reliably.
- You know where these features are on the object and how far away they are from the camera when the image was taken (another part of this imaginary larger project is to slave a Microsoft Kinect to your camera to estimate depth).

- You need to be able to post information about your camera, that with the depth information, gives a reasonably accurate estimate of where the points are in space with respect to the camera.
- The aim of the project is to write code in Matlab to calibrate your camera, so that a Matlab script could be posted with your image. The recipient can then reconstruct distances between features from the image (perhaps to be executed by Octave on-line - see `http://octave-online.net`).

You will be writing algorithms to minimise various cost functions and will be designing mechanisms for inputting data and viewing data. You will be considering how to describe the quality of your resulting model and how to describe the limits on performance to a potential user (the recipient of your image).

You will only have 10 pages to write up your project. Your report is to focus on the technical aspects of the design of the algorithms your propose to use, and how you have tested and characterised their performance limitations. Imagine that your report is the technical white paper underpinning a business plan to launch a new App or Web-service - the final version may not even be implemented in Matlab.

Your code will undoubtedly need a GUI to input data. The GUI will not be the main focus of your report but there should be an indication of how your program inputs data and the nature of the output.

It is important that your code is well structured and designed. Comments should be informative and be consistent with the logical structure of your solution. Assumptions that you make about input parameters and return values that reflect the correctness of the code should be highlighted by your comments. Use comments to underpin the robustness of your code to improper input.

Here is an example of a poor comment in the code:

```
% Set i to 5
i = 5;
```

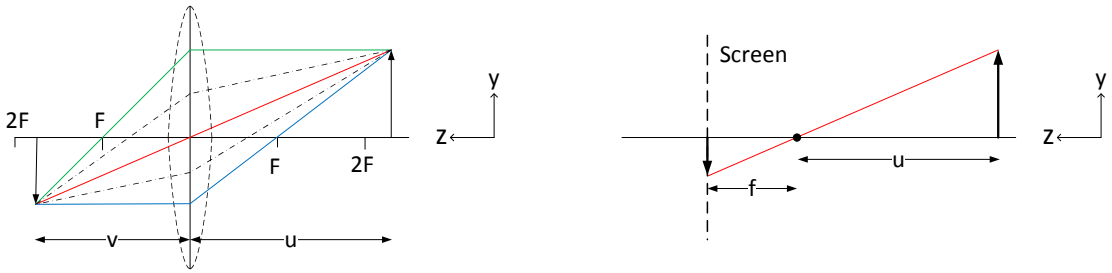
An example of an informative comment:

```
% i must be a positive integer, otherwise the function 'foo' will fail.
i = 5;
```

Please ensure your comments are informative and appropriate!

2 Simple camera models

Engineering often involves abstracting the physics of a system into models that may not have any physical justification, but merely represent the approximate behaviour of the real system. We are going to start with a simple model of a lens to represent our camera and then generate a further set of approximations and abstractions to arrive at a model to start investigating the calibration process.



(a) (b)

Figure 2.1: (a) A simple lens model. (b) A pinhole approximation.

Figure 2(a) shows a simple ray model for a single lens camera. For those of you who have never seen this simple model of a lens before, look at `http://www.physicsclassroom.com/class/refrn`, Lesson 5. The lens is assumed to define a plane whose normal, when passing through the optic centre of the lens defines a ray called the 'principal ray' and a point in the lens from which distances can be measured along this ray. Real camera lenses are not simple and do not have simple focal lengths. However the notion of a focal length is used to define a field of view when the lens is focused at infinity - this field of view being further defined by the size of the imaging element within a camera. The notion of focal length is thus a simple means of defining angles that rays make to the principal ray within the imaging system and is adopted from this simple model of a converging lens.

An object coordinate system is defined on the right with 'y' pointing upwards and 'z' pointing to the left. Light rays are shown leaving an object (a vertical arrow) hitting the lens and converging on an inverted image on the left of the lens. The solid (coloured) rays are those used in constructing the image and the dotted rays illustrate the presumed paths followed by other rays. The focal length of the lens is

F, the object is shown at distance u from the lens and the image at distance v from the lens.

Using the geometry of the ray construction (via similar triangles) we arrive at the simple lens equation (ignoring the the possibility of very close objects $u < F$)

$$\frac{1}{F} = \frac{1}{u} + \frac{1}{v} \tag{2.1}$$

This simple model predicts the following:

1. Doubling the size of the object if u is fixed doubles the size of the image.
2. If $u >> F$ doubling the size of the object and doubling u results in the image remaining the same size.
3. If $u >> F$ then doubling u will result in the image size being approximately halved.

    - These final points assume that for very large u the following approximation is reasonable:

$$v \approx F \quad \text{if} \quad u >> F \tag{2.2}$$

A smartphone camera lens might have a focal length of 35 mm. Thus assumption 3 seems reasonable for object a few metres away. Assumption 2 also seems reasonable given the nature of light rays and the size of a typical lens aperture. We might wish to reconsider assumption 1 a bit further - we may have heard of lens aberrations that can result in the geometric distortion of images. Perhaps we should return to this assumption later. But we are going to accept approximation 2.2 as being valid.

Given our small F big u assumption we now model the camera as having a screen to the left of the lens at a fixed distance. The lens is replaced by a tiny pinhole and we now have a camera obscura model of our camera. Light rays pass through the pinhole and are imaged on a screen (as shown in Figure (b)).

The new approximate model shows a screen at the focal point of the lens, the black dot representing the pinhole. There is still a single coordinate system (shown on the right). The screen is the camera chip in your phone, but the image is upside down and reversed, i.e. both x (not shown) and y swap signs. The swapping of signs is inconvenient, so physical chips have coordinates pointing from right to left and top to bottom determined by the order in which pixels are read out.

The pinhole model has in inverted image. This is still inconvient when we wish to apply, what will later turn out to be, projective geometry in analysing the camera.

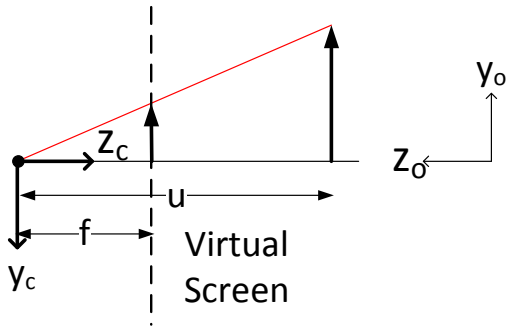


Figure 2.2: An abstract pinhole model.

We thus introduce a yet further abstraction in adopting a virtual screen in front of the lens that marks rays as they pass through the screen, hence generating an image with the same orientation as the object. Figure 2.2 is the final model that we will use for an ideal pinhole camera. There is an object to the right, with a coordinate system with y_o vertical, x_o pointing towards the viewer and z_o pointing to the left (o is for object). A camera coordinate system is located at the lens (the pinhole) with y_c pointing down, x_c pointing towards the viewer (there is an internal reflection generated by the physical pixel reading order) and z_c pointing to the right (c is for camera). There is a virtual screen distance f in front of the pinhole. All coordinates are in millimetres or metres, and the orientation of the coordinates is defined by the normal to the camera's imaging array.

2.1 Introducing pixels and the unit camera

The abstract model is still slightly inconvenient as it uses physical units to describe points in the camera. A digital camera uses light-sensitive areas called pixels. Pixels are are measured in microns and are usually smaller than the resolving power of the lens. It is natural to access images in terms of the address of the pixel in the sensor array - and most camera systems arrange that images are stored inside a computer in an array that is indexed by pixel. We thus need to determine a method representing the transformation from physical units into pixels.

Given that we wish to use a unit-less representation for 'distance' within the digital camera we will introduce two further virtual screens. A useful concept is the 'unit camera'. This is a virtual screen 1 physical unit in front of the the camera pinhole (1 millimetre if millimetres are being used). The coordinates of the unit camera are still physical and correspond to an orthogonal Cartesian coordinate system.

The ray passing through the pinhole (now called the 'camera centre') orthogonal to the imaging array's plane is called the 'principal ray'. Where this ray passes through

a virtual screen the intersection point is called that screen's principal point.

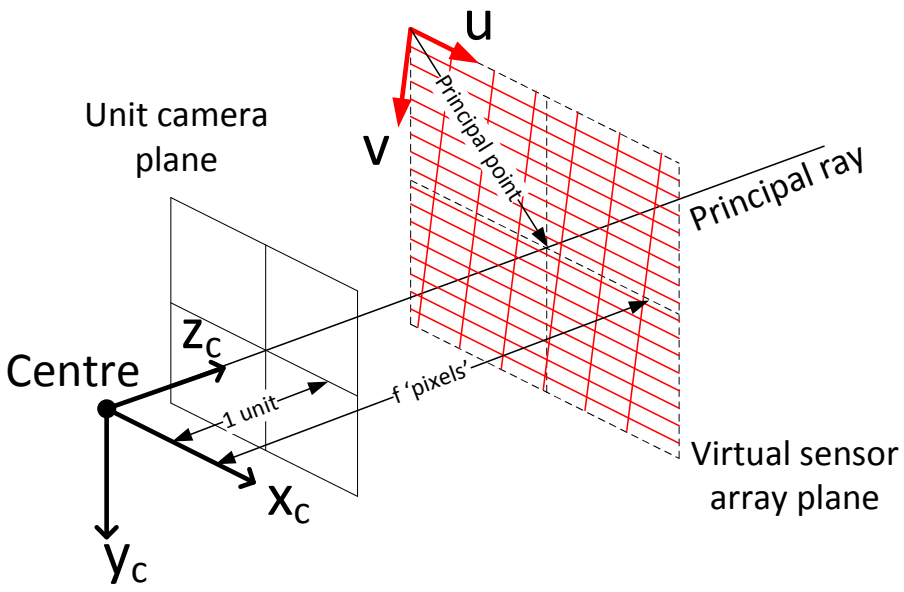


Figure 2.3: The unit camera and virtual sensor array.

Figure 2.3 illustrates the unit camera and the principal ray. We have introduced a further virtual screen at an unspecified *physical* distance, called f, in front of the camera in 'pixel units' - we will be using this distance in the same manner as manufacturers of cameras use 'focal length' to define the characteristics of a lens, i.e. this is the distance of the virtual screen in pixels from the pinhole that a screen must be placed to intersect all the focused rays from infinity that hit the imaging sensor (an array of electronic light-sensitive devices, such as a CCD). Each sensing element defines a pixel. This screen uses pixels as its coordinates, the pixel coordinates being (u, v). The origin of the pixel coordinates is not this screen's principal point, but is located at pixel $\{0, 0\}$ in the array of sensors and is at the top left of the virtual 'pixel' screen. The Principal point is thus at an offset in pixel coordinates. Further, the physical pixels may not be square, thus an x-pixel step may not have the same physical length as a y-pixel step. Also, the pixels may not be rectangular - they may be skewed, i.e. the pixels may be parallelograms. But the pixels define a local coordinate system and the position of a point in this second virtual screen can be expressed by counting pixels in the x and y directions. Pixels are shown as (red) parallelograms and I have chosen to align the u-axis with the x_c-axis and the v-axis is slightly out of line with y_c.

Suppose we have a point on the object $\mathbf{p}_c$, expressed in the camera coordinate frame. This point will define a ray passing through the camera's centre and the point $\mathbf{p}_c$. This ray will intersect both the Unit camera plane and the virtual sensor array plane.

We now calculate the position of the intersection point in the virtual sensor plane given the position in the unit camera plane. To perform this calculation we note the following 'distances':

'pixels' is an unspecified unit of 'distance' in the real world. The virtual sensor array plane is f of these 'pixels' in front of the camera centre. These 'pixels' define an orthogonal coordinate system in the virtual sensor plane with equal length steps in the x and y directions.

u-pixel is a unit step in the u-direction in 'pixels'. For example the horizontal width of a sensor element might be 1.3 'pixels'.

v-pixel is a unit step in the v-direction in 'pixels'. For example the length of the off-vertical side of a sensor element might be 0.9 'pixels'.

The $u-v$ coordinate system does not necessarily have equal length steps and is not necessarily orthogonal as the pixels may not be square - they may be rectangular or even parallelograms.

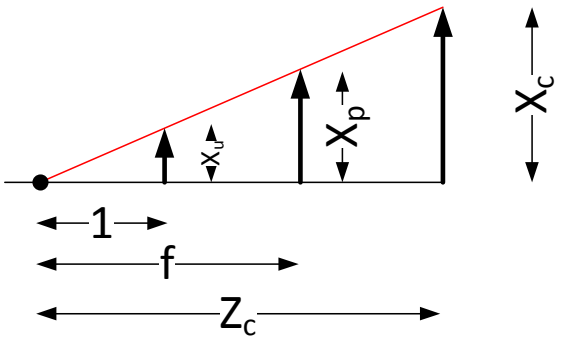


Figure 2.4: Similar triangles arrangement in 'x'.

Let the coordinates of $\mathbf{p}_c$ be (x_c, y_c, z_c) and the coordinates of the intersection point in the unit camera plane by $(x_u, y_u, 1)$ (the plane is unit distance from the centre along $\mathbf{z}_c$) and the position in the sensor array in 'pixels' as (x_p, y_p, f). Then by similar triangles (as illustrated for the x-coordinate in Figure 2.4)

$$\begin{bmatrix} x_u \\ y_u \\ 1 \end{bmatrix} = \frac{1}{z_c} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \tag{2.3}$$

and

$$\begin{bmatrix} x_p \\ y_p \\ f \end{bmatrix} = f \times \begin{bmatrix} x_u \\ y_u \\ 1 \end{bmatrix} = \frac{f}{z_c} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \tag{2.4}$$

We now need to transform from 'pixels' into u and v, i.e. how far to step in sensor elements in the real array from $(0, 0)$; this involves scaling the 'pixel' coordinates and translating by the principal point vector $\mathbf{t}$ (expressed in u and v steps). We also note that u will depend on x_p and y_p, v will only depend on y_p. As the pixels are not square we have

$$\begin{aligned} u &= \alpha \times x_p + \gamma \times y_p + t_u \\ v &= \beta \times y_p + t_v \end{aligned} \tag{2.5}$$

The numbers α, β and γ are the number of 'pixels' to move per step in real physical sensing elements. It is rather inconvenient having these multiplies and adds - so we represent this multiply and add as a matrix multiply in an extra dimension, appending a row $[0, 0, \frac{1}{f}]$ on the bottom.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & \gamma & \frac{t_u}{f} \\ 0 & \beta & \frac{t_v}{f} \\ 0 & 0 & \frac{1}{f} \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ f \end{bmatrix} = \frac{1}{z_c} \begin{bmatrix} \alpha f & \gamma f & t_u \\ 0 & \beta f & t_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \tag{2.6}$$

The product αf is often called the 'x focal length in pixels', the product βf the 'y focal length in pixels' and the product γf is called skewness (often assumed to be zero). But they are not really focal lengths - they arise from skew coordinate changes between the unit camera and the physical sensor array. The vector (t_u, t_v) is the position vector of the principal point in sensor element steps.

2.1.1 Focal length and camera phones

We have used a single lens as our model of a camera. This is gross simplification in that a real camera will have a lens with many elements and the effective focal length can be much shorter than the lens body. The effective focal length of an iPhone 6 is 29mm - which is longer than the width of the phone!

A camera's focal length is a measure of the angle that rays approaching the camera sensor make with the principal axis when the lens is focussed at infinity. A camera

phone has a small sensor close to the lens and not at 'f' from the 'lens'. Most camera phones are designed to generate an image the same size as a 35mm 'film' placed at the focal point of the lens - simulating the behaviour of a traditional physical film camera that used a 35mm format, which is 36mm by 24mm. (Its name come from an old format called '135 film'.)

2.2 Homogeneous coordinates and transformations

The above, somewhat tedious, algebra is a gentle introduction to the subject of Projective Geometry. Projective Geometry is the study of rays through the origin and their transformations. The coordinates used to describe rays are called 'homogeneous coordinates', or 'Points' with a capital 'P'. Equation 2.6 could be written as

$$\begin{bmatrix} uz_c \\ vz_c \\ z_c \end{bmatrix} = \begin{bmatrix} U \\ V \\ z_c \end{bmatrix} = \begin{bmatrix} \alpha f & \gamma f & t_u \\ 0 & \beta f & t_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \tag{2.7}$$

The vector (U, V, z_c) is another representation of the homogeneous vector $(u, v, 1)$ and represents a ray that passes through all 3D points $(\lambda u, \lambda v, \lambda)$ and is called a projective Point.

The matrix

$$\mathbf{K} = \begin{bmatrix} \alpha f & \gamma f & t_u \\ 0 & \beta f & t_v \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

is called the camera 'K-matrix' or 'intrinsic model' and describes how the image in the unit camera is transformed into the real camera. For information: this matrix is in fact an 'affine' transformation of the plane (a sub-group of the set of projective transformations). The matrix is a model of an ideal pin-hole camera. Our task is to estimate this matrix.

2.2.1 Rigid body transformations in 3D

We introduced the notion of homogeneous coordinates when transforming equations of the form $\mathbf{x}_2 = \mathbf{A}\mathbf{x}_1 + \mathbf{b}$ in 2D into $\mathbf{x'}_2 = \mathbf{A}\mathbf{x'}_1$ in 3D, but the coordinates now being homogeneous Points.

A general rigid body transformation in 3D is of the form

$$\mathbf{x}_2 = \mathbf{R}\mathbf{x}_1 + \mathbf{t} \tag{2.9}$$

where $\mathbf{R}$ is a 3×3 rotation matrix and $\mathbf{t}$ is a translation vector i.e. all rigid body transformations in 3D are of the form 'rotation + translation'. We can thus perform the same trick in 3D by pretending that position vectors are 4D, but are homogeneous Points in a larger space. The resulting representation is

$$\begin{bmatrix} \mathbf{x}_2 \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ 1 \end{bmatrix} \tag{2.10}$$

where $\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$ is a 4×4 homogeneous rigid body transformation matrix.

Now imagine pointing the camera at an object. We have defined a coordinate system for the camera in figure 2.3 and the homogeneous coordinates of object Points may be transformed into homogeneous Points in the camera coordinates by multiplying by some homogeneous transformation matrix. But the result is a set of 4-element homogeneous Points. We can remove the '1' by multiplying by a 3×4 matrix, as in

$$\mathbf{x}_2 = \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ 1 \end{bmatrix} \tag{2.11}$$

The resulting matrix is a representation of a projection from 3D into 2D of a point on an object (in the object's coordinate system) via a pinhole camera. This model is called the 'extrinsic model of the camera' (extrinsic means it is not part of the camera but is a description of the world outside the camera, as opposed to intrinsic, which is a property of the camera alone).

2.2.2 The extrinsic and intrinsic camera model

The product of the matrices in equation 2.11 is called the 'Extrinsic' model of the camera. The model depends on the coordinates of the object at which you are pointing the camera. The full model of the camera is the product of the intrinsic and extrinsic components, thus

$$z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \tag{2.12}$$

The model describes how homogeneous Points in the real word map into homogeneous Points in the camera.

3 Building a camera model in Matlab

Any good set of numerical tools must possess some mechanism for testing their performance. Your first task is to write code to model a simple camera. Here is a specification for the program you are to write:

The program is to use the following parameters (the square brackets mean upper and lower limits).

Chip width in x-pixels is an integer in the range [200,4000].

Chip height in y-pixels is an integer in the range [300,5000].

Focal length is to be in mm and is a positive number in the range [1.0,100.0].

Effective width of a pixel in the x direction is in mm and is a positive number in the range [0.0001,0.1].

Effective height of a pixel in the y direction is in mm and is a positive number in the range [0.0001,0.1].

Skewness is a number in the range [-0.1,0.1] x-pixels.

Principal point offset in x direction is a fraction of the width of the chip and is in the range [0.25,0.75].

Principal point offset in y direction is a fraction of the height of the chip and is in the range [0.25,0.75].

The program must generate a camera K matrix whose units are pixels (x-pixel steps in the first row and y-pixel steps in the second row).

Note that expected ranges for each parameter have been specified. Programmers use the notion of 'hygiene' to trap crazy numbers that have been generated by a programming error or poor user input. These ranges are just my guesses as to the range limits for a camera phone - perhaps you can suggest more realistic limits. Your code should check its input against these ranges and generate an error if a parameter is out of range.

Note that I have used an 'effective' width and height of a pixel. By this I mean the size of a pixel should the sensor be placed at the focal point of the 'lens'. The real

size of a pixel in a camera phone is much smaller as the sensor is much closer to the lens. Take the iPhone6. Its pixel size is 1.5 micron and the sensor height is 4.8mm, thus the effective sensor size (on a 35mm camera) 1.5 * 35 / 4.8 microns, or 10.9 microns.

To help you design your first piece of code, I am going to go through three possible ways to attack the problem. There are many strategies that could be followed, but the aim is here is to guide you in developing your programming skills and provide a gentle introduction to Matlab. Consider each approach in turn and then write your own version (perhaps using your own mechanism) and comment on why you have chosen your particular strategy. Your camera model code will be used at all stages during your project development.

3.1 Writing a straight Matlab script

A Matlab script is a set of Matlab instructions that are treated as though you have typed the script instructions into the computer. It is a 'batch' file. The script executes in the same 'context' as your typing of instructions. By 'context' I mean that the script will share all the variables that you have used while typing, and any variables that you use inside the script will still be available when you carry on typing.

```
1  % A straight script called Camera SimpleScriptCameraModel.
2  % An example of a simple script with no function calls.
3  %
4  % This script expects the following parameters to exist in the Matlab
5  % environment (their names should appear in response to a 'who' command).
6  %
7  % ChipWidth - An integer describing the number of horizontal pixels.
8  % ChipHeight -An integer describing the number of vertical pixels.
9  % FocalLength - The camera focal length (between 1.0 and 100.0 mm)
10 % PixelWidth - The pixel width (between 0.001 and 0.1 mm)
11 % PixelHeight - The pixel height (between 0.001 and 0.1 mm)
12 % Skewness - The skewness in u-pixels (between -0.1 and 0.1)
13 % P_u - The offset to the principal point as a fraction of the width
14 % P_v - The offset to the principal point as a fraction of the height
15 %
16 % The resulting K-Matrix is called KMatrix.
17
18
19 % Test in the inputs satisify the design constraints
20 Frac = ChipWidth - fix(ChipWidth);
21 if Frac ~= 0
22     error('ChipWidth is not integer')
23 end
24
25 if ChipWidth < 200 || ChipWidth > 4000
```

```
26      error('ChipWidth is out of range')
27  end
28
29  Frac = ChipHeight - fix(ChipHeight);
30  if Frac ~= 0
31      error('ChipHeight is not integer')
32  end
33
34  if ChipHeight < 300 || ChipHeight > 5000
35      error('ChipHeight is out of range')
36  end
37
38  if FocalLength < 1 || FocalLength > 100.0
39      error('FocalLenght is out of range')
40  end
41
42  % Carry out the rest of the bounds checking ..... I think you get the idea
43
44  % Construct the model
45
46  % The focal length in u-pixels
47  FuPixels = FocalLength / PixelWidth;
48
49  % The focal length in v-pixels
50  FvPixels = FocalLength / PixelHeight;
51
52
53  % Construct the K-Matrix
54  KMatrix = ...
55      [ FuPixels  Skewness  P_u*ChipWidth;...
56            0      FvPixels  P_v*ChipHeight;...
57            0          0          1];
```

../Matlab/SimpleCameraModels/SimpleScriptCameraModel.m

The listing is not complete as I have not tested all the input parameters - but you can see what the requirement is.

Problems

- This script will not work if ALL of the parameters are not defined beforehand.
- Running the script will pollute the Matlab environment with variables such as `FuPixels`.
- Not using functions makes the testing rather long-winded!

This approach to design is not recommended.

Note: I have used the Matlab long line continuation symbol ... to arrange the presentation of the K-Matrix to be clear to the reader of the script.

3.2 Writing a Matlab function with a long parameter list

A Matlab function generates its own 'context' - it has no access to any of the variables you have typed into the computer. Variables are passed into the function in a parameter list and variables are passed back to the calling code in as a set of returned parameters. A function call does not leave any 'footsteps in the snow' in the calling code.

In this function structure the list of parameters is the list of names chosen to contain the variables of interest. The names in the list defining the function do not need to be the same as the names used when calling the function, but using very different names can cause the reader some confusion. If you are not familiar with the difference between a script and a function try out the script and compare it to a function. After using the script and function type 'who' into Matlab to see a list of variable names and compare.

```
1  function [ KMatrix ] = LongListCameraModel( ChipWidth, ChipHeight, ...
2      FocalLength, PixelWidth, PixelHeight,...
3      Skewness,P_u, P_v)
4  %LongListCameraModel builds a camera model taking the following input
5  %parameters.
6  % ChipWidth - An integer describing the number of horizontal pixels.
7  % ChipHeight -An integer describing the number of vertical pixels.
8  % FocalLength - The camera focal length (between 1.0 and 100.0 mm)
9  % PixelWidth - The pixel width (between 0.001 and 0.1 mm)
10 % PixelHeight - The pixel height (between 0.001 and 0.1 mm)
11 % Skewness - The skewness in u-pixels (between -0.1 and 0.1)
12 % P_u - The offset to the principal point in u-pixels
13 % P_v - The offset to the principal point in v-pixels
14 %
15 % The function returns the K-Matrix
16
17 % Test in the inputs satisify the design constraints
18 Frac = ChipWidth - fix(ChipWidth);
19 if Frac ~= 0
20     error('ChipWidth is not integer')
21 end
22
23 Frac = ChipHeight - fix(ChipHeight);
24 if Frac ~= 0
25     error('ChiHeight is not integer')
26 end
27
```

```
28 % Test the ranges
29 SimpleTestRange(ChipWidth,200,4000);
30 SimpleTestRange(ChipHeight,300,5000);
31 SimpleTestRange(FocalLength,1.0,100.0);
32 SimpleTestRange(PixelWidth,0.0001,0.1);
33 SimpleTestRange(PixelHeight,0.0001,0.1);
34 SimpleTestRange(Skewness,-0.1,0.1);
35 SimpleTestRange(P_u,0.25,0.75);
36 SimpleTestRange(P_v,0.25,0.75);
37
38 % The focal length in u-pixels
39 FuPixels = FocalLength / PixelWidth;
40
41 % The focal length in v-pixels
42 FvPixels = FocalLength / PixelHeight;
43
44
45 % Construct the K-Matrix for return
46 KMatrix = ...
47     [ FuPixels Skewness P_u*ChipWidth;...
48           0     FvPixels P_v*ChipHeight;...
49           0          0          1];
50 end
```

../Matlab/SimpleCameraModels/LongListCameraModel.m

```
1  function [] = SimpleTestRange(Parameter, MinVal, MaxVal)
2  % SimpleTestRange
3  % Tests the range of Parameter agains MinVal and MaxVal.
4  % If the test fails an error message results quoting the Parameter
5  % as an identifier and Matlab exits.
6
7  if Parameter < MinVal || Parameter > MaxVal
8      error('An input parameter, %d, was out of range',Parameter)
9  end
10
11
12 end
```

../Matlab/SimpleCameraModels/SimpleTestRange.m

Problems

- You have to get the order of the parameters right every time you call this function as the call need not use the same names as those used in the function.
- The error report is a bit cryptic - it does not tell you which parameter is out of range, only its value.

Note that I have included another function `SimpleTestRange` that generates an error message if the input parameter is out of range. Note how compact the testing is now.

3.3 Writing a Matlab function taking a single vector of parameters

This is a common strategy used when writing Matlab functions. You put all the variables into a single vector of variables and with some suggestive name, such as 'Variables'. The function then has to unpack the variables and to use them.

```
1  function [ KMatrix ] = SingleVectorCameraModel( Parameters )
2  %SingleVectorCameraModel
3  % Builds a camera model using a single vector of Parameters
4  % and returns the resulting K-Matrix in KMatrix.
5  %
6  % There must be 8 parameters passed ordered as
7  % (1) ChipWidth - An integer describing the number of horizontal pixels.
8  % (2) ChipHeight - An integer describing the number of vertical pixels.
9  % (3) FocalLength - The camera focal length (between 1.0 and 100.0 mm)
10 % (4) PixelWidth - The pixel width (between 0.001 and 0.1 mm)
11 % (5) PixelHeight - The pixel height (between 0.001 and 0.1 mm)
12 % (6) Skewness - The skewness in u-pixels (between -0.1 and 0.1)
13 % (7) P_u - The offset to the principal point as a fraction of the width
14 % (8) P_v - The offset to the principal point as a fraction of the height
15
16 % Are there 8 parameters?
17 if length(Parameters) ~= 8
18     error('There must be 8 Parameters passed')
19 end
20
21 % We can CHOOSE to create internal variables for OUR convenience
22 % we could just use Parameter(i) all the way through
23 ChipWidth = Parameters(1);
24 ChipHeight = Parameters(2);
25 FocalLength = Parameters(3);
26 PixelWidth = Parameters(4);
27 PixelHeight = Parameters(5);
28 Skewness = Parameters(6);
29 P_u = Parameters(7);
30 P_v = Parameters(8);
31
32 % Test in the inputs satisify the design constraints
33 Frac = ChipWidth - fix(ChipWidth);
34 if Frac ~= 0
35     error('ChipWidth is not integer')
36 end
37
```

```
38 Frac = ChipHeight - fix(ChipHeight);
39 if Frac ~= 0
40     error('ChiHeight is not integer')
41 end
42
43 % Test the ranges with a less cryptic tester
44 TestRange(ChipWidth,200,4000,'ChipWidth');
45 TestRange(ChipHeight,300,5000,'ChipHeight');
46 TestRange(FocalLength,1.0,100.0,'FocalLength');
47 TestRange(PixelWidth,0.0001,0.1,'PixelWidth');
48 TestRange(PixelHeight,0.0001,0.1,'PixelHeight');
49 TestRange(Skewness,-0.1,0.1,'Skewness');
50 TestRange(P_u,0.25,0.75,'P_u');
51 TestRange(P_v,0.25,0.75,'P_v');
52
53 % The focal length in u-pixels
54 FuPixels = FocalLength / PixelWidth;
55
56 % The focal length in v-pixels
57 FvPixels = FocalLength / PixelHeight;
58
59 % Construct the K-Matrix for return
60 KMatrix = ...
61     [ FuPixels Skewness P_u*ChipWidth;...
62           0     FvPixels P_v*ChipHeight;...
63           0          0         1];
64
65 end
```

../Matlab/SimpleCameraModels/SingleVectorCameraModel.m

```
1  function [] = TestRange(Parameter, MinVal, MaxVal, Name)
2  % TestRange
3  % Tests the range of Parameter agains MinVal and MaxVal.
4  % If the test fails an error message results quoting the Parameter
5  % as an identifier and Matlab exits.
6
7
8  if Parameter < MinVal || Parameter > MaxVal
9      error('Input parameter %s, value %d, was out of range', ...
10         Name,Parameter)
11 end
12
13
14 end
```

../Matlab/SimpleCameraModels/TestRange.m

The new code passes a single vector as Parameters. The code tests the length to make sure that there are as least the right number of variables and then checks the

range using a more informative test.

The examples above illustrate that there are many ways to solve the same problem and that the good use of the structures that a language provides can aid in the readability of the code as well as in debugging. The examiners will be looking for evidence of thought put into the design of your code. My code examples will become increasingly incomplete as this project description progresses, only including key points and not including much error checking - this is for you to develop.

As a final point about the usefulness of comments, set the Matlab path so that your copy of `SingleVectorCameraModel` is not grayed out in the Current Folder Window and type `help SingleVectorCameraModel` into the Command Window. Your comments at the start of the function will be displayed, i.e. instruction on how to use the function and its purpose.

4 Rigid body coordinate transformations

Now that you have built a camera model we need to consider how to generate Points that define parts of an object so that we can generate images of these points in the camera.

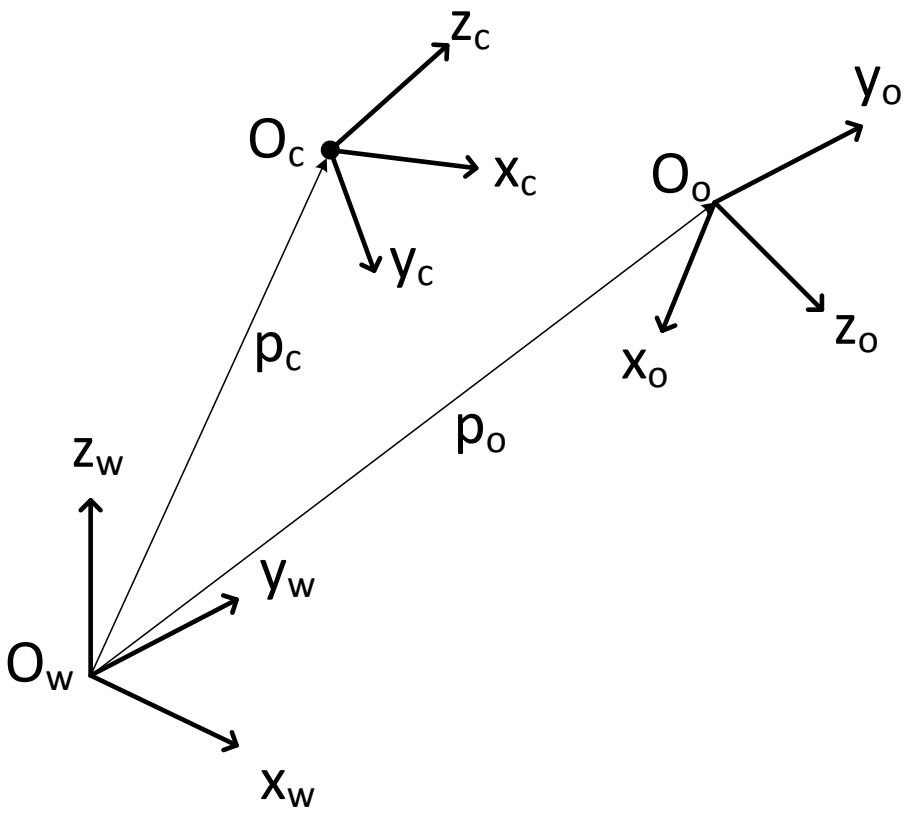


Figure 4.1: A set of coordinates.

Figure 4.1 illustrates 3 coordinate systems useful for considering how to generate images in the camera.

The world coordinates
In the bottom left, with subscript 'w', are a set of world coordinates (for example the room that contains the camera). The origin $\mathbf{O}_w$ has vector representation $\mathbf{O}_w = [0, 0, 0]$. The world coordinates have vector representations $\mathbf{x}_w = [1, 0, 0]$, $\mathbf{y}_w = [0, 1, 0]$ and $\mathbf{z}_w = [0, 0, 1]$.

The camera coordinates
The camera centre is the origin of the camera coordinates in the top left. The

position vector of the camera origin $\mathbf{O}_c$ with respect to the world coordinates is $\mathbf{p}_c = [p_{cx}, p_{cy}, p_{cz}]$. The camera coordinate axes expressed in world coordinates are $\mathbf{x}_c = [x_{cx}, x_{cy}, x_{cz}]$, $\mathbf{y}_c = [y_{cx}, y_{cy}, y_{cz}]$, $\mathbf{z}_c = [z_{cx}, z_{cy}, z_{cz}]$.

The object coordinates

The origin of the object coordinates is in the top right. The position vector of the object origin $\mathbf{O}_o$ with respect to the world coordinates is $\mathbf{p}_o = [p_{ox}, p_{oy}, p_{oz}]$. The camera coordinate axes expressed in world coordinates are $\mathbf{x}_o = [x_{ox}, x_{oy}, x_{oz}]$, $\mathbf{y}_o = [y_{ox}, y_{oy}, y_{oz}]$, $\mathbf{z}_o = [z_{ox}, z_{oy}, z_{oz}]$.

Rigid body homogeneous transformations

Equation 2.11 describes how to transform a Point in an object's coordinates into a Camera Point using the transformation matrix $\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$, a projection matrix $\begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix}$ and the K-matrix. We need to adopt a representation for which set of coordinates is being transformed into which other set of coordinates.

Let

$$\mathbf{T}_{A\to B} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \tag{4.1}$$

represent a homogeneous transformation matrix transforming coordinates of a Point in coordinate system 'A' into coordinate system 'B' (in 3D space, but each Point is represented as a homogeneous coordinate set with 4 entries).

Using the above representation

$$\mathbf{T}_{C\to W} \begin{bmatrix} x_{cx} & y_{cx} & z_{cx} & p_{cx} \\ x_{cy} & y_{cy} & z_{cy} & p_{cy} \\ x_{cz} & y_{cz} & z_{cz} & p_{cz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

represents the transformation that takes Points expressed in Camera coordinates and transforms them into World coordinates.

$$\mathbf{T}_{O\to W} \begin{bmatrix} x_{ox} & y_{ox} & z_{ox} & p_{ox} \\ x_{oy} & y_{oy} & z_{oy} & p_{oy} \\ x_{oz} & y_{oz} & z_{oz} & p_{oz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

represents the transformation that takes Points in Object coordinates and transforms them into World coordinates.

Suppose that we have a Point in World coordinates and wish to express this Point in Object coordinates, then we only need to find the transformation $\mathbf{T}_{W\to O} = \mathbf{T}^{-1}_{O\to W}$, i.e. transformation in the of the reverse representation is just the inverse of the forward transformation.

Now suppose we have a Point on an object in object coordinates $\mathbf{q}_o = [q_{xo}, q_{yo}, q_{zo}]$, and we wish to express this Point in Camera coordinates. The object Point may be expressed in World coordinates as

$$\mathbf{q}_w = \mathbf{T}_{O\to W}\mathbf{q}_o \tag{4.4}$$

We now have the Point in World coordinates, so we can express it in Camera coordinates

$$\mathbf{q}_c = \mathbf{T}_{W\to C}\{\mathbf{T}_{O\to W}\mathbf{q}_o\} = \mathbf{T}^{-1}_{C\to W}\{\mathbf{T}_{O\to W}\mathbf{q}_o\} \tag{4.5}$$

The result of the above is a transformation matrix from Object to Camera coordinates $\mathbf{T}_{O\to C}$

$$\mathbf{T}_{O\to C} = \mathbf{T}^{-1}_{C\to W}\mathbf{T}_{O\to W} \tag{4.6}$$

Matlab and matrix inverses

Matlab has the instruction `inv` that will find the inverse of a non-singular square matrix. Should you use it?

Consider equation 4.6. In Matlab we could write

```
T_oc = inv(T_cw)*T_ow;
```

where the subscripts represent the various transformations needed.

In fact the Matlab inversion algorithm does not generate a good answer to the above problem if the matrix is poorly conditioned. It is also rather slow. A much better way is to use the Matlab internal linear equation solver by writing

```
T_oc = T_cw \ T_ow;
```

The method uses the Matlab matrix backslash operator `mldivide`. See the Matlab documentation on the matrix inverse for more details.

5 Simulating a camera

You have written code to construct the K matrix of a camera. Your task is now to take a set of 3-D points and project these points into the camera that you have modeled. The process you should follow is:

1. Define a set of points in an object frame. For example, you could choose the vertices of a cube. In my solution I have defined a matrix that is $4 \times 2n$, where n is the number of vertices. Each pair of vertices, when projected into the camera, define a line that can be drawn by the Matlab `plot` routine.
2. Define the position and orientation of the object.
3. Define the position and orientation of the Camera.
4. Project the points into the camera.

5.1 How to build a rotation matrix

Once you have chosen a set of points you need to be able to define the coordinate frames $\mathbf{T}_{O \to W}$ and $\mathbf{T}_{C \to W}$. So how do we specify a rotation matrix?

5.1.1 Properties of rotation matrices

Given a rotation matrix $\mathbf{R}$ there are two key properties:

1. The inverse of a rotation matrix is its transpose, i.e. $\mathbf{R}^T\mathbf{R} = \mathbf{R}\mathbf{R}^T = \mathbf{I}$.
2. The determinant of a rotation matrix is 1, i.e. $|\mathbf{R}| = 1$.

The above implies that the columns of a rotation matrix are orthogonal and form a right-handed coordinate system. This suggests the following strategy:

1. Generate a random 3-element non-zero vector $\mathbf{x}$.
2. Divide $\mathbf{x}$ by its norm to get a unit length vector $\hat{\mathbf{x}}$ i.e. $\hat{\mathbf{x}} = \mathbf{x}/|\mathbf{x}|$.
3. Generate a second random 3-element non-zero vector $\mathbf{y}$ not equal to $\mathbf{x}$.
4. Project $\mathbf{y}$ onto $\hat{\mathbf{x}}$ and remove this component along $\hat{\mathbf{x}}$ from $\mathbf{y}$, i.e. $\mathbf{y}_2 = \mathbf{y} - [\hat{\mathbf{x}} \bullet \mathbf{y}]\,\hat{\mathbf{x}}$.

5. Divide $\mathbf{y}_2$ by its norm to get a unit vector $\hat{\mathbf{y}} = \mathbf{y}_2/|\mathbf{y}_2|$.
6. Generate a third orthogonal unit vector by finding the cross product between $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$, i.e. $\hat{\mathbf{z}} = \hat{\mathbf{x}} \times \hat{\mathbf{y}}$.
7. Form the rotation matrix by writing the 3 vectors next to each other, i.e. $R = [\hat{\mathbf{x}}\ \hat{\mathbf{y}}\ \hat{\mathbf{z}}]$.

Here is a Matlab function that performs the above

```
function [ R ] = RandomRotationMatrix()
%RandomRotationMatrix
% Generates a random rotation matrix by starting with
% two random vectors and then normalising, projecting
% out components and then generating the final
% column by using the cross-product.

% Make sure we have a random matrix larger than 'eps'
xnorm = 0;
while xnorm < eps
    x = rand(3,1);
    xnorm = norm(x);
end
% The 'random' unit vector
xhat = x / xnorm;

ynorm = 0;
while ynorm < eps
    y = rand(3,1);
    % Project out xhat by finding the dot product and removing from y
    y = y - (y'*xhat)*xhat;
    ynorm = norm(y);
end
% The 'random' unit y vector
yhat = y / ynorm;

% Find the third vector as the cross product of x and y
zhat = cross(xhat,yhat);

% Construct the rotation matrix for return
R = [xhat yhat zhat];

end
```

../Matlab/RotationMatrices/RandomRotationMatrix.m

The parameter `eps` is an internal Matlab variable that is a small number such that `1.0 + eps` evaluates to 1.0 and is a measure of 'zero' for finite precision arithmetic.

Run the above script and type the following into the Command Window:

```
>> R = RandomRotationMatrix;
>> R'R
>> det(R)
```

Do you get a unit matrix and `1.0` as expected?

Now type the following:

```
>> [U,W,V] = eig(R)
>> W = diag(W)
```

You will get some complex matrices and `W` will be the eigenvalues and `U` the corresponding eigenvectors. Notice that one of the eigenvalues will be reported as `1.0000 + 0.0000i`. This eigenvalue is essentially `1.0` and is the eigenvalue associated with the real axis of rotation, which is the real eigenvector in `U`.

The complex eigenvalues correspond to the complex numbers $e^{\pm i\theta}$ where θ is the angle of rotation.

5.1.2 Building a rotation matrix using the axis of rotation and angle of rotation

The rotation matrix built above can be described using 4 parameters: the components of a unit vector corresponding to the axis of rotation and an angle of rotation.

As a matter of interest, these 4 parameters can be used to create an algebraic object called a 'quaternion' (discovered by Hamilton in 1843) that have many of the properties of complex numbers and the unit quaternion can be thought of as a representation of a rotation matrix where

$$\hat{\mathbf{q}} = [\cos\theta/2; \sin\theta/2\{x, y, z\}] \tag{5.1}$$

We will not be using the above, but we will be using another formula discovered in the 19th century, Rodrigue's Rotation Formula.

Suppose we are given the unit length axis of rotation $\{x, y, z\}$ and the angle of rotation θ. We first construct the matrix representation, $\mathbf{K}$, of the vector cross product by the axis of rotation (a *different* K from the K-matrix!). Then

$$\mathbf{K} = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} \tag{5.2}$$

The rotation matrix is obtained by considering the projection of test vectors onto the axis of rotation and into the plane orthogonal to the axis of rotation, and the effects of the rotation on these components. After some rather tedious algebra we obtain the following expression (Rodrigue's Rotation Formula) for the rotation matrix

$$\mathbf{R} = \mathbf{I} + \sin\theta\mathbf{K} + (1 - \cos\theta)\,\mathbf{K}^2 \tag{5.3}$$

The above is not convenient for representing rotations in a compact form for optimization. The form we shall use later is called the angle-axis representation, ϕ and is defined as

$$\phi = \theta\hat{\mathbf{x}} \tag{5.4}$$

The angle θ is a positive number and is encoded as the magnitude of the vector. Thus only positive angles of rotation can be represented. If the angle of rotation is negative one has to reverse the direction of the vector $\hat{\mathbf{x}}$. In my final stages of this project I will use an modified version of the angle-axis rotation, called the Shifted Angle-Axis rotation representation where

$$\phi = (\theta + n\pi)\,\hat{\mathbf{x}} \tag{5.5}$$

This representation admits a non-zero, and hence continuous, axis of rotation as the angle passes through zero. The number n is chosen to shift problematic singularities away from the working range of the representation - I use $n = 4$ in my code. The angle and axis of rotation is represented as a single 3-element vector whose information must be unpacked in order to construct a proper rotation matrix.

Here is some code to generate a rotation matrix using Rodrigue's Rotation Formula

```
1  function [ R ] = RodriguesRotation(RotationAxis,RotationAngle )
2  %RodriguesRotation
3  % Generates a rotation matrix given the axis of rotation and the
4  % angle of rotation in radians.  This is not an angle-axis
5  % representation (where angle is encoded as the norm of the axis).
6  %
7  % RotationAxis must be a vector with 3 elements and must be non
8  % zero.  The code normalises the vector and thus it does not matter
9  % if the vector is not of unit lenth.
10 % RotationAngle is in radians.
11 %
12 % The algorithm used is Rodrigues Rotation Formula
13
14 % Peform tests on correctness of input
15 if length(RotationAxis) ~=3
16     error('The rotation axis must only have 3 elements')
```

```
17 end
18
19 NormAxis = norm(RotationAxis);
20
21 if NormAxis < eps
22     error('Cannot normalise the axis reliably');
23 end
24
25 % This step may not be necessary if the axis is already unit norm
26 RotationAxis = RotationAxis / NormAxis;
27
28 % Formulate the cross product matrix
29 K = [      0        -RotationAxis(3) RotationAxis(2); ...
30     RotationAxis(3)    0            -RotationAxis(1) ; ...
31    -RotationAxis(2) RotationAxis(1)         0];
32
33 % Compute the rotation matrix using Rodrigues Rotation Formula
34 R = eye(3) + sin(RotationAngle)*K + (1 - cos(RotationAngle))*K^2;
35
36 end
```

../Matlab/RotationMatrices/RodriguesRotation.m

5.2 Is your object point behind the camera?

Before we consider how to generate an image of an object remember that only points in front of the camera can be seen. Thus only points with a positive z-component in the camera coordinates will be visible in the camera!

You may find the following helpful when thinking about how to reason about the relative positions and directions of objects.

Some more details on 4 × 4 transformation matrices

We have presented the notion of a homogeneous transformation matrix that represents a change of coordinates of points in 3D space using homogeneous Points that have 4 entries. These matrices are of the form

$$\begin{bmatrix} x_1 & y_1 & z_1 & p_1 \\ x_2 & y_2 & z_2 & p_2 \\ x_3 & y_3 & z_3 & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} \tag{5.6}$$

Consider the 3D origin point $[x, y, z]^T = [0, 0, 0]^T$. Its homogeneous representation

is $[0, 0, 0, 1]$. How does this Point transform? The transformation is given in equation 5.6 and demonstrates that the transformation takes the origin to the vector $\mathbf{p}$.

So how do we represent a coordinate axis? A coordinate axis is a vector that is free to move about in space, provided that the vector remains parallel to its reference direction. Such 'free' vectors are represented as $[a, b, c, 0]$ (technically a Point at infinity). The x-axis is thus represented as $[1, 0, 0, 0]$ and transforms as shown in equation 5.7.

$$\begin{bmatrix} x_1 & y_1 & z_1 & p_1 \\ x_2 & y_2 & z_2 & p_2 \\ x_3 & y_3 & z_3 & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 0 \end{bmatrix} \tag{5.7}$$

We thus see that a 'free' vector representing an axis is transformed into a rotated 'free' vector representing a new x-axis. We are thus able to represent two types of object:

1. Points representing the position vectors of 3D points with their 4th element set to 1.

2. Points representing directions, or 'free' vectors, with their 4th element set to 0.

The above is included for information - our estimation task will only involve using Points with the last element set to '1'.

5.3 The simulation task

Use the components suggested above to generate an image of an object as viewed by the camera. Choose a location and orientation for the camera and a location and orientation for the object. Think about how to specify the locations of the various components so that there is a good chance that the object is largely visible in the camera.

Perhaps you don't like some of the algorithms used above. Can you find better ways of performing your task? Don't try and be too fancy - hidden line removal is not required!

Figure 5.1 is the result of running my code `RunSimulateCamera.m` that predicts the image of a 1m cube from 10m using a camera model based on the iPhone6. Note

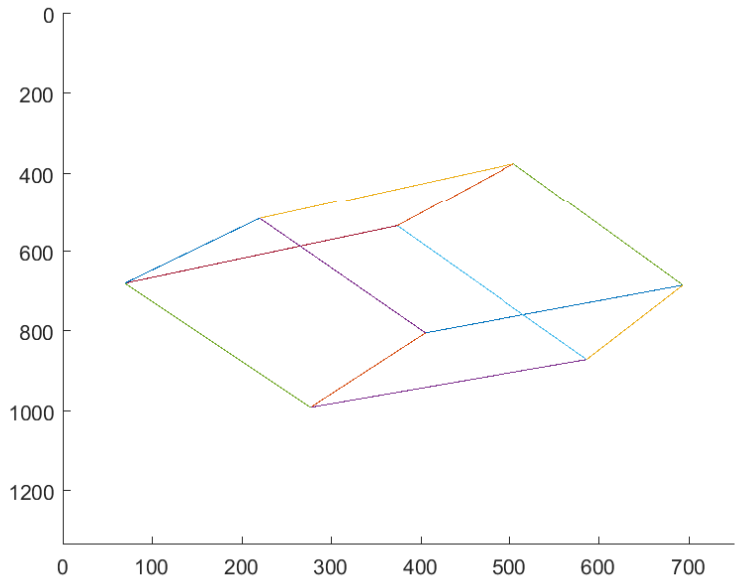


Figure 5.1: The result of simulating the viewing of a cube.

that I have used the instruction '`axis ij`' to make the y-axis run from the top of the page downwards.

Here is some of my code:

```
% Simulates viewing a 3D object (a cube) to demonstrate the
% structures built up to calibrate a camera.


% Construct a Camera
[KMatrix, CameraHeight, CameraWidth] = BuildCamera;

% Construct an object in its own frame
Cube = BuildCube;

% Position the object in space
T_ow = PositionObject;

% Position the camera so that it is likely that the object can be seen
T_cw = PositionCamera(T_ow);

% Look at what we have
ViewCamera(Cube,T_ow,KMatrix,CameraHeight,CameraWidth,T_cw)
```

../Matlab/SimulateAnObject/RunSimulateCamera.m

```
function [ T_ow ] = PositionObject()
%PositionObject
% Defines a 4x4 homogeneous transformation that places an object
% in space at a random location in world coordinates and at
% a random orientation.  T_ow means T:O->W, i.e. the transformation
% of Points in object coordinates into Points in World coordinates.
```

```
7
8  T_ow = zeros(4,4);
9
10 % The bottom row is [ 0 0 0 1]
11 T_ow(4,:) = [0 0 0 1];
12
13 % Put at some random location
14 T_ow(1:3,4) = 2000*rand(3,1);
15
16 % With a random orientation
17 T_ow(1:3,1:3) = RandomRotationMatrix;
18
19 end
```

../Matlab/SimulateAnObject/PositionObject.m

```
1  function [T_cw ] = PositionCamera(T_ow)
2  %PositionCamera
3  % Generates a 'random' camera frame that has a good chance of the
4  % object being visible in the camera when the object is a 1m cube.
5  %
6  % Input T_ow is the 4x4 object frame in world coordinates
7  % T_cw is the 4x4 camera frame in world coordinates.
8
9  % Assign space for the camera frame
10 T_cw = zeros(4);
11
12 % Set the homogeneous multiplier to 1
13 T_cw(4,4) = 1;
14
15 % extract the object origin
16 ObjectOrigin = T_ow(1:3,4);
17
18 % View the camera from about 10 metres or so (unrelated to the object
19 % frame).
20 InitialViewVector = 10000*rand(3,1);
21
22 % Define the origin of the camera frame in world coordinates
23 T_cw(1:3,4) = ObjectOrigin - InitialViewVector;
24
25 % Define the camera z-axis as pointing at the object origin
26 Normz = norm(InitialViewVector);
27 if Normz < eps
28     error('Unable to normalize the camera z-axis');
29 end
30 % Define a unit vector
31 InitialCameraz = InitialViewVector / Normz;
32
33 % Perturb the initial z axis a bit
34 CameraZ = InitialCameraz - 0.01*rand(3,1);
```

```
35
36 % ... and normalize again (no need to check norm)
37 CameraZ = CameraZ / norm(CameraZ);
38
39 % Define a random camera x-axis
40 CameraX = rand(3,1);
41 % project out the z-axis
42 CameraX = CameraX - (CameraZ'*CameraX)*CameraZ;
43 % normalize the x-axis
44 Normx = norm(CameraX);
45 if Normx < eps
46     error('Unable to normalize the camera x-axis')
47 end
48 CameraX = CameraX/Normx;
49
50 % Define the y-axis
51 CameraY = cross(CameraZ,CameraX);
52
53 % Complete the transformation matrix
54 T_cw(1:3,1:3) = [CameraX CameraY CameraZ];
55
56 end
```

../Matlab/SimulateAnObject/PositionCamera.m

```
1  function [] = ViewCamera(ObjectLines,T_ow,KMatrix,CameraHeight,CameraWidth
       , ...
2      T_cw)
3  %ViewCamera
4  % Takes an object described by a set of lines passed in ObjectLines
5  % and draws a picture of the camera's view of the object.
6  %
7  % ObjectLines is 4x2n where each column is a homogeneous Point in
8  % the objects frame.  The object is defined as pairs of Points that
9  % should have a line drawn between them.
10 % T_ow is a 4x4 homogeneous transformation matrix describing the
11 % object's frame.
12 %
13 % KMatrix is the K-Matrix of the camera in pixels.
14 % CameraHeight is the number of vertical pixels.
15 % CameraWidth is the number of horizontal pixels.
16 % T_cw is the 4x4 Camera frame in world coordinates.
17
18 % Check sizes
19 s = size(ObjectLines);
20 if s(1) ~=4 || mod(s(2),2) ~= 0
21     error('ObjectLines has an invalid size')
22 end
23
24 s = size(T_ow);
```

```
25 if s(1) ~=4 || s(2) ~= 4
26     error('T_ow has an invalid size')
27 end
28
29 s = size(KMatrix);
30 if s(1) ~=3 || s(2) ~= 3
31     error('KMatrix has an invalid size')
32 end
33
34 s = size(T_ow);
35 if s(1) ~=4 || s(2) ~= 4
36     error('T_oc has an invalid size')
37 end
38
39 % We could perform other tests to make the code bomb proof
40
41 % Transform the object into world coordinates
42 ObjectLines = T_ow*ObjectLines;
43
44 % Transform the object into camera coordinates using the backslash
45 % operator
46 ObjectLines = T_cw\ObjectLines;
47
48 % Project out the 4th coordinate and multiply by the KMatrix
49 ObjectLines = KMatrix * ObjectLines(1:3,:);
50
51 % We now have a set of homogeneous Points representing 2D points.
52 % We need to normalise these Points to get 2D points.
53 s = size(ObjectLines);
54 for j = 1:s(2)
55     ObjectLines(1:2,j) = ObjectLines(1:2,j) / ObjectLines(3,j);
56 end
57
58 % Throw away the normalising components
59 ObjectLines = ObjectLines(1:2,:);
60
61 % Genrating the image
62 figure(1)
63 clf
64
65 axis([0 CameraWidth 0 CameraHeight])
66 hold on
67 % Count through the Point pair
68 for j = 1 : 2 : s(2)
69     plot([ObjectLines(1,j) ObjectLines(1,j+1)], ...
70         [ObjectLines(2,j) ObjectLines(2,j+1)])
71 end
72
73 axis ij
```

```
74
75 end
```

../Matlab/SimulateAnObject/ViewCamera.m

6 Calibration objects

The calibration process consists of taking an object with known shape, imaging the object, and then using the known distances between points on the known object to calibrate the camera.

The most common calibration techniques use an object often referred to as a 'Tsai Tile'. This is a black and white chequerboard pattern that is either superimposed on the surfaces of a cube or on a flat plate.

There are difficulties in manufacturing a calibration cube and sticking a pattern accurately on the surface. There are also issues regarding the proper identification of features on the cube. We are thus going to use a flat plate as our calibration object.

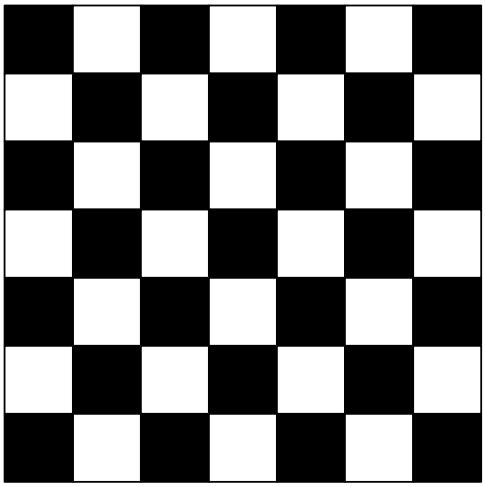


Figure 6.1: A Tsai Tile calibration pattern.

Figure 6.1 illustrates a typical Tsai Tile calibration pattern - a strong chequerboard that is easy for computer vision algorithms to find. These patterns are usually printed on paper and glued to piece of flat metal plate.

In order to calibrate the camera we need to image this tile using at least 3 different orientations of the camera principal axis with respect to the normal to the tile's plane (as will be explained later). It is preferable that a large number of grid squares are visible in each image. For each image the location of all neighbouring points at the corners of the grid squares needs to be found in the image so that the *relative positions* of these points in the physical pattern is known, i.e. the origin of the grid pattern on the object is arbitrary. (Although in a simulation there is little point in

re-assigning the origin of the calibration grid - but for a physical grid there is no need to know where the grid origin is in the plane.)

The steps involved are:

1. For every image, choose a grid point to act as the origin of the real grid. It does not matter which grid point is used as the origin, but all relative distances to the identified points in the real tile must be known given this point, and each of these points must be matched to its image in the camera.
2. For every image a special type of transformation matrix called a homography is found (described below).
3. The homographies are used to estimate the positon and orientation of each tile in camera coordinates.
4. A K-matrix is estimated given these positions and orientations.
5. All of the parameters (positions, orientations and the K-matrix) are adjusted slightly to minimize a cost function based on simulating the image of the grid and comparing the simulation with the real image (called bundle adjustment).

6.1 Perspectivities and homographies

If one draws rays through a common centre (called a projective centre) that pass through two planes, these rays define a mapping between the two planes called a Perspectivity. Any pattern on one plane will be in persepective with respect to the resulting pattern on the other plane. This property was discovered by artists during the Renaissance and used to make paintings look more life-like.

Figure 6.2 is a copy of Brunelleschi's figure, describing perspective, that was drawn in the early 15th century. The perspective transformation can be identified with a homogeneous 3×3 matrix transformation, that can itself be identified with a rigid 3D transformation between the planes. A perspectivity is thus a 3×3 matrix; but it has a constrained structure (just for information, it must represent a rotation of the plane about the projective centre plus a dilation along some axis).

Let us revisit the extrinsic model described by equation 2.12.

$$\mathbf{x}_2 = \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ 1 \end{bmatrix} \tag{6.1}$$

The vector $\mathbf{x}_1$ describes 3D points in the object in the object's coordinate system and $\mathbf{x}_2$ is the corresponding homogeneous Point (capital P) in the unit camera frame.

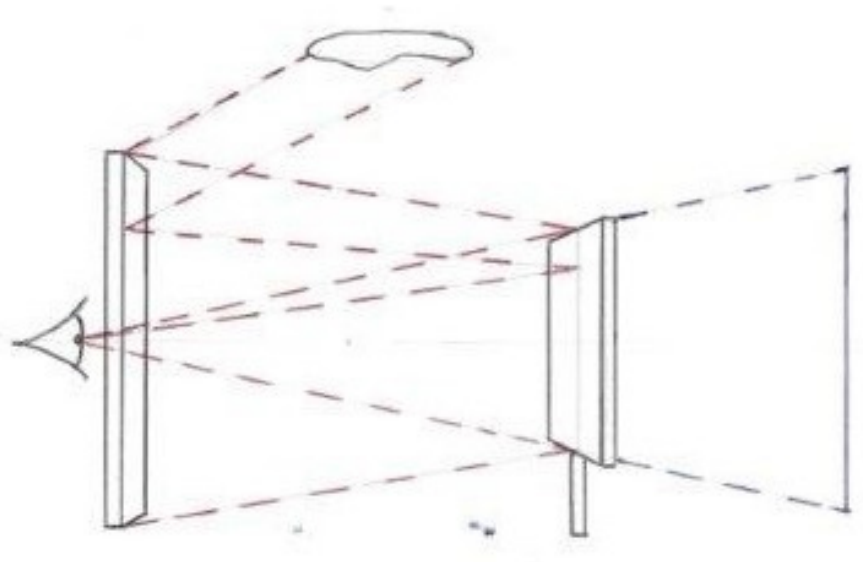


Figure 6.2: Brunelleschi's perpective diagram.

But we have chosen one of the grid points in the calibration pattern as our origin. Thus, all Points on the grid (in our chosen object coordinate system) have the form

$$\begin{bmatrix} \mathbf{x}_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} \tag{6.2}$$

i.e. the z component is zero.

Now let the rotation matrix $\mathbf{R} = [\mathbf{r}_1\ \mathbf{r}_2\ \mathbf{r}_3]$ and write out the equation 6.1 again

$$\mathbf{x}_2 = \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} [\mathbf{r}_1\ \mathbf{r}_2\ \mathbf{r}_3] & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} = [\mathbf{r}_1\ \mathbf{r}_2\ \mathbf{t}] \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.3}$$

Equation 6.3 represents a map between two planes defining a ray mapping between figures that are in perspective (by construction). Thus $[\mathbf{r}_1\mathbf{r}_2\mathbf{t}]$ must be a perspectivity and it represents a homogeneous transformation between Points in the two planes (the calibration grid plane and the unit camera plane).

The homogeneous points in the unit camera plane are transformed into homogeneous points in the sensor via the K-matrix, thus

$$\begin{bmatrix} U \\ V \\ S \end{bmatrix} = \mathbf{K}[\mathbf{r}_1\ \mathbf{r}_2\ \mathbf{t}] \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.4}$$

represents the homogeneous transformation between grid points and the homogeneous representation of their image in the sensor plane (where S is the homogeneous scaling). The final position of the image in the sensor plane is given by $u = U/S$ and $v = V/S$. The matrix $\mathbf{H} = \mathbf{K}[\mathbf{r}_1\ \mathbf{r}_2\ \mathbf{t}]$ is a more general 3×3 homogeneous transformation matrix called a homography.

6.2 Estimating a single homography

Each image of the calibration grid is associated with a homography representing a perspective transformation of the grid into a unit camera followed by an affine transformation into the sensor plane. We need to identify each of these homographies.

We first write out the equation 6.4 using the components of the product homography **H**

$$S \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.5}$$

We first note that we can multiply both sides of this equation by a constant. We can thus scale both sides by h_{33} and set the bottom right corner of the homography to 1.

We now multiply out the terms and divide throughout by S to get

$$\begin{aligned} u &= \frac{x h_{11} + y h_{12} + h_{13}}{x h_{31} + y h_{32} + 1} \\ v &= \frac{x h_{21} + y h_{22} + h_{23}}{x h_{31} + y h_{32} + 1} \end{aligned} \tag{6.6}$$

Note that in the two equations above we know u and v as well as x and y but not the elements h. We cross-multiply and re-express these two equations in matrix form with the elements of the homography expressed as an eight element vector of

unknowns (remembering that we have scaled **H** so that $h_{33} = 1$).

$$\begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -ux & -uy \\ 0 & 0 & 0 & x & y & 1 & -vx & -vy \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} \tag{6.7}$$

The matrix on the left has two rows and, in general, has a rank of 2. However, this matrix equation was generated from a single point matched between the image and the calibration grid. If we use 4 points instead of 1, the extra 3 points will generate 6 more rows that are all, in general, independent. All the rows can be stacked to generate a square matrix that is in general non-singular - thus allowing us to find the matrix **H**.

Re-write equation 6.7 as

$$\phi \mathbf{h} = \mathbf{p} \tag{6.8}$$

where ϕ is the 2×8 matrix on the left, **h** is the vector of homography elements **p** is the two-element vector $[u, v]^T$. Now label each point in the calibration grid with the letter 'j' and stack all of the matrices for all n points in the calibration grid to obtain

$$\begin{bmatrix} \phi_1 \\ \vdots \\ \phi_j \\ \vdots \\ \phi_n \end{bmatrix} \mathbf{h} = \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_j \\ \vdots \\ \mathbf{p}_n \end{bmatrix} \tag{6.9}$$

This is an over-constrained matrix equation of the form $\mathbf{Ax} = \mathbf{b}$ with many more equations than unknowns (providing n is larger than 4). In reality, the estimates of the image points $[u, v]$ will be noisy, thus these equations will not be perfectly consistent for all of the points in the image and there is in general no exact solution.

6.2.1 Minimum norm solutions and scaling

One way of defining the 'best' solution to $\mathbf{Ax} = \mathbf{b}$ when we have a large number of equations that may be inconsistent is to find the solution that minimizes the norm

of the back-substitution error, i.e.

$$\min |\mathbf{e}|^2 = \min |\mathbf{A}\mathbf{x} - \mathbf{b}|^2 \tag{6.10}$$

The above minimization is called 'least-squares' and assumes that the errors only occur in the vector $\mathbf{b}$, are evenly distributed and are independent (I.I.D. and white). But the 'noisy' measurements u and v appear in $\mathbf{A}$ as well as in $\mathbf{b}$. Thus a simple least squares solution $\mathbf{x}$ may not be the 'best' solution. But if the noise is small any bias introduced through not taking into account perturbations to $\mathbf{A}$ should be small provided the matrix $\mathbf{A}$ has rank 8 under all small errors in the measurements.

Solving the least-squares problem using generalized inverses

We first note that the square of the norm of a vector is just its dot product with itself, i.e.

$$|\mathbf{e}|^2 = \mathbf{e}^T\mathbf{e} \tag{6.11}$$

Thus we expand the square of the norm of the defining equation

$$|\mathbf{A}\mathbf{x} - \mathbf{b}|^2 = \mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x} - 2\mathbf{x}^T\mathbf{A}\mathbf{b} + \mathbf{b}^T\mathbf{b} \tag{6.12}$$

We then differentiate partially with respect to all the elements of $\mathbf{x}$ and equate to zero

$$2\mathbf{A}^T\mathbf{A}\mathbf{x} - 2\mathbf{A}\mathbf{b} = \mathbf{0} \tag{6.13}$$

The set of parameters that minimize the norm is thus

$$\mathbf{x} = \left[\mathbf{A}^T\mathbf{A}\right]^{-1}\mathbf{A}\mathbf{b} \tag{6.14}$$

This is called a pseudo-inverse solution of the least-squares problem. But is this a good idea?

Solving the least-squares problem using the singular-value decomposition

Recall that it was pointed out previously that using the Matlab '`inv`' function can result in poor accuracy and be slow.

A better way to solve the least squares problem is to use the singular value decomposition of the matrix $\mathbf{A}$. Any real matrix, square or otherwise, defines two sets of vectors that are the eigenvectors of the matrices $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$. The **non-zero** eigenvalues of these two matrices are identical and are called the singular values of the matrix $\mathbf{A}$. The matrix $\mathbf{A}$ can then be written as

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^T \tag{6.15}$$

The matrices **U** and **V** are square *orthonormal* matrices and **D** is rectangular. **D** is zero except for a non-zero diagonal set of elements in its top left corner. These non-zero elements are the singular values of **A**. This decomposition can be computed in Matlab by the code `[U,D,V] = svd(A);`.

We can now generate a pseudo-inverse of the matrix **A**, using an abuse of notation,

$$\mathbf{A}^{-1} = \mathbf{V}\mathbf{D}^{-1}\mathbf{U}^T \tag{6.16}$$

where $\mathbf{D}^{-1}$ here means transposing and taking the inverse of the non-zero set of entries in the top left corner (not its 'real' inverse).

The matrix **A** has 2n rows and 8 columns, where $2n > 8$ and n is the number of calibration grid Points that are visible in the image. Thus **U** will be $2n \times 2n$, **D** will be $2n \times 8$ and **V** will be 8×8. The matrix **D** will have an 8×8 diagonal matrix block in its first 8 rows that contains the non-zero singular values along the block's diagonal; the rest of **D** will be zero. The pseudo-inverse of **D** will be an $8 \times 2n$ matrix with an 8×8 diagonal matrix block in the extreme left, with the inverses of the singular values along the block's diagonal, the rest being zeros. Such a matrix structure is a bit of a waste because there will be $2n-8$ columns of **U** multiplied by lots of zeros in the right of the matrix **D**, thus taking no part in the inverse. Matlab has a specialized form of the `svd` algorithm that is called using the code `[U,D,V]=svd(A,'econ');`. For our case the matrix **U** will now be $2n \times 8$, **D** will be 8×8 and diagonal, and **V** will still be 8×8. This form makes the computation of the generalized inverse much easier as we now do not need to transpose **D** and the correct columns of **U** are automatically made available.

In a practical application one needs to decide on a numerical definition of 'non-zero'. If the smallest singular value is very small, then using this value in the inverse calculation will lead to large entries being added to the solution. The ratio of the largest to smallest singular value is a measure of the conditioning of the matrix and some applications do not include very small singular values in the calculation. It is wise to check the ratio of the largest to smallest singular value before performing such inverse calculations.

Scaling

Given the above discussion of the conditioning of the matrix inversion problem one might ask what might be done to reduce the problem.

Poor conditioning often arises from the choice of units used to represent measurements. Here the units might be pixels and millimetres. Often one removes the problem of choice of units by choosing a unitless representation. For the case of

the camera 'pixel' unit one could place a new origin at the centre of the image and choose the maximum pixel offset to be 1.0. A pixel would then correspond to 1.0 / half the width or height of the image. Similarly, we could adopt a scaling for the grid in which the whole grid is considered to be 2.0 units on a side and the size of the squares scaled to fractions of a side. Adopting such scaling would result in all measurements lying between -1.0 and 1.0 and helps to improve the conditioning of the estimation task. The correct units are then re-imposed on completion by appropriate scaling of the resulting homography.

Outliers

We have assumed above that all measurements in the image are 'valid', i.e. the positions of the corners in the grid are correct and matched to the correct grid points in the calibration object. Any error present is assumed to be 'white' and that the errors fit a simple noise model.

However, the measurements themselves have been generated by another estimation algorithm (a computer vision algorithm outside the scope of this project) and this algorithm may generate nonsensical measurements (perhaps a cat walked in front of the camera just as the picture was taken). Such measurements (that do not fit the assumed error process model) are called outliers and some means of detecting their presence is required.

The most common approach to rejecting outliers is to use Random Sample Consensus (RANSAC). A brief description of how to program RANSAC is given in Wikipedia `https://en.wikipedia.org/wiki/Random_sample_consensus`

We have found that 4 correspondences between grid Points and image points will generate an 8×8 matrix $\mathbf{A}$ that in general should have rank 8. Choose 4 such matching points at random and calculate the resulting homography. Use this homography to predict the positions of all the other visible grid points in the image and compute the error in position of each of these points when compared to the positions generated by the corner estimation algorithm. The random 4 points is a set of 'hypothetical inliers'. We now need to find how many errors for the remaining points were acceptable. You will need to choose a measure of 'reasonable error' for this task. All those points that agree with the predictions (their errors are less than a maximum that you choose to reflect a 'reasonable' error) of the model generated by the 4 points chosen at random. This larger set of points is called the 'consensus' set. Make a note of the number of points that lie within the consensus set. Repeat this process with a large number of other randomly chosen 4 points, each resulting

model having its own consensus set. Choose the model with the largest consensus set and re-estimate using the least-squares estimation technique described above, but only using the 'best' random 4 points and its consensus set. This is the RANSAC consensus least-squares estimate.

It is possible to put the above technique into a probabilistic framework in order to guide the choice of error and how many sets of 4 random points one should choose to test.

6.3 Your initial estimation task

Your task now is to write code that, given a set of Points in a calibration grid and corresponding points in the camera image, computes the least squares estimate of the homography mapping the Points into the image. Consider how you will test your code using the camera simulation code that you wrote earlier and how you will test how the solution changes when the image points are perturbed by a small amount of image noise. Do you wish to add 'outliers' and use RANSAC to reject these? If you do, think carefully about how to model the generation of such outliers.

These are the minimum set of steps that you will need to follow:

1. Decide on the size of the grid (in mm), how many points should appear in the grid and how to choose the location of the grid. The grid should 'fill' the image, i.e. the image of the grid should be larger than the camera's image size. Use the components from your simulation program to help you make these decisions. I used a 1m grid with tiles that are square and 10mm per side.

2. 'Real' computer vision programs generate estimates of the grid point locations in the image with some noise. Decide on a noise level and consider how you might simulate such errors in locating grid points in your simulation. I used a variance of 0.5 pixels and used the Matlab function `randn` to generate image noise.

3. Decide on how you will test the robustness of your code and how many runs you will perform to generate some statistics on how your code performs. Are you going to test for robustness against outliers? I used a probability of an outlier of 0.05 and ran Ransac 50 times per run with a reject error norm of 3 pixels (the norm of the error vector).

Note the following:

- The positions reported back of tile corners from any reasonable vision algorithm will not be integer and will have fractional parts. You may assume the measurements are reported at the default precision of Matlab.
- Do not be surprised if some of the consensus sets are empty. A random selection of points from the measurements may result in the initial estimation matrix being very badly conditioned and generating wildly inaccurate estimates of the homography. The number of Ransac runs needed to get an estimate will be larger than you think and certainly larger than the number reported in the analysis presented in the Wikepedia article on Ransac. Choosing this number is an art, not a science!

I provide code snippets below of some of the more challenging aspects of the code. Only use these for ideas as to how you will tackle the problem. Many of the functions required are not listed here.

Here is the script I wrote for running the code:

```
% Example script for estimation task 1
%
% This script performs the following actions:
% 1. Constructs a camera model loosely based on an iPhone6
% 2. Constructs a calibration grid 1m on a side with 10mm grid spacing.
% 3. Positions the grid somewhere in space.
% 4. Places the camera in a'random' location so that the grid
% fills the camera image.  Check this location to see if the
% grid outline is outside the image; if not, more the camera nearer
% to the grid.  This is NOT a smart way of positioning things, but
% it models the likely behaviour of a human user.
% 5. Compute the image of the grid and constructs a set of grid - image
% correspondences and add 'noise' ot the image.
% 6. Convert some of the correspondences to outliers.
% 7. Carry out a RANSAC estimation of the homography.

% 1. Construct the Camera model
[KMatrix, CameraHeight, CameraWidth] = BuildCamera;

% 2. Construct a 1m by 1m grid with 10mm tiles in the grid frame
% The grid is a set of 4-element vectors [x y 0 1]'.
GridWidth = 1000;
GridIncrement = 10;
CalibrationGrid = BuildGrid(GridIncrement,GridWidth);

% 3. Choose somewhere in space for the grid
% T_ow is the 4x4 tranformation matrix from grid to world.
T_ow = PositionGrid();

```

```
% 4 Choose a 'random' location for the camera that fills the image.
% T_cw is the 4x4 transformation matrix from camera to world
T_cw = FillImage(T_ow,KMatrix,GridWidth,CameraHeight,CameraWidth);

% 5 We now fill the camera with a noisy image of the grid and generate
% the point correpondences.
% Correspond is a set of pairs of vectors of the form [[u v]' [x y]']
% for each grid corner that lies inside the image.
Correspond = BuildNoisyCorrespondences(T_ow,T_cw,CalibrationGrid, ...
    KMatrix,CameraHeight,CameraWidth);

% 6. Add in some 'outliers' by replacing [u v]' with a point
% somewhere in the image.
% Define the Outlier probability
pOutlier = 0.05;
for j = 1:length(Correspond)
    if rand < pOutlier
        % This is an outlier - so put the point anwhere in the image.
        Correspond(1,j) = rand * (CameraWidth-1);
        Correspond(2,j) = rand * (CameraHeight-1);
    end
end

figure(1)
plot(Correspond(1,:),Correspond(2,:),'.')
title('The noisy measurements of the tile corners')
axis ij

% 7 Perform the Ransac estimation - output the result for inspection
% If the Ransac fails it retuns a zero Homography
Maxerror = 3;     % The maximum error allowed before rejecting a point.
RansacRuns = 50;  % The number of runs when creating the consensus set.
[Homog,BestConsensus] = RansacHomog(Correspond,Maxerror,RansacRuns);

% If you want to test the result, we can construct the homography
% for the system from its definition.
%
% First find the object frame in the camera frame
T_oc = T_cw \ T_ow;
% Construct the non-normalized homography from K*[x y t]
OrigHomog = KMatrix * [T_oc(1:3,1) T_oc(1:3,2) T_oc(1:3,4)];
% And normalise so that (3,3) is 1.0 - output for inspection
OrigHomog = OrigHomog / OrigHomog(3,3);
```

../Matlab/EstimateHomography/RunEstimateHomography.m

Here is my Ransac code

```
function [ Homog ,BestConsensus] = RansacHomog( Correspond,MaxError,NRuns)
%RansacHomog Runs a Ransac estimation of the homography
```

```
% passed as pairs of points in Correspond where Correspond
% is a set of 4-vectors of the form [[u v]';[x y]']
%
% MaxError is the maximum error for a point to be rejected in the
% consensus set.
% NRuns is the number of times to run the estimator
%
% Homog is the homography that has been identified.  If no homography
% can be computed, then a 3x3 zero matrix is returned.
%
% 1. For each of NRuns, choose a set of 4 points.
% 2. Use the 4 points to generate a regression matrix and a data vector.
% 3. If the regressor is full rank, estimate this homography.
% 4. Go through the points and accept those whose error norm is less
%    than that set by MaxError.
% 5. Record the set of points in the largest consensus set.
% 6. If the consensus set is not empty, carry out a least squares
%    estimate of the homogrphy using svd.

% The number of points available
n = length(Correspond);

% Allocate space for the homography
Homog = zeros(3);

% The number of points in the best consensus
nBest = 0;

for Runs = 1:NRuns

    RankTest = 1;
    while RankTest == 1
        % RankTest is set to 1 if the 4 points do not give
        % a full rank matrix in the estimator.
        RankTest = 0;

        % 1 Choose a sample set
        SamplePoints = zeros(1,4);

        % The first point is a random integer between 1 and n inclusive
        SamplePoints(1) = 1 + fix(n*rand);
        if SamplePoints(1) > n
            SamplePoints(1) = 1;
        end

        % Choose the next three points ensuring that there are no repeats
        for j = 2:4
            % searching is a toggle that is triggered if there is a repeat
            searching = 1;
```

```
52              while searching == 1
53                  searching = 0;
54                  % Initial sample point
55                  SamplePoints(j) = 1 + fix(n*rand);
56                  if SamplePoints(j) > n
57                      SamplePoints(j) = n;
58                  end
59
60                  % Is the new point a repeat?  If so, go around again
61                  for k = 1:j-1
62                      if SamplePoints(k) == SamplePoints(j)
63                          searching = 1;
64                      end
65                  end
66              end
67          end
68
69
70          % 2. Allocate space for the regressor and the data
71          Regressor = zeros(8);
72          DataVec = zeros(8,1);
73
74          % Fill in the regressor and data vector given the samples to take
75          for j = 1:4
76              % row indices into the matrix and vector for this data point
77              r1 = j*2-1;
78              r2 = j*2;
79              [Regressor(r1:r2,:),DataVec(r1:r2)] = ...
80                  HomogRowPair(Correspond(:,SamplePoints(j)));
81
82          end
83
84          % 3. Is the regressor full rank?
85          if rank(Regressor) > 7
86              HomogVec = Regressor \ DataVec;
87
88              % The homography for this sample
89              Homog(1,1) = HomogVec(1);
90              Homog(1,2) = HomogVec(2);
91              Homog(1,3) = HomogVec(3);
92              Homog(2,1) = HomogVec(4);
93              Homog(2,2) = HomogVec(5);
94              Homog(2,3) = HomogVec(6);
95              Homog(3,1) = HomogVec(7);
96              Homog(3,2) = HomogVec(8);
97              Homog(3,3) = 1;
98
99              % The number of points in the current consensus set and the
                   set
```

```
100             % itself
101             nCurrent = 0;
102             CurrentConsensus = zeros(1,n);
103             % 4. Go through all the points and see how large the error is
104             for j = 1:n
105                 HomogenousPoint = Homog * [Correspond(3,j);Correspond(4,j)
                        ;1];
106                 HomogenousPoint(1) = HomogenousPoint(1) / HomogenousPoint
                        (3);
107                 HomogenousPoint(2) = HomogenousPoint(2) / HomogenousPoint
                        (3);
108
109                 ThisError = norm(HomogenousPoint(1:2) ...
110                     - [Correspond(1,j);Correspond(2,j)]);
111
112                 if ThisError < MaxError
113                     nCurrent = nCurrent+1;
114                     CurrentConsensus(nCurrent) = j;
115                 end
116             end
117
118             % 5. How well did this sample do?
119             if nCurrent > nBest
120                 nBest = nCurrent;
121                 BestConsensus = CurrentConsensus;
122             end
123         else
124             RankTest = 1;
125         end
126     end
127 end
128
129 % 6. BestConsensus now contains the largest set of consistent estimates.
130 % Use this set to estimate the homography using a robust inverse
131 if nBest > 0
132     % The number of measurements in the consensus set
133     Regressor = zeros(2*nBest,8);
134     DataVec = zeros(2*nBest,1);
135
136     % Construct the regressor
137     for j = 1:nBest
138         r1 = j*2-1;
139         r2 = j*2;
140         % HomogRowPair generates 2 rows of the 8 column matrix
141         % that multiplies the unknown vector of homography elements
142         % to get the vector of measurements.
143         [Regressor(r1:r2,:),DataVec(r1:r2)] = ...
144             HomogRowPair(Correspond(:,BestConsensus(j)));
145     end
```

```
146
147     % Find the singular value decomposition in order to compute the
148     % robust pseudo-inverse.
149     [U,D,V] = svd(Regressor,'econ');
150
151     % The condition number of the computation - a measure of how reliable
152     % the inversion is.
153     if D(8,8) < eps
154         Condition = 1.0E16;     % Just a very big number.
155     else
156         Condition = D(1,1) / D(8,8);
157     end
158
159     if Condition > 1.0e8
160         % A very poor condition number - signal that there is no homograpy
161         Homog = zeros(3);
162     else
163         % The conditioning is good
164         % Invert the diagonal matrix of singular values
165         for j = 1:8
166             D(j,j) = 1.0 / D(j,j);
167         end
168
169         HomogVec = V*(D*(U'*DataVec));
170
171         % construct the homography
172         Homog(1,1) = HomogVec(1);
173         Homog(1,2) = HomogVec(2);
174         Homog(1,3) = HomogVec(3);
175         Homog(2,1) = HomogVec(4);
176         Homog(2,2) = HomogVec(5);
177         Homog(2,3) = HomogVec(6);
178         Homog(3,1) = HomogVec(7);
179         Homog(3,2) = HomogVec(8);
180         Homog(3,3) = 1;
181
182     end
183
184 else
185     % Signal that no homography could be found
186     Homog = zeros(3);
187     BestConsensus = 0;
188 end
189
190 end
```

../Matlab/EstimateHomography/RansacHomog.m

This is how I fill the image with the grid points

```
function [ T_cw ] = FillImage(T_ow,KMatrix,GridWidth, ...
    CameraHeight,CameraWidth)
%FillImage Takes in a Calibration grid definition and positions the
% camera so the the image of the grid fills the camera image.
%
% T_ow is the Calibration Grid frame
% KMatrix is the intrinic camera model
% Gridwidth is the length of a size of the whole grid im mm
% CameraHeight and CameraWidth are the sizes of the image in pixels

% Define the Left Top, Left Bottom, Right Top and Right Bottom of the grid
    .
GridCorners = [-GridWidth/2 -GridWidth/2 GridWidth/2 GridWidth/2; ...
    GridWidth/2 -GridWidth/2 GridWidth/2 -GridWidth/2;
    0 0 0 0;
    1 1 1 1];

% Compute the positions of the GridCorners in the world
GridCorners = T_ow*GridCorners;

% We have a 1m by 1m grid somewhere in space and we need to view
% the grid from the camera.  We view from a random location based
% on a 'distance' called CameraBaseDistance.  this is an initial estimate.
CameraBaseDistance = 2000;

% Keep reducing the distance until all 4 corners are outside the image
% and InsideImage is a flag that records the failure of our objective
% thus triggering a move towards the object and a retry.
InsideImage = 1;
while InsideImage == 1

    InsideImage = 0;

    % PositionCamera makes the camera 'almost' point
    % at the object, but introduces randomness so we have
    % a range of angles betweeen the grid's normal and the
    % camera z-axis.
    T_cw = PositionCamera(T_ow,CameraBaseDistance);

    % Compute where corners are in the unit camera frame
    UnitCorners = (T_cw \ GridCorners);
    % Project into the unit camera
    UnitCorners = UnitCorners(1:3,:);
    % And convert to camera pixels and label them as homogeneous.
    HomogeneousCorners = KMatrix * UnitCorners;

    % Dehomogenise the image of the corners of the grid
    Corners = zeros(2,4);
    for j = 1:4
```

```
49          Corners(1:2,j) = ...
50              HomogeneousCorners(1:2,j) / HomogeneousCorners(3,j);
51          if Corners(1,j) > 0 && Corners(1,j) < CameraWidth-1
52              % The u component is not outside the image - keep searching
53              InsideImage = 1;
54          end
55          if Corners(2,j) > 0 && Corners(2,j) < CameraHeight-1
56              InsideImage = 1;
57          end
58      end
59
60      % Move the camera nearer to the object if any of the corners are
61      % inside the image.  The next time around we have a better chance
62      % of achieving our objective.
63      if InsideImage == 1
64          CameraBaseDistance = CameraBaseDistance * 0.9;
65      end
66
67  end
68
69  end
```

../Matlab/EstimateHomography/FillImage.m

This is how I position the camera

```
1  function [T_cw ] = PositionCamera(T_ow,Distance)
2  %PositionCamera
3  % Generates a 'random' camera frame that almost points
4  % at the origin of the calibration object back along the object's
5  % z-axis (which is perturbed).  Aim is to 'fill' the camera image
6  % with calibration grid.
7  %
8  % T_ow is the 4x4 frame of the calibration grid in world coordinates.
9  % Distance is the base distance to the grid from the camera
10 % along the normal and a random factor of 0.1 times the distance is
11 % added.
12 %
13 % T_cw is the 4x4 camera frame in world coordinates.
14
15 % Assign space for the camera frame
16 T_cw = zeros(4);
17
18 % Set the homogeneous multiplier to 1
19 T_cw(4,4) = 1;
20
21 % extract the object origin
22 ObjectOrigin = T_ow(1:3,4);
23
24 % View the camera from about 'Distance' with a bit of randomness
```

```
% along the negative z-axis of the grid.  This vector will
% be almost parallel to the camera z axis.
InitialViewVector = -Distance*T_ow(1:3,3) + 0.1*Distance*rand(3,1);

% Define the origin of the camera frame in world coordinates
T_cw(1:3,4) = ObjectOrigin - InitialViewVector;

% Define the camera z-axis as pointing at the object origin
Normz = norm(InitialViewVector);
if Normz < eps
    error('Unable to normalize the camera z-axis');
end
% Define a unit vector pointing at the object.
InitialCameraz = InitialViewVector / Normz;

% Perturb the initial z axis a bit more for luck
CameraZ = InitialCameraz - 0.01*(rand(3,1)-0.5);

% ... and normalize again (no need to check norm)
CameraZ = CameraZ / norm(CameraZ);

% Define a random camera x-axis
CameraX = rand(3,1);
% project out the z-axis
CameraX = CameraX - (CameraZ'*CameraX)*CameraZ;
% normalize the x-axis
Normx = norm(CameraX);
if Normx < eps
    error('Unable to normalize the camera x-axis')
end
CameraX = CameraX/Normx;

% Define the y-axis
CameraY = cross(CameraZ,CameraX);

% Complete the transformation matrix
T_cw(1:3,1:3) = [CameraX CameraY CameraZ];

end
```

../Matlab/EstimateHomography/PositionCamera.m

6.3.1 Developing and debugging your code

The task above is possibly the first time you have put together a complex program. There are many strategies for building complicated code, but here I describe the strategy that I followed.

Build test harnesses

Note that the code is run through the *script* `RunEstimateHomography`. I have written a series of such scripts during the development of the final piece of code at the end of the project. Do you remember the difference between a script and a function? All variables created in a script remain available when the code terminates correctly. Slowly build up your test harness and use it to test the functions that your are writing. I tend to start by writing the detailed code for the function within the test harness and ensure that the code is working correctly before creating a separate function file. I then 'cut and paste' the test harness detail to the new function and then test to see if it works in the same manner as the test harness. The test harness thus grows and shrinks as more functionality is added and code is moved over to functions. These function can then be used in later code as fully tested objects.

6.3.2 Break down your problem into sub-problems

Try and identify coherent logical operations that can be separated into functions. Here are the blocks I used for the current task (in alphabetical order):

Buildcamera.m This function defines the parameters that describes the camera and performs the function of a 'definition file'. The code calls `SingleVectorCameraModel.m` to perform the actual construction of the K-matrix, using code tested in an earlier task. It is more usual in other languages to read the definition file from the construction function - but the approach used here is designed for use within the Matlab interpreted environment and to fit in with the code development path defined by the sequence of tasks set down in this project.

BuildGrid.m This function separates the task for constructing an object from the main script. I have defined the representation as lists of Points, which are 4-vectors. This function just returns a list of points that correspond to the corners of the grid.

BuildNoisyCorrespondence.m This function takes sets of 4-vectors in world coordinates and computes where they end up in the camera image. The representation is chosen for constructing grid-point to image-point correspondences and the output is a list of 4-vectors that are NOT homogeneous points, but pairs of points in the form $[[u, v]^T; [xy]^T]$. Normally distributed noise is added before the correspondences are returned to the calling funciton.

FillImage.m This function creates the camera frame so that the image of the grid fills the camera. The code defines an initial distance between the camera and the grid and repeatedly calls `PositionCamera` with a reducing distance until the grid fills the image.

HomogRowPair.m Implementing mathematical functions with complex entries is very error-prone. Thus I separated out the construction of the pair of rows that form the homography estimation matrix into a function. I could then test this function to ensure that the correct equations are implemented.

PositionCamera.m This function positions the camera in a semi-random location so that there is a good chance of getting a 'good' image of the grid. The code defines a viewing vector that is slightly off normal to the plane and uses this vector to compute the camera frame. One has to be careful with 'slightly off-normal' to ensure that there is a good spread of angles between the principal vector and the normal to the grid.

PositionGrid.m This function just creates a random 4×4 homogeneous transformation matrix defining an 'object frame' as tested in a previous task above.

RandomRotationMatrix.m This function is called from `PositionGrid.m` to create a random rotation matrix.

RansacHomog.m This function implements a Ransac algorithm for estimating the Homography. It was written initially as part of `RunEstimateHomography.m` and each component tested before copying to a function.

RunEstimateHomography.m This script is the test harness used to run and develop the code. Place `plot` instructions in here to check on the output of the code. I added and deleted various plot instructions as the code was developed. These were added to make sure the code was doing what is intended.

SingleVectorCameraModel.m This function was one of those constructed in part of the first programming task and is being re-used.

TestRange.m This function is used during camera construction to test the validity of the camera description. It is being re-used from an earlier task.

ViewCamera.m This function is just my way of presenting the data. It is used to test the output during development and is not part of the final code.

6.3.3 Create sub-problems for which you 'know' the answer

As I developed the code, I commented out the 'noise' addition and added code to compute the actual homography (you will find this test code at the end of `RunEstimateHomography.m`). One can then test the estimator without the presence of outliers or noise. If there is no noise or outliers, then the consensus set from Ransac should be the entire set of points. If it is not, you will need to investigate where your code is failing.

I switched noise on and off by using a comment. You could perform the same function by passing a parameter, such as the noise variance, to change the level of noise.

In fact, the entire project contains the test of the optimizer as an objective. We are only inputting simulated image data and we know the answer (we compute the K-matrix before we 'estimate' it). The percentage of outliers, the spacing of the grid and the noise level in the estimation of the grid corners can all be varied to determine how the code's performance varies as these key parameters are changed. You can also change the range of angles from which the grid is viewed to see how this affects the quality of estimation.

6.4 Using the properties of perspectivities to estimate the K-matrix

The previous sections describe how one estimates a single homography. We have a single image and we use a large number of grid-image pairs of Points to generate the homography $\mathbf{H}$ that is the product of the intrinsic model of the camera (the K-matrix) and an extrinsic model of the unit camera (a perspectivity). We now take *at least* three independent images of the calibration grid taken from a range of directions (the angles between the principal axis of the camera and the calibration grid normal must be independent and span a 'reasonable' range, say, at least 20 degrees). The need for the independence is to satisfy a rank requirement (described below) and the reason for the 'reasonable' range is to ensure good conditioning of the ensuing estimation process (sometimes referred to as 'sufficient excitation').

Recapping our reasoning so far:

We have assumed that the camera model satisfies

$$\mathbf{H} = \mathbf{K} \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{t} \end{bmatrix}. \tag{6.17}$$

Our estimator as used the fact that $\mathbf{H}$ is homogeneous and set its bottom right

corner to '1'. We have thus generated an estimate of

$$\mathbf{H} = \lambda \mathbf{K} \left[\begin{array}{ccc} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{t} \end{array} \right], \tag{6.18}$$

where λ is some unknown scaling. We have performed the above on each of the images. Each image was generated by the same K-matrix, but have different perspectivities and scalings. Let each image be given a label 'j'. Then

$$\mathbf{K}^{-1}\mathbf{H}_j = \lambda_j \left[\begin{array}{ccc} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{t} \end{array} \right]_j. \tag{6.19}$$

We now use the orthogonality properties of rotation matrices by multiplying on the left by the transpose of the matrix on the right.

$$\mathbf{H}_j^T \left(\mathbf{K}^{-1}\right)^T \mathbf{K}^{-1}\mathbf{H}_j = \lambda_j^2 \begin{bmatrix} 1 & 0 & \bullet \\ 0 & 1 & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix}_j, \tag{6.20}$$

where $\bullet$ denotes a matrix entry that is going to be ignored.

Denote $\left(\mathbf{K}^{-1}\right)^T \mathbf{K}^{-1}$ by the symmetric matrix ϕ. Then the orthogonality properties of the perspectivity imply that ϕ satisfies the following constraints (defined by the 2×2 upper left unit matrix)

$$\begin{aligned} \mathbf{h}_{1\,j}^T \phi \mathbf{h}_{1j} - \mathbf{h}_{2\,j}^T \phi \mathbf{h}_{2j} &= 0 \\ \mathbf{h}_{1\,j}^T \phi \mathbf{h}_{2j} &= 0, \end{aligned} \tag{6.21}$$

where $\mathbf{h}_{1j}$ denotes the first column of $\mathbf{H}_j$ and similarly for column 2. Each of these expressions is linear in the elements of the symmetric ϕ that has 6 independent components. Expanding the inner products we obtain the constraint equation for image j

$$\psi_j \begin{bmatrix} \phi_{11} \\ \phi_{12} \\ \phi_{13} \\ \phi_{22} \\ \phi_{23} \\ \phi_{33} \end{bmatrix} = \mathbf{0} \tag{6.22}$$

where ψ_j is the matrix

$$\begin{bmatrix} h_{11}^2 - h_{12}^2 & 2\left(h_{11}h_{21} - h_{12}h_{22}\right) & 2\left(h_{11}h_{31} - h_{12}h_{32}\right) & h_{21}^2 - h_{22}^2 & 2\left(h_{21}h_{31} - h_{22}h_{32}\right) & h_{31}^2 - h_{32}^2 \\ h_{11}h_{12} & h_{11}h_{22} + h_{21}h_{12} & h_{11}h_{32} + h_{31}h_{12} & h_{21}h_{22} & h_{21}h_{32} + h_{31}h_{22} & h_{31}h_{32} \end{bmatrix}_j \tag{6.23}$$

Each image generates 2 equations that populate a 2×6 matrix. Stack these matrices on top of each other to generate a $2n\times 6$ matrix where $n > 2$ (and I suggest at least 6 images, but you can technically use 3 independent images).

Equation 6.22 implies that a kernel must exist, i.e. a solution for ϕ exists that makes the right hand side zero. But the measurements are subject to noise and error - so how do we solve this problem if there is no vector that makes the right hand side zero?

As the matrix problem will probably have no 'real' kernel (because of noise) we assume that the best estimate is the vector that minimizes the norm of the product $\mathbf{Ax}$ where $\mathbf{A}$ is the stacked matrix defined above. The solution for $\mathbf{x}$ is the right singular vector of the matrix $\mathbf{A}$ associated with the smallest singular value. In Matlab, this vector is found by the following:

```
% Find the singular value decomposition
[U,D,V] = svd(A,'econ');
% Convert the diagnonal matrix D into a vector
% (it should have been square as we are using the 'econ' version of svd)
d = diag(D);
% Find the smallest singular value, element I
[m,I] = min(d);
% Find the upper triangular part of the solution.
% The smaller m is, the better.
% But the next larger singular value must be >> than m.
PhiVector = V(:,I);
```

Once we have found the best estimate of the kernel, we assemble the matrix ϕ and then note that

$$\phi = \left(\mathbf{K}^{-1}\right)^T \mathbf{K}^{-1} \tag{6.24}$$

is the product of a lower triangular matrix, $(\mathbf{K}^{-1})^T$ and an upper triangular matrix $\mathbf{K}^{-1}$. You will meet a similar product to this in A1 called the LU factorization of a matrix. But the matrix product we are considering must have a more special decomposition that a simple LU decomposition as it is an 'inner product'. This special decomposition is called the Cholesky Factorization and is found using the `chol` command. See `https://en.wikipedia.org/wiki/Cholesky_decomposition` for a description of this decomposition. Here is an example piece of final code with no error checking

```
InvK = chol(phi);
% Solve for the inverse to get the K-matrix
K = InvK \ eye(3);
% Normalize the lower right element to 1.
K = K / K(3,3);
```

6.5 Your second estimation task

Use the estimation results from your first estimation task to construct an estimator for the K-matrix. In this task you should:

1. Decide on a scaling for the grid points and image point locations so that the maximum magnitude of a coordinate in any direction is 1.0.
2. Estimate the homographies for at least 6 images using scaled data.
3. Find the matrix ϕ using the construction described above.
4. Compute the K-matrix.
5. Consider the impact of scaling that you have performed and rescale your results so that the unit camera is at 1mm and the K-matrix is expressed in pixels.

Scaling again

The grid coordinates are in mm. I used a 1m by 1m grid with 10mm squares in my code. Look back to the constraint equation 6.20. We arrived at the entries of a 2×6 matrix by only considering the orthogonality properties of the persepectivity between the grid and the unit camera. Thus changing the scale of the grid from [-500, 500] (in my code) to [-1, 1] does not change any of the equations defining the K-matrix (which is measured in pixels). All that we are doing is redefining the units used and changing where the unit camera is located in space - it is only the extrinsic model that is changing. Thus scaling of the grid only impacts the numbers that appear in the homography estimation and (hopefully) improves the conditioning of the calculations.

Scaling the image corner locations is different. We multiply the u and v coordinates by 'Scaling' and then subtract 1.0. In my code I only use a singe scaling, but you could choose to use different scaling in u and v. Now consider how we undo the scaling after estimating the intrinsic model. Adopting the letter 's' for scaling, we have the new coordinates are equal to 'shift' and 'scale' times K times perspectivity,

i.e.

$$\begin{bmatrix} U_S \\ V_S \\ S_S \end{bmatrix} = \begin{bmatrix} s & 0 & -1 \\ 0 & s & -1 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{K} \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{t} \end{bmatrix} \tag{6.25}$$

Inverting the scaling matrix, we have

$$\begin{bmatrix} s & 0 & -1 \\ 0 & s & -1 \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1/s & 0 & 1/s \\ 0 & 1/s & 1/s \\ 0 & 0 & 1 \end{bmatrix} \tag{6.26}$$

Thus the actions needed are

1. Find K using the constraints on the perspectivity.
2. Normalize K to 1.0 in the bottom right.
3. Add 1 to the first 2 entries of the third column of K.
4. Multiply the first two rows of K by 1/s.

When writing your code ensure that the matrix ϕ is positive definite before calling the Matlab function `chol` by ensuring all the eigenvalues of ϕ are real and positive. Also make sure you test the bottom right element of the K-matrix before scaling this element to 1.0 that the element is not too 'small'.

Here is my code for estimating the seed K-matrix. Note that I have used the Matlab object `cell` to record the data used in the estimation.

```
% This script runs the estimation of the KMatrix
%
%
% 1. Constructs a camera model loosely based on an iPhone6
% 2. Constructs a calibration grid 1m on a side with 10mm grid spacing.
% 3. Positions the grid somewhere in space.
%
% Perform the following actions on each image, repeating an image
% if the homography estimation failed.
% 4. Places the camera somewhere in space to generate a full image
%    of tile square corner locations.
% 5. Generate the noisy image of the grid.
% 6. Add in some outliers.
% 7. Perform a Ransac estimate of the homography
%
% Once the homographies have been estimated
```

```
% 8. Build the regressor for estimating the K-matrix
% 9. Carry out the Cholesky factorization and invert.

% The number of calibration images to use
nImages = 6;

% 1. Construct the Camera model
[KMatrix, CameraHeight, CameraWidth] = BuildCamera;

% 2. Construct a 1m by 1m grid with 10mm tiles in the grid frame
% The grid is a set of 4-element vectors [x y 0 1]'.
GridWidth = 1000;
GridIncrement = 10;
CalibrationGrid = BuildGrid(GridIncrement,GridWidth);

% 3. Choose somewhere in space for the grid
% T_ow is the 4x4 tranformation matrix from grid to world.
T_ow = PositionGrid();

% Define the scaling to use
if CameraHeight > CameraWidth
    CameraScale = 2.0 / CameraHeight;
else
    CameraScale = 2.0 / CameraWidth;
end
GridScale = 2.0/GridWidth;

% Generate the calibration images and the homographies
% Store Homographies and concensus sets in a Matlab Cell Array
% called HomogData.
HomogData = cell(nImages,3);

for CalImage = 1:nImages

    % Keep looking for homographies until we get a non-zero result.
    % 'Estimating' is a toggle.
    Estimating = 1;
    while Estimating == 1
        % The default is 'Success', i.e. Estimating = 0
        Estimating = 0;

        % 4 Choose a 'random' location for the camera that fills the image
            .
        % T_cw is the 4x4 transformation matrix from camera to world
        T_cw = FillImage(T_ow,KMatrix,GridWidth,CameraHeight,CameraWidth);

        % 5 We now fill the camera with a noisy image of the grid and
            generate
        % the point correpondences.
```

```
64          % Correspond is a set of pairs of vectors of the form [[u v]' [x y
               ]']
65          % for each grid corner that lies inside the image.
66          Correspond = BuildNoisyCorrespondences(T_ow,T_cw,CalibrationGrid,
               ...
67              KMatrix,CameraHeight,CameraWidth);
68
69          % 6. Add in some 'outliers' by replacing [u v]' with a point
70          % somewhere in the image.
71          % Define the Outlier probability
72          pOutlier = 0.05;
73          for j = 1:length(Correspond)
74              r = rand;
75              if r < pOutlier
76                  Correspond(1,j) = rand * (CameraWidth-1);
77                  Correspond(2,j) = rand * (CameraHeight-1);
78              end
79          end
80
81          % Now scale the grid and camera to [-1, 1] to improve
82          % the conditioning of the Homography estimation.
83          Correspond(1:2,:) = Correspond(1:2,:)*CameraScale - 1.0;
84          Correspond(3:4,:) = Correspond(3:4,:)*GridScale;
85
86          % 7. Perform the Ransac estimation - output the result for
               inspection
87          % If the Ransac fails it retuns a zero Homography
88          Maxerror = 3.0; % The maximum error allowed before rejecting a
               point.
89          % I am using a variance of 0.5 pixels so sigma is sqrt(0.5)
90          % 3 pixels in the *NORM* is 3 sigma as there are 2 errors involved
91          % in the norm (u and v).
92          %
93          % Note: The above is in pixels - so scale before Ransac!
94          RansacRuns = 50;  % The number of runs when creating the consensus
               set.
95          [Homog, BestConsensus] = ...
96              RansacHomog(Correspond,Maxerror*CameraScale,RansacRuns);
97
98          if Homog(3,3) > 0
99              % This image worked.  So record the homography and the
100             % consensus set
101             HomogData{CalImage,1} = Homog;
102             HomogData{CalImage,2} = Correspond;
103             HomogData{CalImage,3} = BestConsensus;
104         else
105             % The estimate failed.  So go around again.
106             Estimating = 1;
107         end
```

```
108
109     end % end of the while Estimating == 1 loop
110 end % end of the nImages loop
111
112 % 8. Build the regressor for estimating the Cholesky product
113 Regressor = zeros(2*nImages,6);
114 for CalImage = 1:nImages
115     r1 = 2*CalImage-1;
116     r2 = 2*CalImage;
117     Regressor(r1:r2,:) = KMatrixRowPair(HomogData{CalImage,1});
118 end
119
120 % Find the kernel
121 [U,D,V] = svd(Regressor,'econ');
122 D = diag(D);
123 [M,I] = min(D);
124 % K is the estimate of the kernel
125 K = V(:,I);
126
127 % The matrix to be constructed needs to be positive definite
128 % It is necessary that K(1) be positive.
129 if K(1) < 0
130     K = -K;
131 end
132
133 % Construct the matrix Phi from the kernel
134 Phi = zeros(3);
135
136 Phi(1,1) = K(1);
137 Phi(1,2) = K(2);
138 Phi(1,3) = K(3);
139 Phi(2,2) = K(4);
140 Phi(2,3) = K(5);
141 Phi(3,3) = K(6);
142
143 % Add in the symmetric components
144 Phi(2,1) = Phi(1,2);
145 Phi(3,1) = Phi(1,3);
146 Phi(3,2) = Phi(2,3);
147
148 % Check if the matrix is positive definite
149 e = eig(Phi);
150 for j = 1:3
151     if e(j) <= 0
152         error('The Cholesky product is not positive definite')
153     end
154 end
155
156 % 9. Carry out the Cholesky factorization
```

```
157 KMatEstimated = chol(Phi);
158
159 % Invert the factor
160 KMatEstimated = KMatEstimated \ eye(3);
161
162 % The scaling of the grid has no impact on the scaling of
163 % the K-matrix as the vector 't' takes no part in the estimate
164 % of Phi.  Only the image scaling has an impact.
165
166 % first normalize the KMatrix
167 KMatEstimated = KMatEstimated / KMatEstimated(3,3);
168
169 % Add 1.0 to the translation part of the image
170 KMatEstimated(1,3) = KMatEstimated(1,3) + 1;
171 KMatEstimated(2,3) = KMatEstimated(2,3) + 1;
172
173 % Rescale back to pixels
174 KMatEstimated(1:2,1:3) = KMatEstimated(1:2,1:3) / CameraScale;
```

../Matlab/EstimateKMatrix/RunEstimateKMatrix.m

7 Dealing with estimation bias - bundle adjustment

The objective of the estimation tasks so far has been to generate a good 'seed' to start the optimization of the final model. We could not just 'optimize' the problem *ab initio* as the model is highly non-linear and involves rotations that pose particular problems for optimizers. We have thus assumed that all errors introduced at each stage are 'small' and that we can use the geometric constraints imposed by the calibration object to reason about the generation of a good initial estimate. All of the methods adopted so far have assumed that errors are independent and are 'white'. But the estimation of the initial homographies introduces bias, and analyzing how errors in the homographies impact the kernel computation leading to the matrix ϕ would be problematic. It is unlikely that the estimate that we arrive at above is 'optimal'.

7.1 Back projection error - the minimum variance solution

Given a model of the camera we can predict where the image points should be for every grid Point if we know where the calibration object is located. Computing the image location of a grid point given a model is called back projection but the back projected positions will not agree with the measured position in the image as we will not have the correct model. These errors are back projection errors.

We have a set of point-Point correspondences from the final consensus set for each homography that was estimated. We multiplied these homographies by the inverse of the camera K-matrix to get a set of perspectivities that encode the position of the object in each image, expressed in the camera frame. As a result we can be confident that the grid Points within these consensus sets are properly matched to corresponding image points. We thus assume that the back projection errors in the image locations within these consensus sets are white and unbiased, i.e. the expected value of the image point position errors is zero. If we were to compute the sum of the squares of the errors and we knew the true positions of the image points, then the result would be a minimum when using the true points (we assume the expected values are the 'true' values) i.e. variance is at a minimum when the expected value of a measurement is subtracted from each measurement before squaring and adding. We can use this fact to optimize our model.

Take all the points that lie within the maximal consensus sets (rejecting all outliers found in the final phases of the homography estimation). Define a cost function to be the sum of the square of the differences between the image locations (in pixels) and the predicted locations (given the position of the calibration object). Our objective is to minimize this cost by adjusting the K-matrix and the supposed locations of the calibration object in each image. If we achieve a minimum, as the errors are actually in pixels, they have the same statistics as the measurement process, i.e. the bias introduced by the complex geometric manipulations above should be removed. The resulting model should be 'optimal' in the sense that it is a minimum variance estimate of the camera model when the variance is computed in the measurement space. This process is called 'bundle adjustment'.

7.2 Estimating the initial calibration object locations

We are given for each image 'j'

$$\mathbf{H}_j = \lambda_j \mathbf{K} \left[\begin{array}{ccc} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{t} \end{array} \right]_j . \tag{7.1}$$

Thus

$$\lambda_j \left[\begin{array}{ccc} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{t} \end{array} \right]_j = \mathbf{K}^{-1} \mathbf{H}_j \tag{7.2}$$

However, there are errors in the estimates of all of the parameters in the model. Thus the perspectivity is unlikely to even be a true perspectivity (it probably will not be an exact rigid body rotation and dilation of a plane). We thus perform the following operations:

1. Find the norm of the first column of the matrix $\mathbf{K}^{-1}\mathbf{H}_j$. This is our initial estimate of λ_j. Divide $\mathbf{K}^{-1}\mathbf{H}_j$ by λ_j, thus normalizing the first column and generating our first estimate of $\mathbf{r}_1$.

2. Subtract $\left(\mathbf{r}_1^T \mathbf{r}_2\right) \mathbf{r}_1$ from $\mathbf{r}_2$ to obtain $\hat{\mathbf{r}}_2$. This operation makes the first two vectors orthogonal.

3. Divide $\hat{\mathbf{r}}_2$ by its norm to get a new $\mathbf{r}_2$.

4. Find $\mathbf{r}_3 = \mathbf{r}_1 \times \mathbf{r}_2$ and construct the matrix $\left[\begin{array}{ccc} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 \end{array} \right]$. This is the initial rotation matrix of the 4×4 transformation matrix that converts Points in the object coordinates to Points in the unit camera coordinates.

5. Find the eigenvector of the rotation matrix associated with the eigenvalue 1.0. This is the axis of rotation. Note that this vector can be multiplied by -1, i.e. the final answer may be anti-parallel to the result reported by Matlab.

6. Find the sum of the diagonal elements of the rotation matrix (the trace). This sum is equal to $1 + 2\cos\theta$ where θ is the angle of rotation. Solve for θ in the range $[-\pi\ \pi)$.

7. Plug the axis of rotation and angle of rotation into Rodrigue's formula (equation 5.3) for the axis of rotation and and angle of rotation. Compare with the rotation matrix that has just been constructed and change the direction of the axis of rotation by multiplying it by -1 if there is a large error. Use the axis of rotation and angle of rotation to construct a 3-element shifted angle-axis rotation of the rotation matrix (as described by equation 5.4). The purpose of this step is to obtain a *representation* of the rotation part of the transformation matrix ready for optimization and as the angle of rotation is represented as the norm of the angle-axis representation it must be a positive number.

8. Construct the transformation matrix $\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$. This matrix is our initial estimate of how Points in the object frame are transformed into Points in the unit camera frame.

At the end of the above process we have the following parameters:

K-matrix 5 independent parameters in pixels (or scaled units) describing the camera (noting the bottom right of the matrix is set to 1).

Angle-axis 3 parameters for each image, describing the rotation part of the transformation matrix.

Position vector 3 measurements in millimetres for each image (or in scaled units if the grid is scaled) describing the translation between the chosen object origin and the camera origin.

If there are n images, there are $5 + 6 \times n$ parameters to estimate. The proposed optimization method is called the Levenberg-Marquardt algorithm.

7.3 Levenberg-Marquardt

We need to minimize the variance of the back-projection errors of the consensus sets of calibration grid-Point image grid-point correspondences. We start with the initial sets of parameters as described above. The process required is

1. Convert the current set of 'best' positional parameters to a set of n 3×3 Perspectivites.

2. Predict the positions of the points in the image using these transformation matrices and the current best camera K-matrix.

3. Subtract the actual image positions from the predicted image positions to generate a set of image errors in pixels.

4. Square and sum the errors.

5. Adjust the parameters and go back to the start. Stop when the error reaches a minimum.

We will use the Levenberg-Marquardt algorithm to adjust the parameters and to decide when a minimum as been reached.

We have a cost function that is the sum of squares of errors (a chi-squared statistic that, for convenience, we will call E) that is a non-linear function of a set of adjustable parameters $\mathbf{p}$ (elements of the K-matrix plus a set of rotations and translations). If we perturb the parameters, then the Taylor series of the error function is

$$E(\mathbf{p}+\delta\mathbf{p}) = E_0 + \left(\frac{\partial E}{\partial \mathbf{p}}\right)^T \delta\mathbf{p} + \frac{1}{2}\delta\mathbf{p}^T \left(\frac{\partial^2 E}{\partial \mathbf{p}^2}\right)\delta\mathbf{p} + \text{higher order terms} \quad (7.3)$$

The above is a condensed vector representation of a Taylor series in many dimensions. It is just the same as the single variable Taylor series that you saw in P1 but now has partial derivatives that have the following names and meanings:

- E_0 is the value of the cost function with the *current* set of parameters. After each round of adjustment we have a new value of E_0.

- $\delta\mathbf{p}$ is the set of small changes to the parameters that we are going to add.

- $\left(\frac{\partial E}{\partial \mathbf{p}}\right)^T \delta\mathbf{p}$ is shorthand for $\sum_k \frac{\partial E}{\partial p_k}\delta p_k$. $\left(\frac{\partial E}{\partial \mathbf{p}}\right)^T$ is called the gradient or direction or steepest descent (expressed as a row vector).

- $\frac{\partial^2 E}{\partial \mathbf{p}^2}$ is a matrix of all the second derivatives of the error function with respect to the parameters and is called the Hessian. If this matrix is positive definite (all of its eigenvalues are positive and real) and $\left(\frac{\partial E}{\partial \mathbf{p}}\right)^T = \mathbf{0}$, then we have a minimum.

Before we move to the Levenberg-Marquardt algorithm itself, we will first present some useful ideas.

7.3.1 Steepest descent

The simplest way to reduce the error is to find $\left(\frac{\partial E}{\partial \mathbf{p}}\right)^T$ and choose $\delta\mathbf{p}$ to be parallel to this 'steepest descent' direction and keep going 'downhill' until E starts to increase when you reach the bottom. This is called the method of steepest descent and is not recommended. First, you don't know how far to go for each step downhill, and second the direction of steepest descent changes as the parameters change! But it is a good method for making small exploratory steps.

7.3.2 Newton steps

The Hessian is the derivative of the gradient, thus

$$\frac{\partial E}{\partial \mathbf{p}}(\mathbf{p}+\delta\mathbf{p}) \approx \frac{\partial E}{\partial \mathbf{p}}(\mathbf{p}) + \frac{\partial^2 E}{\partial \mathbf{p}^2}\delta\mathbf{p}. \tag{7.4}$$

At the minimum $\left(\frac{\partial E}{\partial \mathbf{p}}\right) = \mathbf{0}$. So we wish

$$\mathbf{0} = \frac{\partial E}{\partial \mathbf{p}}(\mathbf{p}) + \frac{\partial^2 E}{\partial \mathbf{p}^2}\delta\mathbf{p}, \tag{7.5}$$

i.e.

$$\delta\mathbf{p} = -\left(\frac{\partial^2 E}{\partial \mathbf{p}^2}\right)^{-1}\frac{\partial E}{\partial \mathbf{p}}(\mathbf{p}) \tag{7.6}$$

will generate a jump to the minimum should the Hessian be constant and positive definite. This is a Newton step. It is unlikely to work unless you are very near to the minimum.

7.3.3 Computing an approximation to the Hessian - the Jacobian

Let us label each image measurement as m_k, where we do not distinguish between u and v measurements. The predicted measurement k will be a nonlinear function of all the parameters, let us call these functions f_k. There will be as many functions as measurements included in the estimation. Then the error function is given by

$$E = \frac{1}{2}\sum_k \left(f_k(\mathbf{p}) - m_k\right)^2 \tag{7.7}$$

There will be a set of parameters, the number depending on the number of images included in the estimation. Label each parameter by the letter i. Then

$$\frac{\partial E}{\partial p_i} = \sum_k \left(f_k(\mathbf{p}) - m_k\right)\frac{\partial f_k}{\partial p_i} \tag{7.8}$$

The Hessian is the matrix of second derivatives, so let us represent a second parameter with the letter j. Then

$$\frac{\partial^2 E}{\partial p_i \partial p_j} = \sum_k \frac{\partial f_k}{\partial p_j}\frac{\partial f_k}{\partial p_i} + (f_k(\mathbf{p}) - m_k)\frac{\partial^2 f_k}{\partial p_i \partial p_j} \tag{7.9}$$

If we are near to the minimum, which we assume is the case given the care we have used in finding a good starting 'seed' via linear algebra, we note that the measurements should be very close to the estimates, thus $f_k(\mathbf{p}) - m_k \approx 0$ and this term multiplies the second derivatives of the measurement functions inside the sum. We thus drop the second term to get

$$\frac{\partial^2 E}{\partial p_i \partial p_j} \approx \sum_k \frac{\partial f_k}{\partial p_j}\frac{\partial f_k}{\partial p_i} \tag{7.10}$$

Having considered the first and second order derivatives of the error function we now return to the gradient and Hessian.

The gradient is a row vector and using equation 7.8 is given by

$$\left(\frac{\partial E}{\partial \mathbf{p}}\right)^T = \sum_k (f_k(\mathbf{p}) - m_k)\left[\begin{array}{cccc} \frac{\partial f_k}{\partial p_1} & \frac{\partial f_k}{\partial p_2} & \cdots & \frac{\partial f_k}{\partial p_{5+6n}} \end{array}\right] \tag{7.11}$$

The gradient is thus the weighted sum of row vectors, each row vector being the derivatives of a particular non-linear function associated with a measurement with respect to each parameter in turn (and many of these entries will be zero for measurements for an image with respect to parameters describing another image). If we stack all of these row vector up we will construct a rectangular matrix called the Jacobian. This matrix will have as many rows as there are measurements and as many columns as there are parameters. The gradient function can then be expressed as

$$\left(\frac{\partial E}{\partial \mathbf{p}}\right)^T = \mathbf{e}^T\mathbf{J} \tag{7.12}$$

where $\mathbf{e}$ is a vector of measurement errors $f_k(\mathbf{p}) - m_k$ and $\mathbf{J}$ is the Jacobian.

The Hessian $\mathbf{H}$ can then be expressed as

$$\mathbf{H} \approx \frac{\partial^2 E}{\partial p_i \partial p_j} = \mathbf{J}^T\mathbf{J}. \tag{7.13}$$

7.3.4 Combining steepest descent and Newton steps

The essence of the Levenberg-Marquardt algorithm is to combine the method of steepest descent and Newton steps. The algorithm consists of the following:

1. Compute the error vector $\mathbf{e}$ using the current set of parameters.
2. Choose a weighting μ.
3. Solve $\left(\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}\right)\delta\mathbf{p} = -\mathbf{J}^T\mathbf{e}$.
4. Recompute the total cost E for the new parameters $\mathbf{p} + \delta\mathbf{p}$.
5. If the cost increased, then reject the new parameters, increase μ, and repeat the above. If the cost went down, accept the new parameters, reduce μ and repeat the above.

The above algorithm causes the search to move from steepest descent to Newton steps depending on how well the last search performed. If the problem is behaving badly, the weighting μ is increased, reducing the step size and causing the algorithm to just search downhill. If the algorithm is doing well it assumes it is close to the minimum and so relies more on Newton steps.

Note that your code should consider carefully the block structuring of the Jacobian and avoid unnecessary matrix multiplies because of these zeros.

Stopping criterion

The minimum is reached when the gradient is small. One possible measure of 'small' is the norm of the vector $\mathbf{J}^T\mathbf{e}$ divided by the number of measurements is small. But think about the units of each of the derivatives - the unscaled measurement is in pixels and the units used for the parameters are pixels, radians and millimetres, so the derivatives are pixels per pixel, pixels per radian and pixels per millimetre. You may wish to weight the elements of the derivative before computing the norm. Some discussion of the stopping criterion will be looked for in your report. This consideration is another advantage of scaling.

How to vary μ

First we need to consider the initial value of μ. I suggest finding the largest value of the diagonal of $\mathbf{J}^T\mathbf{J}$ and use a small multiple of this value, say 0.1 times or 0.01 times.

Once the initial value of μ has been chosen I suggest you use a method proposed by K Madsen, H.B. Nielsen and O.Tingleff in 2004. This method involves comparing the reduction in the cost function when compared to a predicted reduction.

If we know the proposed change in the parameters the Taylor equation 7.3 may be used to predict the change in error

$$\Delta E_{\text{pred}} \approx \left(\frac{\partial E}{\partial \mathbf{p}}\right)^T \delta\mathbf{p} \tag{7.14}$$

If the total error goes down we also compute the *actual* change in cost ΔE and define the cost ratio

$$\rho = \frac{\Delta E}{\Delta E_{\text{pred}}} \tag{7.15}$$

If $\rho > 1$ then the cost went down by more than we were expecting, if $\rho < 1$ then it went down by less than we were expecting.

We next define a number ν that is initially set to 2. If the cost goes up then ν is multiplied by 2. If the cost goes down then ν is reset to 2. The number ν controls the rate of growth of the damping variable μ in the algorithm. The value of μ is adjusted by the following algorithm on every iteration of the main Levenberg-Marquardt algorithm:

```
if rho < 0
  % The cost has gone down (accept the new parameters)
  nu = 2;
  mu = mu * max([1/3,1-(2*rho-1)^3]);
else
  % The cost has gone up (reject the new parameters)
  mu = mu * nu;
  nu = nu * 2;
end
```

The above defines a regime in which the damping μ changes smoothly should the cost be decreasing, but ramps up very quickly if the cost goes up.

Computing the Jacobian

The elements in the Jacobian are computed from denormalized components of a homogeneous vector, where both the numerator and denominator of the ratios involved

are themselves complex functions of offsets and rotations. Thus analytic computation is not really feasible. We thus use the forward difference, i.e. change each variable in turn by a small amount and compute the new position error and divide by the change in parameter. Be careful that you do not choose too small a change in the parameter, otherwise you will suffer from numerical roundoff and get a poor estimate. Also be aware of the very different units that characterize each of the different parameters, i.e. a change of 1mm in the vector **t** has a very different effect to a change of 1 radian in the angle-axis representations of the rotation matrices. Also note where blocks of zeros will occur in the Jacobian because changes to the position associated with image 'i' will not affect the projected position of the grid Points in image 'j'.

The block structure of the Jacobian

The variables that need to be adjusted are the 5 unknown elements of the intrinsic model, which is an upper triangular matrix with the bottom right element fixed at 1.0, and 6 parameters describing the location of the calibration grid in each image (the position of the grid with respect to the camera in each calibration image). Changing the parameters in the intrinsic model will change the errors for each image. But changing the location of the grid in one image will have no effect on the errors associated with the other images. Thus the Jacobian in block structured. Figure 7.1

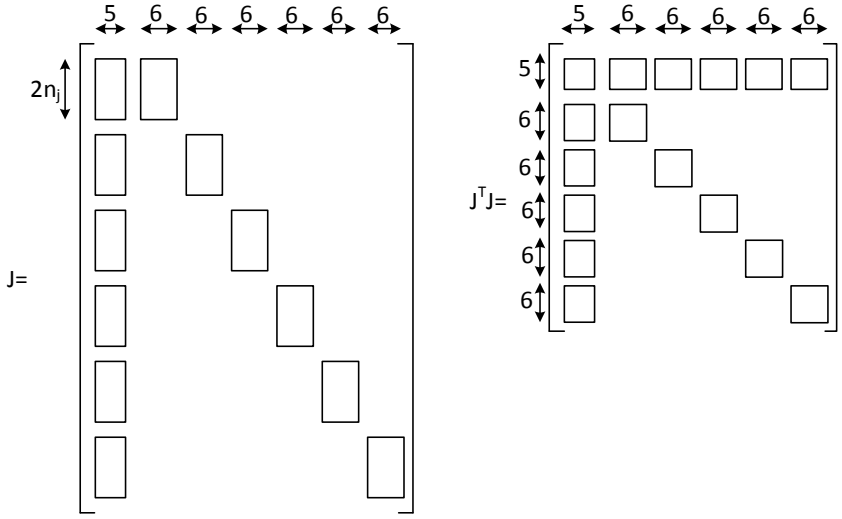


Figure 7.1: The Jacobian block structure.

illustrates the block structure of the Jacobian **J** and the product $\mathbf{J}^T\mathbf{J}$. The columns of the Jacobian correspond to each parameter that is being adjusted. The first five columns are the K-Matrix parameters. There are then groups of 6 parameters for

each image, 3 rotation and 3 translations per image. The rectangles represent the non-zero blocks of entries. If there are n_j valid points in an image's consensus set, there there will be $2n_j$ measurements (u and v). Thus each rectangle will have height $2n_j$, each row corresponding to the changes in the error of a single measurement.

Only the blocks need to be stored, and I use the Matlab `cell` structure here because the number of valid measurements is different in each image. The block structure can be used to advantage when computing $\mathbf{J}^T\mathbf{J}$, which is symmetric and is shown on the right of the figure. Note that $\mathbf{J}^T\mathbf{J}$ is $(5+6n)\times(5+6n)$. There is one 5×5 block on the diagonal corresponding to the inner product of the first 5 columns of the Jacobian. There are n 6×6 blocks down the diagonal corresponding to the inner products of each of the independent image columns with themselves, one per image. The off-diagonal blocks at the top are associated with the inner products of the K-matrix parameter blocks with the position dependent blocks for each individual image.

My Code

In my solution I decided to use a shifted form of the angle-axis representation by adding 4π to the angle of rotation before scaling the axis of rotation. This has the advantage of avoiding discontinuities around a rotation angle of zero. I keep a watch on far the representation moves from 4π to avoid unstable increases or decreases in rotation angle. This representation still has discontinuities, but they have been shifted outside the codes presumed working range.

The following two listings are the top level script to run the estimation using all the functions developed so far plus my implementation of Levenberg-Marquardt. I include the optimizer as this is challenging code to write. A typical run requires 4 iterations to reach the stopping condition set - try different stopping conditions and investigate how the code behaves as it finds the best solution.

Please do not just copy this code - think about how I have approached the problem and provide your own solutions. I have not included the code for the Jacobian and error computation. This is for you to write.

```
1 % This script performs a full estimation and optimisation of
2 % a camera K-matrix.
3 %
4 % 1. Construct a camera model loosely based on an iPhone6
5 % 2. Construct a calibration grid 1m on a side with 10mm grid spacing.
6 % 3. Position the grid somewhere in space.
7 %
8 % Perform the following actions on each image, repeating an image
9 % if the homography estimation failed.
```

```
% 4. Place the camera somewhere in space to generate a full image
%    of tile square corner locations.
% 5. Generate the noisy image of the grid.
% 6. Add in some outliers.
% 7. Perform a Ransac estimate of the homography
%
% Once the homographies have been estimated
% 8. Build the regressor for estimating the K-matrix
% 9. Carry out the Cholesky factorization and invert.  This generates
% an initial model of the K-Matrix

% The number of calibration images to use
nImages = 6;

% 1. Construct the Camera model
[KMatrix, CameraHeight, CameraWidth] = BuildCamera;

% 2. Construct a 1m by 1m grid with 10mm tiles in the grid frame
% The grid is a set of 4-element vectors [x y 0 1]'.
GridWidth = 1000;
GridIncrement = 10;
CalibrationGrid = BuildGrid(GridIncrement,GridWidth);

% 3. Choose somewhere in space for the grid
% T_ow is the 4x4 tranformation matrix from grid to world.
T_ow = PositionGrid();

% Define the scaling to use
if CameraHeight > CameraWidth
    CameraScale = 2.0 / CameraHeight;
else
    CameraScale = 2.0 / CameraWidth;
end
GridScale = 2.0/GridWidth;

% Generate the calibration images and the homographies
% Store Homographies and concensus sets in a Matlab Cell Array
% called HomogData.
HomogData = cell(nImages,3);
% Define the numbers to access the DataCell
NHOMOGRAPHY = 1;
NCORRESPOND = 2;
NCONSENSUS = 3;

for CalImage = 1:nImages

    % Keep looking for homographies until we get a non-zero result.
    % 'Estimating' is a toggle.
    Estimating = 1;
```

```
    while Estimating == 1
        % The default is 'Success', i.e. Estimating = 0
        Estimating = 0;

        % 4 Choose a 'random' location for the camera that fills the
        % image.
        % T_cw is the 4x4 transformation matrix from camera to world
        T_cw = FillImage(T_ow,KMatrix,GridWidth,CameraHeight,CameraWidth);

        % 5 We now fill the camera with a noisy image of the grid and
        % generate the point correpondences.
        % Correspond is a set of pairs of vectors of the form
        % [[u v]' [x y]'] for each grid corner that lies inside the
        % image.
        Correspond = BuildNoisyCorrespondences(T_ow,T_cw,...
            CalibrationGrid,KMatrix,CameraHeight,CameraWidth);

        % 6. Add in some 'outliers' by replacing [u v]' with a point
        % somewhere in the image.
        % Define the Outlier probability
        pOutlier = 0.05;
        for j = 1:length(Correspond)
            r = rand;
            if r < pOutlier
                Correspond(1,j) = rand * (CameraWidth-1);
                Correspond(2,j) = rand * (CameraHeight-1);
            end
        end

        % Now scale the grid and camera to [-1, 1] to improve
        % the conditioning of the Homography estimation.
        Correspond(1:2,:) = Correspond(1:2,:)*CameraScale - 1.0;
        Correspond(3:4,:) = Correspond(3:4,:)*GridScale;

        % 7. Perform the Ransac estimation
        % If the Ransac fails it retuns a zero Homography

         % The maximum error allowed before rejecting a point.
        Maxerror = 3.0;
        % I am using a variance of 0.5 pixels so sigma is sqrt(0.5)
        % 3 pixels in the *NORM* is 3 sigma as there are 2 errors involved
        % in the norm (u and v).
        %
        % Note: The above is in pixels - so scale before Ransac!
        % The number of runs when creating the consensus set.
        RansacRuns = 50;
        [Homog, BestConsensus] = ...
            RansacHomog(Correspond,Maxerror*CameraScale,RansacRuns);

```

```
        if Homog(3,3) > 0
            % This image worked.  So record the homography and the
            % consensus set
            HomogData{CalImage,NHOMOGRAPHY} = Homog;
            HomogData{CalImage,NCORRESPOND} = Correspond;
            HomogData{CalImage,NCONSENSUS} = BestConsensus;
        else
            % The estimate failed.  So go around again.
            Estimating = 1;
        end

    end % end of the while Estimating == 1 loop
end % end of the nImages loop

% 8. Build the regressor for estimating the Cholesky product
Regressor = zeros(2*nImages,6);
for CalImage = 1:nImages
    r1 = 2*CalImage-1;
    r2 = 2*CalImage;
    Regressor(r1:r2,:) = KMatrixRowPair(HomogData{CalImage,1});
end

% Find the kernel
[U,D,V] = svd(Regressor,'econ');
D = diag(D);
[M,I] = min(D);
% K is the estimate of the kernel
K = V(:,I);

% The matrix to be constructed needs to be positive definite
% It is necessary that K(1) be positive.
if K(1) < 0
    K = -K;
end

% Construct the matrix Phi from the kernel
Phi = zeros(3);

Phi(1,1) = K(1);
Phi(1,2) = K(2);
Phi(1,3) = K(3);
Phi(2,2) = K(4);
Phi(2,3) = K(5);
Phi(3,3) = K(6);

% Add in the symmetric components
Phi(2,1) = Phi(1,2);
Phi(3,1) = Phi(1,3);
Phi(3,2) = Phi(2,3);
```

```
157
158 % Check if the matrix is positive definite
159 e = eig(Phi);
160 for j = 1:3
161     if e(j) <= 0
162         error('The Cholesky product is not positive definite')
163     end
164 end
165
166 % 9. Carry out the Cholesky factorization
167 KMatEstimated = chol(Phi);
168
169 % Invert the factor
170 KMatEstimated = KMatEstimated \ eye(3);
171
172 % The scaling of the grid has no impact on the scaling of
173 % the K-matrix as the vector 't' takes no part in the estimate
174 % of Phi.  Only the image scaling has an impact.
175
176 % first normalize the KMatrix
177 KMatEstimated = KMatEstimated / KMatEstimated(3,3);
178
179 % Optimize the K-matrix
180 OptKMatrix = OptimiseKMatrix(KMatEstimated,HomogData);
181
182 % Add 1.0 to the translation part of the image
183 KMatEstimated(1,3) = KMatEstimated(1,3) + 1;
184 KMatEstimated(2,3) = KMatEstimated(2,3) + 1;
185
186 % Rescale back to pixels
187 KMatEstimated(1:2,1:3) = KMatEstimated(1:2,1:3) / CameraScale;
```

../Matlab/OptimiseKMatrix/RunOptimiseKMatrix.m

```
 1 function [ KMatrix ] = OptimiseKMatrix( InitialKMatrix,Data )
 2 %OptimiseKMatrix Optimises a K-matrix using Levenberg-Marquardt
 3 %
 4 % InitialKMatrix is the 'seed' K-matrix to start the optimization.
 5 % Data is a Matlab cell structure containing homogrpahies,
 6 % corresponding points in the form [[u v]' [x y]'], and the
 7 % indices of the consensus sets generated during the Ransac
 8 % estimation of the homographies.
 9 %
10 % The algorithm (Levenberg-Marquadt)
11 % 1. Compute a shifted angle-axis representation of the
12 %    rotation matrix.  Store with the initial translation vector.
13 % 2. Compute the error vector, e,  and total error.
14 %    Compute the Jacobian, J, and J'J.  Find the maximum element of
15 %    J'J, multiply by, say, 0.1 and set mu to this value.
16 % 3. Solve (J'J + muI)dp = -J'e
```

```
% 4. If e'J is 'small' we have converged - terminate.
% 5. Compute predicted change in the error as e'Jdp
% 6. Save current parameters and compute the error with new parameters.
% 7. Find change in error divided by predicted change in error = gain.
% 8. a) If the error went up recompute mu using equation from notes and
%    restore the temporary parameters and repeat.
%    b) If the error went down accept the new parameters and recompute
%    mu using the equation from the notes.
% 9. Go to step 3.
%
% A note on the Jacobian.
% The Jacobian is expensive to compute, thus I keep the same Jacobian
% until it becomes a poor predictor of the change in error - I then
% recompute.

% Carry out your sanity checks here ....

% Initialise the KMatrix
KMatrix = InitialKMatrix;

% Normalize the K-matrix (just in case0
KMatrix = KMatrix / KMatrix(3,3);

% Define the numbers to access the DataCell
NHOMOGRAPHY = 1;
NCORRESPOND = 2;
NCONSENSUS = 3;

% Extract the number of images used in the estimation
s = size(Data);
nImages = s(1);

% A rotation matrix to be constructed.
RotMat = zeros(3);

% 1. We first need to use the initial KMatrix to generate
% a perspectivity for each image and hence the parameterization
% of the grid's frame in the camera frame.
FrameParameters = cell(nImages,2);
% Labels for accessing the cell
NANGLE = 1;
NTRANSLATION = 2;

for nHomog = 1:nImages
    % Extract the homography from the data and convert to
    % a perspectivity
    Perspectivity = KMatrix \ Data{nHomog,NHOMOGRAPHY};

    % The Perspectivity is scaled so that its bottom right hand
```

```
66      % is a unit vector - we need the first column to have a norm
67      % of 1 (a column of a rotation matrix)
68      Perspectivity = Perspectivity / norm(Perspectivity(:,1));
69
70      % The translation part of the perspectivity in the camera frame
71      Translation = Perspectivity(:,3);
72
73      % Start building the rotation matrix
74      RotMat(:,1:2) = Perspectivity(:,1:2);
75      % Project out the first column from the second
76      RotMat(:,2) = RotMat(:,2) - (RotMat(:,1)'*RotMat(:,2))*RotMat(:,1);
77      RotMat(:,2) = RotMat(:,2) / norm(RotMat(:,2));
78      % And complete with a cross product
79      RotMat(:,3) = cross(RotMat(:,1),RotMat(:,2));
80
81      % Find the angle of rotation usng the trace identity
82      CosTheta = (trace(RotMat)-1)/2;
83
84      % Make sure the answer is valid
85      if CosTheta > 1
86          CosTheta = 1;
87      end
88
89      if CosTheta < -1
90          CosTheta = -1;
91      end
92      % Theta is the angle of rotation
93      Theta = acos(CosTheta);
94
95      % Find the axis of rotation by finding the real eigenvector
96      [V,D] = eig(RotMat);
97      D = diag(D);
98      % I will be the real vector.
99      [M,I] = max(real(D));
100     RotAxis = real(V(:,I));
101     RotAxis = RotAxis / norm(RotAxis);
102
103     % We need to check if the axis is pointing in the 'right' direction
104     % consistent with Theta using Rodrigue's formula
105     Rplus = Rodrigues(RotAxis,Theta);
106     Rminus = Rodrigues(-RotAxis,Theta);
107
108     % Reverse the axis if minus Theta is the best match
109     if norm(RotMat-Rminus) < norm(RotMat-Rplus)
110         RotAxis = -RotAxis;
111     end
112
113     % Now shift the angle by 4pi so that we can have negative as
114     % well as positive angles
```

```
115     Theta = Theta+4*pi;
116     % Generate a shifted angle-axis representation of the rotation.
117     RotAxis = RotAxis*Theta;
118 
119     % Record the initial position of the grid
120     FrameParameters{nHomog,NANGLE} = RotAxis;
121     FrameParameters{nHomog,NTRANSLATION} = Translation;
122 
123 
124 end
125 
126 % Now go through the LM steps.  There are a variable number of
127 % measrements, so use another Matlab cell structure to store the
128 % various components.
129 % The components are:
130 % 1. Error Vector.
131 % 2. KMatrixJacobian.
132 % 3. FrameParametersJacobian.
133 % The above correspond to the following cell entries.
134 NERRORVECTOR = 1;
135 NKMATJACOB = 2;
136 NFRAMEJACOB = 3;
137 
138 OptComponents = cell(nImages,3);
139 
140 % Allocate space for J'J
141 ProblemSize = 5+6*nImages;
142 JTransposeJ = zeros(ProblemSize);
143 
144 % Initialise the total error
145 CurrentError = 0.0;
146 
147 % Initialise the Gradient as a column vector
148 Gradient = zeros(ProblemSize,1);
149 
150 % 2. Compute the inital error vector and the Jacobians and the inner
151 % product
152 for j = 1:nImages
153     % The error vector for each image
154     OptComponents{j,NERRORVECTOR} = ComputeImageErrors( KMatrix, ...
155         FrameParameters{j,NANGLE}, FrameParameters{j,NTRANSLATION},...
156         Data{j,NCORRESPOND}, Data{j,NCONSENSUS});
157 
158     % Compute the initial error
159     CurrentError = CurrentError + 0.5 * ...
160         OptComponents{j,NERRORVECTOR}'*OptComponents{j,NERRORVECTOR};
161 
162     % The Jacobian for each image
163     [OptComponents{j,NKMATJACOB}, OptComponents{j,NFRAMEJACOB}] = ...
```

```
164         SingleImageJacobian( KMatrix,...
165            FrameParameters{j,NANGLE}, FrameParameters{j,NTRANSLATION},...
166            Data{j,NCORRESPOND}, Data{j,NCONSENSUS});
167
168     % The top 5x5 block is the sum of all the inner products of the
169     % K-matrix Jacobian blocks.
170     JTransposeJ(1:5,1:5) = JTransposeJ(1:5,1:5)+ ...
171         OptComponents{j,NKMATJACOB}' * OptComponents{j,NKMATJACOB};
172
173     % The diagonal image block associated with the frame parameters
174     StartRow = 6 + (j-1)*6;
175     EndRow = StartRow+5;
176     JTransposeJ(StartRow:EndRow,StartRow:EndRow) = ...
177         OptComponents{j,NFRAMEJACOB}' * OptComponents{j,NFRAMEJACOB};
178
179     JTransposeJ(1:5,StartRow:EndRow) = ...
180         OptComponents{j,NKMATJACOB}' * OptComponents{j,NFRAMEJACOB};
181
182      JTransposeJ(StartRow:EndRow,1:5) =  JTransposeJ(1:5,StartRow:EndRow)
           ';
183
184      % Compute the gradient vector
185      Gradient(1:5) = Gradient(1:5) + ...
186          OptComponents{j,NKMATJACOB}'*OptComponents{j,NERRORVECTOR};
187      Gradient(StartRow:EndRow) = OptComponents{j,NFRAMEJACOB}'*...
188          OptComponents{j,NERRORVECTOR};
189
190 end
191
192 % The initial value of mu
193 mu = max(diag(JTransposeJ)) * 0.1;
194
195 % The initial value of the exponential growth factor nu
196 % This variable is used to increase mu if the error goes up.
197 nu = 2;
198
199 % Now perform the optimisation
200 Searching = 1;
201 Iterations = 0;
202 MaxIterations = 100;
203 while Searching == 1
204
205     Iterations = Iterations + 1;
206
207     if Iterations > MaxIterations
208         error('Number of interations is too high')
209     end
210
211     % 4. Test for convergence - choose a size for the gradient
```

```
212     if norm(Gradient)/ProblemSize < 0.001
213         break;  % Leave the loop
214     end
215 
216     % 3. Solve for the change to parameters
217     dp = -(JTransposeJ + mu*eye(ProblemSize)) \ Gradient;
218 
219     % Step 5.
220     PredictedChange = Gradient'*dp;
221 
222     % 6. Define the new test parameters
223     KMatPerturbed = KMatrix;
224     FrameParametersPerturbed = FrameParameters;
225 
226     KMatPerturbed(1,1) = KMatPerturbed(1,1) + dp(1);
227     KMatPerturbed(1,2) = KMatPerturbed(1,2) + dp(2);
228     KMatPerturbed(1,3) = KMatPerturbed(1,3) + dp(3);
229     KMatPerturbed(2,2) = KMatPerturbed(2,2) + dp(4);
230     KMatPerturbed(2,3) = KMatPerturbed(2,3) + dp(5);
231 
232     % Initialise the error for the latest test
233     NewError = 0;
234     for j = 1:nImages
235 
236         % Perturb the image location
237         StartRow = 6 + (j-1)*6;
238         FrameParametersPerturbed{j,NANGLE} = ...
239             FrameParametersPerturbed{j,NANGLE} + dp(StartRow:StartRow+2);
240 
241         FrameParametersPerturbed{j,NTRANSLATION} = ...
242             FrameParametersPerturbed{j,NTRANSLATION} + ...
243             dp(StartRow+3:StartRow+5);
244 
245         % ... And compute the error vector for this image
246         OptComponents{j,NERRORVECTOR} = ...
247             ComputeImageErrors( KMatPerturbed, ...
248             FrameParametersPerturbed{j,NANGLE},...
249             FrameParametersPerturbed{j,NTRANSLATION},...
250             Data{j,NCORRESPOND}, Data{j,NCONSENSUS});
251 
252         % Compute the error
253         NewError = NewError + 0.5 * ...
254             OptComponents{j,NERRORVECTOR}'*OptComponents{j,NERRORVECTOR};
255 
256     end
257 
258     ChangeInError = NewError - CurrentError;
259     % Step 7.
260     Gain = ChangeInError / PredictedChange;
```

```
261
262     % Step 8.
263     if ChangeInError > 0
264         % Error has gone up.  Increase mu
265         mu = mu*nu;
266         nu = nu*2;
267     else
268         % Error has gone down.  Update mu
269         nu = 2; % Default start value of nu
270         mu = mu * max([1/3,(1-(2*Gain-1)^3)]);
271
272         % Update the parameters
273         KMatrix = KMatPerturbed;
274         FrameParameters = FrameParametersPerturbed;
275
276         % Update the error
277         CurrentError = NewError;
278
279         % The Jacobian is expensive to compute - only recompute
280         % if the gain is low, i.e. the Jacobian is not accurate
281         if Gain < 1/3
282             % The gain is poor - recompute the Jacobian and the Gradient
283             JTransposeJ = zeros(ProblemSize);
284             for j = 1:nImages
285                 % The Jacobian
286                 [OptComponents{j,NKMATJACOB},...
287                     OptComponents{j,NFRAMEJACOB}] = ...
288                     SingleImageJacobian( KMatrix,...
289                     FrameParameters{j,NANGLE}, ...
290                     FrameParameters{j,NTRANSLATION},...
291                     Data{j,NCORRESPOND}, Data{j,NCONSENSUS});
292
293                 % The top 5x5 block is the sum of all the inner products
294                 % of the
295                 % K-matrix Jacobian blocks.
296                 JTransposeJ(1:5,1:5) = JTransposeJ(1:5,1:5)+ ...
297                     OptComponents{j,NKMATJACOB}' * ...
298                     OptComponents{j,NKMATJACOB};
299
300                 % The diagonal image block associated with the frame
301                 % parameters
302                 StartRow = 6 + (j-1)*6;
303                 EndRow = StartRow+5;
304                 JTransposeJ(StartRow:EndRow,StartRow:EndRow) = ...
305                     OptComponents{j,NFRAMEJACOB}' * ...
306                     OptComponents{j,NFRAMEJACOB};
307
308                 JTransposeJ(1:5,StartRow:EndRow) = ...
309                     OptComponents{j,NKMATJACOB}' * ...
```

```
310                     OptComponents{j,NFRAMEJACOB};
311
312                 JTransposeJ(StartRow:EndRow,1:5) =  ...
313                     JTransposeJ(1:5,StartRow:EndRow)';
314             end
315         end
316
317         % Compute the new gradient
318         Gradient = zeros(ProblemSize,1);
319         for j = 1:nImages
320             Gradient(1:5) = Gradient(1:5) + ...
321                 OptComponents{j,NKMATJACOB}'*OptComponents{j,NERRORVECTOR
                    };
322             Gradient(StartRow:EndRow) = OptComponents{j,NFRAMEJACOB}'*...
323                 OptComponents{j,NERRORVECTOR};
324         end
325
326     end
327
328
329 end
330
331 end
```

../Matlab/OptimiseKMatrix/OptimiseKMatrix.m

7.3.5 Some comments on the code structure

I have asked you to write your own Levenberg-Marquardt code. Such a task would unusual in the real world unless there are specific speed requirements - normally one would use an existing package. But in our case the Jacobian as a specific block structure that can be used to speed up computation. The code I wrote is aimed at the task of estimating a K-matrix. You may wish to consider how you might make the code more generic to optimizing general quadratic cost functions. I have also not used the singular value decomposition to compute the update to the parameters (the addition of $\mu\mathbf{I}$ makes this more difficult). But the solution of the over-constrained linear problem is only part of a Newton step that is approximate. So the extra cost involved may not be worth it.

I have also used a simple stopping criterion. Do you think the stopping criterion is appropriate? Why did I divide by the number of images? What is actually being tested here? Can you think of other termination conditions that might be better (how about the infinity norm - and how do you compute it?).

8 Recap and further issues

If you made it to here, then congratulations! You have found a good estimate of a camera provided that lens distortion is not too great. A fully commercial solution would next need to consider how to deal with lens non-linearities and how our initial assumptions underlying our simple pin-hole model break down. However, this is a B1 project and asking you to estimate a real non-linear model would be too much in the context of this project. (Although much of the approach above is close to how it is done in reality.)

All estimation techniques have similar features to those illustrated in the project above:

1. Adopt a simple physical model.
2. Consider how the real system might deviate from the simple model and try and bound these deviations so that your simple model is as linear as possible.
3. Identify constraints that impose structure on the numerical characteristics of your model. In our example we have used a plane that results in a perspective transformation that can be separated from a simple model of a projective camera.
4. Use the geometric or other regularities to identify possible least-squares problems. You may need to make big assumptions about where noise processes impact on the process - but the aim is to generate a seed solution, so a small amount of bias is probably acceptable.
5. Use your knowledge of linear algebra to construct estimates of the structures defined by your constraints. The end result is an initial model.
6. Refine your model by revisiting the errors in your model in the original problem space, where the statistics of the errors are well understood. Use the properties of a minimum variance estimate to optimize your model in this space.

When writing your report consider what you are doing at each stage and write your comments to match the process. Your report should match how the code builds up its model and how problems are reported or overcome by the code. The conclusion should be an assessment of how well the code performs, how it is structured, and

what further stages of development would be required to achieve a fully functional product.