Contents

Chapter 1

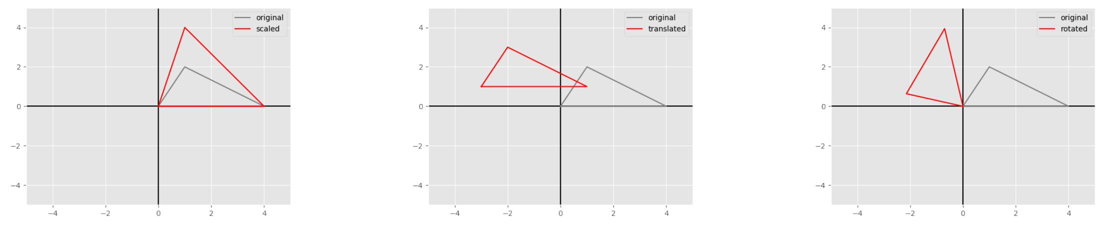
Prerequisites

In this chapter, we will discuss essential building blocks needed for SfM (Structure from Motion), or generally speaking, 3D vision. In particular, we will first discuss basic geometric transformations which are widely used in computer graphics. We will see visual effects on geometric entities (triangle, circle etc), when transformed through some basic geometric transformations. Afterwards, we will assemble transformations in a particular way to form a very basic camera model – a set of geometric principles that governs the process of projecting 3D world points onto 2D image plane. Lastly, we will discuss some miscellaneous topics that will be used in our final SfM pipeline.

1.1 Transformations

1.1.1 2D Transformations

Given a point P having coordinates $(x, y)^T$, some basic geometric transformations are as follows:



(a) Scaled $(s_x = 1, s_y = 2)$ (b) Translation $(t_x = -3, t_y = 1)$ (c) Rotation $(\theta = 100°)$

Figure 1.1: Transformations

1. **Scaling** is used to either shrink or expand an object graphically, as shown in figure 1.1(a). Scaling can be obtained by multiplying the original coordinates x and y with s_x and s_y respectively, as shown in equation below. s_x and s_y determines the degree of expansion (or compression) in x and y dimension respetively. Scaling factor more than 1 expands the objects while scaling factor below 1 shrinks the objects.

$$x' = s_x \times x \qquad y' = s_y \times y \tag{1.1}$$

We can rewrite the above equations in terms of matrix vector product, as shown below:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1.2}$$

2. **Translation** displaces an object to another location, as shown in figure 1.1(b). Transformation can be achieved by adding a translation factor t_x and t_y to x and y respectively, as shown in the equation below. t_x and t_y determines the vector (i.e magnitude and direction) of displacement in x and y coordinates respectively.

$$x' = x + t_x \qquad y' = y + t_y \tag{1.3}$$

We can rewrite the above equations in terms of vector algebra, as shown below:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{1.4}$$

3. **Rotation** rotates an object at particular angle θ from its origin, as shown in figure 1.1(c). Mathematically, the rotated points x' and y' can be obtained using the equation below:

$$x' = xcos\theta - ysin\theta \qquad y' = xsin\theta + ycos\theta \tag{1.5}$$

We can rewrite the above equations in terms of matrix vector product, as shown below:

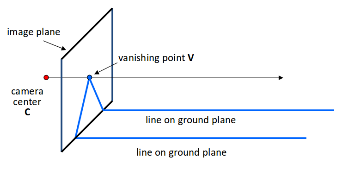
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1.6}$$

Furthermore, the rotation matrix R must satisfy following properties:

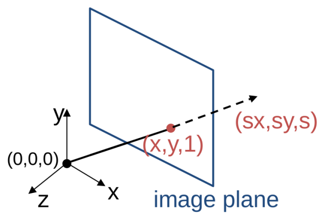
(a) R must be an orthogonal matrix i.e $R^{-1} = R^T$. Graphically, it means that the columns of R must be perpendicular to each other.

(b) det $R = \pm 1$. The rotation matrix R is improper if det $R = -1$, and proper if det $R = +1$. The graphical notion of improper and proper matrices will be explained in section 1.1.3.

1.1.2 Homogeneous Coordinates



(a) Projective Geometry



(b) Homogeneous Coordinates

Figure 1.2: Projective Geometry and Homogeneous Coordinates

Homogeneous Coordinates make calculations of graphics and geometry possible in projective space. To illustrate this, configure two parallel lines in euclidean space (i.e on ground plane), and their projections on projective space (i.e on image plane), as shown in figure 1.2(a). Notice that the parallel lines – which do not meet in euclidean space – do meet in projective space. Theoretically speaking, parallel lines should meet at infinity in projective space, and mustn't meet in euclidean space. Given this scenario, we cannot use cartesian (∞, ∞) to represent point at infinity since it becomes meaningless in euclidean space. To overcome this issue, conventional cartesian coordinates aren't sufficient; we adopt homogeneous coordinates which are explained next.

Given a point $P = (x, y)^T$ in cartesian coordinate system, we can append a dummy coordinate to form $P' = (x, y, 1)^T$ in homogeneous coordinate system. To go back, we can simply discard the last element.

We can generalize this idea further: given a point $P = (x, y)^T$ in cartesian coordinate system, we can use $P' = (sx, sy, s)$ to represent the same point homogeneous coordinates. Again, to go back, we can divide $P' = (sx, sy, s)$ with s to obtain the original point P. Graphically, a point P in cartesian coordinate system corresponds to a ray in homogeneous coordinate system going from origin $O = (0, 0, 0)$ to (sx, sy) on a plane perpendicular to $z - axis$ which is s units away (Figure 1.2(b)).

Now that we know how to form homogenous coordinates, let's see how can we exploit this to represent a point at infinity. Consider a cartesian point $P = (2, 3)^T$, and corresponding homogeneous point $P' = (2, 3, 1)^T$. If the point moves towards infinity in euclidean space, it can be represented as $P' = (2, 3, 0)$ in homogeneous system which gives us $P = (2/0, 3/0) = (\infty, \infty)$ in euclidean space as required. Furthermore, using homogeneous coordinates, we can rewrite the transformations we defined in previous section as follows:

1. **Scaling**

$$k \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{1.7}$$

2. **Translation**

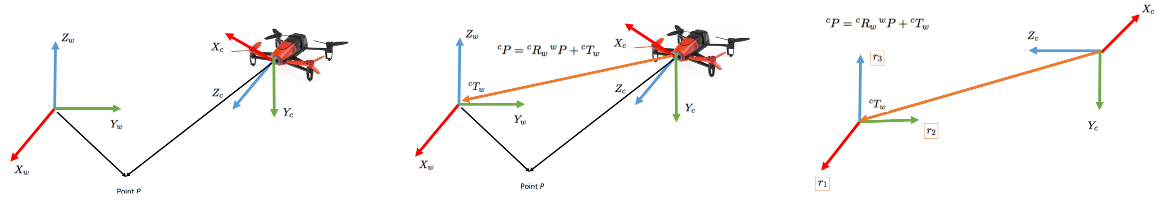
$$k \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{1.8}$$

3. **Rotation**

$$k \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{1.9}$$

Homogeneous coordinates allowed us to rewrite all transformations in matrix vector product. This property will enable us to combine successive transformations in one matrix seamlessly.

1.1.3 Coordinate Transformations



(a) Point P in 2 coordinate systems (b) Geometric Interpretation of t_W^C (c) Geometric Interpretation of R_W^C

Figure 1.3: Multiple Coordinate Systems (X_W, Y_W, Z_W and X_C, Y_C, Z_C represent three axis of world coordinate system and camera coordinate system respectively.)

As we shall see later, we will often need to convert points from one coordinate system to another. Consider a point P in two coordinate systems: a world coordinate frame – often a fixed reference point – and a camera coordinate system – mobile with respect to world coordinate system (Figure 1.3(a)). It is fairly easy to see that one coordinate system can be rotated and/or translated in a particular way to match another coordinate system. So, it follows that a point in one coordinate system (world frame) can be rotated and translated to be represented in another coordinate system (camera frame), as shown in the equation below.

$$P^C = R_W^C P^W + t_W^C \tag{1.10}$$

Where P^C and P^W represent points in camera and world coordinate systems respectively, R_W^C and t_W^C represent rotation and translation transformations respectively from world to camera coordinate system.

Let's now try to understand the geometric meaning of R_W^C and t_W^C. Visualizing t_W^C vector is easier: substituting $P^W = 0$ in equation 1.10, we get:

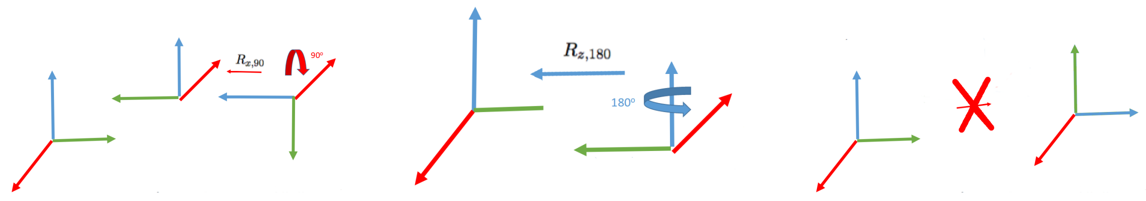
$$P^C = t_W^C \tag{1.11}$$

Since P^C geometrically corresponds to a vector originating from origin of camera coordinate sytem to origin of world coordinate system (Figure 1.3(b)), it follows that t_W^C is the same vector. We can visualize $R_W^C = (r_1, r_2, r_3)$ using the same reasoning: substituting $P^W = (1, 0, 0)^T$ in equation 1.10 and assuming that $t_W^C = 0$, we get:

$$P^C = r_1 \tag{1.12}$$

Since we know that P^C geometrically corresponds to a vector pointing towards world axis in camera coordinate system (Figure 1.3(c)), r_1 is that same vector. Thus, to generalize, rotation columns are the world axis expressed in camera coordinate system. To illustrate further, in the particular case of figure 1.3(c), $r_2 = (0, 0, -1)^T$ since this vector points to 2nd axis of world coordinate system using the notation of camera coordinate system. Using the same reasoning for r_3, we get:

$$R_W^C = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{1.13}$$



(a) Proper Rotation Matrix: Rotating 90 around x-axis

(b) Proper Rotation Matrix: Rotating 180 around z-axis

(c) Improper Rotation Matrix

Figure 1.4: Proper and Improper Rotation Matrices

Having said this, we're now ready to understand the notion of improper and proper matrices. The rotation matrix R_W^C in equation 1.13 is indeed a proper matrix i.e det $R = 1$. However, another rotation matrix $\hat{R}_W^C = -R_W^C$ would have determinant of -1, as you might expect; and thus, it's an improper matrix. Notice that the relative orientation of axes is preserved during proper matrix transformation, unlike improper matrix transformation. This effect is best illustrated through right hand rule: point the forefinger of your right hand in the direction of x-axis, middle finger in the direction of y-axis, your thumb should now point towards the third axis. If you can orient your hand in a way to form new coordinate system, it means that the relative orientation of axes is preserved during transformation (Figure 1.4(a) and (b)), otherwise it's not preserved (1.4(c)).

1.2 Camera Models

Armed with the concept of coordinate transformations, we are now ready to understand camera models. A camera model is responsible for transforming visual 3D world into 2D camera pixels (Figure 1.5). Camera Models can be abstractly divided in two transformations; (1) 3D world in third person view to 3D world in first person view, and (2) 3D world in first person view to 2D pixels.

1.2.1 Extrinsic Parameters: From Third-Person 3D world to First-Person 3D World

Real world 3D points are represented in world coordinate system (i.e third person 3D world). However, we need 3D points to be in camera coordinate system (i.e first person 3D world) to ultimately transform the

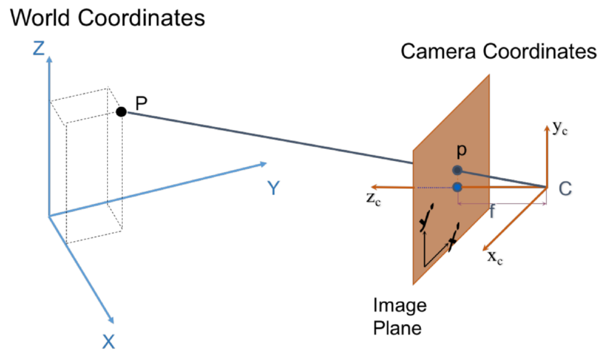
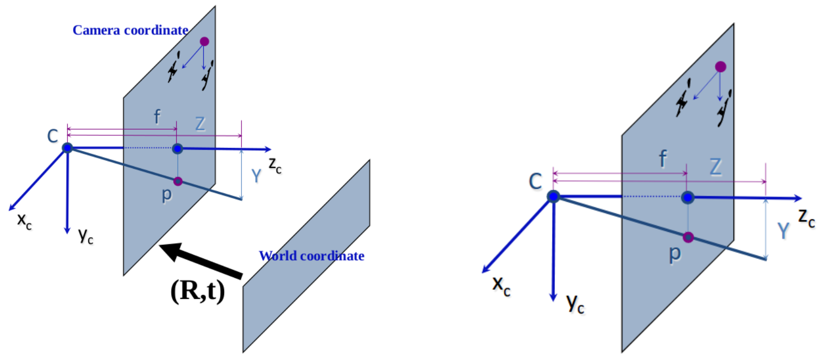


Figure 1.5: Camera Model Transformation



(a) From World Coordinate System to Camera Coordinate System

(b) From Camera Coordinate System to Image Plane Coordinate System

Figure 1.6: Extrinsic and Intrinsic Parameters

points in 2D pixels. Therefore, we require a mapping from world coordinate system to camera coordinate system, as shown in figure 1.6(a). Given a 3D point in world coordinate system P_W, we can compute 3D point in camera coordinate system P_C as:

$$P_C = RP_W + t \tag{1.14}$$

Where R is 3x3 rotation matrix denoting the orientation of camera w.r.t the world, and t is 3x1 vector represents the placement of camera in world coordinate system; these two parameters are collectively called *Exstrinsic Parameters.*

1.2.2 Intrinsic Parameters: From First-Person 3D world to 2D pixels

Now that we have transformed the points into camera coordinate system, we shall transform it once again to obtain the representation in 2D pixels representation. The parameters through which this transformation occurs are known as *Intrinsic Parameters.* Abstractly speaking, camera's intrinsic parameters encompass its internal parameters including focal length, image sensor format, principal point (explained below). Furthermore, constructing camera's intrinsic parameters can be broken down in two following steps:

1. **Mapping 3D Coordinates to Image Plane Coordinates:** Consider a camera having optical centre C and three axes X_C, Y_C, Z_C projecting a 3D point $P_C = (X_C, Y_C, Z_C)$ onto a 2D point $p = (x', y')$ on image plane (which is f units away), as shown in figure 1.6(b). Mathematically, we can represent it as:

$$x' = f\frac{X_C}{Z_C} \qquad \rightarrow \qquad Z_C x' = fX_C \tag{1.15}$$

$$y' = f\frac{Y_C}{Z_C} \qquad \rightarrow \qquad Z_C y' = fY_C \tag{1.16}$$

$$Z_C = Z_C \tag{1.17}$$

The above equations can be encoded in a matrix vector product as follows:

$$Z_C \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} \tag{1.18}$$

2. **Mapping Image Plane Coordinates to Pixel Coordinates:** Although we have projected points in the image plane, the range of transformed values are real numbers. However, pixels indices in a digital image are integers. Furthermore, the origin of digital image coordinates is at the top left corner of the image plane instead of at the center of image plane. Therefore, we need to convert coordinates from image plane coordinate system (x', y') to pixel coordinate system (u_{img}, v_{img}). We can do so using the following transformation:

$$\begin{bmatrix} u_{img} \\ v_{img} \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_x & s & p_x \\ 0 & \alpha_y & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{1.19}$$

Although detailed explanation of the matrix entries is beyond the scope of this text, each of the entries is briefly described as follows:

- **Scaling Factors** α_x, α_y determine the size of pixel width and height, respectively.
- **Principal Point** p_x, p_y represents the point of intersection of image plane with optical axis.
- **Slanting Factor** s represent the skew factor when image plane is not normal to optical axis (often the case in practice).

Finally, we can combine the parameters to form Instrinsic Parameters (also called *Camera Calibration Matrix K*):

$$z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_x & s & p_x \\ 0 & \alpha_y & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} \tag{1.20}$$

$$z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \overbrace{\begin{bmatrix} \alpha_x f & sf & p_x & 0 \\ 0 & \alpha_y f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}^{K} \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} \tag{1.21}$$

1.2.3 Putting it Together

We can assemble extrinsic and intrinsic parameters to construct end-to-end transformation matrix that, given a point P_W in world coordinate system, computes a point p_{img} in pixel coordinate system. Mathematically, it can be composed as follows:

$$p_{img} = \overbrace{K}^{IntrinsicParameters} \times \overbrace{[R|t]}^{ExtrinsicParameters} \times P_W \tag{1.22}$$

$$p_{img} = \overbrace{\underbrace{\begin{bmatrix} \alpha_x & s & p_x \\ 0 & \alpha_y & p_y \\ 0 & 0 & 1 \end{bmatrix}}_{Pixels} \times \underbrace{\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{ImagePlane}}^{IntrinsicParameters} \times \overbrace{\underbrace{[I|t]}_{Translation} \times \underbrace{\begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix}}_{Rotation}}^{ExtrinsicParameters} \times P_W \tag{1.23}$$

1.3 Miscellaneous

1.3.1 Point Matching using Feature Keypoints and Descriptors

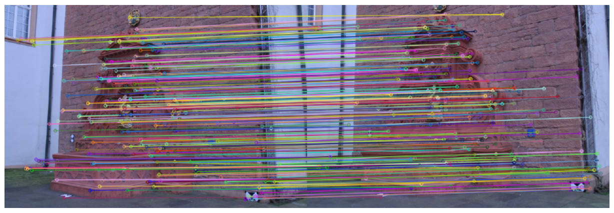


Figure 1.7: 2D Point Correspondences: Two ends of line represent a pair of image points displaying the same 3D point

As we shall see later, we will need to find *2D point correspondences* – a pair of points in two images that refer to the same point in 3D world – as shown in figure 1.7. To do so, we commonly employ image features which contain *feature keypoints* and *feature descriptors*.

- **Feature Keypoints** refer to 2D positions (i.e x-y coordinates) of interesting points in the image that might be helpful in computing similarities between other images.
- **Feature Descriptors** represents a keypoint through a vector containing visual description. It is used to actually compute similarities between image features.

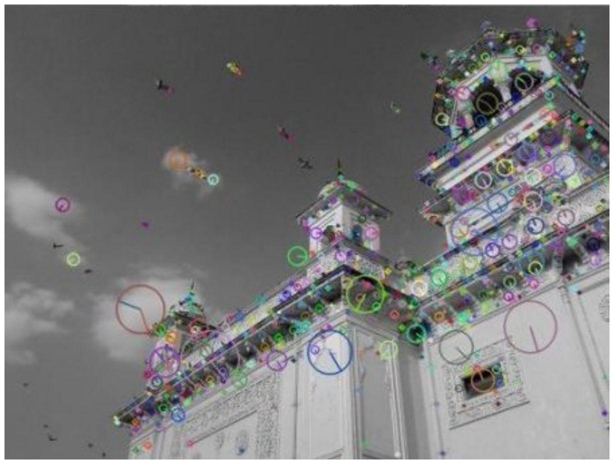


Figure 1.8: SIFT keypoints (Radius and direction of circles represent size and orientation of keypoints)

Two commonly used image features are SIFT (Scale Invariant Feature Transform) and SURF (Speeded Up Robust Features) as shown in figure 1.8; however, their internal workings and methodologies is beyong the scope of this text.

1.3.2 Singular Value Decomposition (SVD) for Least Squares

Often, we need to solve $Ax = 0$, but due to noise in the data, this equation does not hold exactly in practice. Due to which the problem reduces to

$$\underset{x}{min} \, ||Ax|| \tag{1.24}$$

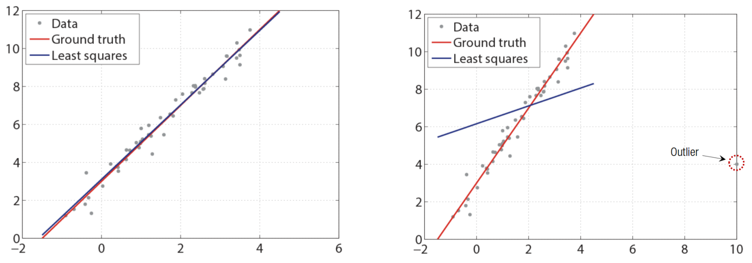
Thus, we try to find, not the exact nullspace of A, but rather a vector x close to nullspace. This vector corresponds to the row-space vector of A which contributes minimally in matrix A construction. Thus

$$x = V(:, end) \tag{1.25}$$

Where

$$A = UDV^T \tag{1.26}$$

1.3.3 Random Sample Consensus (RANSAC)



(a) Outlier Absent (b) Outlier Present

Figure 1.9: Least Squares

As we shall see later, we will often need to estimate parameters from a set of observed data. Since data almost always contains outliers, parameter estimation on full data will generally give us infeasible results. To illustrate this, consider an example of least squares: suppose we have to fit a line through the data. Formally speaking, given N points $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$, the goal is to estimate the line represented by e, f, g such that perpendicular distance E between points and the line is minimized; this can be represented mathematically as mentioned below:

$$E = (ex_1 + fy_1 + g)^2, (ex_2 + fy_2 + g)^2, \ldots, (ex_N + fy_N + g)^2 = \sum_{i}^{N} (ex_i + fy_i + g)^2 \tag{1.27}$$

$$E = \left\| \begin{bmatrix} x_1 & y_1 & 1 \\ .. & .. & .. \\ x_N & y_N & 1 \end{bmatrix} \begin{bmatrix} e \\ f \\ g \end{bmatrix} \right\|^2 = \|Ax\|^2 \tag{1.28}$$

Although, the solution for this equation can be computed very efficiently via SVD (as explained in section 1.3.2), it is extremely sensitive to outliers, as shown in figure 1.9.

To overcome this problem, RANSAC can be used. RANSAC is an iterative process to estimate the parameters of a mathematical model from observed data, when outliers are not supposed to be taken into account. The strategy is to find the model which accords which maximum number of samples. The pseudocode is mentioned in algorithm 1.

Now, the question is, roughly how many iterations are required to find a "good" estimate? It depends on some factors such as desired confidence of solution, number of data samples, and number of inliers. Let's consider the following:

$$\text{Probability of choosing an inlier (w)} = \frac{numberofinliers}{numberofsamples}$$

$$\text{Probability of building a correct model} = w^n$$

Algorithm 1 Random Sample Consensus (RANSAC)

```
1: function RANSAC
2:     while iterations are not complete do
3:         Randomy sample N points
4:         Estimate parameters
5:         Count inliers
6:         Update best parameters
7:     end while
8:     Return best parameters
9: end function
```

$$\text{Probability of not building a correct model after } K \text{ iterations} = (1 - w^n)^K$$

$$(1 - w^n)^K = 1 - p \text{ where p is the desired RANSAC success rate.}$$

In the above equation, we can solve for K, like so:

$$K = \frac{log(1 - p)}{log(1 - w^n)} \tag{1.29}$$