Chapter 9

Least Squares Approximation

Galileo Galilei (1564-1642) said, "The universe cannot be read until we have learned the language and become familiar with the characters in which it is written. It is written in mathematical language, and the letters are triangles, circles and other geometrical figures, without which means it is humanly impossible to comprehend a single word. Without these, one is wandering about in a dark labyrinth." Carl Friedrich Gauss (1777-1855), who is credited with developing the fundamentals of the basis for least squares analysis at the age of eighteen, referred to mathematics as "the Queen of the Sciences." Over a century later, mathematics is still seen throughout the world as an essential tool in many fields, including natural science, engineering, medicine, economics, and social science. In his speech *"10 Lessons of an MIT Education,"* Prof. Gian-Carlo Rota told the MIT Alumni Association that mathematics is still the queen of the sciences, and "MIT is one huge applied mathematics department; you can find applied mathematicians in practically every department at MIT except mathematics."

Unfortunately, not every software developer realizes the importance of mathematics. It is not uncommon for developers to search for answers online when developing algorithms. After reading some articles or postings, they think that they understand the problem and known how to solve it. This often leads to insufficient, if not incorrect, implementations. In this chapter, I will give some examples that may look simple but actually require developers to scratch their heads and dig deep into their memory for mathematics they learned in high schools and colleges. I hope that readers will see how mathematics plays an important role in software development when working on these examples and that they will be motivated to do more math on their own before implementing any algorithms.

In many applications such as linear regression, reverse engineering, image processing, metrology, and computer aided geometric modeling, we often want to find the best fit line, circle, and plane to the given data. Solutions to these problems are commonly studied in *point cloud* computing. A point cloud refers to a collection of points obtained by digitizing devices such as a 3D laser scanner. These 3D points typically represent the external surface of a geometric object. Point clouds are used for many purposes, including feature recognition, reverse engineering, and metrology inspection. For example, a manufactured part may be laser scanned to obtain a collection of points. The point cloud is then aligned and compared with a CAD model to check for differences. These differences can be displayed as color maps that give a visual indicator of the deviation between the manufactured part and the CAD model. Geometric dimensions and tolerances can also be extracted directly from the point cloud.

While point clouds can be rendered and inspected, they are usually not directly usable in most CAD applications. They often need to be converted to triangle meshes or approximated by polynomial curves and surfaces. This process is commonly known as *reverse engineering.* It is beyond the scope of this chapter to discuss how to approximate a point cloud by, for example, spline curves and surfaces. Instead, we shall focus on four fundamental methods: approximating a point cloud by line, circle, arc, and plane. By understanding the techniques described in this chapter, readers should know where to look for more information to solve their own point cloud computation problems.

9.1 Least Squares Line

Given a collection of points, it is often desirable to know whether these points can be represented by a straight line. To achieve this goal, we first find the "best fit" line for the data points and then compute the deviation between the line and the points. If the largest deviation is less than the given tolerance, the point cloud is said to degenerate to a straight line.

A line is uniquely defined by two distinct points. When there are more than two points, the *least squares approximation* is often used to find the best fit line, although other approximation methods are also used. The process of attempting to fit a linear model to observed data is known as *linear regression* in statistics. Searching online for *"least squares"* or *"best fit"* line, one is most likely to find the method described in this section.

A line not perpendicular to the x-axis may be represented either explicitly as $y = kx + c$ or implicitly as $y - kx - c = 0$. Assume this line is used to approximate the given set of points $\boldsymbol{p}_i$ $(i = 1, 2, \cdots, n)$. Then, the squared error with respect to $\boldsymbol{p}_i = (x_i, y_i)$ is $(y_i - kx_i - c)^2$. Accordingly, the sum of squared errors is

$$\mathcal{E} = \sum_{i=1}^{n} (y_i - kx_i - c)^2. \tag{9.1.1}$$

From calculus it is known that $\mathcal{E}$ is minimized if

$$\begin{aligned} \frac{\partial \mathcal{E}}{\partial k} &= 2\sum_{i=0}^{n} (y_i - kx_i - c)x_i = 0 \\ \frac{\partial \mathcal{E}}{\partial c} &= 2\sum_{i=0}^{n} (y_i - kx_i - c) = 0 \end{aligned} \tag{9.1.2}$$

The above equations may alternatively be represented as

$$\begin{aligned} a_{00}k + a_{01}c &= b_0 \\ a_{10}k + a_{11}c &= b_1 \end{aligned} \tag{9.1.3}$$

where,

$$\begin{aligned} a_{00} &= \sum x_i^2, & a_{01} &= \sum x_i, & b_0 &= \sum x_i y_i \\ a_{10} &= \sum x_i, & a_{11} &= n, & b_1 &= \sum y_i \end{aligned}$$

Since it is assumed that the line is not perpendicular to the x-axis, we have

$$\begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix} \neq 0.$$

Therefore,

$$k = \frac{\begin{vmatrix} b_0 & a_{01} \\ b_1 & a_{11} \end{vmatrix}}{\begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix}}, \quad c = \frac{\begin{vmatrix} a_{00} & b_0 \\ a_{10} & b_1 \end{vmatrix}}{\begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix}}.$$

If the line is perpendicular to the x-axis, we may represent it as $x = \bar{k}y + \bar{c}$. Accordingly, we can solve the following linear system of equations to obtain optimized $\bar{k}$ and $\bar{c}$

$$\bar{a}_{00}\bar{k} + \bar{a}_{01}\bar{c} = \bar{b}_0$$

$$\bar{a}_{10}\bar{k} + \bar{a}_{11}\bar{c} = \bar{b}_1$$

where,

$$\begin{aligned} \bar{a}_{00} &= \sum y_i^2, & \bar{a}_{01} &= \sum y_i, & \bar{b}_0 &= \sum x_i y_i \\ \bar{a}_{10} &= \sum y_i, & \bar{a}_{11} &= n, & \bar{b}_1 &= \sum x_i \end{aligned}$$

In actual implementation, it is recommended to compare both determinants

$$\begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix} \quad \text{and} \quad \begin{vmatrix} \bar{a}_{00} & \bar{a}_{01} \\ \bar{a}_{10} & \bar{a}_{11} \end{vmatrix}$$

and choose the method that has the larger absolute value of a determinant to avoid numerical noise. When the given points sit far away from the origin, numerical noise can further be reduced by transforming the points such that the *centroid* (or the mass of center) coincides with the origin.

Referring to equation (9.1.2), $\dfrac{\partial \mathcal{E}}{\partial c} = 0$ indicates

$$\sum y_i - k\sum x_i - \sum c = 0 \quad \Longrightarrow \quad c = \frac{\sum y_i}{n} - k\frac{\sum x_i}{n} = y_c - kx_c$$

which implies that the best fit line passes through the centroid (x_c, y_c). Therefore, $c = 0$ if we translate all points such that the centroid (x_c, y_c) coincides with the origin of the coordinate system. Such translation is done by subtracting (x_c, y_c) from all points. Denoting $\bar{x}_i = x_i - x_c$ and $\bar{y}_i = y_i - y_c$, the error equation (9.1.1) becomes

$$\mathcal{E} = \sum_{i=1}^{n} (\bar{y}_i - k\bar{x}_i)^2.$$

$\mathcal{E}$ is minimized if $\dfrac{d\mathcal{E}}{k} = 0$, which yields

$$\sum_{i=1}^{n} \bar{x}_i\bar{y}_i - k\sum_{i=1}^{n} \bar{x}_i^2 = 0 \quad \Longleftrightarrow \quad k = \frac{\sum_{i=1}^{n} \bar{x}_i\bar{y}_i}{\sum_{i=1}^{n} \bar{x}_i^2}.$$

If a line is represented by $\bar{x} - \bar{k}\bar{y} = 0$, the slope of the best fit line can similarly be computed as

$$\bar{k} = \frac{\sum_{i=1}^{n} \bar{x}_i\bar{y}_i}{\sum_{i=1}^{n} \bar{y}_i^2}.$$

The following implementation chooses the larger denominator for numerical stability:

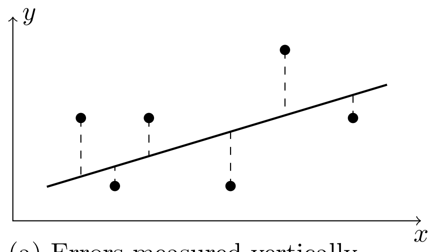
```
/*---------------------------------------------------------------
API Name
   apiBestFitLineV1

Description
   Given a collection of points, it computes a best fit line.
   The least squares error is measured along y-axis.

Signature
   void apiBestFitLineV1(std::vector<GPosition2d> &points,
                         GPosition2d &pt, GVector2d &dirV,
                         apiError &rc)

    INPUT:
      points   collection of points
    OUTPUT:
      pt       point on the best fit line
      dirV     direction vector of the best fit line
      rc       error code: api_OK if not error.

History

   Y.M. Li     03/10/2013 : Creation date
---------------------------------------------------------------*/
#include "MyHeader.h"

void apiBestFitLineV1(std::vector<GPosition2d> &points,
        GPosition2d &pt, GVector2d &dirV, apiError &rc)
{
   int      i, n;
   double   sum_xx, sum_yy, sum_xy, x_i, y_i, k, temp;

   // Compute the center of mass:
   rc = api_OK;
   n = (int)points.size();
   pt.x = pt.y = 0.0;
   for (i=0; i<n; i++)
   {
      pt.x += points[i].x;
      pt.y += points[i].y;
   }
   pt.x /= n;
   pt.y /= n;

   // Transform points and compute summations
   sum_xx = sum_yy = sum_xy = 0.0;
   for (i=0; i<n; i++)
   {
      x_i = points[i].x - pt.x;
      y_i = points[i].y - pt.y;
      sum_xx += x_i * x_i;
      sum_yy += y_i * y_i;
      sum_xy += x_i * y_i;
   }

   if (sum_xx < machine_epsilon && sum_yy < machine_epsilon)
   {
      rc = api_NOSOLUTION;
      goto wrapup;
   }
   if (sum_xx > sum_yy)
   {
      k = sum_xy / sum_xx;
      // Line direction vector is (1/sqrt(1+k^2), k/sqrt(1+k^2):
      temp = 1.0 + k * k;
      dirV.x = 1.0 / sqrt(temp);
      dirV.y = k * dirV.x;
   }
   else
   {
      k = sum_xy / sum_yy;
      // Line direction vector is (k/sqrt(1+k^2), 1/sqrt(1+k^2)):
      temp = 1.0 + k * k;
      dirV.y = 1.0 / sqrt(temp);
      dirV.x = k * dirV.y;
   }

wrapup:

   return;
}
```

In many applications such as computer graphics and computer-aided design, a line is often represented in *vector-valued parametric form* as

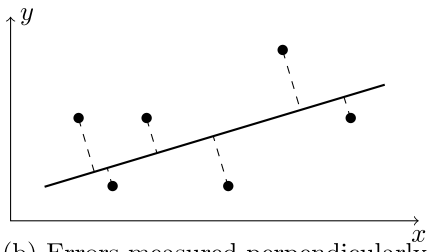
$$\boldsymbol{p}(t) = \boldsymbol{p}_0 + t\boldsymbol{v},$$

where $\boldsymbol{p}_0$ is a point on the line, $\boldsymbol{v}$ the direction vector, and t the arc length parameter. For this reason, the line slope is converted to the direction vector in the above implementation.

The algorithm described above is readily available online, and the implementation of the algorithm is straightforward. Without doing math, many developers may think that they have found the best fit line and hence fail to realize that the obtained least squares line is the "best fit" to data points only in the given coordinate system because the error measurements are done vertically as shown in figure 9.1 (a). If the coordinate system is rotated by some angle, this line may not be the best fit to the data points. Such behavior is known as *non-invariant* under coordinate rotation. In computer-aided geometric modeling and many other applications, it is often desirable to have a property that is *invariant* under coordinate transformations. To find the best fit line that is invariant under coordinate rotations, we need to measure a distance from any point to the line orthogonally as shown in figure 9.1 (b), which is the shortest (or Euclidean) distance.



(a) Errors measured vertically



(b) Errors measured perpendicularly

Figure 9.1: Best fit lines

The best fit line obtained by measuring error orthogonally will be discussed in a later section.

9.2 Least squares circle

Fitting a circle to given points in a plane is a problem that arises in many application areas such as computer graphics, image processing, computer numerical controlled (CNC) machining, metrology, and statistics. The orthogonal least squares circle is a non-linear problem that is often solved by first approximating the model by a linear one to obtain initial estimation and then refining the solution via, for example, the Gauss-Newton algorithm. In this section, we will not discuss the non-linear least squares circle but a quasi-orthogonal "best fit" circle that can be solved via linear least squares approximation.

Writing a circle in a general form gives

$$x^2 + y^2 + 2ax + 2by + c = 0. \tag{9.2.1}$$

By factorizing the above equation we have

$$(x + a)^2 + (y + b)^2 - (a^2 + b^2 - c) = 0.$$

Hence, it is readily seen that the center of the circle is $(-a, -b)$, and the radius is $r = \sqrt{a^2 + b^2 - b}$. If this circle is used to approximate a collection of points, then the shortest distance from $\boldsymbol{p}_i = (x_i, y_i)$ to the circle is

$$\left|\sqrt{(x_i + a)^2 + (y_i + b)^2} - r\right|.$$

Consequently, the squared distance is

$$(x_i + a)^2 + (y_i + b)^2 - 2r\sqrt{(x_i + a)^2 + (y_i + b)^2} + r^2,$$

which leads to solving a system of non-linear equations. It is noted that

$$r = \sqrt{(x_i + a)^2 + (y_i + b)^2}$$

when $\boldsymbol{p}_i$ lies on the circle. To avoid solving a non-linear least squares problem, we may measure the deviation from $\boldsymbol{p}_i$ to the circle as

$$\left|\left(\sqrt{(x_i + a)^2 + (y_i + b)^2}\right)^2 - r^2\right| = \left|x_i^2 + y_i^2 + 2ax_i + 2by_i + c\right|.$$

Accordingly, the sum of squared errors is given by

$$\mathcal{E} = \sum_{i=1}^{n} (x_i^2 + y_i^2 + 2ax_i + 2by_i + c)^2.$$

From calculus, $\mathcal{E}$ is minimized if

$$\frac{\partial \mathcal{E}}{\partial a} = 0, \quad \frac{\partial \mathcal{E}}{\partial b} = 0, \quad \frac{\partial \mathcal{E}}{\partial c} = 0.$$

Explicitly, we need to solve

$$\begin{aligned}
2\sum_{i=1}^{n} x_i^2 a + 2\sum_{i=1}^{n} x_i y_i b + \sum_{i=1}^{n} x_i c + \sum_{i=1}^{n} (x_i^2 + y_i^2) x_i = 0 \\
2\sum_{i=1}^{n} x_i y_i a + 2\sum_{i=1}^{n} y_i^2 b + \sum_{i=1}^{n} y_i c + \sum_{i=1}^{n} (x_i^2 + y_i^2) y_i = 0 \\
2\sum_{i=1}^{n} x_i a + 2\sum_{i=1}^{n} y_i b + nc + \sum_{i=1}^{n} (x_i^2 + y_i^2) = 0
\end{aligned}$$

Let

$$\begin{array}{llll} a_{00} = 2\sum x_i^2, & a_{01} = 2\sum x_i y_i, & a_{02} = \sum x_i, & b_0 = -\sum(x_i^2 + y_i^2)x_i \\ a_{10} = a_{01}, & a_{11} = 2\sum y_i^2, & a_{12} = \sum y_i, & b_1 = -\sum(x_i^2 + y_i^2)y_i \\ a_{20} = 2a_{02}, & a_{21} = 2a_{12}, & a_{22} = n, & b_2 = -\sum(x_i^2 + y_i^2). \end{array}$$

Then, the least squares circle is obtained by solving the following system of linear equations:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}. \tag{9.2.2}$$

The obtained best fit circle is invariant under coordinate rotations. The implementation of least squares circle is listed below.

```
/*-----------------------------------------------------------------
API Name
   apiBestFitCircle

Description
   Given a collection of points, it computes a best fit circle.

Signature
   void apiBestFitCircle(std::vector<GPosition2d> &points,
                         GPosition2d &center, double &radius,
                         apiError &rc)

    INPUT:
      points   collection of points
    OUTPUT:
      center   center of circle
      radius   radius of circle
      rc       error code: api_OK if not error.

History

   Y.M. Li     03/10/2013 : Creation date
-----------------------------------------------------------------*/
#include "MyHeader.h"

void apiBestFitCircle(std::vector<GPosition2d> &points,
      GPosition2d &center, double &radius, apiError &rc)
{
   int      i, n;
   double   x_c, y_c, x_i, y_i, xx, xy, yy;
   double   temp, a, b, c;
   vector<vector<double>> aug_mat;
```

```
// Initialize STL vector
rc = api_OK;
n = (int)points.size();
aug_mat.resize(3);
for (i=0; i<3; i++)
{
   aug_mat[i].resize(4);
}

// Compute centroid
x_c = points[0].x;
y_c = points[0].y;
for (i=1; i<n; i++)
{
   x_c += points[i].x;
   y_c += points[i].y;
}
x_c /= n;
y_c /= n;

// Solve linear least squares approximation
for (i=0; i<n; i++)
{
   x_i = points[i].x - x_c;
   y_i = points[i].y - y_c;
   xx = x_i * x_i;
   xy = x_i * y_i;
   yy = y_i * y_i;

   aug_mat[0][0] += xx;
   aug_mat[0][1] += xy;
   aug_mat[0][2] += x_i;
   aug_mat[1][1] += yy;
   aug_mat[1][2] += y_i;

   temp = xx + yy;
   aug_mat[0][3] -= temp * x_i;
   aug_mat[1][3] -= temp * y_i;
   aug_mat[2][3] -= temp;
}

aug_mat[0][0] *= 2.0;
aug_mat[0][1] *= 2.0;
aug_mat[1][1] *= 2.0;
aug_mat[1][0] = aug_mat[0][1];
aug_mat[2][0] = 2 * aug_mat[0][2];
```

```
   aug_mat[2][1] = 2 * aug_mat[1][2];
   aug_mat[2][2] = n;

   apiGaussPPivot(aug_mat, rc);
   if (rc == api_OK)
   {
      a = aug_mat[0][3];
      b = aug_mat[1][3];
      c = aug_mat[2][3];
      radius = sqrt(a * a + b * b - c);
      center.x = x_c - a;
      center.y = y_c - b;
   }
}
```

When the given points are far away from the origin, it is recommended to transform the points to the local coordinate system whose origin coincides with the center of mass of n points. The data file `BestFitCircle.d1` was obtained by stroking a circle centered at $(0, 0)$ with the radius being 0.5 units. These points were rounded to the third decimal place and then purposely moved far away from the origin by the translation vector $(2 \times 10^5, 10^5)$. Therefore, the best fit circle should be centered at $(2 \times 10^5, 10^5)$ with the radius being roughly 0.5 units with three decimal places. Executing the above program with `BestFitCircle.d1` gives the following expected results:

```
center = (2.000000000012346e+005, 1.000000000000000e+005)
radius = 4.999678963300172e-001
```

But if we did not transform the points such that the centroid coincides with the origin, the least squares circle would be a very poor approximation to the points due to numerical instability.

9.3 Least squares arc with endpoints constraints

Referring to Figure 9.2, a planar plate has been cut by a circular cutter. In computer-aided geometric modeling, such geometric operation is modeled by a Boolean subtraction between a cylinder and the plate. In theory, the intersection curve should be a circular arc. In practice, however, it may have to be represented by a non-circular curve when the cylinder axis is not truly perpendicular to the plate due to numerical noise. In exporting/importing this model for manufacture, it is desirable to represent the cutting path by a circular arc. One possible approach is to stroke the curve to obtain enough sampling points and apply the least squares approximation outlined in the previous section to create the best fit circle. It is then trimmed to obtain the best circular arc (referring to chapter 10 for detail). To have a continuous outer profile (or path) of the plate, the trimmed arc has to interpolate two end points $\boldsymbol{p}_1$ and $\boldsymbol{p}_n$, which is not guaranteed by the method discussed in the previous section. Therefore, a new approximation method needs to be developed to meet the requirement.

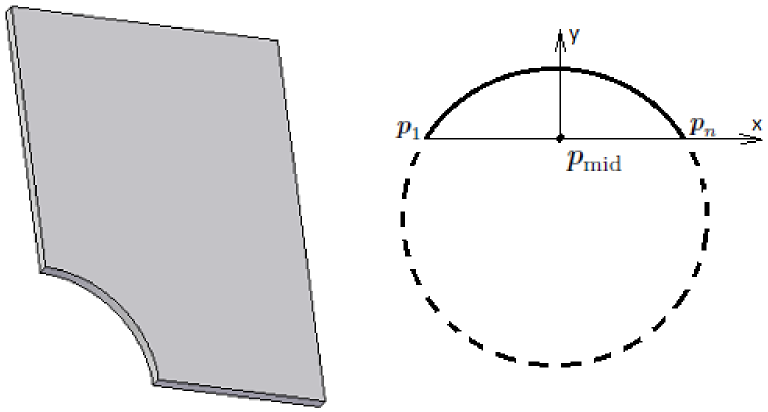


Figure 9.2: Plate cut by circular cutter

Requiring the best fit circle to interpolate two end points $\boldsymbol{p}_1 = (x_1, y_1)$ and $\boldsymbol{p}_n = (x_n, y_n)$ means the circle equation has to meet the following two conditions:

$$x_1^2 + y_1^2 + 2x_1 a + 2y_1 b + c = 0$$
$$x_n^2 + y_n^2 + 2x_n a + 2y_n b + c = 0$$

Recall the fifth question in the preface where candidates were asked what went wrong in deriving the best fit arc. With the added two constraints, we need to eliminate two variables and solve for one. However, the approach given in the preface failed to do so.

In this section, we will discuss how to create a best fit circle that interpolates the endpoints. To simplify the equation and minimize numerical noise, we first compute the mid-point of (x_1, y_1) and (x_n, y_n) and then transform the given points to the new coordinates system whose origin is at the mid-point $\boldsymbol{p}_{\rm mid}$. The transformation is

$$\bar{x}_i = x_i - x_{\rm mid}$$

$$\bar{y}_i = y_i - y_{\rm mid}$$

We then rotate the chord defined by (x_1, y_1) and (x_n, y_n) so that it coincides with the x-axis as shown in Figure 9.2. The rotation angle is determined by

$$\theta = -\arctan\left(\frac{y_n - y_1}{x_n - x_1}\right).$$

Therefore, the rotation transformation is

$$\hat{x}_i = \bar{x}_i \cos\theta - \bar{y}_i \sin\theta$$

$$\hat{y}_i = \bar{x}_i \sin\theta + \bar{y}_i \cos\theta$$

After transformation, the circle would be symmetric to the y-axis and, hence, a in equation (9.2.1) is zero. Accordingly, we have

$$\hat{x}^2 + \hat{y}^2 + 2b\hat{y} + c = 0. \tag{9.3.1}$$

It is also known that $\hat{y}_1$ and $\hat{y}_n$ are zeros in the new coordinate system. By plugging in $(\hat{y}_1, \hat{x}_1)$ (or $(\hat{y}_n, \hat{x}_n)$) into equation (9.3.1), we have $c = -\hat{x}_1^2$ (or $c = -\hat{x}_n^2$) (i.e., the circle interpolates the endpoints). Therefore, our task boils down to find b so that the circle best fits the remaining points in a least squares sense. Let

$$\mathcal{E}(b) = \sum_{i=1}^{n} \left(\hat{x}_i^2 + \hat{y}_i^2 + 2\hat{y}_i b + c\right)^2$$

Minimizing the summation of squared errors is equivalent to solving $\dfrac{d\mathcal{E}}{db} = 0$, i.e.,

$$2\sum_{i=1}^{n} \left(\hat{x}_i^2 + \hat{y}_i^2 + 2\hat{y}_i b + c\right)\hat{y}_i = 0.$$

The above equation is met if

$$b = -\frac{\displaystyle\sum_{i=1}^{n}(\hat{x}_i^2 + \hat{y}_i^2 + c)\hat{y}_i}{\displaystyle 2\sum_{i=1}^{n}\hat{y}_i^2}.$$

Consequently, the center of the circular arc is $(0, -b)$, and the radius is $\sqrt{b^2 - c}$. It should be pointed out that the obtained center of circle is defined in the new coordinate system. Therefore, we need to transform it back to the original coordinate system:

$$x_{\mathrm{c}} = x_{\mathrm{mid}} - b\sin(\theta)$$

$$y_{\mathrm{c}} = y_{\mathrm{mid}} - b\cos(\theta)$$

The implementation of least squares arc is listed below.

```
/*---------------------------------------------------------------
API Name
   apiBestFitArc

Description
   Given a collection of points, it computes a best fit arc
   that interpolates the start and end points.

Signature
   void apiBestFitArc(std::vector<GPosition2d> &points,
                      GPosition2d &center, double &radius,
                      apiError &rc)

    INPUT:
      points   collection of points
    OUTPUT:
      center   center of arc
      radius   radius of arc
      rc       error code: api_OK if not error.

History

   Y.M. Li     03/10/2013 : Creation date
-----------------------------------------------------------------*/
#include "MyHeader.h"

void apiBestFitArc(std::vector<GPosition2d> &points,
                   GPosition2d &center,
                   double &radius, apiError &rc)
{
   int      i, n;
   double   x_i, y_i, x_mid, y_mid, dx, dy;
   double   theta, bid1, bid2, sin_theta, cos_theta, b, c;
   vector<vector<double>> aug_mat;

   // Check if two endpoints coincide
   rc = api_OK;
   n = (int)points.size();
   dx = points[n-1].x - points[0].x;
   dy = points[n-1].y - points[0].y;
   bid1 = sqrt(dx * dx + dy * dy);
   if (bid1 < 1,0e-6)
   {
      rc = api_INVALIDARG;
      goto wrapup;
   }

   // Compute transformation
   x_mid = 0.5 * (points[0].x + points[n-1].x);
   y_mid = 0.5 * (points[0].y + points[n-1].y);
   theta = -atan2(points[n-1].y-points[0].y,points[n-1].x-points[0].x);
   sin_theta = sin(theta);
   cos_theta = cos(theta);
   bid1 = bid2 = 0.0;
   for (i=0; i<n; i++)
   {
      dx =  points[i].x - x_mid;
      dy =  points[i].y - y_mid;
      x_i = dx * cos_theta - dy * sin_theta;
      y_i = dx * sin_theta + dy * cos_theta;
      if (0 == i)
         c = -x_i * x_i;
      bid1 += (x_i * x_i + y_i * y_i + c) * y_i;
      bid2 += y_i * y_i;
   }

   if (bid2 < zero_tol)
   {
      // It may happen if all points are coincident
      rc = api_INVALIDARG;
      goto wrapup;
   }

   b = -bid1 / (2.0 * bid2);
   radius = sqrt(fabs(b * b - c));

   // Transform the center back to original coordinate system
   center.x = x_mid - b * sin_theta;
   center.y = y_mid - b * cos_theta;

wrapup:

   return;
}
```

9.4 Least squares approximation

In this section, we generalize the least squares approximation and derive the so-called *normal equations* to solve least squares approximation problems. Let's consider the least squares line again. Rewriting the line equation $y = kx + b$ in matrix form gives

$$\begin{bmatrix} x & 1 \end{bmatrix} \begin{bmatrix} k \\ b \end{bmatrix} = (y).$$

Given n points $\boldsymbol{p}_i = (x_i, y_i)$, we have n equations for two unknowns k and b

$$\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix} \begin{bmatrix} k \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix},$$

which is over-determined. In general, a linear system $A\boldsymbol{x} = \boldsymbol{b}$ is said to be *over-determined* if A is a matrix of $n \times m$, where $n > m$. In this case, we usually do not have an exact solution for $\boldsymbol{x}$. The best we can do is to minimize the *residual* or error $\boldsymbol{E} = A\boldsymbol{x} - \boldsymbol{b}$. When $\|\boldsymbol{E}\|$ is minimized, the solution $\widehat{\boldsymbol{x}}$ is a least squares solution of $A\boldsymbol{x} = \boldsymbol{b}$.

To find $\widehat{\boldsymbol{x}}$, we minimize the norm of residual squared

$$\|\boldsymbol{E}\|^2 = \|A\boldsymbol{x} - \boldsymbol{b}\|^2 = \boldsymbol{x}^T A^T A\boldsymbol{x} - 2\boldsymbol{b}^T A\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{b}$$

where A^T, $\boldsymbol{x}^T$ and $\boldsymbol{b}^T$ are transpose of A, $\boldsymbol{x}$, and $\boldsymbol{b}$ respectively. The norm of squared residual is minimized by setting the gradient with respect to $\boldsymbol{x}$ to zero:

$$\nabla\|\boldsymbol{E}\|^2 = 2A^TA\boldsymbol{x} - 2A^T\boldsymbol{b} = 0$$

which gives us the *normal equations* $A^TA\boldsymbol{x} = A^T\boldsymbol{b}$. In other words, the least squares solution for $\widehat{\boldsymbol{x}}$ can be obtained by simply solving a system of linear equations (i.e., the normal equations).

We will now look at how to find the least squares line via solving normal equations. Writing a line $kx + c = y$ in matrix form gives

$$\begin{bmatrix} x & 1 \end{bmatrix} \begin{bmatrix} k \\ c \end{bmatrix} = y.$$

Given n points, we have

$$A = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} k \\ b \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

Thus,

$$A^TA = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ 1 & 1 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix} = \begin{bmatrix} \sum x_i^2 & \sum x_i \\ \sum x_i & n \end{bmatrix}$$

$$A^T\boldsymbol{b} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ 1 & 1 & \cdots & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \sum x_iy_i \\ \sum y_i \end{bmatrix}$$

which matches (9.1.3). As another example, we write circle equation (9.2.1) in the matrix form:

$$\begin{bmatrix} 2x & 2y & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = -(x^2 + y^2).$$

Given n points, we have

$$A^TA = \begin{bmatrix} 2x_1 & 2x_2 & \cdots & 2x_n \\ 2y_1 & 2y_2 & \cdots & 2y_n \\ 1 & 1 & \cdots & 1 \end{bmatrix} \begin{bmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \vdots & \vdots & \vdots \\ 2x_n & 2y_n & 1 \end{bmatrix} = \begin{bmatrix} 4\sum x_i^2 & 4\sum x_iy_i & 2\sum x_i \\ 4\sum x_iy_i & 4\sum y_i^2 & 2\sum y_i \\ 2\sum x_i & 2\sum y_i & n \end{bmatrix}$$

$$A^T\boldsymbol{b} = -\begin{bmatrix} 2x_1 & 2x_2 & \cdots & 2x_n \\ 2y_1 & 2y_2 & \cdots & 2y_n \\ 1 & 1 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_n^2 + y_n^2 \end{bmatrix} = -\begin{bmatrix} 2\sum(x_i^2 + y_i^2)x_i \\ 2\sum(x_i^2 + y_i^2)y_i \\ \sum(x_i^2 + y_i^2) \end{bmatrix}$$

which matches (9.2.2) after cancelling the factor 2 in the first and second row. The approach we just discussed is the most direct way of solving a linear least squares problem, as it avoids direct computation of partial derivatives and tedious algebraic manipulations. This is especially desirable in consideration that it is not uncommon to have tens of thousands of data points and a few hundred variables. For the case of best fit line, A is simple so that A^TA and $A^T\boldsymbol{b}$ are readily derived. When the dimension of matrix A becomes high, A^TA and $A^T\boldsymbol{b}$ are usually computed via a matrix multiplication program (e.g., `apiAtAMatrix`).

As will be seen in subsequent sections, we may encounter the case in which $A^TA\boldsymbol{x} = \boldsymbol{0}$. Obviously, the minimum is archived when $\boldsymbol{x} = \boldsymbol{0}$, but this is uninteresting. Therefore, we repose the problem by constraining the solution so it only considers vectors $\boldsymbol{x}$ of some fixed length, e.g., $\|\boldsymbol{x}\| = 1$. We can solve this constrained least squares problem using the Lagrange multipliers by finding the $\boldsymbol{x}$ that minimizes

$$A^TA\boldsymbol{x} + \lambda\boldsymbol{x} = 0$$

which says that $\boldsymbol{x}$ is an eigenvector of A^TA. Since A^TA is symmetric, all eigenvalues are real. If A^TA is also positive definite, then all eigenvalues are real and positive. A matrix B is *positive definite* if $\boldsymbol{x}^TB\boldsymbol{x} > 0$ for all non-zero column vector $\boldsymbol{x}$. It is noted that

$$\boldsymbol{x}^T(A^TA)\boldsymbol{x} = (\boldsymbol{x}^TA^T)(A\boldsymbol{x}) = (A\boldsymbol{x})^T(A\boldsymbol{x}).$$

Since $A\boldsymbol{x}$ gives a transformed vector, $(A\boldsymbol{x})^T(A\boldsymbol{x})$ is equivalent to the dot product of the same vector and, hence, $(A\boldsymbol{x})^T(A\boldsymbol{x}) > 0$. Accordingly, A^TA is positive definite, and all the eigenvalues of A^TA are positive, and the eigenvector with smallest eigenvalue is thus the desired solution of least squares approximation.

9.5 Orthogonal least squares line

The least squares line obtained in section one is the "best fit" line to data only in the given coordinate system because the error measurements are done along the y-axis. If the coordinate system is rotated by some angle, this line may no longer be the best fit to the data points. In this section, we will discuss how to obtain the least squares line that is invariant under coordinate transformation.

A line in the general (or standard) form is written as $\bar{a}x + \bar{b}y - \bar{c} = 0$, where $\bar{a}$ and $\bar{b}$ are not both zero. The shortest distance from any point $\boldsymbol{p}_i = (x_i, y_i)$ to the line is

$$d = \frac{\bar{a}x_i + \bar{b}y_i - \bar{c}}{\sqrt{\bar{a}^2 + \bar{b}^2}}.$$

Since $\sqrt{\bar{a}^2 + \bar{b}^2} \neq 0$, we can divide $\bar{a}$, $\bar{b}$, and $\bar{c}$ by $\sqrt{\bar{a}^2 + \bar{b}^2}$ to obtain the *normalized line*

$$ax + by - c = 0, \tag{9.5.1}$$

where $\sqrt{a^2 + b^2} = 1$ and a and b are the components of a unit vector that is perpendicular to the line. Accordingly, the shortest distance from $\boldsymbol{p}_i$ to the normalized line is

$d_i = ax_i + by_i - c$. If all points are transformed such that the center of mass coincides with the origin, the least squares line passes through the origin and hence $c = 0$. In this case, the above normalized line reduces to $ax + by = 0$ or in matrix form

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = 0.$$

Given n points, we have

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = 0$$

which is the form $A^T A\boldsymbol{x} = \boldsymbol{0}$ that can only be solved via computation of eigenvalues and eigenvectors. Since A is very simple, we compute $A^T A$ manually here:

$$A^T A = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix} = \begin{bmatrix} \sum \bar{x}_i^2 & \sum \bar{x}_i\bar{y}_i \\ \sum \bar{x}_i\bar{y}_i & \sum \bar{y}_i^2 \end{bmatrix}.$$

The above 2×2 matrix has two eigenvalues λ_1 and λ_2 and two corresponding eigenvectors $\boldsymbol{n}_1 = (a_1, b_1)$ and $\boldsymbol{n}_2 = (a_2, b_2)$. As was discussed in the previous section, the eigenvector associated with the smaller eigenvalue defines the best fit line. Assume that $\lambda_1 > \lambda_2$. Then, $\boldsymbol{n}_1$ and $\boldsymbol{n}_2$ define the normal direction of the worst and best fit line respectively. Since $\boldsymbol{n}_1$ and $\boldsymbol{n}_2$ are mutually orthogonal, $\boldsymbol{n}_1$ is the normal direction of the worst fit line but the tangent direction of the best fit line. Therefore, the tangent direction of the best fit line is the eigenvector associated with the larger eigenvalue that can be found via `apiEigenByPower` developed in chapter 6.

The best fit line obtained via orthogonal least squares approximation is invariant under any coordinate rotation. In other words, this line best fits the data in both the least-squares and geometric sense. The implementation to compute the orthogonal least squares line is listed below:

```
/*-------------------------------------------------------------
API Name
   apiBestFitLine

Description
   Given a collection of points, it computes a best fit line.
   The least squares error is measured orthogonal to the line.

Signature
   void apiBestFitLine(std::vector<GPosition2d> &points,
                       GPosition2d &pt, GVector2d &dirV,
                       apiError &rc)
```

```
    INPUT:
      points   collection of points
    OUTPUT:
      pt       point on the best fit line
      dirV     direction vector of the best fit line
      rc       error code: api_OK if not error.

History

   Y.M. Li     03/10/2013 : Creation date
-----------------------------------------------------------------*/
#include "MyHeader.h"

void apiBestFitLine(std::vector<GPosition2d> &points,
      GPosition2d &pt, GVector2d &dirV, apiError &rc)
{
   int            i, n;
   double         x, y, eigenValue;
   vector<double> eigenVector;
   vector<vector<double>> Amat;

   // Initialize memory
   rc = api_OK;
   n = (int)points.size();
   Amat.resize(2);
   for (i=0; i<2; i++)
      Amat[i].resize(2);
   eigenVector.resize(2);

   pt.x = pt.y = 0.0;
   for (i=0; i<n; i++)
   {
      pt.x += points[i].x;
      pt.y += points[i].y;
   }
   pt.x /= n;
   pt.y /= n;

   for (i=0; i<n; i++)
   {
      x = points[i].x - pt.x;
      y = points[i].y - pt.y;
      Amat[0][0] += x * x;
      Amat[0][1] += x * y;
      Amat[1][1] += y * y;
   }
   Amat[1][0] = Amat[0][1];
   eigenVector[0] = 1.0;
   apiEigenByPower(Amat, machine_epsilon, eigenValue, eigenVector, rc);
   if (rc != api_OK)
      goto wrapup;

   dirV.x = eigenVector[0];
   dirV.y = eigenVector[1];

wrapup:

   return;
}
```

Again, the line is represented by the centroid point `pt` and tangent direction vector `dirV`.

It is simpler to solve a linear least squares problem via its normal equation because a direct computation of partial derivatives is avoided. Otherwise, we have to write the sum of squared distance as

$$\mathcal{E} = \sum_{i=1}^{n} (ax_i + by_i)^2$$

and solve the following equations:

$$\frac{\partial \mathcal{E}}{\partial a} = 2\sum_{i=0}^{n} (ax_i + by_i)x_i = 0$$
$$\frac{\partial \mathcal{E}}{\partial b} = 2\sum_{i=0}^{n} (ax_i + by_i)y_i = 0$$

By some algebraic manipulations, we can write the above two equation in matrix form as

$$\begin{bmatrix} \sum \bar{x}_i^2 & \sum \bar{x}_i\bar{y}_i \\ \sum \bar{x}_i\bar{y}_i & \sum \bar{y}_i^2 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} g_{00} & g_{01} \\ g_{10} & g_{11} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = 0.$$

The above system has a trivial solution $a = b = 0$. For non-trivial solutions we need to find λ such that

$$\begin{bmatrix} g_{00} & g_{01} \\ g_{10} & g_{11} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \lambda \begin{bmatrix} a \\ b \end{bmatrix},$$

which is an eigenvalue problem.

9.6 Least squares plane

Similar to the least squares line, we may write a plane as $z = \bar{a}x + \bar{b}y + \bar{c}$ and minimize

$$\mathcal{E} = \sum_{i=1}^{n} (\bar{a}x_i + \bar{b}y_i + \bar{c} - z_i)^2$$

to obtain a simple least squares plane. Since the errors are measured along the direction of z-axis, this "best fit" plane is not invariant under coordinate rotation. In this section, we are concerned only with the orthogonal least squares plane.

A plane in 3D space is defined with *normal vector* $\boldsymbol{n}$ (a vector that is perpendicular to the plane) and an arbitrary point on the plane. For our purpose, we describe a plane as an equation in the form $\bar{a}x + \bar{b}y + \bar{c}z + \bar{d} = 0$. If $\bar{a}$, $\bar{b}$, $\bar{c}$, and $\bar{d}$ are divided by $\sqrt{\bar{a}^2 + \bar{b}^2 + \bar{c}^2}$ and denoted respectively by a, b, c, and d, we obtain the normalized plane

$$ax + by + cz + d = 0$$

where a, b, and c furnish the components of plane normal. Since $\sqrt{a^2 + b^2 + c^2} = 1$, the plane normal $\boldsymbol{n} = (a, b, c)$ is a unit vector. The shortest distance from an arbitrary point $\boldsymbol{p}_i = (x_i, y_i, z_i)$ to the normalized plane is

$$d_i = |ax_i + by_i + cz_i + d|.$$

A plane is uniquely defined by three non-collinear points. If more than three points are given, the approximation process is sought to find the best fit plane. Before proceeding, we want to show that the best fit plane passes through the center of mass. Let

$$\mathcal{E} = \sum_{i=1}^{n} (ax_i + by_i + cz_i + d)^2.$$

From multi-variable calculus, it is known that the following condition has to be met for the best fit plane

$$\frac{\partial \mathcal{E}}{\partial d} = 0,$$

which is equivalent to

$$a\sum_{i=1}^{n} x_i + b\sum_{i=1}^{n} y_i + c\sum_{i=1}^{n} z_i + \sum_{i=1}^{n} d = 0.$$

Therefore, the best fit plane passes through the center of mass (x_c, y_c, z_c) because

$$-d = a\frac{\sum x_i}{n} + b\frac{\sum y_i}{n} + c\frac{\sum z_i}{n} = ax_c + by_c + cz_c.$$

Transforming all given points such that the center of mass coincides with the origin reduces the normalized plane to $ax + by + cz = 0$ or, in matrix form,

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = 0.$$

Accordingly, the normal equations are

$$(A^T A)\boldsymbol{n} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ z_1 & z_2 & \cdots & z_n \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \boldsymbol{0}$$

With some algebraic computations, we have

$$(A^T A)\boldsymbol{n} = \begin{bmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ g_{20} & g_{21} & g_{22} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \boldsymbol{0}$$

where

$$\begin{aligned} g_{00} &= \sum_{i=1}^{n} \bar{x}_i^2, & g_{01} &= \sum_{i=1}^{n} \bar{x}_i \bar{y}_i, & g_{02} &= \sum_{i=1}^{n} \bar{x}_i \bar{z}_i, \\ g_{10} &= g_{01}, & g_{11} &= \sum_{i=1}^{n} \bar{y}_i^2, & g_{12} &= \sum_{i=1}^{n} \bar{y}_i \bar{z}_i, \\ g_{20} &= g_{02}, & g_{21} &= g_{12}, & g_{22} &= \sum_{i=1}^{n} \bar{z}_i^2. \end{aligned}$$

This is again an eigenvalue problem. $A^T A$ is symmetric and positive definite, so it has three non-negative eigenvalues that define three eigenvectors $\boldsymbol{n}_k = (a_k, b_k, c_k)$ $(k = 1, 2, 3)$ that are mutually orthogonal. The eigenvector defined by the smallest eigenvalue is the normal of the best fitting plane. Recall that `apiEigenByPower` is designed to find the largest eigenvalue. It is readily proven that the smallest eigenvalue and corresponding eigenvector can be obtained by applying the power method to the inverse matrix G^{-1}. The implementation is given below:

```
/*----------------------------------------------------------------
API Name
   apiBestFitPlane

Description
   Given a collection of points, it computes a best fit plane.
   The least squares error is measured orthogonal to the plane.

Signature
   void apiBestFitPlane(std::vector<GPosition> &points,
           GPosition &pt, GVector &dirV, apiError &rc)

    INPUT:
      points   collection of points
    OUTPUT:
      pt       a point on the best fit plane
      dirV     normal direction of the plane
      rc       error code: api_OK if not error.

History

   Y.M. Li     03/10/2013 : Creation date
-----------------------------------------------------------------*/
#include "MyHeader.h"
```

```
void apiBestFitPlane(std::vector<GPosition> &points,
        GPosition &pt, GVector &dirV, apiError &rc)
{
   int            i, j, n;
   double         x, y, z, eigenValue;
   vector<double> eigenVector;
   vector<vector<double>> Amat;
   vector<vector<double>> aug_mat;

   // Initialize memory
   rc = api_OK;
   n = (int)points.size();
   Amat.resize(3);
   for (i=0; i<3; i++)
      Amat[i].resize(3);
   eigenVector.resize(3);

   // Compute center of mass
   pt.x = pt.y = pt.z = 0.0;
   for (i=0; i<n; i++)
   {
      pt.x += points[i].x;
      pt.y += points[i].y;
      pt.z += points[i].z;
   }
   pt.x /= n;
   pt.y /= n;
   pt.z /= n;

   for (i=0; i<n; i++)
   {
      x = points[i].x - pt.x;
      y = points[i].y - pt.y;
      z = points[i].z - pt.z;
      Amat[0][0] += x * x;
      Amat[0][1] += x * y;
      Amat[0][2] += x * z;
      Amat[1][1] += y * y;
      Amat[1][2] += y * z;
      Amat[2][2] += z * z;
   }
   Amat[1][0] = Amat[0][1];
   Amat[2][0] = Amat[0][2];
   Amat[2][1] = Amat[1][2];

   // Compute the inverse matrix
   aug_mat.resize(3);
   for (i=0; i<3; i++)
   {
      aug_mat[i].resize(6);
      for (j=0; j<3; j++)
      {
         aug_mat[i][j] = Amat[i][j];
      }
      aug_mat[i][3+i] = 1.0;
   }
   apiGaussJordan(aug_mat, rc);
   if (rc != api_OK)
      goto wrapup;
   for (i=0; i<3; i++)
   {
      for (j=0; j<3; j++)
         Amat[i][j] = aug_mat[i][3+j];
   }

   // Get the eigenvalue and eigenvector
   eigenVector[2] = 1.0;
   apiEigenByPower(Amat, machine_epsilon, eigenValue, eigenVector, rc);
   if (rc != api_OK)
      goto wrapup;

   dirV.x = eigenVector[0];
   dirV.y = eigenVector[1];
   dirV.z = eigenVector[2];

wrapup:

   return;
}
```

If an API is available to compute all three eigenvalues and corresponding eigenvectors, then the given 3D points are planar, linear, or coincident if one, two, or all of the three eigenvalues are zeros respectively. If all three eigenvalues are non-zero and equal, then the given points are symmetric (e.g., points from a sphere surface).

9.7 Summary

In the age of the internet, one can virtually find any information he or she wants. I have seen numerous cases in which developers search online for algorithms and implement them without a thorough grasp of the underlying math. As a result, their implementations are either insufficient or wrong. To draw readers' attention to this common problem, we presented the least squares line, circle, and plane to illustrate why a "simple and straightforward" application may not have such simple math behind.

Without doing the math, developers may simply implement the posted online algorithms without knowing the obtained "best fit" line and plane are coordinate system dependent. If their intention is to obtain the least squares line and plane that is invariant under coordinate transformations, they should actually look for the orthogonal or *total* least squares approximation that minimizes the sum of Euclidian geometric distance. In this case, they will find that the underlying math is then not so simple, and the information about it is not readily available online.

The orthogonal least squares line and plane are obtained via solving the eigenvalues and eigenvectors problems. The API `apiPowerMethod` is designed to find the dominant eigenvalue and associated eigenvector. In real world applications, the QR decomposition algorithm is often used to find all eigenvalues and eigenvectors.

By understanding the least squares line, circle, and plane intuitively, we expanded the technique to a generic linear least squares approximation and derived the normal equations. It is the most direct way of solving a linear least squares problem, as it avoids direct computation of partial derivatives and tedious algebraic manipulations. It should be pointed out that normal equations and QR decomposition only work for full-ranked matrices. If a matrix is rank-deficient, *singular value decomposition* (SVD) is usually used to solve least squares approximation problems, which is considered to be numerically more stable but computationally more expensive algorithm. A simple example to get a singular or rank-deficient matrix is to sample points from a line that passes through the origin and to use these points to compute the least squares line. In this case, all equations are linearly dependent, which results in a rank-deficiency matrix.

The linear least squares approximation is widely used in many fields. The technique and implementation skill learned in this chapter lay a good foundation for readers to solve their own application problems.