Least Squares Fitting of Data by Linear or Quadratic Structures

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: July 15, 1999
Last Modified: September 7, 2021

Contents

1 Introduction

This document describes least-squares minimization algorithms for fitting point sets by linear structures or quadratic structures. The organization is somewhat different from that of the previous version of the document. Modifications include the following.

- A section on the general formulation for nonlinear least-squares fitting is now available. The standard approach is to estimate parameters using numerical minimizers (Gauss–Newton or Levenberg–Marquardt).
- A new algorithm for fitting points by a circle, sphere or hypersphere is provided. The algorithm is non-iterative, so the computation time is bounded and small.
- In the previous version, the sections about fitting of points by ellipses or ellipsoids were severely lacking details and not useful for developing algorithms. Several algorithms are now provided for such fitting, including a general approach for fitting points by hyperellipsoids.
- The document for fitting points by a cylinder has been moved to this document. The website hyperlink to the cylinder document has been redirected to this document.
- A section has been added for fitting points by a single-sided cone.
- Pseudocode is now provided for each of the algorithms. Hyperlinks still exist for those algorithms implemented in the GTE source code.

Other documents using least-squares algorithms for fitting points with curve or surface structures are available at the website. The document for fitting points with a torus is new to the website (as of August 2018).

- Least-Squares Fitting of Data with Polynomials
- Least-Squares Fitting of Data with B-Spline Curves
- Least-Squares Reduction of B-Spline Curves
- Fitting 3D Data with a Helix
- Least-Squares Fitting of Data with B-Spline Surfaces
- Fitting 3D Data with a Torus

The document Least-Squares Fitting of Segments by Line or Plane describes a least-squares algorithm where the input is a set of line segments rather than a set of points. The output is a line (segments in n dimensions) or a plane (segments in 3 dimensions) or a hyperplane (segments in n dimensions).

2 The General Formulation for Nonlinear Least-Squares Fitting

Let $\boldsymbol{F}(\boldsymbol{p}) = (F_0(\boldsymbol{p}), F_1(\boldsymbol{p}), \ldots, F_{n-1}(\boldsymbol{p}))$ be a vector-valued function of the parameters $\boldsymbol{p} = (p_0, p_1, ..., p_{m-1})$. The nonlinear least-squares problem is to minimize the real-valued error function $E(\boldsymbol{p}) = |\boldsymbol{F}(\boldsymbol{p})|^2$.

Let $J = d\boldsymbol{F}/d\boldsymbol{p} = [dF_r/dp_c]$ denote the Jacobian matrix, which is the matrix of first-order partial derivatives of the components of $\boldsymbol{F}$. The matrix has n rows and m columns, and the indexing (r, c) refers to row r and column c. A first-order approximation is

$$\boldsymbol{F}(\boldsymbol{p} + \boldsymbol{d}) \doteq \boldsymbol{F}(\boldsymbol{p}) + J(\boldsymbol{p})\boldsymbol{d} \tag{1}$$

where $\boldsymbol{d}$ is an $m \times 1$ vector with small length. Consequently, an approximation to the error function is

$$E(\boldsymbol{p} + \boldsymbol{d}) = |\boldsymbol{F}(\boldsymbol{p} + \boldsymbol{d})|^2 = |\boldsymbol{F}(\boldsymbol{p}) + J(\boldsymbol{p})\boldsymbol{d}|^2 \tag{2}$$

The goal is to choose $\boldsymbol{d}$ to minimize $|\boldsymbol{F}(\boldsymbol{p}) + J(\boldsymbol{p})\boldsymbol{d}|^2$ and, hopefully, with $E(\boldsymbol{p} + \boldsymbol{d}) < E(\boldsymbol{p})$. Choosing an initial $\boldsymbol{p}_0$, the hope is that the algorithm generates a sequence $\boldsymbol{p}_i$ for which $E(\boldsymbol{p}_{i+1}) < E(\boldsymbol{p}_i)$ and, in the limit, $E(\boldsymbol{p}_j)$ approaches the global minimum of E. The algorithm is referred to as Gauss–Newton iteration.

For a single Gauss–Newton iteration, we need to choose $\boldsymbol{d}$ to minimize $|\boldsymbol{F}(\boldsymbol{p}) + J(\boldsymbol{p})\boldsymbol{d}|^2$ where $\boldsymbol{p}$ is fixed. This is a linear least-squares problem which can be formulated using the normal equations

$$J^{\mathsf{T}}(\boldsymbol{p})J(\boldsymbol{p})\boldsymbol{d} = -J^{\mathsf{T}}(p)\boldsymbol{F}(\boldsymbol{p}) \tag{3}$$

The matrix $J^{\mathsf{T}}J$ is positive semidefinite. If it is invertible, then

$$\boldsymbol{d} = -(J^{\mathsf{T}}(\boldsymbol{p})J(\boldsymbol{p}))^{-1}\boldsymbol{F}(\boldsymbol{p}) \tag{4}$$

If it is not invertible, some other algorithm must be used to choose $\boldsymbol{d}$; one option is to use gradient descent for the step. A Cholesky decomposition can be used to solve the linear system.

During Gauss–Newton iteration, if E does not decrease for a step of the algorithm, one can modify the algorithm to Levenberg–Marquardt iteration. The idea is to smooth the linear system to

$$\left(J^{\mathsf{T}}(\boldsymbol{p})J(\boldsymbol{p}) + \lambda I\right)\boldsymbol{d} = -J^{\mathsf{T}}(p)\boldsymbol{F}(\boldsymbol{p}) \tag{5}$$

where I is the identity matrix of appropriate size and $\lambda > 0$ is the smoothing factor. The strategy for choosing the initial λ and how to adjust it as you compute iterations depends on the problem at hand.

For a more detailed discussion, see Gauss–Newton algorithm and Levenberg–Marquardt algorithm. Implementations of the Cholesky decomposition, Gauss–Newton method and Levenberg–Marquardt method in GTE can be found in CholeskyDecomposition.h, GaussNewtonMinimizer.h and LevenbergMarquardtMinimizer.h.

3 Affine Fitting of Points Using Height Fields

We have a set of measurements $\{(\boldsymbol{X}_i, h_i)\}_{i=1}^m$ for which $\boldsymbol{X}_i \in \mathbb{R}^n$ are sampled independent variables and $h_i \in \mathbb{R}$ is a sampled dependent variable. The hypothesis is that h is related to $\boldsymbol{X}$ via an affine transformation $h = \boldsymbol{A} \cdot \boldsymbol{X} + b$, where $\boldsymbol{A}$ is an $n \times 1$ vector of constants and b is a scalar constant. The goal is to estimate $\boldsymbol{A}$ and b from the samples. The choice of name h stresses that the measurement errors are in the direction of *height* above the plane containing the $\boldsymbol{X}$ measurements.

3.1 Fitting by a Line in 2 Dimensions

The measurements are $\{(x_i, h_i\}_{i=1}^m$ where x is an independent variable and h is a dependent variable. The affine transformation we want to estimate is $h = ax + b$, where a and b are scalars. This defines a line that

best fits the samples in the sense that the sum of the squared errors between the h_i and the line values $ax_i + b$ is minimized. Note that the error is measured only in the h-direction.

Define the error function for the least-squares minimization to be

$$E(a,b) = \sum_{i=1}^{m} [(ax_i + b) - h_i]^2 \tag{6}$$

This function is nonnegative and its graph is a paraboloid whose vertex occurs when the gradient satisfies $\nabla E(a,b) = (\partial E/\partial a, \partial E/\partial b) = (0,0)$. This leads to a system of two linear equations in a and b which can be easily solved. Precisely,

$$\begin{aligned} 0 &= \partial E/\partial a = 2\textstyle\sum_{i=1}^{m} [(ax_i + b) - h_i]x_i \\ 0 &= \partial E/\partial b = 2\textstyle\sum_{i=1}^{m} [(ax_i + b) - h_i] \end{aligned} \tag{7}$$

and so

$$\begin{bmatrix} \sum_{i=1}^{m} x_i^2 & \sum_{i=1}^{m} x_i \\ \sum_{i=1}^{m} x_i & \sum_{i=1}^{m} 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m} x_i h_i \\ \sum_{i=1}^{m} h_i \end{bmatrix} \tag{8}$$

The system is solved by standard numerical algorithms. If implemented directly, this formulation can lead to an ill-conditioned linear system. To avoid this, you should first compute the averages $\bar{x} = (\sum_{i=1}^{m} x_i)/m$ and $\bar{h} = (\sum_{i=1}^{m} h_i)/m$ and subtract them from the data. The fitted line is of the form $h - \bar{h} = \bar{a}(x - \bar{x}) + \bar{b}$. The linear system of equations that determines the coefficients is

$$\begin{bmatrix} \sum_{i=1}^{m} (x_i - \bar{x})^2 & 0 \\ 0 & m \end{bmatrix} \begin{bmatrix} \bar{a} \\ \bar{b} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m} (x_i - \bar{x})(h_i - \bar{h}) \\ 0 \end{bmatrix} \tag{9}$$

and has solution

$$\bar{a} = \frac{\sum_{i=1}^{m} (x_i - \bar{x})(h_i - \bar{h})}{\sum_{i=1}^{m} (x_i - \bar{x})^2}, \quad \bar{b} = 0 \tag{10}$$

In terms of the original inputs, $a = \bar{a}$ and $b = \bar{h} - \bar{a}\bar{x}$.

3.1.1 Pseudocode for Fitting by a Line

Listing 1 contains pseudocode for fitting a height line to points in 2 dimensions.

Listing 1. Pseudocode for fitting a height line to points in 2 dimensions. The number of input points must be at least 2. The returned Boolean value is `true` as long as the numerator of equation (10) is positive; that is, when the points are not all the same point. An implementation in a slightly more general framework is ApprHeightLine2.h.

```
bool FitHeightLine(int numPoints, Vector2 points[],
    Real& barX, Real& barH, Real& barA)
{
    // Compute the mean of the points.
    Vector2 mean = { 0, 0 };
    for (int i = 0; i < numPoints; ++i)
    {
        mean += points[i];
```

```
    }
    mean /= numPoints;

    // Compute the linear system matrix and vector elements.
    Real xxSum = 0, xhSum = 0;
    for (int i = 0; i < numPoints; ++i)
    {
        Vector2 diff = points[i] - mean;
        xxSum += diff[0] * diff[0];
        linear += diff[0] * diff[1];
    }

    // Solve the linear system.
    if (xxSum > 0)
    {
        // Compute the fitted line h(x) = barH + barA * (x - barX).
        barX = mean[0];
        barH = mean[1];
        barA = linear / xxSum;
        return true;
    }
    else
    {
        // The output is invalid.  The points are all the same.
        barX = 0;
        barH = 0;
        barA = 0;
        return false;
    }
}
```

3.2 Fitting by a Plane in 3 Dimensions

The measurements are $\{(x_i, y_i, h)\}_{i=1}^m$ where x and y are independent variables and h is a dependent variable. The affine transformation we want to estimate is $h = a_0x + a_1y + b$, where a_0, a_1 and b are scalars. This defines a plane that best fits the samples in the sense that the sum of the squared errors between the h_i and the plane values $a_0x_i + a_1y_i + b$ is minimized. Note that the error is measured only in the h-direction.

Define the error function for the least-squares minimization to be

$$E(a_0, a_1, b) = \sum_{i=1}^{m} [(a_0x_i + a_1y_i + b) - h_i]^2 \tag{11}$$

This function is nonnegative and its graph is a hyperparaboloid whose vertex occurs when the gradient satisfies $\nabla E(a_0, a_1, b) = (\partial E/\partial a_0, \partial E/\partial a_1, \partial E/\partial b) = (0, 0, 0)$. This leads to a system of three linear equations in a_0, a_1 and b which can be easily solved. Precisely,

$$\begin{aligned} 0 &= \partial E/\partial a_0 = 2\textstyle\sum_{i=1}^m [(Ax_i + By_i + C) - z_i]x_i \\ 0 &= \partial E/\partial a_1 = 2\textstyle\sum_{i=1}^m [(Ax_i + By_i + C) - z_i]y_i \\ 0 &= \partial E/\partial b = 2\textstyle\sum_{i=1}^m [(Ax_i + By_i + C) - z_i] \end{aligned} \tag{12}$$

and so

$$\begin{bmatrix} \sum_{i=1}^m x_i^2 & \sum_{i=1}^m x_iy_i & \sum_{i=1}^m x_i \\ \sum_{i=1}^m x_iy_i & \sum_{i=1}^m y_i^2 & \sum_{i=1}^m y_i \\ \sum_{i=1}^m x_i & \sum_{i=1}^m y_i & \sum_{i=1}^m 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ b \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^m x_ih_i \\ \sum_{i=1}^m y_ih_i \\ \sum_{i=1}^m h_i \end{bmatrix} \tag{13}$$

The solution is solved by standard numerical algorithms. If implemented directly, this formulation can lead to an ill-conditioned linear system. To avoid this, you should first compute the averages $\bar{x} = (\sum_{i=1}^{m} x_i)/m$, $\bar{y} = (\sum_{i=1}^{m} y_i)/m$ and $\bar{h} = (\sum_{i=1}^{m} h_i)/m$ and subtract them from the data. The fitted plane is of the form $h - \bar{h} = \bar{a}_0(x - \bar{x}) + \bar{a}_1(y - \bar{y}) + \bar{b}$. The linear system of equations that determines the coefficients is

$$\begin{bmatrix} \ell_{00} & \ell_{01} & 0 \\ \ell_{01} & \ell_{11} & 0 \\ 0 & 0 & m \end{bmatrix} \begin{bmatrix} \bar{a}_0 \\ \bar{a}_1 \\ \bar{b} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m}(x_i - \bar{x})^2 & \sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y}) & 0 \\ \sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y}) & \sum_{i=1}^{m}(y_i - \bar{y})^2 & 0 \\ 0 & 0 & m \end{bmatrix} \begin{bmatrix} \bar{a}_0 \\ \bar{a}_1 \\ \bar{b} \end{bmatrix} \tag{14}$$

$$= \begin{bmatrix} \sum_{i=1}^{m}(h_i - \bar{h})(x_i - \bar{x}) \\ \sum_{i=1}^{m}(h_i - \bar{h})(y_i - \bar{y}) \\ 0 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ 0 \end{bmatrix}$$

and has solution

$$\bar{a}_0 = \frac{\ell_{11} r_0 - \ell_{01} r_1}{\ell_{00}\ell_{11} - \ell_{01}^2}, \quad \bar{a}_1 = \frac{\ell_{00} r_1 - \ell_{01} r_0}{\ell_{00}\ell_{11} - \ell_{01}^2}, \quad \bar{b} = 0 \tag{15}$$

In terms of the original inputs, $a_0 = \bar{a}_0$, $a_1 = \bar{a}_1$ and $b = \bar{h} - \bar{a}_0\bar{x} - \bar{a}_1\bar{y}$.

3.2.1 Pseudocode for Fitting by a Plane

Listing 2 contains pseudocode for fitting a height plane to points in 3 dimensions.

Listing 2. Pseudocode for fitting a height plane to points in 3 dimensions. The number of input points must be at least 3. The returned Boolean value is true as long as the matrix of the linear system has nonzero determinant. An implementation in a slightly more general framework is ApprHeightPlane3.h.

```
bool FitHeightPlane(int numPoints, Vector3 points[],
    Real& barX, Real& barY, Real& barH, Real& barA0, Real& barA1)
{
    // Compute the mean of the points.
    Vector3 mean = { 0, 0, 0 };
    for (int i = 0; i < numPoints; ++i)
    {
        mean += points[i];
    }
    mean /= numPoints;

    // Compute the linear system matrix and vector elements.
    Real xxSum = 0, xySum = 0, xhSum = 0, yySum = 0, yhSum = 0;
    for (int i = 0; i < numPoints; ++i)
    {
        Vector3 diff = points[i] - mean;
        xxSum += diff[0] * diff[0];
        xySum += diff[0] * diff[1];
        xhSum += diff[0] * diff[2];
        yySum += diff[1] * diff[1];
        yhSum += diff[1] * diff[2];
    }

    // Solve the linear system.
    Real det = xxSum * yySum - xySum * xySum;
    if (det != 0)
    {
        // Compute the fitted plane h(x,y) = barH + barA0 * (x - barX) + barA1 * (y - barY).
```

```
        barX = mean[0];
        barY = mean[1];
        barH = mean[2];
        barA0 = (yySum * xhSum - xySum * yhSum) / det;
        barA1 = (xxSum * yhSum - xySum * xhSum) / det;
        return true;
    }
    else
    {
        // The output is invalid.  The points are all the same or they are collinear.
        barX = 0;
        barY = 0;
        barH = 0;
        barA0 = 0;
        barA1 = 0;
        return false;
    }
}
```

3.3 Fitting by a Hyperplane in $n+1$ Dimensions

The measurements are $\{(\boldsymbol{X}_i, h_i)\}_{i=1}^m$ where the n components of $\boldsymbol{X}$ are independent variables and h is a dependent variable. The affine transformation we want to estimate is $h = \boldsymbol{A} \cdot \boldsymbol{X} + b$, where $\boldsymbol{A}$ is an $n \times 1$ vector of constants and b is a scalar constant. This defines a hyperplane that best fits the samples in the sense that the sum of the squared errors between the h_i and the hyperplane values $\boldsymbol{A} \cdot \boldsymbol{X}_i + b$ is minimized. Note that the error is measured only in the h-direction.

Define the error function for the least-squares minimization to be

$$E(\boldsymbol{A}, b) = \sum_{i=1}^{m} [(\boldsymbol{A} \cdot \boldsymbol{X}_i + b) - h_i]^2 \tag{16}$$

This function is nonnegative and its graph is a hyperparaboloid whose vertex occurs when the gradient satisfies $\nabla E(\boldsymbol{A}, b) = (\partial E/\partial \boldsymbol{A}, \partial E/\partial b) = (\boldsymbol{0}, 0)$. This leads to a system of $n+1$ linear equations in $\boldsymbol{A}$ and b which can be easily solved. Precisely,

$$\begin{aligned} \boldsymbol{0} &= \partial E/\partial \boldsymbol{A} = 2\textstyle\sum_{i=1}^{m} [(\boldsymbol{A} \cdot \boldsymbol{X}_i + b) - h_i]\boldsymbol{X}_i \\ 0 &= \partial E/\partial b = 2\textstyle\sum_{i=1}^{m} [(\boldsymbol{A} \cdot \boldsymbol{X}_i + b) - h_i] \end{aligned} \tag{17}$$

and so

$$\begin{bmatrix} \sum_{i=1}^{m} \boldsymbol{X}_i \boldsymbol{X}_i^{\mathsf{T}} & \sum_{i=1}^{m} \boldsymbol{X}_i \\ \sum_{i=1}^{m} \boldsymbol{X}_i^{\mathsf{T}} & \sum_{i=1}^{m} 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{A} \\ b \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m} h_i \boldsymbol{X}_i \\ \sum_{i=1}^{m} h_i \end{bmatrix} \tag{18}$$

The solution is solved by standard numerical algorithms. If implemented directly, this formulation can lead to an ill-conditioned linear system. To avoid this, you should first compute the averages $\bar{\boldsymbol{X}} = (\sum_{i=1}^{m} \boldsymbol{X}_i)/m$ and $\bar{h} = (\sum_{i=1}^{m} h_i)/m$ and subtract them from the data. The fitted hyperplane is of the form $h - \bar{h} = \bar{\boldsymbol{A}} \cdot (\boldsymbol{X} - \bar{\boldsymbol{X}}) + \bar{b}$. The linear system of equations that determines the coefficients is

$$\begin{bmatrix} \sum_{i=1}^{m} (\boldsymbol{X}_i - \bar{\boldsymbol{X}}) (\boldsymbol{X}_i - \bar{\boldsymbol{X}})^{\mathsf{T}} & \boldsymbol{0} \\ \boldsymbol{0}^{\mathsf{T}} & m \end{bmatrix} \begin{bmatrix} \bar{\boldsymbol{A}} \\ \bar{b} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m} (h_i - \bar{h}) (\boldsymbol{X}_i - \bar{\boldsymbol{X}}) \\ 0 \end{bmatrix} \tag{19}$$

and has solution

$$\bar{\boldsymbol{A}} = \left(\sum_{i=1}^{m} (\boldsymbol{X}_i - \bar{\boldsymbol{X}})(\boldsymbol{X}_i - \bar{\boldsymbol{X}})^{\mathsf{T}} \right)^{-1} \left(\sum_{i=1}^{m} (h_i - \bar{h})(\boldsymbol{X}_i - \bar{\boldsymbol{X}}) \right), \quad \bar{b} = 0 \tag{20}$$

In terms of the original inputs, $\boldsymbol{A} = \bar{\boldsymbol{A}}$ and $b = \bar{h} - \bar{\boldsymbol{A}} \cdot \bar{\boldsymbol{X}}$.

3.3.1 Pseudocode for Fitting a Hyperplane

Listing 3 contains pseudocode for fitting a height hyperplane to points in $n+1$ dimensions.

Listing 3. Pseudocode for fitting a height hyperplane to points in $n+1$ dimensions. The number of input points must be at least n. The returned Boolean value is `true` as long as the matrix of the linear system has nonzero determinant.

```
bool FitHeightHyperplane(int numPoints, Vector<n + 1> points[],
    Vector<n>& barX, Real& barH, Vector<n>& barA)
{
    // Compute the mean of the points.
    Vector<n + 1> mean = Vector<n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        mean += points[i];
    }
    mean /= numPoints;

    // Compute the linear system matrix and vector elements.  The function
    // Vector<n> Head<n>(Vector<n + 1> V) returns (V[0],...,V[n-1]).
    Matrix<n,n> L = Matrix<n,n>::ZERO;
    Vector<n> R = Vector<n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        Vector<n + 1> diff = points[i] - mean;
        Vector<n> XminusBarX = Head<n>(diff);
        Real HminusBarH = diff[n];
        L += OuterProduct(XminusBarX, XminusBarX);  // (X[i]-barX[i])*(X[i]-barX[i])^T
        R += HminusBarH * XminusBarX;
    }

    // Solve the linear system.
    Real det = Determinant(L);
    if (det != 0)
    {
        // Compute the fitted plane h(X) = barH + Dot(barA, X - barX).
        barX = Head<n>(mean);
        barH = mean[n];
        barA = SolveLinearSystem(L, R);  // solve L*A = R
        return true;
    }
    else
    {
        // The output is invalid.  The points appear not to live on a
        // hyperplane; they might live in an affine subspace of dimension
        // smaller than n.
        barX = Vector<n>::ZERO;
        barH = 0;
        barA = Vector<n>::ZERO;
        return false;
    }
}
```

---

4 Affine Fitting of Points Using Orthogonal Regression

We have a set of measurements $\{\boldsymbol{X}_i\}_{i=1}^m$ for which $\boldsymbol{X}_i \in \mathbb{R}^n$ are sampled independent variables. The hypothesis is that the points are sampled from a k-dimensional affine subspace in n-dimensional space. Such a space is referred to as a *k-dimensional flat*. The classic cases include fitting a line to points in n dimensions and fitting a plane to points in 3 dimensions. The latter is a special case of fitting a hyperplane, an $(n-1)$-dimensional flat, to points in n dimensions.

In the height-field fitting algorithms, the least-squares errors were measured in a specified direction (the height direction). An alternative is to measure the errors in the perpendicular direction to the purported affine subspace. This approach is referred to as *orthogonal regression*.

4.1 Fitting by a Line [1 Dimension]

The algorithm may be applied to sample points $\{\boldsymbol{X}_i\}_{i=1}^m$ in any dimension n. Let the line have origin $\boldsymbol{A}$ and unit-length direction $\boldsymbol{D}$, both $n \times 1$ vectors. Define $\boldsymbol{Y}_i = \boldsymbol{X}_i - \boldsymbol{A}$, which can be written as $\boldsymbol{Y}_i = (\boldsymbol{D} \cdot \boldsymbol{Y}_i)\,\boldsymbol{D} + \boldsymbol{D}_i^\perp$ where $\boldsymbol{D}_i^\perp$ is the perpendicular vector from $\boldsymbol{X}_i$ to its projection on the line. The squared length of this vector is $|\boldsymbol{D}_i^\perp|^2 = |\boldsymbol{Y}_i - d_i\boldsymbol{D}|^2$. The error function for the least-squares minimization is $E(\boldsymbol{A}, \boldsymbol{D}) = \sum_{i=1}^m |\boldsymbol{D}_i^\perp|^2$. Two alternate forms for this function are

$$E(\boldsymbol{A}, \boldsymbol{D}) = \sum_{i=1}^m \left(\boldsymbol{Y}_i^{\mathsf{T}} \left(I - \boldsymbol{D}\boldsymbol{D}^{\mathsf{T}}\right) \boldsymbol{Y}_i\right) \tag{21}$$

and

$$E(\boldsymbol{A}, \boldsymbol{D}) = \boldsymbol{D}^{\mathsf{T}} \left(\sum_{i=1}^m \left((\boldsymbol{Y}_i \cdot \boldsymbol{Y}_i) I - \boldsymbol{Y}_i \boldsymbol{Y}_i^{\mathsf{T}}\right)\right) \boldsymbol{D} = \boldsymbol{D}^{\mathsf{T}} M \boldsymbol{D} \tag{22}$$

where M is a positive semidefinite symmetric matrix that depends on $\boldsymbol{A}$ and the $\boldsymbol{Y}_i$ but not in $\boldsymbol{D}$.

Compute the derivative of equation (21) with respect to $\boldsymbol{A}$ to obtain

$$\frac{\partial E}{\partial \boldsymbol{A}} = -2\left[I - \boldsymbol{D}\boldsymbol{D}^{\mathsf{T}}\right] \sum_{i=1}^m \boldsymbol{Y}_i \tag{23}$$

At a minimum value of E, it is necessary that this derivative is zero, which it is when $\sum_{i=1}^m \boldsymbol{Y}_i = 0$, implying $\boldsymbol{A} = (1/m)\sum_{i=1}^m \boldsymbol{X}_i$, the average of the sample points. In fact there are infinitely many solutions, $\boldsymbol{A} + s\boldsymbol{D}$, for any scalar s. This is simply a statement that $\boldsymbol{A}$ is a point on the best-fit line, but any other point on the line may serve as the origin for that line.

Equation (22) is a quadratic form $\boldsymbol{D}^{\mathsf{T}} M \boldsymbol{D}$ whose minimum is the smallest eigenvalue of M, computed using standard eigensystem solvers. A corresponding unit length eigenvector $\boldsymbol{D}$ completes our construction of the least-squares line. The covariance matrix of the input points is $C = \sum_{i=1}^m \boldsymbol{Y}_i \boldsymbol{Y}_i^{\mathsf{T}}$. Defining $\delta = \sum_{i=1}^m \boldsymbol{Y}_i^{\mathsf{T}} \boldsymbol{Y}_i$, we see that $M = \delta I - C$, where I is the identity matrix. Therefore, M and C have the same eigenspaces. The eigenspace corresponding to the minimum eigenvalue of M is the same as the eigenspace corresponding to the maximum eigenvalue of C. In an implementation, it is sufficient to process C and avoid the additional cost to compute M.

4.1.1 Pseudocode for the General Case

Listing 4 contains pseudocode for fitting a line to points in n dimensions with $n \geq 2$.

Listing 4. Pseudocode for fitting a line to points in n dimensions using orthogonal regression. The number of input points must be at least 2. The returned Boolean value is true as long as the covariance matrix of the linear system has a 1-dimensional eigenspace for the maximum eigenvalue of the covariance matrix.

```
bool FitOrthogonalLine(int numPoints, Vector<n> points[],
    Vector<n>& origin, Vector<n>& direction)
{
    // Compute the mean of the points.
    Vector<n> mean = Vector<n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        mean += points[i];
    }
    mean /= numPoints;

    // Compute the covariance matrix of the points.
    Matrix<n,n> C = Matrix<n,n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        Vector<n> diff = points[i] - mean;
        C += OuterProduct(diff, diff);  // diff * diff^T
    }

    // Compute the eigenvalues and eigenvectors of C, where the eigenvalues are sorted
    // in nondecreasing order (eigenvalues[0] <= eigenvalues[1] <= ...).
    Real eigenvalues[n];
    Vector<n> eigenvectors[n];
    SolveEigensystem(C, eigenvalues, eigenvectors);

    // Set the output information.
    origin = mean;
    direction = eigenvectors[n-1];

    // The fitted line is unique when the maximum eigenvalue has multiplicity 1.
    return eigenvalues[n-2] < eigenvalues[n-1];
}
```

Specializations for 2 and 3 dimensions are simple, computing only the upper-triangular elements of C and passing them to specialized eigensolvers for 2 and 3 dimensions. Implementations are ApprOrthogonalLine2.h and ApprOrthogonalLine3.h.

4.2 Fitting by a Hyperplane [$(n-1)$ Dimensions]

The algorithm may be applied to sample points $\{\boldsymbol{X}_i\}_{i=1}^m$ in any dimension n. Let the hyperplane be defined implicitly by $\boldsymbol{N} \cdot (\boldsymbol{X} - \boldsymbol{A}) = 0$, where $\boldsymbol{N}$ is a unit-length normal to the hyperplane and $\boldsymbol{A}$ is a point on the hyperplane. Define $\boldsymbol{Y}_i = \boldsymbol{X}_i - \boldsymbol{A}$, which can be written as $\boldsymbol{Y}_i = (\boldsymbol{N} \cdot \boldsymbol{Y}_i)\boldsymbol{N} + \boldsymbol{N}_i^{\perp}$ where $\boldsymbol{N}_i^{\perp}$ is a vector that is perpendicular to $\boldsymbol{N}$. The squared length of the projection of $\boldsymbol{Y}_i$ onto the normal line for the hyperplane is $(\boldsymbol{N} \cdot \boldsymbol{Y}_i)^2$. The error function for the least-squares minimization is $E(\boldsymbol{A}, \boldsymbol{N}) = \sum_{i=1}^m (\boldsymbol{N} \cdot \boldsymbol{Y}_i)^2$. Two alternate forms for this function are

$$E(\boldsymbol{A}, \boldsymbol{N}) = \sum_{i=1}^{m} \left(\boldsymbol{Y}_i^{\mathsf{T}} \left(\boldsymbol{N}\boldsymbol{N}^{\mathsf{T}} \right) \boldsymbol{Y}_i \right) \tag{24}$$

and

$$E(\boldsymbol{A}, \boldsymbol{N}) = \boldsymbol{N}^{\mathsf{T}} \left(\sum_{i=1}^{m} \boldsymbol{Y}_i \boldsymbol{Y}_i^{\mathsf{T}} \right) \boldsymbol{N} = \boldsymbol{N}^{\mathsf{T}} C \boldsymbol{N} \tag{25}$$

where $C = \sum_{i=1}^{m} \boldsymbol{Y}_i \boldsymbol{Y}_i^{\mathsf{T}}$ is the covariance matrix of the $\boldsymbol{Y}_i$.

Compute the derivative of equation (24) with respect to $\boldsymbol{A}$ to obtain

$$\frac{\partial E}{\partial \boldsymbol{A}} = 2 \left(\boldsymbol{N} \boldsymbol{N}^{\mathsf{T}} \right) \sum_{i=1}^{m} \boldsymbol{Y}_i \tag{26}$$

At a minimum value of E, it is necessary that this derivative is zero, which it is when $\sum_{i=1}^{m} \boldsymbol{Y}_i = 0$, implying $\boldsymbol{A} = (1/m) \sum_{i=1}^{m} \boldsymbol{X}_i$, the average of the sample points. In fact there are infinitely many solutions, $\boldsymbol{A} + \boldsymbol{W}$, where $\boldsymbol{W}$ is any vector perpendicular to $\boldsymbol{N}$. This is simply a statement that the average is on the best-fit hyperplane, but any other point on the hyperplane may serve as the origin for that hyperplane.

Equation (25) is a quadratic form $\boldsymbol{N}^{\mathsf{T}} C \boldsymbol{N}$ whose minimum is the smallest eigenvalue of C, computed using standard eigensystem solvers. A corresponding unit-length eigenvector $\boldsymbol{N}$ completes our construction of the least-squares hyperplane.

4.2.1 Pseudocode for the General Case

Listing 5 contains pseudocode for fitting a hyperplane to points in n dimensions with $n \geq 3$.

Listing 5. Pseudocode for fitting a line to points in n dimensions using orthogonal regression. The number of input points must be at least n. The returned Boolean value is true as long as the covariance matrix of the linear system has a 1-dimensional eigenspace for the minimum eigenvalue of the covariance matrix.

```
bool FitOrthogonalHyperplane(int numPoints, Vector<n> points[],
    Vector<n>& origin, Vector<n>& normal)
{
    // Compute the mean of the points.
    Vector<n> mean = Vector<n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        mean += points[i];
    }
    mean /= numPoints;

    // Compute the covariance matrix of the points.
    Matrix<n,n> C = Matrix<n,n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        Vector<n> diff = points[i] - mean;
        C += OuterProduct(diff, diff);  // diff * diff^T
    }

    // Compute the eigenvalues and eigenvectors of M, where the eigenvalues are sorted
    // in nondecreasing order (eigenvalues[0] <= eigenvalues[1] <= ...).
    Real eigenvalues[n];
    Vector<n> eigenvectors[n];
    SolveEigensystem(C, eigenvalues, eigenvectors);

    // Set the output information.
    origin = mean;
    normal = eigenvectors[0];

    // The fitted hyperplane is unique when the minimum eigenvalue has multiplicity 1.
    return eigenvalues[0] < eigenvalues[1];
}
```

A specialization for 3 dimensions is simple, computing only the upper-triangular elements of C and passing them to a specialized eigensolver for 3 dimensions. An implementations is ApprOrthogonalPlane3.h.

4.3 Fitting by a Flat [k Dimensions]

Orthogonal regression to fit n-dimensional points by a line or by a hyperplane can be generalized to fitting by an affine subspace called a *k-dimensional flat*, where you may choose k such that $1 \le k \le n-1$. A line is a 1-flat and a hyperplane is an $(n-1)$-flat.

For dimension $n = 3$, we fit with flats that are either lines ($k = 1$) or planes ($k = 2$). For dimensions $n \ge 4$ and flat dimensions $1 \le k \le n-1$, the generalization of orthogonal regression is the following. The bases mentioned here are for the linear portion of the affine subspace; that is, the basis vectors are relative to an origin at a point $\boldsymbol{A}$. The flat has an orthonormal basis $\{\boldsymbol{F}_j\}_{j=1}^{k}$ and the orthogonal complement has an orthonormal basis $\{\boldsymbol{P}_j\}_{j=1}^{n-k}$. The union of the two bases is an orthonormal basis for $\mathbb{R}^n$. Any input point $\boldsymbol{X}_i$, $1 \le i \le m$, can be represented by

$$\boldsymbol{X}_i = \boldsymbol{A} + \sum_{j=1}^{k} f_{ij}\boldsymbol{F}_j + \sum_{j=1}^{n-k} p_{ij}\boldsymbol{P}_j = \left(\boldsymbol{A} + \sum_{j=1}^{k} f_{ij}\boldsymbol{F}_j\right) + \left(\sum_{j=1}^{n-k} p_{ij}\boldsymbol{P}_j\right) \tag{27}$$

The left-parenthesized term is the portion of $\boldsymbol{X}_i$ that lives in the flat and the right-parenthesized is the portion that is the deviation of $\boldsymbol{X}_i$ from the flat. The least-squares problem is about choosing the two bases so that the sum of squared lengths of the deviations is as small as possible.

Define $\boldsymbol{Y}_i = \boldsymbol{X}_i - \boldsymbol{A}$. The basis coefficients are $f_{ij} = \boldsymbol{F}_j \cdot \boldsymbol{Y}_i$ and $p_{ij} = \boldsymbol{P}_j \cdot \boldsymbol{Y}_i$. The squared length of the deviation is $\sum_{j=1}^{n-k} p_{ij}^2$. The error function for the least-squares minimization is the sum of the squared lengths for all inputs, $E = \sum_{i=1}^{m}\sum_{j=1}^{n-k} p_{ij}^2$. Two alternate forms for this function are

$$E(\boldsymbol{A}, \boldsymbol{P}_1, \ldots \boldsymbol{P}_{n-k}) = \sum_{i=1}^{m} \boldsymbol{Y}_i^{\mathsf{T}} \left(\sum_{j=1}^{n-k} \boldsymbol{P}_j\boldsymbol{P}_j^{\mathsf{T}}\right) \boldsymbol{Y}_i \tag{28}$$

and

$$E(\boldsymbol{A}, \boldsymbol{P}_1, \ldots \boldsymbol{P}_{n-k}) = \sum_{j=1}^{n-k} \boldsymbol{P}_j^{\mathsf{T}} \left(\sum_{i=1}^{m} \boldsymbol{Y}_i\boldsymbol{Y}_i^{\mathsf{T}}\right) \boldsymbol{P}_j = \sum_{j=1}^{n-k} \boldsymbol{P}_j^{\mathsf{T}} C \boldsymbol{P}_j \tag{29}$$

where $C = \sum_{i=1} \boldsymbol{Y}_i\boldsymbol{Y}_i^{\mathsf{T}}$.

Compute the derivative of equation (28) with respect to $\boldsymbol{A}$ to obtain

$$\frac{\partial E}{\partial \boldsymbol{A}} = 2\left(\sum_{j=1}^{n-k} \boldsymbol{P}_j\boldsymbol{P}_j^{\mathsf{T}}\right) \sum_{i=1}^{m} \boldsymbol{Y}_i \tag{30}$$

At a minimum value of E, it is necessary that this derivative is zero, which it is when $\sum_{i=1}^{m} \boldsymbol{Y}_i = 0$, implying $\boldsymbol{A} = (1/m)\sum_{i=1}^{m} \boldsymbol{X}_i$, the average of the sample points. In fact there are infinitely many solutions, $\boldsymbol{A} + \boldsymbol{W}$, where $\boldsymbol{W}$ is any vector in the orthogonal complement of the subspace spanned by the $\boldsymbol{P}_j$; this subspace is

the one spanned by the $\boldsymbol{F}_j$. This is simply a statement that the average is on the best-fit flat, but any other point on the flat may serve as the origin for that flat. Because we are choosing $\boldsymbol{A}$ to be the average of the input points, the matrix C is the covariance matrix for the input points.

The last term in equation (29) is a sum of quadratic forms involving the matrix C and the vectors $\boldsymbol{P}_j$ that are a basis (unit length, mutually perpendicular). The minimum value of the quadratic form is the smallest eigenvalue λ_1 of C, so we may choose $\boldsymbol{P}_1$ to be a unit-length eigenvector of C corresponding to λ_1. We must choose $\boldsymbol{P}_2$ to be unit length and perpendicular to $\boldsymbol{P}_1$. If the eigenspace for λ_1 is 1-dimensional, the next smallest value we can attain by the quadratic form is the smallest eigenvalue λ_2 for which $\lambda_1 < \lambda_2$. $\boldsymbol{P}_2$ is chosen to be a corresponding unit-length eigenvector. However, if λ_1 has an eigenspace of dimension larger than 1, we can choose $\boldsymbol{P}_2$ in that eigenspace but which is perpendicular to $\boldsymbol{P}_1$.

Generally, let $\{\lambda_\ell\}_{\ell=1}^r$ be the r distinct eigenvalues of the covariance matrix C; we know $1 \le r \le n$ and $\lambda_1 < \lambda_2 < \cdots < \lambda_r$. Let the dimension of the eigenspace for λ_ℓ be $d_\ell \ge 1$; we know that $\sum_{\ell=1}^r d_\ell = n$. List the eigenvalues and eigenvectors in order of increasing eigenvalue, including repeated values,

$$\underbrace{\begin{matrix}\lambda_1 & \cdots & \lambda_1 \\ \boldsymbol{V}_1^1 & \cdots & \boldsymbol{V}_{d_1}^1\end{matrix}}_{d_1 \text{ terms}} \quad \underbrace{\begin{matrix}\lambda_2 & \cdots & \lambda_2 \\ \boldsymbol{V}_1^2 & \cdots & \boldsymbol{V}_{d_2}^2\end{matrix}}_{d_2 \text{ terms}} \quad \cdots \quad \underbrace{\begin{matrix}\lambda_r & \cdots & \lambda_r \\ \boldsymbol{V}_1^r & \cdots & \boldsymbol{V}_{d_r}^r\end{matrix}}_{d_r \text{ terms}} \tag{31}$$

The list has n items. The eigenvalue d_ℓ has an eigenspace with orthonormal basis $\{\boldsymbol{V}_j^\ell\}_{j=1}^{d_\ell}$. In this list, choose the first k eigenvectors to be $\boldsymbol{P}_1$ through $\boldsymbol{P}_k$ and choose the last $n-k$ eigenvectors to be $\boldsymbol{F}_1$ through $\boldsymbol{F}_{n-k}$.

It is possible that one (or more) of the $\boldsymbol{P}_j$ and one (or more) of the $\boldsymbol{F}_j$ are in the eigenspace for the same eigenvalue. In this case, the fitted flat is not unique, and one should re-examine the choice of dimension k for the fitted flat. This is analogous to the following situations in dimension $n = 3$:

- The input points are nearly collinear but you are trying to fit those points with a plane. The covariance matrix likely has two distinct eigenvalues $\lambda_1 < \lambda_2$ with $d_1 = 2$ and $d_2 = 1$. The basis vectors are $\boldsymbol{P}_1 = \boldsymbol{V}_1^1$, $\boldsymbol{F}_1 = \boldsymbol{V}_2^1$, and $\boldsymbol{F}_2 = \boldsymbol{V}_1^2$. The first two of these are from the same eigenspace.
- The input points are spread out over a portion of a plane (and are not nearly collinear) but you are trying to fit those points with a line. The covariance matrix likely has two distinct eigenvalues $\lambda_1 < \lambda_2$ with $d_1 = 1$ and $d_2 = 2$. The basis vectors are $\boldsymbol{P}_1 = \boldsymbol{V}_1^1$, $\boldsymbol{P}_2 = \boldsymbol{V}_1^2$, and $\boldsymbol{F}_1 = \boldsymbol{V}_2^2$. The last two of these are from the same eigenspace.
- The input points are not well fit by a flat of any dimension. For example, your input points are uniformly distributed over a sphere. The covariance matrix likely has one distinct eigenvalue λ_1 of multiplicity $d_1 = 3$. Neither a line nor a plane is a good fit to the input points—in either case, a $\boldsymbol{P}$-vector and an $\boldsymbol{F}$-vector are in the same eigenspace.

The computational algorithm is to compute the average $\boldsymbol{A}$ and covariance matrix of the points. Use an eigensolver whose output eigenvalues are sorted in nondecreasing order. Choose the $\boldsymbol{F}_j$ to be the last $n-k$ eigenvectors output by the eigensolver.

4.3.1 Pseudocode for the General Case

Listing 6 contains pseudocode for fitting a k-dimensional flat to points in n-dimensions where $n \ge 2$ and $1 \le k \le n-1$.

Listing 6. Pseudocode for fitting a k-dimensional flat to points in n dimensions using orthogonal regression. The number of input points must be at least $k+1$. The returned Boolean value is true as long as the covariance matrix of the linear system has a basis of eigenvectors sorted by nondecreasing eigenvalues for which the following holds. The subbasis that spans the linear space of the flat and the subbasis that spans the orthogonal complement of the linear space of the flat do not both contain basis vectors from the same eigenspace.

```
bool FitOrthogonalFlat(int numPoints, Vector<n> points[],
    Vector<n>& origin, Vector<k>& flatBasis, Vector<n-k>& complementBasis)
{
    // Compute the mean of the points.
    Vector<n> mean = Vector<n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        mean += points[i];
    }
    mean /= numPoints;

    // Compute the covariance matrix of the points.
    Matrix<n,n> C = Matrix<n,n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        Vector<n> diff = points[i] - mean;
        C += OuterProduct(diff, diff);  // diff * diff^T
    }

    // Compute the eigenvalues and eigenvectors of M, where the eigenvalues are sorted
    // in nondecreasing order (eigenvalues[0] <= eigenvalues[1] <= ...).
    Real eigenvalues[n];
    Vector<n> eigenvectors[n];
    SolveEigensystem(C, eigenvalues, eigenvectors);

    // Set the output information.  The basis for the fitted flat corresponds
    // to the largest variances and the basis for the complement of the fitted
    // flat corresponds to the smallest variances.
    origin = mean;
    for (int i = 0; i < n - k; ++i)
    {
        complementBasis[i] = eigenvectors[i];
    }
    for (int i = 0; i < k; ++i)
    {
        flatBasis[i] = eigenvectors[n - k + i];
    }

    // The fitted flat and its complement do not have vectors from the same eigenspace.
    return eigenvalues[n - k - 1] < eigenvalues[n - k];
}
```

In the special case of fitting with a line ($k=1$), the line direction is the only element in flatBasis. In the special case of fitting with a hyperplane ($k=n-1$), the hyperplane normal is the only element in complementBasis.

5 Fitting a Hypersphere to Points

The classic cases are fitting 2-dimensional points by circles and 3-dimensional points by spheres. In n-dimensions, the objects are called hyperspheres, defined implicitly by the quadratic equation $|\boldsymbol{C}-\boldsymbol{X}|^2 = r^2$, where $\boldsymbol{C}$ is the center and r is the radius. Three algorithms are presented for fitting points by hyperspheres.

5.1 Fitting Using Differences of Lengths and Radius

The sample points are $\{\boldsymbol{X}_i\}_{i=1}^m$. The least-squares error function involves the squares of differences between lengths and radius,

$$E(\boldsymbol{C}, r) = \sum_{i=1}^{m} \left(|\boldsymbol{C} - \boldsymbol{X}_i| - r\right)^2 \tag{32}$$

The minimization is based on computing points where the gradient of E is zero. The partial derivative with respect to r is

$$\frac{\partial E}{\partial r} = -2 \sum_{i=1}^{m} \left(|\boldsymbol{C} - \boldsymbol{X}_i| - r\right) \tag{33}$$

Setting the derivative equal to zero and solving for the radius,

$$r = \frac{1}{m} \sum_{i=1}^{m} |\boldsymbol{C} - \boldsymbol{X}_i| \tag{34}$$

which says that the radius is the average of the distances from the sample points to the center $\boldsymbol{C}$. The partial derivative with respect to $\boldsymbol{C}$ is

$$\begin{aligned} \frac{\partial E}{\partial \mathbf{C}} &= 2 \textstyle\sum_{i=1}^{m} \left(|\boldsymbol{C} - \boldsymbol{X}_i| - r\right) \frac{\partial |\mathbf{C} - \mathbf{X}_i|}{\partial \mathbf{C}} \\ &= 2 \textstyle\sum_{i=1}^{m} \left(|\boldsymbol{C} - \boldsymbol{X}_i| - r\right) \frac{\mathbf{C} - \mathbf{X}_i}{|\mathbf{C} - \mathbf{X}_i|} \\ &= 2 \textstyle\sum_{i=1}^{m} \left((\boldsymbol{C} - \boldsymbol{X}_i) - r \frac{\mathbf{C} - \mathbf{X}_i}{|\mathbf{C} - \mathbf{X}_i|}\right) \end{aligned} \tag{35}$$

Setting the derivative equal to zero and solving for the center,

$$\boldsymbol{C} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{X}_i + r \frac{1}{m} \sum_{i=1}^{m} \frac{\mathbf{C} - \mathbf{X}_i}{|\mathbf{C} - \mathbf{X}_i|} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{X}_i + \left(\frac{1}{m} \sum_{i=1}^{m} |\boldsymbol{C} - \boldsymbol{X}_i|\right) \left(\frac{1}{m} \sum_{i=1}^{m} \frac{\mathbf{C} - \mathbf{X}_i}{|\mathbf{C} - \mathbf{X}_i|}\right) \tag{36}$$

The average of the samples is $\bar{\boldsymbol{X}}$. Define length $L_i = |\boldsymbol{C} - \boldsymbol{X}_i|$; the average of the lengths is $\bar{L}$. Define unit-length vector $\boldsymbol{U}_i = (\boldsymbol{C} - \boldsymbol{X}_i)/|\boldsymbol{C} - \boldsymbol{X}_i|$; the average of the unit-length vectors is $\bar{\boldsymbol{U}}$. Equation (36) becomes

$$\boldsymbol{C} = \bar{\boldsymbol{X}} + \bar{L}\bar{\boldsymbol{U}} =: \boldsymbol{F}(\boldsymbol{C}) \tag{37}$$

were the last equality defines the vector-valued function $\boldsymbol{F}$. The function depends on the independent variable $\boldsymbol{C}$ because both $\bar{L}$ and $\bar{\boldsymbol{U}}$ depend on $\boldsymbol{C}$. Fixed-point iteration can be applied to solve equation (37),

$$\boldsymbol{C}_0 = \bar{\boldsymbol{X}}; \;\; \boldsymbol{C}_{i+1} = \boldsymbol{F}(\boldsymbol{C}_i), \;\; i \ge 0 \tag{38}$$

Depending on the distribution of the samples, it is possible to choose a different initial guess for $\boldsymbol{C}_0$ that (hopefully) leads to faster convergence.

5.1.1 Pseudocode for the General Case

Listing 7 contains pseudocode for fitting a hypersphere to points. The case $n = 2$ is for circles and the case $n = 3$ is for spheres.

Listing 7. Fitting a hypersphere to points using least squares based on squared differences of lengths and radius. If you want the incoming hypersphere center to be the initial guess for the center, set inputCenterIsInitialGuess; otherwise, the initial guess is computed to be the average of the samples. The maximum number of iterations is also specified. The returned function value is the number of iterations used.

```
int FitHypersphere(int numPoints, Vector<n> X[], int maxIterations, bool inputCenterIsInitialGuess,
    Vector<n>& center, Real& radius)
{
    // Compute the average of the data points.
    Vector<n> averageX = X[0];
    for (int i = 1; i < numPoints; ++i)
    {
        averageX += X[i];
    }
    averageX /= numPoints;

    // The initial guess for the center is either the incoming center of the
    // average of the sample points.
    if (!inputCenterIsInitialGuess)
    {
        center = averageX;
    }

    int iteration;
    for (iteration = 0; iteration < maxIterations; ++iteration)
    {
        // Update the estimate for the center.
        Vector<n> previousCenter = center;

        // Compute average L and average U.
        Real averageL = 0;
        Vector<n> averageU = Vector<n>::ZERO;
        for (int i = 0; i < numPoints; ++i)
        {
            Vector<n> CmXi = center - X[i];
            Real length = Length(CmXi);
            if (length > 0)
            {
                averageL += length;
                averageU -= CmXi / length;
            }
        }
        averageL /= numPoints;
        averageU /= numPoints;

        center = averageX + averageL * averageU;
        radius = averageL;

        // Test for convergence.
        if (center == previousCenter)
        {
            break;
        }
    }

    return ++iteration;
}
```

The convergence test uses an exact equality, that the previous center $\boldsymbol{C}'$ and the current center $\boldsymbol{C}$ are the same. In practice you might want to specify a small $\varepsilon > 0$ and instead exit when $|\boldsymbol{C} - \boldsymbol{C}'| \leq \varepsilon$.

Specializations for 2 and 3 dimensions have the same implementation as the general case. Implementations are ApprCircle2.h and ApprSphere3.h.

5.2 Fitting Using Differences of Squared Lengths and Squared Radius

The sample points are $\{\boldsymbol{X}_i\}_{i=1}^m$. The least-squares error function involves the squares of differences between squared lengths and squared radius,

$$E(\boldsymbol{C}, r^2) = \sum_{i=1}^{m} \left(|\boldsymbol{C} - \boldsymbol{X}_i|^2 - r^2\right)^2 \tag{39}$$

The minimization is based on computing points where the gradient of E is zero. The partial derivative with respect to r^2 is

$$\frac{\partial E}{\partial r^2} = -2\sum_{i=1}^{m} \left(|\boldsymbol{C} - \boldsymbol{X}_i|^2 - r^2\right) \tag{40}$$

Define $\boldsymbol{\Delta}_i = \boldsymbol{C} - \boldsymbol{X}_i$. Setting the derivative to zero and solving for the squared radius,

$$r^2 = \frac{1}{m}\sum_{i=1}^{m} |\boldsymbol{C} - \boldsymbol{X}_i|^2 = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{\Delta}_i^{\mathsf{T}}\boldsymbol{\Delta}_i \tag{41}$$

which says that the squared radius is the average of the squared distances from the sample points to the center $\boldsymbol{C}$. The partial derivative with respect to $\boldsymbol{C}$ is

$$\frac{\partial E}{\partial \boldsymbol{C}} = 4\sum_{i=1}^{m} \left(|\boldsymbol{C} - \boldsymbol{X}_i|^2 - r^2\right)(\boldsymbol{C} - \boldsymbol{X}_i) = 4\sum_{i=1}^{m} \left(\boldsymbol{\Delta}_i^{\mathsf{T}}\boldsymbol{\Delta}_i - r^2\right)\boldsymbol{\Delta}_i \tag{42}$$

Setting this to zero, the center and radius must satisfy

$$\sum_{i=1}^{m} \left(\boldsymbol{\Delta}_i^{\mathsf{T}}\boldsymbol{\Delta}_i - r^2\right)\boldsymbol{\Delta}_i = \boldsymbol{0} \tag{43}$$

Expanding the squared lengths,

$$\boldsymbol{\Delta}_i^{\mathsf{T}}\boldsymbol{\Delta}_i = |\boldsymbol{C}|^2 - 2\boldsymbol{C}^{\mathsf{T}}\boldsymbol{X}_i + |\boldsymbol{X}_i|^2 \tag{44}$$

Substituting into equation (41),

$$r^2 = |\boldsymbol{C}|^2 - 2\boldsymbol{C}^{\mathsf{T}}\left(\frac{1}{m}\sum_{i=1}^{m}\boldsymbol{X}_i\right) + \frac{1}{m}\sum_{i=1}^{m}|\boldsymbol{X}_i|^2 = |\boldsymbol{C}|^2 - 2\boldsymbol{C}^{\mathsf{T}}\boldsymbol{A} + \frac{1}{m}\sum_{i=1}^{m}|\boldsymbol{X}_i|^2 \tag{45}$$

where $\boldsymbol{A} = \left(\sum_{i=1}^m \boldsymbol{X}_i\right)/m$ is the average of the samples. Define $\boldsymbol{Y}_i = \boldsymbol{X}_i - \boldsymbol{A}$. Some algebra will show that

$$\begin{aligned}
\boldsymbol{\Delta}_i^{\mathsf{T}}\boldsymbol{\Delta}_i - r^2 &= |\boldsymbol{C}|^2 - 2\boldsymbol{C}^{\mathsf{T}}\boldsymbol{X}_i + |\boldsymbol{X}_i|^2 - \left(|\boldsymbol{C}|^2 - 2\boldsymbol{C}^{\mathsf{T}}\boldsymbol{A} + \tfrac{1}{m}\textstyle\sum_{j=1}^m |\boldsymbol{X}_j|^2\right) \\
&= -2\boldsymbol{C}^{\mathsf{T}}\boldsymbol{Y}_i + |\boldsymbol{X}_i|^2 - \tfrac{1}{m}\textstyle\sum_{j=1}^m |\boldsymbol{X}_j|^2 \\
&= -2(\boldsymbol{C} - \boldsymbol{A})^{\mathsf{T}}\boldsymbol{Y}_i + |\boldsymbol{Y}_i|^2 - \tfrac{1}{m}\textstyle\sum_{j=1}^m |\boldsymbol{Y}_j|^2 \\
&= -2(\boldsymbol{C} - \boldsymbol{A})^{\mathsf{T}}\boldsymbol{Y}_i + B_i
\end{aligned} \tag{46}$$

where the last equality defines B_i. Equation (43) becomes

$$\begin{aligned}
\boldsymbol{0} &= \textstyle\sum_{i=1}^m \left(\boldsymbol{\Delta}_i^{\mathsf{T}}\boldsymbol{\Delta}_i - r^2\right)\boldsymbol{\Delta}_i \\
&= \textstyle\sum_{i=1}^m \left(-2(\boldsymbol{C} - \boldsymbol{A})^{\mathsf{T}}\boldsymbol{Y}_i + B_i\right)((\boldsymbol{C} - \boldsymbol{A}) - \boldsymbol{Y}_i) \\
&= \left((\boldsymbol{C} - \boldsymbol{A})^{\mathsf{T}}\textstyle\sum_{i=1}^m \boldsymbol{Y}_i\right)(\boldsymbol{C} - \boldsymbol{A}) + 2\left(\textstyle\sum_{i=1}^m \boldsymbol{Y}_i\boldsymbol{Y}_i^{\mathsf{T}}\right)(\boldsymbol{C} - \boldsymbol{A}) + \left(\textstyle\sum_{i=1}^m B_i\right)(\boldsymbol{C} - \boldsymbol{A}) - \textstyle\sum_{i=1}^m B_i\boldsymbol{Y}_i
\end{aligned} \tag{47}$$

It is easily shown that $\sum_{i=1}^{m} \boldsymbol{Y}_i = \boldsymbol{0}$ and $\sum_{i=1}^{m} B_i = 0$; therefore,

$$\boldsymbol{0} = 2\left(\sum_{i=1}^{m} \boldsymbol{Y}_i \boldsymbol{Y}_i^{\mathsf{T}}\right)(\boldsymbol{C} - \boldsymbol{A}) - \sum_{i=1}^{m}\left(\boldsymbol{Y}_i^{\mathsf{T}}\boldsymbol{Y}_i\right)\boldsymbol{Y}_i \tag{48}$$

The least-squares center is obtained by solving the previous equation,

$$\boldsymbol{C} = \boldsymbol{A} + \frac{1}{2}\left(\sum_{i=1}^{m} \boldsymbol{Y}\boldsymbol{Y}_i^{\mathsf{T}}\right)^{-1} \sum_{i=1}^{m}\left(\boldsymbol{Y}_i^{\mathsf{T}}\boldsymbol{Y}_i\right)\boldsymbol{Y}_i \tag{49}$$

5.2.1 Pseudocode for the General Case

Listing 8 contains pseudocode for fitting a hypersphere to points. The case $n = 2$ is for circles and the case $n = 3$ is for spheres.

Listing 8. Fitting a hypersphere to points using least squares based on squared differences of squared lengths and square radius. The algorithm requires inverting the covariance matrix. If the matrix is invertible, the output center and radius are valid and the function returns true. If the matrix is not invertible, the function returns false, and the center and radius are invalid (but set to zero so at least they are initialized).

```
bool FitHypersphere(int numPoints, Vector<n> X[], Vector<n>& center, Real& radius)
{
    // Compute the average of the data points.
    Vector<n> A = Vector<n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        A += X[i];
    }
    A /= numPoints;

    // Compute the covariance matrix M of the Y[i] = X[i]-A and the right-hand side R of the linear
    // system M*(C-A) = R.
    Matrix<n, n> M = Matrix<n, n>::ZERO;
    Vector<n> R = Vector<n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        Vector<n> Y = X[i] - A;
        Matrix<n, n> YYT = OuterProduct(Y, Y);  // Y*Transpose(Y)
        Real YTY = Dot(Y, Y);  // Transpose(Y)*Y
        M += YYT;
        R += YTY * Y;
    }
    R /= 2;

    // Solve the linear system M*(C-A) = R for the center C.  The function 'bool Solve(M, R, S)' tries
    // to solve the linear system M*S = R.  If M is invertible, the function returns true and S is the
    // solution.  If M is not invertible, the function returns false and S is invalid.
    Vector<n> CmA;
    if (Solve(M, R, CmA))
    {
        center = A + CmA;
        Real rsqr = 0;
        for (int i = 0; i < numPoints; ++i)
        {
            Vector<n> delta = X[i] - center;
            rsqr += Dot(delta, delta);
        }
        rsqr /= numPoints;
        radius = sqrt(rsqr);
```

```
        return true;
    }
    else
    {
        center = Vector<n>::ZERO;
        radius = 0;
        return false;
    }
}
```

5.2.2 Pseudocode for Circles

Listing 9 is a specialization of Listing 8 for circles in 2 dimensions. The matrix inversion requires only computing the upper-triangular part of the covariance matrix and uses cofactors for inversion.

Listing 9. Fitting a circle to points using least squares based on squared differences of squared lengths and square radius. The algorithm requires inverting the covariance matrix. If the matrix is invertible, the output center and radius are valid and the function returns true. If the matrix is not invertible, the function returns false, and the center and radius are invalid (but set to zero so at least they are initialized).

```
bool FitCircle(int numPoints, Vector2 X[], Vector2& center, Real& radius)
{
    // Compute the average of the data points.
    Vector2 A = { 0, 0 };
    for (int i = 0; i < numPoints; ++i)
    {
        A += X[i];
    }
    A /= numPoints;

    // Compute the covariance matrix M of the Y[i] = X[i]-A and the right-hand side R of the linear
    // system M*(C-A) = R.
    Real M00 = 0, M01 = 0, M11 = 0;
    Vector2 R = { 0, 0 };
    for (int i = 0; i < numPoints; ++i)
    {
        Vector2 Y = X[i] - A;
        Real Y0Y0 = Y[0] * Y[0], Y0Y1 = Y[0] * Y[1], Y1Y1 = Y[1] * Y[1];
        M00 += Y0Y0;  M01 += Y0Y1;  M11 += Y1Y1;
        R += (Y0Y0 + Y1Y1) * Y;
    }
    R /= 2;

    // Solve the linear system M*(C-A) = R for the center C.
    Real det = M00 * M11 - M01 * M01;
    if (det != 0)
    {
        center[0] = A[0] + (M11 * R[0] - M01 * R[1]) / det;
        center[1] = A[1] + (M00 * R[1] - M01 * R[0]) / det;
        Real rsqr = 0;
        for (int i = 0; i < numPoints; ++i)
        {
            Vector2 delta = X[i] - center;
            rsqr += Dot(delta, delta);
        }
        rsqr /= numPoints;
        radius = sqrt(rsqr);
        return true;
    }
    else
    {
        center = { 0, 0 };
```

```
        radius = 0;
        return false;
    }
}
```

5.2.3 Pseudocode for Spheres

Listing 10 is a specialization of Listing 8 for spheres in 3 dimensions. The matrix inversion requires only computing the upper-triangular part of the covariance matrix and uses cofactors for inversion.

Listing 10. Fitting a sphere to points using least squares based on squared differences of squared lengths and square radius. The algorithm requires inverting the covariance matrix. If the matrix is invertible, the output center and radius are valid and the function returns true. If the matrix is not invertible, the function returns false, and the center and radius are invalid (but set to zero so at least they are initialized).

```
bool FitSphere(int numPoints, Vector3 X[], Vector3& center, Real& radius)
{
    // Compute the average of the data points.
    Vector3 A = { 0, 0, 0 };
    for (int i = 0; i < numPoints; ++i)
    {
        A += X[i];
    }
    A /= numPoints;

    // Compute the covariance matrix M of the Y[i] = X[i]-A and the right-hand side R of the linear
    // system M*(C-A) = R.
    Real M00 = 0, M01 = 0, M02 = 0, M11 = 0, M12 = 0, M22 = 0;
    Vector3 R = { 0, 0, 0 };
    for (int i = 0; i < numPoints; ++i)
    {
        Vector2 Y = X[i] - A;
        Real Y0Y0 = Y[0] * Y[0], Y0Y1 = Y[0] * Y[1], Y0Y2 = Y[0] * Y[2];
        Real Y1Y1 = Y[1] * Y[1], Y1Y2 = Y[1] * Y[2], Y2Y2 = Y[2] * Y[2];
        M00 += Y0Y0;  M01 += Y0Y1;  M02 += Y0Y2;
        M11 += Y1Y1;  M12 += Y1Y2;  M22 += Y2Y2;
        R += (Y0Y0 + Y1Y1 + Y2Y2) * Y;
    }
    R /= 2;

    // Solve the linear system M*(C-A) = R for the center C.
    Real cof00 = M11 * M22 - M12 * M12;
    Real cof01 = M02 * M12 - M01 * M22;
    Real cof02 = M01 * M12 - M02 * M11;
    Real det = M00 * cof00 + M01 * cof01 + M02 * cof02;
    if (det != 0)
    {
        Real cof11 = M00 * M22 - M02 * M02;
        Real cof12 = M01 * M02 - M00 * M12;
        Real cof22 = M00 * M11 - M01 * M01;
        center[0] = A[0] + (cof00 * R[0] + cof01 * R[1] + cof02 * R[2]) / det;
        center[1] = A[1] + (cof01 * R[0] + cof11 * R[1] + cof12 * R[2]) / det;
        center[2] = A[2] + (cof02 * R[0] + cof12 * R[1] + cof22 * R[2]) / det;
        Real rsqr = 0;
        for (int i = 0; i < numPoints; ++i)
        {
            Vector3 delta = X[i] - center;
            rsqr += Dot(delta, delta);
        }
        rsqr /= numPoints;
        radius = sqrt(rsqr);
        return true;
```

```
    }
    else
    {
        center = { 0, 0, 0 };
        radius = 0;
        return false;
    }
}
```

5.3 Fitting the Coefficients of a Quadratic Equation

The general quadratic equation that represents a hypersphere in n dimensions is

$$b_0 + \boldsymbol{b}_1 \cdot \boldsymbol{X} + b_2 |\boldsymbol{X}|^2 = 0 \tag{50}$$

where b_0 and $b_2 \neq 0$ are scalar constants and $\boldsymbol{b}_1$ is an $n \times 1$ vector of scalars. Define

$$\boldsymbol{b} = \begin{bmatrix} b_0 \\ \boldsymbol{b}_1 \\ b_2 \end{bmatrix}, \quad \boldsymbol{V} = \begin{bmatrix} 1 \\ \boldsymbol{X} \\ |\boldsymbol{X}|^2 \end{bmatrix} \tag{51}$$

which are both $(n+2) \times 1$ vectors. The quadratic equation is $\boldsymbol{b} \cdot \boldsymbol{V} = 0$. Because $\boldsymbol{b}$ is not zero, we can remove a degree of freedom by requiring $|\boldsymbol{b}| = 1$.

Given samples $\{\boldsymbol{X}_i\}_{i=1}^m$, we can estimate the constants using a least-squares algorithm where the error function is

$$E(\boldsymbol{b}) = \sum_{i=1}^{m} (\boldsymbol{b} \cdot \boldsymbol{V}_i)^2 = \boldsymbol{b}^{\mathsf{T}} \left(\sum_{i=1}^{m} \boldsymbol{V}_i \boldsymbol{V}_i^{\mathsf{T}} \right) \boldsymbol{b} = \boldsymbol{b}^{\mathsf{T}} V \boldsymbol{b} \tag{52}$$

where as a tuple, $\boldsymbol{V}_i = (1, \boldsymbol{X}_i, |\boldsymbol{X}_i|^2)$, and where $V = \sum_{i=1}^m \boldsymbol{V}_i \boldsymbol{V}_i^{\mathsf{T}}$ is a positive semidefinite matrix (symmetric, eigenvalues are nonnegative). The error function is minimized by a unit-length eigenvector $\boldsymbol{b}$ that corresponds to the minimum eigenvalue of V. The equation (50) is factored into

$$\left| \boldsymbol{X} + \frac{\boldsymbol{b}_1}{2b_2} \right|^2 = \left| \frac{\boldsymbol{b}_1}{2b_2} \right|^2 - \frac{b_0}{b_2} \tag{53}$$

from which we see that the hypersphere center is $\boldsymbol{C} = -\boldsymbol{b}_1/(2b_2)$ and the radius is $r = \sqrt{(|\boldsymbol{b}_1|^2 - 4b_0 b_2)/(4b_2^2)}$.

As is typical of these types of problems, it is better to subtract translate the samples by their average to obtain numerical robustness when computing with floating-point arithmetic. Define $\boldsymbol{A} = (\sum_{i=1}^m \boldsymbol{X}_i)$ and $\boldsymbol{Y}_i = \boldsymbol{X}_i - \boldsymbol{A}$. Equation (50) becomes

$$f_0 + \boldsymbol{f}_1 \cdot \boldsymbol{Y} + f_2 |\boldsymbol{Y}|^2 = 0 \tag{54}$$

where

$$\boldsymbol{b} = \begin{bmatrix} b_0 \\ \boldsymbol{b}_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} f_0 - \boldsymbol{f}_1 \cdot \boldsymbol{A} + f_2 |\boldsymbol{A}|^2 \\ \boldsymbol{f}_1 - 2f_2 \boldsymbol{A} \\ f_2 \end{bmatrix} = \begin{bmatrix} 1 & -\boldsymbol{A}^{\mathsf{T}} & |\boldsymbol{A}|^2 \\ \boldsymbol{0} & I & -2\boldsymbol{A} \\ 0 & \boldsymbol{0}^{\mathsf{T}} & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ \boldsymbol{f}_1 \\ f_2 \end{bmatrix} = M \boldsymbol{f} \tag{55}$$

with I the $n \times n$ identity matrix and $\mathbf{0}$ the $n \times 1$ zero vector. The last equality defines the $(n+2) \times (n+2)$ upper-triangular matrix M and the $(n+2) \times 1$ vector $\boldsymbol{f}$. The error function is now

$$E(\boldsymbol{f}) = \boldsymbol{b}^{\mathsf{T}} V \boldsymbol{b} = (R\boldsymbol{f})^{\mathsf{T}} V (R\boldsymbol{f}) = \boldsymbol{f}^{\mathsf{T}} \left(R^{\mathsf{T}} V R\right) \boldsymbol{f} = \boldsymbol{f}^{\mathsf{T}} W \boldsymbol{f} \tag{56}$$

where the last equation defines the positive semidefinite matrix W. Observe that

$$W = R^{\mathsf{T}} \left(\sum_{i=1}^{m} \boldsymbol{V}_i \boldsymbol{V}_i^{\mathsf{T}} \right) R = \sum_{i=1}^{m} \left(R^{\mathsf{T}} \boldsymbol{V}_i\right) \left(R^{\mathsf{T}} \boldsymbol{V}_i\right)^{\mathsf{T}} = \sum_{i=1}^{m} \boldsymbol{W}_i \boldsymbol{W}_i^{\mathsf{T}} \tag{57}$$

where

$$\boldsymbol{W}_i = R^{\mathsf{T}} \boldsymbol{V}_i = \begin{bmatrix} 1 & \mathbf{0}^{\mathsf{T}} & 0 \\ -\boldsymbol{A} & I & \mathbf{0} \\ |\boldsymbol{A}|^2 & -2\boldsymbol{A}^{\mathsf{T}} & 1 \end{bmatrix} \begin{bmatrix} 1 \\ \boldsymbol{X}_i \\ |\boldsymbol{X}_i|^2 \end{bmatrix} = \begin{bmatrix} 1 \\ \boldsymbol{X}_i - \boldsymbol{A} \\ |\boldsymbol{X}_i - \boldsymbol{A}|^2 \end{bmatrix} \tag{58}$$

Therefore, we can subtract $\boldsymbol{A}$ from the samples, compute the matrix W, extract its eigenvector $\boldsymbol{f}$ corresponding to its minimum eigenvalue, compute $\boldsymbol{b} = R\boldsymbol{f}$ and then the center and radius using equation (53). The matrix W written as a block matrix *and* divided by the number of points m to keep the numerical intermediate values on the order of the sample values is

$$W = \begin{bmatrix} 1 & \mathbf{0}^{\mathsf{T}} & \frac{1}{m}\sum_{i=1}^{m} |\boldsymbol{Y}_i|^2 \\ \mathbf{0} & \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{Y}_i \boldsymbol{Y}_i^{\mathsf{T}} & \frac{1}{m}\sum_{i=1}^{m} |\boldsymbol{Y}_i|^2 \boldsymbol{Y}_i \\ \frac{1}{m}\sum_{i=1}^{m} |\boldsymbol{Y}_i|^2 & \frac{1}{m}\sum_{i=1}^{m} |\boldsymbol{Y}_i|^2 \boldsymbol{Y}_i & \frac{1}{m}|\boldsymbol{Y}_i|^4 \end{bmatrix} \tag{59}$$

5.3.1 Pseudocode for the General Case

Listing 11 contains pseudocode for fitting a hypersphere to points. The case $n = 2$ is for circles and the case $n = 3$ is for spheres.

Listing 11. Fitting a hypersphere to points using least squares to fit the coefficients of a quadratic equation defining the hypersphere. The algorithm requires computing the squared radius in terms of the components of an eigenvector. For samples not really distributed well on a hypersphere, the purported squared radius might be negative. The function returns true when that computation is nonnegative or false when it is negative.

```
bool FitHypersphere(int numPoints, Vector<n> X[], Vector<n>& center, Real& radius)
{
    // Compute the average of the data points and the squared length of the average.
    Vector<n> A = Vector<n>::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        A += X[i];
    }
    A /= numPoints;
    Real sqrLenA = Dot(A, A);

    // Compute the components of W.  Block(r, c, rsize, csize) is an accessor to the block
    // whose upper-left location is (r, c) and whose size is rsize-by-csize.  The indices
    // r and c are zero-based.
    Matrix<n + 2, n + 2> W = Matrix<n + 2, n + 2>::ZERO;
```

```
    for (int i = 0; i < numPoints; ++i)
    {
        Vector<n> Y = X[i] - A;
        Matrix<n, n> YYT = OuterProduct(Y, Y);  // Y*Transpose(Y)
        Real YTY = Dot(Y, Y);  // Transpose(Y)*Y
        Vector YTYY = YTY * Y;
        Real YTYYTY = YTY * YTY;
        W.Block(0, 0, 1, 1) += 1;
        W.Block(0, n + 1, 1, 1) += YTY;
        W.Block(1, 1, n, n) += YYT;
        W.Block(1, n + 1, n, 1) += YTYY;
        W.Block(n + 1, 0, 1, 1) += YTY;
        W.Block(n + 1, 1, 1, n) += YTYY;
        W.Block(n + 1, n + 1, 1, 1) += YTYYTY;
    }
    W /= numPoints;

    // Compute the eigenvalues and eigenvectors of M, where the eigenvalues are sorted
    // in nondecreasing order (eigenvalues[0] <= eigenvalues[1] <= ...).
    Real eigenvalues[n + 2];
    Vector<n> eigenvectors[n + 2];
    SolveEigensystem(W, eigenvalues, eigenvectors);

    // Block(i, isize) is an accessor to the block whose initial location is i and whose
    // size is isize.
    Real f0 = eigenvectors[0].Block(0, 1);
    Vector<n> f1 = eigenvectors[0].Block(1, n);
    Real f2 = eigenvectors[0].Block(n + 1, 1);
    Real b0 = f0 - Dot(f1, A) + f2 * sqrLenA;
    Vector<n> b1 = f1 - 2 * f2 * A;
    Real b2 = f2;

    if (b2 != 0)
    {
        Real discr = Dot(b1, b1) - 4 * b0 * b2;
        if (discr >= 0)
        {
            center = -b1 / (2 * b2);
            radius = sqrt(discr / (4 * b2 * b2));
            return true;
        }
    }

    center = Vector<n>::ZERO;
    radius = 0;
    return false;
}
```

5.3.2 Pseudocode for Circles

Listing 12 contains pseudocode for fitting a circle to points.

Listing 12. Fitting a circle to points using least squares to fit the coefficients of a quadratic equation defining the hypersphere. The algorithm requires computing the squared radius in terms of the components of an eigenvector. For samples not really distributed well on a hypersphere, the purported squared radius might be negative. The function returns true when that computation is nonnegative or false when it is negative.

```
bool FitCircle(int numPoints, Vector2 X[], Vector2& center, Real& radius)
{
    // Compute the average of the data points and the squared length of the average.
    Vector2 A = { 0, 0 };
    for (int i = 0; i < numPoints; ++i)
```

```
    {
        A += X[i];
    }
    A /= numPoints;
    Real sqrLenA = Dot(A, A);

    // Compute the upper-triangular components of W.
    Matrix4x4 W = Matrix4x4::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        Vector2 Y = X[i] - A;
        Real Y0Y0 = Y[0] * Y[0], Y0Y1 = Y[0] * Y[1], Y1Y1 = Y[1] * Y[1];
        Real RR = Y0Y0 + Y1Y1, RRRR = RR * RR;
        Real Y0RR = Y0 * RR, Y1RR = Y1 * RR;
        W(0, 3) += RR;
        W(1, 1) += Y0Y0;
        W(1, 2) += Y0Y1;
        W(1, 3) += Y0RR;
        W(2, 2) += Y1Y1;
        W(2, 3) += Y1RR;
        W(3, 3) += RRRR;
    }
    W /= numPoints;
    W(0, 0) = 1;

    // Fill in the lower-triangular components of W.
    W(3, 0) = W(0, 3);
    W(2, 1) = W(1, 2);
    W(3, 1) = W(1, 3);
    W(3, 2) = W(2, 3);

    // Compute the eigenvalues and eigenvectors of M, where the eigenvalues are sorted
    // in nondecreasing order
    //   eigenvalues[0] <= eigenvalues[1] <= eigenvalues[2] <= eigenvalues[3]
    Real eigenvalues[4];
    Vector4 eigenvectors[4];
    SolveEigensystem(W, eigenvalues, eigenvectors);

    Real f0 = eigenvectors[0][0];
    Vector2 f1 = { eigenvectors[0][1], eigenvectors[0][2] };
    Real f2 = eigenvectors[0][3];
    Real b0 = f0 - Dot(f1, A) + f2 * sqrLenA;
    Vector2 b1 = f1 - 2 * f2 * A;
    Real b2 = f2;

    if (b2 != 0)
    {
        Real discr = Dot(b1, b1) - 4 * b0 * b2;
        if (discr >= 0)
        {
            center = -b1 / (2 * b2);
            radius = sqrt(discr / (4 * b2 * b2));
            return true;
        }
    }

    center = Vector2::ZERO;
    radius = 0;
    return false;
}
```

5.3.3 Pseudocode for Spheres

Listing 13 contains pseudocode for fitting a sphere to points.

Listing 13. Fitting a sphere to points using least squares to fit the coefficients of a quadratic equation defining the hypersphere. The algorithm requires computing the squared radius in terms of the components of an eigenvector. For samples not really distributed well on a hypersphere, the purported squared radius might be negative. The function returns true when that computation is nonnegative or false when it is negative.

```
bool FitSphere(int numPoints, Vector3 X[], Vector3& center, Real& radius)
{
    // Compute the average of the data points and the squared length of the average.
    Vector3 A = { 0, 0, 0 };
    for (int i = 0; i < numPoints; ++i)
    {
        A += X[i];
    }
    A /= numPoints;
    Real sqrLenA = Dot(A, A);

    // Compute the upper-triangular components of W.
    Matrix5x5 W = Matrix5x5::ZERO;
    for (int i = 0; i < numPoints; ++i)
    {
        Vector3 Y = X[i] - A;
        Real Y0Y0 = Y[0] * Y[0], Y0Y1 = Y[0] * Y[1], Y1Y1 = Y[1] * Y[1];
        Real Y1Y1 = Y[1] * Y[1], Y1Y2 = Y[1] * Y[2], Y2Y2 = Y[2] * Y[2];
        Real RR = Y0Y0 + Y1Y1 + Y2Y2, RRRR = RR * RR;
        Real Y0RR = Y0 * RR, Y1RR = Y1 * RR, Y2RR = Y2 * RR;
        W(0, 4) += RR;
        W(1, 1) += Y0Y0;
        W(1, 2) += Y0Y1;
        W(1, 3) += Y0Y2;
        W(1, 4) += Y0RR;
        W(2, 2) += Y1Y1;
        W(2, 3) += Y1Y2;
        W(2, 4) += Y1RR;
        W(3, 3) += Y2Y2;
        W(3, 4) += Y2RR;
        W(4, 4) += RRRR;
    }
    W /= numPoints;
    W(0, 0) = 1;

    // Fill in the lower-triangular components of W.
    W(4, 0) = W(0, 4);
    W(2, 1) = W(1, 2);
    W(3, 1) = W(1, 3);
    W(4, 1) = W(1, 4);
    W(3, 2) = W(2, 3);
    W(4, 2) = W(2, 4);
    W(4, 3) = W(3, 4);

    // Compute the eigenvalues and eigenvectors of M, where the eigenvalues are sorted
    // in nondecreasing order
    //   eigenvalues[0] <= eigenvalues[1] <= eigenvalues[2] <= eigenvalues[3] <= eigenvalues[4]
    Real eigenvalues[5];
    Vector5 eigenvectors[5];
    SolveEigensystem(W, eigenvalues, eigenvectors);

    Real f0 = eigenvectors[0][0];
    Vector3 f1 = { eigenvectors[0][1], eigenvectors[0][2], eigenvectors[0][3] };
    Real f2 = eigenvectors[0][4];
    Real b0 = f0 - Dot(f1, A) + f2 * sqrLenA;
    Vector3 b1 = f1 - 2 * f2 * A;
    Real b2 = f2;

    if (b2 != 0)
    {
        Real discr = Dot(b1, b1) - 4 * b0 * b2;
        if (discr >= 0)
        {
            center = -b1 / (2 * b2);
```

```
            radius = sqrt(discr / (4 * b2 * b2));
            return true;
        }
    }

    center = Vector3::ZERO;
    radius = 0;
    return false;
}
```

---

6 Fitting a Hyperellipsoid to Points

The classic cases are fitting 2-dimensional points by ellipses and 3-dimensional points by ellipsoids. In n-dimensions, the objects are called hyperellipsoids, defined implicitly by the quadratic equation

$$(\boldsymbol{X}-\boldsymbol{C})^{\mathsf{T}} S(\boldsymbol{X}-\boldsymbol{C}) = 1 \tag{60}$$

The center of the hyperellipsoid is the $n \times 1$ vector $\boldsymbol{C}$, the $n \times n$ matrix S is positive definite, and the $n \times 1$ vector $\boldsymbol{X}$ is any point on the hyperellipsoid. The matrix S can be factored using an eigendecomposition, $S = RDR^{\mathsf{T}}$, where R is an $n \times n$ rotation matrix whose $n \times 1$ columns are unit-length vectors $\boldsymbol{U}_i$ for $0 \le i < n$ that form a right-handed orthonormal basis. The $n \times n$ matrix D is diagonal with diagonal elements $d_i = 1/e_i^2$ for $0 \le i < n$. The extent $e_i > 0$ is the distance from $\boldsymbol{C}$ to the two extreme points of the hyperellipsoid in the direction $\boldsymbol{U}_i$. The matrix S can be expressed as

$$S = \sum_{i=0}^{n-1} \frac{\boldsymbol{U}_i \boldsymbol{U}_i^{\mathsf{T}}}{e_i^2} = \sum_{i=0}^{n-1} \frac{\boldsymbol{V}_i \boldsymbol{V}_i^{\mathsf{T}}}{e_i^2 |\boldsymbol{V}_i|^2} \tag{61}$$

The second equality uses vectors $\boldsymbol{V}_i$ that are not necessarily unit length, which is useful when robust computations require arbitrary precision arithmetic for exact results. The length of $\boldsymbol{V}_i$ is not necessary to compute; rather, we compute $|\boldsymbol{V}_i|^2 = \boldsymbol{V}_i \cdot \boldsymbol{V}_i$ to eliminate floating-point rounding errors that occur when normalizing $\boldsymbol{V}_i$ to obtain $\boldsymbol{U}_i$.

Let the samples be $\{\boldsymbol{X}_i\}_{i=1}^m$. A nonlinear least-squares error function used for the fitting is

$$E(\boldsymbol{C}, S) = \frac{1}{m} \sum_{i=1}^{m} \left[(\boldsymbol{X}_i - \boldsymbol{C})^{\mathsf{T}} S (\boldsymbol{X}_i - \boldsymbol{C}) - 1 \right]^2 \tag{62}$$

An iterative algorithm is presented for locating the parameters $\boldsymbol{C}$ and S. It is based on gradient descent, which requires derivative computations. The symmetric matrix S has $n(n+1)/2$ parameters consisting of the upper-triangular part of the matrix. The vector $\boldsymbol{C}$ has n parameters.

Define $\boldsymbol{\Delta}_i = \boldsymbol{X}_i - \boldsymbol{C}$. The first-order derivative of E with respect to the center is

$$\frac{\partial E}{\partial \boldsymbol{C}} = \frac{-4}{m} \sum_{i=1}^{m} \left(\boldsymbol{\Delta}_i^{\mathsf{T}} S \boldsymbol{\Delta}_i - 1 \right) S \boldsymbol{\Delta}_i \tag{63}$$

The first-order derivative of E with respect to the component s_{rc} of S is

$$\frac{\partial E}{\partial s_{rc}} = \begin{cases} \frac{2}{m} \sum_{i=1}^{m} \left(\boldsymbol{\Delta}_i^{\mathsf{T}} S \boldsymbol{\Delta}_i - 1 \right) \boldsymbol{\Delta}_i^{\mathsf{T}} B_{rr} \boldsymbol{\Delta}_i, & r = c \\ \frac{2}{m} \sum_{i=1}^{m} \left(\boldsymbol{\Delta}_i^{\mathsf{T}} S \boldsymbol{\Delta}_i - 1 \right) \boldsymbol{\Delta}_i^{\mathsf{T}} \left(B_{rc} + B_{cr} \right) \boldsymbol{\Delta}_i, & r \ne c \end{cases} \tag{64}$$

where B_{rc} is the matrix whose elements are all 0 except for the entry in row r and column c which is 1.

The algorithm uses a 2-Step gradient step whereby the center and matrix are alternately updated.

6.1 Updating the Estimate of the Center

Given initial guesses $\boldsymbol{C}$ and S for the hyperellipsoid, the center can be updated to $\boldsymbol{C}' = \boldsymbol{C} + t\boldsymbol{N}$ for $t \geq 0$, where $\boldsymbol{N} = -\partial E/\partial \boldsymbol{C}$ is the negative of the gradient of E in the $\boldsymbol{C}$ direction at the initial guess. This is one step of the gradient descent algorithm, searching in the direction of largest decrease of E in the $\boldsymbol{C}$ parameter. Define the function

$$\begin{aligned} g(t) &= E(\boldsymbol{C} + t\boldsymbol{N}, S) \\ &= \tfrac{1}{m}\textstyle\sum_{i=1}^{m}\left[(t\boldsymbol{N} - \boldsymbol{\Delta}_i)^{\mathsf{T}} S\,(t\boldsymbol{N} - \boldsymbol{\Delta}_i) - 1\right]^2 \\ &= \tfrac{1}{n}\textstyle\sum_{i=1}^{m}\left[\left(\boldsymbol{N}^{\mathsf{T}} S\boldsymbol{N}\right)t^2 - \left(2\boldsymbol{N}^{\mathsf{T}} S\boldsymbol{\Delta}_i\right)t + \left(\boldsymbol{\Delta}_i^{\mathsf{T}} S\boldsymbol{\Delta}_i - 1\right)\right]^2 \\ &= \tfrac{1}{n}\textstyle\sum_{i=1}^{m}\left(ct^2 - 2b_i t + a_i\right)^2 \end{aligned} \tag{65}$$

where $c = \boldsymbol{N}^{\mathsf{T}} S\boldsymbol{N}$, $b_i = \boldsymbol{N}^{\mathsf{T}} S\boldsymbol{\Delta}_i$ and $a_i = \boldsymbol{\Delta}_i^{\mathsf{T}} S\boldsymbol{\Delta}_i - 1$. Some algebra will show that

$$g(t) = g_0 + g_1 t + g_2 t^2 + g_3 t^3 + g_4 t^4 \tag{66}$$

where

$$g_0 = \frac{1}{m}\sum_{i=1}^{m} a_i^2,\;\; g_1 = \frac{-4}{m}\sum_{i=1}^{m} a_i b_i,\;\; g_2 = \frac{4}{m}\sum_{i=1}^{m} b_i^2 + \frac{2c}{m}\sum_{i=1}^{m} a_i,\;\; g_3 = \frac{-4c}{m}\sum_{i=1}^{m} b_i,\;\; g_4 = c^2 \tag{67}$$

The derivative is $g'(t) = g_1 + 2g_2 t + 3g_3 t^2 + 4g_4 t^3$.

Observe that $g(t) \geq 0$ because it is the nonnegative error function evaluated along a line in the domain. In practice, one never expects the minimum of E to be zero, so $g(t) > 0$. In particular, $g(0) > 0$. The search is in the largest direction of decrease for E along any line through the initial guess, which implies $g'(0) \leq 0$. If $g'(0) = 0$ with $g(0)$ a local minimum, the line search terminates. If $g'(0) < 0$, notice that the coefficient $g_4 > 0$, so $g'(\infty) = \infty$. These conditions guarantee that $g'(t)$ has at least one root for $t > 0$. Of all such roots, let T be the root for which $g(T)$ is the smallest value. It must be the case that $g(T) < g(0)$, so in fact $E(\boldsymbol{C}, S) < E(\boldsymbol{C} + T\boldsymbol{N})$ and the updated estimate for the center is $\boldsymbol{C}' = \boldsymbol{C} + T\boldsymbol{N}$.

6.2 Updating the Estimate of the Matrix

Given initial guesses $\boldsymbol{C}$ and S for the hyperellipsoid, the matrix can be updated to $S' = S + tN$ for $t \geq 0$, where $N = -\partial E/\partial S$ is the negative of the gradient of E in the S direction at the initial guess. This is another step of the gradient descent algorithm, searching in the direction of largest decrease of E in the S parameter. Define the function

$$\begin{aligned} h(t) &= E(\boldsymbol{C}, S + tN) \\ &= \tfrac{1}{m}\textstyle\sum_{i=1}^{m}\left[\boldsymbol{\Delta}_i^{\mathsf{T}}(S + tN)\boldsymbol{\Delta}_i - 1\right]^2 \\ &= \tfrac{1}{m}\textstyle\sum_{i=1}^{m}\left[\left(\boldsymbol{\Delta}_i^{\mathsf{T}} N\boldsymbol{\Delta}_i\right)t + \left(\boldsymbol{\Delta}_i^{\mathsf{T}} S\boldsymbol{\Delta}_i - 1\right)\right]^2 \\ &= \tfrac{1}{m}\textstyle\sum_{i=1}^{m}\left(b_i t + a_i\right)^2 \end{aligned} \tag{68}$$

where $b_i = \mathbf{\Delta}_i^{\mathsf{T}} N \mathbf{\Delta}_i$ and $a_i = \mathbf{\Delta}_i^{\mathsf{T}} S \mathbf{\Delta}_i - 1$. Some algebra will show that

$$h(t) = h_0 + h_1 t + h_2 t^2 \tag{69}$$

where

$$h_0 = \frac{1}{m}\sum_{i=1}^{m} a_i^2, \;\; h_1 = \frac{2}{m}\sum_{i=1}^{m} a_i b_i, \;\; h_2 = \frac{1}{m}\sum_{i=1}^{m} b_i^2 \tag{70}$$

The derivative is $h'(t) = h_1 + 2h_2 t$.

Observe that $h(t) \geq 0$ because it is the nonnegative error function evaluated along a line in the domain. In practice, one never expects the minimum of E to be zero, so $h(t) > 0$. In particular, $h(0) > 0$. The search is in the largest direction of decrease for E along any line through the initial guess, which implies $h'(0) \leq 0$. If $h'(0) = 0$ with $h(0)$ a local minimum, the line search terminates. If $h'(0) < 0$, notice that the coefficient $h_2 > 0$, so $h'(\infty) = \infty$. These conditions guarantee that $h'(t)$ has at least one root for $t > 0$. Of all such roots, let T be the root for which $h(T)$ is the smallest value. It must be the case that $h(T) < h(0)$.

We need to ensure that $S + TN$ is positive definite. If it is positive definite, the updated estimate for the matrix is $S' = S + TN$. If it is not positive definite, the root is halved and $S' = S + (T/2)N$ is tested for positive definiteness. If it is still not positive definite, the halving is repeated until a power $p > 0$ is obtained for which $S' = S + (T/2^p)N$ is positive definite.

The test for positive definiteness uses Sylvester's criterion which says that S is positive definite if all of its leading principal minors are positive. The minors are determinants of the upper-left 1×1 block, the upper-left 2×2 block, and so on through the determinant of S itself. For $n = 2$, S is positive definite when

$$\det\begin{bmatrix} s_{00} \end{bmatrix} > 0, \;\; \det\begin{bmatrix} s_{00} & s_{01} \\ s_{01} & s_{11} \end{bmatrix} > 0 \tag{71}$$

For $n = 3$, S is positive definite when

$$\det\begin{bmatrix} s_{00} \end{bmatrix} > 0, \;\; \det\begin{bmatrix} s_{00} & s_{01} \\ s_{01} & s_{11} \end{bmatrix} > 0, \;\; \det\begin{bmatrix} s_{00} & s_{01} & s_{02} \\ s_{01} & s_{11} & s_{12} \\ s_{02} & s_{12} & s_{22} \end{bmatrix} > 0 \tag{72}$$

6.3 Pseudocode for the Algorithm

The initial guesses for $\boldsymbol{C}$ and S can be provided by the caller or they can be estimated internally. In the Geometric Tools implementation, the internal algorithm is to compute an oriented bounding box for the input points and use the box center as the initial $\boldsymbol{C}$. The initial S is compute from the box axes and box extents using equation (61). Listing 14 contains pseudocode for the algorithm.

Listing 14. Pseudocode for fitting a hyperellipsoid to points. The number of iterations is specified by the caller. The function can be called multiple times, the first time allowing the internal estimate of the initial center and matrix. A subsequent call uses the hyperellipsoid from the previous call. The function returns the error function value that can be monitored by the caller to decide whether or not to make subsequent calls.

```
template <size_t N, typename Real>
Real DoFit(vector<Vector<N, Real>> const& points, size_t numIterations,
    bool useHyperellipsoidForInitialGuess, Hyperellipsoid<N, Real>& hyperellipsoid)
{
    Vector<N, Real> C;
    Matrix<N, N, Real> A;  // the zero matrix
    if (useHyperellipsoidForInitialGuess)
    {
        C = hyperellipsoid.center;
        for (size_t i = 0; i < N; ++i)
        {
            Vector<N, Real> product = OuterProduct(hyperellipsoid.axis[i], hyperellipsoid.axis[i]);
            A += product / (hyperellipsoid.extent[i] * hyperellipsoid.extent[i]);
        }
    }
    else
    {
        OrientedBox<N, Real> box = GetBoundingBox(points);
        C = box.center;
        for (size_t i = 0; i < N; ++i)
        {
            Vector<N, Real> product = OuterProduct(box.axis[i], box.axis[i]);
            A += product / (box.extent[i] * box.extent[i]);
        }
    }

    Real error = ErrorFunction(points, C, A);
    for (size_t i = 0; i < numIterations; ++i)
    {
        error = UpdateMatrix(points, C, A);
        error = UpdateCenter(points, A, C);
    }

    // Extract the hyperellipsoid axes and extents.
    std::array<Real, N> eigenvalue;
    std::array<Vector<N, Real>, N> eigenvector;
    DoEigendecomposition(A, eigenvalue, eigenvector);

    hyperellipsoid.center = C;
    for (size_t i = 0; i < N; ++i)
    {
        hyperellipsoid.axis[i] = eigenvector[i];
        hyperellipsoid.extent[i] = 1 / sqrt(eigenvalue[i]);
    }

    return error;
}
```

---

7 Fitting a Cylinder to 3D Points

This document describes an algorithm for fitting a set of 3D points with a cylinder. The assumption is that the underlying data is modeled by a cylinder and that errors have caused the points not to be exactly on the cylinder. You could very well try to fit a random set of points, but the algorithm is not guaranteed to produce a meaningful solution.

7.1 Representation of a Cylinder

An infinite cylinder is specified by an axis containing a point $\boldsymbol{C}$ and having unit-length direction $\boldsymbol{W}$. The radius of the cylinder is $r > 0$. Two more unit-length vectors $\boldsymbol{U}$ and $\boldsymbol{V}$ may be defined so that $\{\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}\}$

is a right-handed orthonormal set; that is, all vectors are unit-length, mutually perpendicular, and with $\boldsymbol{U} \times \boldsymbol{V} = \boldsymbol{W}$, $\boldsymbol{V} \times \boldsymbol{W} = \boldsymbol{U}$, and $\boldsymbol{W} \times \boldsymbol{U} = \boldsymbol{V}$. Any point $\boldsymbol{X}$ may be written uniquely as

$$\boldsymbol{X} = \boldsymbol{C} + y_0\boldsymbol{U} + y_1\boldsymbol{V} + y_2\boldsymbol{W} = \boldsymbol{C} + R\boldsymbol{Y} \tag{73}$$

where R is a rotation matrix whose columns are $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{W}$ and where $\boldsymbol{Y}$ is a column vector whose rows are y_0, y_1, and y_2. To be on the cylinder, we need

$$\begin{aligned} r^2 &= y_0^2 + y_1^2 \\ &= (\boldsymbol{U} \cdot (\boldsymbol{X} - \boldsymbol{C}))^2 + (\boldsymbol{V} \cdot (\boldsymbol{X} - \boldsymbol{C}))^2 \\ &= (\boldsymbol{X} - \boldsymbol{C})^{\mathsf{T}}(\boldsymbol{U}\boldsymbol{U}^{\mathsf{T}} + \boldsymbol{V}\boldsymbol{V}^{\mathsf{T}})(\boldsymbol{X} - \boldsymbol{C}) \\ &= (\boldsymbol{X} - \boldsymbol{C})^{\mathsf{T}}(I - \boldsymbol{W}\boldsymbol{W}^{\mathsf{T}})(\boldsymbol{X} - \boldsymbol{C}) \end{aligned} \tag{74}$$

where I is the identity matrix. Because the unit-length vectors form an orthonormal set, it is necessary that $I = \boldsymbol{U}\boldsymbol{U}^{\mathsf{T}} + \boldsymbol{V}\boldsymbol{V}^{\mathsf{T}} + \boldsymbol{W}\boldsymbol{W}^{\mathsf{T}}$. A finite cylinder is obtained by bounding the points in the axis direction,

$$|y_2| = |\boldsymbol{W} \cdot (\boldsymbol{X} - \boldsymbol{C})| \le h/2 \tag{75}$$

where $h > 0$ is the height of the cylinder.

7.2 The Least-Squares Error Function

Let $\{\boldsymbol{X}_i\}_{i=1}^n$ be the input point set. An error function for a cylinder fit based on Equation (74) is

$$E(r^2, \boldsymbol{C}, \boldsymbol{W}) = \sum_{i=1}^{n} \left[F(\boldsymbol{X}_i; r^2, \boldsymbol{C}, \boldsymbol{W})\right]^2 = \sum_{i=1}^{n} \left[(\boldsymbol{X}_i - \boldsymbol{C})^{\mathsf{T}} \left(I - \boldsymbol{W}\boldsymbol{W}^{\mathsf{T}}\right)(\boldsymbol{X}_i - \boldsymbol{C}) - r^2\right]^2 \tag{76}$$

where the cylinder axis is a line containing point $\boldsymbol{C}$ and having unit-length direction $\boldsymbol{W}$ and the cylinder radius is r. Thus, the error function involves 6 parameters: 1 for the squared radius r^2, 3 for the point $\boldsymbol{C}$, and 2 for the unit-length direction $\boldsymbol{W}$. These parameters form the 6-tuple $\boldsymbol{q}$ in the generic discussion presented previously.

For numerical robustness, it is advisable to subtract the sample mean $\boldsymbol{A} = (\sum_{i=1}^n \boldsymbol{X}_i)/n$ from the samples, $\boldsymbol{X}_i \leftarrow \boldsymbol{X}_i - \boldsymbol{A}$. This preconditioning is assumed in the mathematical derivations to follow, in which case $\sum_{i=1}^n \boldsymbol{X}_i = \boldsymbol{0}$.

In the following discussion, define

$$P = I - \boldsymbol{W}\boldsymbol{W}^{\mathsf{T}}, \;\; r_i^2 = (\boldsymbol{C} - \boldsymbol{X}_i)^{\mathsf{T}} P (\boldsymbol{C} - \boldsymbol{X}_i) \tag{77}$$

The matrix P represents a projection onto a plane with normal $\boldsymbol{W}$, so $P^2 = P$ and depends only on the direction $\boldsymbol{W}$. The term r_i^2 depends on the center $\boldsymbol{C}$ and the direction $\boldsymbol{W}$. The error function is written concisely as $E = \sum_{i=1}^n (r_i^2 - r^2)^2$.

7.3 An Equation for the Radius

The partial derivative of the error function with respect to the squared radius is $\partial E/\partial r^2 = -2\sum_{i=1}^{n}(r_i^2 - r^2)$. Setting this to zero, we have the constraint

$$0 = \sum_{i=1}^{n}(r_i^2 - r^2) \tag{78}$$

which leads to

$$r^2 = \frac{1}{n}\sum_{i=1}^{n} r_i^2 \tag{79}$$

Thus, the squared radius is the average of the squared distances of the projections of $\boldsymbol{X}_i - \boldsymbol{C}$ onto a plane containing $\boldsymbol{C}$ and having normal $\boldsymbol{W}$. The right-hand side depends on the parameters $\boldsymbol{C}$ and $\boldsymbol{W}$.

Observe that

$$\begin{aligned}
r_i^2 - r^2 &= r_i^2 - \tfrac{1}{n}\textstyle\sum_{j=1}^{n} r_j^2 \\
&= (\boldsymbol{C} - \boldsymbol{X}_i)^{\mathsf{T}} P(\boldsymbol{C} - \boldsymbol{X}_i) - \tfrac{1}{n}\textstyle\sum_{j=1}^{n}(\boldsymbol{C} - \boldsymbol{X}_j)^{\mathsf{T}} P(\boldsymbol{C} - \boldsymbol{X}_j) \\
&= \boldsymbol{C}^{\mathsf{T}} P\boldsymbol{C} - 2\boldsymbol{X}_i^{\mathsf{T}} P\boldsymbol{C} + \boldsymbol{X}_i^{\mathsf{T}} P\boldsymbol{X}_i - \tfrac{1}{n}\textstyle\sum_{j=1}^{n}\left(\boldsymbol{C}^{\mathsf{T}} P\boldsymbol{C} - 2\boldsymbol{X}_j^{\mathsf{T}} P\boldsymbol{C} + \boldsymbol{X}_j^{\mathsf{T}} P\boldsymbol{X}_j\right) \\
&= \boldsymbol{C}^{\mathsf{T}} P\boldsymbol{C} - 2\boldsymbol{X}_i^{\mathsf{T}} P\boldsymbol{C} + \boldsymbol{X}_i^{\mathsf{T}} P\boldsymbol{X}_i - \boldsymbol{C}^{\mathsf{T}} P\boldsymbol{C} + \tfrac{2}{n}\left(\textstyle\sum_{j=1}^{n}\boldsymbol{X}_j^{\mathsf{T}}\right) P\boldsymbol{C} - \tfrac{1}{n}\textstyle\sum_{j=1}^{n}\boldsymbol{X}_j^{\mathsf{T}} P\boldsymbol{X}_j \\
&= \left(\tfrac{1}{n}\textstyle\sum_{j=1}^{n}\boldsymbol{X}_j^{\mathsf{T}} - \boldsymbol{X}_i^{\mathsf{T}}\right) 2P\boldsymbol{C} + \left(\boldsymbol{X}_i^{\mathsf{T}} P\boldsymbol{X}_i - \tfrac{1}{n}\textstyle\sum_{j=1}^{n}\boldsymbol{X}_j^{\mathsf{T}} P\boldsymbol{X}_j\right) \\
&= -\boldsymbol{X}_i^{\mathsf{T}} 2P\boldsymbol{C} + \left(\boldsymbol{X}_i^{\mathsf{T}} P\boldsymbol{X}_i - \tfrac{1}{n}\textstyle\sum_{j=1}^{n}\boldsymbol{X}_j^{\mathsf{T}} P\boldsymbol{X}_j\right)
\end{aligned} \tag{80}$$

The last equality is based on the precondition $\sum_{j=1}^{n}\boldsymbol{X}_j = \boldsymbol{0}$.

7.4 An Equation for the Center

The partial derivative with respect to the center is $\partial E/\partial \boldsymbol{C} = -4\sum_{i=1}^{n}(r_i^2 - r^2)P(\boldsymbol{X}_i - \boldsymbol{C})$. Setting this to zero, we have the constraint

$$\begin{aligned}
0 &= \textstyle\sum_{i=1}^{n}(r_i^2 - r^2)P(\boldsymbol{X}_i - \boldsymbol{C}) \\
&= \textstyle\sum_{i=1}^{n}(r_i^2 - r^2)P\boldsymbol{X}_i - \left[\sum_{i=1}^{n}(r_i^2 - r^2)\right] P\boldsymbol{C} \\
&= \textstyle\sum_{i=1}^{n}(r_i^2 - r^2)P\boldsymbol{X}_i
\end{aligned} \tag{81}$$

where the last equality is a consequence of Equation (78). Multiply equation (80) by $P\boldsymbol{X}_i$, sum over i, and use equation (81) to obtain

$$\begin{aligned}
0 &= -2P\left(\textstyle\sum_{i=1}^{n}\boldsymbol{X}_i\boldsymbol{X}_i^{\mathsf{T}}\right) P\boldsymbol{C} + \textstyle\sum_{i=1}^{n}\left(\boldsymbol{X}_i^{\mathsf{T}} P\boldsymbol{X}_i\right) P\boldsymbol{X}_i - \left(\tfrac{1}{n}\textstyle\sum_{j=1}^{n}\boldsymbol{X}_j^{\mathsf{T}} P\boldsymbol{X}_j\right) P\textstyle\sum_{i=1}^{n}\boldsymbol{X}_i \\
&= -2P\left(\textstyle\sum_{i=1}^{n}\boldsymbol{X}_i\boldsymbol{X}_i^{\mathsf{T}}\right) P\boldsymbol{C} + \textstyle\sum_{i=1}^{n}\left(\boldsymbol{X}_i^{\mathsf{T}} P\boldsymbol{X}_i\right) P\boldsymbol{X}_i
\end{aligned} \tag{82}$$

where the last equality is based on the precondition $\sum_{i=1}^{n}\boldsymbol{X}_i = \boldsymbol{0}$. We wish to solve this equation for $\boldsymbol{C}$, but observe that $\boldsymbol{C} + t\boldsymbol{W}$ are valid centers for all t. It is sufficient to compute a center that has no component

in the $\boldsymbol{W}$-direction; that is, we may construct a point for which $\boldsymbol{C} = P\boldsymbol{C}$. It suffices to solve Equation (82) for $P\boldsymbol{C}$ written as the linear system $A(P\boldsymbol{C}) = \boldsymbol{B}/2$ where

$$A = P\left(\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{X}_i\boldsymbol{X}_i^{\mathsf{T}}\right)P, \quad \boldsymbol{B} = \frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{X}_i^{\mathsf{T}}P\boldsymbol{X}_i\right)P\boldsymbol{X}_i \tag{83}$$

The projection matrix is symmetric, $P = P^{\mathsf{T}}$, a condition that leads to the right-hand side of the equation defining A. We have used $P = P^2$ to introduce an additional P factor, $\boldsymbol{X}_i^{\mathsf{T}}P\boldsymbol{X}_i = \boldsymbol{X}_i^{\mathsf{T}}P^2\boldsymbol{X}_i = \boldsymbol{X}_i^{\mathsf{T}}P^{\mathsf{T}}P\boldsymbol{X}_i$, which leads to the right-hand side of the equation defining $\boldsymbol{B}$.

The matrix A is singular because the projection matrix P is singular, so we cannot directly invert A to solve the equation. The linear system involves terms that live only in the plane perpendicular to $\boldsymbol{W}$, so in fact the linear system reduces to two equations in two unknowns in the projection space and is solvable as long as the coefficient matrix is invertible.

Choose $\boldsymbol{U}$ and $\boldsymbol{V}$ so that $\{\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}\}$ is a right-handed orthonormal set; then $P\boldsymbol{X}_i = \mu_i\boldsymbol{U} + \nu_i\boldsymbol{V}$ and $P\boldsymbol{C} = k_0\boldsymbol{U} + k_1\boldsymbol{V}$, where $\mu_i = \boldsymbol{U}\cdot\boldsymbol{X}_i$, $\nu_i = \boldsymbol{V}\cdot\boldsymbol{X}_i$, $k_0 = \boldsymbol{U}\cdot P\boldsymbol{C}$, and $k_1 = \boldsymbol{V}\cdot P\boldsymbol{C}$. The matrix A becomes

$$A = \left(\frac{1}{n}\sum_{i=1}^{n}\mu_i^2\right)\boldsymbol{U}\boldsymbol{U}^{\mathsf{T}} + \left(\frac{1}{n}\sum_{i=1}^{n}\mu_i\nu_i\right)\left(\boldsymbol{U}\boldsymbol{V}^{\mathsf{T}} + \boldsymbol{V}\boldsymbol{U}^{\mathsf{T}}\right) + \left(\frac{1}{n}\sum_{i=1}^{n}\nu_i^2\right)\boldsymbol{V}\boldsymbol{V}^{\mathsf{T}} \tag{84}$$

and the vector $\boldsymbol{B}$ becomes

$$\boldsymbol{B} = \frac{1}{n}\sum_{i=1}^{n}(\mu_i^2 + \nu_i^2)(\mu_i\boldsymbol{U} + \nu_i\boldsymbol{V}) \tag{85}$$

The vector $A(P\boldsymbol{C})$ becomes

$$A(P\boldsymbol{C}) = \left(\frac{k_0}{n}\sum_{i=1}^{n}\mu_i^2 + \frac{k_1}{n}\sum_{i=1}^{n}\mu_i\nu_i\right)\boldsymbol{U} + \left(\frac{k_0}{n}\sum_{i=1}^{n}\mu_i\nu_i + \frac{k_1}{n}\sum_{i=1}^{n}\nu_i^2\right)\boldsymbol{V} \tag{86}$$

Equating this to $\boldsymbol{B}/2$ and grouping the coefficients for $\boldsymbol{U}$ and $\boldsymbol{V}$ leads to the linear system

$$\begin{bmatrix} \frac{1}{n}\sum_{i=1}^{n}\mu_i^2 & \frac{1}{n}\sum_{i=1}^{n}\mu_i\nu_i \\ \frac{1}{n}\sum_{i=1}^{n}\mu_i\nu_i & \frac{1}{n}\sum_{i=1}^{n}\nu_i^2 \end{bmatrix}\begin{bmatrix} k_0 \\ k_1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} \frac{1}{n}\sum_{i=1}^{n}(\mu_i^2+\nu_i^2)\mu_i \\ \frac{1}{n}\sum_{i=1}^{n}(\mu_i^2+\nu_i^2)\nu_i \end{bmatrix} \tag{87}$$

The coefficient matrix is the covariance matrix of the projection of the samples onto the plane perpendicular to $\boldsymbol{W}$. Intuitively, this matrix is invertible as long as the projections do not lie on a line. If the matrix is singular (or nearly singular numerically), the original samples lie on a plane (or nearly lie on a plane numerically). They are not fitted well by a cylinder or, if you prefer, they are fitted by a cylinder with infinite radius.

The matrix system of Equation (87) has solution

$$\begin{bmatrix} k_0 \\ k_1 \end{bmatrix} = \frac{1}{2\left(\frac{1}{n}\sum_{i=1}^{n}\mu_i^2\frac{1}{n}\sum_{i=1}^{n}\nu_i^2 - \left(\frac{1}{n}\sum_{i=1}^{n}\mu_i\nu_i\right)^2\right)}\begin{bmatrix} \frac{1}{n}\sum_{i=1}^{n}\nu_i^2 & -\frac{1}{n}\sum_{i=1}^{n}\mu_i\nu_i \\ -\frac{1}{n}\sum_{i=1}^{n}\mu_i\nu_i & \frac{1}{n}\sum_{i=1}^{n}\mu_i^2 \end{bmatrix}\begin{bmatrix} \frac{1}{n}\sum_{i=1}^{n}(\mu_i^2+\nu_i^2)\mu_i \\ \frac{1}{n}\sum_{i=1}^{n}(\mu_i^2+\nu_i^2)\nu_i \end{bmatrix} \tag{88}$$

which produces the cylinder center $P\boldsymbol{C} = k_0\boldsymbol{U} + k_1\boldsymbol{V}$; use this instead of $\boldsymbol{C}$ in Equation (74).

Although the solution appears to depend on the choice of $\boldsymbol{U}$ and $\boldsymbol{V}$, it does not. Let $\boldsymbol{W} = (w_0, w_1, w_2)$ and define the skew symmetric matrix

$$S = \begin{bmatrix} 0 & -w_2 & w_1 \\ w_2 & 0 & -w_0 \\ -w_1 & w_0 & 0 \end{bmatrix} \tag{89}$$

By definition of skew symmetry, $S^{\mathsf{T}} = -S$. This matrix represents the cross product operation: $S\boldsymbol{\xi} = \boldsymbol{W} \times \boldsymbol{\xi}$ for any vector $\boldsymbol{\xi}$. Because $\{\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}\}$ is a right-handed orthonormal set, it follows that $S\boldsymbol{U} = \boldsymbol{V}$ and $S\boldsymbol{V} = -\boldsymbol{U}$. It can be shown also that $S = \boldsymbol{V}\boldsymbol{U}^{\mathsf{T}} - \boldsymbol{U}\boldsymbol{V}^{\mathsf{T}}$. Define matrix $\hat{A}$ by

$$\hat{A} = \left(\frac{1}{n}\sum_{i=1}^{n}\nu_i^2\right)\boldsymbol{U}\boldsymbol{U}^{\mathsf{T}} - \left(\frac{1}{n}\sum_{i=1}^{n}\mu_i\nu_i\right)\left(\boldsymbol{U}\boldsymbol{V}^{\mathsf{T}} + \boldsymbol{V}\boldsymbol{U}^{\mathsf{T}}\right) + \left(\frac{1}{n}\sum_{i=1}^{n}\mu_i^n\right)\boldsymbol{V}\boldsymbol{V}^{\mathsf{T}} = SAS^{\mathsf{T}} \tag{90}$$

Effectively, this generates a 2×2 matrix that is the adjugate of the 2×2 matrix representing A. It has the property

$$\hat{A}A = \delta P, \quad \delta = \sum_{i=1}^{n}\mu_i^2\sum_{i=1}^{n}\nu_i^2 - \left(\sum_{i=1}^{n}\mu_i\nu_i\right)^2 \tag{91}$$

The trace of a matrix is the sum of the diagonal entries. Observe that $\text{Trace}(P) = 2$ because $|\boldsymbol{W}|^2 = 1$. Taking the trace of $\hat{A}A = \delta P$, we obtain $2\delta = \text{Trace}(\hat{A}A)$. The cylinder center is obtained by multiplying $A(P\boldsymbol{C}) = \boldsymbol{B}/2$ by $\hat{A}$, using the definitions in equation (83) and using equation (91),

$$P\boldsymbol{C} = \frac{\hat{A}}{\text{Trace}(\hat{A}A)}\left(\frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{X}_i^{\mathsf{T}}P\boldsymbol{X}_i\right)P\boldsymbol{X}_i\right) = \frac{\hat{A}}{\text{Trace}(\hat{A}A)}\left(\frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{X}_i^{\mathsf{T}}P\boldsymbol{X}_i\right)\boldsymbol{X}_i\right) \tag{92}$$

where the last equality uses $\hat{A}P = \hat{A}$ because $S^{\mathsf{T}}P = S^{\mathsf{T}}$. Equation (92) is independent of $\boldsymbol{U}$ and $\boldsymbol{V}$ but dependent on $\boldsymbol{W}$.

7.5 An Equation for the Direction

Let the direction be parameterized as $\boldsymbol{W}(\boldsymbol{s}) = (w_0(\boldsymbol{s}), w_1(\boldsymbol{s})), w_2(\boldsymbol{s}))$, where $\boldsymbol{s}$ is a 2-dimensional parameter. For example, spherical coordinates is such a parameterization: $\boldsymbol{W} = (\cos s_0 \sin s_1, \sin s_0 \sin s_1, \cos s_1)$ for $s_0 \in [0, 2\pi)$ and $s_1 \in [0, \pi/2]$, where $w_2(s_0, s_1) \geq 0$. The partial derivatives of E are

$$\frac{\partial E}{\partial s_k} = 2\sum_{i=1}^{n}(r_i^2 - r^2)(\boldsymbol{C} - \boldsymbol{X}_i)^{\mathsf{T}}\frac{\partial P}{\partial s_k}(\boldsymbol{C} - \boldsymbol{X}_i) \tag{93}$$

Solving $\partial E/\partial s_k = 0$ in closed form is not tractable. It is possible to generate a system of polynomial equations in the components of $\boldsymbol{W}$, use elimination theory to obtain a polynomial in one variable, and then find its roots. This approach is generally tedious and not robust numerically.

Alternatively, we can skip root finding for $\partial E/\partial s_k = 0$, instead substituting Equations (80) and (92) directly into the error function $E/n = \frac{1}{n}\sum_{i=1}^{n}(r_i^2 - r^2)^2$ to obtain a nonnegative function,

$$G(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left[\boldsymbol{X}_i^{\mathsf{T}}P\boldsymbol{X}_i - \frac{1}{n}\sum_{j=1}^{n}\boldsymbol{X}_j^{\mathsf{T}}P\boldsymbol{X}_j - 2\boldsymbol{X}_i^{\mathsf{T}}\frac{\hat{A}}{\text{Trace}(\hat{A}A)}\left(\frac{1}{n}\sum_{j=1}^{n}\left(\boldsymbol{X}_j^{\mathsf{T}}P\boldsymbol{X}_j\right)\boldsymbol{X}_j\right)\right]^2 \tag{94}$$

A numerical algorithm for locating the minimum of G can be used. Or, as is shown in the sample code, the domain for (s_0, s_1) may be partitioned into samples at which G is evaluated. The sample producing the minimum G-value determines a reasonable direction $\boldsymbol{W}$. The center $\boldsymbol{C}$ and squared radius r^2 are inherent in the evaluation of G, so in the end we have a fitted cylinder. The evaluations of G are expensive for a large number of samples: equation (94) contains summations in a term that is then squared, followed by another summation.

Some algebraic manipulation lead to encapsulating the summations, allowing us to precompute the summations and represent $G(\boldsymbol{W})$ as a rational polynomial of the components of $\boldsymbol{W}$. This approach increases the performance on the CPU. It also allows an efficient implementation for massively parallel performance on the GPU—one GPU thread per direction vector that is sampled from a hemisphere.

The projection matrix P is determined by its upper-triangular elements $\boldsymbol{p} = (p_{00}, p_{01}, p_{02}, p_{11}, p_{12}, p_{22})$. We can write the following, where $\boldsymbol{p}$ is represented as a 6×1 vector,

$$\boldsymbol{X}_i^{\mathsf{T}} P \boldsymbol{X}_i - \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{X}_j^{\mathsf{T}} P \boldsymbol{X}_j = \boldsymbol{p} \cdot (\boldsymbol{\xi}_i - \boldsymbol{\mu}) = \boldsymbol{p} \cdot \boldsymbol{\delta}_i \tag{95}$$

The 6×1 vectors $\boldsymbol{\xi}_i$, $\boldsymbol{\mu}$ and $\boldsymbol{\delta}_i$ are defined next (written as 6-tuples). As a 3-tuple, let $\boldsymbol{X}_i = (x_i, y_i, z_i)$,

$$\boldsymbol{\xi}_i = \left(x_i^2, 2x_iy_i, 2x_iz_i, y_i^2, 2y_iz_i, z_i^2\right), \quad \boldsymbol{\mu} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{\xi}_i, \quad \boldsymbol{\delta}_i = \boldsymbol{\xi}_i - \boldsymbol{\mu} \tag{96}$$

We can also write

$$\frac{1}{n} \sum_{j=1}^{n} \left(\boldsymbol{X}_j^{\mathsf{T}} P \boldsymbol{X}_j\right) \boldsymbol{X}_j = \left(\frac{1}{n} \sum_{j=1}^{n} \boldsymbol{X}_j \boldsymbol{\xi}_j^{\mathsf{T}}\right) \boldsymbol{p} = \left(\frac{1}{n} \sum_{j=1}^{n} \boldsymbol{X}_j \boldsymbol{\delta}_j^{\mathsf{T}}\right) \boldsymbol{p} \tag{97}$$

The last equality is true because $\sum_{j=1}^{n} \boldsymbol{X}_j = \boldsymbol{0}$ implies $\sum_{j=1}^{n} \boldsymbol{X}_j \boldsymbol{\mu}^{\mathsf{T}} = 0$. Define

$$Q = \hat{A} / \operatorname{Trace}(\hat{A}A), \quad F_0 = \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{X}_j \boldsymbol{X}_j^{\mathsf{T}}, \quad F_1 = \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{X}_j \boldsymbol{\delta}_j^{\mathsf{T}}, \quad F_2 = \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{\delta}_j \boldsymbol{\delta}_j^{\mathsf{T}} \tag{98}$$

where Q and F_0 are 3×3, F_1 is 3×6 and F_2 is 6×6; then

$$\begin{aligned} G(\boldsymbol{W}) &= \tfrac{1}{n} \textstyle\sum_{i=1}^{n} \left[\boldsymbol{p}^{\mathsf{T}} \boldsymbol{\delta}_i - 2\boldsymbol{X}_i^{\mathsf{T}} Q F_1 \boldsymbol{p}\right]^2 \\ &= \tfrac{1}{n} \textstyle\sum_{i=1}^{n} \left[\left(\boldsymbol{p}^{\mathsf{T}} \boldsymbol{\delta}_i\right)^2 - 4\left(\boldsymbol{p}^{\mathsf{T}} \boldsymbol{\delta}_i\right)\left(\boldsymbol{X}_i^{\mathsf{T}} Q F_1 \boldsymbol{p}\right) + 4\left(\boldsymbol{X}_i^{\mathsf{T}} Q F_1 \boldsymbol{p}\right)^2\right] \\ &= \boldsymbol{p}^{\mathsf{T}} F_2 \boldsymbol{p} - 4\boldsymbol{p}^{\mathsf{T}} F_1^{\mathsf{T}} Q F_1 \boldsymbol{p} + 4\boldsymbol{p}^{\mathsf{T}} F_1^{\mathsf{T}} Q^{\mathsf{T}} F_0 Q F_1 \boldsymbol{p} \end{aligned} \tag{99}$$

The precomputations for the input samples $\{\boldsymbol{Y}_i\}_{i=1}^{n}$ is the following and must be done so in the order specified. These steps are independent of direction vectors $\boldsymbol{W}$.

1. Compute $\boldsymbol{A} = \frac{1}{n}\left(\sum_{i=1}^{n} \boldsymbol{Y}_i\right)$.
2. Compute $\boldsymbol{X}_i = \boldsymbol{Y}_i - \boldsymbol{A}$ for all i.
3. Compute $\boldsymbol{\mu} = \frac{1}{n} \sum_{i=1}^{n} (x_i^2, 2x_iy_i, 2x_iz_i, y_i^2, 2y_iz_i, z_i^2)$, $\boldsymbol{\delta}_i = (x_i^2, 2x_iy_i, 2x_iz_i, y_i^2, 2y_iz_i, z_i^2) - \boldsymbol{\mu}$ for all i.

4. Compute $F_0 = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{X}_i\boldsymbol{X}_i^{\mathsf{T}}$, $F_1 = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{X}_i\boldsymbol{\delta}_i^{\mathsf{T}}$, $F_2 = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{\delta}_i\boldsymbol{\delta}_i^{\mathsf{T}}$.

For each specified direction $\boldsymbol{W}$, do the following steps.

1. Compute the projection matrix $P = I - \boldsymbol{W}\boldsymbol{W}^{\mathsf{T}}$ and the skew-symmetric matrix S.
2. Compute $A = PF_0P$, $\hat{A} = SAS^{\mathsf{T}}$, $\hat{A}A$, $\text{Trace}(\hat{A}A)$.
3. Compute $Q = \hat{A}/\,\text{Trace}(\hat{A}A)$.
4. Store the upper-triangular entries of P in $\boldsymbol{p}$.
5. Compute $\boldsymbol{\alpha} = F_1\boldsymbol{p}$ and $\boldsymbol{\beta} = Q\boldsymbol{\alpha}$.
6. Compute $G = \boldsymbol{p}^{\mathsf{T}}F_2\boldsymbol{p} - 4\boldsymbol{\alpha}^{\mathsf{T}}\boldsymbol{\beta} + 4\boldsymbol{\beta}^{\mathsf{T}}F_0\boldsymbol{\beta}$.
7. The corresponding center is $P\boldsymbol{C} = \boldsymbol{\beta}$.
8. The corresponding squared radius is $r^2 = \frac{1}{n}\sum_{i=1}^{n} r_i^2$ where $r_i^2 = (P\boldsymbol{C} - P\boldsymbol{X}_i)^{\mathsf{T}}(P\boldsymbol{C} - P\boldsymbol{X}_i)$. This factors to $r^2 = \boldsymbol{p}\cdot\boldsymbol{\mu} + \boldsymbol{\beta}^{\mathsf{T}}\boldsymbol{\beta}$.

The sample application that used equation (94) directly was really slow. On an Intel® Core™ i7-6700 CPU at 3.40 GHz, the single-threaded version for 10765 points required 129 seconds and the multithreaded version using 8 hyperthreads required 22 seconds. The evaluation of G using the precomputed summations is much faster. The single-threaded version required 85 milliseconds and the multithreaded version using 8 hyperthreads required 22 milliseconds.

7.6 Fitting for a Specified Direction

Although one may apply root-finding or minimization techniques to estimate the global minimum of E, as shown previously, in practice it is possible first to obtain a good estimate for the direction $\boldsymbol{W}$. Using this direction, we may solve for $\boldsymbol{C} = P\boldsymbol{C}$ in Equation (92) and then r^2 in Equation (79).

For example, suppose that the $\boldsymbol{X}_i$ are distributed approximately on a section of a cylinder so that the least-squares line that fits the data provides a good estimate for the direction $\boldsymbol{W}$. This vector is a unit-length eigenvector associated with the largest eigenvalue of the covariance matrix $A = \sum_{i=1}^{n} \boldsymbol{X}_i\boldsymbol{X}_i^{\mathsf{T}}$. We may use a numerical eigensolver to obtain $\boldsymbol{W}$, and then solve the aforementioned equations for the cylinder center and squared radius.

The distribution can be such that the estimated direction $\boldsymbol{W}$ from the covariance matrix is not good, as is shown in the experiments of the next section.

7.7 Pseudocode and Experiments

The simplest algorithm to implement involves the minimization of the function G in Equation (94). An implemention is shown in Listing 15.

Listing 15. Pseudocode for preprocessing the sample points. The output X[] is the array of sample points translated by the average. The other outputs are mu for $\boldsymbol{\mu}$, F0 for F_0, F1 for F_1 and F2 for F_2. The pseudocode is algo given for evaluating the function $G(\boldsymbol{W})$ and generating the corresponding $P\boldsymbol{C}$ and r^2.

```
void Preprocess(int n, Vector3 points[n], Vector3 X[n],
    Vector3& average, Vector6& mu, Matrix3x3& F0, Matrix3x6& F1, Matrix6x6&F2)
{
    average = { 0, 0, 0 };
    for (int i = 0; i < n; ++i)
    {
        average += points[i];
    }
    average /= n;
    for (int i = 0; i < n; ++i)
    {
        X[i] = points[i] - average;
    }

    Vector6 zero = { 0, 0, 0, 0, 0, 0 };
    Vector6 products[n];
    MakeZero(products);
    mu = zero;
    for (int i = 0; i < n; ++i)
    {
        products[i][0] = X[i][0] * X[i][0];
        products[i][1] = X[i][0] * X[i][1];
        products[i][2] = X[i][0] * X[i][2];
        products[i][3] = X[i][1] * X[i][1];
        products[i][4] = X[i][1] * X[i][2];
        products[i][5] = X[i][2] * X[i][2];
        mu[0] += products[i][0];
        mu[1] += 2 * products[i][1];
        mu[2] += 2 * products[i][2];
        mu[3] += products[i][3];
        mu[4] += 2 * products[i][4];
        mu[5] += products[i][5];
    }
    mu /= n;

    MakeZero(F0);
    MakeZero(F1);
    MakeZero(F2);
    for (int i = 0; i < n; ++i)
    {
        Vector6 delta;
        delta[0] = products[i][0] - mu[0];
        delta[1] = 2 * products[i][1] - mu[1];
        delta[2] = 2 * products[i][2] - mu[2];
        delta[3] = products[i][3] - mu[3];
        delta[4] = 2 * products[i][4] - mu[4];
        delta[5] = products[i][5] - mu[5];
        F0(0, 0) += products[i][0];
        F0(0, 1) += products[i][1];
        F0(0, 2) += products[i][2];
        F0(1, 1) += products[i][3];
        F0(1, 2) += products[i][4];
        F0(2, 2) += products[i][5];
        F1 += OuterProduct(X[i], delta);
        F2 += OuterProduct(delta, delta);
    }
    F0 /= n;
    F0(1, 0) = F0(0, 1);
    F0(2, 0) = F0(0, 2);
    F0(2, 1) = F0(1, 2);
    F1 /= n;
    F2 /= n;
}

Real G(int n, Vector3 X[n], Vector3 mu, Matrix3x3 F0, Matrix3x6 F1, Matrix6x6 F2, Vector3 W,
    Vector3& PC, Real& rSqr)
{
```

```
    Matrix3x3 P = Matrix3x3::Identity() - OuterProduct(W, W);  // P = I - W * W^T
    // S = {{0, -w2, w1}, {w2, 0, -w0}, {-w1, w0, 0}}, inner braces are rows
    Matrix3x3 S(0, -W[2], W[1], W[2], 0, -W[0], -W[1], W[0], 0);
    Matrix3x3 A = P * F0 * P;
    Matrix3x3 hatA = -(S * A * S);
    Matrix3x3 hatAA = hatA * A;
    Real trace = Trace(hatAA);
    Matrix3x3 Q = hatA / trace;
    Vector6 p = { P(0, 0), P(0, 1), P(0, 2), P(1, 1), P(1, 2), P(2, 2) };
    Vector3 alpha = F1 * p;
    Vector3 beta = Q * alpha;
    Real error = (Dot(p, F2 * p) - 4 * Dot(alpha, beta) + 4 * Dot(beta, F0 * beta)) / n;
    PC = beta;
    rsqr = Dot(p, mu) + Dot(beta, beta);
    return error;
}
```

The fitting is performed by searching a large number of directions $\boldsymbol{W}$, as shown in Listing 16.

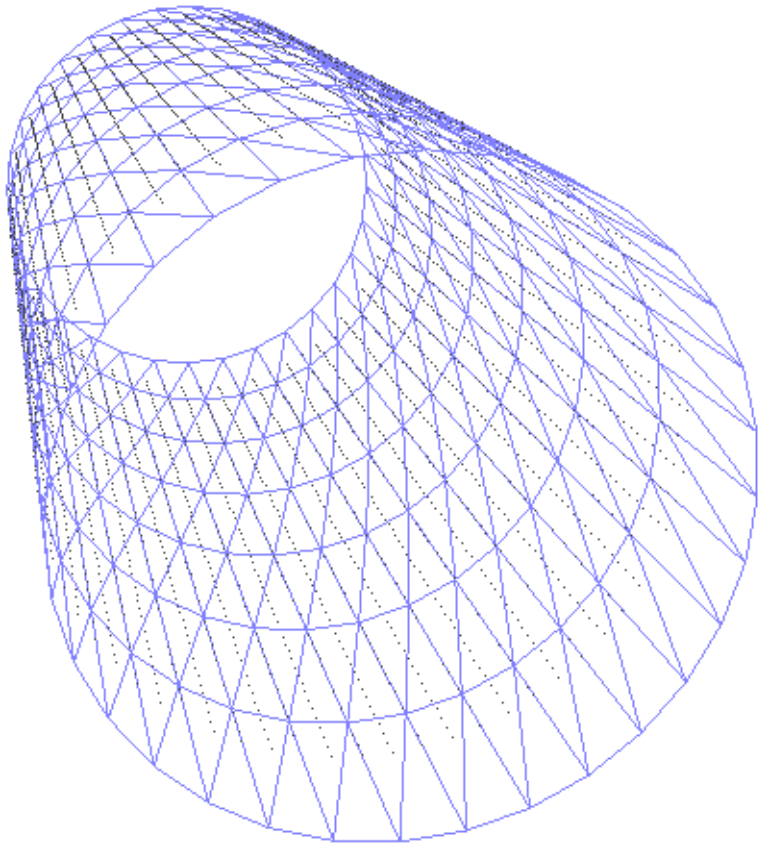
Listing 16. Fitting a cylinder to a set of points.

```
// The X[] are the points to be fit.  The outputs rSqr, C, and W are the
// cylinder parameters.  The function return value is the error function
// evaluated at the cylinder parameters.
Real FitCylinder (int n, Vector3 points[n], Real& rSqr, Vector3& C, Vector3& W)
{
    Vector3 X[n];
    Vector3 mu;
    Matrix3x3 F0;
    Matrix3x6 F1;
    Matrix6x6 F2;
    Preprocess(n, points, X, average, mu, F0, F1, F2);

    // Choose imax and jmax as desired for the level of granularity you
    // want for sampling W vectors on the hemisphere.
    Real minError = infinity;
    W = Vector3::Zero();
    C = Vector3::Zero();
    rSqr = 0;
    for (int j = 0; j <= jmax; ++j)
    {
        Real phi = halfPi * j / jmax;  // in [0,pi/2]
        Real csphi = cos(phi), snphi = sin(phi);
        for (int i = 0; i < imax; ++i)
        {
            Real theta = twoPi * i / imax;  // in [0,2*pi)
            Real cstheta = cos(theta), sntheta = sin(theta);
            Vector3 currentW(cstheta * snphi, sntheta * snphi, csphi);
            Vector3 currentC;
            Real currentRSqr;
            Real error = G(n, X, mu, F0, F1, F2, currentW, currentC, currentRSqr);
            if (error < minError)
            {
                minError = error;
                W = currentW;
                C = currentC;
                rSqr = currentRSqr;
            }
        }
    }

    // Translate the center to the original coordinate system.
    C += average;

    return minError;
}
```

The following experiments show how well the cylinder fitting works.

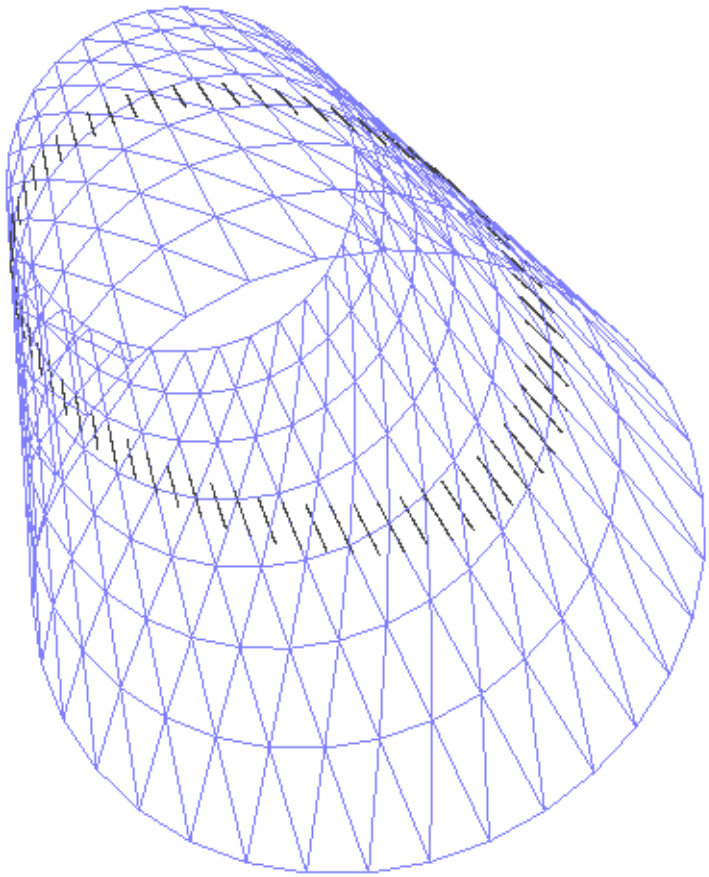
A regular lattice of 64×65 samples were chosen for a cylinder centered at the origin, with axis direction $(0, 0, 1)$, with radius 1, and height 4. The samples are $(\cos\theta_j, \sin\theta_j, z_{ij})$, where $\theta_j = 2\pi j/64$ and $z_{ij} = -2 + 4i/64$ for $0 \le i \le 64$ and $0 \le j < 64$. The fitted center, radius, and axis direction are the actual ones (modulo numerical round-off errors). Figure 1 shows a rendering of the points (in black) and a wire frame of the fitted cylinder (in blue).

Figure 1. Fitting of samples on a cylinder ring that is cut perpendicular to the cylinder axis.



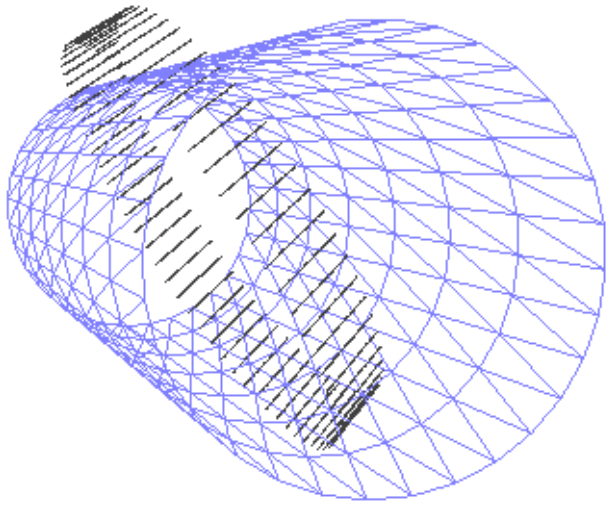
A lattice of samples was chosen for the same cylinder but the samples are skewed to lie on a cut that is not perpendicular to the cylinder axis. The samples are $(\cos\theta_j, \sin\theta_j, z_{ij})$, where $\theta_j = 2\pi j/64$ for $0 \le j < 64$ and $z_{ij} = -b + \cos\theta_j + 2bi/64$ for $b = 1/4$ and $0 \le i \le 64$. The fitted center, radius, and axis direction are the actual ones (modulo numerical round-off errors). Figure 2 shows a rendering of the points (in black) and a wire frame of the fitted cylinder (in blue).

Figure 2. Fitting of samples on a cylinder ring that is cut skewed relative to the cylinder axis.



In this example, if you were to compute the covariance matrix of the samples and choose the cylinder axis direction to be the eigenvector in the direction of maximum variance, that direction is skewed relative to the original cylinder axis. The fitted parameters are (approximately) $\boldsymbol{W} = (0.699, 0, 0.715)$, $P\boldsymbol{C} = (0, 0, 0)$, and $r^2 = 0.511$. Figure 3 shows a rendering of the points (in black) and a wire frame of the fitted cylinder (in blue).

Figure 3. Fitting of samples on a cylinder ring that is cut skewed relative to the cylinder axis. The direction $\boldsymbol{W}$ was chosen to be an eigenvector corresponding to the maximum eigenvalue of the covariance matrix of the samples.



You can see that the fitted cylinder is not a good approximation to the points.

Figure 4 shows a point cloud generated from a DIC file (data set courtesy of Carolina Lavecchia) and the fitted cylinder. The data set has nearly 11000 points.

Figure 4. A view of a fitted point cloud.

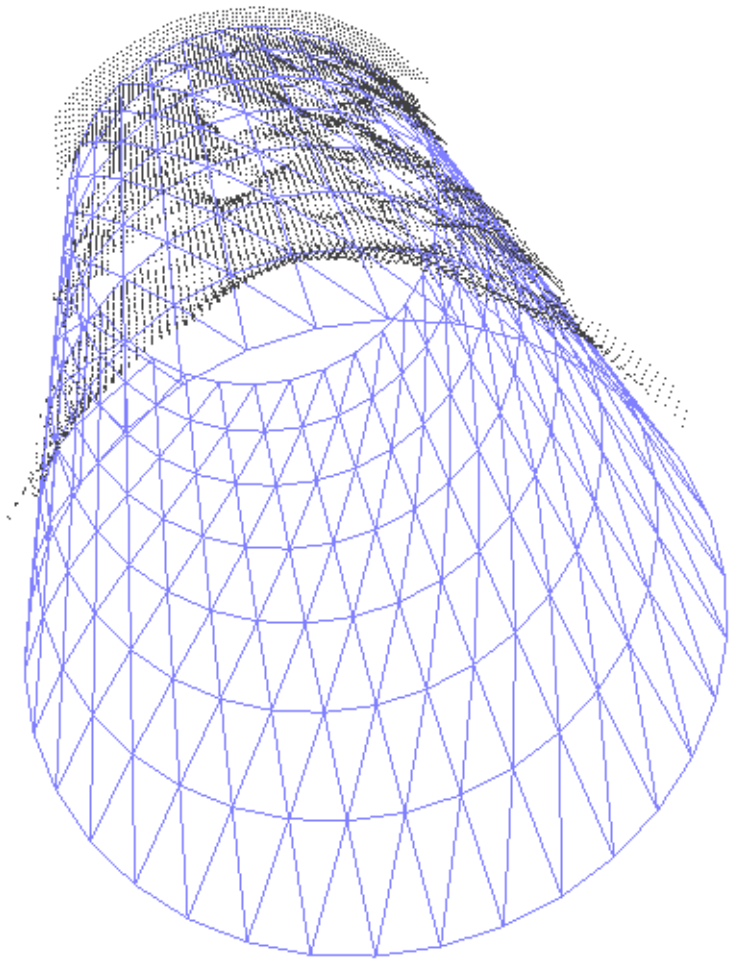
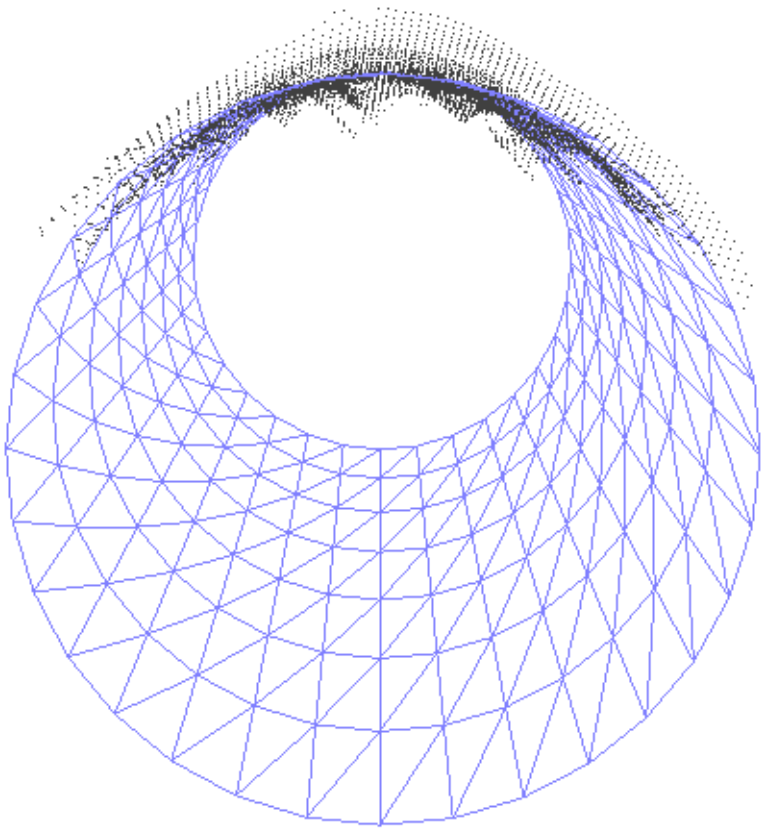


Figure 5 shows a different view of the point cloud and cylinder.

Figure 5. Another view of a fitted point cloud, looking along the top of the cyliner to get an idea of how well the cylinder fits the data.



7.8 Fitting a Cylinder to a Triangle Mesh

Imagine having a triangle mesh that approximates a portion of an infinite cylinder. When the triangles have small area and have normal vectors perpendicular to the cylinder axis, the projection of the triangles onto a plane perpendicular to the cylinder axis covers a region also of small area. If you were to project the cylinder itself onto the plane, you get a circle (not a disk) that covers a region of zero area. This suggests locating a direction $\boldsymbol{D}$ on the hemisphere of points (x, y, z) with $z \geq 0$ for which the sum of areas of projections of the triangles of the mesh is minimized.

Let the mesh have n points $\{\boldsymbol{P}_i\}_{i=0}^{n=1}$ and m triangles, each a triple (i_0, i_1, i_2) of indices into the points. Given a unit-length direction $\boldsymbol{D}$, let $\boldsymbol{U}$ and $\boldsymbol{V}$ be vectors for which $\{\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{D}\}$ is a right-handed orthonormal set. For numerical robustness, compute the average $\boldsymbol{A} = \left(\sum_{i=0}^{n-1} \boldsymbol{P}_i\right)/n$ and replace $\boldsymbol{P}_i \leftarrow (\boldsymbol{P}_i - \boldsymbol{A})$. The points are projected onto a plane perpendicular to $\boldsymbol{D}$; they are $(u_i, v_i) = (\boldsymbol{U} \cdot \boldsymbol{P}_i, \boldsymbol{V} \cdot \boldsymbol{P}_i)$. Given the vertices of a

projected triangle, say (u_0, v_0), (u_1, v_1) and (u_2, v_2), the area of the projected triangle is

$$A = \frac{1}{2} \left| (u_1 - u_0, v_1 - v_0) \cdot (u_2 - u_0, v_2 - v_0)^{\perp} \right| \tag{100}$$

where $(u, v)^{\perp} = (v, -u)$. The areas can be summed over all projected triangles, ignoring the overlap of projections; that is, the sum of areas is generally larger than the area of the union of the projections.

Pseudocode for a simple search over the hemisphere of directions to locate a minimum sum of areas is shown in Listing 17.

Listing 17. A simple search over the hemisphere of directions to locate a direction that minimizes the sum of areas of projected triangles. The factor 1/2 is ignored, so the minimum is over twice the area.

```
struct Cylinder { Vector3 center, direction; Real radius, height; };
struct Circle { Vector2 center; Real radius; };

Cylinder FitMeshByCylinder(int numPoints, Vector3 points[], int numTriangles, int3 triangles[])
{
    // Translate the points so that the average is the origin (for numerical robustness).
    Vector3 localPoints[] = points[];
    Vector3 average = { 0, 0, 0 };
    for (int i = 0; i < numPoints; ++i)
    {
        average += points[i];
    }
    average /= numPoints;
    for (int i = 0; i < numPoints; ++i)
    {
        localPoints = points[i] - average;
    }

    // Locate the direction D that minimizes the sum of the areas of the
    // projected triangles.
    Vector3 D;
    SearchForMinimum(numPoints, localPoints, numTriangles, triangles, D);

    // Project the points onto a plane perpendicular to D. This is the
    // setup for fitting the projected points by a circle.
    Vector3 U, V;
    ComputeOrthonormalBasis(D, U, V);
    Vector2 projections[numPoints];
    Real hmin = infinity, hmax = 0;
    for (int i = 0; i < numPoints; ++i)
    {
        T h = Dot(D, localPoints[i]);
        hmin = min(h, hmin);
        hmax = max(h, hmax);
        projections[i][0] = Dot(U, localPoints[i]);
        projections[i][1] = Dot(V, localPoints[i]);
    }

    // The file ApprCircle2.h has a class with member function FitUsingSquaredLength
    // that can be used for the fitting of the projected points by a circle.
    Circle circle;
    FitPointsByCircle(numPoints, projections, circle);

    // Inverse project the circle to obtain cylinder parameters.
    cylinder.center = average + (circle.center[0] * U + circle.center[1] * V) + ((hmax + hmin) / 2) * D;
    cylinder.direction = D;
    cylinder.radius = circle.radius;
    cylinder.height = hmax - hmin;
}

void SearchForMinimum(int numPoints, Vector3 points[], int numTriangles, int3 triangles[],
    Vector3& minDirection, Real& minMeasure)
```

```
{
    // Handle the north pole (0,0,1) separately.
    minDirection = { 0, 0, 1 };
    Real minMeasure = GetMeasure(minDirection, numPoints, points, numTriangles, triangles);

    // Process a regular grid of (theta,phi) angles.
    for (int j = 1; j <= numPhiSamples; ++j)
    {
        Real phi = (pi / 2) * j / numPhiSamples;  // in [0,pi/2]
        Real csphi = cos(phi), snphi = sin(phi);
        for (int i = 0; i < numThetaSamples; ++i)
        {
            Real theta = 2 * pi * i / numThetaSamples;  // in [0,2*pi)
            Real cstheta = cos(theta), std::sin(theta);
            Vector3 direction = { cstheta * snphi, sntheta * snphi, csphi };
            Real measure = GetMeasure(direction, numPoints, points, numTriangles, triangles);
            if (measure < minMeasure)
            {
                minDirection = direction;
                minMeasure = measure;
            }
        }
    }
}

Real GetMeasure(Vector3 D, int numPoints, Vector3 points[], int numTriangles, int3 triangles[])
{
    Vector3 U, V;
    ComputeOrthonormalBasis(D, U, V);
    Vector2 projections[numPoints];
    for (int i = 0; i < numPoints; ++i)
    {
        projections[i][0] = Dot(U, points[i]);
        projections[i][1] = Dot(V, points[i]);
    }

    // Add up 2*area of the triangles.
    Real measure = 0;
    for (int t = 0; t < numTriangles; ++t)
    {
        Vector2 V[3];
        for (int i = 0; i < 3; ++i)
        {
            V[i] = projections[triangles[t][i]];
        }
        measure += abs(edge10[0] * edge20[1] - edge10[1] * edge20[0]);
    }
    return measure;
}
```

---

8 Fitting a Cone to 3D Points

The cone vertex is $\boldsymbol{V}$, the unit-length axis direction is $\boldsymbol{U}$ and the cone angle is $\theta \in (0, \pi/2)$. The cone is defined algebraically by those points $\boldsymbol{X}$ for which

$$\boldsymbol{U} \cdot \frac{\boldsymbol{X} - \boldsymbol{V}}{|\boldsymbol{X} - \boldsymbol{V}|} = \cos(\theta) \tag{101}$$

This can be written as a quadratic equation

$$(\boldsymbol{X} - \boldsymbol{V})^{\mathsf{T}}(\cos(\theta)^2 I - \boldsymbol{U}\boldsymbol{U}^{\mathsf{T}})(\boldsymbol{X} - \boldsymbol{V}) = 0 \tag{102}$$

with the implicit constraint that $\boldsymbol{U}\cdot(\boldsymbol{X}-\boldsymbol{V}) > 0$; that is, $\boldsymbol{X}$ is on the "positive" cone. Define $\boldsymbol{W} = \boldsymbol{U}/\cos(\theta)$, so $|\boldsymbol{W}| > 1$ and

$$F(\boldsymbol{X};\boldsymbol{V},\boldsymbol{W}) = (\boldsymbol{X}-\boldsymbol{V})^{\mathsf{T}}(I - \boldsymbol{W}\boldsymbol{W}^{\mathsf{T}})(\boldsymbol{X}-\boldsymbol{V}) = 0 \tag{103}$$

The nonlinear least-squares fitting of points $\{\boldsymbol{X}_i\}_{i=0}^{n-1}$ computes $\boldsymbol{V}$ and $\boldsymbol{W}$ to minimize the error function

$$E(\boldsymbol{V},\boldsymbol{W}) = \sum_{i=0}^{n-1} F(\boldsymbol{X}_i;\boldsymbol{V},\boldsymbol{W})^2 \tag{104}$$

This can be solved using standard iterative minimizers such as the Gauss–Newton method and Levenberg–Marquardt method. The partial derivatives of the F terms are needed for $\partial E/\partial \boldsymbol{V}$ and $\partial E/\partial \boldsymbol{W}$ in order to compute the Jacobian matrix,

$$\frac{\partial F}{\partial \boldsymbol{V}} = 2\left(\boldsymbol{\Delta} - (\boldsymbol{W}\cdot\boldsymbol{\Delta})\,\boldsymbol{W}\right), \quad \frac{\partial F}{\partial \boldsymbol{W}} = -2\,(\boldsymbol{W}\cdot\boldsymbol{\Delta})\,\boldsymbol{\Delta} \tag{105}$$

where $\boldsymbol{\Delta} = \boldsymbol{X} - \boldsymbol{V}$. Implementations are found in ApprCone3.h.

8.1 Initial Choice for the Parameters of the Error Function

Without any application-specific knowledge about the sample points, the simplest initial choice for the center, axis directions and axis extents is based on assuming the sample points are dense on a frustum of a cone and then using integrals of various quantities over that frustum. The cone frustum surface is parameterized by

$$\boldsymbol{P}(h,\phi) = \boldsymbol{V} + h\boldsymbol{U} + h\tan\theta(\cos\phi\boldsymbol{W}_0 + \sin\phi\boldsymbol{W}_1), \quad h \in [h_0,h_1], \quad \phi \in [0,2\pi) \tag{106}$$

where $0 \le h_0 < h_1$. The set $\{\boldsymbol{U},\boldsymbol{W}_0,\boldsymbol{W}_1\}$ is a right-handed orthonormal basis for $\mathbb{R}^3$; that is, the vectors are unit length, mutually perpendicular, and $\boldsymbol{U} = \boldsymbol{W}_0 \times \boldsymbol{W}_1$. We want to use the sample points $\boldsymbol{X}_i$ to determine an initial choice for the cone axis direction $\boldsymbol{U}$, the cone angle θ and the height bounds h_0 and h_1 of the cone frustum. The integrals involve ratio expressions that are defined by

$$\rho_n = \frac{\int_{h_0}^{h_1} h^{n-1}\,dh}{\int_{h_0}^{h_1} h\,dh} = \frac{\frac{1}{n}(h_1^n - h_0^n)}{\frac{1}{2}(h_1^2 - h_0^2)} \tag{107}$$

Define the rotation matrix $R = [\boldsymbol{U}\,\boldsymbol{W}_0\,\boldsymbol{W}_1]$ whose columns are the specified basis vectors and define $\boldsymbol{Y}(h,\phi)$ to be the 3×1 vector which in 3-tuple form is $(h, h\tan\theta\cos\phi, h\tan\theta\sin\phi)$. The parameterization of the surface is concisely $\boldsymbol{P}(h,\phi) = \boldsymbol{V} + R\boldsymbol{Y}(h,\phi)$. Several integrals are formulated next and involve the following integrals. For notational convenience, I will use $\boldsymbol{Y}$ with the understanding that it depends on h and ϕ.

The element of surface area for the cone frustum surface is

$$\begin{aligned}
dA &= \left|\tfrac{\partial \mathbf{P}}{\partial h} \times \tfrac{\partial \mathbf{P}}{\partial \phi}\right| dh\,d\phi \\
&= |(\boldsymbol{U} + \tan\theta\,(\cos\phi\boldsymbol{W}_0 + \sin\phi\boldsymbol{W}_1)) \times (h\tan\theta\,(-\sin\phi\boldsymbol{W}_0 + \cos\phi\boldsymbol{W}_1))|\,dh\,d\phi \\
&= (h\tan\theta)\,|-\sin\phi\boldsymbol{U}\times\boldsymbol{W}_0 + \cos\phi\boldsymbol{U}\times\boldsymbol{W}_1 + \tan\theta\boldsymbol{W}_0\times\boldsymbol{W}_1|\,dh\,d\phi \\
&= h\tan\theta\sqrt{\sin^2\phi + \cos^2\phi + \tan^2\theta}\,dh\,d\phi \\
&= h\tan\theta\sqrt{1+\tan^2\theta}\,dh\,d\phi \\
&= h\tan\theta\sqrt{1/\cos^2\theta}\,dh\,d\phi \\
&= (h\tan\theta/\cos\theta)\,dh\,d\phi
\end{aligned} \tag{108}$$

and the surface area of the cone frustum is

$$A = \int_{h_0}^{h_1}\int_0^{2\pi} dA = \int_{h_0}^{h_1}\int_0^{2\pi} (h\tan\theta/\cos\theta)\, dh\, d\phi = \pi\,\frac{\tan\theta}{\cos\theta}(h_1^2 - h_0^2) \tag{109}$$

The average of $\boldsymbol{Y}$ over the cone frustum surface is

$$\begin{aligned}
\bar{\boldsymbol{Y}} &= \tfrac{1}{A}\textstyle\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{Y}\, dA \\
&= \tfrac{1}{\pi(h_1^2-h_0^2)}\textstyle\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{Y}\, h\, dh\, d\phi \\
&= \tfrac{1}{\pi(h_1^2-h_0^2)}\textstyle\int_{h_0}^{h_1}\int_0^{2\pi} \begin{bmatrix} 1 \\ \tan\theta\cos\phi \\ \tan\theta\sin\phi \end{bmatrix} h^2\, dh\, d\phi \\
&= \rho_3 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}
\end{aligned} \tag{110}$$

The average of $\boldsymbol{Y}^\mathsf{T}\boldsymbol{Y}$ over the cone frustum surface is

$$\begin{aligned}
\overline{\boldsymbol{Y}^\mathsf{T}\boldsymbol{Y}} &= \tfrac{1}{A}\textstyle\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{Y}^\mathsf{T}\boldsymbol{Y}\, dA \\
&= \tfrac{1}{\pi(h_1^2-h_0^2)}\textstyle\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{Y}^\mathsf{T}\boldsymbol{Y}\, h\, dh\, d\phi \\
&= \tfrac{1}{\pi(h_1^2-h_0^2)}\textstyle\int_{h_0}^{h_1}\int_0^{2\pi} (1+\tan^2\theta)\, h^3\, dh\, d\phi \\
&= \rho_4 \sec^2\theta
\end{aligned} \tag{111}$$

The average of $\boldsymbol{Y}\boldsymbol{Y}^\mathsf{T}$ over the cone frustum surface is

$$\begin{aligned}
\overline{\boldsymbol{Y}\boldsymbol{Y}^\mathsf{T}} &= \tfrac{1}{A}\textstyle\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{Y}\boldsymbol{Y}^\mathsf{T}\, dA \\
&= \tfrac{1}{\pi(h_1^2-h_0^2)}\textstyle\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{Y}\boldsymbol{Y}^\mathsf{T}\, h\, dh\, d\phi \\
&= \tfrac{1}{\pi(h_1^2-h_0^2)}\textstyle\int_{h_0}^{h_1}\int_0^{2\pi} \begin{bmatrix} 1 & \tan\theta\cos\phi & \tan\theta\sin\phi \\ \tan\theta\cos\phi & \tan^2\theta\cos^2\phi & \tan^2\theta\sin\phi\cos\phi \\ \tan\theta\sin\phi & \tan^2\theta\sin\phi\cos\phi & \tan^2\theta\sin^2\phi \end{bmatrix} h^3\, dh\, d\phi \\
&= \rho_4\,\mathrm{Diag}\left(1, \tfrac{1}{2}\tan^2\theta, \tfrac{1}{2}\tan^2\theta\right)
\end{aligned} \tag{112}$$

The average of $\boldsymbol{Y}\boldsymbol{Y}^{\mathsf{T}}\boldsymbol{Y}$ over the cone frustum surface is

$$\begin{aligned}
\overline{\boldsymbol{Y}\boldsymbol{Y}^{\mathsf{T}}\boldsymbol{Y}} &= \tfrac{1}{A}\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{Y}\boldsymbol{Y}^{\mathsf{T}}\boldsymbol{Y}\, dA \\
&= \tfrac{1}{\pi(h_1^2-h_0^2)}\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{Y}\boldsymbol{Y}^{\mathsf{T}}\boldsymbol{Y}\, h\, dh\, d\phi \\
&= \tfrac{1}{\pi(h_1^2-h_0^2)}\int_{h_0}^{h_1}\int_0^{2\pi} (1+\tan^2\theta) \begin{bmatrix} 1 \\ \tan\theta\cos\phi \\ \tan\theta\sin\phi \end{bmatrix} h^4\, dh\, d\phi \\
&= \rho_5 \sec^2\theta \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}
\end{aligned} \tag{113}$$

In the following, $\boldsymbol{P}$ is the surface parameterization with the understanding that it depends on h and ϕ. The average of the points over the cone frustum surface is

$$\begin{aligned}
\boldsymbol{C} &= \tfrac{1}{A}\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{P}\, dA \\
&= \tfrac{1}{\pi(h_1^2-h_0^2)}\int_{h_0}^{h_1}\int_0^{2\pi} \left(h\boldsymbol{V} + h^2\boldsymbol{U} + h^2\tan\theta\left(\cos\phi\boldsymbol{W}_0 + \sin\phi\boldsymbol{W}_1\right)\right) dh\, d\phi \\
&= \boldsymbol{V} + \rho_3\boldsymbol{U}
\end{aligned} \tag{114}$$

Define the difference $\boldsymbol{\Delta} = \boldsymbol{V} - \boldsymbol{C} = -\rho_3\boldsymbol{U}$, in which case $\boldsymbol{P} - \boldsymbol{C} = \boldsymbol{\Delta} + R\boldsymbol{Y}$. Define $\boldsymbol{Z} = \boldsymbol{P} - \boldsymbol{C}$ and define $\boldsymbol{i}$ be the 3×1 vector which as a 3-tuple is $(1,0,0)$. Observe that $R\boldsymbol{i} = \boldsymbol{U}$. The average of $\boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}$ over the cone frustum surface is

$$\begin{aligned}
\overline{\boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}} &= \tfrac{1}{A}\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}\, dA \\
&= \tfrac{1}{A}\int_{h_0}^{h_1}\int_0^{2\pi} \left(\boldsymbol{\Delta}\boldsymbol{\Delta}^{\mathsf{T}} + \boldsymbol{\Delta}\boldsymbol{Y}^{\mathsf{T}}R^{\mathsf{T}} + R\boldsymbol{Y}\boldsymbol{\Delta}^{\mathsf{T}} + R\boldsymbol{Y}\boldsymbol{Y}^{\mathsf{T}}R^{\mathsf{T}}\right) dA \\
&= \boldsymbol{\Delta}\boldsymbol{\Delta}^{\mathsf{T}} + \boldsymbol{\Delta}\overline{\boldsymbol{Y}}^{\mathsf{T}}R^{\mathsf{T}} + R\overline{\boldsymbol{Y}}\boldsymbol{\Delta}^{\mathsf{T}} + R\overline{\boldsymbol{Y}\boldsymbol{Y}^{\mathsf{T}}}R^{\mathsf{T}} \\
&= (-\rho_3\boldsymbol{U})(-\rho_3\boldsymbol{U})^{\mathsf{T}} + (-\rho_3\boldsymbol{U})(\rho_3\boldsymbol{i})^{\mathsf{T}}R^{\mathsf{T}} + R(\rho_3\boldsymbol{i})(-\rho_3\boldsymbol{U})^{\mathsf{T}} \\
&\quad + R(\rho_4\,\mathrm{Diag}\left(1, \tfrac{1}{2}\tan^2\theta, \tfrac{1}{2}\tan^2\theta\right))R^{\mathsf{T}} \\
&= (\rho_4 - \rho_3^2)\boldsymbol{U}\boldsymbol{U}^{\mathsf{T}} + \tfrac{1}{2}\rho_4\tan^2\theta(\boldsymbol{W}_0\boldsymbol{W}_0^{\mathsf{T}} + \boldsymbol{W}_1\boldsymbol{W}_1^{\mathsf{T}}) \\
&= R\,\mathrm{Diag}\left(\rho_4 - \rho_3^2, \tfrac{1}{2}\rho_4\tan^2\theta, \tfrac{1}{2}\rho_4\tan^2\theta\right)R^{\mathsf{T}}
\end{aligned} \tag{115}$$

The matrix $\overline{\boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}}$ is the covariance matrix of the cone frustum surface points and its eigendecomposition is

given by the right-hand side of equation (115). The average of $\overline{\boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}\boldsymbol{Z}}$ over the cone frustum surface is

$$\begin{aligned}
\overline{\boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}\boldsymbol{Z}} &= \tfrac{1}{A}\int_{h_0}^{h_1}\int_0^{2\pi} \boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}\boldsymbol{Z}\,dA \\
&= \tfrac{1}{A}\int_{h_0}^{h_1}\int_0^{2\pi} \Big((\boldsymbol{\Delta}^{\mathsf{T}}\boldsymbol{\Delta})\boldsymbol{\Delta} + (2\boldsymbol{\Delta}\boldsymbol{\Delta}^{\mathsf{T}}R)\boldsymbol{Y} + (\boldsymbol{Y}^{\mathsf{T}}\boldsymbol{Y})\boldsymbol{\Delta}(\boldsymbol{\Delta}^{\mathsf{T}}\boldsymbol{\Delta})R\boldsymbol{Y} + \\
&\quad + (2R\boldsymbol{Y}\boldsymbol{Y}^{\mathsf{T}}R^{\mathsf{T}})\boldsymbol{\Delta} + R\boldsymbol{Y}\boldsymbol{Y}^{\mathsf{T}}\boldsymbol{Y}\Big) \\
&= (\boldsymbol{\Delta}^{\mathsf{T}}\boldsymbol{\Delta})\boldsymbol{\Delta} + (2\boldsymbol{\Delta}\boldsymbol{\Delta}^{\mathsf{T}}R)\overline{\boldsymbol{Y}} + \overline{\boldsymbol{Y}^{\mathsf{T}}\boldsymbol{Y}}\boldsymbol{\Delta} + (\boldsymbol{\Delta}^{\mathsf{T}}\boldsymbol{\Delta})R\overline{\boldsymbol{Y}} + 2R\overline{\boldsymbol{Y}\boldsymbol{Y}^{\mathsf{T}}}R^{\mathsf{T}}\boldsymbol{\Delta} + R\overline{\boldsymbol{Y}\boldsymbol{Y}^{\mathsf{T}}\boldsymbol{Y}} \\
&= (-\rho_3^3 + 2\rho_3^3 - \rho_3\rho_4\sec^2\theta + \rho_3^3 - 2\rho_3\rho_4 + \rho_5\sec^2\theta)\boldsymbol{U} \\
&= (2p_3(\rho_3^2 - \rho_4) + (\rho_5 - \rho_3\rho_4)\sec^2\theta)\boldsymbol{U}
\end{aligned} \tag{116}$$

Equations (115) and (116) can be manipulated to obtain 3 equations in the 3 unknowns h_0, h_1 and $\tan\theta$. In this sense, if we have good estimates for $\boldsymbol{C}$, $\overline{\boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}}$ and $\overline{\boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}\boldsymbol{Z}}$, we can reconstruct the cone frustum from a dense set of samples on or near the frustum.

8.1.1 Simple Attempt to Reconstruct Height Extremes

The cone parameterization in terms of the centroid $\boldsymbol{C}$ is

$$\boldsymbol{P}(h,\phi) = \boldsymbol{C} + (h - p_3)\boldsymbol{U} + h\tan\theta(\cos\phi\boldsymbol{W}_0 + \sin\phi\boldsymbol{W}_1) \tag{117}$$

The minimum and maximum of the projections onto $\boldsymbol{U}$ relative to $\boldsymbol{C}$ are $\hat{h}_0 = h_0 - p_3$ and $\hat{h}_1 = h_1 - p_3$, respectively. Recall that $p_3 = ((h_1^3 - h_0^3)/3)/((h_1^2 - h_0^2)/2)$, so we have two equations in h_0 and h_1 that we can solve for in terms of $\hat{h}_0$ and $\hat{h}_1$,

$$h_0 = \frac{\hat{h}_0^2 + \hat{h}_0\hat{h}_1 - 2\hat{h}_1^2}{3(\hat{h}_0 + \hat{h}_1)}, \quad h_1 = \frac{\hat{h}_0^1 + \hat{h}_1\hat{h}_0 - 2\hat{h}_0^2}{3(\hat{h}_1 + \hat{h}_0)} \tag{118}$$

Because $\boldsymbol{C}$ is the centroid, it is necessary that $\hat{h}_0 < 0 < \hat{h}_1$. Also notice that $h_1 - h_0 = \hat{h}_1 - \hat{h}_0$, and the right-hand side is a robust estimate of $h_1 - h_0$.

Although this is a simple mathematical problem given infinitely many points on a cone frustum surface, in practice it has problems. Listing 18 contains code to generate a rectangular mesh of points on a cone frustum that leads to $\hat{h}_1$ being nearly equal to $-\hat{h}_0$, which causes the denominator in the h_0 and h_1 reconstruction equations to be nearly zero. The reconstructed h_0 and h_1 are nowhere near what they should be.

Listing 18. The reconstructed h_0 and h_1 for a dense sample of points on the cone frustum are grossly incorrect. The cone axis direction is estimated using equation (116) rather than as an eigenvector from equation (115); however, both estimates are nearly identical and nearly equal to the $(3,2,1)/\sqrt{14}$.

```
Vector3<double> V = { 3.0, 2.0, 1.0 };
Vector3<double> U = { 1.0, 2.0, 3.0 };
Vector3<double> basis[3];
basis[0] = U;
ComputeOrthogonalComplement(1, basis);
U = basis[0];
Vector3<double> W0 = basis[1];
Vector3<double> W1 = basis[2];
```

```
double h0 = 1.0;
double h1 = 2.0;
double theta = GTE_C_PI / 4.0;
double tantheta = std::tan(theta);

size_t const numH = 512, numR = 512;
std::vector<Vector3<double>> X(numH * numR);
for (size_t ih = 0, i = 0; ih < numH; ++ih)
{
    double h = h0 + (h1 - h0) * (double)ih / (double)(numH - 1);
    double r = h * tantheta;
    for (size_t ir = 0; ir < numR; ++ir, ++i)
    {
        double phi = GTE_C_TWO_PI * (double)ir / (double)numR;
        double csphi = std::cos(phi);
        double snphi = std::sin(phi);
        X[i] = V + h * U + r * (csphi * W0 + snphi * W1);
    }
}

Vector3<double> C{ 0.0, 0.0, 0.0 };  // the centroid
for (size_t i = 0; i < X.size(); ++i)
{
    C += X[i];
}
C /= (double)X.size();

Vector3<double> Ufit{ 0.0, 0.0, 0.0 };  // the third-order term
for (size_t i = 0; i < X.size(); ++i)
{
    Vector3<double> diff = X[i] - C;
    Ufit += Dot(diff, diff) * diff;
}
Ufit /= (double)X.size();
Normalize(Ufit);

double H1cen = std::numeric_limits<double>::max();
double h1hat = -h0hat;
for (size_t i = 0; i < X.size(); ++i)
{
    Vector3<double> diff = X[i] - C;
    double h = Dot(Ufit, diff);
    h0hat = std::min(h, h0hat);
    h1hat = std::max(h, h1hat);
}
double hsum = h1hat + h0hat;
double h0reconstruct = (h0hat * (h0hat + h1hat) - 2.0 * h1hat * h1hat) / (3.0 * hsum);
double h1reconstruct = (h1hat * (h0hat + h1hat) - 2.0 * h0hat * h0hat) / (3.0 * hsum);
// h0hat = -0.50000000000238787
// h1hat = 0.50000000000458455
// hsum = -8.7140517258189930e-14
// h0reconstruct = -75871822289.547775 (original was 1.0)
// h1reconstruct = -75871822288.547775 (original was 2.0)
```

Other equations can be derived to attempt reconstruction of h_0 and h_1. For example, we can compute averages of powers of the height relative to the centroid,

$$\frac{1}{A}\int_{h_0}^{h_1}\int_0^{2\pi} (\boldsymbol{U}\cdot(\boldsymbol{P}-\boldsymbol{C}))^2\,dA = \frac{1}{\pi(h_1^2-h_0^2)}\int_{h_0}^{h_1}\int_0^{2\pi}(h-p_3)^2 h\,dh\,d\phi = p_4 - p_3^2 \tag{119}$$

Given a set of sample points, the equation to reconstruct the height extremes is

$$p_4 - p_3^2 = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{U}\cdot(\boldsymbol{X}_i-\boldsymbol{C}))^2 = b \tag{120}$$

where the last equality defines b. We know the estimate $h_1 - h_0 = \hat{h}_1 - \hat{h}_0 = a$ where the last equality defines a. We need to solve $h_1 - h_0 = a$ and $p_4 - p_3^2 = b$. The solution is

$$h_0 = \frac{+(36ab - 3a^3) \pm \sqrt{3}\, a^2 \sqrt{a^2 - 12b}}{6(a^2 - 12b)}, \quad h_1 = \frac{-(36ab - 3a^3) \pm \sqrt{3}\, a^2 \sqrt{a^2 - 12b}}{6(a^2 - 12b)} \tag{121}$$

In Listing 18, after reconstruction of `h0hat` and `h1hat`, add the code

```
double a = h1hat - h0hat;  // 1.0000000000069724
double b = 0.0;
for (size_t i = 0; i < X.size(); ++i)
{
    Vector3<double> diff = X[i] - C;
    double h = Dot(Ufit, diff);
    b += h * h;
}
b /= (double)X.size();  // 0.083659491193733351
double discr = a * a - 12.0 * b;  // -0.0039138943108554258
```

The discriminant $a^2 - 12b^2$ is negative, so there are no real-valued solutions for h_0 and h_1.

Attempts to estimate h_0 and h_1 from the average integral of $(\boldsymbol{U} \cdot (\boldsymbol{P} - \boldsymbol{C}))^3$ also failed when using the point samples to estimate the integrals and then solve the resulting equations.

8.1.2 Attempt to Reconstruct the Cone Axis Direction

The eigendecomposition in equation (115) provides a symbolic expression involving the cone axis direction $\boldsymbol{U}$. In a numerical implementation, we need to identify which eigenvector output by the eigensolver corresponds to $\boldsymbol{U}$. The convariance matrix of the point samples is used to estimate $\overline{\boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}}$. We cannot expect two distinct eigenvalues $\lambda_1 = p_4 - p_3^2$ and $\lambda_2 = (p_4 \tan^2\theta)/4$, the first with multiplicity 1 and the second with multiplicity 2. For if we obtained such a result numerically, selection of $\boldsymbol{U}$ is trivial. Even theoretically we could have a problem when there is a single eigenvalue of multiplicity 3.

If the point samples are nearly on a cone frustum, let the numerically computed eigenvalues be sorted as $\lambda_1 \le \lambda_2 \le \lambda_3$. We could compute $\lambda_2 - \lambda_1$ and $\lambda_3 - \lambda_2$ and claim that the minimum difference is due to numerical rounding errors. The conclusion is that those two eigenvalues are theoretically a single eigenvalue. The other eigenvalue is the one associated with $\boldsymbol{U}$. Some experiments showed that this is not a reliable way to select $\boldsymbol{U}$.

Instead we can use equation (116) to select $\boldsymbol{U}$ unambiguously. The integral $\overline{\boldsymbol{Z}\boldsymbol{Z}^{\mathsf{T}}\boldsymbol{Z}}$ is estimated by a summation involving the point samples as shown in Listing 18. The normalization of the summation is the estimate for $\boldsymbol{U}$, named `Ufit` in the listing.

8.1.3 Attempt to Reconstruct the Cone Vertex

As mentioned previously, reconstructing h_0 and h_1 would allow us to compute p_3 and then $\boldsymbol{V} = \boldsymbol{C} - p_3\boldsymbol{U}$, but the reconstruction attempts can fail numerically. A more reliable algorithm for estimating $\boldsymbol{V}$ is shown here. The goal is to estimate p_3 directly rather than estimating h_0 and h_1 and then computing p_3 from them.

In the following, $\boldsymbol{U}$ is the estimate of the cone axis direction obtained from equation (116). We want to estimate a positive t for which $\boldsymbol{V} = \boldsymbol{C} - t\boldsymbol{U}$. At the same time we need to estimate $s = \cos^2\theta$ for the cone

angle. The algorithm will use the discrete least-squares error function for a specified $\boldsymbol{U}$. Define $\boldsymbol{\Delta}_i = \boldsymbol{X}_i - \boldsymbol{C}$, $a_i = \boldsymbol{U} \cdot \boldsymbol{\Delta}_i$ and $b_i = \boldsymbol{\Delta}_i \cdot \boldsymbol{\Delta}_i$. Define

$$F_i = (\boldsymbol{X}_i - \boldsymbol{V})^{\mathsf{T}} \left(sI - \boldsymbol{U}\boldsymbol{U}^{\mathsf{T}} \right) (\boldsymbol{X}_i - \boldsymbol{V}) = s\left(t^2 + 2a_i t + b_i\right) - (t + a_i)^2 \tag{122}$$

The least-squares error function is $E = (\sum_{i=1}^{n} F_i^2)/n$. The derivatives of the F_i terms are

$$\frac{\partial F_i}{\partial s} = t^2 + 2a_i t + b_i, \quad \frac{\partial F_i}{\partial t} = 2(s-1)(t + a_i) \tag{123}$$

Some algebra will show that $\partial E/\partial t = 0$ and $\partial E/\partial s = 0$ lead to

$$sp_3(t) - q_3(t) = 0, \quad sp_4(t) - q_4(t) = 0 \tag{124}$$

where $p_d(t)$ and $q_d(t)$ are polynomials of degree d. Specifically,

$$\begin{aligned} p_3(t) &= t^3 + e_0 t^2 + e_1 t + e_2 \\ q_3(t) &= t^3 + e_0 t^2 + e_3 t + e_4 \\ p_4(t) &= t^4 + f_0 t^3 + f_1 t^2 + f_2 t + f_3 \\ q_4(t) &= t^4 + f_0 t^3 + f_4 t^2 + f_5 t + f_6 \end{aligned} \tag{125}$$

where

$$\begin{aligned} e_0 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} 3a_i & f_0 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} 4a_i \\ e_1 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} (2a_i^2 + b_i) & f_1 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} (4a_i^2 + 2b_i) \\ e_2 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} a_i b_i & f_2 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} 4a_i b_i \\ e_3 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} 3a_i^2 & f_3 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} b_i^2 \\ e_4 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} a_i^3 & f_4 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} (5a_i^2 + b_i) \\ & & f_5 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} (2a_i^3 + 2a_i b_i) \\ & & f_6 &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} a_i^2 b_i \end{aligned} \tag{126}$$

Define $c_{rs} = \frac{1}{n}\sum_{i=1}^{n} a_i^r b_i^s$. We can solve one st-equation for $s = q_4(t)/p_4(t)$, substitute into the other st-equation and then multiply by $p_4(t)$ to obtain

$$0 = p_3(t)q_4(t) - p_4(t)q_3(t) = g_4 t^4 + g_3 t^3 + g_2 t^2 + g_1 t + g_0 \tag{127}$$

where

$$\begin{aligned} g_4 &= c_{30} - c_{11} + c_{10}(c_{01} - c_{20}) \\ g_3 &= c_{21} - c_{02} + c_{01}(c_{01} + c_{20}) + 2(c_{10}(c_{30} - c_{11}) - c_{20}^2) \\ g_2 &= 3(c_{11}(c_{01} - c_{20}) + c_{10}(c_{21} - c_{02})) \\ g_1 &= c_{01}c_{21} - 3c_{02}c_{20} + 2(c_{20}c_{21} - c_{11}(c_{30} - c_{11})) \\ g_0 &= c_{11}c_{21} - c_{02}c_{30} \end{aligned} \tag{128}$$

The least-squares error function is smooth and nonnegative, so there must be at least one (s, t) for which $\nabla E(s, t) = (0, 0)$. In theory, this means $g(t) = 0$ must have at least one real-valued root. Naturally, the

quartic root solver must guard against floating-point round-off errors to guarantee this. The GTE quartic root solver can be executed with rational arithmetic to correctly classify the roots, after which rounding errors will not change the classification. For each root $t \geq 0$, compute $s = q_3(t)/p_3(t)$. If $s \in (0, 1)$, we have a valid angle because $s = \cos^2\theta$. Evaluate $E(s, t)$. Of all such valid pairs (s, t), choose the one whose E-value is minimum. This pair is used to initialize $\boldsymbol{V}$ and $\cos\theta$.

Using floating-point arithmetic, it is unlikely but possible that there is no pair (s, t) generated by the algorithm of the previous paragraph. An implementation must guard against this. Although most likely bad estimates, choose the initial $\boldsymbol{V}$ to be the average $\boldsymbol{C}$ and chose the angle arbitrarily to be $\pi/4$ which is the midpoint of the domain of the angles $(0, \pi/2)$.

Listing 18 can be modified by replacing all the code after the line Normalize(Ufit) with that shown in Listing 19. The new listing shows an implementation for the initial guesses for $\boldsymbol{V}$ and $\cos^2\theta$.

Listing 19. Code for the initial guesses for the cone vertex and cone angle. The numbers in the comments are those produced by the example in Listing 18.

```
double c10 = 0.0, c20 = 0.0, c30 = 0.0, c01 = 0.0, c02 = 0.0, c11 = 0.0, c21 = 0.0;
for (size_t i = 0; i < X.size(); ++i)
{
    Vector3<double> diff = X[i] - C;
    double ai = Dot(Ufit, diff);
    double bi = Dot(diff, diff);
    c10 += ai;
    c20 += ai * ai;
    c30 += ai * ai * ai;
    c01 += bi;
    c02 += bi * bi;
    c11 += ai * bi;
    c21 += ai * ai * bi;
}
c10 /= (double)X.size();
c20 /= (double)X.size();
c30 /= (double)X.size();
c01 /= (double)X.size();
c02 /= (double)X.size();
c11 /= (double)X.size();
c21 /= (double)X.size();

// The coefficients for polynomials p3(t) and q3(t).
double e0 = 3.0 * c10;
double e1 = 2.0 * c20 + c01;
double e2 = c11;
double e3 = 3.0 * c20;
double e4 = c30;

// The coefficients for polynomials p4(t) and q4(t).
double f0 = 4.0 * c10;
double f1 = 4.0 * c20 + 2.0 * c01;
double f2 = 4.0 * c11;
double f3 = c02;
double f4 = 5.0 * c20 + c01;
double f5 = 2.0 * c30 + 2.0 * c11;
double f6 = c21;

// The coefficients for the quartic polynomial g(t).
double g0 = c11 * c21 - c02 * c30;  // 0.053566286603418618
double g1 = c01 * c21 - 3.0 * c02 * c20 + 2.0 * (c20 * c21 - c11 * (c30 - c11));  // -0.98354780514088114
double g2 = 3.0 * (c11 * (c01 - c20) + c10 * (c21 - c02));  // 1.7570948908739785
double g3 = c21 - c02 + c01 * (c01 + c20) + 2.0 * (c10 * (c30 - c11) - c20 * c20);  // -0.37366801803251715
double g4 = c30 - c11 + c10 * (c01 - c20);  // -0.25097847358125042

// Compute the roots of g(t) = 0.
std::map<double, int> rmMap;
```

```
RootsPolynomial<double>::SolveQuartic(g0, g1, g2, g3, g4, rmMap);
// root[0] = -3.6829478517723415
// root[1] = 0.061025506205900915
// root[2] = 0.63307745500821921
// root[3] = 1.4999999999961315

std::vector<std::array<double, 3>> info;
double s, t;
for (auto const& element : rmMap)
{
    t = element.first;
    if (t > 0.0)
    {
        s = (e4 + t * (e3 + t * (e0 + t))) / (e2 + t * (e1 + t * (e0 + t)));
        if (0.0 < s && s < 1.0)
        {
            double error = 0.0;
            for (size_t i = 0; i < X.size(); ++i)
            {
                Vector3<double> diff = X[i] - C;
                double ai = Dot(Ufit, diff);
                double bi = Dot(diff, diff);
                double tpai = t + ai;
                double Fi = s * (bi + t * (2.0 * ai + t)) - tpai * tpai;
                error += Fi * Fi;
            }
            error /= (double)X.size();
            std::array<double, 3> item = { s, t, error };
            info.push_back(item);
        }
    }
}

if (info.size() > 0)
{
    std::array<double, 3> minItem = info.front();
    for (auto const& item : info)
    {
        if (item[2] < minItem[2])
        {
            minItem = item;
        }
    }

    // minItem = { minS, minT, minError }
    Vfit = C - minItem[1] * Ufit;
    cosAngleSqrFit = minItem[0];

    // We do not need this for the minimizer, but let's reconstruct
    // the heights to see how close we get to the original ones
    // (h0 = 1, h1 = 2).  Knowing the estimate h1 - h0 = 1, solve
    //   minT = p3 = ((h1^3-h0^3)/3)/((h1^2-h0^2)/2)
    // for h0 and h1.
    double h0 = (6.0 + std::sqrt(276)) / 24.0;  // 0.94221865524317294
    double h1 = 1.0 + h0;  // 1.9422186552431731

    // The original cone has p3 = 14/9 = 1.555555 and the minimizing
    // t-value is 1.4999999999961315.
}
else
{
    Vfit = C;
    cosAngleSqrFit = sqrt(0.5);  // angleFit is pi/4
}
```

9 Fitting a Paraboloid to 3D Points of the Form $(x, y, f(x, y))$

Given a set of samples $\{(x_i, y_i, z_i)\}_{i=1}^m$ and assuming that the true values lie on a paraboloid

$$z = f(x, y) = p_1x^2 + p_2xy + p_3y^2 + p_4x + p_5y + p_6 = \boldsymbol{P} \cdot \boldsymbol{Q}(x, y) \tag{129}$$

where $\boldsymbol{P} = (p_1, p_2, p_3, p_4, p_5, p_6)$ and $\boldsymbol{Q}(x, y) = (x^2, xy, y^2, x, y, 1)$, select $\boldsymbol{P}$ to minimize the sum of squared errors

$$E(\boldsymbol{P}) = \sum_{i=1}^{m} (\boldsymbol{P} \cdot \boldsymbol{Q}_i - z_i)^2 \tag{130}$$

where $\boldsymbol{Q}_i = \boldsymbol{Q}(x_i, y_i)$. The minimum occurs when the gradient of E is the zero vector,

$$\nabla E = 2 \sum_{i=1}^{m} (\boldsymbol{P} \cdot \boldsymbol{Q}_i - z_i)\boldsymbol{Q}_i = \boldsymbol{0} \tag{131}$$

Some algebra converts this to a system of 6 equations in 6 unknowns:

$$\left(\sum_{i=1}^{m} \boldsymbol{Q}_i \boldsymbol{Q}_i^{\mathsf{T}} \right) \boldsymbol{P} = \sum_{i=1}^{m} z_i \boldsymbol{Q}_i \tag{132}$$

The product $\boldsymbol{Q}_i\boldsymbol{Q}_i^{\mathsf{T}}$ is a product of the 6×1 matrix $\boldsymbol{Q}_i$ with the 1×6 matrix $\boldsymbol{Q}_i^{\mathsf{T}}$, the result being a 6×6 matrix.

Define the 6×6 symmetric matrix $A = \sum_{i=1}^{m} \boldsymbol{Q}_i\boldsymbol{Q}_i^{\mathsf{T}}$ and the 6×1 vector $\boldsymbol{B} = \sum_{i=1}^{m} z_i\boldsymbol{Q}_i$. The choice for $\boldsymbol{P}$ is the solution to the linear system of equations $A\boldsymbol{P} = \boldsymbol{B}$. The entries of A and $\boldsymbol{B}$ indicate summations over the appropriate product of variables. For example, $s(x^3y) = \sum_{i=1}^{m} x_i^3 y_i$:

$$\begin{bmatrix} s(x^4) & s(x^3y) & s(x^2y^2) & s(x^3) & s(x^2y) & s(x^2) \\ s(x^3y) & s(x^2y^2) & s(xy^3) & s(x^2y) & s(xy^2) & s(xy) \\ s(x^2y^2) & s(xy^3) & s(y^4) & s(xy^2) & s(y^3) & s(y^2) \\ s(x^3) & s(x^2y) & s(xy^2) & s(x^2) & s(xy) & s(x) \\ s(x^2y) & s(xy^2) & s(y^3) & s(xy) & s(y^2) & s(y) \\ s(x^2) & s(xy) & s(y^2) & s(x) & s(y) & s(1) \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{bmatrix} = \begin{bmatrix} s(zx^2) \\ s(zxy) \\ s(zy^2) \\ s(zx) \\ s(zy) \\ s(z) \end{bmatrix} \tag{133}$$

An implementation is ApprParaboloid3.h.