Deduction of Explicit, Non-Iterative Expressions for Orthogonal Regression Analysis in Two and Three Dimensions

Diego Velez
delta@velez.ca

Abstract

Simple linear regression analysis techniques, such as the method of Total Least Squares, account for errors in the dependent variables but assume the independent variable (or variables) to be error-free. A suitable approach for dealing with errors in all variables is known as the "Orthogonal" or "Deming" regression.

In spite of the many fields where such technique is applicable (e.g., computer graphics, robotics, data fitting, artificial intelligence, and others), we appear to lacking an open, complete (e.g., ***non-reliant on external eignevalue solvers***), and practical guide for implementing it, for two and three-dimensional data.

This article outines the deduction and implementation of explicit, non-iterative techniques, and provides functional examples in the C programming language.

1 Motivation

Simple regression analysis is usually taught in most science and engineering disciplines. Twenty-five years after my final exam on Statistics for Engineering, I still vividly remember my final exam: it consisted of determining the optimal ratios of two compounds, in order to maximize the strength of their mixture. The strength values were based on experimental data. My fellow classmates and I had to perform a classic (*vertical*, non-orthogonal) regression analysis, with two independent variables (the percentual ratios of each substance) and one dependent variable, which was, again, the strength.

Having prepared well for that exam, I did not sweat while performing a two-variable *Least Squares* analysis, which indicated that the strength behaved as a two-dimensional linear function (a plane, if seen geometrically) of the composition ratios of the materials.

I clearly remember this anecdote because, as I later found out, my professor had a serious conceptual mishap: he considered the right answer to be *"there are no optimal ratios for the mixture, because its strength behaves mathematically as a plane, and, since planes grow indefinitely, they have no minima nor maxima"*. Yet, I realized that it could not be an unlimited plane, given that (due to both logic and physics) each one of the ratios had to be positive; each ratio could not exceed 100%; and the sum of all ratios had to total 100%.

I was the only student of my class (maybe because of my lifetime-long affinity towards 3D graphics and its parent, geometry) who realized that was not a plane but a three-dimensional *sector* of a plane (in fact, a triangular area), which *had* maxima and minima, and were (in that case) determined by the weakest and strongest compounds. My answer was: *"as shown by the mathematical model, and considering the composition constrains, the strongest mixture is obtained by just using compound A"*.

No argument I made was able to change my professor's mind and to rightfully adjust my final exam's mark from 80% to 100%.

Two decades and a half after that test, I find myself building a software module in which a surface made by three-dimensional points is stored in memory as a two-dimensional grid. Each cell of the grid consists of 4 points: $P_{i,j}$, $P_{i,j+1}$, $P_{i+1,j}$, and $P_{i+1,j+1}$, which *approximate* a segment of a plane. As we know, a plane is determined by three points. Therefore, the question arises: for each cell in the grid, ¿what plane better fits its corners? Granted, if the grid is fine enough, one could just pick *any* 3 points, accepting some error between the plane and the unchosen 4^{th} point. An initial thought was to use the *Least Squares* method, just as I did many years ago in *that* final exam. This, however, presented two difficulties:

- The distance from each of the points to the plane must be evaluated orthogonally, not vertically.
- If z is chosen as the dependent variable, *Least Squares* fails when dealing with vertical planes, such as: $\Pi : \langle 0, 0, k\rangle \cdot \langle x, y, z\rangle = 0$

The next sections detail the deduction of explicit expressions for performing an Orthogonal or Deming regression.

2 Two-Dimensional Orthogonal Regression

Given a set of two-dimensional points P_i, for $i = 1, 2, ..., n$ the objective is determining the line

$$\vec{L} : \vec{Q} + \lambda\vec{A}$$

such as $\vec{Q}$ and $\vec{A}$ minimize the sum of the distances[1] from each P_i to $\vec{L}$

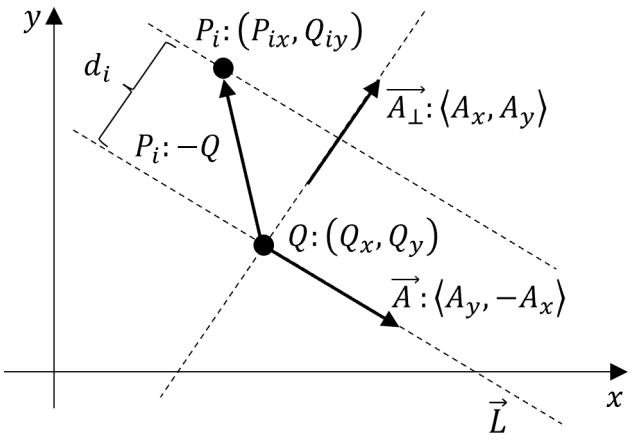

Figure 1: Orthogonal distance from a given Point to a Line in two dimensions

Although the solution of the two-dimensional case can be obtained using the direction vector of the line $\vec{A}$, solving for its normal vector $\vec{A_\perp}$ simplifies (as we shall see) the notation, and makes it compatible with the three-dimensional case, where the normal form is preferable. Once $\vec{A_\perp} : \langle Ax, Ay\rangle$ is found, it is easy to obtain the direction vector of the line $\vec{A} = \langle Ay, -Ax\rangle$

[1]More precisely, the sum of the Gaussian squares of the distances. Other types of *metrics* should be feasable, too.

The distance to $\vec{L}$ is given by:

$$d_i = \frac{\left(\vec{P} - \vec{Q}\right) \cdot \vec{A_\perp}}{\left\|\vec{A_\perp}\right\|}$$

$$d_i = \frac{(\vec{P} - \vec{Q}) \cdot \langle A_x, A_y \rangle}{\sqrt{{A_x}^2 + {A_y}^2}}$$

$$d_i = \frac{\langle P_{ix} - Q_{ix}, P_{iy} - Q_{iy} \rangle \cdot \langle A_x, A_y \rangle}{\sqrt{{A_x}^2 + {A_y}^2}}$$

$$d_i = \frac{A_x(P_{ix} - Q_{ix}) + A_y(P_{iy} - Q_{iy})}{\sqrt{{A_x}^2 + {A_y}^2}}$$

The square of the distance is:

$${d_i}^2 = \frac{\left[A_x \left(P_{ix} - Q_{ix}\right) + A_y \left(P_{iy} - Q_{iy}\right)\right]^2}{{A_x}^2 + {A_y}^2}$$

The sum of the squares of the distances (or *Gaussian* metric) is:

$$\sum_{i=1}^{n} {d_i}^2 = \frac{\left[A_x \left(P_{ix} - Q_{ix}\right) + A_y \left(P_{iy} - Q_{iy}\right)\right]^2}{{A_x}^2 + {A_y}^2} \tag{1}$$

In order to find the Minima, we partially differentiate (1) against Q_x and Qy, and make equal to zero:

$$0 = \frac{\partial}{\partial Q_x} {d_i}^2 = \frac{1}{{A_x}^2 + {A_y}^2} \left[-2{A_x}^2 \sum_{i=1}^{n} \left(P_{ix} - Q_x\right) - 2A_x A_y \sum_{i=1}^{n} \left(P_{iy} - Q_y\right) \right]$$

$$0 = \frac{\partial}{\partial Q_y} {d_i}^2 = \frac{1}{{A_x}^2 + {A_y}^2} \left[-2A_x A_y \sum_{i=1}^{n} \left(P_{ix} - Q_x\right) - 2{A_y}^2 \sum_{i=1}^{n} \left(P_{iy} - Q_y\right) \right]$$

A_x and A_y were defined as the components of *any* director-vector perpendicular to line $\vec{L}$. For convenience, we can choose $\vec{A_\perp}$ to be an unitary vector, whose magnitude (and its square, ${A_x}^2 + {A_y}^2$) are constant. Therefore,

$$0 = -\cancel{2}{A_x}^2 \sum_{i=1}^{n} \left(P_{ix} - Q_x\right) - \cancel{2}A_x A_y \sum_{i=1}^{n} \left(P_{iy} - Q_y\right) \tag{2}$$

$$0 = -\cancel{2}{A_x}^2 \sum_{i=1}^{n} \left(P_{ix} - Q_x\right) - \cancel{2}A_x A_y \sum_{i=1}^{n} \left(P_{iy} - Q_y\right) \tag{3}$$

It can be seen that choosing:

$$\overline{x} = Q_x = \frac{1}{n} \sum_{i=1}^{n} P_{ix}$$

$$\overline{y} = Q_y = \frac{1}{n}\sum_{i=1}^{n} P_{iy}$$

Solves (2) and (3) in the general case in which A_x and A_y are not both zero.

Equation (1) can be re-written as:

$$\sum_{i=1}^{n} {d_i}^2 = \frac{1}{{A_x}^2 + A_y^2}\left[{A_x}^2\sum_{i=1}^{n}(P_{ix}-\overline{x})^2 + 2A_xA_y\sum_{i=1}^{n}(P_{ix}-\overline{x})(P_{iy}-\overline{y}) + {A_y}^2\sum_{i=1}^{n}(P_{iy}-\overline{y})^2\right] \tag{4}$$

This sum is partially derived and made equal to zero, in order to find maxima and minima:

$$0 = \frac{\partial}{\partial A_x}\sum_{i=1}^{n} {d_i}^2 = \cancel{2}A_x\sum_{i=1}^{n}(P_{ix}-\overline{x})^2 + \cancel{2}A_y\sum_{i=1}^{n}(P_{ix}-\overline{x})(P_{iy}-\overline{y})$$

$$0 = \frac{\partial}{\partial A_x}\sum_{i=1}^{n} {d_i}^2 = \cancel{2}A_x\sum_{i=1}^{n}(P_{ix}-\overline{x})(P_{iy}-\overline{y}) + \cancel{2}A_y\sum_{i=1}^{n}(P_{iy}-\overline{y})^2$$

Rewriting in matrix form:

$$\begin{bmatrix} \sum_{i=1}^{n}(P_{ix}-\overline{x})^2 & \sum_{i=1}^{n}(P_{ix}-\overline{x})(P_{iy}-\overline{y}) \\ \sum_{i=1}^{n}(P_{ix}-\overline{x})(P_{iy}-\overline{y}) & \sum_{i=1}^{n}(P_{iy}-\overline{y})^2 \end{bmatrix}\begin{bmatrix} A_x \\ A_y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5}$$

Since we considered A_x and A_y to be the components of a unitary vector, $\vec{A}$ can be presented in angular terms as:

$$\begin{bmatrix} \sum_{i=1}^{n}(P_{ix}-\overline{x})^2 & \sum_{i=1}^{n}(P_{ix}-\overline{x})(P_{iy}-\overline{y}) \\ \sum_{i=1}^{n}(P_{ix}-\overline{x})(P_{iy}-\overline{y}) & \sum_{i=1}^{n}(P_{iy}-\overline{y})^2 \end{bmatrix}\begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{6}$$

This system can now be solved trigonometrically. It is convenient to define some auxiliary variables, noting this is a symmetric matrix. Let:

$$k_{xx} = \sum_{i=1}^{n}(P_{ix}-\overline{x})^2$$

$$k_{xy} = \sum_{i=1}^{n}(P_{ix}-\overline{x})(P_{iy}-\overline{y})$$

$$k_{yy} = \sum_{i=1}^{n}(P_{iy}-\overline{y})^2$$

Equation (6) becomes:

$$\begin{bmatrix} k_{xx} & k_{xy} \\ k_{xy} & k_{yy} \end{bmatrix}\begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{7}$$

$$k_{xx}\cos(\theta) + k_{xy}\sin(\theta) = 0$$

$$k_{xx}\cos(\theta) = -k_{xy}\sin(\theta)$$

As long as $k_{xy} \neq 0$:

$$\frac{k_{xx}}{-k_{xy}} = \frac{\sin(\theta)}{\cos(\theta)} = \tan(\theta)$$

Regardless of the data, k_{xx} is always positive, due to being the sum of squares; whereas k_{xy} can either be positive or negative.

$$\theta = \arctan\left(\frac{|k_{xx}|}{-k_{xy}}\right)$$

$$\cos(\theta) = \cos\left[\arctan\left(\frac{|k_{xx}|}{-k_{xy}}\right)\right] = \frac{-k_{xy}}{\sqrt{{k_{xx}}^2 + {k_{xy}}^2}} \tag{8}$$

$$\sin(\theta) = \sin\left[\arctan\left(\frac{|k_{xx}|}{-k_{xy}}\right)\right] = \frac{|k_{xx}|}{\sqrt{{k_{xx}}^2 + {k_{xy}}^2}} \tag{9}$$

Substituting for k_{xx} and k_{xy} :

$$A_x = \cos(\theta) = \frac{-\sum_{i=1}^{n}(P_{ix} - \overline{x})(P_{iy} - \overline{y})}{\sqrt{\left[\sum_{i=1}^{n}(P_{iy} - \overline{y})^2\right]^2 + \left[\sum_{i=1}^{n}(P_{ix} - \overline{x})(P_{iy} - \overline{y})\right]^2}} \tag{10}$$

Likewise,

$$A_y = \sin(\theta) = \frac{\sum_{i=1}^{n}(P_{ix} - \overline{x})^2}{\sqrt{\left[\sum_{i=1}^{n}(P_{iy} - \overline{y})^2\right]^2 + \left[\sum_{i=1}^{n}(P_{ix} - \overline{x})(P_{iy} - \overline{y})\right]^2}} \tag{11}$$

$\vec{A_\perp} : \langle A_x, A_y \rangle$ is the solution for the vector perpendicular to the line. The direction vector of the line is: $\vec{L}$ is $\langle A_y, -A_x \rangle$

$$\vec{L} = \left\langle \frac{1}{n}\sum_{i=1}^{n} P_{ix}, \frac{1}{n}\sum_{i=1}^{n} P_{iy} \right\rangle + \frac{\lambda}{\sqrt{\left[\sum_{i=1}^{n}(P_{iy} - \overline{y})^2\right]^2 + \left[\sum_{i=1}^{n}(P_{ix} - \overline{x})(P_{iy} - \overline{y})\right]^2}} \left\langle \sum_{i=1}^{n}(P_{ix} - \overline{x})^2, \sum_{i=1}^{n}(P_{ix} - \overline{x})(P_{iy} - \overline{y}) \right\rangle \tag{12}$$

The parametric form avoids the inconvenience of dealing with *principal values* (as it is often the case under trigonometric approaches), making this solution adequate for computing applications.

3 Implementation and Validation for Two-Dimensional variables

The code below is available and can also be executed online, without a local compiler, at `https://onlinegdb.com/gV4z7XA8m`.

The software loads X and Y data defined as $y = 2x$, plus a small amount of noise. At completion, the program prints out the parametric form of the line.

3.1 Source code for 2D Orthogonal Regression

```
#include <stdio.h>
#include <math.h> /* Needed to calculate square roots */
#include <time.h> /* Needed to seed the random number generator */
#include <stdlib.h> /* Needed for rand() */

/* Copyright (C) 2021, Diego Velez

The code below is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License */


double sum_sqr(double V[], double avg,int n)
{
    int i;
    double sum=0.0;
    for (i=0;i<n;i++)
        sum+=(V[i]-avg)*(V[i]-avg);
    return sum;
}

double sum_prod(double X[], double x_avg, double Y[], double y_avg, int n)
{
    int i;
    double sum=0.0;
    for (i=0;i<n;i++)
        sum+=(X[i]-x_avg)*(Y[i]-y_avg);
    return sum;
}

double avg(double V[],int n)
{
    int i;
    double sum=0.0;
    for (i=0;i<n;i++)
```

```
        sum+=V[i];
    return sum/n;
}

double randf(double low, double high)
{
  return (rand()/(float)(RAND_MAX))*fabs(low-high)+low;
}

int main()
{
    srand(time(NULL));

    double X[5], Y[5];
    X[0]=0+0*randf(-0.2,0.2); Y[0]=0+0*randf(-0.2,0.2);
    X[1]=1+0*randf(-0.2,0.2); Y[1]=2+0*randf(-0.2,0.2);
    X[2]=2+0*randf(-0.2,0.2); Y[2]=4+0*randf(-0.2,0.2);
    X[3]=3+0*randf(-0.2,0.2); Y[3]=6+0*randf(-0.2,0.2);
    X[4]=4+0*randf(-0.2,0.2); Y[4]=8+0*randf(-0.2,0.2);

    double x_avg=avg(X,5);
    double y_avg=avg(Y,5);

    double k_xx = sum_sqr(X, x_avg, 5);
    double k_xy = sum_prod(X, x_avg, Y, y_avg, 5);

    printf("k_xx=%4.3f, k_xy=%4.3f\n",k_xx, k_xy);
    printf("Line: L<x,y> = <%4.3f,%4.3f> + t*<%4.3f,%4.3f>, slope = %f",
    x_avg, y_avg, k_xx/sqrt(k_xx*k_xx+k_xy*k_xy), k_xy/sqrt(k_xx*k_xx+k_xy*k_xy),k_xy/k_xx);
    return 0;
}
```

3.2 Results

When X and Y incorporate a certain random component or *noise*, say, in the range $[-0.2; 0.2]$, a typical result is:

```
Line: L<x,y> = <1.918,3.982> + t*<0.423,0.906>, slope = 2.144033
```

Similarly, when X or Y have no noise, we obtain the parametric form of the line $y = 2x$

```
Line: L<x,y> = <2.000,4.000> + t*<0.447,0.894>, slope = 2.000000
```

4 Three-Dimensional Orthogonal Regression

A plane can be expressed in its normal form as: $\Pi : \vec{A_\perp} \cdot \left(\vec{X} - \vec{Q}\right) = 0$, where $\vec{A_\perp} : \langle A_x, A_y, A_z \rangle$ is a vector perpendicular to the plane and $\vec{Q}$ is a vector to a point Q in the plane.

The objective is finding expressions for Q and $\vec{A_\perp}$ such that the sum of the distances from each P_i to the plane is minimized.

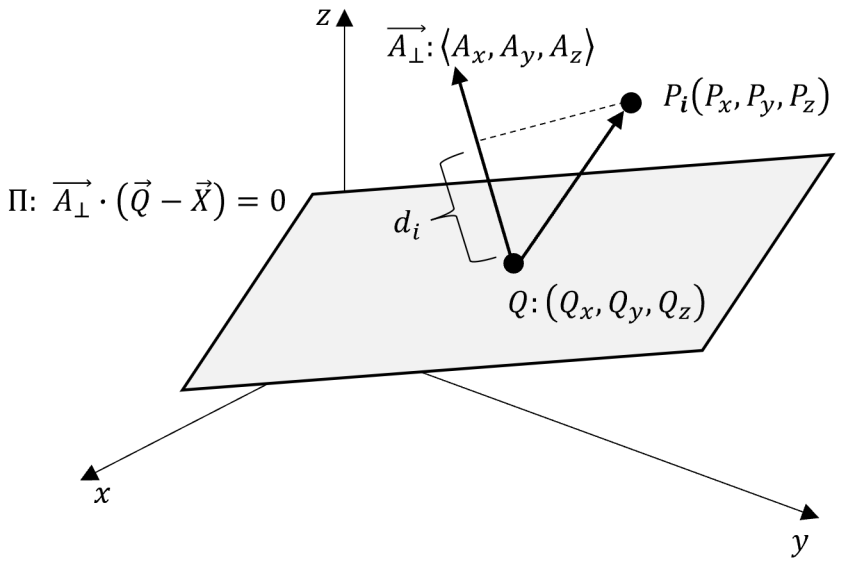

Figure 2: Orthogonal distance from a given Point to a Line in three dimensions

The distance from the point to the plane is:

$$d_i = \frac{\left(\vec{P} - \vec{Q}\right) \cdot \vec{A_\perp}}{\left\|\vec{A_\perp}\right\|}$$

$$d_i = \frac{(\vec{P} - \vec{Q}) \cdot \langle A_x, A_y, A_z \rangle}{\sqrt{{A_x}^2 + {A_y}^2 + {A_z}^2}}$$

$$d_i = \frac{\langle P_{ix} - Q_{ix}, P_{iy} - Q_{iy}, P_{iz} - Q_{iz} \rangle \cdot \langle A_x, A_y, A_z \rangle}{\sqrt{{A_x}^2 + {A_y}^2 + {A_z}^2}}$$

$$d_i = \frac{A_x(P_{ix} - Q_{ix}) + A_y(P_{iy} - Q_{iy}) + A_z(P_{iz} - Q_{iz})}{\sqrt{{A_x}^2 + {A_y}^2 + {A_z}^2}}$$

The square of the distance is:

$${d_i}^2 = \frac{\left[A_x\left(P_{ix} - Q_{ix}\right) + A_y\left(P_{iy} - Q_{iy}\right) + A_z\left(P_{iz} - Q_{iz}\right)\right]^2}{{A_x}^2 + {A_y}^2 + {A_z}^2}$$

The sum of the squares of the distances (or *Gaussian* metric) is:

$$\sum_{i=1}^{n} {d_i}^2 = \frac{\left[A_x\left(P_{ix} - Q_{ix}\right) + A_y\left(P_{iy} - Q_{iy}\right) + A_z\left(P_{iz} - Q_{iz}\right)\right]^2}{{A_x}^2 + {A_y}^2 + {A_z}^2} \tag{13}$$

In order to find the Minima, we partially differentiate against Q_x and Qy, and make equal to zero:

$$0 = \frac{\partial}{\partial Q_x} {d_i}^2 = \frac{1}{{A_x}^2 + {A_y}^2 + {A_z}^2} \left[-2{A_x}^2 \sum_{i=1}^{n} \left(P_{ix} - Q_x\right) - 2A_x A_y \sum_{i=1}^{n} \left(P_{iy} - Q_y\right) - 2A_x A_z \sum_{i=1}^{n} \left(P_{iz} - Q_z\right) \right]$$

$$0 = -2{A_x}^2 \sum_{i=1}^{n} (P_{ix} - Q_x) - 2A_x A_y \sum_{i=1}^{n} (P_{iy} - Q_y) - 2A_x A_z \sum_{i=1}^{n} (P_{iz} - Q_z) \tag{14}$$

Similarly, we can obtain:

$$0 = -2A_x A_y \sum_{i=1}^{n} (P_{ix} - Q_x) - 2{A_y}^2 \sum_{i=1}^{n} (P_{iy} - Q_y) - 2A_y A_z \sum_{i=1}^{n} (P_{iz} - Q_z) \tag{15}$$

$$0 = -2A_x A_z \sum_{i=1}^{n} (P_{ix} - Q_x) - 2A_y A_z \sum_{i=1}^{n} (P_{iy} - Q_y) - 2{A_z}^2 \sum_{i=1}^{n} (P_{iz} - Q_z) \tag{16}$$

It can be seen that choosing:

$$\overline{x} = Q_x = \frac{1}{n} \sum_{i=1}^{n} P_{ix}$$

$$\overline{y} = Q_y = \frac{1}{n} \sum_{i=1}^{n} P_{iy}$$

$$\overline{z} = Q_z = \frac{1}{n} \sum_{i=1}^{n} P_{iz}$$

Solves (14), (15), and (16) in the general case in which A_x, A_y, and A_z are not all zero.

Expression (13) can now be partially differentiated[2] against A_x, A_y, and A_z, to outline a new system of equations

$$0 = \frac{\partial}{\partial A_x} \sum_{i=1}^{n} {d_i}^2 = \cancel{2}A_x \sum_{i=1}^{n} (P_{ix} - \overline{x})^2 + \cancel{2}A_y \sum_{i=1}^{n} (P_{ix} - \overline{x})(P_{iy} - \overline{y}) + \cancel{2}A_z \sum_{i=1}^{n} (P_{ix} - \overline{x})(P_{iz} - \overline{z}) \tag{17}$$

$$0 = \frac{\partial}{\partial A_y} \sum_{i=1}^{n} {d_i}^2 = \cancel{2}A_x \sum_{i=1}^{n} (P_{ix} - \overline{x})(P_{iy} - \overline{y}) + \cancel{2}A_y \sum_{i=1}^{n} (P_{iy} - \overline{y})^2 + \cancel{2}A_z \sum_{i=1}^{n} (P_{iy} - \overline{y})(P_{iz} - \overline{z}) \tag{18}$$

$$0 = \frac{\partial}{\partial A_z} \sum_{i=1}^{n} {d_i}^2 = \cancel{2}A_x \sum_{i=1}^{n} (P_{ix} - \overline{x})(P_{iz} - \overline{z}) + \cancel{2}A_y \sum_{i=1}^{n} (P_{iy} - \overline{y})(P_{iz} - \overline{z}) + \cancel{2}A_z \sum_{i=1}^{n} (P_{iz} - \overline{z})^2 \tag{19}$$

This can be represented in matrix form as:

$$\begin{bmatrix} \sum_{i=1}^{n} (P_{ix} - \overline{x})^2 & \sum_{i=1}^{n} (P_{ix} - \overline{x})(P_{iy} - \overline{y}) & \sum_{i=1}^{n} (P_{ix} - \overline{x})(P_{iz} - \overline{z}) \\ \sum_{i=1}^{n} (P_{ix} - \overline{x})(P_{iy} - \overline{y}) & \sum_{i=1}^{n} (P_{iy} - \overline{y})^2 & \sum_{i=1}^{n} (P_{iy} - \overline{y})(P_{iz} - \overline{z}) \\ \sum_{i=1}^{n} (P_{ix} - \overline{x})(P_{iz} - \overline{z}) & \sum_{i=1}^{n} (P_{iy} - \overline{y})(P_{iz} - \overline{z}) & \sum_{i=1}^{n} (P_{iz} - \overline{z})^2 \end{bmatrix} \begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{20}$$

[2]Note that ${A_x}^2 + {A_y}^2 + {A_z}^2$ is a constant. Therefore, this factor can be omitted when taking the derivative and making equal to zero.

A_x, A_y and A_z were defined as the components of *any* director-vector perpendicular to line $\vec{L}$. For convenience, we can set $\vec{A_\perp}$ as an unitary vector, where ${A_x}^2 + {A_y}^2 + {A_z}^2 = 1$. Considering this, the system can be presented in terms of the spherical angles θ and ϕ. The cells in the matrix can all be computed from the data, and its notation is defined as follows to facility its algebraic handling, as follows:

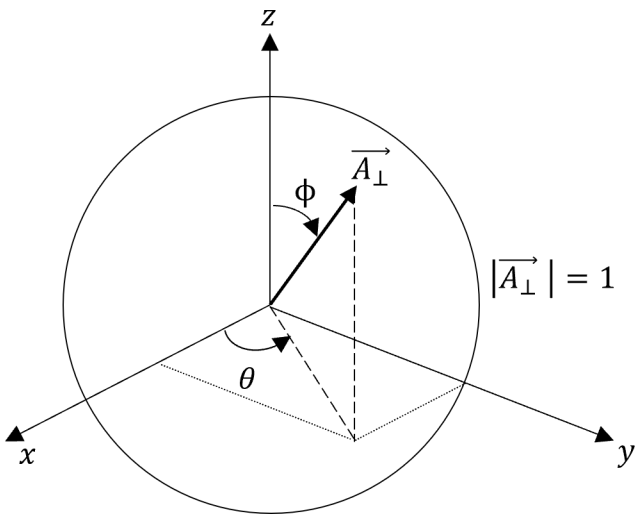

Figure 3: Spherical coordinate system, ISO convention, commonly used in Mathematics.

$$\begin{bmatrix} k_{xx} & k_{xy} & k_{xz} \\ k_{xy} & k_{yy} & k_{yz} \\ k_{xz} & k_{yz} & k_{zz} \end{bmatrix} \begin{bmatrix} \cos\theta \sin\phi \\ \sin\theta \sin\phi \\ \cos\phi \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{21}$$

Multiplying the 1st row by $-\frac{k_{zz}}{k_{xz}}$ (as long as $k_{xz} \neq 0$)

$$\begin{bmatrix} -\frac{k_{xx}k_{zz}}{k_{xz}} & -\frac{k_{xy}k_{zz}}{k_{xz}} & -\frac{k_{xz}k_{zz}}{k_{xz}} \end{bmatrix} \begin{bmatrix} \cos\theta \sin\phi \\ \sin\theta \sin\phi \\ \cos\phi \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Adding row 3)

$$\begin{bmatrix} k_{xz} - \frac{k_{xx}k_{zz}}{k_{xz}} & k_{yz} - \frac{k_{xy}k_{zz}}{k_{xz}} & \cancel{k_{zz} - \frac{k_{xz}k_{zz}}{k_{xz}}} \end{bmatrix} \begin{bmatrix} \cos\theta \sin\phi \\ \sin\theta \sin\phi \\ \cos\phi \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

More simply

$$\begin{bmatrix} k_{xz} - \frac{k_{xx}k_{zz}}{k_{xz}} & k_{yz} - \frac{k_{xy}k_{zz}}{k_{xz}} \end{bmatrix} \begin{bmatrix} \cos\theta \sin\phi \\ \sin\theta \sin\phi \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Multiplying by k_{xz}

$$\begin{bmatrix} {k_{xz}}^2 - k_{xx}k_{zz} & k_{yz}k_{xz} - k_{xy}k_{zz} \end{bmatrix} \begin{bmatrix} \cos\theta \sin\phi \\ \sin\theta \sin\phi \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

In general, $\sin(\phi) \neq 0$, therefore

$$\begin{bmatrix} {k_{xz}}^2 - k_{xx}k_{zz} & k_{yz}k_{xz} - k_{xy}k_{zz} \end{bmatrix} \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\cos(\theta)\left({k_{xz}}^2 - k_{xx}k_{zz}\right) + \sin(\theta)\left(k_{yz}k_{xz} - k_{xy}k_{zz}\right) = 0$$

$$\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)} = \frac{k_{xx}k_{zz} - {k_{xz}}^2}{k_{yz}k_{xz} - k_{xy}k_{zz}} \tag{22}$$

In order to determine the sine and cosine components, we should take into account the signs and angle conventions. $\tan\frac{a}{b} = \tan\frac{-a}{-b}$, in spite of (a, b) and $(-a, -b)$ being in opposite quadrants (the radar vectors are off by π radians). One possible approach is to first compute the angle using a 4-quadrant tangent function, $\arctan(y, x)$, and then calculate the sine and cosines, but doing this would be computationally wasteful, specially in the context of high speed, real-time applications (such as computer graphics or robotics). Given that any of the two normal vectors to the plane $\vec{A}$ and $-\vec{A}$ are suitable, the steps below do not strictly assert the signs, accepting that the expressions may produce either the conventional normal vector or its opposite. It is left to the reader to strictly track the signs and angle conventions when necessary.

$$\sin(\theta) = \frac{k_{xx}k_{zz} - {k_{xz}}^2}{\sqrt{\left(k_{xx}k_{zz} - {k_{xz}}^2\right)^2 + \left(k_{yz}k_{xz} - k_{xy}k_{zz}\right)^2}} \tag{23}$$

$$\cos(\theta) = \frac{k_{yz}k_{xz} - k_{xy}k_{zz}}{\sqrt{\left(k_{xx}k_{zz} - {k_{xz}}^2\right)^2 + \left(k_{yz}k_{xz} - k_{xy}k_{zz}\right)^2}} \tag{24}$$

From the first row of system, we know:

$$k_{xx}\cos\theta\sin\phi + k_{xy}\sin\theta\sin\phi + k_{xz}\cos\phi = 0$$

In general, $\sin(\phi) \neq 0$, therefore

$$k_{xx}\cos\theta + k_{xy}\sin\theta + \frac{k_{xz}}{tan\phi} = 0$$

$$\tan\phi = \frac{-k_{xz}}{k_{xx}\cos\theta + k_{xy}\sin\theta} \tag{25}$$

$$\sin\phi = \frac{-k_{xz}}{\sqrt{{k_{xz}}^2 + \left(k_{xx}\cos\theta + k_{xy}\sin\theta\right)^2}} \tag{26}$$

$$\cos\phi = \frac{k_{xx}\cos\theta + k_{xy}\sin\theta}{\sqrt{{k_{xz}}^2 + \left(k_{xx}\cos\theta + k_{xy}\sin\theta\right)^2}} \tag{27}$$

4.1 Wrapping-up

The angular components $\sin\theta$ and $\cos\theta$ can easily be computed from the data vectors as per the definition of the k_{ij} sums defined in (21), without calling any trigonometric functions. Furthermore, all k_{ij} can be determined using a single loop[3].

$\sin\phi$ and $\cos\phi$ require (in addition to the data vectors), the values for $\sin\theta$ and $\cos\theta$ which were just determined.

The vector normal to the regression plane is, therefore:

$$\begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} = \begin{bmatrix} \cos\theta\sin\phi \\ \sin\theta\sin\phi \\ \cos\phi \end{bmatrix} \tag{28}$$

The solution to (14), (15), and (16) offers a point belonging to the regression plane, which is:

$$\begin{bmatrix} \overline{x} \\ \overline{y} \\ \overline{z} \end{bmatrix} \tag{29}$$

These two elements allow representing the regression plane, in Normal Form, as:

$$\langle \cos\theta\sin\phi,\ \ \sin\theta\sin\phi,\ \ \cos\phi\rangle \cdot \langle x-\overline{x},\ \ y-\overline{x},\ \ z-\overline{z}\rangle = 0 \tag{30}$$

Representation of the plane in other forms, if required, is simple, with abundant literature on the subject. Conversions to the most common forms are included and performed at the end of the included source code.

5 Implementation and Validation for Three-Dimensional variables

The code below is available and can also be executed online, without a local compiler, at:
https://onlinegdb.com/0cp0JNwgE

The software first builds a linear array of three-dimensional points X[i], Y[i] and Z[i], with a length of n^2 elements (this is a grid of n by n elements varying in auxiliary parameters u and v, building data for the plane

$$\Pi : \langle P_x, P_y, P_z\rangle + u\,\langle U_x, U_y, U_z\rangle +, v\,\langle V_x, V_y, V_z\rangle$$

The simulated data incorporates an adjustable amount of noise set by factor s. Given this array, a point $Q : \langle \bar{x}, \bar{y}, \bar{z}\rangle$ is first determined. Secondly, the $\vec{A_\perp}$ is determined by using the expressions outlined in this article.

A few verification checks are then performed. The magnitude or norm of $\vec{A_\perp}$ is printed and should be one. Also, the vector perpendicular to the plane is obtained $\vec{C} = \vec{U} \times \vec{V}$. The cross product of $\vec{C}$ and $\vec{A}$ should be a zero vector or close, depending of the system's accuracy and the amount of noise. Finally, the angle between $\vec{C}$ and $\vec{A_\perp}$ is computed (this value is expected to be close to zero) , using

$$\alpha = \arccos\left[\frac{\vec{C}\cdot\vec{A}}{\|C\|\,\|A\|}\right]$$

[3]The associated C code in this article implements multiple loops in order to improve user's readability.

5.1 Source code for 3D Orthogonal Regression

```
#include <stdio.h>
#include <math.h> /* Needed to calculate square roots */
#include <time.h> /* Needed to seed the random number generator */
#include <stdlib.h> /* Needed for rand() */

/* Copyright (C) 2021, Diego Velez

The code below is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License */

double sum_sqr (double A[], double avg, int n)
{
    int i;
    double sum = 0.0;
    for (i=0; i<n; i++)
        sum += (A[i] - avg) * (A[i] - avg);
    return sum;
}

double pow2 (double x)
{
  return x*x;
}

double sum_prod (double A[], double a_avg, double B[], double b_avg, int n)
{
    int i;
    double sum = 0.0;
    for (i=0; i<n; i++)
        sum += (A[i] - a_avg) * (B[i] - b_avg);
    return sum;
}

double avg (double A[], int n)
{
    int i;
    double sum = 0.0;
    for (i=0; i<n; i++)
        sum += A[i];
    return sum / n;
}
```

```
void cross (double ax,  double ay,  double az,
            double bx,  double by,  double bz,
            double *cx, double *cy, double *cz)
{
    *cx = ay*bz - az*by;
    *cy = az*bx - ax*bz;
    *cz = ax*by - ay*bx;
}

double norm(double ax,  double ay,  double az)
{
    return sqrt(ax*ax+ay*ay+az*az);
}

double randf (double low, double high)
{
  return (rand () / (float) (RAND_MAX)) * fabs (low - high) + low;
}

void main ()
{
    srand (time (NULL));
    const int n=3;
    double X[n*n], Y[n*n], Z[n*n];
    double s = 0.2;     /* variance */

    /* Simulate a plane Plane: P<x,y,z> = u<Ux,Uy,Uz> + v<Vx,Vy,Vz> +*/
    double Px =  0.50;
    double Py =  1.75;
    double Pz = -1.20;

    double Ux =  3.25;
    double Uy =  0.20;
    double Uz = -0.33;

    double Vx = -0.10;
    double Vy =  2.75;
    double Vz =  0.70;

    int i, j;
    double u_min=-5.0;
    double u_max= 5.0;
    double v_min=-5.0;
    double v_max= 5.0;

    for (i=0; i<n; i++)
    {
       double v=v_min+(v_max-v_min)*(double)i/n;
        for (j = 0; j < n; j++)
      {
          double u=u_min+(u_max-u_min)*(double)j/n;
          X[n*i+j] =  Px + u*Ux + v*Vx + s*randf (-1.0, 1.0);
           Y[n*i+j] =  Py + u*Uy + v*Vy + s*randf (-1.0, 1.0);
```

```
        Z[n*i+j] =  Pz + u*Uz + v*Vz + s*randf (-1.0, 1.0);

       printf ("V[%i]= \t<%4.6f, \t%4.6f, \t%4.6f>\n", n*i+j,
       X[n*i+j], Y[n*i+j], Z[n*i+j]);
     }
  }
  double x_avg = avg (X, n*n);
  double y_avg = avg (Y, n*n);
  double z_avg = avg (Z, n*n);

  double k_xx = sum_sqr (X, x_avg, n*n);
  double k_yy = sum_sqr (Y, y_avg, n*n);
  double k_zz = sum_sqr (Z, z_avg, n*n);
  double k_xy = sum_prod (X, x_avg, Y, y_avg, n*n);
  double k_xz = sum_prod (X, x_avg, Z, z_avg, n*n);
  double k_yz = sum_prod (Y, y_avg, Z, z_avg, n*n);

  double sin_tetha = (k_xx*k_zz-k_xz*k_xz)/
       sqrt(pow2(k_xx*k_zz-k_xz*k_xz)+pow2(k_yz*k_xz-k_xy*k_zz));
  double cos_tetha = (k_yz*k_xz-k_xy*k_zz)/
       sqrt(pow2(k_xx*k_zz-k_xz*k_xz)+pow2(k_yz*k_xz-k_xy*k_zz));

  double sin_phi = -k_xz/sqrt(k_xz*k_xz + pow2 (k_xx*cos_tetha + k_xy*sin_tetha));
  double cos_phi = (k_xx*cos_tetha + k_xy*sin_tetha)/
       sqrt(k_xz*k_xz + pow2 (k_xx*cos_tetha + k_xy*sin_tetha));

  double Ax = cos_tetha * sin_phi;
  double Ay = sin_tetha * sin_phi;
  double Az = cos_phi;

  double Cx, Cy, Cz;
  cross (Ux, Uy, Uz, Vx, Vy, Vz, &Cx, &Cy, &Cz);

  printf ("\nTheoretical:");
  printf ("\n\nVector Normal to the plane: U cross V: <%4.8f, %4.8f, %4.8f>",
          Cx, Cy, Cz);


  printf ("\n\nVector Normal to the plane (normalized): <%4.8f, %4.8f, %4.8f>",
         Cx/norm(Cx,Cy,Cz),
         Cy/norm(Cx,Cy,Cz),
         Cz/norm(Cx,Cy,Cz));
  printf ("\n\n\nCalculated:");
  printf ("\n\nVector Normal to the plane: A: <Ax, Ay, Az> = <%4.8f, %4.8f, %4.8f>",
          Ax, Ay, Az);
  printf ("\nMagnitude (should be 1.000) = %4.8f", sqrt (Ax * Ax + Ay * Ay + Az * Az));

  double Dx, Dy, Dz;
  cross (Ax, Ay, Az, Cx, Cy, Cz, &Dx, &Dy, &Dz);

  printf ("\n\nPlane equation in Normal Form:");
  printf ("\n\tA dot (X-Q) = 0");
  printf ("\n\t<%4.4f, %4.4f, %4.4f> dot <X%+4.4f, Y%+4.4f, Z%+4.4f> = 0",
```

```
            Ax, Ay, Az, -x_avg, -y_avg, -z_avg);
   printf ("\n\nPlane␣equation␣in␣Cartesian␣Form:");
   double D=x_avg*Ax + y_avg*Ay + z_avg*Az;
   printf ("\n\t%4.4f*x␣%+4.4f*y␣%+4.4f*z␣=␣%4.4f",
            Ax, Ay, Az, D);
   printf ("\n\nPlane␣equation␣with␣z␣as␣a␣function␣of␣x␣and␣y:");
   printf ("\n\tz␣=␣f(x,y)␣=␣%4.4f*x␣%+4.4f*y␣%+4.4f",
            -Ax/Az, -Ay/Az, D/Az);

   printf ("\n\n\nVerification:");
   printf ("\n\nTheor.␣vector␣cross␣calculated:␣<%4.8f,␣%4.8f,␣%4.8f>", Dx, Dy, Dz);

   double angle = acos((Ax*Cx+Ay*Cy+Az*Cz)/(norm(Ax,Ay,Az)*norm(Cx,Cy,Cz)));
   printf("\n\nAngle␣between␣theoretical␣and␣calculated␣normal␣vectors␣(0␣or␣pi):");
   printf("\n\tangle␣=␣acos[%4.4f/(%4.4f␣*␣%4.4f)]␣%4.4f␣radians␣(%4.4f␣degrees)",
            Ax*Cx+Ay*Cy+Az*Cz, norm(Ax,Ay,Az),norm(Cx,Cy,Cz),angle, 180*angle/M_PI);

   /* A rough, inneficient -yet easy- iterative solution */

   double Ax_it, Ay_it, Az_it;
   double Ax_min, Ay_min, Az_min;
   double phi, theta, delta_theta;
   int n_phi = 180; /* We'll sweep the orientation every 1 degree */
  double min_sum_distance_sqr=0;
   delta_theta=M_PI/n_phi;
  for (i=0; i<n_phi; i++)
  {
     phi=i*M_PI/n_phi;
     /* We use a variable step in theta in order to make it consistent with that of phi */
     if (i!=0)
        delta_theta=0.5*fabs((M_PI/n_phi)/sin(phi));
     else
        delta_theta=0.00001;

     for (theta=0; theta<2*M_PI; theta+=delta_theta)
     {
        Ax_it=cos(theta)*sin(phi);
        Ay_it=sin(theta)*sin(phi);
        Az_it=cos(phi);

        double sum_distance_sqr=0;
        for (j=0; j<n*n; j++)
        {
           double temp=Ax_it*(X[j]-x_avg)+Ay_it*(Y[j]-y_avg)+Az_it*(Z[j]-z_avg);
           sum_distance_sqr+=temp*temp;
        }

        if (i==0)
        {
           min_sum_distance_sqr=sum_distance_sqr;
           Ax_min=Ax_it;
           Ay_min=Ay_it;
           Az_min=Az_it;
```

```
         }
         else
         {
            if (sum_distance_sqr<min_sum_distance_sqr)
            {
               min_sum_distance_sqr=sum_distance_sqr;
               Ax_min=Ax_it;
               Ay_min=Ay_it;
               Az_min=Az_it;
            }
         }
      }
   }
   printf ("\n\n\nIterative␣solution:");

   printf ("\n\nVector␣Normal␣to␣the␣plane␣(iterative):");
   printf ("\n\t<%4.8f,␣%4.8f,␣%4.8f>", Ax_min, Ay_min, Az_min);

   angle = acos((Ax*Ax_min+Ay*Ay_min+Az*Az_min)/(norm(Ax,Ay,Az)*norm(Ax_min,Ay_min,Az_min)));
   printf("\n\nAngle␣between␣calculated␣and␣iterative␣␣normal␣vectors:␣%4.8f␣rad␣(%4.8f␣degrees)
       ", angle,angle*180/M_PI);
   printf("\n\t%4.8f␣rad␣(%4.8f␣degrees)", angle,angle*180/M_PI);

   return;
}
```

5.2 Results

$\vec{A_\perp}$ is confirmed to be unitary and to be orthogonal to plane, which is confirmed by obtaining the theoretical direction vector of the plane $\vec{C} = \vec{U} \times \vec{V}$, and also by obtaining the angle between the theoretical vector $\vec{C}$ and the calculated $\vec{A_\perp}$.

A typical output is as follows:

```
V[0]=   <-15.308801,    -13.120833,     -3.095721>
V[1]=   <-4.536799,     -12.399571,     -4.144136>
V[2]=   <6.402709,      -11.787633,     -5.164318>
V[3]=   <-15.491950,    -3.768682,      -0.631038>
V[4]=   <-4.827151,     -3.160220,      -1.871078>
V[5]=   <6.219981,      -2.343084,      -3.034443>
V[6]=   <-15.814618,    5.484898,       1.616617>
V[7]=   <-4.883876,     5.863204,       0.599545>
V[8]=   <5.671966,      6.700987,       -0.732638>

Theoretical:

Vector Normal to the plane: U cross V: <1.04750000, -2.24200000, 8.95750000>

Vector Normal to the plane (normalized): <0.11271874, -0.24125577, 0.96389322>
```



```
Calculated:

Vector Normal to the plane: A: <Ax, Ay, Az> = <-0.11588611, 0.23905117, -0.96406688>
Magnitude (should be 1.000) = 1.00000000

Plane equation in Normal Form:
        A dot (X-Q) = 0
        <-0.1159, 0.2391, -0.9641> dot <X+4.7298, Y+3.1701, Z+1.8286> = 0

Plane equation in Cartesian Form:
        -0.1159*x +0.2391*y -0.9641*z = 1.5532

Plane equation with z as a function of x and y:
        z = f(x,y) = -0.1202*x +0.2480*y -1.6111


Verification:

Theor. vector cross calculated: <-0.02013713, 0.02818975, 0.00941056>

Angle between theoretical and calculated normal vectors (0 or pi):
        angle = acos[-9.2930/(1.0000 * 9.2930)] 3.1377 radians (179.7787 degrees)


Iterative solution:

Vector Normal to the plane (iterative):
        <-0.11309772, 0.23280078, -0.96592583>

Angle between calculated and iterative  normal vectors: 0.00709213 rad (0.40634908 degrees)
        0.00709213 rad (0.40634908 degrees)
```

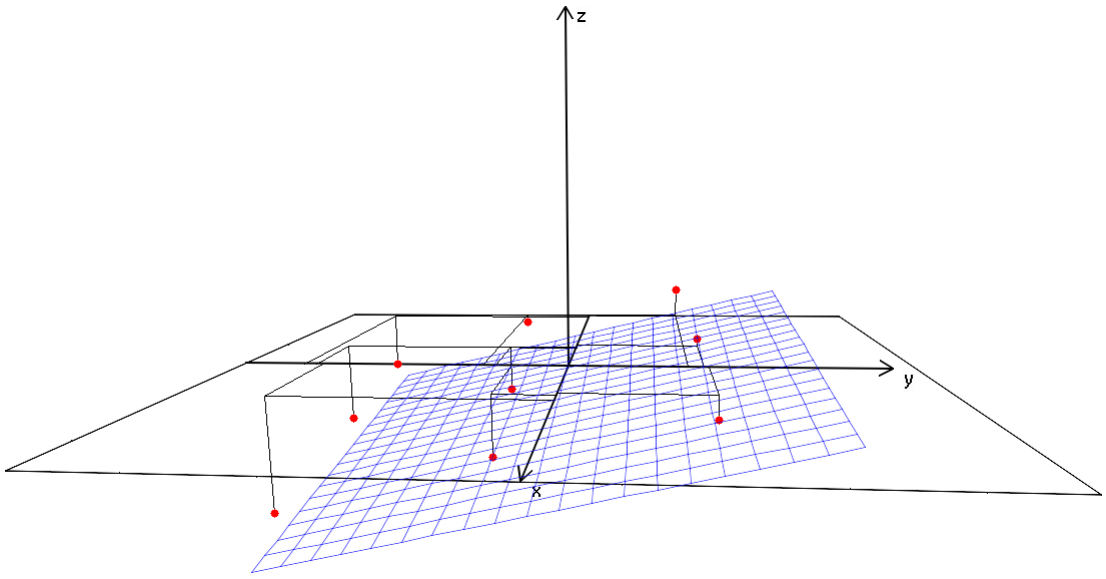

Figure 4: Regression plane, computed from the data points. Points may appear "scattered" due to the coordinate system being arbitrarily oriented.

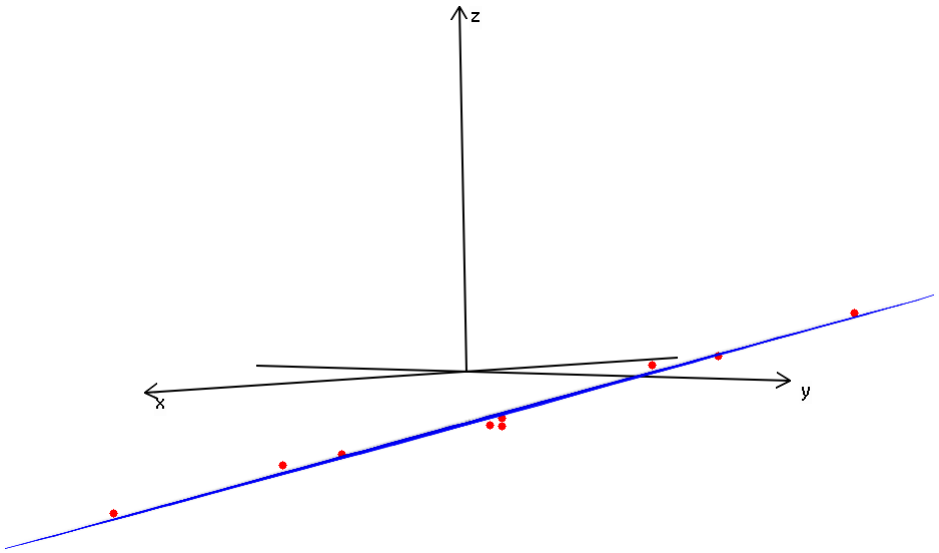

Figure 5: Regression plane, computed from the data points. The view angle has been set to be perpendicular to the plane. Data with low variance ($s = 0.5$), making the points almost co-planar.

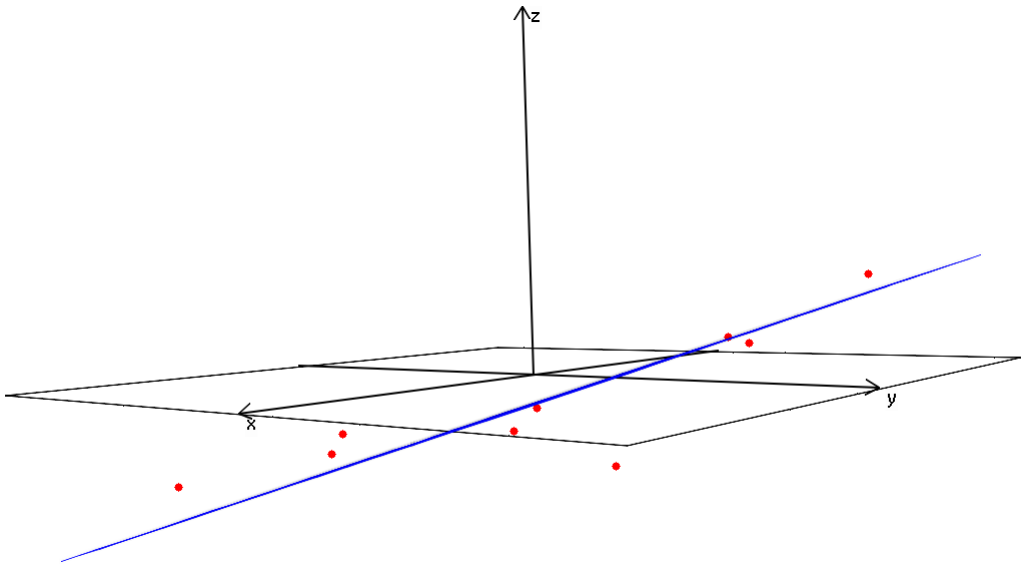

Figure 6: Regression plane, computed from a different set of data points (not printed in this document); this time a higher variance has been used to simulate the data($s = 2.0$). As can be seen, the error from each point to the plane is now significant.

6 Special scenarios

Certain conditions in the data can bring undetermined solutions. For example, in expressions (20) and (21), the characteristic value $\left(k_{xx}k_{zz} - {k_{xz}}^2\right)^2 + \left(k_{yz}k_{xz} - k_{xy}k_{zz}\right)^2$ can be zero when $x = 0$ or $z = 0$ for all data samples. Under such scenario, the problem can be shaped as a two-dimensional regression, with no need to further compute $\sin\phi$, $\cos\phi$. It is left to the reader to implement appropriate error-checking safeguards to handle ill-formed data sets.

7 Conclusions

A direct method for performing orthogonal regression analysis has been presented.

The formulae and associated supporting code allow obtaining the Normal Form representation of the orthogonal regression plane based on real (or simulated) two and three dimensional data.

Not only the outlined solution is simple enough as to be accessible by either intermediate or advanced readers, but it is also highly efficient, as it makes no use of trigonometric or transcendental functions; does not rely on external eigenvalue solvers; it is direct and makes no use of iterative methods. Finally, it also avoids implicitly repeated computations.

Due to this, it concurrently achieves three goals in algorithmic design: *simplicity*, *speediness*, and *lightness in the use of resources*.

8 Bibliography

In spite the abundance of conceptual sources, a practical solution to the orthogonal regression analysis seems to be elusive, as it has not been possible to locate a complete, suitable, and simple implementation. A valuable article outlining both an iterative and a direct solution is:

[1] JACQUELIN, JEAN, *Regressions et trajectoires en 3D, Pages 13-27*, . Retrived from `https://www.scribd.com/doc/31477970/Regressions-et-trajectoires-3D#scribd`

Iterative approaches are not deemed optimal for real-time and high-speed applications, since they rely on successive approximations. Furthermore, they impose on the user the burden of defining appropriate tolerances as well as a maximum number of steps, both of which may be unknown beforehand or may vary according to the application. Iterative systems bring also difficulties when dealing with degenerated data-sets, which can otherwise be easily detected when using direct methods.

JACQUELIN's proposed iterative solution relies on the use of a 3^{rd} degree polynomial, which requires the handling of multiple solutions. Likewise, it relies on trigonometric functions, which impose a toll on speediness.

Other works by the same author, which are (for the most part) related to regression analysis, can be consulted at `https://www.scribd.com/user/10794575/JJacquelin`

