PCA Bounding Volume Boxes

russoev

May 2018

1 What is this / Motivation

This is a small demo showing the use of PCA (Principle Component Analysis) in order to draw what is known as a bounding volume (in this case a box) around a set of random points in 3D space. This is particularly useful in what is referred to in 3D game development as *bounding volume tests*, which is one method for determining if a 3D object in space is currently visible or not.

2 The Covariance Matrix

The PCA relies on a construction known as the covariance matrix, which represents how closely two coordinates vary with each other. For example, if the entry corresponding to two coordinates is zero, there is no correlation between the data for those coordinates. We first arrive at the mean vector for the points, which is essentially the central point

$$\mathbf{m} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{P}_i \tag{1}$$

Next, we use the mean vector to compute the covariance matrix as follows

$$\mathbf{C} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{P}_i - \mathbf{m})(\mathbf{P}_i - \mathbf{m})^T \tag{2}$$

For example, the entry denoting the correlation between the x and y coordinates is calculated by

$$C_{12} = \frac{1}{N}\sum_{i=1}^{N}(x_i - m_x)(y_i - m_y) \tag{3}$$

The correlation isn't affected by the order of the two coordinates, so we are left with a symmetric matrix. One convenient property of symmetric matrices is that they always have N real eigenvalues, N being the dimension of the NxN matrix. Our goal is to *diagonalize* the covariance matrix, so we are left with three principal uncorrelated axes that we can use as the basis for our transformation.

Fortunately, because the correlation matrix is symmetric, the eigenvectors are orthogonal and can serve as the transformation matrix we are looking for:

$$\mathbf{C}' = \mathbf{A}\mathbf{C}\mathbf{A}^T \tag{4}$$

3 Jacobi Iterative Method

In my program, I used the Jacobi Iterative method to compute the eigenvalues and vectors for the 3 by 3 covariance matrix (although only the eigenvectors are needed this demonstration). The Jacobi Iterative method works based off the principle that symmetric square matrices can always be diagonalized by a matrix with orthogonal columns (its eigenvectors). At every step of the iteration, one of the non-zero off-diagonal entries is "eliminated" through a decomposition with a special matrix.

$$\mathbf{M}_k = \mathbf{R}_k^T \mathbf{M}_{k-1} \mathbf{R}_k \tag{5}$$

Once enough iterations are performed, the off diagonal entries tend to zero, and the resultant diagonal matrix represents equals the eigenvalues, and the product of the transpose matrices used at every step is the matrix of eigenvectors.

The special matrix used in the decomposition is known as a Givens Rotation Matrix. An example of a 2x2 Givens Matrix is

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & cos\theta \end{bmatrix} \tag{6}$$

You may recognize this as a matrix denoting a counterclockwise rotation by angle θ. Notice how the columns form an orthonormal basis of R^2. If we carefully select an angle θ, then we can use this matrix in our eigenvalue decomposition to get a diagonal matrix. In the 3 by 3 case, here are our rotation matrices:

$$\mathbf{R}^{(12)} = \begin{bmatrix} c & s & 0 \\ \text{-}s & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{R}^{(13)} = \begin{bmatrix} c & 0 & s \\ 0 & 1 & 0 \\ \text{-}s & 0 & c \end{bmatrix} \mathbf{R}^{(23)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & \text{-}s & c \end{bmatrix} \tag{7}$$

Here c is substituted for $\cos\theta$, and s for $\sin\theta$. Using the formula $\mathbf{M}' = \mathbf{R}^{(pq)T}\mathbf{M}\mathbf{R}^{(pq)}$ we can explicitly compute the entries of M' for the next iteration:

$$\left.\begin{aligned} M'_{ii} &= M_{ii} \\ M'_{ip,pi} &= cM_{ip} - sM_{iq} \\ M'_{iq,qi} &= sM_{ip} + cM_{iq} \end{aligned}\right\} \quad \text{if } i \neq p \text{ and } i \neq q;$$

$$M'_{pp} = c^2 M_{pp} + s^2 M_{qq} - 2scM_{pq}$$

$$M'_{qq} = s^2 M_{pp} + c^2 M_{qq} + 2scM_{pq}$$

$$M'_{pq,qp} = sc\left(M_{pp} - M_{qq}\right) + \left(c^2 - s^2\right) M_{pq}$$

All we need to do now is work backwards so $M_{pq} = 0$ (the indices p,q and q,p represent the diagonal entries we want to get rid of at this step), so we must find an angle θ that satisfies

$$\frac{c^2 - s^2}{sc} = \frac{M_{pp} - M_{qq}}{M_{pq}} \tag{8}$$

Using the trigonometric identities

$$\sin 2\alpha = 2 \sin\alpha \cos\alpha \quad \cos 2\alpha = \cos^2\alpha - \sin^2\alpha \tag{9}$$

We set

$$u = \frac{1}{\tan 2\theta} = \frac{c^2 - s^2}{sc} = \frac{M_{pp} - M_{qq}}{M_{pq}} \tag{10}$$

While you could just use inverse tangent at this point to calculate the angle and plug it back in for sin and cosine, there is a simpler method by noting that

$$t^2 + 2ut - 1 = 0 \tag{11}$$

By the quadratic formula we have

$$t = \text{-}u \pm \sqrt{u^2 + 1} \tag{12}$$

Now, the trig identity $t^2 + 1 = 1/c^2$ easily leads us to computed values for s and c

$$c = \frac{1}{\sqrt{t^2 + 1}} \qquad\qquad s = ct \tag{13}$$

It can be shown that by repeatedly using the formulas above for the next matrix in the iteration, the off-diagonal entries will converge to zero and we will be left with a diagonal matrix and the eigenvectors we're after.

4 Building a Box

We now have the eigenvectors for the covariance matrix, which represents a basis for an uncorrelated set of axes that theoretically distributes the point data equally. Our next step towards a box is to compute the planes that make up the faces of the box. The planes are determined by the *extrema* of the data, which can be simply computed by getting the maximum and minimum dot products of all our points with our eigenvectors.

A plane can be represented by a normal vector and an offset D, such that

$$Ax + By + Cz + D = 0 \tag{14}$$

Another way to put it is that for any point on the plane **P**

$$\mathbf{N} \cdot \mathbf{P} + D = 0 \tag{15}$$

Or for short, $\langle \mathbf{N}, D \rangle$. The extrema represent our D values for our box planes in this case. If our eigenvectors are denoted as $\mathbf{R}, \mathbf{S}$, and $\mathbf{T}$ we're left with the six plane equations

$$\langle \mathbf{R}, \text{-}\mathit{minDotR} \rangle \qquad \langle \textbf{-R}, \mathit{maxDotR} \rangle \tag{16}$$

$$\langle \mathbf{S}, \text{-}\mathit{minDotS} \rangle \qquad \langle \textbf{-S}, \mathit{maxDotS} \rangle \tag{17}$$

$$\langle \mathbf{T}, \text{-}\mathit{minDotT} \rangle \qquad \langle \textbf{-T}, \mathit{maxDotT} \rangle \tag{18}$$

The values such as minDotR represent the minimum dot product of **R** with any point **P**, the others are similarly determined.

All I needed to do now in order to draw the box was compute its vertices, which is as simple as finding the exact point where 3 of the cube faces intersect. In other words, given three plane equations, we just need to solve the simple system

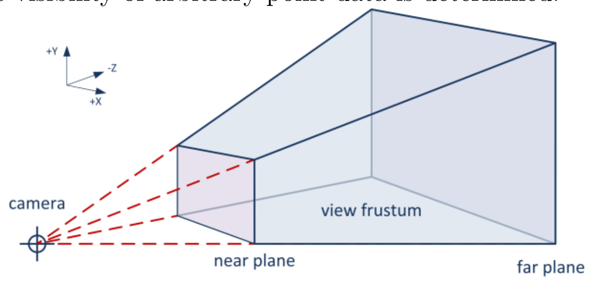
$$\begin{bmatrix} (\mathbf{N}_1)_x & (\mathbf{N}_1)_y & (\mathbf{N}_1)_z \\ (\mathbf{N}_2)_x & (\mathbf{N}_2)_y & (\mathbf{N}_2)_z \\ (\mathbf{N}_3)_x & (\mathbf{N}_3)_y & (\mathbf{N}_3)_z \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} -\mathbf{D}_1 \\ -\mathbf{D}_2 \\ -\mathbf{D}_3 \end{bmatrix} \tag{19}$$

To do this in javascript I used an explicit formula for computing 3 by 3 matrix inverses given by multiplying the transpose of the cofactor matrix by the reciprocal of the determinant.

Hooray, we now have a box around our point data.

5 Addendum

You may still be wondering why this is useful for determining object visibility in a 3D rendering scenario. Well, boxes aren't the only bounding volume constructed on point data, there are also spheres, ellipsoids, cylinders, and others I'm sure. What you're seeing when you play a 3D game is all determined by a 3D object called the *view frustrum*, a sort of truncated cone that transforms 3D objects in absolute world space to a realistic ”view space” by taking into account perspective and field-of-view. What bounding volume tests do is test for intersection between simple geometric entities (boxes for instance) with the frustrum, and only display them if the test passes. This is the essence of how the visibility of arbitrary point data is determined.



6 References

Most of all the math I incorporated into this project is taken from Eric Lengeyel’s excellent and comprehensive primer on 3D mathematics in video games:

Mathematics for 3D Game Programming and Computer Graphics. The 3D plotting library is courtesy of http://plotly.org. The cat, car, and house were all originally free 3D models from http://free3d.org. The remainder of the code and implementation is my own.