THE MONACO VARIATION ON PRINCIPAL COMPONENT ANALYSIS

A Novel Procedure for processing Multi-Variate Data

Kevin R. Lawrence

March, 2020

Table of Contents

I. Abstract

Principal Component Analysis (PCA) is a useful method of summarizing multi-dimensional datasets. PCA can take a dataset that may be extraordinarily or unnecessarily large and compose a new, smaller, dataset that optimally summarizes the original. It does this by combining redundant features that measures redundantly related properties. Consider the following simple hypothetical example; imagine a dataset that aggregated characteristics across all types of cars, there would be things like height, weight, length, color, gas mileage, and a number of other properties. Out of that dataset, across all cars, certain characteristics will begin to stick together. Perhaps as the height and length increase, as does the weight, and gas mileage inversely. It would certainly be useful to have one variable that could summarize all those features. Taking all those variables and combining them in a way that is based on the amount of variance each contributes to the dataset that we can call "compactness". PCA looks for these properties that show as much variation across the entire dataset, and construct variables that best represent those properties with as little redundancy as possible.

Constructing these variables is only half of the process of PCA, once the variables have been generated, there is still the question of "How much of the variance is needed to significantly summarize the data?" To answer this question, a set of stopping rules is needed. These are rules for deciding a non-trivial amount of principal components to keep. Dr. Pedro R. Peres-Neto et al. completed a comprehensive comparison of existing stopping rules which determined the most useful methods to be. The purpose of this paper is intended to both validate the findings of Dr. Peres-Neto, and lay out a new variation process, meta to the PCA algorithm, for selecting the most efficient stopping process for a particular dataset.

This variation was designed to be utilized in audio-fingerprinting for discrete signal analysis.

II. Statistical Background

In order to discuss how PCA is performed, a bit of mathematical context is necessary. These functions are multi-purpose and will be recycled throughout the algorithm. For accessibility, these will packaged together in a statistical suite (statistics.cpp).

A. Transpose

It is often helpful to arrange the data in a two-dimensional tabular format. If the data is organized by the independent variable (i), which it often is, it may be necessary to reformat the table such that it is in order according the dominant variable (j). The simplest way to reformat a matrix in such a way, is to take each variable at position X_{ij} and move it to position X_{ji}.

```
vector<vector<double> > TRANSPOSE(vector<vector<double> > s) {
    vector<vector<double> > st; //Vector to hold transpose of s
    int spop = s.size(); //Population size
    int ssamp = s[0].size(); //Sample size

    for(int i = 0; i < ssamp; i++){
        vector<double> str; //Row for transpose matrix.
        for(int j = 0; j < spop; j++){
            str.push_back(s[j][i]); //Swap (i,j) to (j,i);
        }
        st.push_back(str); //Aggregate in st.
    }
    return st; //Return transpose.
}
```

B. Mean

The sum of all values for a variable divided by the number of values.

$$\mu_x = \sum_{i=1}^{n} x_i / n$$

```
double MEAN(vector<double> s){
    double nsize = s.size(); //Sample size.
    return accumulate(s.begin(), s.end(), 0.0)/nsize; //Average
}
```

C. Standard Deviation

The average distance from the mean of the dataset to a point.

$$\sigma_x = \sqrt{\sum_{i=1}^{n}(x_i - \mu_x)^2 / (n-1)}$$

```
double STANDARD_DEVIATION(vector<double> s){
    double m = MEAN(s); //Mean of sample
    double nsize = s.size();  //Sample size
    vector<double> stddev; //Vector to hold standard deviation.

    for(int i = 0; i < nsize; i++){
        stddev.push_back(pow((s[i]-m), 2)); //sample minus mean squared.
    }
    ///Standard deviation.
    return sqrt(accumulate(stddev.begin(), stddev.end(), 0.0)/(nsize-1));
}
```

D. Standardization

Ensures that each variable has a standard deviation of 1 and a mean of 0. For each value, subtract the mean and divide by the standard deviation.

$$X_{std} = {(x_i - \mu_x)}/{\sigma_x}$$

```
void STANDARDIZE(vector<vector<double> > &s){
    double n_pop = s.size(); //Population size
    double n_sample = s[0].size();  //Sample Size
    vector<vector<double> > st = TRANSPOSE(s); //Transpose data into columns.
    vector<vector<double> > standardized_matrix; //to hold new standardized matrix

    for(int i = 0; i < n_sample; i++){
        double mean_st = MEAN(st[i]); //Mean of individual vector
        double stddev_st =  STANDARD_DEVIATION(st[i]); //Corresponding Standard Deviation
        vector<double> std_row; //Rows of new matrix.

        for(int j = 0; j < n_pop; j++){
            std_row.push_back((st[i][j] - mean_st)/stddev_st); //Standardize dataset.
        }
        standardized_matrix.push_back(std_row); //Aggregate rows.
    }
    s = standardized_matrix; //Reflect in referenced vector.
}
```

E. Dot-Product

Takes two equal length sequences and returns the sum of the products of the corresponding entries.(Product of the Euclidian magnitudes).

```
double DOT_PRODUCT(vector<double> a, vector<double> b){
    if(a.size() != b.size()){
        cout<<"Error! Bad data sizing when calculating dot products. . .
            Eleanor was forced to exit. "<<endl; //Exception handling.
        exit(0);
    }
    int sizer = a.size(); // Equal to b.size()
    vector<double> accum; //To hold dot products.

    for(int i = 0; i < sizer; i++){
        accum.push_back(a[i] * b[i]);  //Dot product.
    }
    return(accumulate(accum.begin(), accum.end(), 0.0));  //Return accumulated sum.
}
```

F. Covariance

Measure of how much the variables vary from mean with respect to each other.

$$COV(x,y) = \sum_{i=0}^{n}(x_{std} * y_{std}) \Big/ (n-1)$$

G. Covariance Matrix

A square matrix given by the covariance between each pair of elements in the dataset. This element will serve as the input for the next portion of the algorithm.

```
vector<vector<double> > COVARIANCE_MATRIX(vector<vector<double> > s){
    double n_sample = s[0].size(); //Sample size
    STANDARDIZE(s); //Standardize matrix.
    vector<vector<double> > covariance_final; //Matrix to hold covariance

    for(int i = 0; i < n_sample; i++){
        vector<double> cov_r; //Covariance row.

        for(int j = 0; j < n_sample; j++){
            double dot = DOT_PRODUCT(s[i], s[j]); //Dot product of components
            cov_r.push_back(dot/(n_pop-1)); //Covariance Calculation.
        }
        covariance_final.push_back(cov_r); //Aggregate rows
    }
    return covariance_final;
}
```

H. Moduli

The sum of the moduli (in this case, absolute-value) of the supra-diagonal elements in the matrix will be used in calculating threshold values for pivots during the first three sweeps of Jacobi Rotations. (See III.A, III.C, V.A)

```
double MODULI(vector<vector<double> > cm){
    double order = cm.size();  //Order of covariance matrix.
    double modulus  = 0.0; //Modulus value.

    for(int i = 0; i < order; i++){
        for(int j = i+1; j < order; j++){
            modulus += fabs(cm[i][j]); //Aggregate modulus values.
        }
    }
    return modulus; //Return value.
}
```

I. Diagonals

The diagonal of a matrix is a vector of the elements running from the top right to the bottom left. The values in the matrix other than its diagonal are other values are collectively called **off-diagonal** elements. Of the off-diagonal elements, those above the diagonal are called the **supra-diagonal** elements**.** This terminology will be used throughout the paper.

```
vector<double> DIAGONAL(vector<vector<double> > cm){
    int order = cm.size(); //Order of covariance matrix
    vector<double> diag; //Vector to hold diagonal elements

    for(int i = 0; i < order; i++){
        diag.push_back(cm[i][i]); //Add diagonal elements
    }
    return diag;
}
```

J. Identity Matrix

Also called a unit matrix, the identity matrix is a diagonal of ones in a square matrix of zeros. It produces a product equal to any matrix of that it is multiplied by. This analogus to the way that the product of any number and one is itself.

```
vector<vector<double> > IDENTITY_MATRIX(int n){ //Returns an identity matrix of order n.
    vector<vector<double> > identity; //Matrix vector to return.
    for(int i = 0; i < n; i++){
        vector<double> irow;  //Row to fill vector.
        for(int j = 0; j < n; j++){
            double ident = i == j ? 1.0 : 0.0; //1 if diagonal, else 0.
            irow.push_back(ident); //Accumulate in row.
        }
        identity.push_back(irow); //Accumulate in matrix vector.
    }
    return identity; //Return identity matrix.
}
```

K. Matrix Inversion (Gauss-Jordan Method)

The invert of a single variable X, is the number which, when multiplied by X, equals one

$$X \times X^{-1} = 1$$

The invert of a matrix of variables size p x q, A_{pq} , can be defined as the matrix which when multiplied by A_{pq} equals an identity matrix of size p x q.

$$A_{pq} * A_{pq}^{-1} = ident_{pq}$$

The equation above has only one unknown variable, and can be rearranged so that it is solvable by means of Jordan-Gauss reduction. The steps go as follows:
Generate an identity matrix of size p x q (A_{pq}^{*}) use it alongside invertible matrix A_{pq}
Perform elementary operations on both matrices according to the following rules:

Swapping two rows of A_{pq} multiplies A_{pq}^{*} t by -1
Multiplying a row of A_{pw} by a nonzero scalar multiplies A_{pq}^{*} by the same scalar
Adding to one row of A_{pq} a scalar multiple of another A_{pq} row does not change A_{pq}^{*}.
After the transformation is complete, A_{pq}^{*} will be the invert of A_{pq}

```
void INVERSE(vector<vector<double> > &c){
    int order = c.size(); //Order of matrix to invert.
    if(order == 1){
        c[0][0] = 1.0/c[0][0]; //Return inversion of single-unit matrix.
        return;
    }
    vector<vector<double> > cid = IDENTITY_MATRIX(order); //Create an indentity matrix.

    for(int i = 0; i < order; i++){
        double one = c[i][i]; //Check diagonal for value.
        if(one != 1){
            //Swap vectors if one == 0 and there is an available vector to swap with.
            if(one == 0 && i < order-1){
                vector<double> swapee = c[i+1]; //set swap vector.
                c[i+1] = c[i]; //Set swapped vector.
                c[i] = swapee; //Set remaining vector with swap vector.
                i = i-1; //Reset iteration.
                continue;
            }
            //End if one == 0 and no remaining vectors.
            if(one == 0 && i == order-1){
                continue;
            }
            for(int k = 0; k < order; k++){
                cid[k][i] = cid[k][i]/one; //Accumulate corrections in identity matrix.
                c[k][i] = c[k][i]/one; //Force value to 1.
            }
        }
        for(int j = 0; j < order; j++){
            if(i == j){continue;} //Redundant fail-safe.
            double val = c[i][j]; //Off-diagonal elements
            if(val == 0){continue;} //If value is already 0, continue.
            for(int k = 0; k < order; k++){
                cid[k][j] = cid[k][j] - val*cid[k][i];//Accumulate corrections in identity.
                c[k][j] = c[k][j] - val*c[k][i]; //Force value to 0.
            }
        }
    }
    c = cid; //Reflect inversion in referenced vector.
    return;
}
```

III. Eigenvectors and Eigenvalues

The covariance matrix is both orthogonal and symmetric, which means that it is equal to its transpose. Performing orthogonal transformations (Jacobi rotations) around pivot values produces a new matrix that is similar to (in the same direction as) the original. This particular implementation of Jacobi's Algorithm is based off of Rutishauser's modification to the original which is now commonly known as the "LR Algorithm" These rotations are applied iteratively by means of row-wise scanning on values greater than a selective threshold. Each iteration of these rotations is called a sweep. After the first three sweeps, the threshold is set to zero. Only the components above the diagonal need to be taken into consideration due to the symmetry of the input matrix from II.F. This preserves the data values destroyed during the rotations in the infra-diagonal elements. The diagonal elements are then the approximate eigenvalues for the matrix. The following concepts apply at the sweep-level of the algorithm, the following section will address the rotation-level.

A. Initial Threshold

A threshold filter can be used for the first few iterations to increase the efficiency of the algorithm, however it should eventually be removed order to ensure convergence. The first three rounds will use a threshold value given by:

$$thresh = 0.2 * \frac{sm^2}{n}$$

i. Where sm represents the sum of the moduli of the super-diagonal elements
ii. And n represents the number of super-diagonal elements.

B. Convergence

Rutishauser's variation on Jacobi's Algorithm made the process fool-proof. As such this implementation is designed such that no termination limit is required, it will proceed for as long as it is necessary. However, it is absolutely reasonable to be skeptical of anyone who describes their method as "fool-proof," as such, a safeguard has been added as an input to the function.

C. Termination

The procedure will inevitably reach convergence with all super-diagonal elements 0, where upon the process will be terminated. Checking for to determine convergence bundles nicely with the threshold calculation.

```
bool check_convergence(jacobi_transform &jf){
    double order = jf.covariance_matrix.size();  //Order of covariance matrix
    double thold = MODULI(jf.covariance_matrix);  //Modulus of covariance matrix
    thold = 0.2 * pow(thold/order, 2);  //Rutishauser's threshold formula
    jf.thresh = thold;  //Set threshold.
    return thold == 0.0 ? false : true; //Return boolean.
}
```

Notes:
(i) See section II.H for MODULI function
(ii) See section V.A for implementation.

IV. Preparations for the Method.

A. Data Structure Definitions

Rutishauser's variation on Jacobi's Method is an iterative process has lots of moving parts, wound together like clockwork. It would be helpful to have a couple of structures so that it is easier to keep track of all the components as they're being modified.

```
struct pivot{
    int I; //Pivot column
    int J; //Pivot row
};

struct jacobi_transform{
    vector<vector<double> > orig_data; // Saved copy of original data
    vector<vector<double> > eigenvectors; //OUTPUT Eigenvectors
    vector<vector<double> > covariance_matrix; //Working Covariance Matrix
    vector<double> eigenvalues; //OUTPUT Eigenvalues
    double thresh; //Working Threshold
    int it_count; //Working Iteration Count
    int it_max; //INPUT Const M
    vector<double> n_ev; //Transfer vector (donor)
    vector<double> o_ev; //Intermediary vector (acceptor)
    pivot p; //Working pivot

};
```

B. Identity Matrix Initialization.

An identity matrix (also called a unit matrix) is a matrix composed of zeros with ones along the diagonal. It is matrix equivalent of the number one as it does not change any element of another matrix when combined with it.

```
void init_identity(jacobi_transform &jf){
    int order = jf.covariance_matrix.size(); //Order of covariance matrix
    vector<vector<double> > seigen = IDENTITY_MATRIX(order); //Create identity matrix
    jf.eigenvectors = seigen; //Set identity matrix
    return;
}
```

C. Transformation Structure Initialization

The Jacobi method does not actually operate on the covariance matrix(jm.covariance_matrix), but instead, transfers them at the beginning of the process into an eigenvalue vector (jm.eigenvalues) and performs the operations on jm.eigenvalue[i] instead of jm.covariance_matrix[i,i]. A number of other variables need to be initialized as well before the row-sweeps begin, which can be added to the kernel of the algorithm as seen below.

```
jacobi_transform eigenvalue_calc(vector<vector<double> > c, int MAX_ITERATIONS){
    int ITERATION_COUNT = 0;//Counter for iterations
    int order = c[0].size(); //Number of dimensions in the dataset
    vector<double> nev(order, 0.0); // initialize an empty vector to hold corrections

    jacobi_transform jf; //create a jacobi_transformation structure
    jf.it_max = MAX_ITERATIONS; //Set max value for iterations.
    jf.orig_data = c; //save a copy of original data
    jf.covariance_matrix = COVARIANCE_MATRIX(c); //Calculate covariance matrix.
    init_identity(jf);//initialize identity matrix.
    jf.eigenvalues = get_diagonal(jf.covariance_matrix); //set initial eigenvalues
    jf.o_ev = jf.eigenvalues; //keep a copy of initial eigenvalues
    jf.n_ev = nev; //set the correction vector.

    do{
        ITERATION_COUNT++;//Count the iterations
        jf.it_count = ITERATION_COUNT; //set the iteration count
        if(check_convergence(jf) == false){break;} //Check for convergence
        if(ITERATION_COUNT > 3){jf.thresh = 0;} //Check threshold
        jacobi_rotation(jf); //perform the rotation

    }while(ITERATION_COUNT < MAX_ITERATIONS);

    return jf;
}
```

V. Jacobi Rotations

A. Eliminating Small Off-Diagonal Elements

If the pivot element a[i, j] is small compared to a[i, i] and small compared to a[j, j], then a[i, j] is set to zero and the p, g-rotation is skipped. In computer code, it is necessary to use epsilon values for comparisons in order to minimize roundoff errors. In c++ (and other like languages), the "==" will have sufficient precision when coupled with an epsilon coefficient of 100. These omissions produce no more error than had the rotation been performed, however they should be suppressed for the first three iterations in order that it can be used on perturbed diagonal matrices For the first three iterations, the threshold value (from section III.C) will be used to determine the eligibility of the pivot.

```
bool pivot_check(jacobi_transform &jf, double &s){
    s = 100.0 * fabs(jf.covariance_matrix[jf.p.I][jf.p.J]);//epsilon value
    double epi = s + fabs(jf.eigenvalues[jf.p.I]); //i + epsilon
    double epj = s + fabs(jf.eigenvalues[jf.p.J]);//j + epsilon

    if(jf.it_count > 3 && epi == fabs(jf.eigenvalues[jf.p.I]) &&
       epj == fabs(jf.eigenvalues[jf.p.J])){
        jf.covariance_matrix[jf.p.I][jf.p.J] = 0.0; //Annihilate tiny off-diagonal elements
        return false;
    }
    else if(fabs(jf.covariance_matrix{jf.p.I}[jf.p.J]) < jf.thresh &&
            jf.it_count < 3){ //Compare to threshold value for first three.
        return false;
    }
    return true;
}
```

B. Preparing for the Rotation

Each iteration of the algorithm processes the values above the diagonal for every row of the covariance matrix. Each pivot is checked using the method above (V.A), if the function returns true, it proceeds to set up for the rotation. This algorithm uses Rutihshauser's variation of Jacobi's algorithm which uses the following formulae to determine the elements of the covariance matrix. Consider covariance matrix cov[n, n], with diagonal separated as vector eigen[n] for pivot element p[i,j] for the (i x j)th rotation of a given sweep.

$$\Delta = eigen[j] - eigen[i]$$

$$\varepsilon = 100\ x\ cov\{i,j]$$

$$tan\vartheta\ (t) = \begin{cases} \dfrac{cov[i,j]}{\Delta} & abs(\Delta) \approx abs(\Delta) + \varepsilon \\ \vartheta = 0.5 * \Delta / cov[i,j] \begin{cases} \dfrac{1}{abs(\vartheta) + \sqrt{1+\vartheta^2}} & \vartheta > 0 \\ -\dfrac{1}{abs(\vartheta) + \sqrt{1+\vartheta^2}} & \vartheta < 0 \end{cases} \end{cases}$$

$$cos\vartheta\ (c) = \frac{1}{1+t^2}$$

$$sin\vartheta\ (s) = \frac{t}{1+t^2}$$

$$\tau = \tan(\vartheta/2) = \frac{s}{1+c}$$

C. Updating the Diagonal Elements.

Once the values above (V.B) have been calculated, Rutishauser's algorithm updates the eigenvector (eigen[n]) element at indices i and j.

$$eigen[i] = eigen[i] - t * cov[i,j]$$

$$eigen[j] = eigen[j] + t * cov[i,j]$$

Notes:

(i) These values are accumulated in vector n_ev which is initialized with zeros.

(ii) n_ev is aggregated in o_ev then reset at the beginning of each iteration (see section V.F for details)

D. Annihilating Pivot

The termination of the loop is dependent on the elimination of the matrix's off-diagonal elements. Now that the values have been extracted, the matrix is ready for rotation and pivot can be set to zero.

$$cov[j,i] = 0$$

E. Updating the Off-Diagonal Elements

Both dimensions of the upper off-diagonal elements (k) of matrix cov[n,n] need to be updated at pivotal indices j and i.

$$cov[i,k] = cov[i,k] - s \times (cov[j,k] + \tau \times cov[i,k])$$

$$cov[j,k] = cov[j,k] + s \times (cov[i,k] - \tau \times cov[j,k])$$

$$cov[k,j] \equiv cov[j,k]$$

$$cov[i,k] \equiv cov[k,i]$$

F. **Accumulating Matrix Corrections**

Each rotation updates the values for jacbobi_transform.n_ev, which is aggregated into jacobi_transform.o_ev after each sweep, and then used to generate the new values and better values for jm.eigenvalues for the next sweep.

```
void jacobi_rotation(jacobi_transform &jf){
   int order = jf.covariance_matrix.size();

    for(int i = 0; i < order; i++){
        for(int j = (i+1); j < order; j++){
            double epsilon = 0; //Declare an epsilon value
            jf.p.I = i; // set column of current sweep
            jf.p.J = j;  // set row of current sweep

            if (pivot_check(jf, epsilon) == true){
                double delta = jf.eigenvalues[j] - f.eigenvalues[i];// difference of diagonals
                double theta = 0.5 * (delta) / jf.covariance_matrix[i][j]; //angle of rotation
                double tangent = ((fabs(delta) + epsilon) == fabs(delta)) ?
                    jf.covariance_matrix[i][j]/delta : theta < 0 ?
                    -1.0/(fabs(theta) + sqrt(1 + pow(theta, 2))) :
                    1.0/(fabs(theta) + sqrt(1 + pow(theta, 2))); //Tangent of theta
                double cosine = 1.0 / sqrt ( 1.0 + pow(tangent, 2)); //cosine of theta
                double sine = tangent * cosine; // sine of theta
                double tau = sine / ( 1.0 + cosine ); //tan of (theta/2)
                delta = tangent * jf.covariance_matrix[i][j]; //update delta

                //accumulate changes to diagonal elements
                jf.n_ev[i] = (double) jf.n_ev[i] - delta;
                jf.n_ev[j] = (double) jf.n_ev[j] + delta;
                jf.eigenvalues[i] = (double) jf.eigenvalues[i] - delta;
                jf.eigenvalues[j] = (double) jf.eigenvalues[j] + delta;
                jf.covariance_matrix[i][j] = 0.0; //anihilate pivot from covariance matrix.

                //perform the rotation
                for (int k = 0; k < i; k++){
                    double g = jf.covariance_matrix[k][i];
                    double h = jf.covariance_matrix[k][j];
                    jf.covariance_matrix[k][i] = g - sine * (h + g * tau);
                    jf.covariance_matrix[k][j] = h + sine * (g - h * tau);
                }
                for (int k = i + 1; k < j; k++){
                    double g = jf.covariance_matrix[i][k];
                    double h = jf.covariance_matrix[k][j];
                    jf.covariance_matrix[i][k] = g - sine * (h + g * tau);
                    jf.covariance_matrix[k][j] = h + sine * (g - h * tau);
                }
                for (int k = j + 1; k < order; k++){
                    double g = jf.covariance_matrix[i][k];
                    double h = jf.covariance_matrix[j][k];
                    jf.covariance_matrix[i][k] = g - sine * (h + g * tau);
                    jf.covariance_matrix[j][k] = h + sine * (g - h * tau);
                }
                //accumulate information in eigenvectors matrix.
                for (int k = 0; k < order; k++){
                    double g = jf.eigenvectors[k][i];
                    double h = jf.eigenvectors[k][j];
                    jf.eigenvectors[k][i] = g + sine * (h - g * tau);
                    jf.eigenvectors[k][j] = h - sine * (g + h * tau);
                }
            }
        }
    }
    for (int i = 0; i < order; i++ ){
        jf.o_ev[i] = jf.o_ev[i] + jf.n_ev[i]; //aggregate the corrections
        jf.eigenvalues[i] = jf.o_ev[i]; //transfer the corrections to eigenvalues
        jf.n_ev[i] = 0; //reset the vector.
    }
}
```

VI. Preparing to Stop.

The next few sections will focus on various criteria used to determine how many principal components are needed to significantly summarize the dataset. The Monaco Variation is unique in that the technique that is used to determine the stopping rules is dynamic, meaning that the algorithm could stop with a number of different potential principal components. Sorting the data before beginning this process will save a lot of time and code.

A. Addressing the Backup.

Producing the actual values for the principal component requires the initial standardized dataset generated all the way back in II.B. This is the purpose of keeping a copy of the original values in the jacobi_transform structure (jt.orig_data). By repurposing the STANDARDIZE function that was used to generate the covariance_matrix, a backup function can be easily built to transform it into a standardized version of the original data.

B. Zipping the Eigenvalues and Eigenvectors.

With the help of the eigen_unit structure, the tasks of loading and sorting the components can be combined into a single condensed function.

```
void load_components(jacobi_transform jf, vector<eigen_unit> &e_units){
    int order = jf.eigenvectors.size(); //Order of eigenvector matrix
    for(unsigned int i = 0; i < order; i++){
        eigen_unit ev(jf.eigenvectors[i], jf.eigenvalues[i]); //Zip Eigenvalue and Eigenvector
        e_units.push_back(ev); //accumulate zipped unit in reference vector
    }
    return;
}
```

C. Structuring the Component Data

The eigenvalues tell how much variance there is in the direction of the eigenvector. Sorting the eigenvectors by their corresponding eigenvalue organizes the components of the dataset such that the highest eigenvalue is the first principal component, the second-highest, is the second principal component, so on and so forth. For convenience and organization, the eigenvector-eigenvalue combination will be then be loaded into a vector, reordered with a custom sorting operation, and zipped into a structure along with the standardized data.

```
struct eigen_unit{
    vector<double> data; //Original vector of data (used later)
    vector<double> eigenvector; // Corresponding Eigenvector
    double eigenvalue; //Corresponding Eigenvalue
    eigen_unit(vector<double> ev, double eval){
        eigenvector = ev; //Set Eigenvector
        eigenvalue = eval; //Set Eigenvalue
        /*
         No need to set data element, as a copy of the original data will be stored in
         component_data.original_data, the property will be used later for sorting in
         Velicer's MAP criteria calculation.
         */
    }
    struct by_eigenvalue {
        bool operator()(eigen_unit const &a, eigen_unit const &b){
            return a.eigenvalue < b.eigenvalue; //  To sort by eigenvalue
        }
    };
};
```

```
struct component_data {
    vector<eigen_unit> comp; //components
    vector<vector<double> > od;   //original data
}
```

VII. Stopping Criteria

There are various processes that have been developed in order to justify how much of the variance needs to be accounted for in order to properly summarize the data. In order for PCA to be useful, the number of principal components ranges somewhere greater than zero and less than N, the total number of components. The working data gives essentially a rank order of these possible principal components, in terms of the amount of variance that the specific components account for. Dr. Pedro R. Peres-Neto

A. Kaiser-Guttman Criterion

The Kaiser-Guttman Criterion is based on the observation that the average of all eigenvalues is one, and PCA extracts those factors with an eigenvalue greater than this average value. It is based on the arbitrary assumption that factors that are "less than average" are insignificant, which turns out to be a very poor way to determine principal components (in comparison to others). It is one of the worst ways to determine which principal components to keep, however, modern popular statistical software (such as SAS and SPSS) still use the Kaiser-Guttman criterion as the default method. The justification for using this criterion in the Monaco variation is that it is fast, intuitive, widely used, and (theoretically) better than had nothing been done at all. The method may end up being used primarily as a ground for benchmarking comparisons.

```
void Kaiser_Guttman(component_data cd, vector<int> &c){
    sort(cd.comp.begin(), cd.comp.end(), eigen_unit::by_eigenvalue()); //Sort by Eigenvalue
    int csize = cd.comp.size(); //Order of component matrix.

    for(int i = 0; i < csize; i++){
        if(cd.comp[i].eigenvalue>=1.0){ //"If greater than most"
             c[i] = 1; //Reflect in referenced vector.
        }
    }
    return;
}
```

B. Horn's Parallel Analysis (Monte-Carlo Simulations)

John Horn was a cognitive psychologist that created a psychometric method of determining the optimal questions for IQ tests so that they best intelligence. He used a method was based on the results of multiple Monte-Carlo simulations (MCS). A MCS-based method uses random sampling, along with the law of large numbers and time, can produce the answer to any question with a probabilistic answer. Consider the following:

> Suppose the probability of a coinflip ending either heads or tails is unknown, and a researcher decides to determine the value using the **Monte-Carlo Method.** He could do so by dumping a box containing a large number of coins on the floor and counting how many are heads or tails.
>
> The researcher could have come to the same value by *simulating* this process. The **Simulation** would generate a random number between one and zero, and determine each outcome by a preset assignment (i.e. heads == 0, tails == 1).
>
> Then the researcher could run a **Monte-Carlo Simulation** which would be running the simulation for a large number of times and aggregating the return values, and averaging the results.

Horn took the idea of the Monte-Carlo method and applied it to the problem of PCA stopping. The process used in the Monaco variation is an implementation based off the original procedure by John Horn, it goes as follows:

(1) Generate random values respecting the original dimensions of the data matrix.
(2) Standardize the matrix of generated values.
(2) Calculate the eigenvalues for the standardized matrix.
(3) Repeat steps 1 -3 a total of 1000 time, retaining the eigenvalues.
(4) Retain the components greater than the mean from the generated variates.

Notes:

(i) The procedure takes a flag argument because it shares a function with the Buja's_Modification which uses the same simulation kernel, to perform Horn's Parallel analysis use flag = 0.

(ii) Information on the Buja's Modification (flag = 1) can be found in the following Section (VII.C).

```
/*
flag == 0 for Horn's Parallel Analysis.
flag == 1 for Buja-Eyuboglu Modification.
*/

void Monte_Carlo(component_data cd, vector<int> &c, int flag){
    sort(cd.comp.begin(), cd.comp.end(), eigen_unit::by_eigenvalue()); //Sort eigenvalues
    int order = cd.comp.size(); //Order of component matrix.
    int Y = cd.original_data.size(); //Original data size.
    int SIM_COUNT = 0; //Simulation counter
    vector<vector<double> > simulations; //Matrix to hold results from simulations

    while(SIM_COUNT < MAX_SIM_COUNT){ //MAX_SIM_COUNT == 999
        vector<vector<double> > sim; // Matrix to hold simulation

        for(int i = 0; i < Y; i++){
            vector<double> simrow; //Row of sim

            for(int j = 0; j < order; j++){
                double r = rand(); //srand(time(null)) in Monaco Function
                simrow.push_back(r); //Aggregate random values.
            }
            sim.push_back(simrow); //Aggregate simulation rows.
        }
        jacobi_transform jt = eigenvalue_calc(sim ,order*Y); //Eigenvalues for simulation
        sort(jt.eigenvalues.begin(), jt.eigenvalues.end()); //Sort simulated eigenvalues.
        simulations.push_back(jt.eigenvalues); //Accumulate data.
        SIM_COUNT++; //Iterate SIM_COUNT
    }
    simulations = TRANSPOSE(simulations); //Transpose simulations.

    for(int i = 0; i < order; i++){
        double av = MEAN(simulations[i]); //Mean of simulations.
        double sd = STANDARD_DEVIATION(simulations[i]); //Standard deviations of simulations

        if(flag == 1 && cd.comp[i].eigenvalue > (av+1.96*sd/sqrt(Y))){ //Confidence intervals.
            c[i] = 1; //Reflect in referenced vector.
        }
        else if(flag == 0 && cd.comp[i].eigenvalue > av){ //For Average Monte-Carlo
            c[i] = 1;  //Reflect in referenced vector.
        }
    }
    return;
}
```

C. Buja-Eyuboglu Modification

A number of modifications and extensions of parallel analysis were formalized by Buja & Eyuboglu in 2000 which were reviewed in the comparative study done by Dr. Pedro R. Peres-Neto. The study found that replacing the mean with the confidence interval of the simulated variants worked better in some instances. The simulations run the same as Horn's Parallel Analysis, so to save code, the kernel can be reused using a flag argument (as shown above flag = 1 for the modification). The procedure goes as follows:

(1) 999 Monte-Carlo Simulations are executed and stored in the same way as in Horn's Parallel Analysis (VII.B) steps 1-3.
(2) Calculate for each axis the percentile intervals (e.g., 95% for an alpha = 0.05).
(3) Use intervals as standard critical values for assessing eigenvalues.

D. Anderson-Braak Permutation Test

A variation on the Monte-Carlo Simulations, the Anderson-Braak Permutation test use simulations generated randomizing the original data set with respect to its original axis. The procedure goes as follows:

(1) Transpose the original data so that the data is in columns with respect to the dependent variable.
(2) Shuffle the data within the columns, generate the eigenvalues for the shuffled matrix using eigenvalue_calc.
(3) Keep track of the number of times the generated eigenvalue is greater than the observed, use the final count to estimate the p-value for each axis in determining the number of non-trival components.

```
void Anderson_Braak(component_data cd, vector<int> &c){
    int order = cd.comp.size(); //Order of component matrix.
    int SIM_COUNT = 0; //Simulation counter
    vector<int> randoms(order, 1);//Initial vector to hold randoms
    //Initialize with 1's so to account for observed Eigenvalue
    int Y = cd.original_data.size(); //Original data size.
    cd.original_data = TRANSPOSE(cd.original_data); //Transpose original data.

    while(SIM_COUNT < MAX_SIM_COUNT){ //MAX_SIM_COUNT == 999

        for(int i = 0; i < order; i++){
            //srand(time(0)); in Monaco Function.
            random_shuffle(cd.original_data[i].begin(), cd.original_data[i].end());
        }
        vector<vector<double> > sim = TRANSPOSE(cd.original_data);
        //Transpose the original data back.
        jacobi_transform jt = eigenvalue_calc(sim, Y*order);  //Calculate eigenvalues.

        for(int i = 0; i < order; i++){

            if(jt.eigenvalues[i] > cd.comp[i].eigenvalue){ //if calculated > observed
                randoms[i] = randoms[i]+1; //Iterate the count of randoms
            }
        }
        SIM_COUNT++; //Iterate the simulation count.
    }
    for(int i = 0; i < order; i++){
        if(randoms[i]<=50){ //Estimated P-Value limit.
            c[i] = 1; //Reflect in referenced vector.
        }
    }
    return;
}
```

```
void INVERSE(vector<vector<double> > &c){
    int order = c.size(); //Order of matrix to invert.
    if(order == 1){
        c[0][0] = 1.0/c[0][0]; //Return inversion of single-unit matrix.
        return;
    }
    vector<vector<double> > cid = identity_matrix(order); //Create an indentity matrix.

    for(int i = 0; i < order; i++){
        double one = c[i][i]; //Check diagonal for value.
        if(one != 1){
            //Swap vectors if one == 0 and there is an available vector to swap with.
            if(one == 0 && i < order-1){
                vector<double> swapee = c[i+1]; //set swap vector.
                c[i+1] = c[i]; //Set swapped vector.
                c[i] = swapee; //Set remaining vector with swap vector.
                i = i-1; //Reset iteration.
                continue;
            }
            //End if one == 0 and no remaining vectors.
            if(one == 0 && i == order-1){
                continue;
            }
            for(int k = 0; k < order; k++){
                cid[k][i] = cid[k][i]/one; //Accumulate corrections in identity matrix.
                c[k][i] = c[k][i]/one; //Force value to 1.
            }
        }
        for(int j = 0; j < order; j++){
            if(i == j){continue;} //Redundant fail-safe.
            double val = c[i][j]; //Off-diagonal elements
            if(val == 0){continue;} //If value is already 0, continue.
            for(int k = 0; k < order; k++){
                cid[k][j] = cid[k][j] - val*cid[k][i]; //Accumulate corrections in identity.
                c[k][j] = c[k][j] - val*c[k][i]; //Force value to 0.
            }
        }
    }
    c = cid; //Reflect inversion in referenced vector.
    return;
}
```

E. Velicer's Minimum Average Partial Test

The Minimum Average Partial Test, takes the columns of the original data matrix and sorts them by their corresponding eigenvalues. It then creates a covariance matrix from the sorted dataset. The inversion of the covariance matrix is used to produce the partial correlation matrix of the data after removing N out of M components. Velicer's MAP test bases its acceptance criterion on the minimum of the average of the squared off-diagonal partial correlations in the matrix as N proceeds to (M-1).

```
void Minimum_Average_Partial(component_data cd, vector<int> &c){
    double MAP = numeric_limits<double>::max(); //Maximum numerical limit of double
    int MAP_index = 0; //MAP index tracker.
    cd.original_data = TRANSPOSE(cd.original_data); //Transpose original data.
    int order = cd.original_data.size(); //Order of component matrix.

    for(int j = 0; j < order; j++){
            cd.comp[j].data = cd.original_data[j]; //Set data properties.
    }
    sort(cd.comp.begin(), cd.comp.end(), eigen_unit::by_eigenvalue()); //Sort by eigenvalue
    vector<vector<double> > od; //Original "sorted" data vector.
    for(int j = 0; j < order; j++){
        od.push_back(cd.comp[j].data); //Fill in with original data in sorted form.
    }
    od = TRANSPOSE(od);  //Transpose sorted original data.
    vector<vector<double> > cm = covariance_matrix_calc(od); //Calculate covariance matrix.
    INVERSE(pcm); //Invert covariance matrix.

    for(int i = 0; i < order-1; i++){
        vector<vector<double> > crmn; //Partial covariance matrix.
        int diff = order - (i+1); //Number of partialed out components.
        for(int j = 0; j < diff; j++){
            vector<double> crmnr; //Partial Correlation Row.
            for(int k = 0; k < diff; k++){
                double rjk = pcm[j][k]; //Inverted position j,k
                double rjj =  pcm[j][j]; //Inverted position j, j
                double rkk = pcm[k][k]; //Inverted position k, ,
                double r = -1*rjk/sqrt(rjj*rkk); //Partial Correlation Coefficient
                crmnr.push_back(r); //Aggregate Partial Correlations
            }
            crmn.push_back(crmnr); //Aggregate Partial Correlation Rows.
        }
        double porder = crmn.size(); //Order of Partial Correlation Matrix
        vector<double> ap; //Average partial
        double counter = 0.0; //Counter.
        for(int j = 0; j < porder; j++){
            for(int k = 0; k < porder; k++){
                if(j == k){continue;} //Skip diagonal values.
                ap.push_back(pow(crmn[j][k], 2)); //Aggregate off-diagonal values.
                counter++; //Iterate Counter.
            }
        }
        double dap = accumulate(ap.begin(), ap.end(), 0.0)/counter; //Calculate Average.
        if(dap < MAP){ //Check if minimum.
            MAP = dap; //Set minimum value.
            MAP_index = i; //Set minimum index.
        }
    }
    for(int i = 0; i < MAP_index; i++){
        c[i] = 1; //Reflect in reference vector.
    }
    return;
}
```

References

[1] Peres-Neto, Pedro & Jackson, Donald & Somers, Keith. (2005). How Many Principal Components? Stopping Rules for Determining the Number of Non-Trivial Axes Revisited. Computational Statistics & Data Analysis. 49. 974-997. 10.1016/j.csda.2004.06.015.

[2] Rutishauser, "The Jacobi Method for Real Symmetric Matricies", in [30] 202-211

[3] Velicer, Wayne. (1976). Determining the Number of Components from the Matrix of Partial Correlations. Psychometrika. 41. 321-327. 10.1007/BF02293557.

[4] Horn, J.L. A rationale and test for the number of factors in factor analysis. Psychometrika 30, 179–185 (1965). https://doi.org/10.1007/BF02289447

[5] Anderson, M. & ter Braak, Cajo. (2010). Permutation tests for multi-factorial analysis of variance. Journal of Statistical Computation and Simulation. 73. 85-113. 10.1080/00949650215733.

[6] Buja, Andreas & Eyuboglu, Nermin. (2000). Remarks on Parallel Analysis. Multivariate Behavioral Research. 27. 10.1207/s15327906mbr2704_2.