BRYN MAWR COLLEGE

Department of Physics

Computational Module 8 – Analyzing Data

Prerequisite modules: Module 6 – Solution of Linear Equations; Module 7 – Eigenequations
Estimated completion time: 3 hours
Learning objectives: Become familiar with the basic approach to numerical least-squares fitting of data, and the technique of principal components analysis

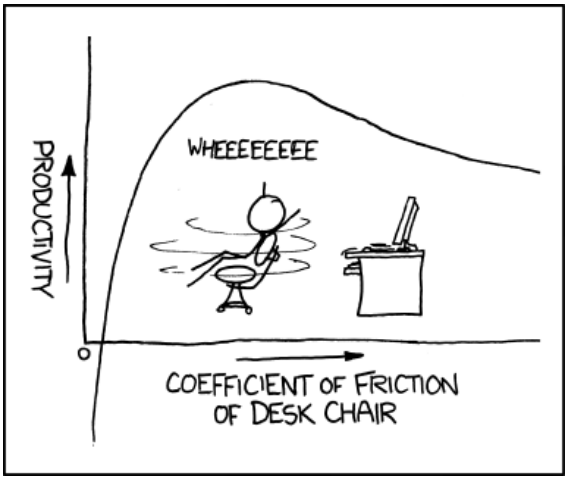


Image credit: xkcd.com

Extracting information from data is a central activity in all of the sciences and engineering, as well as other quantitative fields such as economics. One common need is to fit a set of data using a linear or nonlinear curve. Most numerical software — including spreadsheets, Mathematica, MATLAB, etc. — will offer a range of options for performing such fits. This module will explore some of the basic ideas underlying those fitting methods, familiarity with which might be helpful in fitting data that's poorly modeled by standard functions. As another example of a data analysis technique, this module also introduces principal components analysis, used for finding relationships between correlated variables. This module barely scratches the surface of the body of established data analysis techniques. For more information, see the Wikipedia page on data analytics.

8.1 Linear Least-Squares Data Fitting

The general problem we are considering here[1] is one in which we have N data points of the form (x_i, y_i) that obey some assumed relationship $y = y(x)$, so that for each point

$$y_i = y(x_i). \tag{1}$$

Here, we're thinking of x as the independent variable and y as the dependent variable. Our goal is to represent the relationship $y(x)$ in terms of some parameters a_j which will correspond to coefficients of ***basis*** functions $F_j(x)$ in a sum of M terms, as in

$$y(x) = \sum_{j=1}^{M} a_j F_j(x). \tag{2}$$

The expression on the right is called a ***linear model*** because the *coefficients* enter the expression linearly; the $F_j(x)$ themselves generally are *non*-linear functions of x, however. A common choice for the set of functions F_j is the polynomials — $F_1(x) = 1, F_2(x) = x, F_3(x) = x^2$, etc. — but they could also be $\sin(x)$ or $\cos(x)$ (in a Fourier series), exponential functions, or whatever else might be appropriate for the problem at hand. Once the set of functions is chosen, the most common criterion to use in determining the coefficients is the ***least squares*** condition, in which we require that the following sum-of-squares expression be a minimum:

$$S \equiv \sum_{i=1}^{N} r_i^2 = \sum_{i=1}^{N} \left[y_i - \sum_{j=1}^{M} a_j F_j(x_i) \right]^2. \tag{3}$$

For a given i, the term in brackets represents the difference between the i^{th} data value, y_i, and the approximation to it resulting from the model (i.e., the best-fit curve). This difference, represented by r_i, is called a ***residual***, shown in Figure 1 as the vertical orange line segments.[2] The goal of least-squares fitting is to find the set of coefficients a_i that minimizes the sum of squared residuals. In the best-case scenario, the model exactly fits all of the data, so all of the residuals vanish and the least-squares sum is zero. [This will happen automatically if the basis functions F_j are independent, and if $M \geq N$; i.e., the number of basis functions is greater than or equal to the number of data points. This would occur, for instance, if we fit two points with a straight line (of the form $a_0 + a_1 x$, using two basis functions) or a quadratic polynomial ($a_0 + a_1 x + a_2 x^2$, using three basis functions).] It should be noted that if the basis functions are not independent — i.e., if $\sum_{j=1}^{M} b_j F_j(x_i) = 0$ $(i = 1, 2, \ldots, N)$ for some set of b_j — then the requirement that the sum above be minimized will not uniquely determine the a_i, thus more than one set of a_i could minimize S.

[1] The material of this section is adapted mainly from *Computer Methods for Mathematical Computations*, by Forsythe, Malcolm and Moler, and partly from *Scientific Computing: An Introductory Survey, 2e* by Heath.

[2] Image credit: http://www.zweigmedia.com/RealWorld/calctopic1/regression.html

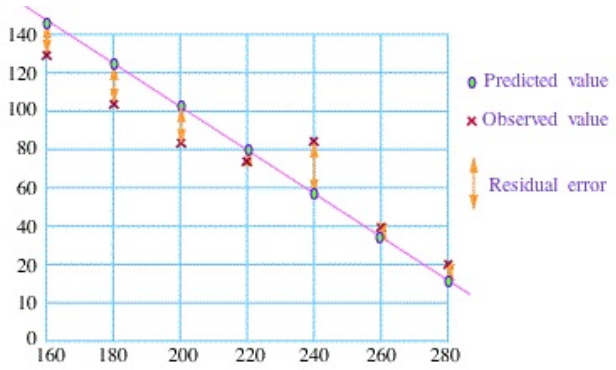


Figure 1: Residuals in a linear fit

Multiple approaches exist for finding the a_i that minimize Eq. (3). The obvious one is to use calculus: just as we minimize a function $f(x)$ by requiring $df/dx = 0$, in this case involving multiple a_i variables we should differentiate S with respect to *each* of the a_i, and require the results to be zero; that is, require $\partial S/\partial a_k = 0$ for all k ($k = 1, 2, \ldots, N$). Doing this, and reversing the order of the sums, the result is

$$\sum_{j=1}^{M} \left[\sum_{i=1}^{N} F_j(x_i) F_k(x_i) \right] a_j = \sum_{i=1}^{N} y_i F_k(x_i). \tag{4}$$

Breakpoint 1: Derive this result by differentiating the previous equation and setting the result to 0. (Hints: Think of each of the other a's as constants when differentiating with respect to a particular a_k. It may help to expand a few terms in the sums, take their derivatives, and then rewrite the sums.)

This expression represents a set of M simultaneous equations in the M unknown a_j's, called the ***normal equations***, which can be written in the matrix form $\mathbf{Ma} = \mathbf{q}$, where the matrix $\mathbf{M}$ and vector $\mathbf{q}$ have components

$$M_{kj} = \sum_{i=1}^{N} F_k(x_i) F_j(x_i), \tag{5}$$

$$q_k = \sum_{i=1}^{N} y_i F_k(x_i), \tag{6}$$

and the components of vector $\mathbf{a}$ are just the a_j. Note that the elements of $\mathbf{M}$ depend only on the basis functions, not on the data values (the y_i), while the elements of $\mathbf{q}$ depend on both.

As an example, suppose we wish to fit N data points (x_i, y_i) with a straight line, $y(x) = a_1 + a_2 x$ (so $F_1(x_i) = 1$ and $F_2(x_i) = x_i$). Then $M_{11} = \sum 1 \cdot 1$, $M_{12} = \sum 1 \cdot x_i = M_{21}$, $M_{22} = \sum x_i \cdot x_i$, and

$q_1 = \sum y_i \cdot 1$, $q_2 = \sum y_i \cdot x_i$. Hence, $\mathbf{M}$ and $\mathbf{q}$ take the forms

$$\mathbf{M} = \begin{pmatrix} N & \sum x_i \\ \sum x_i & \sum x_i^2 \end{pmatrix} \quad \text{and} \quad \mathbf{q} = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \end{pmatrix}, \tag{7}$$

where the sums run from 1 to N. The normal equations can be solved by one of the methods introduced in Module 6, except in the not uncommon situation that $\mathbf{M}$ has a form that is singular or nearly so, which can lead to large errors in the computed a_j's. As noted in that module, a new method known as ***singular value decomposition*** (SVD) can be used to treat such situations. Since that method works perfectly well in the case of non-singular matrices too, it is the technique of choice for linear least-squares fitting. The SVD approach is fairly complicated so it will not be discussed here, but an explanation is provided in the Appendix for the interested reader. Fortunately, a SVD function is provided in the `numpy.linalg` package.

Breakpoint 2: Write out the forms of $\mathbf{M}$ and $\mathbf{q}$ for the case that we wish to fit N data points with a quadratic polynonial, $a_0 + a_1 x + a_2 x^2$.

It's worth mentioning that often one wants to fit a set of data with known measurement uncertainties. If we denote by δ_i the uncertainty in the data value y_i, then in the presence of measurement uncertainties the definitions of the $\mathbf{M}$ matrix and $\mathbf{q}$ vector in the normal equations are modified to

$$M_{kj} = \sum_{i=1}^{N} \frac{1}{\delta_i^2} F_k(x_i) F_j(x_i), \tag{8}$$

$$q_k = \sum_{i=1}^{N} \frac{1}{\delta_i^2} y_i F_k(x_i). \tag{9}$$

The rest of the analysis in this situation proceeds just as in the case of no specified uncertainties (which corresponds to $\delta_i = 1$, for all i).

8.2 Principal Components Analysis

A powerful technique for finding connections among data involving interrelated variables – a technique that relies on finding eigenvectors – is known as ***principal components analysis*** (PCA), which originally derived from the principal axis theorem in mechanics. (This technique has many variations, and goes by many names, depending on the field of application.) It is used in image and signal analysis to identify features of interest, in genomics to identify subpopulations with related characteristics, and in many other application areas. (See the page for PCA on Wikipedia for more information.)

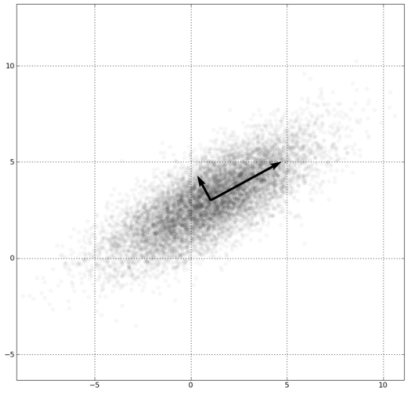


Figure 2: Example PCA data for two variables

Imagine that you have a set of data representing two related variables; for instance, the heights and weights of the students in a college. If you were to plot the data in a graph, it might look something like Figure 2 (ignore the scales on the axes).[3] If we suppose that the horizontal axis represents weight and the vertical axis represents height, then we can see that there's a relationship between them. Generally speaking, greater height correlates with greater weight, but the relationship is not exact — two people can have the same weight but significantly different heights. Note the long vector pointing up and to the right — it identifies the "direction" within the data along which it varies the most, and this direction reveals the strongest relationship between the heights and weights. If there were no variation in the orthogonal direction (indicated by the short vector), then the relationship would be a perfectly linear one. On the other hand, if both vectors were of the same length, then there'd be no correlation between the variables (the "blob" of data would fill a circular region in the plot, with no preferred axis).

The aim of PCA is to identify the "long vector," and the orthogonal ones too, in a set of data representing correlated variables. (This is analogous to finding the principal axes of rotation of an extended body.) While it might be possible to do this by eye in the case of just two variables by making a plot like Figure 2, it becomes quite difficult when more than two variables are involved. (One would have to plot the data in three or more dimensions to see the relationships among the variables.)

PCA involves finding the eigenvectors of a certain matrix constructed from the data for the variables of interest. For example, let x_i and y_i denote the values taken by two variables; e.g., in our previous example the x_i would represent the weights of the students and the y_i would represent the heights. Further, let $\bar{x}$ and $\bar{y}$ represent the averages of the respective variables. You may know from a previous course that in the case of n measurements of one variable we define the ***variance*** (the

[3] From Wikipedia page on PCA.

square of the standard deviation) as

$$var(x) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \,. \tag{10}$$

The ***covariance*** of two variables is a generalization of the variance, and is defined as

$$cov(x, y) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x}) (y_i - \bar{y}) \,. \tag{11}$$

Notice from these two expressions that we can consider the variance as just a special case of the covariance, in which the two variables are one and the same; that is, $var(x) = cov(x, x)$. Further note that we can think of $cov(x, y)$ essentially as an inner product of two vectors representing the deviations of the two variables from their means. If the two variables fairly consistently vary in the same way, so that the signs of the deviations $(x_i - \bar{x})$ and $(y_i - \bar{y})$ generally are the same for the various i values, then their covariance will be large; if the signs of the two expressions vary randomly, so that they're the same as often as they're opposite, then there's little correlation between them and their covariance will be small. This is very much like the way that the inner product of two vectors is substantial when their components are similar (so they're nearly parallel) but small when their components are very different (much less parallel).

It is useful to combine the various possible pairings of variable covariances into a single matrix, the ***covariance matrix***, which for two variables has the form

$$\begin{pmatrix} cov(x, x) & cov(x, y) \\ cov(y, x) & cov(y, y) \end{pmatrix} .$$

The generalization to more than two variables is straightforward (e.g., for three variables the covariance matrix would be 3×3, with the third row and column involving the third variable paired with itself and the other two variables.)

It is the eigenvectors of the covariance matrix that correspond to the vectors in Figure 2. Specifically, the eigenvector corresponding to the largest eigenvalue is the one that reveals the greatest variance in the data; i.e., the direction of the long vector in Figure 2. The eigenvector corresponding to the next-largest eigenvalue is the one that shows the next-greatest variance in the data, and so on.

Recap

- Least-squares fitting of a set of data using a series expansion in user-selected basis functions amounts to solving a set of simultaneous equations — the normal equations $\mathbf{Ma} = \mathbf{q}$ — for the vector $\mathbf{a}$ representing the coefficients of the basis functions. This can be done using the methods of Module 6.
- Principal components analysis finds the strongest relationships between correlated variables as the eigenvectors of the covariance matrix corresponding to the largest eigenvalues. The methods of Module 7 apply here.
- From the Appendix: singular value decomposition is a method of factorizing matrices that permits the solution of equations like the normal equations even in the case of singular $\mathbf{M}$. The Python function `svd` in the `numpy.linalg` package implements this method. [The reader who wishes to use this method without reading the entire Appendix can start at the paragraph just before Eq. (21).]

Appendix: Singular Value Decomposition

The method we need for solving problems involving an $\mathbf{M}$ that may be singular, or close to it, is known as ***singular value decomposition***, or SVD — another way of factoring a matrix.[4] It turns out that SVD has many useful applications; for example, it can be used for data compression of images, and Netflix even uses it as a key element of its movie recommendation system!

You may have learned in linear algebra that every symmetric matrix $\mathbf{A}$ can be factored in the form $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$, where $\mathbf{Q}$ is an orthogonal matrix (so $\mathbf{Q}^{-1} = \mathbf{Q}^T$) and $\mathbf{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\mathbf{A}$. The natural question that arises is whether a similar factorization can be performed for a non-symmetric matrix. The answer is yes — sort of — and SVD provides such a factorization.

To understand SVD, it will be helpful to have a geometrical notion in mind; namely, that multiplication by a matrix changes one vector into another one. Consider the x-y plane, and the effect of multiplying the vector representing some point (x, y) in the plane by a matrix; e.g.,

$$\begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3x \\ y \end{pmatrix} \tag{12}$$

The effect of multiplication by this matrix has been to stretch the x axis by a factor of 3; for instance, the point (1,1) would have been taken to (3,1), the point (0,1) would not be changed, and the point (1,0) would have been moved to (3,0).[5] Thus, this matrix would stretch the unit square defined by the points (0,0), (0,1), (1,0), (1,1) into a rectangle of width 3 and height 1. Likewise, switching the 3 and the 1 in the matrix would result in a new matrix that stretches the plane *vertically*, turning the unit square into a rectangle with width 1 and height 3.

Going a step further, if we replace the 1 in the original matrix with a 2, the resulting matrix would stretch the plane both along the x axis (by a factor of 3) and along the y axis (by a factor of 2).

Now consider the effect of a more general *symmetric* matrix $\mathbf{A}_1 = \left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$:

$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2x + y \\ x + 2y \end{pmatrix}. \tag{13}$$

This matrix would produce the following transformations of the corners of the unit square: (0,0) $\rightarrow$ (0,0); (0,1) $\rightarrow$ (1,2); (1,0) $\rightarrow$ (2,0); and (1,1) $\rightarrow$ (3,3). This transformation changes the unit square into the diamond shape shown on the right in Figure 3. (In the figure, the matrix action is labeled by M). The square has been stretched along a 45° line.

[4] The material of this section, including the figures, is taken mainly from http://www.ams.org/samplings/feature-column/fcarc-svd . Some material also comes from *Scientific Computing* by Heath.

[5] This and the following transformations can be confirmed, and new ones explored, using the notebook `Matrix_Transformations.ipynb`, which can be found where these Modules are posted.

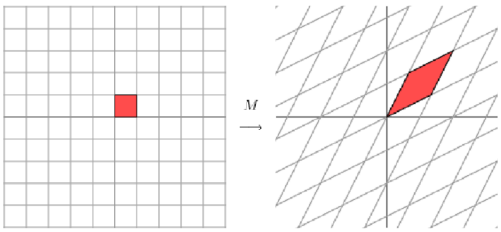


Figure 3: Transformation of the unit square by matrix $\mathbf{A}_1$

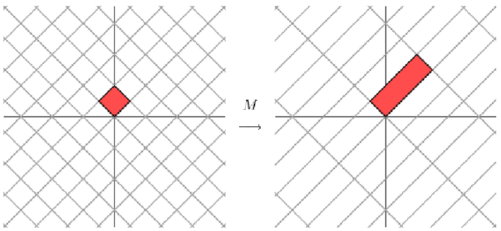


Figure 4: Transformation of a rotated unit square by matrix $\mathbf{A}_1$

We can get a different perspective on what's going on here if we imagine a coordinate system rotated 45° from the original one, such that the unit square is rotated into a "unit diamond." As Figure 4 shows, in this rotated coordinate system matrix $\mathbf{A}_1$ has the effect of stretching the unit diamond by a factor of 3 along the rotated x axis. [You can convince yourself of this by operating with $\mathbf{A}_1$ on the corners of the unit diamond, at (0,0), $\frac{1}{\sqrt{2}}(-1,1)$, $(0,\sqrt{2})$, and $\frac{1}{\sqrt{2}}(1,1)$ in the original coordinate system.] This simple stretching (along a new axis) is a consequence of the fact that $\mathbf{A}_1$ is a symmetric matrix. (Symmetric matrices also can produce reflection about a line.) Note that in this case, the rotated x and y axes remain perpendicular to each other after the transformation produced by $\mathbf{A}_1$. That implies that the eigenvectors of $\mathbf{A}_1$ lie along those rotated axes.

Now let's explore what happens if we use a non-symmetric matrix, e.g. $\mathbf{A}_2 = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$. As shown in Figure 5, this has the effect of "sliding" the top of the unit square sideways, as would happen to a loosely bound book (viewed along its spine) if you pushed the top cover sideways while holding the bottom cover in place. This kind of transformation is called ***shear*** (it happens, for example, in moving fluids). With the previous discussion in mind, the obvious question to ask is whether some rotation of the original coordinate system would result in this shear being transformed into a simpler transformation. Since we seem to have a theme going here, you probably won't be surprised to learn that the answer is "yes." In fact, the necessary rotation angle, as illustrated in the left-hand part of Figure 6, is about 58.28°. Note that in this case the matrix seems to both stretch and rotate the square. That is, a shear corresponds to both a rotation and a stretching. As a result of this combined action, while the final coordinate axes resulting from the original rotation and the action of $\mathbf{A}_2$ are perpendicular to each other (see the right-hand part of the figure), they are not aligned with the (rotated) axes in the left-hand part of the figure.

Computing the SVD corresponds to finding an orthogonal grid (the left-hand images in Figures 4

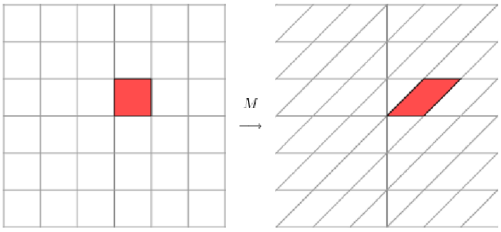


Figure 5: Transformation of the unit square by matrix $\mathbf{A}_2$

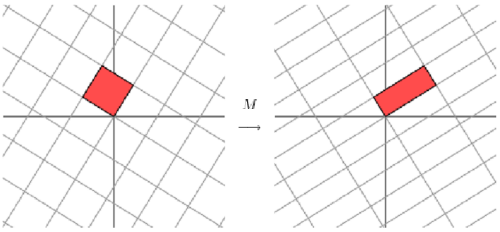


Figure 6: Transformation of a rotated unit square by matrix $\mathbf{A}_2$

and 6) that is transformed into another orthogonal grid (the right-hand images in those figures) by the matrix of interest ($\mathbf{A}_1$ and $\mathbf{A}_2$ in the examples above). Put another way, in this 2-D example the SVD of a matrix $\mathbf{A}$ finds orthogonal unit vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ such that $\mathbf{A}\mathbf{v}_1$ and $\mathbf{A}\mathbf{v}_2$ also are orthogonal.

How is this done? The method is suggested by Figure 7, which shows a unit circle superimposed on the grid of Figure 6. Under the action of $\mathbf{A}$, the unit circle is transformed into an ellipse whose major and minor axes are the transformed vectors of interest, $\mathbf{A}\mathbf{v}_1$ and $\mathbf{A}\mathbf{v}_2$. That is, $\mathbf{A}\mathbf{v}_1$ and $\mathbf{A}\mathbf{v}_2$ are the extrema (the maximum and minimum) of $|\mathbf{A}\mathbf{v}|$ for $\mathbf{v}$ on the unit circle. As it turns out, these extrema are the eigenvectors of the matrix $\mathbf{A}^T\mathbf{A}$. Being a symmetric matrix, its eigenvectors belonging to different eigenvalues are orthogonal, as we need.

We note that if $\mathbf{u}_1$ and $\mathbf{u}_2$ are unit vectors in the directions of $\mathbf{A}\mathbf{v}_1$ and $\mathbf{A}\mathbf{v}_2$, then we can write $\mathbf{A}\mathbf{v}_1 = \sigma_1\mathbf{u}_1$ and $\mathbf{A}\mathbf{v}_2 = \sigma_2\mathbf{u}_2$. We then may express a general vector $\mathbf{w}$ in terms of the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ as

$$\mathbf{w} = (\mathbf{w}\cdot\mathbf{v}_1)\,\mathbf{v}_1 + (\mathbf{w}\cdot\mathbf{v}_2)\,\mathbf{v}_2, \tag{14}$$

and then multiplying both sides by $\mathbf{A}$ we get

$$\mathbf{A}\mathbf{w} = (\mathbf{w}\cdot\mathbf{v}_1)\,\mathbf{A}\mathbf{v}_1 + (\mathbf{w}\cdot\mathbf{v}_2)\,\mathbf{A}\mathbf{v}_2, \tag{15}$$

$$= (\mathbf{w}\cdot\mathbf{v}_1)\,\sigma_1\mathbf{u}_1 + (\mathbf{w}\cdot\mathbf{v}_2)\,\sigma_2\mathbf{u}_2. \tag{16}$$

We can replace the dot products using the transposes of the $\mathbf{v}$ vectors (we can think of the transposes as row vectors), getting

$$\mathbf{A}\mathbf{w} = (\mathbf{v}_1^T\mathbf{w})\,\sigma_1\mathbf{u}_1 + (\mathbf{v}_2^T\mathbf{w})\,\sigma_2\mathbf{u}_2, \tag{17}$$

$$= \mathbf{u}_1\,\sigma_1\,\mathbf{v}_1^T\,\mathbf{w} + \mathbf{u}_2\,\sigma_2\,\mathbf{v}_2^T\,\mathbf{w}, \tag{18}$$

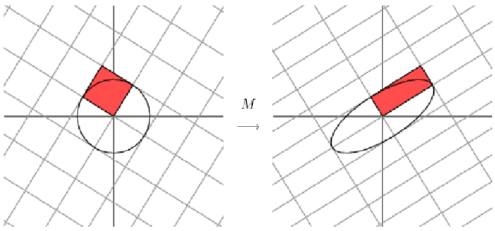


Figure 7: Transformation of a unit circle by matrix $\mathbf{A}_2$

where the rearrangement of the second equation is possible since the products $\mathbf{v}_i^T\mathbf{w}$ are scalars. Since this equation is true for *any* $\mathbf{w}$, we can extract it from the equation to get

$$\mathbf{A} = \mathbf{u}_1\,\sigma_1\,\mathbf{v}_1^T + \mathbf{u}_2\,\sigma_2\,\mathbf{v}_2^T. \tag{19}$$

If we imagine assembling the $\mathbf{u}_i$'s and $\mathbf{v}_j$'s into the columns of matrices $\mathbf{U}$ and $\mathbf{V}$ respectively, and making the σ_i's into a diagonal matrix $\mathbf{\Sigma}$, then we can rewrite the above equation as

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T. \tag{20}$$

This is the singular value decomposition of $\mathbf{A}$. The three matrices can be interpreted geometrically: $\mathbf{U}$ and $\mathbf{V}^T$ perform rotations and $\mathbf{\Sigma}$ causes stretching. (Note that the bold $\mathbf{\Sigma}$ matrix symbol is different from the un-bold summation symbol, Σ.) The SVD therefore reveals that the action of any 2-D matrix can be reduced to a rotation, followed by a stretch, followed by another rotation.

Breakpoint 3: Write out the components of $\mathbf{A}$ determined from these matrices:

$$\mathbf{U} = \begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix}, \qquad \mathbf{\Sigma} = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}, \qquad \mathbf{V} = \begin{pmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{pmatrix}.$$

The same procedure presented above is possible even if $\mathbf{A}$ is a non-square, $M \times N$ matrix (which can be thought of as transforming an M-dimensional "cube" to an N-dimensional "rectangle" or, more commonly, an M-dimensional "sphere" to an N-dimensional "ellipsoid"). In that general situation, $\mathbf{U}$ is an $M \times M$ orthogonal matrix, $\mathbf{V}$ is an $N \times N$ orthogonal matrix, and $\mathbf{\Sigma}$ is an $M \times N$ "diagonal" matrix with elements

$$\sigma_{ij} = \begin{cases} 0, & i \neq j; \\ \sigma_i \geq 0, & i = j. \end{cases}$$

Often these are ordered largest to smallest, so that $\sigma_i \geq \sigma_{i+1}$. (By "orthogonal" in the description of $\mathbf{U}$ and $\mathbf{V}$ above, we mean that each matrix's columns are normalized and mutually orthogonal.)

The values σ_i are termed the ***singular values*** of $\mathbf{A}$, and the columns of $\mathbf{U}$ and $\mathbf{V}$ are known as the left and right ***singular vectors***, respectively. The singular values turn out to be the square roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$, and the columns of $\mathbf{U}$ and $\mathbf{V}$ are orthonormal eigenvectors of $\mathbf{A}\mathbf{A}^T$ and $\mathbf{A}^T\mathbf{A}$, respectively. (A later Module will discuss how to find eigenvalues and eigenvectors numerically.) Note the difference between the SVD factorization and the one for symmetric matrices noted earlier, $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$: in SVD, the two orthogonal matrices sandwiching $\mathbf{A}$ are, in general, very different from each other, not simply transposes of one another.

Now let's come back to our original goal of solving for $\mathbf{a}$ in the normal equations $\mathbf{M}\mathbf{a} = \mathbf{q}$. We start by finding the SVD of matrix $\mathbf{M}$: $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. Then using that in the normal equations, we have

$$\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{a} = \mathbf{q}. \tag{21}$$

Left-multiplying both sides first by $\mathbf{U}^{-1} = \mathbf{U}^T$, then by $\mathbf{\Sigma}^{-1}$, and finally by $(\mathbf{V}^T)^{-1} = \mathbf{V}$, we obtain

$$\mathbf{a} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^T\mathbf{q}. \tag{22}$$

Because $\mathbf{\Sigma}$ is diagonal, so is $\mathbf{\Sigma}^{-1}$ (the non-zero diagonal elements of $\mathbf{\Sigma}^{-1}$ are simply the reciprocals of the corresponding diagonal elements of $\mathbf{\Sigma}$) and, as a result, the expression above reduces to

$$\mathbf{a} = \sum_{\sigma_i \neq 0} \frac{1}{\sigma_i} \left(\mathbf{u}_i^T\mathbf{q}\right) \mathbf{v}_i, \tag{23}$$

In words, we compute $\mathbf{a}$ from a sum of terms associated with the nonzero singular values: if $\sigma_i \neq 0$, then the i^{th} term involves taking the inner product of the i^{th} column of $\mathbf{U}$ (as a row vector) with the (column) vector $\mathbf{q}$, dividing by the i^{th} singular value, and using the resulting scalar to multiply the i^{th} column of $\mathbf{V}$. This procedure results in a series of column vectors that add up to give $\mathbf{a}$.

While the singular values and vectors are related to $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$, algorithms for computing the SVD work directly from $\mathbf{A}$ rather than from the matrix products, because any errors in $\mathbf{A}$ would be magnified in those products. SVD programs, which may make use of the QR algorithm mentioned in Module 6, are quite complicated, so they will not be discussed here. Fortunately, the `numpy.linalg` package includes an `svd` function which takes an $M \times N$ matrix $\mathbf{A}$ and returns the three matrix factors of Eq. (20).

It's worth noting that the inverse of a square, invertible matrix $\mathbf{A}$ is $\mathbf{A}^{-1} = \mathbf{V}\Sigma^{-1}\mathbf{U}^T$, and therefore the SVD can be used to numerically determine the inverse of such a matrix.

Finally, the reason that SVD is useful in data compression, e.g. of images or music, is that the part of the data associated with the largest singular values is the most important. That is, in the general form of Eq. (19), which may contain many terms of the form shown there, the dominant terms are

the ones with the largest σ values. Therefore, to reduce the size of a data set with minimal loss of information, the data associated with the *smallest* singular values may be removed.

Consider a greyscale image composed of pixels having values in a finite range, usually 0 to 255, representing 256 shades of grey from pure black to pure white. The values of those image pixels are represented by our matrix $\mathbf{A}$. We compute the SVD of this matrix, and then to compress it we keep only the r largest singular values in $\mathbf{\Sigma}$, setting the others to zero. This gives us a modified matrix $\mathbf{\Sigma}_r$. We then construct the compressed matrix $\mathbf{A}_{\text{comp}}$ as

$$\mathbf{A}_{\text{comp}} = \mathbf{U}\,\mathbf{\Sigma}_r\mathbf{V}^T. \tag{24}$$

Many of the values in $\mathbf{A}_{\text{comp}}$ would be the same, namely zero, and there are significantly more-efficient methods of storing such ***sparse matrices*** in memory, and of transmitting them, compared with non-sparse matrices.

Exercises

Exercise #1

Suppose you want to fit a straight line, $y(x) = a_1 + a_2 x$, to the points (0.0, 1.0), (1.0, 2.0), (3.0, 3.0). (a) Find $\mathbf{M}$ and $\mathbf{q}$ for this situation. (b) Using your code from the previous Module to implement Gaussian elimination with back-substitution, solve the associated normal equations for the two a_i coefficients. (Make sure that your inputs are floating point numbers.) (c) Plot the data points and your fit on the same graph. How is the fit?

Exercise #2

Fit the function $y(x) = a_1 x + a_2 e^x$, to the points (1.0, 1.5), (2.0, 3.0), (3.0, 7.0). (a) Find $\mathbf{M}$ and $\mathbf{q}$ for this situation. (b) Using your code from the previous Module to implement Gaussian elimination with back-substitution, solve the associated normal equations for the two a_i coefficients. (c) Plot the data points and your fit on the same graph. How does the fit look? (Note: you will want to use at least 20 points, and maybe more, to generate a smooth curve for your fit.)

Exercise #3

(a) Write a program to compute the covariance matrix, given 1-D arrays of data for two variables. (Your program should check that the arrays have the same length, and print an error message and exit if not.)

(b) Test your program by computing the covariance matrix for this data:[6]

$x = [2.5, 0.5, 2.2, 1.9, 3.1, 2.3, 2.0, 1.0, 1.5, 1.1]$,

$y = [2.4, 0.7, 2.9, 2.2, 3.0, 2.7, 1.6, 1.1, 1.6, 0.9]$.

(c) Compute the eigenvalues and eigenvectors for the covariance matrix and identify the directions of the eigenvectors. (d) Plot the raw data, and estimate whether the eigenvector corresponding to the largest eigenvalue aligns with the long axis of the data, and whether the other eigenvector is perpendicular to the first. (You can compute their inner product to confirm orthogonality.)

Exercise #4

One of the nice features of Python is the possibility of feeding a variable number of inputs to a function. That feature can be used to help generalize your covariance function to the case of more than two correlated variables. The syntax is simple: to feed in an arbitrary set of inputs, you can use the format `def func(*args)`, where `*args` stands for as many comma-separated inputs as you need; that is, you can call your covariance function as `covar(x,y,z,w)` if you define it with a single argument of `*args`. (Note: the particular name "`args`" is not required, but the "`*`" is.) If you know that you will provide a certain minimum number of inputs – two in the case of the covariance function – then you can specify those as explicit inputs and use `*args` to represent any additional inputs. In other words, you could define your covariance function as `def covar(x,y,*args)`, and still call it as `covar(x,y,z,w)`.

[6] Data from *A Tutorial on Principal Components Analysis*, by Lindsay Smith.

(The `*args` must come *after* the required inputs.) This alternative method might be preferable from a good-programming perspective, as it immediately shows anyone who reads your code that your function must be supplied at least two inputs. In any case, use this feature to generalize your covariance function to work with any number of input 1-D arrays, and test it on the data of Exercise #3. (Of course, you might have to alter other parts of your program, depending on how you wrote it!)

Exercise #5 (Requires the Appendix to be read)

Write a function to solve the normal equations ($\mathbf{Ma} = \mathbf{q}$) using `numpy`'s `svd` function and Eq. (23). The function should take $\mathbf{M}$ and $\mathbf{q}$ as inputs and return $\mathbf{a}$ as the output, constructing it by using the $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}$ matrices associated with $\mathbf{M}$ as obtained from the `svd` function. Test your function with the specific $\mathbf{M}$ and $\mathbf{q}$ that you found in Exercise #1, and compare its output with your result from that exercise. (After importing `svd` from `numpy.linalg`, call it like this: `u,s,v = svd(M)`.)

Breakpoint Answers

Breakpoint 1

$$S = \sum_{i=1}^{N} \left[y_i - \sum_{j=1}^{M} a_j F_j(x_i) \right]^2 .$$

Take the partial derivative with respect to a *particular* a (e.g., a_k) treating the other a_i's as constants, and set the result equal to 0 (the chain rule is used):

$$\frac{\partial S}{\partial a_k} = \sum_{i=1}^{N} 2\left[-F_k(x_i)\right] \left[y_i - \sum_{j=1}^{M} a_j F_j(x_i) \right] = 0.$$

Then do the algebra:

$$\sum_{i=1}^{N} -2F_k(x_i)y_i + 2\sum_{i=1}^{N} F_k(x_i) \sum_{j=1}^{M} a_j F_j(x_i) = 0$$

$$\sum_{i=1}^{N} F_k(x_i) \sum_{j=1}^{M} a_j F_j(x_i) = \sum_{i=1}^{M} y_i F_k(x_i)$$

$$\sum_{j=1}^{M} \left[\sum_{i=1}^{N} F_j(x_i) F_k(x_i) \right] a_j = \sum_{i=1}^{M} y_i F_k(x_i).$$

Breakpoint 2

Here, $M_{11} = \sum 1 \cdot 1$, $M_{12} = \sum 1 \cdot x_i = M_{21}$, $M_{22} = \sum x_i \cdot x_i$, $M_{13} = \sum 1 \cdot x_i^2 = M_{31}$, $M_{23} = \sum x_i \cdot x_i^2 = M_{32}$, $M_{33} = \sum x_i^2 \cdot x_i^2 = M_{33}$; also, $q_1 = \sum y_i \cdot 1$, $q_2 = \sum y_i \cdot x_i$, and $q_3 = \sum y_i \cdot x_i^2$. So,

$$\mathbf{M} = \begin{pmatrix} N & \sum x_i & \sum x^2 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \end{pmatrix} \quad \text{and} \quad \mathbf{q} = \begin{pmatrix} \sum y_i \\ \sum y_i x_i \\ \sum y_i x_i^2 \end{pmatrix}.$$

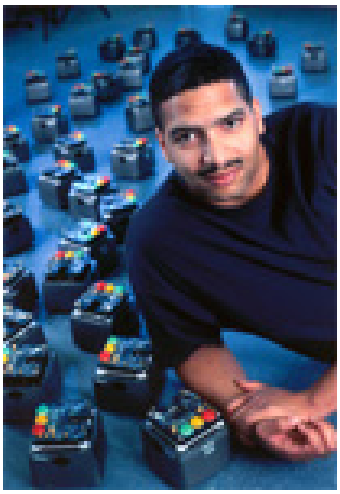
Breakpoint 3

First we compute $\mathbf{\Sigma V}^T$ with the given matrices, then bring in $\mathbf{U}$:

$$\mathbf{\Sigma V}^T = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} v_{11} & v_{21} \\ v_{12} & v_{22} \end{pmatrix} = \begin{pmatrix} \sigma_1 v_{11} & \sigma_1 v_{21} \\ \sigma_2 v_{12} & \sigma_2 v_{22} \end{pmatrix},$$

$$\mathbf{U\Sigma V}^T = \begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix} \begin{pmatrix} \sigma_1 v_{11} & \sigma_1 v_{21} \\ \sigma_2 v_{12} & \sigma_2 v_{22} \end{pmatrix} = \begin{pmatrix} u_{11}\sigma_1 v_{11} + u_{12}\sigma_2 v_{12} & u_{11}\sigma_1 v_{21} + u_{12}\sigma_2 v_{22} \\ u_{21}\sigma_1 v_{11} + u_{22}\sigma_2 v_{12} & u_{21}\sigma_1 v_{21} + u_{22}\sigma_2 v_{22} \end{pmatrix}.$$

Scientist Profile



James McLurkin is a roboticist and professor of computer of science at Rice University. He was born in 1972 and grew up in Baldwin, N.Y. Dr. McLurkin obtained a Bachelors degree in Electrical Engineering from MIT with a Mechanical Engineering minor, a Masters degree in Electrical Engineering from UC Berkeley, and a Ph.D in Computer Science from MIT. Dr. McLurkin developed his interest in engineering from Lego construction. He studies swarm intelligence of robotic systems, which involves coordinating large systems of small robots. Maximizing computation speed and managing algorithmic complexity are very important in McLurkin's work. As systems of robots increase in size and their tasks increase in sophistication, the algorithms grow increasingly complex. Because each robot must relay and respond to the varying communications of up to hundreds of others, efficiency is key. A benefit of using a swarm of simple robots over a single complex robot is that the ability of the swarm to complete tasks can grow in a way that mimics intelligence, not unlike the sophisticated emergent behavior of ant colonies. (In fact, Dr. McLurkin keeps ants as pets and has used them as inspiration for his groundbreaking work throughout his undergraduate, graduate, and industry careers.) Another benefit of a swarm approach is its built-in redundancy — even if a few robots fail, the overall task may still be completed by those remaining. Swarm intelligence algorithms have many other uses, ranging from understanding biological systems, to controlling nanorobots, to intelligence applications. When he is not advancing his research, Dr. McLurkin spends much of his time teaching summer programs to Boston high school students in science and technology, and engaging in speaking events.