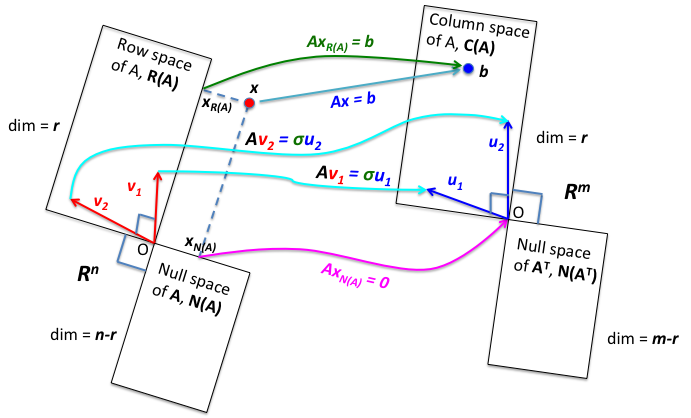
# **Singular Value Decomposition (SVD).**

We have seen that if we have a square matrix ***A*** we may be able to diagonalize it by means of an *eigen decomposition* **.** However, very often the *eigenvectors* in ***S*** have two big problems: they are NOT orthogonal unless ***A*** is symmetric (in which case ), and if they are not orthogonal, they may not be all linearly independent for ***S*** to be invertible. Finally, even when these conditions are met, the *eigen decomposition* is possible only for the very small class of matrices that are square. The ***Singular Value Decomposition*** of any (square or rectangular, symmetric or non-symmetric) matrix ***A*** solves these problems.

We recall here that when a matrix ***A*** acts as an *operator* on vector ***x***, the operation moves the projection of vector ***x*** in the row space to the column space.



Now we are looking for an orthogonal (really orthonormal, all unit vectors) basis ***V*** of dimensions ***r*** in the row space that gets moved to a scaled orthogonal (orthonormal) basis ***U*** of dimensions ***r*** in the column space, such that:

which is going to be for rectangular matrices the equivalent of:

for square matrices. Just as we expanded this into , expanding we write:

*m* x *r* *m* x *n* *n* x *r* *m* x *r* *r* x *r m* x *r*

This is the heart of the **SVD,** but we can add*n-r*more orthonormal ***v*** vectors taking them from the null space of ***A*,** and *m-r*more orthonormal ***u*** vectorstaking themfrom the left null space of ***A***. They will be orthonormal bases for the two null spaces and therefore they will be orthogonal to the first *r* ***v*** and ***u*** vectors.

*m* x *n* *m* x *n* *n* x *n* *m* x *m* *m* x *n m* x *n*

and bring ***V*** (orthogonal) on the other side as ***VT***:

*m* x *n* *m* x *m* *m* x *n* *n* x *n*

In conclusion, we need to find 2 orthogonal matrices ***V*** and ***U*** such that we can factorize ***A*** as

and obviously, we can't find both simultaneously. Therefore, we have to find a strategy that allows us to make either ***V*** or ***U*** disappear in the equation above. For example, we can multiply both sides of the equation by ***AT*** on the left.

Thus, the final result is:

We notice that (*n* x *n*) is symmetric (positive definite or semidefinite) and thus the equation is nothing but the *eigen decomposition* of , with its *eigenvectors* and its *eigenvalues.*

In a similar way, we can multiply both sides by on the right.

And the final result this time is:

Again, we notice that (*m* x *m*) is symmetric (positive definite or semidefinite) and thus the equation is nothing but the *eigen decomposition* of , with its *eigenvectors* and its *eigenvalues.* It follows that the *singular values* ***σ*** (the entries on the diagonal of )of ***A*** are ≥ 0.

A = rand(4,2)\*rand(2,3), r = rank(A)

AtA = A'\*A

[V,Svs] = eigs(AtA)

AAt = A\*A'

[U,Sus] = eigs(AAt)

S = sqrt(diag(Svs))

We have now completed the factorization of ***A*** as:

and found that:

Based on this result we can define the following bases:

[U\_svd,S\_svd,V\_svd] = svd(A)

V = orth(A')

Vn = null(A)

U = orth(A)

Uln = null(A')

The following relations hold:

and from :

the following:

Notice that if ***A*** is symmetric . Since the eigenvectors of are the same as the eigenvectors of and its eigenvalues are the square of the eigenvalues of , it follows that the singular value decomposition of a symmetric matrix is the same as its eigen decomposition (singular values of are the square root of the eigenvalues of , which are the eigenvalues of ) and in that case the ***U*** and ***V*** vectors are identical.

**Important Note:**  In fundamental applications throughout science the *singular values* of ***A*** have an extremely useful property: they essentially drop to 0 after the first ***r*** *singular vectors* (***σr+1*** is very small). Then, that *threshold value* of ***r*** is the *effective rank* of the matrix ***A***. The actual rank of ***A*** may be larger, and very noisy data can deceptively lead to full rank.

A = [2 7 6;4 14 12;1 3 3], rank(A)

noise = rand(3)/100000

A\_noisy = A + noise, rank(A\_noisy)

[U,S,V] = svd(A\_noisy)

Notice how the **SVD** can be considered as a summation of matrices of *rank 1* and dimensions *m* x *n*, each one derived from an outer product , *ordered* and *scaled* by the size of the *singular values*:

Since , the SVD decomposition tell us that any matrix can be reconstructed as the *summation* of *r* matrices of ***rank =* 1**, each added with a different weight :

We can add only the *rank 1* matrices with the *largest* singular values in the reconstruction, thus effectively producing the corresponding matrix ***Anew*** of rank ***rnew*** *<* ***r***. This is a very effective way of removing the 'noise' in ***A***.

A\_denoised = U(:,1:2)\*S(1:2,1:2)\*V(:,1:2)'

rank(A\_denoised)

At this point we notice something very important: by choosing the bases ***V*** and ***U***, ***A*** multiplies in the row space to obtain in the column space. Thus, if:

then (if it exists) should do the opposite!

Until this moment we said "if existed", but now we can see how a matrix that multiplies to obtain can be obtained: we call it the ***pseudoinverse*** of ***A*** and is represented with the symbol ***A+***. Thus, if ***A*** is *m* x *n*, we obtain ***A+*** of size *n* x *m*:

*n* x *m* *n* x *n* *n* x *m* *m* x *m*

Strictly speaking is not the inverse of , but since the rows and columns of beyond *r* are all 0, the product above is the same as:

*n* x *m* *n* x *r* *r* x *r* *r* x *m*

with as the true inverse of .

r = rank(A)

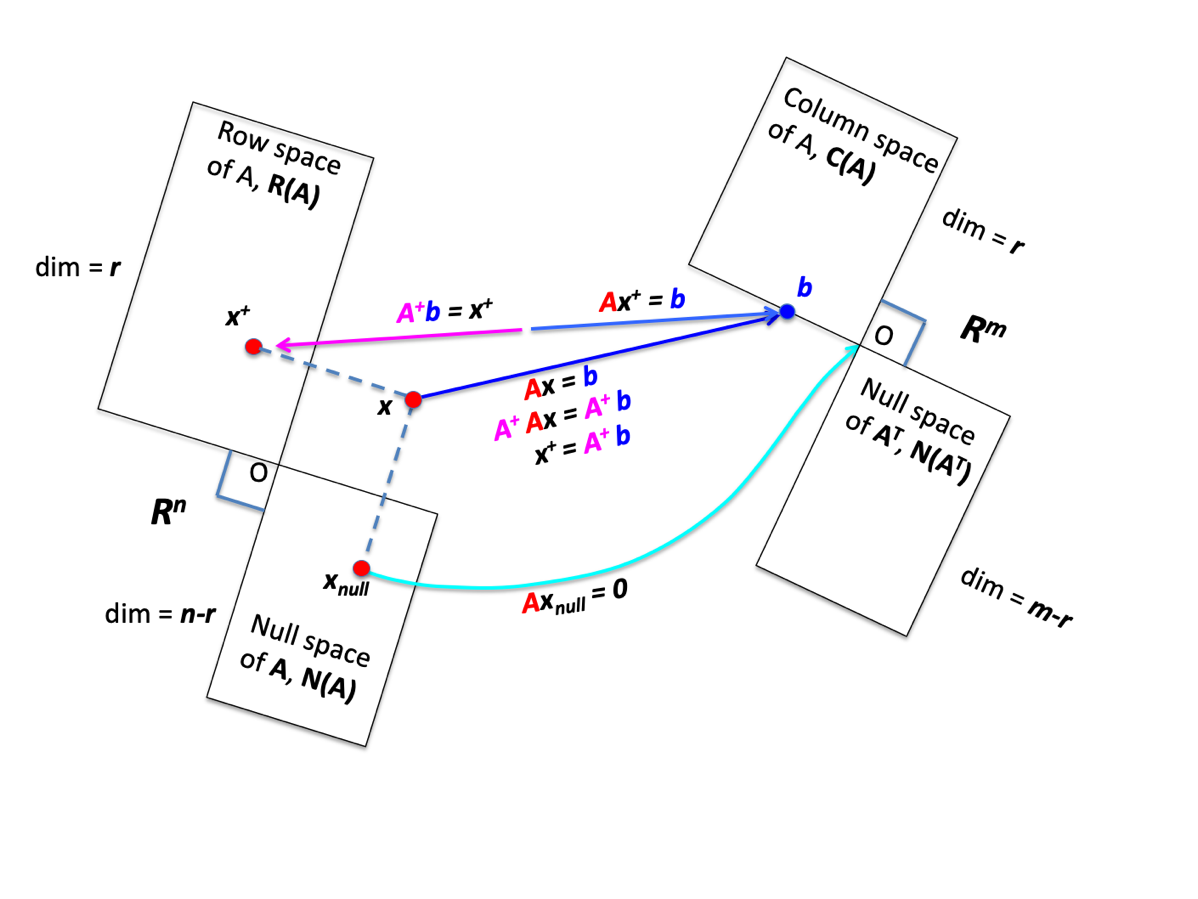
S\_pos = diag(ones(r,1)./diag(S\_svd(:,1:r)))

pinvA = V\_svd(:,1:r)\*S\_pos(:,1:r)\*U\_svd(:,1:r)'

pinvA = pinv(A)

Instead the pseudoinverse of the diagonal matrix gives a product that is as close as possible to the identity matrix (it is partly ***I*** and partly 0).

Notice that if existed, then , instead the products .In fact, both and are projection matrices:

The following scheme clarifies the relationship between ***A*** and ***A+***:

***x*** has projection ***x+*** in the row space and ***xnull*** in the null space***. A*** converts both ***x*** and ***x+*** to ***b*** because ***xnull*** is converted to 0. The pseudoinverse ***A+*** can only converts ***b*** to ***x+*** because the component ***xnull*** was converted to 0 and there is no way to get it back and sum it to ***x+*** to regenerate ***x***. We can check this with a numerical example that uses a singular matrix ***A***:

A = [2 7 6;4 14 12;1 3 3], [m,n] = size(A), rank(A)

RA = orth(A'), NA = null(A)

x\_p = RA(:,1), x\_n = NA

x = x\_p + x\_n

b = A\*x

First, we calculate the pseudo inverse:

inv(A)

pinvA = pinv(A)

x\_p\_rec = pinvA\*b

Now we check by projecting ***x*** onto the row space of ***A***, and we also check that indeed is a projection matrix:

P = RA/(RA'\*RA)\*RA'

P = pinvA\*A

x\_p = P\*x

P\*P

The *pseudoinverse* ***A+*** has some interesting applications. Consider again the above numerical example, but imagine this time we know ***x*** and ***b*** and we want to derive the matrix ***A*** such that . In principle, we would like to obtain ***A*** by multiplying ***b*** by the inverse of ***x***, ***x-1***, but ***x*** is just a column vector and therefore has no inverse. Instead, we can multiply ***b*** by the *pseudoinverse* of ***x***, ***x+***.

We can compare the recovered ***Arecov*** versus the original ***A*** matrix:

pinv\_x = pinv(x)

A\_recov = b\*pinv\_x, A

b = A\*x, b = A\_recov\*x

b = A\*(0.5\*x), b = A\_recov\*(0.5\*x)

b = A\*([1 2 3]'.\*x), b = A\_recov\*([1 2 3]'.\*x)

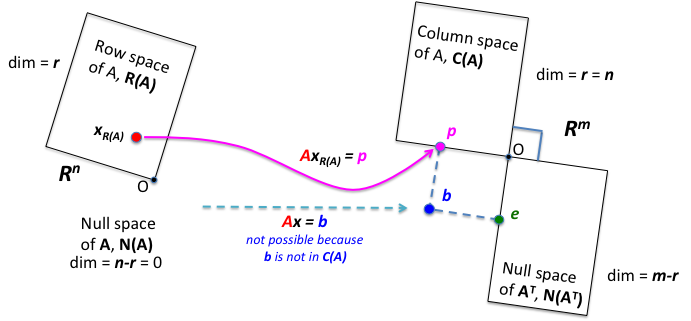
Thus, the two matrices are effectively identical for *that particular vector* ***x*** (or any multiple of ***x***), although they are not for any vector different from ***x***! That is because for every value of ***x*** there is going to be a different specific *pseudoinverse* ***x+***. In fact, the rank of ***A*** is 2, while the rank of ***Arecov*** is 1, and therefore its *column space* contains only 1 line in the direction of ***b***! As a consequence, when ***Arecov*** multiplies any vector ***v*** different from ***x***, the result of the multiplication will always be an exact multiple of ***Ax = b***:

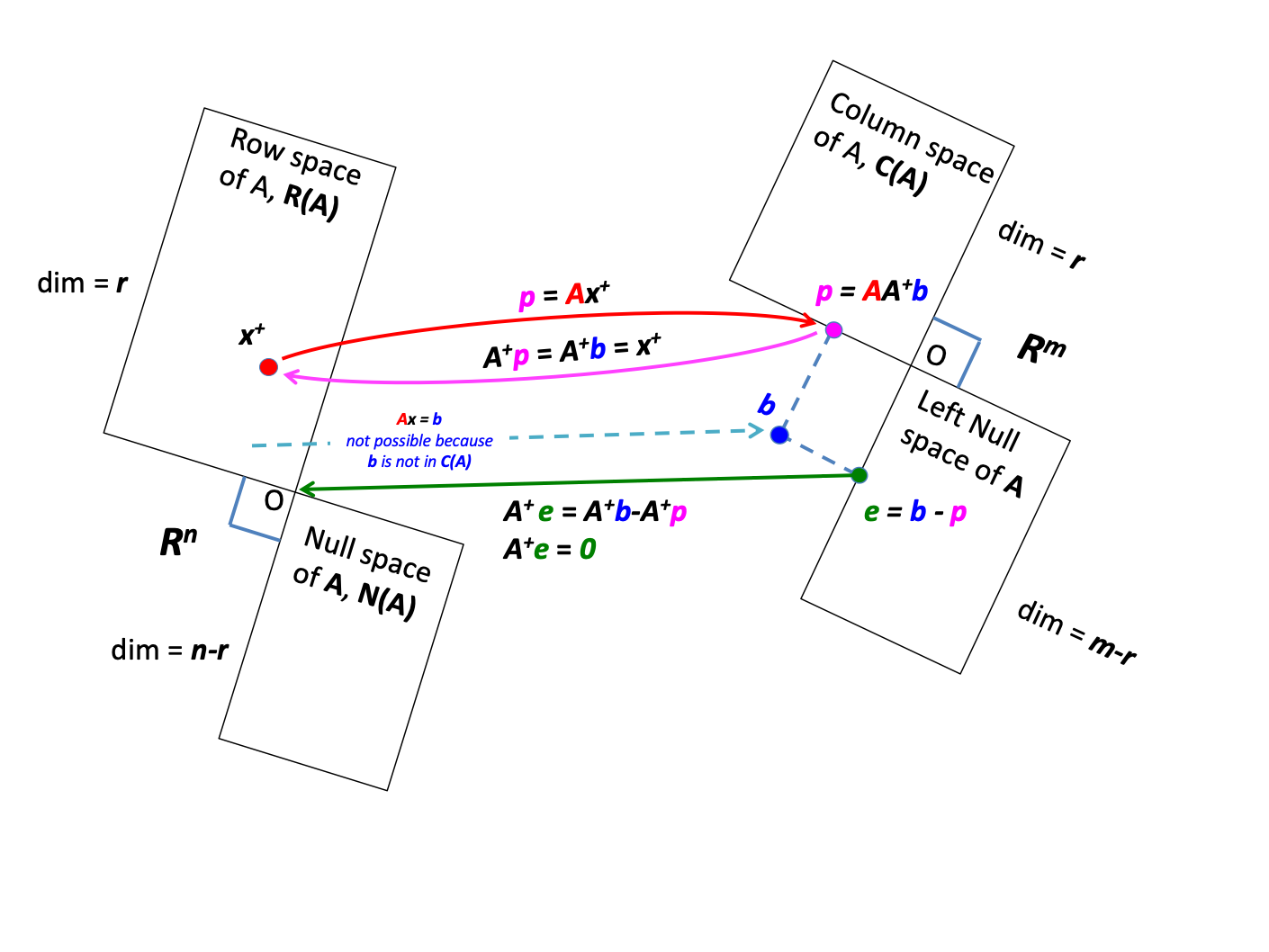
b1 = A\*x

b2 = A\_recov\*([1 2 3]'.\*x)

b2./b1

It also follows that if and are matrices, and , and only for that .

Another application of the *pseudoinverse* is in *least squares* methods. In previous chapters we found that the best solution to an unsolvable system ***Ax*** = ***b*** can be obtained by splitting ***b*** into ***p*** (projection of ***b*** in the *column space* of ***A***) and ***e*** (projection of ***b*** in the *left null space* of ***A***), and using the normal equation ***ATAx*** = ***ATb*** to solve for ***x***, provided ***ATA*** is invertible (rank of ***A*** = *n*, nullspace of ***A*** empty, ***ATA*** is *positive definite*).

However, we may have a situation in which the rank of ***A***is less than *n*, and therefore the null space of ***A*** and ***ATA*** is not empty (***ATA*** is *positive semidefinite*). In this situation, ***ATA*** is not invertible, and thus the *normal equation* cannot be used to find a unique solution ***Ax = p*** where ***p*** is the projection of ***b*** in the column space of ***A***. In fact, there is an infinite number of solutions of the system ***Ax*** = ***p*** because any vector (or multiple of it) from the null space of ***A*** can be added to the projection ***x+*** of ***x*** into the row space of ***A*** to give another solution for ***x*** without affecting the result of the product ***Ax*** = ***p***.

Furthermore, we notice that ,as already the first product,since(in the *left null space* of ***A***) is perpendicular to (the *column space of* ***A***), and therefore ***A+*** converts to **0** the projection of ***b*** in the *left* *null space* of ***A***. As a consequence, , and is the shortest of all possible solutions. In conclusion:

The shortest least-squares solution to is .

A = [2 7 6;4 14 12;1 3 3] , rank(A)

b = [1.2 1.9 3.1]'

AtA = A'\*A , rank(AtA)

inv(AtA)

x = pinv(A)\*b

NA = null(A)

x2 = x -7.6\*NA

b , A\*x , A\*x2

norm(x) , norm(x2)

null(A') , null(pinv(A)) % null space of A and A+ are the same

is the solution with the smallest norm because it does not contain any contribution from the *null space* of ***A***.

SVD and PCA.

The most general application of **SVD** is in the analysis of complex tables in which *m*properties are observed in *n*samples (or viceversa). The goal is always to find a meaning in those *m, n* numbers. The success of the **SVD** is to find the combinations of properties and samples that are uncorrelated. In matrix language, shows the combinations (contained inside corresponding ***U****'s* and ***V****'s*) that produce the diagonal matrix as .

In a typical example each row of ***A*** represents a *sample* (an observation) and each column represents a *property*. In one direction ***C*** = ***ATA*** (*n*x*n*) represents the correlation between properties (*'property correlation matrix'*); in the other direction ***C*** = ***AAT*** (*m*x*m*) represents the correlation between samples (*'sample correlation matrix'*). Some applications focus on ***ATA*** (typically with *m>n*) and others on ***AAT*** (typically with *n>m*). In principle we could analyze both ***ATA*** (*n*x*n*) and ***AAT*** (*m*x*m*) (which are both square matrices) by means of an *eigen decomposition*. However, the big advantage of the **SVD** is that we solve both *eigenvalue* problems at once: we get both sets of *eigenvectors*, the ***U****'s* (for ***AAT***) and the ***V****'s* (for ***ATA***).

As we have seen, the *eigen* factorization of a covariance matrix provides a basis ***S*** (of orthogonal *eigenvectors*) in which the covariance matrix ***C*** is *diagonal*, that is, there is no covariation at all. In fact the diagonal matrix of eigenvalues is *similar* in ***S*** space to ***C***:

The column vectors in the *eigenvector* matrix ***S*** are the *'principal components* *axes*’, most often called the . The are the coordinates of the centered data in *eigenvector* basis, or more simply, the '*principal components*’ of the centered data ***cA***.

When we discussed the PCA derivation by means of the covariance matrix we mentioned that it is frequent practice to organize the data as an *m* x *n* *data matrix* ***X*** with *n* variables as different columns and *m* observations as different rows. In this case, we calculate the *m* x *n* *score matrix* as:

where is the*n* x *n* *loading matrix*. Then, in order to make , the *f-components* PCA model of , we calculate the approximation:

where is *m* x *f* and is the *f* x *n*. This PCA derivation is particularly convenient when used in combination with the SVD of ***X*** as:

which shows that a truncated SVD provides directly both the *score matrix* and the *loading matrix*.

In the case of an *m* x *n* *data matrix* ***X*** with *m* variables and *n* observations, we calculate the *m* x *n* *score matrix* ***T*** as:

where is the*m* x *m* *loading matrix*. Then, in order to make , the *f-components* PCA model of , we calculate the approximation:

where ***Pf*** is the *m* x *f* *loading matrix*, and ***T f*** is the *f* x *n*  *score matrix*. This PCA derivation is particularly convenient when used in combination with the SVD of ***X*** as:

Thus a single SVD provides all the factorizations required for PCA regardless of whether the variables are arranged as rows or columns. Therefore, the difference between the two forms of PCA is only given by the centering of the observations, which in one case, (*m* x *n* *data matrix* ***X*** with *m* observations and *n* variables) requires subtracting from each column the mean of all the observations (= *m* rows) in that column, and in the other case (*m* x *n* *data matrix* ***X*** with *m* variables and *n* observations) requires subtracting from each row the mean of all the observations (= *n* columns) in that rows.

**SPECIAL TOPICS:**

**The Trace operator**

We have seen how the *trace* of a square matrix ***A*** is defined as the sum of the elements on the main diagonal. The trace of a matrix is equal to the sum of the *eigenvalues*, and it is *invariant* with respect to a change of basis. The operation of calculating the trace of a matrix is represented by the *trace operator*, usually abbreviated with the symbol *Tr* or *tr*. The following are some of the most important properties of the *trace* operator:

Tr(***A***+***B***) = Tr(***A***)+Tr(***B***)

Tr(***AB***) = Tr(***BA***)

Tr(**AT**) = Tr(***A***)

Tr(c***A***) = cTr(***A***)

Tr(***ATB***) = Tr(***ABT***) = Tr(***BTA***) = Tr(***BAT***)

Tr(***ABCD***) = Tr(***BCDA***) = Tr(***CDAB***) = Tr(***DABC***) *Cyclic permutations*

Note that in general *non-cyclic permutations* are not allowed:

Tr(***ABC***) ≠ Tr(***ACB***)

Unless is the product of 3 *symmetric* matrices, in which case any permutation is allowed:

Tr(***ABC***) = Tr(***ATBTCT***) = Tr(***AT***(***CB***)***T***) = Tr((***CB***)***TAT***) = Tr((***ACB***)***T***) = Tr(***ACB***)

The trace is similarity-invariant:

Tr(***P-1AP***) = Tr(***A***)

The trace of the *Kronecker tensor* product of two matrices (CHAPTER 1) is the product of their traces:

Tr(***A***⊗***B***) = Tr(***A***)Tr(***B***)

If ***A*** is *symmetric* and ***B*** is *antisymmetric*, then:

Tr(***AB***) = 0

Finally, if we define the *vectorization* operator *vec* as the operator that by concatenating its columns converts a matrix into a column vector, then

Tr(***ATB***) = vec(***A***)***T***vec(***B***)

**RMSD superposition of two conformations of a biological molecule.**

We start by selecting two conformations from an MD simulations as saving them as two arrays (point sets) ***A*** and ***B*** of dimensions *m* x *n*, where *m*=3 is the number of coordinates and *n* is the number of atoms.

close all, clear, clc

trj\_mat = dlmread('DATABASE/md\_trj\_matrix.txt');

[nframes,ncoords] = size(trj\_mat);

n = ncoords/3;

First structure

A = trj\_mat(1,:);

A\_x = A(1:3:end);A\_y = A(2:3:end);A\_z = A(3:3:end);

A = [A\_x;A\_y;A\_z];

Last structure

B = trj\_mat(end,:);

B\_x = B(1:3:end);B\_y = B(2:3:end);B\_z = B(3:3:end);

B = [B\_x;B\_y;B\_z];

We save the original B coordinates

B\_orig = B;

To make sure the two structures are well separated we translate ***A*** by an arbitrary*translation vector*, and rotate ***A*** with respect to ***B*** using an arbitrary *rotation* matrix that we build as the product of 3 consecutive rotation matrices around the *x* axis, the *y’* axis, the *z’’* axis (CHAPTER 2).

Translation

trans\_vec = [12 -24 -9]';

A = A + trans\_vec(:,ones(1,n));

Rotation

psi = 30;theta = 210;phi = 55;

Q1 = [1 0 0;0 cosd(phi) sind(phi);0 -sind(phi) cosd(phi)];

Q2 = [cosd(theta) 0 -sind(theta);0 1 0;sind(theta) 0 cosd(theta)];

Q3 = [cosd(psi) sind(psi) 0;-sind(psi) cosd(psi) 0;0 0 1];

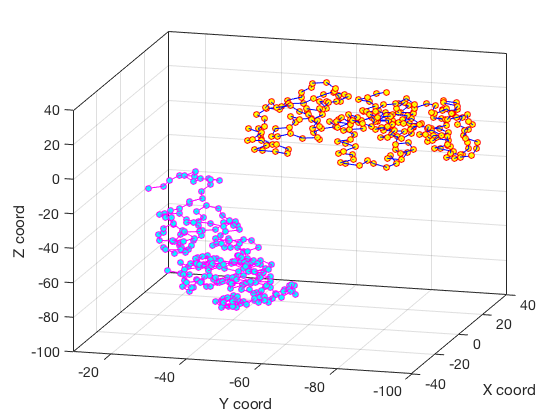
% Q\_for = Q3\*Q2\*Q1;

We take the transpose of each matrix if we want to rotate the actual structure, rather than the frame:

Q\_for = Q1'\*Q2'\*Q3';

A = Q\_for\*A;

We save the original translated and rotated A coordinates

A\_orig = A;

ABrot\_structure = figure;

plot3(A(1,:),A(2,:),A(3,:),'-bo','Linewidth',1.0,...

'MarkerEdgeColor','r','MarkerFaceColor','y')

box('on'); grid('on'); hold on

xlabel('X coord'); ylabel('Y coord'); zlabel('Z coord')

plot3(B(1,:),B(2,:),B(3,:),'-mo','Linewidth',1.0,...

'MarkerEdgeColor','m','MarkerFaceColor','c')

Diff = A - B;

rmsd\_orig = sqrt((Diff(:)'\*Diff(:))/n)

The optimal superposition of ***A*** and ***B*** requires both a translation and a rotation. The translation is obtained by translating the coordinates of one set of coordinates (e.g. set ***A***) so that its centroid coincides with the other set's (e.g. set ***B***) centroid. The centroid of a point set is simply the average position of all its points:

We can then redefine each point in two sets ***A*** and ***B*** as a deviation from the centroid:

Given this notation relative to the centroid, we can explicitly set the centroids to be equal and proceed with the rotational part of the alignment.

mA = mean(A,2); % the centroid of A

mB = mean(B,2); % the centroid of B

A = A - mA(1,ones(1,n)); % translating A to center the origin

B = B - mB(1,ones(1,n)); % translating B to center the origin

If we represent the *centered* coordinates of the two conformations ***A*** and ***B*** with vectors ***a*** and ***b***, we need to find a 3x3 orthonormal matrix ***Q*** such that ***Qa*** aligns as best as possible with ***b***. This problem can be stated as the minimization of the following quantity:

It is worth noting that here ***Q*** cannot be any orthogonal matrix, but only one with *right handed basis*. Therefore, *reflection matrices ,* having determinant = -1, are excluded from the solution. When E is a minimum, the square root of its value becomes the *least RMSD (lRMSD)* between ***a*** and ***b***.

The first attempt to solve this problem in matrix form was offered by Kabsch, who used *Lagrange multipliers* (CHAPTER 7) to solve the minimization problem of finding the optimal rotation ***Q***. However, a simpler and more intuitive method is based on matrix algebra and properties.

If we represent the two point sets, rather than as one-dimensional vectors, by using the original two 3 x *n* matrices (*n* atoms, 3 coordinates for each) ***A*** and ***B***, and we introduce ***Arot*** = ***QA,*** the minimum can be redefined as:

where Tr(***A***) stands for the trace of matrix ***A***. Using the properties of the *trace operator* the right-hand side of the minimum expression can be expanded as:

The first two terms in the expansion above represent the sum of the squares of the components in***arot*** and ***b***, so it can be rewritten as:

Note that the ***a*** components do not need to be rotated (i.e., ***arot***) since the rotation ***Q*** around the origin does not change the length of ***a***. Furthermore, since the first two terms don’t depend on ***Q***,minimizing *E* is equivalent to maximizing . Remembering that ***Arot*** = ***QA***, the quantity to maximize is then:

From the properties of the trace operator, this is equivalent to:

We notice that since ***A*** and ***B*** are centered, the product is a 3 x 3 matrix proportional by *n* to the cross-covariance matrix of the coordinates between the two sets. Singular Value Decomposition of yields:

Again from the properties of the trace operator, we obtain that:

If we introduce the 3x3 matrix , we can rewrite the above expression as:

The last equality derives from the fact that ***T*** is the product of orthonormal matrices, and thus is itself an orthonormal matrix with det(***T***)=+/-1. This means that the absolute value of each element of this matrix is no more than 1. Consequently, the maximum value of the left hand side of the equation is reached when the diagonal elements of ***T*** are equal to 1, and since it is an orthonormal matrix, all other elements must be 0. This results in:

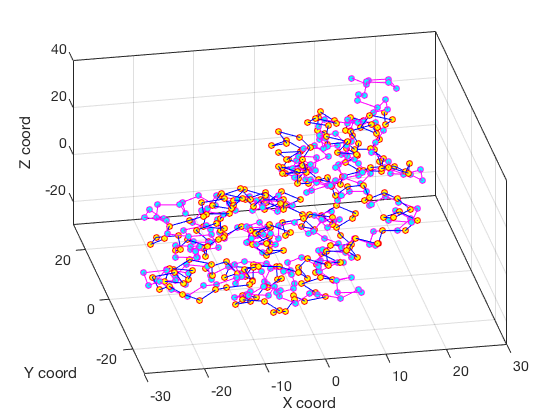
Since both and are orthogonal, multiplying by on the left and by on the right we obtain a solution for ***Q***:

This formula ensures that ***Q*** is orthogonal. The only remaining detail is to make sure that ***Q*** is a *proper rotation*, as it could happen that det(***Q***) = -1 if its rows/columns do not make up a right-handed basis. When this happens, we need to compromise between two goals: maximizing and respecting the constraint det(***Q***)=+1. This is achieved by choosing the second largest value of . Since:

with:

and

then the second largest value occurs when ***T*11**=***T*22**=+1 and ***T*33**=-1. In this case ***T*** is no longer the identity matrix, but has the lower-right corner set to -1. It follows that:



This result provides a unified way to represent the solution. If det(***Q***)>0, ***T*** is the identity; otherwise, it has a -1 as its last element. These facts can be expressed in a single formula for the optimal rotation ***Q***:

where *d* = sign(det(***VUT***)).

D = 3;

C = A\*B'; % cross-covariance matrix

[U,S,V] = svd(C) ; % singular value decomposition

I = eye(D) ;

I(D,D) = sign(det(V\*U'));

Q = V\*I\*U' ;

The translation vector is obtained as the difference between the centroid of the ***B*** point set and the *rotated* centroid of the ***A*** point set:

r = mB - Q\*mA ;

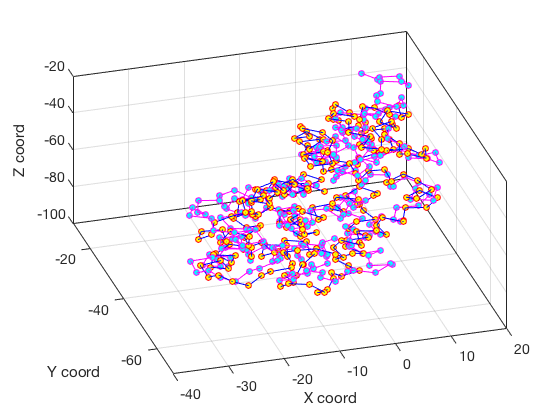
Diff1 = Q\*A - B ;

Arot = Q\*A;

ABrot\_structure = figure;

plot3(Arot(1,:),Arot(2,:),Arot(3,:),'-bo','Linewidth',1.0,...

'MarkerEdgeColor','r','MarkerFaceColor','y')

box('on'); grid('on')

xlabel('X coord'); ylabel('Y coord'); zlabel('Z coord')

hold on

plot3(B(1,:),B(2,:),B(3,:),'-mo','Linewidth',1.0,...

'MarkerEdgeColor','m','MarkerFaceColor','c')

lrmsd = sqrt((Diff1(:)'\*Diff1(:))/n)

We can check that the solution is correct by applying the optimal rotation *followed by* the optimal translation to the original ***A*** coordinates to superimpose ***A*** onto ***B***.

Arot2 = Q\*A\_orig + r(:,ones(1,n));

Diff2 = Arot2 - B\_orig ;

lrmsd2 = sqrt((Diff2(:)'\*Diff2(:))/n)

ABrot2\_structure = figure;

plot3(Arot2(1,:),Arot2(2,:),Arot2(3,:),'-bo','Linewidth',1.0,...

'MarkerEdgeColor','r','MarkerFaceColor','y')

box('on'); grid('on')

xlabel('X coord'); ylabel('Y coord'); zlabel('Z coord')

hold on

plot3(B\_orig(1,:),B\_orig(2,:),B\_orig(3,:),'-mo','Linewidth',1.0,...

'MarkerEdgeColor','m','MarkerFaceColor','c')

Notice how in this case ***A*** has been superimposed to the original position of ***B*** (not centered at the origin).

In conclusion, the algorithm for the RMSD superimposition of two conformations of the same biological molecule can summarized in the following steps:

1. Represent the molecule coordinates as two arrays ***A*** and ***B*** of dimensions 3 x *n* (where *n* is the number of atoms)
2. Center each array by subtracting the coordinates of their *centroids*.
3. Calculate the cross-covariance matrix ***C*** *=* ***ABT***
4. Calculate the SVD of ***C*** as
5. The optimal rotation matrix to superimpose ***A*** to ***B*** is , where*d* is the value of the determinant of .
6. The optimal translation is obtained as the difference between the centroid of the ***B*** array and the *rotated* centroid of the ***A*** array.

A MATLAB function *Kabsch.m* is provided in the directory GENERAL\_SCRIPTS\_FUNCTIONS to carry out the RMSD superposition of a point set ***A*** onto a point set ***B***, which also includes the option of assigning weights to each point.

In our example, the same result achieved with the code shown above, would be obtained by calling the Kabsch function with the following syntax:

[Q\_k,r\_k,lrmsd\_k] = Kabsch(A\_orig,B\_orig)

**SPECIAL TOPICS: Principal Component Regression and Partial Least Squares.**

We have seen how a *multiple linear regression* (MLR) model relates a *response* to *p* *predictors* according to:

Determination of the regression coefficients for this linear model relies on the independence of the predictors. However, when the predictors are correlated and the columns of the *design* *matrix* have an approximate linear dependence, the matrix  becomes close to singular. As a result, the least-squares estimate:

becomes highly sensitive to random errors in the observed response . This situation of *multicollinearity* can arise, for example, when data are collected without a well-planned experimental design. We have already discussed *Stepwise regression* and *Lasso* (CHAPTER 6) as a way to address this problem by *shrinking* the number of predictors, and/or selecting those with minimal redundancy between them.

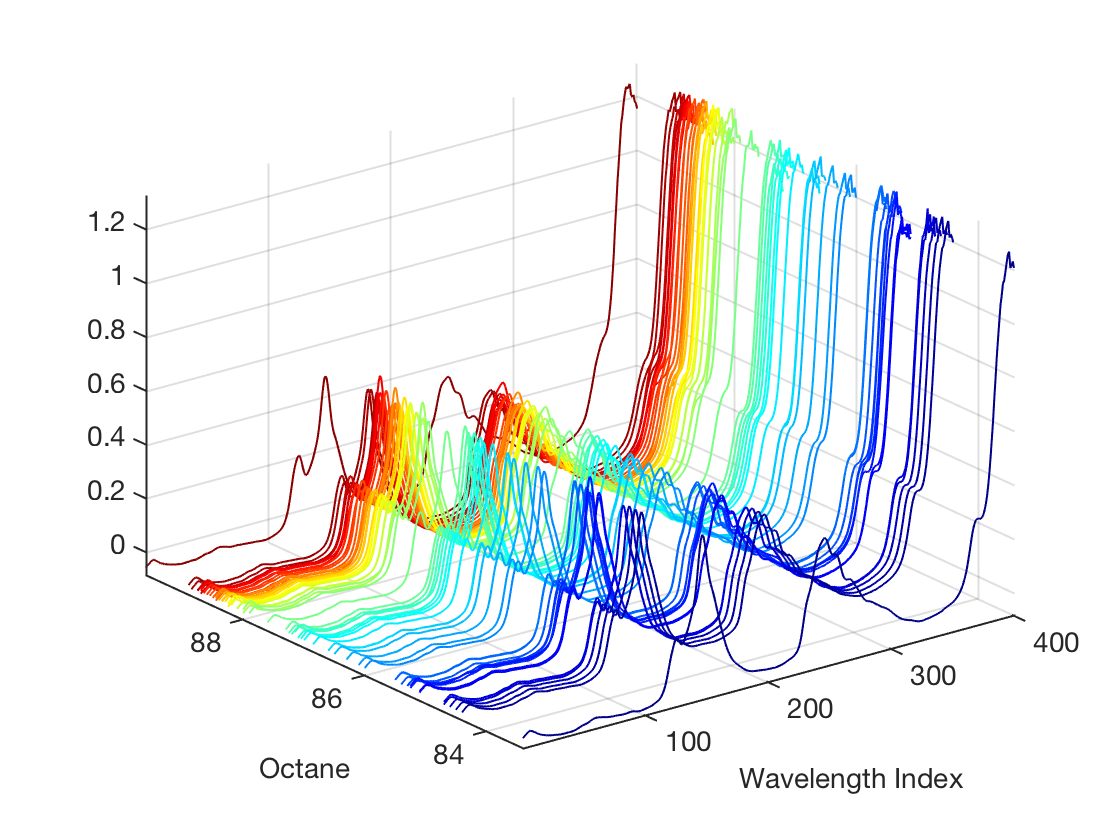
An alternative strategy to removing redundant (correlated) predictors is to use the representation of these predictors into a space where the components that contribute most to the response can be selected: these are usually referred to as the *latent variables* that affect the response.

*Principal Components Regression* (PCR) and *Partial Least Squares Regression* (PLSR) are two of these alternative methods that can be used to model a response (*dependent* variable) when there are a large number of predictors (*independent* variables), and those predictors are highly correlated. Both methods construct new predictor variables, known as *components*, as linear combinations of the original predictor variables. PCR creates these components to explain the observed variability in the predictor variables, without considering the response variable at all. PLSR takes also the response variable into account.

We will use an example derived from the MATLAB Statistics Toolbox to describe a common application of PCR and PLSR. A dataset consisting of the spectral intensities of 60 samples of gasoline at 401 wavelengths, and their octane ratings can be loaded directly from inside MATLAB:

load spectra

whos NIR octane



[~,h] = sort(octane);

oldorder = get(gcf,'DefaultAxesColorOrder');

set(gcf,'DefaultAxesColorOrder',jet(60));

plot3(repmat(1:401,60,1)',…

repmat(octane(h),1,401)',NIR(h,:)');

set(gcf,'DefaultAxesColorOrder',oldorder);

xlabel('Wavelength Index');

ylabel('Octane'); axis('tight');

grid on

X = NIR;

Y = octane;

[n,p] = size(X);

The goal here is to use the information contained in this dataset to develop a *model* that relates the Near-Infra-Red (NIR) spectrum of a generic gasoline sample to its octane value.

To generate a PCR model with *n* principal components we first perform Principal Components Analysis on ***X***, retaining *n* principal components. The calculated *scores* are the *latent variables*: PCR is then just a linear regression of these *n* latent variables on the response variable.

[PCALoadings,PCAScores,PCAVar] = pca(X,'Economy',true);

The following figure shows the fraction of variance in ***X*** explained by the different principal components used in the PCR.

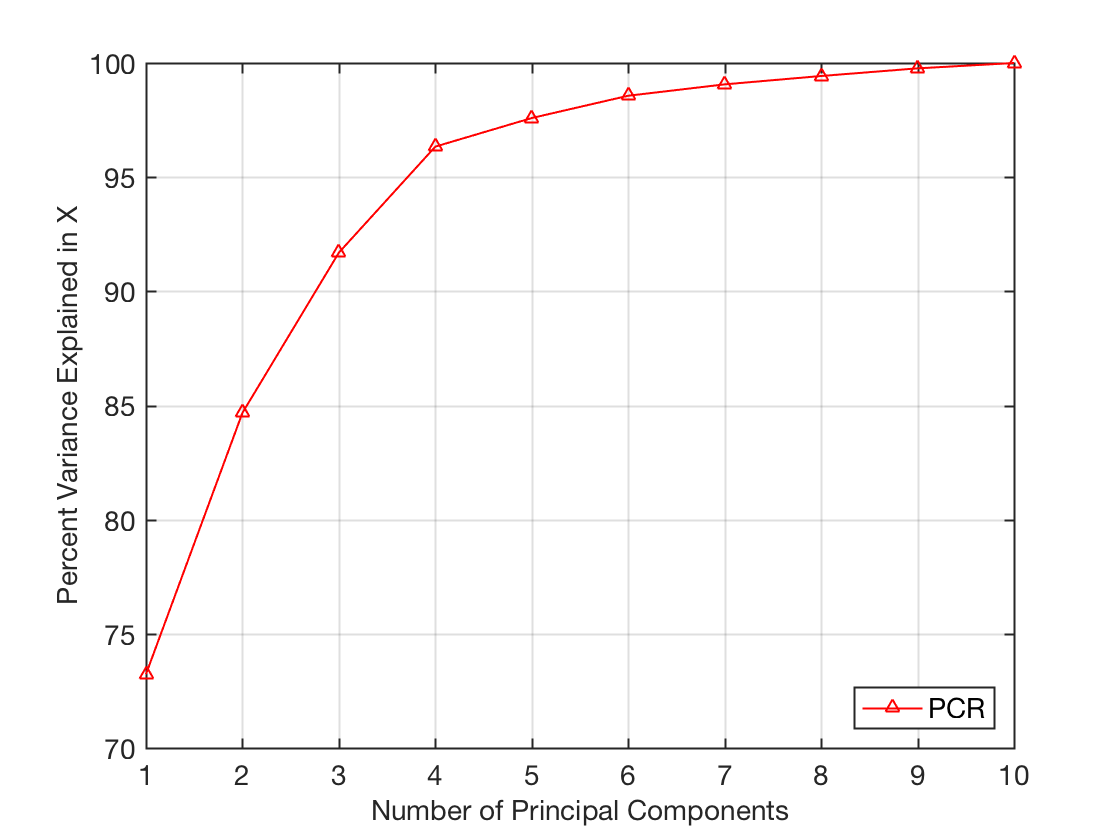
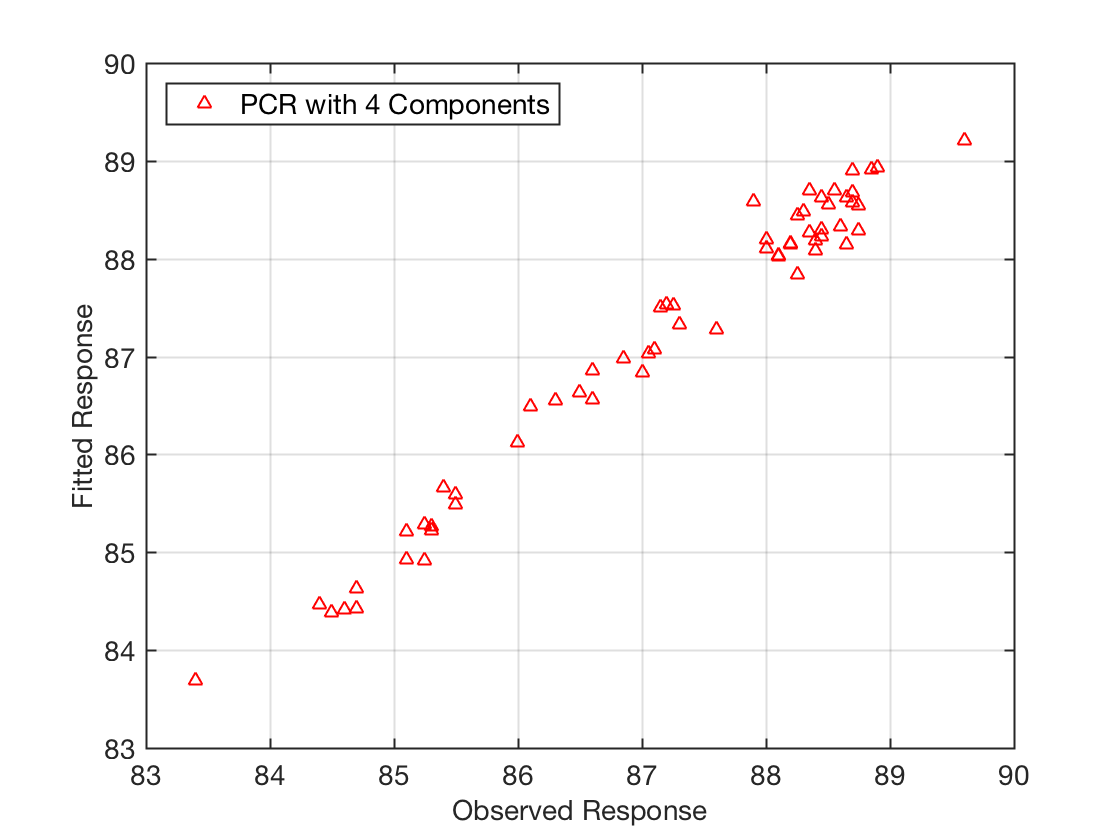
plot(1:10,100\*cumsum(PCAVar(1:10))/sum(PCAVar(1:10)),'r-^');

xlabel('Number of Principal Components');

ylabel('Percent Variance Explained in X');

legend({'PCR'},'location','SE');

grid on



Based on this result we decide to use only 4 *components* in the regression:

betaPCR = regress(y-mean(y), PCAScores(:,1:4));

To make the PCR results easier to interpret in terms of the original spectral data, we bring back the regression coefficients into standard space.

betaPCR = PCALoadings(:,1:4)\*betaPCR;

Since:

we calculate the constant regression coefficient as:

beta0PCR = mean(y) - mean(X)\*betaPCR;

betaPCR = [beta0PCR; betaPCR];

We have a total of 402 regression coefficients:

yfitPCR = [ones(n,1) X]\*betaPCR;

Finally, we plot the fitted vs. the observed response for the PCR fits.

plot(Y,yfitPCR,'r^');

xlabel('Observed Response');

ylabel('Fitted Response');

legend({'PCR with 4 Components'},…

'location','NW');

A useful way to rationalize the PCR process is to use the *psudoinverse* after deriving a PCA reduced data set with only 4 components:

meanX = mean(X);

cX = X - repmat(mean(X),60,1);

[coeff,score,latent] = pca(X);

rX = score(:,1:4)\*coeff(:,1:4)' + repmat(meanX,60,1)

or using explicitly SVD. Since the rank of the augmented matrix [ones(n,1) rX] is 5 we can’t use the normal equation, but the pseudoinverse gives the shortest least squares solution:

[U,S,V] = svd(cX,'econ');

rX = U(:,1:4)\*S(1:4,1:4)\*V(:,1:4)' + repmat(meanX,60,1);

betaPCR\_mp = pinv([ones(n,1) rX])\*Y

Again, we have a total of 402 regression coefficients:

yfitPCR\_mp = [ones(n,1) rX]\*betaPCR\_mp;

We plot fitted vs. observed response for the PCR fit:

plot(Y,yfitPCR\_mp,'r^');

xlabel('Observed Response');

ylabel('Fitted Response');

legend({'PCR with 4 Components'},'location','NW');

grid on

The fitted response obtained using the pseudoinverse is identical to that derived previously, but the norm of the psudoinverve solution for the regression coefficients is of course much smaller:

norm(betaPCR) % ans = 102.9070

norm(betaPCR\_mp) % ans = 39.2641

Finally, we calculate the *r*-squared for the PCR fit:

SST = sum((Y-mean(Y)).^2);

SSE\_PCR = sum((y-yfitPCR).^2);

rsquaredPCR = 1 - SSE\_PCR/SST % rsquaredPCR = 0.9769

We note here that, ideally, the choice of the number of components should be based on the goal of minimizing the expected error when predicting the response from future observations on the predictor variables. *Cross-validation* is the statistically most sound method for choosing the number of components in PCR. It avoids overfitting data by not reusing the same data to both fit a model and to estimate prediction error.

For PCR, the MATLAB function *crossval* combined with the MATLAB function *pcrsse* (which computes the sum of squared errors for PCR), can be used to estimate the *mean squared error of the prediction* (MSEP) using, for example, 10-fold cross-validation.

PCRmsep = sum(crossval(@pcrsse,X,Y,'KFold',10),1) / n;

The MSEP curve for PCR confirms that four components are necessary to get the best prediction:

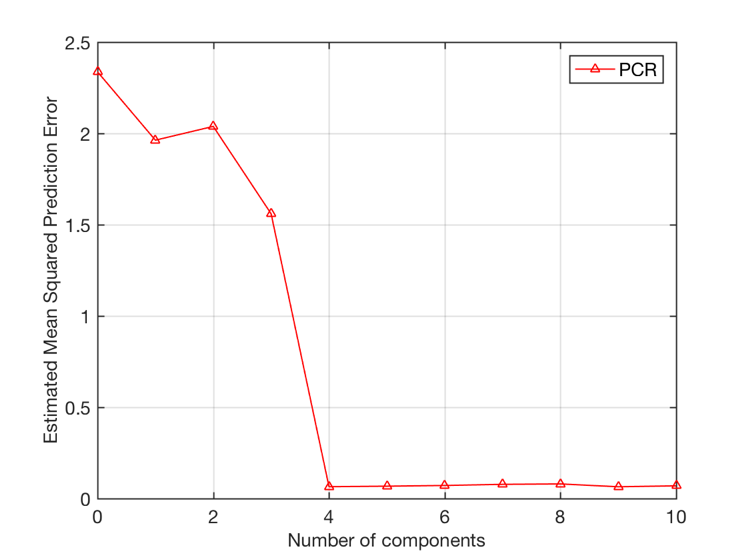
plot(0:10,PCRmsep,'r-^');

xlabel('Number of components');

ylabel('Estimated Mean Squared Prediction Error');

legend({'PCR'},'location','NE');

grid on

In fact, the second component in PCR actually increases slightly the prediction error of the model, suggesting that the combination of predictor variables contained in that component is not strongly correlated with the response ***y***.  Again, that's because PCR constructs components to explain variation in ***X***, not in ***y***.

Now let’s consider the corresponding analysis using *Partial Least Squares (PLS) Regression*.

We have seen that in principal component regression (PCR), the principal components of ***X*** (the *scores*) are used as regressors on ***Y***. The orthogonality of the principal components eliminates the multicollinearity problem. But, the problem of choosing an optimum subset of predictors remains. The PCR strategy is to keep only a few components, but they are chosen to explain ***X*** rather than ***Y***, and so, nothing guarantees that the *principal components*, which “explain” ***X***, are relevant for ***Y***.

By contrast, PLSR finds components from ***X*** that are also relevant for ***Y***. Specifically, PLSR searches for a set of components (latent vectors = scores = factors) that performs a simultaneous decomposition of ***X*** and ***Y*** with the constraint that these components explain as much as possible of the covariance between ***X*** and ***Y***. This step, which generalizes PCR, is followed by a regression step where the decomposition of ***X*** is used to predict ***Y***.

The process starts by considering two matrices ***X*** and ***Y***: the values of *i* observations of *j* predictors are stored in the *i* x *j* matrix ***X***; the values of *i* observations of *k* responses are stored in the *i* x *k* matrix ***Y***.

We can do PCA on both matrices to derive their lower rank approximation:

***X*** *=* ***TPT + E***

***Y*** *=* ***UQT + F***

where, using the language of PCA, ***T*** and ***U*** are the *scores* and ***P*** and ***Q*** are the *loadings*.

The core of PLSR is that it identifies two sets of *scores* ***T*** and ***U*** such that they are maximally *covariant*, that is, ***u1*** is maximally covariant with ***t1***, ***u2*** is maximally covariant with ***t2***, ***u3*** is maximally covariant with ***t3***, and so on.



This means that if we know ***T*** we can directly calculate ***U***. Thus, we can re-write the two PCA factorizations as:

***X*** *=* ***TPT + E***

***Y*** *=* ***TBCT + F***

which shows that PLS regression decomposes both ***X*** and ***Y*** as a product of a common set of orthogonal factors ***T***, with ***TTT = I***, and a set of specific loadings (***PT*** and ***BCT***). In order to accomplish this result, the loadings ***PT*** and ***BCT*** of PLS are vectors rotated with respect to the corresponding ***PT*** and ***QT*** loadings of PCA.

***Y*** is estimated as where ***B*** is a diagonal matrix. The columns of ***T*** (the ***X*** scores) are the *latent vectors* (PLS *components* or *factors*). When their number is equal to the rank of ***X***, they perform an exact decomposition of ***X***. However, they only estimate ***Y*** (i.e., in general is not equal to ***Y***).

The latent vectors ***T*** can be chosen in a lot of different ways. In the original NIPALS (Non-linear Partial Least Squares) algorithm, in order to specify ***T***, after centering, and/or normalizing (z-scoring) ***X*** and ***Y***, PLS regression finds two sets of weights ***w*** and ***c*** in order to create (respectively) a linear combination of the columns of ***X*** and ***Y*** such that their covariance is maximal (note that cov(***X***, ***Y***) = ***XTY***). Specifically, the goal is to obtain a first pair of vectors:

***t1****=* ***Xw1***

***u1****=* ***Yc1***

with the constraints that ***wTw***= 1, ***tTt*** = 1 and cov(***t, u***) = ***tTu*** be maximal. When the first latent vector is found, it is subtracted from both ***X*** and ***Y*** and the procedure is re-iterated until ***X*** becomes a null matrix.

The properties of PLS regression can be analyzed from a sketch of the original algorithm. The first step is to create two matrices: ***E****=* ***X***and ***F****=* ***Y***. These matrices can also be normalized (i.e., transformed into Z scores).

The NIPALS algorithm starts with a random initialization of the ***Y*** space score vector ***u*** and repeats a sequence of the following steps until convergence of , , , and :

(normalize)

(normalize)

The ***X***- and ***Y***-space score vectors ***t*** and ***u*** are then given by:

The rationale behind the NIPALS iteration is not immediately obvious, but it can be understood by noticing that at convergence the iteration sequence can be represented as:

***w*** *=* ***XTu*** *=* ***XTYc*** *=****XTYYTt*** *=****XTYYTXw***

which shows clearly that ***w*** is an eigenvector of ***XTYYTX***, or equivalently, a left singular vector of ***XTY***. Now, if we want to maximize the covariance between ***X*** and ***Y*** we need to represent ***X*** in a space in which the covariance matrix cov(***X, Y***) = ***XTY*** is diagonal. To this end, we do an SVD of ***XTY*** and ***w1*** is the 1st left singular vector of ***XTY*** (1st eigenvector of ***XTYYTX***). In order to carry out a change of basis to ***w*** space, we need recall that here we are using the convention that the variables of ***X*** are the columns (and the rows are the observation), thus:

***t1T*** *=* ***w1TXT***

***t1*** *=* ***Xw1***

Likewise, if we want to maximize the covariance between ***Y*** and ***X*** we need to represent ***Y*** in a space in which the covariance matrix cov(***Y, X***) = ***YTX*** is diagonal. We do an SVD of ***YTX*** and ***c1*** is the 1st left singular vector of ***YTX*** (1st eigenvector of ***YTXXTY***), or equivalently, the 1st right singular vector of ***XTY***.

***u1*** *=* ***Yc1***

which shows that ***t*** and ***u*** are linear combinations of the columns of ***X*** and ***Y***, respectively, and that indeed ***w1*** and ***c1*** are the set of weights that maximize the covariance between ***t1*** and ***u1***.

Once ***t1*** and ***u1*** are obtained we can determine the loading for the 1st PLS component for ***X*** and ***Y***. We recall here that:

***X*** *=* ***TPT***

***XT*** *=* ***PTT***

***XTT*** *=* ***PTTT*** *=* ***P***

Thus:

and likewise:

and since ***u1*** *=* ***t1****b1* we derive:

***t1Tu1*** *=* ***t1Tt1****b1*

*b1 =* ***t1Tu1*** = ***u1Tt1***

Next we subtract (i.e., partial out) the effect of ***t***from both error matrices ***E*** and ***F***. This operation, defined as *deflation* of the original matrices ***E*** and ***F***, is performed in different ways in different implementations of PLS.

In the methods called PLS1 (in which ***y*** is a vector) and PLS2 (in which ***Y*** is a matrix), which are the most frequently used versions of PLS, ***E*** and ***F*** are deflated as follows:

***E****=* ***E − t1p1T****=* ***E - t1t1TE***

***F****=* ***F −*** *b1****t1c1T***

In the method called PLS Mode A, ***E*** and ***F*** are deflated as follows:

***E****=* ***E − t1p1T****=* ***E - t1t1TE***

***F****=* ***F – u1q1T*** *=* ***F -****b1****t1q1T***

The vectors ***t1, u1, w1, c1,*** and ***p1***are then stored in the corresponding matrices, and the scalar *b1* is stored as a diagonal element of ***B***. The sum of squares of ***X*** and ***Y*** explained by the latent vector is computed as ***pTp*** and ***b2***, respectively, and the proportion of variance explained is obtained by dividing the explained sum of squares by the corresponding total sum of squares of ***X*** and ***Y***.

At the end of this iteration, if ***E*** is a null matrix, then the whole set of latent vectors has been found, otherwise the procedure can be re-iterated starting from the new values of ***E*** and ***F***.

Once the iterative process is complete, the dependent variables are predicted using the multivariate regression formula:

but:

***X*** *=* ***TPT***

***T*** *=* ***XPT+***

Where ***PT+*** is the pseudoinverse of ***PT***. It follows that:

where:

This last equation assumes that both ***X*** and ***Y*** have been standardized prior to the prediction.

In order to predict a non-standardized matrix ***Y*** from a non-standardized matrix ***X***, we use which is obtained by reintroducing the original units into and adding a first column corresponding to the intercept (that is, when using the original units, ***X*** needs to be augmented with a first columns of 1, as in multiple regression).

Another form of PLS, denoted BPLS, is particularly popular for the analysis of brain imaging data (probably because it requires much less computational time, which is critical taking into account the very large size of brain imaging data sets). Just like standard PLS regression, BPLS regression starts with the cross-covariance matrix:

***S*** *=**cov(****X,Y****)**=* ***XTY***

SVD decomposition of ***S*** yields:

In BPLS regression, the latent variables for ***X*** and ***Y*** are obtained directly as:

***T*** *=* ***XW***

***U*** *=* ***YC***

Because *BPLS regression* uses a single SVD to compute the latent variables, they will be identical if the roles of ***X*** and ***Y*** are reversed: BPLS regression treats ***X*** and ***Y*** symmetrically.

Another variety of PLS analysis is offered in the MATLAB Statistics Toolbox with the function *plsregress*. This function uses the SIMPLS algorithm, which finds the ***w*** weights by SVD and applies deflation steps directly to the covariance matrix ***XTY***. The criterion of the mutually orthogonal score vectors ***t*** is kept. It has been shown that SIMPLS is equal to PLS1 but differs from PLS2 when applied to the multidimensional matrix ***Y***.

For example, we can use *plsregress* with 10 *components*:

[Xloadings,Yloadings,Xscores,Yscores,betaPLS10, PctVarPLS10] = plsregress(X,Y,10);

The matrix contains the predictor scores (the *PLS components or factors or latent variables*) as linear combinations of the columns (variables) in , the centered vector.   is an *n* x *ncomp* orthonormal matrix with rows corresponding to observations and columns to components.

The process starts with centering the predictor matrix and the response vector (or matrix) to derive and :

cX = X - repmat(mean(X),n,1)

cY = Y - repmat(mean(Y),n,1)

The predictor score matrix is obtained as a linear combination of the columns of :

with an appropriate *weight matrix* .

PLS tries to find a linear decomposition (linear regression model) of c***X*** and c***Y*** such that:

where:

are matrices of regression coefficients (*loadings*) for and , and and are error (noise) terms.

P\_t = T\cX;

Q\_t = T\cY;

Xres = cX - T\*P'

Yres = cY - T\*Q'

The matrix ***U***, the responses that have maximal covariance with each of the PLS components ***T***, is obtained as:

U = cY\*Q;

U = cY\*cY'\*T;

Once the loadings are computed, the above regression model:

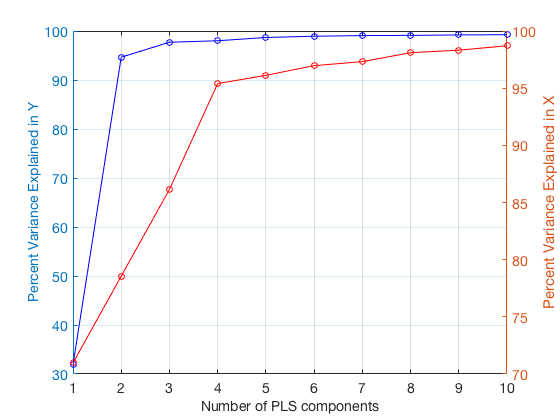
is equivalent to:

which can be used as a predictive regression model, with

From this *beta* the final *betapls* is calculated adding a constant term:

beta = W\*Q';

beta\_pls = [meanY - meanX\*beta; beta];

The fit of a model with up to ten components can be used to make a choice of a simpler model with fewer components. For example, we can plot the percent of variance explained in the predictor and response variable as a function of the number of components.

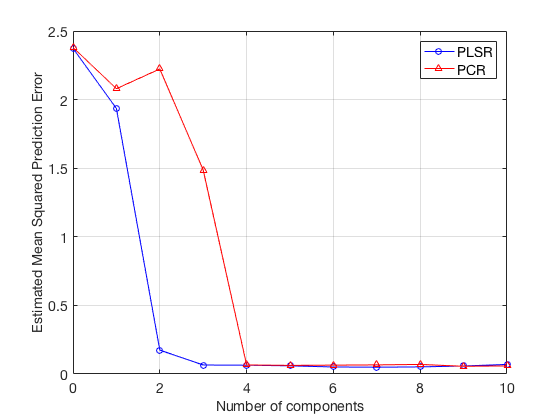
yyaxis left

plot(1:10,cumsum(100\*PctVarPLS10(2,:)),'-bo');

xlabel('Number of PLS components');

ylabel('Percent Variance Explained in Y');

yyaxis right

plot(1:10,cumsum(100\*PctVarPLS10(1,:)),'-ro');

xlabel('Number of PLS components');

ylabel('Percent Variance Explained in x');

This analysis shows that including 4 PLS components we get ~90 of the predictor variance and ~98% of the response variance explained.

However, we can use cross-validation (CV) to estimate the optimal number of components in PLSR.  *plsregress* has an option to estimate the *mean squared prediction error* (MSEP) by cross-validation: here we use 10-fold CV.

[P,Q,T,U,beta,pctVar,PLSmsep] = plsregress(X,Y,10,'CV',10);

The MSEP curve for PLSR indicates that just three components offer optimal performance.  On the other hand, PCR needs four components to get the same prediction accuracy.

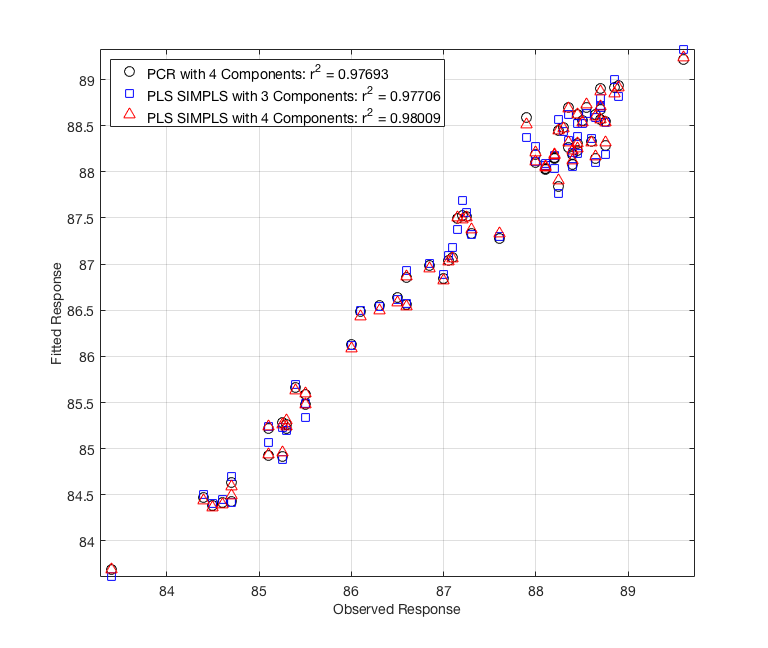
plot(0:10,PLSmsep(2,:),'b-o',0:10,PCRmsep,'r-^');

xlabel('Number of components');

ylabel('Estimated Mean Squared Prediction Error');

legend({'PLSR' 'PCR'},'location','NE');

We can now compare the regression obtained with PCR and PLS using in both cases 4 components, and with PLS using only 3 components:



yfitPLS4 = [ones(n,1) X]\*betaPLS4;

SST = sum((y-mean(y)).^2);

SSE\_PLS4 = sum((y-yfitPLS4).^2);

rsquaredPLS4 = 1 - SSE\_PLS4/SST

[P,Q,T,U,betaPLS3] = plsregress(X,Y,3);

yfitPLS3 = [ones(n,1) X]\*betaPLS3;

SSE\_PLS3 = sum((Y-yfitPLS3).^2);

rsquaredPLS3 = 1 - SSE\_PLS3/SST

We plot fitted vs. observed response for the PLS fit:

PCR\_vs\_PLS = figure

set(PCR\_vs\_PLS,'Unit','Normalized','Position',[0.4 0.4 0.4 0.6])

plot(Y,yfitPCR,'LineStyle','none','Marker','o','MarkerSize',10,'Color','k')

hold on

plot(Y,yfitPLS3,'LineStyle','none','Marker','s','MarkerSize',10,'Color','b')

plot(Y,yfitPLS4,'LineStyle','none','Marker','^','MarkerSize',10,'Color','r')

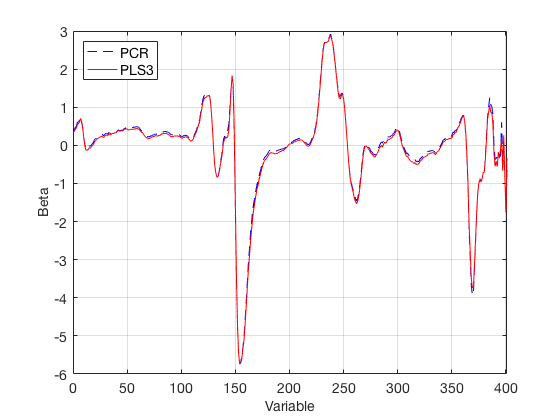
xlabel('Observed Response');

ylabel('Fitted Response');

legend({['PCR with 4 Components: r^2 = ' num2str(rsquaredPCR)] ...

    ['PLS SIMPLS with 3 Components: r^2 = ' num2str(rsquaredPLS3)] ...

    ['PLS SIMPLS with 4 Components: r^2 = ' num2str(rsquaredPLS4)]},'location','NW');

grid on; axis equal

The regression coefficients from the 4 component PCR and the 3 component PLSR are essentially identical:

PCR\_vs\_PLS\_beta = figure

figure;

plot(1:401,betaPCR(2:end),'--b',…

1:401,betaPLS3(2:end),'-r');

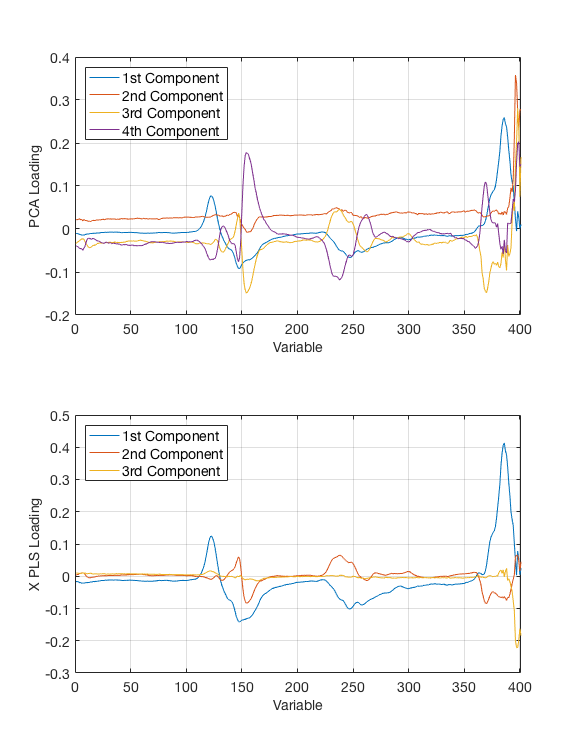
xlabel('Variable');

ylabel('Beta');

xlim([0 401])

legend({'PCR' 'PLS3' },'location','NW');

grid on

We can understand the physical meaning of these coefficients by noticing that for each gasoline the predicted octane rating is the dot product of the IR spectrum times the coefficients. Thus, if a spectrum has a strong positive peak at a wavelength with a big negative coefficients the resulting effect will be a decrease in the predicted octane rating, and viceversa.

As to the meaning of scores and loadings, they are exactly identical in PCR and PLSR. The loadings ***P****’s are a basis for the row space of* ***X***, that is, the individual spectral components whose linear combination determined by the scores ***T*** produces the spectra in ***X***:

PCR\_vs\_PLS\_loadings = figure;

set(PCR\_vs\_PLS\_loadings,'Unit',…

'Normalized','Position',[0.4 0.3 0.3 0.7])

subplot(2,1,1)

plot(1:401,PCALoadings(:,1:4),'-');

xlabel('Variable');ylabel('PCA Loading');

legend({'1st Component' '2nd Component' '3rd Component'  '4th Component'},'location','NW');

xlim([0 401]);grid on

subplot(2,1,2)

plot(1:401,P(:,1:3),'-');

xlabel('Variable');ylabel('X PLS Loading');

legend({'1st Component' '2nd Component' '3rd Component'},'location','NW');

xlim([0 401]);grid on