Scientific Computing, Fall 2022
http://www.math.nyu.edu/faculty/goodman/teaching/SciComp2022/index.html

Scientific Computing

Jonathan Goodman, Fall, 2022

1 Linear Algebra, matrix factorization

Prerequisites. This Section assumes a familiarity with linear algebra at a "good undergraduate level". Specific topics assumed include the relation between matrices and linear transformations, formulation of systems of linear equations in terms of matrices and solution via LU factorization, solvability conditions, subspaces and bases for subspaces, change of basis and invariance of dimension, eigenvalues and eigenvectors. The linear algebra book by Gilbert Strang is a good source for "review".

This Section and the next describe Computational linear algebra. This Section covers some common linear algebra problems and solution strategies that rely on matrix factorizations. Most computational linear algebra is done this way. The user formulates a linear problem and creates a strategy for solving it that uses information from one of the standard matrix factorizations. The corresponding code uses available linear algebra software to compute the factorization. In Python, this linear algebra software is in `numpy.linalg` or in `scipy`.

What's missing from this Section is guidance on how to choose between different strategies to solve a given problem. For example, a linear system of equations may be solved using the LU factorization, or QR or SVD. The LU factorization may be simpler and faster, but the SVD allows you to treat ill-conditioned problems with more numerical stability. Numerical stability and conditioning are not discussed here, but are discussed in future classes. Also delayed is a discussion of the computational algorithms used to get the factorizations. It may seem strange to put the topics in this order, but the other orders seem strange too. The full picture involving problems, approaches, conditioning (perturbation theory) and algorithms should be clear.

Approximate arithmetic, quantitative vs. qualitative properties. Computational linear algebra is an important part of scientific computing because many problems are linear, and because linear approximations are useful in many non-linear problems. The techniques and software of computational linear algebra are so powerful that we look for ways to apply them whenever possible.

Working with linear algebra means going back and forth between concrete and abstract ways of thinking. In the most concrete view a vector is a one-index array of numbers (components) and a matrix is a two-index square or rectangular array. The other extreme is the abstract theory of vector spaces and linear transformations between them. Concrete and abstract reasoning are often used together. For example, there are linear transformations such as the FFT (discussed in a later Section) that are implemented by a code that does not store or compute the entries of the FFT matrix.

Doing practical computational linear algebra can feel different from doing theoretical linear algebra. Theoretical linear algebra has theorems that depend on whether a number is zero, whether eigenvalues are equal, or real, etc. The code `zeroDemo.py` provides an example. One experiment involves the matrix

$$A = \begin{pmatrix} \pi & e & \sqrt{2} \\ e & \sqrt{2} & \pi \\ \pi + e & e + \sqrt{2} & \sqrt{2} + \pi \end{pmatrix}$$

The experiment could work with different entries in the first two rows, so long as they are not represented exactly in floating point. Mathematically, the determinant is zero because the third row is the sum of the first two rows. The computed determinant (double precision arithmetic, top notch algorithm) is `-6.5418e-16`. This number is within roundoff of zero, but it is not actually zero. A second experiment involves the $n \times n$ matrix with entries $A_{jk} = u^{j+k}$. This is an example of a *Vandermonde* matrix and is non-singular. With $u = .9$ and $n = 11$, the determinant was computed to be `-5.3623e-39` (smaller than the determinant of the singular matrix). This computation has high relative accuracy, since it agrees with the Vandermonde formula for the determinant of a Vandermonde matrix. Clearly the computer cannot tell whether A is singular by testing the determinant.

The singularity or inevitability of a matrix is a qualitative question that "the computer" cannot answer, as `zeroDemo.py` shows. Other qualitative properties of a matrix include being positive definite, and being diagonalizable (n linearly independent eigenvectors). Such qualitative questions should be replaced by quantitative ones that the computer has a better chance of answering reliably. Instead of asking whether A is invertible, we can ask how accurately A^{-1} may be computed in floating point, which is a quantitative question. The accuracy of A^{-1} depends on the accuracy of the computer arithmetic (e.g., single or double precision) and the conditioning of the matrix inversion problem. The linear algebra software in `numpy` includes functions that estimate condition number reliably enough to know how accurate a computation of A^{-1} might be.

Computational software should be written and used with the understanding that it will fail on some problems. For example, a routine `matrix_inverse(A)` should fail if the entries of A^{-1} are outside the range of floating point numbers. A deeper discussion of which matrices are invertible in floating point is coming in a future class. All the routines in `numpy.linalg` report failure when appropriate. Any routine that uses this software should look for error reports and act accordingly.

Factorizations and the computational kernal. This is the first of several Sections on numerical linear algebra. This Section focuses on matrix factorizations, what they are and how they used. A *matrix factorization* is an expression of a matrix A as a product of several matrices $A = BC\cdots$, where the factors B, C, etc., have specified properties. In many cases an analysis of a matrix or linear transformation may be formulated as constructing a matrix factorization. Such formulations are useful because there may be several ways to construct a matrix factorization other than the "direct" method that led to it.

The focus on matrix factorizations is a guiding principle of modern computational linear algebra. In several senses, matrix factorizations form the *computational kernel* of that field. If a solution strategy involves matrix factorizations, probably most of the computer time is spent doing the factorizations. Identifying a small set of clearly defined factorizations has allowed the development of amazing (optimized, reliable, easy to use) software that is widely available. Optimizing these core tasks the best way to speed up overall computations. This involves mathematical problems and also The best widely available software computes factorizations of large matrices far faster and more reliably than anything a small team could created by themselves.

There are many good books on linear algebra and numerical linear algebra. There is the basic linear algebra book of Strang and the beautiful but more abstract book of Lax. For numerical linear algebra, the book of Demmel has much detail (some outdated) and important material on conditioning and perturbation theory. A shorter and gentler book is by Trefethen and Bau. The old but still great book by Dahlquist and Björk is a great source for almost anything related to numerical computing, including numerical linear algebra.

2 *LU* factorization

The *LU* factorization, which many readers will be familiar with, most often used to store most of the work needed to solve a system of linear equations. The direct solution of systems of linear equations starts with an *elimination* phase in which variables are 'eliminated" from equations. This allows the "substitution" phase in which the values of the variables are found one by one. For an $n \times n$ matrix, the elimination phase takes $O(n^3)$ operations while substitution takes $O(n^2)$. The *LU* factorization stores the result of the elimination phase.

Here, we give a backwards explanation of *LU* factorization. First we say what it is, then (next Section) we say how accurate it might be, then, finally, we give some hint how it is computed. This re-ordering of the material is motivated in part by the fact that most users will use "black box" software to compute the factorization. The algorithms behind that software are too sophisticated and specialized to be described in a beginning Scientific Computing course. Instead, people need to know how the factors are used and how accurate the computations are likely to be, or when they are likely to be inaccurate.

The *LU* factorization more than just

$$A = LU \ , \ \ L \text{ lower triangular, } U \text{ upper triangular.}$$

In fact, there are square matrices A that cannot be expressed as a product in this way. An example is $A = \begin{pmatrix} 0 & 2 \\ 3 & 4 \end{pmatrix}$. This A is non-singular, so it cannot be written as a product where one of the factors is singular. A strict (lower triangular)∗(upper triangular) would be

$$\begin{pmatrix} 0 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} a & 0 \\ b & c \end{pmatrix} \begin{pmatrix} d & e \\ 0 & f \end{pmatrix}$$

We equate the individual matrix entries on the left and right, which gives

$$\begin{aligned} A_{11} : &\quad 0 = ad \\ A_{12} : &\quad 2 = ae \\ A_{21} : &\quad 3 = bd \\ A_{22} : &\quad 4 = be + cf \end{aligned}$$

The A_{11} equation gives $ad = 0$, which implies $a = 0$ or $d = 0$ (or both). If $a = 0$ then the first factor on the right is $\begin{pmatrix} 0 & 0 \\ b & c \end{pmatrix}$, which is singular. If $d = 0$ then the second factor is singular. Thus, the strict LU factorization implies that one of the factors is singular, which contradicts the fact that the product is non-singular.

The practical LU factorization is either $A = PLU$, or $A = P_r LUP_c$, where the various P matrices represent permutations of the equations or the variables. To motivate this, suppose the example above came from a system of linear equations, written in various ways

$$Ax = b$$

$$\begin{pmatrix} 0 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$\begin{aligned} 0 \cdot x_1 + 2 \cdot x_2 &= b_1 \\ 3 \cdot x_1 + 4 \cdot x_2 &= b_2 \ . \end{aligned}$$

In the latter form, the one that gives individual linear equations, we can list the equations in the opposite order

$$\begin{aligned} 3 \cdot x_1 + 4 \cdot x_2 &= b_2 \\ 0 \cdot x_1 + 2 \cdot x_2 &= b_1 \ . \end{aligned}$$

In matrix form, this is

$$\begin{pmatrix} 3 & 4 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_2 \\ b_1 \end{pmatrix} \quad .$$

The matrix involved has lower/upper factorization

$$\begin{pmatrix} 3 & 4 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 3 & 4 \\ 0 & 2 \end{pmatrix} \ .$$

The LU factorization was possible after a permutation that re-ordered the equations.

The permutation is included in the LU factorization formalism in the form of a permutation matrix P. A permutation is a re-ordering of the numbers $(1, 2, \ldots, n)$. The new order is specified using a mapping π that puts k in position $\pi(k)$. For example, if $n = 3$ and $\pi(1) = 3$, $\pi(2) = 1$, and $\pi(3) = 2$, then the new order defined by π is $(2, 3, 1)$. A mapping $(1, \ldots, n) \xrightarrow{\pi} (1, \ldots, n)$ is a permutation if all values are taken and no value is repeated. The "pidgenhole principle" is that these conditions are equivalent. A *permutation matrix* corresponding to π is the $n \times n$ matrix with all zeros except that column k has a a 1 in row $\pi(k)$. In our example, since $\pi(1) = 3$, column 1 of P is $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$. This has the effect that if $y = Px$, then $y_3 = x_1$. The full permutation matrix is

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} .$$

You can see that $y_{\pi(k)} = x_k$. Column k of P determines the effect of x_k on $y = Px$. The entry in row j determines the effect of x_k on y_j. If this column is all zero except for 1 in row $\pi(k)$, this makes $y_{\pi(k)}$ equal to x_k. That is, the permutation matrix P permutes the entries of the vector.

The 2×2 example above involved the permutation $(1, 2) \to (2, 1)$. The corresponding permutation matrix is $P = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. This matrix has $P^{-1} = P$, which is not true of general permutation matrices. The example has (multiplying by P in the last step)

$$\begin{aligned} P^{-1}A &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 2 \\ 3 & 4 \end{pmatrix} \\ &= \begin{pmatrix} 3 & 4 \\ 0 & 2 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 3 & 4 \\ 0 & 2 \end{pmatrix} \\ A &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 3 & 4 \\ 0 & 2 \end{pmatrix} \\ A &= PLU \ . \end{aligned}$$

Putting the permutation matrix in front of the factors LU corresponds to re-ordering the equations in the equation system, which is equivalent to permuting the rows of the matrix A. It is called *partial pivoting*.

Practical LU factorization does partial pivoting (row exchange) or *full* pivoting (row and column exchange) to make the factorization algorithm stable. The example above was misleading in one sense. It used pivoting to avoid having a

0 in the (1,1) position of the matrix A. That is necessary for A to have a representation in the form $A = LU$ in the mathematical sense. But pivoting is used not mainly to prevent mathematical impossibliities, but to prevent numerical instability that results from dividing by small numbers. Full pivoting results in $A = P_r LU P_c$, where P_r corresponds to permuting the rows of the matrix, and P_c permutes the columns. Permuting the rows is the same as permuting the equations in an equation system. Permuting the columns is the same as permuting the components of the unknown x.

The $A = PLU$ factorization can be used to solve an equation system through permutation, *forward substitution* and *backward substitution*, which together may be called back substitution. We write the equations $Ax = b$ in factored form as $PLUx = b$, then

$$\text{find } c \text{ with } Pc = b \ , \quad (c = P^{-1}b) \tag{1}$$

$$\text{find } y \text{ with } Ly = x \ , \quad (y = L^{-1}c) \tag{2}$$

$$\text{find } x \text{ with } Ux = y \ , \quad (x = U^{-1}y) \tag{3}$$

The final x satisfies $Ax - b$ because

$$\begin{aligned} Ax &= PLUx \\ &= PL(Ux) \\ &= PLy \\ &= P(Ly) \\ &= Pc \\ &= b \end{aligned}$$

The calculation is written out in such detail with parentheses to emphasize the fact that it depends on the associative property of matrix multiplication. Associativity is the key to many clever algorithms is numerical linear algebra.

The steps (1), (2), and (3) are done using “algorithms” rather than explicitly computing the inverse matrices P^{-1}, L^{-1} and U^{-1}. For step (1), we use the definition of P in the form $c_{\pi(k)} = b_k$ to get the numbers c_j (thanks to the pidgenhole principle). For step (2), we write it out writing l_{jk} for the entries of L and using the fact that L is lower triangular with ones on its diagonal:

$$\begin{pmatrix} 1 & 0 & 0 & \cdots \\ l_{21} & 1 & 0 & \cdots \\ l_{31} & l_{32} & 1 & \\ & \vdots & & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \end{pmatrix}$$

We write out the equations corresponding to the rows and say what we do with that information

$$\begin{array}{lrcl}
\text{(row 1)} & y_1 = c_1 & \Longrightarrow & y_1 = c_1 \\
\text{(row 2)} & l_{21}y_1 + y_2 = c_2 & \Longrightarrow & y_2 = c_2 - l_{21}y_1 \\
\text{(row 3)} & l_{31}y_1 + l_{32}y_2 + y_3 = c_3 & \Longrightarrow & y_3 = c_3 - l_{31}y_1 - l_{32}y_2 \\
 & \text{etc.} & &
\end{array}$$

This is forward substitution because we find the y_k in the forward order y_1, then y_2, etc. It is possible to form the matrix L^{-1}, but it is not necessary. The action of L^{-1} can instead be implemented in the form just given.

Getting x from y in step (3) is similar. The matrix U is upper triangular so the matrix form is

$$\begin{pmatrix} \ddots & & & & \\ & & & & \vdots \\ \cdots & & 0 & u_{n-1,n-1} & u_{n-1,n} \\ & \cdots & 0 & & u_{nn} \end{pmatrix} \begin{pmatrix} \vdots \\ \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} \vdots \\ \\ y_{n-1} \\ y_n \end{pmatrix} .$$

The individual equations corresponding to the rows are written out, but starting from the last one, which is row n.

$$\begin{array}{lrcl}
\text{(row } n\text{)} & u_{nn}x_n = y_n & \Longrightarrow & x_n = \dfrac{1}{u_{nn}}\, y_n \\
\text{(row } n-1\text{)} & u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n = y_{n-1} & \Longrightarrow & x_{n-1} = \dfrac{1}{u_{n-1,n-1}} \left(y_{n-1} - u_{n-1,n}y_n \right) \\
 & \text{etc.} & &
\end{array}$$

The components x_k are found one-by-one, starting from x_n. This process is *backward substitution* because it starts from the $k = n$ rather than $k = 1$.

The LU factorization of A can be used to evaluate the determinant.

3 QR factorization, least squares

The QR factorization is a representation of an $m \times n$ matrix A as the product of an $m \times m$ (square) orthogonal matrix Q and an $m \times n$ upper triangular matrix R. R and A have the same shape, which could be square or rectangular. Upper triangular means that the entries of R are zero below the diagonal: $r_{jk} = 0$ if $j > k$. All three cases $m < n$, $m = n$ and $m > n$ occur in applications.

A real square matrix Q is *orthogonal* if it satisfies $QQ^T = I$. This is equivalent to $Q^TQ = I$ because $AB = I$ is equivalent to $BA = I$ for any square matrices A and B. In terms of columns, Q is orthogonal if its columns are an ortho-normal basis for $\mathbb{R}^n$. To see this more explicitly, write Q in terms of its columns $q_k \in \mathbb{R}^n$

$$Q = \begin{pmatrix} \vdots & & \vdots \\ q_1 & \cdots & q_n \\ \vdots & & \vdots \end{pmatrix} .$$

We write Q^TQ as

$$Q^TQ = \begin{pmatrix} \cdots & q_1^T & \cdots \\ & \vdots & \\ \cdots & q_n^T & \cdots \end{pmatrix} \begin{pmatrix} \vdots & & \vdots \\ q_1 & \cdots & q_n \\ \vdots & & \vdots \end{pmatrix}$$

For any matrix Q (orthogonal or not), the (j,k) entry of Q^TQ seen to be $q_j^Tq_k$. If $Q^TQ = I$, then $q_j^Tq_k = 0$ when $j \neq k$ so the column vectors q_k are orthogonal to each other. The diagonal entries of the identity matrix are 1, so $q_j^Tq_j = 1$. Recall that if x and y are any two column vectors, then (using various notations for inner product)

$$x^Ty = \sum_{k=1}^{n} x_ky_k = x \cdot y = \langle x, y \rangle .$$

The *2 norm* of a vector x is given by

$$\sqrt{x^Tx} = \left(\sum_{k=1}^{n} x_k^2 \right)^{\frac{1}{2}} = \|x\|_2$$

The columns of an orthogonal matrix are "normalized" to have $\|q_j\|_2 = 1$. A system of vectors that are normalized and orthogonal to each other is an *orthonormal* system.

Orthogonal matrices have the important property of preserving norms and inner products of vectors. This means that if x and y are any two column vectors in $\mathbb{R}^n$, then

$$\langle Qx, Qy \rangle = \langle x, y \rangle .$$

This is verified by a matrix/vector calculation, which uses the fact that matrix/vector multiplication is associative ($(AB)C = A(BC)$, $(AB)(CD) = A(BC)D$, etc.) and transpose reverses order ($(AB)^T = B^TA^T$):

$$\begin{aligned} \langle Qx, Qy \rangle &= (Qx)^T (Qy) \\ &= \left(x^TQ^T\right)(Qy) \\ &= x^T \left(Q^TQ\right) y \\ &= x^TIy \\ &= x^Ty \\ &= \langle x, y \rangle . \end{aligned}$$

In particular, orthogonal transformations preserve orthogonality and the preserve length

$$\begin{aligned} \langle x,y\rangle = 0 \Longrightarrow \langle Qx,Qy\rangle = 0 \qquad & \text{(preserve orthogonality)} \\ \|x\|_2 = \|Qx\|_2 \ . \qquad & \text{(preserve length)} \end{aligned}$$

Computations with orthogonal matrices tend to be more stable than computations with general matrices.

The QR factorization is simpler from a mathematical point of view than the LU factorization is because there is no permutation matrix P and no problem with the factorization not existing in certain cases. The QR factorization is more expensive than LU both in operation counts and in memory needed. The number of multiplies needed to find Q and R is roughly twice the number needed to find P, L, and U, as we will see in a later Section. The memory needed also is more, but how much more depends on the implementation. Suppose $n = m$ (square matrices). For LU, you can store all the entries l_{jk} and u_{jk} in a single $n \times n$ array, because off the diagonal either $l_{jk} = 0$ or $u_{jk} = 0$. The diagonal may be used to store u_{kk} because $l_{kk} = 1$ does not need storing. It is impossible to pack both Q and R into a single $n \times n$ array.

If you use QR rather than LU to solve a system of equations, it may be because QR is more stable or because you want to use the information in R to take action if A is badly conditioned. A later Section will discuss conditioning and stability. For now, just note that if $A = QR$ and A is square, then the linear system $Ax = b$ may be solved using $Q^TQ = I$ and back substitution

$$\begin{aligned} Ax &= b \\ QRx &= b \\ Rx &= Q^Tb \qquad (4) \\ x &= R^{-1}\left(Q^Tb\right) \qquad (5) \end{aligned}$$

The step (4) is done by direct matrix/vector multiplication to get $y = Q^Tb$. The step (5) is done using back substitution on the upper triangular system $Rx = y$.

3.1 Least squares via QR

Least squares problems arise when you try to choose a small set of parameters to fit a lot of data using a linear model. An abstract version of “linear model” is m equations that try to predict the numbers b_j using parameters x_k. There are n parameters x_k and m “data values” b_j. The *residuals* in the fitting are m numbers r_j. The definitions are

$$\sum_{k=1}^{n} a_{jk}x_k = b_j + r_j \ . \qquad (6)$$

In matrix form, there is the model matrix A, the parameter vector x, the data vector b and the residual vector r, which satisfy

$$Ax = b + r \ . \qquad (7)$$

The *linear least squares* problem is to find x that minimizes the sum of squares of the fitting residuals,

$$\sum_{j=1}^{m} r_j^2 = \|r\|_2^2 \ .$$

The goal of *least squares* is to choose the parameters x_k to make the sum of squares of the residuals as small as possible – achieve the “least squares”.

Throughout scientific computing, *residual* means the amount by which some equations are not satisfied. It is generally the case that the computer can (approximately) evaluate residuals. The *error* is the difference between an estimated answer and the true mathematical answer. If you could evaluate the error, you would add it to the approximate solution and have the exact solution. The residuals can be known but the errors cannot be. The equations here, written in matrix form, are $Ax = b$. If there are more equations than fitting parameters, which is $m > n$, then it is probably impossible to find an x that satisfies the equations exactly. The question then becomes: how small can the residuals be? If you measure residuals in the $2-$norm, this is the linear least squares problem

The QR factorization of A can be used to solve the least squares problem

$$\min_x \|Ax - b\|_2^2 \ . \tag{8}$$

To see this, use $A = QR$ and the properties of orthogonal matrices:

$$\begin{aligned} \|QRx - b\|_2^2 &= \left\|Q^T\left(QRx - b\right)\right\|_2^2 \\ &= \left\|Q^TQRx - Q^Tb\right\|_2^2 \\ &= \left\|Rx - \widetilde{b}\right\|_2^2 \ . \end{aligned}$$

This shows that we can solve the least squares problem (8) in three steps,

(step 1)	compute the QR factorization of A
(step 2)	compute $\widetilde{b} = Q^Tb$
(step 3)	solve the triangular least squares problem:

$$\min_x \left\|Rx - \widetilde{b}\right\|_2^2 \tag{9}$$

The work needed for step 1 is $O(nm^2)$. Future classes will have some material on work estimates and factorization algorithms. The work for steps 2 and 3 respectively is $O(m^2)$ and $O(n^2)$. Both of these are much smaller than the work for step 1. Matrix factorization, usually, is the most expensive part of a large numerical linear algebra computation.

The QR factorization makes it easy to solve least squares problems because Step 3, and the solution of (9), is easy when R is upper triangular. The quantity being minimized is $\widetilde{r} = Q^T r$. The element of $\widetilde{r}$ are found explicitly as

$$\begin{pmatrix} R_{11} & R_{12} & & \cdots & & R_{1n} \\ 0 & R_{22} & & & & R_{2n} \\ \vdots & & \ddots & & & \vdots \\ & & & \ddots & & \\ & 0 & & & R_{n-1,n-1} & R_{n-1,n} \\ 0 & & \cdots & & 0 & R_{nn} \\ 0 & \cdots & & & \cdots & 0 \\ \vdots & & & & & \vdots \\ 0 & \cdots & & & \cdots & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \\ x_{n-1} \\ x_n \end{pmatrix} - \begin{pmatrix} \widetilde{b}_1 \\ \widetilde{b}_2 \\ \vdots \\ \\ \widetilde{b}_n \\ \widetilde{b}_{n+1} \\ \vdots \\ \widetilde{b}_m \end{pmatrix} = \begin{pmatrix} \widetilde{r}_1 \\ \widetilde{r}_2 \\ \vdots \\ \\ \widetilde{r}_n \\ \widetilde{r}_{n+1} \\ \vdots \\ \widetilde{r}_m \end{pmatrix}$$

This tells you how to choose the components of x to minimize $\widetilde{r}$. Component n is

$$R_n x_n - \widetilde{b}_n = \widetilde{r}_n \ .$$

You achieve $\widetilde{r}_n = 0$ by taking $R_{nn} x_n = \widetilde{b}_n$. After this, you set $\widetilde{r}_{n-1} = 0$ by solving for x_{n-1} in

$$R_{n-1,n-1} x_{n-1} + R_{n-1,n} x_n = \widetilde{b}_{n-1} \ .$$

Everything in this equation is already known, except x_{n-1}. This backward substitution process can continue until all the components of x are found and the first n residuals $\widetilde{r}_1, \ldots, \widetilde{r}_n$ are set to zero. This is the smallest residual you can make, because the rest of the residual components are independent of x because of the zeros in the matrix R. The remaining residuals are equal to the remaining $\widetilde{b}_j$. As a result, the solution to the least squares problem (8) satisfies

$$\min_x \|Ax - b\|_2^2 = \sum_{j=n+1}^{m} \widetilde{b}_j^2 \ .$$

3.2 Orthogonal complements via QR

Suppose you have $m < n$ linearly independent vectors in $\mathbb{R}^n$. It is common (for example in algorithms for constrained optimization) to seek a complementary set of $n-m$ vectors that are orthogonal to this subspace. More precisely, suppose the vectors are $a_1, \cdots, a_m$. We want vectors $q_{m+1}, \cdots, q_n$ that are linearly independent and orthogonal to the vectors a_k, which can be written as

$$q_j^T a_k = 0 \ , \ \text{if } j > m \ , \ \ k \leq m \ . \tag{10}$$

To do this, let A be the $n \times m$ matrix A and take its QR factorization. The desired vectors are the last $n - m$ columns of the orthogonal matrix Q. The orthogonality conditions (10) can be expressed in this matrix notation as

$$q_j^T A = 0 \ .$$

The components in $q_j^T A$ are the numbers $q_j^T a_k$. With the QR factorization, we can write the matrix orthogonality conditions as

$$q_j^T QR = 0 \; .$$

But q_j is orthogonal to every column of Q except column j, so all the entries of $q_j^T Q$ are zero except entry j, which is equal to 1, which may be written as e_j^T. (e_j is the column vector with all zeros except a 1 as component j.) Finally $e_j^T R = 0$ because R has all zero entries below row m (being upper triangular). The algebra is

$$q_j^T A = q_j^T QR = \left(q_j^T Q\right) R = e_j^T R = 0 \; .$$

If you have m vectors a_j, your code might have to copy their components into an $n \times m$ matrix A. Then you ask `numpy` for the QR factorization. Then you copy out the last $n - m$ columns of Q. Note that the vectors from Q of the QR factorization are orthogonal to each other, which implies that they are linearly independent. For many applications you don't need the q_j to be orthogonal. But orthogonality guarantees that the q_j you get will be "well conditioned", in the sense that they will not seem almost linearly dependent to the computer.

4 Symmetric matrices, Cholesky and LDL^T

Many computational linear algebra problems involve symmetric matrices. The normal equations (21) are one example. The *hessian* matrix of a function $f(x)$ of n variables is the $n \times n$ matrix of second partial derivatives

$$H_{jk} = \frac{\partial^2 f}{\partial x_j \, \partial x_k} \; .$$

The Hessian matrix is symmetric. If $A^T = A$, it is natural to look for a factorization of A that preserves the symmetry.

A large symmetric matrix can be stored in roughly half the space of a general $n \times n$ matrix. You don't have to store $A_{jk} = A_{kj}$ if you're also storing A_{kj}. You can, for example, store just the elements in the lower triangle, A_{jk} with $j > k$, plus the diagonals. This is $\frac{1}{2}n(n+1) \approx \frac{1}{2}n^2$ elements.

An LU factorization of A would require n^2 elements if L and U are unrelated. Also, if A is symmetric and $A = LU$, then $A = A^T = U^T L^T$. This is also an LU factorization, because U^T is lower triangular and L^T is upper triangular. If the factorizations are the same, then $U^T = L$, which would give

$$A = LL^T \; . \tag{11}$$

A factorization of this form is a *Cholesky* factorization. The number of elements of L is the same as the number of possibly distinct elements of A, which is approximately $\frac{1}{2}n^2$, so the Cholesky factorization, should there be one, would half as much storage as an LU factorization. The algorithm for computing L, if

L exists, takes half the work of the algorithm to find the LU factorization of an $n \times n$ matrix.

A matrix with a Cholesky factorization must be *positive* (semi) *definite.* A matrix A is *positive definite* if

$$x^T A x > 0 \; , \quad \text{if } x \neq 0 \; .$$

The matrix is positive *semi* definite if

$$x^T A x \geq 0 \; , \quad \text{for all } x \; .$$

If A is positive semi-definite and non-singular, then A is positive definite. This makes semi-definiteness a borderline case. If A has a Cholesky factorization (11) the A is positive semi-definite, because

$$x^T A x = x^T \left(L L^T \right) x = \left(x^T L^T \right) (Lx) \geq 0 \; .$$

The converse is also true: if A is positive semi-definite then A has a Cholesky factorization of the form (11).

A general symmetric matrix is not positive semi-definite. It turns out that any symmetric matrix may be written in the form

$$A = L D L^T \; . \tag{12}$$

Here, L is a lower triangular matrix with $L_{jj} = 1$ (all ones on the diagonal) and D being a diagonal matrix. The work to calculate L and D is essentially the same as the work in finding the Cholesky factorization (when that exists). The storage also is the same, since you can store the n diagonal elements D_{jj} in the spaces you would have used for the diagonal entries L_{jj}. In LDL^T, the ones on the diagonal of L do not have to be stored explicitly.

If you have the LDL^T factorization (12), then you can tell whether A is positive definite or positive semi-definite by looking at the diagonals D_{jj}. We want to know the sign of $x^T A x$. Define a new variable $y = Lx$. This is a one-to-one transformation because L is invertible if L is lower triangular and has ones on the diagonal, so $x = L^{-1} y$. Then

$$\begin{aligned} x^T A x &= x^T L^T D L x \\ &= \left(x^T L^T \right) D \, (Lx) \\ &= y^T D y \\ &= \sum_{j=1}^{n} y_j^2 D_{jj} \; . \end{aligned}$$

If all the D_j are positive, then this is positive whenever $y \neq 0$, which means A is positive definite. If there is a j with $D_{jj} < 0$, then we may choose $y_j = 1$ and $y_k = 0$ for $k \neq j$. This y has $y^T D y = D_{jj} < 0$, so A is definitely *indefinite.* The borderline case is where $D_{jj} \geq 0$ for all j but some of the D_{jj} may be equal to zero.

5 Principal components and SVD

The *singular value decomposition*, usually called the *SVD*, of a matrix A is a factorization of the form

$$A = U\Sigma V^T \ . \tag{13}$$

Assuming A is $m \times n$, the matrices have the form

- U is $m \times m$ and $U^TU = I$ (U is orthogonal)
- V is $n \times n$ and $V^TV = I$ (V is orthogonal)
- Σ is $m \times n$ and "diagonal" in the sense that $\Sigma_{jk} = 0$ if $j \neq k$.
- The diagonal entries of Σ are $\Sigma_{jj} = \sigma_j \geq 0$ and $\sigma_1 \geq \sigma_1 \geq \cdots$.

The σ_j are the *singular values* of A. The columns of V are the *right singular vectors* and the columns of U are the *left singular vectors* of A. The SVD of A is often called the *principal component analysis*, or *PCA*, of A. This term is not as precise as SVD, and it is not clear whether the columns of U or the columns of V are the principal "components".

The SVD has many derivations and many applications.

5.1 Low rank approximation

The *rank* of a matrix A is the dimension of the space spanned by its columns. You might call this the "column rank", but a fundamental theorem of linear algebra is that the column rank is equal to the row rank (dimension of the subspace spanned by the rows). This "row rank equals column rank" theorem may seem surprising because the subspaces involved are different. The column space is column vectors with m components. The row space is row vectors with n components. They are different even if $m = n$ because row vectors are not column vectors.

If the matrix A has rank 1, then every column of A is in the same one dimensional subspace. This means there is a single vector, $w \in \mathbb{R}^m$ so that the columns of A satisfy $a_j = x_j w$, for $j = 1, \cdots, n$. This means that the entries of a_j are $a_{kj} = x_j w_k$. To summarize, A is a rank one matrix if and only if there are column vectors $w \in \mathbb{R}^m$ and $x \in \mathbb{R}^n$ so that

$$A = wx^T \ .$$

Similarly, a matrix has rank r if there are vectors w_i and x_i so that

$$A = \sum_{i=1}^{r} w_i x_i^T \ . \tag{14}$$

There are several seemingly different ways to come to the SVD. We give *variational principles* that define the SVD. A variational principle for something is a definition of that thing as the solution of an optimization problem. In linear algebra, optimizing (maximizing or minimizing) a sum of squares often leads to orthogonality. For the SVD, it is the singular vectors that are orthogonal to each other.

A simple drawing in the plane illustrates the orthogonality principle. Draw any straight line L that does not go through the origin. Then, identify the point $u = (u_x, u_y)$ on L closest to the origin. Observe that the line from the origin to u is orthogonal to L. In formulas, L is defined by a linear constraint $ax + by = c$. We require that a and b are not both zero in order that the "line" not include the whole plane. We require $c \neq 0$ so that L does not go through the origin. The closest point on L to $(0,0)$ satisfies

$$\text{minimize } x^2 + y^2 \ \text{ under the constraint } ax + by = c \ .$$

Minimizing the sum of squares is equivalent to minimizing the length because one is the square root of the other

$$\|(x,y)\|_2 = \sqrt{x^2 + y^2} \ .$$

5.2 Fitting A as a linear transformation

Let A be an $m \times n$ matrix, and x an n component column vector. Then $y = Ax$ is an m component column vector. The SVD expresses the mapping

$$x \ \xrightarrow{A} \ y = Ax$$

as a sum of rank one mappings

$$x \ \xrightarrow{\sigma_j U_j V_j^t} \ y_j = \sigma_j U_j V_j^T x \ .$$

Here is a a sequence of optimizations that gives the pieces

$$x \ \longrightarrow \ \sigma_j U_j V_j^T x \ .$$

First the principal singular value contribution ($j = 1$, $\sigma_1 = \sigma_{\max}$), then the next largest ($j = 2$), and so on.

The principal singular value may be found using a variational definition

$$\sigma_1 = \max_{\|x\|_2 = 1} \|Ax\|_2 \ . \tag{15}$$

Let V_1 be a unit vector that achieves this maximum:

$$\|V_1\|_2 = 1 \ , \ \ \|AV_1\|_2 = \sigma_1 \ .$$

As this says, AV_1 is a vector of length σ_1. This vector may be written as a unit vector, multiplied by σ_1, as in:

$$AV_1 = \sigma_1 U_1 \ , \ \ \|U_1\| = 1 \ .$$

Being careful the way mathematicians are, note that σ_1 is uniquely defined by the variational principle (15), but neither V_1 nor U_1 is defined uniquely. For one thing, we could use $-V_1$ and $-U_1$ instead of V_1 an U_1. But there can be more non-uniqueness than just changing sign. The identity matrix is an extreme example, where V_1 can be any unit vector, corresponding to the principal (and only) singular value $\sigma_1 = 1$.

The next right singular vector V_2 has a more complicated variational principle. We ask for the largest stretch, constraining x to be orthogonal to V_1

$$\sigma_2 = \max_{\|x\|_2=1,\ V_1^T x=0} \|Ax\|_2 \ . \tag{16}$$

We let V_2 be an optimizer:

$$\sigma_2 = \|AV_2\|_2 \ , \ \ \|V_2\| 2 = 1 \ , \ \ V_1^T V_2 = 0 \ .$$

As we did for V_1, we write AV_2 as σ_2 multiplied by a unit vector

$$AV_2 = \sigma_2 U_2 \ , \ \ \|U_2\|_2 = 1 \ .$$

We know $\sigma_2 \leq \sigma_1$ because the variational principle (16) has an extra constraint (optimizes over fewer vectors x) than (15). We do not, yet, know that U_2 is orthogonal to U_1. That extra piece of orthogonality is a consequence of the variational principles. It is the basic math behind the SVD.

This SVD math is contained in the *lemma*: If $x = V_1$ is an optimizer for (15) and if y is orthogonal to x, then Ay is orthogonal to AV_1. One proof of this lemma goes by contradiction: if y is orthogonal to x and $y \neq 0$ and Ay is not orthogonal to AV_1, then V_1 is not optimal. It simplifies the argument a little to replace the optimization problem (15) with the equivalent optimization problem

$$\sigma_1^2 = \max_{x\neq 0} \frac{\|Ax\|_2^2}{\|x\|_2^2} \ . \tag{17}$$

To see the equivalence, suppose x is a maximizer in (17), then $y = \frac{1}{\|x\|_2} x$ is a maximizer in

$$\sigma_1^2 = \max_{\|y\|_2=1} \|Ay\|_2^2 \ .$$

Then you take the square root of both sides to get the original problem (15).

To finish the argument, we take the supposed y perpendicular to V_1 that has Ay not perpendicular to AV_1. We do a calculation to show that if there were such a $y \neq 0$, then $x = V_1$ would not optimize (17). We just take $x = V_1 + ty$ and show that if t is small enough with the correct sign, then the ratio (17) decreases. The calculation has elements that might seem familiar by now. The numerator is

$$\begin{aligned} \|A(V_1 + ty)\|_2^2 &= \left[A(V_1 + ty)\right]^T \left[A(V_1 + ty)\right] \\ &= V_1^T A^T A V_1 + 2t V_1^T A^T A y + O(t^2) \ . \end{aligned}$$

The coefficient of t is the inner product of AV_1 with Ay, so it is not zero if AV_1 is not perpendicular to Ay. The denominator is

$$\begin{aligned}\|V_1 + ty\|_2^2 &= V_1^T V_1 + 2tV_1^T y + O(t^2) \\ &= \|V_1\|_2^2 + O(t^2) \ .\end{aligned}$$

The linear term $2tV_1^T y$ is zero because y is supposed to be perpendicular to V_1.

$$\frac{d}{dt} \left. \frac{\|V_1 + ty\|_2^2}{\|V_1 + ty\|_2^2} \right|_{t=0} = \frac{2V_1^T A^T Ay}{\|V_1\|_2^2} \neq 0 \ .$$

Thus, taking t to be a small positive number or maybe a small negative number will make the ratio larger than its $t = 0$ value, which was supposed to be the the maximum. This shows that there cannot be any such y, if V_1 is the optimizer. Thus, V_2 perpendicular to V_1 and V_1 being optimal implies that U_2 is perpendicular to U_1.

This argument could be done for the latter singular vectors. If $V_1, \cdots, V_k$ are all optimal and have the right orthogonality properties, and if x is perpendicular to all of the V_j, then Ax is perpendicular to all the corresponding U_j. For that reason, U_{k+1} must be orthogonal to $U_1, \cdots, U_k$.

5.3 Fitting A as a data matrix

This approach treats the entries of A as data. For example, A_{jk} could be a measurement of quantity j at time k. In this case, row k of A, which has entries A_{1k}, …, A_{nk} represent the "snapshot" at time k, while column j, which has entries $A_{1,k}$, …, A_{mk}, represents the time series of measurements of quantity j.

A variational problem would be to find a *feature* that best "explains" the columns of A. This feature would be an m component column vector U that can be scaled to fit all the columns of A in the least squares sense. Let A_k be column k of A, which could be thought of as m elements of a time series. Let S_k be a number that "scales" the feature vector U to match A_k as well as it can. The best scaling minimizes the sum of squares difference between A_k and the scaled U, which is

$$\|A_k - S_k U\|_2^2 = \sum_{j=1}^{m} (A_{jk} - S_k U_j)^2 \ . \tag{18}$$

The notation in this expression is not ideal. A_{jk} is entry (j, k) of A, and is entry j of the column vector A_k. S_k is a scaling factor, a number, for column k. The numbers U_j are the m entries of the column vector U. The best scalings S_k may be found by differentiating with respect to S_k and setting the derivative to zero. This gives

$$\sum_{j=1}^{m} U_j \left(A_{jk} - S_k U_j \right) = 0 \implies S_k \; = \; \frac{\displaystyle\sum_{j=1}^{m} U_j A_{jk}}{\displaystyle\sum_{j=1}^{m} U_j^2} \; .$$

This precise form of this formula for S_k is not important, except for two things: One is that S_k is uniquely determined. The other is that we may normalize U so that $\|U\|_2^2 = 1$. If we multiply U by a scale factor a, we can get the same quality of fit by adjusting the scale factors S_k by $\frac{1}{a}$.

The optimal column vector U is the one that minimizes the sum of the fitting errors over all n of the columns. We assume that we use the optimal scaling factor S_k for column k. The U we seek satisfies

$$\min_{\|U\|_2^2=1} \sum_{k=1}^{n} \|A_k - S_k U\|_2^2 \; . \tag{19}$$

A theorem of mathematical analysis implies that there is an optimal U. Too bad, it doesn't say that this optimal U is unique and it does not give an algorithm for finding it.

The quantity being minimized in (19) is

$$\sum_{k=1}^{n} \sum_{j=1}^{m} \left(A_{jk} - S_k U_j \right)^2 \; .$$

The numbers S_k and U_j play a similar role in this expression. You can think of the numbers S_k as the components of an $n-$component column vector, S. Then the numbers $S_j U_k$ are the elements of the matrix $m \times n$ matrix US^T. The *rank* of a matrix is the dimension of the span of the columns of that matrix. The "row rank = column rank" theorem says that this is also the dimension of the space spanned by the rows. The column rank of US^T is one, because all the columns are proportional to U. The row rank is one because all the rows are proportional to S, and because row rank is equal to column rank.

5.4 Tikhonov regularization

6 Summary

- $A = m \times n$ matrix.
  - $x \in \mathbb{R}^n \rightsquigarrow y = Ax \in \mathbb{R}^m$ (a "real" matrix), or $x \in \mathbb{C}^n$, $y \in \mathbb{C}^m$ (a "complex" matrix)
  - $m =$ number of rows, $n =$ number of columns
  - $m = n$ is a square matrix.

- $m < n$ is a "short-fat" matrix. There is $x \neq 0$ with $Ax = 0$.
- $m > n$ is a "tall-thin" matrix. There is y with $Ax \neq y$ for any x.
- entries called a_{jk} or A_{jk}

- L = lower triangular matrix. $L_{jk} = 0$ for $j < k$. Sometimes normalized to have ones on the diagonal: $L_{jj} = 1$. Sometimes not. Often implicitly assumed to be square, not always.

- U = upper triangular matrix, transpose is lower triangular, sometimes normalized to have ones on the diagonal. Often implicitly assumed to be square, not always.

- If A is $m \times n$ then A^T is $n \times m$ with entries

$$A^T_{jk} = A_{kj} \ .$$

The conjugate transpose (or just "conjugate", or "adjoint") takes the complex conjugate of the entries as it transposes

$$A^*_{jk} = \overline{A_{kj}} \ .$$

$A^T = A^*$ if A is a real matrix (has real entries).

- Q = orthogonal matrix: Q us real and $Q^T Q = I$.

- Q = unitary matrix: $Q^* Q = I$. Unitary and orthogonal are the same for real matrices because $Q^* = Q^T$ for real Q.

- A = symmetric matrix: $A = A^T$ (automatically square)

- A = hermitian matrix: $A = A^*$ (automatically square)

- A = positive definite matrix: $x^T A x > 0$ if $x \neq 0$. SPD means symmetric and positive definite. Positive semi-definite allows $x^T A x = 0$ with $x \neq 0$, but not $x^T A x < 0$.

- P = permutation matrix representing a permutation π, which is a re-arrangement of the numbers $1, 2, \ldots, n$ taking j to $\pi(i)$. $Px = y$ means $y_{\pi(j)} = x_j$. For example, if $(1,2,3) \xrightarrow{\pi} (2,3,1)$, which means $\pi(1) = 3$, $\pi(2) = 1$, and $\pi(3) = 2$, then $y = Px$ has $y_{\pi(1)} = x_1$, which is $y_3 = x_1$, etc.

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \ , \quad \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \xrightarrow{P} Px = \begin{pmatrix} x_2 \\ x_3 \\ x_1 \end{pmatrix} \ .$$

- LU factorization of a general square matrix:

$$A = PLU \ \text{ with } \left\{ \begin{array}{rcl} P & = & \text{permutation matrix} \\ L & = & \text{upper triangular, ones on the diagonal} \\ U & = & \text{lower triangular, unconstrained diagonal} \end{array} \right.$$

- Cholesky factorization of a real symmetric positive semi-definite matrix

$$A = LL^T \ , \ \ L \text{ upper triangular, diagonals } L_{jj} \geq 0.$$

- LDL^T factorization of a real symmetric matrix:

$$A = LDL^T \ , \ \ L \text{ is lower triangular ones on the diagonal, } D \text{ is diagonal}$$

- QR factorization of a square matrix

$$A = QR \ , \ \ Q \text{ orthogonal, } R \text{ upper triangular, unconstrained diagonal}$$

- Upper Schur form

$$A = QLQ^* \ , \ \ Q \text{ unitary, } L \text{ upper triangular}$$

  Diagonals L_{jj} are eigenvalues of A, which often are not real even when A is real. If A is real then Q and L are real if A has all real eigenvalues.

- SVD, Singular value decomposition, $m \times n$ matrix A

$$A = U\Sigma V^T \left\{ \begin{array}{lcl} U & = & m \times m \text{ orthogonal, unitary if } A \text{ is complex} \\ \Sigma & = & m \times n \ , \ \ \text{diagonal, } \Sigma_{jk} = 0 \text{ if } j \neq k, \ \Sigma_{jj} \geq 0 \text{ (real even if } A \text{ is complex)} \\ V & = & n \times n \text{ orthogonal, unitary if } A \text{ is complex} \end{array} \right.$$

7 Exercises

1. The *normal equations* are a different way to solve the linear least squares problem (8). The operation count and memory use are less, particularly if the number of fitting variables, n, is much smaller than the number of data values, m. This is because it avoids creating the $m \times m$ matrix Q and instead factors an $n \times n$ matrix. On the other hand, the normal equations algorithm is not as stable as algorithms based on QR or SVD for ill conditioned problems. Also, it does not replace other applications of the QR factorization such as finding an ortho-normal basis of the complementary space.

   (a) A *quadratic form* is a function of $x \in \mathbb{R}^n$ given by

$$q(x) = \frac{1}{2} x^T H x \ . \tag{20}$$

   The gradient of q (or any function of n variables) is the column vector of partial derivatives

$$\nabla q(x) = \begin{pmatrix} \dfrac{\partial q}{\partial x_1} \\ \vdots \\ \dfrac{\partial q}{\partial x_n} \end{pmatrix} \ .$$

Show that if H is symmetric, then

$$\nabla q = Hx \; .$$

Hint. Use the explicit formula

$$x^T H x = \sum_{i=1}^{n} \sum_{j=1}^{n} x_i x_j H_{ij} \; .$$

Be careful to respect the possible repetition of terms in the sum. For example, $x_2 x_3$ occurs both in $x_2 x_3 H_{23}$ and $x_3 x_2 H_{32}$. If H is not symmetric, the formula is

$$\nabla q = \frac{1}{2}\left(H + H^T\right) x \; .$$

It is natural to assume the matrix defining a quadratic form is symmetric because the same quadratic form is given by the *symmetrized* matrix $\frac{1}{2}\left(H + H^T\right)$:

$$x^T H x = x^T \left[\frac{1}{2}\left(H + H^T\right) \right] x \; .$$

(b) Show that minimizing (8) is equivalent to (has the same solution x as) minimizing

$$f(x) = x^T\left(A^T A\right) x - 2b^T A x \; .$$

(c) Show that $\nabla f(x) = A^T A x - A^T b$ so minimizing f is equivalent to solving the *normal equations* involving the $n \times n$ symmetric matrix $H = A^T A$:

$$A^T A x = A^T b \; . \qquad (21)$$

(d) Show that if A has *full rank*, which is n, then $H = A^T A$ is positive definite and has a Cholesky factorization.

8 References

1. G. Strang, Linear Algebra
2. P. Lax, Linear Algebra
3. J. Demmel, Practical Linear Algebra
4. L.N. Trefethen and D. Bau, Numerical Linear Algebra