Chapter 3

Numerical Linear Algebra

This chapter aims to provide a brief introduction to numerical linear algebra. People who are unfamiliar with how to go about (say) solving linear equations, or how to compute eigenvalues and eigenvectors might find this useful for selecting the best routine(s) to solve their particular problem, and to understand the rationale for the way the routines are set up in the way they are.

3.1 What numerical linear algebra is about

There are a number of core operations and tasks that make up numerical linear algebra. At the lowest level these include calculating linear combinations of vectors and inner products, and at the higher level consists of solving linear equations, solving least-squares problems and finding eigenvalues and eigenvectors.

The lower level operations are usually quite straight forward in terms of what they do and what the accuracy of the results are. However, the higher level operations take longer, and more care must be taken with regard to how accurate the answers are. The routines used to perform these higher level operations are more varied and allow a number of different ways of performing the same computation. The difference between them lies often in the speed (or lack of it) and the accuracy of the answers obtained.

There are further complications because of some intrinsic limits to the computations that a computer can do accurately, at least with floating point arithmetic. Floating point arithmetic cannot store numbers to an accuracy (relative to the number stored) better than what is called "*machine epsilon*", or "*unit roundoff*". This quantity is usually denoted by **u**, but is represented in the library by `MACHEPS`. It is also referred to in the ANSI C header file `<float.h>` as `DBL_EPSILON`. For most machines this quantity for double precision is about 2×10^{-16}.

Practically all floating point calculations introduce errors of size of machine epsilon times the size of the quanitities involved; for all intents and purposes, these errors are unavoidable. Perturbations in the *data* of a problem are essentially unavoidable. Algorithms that compute answers that would be exact for slightly perturbed data are called *backward stable*; algorithms which give answers that are close to the exact answer are called *forward stable*. Sometimes the problems that are solved are inherently unstable, or "ill conditioned" (see below). In these circumstances, *no* algorithm can be expected to be *forward* stable. However, well designed algorithms are at least *backward* stable; the answers are exact for slightly perturbed data. The algorithms in Meschach are essentially all backward stable in this sense. Combining these algorithms in programs can sometimes lead to methods that are not stable in this sense. Careful analysis of the algorithm may need to be done to check this. Often the loss of backward stability is because an operation is carried out that hides or disguises the original data of the problem.

3.2 Vector and matrix norms

While it is quite straightforward to talk about the magnitude of a number, it is less so with vectors and matrices as there are a number of different ways of defining it. These "magnitudes" or *norms* must have a number of basic properties in order to be of some use. These properties for vector norms are written out below; the norm itself is written as $\|\ \|$.

$$\begin{gathered} \|x\| \text{ is a non-negative real number} \\ \|x+y\| \le \|x\| + \|y\| \\ \|\alpha x\| = |\alpha|\,\|x\| \qquad \text{where } \alpha \text{ is a real number.} \end{gathered} \tag{3.1}$$

Matrix norms have not only these properties (with x and y replaced with matrices), but an additional one:

$$\|XY\| \le \|X\|\,\|Y\|.$$

Some standard vector norms are

$$\begin{gathered} \|x\|_1 = \sum_i |x_i|, \qquad \|x\|_\infty = \max_i |x_i| \\ \|x\|_2 = \left(\sum_i |x_i|^2 \right)^{1/2}. \end{gathered} \tag{3.2}$$

The last norm ($\|\ \|_2$) is actually the standard or "Euclidean" norm and is the definition of "magnitude" used in geometry and mechanics etc. However, different problems often natural ways of measuring vectors. For example, if e is a vector of errors, then $\|e\|_\infty \le .01$ means that no error is larger than .01.

These vector norms can be computed by the routines `v_norm1()`, `v_norm2()` and `v_norm_inf()`, for the $\|\ \|_1$ norm, the $\|\ \|_2$ norm and the $\|\ \|_\infty$ norm respectively.

Associated with these vector norms are matrix norms that are defined by

$$\|A\| = \max_{x \ne 0} \|Ax\| / \|x\|.$$

The associated matrix norms for the above vector norms are:

$$\begin{gathered} \|A\|_1 = \max_j \sum_i |a_{ij}|, \qquad \|A\|_\infty = \max_i \sum_j |a_{ij}| \\ \|A\|_2 = \big(\text{maximum eigenvalue of } A^T A\big)^{1/2}. \end{gathered} \tag{3.3}$$

Some matrix norms are not associated with any particular vector norm, such as the *Frobenius* norm:

$$\|A\|_F = \left(\sum_{i,j} |a_{ij}|^2 \right)^{1/2}.$$

These matrix norms can be computed by the routines `m_norm1()` for the $\|\ \|_1$ norm, `m_norm_inf()` for the $\|\ \|_\infty$ norm, and `m_norm_frob()` for the Frobenius norm $\|\ \|_F$. The matrix 2-norm has not been implemented as it is a rather expensive operation. The matrix 2-norm is best computed using the SVD, which is discussed later.

3.3 "Ill conditioning" or intrinsically bad problems

Users of numerical routines sometimes find that the results they get are erratic or obviously wrong for some reason or other. Barring programming errors, there are some reasons why this can happen. Often it comes under the heading *ill conditioning*, which means that the problem is inherently difficult.

Whenever the computer does some calculation with real numbers (like 3.1415926 ...) it almost always adds some error to the result whose magnitude is about "machine epsilon" times the magnitude of the result. If such a change in the data can radically change the answer, then the problem or task is called "ill conditioned". This is a property of the problem, not of any algorithm to solve it.

As with most things in numerical analysis, it is a good idea to quantify "how badly conditioned". For the problem of solving linear systems of equations, the measure of conditioning for a particular norm $\| \, \|$ is

$$\kappa(A) = \|A\| \, \|A^{-1}\|$$

which is called the *condition number* of A. The condition numbers for the $\| \, \|_1$, $\| \, \|_2$ or $\| \, \|_\infty$ norms are usually denoted $\kappa_1(A)$, $\kappa_2(A)$ or $\kappa_\infty(A)$ respectively.

A justification of why this is used as a measure of the conditioning of a system of linear equations, is given in the following theorem:

Theorem 3.3.1 *If A is nonsingular and $\|A^{-1}\| \, \|E\| < 1$ and*

$$Ax = b, \quad \textit{and} \quad (A+E)(x+e) = b+f,$$

then

$$\frac{\|e\|}{\|x\|} \leq \frac{\kappa(A)}{1 - \kappa(A)(\|E\|/\|A\|)} \left[\frac{\|E\|}{\|A\|} + \frac{\|f\|}{\|b\|} \right].$$

A proof of this may be found in a number of numerical analysis textbooks such as *Matrix Computations*, by Golub and van Loan, §2.7, pp. 79–80, 2nd Edition, (1989), or in *An Introduction to Numerical Analysis*, by K. Atkinson, Ch. 8, pp. 462–463, 1st Edition, (1979).

Do ill conditioned problems or tasks occur in practice? The answer is "All too often." One family of matrices that are notoriously ill-conditioned are the Hilbert matrices:

$$H_n = \begin{bmatrix} 1 & 1/2 & 1/3 & \dots & 1/n \\ 1/2 & 1/3 & 1/4 & \dots & 1/(n+1) \\ 1/3 & 1/4 & 1/5 & \dots & 1/(n+2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1/n & 1/(n+1) & 1/(n+2) & \dots & 1/(2n-1) \end{bmatrix}.$$

These matrices arise quite naturally in finding best integral-least square error fits for functions in terms of 1, x, x^2, $\dots$, x^{n-1}. The condition number of H_n for $n = 5$ is already $\approx 4.8 \times 10^5$ and for $n = 10$ is $\approx 1.6 \times 10^{13}$. In fact the condition number of H_n for large n increases super-exponentially in n. Because they are so ill-conditioned, they are a favourite family of matrices to test linear equation solvers.

This condition number can be computed in $O(n^3)$ floating point operations essentially by calculating the inverse of the original matrix. Alternatively, it can be *estimated* relatively cheaply (in $O(n^2)$ operations) once the LU factors of the matrix are known. This can be done using the routine `LUcondest()`.

3.4 Least squares and pseudo-inverses

It is quite common, when analysing data, to perform a "least squares fit". For example, if there are three controlled quantities and one measured quantity in an experiment, it is common to fit a linear model:

$$y_i \approx \alpha_1 x_{i,1} + \alpha_2 x_{i,2} + \alpha_3 x_{i,3}$$

where each α_j is a parameter to be fitted, and y_i is the ith measured value, and $x_{i,j}$ is the ith value of the jth controlled quantity.

The "least squares fit" is the α vector that minimises

$$\sum_{i=1}^{m} \left(y_i - (\alpha_1 x_{i,1} + \alpha_2 x_{i,2} + \alpha_3 x_{i,3})\right)^2 .$$

This can be cast in terms of matrices and vectors by setting X to be the matrix of the $x_{i,j}$, and y to be the vector $[y_1, y_2, \ldots, y_m]^T$. Then the approximation is $y \approx X\alpha$, and more specifically, the least squares fit is obtained by minimising $\|y - X\alpha\|_2^2 = (y - X\alpha)^T(y - X\alpha)$. By taking partial derivatives with respect to the α_j's gives the system of linear equations known as the *normal equations*:

$$X^T X\alpha = X^T y.$$

If the columns of X are linearly independent, then the matrix $X^T X$ is positive definite and the Cholesky factorisation can be used to solve this system of equation once $X^T X$ is formed. The following piece of code does this:

```
MAT     *X, *XTX;
VEC     *y, *XTy, *alpha;
  ......
/* set up X and y */
  ......
XTX = mtrm_mlt(X,X,MNULL);
XTy = vm_mlt(X,y,VNULL);
CHfactor(XTX);
alpha = CHsolve(XTX,XTy,VNULL);
```

If the columns of X are *not* linearly independent, then there are redundant variables being set in the experiment: at least one of the variables being set is just a linear combination of the others. In the above piece of code, this may result in an error being raised to the effect that the matrix `XTX` is not positive definite. Whether this happens or not depends on the way that the rounding errors go.

In practice it may well be that some of the set quantities are *nearly*, but not exactly, redundant. The Cholesky factorisation may not be able to pick this up. However, there are other "factorisations" that can. These are the QR factorisation (with column pivoting) and the SVD. The QR factorisation we will return to as another means of solving least squares problems.

3.4.1 Singular Value Decompositions

The SVD or Singular Value Decomposition is analogous in some ways to finding eigenvalues and eigenvectors. The SVD of a matrix X is a decomposition $X = U^T \Sigma V$ where U and V are orthogonal matrices, and Σ is a diagonal matrix. The values on the diagonal of Σ are unique, except for their sign. If the entries of Σ are all nonnegative and ordered so that they are nonincreasing going down the diagonal, then the diagonal entries are called *singular values*, and are denoted by σ_i. The columns of U and V are called *singular vectors*.

How well or ill conditioned a least squares problem is can be determined directly from the singular values. The usual condition number for least square problems is $\kappa_{LS}(X) = \sigma_1/\sigma_n$ where X is $m \times n$ and $m \geq n$. If $\sigma_n = 0$ then X has linearly dependent columns, and the problem cannot be solved to any degree of accuracy. Such a matrix is also referred to as being *rank deficient*.

3.4.2 Pseudo-inverses

Whether a matrix is square or rectangular, rank deficient or has full rank, it always has a *pseudo-inverse*. This is the matrix $X^+ = V^T \Sigma^+ U$ where the ith diagonal of Σ^+ is $1/\sigma_i$ is $\sigma_i \neq 0$ and zero otherwise. This

has a number of useful properties such as the Moore–Penrose properties:

$$\begin{aligned} XX^+X &= X, & (XX^+)^T &= XX^+ \\ X^+XX^+ &= X^+, & (X^+X)^T &= X^+X. \end{aligned} \tag{3.4}$$

This means that XX^+ is an orthogonal projection onto range(X) and X^+X is an orthogonal projection onto range(X^T).

The least squares problem can, in general, be solved by setting $\alpha = X^+y$. This solution is, in fact, the *smallest* α that minimises the sum of errors squared. This approach appears quite simple for providing a way of solving least squares problems (and others) involving rank deficient matrices. However, there are a number of practical difficulties. The first of these is that small perturbations to rank deficient matrices usually result in full rank matrices; the σ_i's that were formerly zero before the perturbation, become nonzero, but small after the perturbation. This means that where Σ^+ had a zero on the diagonal before the perturbation, after the perturbation it has $1/\sigma_i$ which is quite large. In short, the pseudo-inverse is not a continuous function of the matrix entries; small perturbations can give very large changes in the results.

While the SVD can be computed numerically, roundoff error will ensure that almost always the computed σ_i's are all nonzero. In these cases it is important to *estimate* the rank by considering the size of the σ_i's. For such problems an error tolerance is needed to decide how small the σ_i's need to be before they are considered "too small". The choice of such an error tolerance should be based on the size of the errors in the matrix, and their source. If, for example, the values in the X matrix have a measurement error of about 10^{-3}, then a tolerance of about 10 times this should detect near rank deficient matrices. If, on the other hand, the only errors are those from roundoff error, then a value of 100 times unit roundoff (`MACHEPS` in the library) should be adequate.

3.4.3 QR factorisations and least squares

An alternative approach to solving least squares problems for full rank matrices (i.e. those that are not rank deficient) is to use the QR factorisation. This method is also described in section 3 of the tutorial chapter. The QR factorisation of a matrix A is a factorisation $A = QR$ where Q is orthogonal and R is upper triangular.

This QR factorisation is computed by means of *Householder matrices*. These are discussed in more detail in the manual entry for the routines that implements these operations, `hhvec()`, `hhtrvec()`, `hhtrcols()` and `hhtrrows()`. The QR factorisation can also be computed by using *Givens' rotations* which are discussed in the manual entries for `givens()`, `rot_vec()`, `rot_cols()` and `rot_rows()`.

To use this factorisation to solve a linear least squares problem $X\alpha \approx y$ we compute, first, the QR factorisation of $X = QR$. For X $m \times n$ and $m > n$, as the R matrix is upper triangular,

$$R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}.$$

If X has full rank, then R_1 is a nonsingular $n \times n$ matrix. The matrix Q should be split in a consistent way: $Q = [Q_1, Q_2]$.

The residual vector's norm is then

$$\|X\alpha - y\|_2 = \|R\alpha - Q^Tb\|_2 = \left\| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \alpha - \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} y \right\|_2.$$

This means that

$$\|X\alpha - y\|_2^2 = \|R_1\alpha - Q_1^Ty\|_2^2 + \|Q_2^Ty\|_2^2.$$

The minimum 2-norm of $X\alpha - y$ is obtained by solving

$$R_1\alpha = Q_1^Ty$$

and has the value $\|Q_2^T y\|_2$. The code in section 3 of the chapter 1 provides a complete program for solving least squares problems of this sort.

There are some advantages of this method over the "normal equations" approach, of which the main one is accuracy. In the normal equations approach, the system $X^T X\alpha = X^T y$ is solved for α. The error in the computed α in the 2-norm is of the order of $\mathbf{u}\kappa_2(X^T X)$. On the other hand, the error in the computed α for the QR factorisation method is of the order of $\mathbf{u}\kappa_{LS}(X)$. Now if $X = U^T \Sigma V$ is the SVD of X, then

$$X^T X = V^T \Sigma^T \Sigma V = V^T \operatorname{diag}(\sigma_1^2, \ldots, \sigma_n^2) V$$

and the eigenvalues of $X^T X$ are the squares of the singular values of X. So

$$\kappa_2(X^T X) = \|X^T X\|_2 \|(X^T X)^{-1}\|_2 = \sigma_1^2/\sigma_n^2 = \kappa_{LS}(X)^2$$

and forming $X^T X$ effectively squares the condition number of the problem. This is particularly important for badly conditioned problems with $\kappa_{LS}(X) \approx 1/\sqrt{\mathbf{u}}$; for such problems the QR factorisation method would work, but the normal equations approach would fail.

3.5 Eigenvalues and eigenvectors

There are two main classes of problems and algorithms for computing eigenvalues and eigenvectors. They are problems involving symmetric matrices, and problems involving nonsymmetric matrices. The case of symmetric matrices is easier both in theory and practice. It is also less vulnerable to the effects of roundoff error.

Symmetric matrices all have real eigenvalues, and the corresponding eigenvectors are both real and orthogonal. Thus for any symmetric matrix A there is an orthogonal matrix Q such that $Q^T A Q = \Lambda$ where Λ is the diagonal matrix of eigenvalues. If the ith diagonal element of Λ is λ_i, and q_i is the ith column of Q, then $Aq_i = \lambda_i q_i$. Regarding stability of the eigenvalues to perturbations of the matrix A, the ith eigenvalue of $A + E$, denoted $\tilde{\lambda}_i$, satisfies $\lambda_i - \|E\|_2 \leq \tilde{\lambda}_i \leq \lambda_i + \|E\|_2$.

The eigenvectors are not so stable with respect to perturbations of A, especially if eigenvalues are close together. The extreme case is where there is a repeated eigenvalue, in which case the eigenvalues are not essentially unique (up to a scale factor). Instead, there is a two or three or higher dimensional subspace of eigenvectors. If all the eigenvalues are distinct, then for a matrix $A + E$, $\|E\|_2$ "small", the perturbation in the eigenvector q_i is of size roughly bounded by

$$\|E\|_2 \sqrt{\sum_{k \neq i} \frac{1}{(\lambda_k - \lambda_i)^2}}.$$

As for previous problems, the perturbations in A due to roundoff error is roughly $\|E\|_2 \approx \mathbf{u}\|A\|_2$. This means that the eigenvectors would not usually be reliably computed if its eigenvalue is no more than about $\mathbf{u}\|A\|_2$ from other eigenvalues.

The eigenvalues for a symmetric matrix can be computed using the `symmeig()` library routine, which will compute the Q matrix of eigenvectors as well as a vector containing the eigenvalues if desired.

For the nonsymmetric case, a rather different strategy has to be adopted for several reasons:

1. The matrix A may not be diagonalisable; the Jordan canonical form is not numerically stable.
2. The matrix of eigenvectors may not be well conditioned.
3. The eigenvalues may not be real.

The standard strategy used is to compute the *real Schur decomposition*. This is a variant of the complex Schur decomposition which is a factorisation

$$\overline{Q}^T A Q = T$$

where T is upper triangular, and Q is unitary; that is, $\overline{Q}^T Q = I$ where $\overline{Q}$ is the complex conjugate of the matrix Q. The diagonal entries of T are the eigenvalues of A.

For the real case,

$$Q^T A Q = T$$

where T is *block upper triangular* with 1×1 and 2×2 blocks on the diagonal and Q is orthogonal. The eigenvalues of the 1×1 and 2×2 diagonal blocks of T are the eigenvalues of A. This real Schur decomposition is computed by the `schur()` routine. If you wish to obtain the actual eigenvalues and eigenvectors, there are the auxiliary routines `schur_vals()` and `schur_vecs()`. The `schur_vals()` routine computes the (complex) eigenvalues and returns the real and imaginary parts of the eigenvalues. The `schur_vecs()` routine computes the eigenvectors of a matrix by means of its real Schur decomposition, by using one cycle of inverse iteration for each eigenvector. That is, the system

$$(T - \lambda I)\hat{x} = r$$

is solved for $\hat{x}$ where r is a random real vector.

Unfortunately, if there are repeated eigenvalues, this method cannot be expected to give good results: the matrix of eigenvectors would be ill-conditioned. Indeed, it is usually not possible to get a nonsingular matrix of eigenvectors if there are repeated eigenvalues. Consider the general 2×2 matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

This matrix has repeated eigenvalues if and only if $(a-d)^2 = -4bc$. The repeated eigenvalue is $(a+d)/2$. If X is the matrix of eigenvectors, and is nonsingular,then

$$X^{-1} \begin{bmatrix} a & b \\ c & d \end{bmatrix} X = (a+d)/2\, I$$

which implies that

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = (a+d)/2\, I$$

and $a = d$ and $b = c = 0$. Clearly, small perturbations of matrices with repeated eigenvalues usually result in matrices which do not have a nonsingular matrix of eigenvectors.

The proper way to handle the situation of repeated eigenvalues is either to use the Schur decomposition (real or complex), or to use the *Jordan Normal form*. The Jordan Normal form of the matrix A has the form

$$X^{-1}AX = \begin{bmatrix} J_1 & 0 & 0 & \dots & 0 \\ 0 & J_2 & 0 & \dots & 0 \\ 0 & 0 & J_3 & \dots & 0 \\ 0 & 0 & 0 & \dots & J_l \end{bmatrix}$$

where each J_i (called a *Jordan block*) has the form

$$J_i = \begin{bmatrix} \lambda_i & 1 & 0 & \dots & 0 \\ 0 & \lambda_i & 1 & \dots & 0 \\ 0 & 0 & \lambda_i & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & & \lambda_i \end{bmatrix}.$$

Note that J_i may be as small as 1×1 or 2×2.

This form is not favoured by numerical analysts as it is difficult to compute when roundoff errors are present, and the criterion for deciding how big a Jordan block should be is a difficult task as it requires numerically estimating the rank of a number of matrices. Golub and van Loan's *Matrix Computations* discusses the difficulties of computing the Jordan Normal form pp. 390–392 (2nd Edition, 1989). Also, the Schur form can be used for almost all the same purposes as the Jordan Normal form, such as computing matrix exponentials.

3.6 Sparse matrix operations

Sparse matrices are simply matrices where most of the entries are zero. These are important as they can be stored in a more compact way by storing only the nonzero entries and where they lie within the matrix. The zero entries can usually be ignored for most computations. Thus far larger problems can be dealt with, and more quickly, than if array storage is used.

While the previous discussion holds for all matrices whether sparse or not, if sparse matrices are to be used effectively then their sparsity needs to be preserved. This quickly rules out a lot of algorithms which work better for matrices that are not sparse (i.e. *dense*). For example, the Schur decomposition and explicit matrix inverses usually result in intermediate and result matrices where most of the entries are nonzero.

Sparse matrices have a *structure* that dense matrices don't. This is essentially the set of (i, j) entries of a matrix that are nonzero, or at least that have memory allocated for a value. And it is often important to keep this structure and to prevent the number of nonzeros in intermediate matrices from increasing too quickly. The introduction of nonzero entries into sparse matrices is called *fill-in*. Not only does fill-in result in more space required to store the intermediate matrices and result indirectly in more floating point computations, but it also requires some sort of dynamic memory management. (This is easier in 'C' than in Fortran, but still has a cost in both time and memory space.) The routines provided for manipulating sparse matrix data structures hides much of the complexity of the data structures and operations that need to be performed when there is fill-in.

Sparse matrices are also important as they often lend themselves better to iterative rather than the direct methods that have been discussed so far. Often some mix of iterative and direct methods will provide the best performance for solving some large problems.

The direct routines implemented for sparse matrices include sparse Cholesky and sparse LU factorisation, with a number of variants which are provided for control the "structure" of the sparse factorisations. The iterative methods for solving systems of linear equations include pre-conditioned conjugate gradients for solving symmetric, positive definite systems, the CGS method of Sonneveld for solving systems of non-symmetric matrices, and the LSQR method of Paige and Saunders for non-square least squares problems. For eigenvalues, the Lanczos method is provided for symmetric matrices, and the Arnoldi method for nonsymmetric matrices.

Those who are familiar with the standard "classical" iterative methods (Gauss–Jacobi, Gauss–Seidel and Successive Over-Relaxation etc.) may be disappointed that they are not implemented. There are three reasons for this. The first is that the iterative routines that have been implemented do not require an explicit representation of the matrix; all that is needed is a way of forming Ax for any vector x. That is, only a *functional representation* of the matrix (A) is needed. The second is the difficulty in obtaining good convergence with the classical methods. These classical methods require good estimates of convergence rates and the like, and are difficult to turn into general purpose routines when the "rate estimation code" is included. The third is that conjugate gradients (without pre-conditioning) give the same order of convergence as that for SOR with the optimum over-relaxation parameter for standard test problems. It therefore appears that there is not a great deal of reason to implement SOR over conjugate gradient methods, although conjugate gradient methods can be modified to use an SOR-based pre-conditioner M:

$$M = (D + \omega L)D^{-1}(D + \omega L)^T$$

where D is the diagonal part of A, and L is the strictly lower triangular part of A and ω is the (over)relaxation parameter. Solving $Mz = w$ for z can be done essentially by backward and forward substitution and can be easily programmed without explicitly forming M. The Gauss–Seidel pre-conditioner is obtained by setting $\omega = 1$.

The crucial point about iterative methods is that there is usually no natural limit to the number of iterations. A relative precision for the residual must usually be specified, and it needs to be significantly larger than $\mathbf{u}$ (or, as it is represented in the library `MACHEPS`). The number of iterations is also important for the speed with which a system of linear equations is solved. If the relative error tolerance is set to ϵ, then the number of iterations is roughly proportional to $\sqrt{\kappa_2(A)}\ln(1/\epsilon)$ for conjugate gradient methods. For LSQR, it is roughly proportional to $\kappa_{LS}(A)\ln(1/\epsilon)$. For finding eigenvalues of symmetric

matrices, the Lanczos routine finds the bottom eigenvalue to an accuracy of ϵ in time roughly proportional to $\sqrt{(\lambda_k - \lambda_2)/(\lambda_2 - \lambda_1)} \ln(n(\lambda_k - \lambda_1)/\epsilon)$ where $\lambda_1 < \lambda_2 < \cdots < \lambda_k$ are the distinct eigenvalues of A. (i.e. λ_k is the largest eigenvalue of A.)

The use of functional representation also opens up the possibility of pre-conditioning for the CGS and LSQR, and even the Lanczos methods. Here incomplete factorisations may be able to improve performance, such as the incomplete Cholesky factorisation or the incomplete/modified LU factorisation.

Contents