RESEARCH
COUNCILS UK

nag®

CSC / NAG Autumn School on

Core Algorithms in High-Performance Scientific Computing

Maths II

Dwight Barkley

Linear Equations I

Maths II: Linear Equations

Probably the most important problem in numerical linear algebra is:

$$Ax = b$$

where

- A is a given $n \times n$ matrix
- b is a known n-vector
- x is the unknown n-vector

We shall interchangeably use the terms: linear system of equations, linear equations, and linear system.

When A is square with full rank, its inverse A^{-1} exists and there is a unique solution x given by

$$x = A^{-1}b$$

The problem is formally solved.

Even though it would be natural at this point to discuss rank and related topics, a proper discussion would be too distracting here. This is the subject of Math VI where we consider rank deficient (singular), nearly singular, and rectangular matrices. At present, we assume we have a linear system that can in fact be solved.

Even in this case where a unique solution x exists, important issues to consider are:

- Is the method for computing x stable?
- Does one need solutions for several different b all with the same A?
- Does A have structure that can be exploited?
- Does one need A^{-1} explicitly, or just the solution x?
- How can we determine or estimate the accuracy of x?
- Is A nearly singular? If it is, then x is very sensitive to b and the solution $x = A^{-1}b$ may be inaccurate even for a stable method. We consider this in the last lecture.

In this lecture we consider **Direct Methods**. Such methods produce x at the end of a sequence of operations determined only by the problem dimension n (and possibly the structure, e.g. bandwidth, of A) but not by the particular values in A or b. The distinction between these methods and iterative methods, which are more favorable for large sparse systems, will be clear in later lectures after both approaches have been discussed.

2.1 An Example

Consider the following real example of how large systems of equations arise in practice. The inhomogeneous Helmholtz equation is:

$$\left(\nabla^2 + \lambda\right) u = \rho$$

where $u(x, y)$ is the unknown and $\rho(x, y)$ is a given source. The Laplacian ∇^2 in two dimensions is:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

A domain and some boundary conditions must be specified. This equation arises in numerous contexts. When considering waves in hyperbolic PDEs, $\lambda = -k^2$, and k is proportional to the wave frequency. When considering implicit time-stepping schemes for parabolic PDEs, λ will be related to $-\triangle t$ where $\triangle t$ is the time step. For $\lambda = 0$, the equation becomes the Poisson equation.

We saw before how differentiating a function can become a matrix-vector multiplication when a continuous problem is discretized. This case is similar. Skipping most of the details, we consider u on a regular 2D grid with $N \times N$ points with spacing h and use the approximation

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) u(x_j, y_k) \simeq \frac{1}{h^2}\left(u(x_{j-1}, y_k) + u(x_j, y_{k-1}) - 4u(x_j, y_k) + u(x_j, y_{k+1}) + u(x_{j+1}, y_k)\right)$$

We now let $\mathbf{U}$ be the vector representing the N^2 (unknown) values of u on the grid and let $\mathbf{R}$ be the vector of N^2 known values of ρ on the grid. The relationship between the vector indices and the grid indices are given by:

$$U_i = u(x_j, y_k) \quad R_i = \rho(x_j, y_k), \quad \text{where } i = j + Nk$$

Putting this all together (but ignoring here what happens at the boundaries), the discretized Helmholtz equation becomes:

$$\left(\mathbf{D_2} + \lambda \mathbf{I}\right) \mathbf{U} = \mathbf{R}$$

where $\mathbf{D_2}$ is a matrix which approximates ∇^2 according to the above formula. (The 2 in $\mathbf{D_2}$ is for second derivative, not 2 space dimensions.)

Hence, once discretized, the Helmholtz problem becomes a system of linear equations: a known matrix multiplies the unknown solution resulting in a specified right-hand-side:

$$\underbrace{\left(\mathbf{D_2} + \lambda \mathbf{I}\right)}_{A} \underbrace{\mathbf{U}}_{x} = \underbrace{\mathbf{R}}_{b}$$

It is important to appreciate the matrix $A = \mathbf{D_2} + \lambda \mathbf{I}$ that has arisen here:

- For an $N \times N$ grid in two-dimensions, the vectors $\mathbf{U}$ and $\mathbf{R}$ are length N^2 and the matrix A is $N^2 \times N^2$. A small 100×100 grid gives a matrix that is $10^4 \times 10^4$. You can see just how quickly the size of A grows with the grid. Think about in three dimensions.
- With the approximation we use for the Laplacian, the matrix A is banded (ignoring the boundaries). There are non-zero elements only along the diagonal, the first sub- and super-diagonals, and the N^{th} sub- and super-diagonals. The means in particular that the total bandwidth of A is only $2N + 1$. For large N, the relative size of $2N + 1$ compared to N^2 becomes very significant in practice.

This example is very representative of what happens in many real situations in which continuous problems are discretized. While we outlined a particular discretization here, any discretization of the Helmholtz equation would have resulted in a linear system since the Helmholtz equation is linear and the operator acts on the unknown to give the specified right-hand-side.

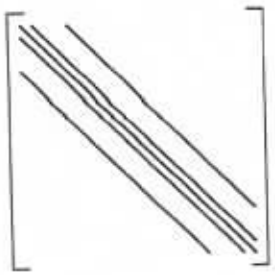

Figure 2.1: Banded matrix arising from discretization of the Helmholtz equation.

2.2 Gaussian-Elimination and Back Substitution

1. Perform a sequence of row operations that renders A upper-triangular (**elimination**)

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$
$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b'_2 \\ b'_3 \end{pmatrix} \tag{2.1}$$
$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b'_2 \\ b''_3 \end{pmatrix}$$

2. Solve for x starting with the last component x_n and working backward (**back-substitution**).

$$x_3 = b''_3/a''_{33}, \quad x_2 = (b'_2 - a_{23}x_3)/a_{22}, \quad x_1 = \dots$$

Any direct method for solving $Ax = b$ is equivalent to this in exact arithmetic. Some methods may be more efficient or more stable, but they cannot be fundamentally different.

2.3 LU decomposition or factorization

In practice Gaussian elimination as just presented is never used in this form. For the same computational cost, one can factor A so that the right-hand-side b does not have to be known in advance. The result is that x can be found cheaply for many different b.

Basics

We will be manipulating rows, and potentially columns, of the linear equations and the following will be helpful to keep in mind:

- Interchanging or taking linear combinations of **rows** in a linear system of equations does not change the order of the components of x.
- Interchanging or taking linear combinations of **columns** in a linear system of equations does change the order of the components of x, but not the order of the components of b.

Elimination by matrix multiply

Each step of the elimination process can be viewed as a multiplication by a unit lower triangular matrix (a lower triangular matrix with 1's on the diagonal). In fact, the unit lower triangular matrix has only one column of non-zero entries below the diagonal. For example, the elimination of the first column in the simple 3×3 example can be written:

$$A' = L_1 A$$

where

$$L_1 = \begin{pmatrix} 1 & & \\ \times & 1 & \\ \times & & 1 \end{pmatrix}$$

$\times$ indicates non-zero values.

Likewise for the second step:

$$A'' = L_2 A' = L_2 L_1 A$$

where

$$L_2 = \begin{pmatrix} 1 & & \\ & 1 & \\ & \times & 1 \end{pmatrix}$$

Noting that A'' is upper triangular, we denote it by U and thus have:

$$L_2 L_1 A = A'' = U$$

Similarly, for the general $n \times n$ case, Gaussian elimination can be expressed

$$L_{n-1} L_{n-2} \cdots L_1 A = U \tag{2.2}$$

where U is upper triangular and each L_j is unit lower triangular with only a single column of non-zero entries off the diagonal. It is possible to invert each of the L_j matrices and to move them to the other side of (2.2). The result is then

$$A = L_1^{-1} \cdots L_{n-2}^{-1} L_{n-1}^{-1} U$$

Each inverse L_j^{-1} has the same structure as L_j, unit lower triangular with only a single column of non-zero entries off the diagonal. Moreover, the L_j^{-1} are easily found from the L_j. The product $L_1^{-1} \cdots L_{n-2}^{-1} L_{n-1}^{-1}$ is unit lower triangular and is obtained by summing the off-diagonal entries. Since $L_1^{-1} \cdots L_{n-2}^{-1} L_{n-1}^{-1}$ is a unit lower triangular matrix, call it L.

Thus we have the LU decomposition of A:

$$A = LU \quad \text{with} \begin{cases} U & \text{upper triangular} \\ L & \text{unit lower triangular} \end{cases}$$

Given the LU decomposition, solving $Ax = b$ is easy. First solve

$$Ly = b \tag{2.3}$$

for y. Since L is unit lower triangular, y is found by forward substitution. Once y is known, x is found by back substitution in the following equation

$$Ux = y \tag{2.4}$$

(To verify that this procedure is algebraically correct, multiply $Ux = y$ by L, giving $LUx = Ly$, but $LU = A$ and $Ly = b$ so $Ax = b$ as required.)

This forward substitution/back substitution phase is commonly referred to simply as back substitution.

Complexity

The computational cost of the LU decomposition for a general matrix is

$$C(n) \sim \frac{2}{3}n^3$$

while the cost of each forward/back substitution is

$$C(n) \sim n^2$$

In short, the work required for solving a general $n \times n$ system of equations is $O(n^3)$. This work is dominated by the LU decomposition
back substitution is very inexpensive by comparison.

For memory, we note that the LU decomposition for a general matrix requires exactly the same memory as the original matrix A. L is unit lower triangular, and so the diagonal elements need not be stored. In fact, the LU decomposition is frequently done "in place" and the matrix A is overwritten, for example by $A \rightarrow L - I + U$.

2.4 Pivoting

The basic LU decomposition just presented is never used in practice since it is numerically unstable (technically it is not backward stable). It is easy to understand the essence of the problem with a simple 2×2 example. Let

$$A = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix}$$

Then the exact LU decomposition is:

$$L = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix} \qquad U = \begin{pmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{pmatrix}$$

In any reasonable precision, $1 - 10^{20}$ would be rounded to -10^{20}. Multiplying L with the approximate U gives:

$$\tilde{A} = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 0 \end{pmatrix}$$

One can easily see that the difference

$$A - \tilde{A} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

is comparable in size to A itself. There is nothing perverse about the matrix itself in that the condition number (defined later) is small and solving $Ax = b$ accurately should present no difficulties.

Essentially the problem is that we took a matrix of roughly unit size and decomposed it into two matrices of very large size. For those matrices to multiply back to the original, some nearly exact cancellation must occur. If L or U gets perturbed a little, this cancellation will not occur correctly and our decomposition will be faulty. Fortunately, there is a solution to the difficulty.

As you can guess, the problem is the element 10^{-20} in the upper left corner of A. The problem is not that it is small relative to the other components, but rather that it is located on a diagonal. (Try permuting the rows of A and performing the LU decomposition.)

Elements along the diagonal, as they are encountered in the elimination process (first a_{11}, then a'_{22} in the example in (2.1)), are known as **pivots**.

The solution to the stability problem is to use **pivoting**: **partial pivoting** (re-ordering rows) or **full pivoting** (re-ordering rows and columns) to bring the available element with largest magnitude to the pivot location before a column elimination. Partial pivoting is standard practice and we will focus only on this case.

Let P be a permutation matrix: P has exactly one 1 in each row and each column. Two examples are:

$$P = \begin{pmatrix} & 1 & \\ 1 & & \\ & & 1 \end{pmatrix} \quad \text{and} \quad P' = \begin{pmatrix} & 1 & \\ & & 1 \\ 1 & & \end{pmatrix}$$

One can easily check that left multiplying a matrix A by P or P' reorders the rows of A:

$$PA = \begin{pmatrix} a_{21} & a_{22} & a_{23} \\ a_{11} & a_{12} & a_{13} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad \text{and} \quad P'A = \begin{pmatrix} a_{31} & a_{32} & a_{33} \\ a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

You can check for fun that right multiplying by a permutation matrix Q, i.e. computing AQ, will re-order the columns of A. Note that if P is diagonal except for two rows and columns, then P will permute just a pair of rows and $PP = I$. We shall call these elementary permutations.

In terms of the previous manipulations giving the LU decomposition, before each elimination we first permute a pair of rows.

For the 3×3 example we started with, rather than use a_{11} as the first pivot, we first search the first column for the largest $|a_{i1}|$. Suppose this were a_{21}. Then we apply an elementary permutation to interchange rows 1 and 2 and then eliminate the first row:

$$A' = L_1 P_1 A$$

where L_1 and A' might be different from before due to the permutation P_1. We have:

$$L_1 = \begin{pmatrix} 1 & & \\ \times & 1 & \\ \times & & 1 \end{pmatrix}$$

where now (since we pivoted first) the non-zero values $\times$ are all less than one in magnitude.

Likewise for the second step, rather than using a'_{22} as a pivot, we first look down the column (diagonal and below) and, if necessary, permute rows and then eliminate:

$$A'' = L_2 P_2 A' = L_2 P_2 L_1 P_1 A$$

As with L_1, L_2 will have no values larger than 1 in magnitude.

You get the idea. For the general $n \times n$ case, Gaussian elimination can be expressed

$$L_{n-1} P_{n-1} L_{n-2} P_{n-2} \dots L_1 P_1 A = U$$

where U is upper triangular.

The next step requires consideration of the structure of the L and P matrices and we shall not justify it here, but it is possible to move all P matrices to the right and L matrices to the left. The L matrices get changed by this process, but their structure does not. We obtain:

$$L'_{n-1} L'_{n-2} \dots L'_1 P_{n-1} P_{n-2} \dots P_1 A = U$$

Each L'_k has unit diagonal and exactly one non-zero column below the diagonal, and no values greater than 1 in magnitude. We can write this as

$$A = PLU$$

where

$$P = (P_{n-1})^{-1} \dots (P_1)^{-1} \qquad L = (L'_1)^{-1} \dots (L'_{n-2})^{-1}(L'_{n-1})^{-1}$$

Finally we arrive at the LU decomposition with partial pivoting (LUPP):

$$A = PLU \quad \text{with} \quad \begin{cases} P & \text{permutation} \\ U & \text{upper triangular} \\ L & \text{unit lower triangular, no element greater than 1 in magnitude} \end{cases}$$

Thus one has an LU decomposition not of A but of $P^{-1}A$. P^{-1} is also a permutation matrix simply related to P. (Some authors write the LU decomposition as $PA = LU$.) The result of considering successive row permutations is that now L is "small": it has no elements greater than 1 in magnitude.

Theory vs Practice

We have presented LU decomposition with partial pivoting, LUPP, in the standard form found in numerical analysis texts. This form is useful for understanding the decomposition and why row interchanges solve the stability problem.
In practice, one uses a good performance library for the LU decomposition and subsequent back substitution. A library might, or might not, actually do the memory copies implied by pivoting. A user seldom needs to be concerned with the details, but should be aware of this important issue.
We have written the combined effect of all row permutations as an $n \times n$ matrix P. In practice the information about the row permutation is always required but will not be stored in such an inefficient form. Generally a single integer array is used to encode the row permutations.

LUPP produces a lower triangular matrix L which is "small", however it is still not guaranteed that U does not somehow become large. There are examples, fortunately artificial, in which U does become exceedingly large for large n. In practice such matrices do not occur.

In general, the LU decomposition of a sparse matrix is dense. This is known as **fill in**. This is a primary reason why direct methods are not well suited for sparse problems. A kind of exception is banded matrices which maintain banded structure under LU decomposition, although with increased bandwidth. (See below).

2.5 QR decomposition

QR decomposition will be discussed in later lectures. Here we note that it too can be used to solve linear equations. The QR decomposition of A is:

$$A = QR \quad \text{with} \quad \begin{cases} Q & \text{unitary (orthogonal if real)} \\ R & \text{upper triangular} \end{cases}$$

Since Q is unitary, $Q^* = Q^{-1}$ so $Ax = b \rightarrow QRx = b \rightarrow Rx = Q^*b$.

Thus, given the QR decomposition, $Ax = b$ is solved by:

$$\begin{aligned} \text{compute} &: \quad y = Q^* b \\ \text{solve} &: \quad Rx = y \end{aligned}$$

This method is generally not used for square, non-singular matrices since it is about a factor of 2 slower than, and offers no advantages to, LU.

2.6 A Few Extensions and Special Cases

LU decomposition for banded matrices

We return to our example from the beginning of the lecture, where we obtained a banded matrix

$$A = (\mathbf{D_2} + \lambda \mathbf{I})$$

Banded matrices such as this arise frequently in practice and it is worth noting two points about LU decomposition for banded matrices.

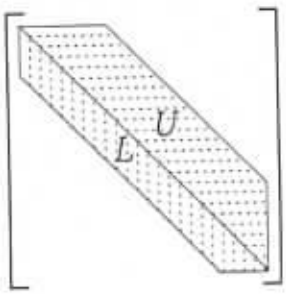

Figure 2.2: LU decomposition of the banded matrix from Fig. 2.1. Note the increase in bandwidth and the fill in within the bands.

The first is that the LU decomposition of a banded matrix is itself banded. If A has lower bandwidth kl and upper bandwidth ku, then the LU decomposition of A with partial pivoting leads to a matrix L with bandwidth kl and a matrix U with bandwidth $ku + kl$. We can trace the increase in bandwidth to the row permutations.

There is a corresponding reduction in the work from $O(n^3)$ to $O(k^2 n)$ where k is the total bandwidth. If $k \ll n$, then this is a huge reduction on work and memory over the full $n \times n$ case. Backsolves also reduce in complexity to $O(kn)$.

The second point is that, in this example, the matrix is sparse between the bands and this sparsity is lost in the LU decomposition.

Symmetric and Hermitian positive definite matrices

A matrix A is symmetric positive definite (real case) or Hermitian positive definite (complex case) if $\langle x^*, Ax \rangle > 0$ for all vectors $x \neq 0$. (See table from Maths I lectures.) While this may seem like a rather specialized case, such matrices arise commonly in practice, in large part because many differential operators have the continuous analog of this property.

All symmetric or Hermitian positive definite matrices have a **Cholesky factorization** of the form:

$$A = R^T R \quad \text{Real} \qquad A = R^* R \quad \text{Complex}$$

where in both cases R is upper triangular with positive diagonal entries.

The memory requirements and computational cost for Cholesky factorization are each essentially 1/2 those for a general matrix A of the same dimension. This is not surprising due to the symmetry of A. The cost of

Cholesky factorization is

$$C(n) \sim \frac{1}{3}n^3$$

The requirement that the matrix be positive definite is essential in that this guarantees that diagonal entries will be positive - a property explicitly required by the factorization as square roots are taken. Pivoting is not required nor used in practice.

From the Cholesky factorization, a system of equations $Ax = b$ is easily solved as before.

Note that the Cholesky factorization is sometimes written as

$$A = LL^* \qquad \text{or} \qquad A = LDL^*$$

where in the second case, D is diagonal and L is unit lower triangular.

Block LU and Cholesky

Consider a matrix M decomposed into blocks as follows:

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

Then the block LU decomposition of M is

$$M = LU = \begin{pmatrix} I & \\ CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & B \\ & Q \end{pmatrix}$$

where $Q = D - CA^{-1}B$ is the **Schur complement** of A in M.

Similarly there is a block Cholesky decomposition:

$$M = R^*R = \begin{pmatrix} A^{1/2} & \\ CA^{-1/2} & A^{1/2} \end{pmatrix} \begin{pmatrix} A^{1/2} & A^{-1/2}B \\ & Q^{1/2} \end{pmatrix}$$

2.7 Two Basic Approaches to Further Efficiency

For general dense A there is little more that can be done apart from finding the best available numerical library to perform the LU decomposition. However, often the matrices that arise come with some sort of structure and one can try to exploit this. The two main approaches (which may in fact reduce to the same) are:

- Minimize the bandwidth of the matrix whose LU decomposition is required.
- Exploit structure of elements of A, using recursion, as in the FFT.

Bandwidth reduction

One case that often arises is that a matrix is banded except for a relatively few entries (a few bad columns for example). There is a trick for solving linear equations with such matrices using banded LU.

The Sherman-Morrison-Woodbury formula is a powerful formula relating the inverse of a matrix $\hat{A}$ to the inverse of a different matrix A,

$$\hat{A} = A + VW^T,$$

where $\hat{A}$ and A are $n \times n$ and where V and W are $n \times p$ matrices. To be useful, A should be banded and $p \ll n$. Examples will be given below.

The inverses of $\hat{A}$ and A are related by:

$$\hat{A}^{-1} = A^{-1} - \left[A^{-1}V\right]\left[I + W^T A^{-1} V\right]^{-1}\left[W^T A^{-1}\right]$$

In the case where $p = 1$, V and W are simply vectors and this reduces to:

$$\hat{A} = A + vw^T$$

$$\hat{A}^{-1} = A^{-1} - \frac{A^{-1}vw^T A^{-1}}{1+\lambda} \qquad \lambda = w^T A^{-1} v$$

Remarks

- While stated as a formula for $\hat{A}^{-1}$, it is not typically used to computed the matrix $\hat{A}^{-1}$, but rather to solve $\hat{A}x = b$ given a good/fast way to solve $Ax = b$.
- **The formula is exact**. It does not require the "perturbation" VW^T to the matrix A to be small relative to A. The matrix formed by VW^T can be large compared to A itself.

Implementation

Implementation is straightforward. Consider for example the specific case in which $\hat{A}$ and A differ except for column j. Then $\hat{A} = A + vw^T$ where

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \qquad w^T = e_j^T = (0, \ldots, 1, \ldots, 0) \quad \rightarrow \quad vw^T = \begin{bmatrix} 0 & \ldots & v_1 & \ldots & 0 \\ 0 & \ldots & v_2 & \ldots & 0 \\ & & \vdots & & \\ 0 & \ldots & v_n & \ldots & 0 \end{bmatrix}$$

$A + vw^T$ is typically referred to as a **rank-1 update of** A.

We suppose we want to solve $\hat{A}x = b$ and that we have a good method to solve $Ax = b$, e.g. by LU decomposition of A. The Sherman-Morrison-Woodbury formula gives

$$\hat{x} = \hat{A}^{-1}b = A^{-1}b - \frac{A^{-1}vw^T A^{-1}b}{1+\lambda}$$

All one needs to do is solve the two problems

$$Ay = b \qquad Az = v$$

using the LU decomposition. (Note that with LAPACK, both equations would be solved at the same time by forming a $n \times 2$ matrix with both b and v.) Then the solution to $\hat{A}x = b$ is:

$$x = y - \frac{zw^T y}{1 + w^T z}$$

or writing in terms of the inner vector product

$$x = y - z\left[\frac{< w, y >}{1 + < w, z >}\right]$$

If V and W are $n \times p$, we get a rank-p update. More computation is required but in principle it is implemented in the same way. The only significant difference is that the denominator $1 + w^T z$ becomes a matrix inverse $\left[I + W^T A^{-1} V\right]^{-1}$. However, this is a $p \times p$ matrix and for p small this is easy to invert (or, more correctly, it is easy to solve a linear system with a small $p \times p$ matrix).

Highly structured matrices

There are a number of particular linear systems that arise that can be solved in fewer than $O(n^3)$ work. Most of the cases are too particular to warrant detailed discussion here. However we give some of the standard examples.

Fourier methods

As we will see when we discuss eigenvalues, it is possible to diagonalize any non-defective $n \times n$ matrix A. This means finding a $n \times n$ matrix V such that

$$VAV^{-1} = D,$$

where D is a diagonal (or quasi-diagonal)$n \times n$ matrix. From this one can easily solve $Ax = b$:

$$\begin{aligned} Ax &= b \\ VAV^{-1}Vx &= Vb \\ DVx &= Vb \\ x &= V^{-1}D^{-1}Vb \end{aligned}$$

This is an insane approach to solving $Ax = b$ in general. However, if A can be diagonalized by a Fourier transform for example, then $V = FT$ (see Divide and Conquer in Maths Lecture I) and we have

$$x = (FT)^{-1}D^{-1}(FT)b$$

The Fourier transform and inverse Fourier transform each take $O(n \log_2 n)$ work and the inversion of the diagonal matrix takes only $O(n)$ work. Hence in this case x can be found in only $O(n \log_2 n)$ work.

Vandermonde matrices

Vandermonde matrices are of the form:

$$A = \begin{bmatrix} 1 & x_1 & \dots & x_1^{n-1} \\ 1 & x_2 & \dots & x_2^{n-1} \\ \vdots & & & \\ 1 & x_n & \dots & x_n^{n-1} \end{bmatrix}$$

or the transpose of this, for some set of numbers $x_1, \cdots, x_n$.

Such matrices arise in polynomial fitting for example,

$$Ac = y$$

where c is the vector of coefficients for polynomial fit to a set of data points $(x_1, y_1), \cdots, (x_n, y_n)$.

Warning - Vandermonde matrices are generally very poorly conditioned.

Nevertheless, in some circumstances one wants to solve linear equations with a Vandermonde matrix. This can be done in $O(n^2)$ operations rather than $O(n^3)$ operations. The way this is done is essentially to compute coefficients of the related Lagrange polynomials. The coefficients give directly the inverse of A and yet can be computed with $O(n^2)$ cost.

Toeplitz matrices

An $n \times n$ Toeplitz matrices is of the form:

$$A = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-n+1} \\ a_1 & a_0 & a_{-1} & \ddots & & \vdots \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{n-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{bmatrix}$$

and so has constant entries along each diagonal.

Due to the structure of Toeplitz matrices, the cost of operations is reduced from the general case. For example, the addition of two Toeplitz matrices can be done in $O(n)$ time; a Toeplitz matrix can be multiplied by a vector in $O(n \log n)$ time; the matrix multiplication of two Toeplitz matrices can be done in $O(n^2)$ time; and a Toeplitz systems of form $Ax = b$ can be solved with $O(n^2)$ cost. The memory cost is also greatly reduced from that of the general case.

Large Triangular Systems

Up until now we have consider the problem of backward (or forward) substitution as a small problem, and it is compared with factorizations. However, in practice linear equations are of the form:

$$Rx = b$$

where R is full upper triangular $n \times n$. Solving for x takes $O(n^2)$ work as we know. However, it may be the case that there is a banded upper triangular matrix B with bandwidth $ku \ll n$ such that BR is also banded with bandwidth J. In this case the linear equations can be written

$$BRx = Bb$$

and the resulting solution x can be found in $O(lu \times n)$ work rather than $O(n^2)$.

2.8 Iterative Improvement

There is a standard technique to improve a solution to a system of linear equations. Suppose we have

$$Ax = b$$

where x is the exact solution. Suppose our linear equation solver instead produces a solution $\bar{x}$. The question is: given the approximate solution $\bar{x}$, can we find a better approximation to the exact solution? The answer is often yes, to some extent. In any case it is often worth trying since this can be done at little additional cost.

Write $\bar{x} = x + \delta x$. If we can find δx, then we know how to correct (improve) our approximate solution to get the exact solution. Multiply A times $\bar{x}$ (which we have) to obtain $\bar{b} = A\bar{x}$. Now $\bar{b}$ will differ from b (else $\bar{x}$ would have been the exact solution), so write it as $\bar{b} = b + \delta b$.

We then have:

$$\begin{aligned} A\bar{x} &= \bar{b} \\ A(x + \delta x) &= b + \delta b \\ A\,x + A\,\delta x &= b + \delta b \\ A\,\delta x &= \delta b \end{aligned}$$

That is, to find our correction δx, we form a new right hand side $\delta b = A\,\bar{x} - b$ and solve:

$$A\delta x = \delta b$$

Our improved solution will be $\bar{x} - \delta x$. Of course this will itself not be exact and can repeat the process a few times. Hence the name iterative improvement. Note that once the matrix A has been factorized, each improvement step requires at most $O(n^2)$ additional work. Many performance libraries optionally provide iterative improvement of solutions computed by LU decomposition.

2.9 Stability and Conditioning

Finally, we consider how accurately we can expect to obtain a solution x to a linear system of equations, given that computers work in finite-precision arithmetic. You might think that by using iterative improvement just presented you could obtain a solution as accurate as numbers are represented in a computer, e.g. 16 digits of accuracy in double precision arithmetic. This is not the case. There is a limit to the accuracy you can expect. (Recall the discussion in Sec. 1.7.)

Let x be the exact solution to the equation $Ax = b$. Suppose now that A is varied by a small amount, $A \rightarrow A + \delta A$. You can think of this variation as arising because A is not represented exactly in a computer. The question is: how much does the exact solution change when A is varied by this small amount? The answer is that δx, the change in x, is bounded by:

$$\frac{||\delta x||}{||x||} \leq ||A||\,||A^{-1}||\frac{||\delta A||}{||A||}$$

(The proof of this can be found in any numerical linear algebra book.)

The quantity $||A||\,||A^{-1}||$ appears frequently in this subject and is defined to be the **condition number of the matrix** A:

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

Thus the bound becomes:

$$\frac{||\delta x||}{||x||} \leq \kappa(A) \frac{||\delta A||}{||A||} \tag{2.5}$$

This should be read as follows. The relative change in the solution (left-hand side) is bounded by the product of the condition number and the relative change in the matrix (right-hand side). The relative errors and condition number depend on the particular norm chosen, but we shall not worry about such details here.

In finite-precision arithmetic, with machine precision $\epsilon_{machine}$, one cannot expect A to be more sharply defined than a relative error of $\epsilon_{machine}$. Hence the best we can expect of our solution is:

$$\frac{||\delta x||}{||x||} = O(\kappa(A)\epsilon_{machine})$$

If $\kappa(A)$ is "small" then the matrix is **well-conditioned** and we can expect to be able to obtain accurate solutions. As $\kappa(A)$ becomes large, the problem becomes **poorly conditioned** and if $\kappa(A)$ diverges the problem becomes **ill-conditioned**. If A is singular, so that $Ax = b$ does not have a solution or the solution is not unique, then by definition $\kappa(A) = \infty$.

Some important points:

- What is considered to be a well-conditioned matrix depends on many factors, but typically in large scale numerical work, a well-conditioned problem may have a seemingly large value of κ. For example, in double precision arithmetic, the relative precision of the result when $\kappa(A) = 10^6$ can still be as high as 10 digits of accuracy and such a problem would not normally be considered poorly conditioned.
- It is very useful to know the condition number of the matrices one is working with. Numerical libraries often can provide an estimate of the condition number (in a particular norm).
- You should not think that the loss of precision of the solution x captured in equation (2.5) is due to the fact that we need to invert a matrix. The same loss of precision occurs for simple matrix-vector multiplication. Let y be given by $y = Ax$. Then the relative accuracy of y is bounded by

$$\frac{||\delta y||}{||y||} \leq \kappa(A) \frac{||\delta A||}{||A||}$$