DRAFT V2.1

From

# *Math, Numerics, & Programming*

# *(for Mechanical Engineers)*

Masayuki Yano
James Douglass Penn
George Konidaris
Anthony T Patera

August 2013

# Unit V

# (Numerical) Linear Algebra 2: Solution of Linear Systems

# Chapter 24

# Motivation



We thank Dr Phuong Huynh of MIT for generously developing and sharing the robot arm model, finite element discretization, and computational results reported in this chapter.

## 24.1 A Robot Arm

In the earlier units we have frequently taken inspiration from applications related to robots — navigation, kinematics, and dynamics. In these earlier analyses we considered systems consisting of relatively few "lumped" components and hence the computational tasks were rather modest. However, it is also often important to address not just lumped components but also the detailed deformations and stresses within say a robot arm: excessive deformation can compromise performance in precision applications; and excessive stresses can lead to failure in large manufacturing tasks.

The standard approach for the analysis of deformations and stresses is the finite element (FE) method. In the FE approach, the spatial domain is first broken into many (many) small regions denoted elements: this decomposition is often denoted a triangulation (or more generally a grid or mesh), though elements may be triangles, quadrilaterals, tetrahedra, or hexahedra; the vertices of these elements define nodes (and we may introduce additional nodes at say edge or face midpoints). The displacement field within each such element is then expressed in terms of a low order polynomial representation which interpolates the displacements at the corresponding nodes. Finally, the partial differential equations of linear elasticity are invoked, in variational form, to yield equilibrium equations at (roughly speaking) each node in terms of the displacements at the neighboring nodes. Very crudely, the coefficients in these equations represent effective spring constants which reflect the relative nodal geometric configuration and the material properties. We may express this system of $n$ equations — one equation for each node — in $n$ unknowns — one displacement (or "degree of freedom") for each node — as $Ku = f$, in which $K$ is an $n \times n$ matrix, $u$ is an $n \times 1$ vector of the unknown displacements, and $f$ is an $n \times 1$ vector of imposed forces or "loads."[1]

We show in Figure 24.1 the finite element solution for a robot arm subject only to the "self-load" induced by gravity. The blue arm is the unloaded (undeformed) arm, whereas the multi-color arm is the loaded (deformed) arm; note in the latter we greatly amplify the actual displacements for purposes of visualization. The underlying triangulation — in particular surface triangles associated

[1] In fact, for this vector-valued displacement field, there are three equations and three degrees of freedom for each (geometric) node. For simplicity we speak of (generalized) nodes equated to degrees of freedom.

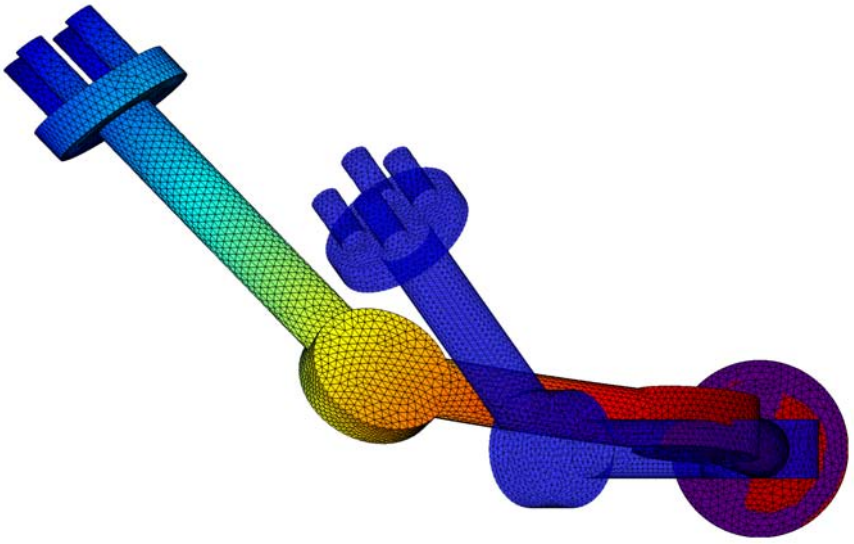


Figure 24.1: Deflection of robot arm.

with volumetric tetrahedral elements — is also shown. In this FE discretization there are $n = 60{,}030$ degrees of freedom (for technical reasons we do not count the nodes at the robot shoulder). The issue is thus how to effectively solve the linear system of equations $Ku = f$ given the very large number of degrees of freedom. In fact, many finite element discretizations result not in $10^5$ unknowns but rather $10^6$ or even $10^7$ unknowns. The computational task is thus formidable, in particular since typically one analysis will not suffice — rather, many analyses will be required for purposes of design and optimization.

## 24.2 Gaussian Elimination and Sparsity

If we were to consider the most obvious tactic — find $K^{-1}$ and then compute $K^{-1}f$ — the result would be disastrous: days of computing (if the calculation even completed). And indeed even if we were to apply a method of choice — Gaussian elimination (or LU decomposition) — without any regard to the actual structure of the matrix $K$, the result would still be disastrous. Modern computational solution strategies must and do take advantage of a key attribute of $K$ — sparseness.[2] In short, there is no reason to perform operations which involve zero operands and will yield zero for a result. In MechE systems sparseness is not an exceptional attribute but rather, and very fortunately, the rule: the force in a (Hookean) spring is just determined by the deformations of nearest neighbors, not by distant neighbors; similar arguments apply in heat transfer, fluid mechanics, and acoustics. (Note this is not to say that the equilibrium displacement at one node does not depend on the applied forces at the other nodes; quite to the contrary, a force applied at one node will yield nonzero displacements at all the nodes of the system. We explore this nuance more deeply when we explain why formation of the inverse of a (sparse) matrix is a very poor idea.)

We present in Figure 24.2 the structure of the matrix $K$: the dots indicate nonzero entries in the matrix. We observe that most of the matrix elements are in fact zero. Indeed, of the 3,603,600,900 entries of $K$, 3,601,164,194 entries are zero; put differently, there are only 2,436,706

[2]Note in this unit we shall consider only *direct* solution methods; equally important, but more advanced in terms of formulation, analysis, and robust implementation (at least if we consider the more efficient varieties), are *iterative* solution methods. In actual practice, most state-of-the-art solvers include some combination of direct and iterative aspects. And in all cases, sparsity plays a key role.

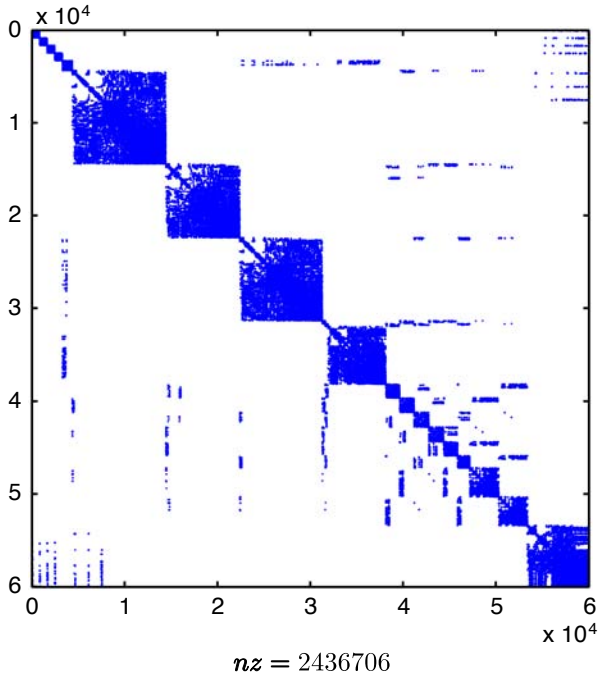



Figure 24.2: Structure of stiffness matrix $K$.

nonzero entries of $K$ — only 0.06% of the entries of $K$ are nonzero. If we exploit these zeros, *both* in our numerical approach and in the implementation/programming, we can now solve our system in reasonable time: about 230 seconds on a Mac laptop (performing a particular version of sparse Gaussian elimination based on Cholesky factorization). However, we can do still better.

In particular, although some operations "preserve" all sparsity, other operations — in particular, Gaussian elimination — result in "fill-in": zero entries become nonzero entries which thus must be included in subsequent calculations. The extent to which fill-in occurs depends on the way in which we order the equations and unknowns (which in turn determines the structure of the matrix). There is no unique way that we must choose to order the unknowns and equations: a particular node say near the elbow of the robot arm could be node (column) "1" — or node (column) "2,345"; similarly, the equilibrium equation for this node could be row "2" — or row "58,901".[3] We can thus strive to find a best ordering which minimizes fill-in and thus maximally exploits sparsity. In fact, this optimization problem is very difficult, but there are efficient heuristic procedures which yield very good results. The application of such a heuristic to our matrix $K$ yields the new (that is, reordered) matrix $K'$ shown in Figure 24.3. If we now reapply our sparse Cholesky approach the computational time is now very modest — only 7 seconds. Hence proper choice of algorithm and an appropriate implementation of the algorithm can reduce the computational effort from days to several seconds.

## 24.3 Outline

In this unit we first consider the well-posedness of linear systems: $n$ equations in $n$ unknowns. We understand the conditions under which a solution exists and is unique, and we motivate — from a physical perspective — the situations in which a solution might not exist or might exist but not be

[3]For our particular problem it is best to permute the unknowns and equations in the same fashion to preserve symmetry of $K$.

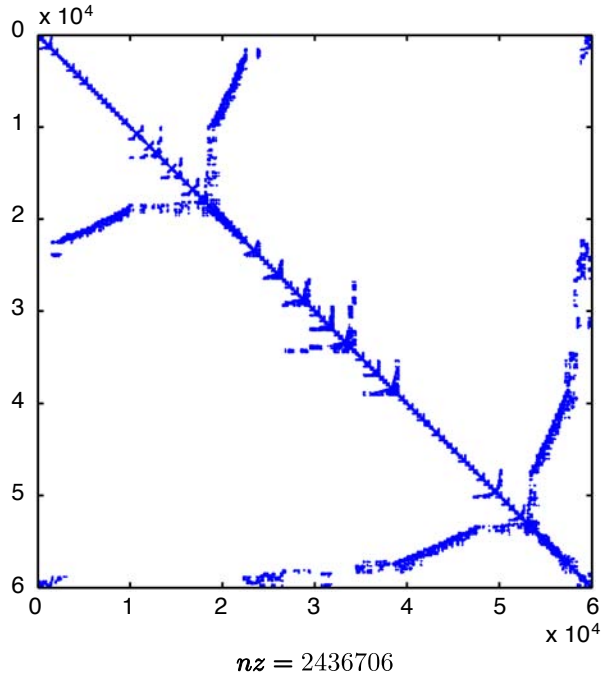



Figure 24.3: Structure of reordered stiffness matrix $K'$.

unique.

We next consider the basic Gaussian eliminate algorithm. We then proceed to Gaussian elimination for sparse systems — motivated by the example and numerical results presented above for the robot arm. Finally, we consider the MATLAB implementation of these approaches. (Note that all results in this chapter are based on MATLAB implementations.)

We notably omit several important topics: we do not consider iterative solution procedures; we do not consider, except for a few remarks, the issue of numerical stability and conditioning.

# Chapter 25

# Linear Systems



## 25.1 Model Problem: $n = 2$ Spring-Mass System in Equilibrium

### 25.1.1 Description

We will introduce here a simple spring-mass system, shown in Figure 25.1, which we will use throughout this chapter to illustrate various concepts associated with linear systems and associated solution techniques. Mass 1 has mass $m_1$ and is connected to a stationary wall by a spring with stiffness $k_1$. Mass 2 has mass of $m_2$ and is connected to the mass $m_1$ by a spring with stiffness $k_2$.

We denote the displacements of mass 1 and mass 2 by $u_1$ and $u_2$, respectively: positive values correspond to displacement away from the wall; we choose our reference such that in the absence of applied forces — the springs unstretched — $u_1 = u_2 = 0$. We next introduce (steady) forces $f_1$ and $f_2$ on mass 1 and mass 2, respectively; positive values correspond to force away from the wall. We are interested in predicting the equilibrium displacements of the two masses, $u_1$ and $u_2$, for prescribed forces $f_1$ and $f_2$.

We note that while all real systems are inherently dissipative and therefore are characterized not just by springs and masses but also dampers, the dampers (or damping coefficients) do not affect the system at equilibrium — since $d/dt$ vanishes in the steady state — and hence for equilibrium considerations we may neglect losses. Of course, it is damping which ensures that the system ultimately achieves a stationary (time-independent) equilibrium.[1]

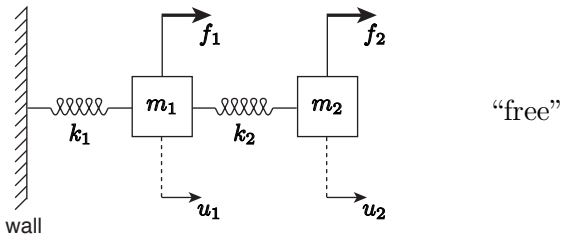



Figure 25.1: A system of two masses and two springs anchored to a wall and subject to applied forces.

[1] In some rather special cases — which we will study later in this chapter — the equilibrium displacement is indeed affected by the initial conditions and damping. This special case in fact helps us better understand the mathematical aspects of systems of linear equations.

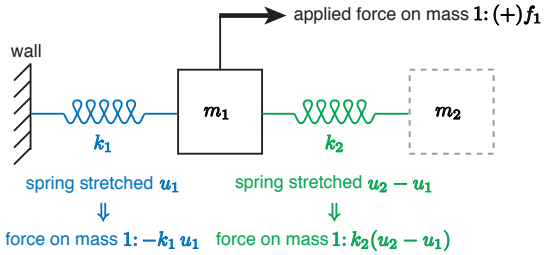



Figure 25.2: Forces on mass 1.

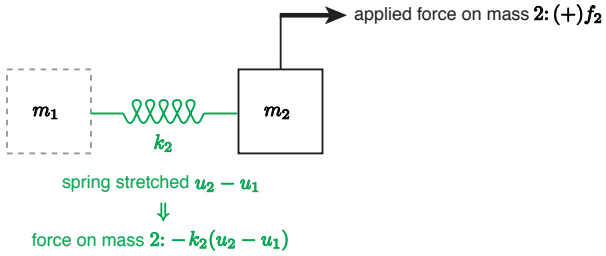



Figure 25.3: Forces on mass 2.

We now derive the equations which must be satisfied by the displacements $u_1$ and $u_2$ at equilibrium. We first consider the forces on mass 1, as shown in Figure 25.2. Note we apply here Hooke's law — a constitutive relation — to relate the force in the spring to the compression or extension of the spring. In equilibrium the sum of the forces on mass 1 — the applied forces and the forces due to the spring — must sum to zero, which yields

$$f_1 - k_1 u_1 + k_2(u_2 - u_1) = 0 \ .$$

(More generally, for a system *not* in equilibrium, the right-hand side would be $m_1 \ddot{u}_1$ rather than zero.) A similar identification of the forces on mass 2, shown in Figure 25.3, yields for force balance

$$f_2 - k_2(u_2 - u_1) = 0 \ .$$

This completes the physical statement of the problem.

Mathematically, our equations correspond to a system of $n = 2$ linear equations, more precisely, 2 equations in 2 unknowns:

$$(k_1 + k_2)\, u_1 - k_2\, u_2 \quad = \quad f_1 \ , \tag{25.1}$$

$$-k_2\, u_1 + k_2\, u_2 \quad = \quad f_2 \ . \tag{25.2}$$

Here $u_1$ and $u_2$ are *unknown*, and are placed on the left-hand side of the equations, and $f_1$ and $f_2$ *are known*, and placed on the right-hand side of the equations. In this chapter we ask several questions about this linear system — and more generally about linear systems of $n$ equations in $n$ unknowns. First, existence: when do the equations have a solution? Second, uniqueness: if a solution exists, is it unique? Although these issues appear quite theoretical in most cases the mathematical subtleties are in fact informed by physical (modeling) considerations. In later chapters in this unit we will ask a more obviously practical issue: how do we solve systems of linear equations efficiently?

But to achieve these many goals we must put these equations in matrix form in order to best take advantage of both the theoretical and practical machinery of linear algebra. As we have already

addressed the translation of sets of equations into corresponding matrix form in Unit III (related to overdetermined systems), our treatment here shall be brief.

We write our two equations in two unknowns as $Ku = f$, where $K$ is a $2\times2$ matrix, $u = (u_1 \;\; u_2)^{\mathrm{T}}$ is a $2 \times 1$ vector, and $f = (f_1 \;\; f_2)^{\mathrm{T}}$ is a $2 \times 1$ vector. The elements of $K$ are the coefficients of the equations (25.1) and (25.2):

$$\begin{array}{ccccc} & & \textit{unknown} & & \textit{known} \\ \begin{pmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{pmatrix} & & \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} & = & \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} \quad \cdots \quad \begin{array}{l} \leftarrow \text{Equation (25.1)} \\ \leftarrow \text{Equation (25.2)} \end{array} \\ K & & u & & f \\ 2 \times 2 & & 2 \times 1 & & 2 \times 1 \end{array} \; . \tag{25.3}$$

We briefly note the connection between equations (25.3) and (25.1). We first note that $Ku = F$ implies equality of the two vectors $Ku$ and $F$ and hence equality of each component of $Ku$ and $F$. The first component of the vector $Ku$, from the row interpretation of matrix multiplication,[2] is given by $(k_1 + k_2)u_1 - k_2u_2$; the first component of the vector $F$ is of course $f_1$. We thus conclude that $(Ku)_1 = f_1$ correctly produces equation (25.1). A similar argument reveals that the $(Ku)_2 = f_2$ correctly produces equation (25.2).

### 25.1.2 SPD Property

We recall that a real $n \times n$ matrix $A$ is Symmetric Positive Definite (SPD) if $A$ is symmetric

$$A^{\mathrm{T}} = A \; , \tag{25.4}$$

and $A$ is positive definite

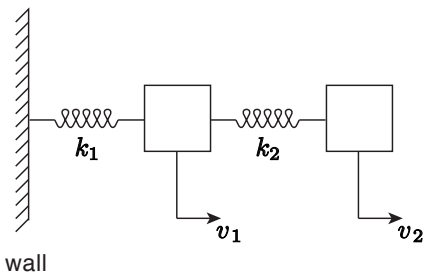
$$v^{\mathrm{T}}Av > 0 \text{ for any } v \neq 0 \; . \tag{25.5}$$

Note in equation (25.5) that $Av$ is an $n \times 1$ vector and hence $v^{\mathrm{T}}(Av)$ is a scalar — a real number. Note also that the positive definite property (25.5) implies that if $v^{\mathrm{T}}Av = 0$ then $v$ must be the zero vector. There is also a connection to eigenvalues: symmetry implies real eigenvalues, and positive definite implies strictly positive eigenvalues (and in particular, no zero eigenvalues).

There are many implications of the SPD property, all very pleasant. In the context of the current unit, an SPD matrix ensures positive eigenvalues which in turn will ensure existence and uniqueness of a solution — which is the topic of the next section. Furthermore, an SPD matrix ensures stability of the Gaussian elimination process — the latter is the topic in the following chapters. We also note that, although in this unit we shall focus on direct solvers, SPD matrices also offer advantages in the context of iterative solvers: the very simple and efficient conjugate gradient method can be applied (only to) SPD matrices. The SPD property is also the basis of minimization principles which serve in a variety of contexts. Finally, we note that the SPD property is often, though not always, tied to a natural physical notion of energy.

We shall illustrate the SPD property for our simple $2 \times 2$ matrix $K$ associated with our spring system. In particular, we now again consider our system of two springs and two masses but now we introduce an arbitrary imposed displacement vector $v = (v_1 \;\; v_2)^{\mathrm{T}}$, as shown in Figure 25.4. In this case our matrix $A$ is given by $K$ where

[2] In many, but not all, cases it is more intuitive to develop matrix equations from the row interpretation of matrix multiplication; however, as we shall see, the column interpretation of matrix multiplication can be very important from the theoretical perspective.




$v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$ any "imposed" displacement

Figure 25.4: Spring-mass system: imposed displacements $v$.

$$K = \begin{pmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{pmatrix} .$$

We shall assume that $k_1 > 0$ and $k_2 > 0$ — our spring constants are strictly positive. We shall return to this point shortly.

We can then form the scalar $v^{\mathrm{T}} K v$ as

$$\begin{aligned} v^{\mathrm{T}} K v &= v^{\mathrm{T}} \underbrace{\begin{pmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}} \\ &= (v_1 \quad v_2) \underbrace{\begin{pmatrix} (k_1 + k_2)v_1 & -k_2 v_2 \\ -k_2 v_1 & k_2 v_2 \end{pmatrix}}_{Kv} \\ &= v_1^2(k_1 + k_2) - v_1 v_2 k_2 - v_2 v_1 k_2 + v_2^2 k_2 \\ &= v_1^2 k_1 + \left( v_1^2 - 2v_1 v_2 + v_2^2 \right) k_2 \\ &= k_1 v_1^2 + k_2 (v_1 - v_2)^2 . \end{aligned}$$

We now inspect this result.

In particular, we may conclude that, under our assumption of positive spring constants, $v^{\mathrm{T}} K v \geq 0$. Furthermore, $v^{\mathrm{T}} K v$ can only be zero if $v_1 = 0$ and $v_1 = v_2$, which in turn implies that $v^{\mathrm{T}} K v$ can only be zero if both $v_1$ and $v_2$ are zero — $v = 0$. We thus see that $K$ is SPD: $v^{\mathrm{T}} K v > 0$ unless $v = 0$ (in which case of course $v^{\mathrm{T}} K v = 0$). Note that if either $k_1 = 0$ or $k_2 = 0$ then the matrix is not SPD: for example, if $k_1 = 0$ then $v^{\mathrm{T}} K v = 0$ for any $v = (c \quad c)^{\mathrm{T}}$, $c$ a real constant; similarly, if $k_2 = 0$, then $v^{\mathrm{T}} K v = 0$ for any $v = (0 \quad c)^{\mathrm{T}}$, $c$ a real constant.

We can in this case readily identify the connection between the SPD property and energy. In particular, for our spring system, the potential energy in the spring system is simply $\frac{1}{2} v^{\mathrm{T}} K v$:

PE (potential/elastic energy) =

$$\underbrace{\frac{1}{2} k_1 v_1^2}_{\substack{\text{energy in} \\ \text{spring 1}}} + \underbrace{\frac{1}{2} k_2 (v_2 - v_1)^2}_{\substack{\text{energy in} \\ \text{spring 2}}} = \frac{1}{2} v^{\mathrm{T}} A v > 0 \qquad (\text{unless } v = 0) ,$$

where of course the final conclusion is only valid for strictly positive spring constants.

Finally, we note that many MechE systems yield models which in turn may be described by SPD systems: structures (trusses, ...); linear elasticity; heat conduction; flow in porous media (Darcy's Law); Stokes (or creeping) flow of an incompressible fluid. (This latter is somewhat complicated by the incompressibility constraint.) All these systems are very important in practice and indeed ubiquitous in engineering analysis and design. However, it is also essential to note that many other very important MechE phenomena and systems — for example, forced convection heat transfer, non-creeping fluid flow, and acoustics — do *not* yield models which may be described by SPD matrices.

## 25.2 Existence and Uniqueness: $n = 2$

### 25.2.1 Problem Statement

We shall now consider the existence and uniqueness of solutions to a general system of $(n =)$ 2 equations in $(n =)$ 2 unknowns. We first introduce a matrix $A$ and vector $f$ as

$$\begin{aligned} 2 \times 2 \text{ matrix} \quad A &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \\ & \qquad ; \\ 2 \times 1 \text{ vector} \quad f &= \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} \end{aligned}$$

our equation for the $2 \times 1$ unknown vector $u$ can then be written as

$$Au = f\,, \quad \text{or} \quad \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}, \quad \text{or} \quad \left.\begin{aligned} A_{11}u_1 + A_{12}u_2 &= f_1 \\ A_{21}u_1 + A_{22}u_2 &= f_2 \end{aligned}\right\}\,.$$

Note these three expressions are equivalent statements proceeding from the more abstract to the more concrete. We now consider existence and uniqueness; we shall subsequently interpret our general conclusions in terms of our simple $n = 2$ spring-mass system.

### 25.2.2 Row View

We first consider the *row* view, similar to the row view of matrix multiplication. In this perspective we consider our solution vector $u = (u_1 \quad u_2)^{\mathrm{T}}$ as a point $(u_1, u_2)$ in the two dimensional Cartesian plane; a *general* point in the plane is denoted by $(v_1, v_2)$ corresponding to a vector $(v_1 \quad v_2)^{\mathrm{T}}$. In particular, $u$ is the particular point in the plane which lies both on the straight line described by the first equation, $(Av)_1 = f_1$, denoted 'eqn1' and shown in Figure 25.5 in blue, *and* on the straight line described by the second equation, $(Av)_2 = f_2$, denoted 'eqn2' and shown in Figure 25.5 in green.

We directly observe three possibilities, familiar from any first course in algebra; these three cases are shown in Figure 25.6. In case $(i)$, the two lines are of different slope and there is clearly one and only one intersection: the solution thus exists and is furthermore unique. In case $(ii)$ the two lines are of the same slope and furthermore coincident: a solution exists, but it is not unique — in fact, there are an infinity of solutions. This case corresponds to the situation in which the two equations in fact contain *identical* information. In case $(iii)$ the two lines are of the same slope but *not* coincident: no solution exists (and hence we need not consider uniqueness). This case corresponds to the situation in which the two equations contain *inconsistent* information.

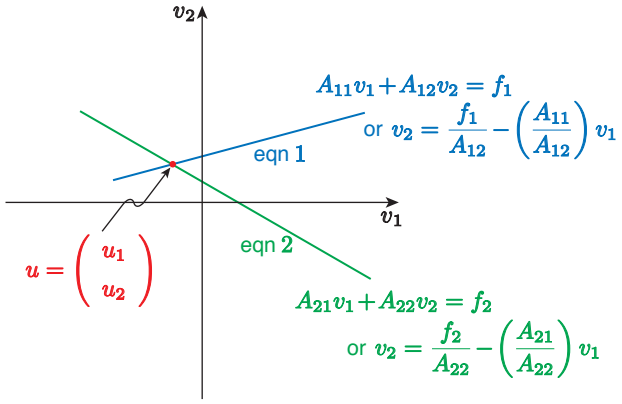
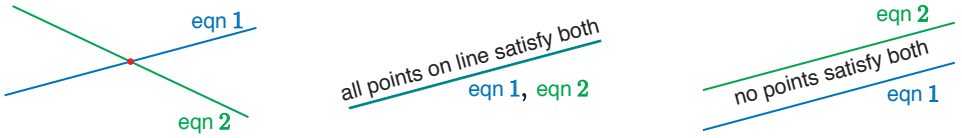



Figure 25.5: Row perspective: $u$ is the intersection of two straight lines.




| $(i)$ | $(ii)$ | $(iii)$ |
|---|---|---|
| exists ✓ | exists ✓ | exists ✗ |
| unique ✓ | unique ✗ | ~~unique~~ |
| | *redundant* information, | *inconsistent* information |
| | *infinity* of solutions | *no* solution |

Figure 25.6: Three possibilities for existence and uniqueness; row interpretation.

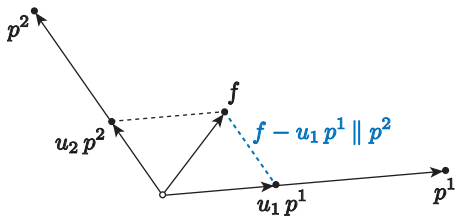



Figure 25.7: Parallelogram construction for the case in case in which a unique solution exists.

We see that the condition for (both) existence and uniqueness is that the slopes of 'eqn1' and 'eqn2' must be different, or $A_{11}/A_{12} \neq A_{21}/A_{22}$, or $A_{11}A_{22} - A_{12}A_{21} \neq 0$. We recognize the latter as the more familiar condition $det(A) \neq 0$. In summary, if $det(A) \neq 0$ then our matrix $A$ is non-singular and the system $Au = f$ has a unique solution; if $det(A) \neq 0$ then our matrix $A$ is singular and either our system has an infinity of solutions or no solution, depending on $f$. (In actual practice the determinant condition is not particularly practical computationally, and serves primarily as a convenient "by hand" check for very small systems.) We recall that a non-singular matrix $A$ has an inverse $A^{-1}$ and hence in case $(i)$ we can write $u = A^{-1}f$; we presented this equation earlier under the assumption that $A$ is non-singular — now we have provided the condition under which this assumption is true.

### 25.2.3 The Column View

We next consider the column view, analogous to the column view of matrix multiplication. In particular, we recall from the column view of matrix-vector multiplication that we can express $Au$ as

$$Au = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \underbrace{\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}}_{p^1} u_1 + \underbrace{\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}}_{p^2} u_2 \quad ,$$

where $p^1$ and $p^2$ are the first and second column of $A$, respectively. Our system of equations can thus be expressed as

$$Au = f \quad \Leftrightarrow \quad p^1 u_1 + p^2 u_2 = f \ .$$

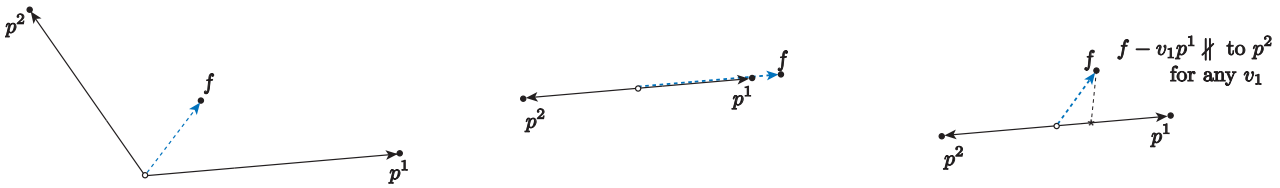
Thus the question of existence and uniqueness can be stated alternatively: is there a (unique?) combination, $u$, of columns $p^1$ and $p^2$ which yields $f$?

We start by answering this question pictorially in terms of the familiar parallelogram construction of the sum of two vectors. To recall the parallelogram construction, we first consider in detail the case shown in Figure 25.7. We see that in the instance depicted in Figure 25.7 there is clearly a unique solution: we choose $u_1$ such that $f - u_1 p^1$ is parallel to $p^2$ (there is clearly only one such value of $u_1$); we then choose $u_2$ such that $u_2 p^2 = f - u_1 p^1$.

We can then identify, in terms of the parallelogram construction, three possibilities; these three cases are shown in Figure 25.8. Here case $(i)$ is the case already discussed in Figure 25.7: a unique solution exists. In both cases $(ii)$ and $(iii)$ we note that

$$p^2 = \gamma p^1 \quad \text{or} \quad p^2 - \gamma p^1 = 0 \ (p^1 \text{ and } p^2 \text{ are linearly dependent})$$

for some $\gamma$, where $\gamma$ is a scalar; in other words, $p^1$ and $p^2$ are colinear — *point in the same direction to within a sign* (though $p^1$ and $p^2$ may of course be of different magnitude). We now discuss these two cases in more detail.




| $(i)$ | $(ii)$ | $(iii)$ |
|---|---|---|
| exists ✓ | exists ✓ | exists ✗ |
| unique ✓ | unique ✗ | ~~unique~~ |
| | (only $p^1$, *or* more $p^1$ and some $p^2$, *or* ...) | |

Figure 25.8: Three possibilities for existence and uniqueness; the column perspective.

In case $(ii)$, $p^1$ and $p^2$ are colinear but $f$ *also* is colinear with $p^1$ (and $p^2$) — say $f = \beta p^1$ for some scalar $\beta$. We can thus write

$$\begin{aligned} f &= p^1 \cdot \beta + p^2 \cdot 0 \\ &= \begin{pmatrix} p^1 & p^2 \end{pmatrix} \begin{pmatrix} \beta \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \underbrace{\begin{pmatrix} \beta \\ 0 \end{pmatrix}}_{u^*} \\ &= Au^* \ , \end{aligned}$$

and hence $u^*$ is *a* solution. However, we also know that $-\gamma p^1 + p^2 = 0$, and hence

$$\begin{aligned} 0 &= p^1 \cdot (-\gamma) + p^2 \cdot (1) \\ &= \begin{pmatrix} p^1 & p^2 \end{pmatrix} \begin{pmatrix} -\gamma \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} -\gamma \\ 1 \end{pmatrix} \\ &= A \begin{pmatrix} -\gamma \\ 1 \end{pmatrix} \ . \end{aligned}$$

Thus, for any $\alpha$,

$$u = \underbrace{u^* + \alpha \begin{pmatrix} -\gamma \\ 1 \end{pmatrix}}_{\text{infinity of solutions}}$$

satisfies $Au = f$, since

$$A\left(u^* + \alpha \begin{pmatrix} -\gamma \\ 1 \end{pmatrix}\right) = Au^* + A\left(\alpha \begin{pmatrix} -\gamma \\ 1 \end{pmatrix}\right)$$

$$= Au^* + \alpha A \begin{pmatrix} -\gamma \\ 1 \end{pmatrix}$$

$$= f + \alpha \cdot 0$$

$$= f \ .$$

This demonstrates that in case $(ii)$ there are an infinity of solutions parametrized by the arbitrary constant $\alpha$.

Finally, we consider case $(iii)$. In this case it is clear from our parallelogram construction that for no choice of $v_1$ will $f - v_1 p^1$ be parallel to $p^2$, and hence for no choice of $v_2$ can we form $f - v_1 p^1$ as $v_2 p^2$. Put differently, a linear combination of two colinear vectors $p^1$ and $p^2$ can not combine to form a vector perpendicular to both $p^1$ and $p^2$. Thus no solution exists.

Note that the vector $(-\gamma \quad 1)^{\mathrm{T}}$ is an eigenvector of $A$ corresponding to a zero eigenvalue.[3] By definition the matrix $A$ has no effect on an eigenvector associated with a zero eigenvalue, and it is for this reason that if we have one solution to $Au = f$ then we may add to this solution *any* multiple — here $\alpha$ — of the zero-eigenvalue eigenvector to obtain yet another solution. More generally a matrix $A$ is non-singular if and only if it has no zero eigenvalues; in that case — case $(i)$ — the inverse exists and we may write $u = A^{-1} f$. On the other hand, if $A$ has any zero eigenvalues then $A$ is singular and the inverse does not exist; in that case $Au = f$ may have either many solutions or no solutions, depending on $f$. From our discussion of SPD systems we also note that $A$ SPD is a sufficient (but not necessary) condition for the existence of the inverse of $A$.

### 25.2.4 A Tale of Two Springs

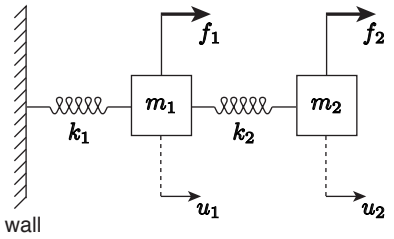
We now interpret our results for existence and uniqueness for a mechanical system — our two springs and masses — to understand the connection between the model and the mathematics. We again consider our two masses and two springs, shown in Figure 25.9, governed by the system of equations

$$Au = f \quad \text{for} \quad A = K \equiv \begin{pmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{pmatrix} \ .$$

We analyze three different scenarios for the spring constants and forces, denoted (I), (II), and (III), which we will see correspond to cases $(i)$, $(ii)$, and $(iii)$, respectively, as regards existence and uniqueness. We present first (I), then (III), and then (II), as this order is more physically intuitive.

(I) In scenario (I) we choose $k_1 = k_2 = 1$ (more physically we would take $k_1 = k_2 = \overline{k}$ for some value of $\overline{k}$ expressed in appropriate units — but our conclusions will be the same) and $f_1 = f_2 = 1$ (more physically we would take $f_1 = f_2 = \overline{f}$ for some value of $\overline{f}$ expressed in appropriate units — but our conclusions will be the same). In this case our matrix $A$ and

[3] All scalar multiples of this eigenvector define what is known as the right nullspace of $A$.



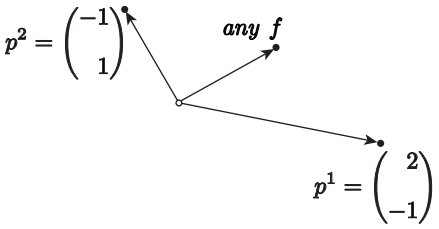

$$f = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}, \qquad u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$$

given to find

Figure 25.9: System of equilibrium equations for two springs and two masses.

associated column vectors $p^1$ and $p^2$ take the form shown below. It is clear that $p^1$ and $p^2$ are not colinear and hence a unique solution exists for any $f$. We are in case $(i)$.

$$A = \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}$$

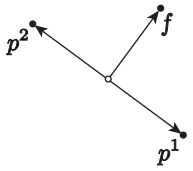



case $(i)$: exists ✓, unique ✓

(III) In scenario (III) we chose $k_1 = 0$, $k_2 = 1$ and $f_1 = f_2 = 1$. In this case our vector $f$ and matrix $A$ and associated column vectors $p^1$ and $p^2$ take the form shown below. It is clear that a linear combination of $p^1$ and $p^2$ can not possibly represent $f$ — and hence no solution exists. We are in case $(iii)$.

$$f = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

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




case $(iii)$: exists ✗, ~~unique~~

We can readily identify the cause of the difficulty. For our particular choice of spring constants in scenario (III) the first mass is no longer connected to the wall (since $k_1 = 0$); thus our spring system now appears as in Figure 25.10. We see that there is a net force on our *system* (of two masses) — the net force is $f_1 + f_2 = 2 \neq 0$ — and hence it is clearly inconsistent to assume equilibrium.[4] In even greater detail, we see that the equilibrium equations for each mass are inconsistent (note $f_{\text{spr}} = k_2(u_2 - u_1)$) and hence we must replace the zeros on the right-hand sides with mass $\times$ acceleration terms. At fault here is not the mathematics but rather the model provided for the physical system.

[4] In contrast, in scenario (I), the wall provides the necessary reaction force in order to ensure equilibrium.

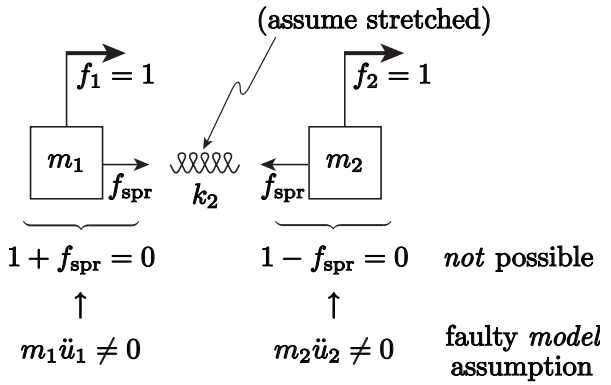
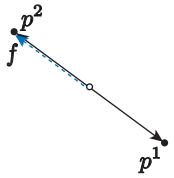



Figure 25.10: Scenario III

(II) In this scenario we choose $k_1 = 0$, $k_2 = 1$ and $f_1 = 1, f_2 = -1$. In this case our vector $f$ and matrix $A$ and associated column vectors $p^1$ and $p^2$ take the form shown below. It is clear that a linear combination of $p^1$ and $p^2$ now *can* represent $f$ — and in fact there are many possible combinations. We are in case $(ii)$.

$$f = \begin{pmatrix} -1 \\ \\ 1 \end{pmatrix}$$

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




case $(ii)$: exists ✓, unique ✗

We can explicitly construct the family of solutions from the general procedure described earlier:

$$\begin{aligned} p^2 &= = \underbrace{-1}_{\gamma} p^1 \ , \\ f &= \underbrace{-1}_{\beta} p^1 \Rightarrow u^* = \begin{pmatrix} = -1 \\ \\ 0 = \end{pmatrix} \\ &\Downarrow \\ u &= u^* + \alpha \begin{pmatrix} -\gamma \\ \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} -1 \\ \\ 0 \end{pmatrix} + \alpha \begin{pmatrix} 1 \\ \\ 1 \end{pmatrix} \end{aligned}$$

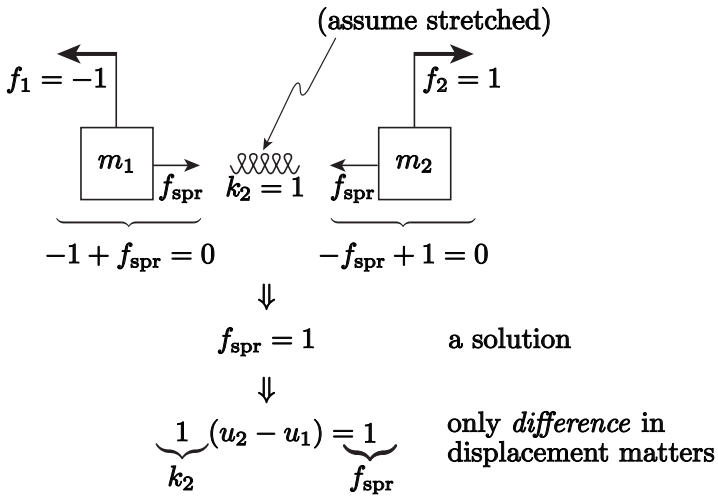



Figure 25.11: Scenario II. (Note on the left mass the $f_1$ arrow indicates the direction of the force $f_1 = -1$, not the direction of positive force.)

for *any* $\alpha$. Let us check the result explicitly:

$$A\left(\begin{pmatrix}-1\\0\end{pmatrix} + \begin{pmatrix}\alpha\\\alpha\end{pmatrix}\right) = \begin{pmatrix}1 & -1\\-1 & 1\end{pmatrix}\begin{pmatrix}1+\alpha\\\alpha\end{pmatrix}$$

$$= \begin{pmatrix}(-1+\alpha)-\alpha\\(1-\alpha)+\alpha\end{pmatrix}$$

$$= \begin{pmatrix}-1\\1\end{pmatrix}$$

$$= f \ ,$$

as desired. Note that the zero-eigenvalue eigenvector here is given by $(-\gamma \quad 1)^{\mathrm{T}} = (1 \quad 1)^{\mathrm{T}}$ and corresponds to an equal (or translation) shift in both displacements, which we will now interpret physically.

In particular, we can readily identify the cause of the non-uniqueness. For our choice of spring constants in scenario (II) the first mass is no longer connected to the wall (since $k_1 = 0$), just as in scenario (III). Thus our spring system now appears as in Figure 25.11. But unlike in scenario (III), in scenario (II) the net force on the system is zero — $f_1$ and $f_2$ point in opposite directions — and hence an equilibrium is possible. Furthermore, we see that each mass is in equilibrium for a spring force $f_{\text{spr}} = 1$. Why then is there not a *unique* solution? Because to obtain $f_{\text{spr}} = 1$ we may choose any displacements $u$ such that $u_2 - u_1 = 1$ (for $k_2 = 1$): the system is not anchored to wall — it just floats — and thus equilibrium is maintained if we shift (or translate) both masses by the same displacement (our eigenvector) such that the "stretch" remains invariant. This is illustrated in Figure 25.12, in which $\alpha$ is the shift in displacement. Note $\alpha$ is *not* determined by the *equilibrium* model; $\alpha$ *could* be determined from a *dynamical* model and in particular would depend on the initial conditions *and* the

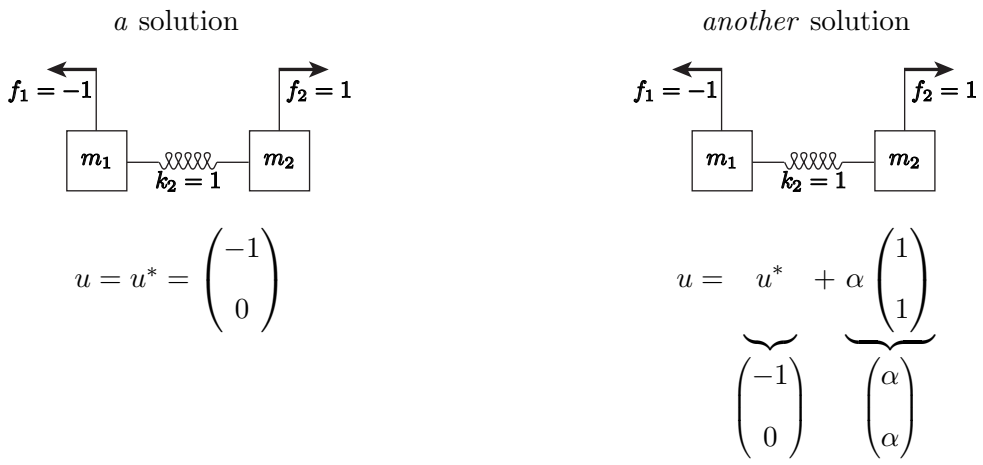



Figure 25.12: Scenario (II): non-uniqueness.

damping in the system.

We close this section by noting that for scenario (I) $k_1 > 0$ and $k_2 > 0$ and hence $A$ ($\equiv K$) is SPD: thus $A^{-1}$ exists and $Au = f$ has a unique solution for any forces $f$. In contrast, in scenarios (II) and (III), $k_1 = 0$, and hence $A$ is no longer SPD, and we are no longer guaranteed that $A^{-1}$ exists — and indeed it does not exist. We also note that in scenario (III) the zero-eigenvalue eigenvector $(1 \quad 1)^{\mathrm{T}}$ is precisely the $v$ which yields zero energy, and indeed a shift (or translation) of our unanchored spring system does not affect the energy in the spring.

## 25.3 A "Larger" Spring-Mass System: $n$ Degrees of Freedom

We now consider the equilibrium of the system of $n$ springs and masses shown in Figure 25.13. (This string of springs and masses in fact is a model (or discretization) of a continuum truss; each spring-mass is a small segment of the truss.) Note for $n = 2$ we recover the small system studied in the preceding sections. This larger system will serve as a more "serious" model problem both as regards existence and uniqueness but even more importantly as regard computational procedures. We then consider force balance on mass 1,

$$\sum \text{ forces on mass } 1 = 0$$
$$\Rightarrow f_1 - k_1 u_1 + k_2(u_2 - u_1) = 0 \ ,$$

and then on mass 2,

$$\sum \text{ forces on mass } 2 = 0$$
$$\Rightarrow f_2 - k_2(u_2 - u_1) + k_3(u_3 - u_2) = 0 \ ,$$

and then on a typical interior mass $i$ (hence $2 \le i \le n-1$)

$$\sum \text{ forces on mass } i = 0 \ (i \neq 1, \ i \neq n)$$
$$\Rightarrow f_i - k_i(u_i - u_{i-1}) + k_{i+1}(u_{i+1} - u_i) = 0 \ ,$$

and finally on mass $n$,

$$\sum \text{ forces on mass } n = 0$$
$$\Rightarrow f_n - k_n(u_n - u_{n-1}) = 0 \ .$$

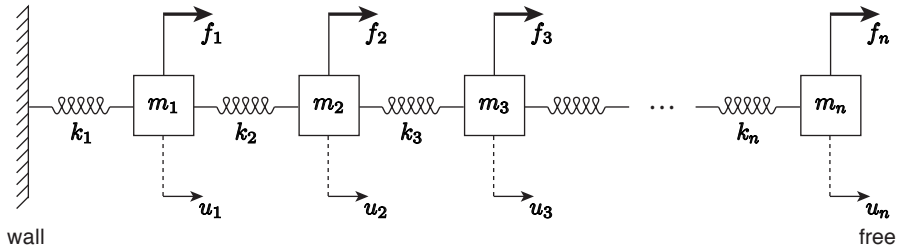



Figure 25.13: System of $n$ springs and masses.

We can write these equations as

$$\begin{array}{rrrrcl}
(k_1+k_2)u_1 & -k_2u_2 & 0\ldots & & = & f_1 \\
-k_2u_1 & +(k_2+k_3)u_2 & -k_3u_3 & 0\ldots & = & f_2 \\
0 & -k_3u_2 & +(k_3+k_4)u_3 & -k_4u_4 & = & f_3 \\
 & & \ddots & & & \vdots \\
 & & \ldots 0 & -k_nu_{n-1} \quad +k_nu_n & = & f_n
\end{array}$$

or

$$\underbrace{\begin{pmatrix}
k_1+k_2 & -k_2 & & & & \\
-k_2 & k_2+k_3 & -k_3 & & 0 & \\
 & -k_3 & k_3+k_4 & -k_4 & & \\
 & & \ddots & \ddots & \ddots & \\
 & 0 & & & -k_n & \\
 & & & -k_n & k_n &
\end{pmatrix}}_{\substack{K \\ n\times n}}
\underbrace{\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix}}_{\substack{u \\ n\times 1}}
=
\underbrace{\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix}}_{\substack{f \\ n\times 1}}$$

which is simply $Au = f$ ($A \equiv K$) but now for $n$ equations in $n$ unknowns.

In fact, the matrix $K$ has a number of special properties. In particular, $K$ is *sparse* — $K$ is mostly zero entries since only "nearest neighbor" connections affect the spring displacement and hence the force in the spring[5]; *tri-diagonal* — the nonzero entries are all on the main diagonal and diagonal just below and just above the main diagonal; *symmetric* — $K^{\mathrm{T}} = K$; and positive definite (as proven earlier for the case $n = 2$) — $\frac{1}{2}(v^{\mathrm{T}}Kv)$ is the potential/elastic energy of the system. Some of these properties are important to establish existence and uniqueness, as discussed in the next section; some of the properties are important in the efficient computational solution of $Ku = f$, as discussed in the next chapters of this unit.

[5]This sparsity property, ubiquitous in MechE systems, will be the topic of its own chapter subsequently.

## 25.4 Existence and Uniqueness: General Case (Square Systems)

We now consider a general system of $n$ equations in $n$ unknowns,

$$\underbrace{A}_{\text{given}} \quad \underbrace{u}_{\text{to find}} = \underbrace{f}_{\text{given}}$$

where $A$ is $n \times n$, $u$ is $n \times 1$, and $f$ is $n \times 1$.

If $A$ has $n$ independent columns then $A$ is non-singular, $A^{-1}$ exists, and $Au = f$ has a unique solution $u$. There are in fact many ways to confirm that $A$ is non-singular: $A$ has $n$ independent columns; $A$ has $n$ independent rows; $A$ has nonzero determinant; $A$ has no zero eigenvalues; $A$ is SPD. (We will later encounter another condition related to Gaussian elimination.) Note all these conditions are necessary and sufficient except the last: $A$ is SPD is only a sufficient condition for non-singular $A$. Conversely, if any of the necessary conditions is *not* true then $A$ is singular and $Au = f$ either will have many solutions or no solution, depending of $f$.[6] In short, all of our conclusions for $n = 2$ directly extend to the case of general $n$.

[6] Note in the computational context we must also understand and accommodate "*nearly*" singular systems. We do not discuss this more advanced topic further here.

# Chapter 26

# Gaussian Elimination and Back Substitution



## 26.1 A $2 \times 2$ System ($n = 2$)

Let us revisit the two-mass mass-spring system ($n = 2$) considered in the previous chapter; the system is reproduced in Figure 26.1 for convenience. For simplicity, we set both spring constants to unity, i.e. $k_1 = k_2 = 1$. Then, the equilibrium displacement of mass $m_1$ and $m_2$, $u_1$ and $u_2$, is described by a linear system

$$\underset{(K)}{A}\, u = f \quad \rightarrow \quad \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}, \tag{26.1}$$

where $f_1$ and $f_2$ are the forces applied to $m_1$ and $m_2$. We will use this $2 \times 2$ system to describe a systematic two-step procedure for solving a linear system: a linear solution strategy based on *Gaussian elimination* and *back substitution*. While the description of the solution strategy may appear overly detailed, we focus on presenting a systematic approach such that the approach generalizes to $n \times n$ systems and can be carried out by a computer.

By row-wise interpretation of the linear system, we arrive at a system of linear equations

$$\begin{aligned} \underset{\text{pivot}}{2}\, u_1 \; - u_2 &= f_1 \\ -1u_1 + u_2 &= f_2. \end{aligned}$$

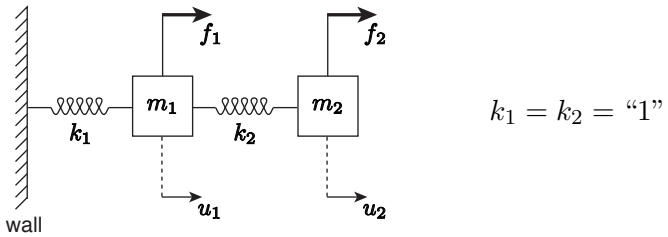



Figure 26.1: $n = 2$ spring-mass system.

We recognize that we can eliminate $u_1$ from the second equation by adding $1/2$ of the first equation to the second equation. The scaling factor required to eliminate the first coefficient from the second equation is simply deduced by diving the first coefficient of the second equation $(-1)$ by the "pivot" — the leading coefficient of the first equation (2) — and negating the sign; this systematic procedure yields $(-(-1)/2) = 1/2$ in this case. Addition of $1/2$ of the first equation to the second equation yields a new second equation

$$\begin{array}{rcl} u_1 - \frac{1}{2}u_2 & = & \frac{1}{2}f_1 \\ -u_1 + u_2 & = & f_2 \\ \hline 0u_1 + \frac{1}{2}u_2 & = & \frac{1}{2}f_1 + f_2 \end{array}.$$

Note that the solution to the linear system is unaffected by this addition procedure as we are simply adding "0" — expressed in a rather complex form — to the second equation. (More precisely, we are adding the same value to both sides of the equation.)

Collecting the new second equation with the original first equation, we can rewrite our system of linear equations as

$$\begin{aligned} 2u_1 - u_2 &= f_1 \\ 0u_1 + \frac{1}{2}u_2 &= f_2 + \frac{1}{2}f_1 \end{aligned}$$

or, in the matrix form,

$$\underbrace{\begin{pmatrix} 2 & -1 \\ 0 & \frac{1}{2} \end{pmatrix}}_{U} \underbrace{\begin{pmatrix} u_1 \\ u_2 \end{pmatrix}}_{u} = \underbrace{\begin{pmatrix} f_1 \\ f_2 + \frac{1}{2}f_1 \end{pmatrix}}_{\hat{f}}.$$

Here, we have identified the new matrix, which is *upper triangular*, by $U$ and the modified right-hand side by $\hat{f}$. In general, an upper triangular matrix has all zeros below the main diagonal, as shown in Figure 26.2; the zero entries of the matrix are shaded in blue and (possibly) nonzero entries are shaded in red. For the $2 \times 2$ case, upper triangular simply means that the $(2, 1)$ entry is zero. Using the newly introduced matrix $U$ and vector $\hat{f}$, we can concisely write our $2 \times 2$ linear system as

$$Uu = \hat{f}. \tag{26.2}$$

The key difference between the original system Eq. (26.1) and the new system Eq. (26.2) is that the new system is upper triangular; this leads to great simplification in the solution procedure as we now demonstrate.

First, note that we can find $u_2$ from the second equation in a straightforward manner, as the equation only contains one unknown. A simple manipulation yields

$$\begin{matrix} \text{eqn 2} \\ \text{of } U \end{matrix} \qquad \frac{1}{2}u_2 = f_2 + \frac{1}{2}f_1 \;\Rightarrow\; u_2 = f_1 + 2f_2$$

Having obtained the value of $u_2$, we can now treat the variable as a "known" (rather than "unknown") from hereon. In particular, the first equation now contains only one "unknown", $u_1$; again,

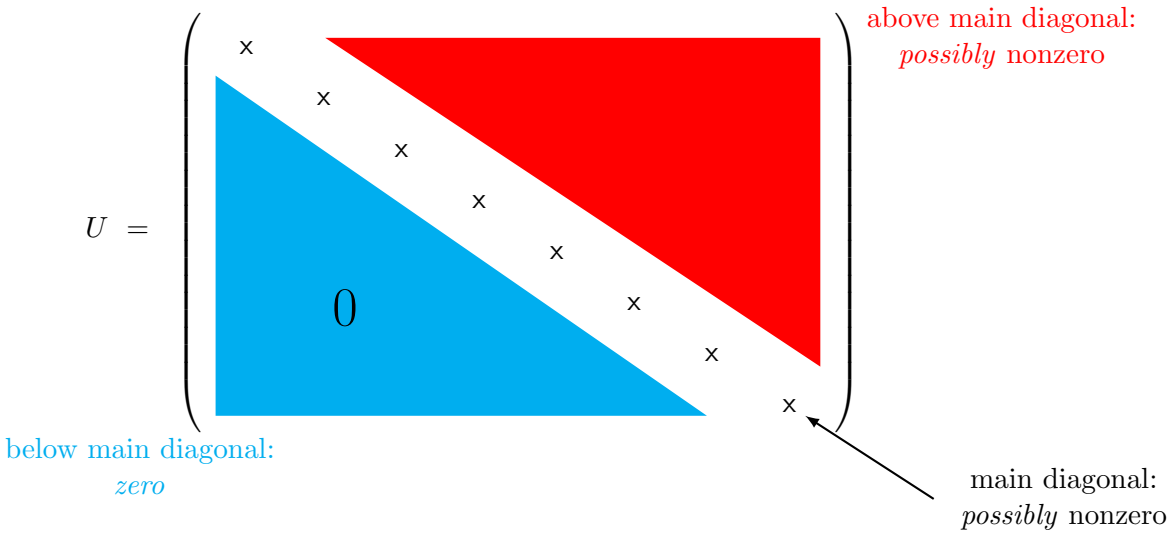



Figure 26.2: Illustration of an upper triangular matrix.

it is trivial to solve for the single unknown of a single equation, i.e.

$$\begin{aligned}
&\text{eqn 1 of } U: && 2u_1 - u_2 = f_1 \\
\Rightarrow\ & && 2u_1 = f_1 + \underbrace{u_2}_{\text{(already know)}} \\
\Rightarrow\ & && 2u_1 = f_1 + f_1 + 2f_2 = 2(f_1 + f_2) \\
\Rightarrow\ & && u_1 = (f_1 + f_2)\ .
\end{aligned}$$

Note that, even though the $2 \times 2$ linear system Eq. (26.2) is still a fully coupled system, the solution procedure for the upper triangular system is greatly simplified because we can sequentially solve (two) single-variable-single-unknown equations.

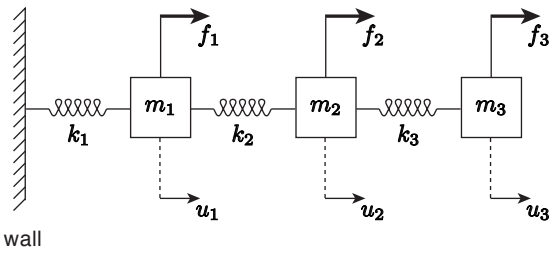
In above, we have solved a simple $2 \times 2$ system using a *systematic* two-step approach. In the first step, we reduced the original linear system into an upper triangular system; this step is called *Gaussian elimination (GE)*. In the second step, we solved the upper triangular system sequentially starting from the equation described by the last row; this step is called *back substitution (BS)*. Schematically, our linear system solution strategy is summarized by

$$\begin{cases}
\text{GE:} \quad Au = f \ \Rightarrow\ Uu = \hat{f} & \text{STEP 1} \\
\text{BS:} \quad Uu = \hat{f} \ \Rightarrow\ u & \text{STEP 2.}
\end{cases}$$

This systematic approach to solving a linear system in fact generalize to general $n \times n$ systems, as we will see shortly. Before presenting the general procedure, let us provide another concrete example using a $3 \times 3$ system.

## 26.2 A $3 \times 3$ System ($n = 3$)

Figure 26.3 shows a three-mass spring-mass system ($n = 3$). Again assuming unity-stiffness springs




$k_1 = k_2 = k_3 = 1$

Figure 26.3: A $n = 3$ spring-mass system.

for simplicity, we obtain a linear system describing the equilibrium displacement:

$$\underset{(K)}{A}\ u = f \quad \rightarrow \quad \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} .$$

As described in the previous chapter, the linear system admits a unique solution for a given $f$.

Let us now carry out Gaussian elimination to transform the system into an upper triangular system. As before, in the first step, we identify the first entry of the first row (2 in this case) as the "pivot"; we will refer to this equation containing the pivot for the current elimination step as the "pivot equation." We then add $(-(-1/2))$ of the "pivot equation" to the second equation, i.e.

$$\begin{array}{cccc} \underset{\text{pivot}}{2} & -1 & 0 & f_1 \\ -1 & 2 & -1 & f_2 \\ 0 & -1 & 1 & f_3 \end{array} \qquad \begin{array}{c} \frac{1}{2} \text{ eqn } 1 \\ +1 \text{ eqn } 2 \ , \\ \\ \end{array}$$

where the system *before* the reduction is shown on the left, and the operation to be applied is shown on the right. The operation eliminates the first coefficient (i.e. the first-column entry, or simply "column 1") of eqn 2, and reduces eqn 2 to

$$0u_1 + \frac{3}{2}u_2 - u_3 = f_2 + \frac{1}{2}f_1 \ .$$

Since column 1 of eqn 3 is already zero, we need not add the pivot equation to eqn 3. (Systematically, we may interpret this as adding $(-(0/2))$ of the pivot equation to eqn 3.) At this point, we have completed the elimination of the column 1 of eqn 2 through eqn 3 $(= n)$ by adding to each appropriately scaled pivot equations. We will refer to this partially reduced system, as "$U$-to-be"; in particular, we will denote the system that has been reduced up to (and including) the $k$-th pivot by $\tilde{U}(k)$. Because we have so far processed the first pivot, we have $\tilde{U}(k = 1)$, i.e.

$$\tilde{U}(k = 1) = \begin{pmatrix} 2 & -1 & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & -1 & 1 \end{pmatrix} .$$

In the second elimination step, we identify the modified second equation (eqn $2'$) as the "pivot equation" and proceed with the elimination of column 2 of eqn $3'$ through eqn $n'$. (In this case, we

modify only eqn $3'$ since there are only three equations.) Here the prime refers to the equations in $\tilde{U}(k=1)$. Using column 2 of the pivot equation as the pivot, we add $(-(-1/(3/2)))$ of the pivot equation to eqn $3'$, i.e.

$$\begin{array}{cccc} 2 & -1 & 0 & f_1 \\ 0 & \underset{\text{pivot}}{\frac{3}{2}} & -1 & f_2+\frac{1}{2}f_1 \\ 0 & -1 & 1 & f_3 \end{array} \qquad \begin{array}{c} \\ \frac{2}{3}\text{ eqn } 2' \\ 1\text{ eqn } 3' \end{array},$$

where, again, the system before the reduction is shown on the left, and the operation to be applied is shown on the right. The operation yields a new system,

$$\begin{array}{ccc} 2 & -1 & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & 0 & \frac{1}{3} \end{array} \qquad \begin{array}{c} f_1 \\ f_2+\frac{1}{2}f_1 \\ f_3+\frac{2}{3}f_2+\frac{1}{3}f_1 \end{array},$$

or, equivalently in the matrix form

$$\begin{pmatrix} 2 & -1 & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & 0 & \frac{1}{3} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2+\frac{1}{2}f_1 \\ f_3+\frac{2}{3}f_2+\frac{1}{3}f_1 \end{pmatrix}$$

$$U \qquad u \qquad = \qquad \hat{f},$$

which is an upper triangular system. Note that this second step of Gaussian elimination — which adds an appropriately scaled eqn $2'$ to eliminate column 3 of all the equations below it — can be reinterpreted as performing the first step of Gaussian elimination to the $(n-1)\times(n-1)$ lower sub-block of the matrix (which is $2\times 2$ in this case). This interpretation enables extension of the Gaussian elimination procedure to general $n\times n$ matrices, as we will see shortly.

Having constructed an upper triangular system, we can find the solution using the back substitution procedure. First, solving for the last variable using the last equation (i.e. solving for $u_3$ using eqn 3),

$$\underset{\text{of } U}{\text{eqn } n(=3)} \qquad \tfrac{1}{3}u_3 = f_3+\tfrac{2}{3}f_2+\tfrac{1}{3}f_1 \quad \Rightarrow \quad u_3 = 3f_3+2f_2+f_1.$$

Now treating $u_3$ as a "known", we solve for $u_2$ using the second to last equation (i.e. eqn 2),

$$\underset{\text{of } U}{\text{eqn } 2} \qquad \tfrac{3}{2}u_2 - \underset{\substack{\text{known;}\\ \text{(move to r.h.s.)}}}{u_3} = f_2+\tfrac{1}{2}f_1$$

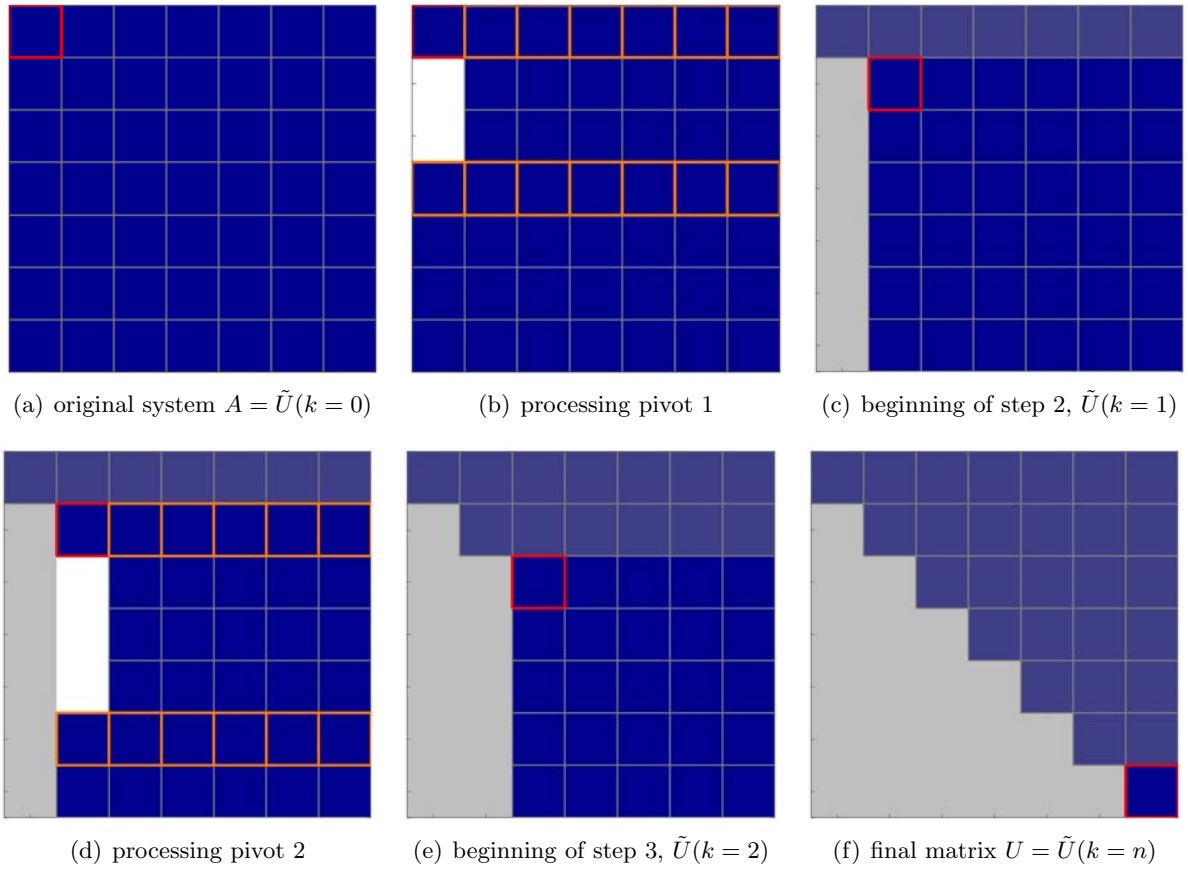
$$\tfrac{3}{2}u_2 = f_2+\tfrac{1}{2}f_1+u_3 \quad \Rightarrow \quad u_2 = 2f_2+f_1+2f_3.$$

Finally, treating $u_3$ and $u_2$ as "knowns", we solve for $u_1$ using eqn 1,

$$\underset{\text{of } U}{\text{eqn } 1} \qquad 2u_1 - \underset{\substack{\text{known;}\\ \text{(move to r.h.s.)}}}{u_2} + \underset{\substack{\text{known;}\\ \text{(move to r.h.s.)}}}{0\cdot u_3} = f_1$$

$$2u_1 = f_1+u_2\ (+0\cdot u_3) \quad \Rightarrow \quad u_1 = f_1+f_2+f_3.$$

Again, we have taken advantage of the upper triangular structure of the linear system to sequentially solve for unknowns starting from the last equation.



(a) original system $A = \tilde{U}(k=0)$ (b) processing pivot 1 (c) beginning of step 2, $\tilde{U}(k=1)$

(d) processing pivot 2 (e) beginning of step 3, $\tilde{U}(k=2)$ (f) final matrix $U = \tilde{U}(k=n)$

Figure 26.4: Illustration of Gaussian elimination applied to a $6 \times 6$ system. See the main text for a description of the colors.

## 26.3 General $n \times n$ Systems

Now let us consider a general $n \times n$ linear system. We will again use a systematic, two-step approach: Gaussian elimination and back substitution:

$$\text{STEP 1:} \quad \underset{n\times n}{A}\ \underset{n\times 1}{u} = \underset{n\times 1}{f} \quad \rightarrow \quad \underset{n\times n}{U}\ \underset{n\times 1}{u} = \underset{n\times 1}{\hat{f}} \qquad \text{(GE)}$$

$$\text{STEP 2:} \quad Uu = \hat{f} \quad \Rightarrow \quad u \qquad \text{(BS)}$$

This time, we will pay particular attention to the operation count required for each step. In addition, we will use the graphical representation shown in Figure 26.4 to facilitate the discussion. In the figure, the blue represents (in general) a nonzero entry, the white represents a zero entry, the red square represents the pivot, the orange squares identify the working rows, the shaded regions represents the rows or columns of pivots already processed, and the unshaded regions represents the rows and columns of pivots not yet processed.

As before, the first step of Gaussian elimination identifies the first equation (eqn 1) as the pivot equation and eliminates the first coefficient (column 1) of the eqn 2 through eqn $n$. To each such row, we add the appropriately scaled (determined by the ratio of the first coefficient of the row and the pivot) pivot row. We must scale (i.e. multiply) and add $n$ coefficients, so the elimination of the first coefficient requires $2n$ operations *per row*. Since there are $n-1$ rows to work on, the

total operation count for the elimination of column 1 of eqn 2 through eqn $n$ is $2n(n-1) \approx 2n^2$. Figure 26.4(b) illustrates the elimination process working on the fourth row. Figure 26.4(c) shows the partially processed matrix with zeros in the first column: $U$-to-be after the first step, i.e. $\tilde{U}(k=1)$.

In the second step, we identify the second equation as the pivot equation. The elimination of column 2 of eqn 3 through eqn $n$ requires addition of an $(n-1)$-vector — an appropriately scaled version of the pivot row of $\tilde{U}(k=1)$ — from the given row. Since there are $n-2$ rows to work on, the total operation count for the elimination of column 2 of eqn 3 through eqn $n$ is $2(n-1)(n-2) \approx 2(n-1)^2$. Note that the work required for the elimination of the second coefficient in this second step is lower than the work required for the elimination of the first coefficient in the first step because 1) we do not alter the first row (i.e. there is one less row from which to eliminate the coefficient) and 2) the first coefficient of all working rows have already been set to zero. In other word, we are working on the lower $(n-1) \times (n-1)$ sub-block of the original matrix, eliminating the first coefficient of the sub-block. This sub-block interpretation of the elimination process is clear from Figures 26.4(c) and 26.4(d); because the first pivot has already been processed, we only need to work on the unshaded area of the matrix.

In general, on the $k^{\text{th}}$ step of Gaussian elimination, we use the $k^{\text{th}}$row to remove the $k^{\text{th}}$ coefficient of eqn $k+1$ through eqn $n$, working on the $(n-k+1) \times (n-k+1)$ sub-block. Thus, the operation count for the step is $2(n-k+1)$. Summing the work required for the first to the $n^{\text{th}}$ step, the total operation count for Gaussian elimination is

$$2n^2 + 2(n-1)^2 + \cdots + 2 \cdot 3^2 + 2 \cdot 2^2 \approx \sum_{k=1}^{n} 2k^2 \approx \frac{2}{3} n^3 \text{ FLOPs} .$$

Note that the cost of Gaussian elimination grows quite rapidly with the size of the problem: as the third power of $n$. The upper-triangular final matrix, $U = \tilde{U}(k=n)$, is shown in Figure 26.4(f).

During the Gaussian elimination process, we must also construct the modified right-hand side $\hat{f}$. In eliminating the first coefficient, we modify the right-hand side of eqn 2 through eqn $n$ ($n-1$ equations), each requiring two operations for multiplication and addition, resulting in $2(n-1) \approx 2n$ total operations. In general, the $k^{\text{th}}$ step of Gaussian elimination requires modification of the $(n-k)$-sub-vector on the right-hand side. Thus, the total operation count for the construction of the right-hand side is

$$2n + 2(n-1) + \cdots + 2 \cdot 3 + 2 \cdot 2 \approx \sum_{k=1}^{n} 2k \approx n^2 \text{ FLOPs} .$$

As the cost for constructing the modified right-hand side scales as $n^2$, it becomes insignificant compared to $2n^3/3$ operations required for the matrix manipulation for a large $n$. Thus, we conclude that the total cost of Gaussian elimination, including the construction of the modified right-hand side, is $2n^3/3$.

Now let us consider the operation count of back substitution. Recall that the $n \times n$ upper

triangular system takes the form

$$\begin{pmatrix} U_{11} & U_{12} & \cdots & \cdots & U_{1n} \\ & U_{22} & & & U_{2n} \\ & & \ddots & & \vdots \\ & 0 & & U_{n-1\,n-1} & U_{n-1\,n} \\ & & & & U_{n\,n} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} \hat{f}_1 \\ \hat{f}_2 \\ \vdots \\ \hat{f}_{n-1} \\ \hat{f}_n \end{pmatrix} .$$

We proceed to solve for the unknowns $u_1, \ldots, u_n$ starting from the last unknown $u_n$ using the $n^{\text{th}}$ equation and sequentially solving for $u_{n-1}, \ldots, u_1$ in that order. Schematically, the solution process takes the form

$$\begin{aligned} &\text{eqn } n\text{:} && U_{n\,n}u_n - \hat{f}_n \;\Rightarrow\; u_n = \frac{\hat{f}_n}{U_{n\,n}} \\ &\text{eqn } n-1\text{:} && U_{n-1\,n-1}u_{n-1} + U_{n-1\,n}u_n = \hat{f}_{n-1} \\ & && \Downarrow \\ & && U_{n-1\,n-1}u_{n-1} = \hat{f}_{n-1} - U_{n-1\,n-1}u_{n-1} \;\Rightarrow\; u_{n-1} \\ &\vdots \\ &\text{eqn } 1\text{:} && U_{11}u_1 + U_{12}u_2 + \cdots + U_{1\,n}u_n = \hat{f}_1 \\ & && \Downarrow \\ & && U_{11}u_1 = \hat{f}_1 - U_{12}u_2 - \cdots - U_{1\,n}u_n \;\Rightarrow\; u_1 \; . \end{aligned}$$

Solving for $u_n$ requires one operation. Solving for $u_{n-1}$ requires one multiplication-subtraction pair (two operations) and one division. In general, solving for $u_k$ requires $(n-k)$ multiplication-subtraction pairs ($2(n-k)$ operations) and one division. Summing all the operations, the total operation count for back substitution is

$$1 + (1+2) + (1+2\cdot 2) + \cdots + (1+2(n-1)) \approx \sum_{k=1}^{N} 2k \approx n^2 \,\text{FLOPs} \; .$$

Note that the cost for the back substitution step scales as the second power of the problem size $n$; thus, the cost of back substitution becomes negligible compared to that of Gaussian elimination for a large $n$.

## 26.4 Gaussian Elimination and LU Factorization

In this chapter, we introduced a systematic procedure for solving a linear system using Gaussian elimination and back substitution. We interpreted Gaussian elimination as a process of triangulating the system matrix of interest; the process relied, in the $k^{\text{th}}$ step, on adding appropriately scaled versions of the $k^{\text{th}}$ equation to all the equations below it in order to eliminate the leading

coefficient. In particular, we also modified the right-hand side of the equation in the triangulation procedure such that we are adding the same quantity to both sides of the equation and hence not affecting the solution. The end product of our triangulation process is an upper triangular matrix $U$ and a modified right-hand side $\hat{f}$. If we are given a new right-hand side, we would have to repeat the same procedure again (in $\mathcal{O}(n^3)$ cost) to deduce the appropriate new modified right-hand side.

It turns out that a slight modification of our Gaussian elimination procedure in fact would permit solution to the problem with a different right-hand side in $\mathcal{O}(n^2)$ operations. To achieve this, instead of modifying the right-hand side in the upper triangulation process, we *record the operations used in the upper triangulation process with which we generated the right-hand side.* It turns out that this recording operation in fact can be done using a lower triangular matrix $L$, such that the modified right-hand side $\hat{f}$ is the solution to

$$L\hat{f} = f, \tag{26.3}$$

where $f$ is the original right-hand side. Similar to back substitution for an upper triangular system, *forward substitution* enables solution to the lower triangular system in $\mathcal{O}(n^2)$ operations. This lower triangular matrix $L$ that records all operations used in transforming matrix $A$ into $U$ in fact is a matrix that satisfies

$$A = LU \ .$$

In other words, the matrices $L$ and $U$ arise from a *factorization* of the matrix $A$ into lower and upper triangular matrices.

This procedure is called *LU factorization.* (The fact that $L$ and $U$ must permit such a factorization is straightforward to see from the fact that $Uu = \hat{f}$ and $L\hat{f} = f$; multiplication of both sides of $Uu = \hat{f}$ by $L$ yields $LUu = L\hat{f} = f$, and because the relationship must hold for any solution-right-hand-side pair $\{u, f\}$ to $Au = f$, it must be that $LU = A$.) The factorization process is in fact identical to our Gaussian elimination and requires $2n^3/3$ operations. Note we did compute all the pieces of the matrix $L$ in our elimination procedure; we simply did not form the matrix for simplicity.

In general the LU decomposition will exist if the matrix $A$ is non-singular. There is, however, one twist: we may need to permute the rows of $A$ — a process known as (partial) pivoting — in order to avoid a zero pivot which would prematurely terminate the process. (In fact, permutations of rows can also be advantageous to avoid small pivots which can lead to amplification of round-off errors.) If even — say in infinite precision — with row permutations we arrive at an exactly zero pivot then this is in fact demonstrates that $A$ is singular.[1]

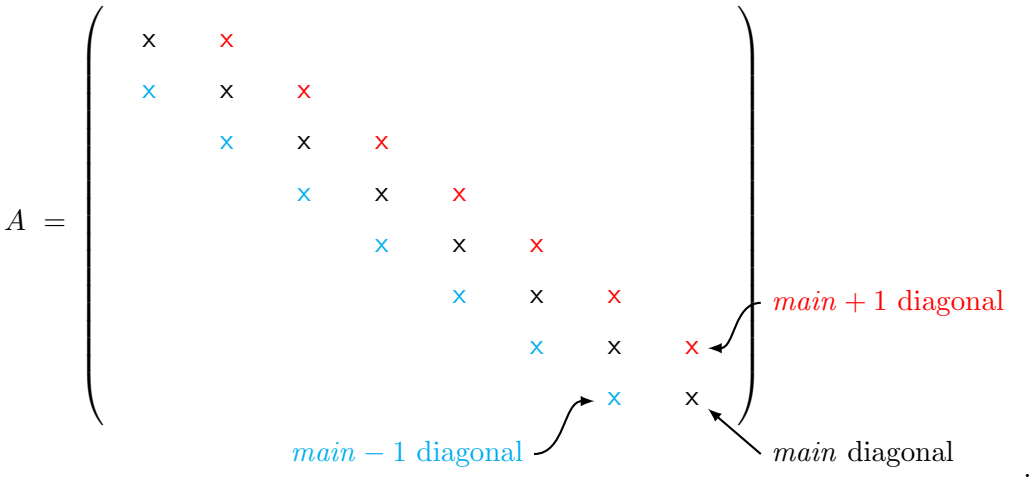
There are some matrices for which no pivoting is required. One such important example in mechanical engineering is SPD matrices. For an SPD matrix (which is certainly non-singular — all eigenvalues are positive) we will never arrive at a zero pivot nor we will need to permute rows to ensure stability. Note, however, that we may still wish to permute rows to improve the efficiency of the LU decomposition for sparse systems — which is the topic of the next section.

## 26.5 Tridiagonal Systems

While the cost of Gaussian elimination scales as $n^3$ for a general $n \times n$ linear system, there are instances in which the scaling is much weaker and hence the computational cost for a large problem

[1]The latter is a more practical test for singularity of $A$ than say the determinant of $A$ or the eigenvalues of $A$, however typically singularity of "mechanical engineering" $A$ matrices are not due to devious cancellations but rather due to upfront modeling errors — which are best noted and corrected prior to LU decomposition.

is relatively low. A *tridiagonal system* is one such example. A tridigonal system is characterized by having nonzero entries only along the main diagonal and the immediate upper and lower diagonal, i.e.




The immediate upper diagonal is called the *super-diagonal* and the immediate lower diagonal is called the *sub-diagonal*. A significant reduction in the computational cost is achieved by taking advantage of the *sparsity* of the tridiagonal matrix. That is, we omit addition and multiplication of a large number of zeros present in the matrix.

Let us apply Gaussian elimination to the $n \times n$ tridiagonal matrix. In the first step, we compute the scaling factor (one FLOP), scale the second entry of the coefficient of the first row by the scaling factor (one FLOP), and add that to the second coefficient of the second row (one FLOP). (Note that we do not need to scale and add the first coefficient of the first equation to that of the second equation because we know it will vanish by construction.) We do not have to add the first equation to any other equations because the first coefficient of all other equations are zero. Moreover, note that the addition of the (scaled) first row to the second row does not introduce any new nonzero entry in the second row. Thus, the updated matrix has zeros above the super-diagonal and *retains the tridiagonal structure of the original matrix* (with the (2,1) entry eliminated, of course); in particular, the updated $(n-1) \times (n-1)$ sub-block is again tridiagonal. We also modify the right-hand side by multiplying the first entry by the scaling factor (one FLOP) and adding it to the second entry (one FLOP). Combined with the three FLOPs required for the matrix manipulation, the total cost for the first step is five FLOPs.

Similarly, in the second step, we use the second equation to eliminate the leading nonzero coefficient of the third equation. Because the structure of the problem is identical to the first one, this also requires five FLOPs. The updated matrix retain the tridiagonal structure in this elimination step and, in particular, the updated $(n-2) \times (n-2)$ sub-block is tridiagonal. Repeating the operation for $n$ steps, the total cost for producing an upper triangular system (and the associated modified right-hand side) is $5n$ FLOPs. Note that the cost of Gaussian elimination for a tridiagonal system scales *linearly* with the problem size $n$: a dramatic improvement compared to $\mathcal{O}(n^3)$ operations required for a general case.

At this point, we have produced an upper triangular system of the form

$$\begin{pmatrix} \times & \times & & & & & & \\ & \times & \times & & & & 0 & \\ & & \times & \times & & & & \\ & & & \times & \times & & & \\ & & & & \times & \times & & \\ & & 0 & & & \times & \times & \\ & & & & & & \times & \times \\ & & & & & & & \times \end{pmatrix} \begin{pmatrix} u_1 \\ \text{3 FLOPs} \\ u_2 \\ \vdots \\ u_{n-2} \\ \text{3 FLOPs} \\ u_{n-1} \\ \text{3 FLOPs} \\ u_n \\ \text{1 FLOP} \end{pmatrix} = \begin{pmatrix} \hat{f}_1 \\ \hat{f}_1 \\ \vdots \\ \hat{f}_{n-2} \\ \hat{f}_{n-1} \\ \hat{f}_n \end{pmatrix}.$$

The system is said to be *bidiagonal* — or more precisely upper bidiagonal — as nonzero entries appear only on the main diagonal and the super-diagonal. (A matrix that has nonzero entries only on the main and sub-diagonal are also said to be bidiagonal; in this case, it would be lower bidiagonal.)

In the back substitution stage, we can again take advantage of the sparsity — in particular the bidiagonal structure — of our upper triangular system. As before, evaluation of $u_n$ requires a simple division (one FLOP). The evaluation of $u_{n-1}$ requires one scaled subtraction of $u_n$ from the right-hand side (two FLOPs) and one division (one FLOP) for three total FLOPs. The structure is the same for the remaining $n-2$ unknowns; the evaluating each entry takes three FLOPs. Thus, the total cost of back substitution for a bidiagonal matrix is $3n$ FLOPs. Combined with the cost of the Gaussian elimination for the tridiagonal matrix, the overall cost for solving a tridiagonal system is $8n$ FLOPs. Thus, the operation count of the entire linear solution procedure (Gaussian elimination and back substitution) scales linearly with the problem size for tridiagonal matrices.

We have achieved a significant reduction in computational cost for a tridiagonal system compared to a general case by taking advantage of the sparsity structure. In particular, the computational cost has been reduced from $2n^3/3$ to $8n$. For example, if we wish to solve for the equilibrium displacement of a $n = 1000$ spring-mass system (which yields a tridiagonal system), we have reduced the number of operations from an order of a billion to a thousand. In fact, with the tridiagonal-matrix algorithm that takes advantage of the sparsity pattern, we can easily solve a spring-mass system with millions of unknowns on a desktop machine; this would certainly not be the case if the general Gaussian elimination algorithm is employed, which would require $\mathcal{O}(10^{18})$ operations.

While many problems in engineering require solution of a linear system with millions (or even billions) of unknowns, these systems are typically sparse. (While these systems are rarely tridiagonal, most of the entries of these matrices are zero nevertheless.) In the next chapter, we consider solution to more general sparse linear systems; just as we observed in this tridiagonal case, the key to reducing the computational cost for large sparse matrices — and hence making the computation tractable — is to study the nonzero pattern of the sparse matrix and design an algorithm that does not execute unnecessary operations.

# Chapter 27

# Gaussian Elimination: Sparse Matrices



In the previous chapter, we observed that the number of floating point operations required to solve a $n \times n$ tridiagonal system scales as $\mathcal{O}(n)$ whereas that for a general (dense) $n \times n$ system scales as $\mathcal{O}(n^3)$. We achieved this significant reduction in operation count by taking advantage of the sparsity of the matrix. In this chapter, we will consider solution of more general sparse linear systems.

## 27.1 Banded Matrices

A class of sparse matrices that often arise in engineering practice — especially in continuum mechanics — is the banded matrix. An example of banded matrix is shown in Figure 27.1. As the figure shows, the nonzero entries of a banded matrix is confined to within $m_\mathrm{b}$ entries of the main diagonal. More precisely,

$$A_{ij} = 0, \quad \text{for} \quad j > i + m_\mathrm{b} \quad \text{or} \quad j < i - m_\mathrm{b},$$

and $A$ may take on any value within the band (including zero). The variable $m_\mathrm{b}$ is referred to as the *bandwidth*. Note that the number of nonzero entries in a $n \times n$ banded matrix with a bandwidth $m_\mathrm{b}$ is less than $n(2m_\mathrm{b} + 1)$.

Let us consider a few different types of banded matrices.

$$\underset{n\times n}{A}\ \underset{n\times 1}{u} = \underset{n\times 1}{f}, \quad A = \begin{array}{c} \\ m_\mathrm{b} \\ \\ \end{array}\!\!\begin{array}{c} \overbrace{\phantom{xxxx}}^{m_\mathrm{b}} \\ \\ \\ \end{array}\!\!\begin{pmatrix} \diagdown\!\diagdown & & & 0 \\ \diagdown\!\diagdown & \diagdown\!\diagdown & & \\ & \diagdown\!\diagdown & \diagdown\!\diagdown & \\ 0 & & \diagdown\!\diagdown & \diagdown\!\diagdown \end{pmatrix}$$

Figure 27.1: A banded matrix with bandwidth $m_\mathrm{b}$.

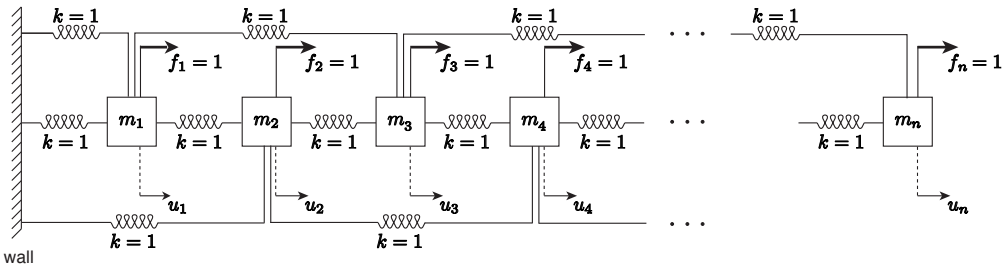



Figure 27.2: A spring-mass system whose equilibrium state calculation gives rise to a pentadiagonal matrix.

**Example 27.1.1** ***Tri*diagonal matrix:** $m_\mathrm{b} = 1$
As we have discussed in the previous two chapters, tridiagonal matrices have nonzero entries only along the main diagonal, sub-diagonal, and super-diagonal. Pictorially, a tridiagonal matrix takes the following form:

$$\text{main diagonal, main } \pm 1 \text{ diagonals} \quad \begin{matrix} & & 0 \\ & \iddots & \\ 0 & & \end{matrix} \ .$$

Clearly the bandwidth of a tridiagonal matrix is $m_\mathrm{b} = 1$. A $n \times n$ tridiagonal matrix arise from, for example, computing the equilibrium displacement of $n$ masses connected by springs, as we have seen in previous chapters.

---

**Example 27.1.2** ***Penta*diagonal matrix:** $m_\mathrm{b} = 2$
As the name suggests, a pentadiagonal matrix is characterized by having nonzero entries along the main diagonal and the two diagonals above and below it, for the total of five diagonals. Pictorially, a pentadigonal matrix takes the following form:

$$\text{main diagonal, main } \pm 1, \pm 2 \text{ diagonals} \quad \begin{matrix} & & 0 \\ & \iddots & \\ 0 & & \end{matrix} \ .$$

The bandwidth of a pentadiagonal matrix is $m_\mathrm{b} = 2$. A $n \times n$ pentadiagonal matrix arise from, for example, computing the equilibrium displacement of $n$ masses each of which is connected to not only the immediate neighbor but also to the neighbor of the neighbors. An example of such a system is shown in Figure 27.2.

---

**Example 27.1.3 "Outrigger" matrix**
Another important type of banded matrix is a matrix whose zero entries are confined to within the $m_\mathrm{b}$ band of the main diagonal but for which a large number of entries between the main diagonal and the most outer band is zero. We will refer to such a matrix as "outrigger." An example of such a matrix is

$$\text{"outrigger"} \quad \begin{pmatrix} & \overbrace{\quad\quad}^{m_\mathrm{b}} & \\ & \iddots & 0 \\ 0 & & \ddots \\ & 0 & \end{pmatrix} \ .$$

In this example, there are five nonzero diagonal bands, but the two outer bands are located far from the middle three bands. The bandwidth of the matrix, $m_{\mathrm{b}}$, is specified by the location of the outer diagonals. (Note that this is *not* a pentadiagonal matrix since the nonzero entries are not confined to within $m_{\mathrm{b}} = 2$.) "Outrigger" matrices often arise from finite difference (or finite element) discretization of partial differential equations in two or higher dimensions.

---

## 27.2 Matrix-Vector Multiplications

To introduce the concept of sparse operations, let us first consider multiplication of a $n \times n$ sparse matrix with a (dense) $n$-vector. Recall that matrix-vector multiplication may be interpreted row-wise or column-wise. In row-wise interpretation, we consider the task of computing $w = Av$ as performing $n$ inner products, one for each entry of $w$, i.e.

$$w_i = \begin{pmatrix} A_{i1} & A_{i2} & \dots & A_{in} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}, \quad i = 1, \dots, n.$$

If the matrix $A$ is dense, the $n$ inner products of $n$-vectors requires $n \cdot (2n) = 2n^2$ FLOPs. However, if the matrix $A$ is sparse, then each row of $A$ contains few nonzero entries; thus, we may skip a large number of trivial multiplications in our inner products. In particular, the operation count for the inner product of a sparse $n$-vector with a dense $n$-vector is equal to twice the number of nonzero entries in the sparse vector. Thus, the operation count for the entire matrix-vector multiplication is equal to twice the number of nonzero entries in the matrix, i.e. $2 \cdot \mathrm{nnz}(A)$, where $\mathrm{nnz}(A)$ is the number of nonzero entries in $A$. This agrees with our intuition because the matrix-vector product requires simply visiting each nonzero entry of $A$, identifying the appropriate multiplier in $v$ based on the column index, and adding the product to the appropriate entry of $w$ based on the row index.

Now let us consider a column interpretation of matrix-vector multiplication. In this case, we interpret $w = Av$ as

$$\begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = v_1 \begin{pmatrix} A_{11} \\ A_{21} \\ \vdots \\ A_{n1} \end{pmatrix} + v_2 \begin{pmatrix} A_{12} \\ A_{22} \\ \vdots \\ A_{n2} \end{pmatrix} + \cdots + v_n \begin{pmatrix} A_{1n} \\ A_{2n} \\ \vdots \\ A_{nn} \end{pmatrix}.$$

If $A$ is sparse then, each column of $A$ contains few nonzero entries. Thus, for each column we simply need to scale these few nonzero entries by the appropriate entry of $v$ and augment the corresponding entries of $w$; the operation count is twice the number of nonzero entries in the column. Repeating the operation for all columns of $A$, the operation count for the entire matrix-vector multiplication is again $2 \cdot \mathrm{nnz}(A)$.

Because the number of nonzero entries in a sparse matrix is (by definition) $\mathcal{O}(n)$, the operation count for sparse matrix-vector product is $2 \cdot \mathrm{nnz}(A) \sim \mathcal{O}(n)$. For example, for a banded matrix with a bandwidth $m_{\mathrm{b}}$, the operation count is at most $2n(2m_{\mathrm{b}} + 1)$. Thus, we achieve a significant reduction in the operation count compared to dense matrix-vector multiplication, which requires $2n^2$ operations.

## 27.3 Gaussian Elimination and Back Substitution

### 27.3.1 Gaussian Elimination

We now consider the operation count associated with solving a sparse linear system $Au = f$ using Gaussian elimination and back substitution introduced in the previous chapter. Recall that the Gaussian elimination is a process of turning a linear system into an upper triangular system, i.e.

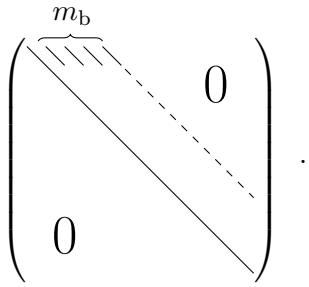
$$\text{STEP 1: } Au = f \rightarrow \underset{\substack{(n\times n)\\ \text{upper}\\ \text{triangular}}}{U} \quad u = \hat{f} \quad .$$

For a $n \times n$ dense matrix, Gaussian elimination requires approximately $\frac{2}{3}n^3$ FLOPs.

#### Densely-Populated Banded Systems

Now, let us consider a $n \times n$ banded matrix with a bandwidth $m_\text{b}$. To analyze the worst case, we assume that all entries within the band are nonzero. In the first step of Gaussian elimination, we identify the first row as the "pivot row" and eliminate the first entry (column 1) of the first $m_\text{b}$ rows by adding appropriately scaled pivot row; column 1 of rows $m_\text{b} + 2, \ldots, n$ are already zero. Elimination of column 1 of a given row requires addition of scaled $m_\text{b} + 1$ entries of the pivot row, which requires $2(m_\text{b} + 1)$ operations. Applying the operation to $m_\text{b}$ rows, the operation count for the first step is approximately $2(m_\text{b} + 1)^2$. Note that because the nonzero entries of the pivot row is confined to the first $m_\text{b} + 1$ entries, addition of the scaled pivot row to the first $m_\text{b} + 1$ rows does not increase the bandwidth of the system (since these rows already have nonzero entries in these columns). In particular, the sparsity pattern of the upper part of $A$ is unaltered in the process.

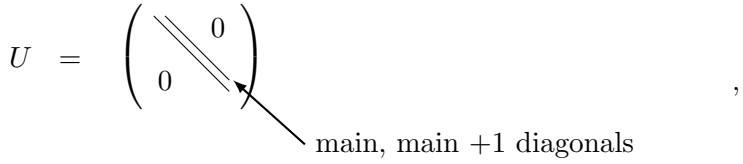
The second step of Gaussian elimination may be interpreted as applying the first step of Gaussian elimination to $(n - 1) \times (n - 1)$ submatrix, which itself is a banded matrix with a bandwidth $m_\text{b}$ (as the first step does not alter the bandwidth of the matrix). Thus, the second elimination step also requires approximately $2(m_\text{b}+1)^2$ FLOPs. Repeating the operation for all $n$ pivots of the matrix, the total operation count for Gaussian elimination is approximately $2n(m_\text{b} + 1)^2 \sim \mathcal{O}(n)$. The final upper triangular matrix $U$ takes the following form:




.

The upper triangular matrix has approximately $n(m_\text{b} + 1) \sim \mathcal{O}(n)$ nonzero entries. Both the operation count and the number of nonzero in the final upper triangular matrix are $\mathcal{O}(n)$, compared to $\mathcal{O}(n^3)$ operations and $\mathcal{O}(n^2)$ entries for a dense system. (We assume here $m_\text{b}$ is fixed independent of $n$.)

In particular, as discussed in the previous chapter, Gaussian elimination of a tridiagonal matrix yields an upper bidiagonal matrix




,

in approximately $5n$ operations (including the formation of the modified right-hand side $\hat{f}$). Similarly, Gaussian elimination of a pentadiagonal system results in an upper triangular matrix of the form

$$U = \begin{pmatrix} & & & 0 \\ & & & \\ & & & \\ 0 & & & \end{pmatrix},$$

main, main +1, +2 diagonals

and requires approximately $14n$ operations.

### "Outrigger" Systems: Fill-Ins

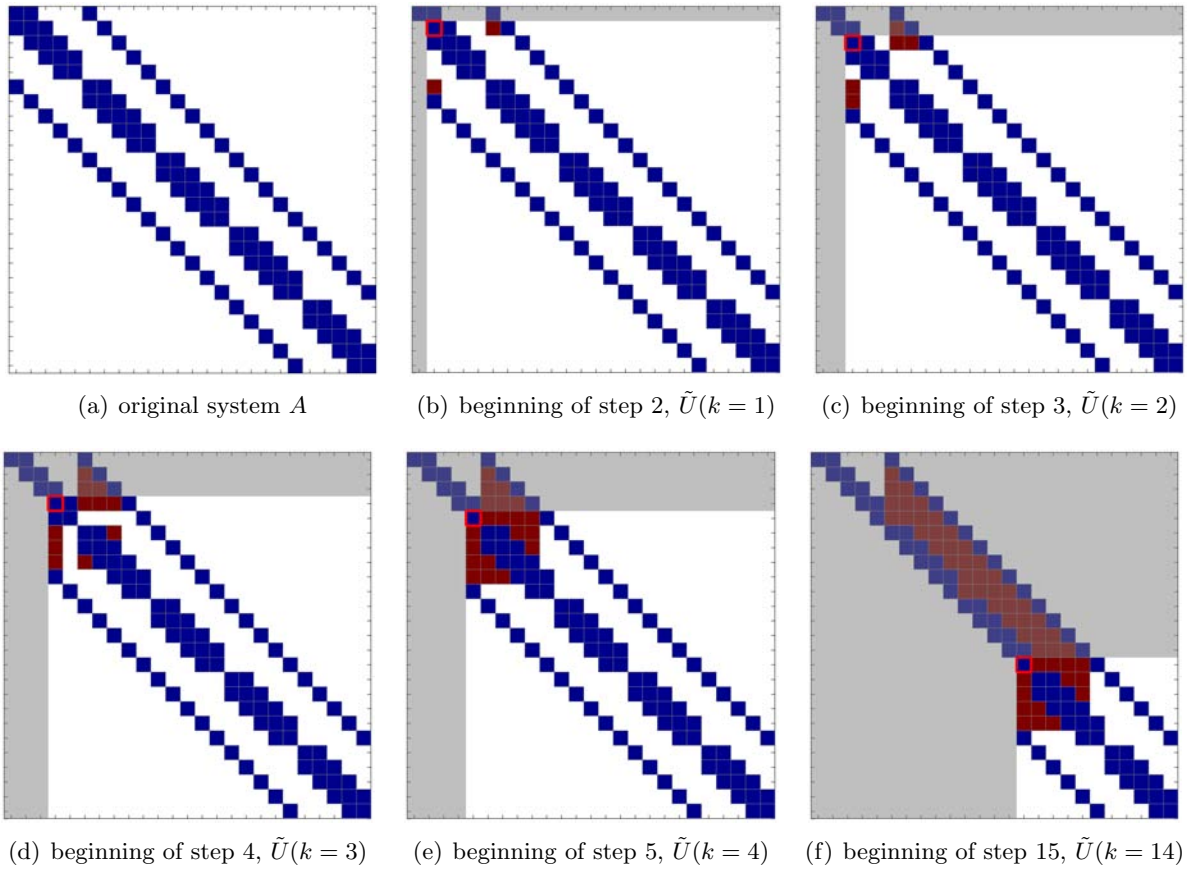
Now let us consider application of Gaussian elimination to an "outrigger" system. First, because a $n \times n$ "outrigger" system with a bandwidth $m_b$ is a special case of a "densely-populated banded" system with a bandwidth $m_b$ considered above, we know that the operation count for Gaussian elimination is at most $n(m_b + 1)^2$ and the number of nonzero in the upper triangular matrix is at most $n(m_b + 1)$. In addition, due to a large number of zero entries between the outer bands of the matrix, we *hope* that the operation count and the number of nonzero are less than those for the "densely-populated banded" case. Unfortunately, inspection of the Gaussian elimination procedure reveals that this reduction in the cost and storage is not achieved in general.

The inability to reduce the operation count is due to the introduction of "fill-ins": the entries of the sparse matrix that are originally zero but becomes nonzero in the Gaussian elimination process. The introduction of fill-ins is best described graphically. Figure 27.3 shows a sequence of matrices generated through Gaussian elimination of a $25 \times 25$ outrigger system. In the subsequent figures, we color code entries of partially processed $U$: the shaded area represents rows *or* columns of pivots already processed; the unshaded area represents the rows *and* columns of pivots not yet processed; the blue represent initial nonzeros in $A$ which remain nonzeros in $U$-to-be; and the red are initial zeros of $A$ which become nonzero in $U$-to-be, i.e. fill-ins.

As Figure 27.3(a) shows, the bandwidth of the original matrix is $m_b = 5$. (The values of the entries are hidden in the figure as they are not important in this discussion of fill-ins.) In the first elimination step, we first eliminate column 1 of row 2 by adding an appropriately scaled row 1 to the row. While we succeed in eliminating column 1 of row 2, note that we introduce a nonzero element in column 6 of row 2 as column 6 of row 1 "falls" to row 2 in the elimination process. This nonzero element is called a "fill-in." Similarly, in eliminating column 1 of row 6, we introduce a "fill-in" in column 2 as column 2 of row 1 "falls" to row 6 in the elimination process. Thus, we have introduced two fill-ins in this first elimination step as shown in Figure 27.3(b): one in the upper part and another in the lower part.

Now, consider the second step of elimination starting from Figure 27.3(b). We first eliminate column 2 of row 3 by adding appropriately scaled row 2 to row 3. This time, we introduce fill in not only from column 7 of row 2 "falling" to row 3, but also from column 6 of row 2 "falling" to row 3. Note that the latter is in fact a fill-in introduced in the first step. In general, once a fill-in is introduced in the upper part, *the fill-in propagates from one step to the next, introducing further fill-ins* as it "falls" through. Next, we need to eliminate column 2 of row 6; this entry was zero in the original matrix but was filled in the first elimination step. Thus, fill-in introduced in the lower part *increases the number of rows whose leading entry must be eliminated in the upper-triangulation process.* The matrix after the second step is shown in Figure 27.3(c). Note that the number of fill-in continue to increase, and we begin to lose the zero entries between the outer bands of the outrigger system.

As shown in Figure 27.3(e), by the beginning of the fifth elimination step, the "outrigger"

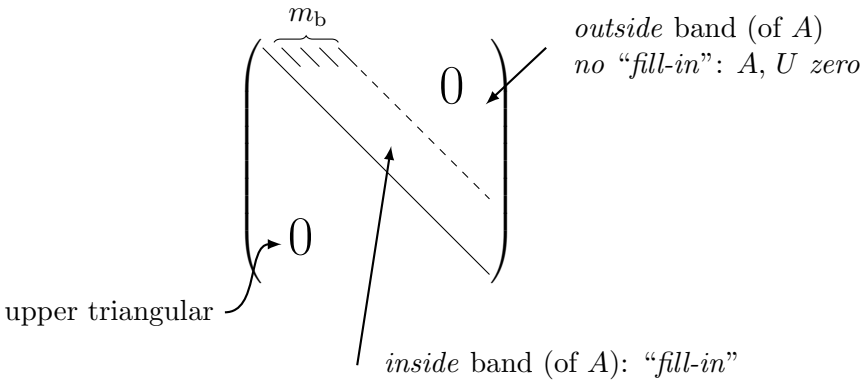


(a) original system $A$ (b) beginning of step 2, $\tilde{U}(k=1)$ (c) beginning of step 3, $\tilde{U}(k=2)$

(d) beginning of step 4, $\tilde{U}(k=3)$ (e) beginning of step 5, $\tilde{U}(k=4)$ (f) beginning of step 15, $\tilde{U}(k=14)$

Figure 27.3: Illustration of Gaussian elimination applied to a $25 \times 25$ "outrigger" system. The blue entries are the entries present in the original system, and the red entries are "fill-in" introduced in the factorization process. The pivot for each step is marked by a red square.

system has largely lost its sparse structure in the leading $(m_\mathrm{b}+1)\times(m_\mathrm{b}+1)$ subblock of the working submatrix. Thus, for the subsequent $n-m_\mathrm{b}$ steps of Gaussian elimination, each step takes $2m_\mathrm{b}^2$ FLOPs, which is approximately the same number of operations as the densely-populated banded case. Thus, the total number of operations required for Gaussian elimination of an outrigger system is approximately $2n(m_\mathrm{b}+1)^2$, the same as the densely-populated banded case. The final matrix takes the form:




Note that the number of nonzero entries is approximately $n(m_\mathrm{b}+1)$, which is much larger than the number of nonzero entries in the original "outrigger" system.

The "outrigger" system, such as the one considered above, naturally arise when a partial differential equation (PDE) is discretized in two or higher dimensions using a finite difference or finite element formulation. An example of such a PDE is the heat equation, describing, for example, the equilibrium temperature of a thermal system shown in Figure 27.4. With a natural ordering of the degrees of freedom of the discretized system, the bandwidth $m_\mathrm{b}$ is equal to the number of grid points in one coordinate direction, and the number of degrees of freedom of the linear system is $n=m_\mathrm{b}^2$ (i.e. product of the number of grid points in two coordinate directions). In other words, the bandwidth is the square root of the matrix size, i.e. $m_\mathrm{b}=n^{1/2}$. Due to the outrigger structure of the resulting system, factorizing the system requires approximately $n(m_\mathrm{b}+1)^2\approx n^2$ FLOPs. This is in contrast to one-dimensional case, which yields a tridiagonal system, which can be solved in $\mathcal{O}(n)$ operations. In fact, in three dimensions, the bandwidth is equal to the product of the number of grid points in two coordinate directions, i.e. $m_\mathrm{b}=(n^{1/3})^2=n^{2/3}$. The number of operations required for factorization is $n(m_\mathrm{b}+1)^2\approx n^{7/3}$. Thus, the cost of solving a PDE is significantly higher in three dimensions than in one dimension *even if both discretized systems had the same number of unknowns.*[1]

### 27.3.2 Back Substitution

Having analyzed the operation count for Gaussian elimination, let us inspect the operation count for back substitution. First, recall that back substitution is a process of finding the solution of an upper triangular system, i.e.

$$\text{STEP 2: } Uu=\hat{f}\rightarrow u \quad .$$

Furthermore, recall that the operation count for back substitution is equal to twice the number of nonzero entries in $U$. Because the matrix $U$ is unaltered, we can simply count the number of

[1] Not unknowns per dimension, but the total number of unknowns.

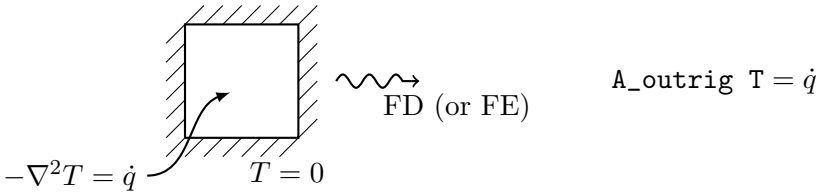



Figure 27.4: Heat equation in two dimensions. Discretization of the equation by finite difference (FD) or finite element (FE) method yields an "outrigger" system.

nonzero entries in the $U$ that we obtain after Gaussian elimination; there is nothing equivalent to "fill-in" — modifications to the matrix that increases the number of entries in the matrix and hence the operation count — in back substitution.

#### Densely-Populated Banded Systems

For a densely-populated banded system with a bandwidth $m_\text{b}$, the number of unknowns in the factorized matrix $U$ is approximately equal to $n(m_\text{b}+1)$. Thus, back substitution requires approximately $2n(m_\text{b}+1)$ FLOPs. In particular, back substitution for a tridiagonal system (which yields an upper bidiagonal $U$) requires approximately $3n$ FLOPs. A pentadiagonal system requires approximately $5n$ FLOPs.

#### "Outrigger"

As discussed above, a $n \times n$ outrigger matrix of bandwidth $m_\text{b}$ produces an upper triangular matrix $U$ whose entries between the main diagonal and the outer band are nonzero due to fill-ins. Thus, the number of nonzeros in $U$ is approximately $n(m_\text{b}+1)$, and the operation count for back substitution is approximately $2n(m_\text{b}+1)$. (Note in particular that even if an outrigger system only have five bands (as in the one shown in Figure 27.3), the number of operations for back substitution is $2n(m_\text{b}+1)$ and *not* $5n$.)

*Begin Advanced Material*

## 27.4 Fill-in and Reordering

The previous section focused on the computational cost of solving a linear system governed by banded sparse matrices. This section introduces a few additional sparse matrices and also discussed additional concepts on Gaussian elimination for sparse systems.

### 27.4.1 A Cyclic System

First, let us show that a small change in a physical system — and hence the corresponding linear system $A$ — can make a large difference in the sparsity pattern of the factored matrix $U$. Here, we consider a modified version of $n$-mass mass-spring system, where the first mass is connected to the last mass, as shown in Figure 27.5. We will refer to this system as a "cyclic" system, as the springs form a circle. Recall that a spring-mass system without the extra connection yields a tridiagonal system. With the extra connection between the first and the last mass, now the $(1,n)$ entry and $(n,1)$ entry of the matrix are nonzero as shown in Figure 27.6(a) (for $n = 25$); clearly, the matrix

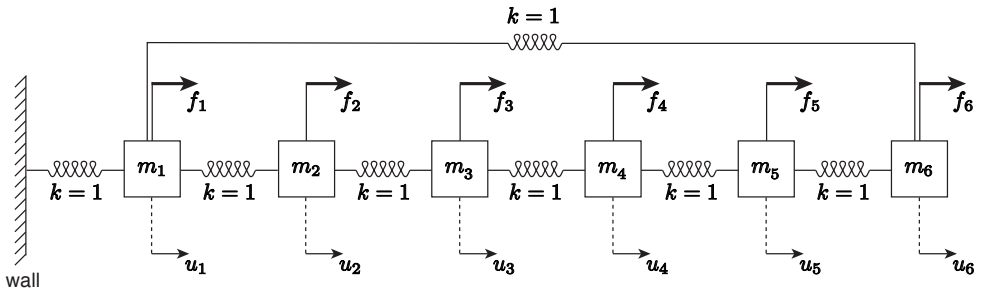
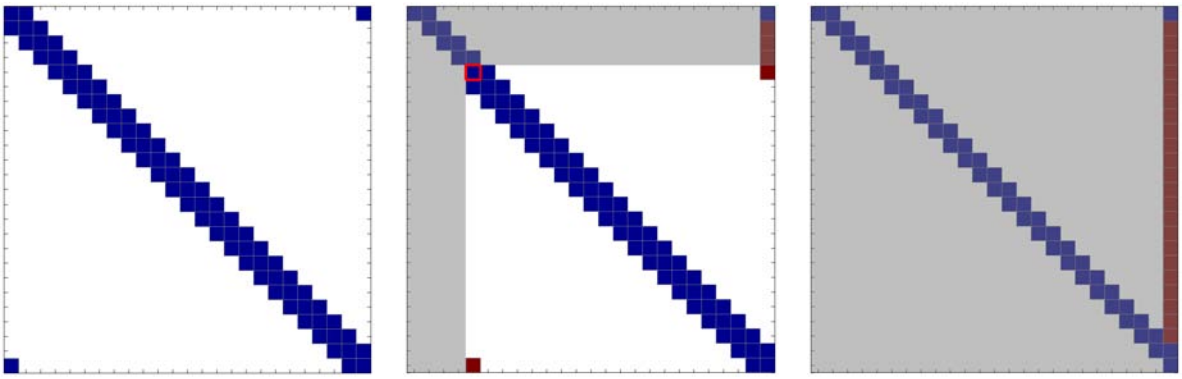



Figure 27.5: "Cyclic" spring-mass system with $n = 6$ masses.



(a) original matrix $A$ (b) beginning of step 5 (c) final matrix $U$

Figure 27.6: Illustration of Gaussian elimination applied to a $25 \times 25$ "arrow" system. The red entries are "fill-in" introduced in the factorization process. The pivot for each step is marked by a red square.

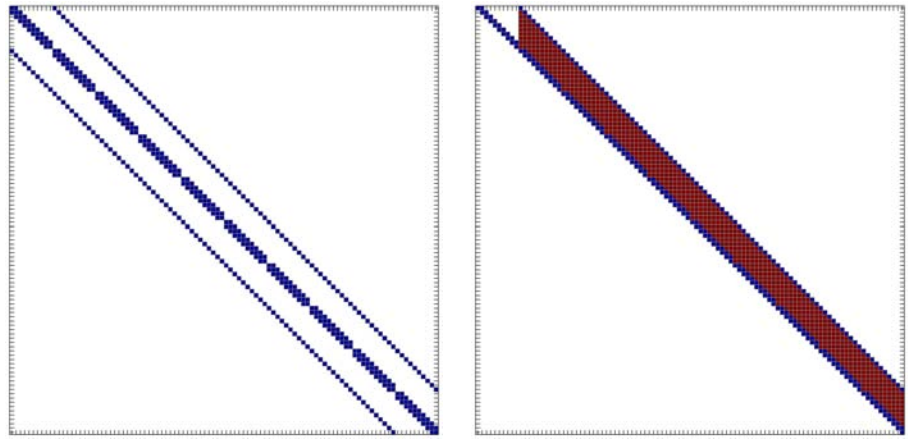
is no longer tridiagonal. In fact, if apply our standard classification for banded matrices, the cyclic matrix would be characterized by its bandwidth of $m_\text{b} = n - 1$.

Applying Gaussian elimination to the "cyclic" system, we immediately recognize that the $(1, n)$ entry of the original matrix "falls" along the last column, creating $n - 2$ fill-ins (see Figures 27.6(b) and 27.6(c)). In addition, the original $(n, 1)$ entry also creates a nonzero entry on the bottom row, which moves across the matrix with the pivot as the matrix is factorized. As a result, the operation count for the factorization of the "cyclic" system is in fact similar to that of a pentadiagonal system: approximately $14n$ FLOPs. Applying back substitution to the factored matrix — which contains approximately $3n$ nonzeros — require $5n$ FLOPs. Thus, solution of the cyclic system — which has just two more nonzero entries than the tridiagonal system — requires more than twice the operations ($19n$ vs. $8n$). However, it is also important to note that this $\mathcal{O}(n)$ operation count is a significant improvement compared to the $\mathcal{O}(n(m_\text{b} + 1)^2) = \mathcal{O}(n^3)$ operation estimate based on classifying the system as a standard "outrigger" with a bandwidth $m_\text{b} = n - 1$.

We note that the fill-in structure of $U$ takes the form of a skyline defined by the envelope of the columns of the original matrix $A$. This is a general principal.

### 27.4.2 Reordering

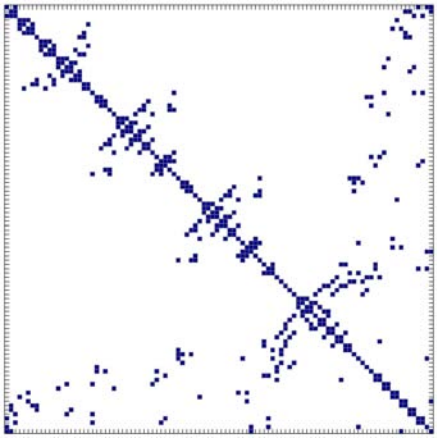
In constructing a linear system corresponding to our spring-mass system, we associated the $j^\text{th}$ entry of the solution vector — and hence the $j^\text{th}$ column of the matrix — with the displacement of the $j^\text{th}$ mass (counting from the wall) and associated the $i^\text{th}$ equation with the force equilibrium condition



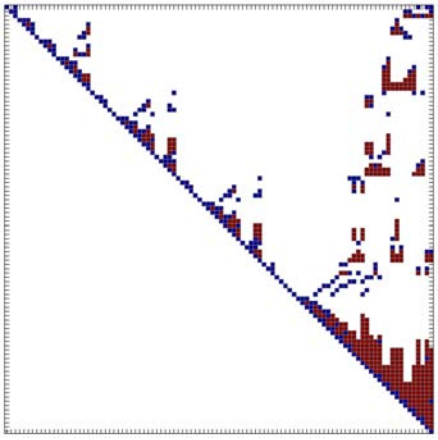
(a) $A$ (natural, $\text{nnz}(A) = 460$)



(b) $U$ (natural, $\text{nnz}(U) = 1009$)



(c) $A'$ (AMD, $\text{nnz}(A') = 460$)



(d) $U'$ (AMD, $\text{nnz}(U') = 657$)

Figure 27.7: Comparison of the sparsity pattern and Gaussian elimination fill-ins for a $n = 100$ "outrigger" system resulting from natural ordering and an equivalent system using the approximate minimum degree (AMD) ordering.

of the $i^{\text{th}}$ mass. While this is arguably the most "natural" ordering for the spring-mass system, we could have associated a given column and row of the matrix with a different displacement and force equilibrium condition, respectively. Note that this "reordering" of the unknowns and equations of the linear system is equivalent to "swapping" the rows of columns of the matrix, which is formally known as permutation. Importantly, we can describe the same physical system using many different orderings of the unknowns and equations; even if the matrices appear different, these matrices describing the same physical system may be considered equivalent, as they all produce the same solution for a given right-hand side (given both the solution and right-hand side are reordered in a consistent manner).

Reordering can make a significant difference in the number of fill-ins and the operation count. Figure 27.7 shows a comparison of number of fill-ins for an $n = 100$ linear system arising from two different orderings of a finite different discretization of two-dimensional heat equation on a $10 \times 10$ computational grid. An "outrigger" matrix of bandwidth $m_{\text{b}} = 10$ arising from "natural" ordering is shown in Figure 27.7(a). The matrix has 460 nonzero entries. As discussed in the previous section, Gaussian elimination of the matrix yields an upper triangular matrix $U$ with approximately

$n(m_b + 1) = 1100$ nonzero entries (more precisely 1009 for this particular case), which is shown in Figure 27.7(b). An equivalent system obtained using the approximate minimum degree (AMD) ordering is shown in Figure 27.7(c). This newly reordered matrix $A'$ also has 460 nonzero entries because permuting (or swapping) rows and columns clearly does not change the number of nonzeros. On the other hand, application of Gaussian elimination to this reordered matrix yields an upper triangular matrix $U$ shown in Figure 27.7(d), which has only 657 nonzero entries. Note that the number of fill-in has been reduced by roughly a factor of two: from $1009 - 280 = 729$ for the "natural" ordering to $657 - 280 = 377$ for the AMD ordering. (The original matrix $A$ has 280 nonzero entries in the upper triangular part.)

In general, using an appropriate ordering can significantly reduced the number of fill-ins and hence the computational cost. In particular, for a sparse matrix arising from $n$-unknown finite difference (or finite element) discretization of two-dimensional PDEs, we have noted that "natural" ordering produces an "outrigger" system with $m_b = \sqrt{n}$; Gaussian elimination of the system yields an upper triangular matrix with $n(m_b + 1) \approx n^{3/2}$ nonzero entries. On the other hand, the number of fill-ins for the same system with an optimal (i.e. minimum fill-in) ordering yields an upper triangular matrix with $\mathcal{O}(n \log(n))$ unknowns. Thus, ordering can have significant impact in both the operation count and storage for large sparse linear systems.

*End Advanced Material*

## 27.5 The Evil Inverse

In solving a linear system $Au = f$, we advocated a two-step strategy that consists of Gaussian elimination and back substitution, i.e.

$$\begin{aligned} \text{Gaussian elimination:} \quad & Au = f \quad \Rightarrow \quad Uu = \hat{f} \\ \text{Back substitution:} \quad & Uu = \hat{f} \quad \Rightarrow \quad u \ . \end{aligned}$$

Alternatively, we *could* find $u$ by explicitly forming the inverse of $A$, $A^{-1}$. Recall that if $A$ is non-singular (as indicated by, for example, independent columns), there exists a *unique* matrix $A^{-1}$ such that

$$AA^{-1} = I \quad \text{and} \quad A^{-1}A = I.$$

The inverse matrix $A^{-1}$ is relevant to solution systems because, in principle, we could

1. Construct $A^{-1}$;
2. Evaluate $u = A^{-1}f$ (i.e. matrix-vector product).

Note that the second step follows from the fact that

$$\begin{aligned} Au &= f \\ A^{-1}Au &= A^{-1}f \\ Iu &= A^{-1}f \, . \end{aligned}$$

While the procedure is mathematically valid, we warn that *a linear system should never be solved by explicitly forming the inverse.*

To motivate why explicitly construction of inverse matrices should be avoided, let us study the sparsity pattern of the inverse matrix for a $n$-mass spring-mass system, an example of which for $n = 5$ is shown in Figure 27.8. We use the column interpretation of the matrix and associate the column $j$ of $A^{-1}$ with a vector $p^j$, i.e.

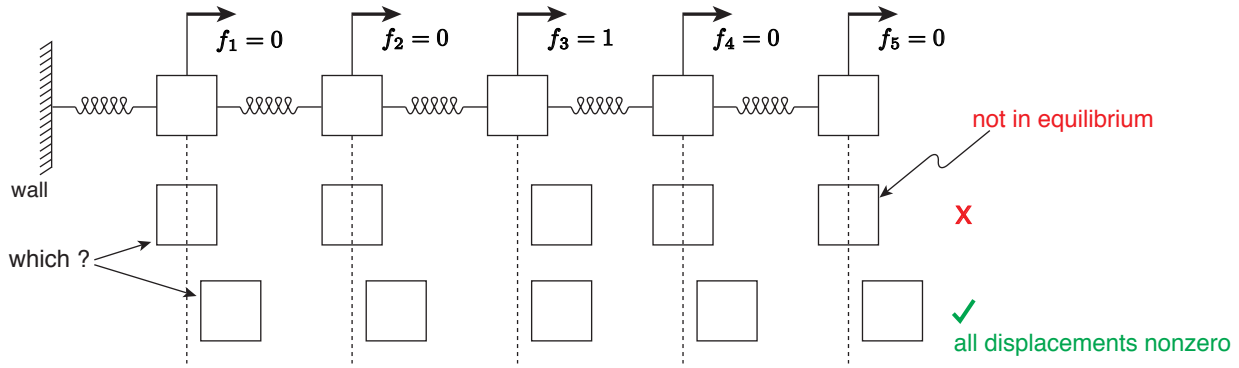
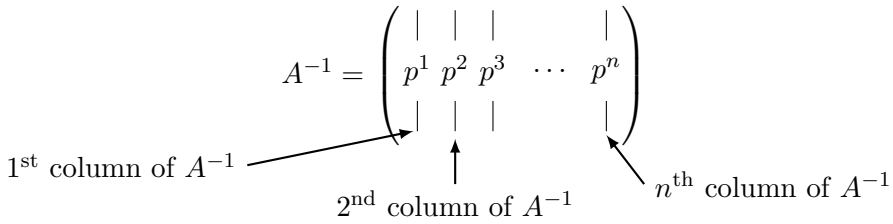



Figure 27.8: Response of a $n = 5$ spring-mass system to unit loading on mass 3.




Since $Au = f$ and $u = A^{-1}f$, we have (using one-handed matrix-vector product),

$$u = A^{-1}f = \begin{pmatrix} | & | & | & & | \\ p^1 & p^2 & p^3 & \cdots & p^n \\ | & | & | & & | \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix}$$

$$= p^1 f_1 + p^2 f_2 + \cdots + p^n f_n \,.$$

From this expression, it is clear that the vector $p^j$ is equal to the displacements of masses due to the unit force acting on mass $j$. In particular the $i^{\text{th}}$ entry of $p^j$ is the displacement of the $i^{\text{th}}$ mass due to the unit force on the $j^{\text{th}}$ mass.

Now, to deduce nonzero pattern of a vector $p^j$, let us focus on the case shown in Figure 27.8; we will deduce the nonzero entries of $p^3$ for the $n = 5$ system. Let us consider a sequence of events that takes place when $f_3$ is applied (we focus on qualitative result rather than quantitative result, i.e. whether masses move, not by how much):

1. Mass 3 moves to the right due to the unit load $f_3$.

2. Force exerted by the spring connecting mass 3 and 4 increases as the distance between mass 3 and 4 decreases.

3. Mass 4 is no longer in equilibrium as there is a larger force from the left than from the right (i.e. from the spring connecting mass 3 and 4, which is now compressed, than from the spring connecting mass 4 and 5, which is neutral).

4. Due to the unbalanced force mass 4 moves to the right.

5. The movement of mass 4 to the left triggers a sequence of event that moves mass 5, just as the movement of mass 3 displaced mass 4. Namely, the force on the spring connecting mass 4 and 5 increases, mass 5 is no longer in equilibrium, and mass 5 moves to the right.

Thus, it is clear that the unit load on mass 3 not only moves mass 3 but also mass 4 and 5 in Figure 27.8. Using the same qualitative argument, we can convince ourselves that mass 1 and 2 must also move when mass 3 is displaced by the unit load. Thus, in general, the unit load $f_3$ on mass 3 results in displacing all masses of the system. Recalling that the $i^{\text{th}}$ entry of $p^3$ is the displacement of the $i^{\text{th}}$ mass due to the unit load $f_3$, we conclude that all entries of $p^3$ are nonzero. (In absence of damping, the system excited by the unit load would oscillate and never come to rest; in a real system, intrinsic damping present in the springs brings the system to a new equilibrium state.)

Generalization of the above argument to a $n$-mass system is straightforward. Furthermore, using the same argument, we conclude that forcing of any of one of the masses results in displacing all masses. Consequently, for $p^1, \ldots, p^n$, we have

$$\begin{aligned}
u[\text{for } f = (1 \;\; 0 \;\; \cdots \;\; 0)^{\mathrm{T}}] &= p^1 \quad \leftarrow \quad \text{nonzero in all entries!} \\
u[\text{for } f = (0 \;\; 1 \;\; \cdots \;\; 0)^{\mathrm{T}}] &= p^2 \quad \leftarrow \quad \text{nonzero in all entries!} \\
&\vdots \\
u[\text{for } f = (0 \;\; 0 \;\; \cdots \;\; 0)^{\mathrm{T}}] &= p^n \quad \leftarrow \quad \text{nonzero in all entries!}
\end{aligned}$$

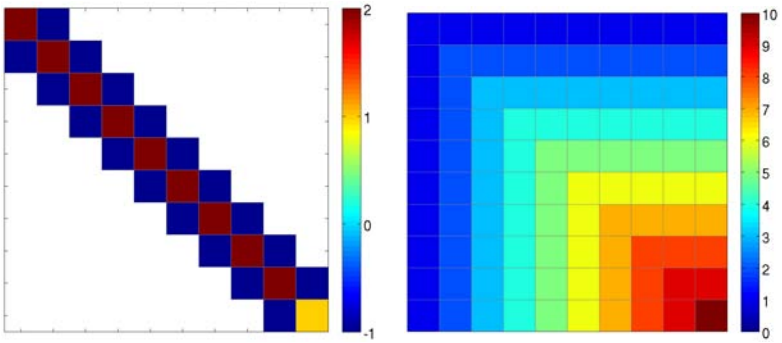
Recalling that $p^j$ is the $j^{\text{th}}$ column of $A^{-1}$, we conclude that

$$A^{-1} = \begin{pmatrix} p^1 & p^2 & \cdots & p^n \\ & & & \end{pmatrix}$$

is *full even though (here) $A$ is tridiagonal.* In general $A^{-1}$ does not preserve sparsity of $A$ and is in fact often full. This is unlike the upper triangular matrix resulting from Gaussian elimination, which preserves a large number of zeros (modulo the fill-ins).

Figure 27.9 shows the system matrix and its inverse for the $n = 10$ spring-mass system. The colors represent the value of each entries; for instance, the $A$ matrix has the typical $[-1 \;\; 2 \;\; -1]$ pattern, except for the first and last equations. Note that the inverse matrix is not sparse and is in fact full. In addition, the values of each column of $A^{-1}$ agrees with our physical intuition about the displacements to a unit load. For example, when a unit load is applied to mass 3, the distance between the wall and mass 1 increases by 1 unit, the distance between mass 1 and 2 increases by 1 unit, and the distance between mass 3 and 2 increases by 1 unit; the distances between the remaining masses are unaltered because there is no external force acting on the remaining system at equilibrium (because our system is not clamped on the right end). Accumulating displacements starting with mass 1, we conclude that mass 1 moves by 1, mass 2 moves by 2 (the sum of the increased distances between mass 1 and 2 and mass 2 and 3), mass 3 moves by 3, and all the remaining masses move by 3. This is exactly the information contained in the third column of $A^{-1}$, which reads $[1 \;\; 2 \;\; 3 \;\; 3 \;\; \ldots \;\; 3]^{\mathrm{T}}$.

In concluding the section, let us analyze the operation count for solving a linear system by explicitly forming the inverse and performing matrix-vector multiplication. We assume that our $n \times n$ matrix $A$ has a bandwidth of $m_{\text{b}}$. First, we construct the inverse matrix one column at a time




(a) $A$ (b) $A^{-1}$

Figure 27.9: Matrix $A$ for the $n = 10$ spring-mass system and its inverse $A^{-1}$. The colors represent the value of each entry as specified by the color bar.

by solving for the equilibrium displacements associated with unit load on each mass. To this end, we first compute the LU factorization of $A$ and then repeat forward/backward substitution. Recalling the operation count for a single forward/backward substitution is $\mathcal{O}(nm_{\text{b}}^2)$, the construction of $A^{-1}$ requires

$$
\begin{aligned}
Ap^1 &= \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \Rightarrow \quad p^1 \quad \mathcal{O}(nm_{\text{b}}^2) \text{ FLOPs} \\
&\vdots \\
Ap^n &= \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} \quad \Rightarrow \quad p^n \quad \mathcal{O}(nm_{\text{b}}^2) \text{ FLOPs}
\end{aligned}
$$

for the total work of $n \cdot \mathcal{O}(nm_{\text{b}}^2) \sim \mathcal{O}(n^2 m_{\text{b}}^2)$ FLOPs. Once we formed the inverse matrix, we can solve for the displacement by performing (dense) matrix-vector multiplication, i.e.

$$
\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \underbrace{\begin{pmatrix} \times & \times & \cdots & \times \\ \times & \times & \cdots & \times \\ \vdots & \vdots & \ddots & \vdots \\ \times & \times & \cdots & \times \end{pmatrix}}_{A^{-1} \text{ (full)}} \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix} \qquad \mathcal{O}(n \cdot n) = \mathcal{O}(n^2) \text{ FLOPs} \qquad .
$$

one-handed or two-handed

Thus, both the construction of the inverse matrix and the matrix-vector multiplication require $\mathcal{O}(n^2)$ operations. In contrast recall that Gaussian elimination and back substitution solves a sparse linear system in $\mathcal{O}(n)$ operations. Thus, a large sparse linear system should never be solved by explicitly forming its inverse.

# Chapter 28

# Sparse Matrices in Matlab



Throughout this chapter we shall assume that $A$ is an $n \times n$ sparse matrix. By "sparse" here we mean that most of the entries of $A$ are zero. We shall define the number of nonzero entries of $A$ by $\texttt{nnz}(A)$. Thus, by our assumption on sparsity, $\texttt{nnz}(A)$ is small compared to $n^2$; in fact, in all of our examples, and indeed in many MechE examples, $\texttt{nnz}(A)$ is typically $cn$, for a constant $c$ which is $\mathcal{O}(1)$ — say $c = 3$, or 4, or 10. (We will often consider families of matrices $A$ in which case we could state more precisely that $c$ is independent of $n$.)

## 28.1 The Matrix Vector Product

To illustrate some of the fundamental aspects of computations with sparse matrices we shall consider the calculation of the matrix vector product, $w = Av$, for $A$ a given $n \times n$ sparse matrix as defined above and $v$ a given $n \times 1$ vector. (Note that we considering here the simpler forward problem, in which $v$ is known and $w$ unknown; in a later section we consider the more difficult "inverse" problem, in which $w$ is known and $v$ is unknown.)

### 28.1.1 A Mental Model

We first consider a mental model which provides intuition as to how the sparse matrix vector product is calculated. We then turn to actual MATLAB implementation which is different in detail from the mental model but very similar in concept. There are two aspects to sparse matrices: how these matrices are stored (efficiently); and how these matrices are manipulated (efficiently). We first consider storage.

#### Storage

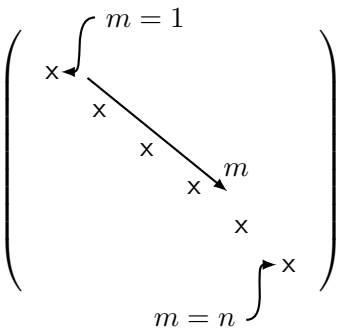
By definition our matrix $A$ is mostly zeros and hence it would make no sense to store all the entries. Much better is to just store the $\texttt{nnz}(A)$ nonzero entries with the convention that all other entries are indeed zero. This can be done by storing the indices of the nonzero entries as well as the values,

as indicated in (28.1).

$$\left.\begin{array}{l} I(m), J(m),\ 1 \le m \le \texttt{nnz}(A)\colon \\ \qquad \text{indices } i = I(m),\ j = J(m) \\ \qquad \text{for which } A_{ij} \neq 0 \\ V_A(m),\ 1 \le m \le \texttt{nnz}(A)\colon \\ \qquad \text{value of } A_{I(m),J(m)} \end{array}\right\} \begin{array}{l} \mathcal{O}(\texttt{nnz}(A)) \text{ storage} \\ \ll n^2 \text{ if sparse} \end{array} . \tag{28.1}$$

Here $m$ is an index associated to each nonzero entry, $I(m), J(m)$ are the indices of the $m^{\text{th}}$ nonzero entry, and $V_A(m)$ is the value of $A$ associated with this index, $V_A(m) \equiv A_{I(m),J(m)}$. Note that $V_A$ is a vector and not a matrix. It is clear that if $A$ is in fact dense then the scheme (28.1) actually requires more storage than the conventional non-sparse format since we store values of the indices as well as the values of the matrix; but for sparse matrices the scheme (28.1) can result in significant economies.

As a simple example we consider $A = I$, the $n \times n$ identity matrix, for which $\texttt{nnz}(A) = n$ — a very sparse matrix. Here our sparse storage scheme can be depicted as in (28.2).




$$\left.\begin{array}{lll} m = 1\colon & I(1) = 1,\ J(1) = 1 & V_A(1) = 1 \\ m = 2\colon & I(2) = 2,\ J(2) = 2 & V_A(2) = 1 \\ & \vdots & \\ m = n\colon & I(n) = 1,\ J(n) = n & V_A(n) = 1 \end{array}\right\} \begin{array}{l} \mathcal{O}(n)\ (\ll n^2)\ . \\ \text{storage} \end{array} \tag{28.2}$$

We note that the mapping between the index $m$ and the nonzero entries of $A$ is of course non-unique; in the above we identify $m$ with the row (or equivalently, column) of the entry on the main diagonal, but we could equally well number backwards or "randomly."

### Operations

We now consider the *sparse* matrix-vector product in terms of the storage scheme introduced above. In particular, we claim that

$$\underbrace{w}_{\text{to find}} = A \underbrace{v}_{\text{given}}$$

can be implemented as

$w = \texttt{zeros}(n, 1)$

for $m = 1\colon \texttt{nnz}(A)$

$$w(I(m)) = w(I(m)) + \underbrace{V_A(m)}_{A_{I(m),J(m)}} \times v(J(m))$$

end

We now discuss why this algorithm yields the desired result.

We first note that for any $i$ such that $I(m) \neq i$ for any $m$ — in other words, a row $i$ of $A$ which is entirely zero — $w(i)$ should equal zero by the usual row interpretation of the matrix vector product: $w(i)$ is the inner product between the $i^{\text{th}}$ row of $A$ — all zeros — and the vector $v$, which vanishes for any $v$.

$$i \rightarrow \underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 & 0 \\ & & & & \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \times \\ \times \\ \vdots \\ \times \end{pmatrix}}_{v} = \underbrace{\begin{pmatrix} 0 \\ \\ \\ \end{pmatrix}}_{w}$$

On the other hand, for any $i$ such that $I(m) = i$ for some $m$,

$$w(i) = \sum_{j=1}^{n} A_{ij} v_j = \sum_{\substack{j=1 \\ A_{ij} \neq 0}}^{n} A_{ij} v_j.$$

In both cases the sparse procedure yields the correct result and furthermore does not perform all the unnecessary operations associated with elements $A_{ij}$ which are zero and which will clearly not contribute to the matrix vector product: zero rows of $A$ are never visited; and in rows of $A$ with nonzero entries only the nonzero columns are visited. We conclude that the operation count is $\mathcal{O}(\texttt{nnz}(A))$ which is much less than $n^2$ if our matrix is indeed sparse.

We note that we have not necessarily used all possible sparsity since in addition to zeros in $A$ there may also be zeros in $v$; we may then not only disregard any row of $A$ which are is zero but we may also disregard any column $k$ of $A$ for which $v_k = 0$. In practice in most MechE examples the sparsity in $A$ is much more important to computational efficiency and arises much more often in actual practice than the sparsity in $v$, and hence we shall not consider the latter further.

### 28.1.2 Matlab Implementation

It is important to recognize that sparse is an attribute associated not just with the matrix $A$ in a linear algebra or mathematical sense but also an attribute in MATLAB (or other programming languages) which indicates a particular data type or class (or, in MATLAB , attribute). In situations in which confusion might occur we shall typically refer to the former simply as sparse and the latter as "declared" sparse. In general we will realize the computational savings associated with a mathematically sparse matrix $A$ only if the corresponding MATLAB entity, `A`, is also declared sparse — it is the latter that correctly invokes the sparse storage and algorithms described in the previous section. (We recall here that the MATLAB implementation of sparse storage and sparse methods is conceptually similarly to our mental model described above but not identical in terms of details.)

#### Storage

We proceed by introducing a brief example which illustrates most of the necessary MATLAB functionality and syntax. In particular, the script

```
n = 5;
```

```
K = spalloc(n,n,3*n);

K(1,1) = 2;
K(1,2) = -1;
for i = 2:n-1
    K(i,i) = 2;
    K(i,i-1) = -1;
    K(i,i+1) = -1;
end
K(n,n) = 1;
K(n,n-1) = -1;

is_K_sparse = issparse(K)

K

num_nonzeros_K = nnz(K)

spy(K)

K_full = full(K)

K_sparse_too = sparse(K_full)
```

yields the output

```
is_K_sparse =

     1

K =

   (1,1)       2
   (2,1)      -1
   (1,2)      -1
   (2,2)       2
   (3,2)      -1
   (2,3)      -1
   (3,3)       2
   (4,3)      -1
   (3,4)      -1
   (4,4)       2
   (5,4)      -1
   (4,5)      -1
   (5,5)       1

num_nonzeros_K =
```

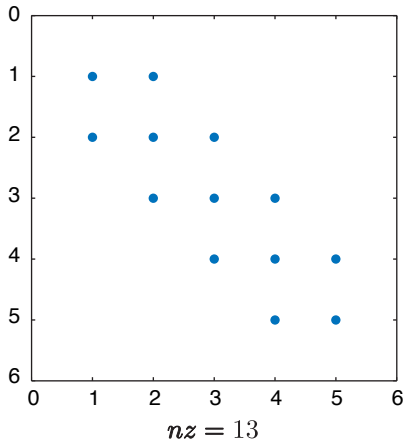



Figure 28.1: Output of `spy(K)`.

```
    13

K_full =

     2    -1     0     0     0
    -1     2    -1     0     0
     0    -1     2    -1     0
     0     0    -1     2    -1
     0     0     0    -1     1

K_sparse_too =

   (1,1)       2
   (2,1)      -1
   (1,2)      -1
   (2,2)       2
   (3,2)      -1
   (2,3)      -1
   (3,3)       2
   (4,3)      -1
   (3,4)      -1
   (4,4)       2
   (5,4)      -1
   (4,5)      -1
   (5,5)       1
```

as well as Figure 28.1.

We now explain the different parts in more detail:

First, `M = spalloc(n1,n2,k)` (*i*) creates a *declared* sparse array `M` of size `n1`× `n2` with allocation for `k` nonzero matrix entries, and then (*ii*) initializes the array `M` to all zeros. (If later

in the script this allocation may be exceeded there is no failure or error, however efficiency will suffer as memory will not be as contiguously assigned.) In our case here, we anticipate that `K` will be tri-diagonal and hence there will be less than `3*n` nonzero entries.

Then we assign the elements of $K$ — we create a simple tri-diagonal matrix associated with $n = 5$ springs in series. Note that although $K$ is declared sparse we do not need to assign values according to any complicated sparse storage scheme: we assign and more generally refer to the elements of $K$ with the usual indices, and the sparse storage bookkeeping is handled by MATLAB under the hood.

We can confirm that a matrix $M$ is indeed (declared) sparse with `issparse` — `issparse(M)` returns a logical 1 if `M` is sparse and a logical 0 if `M` is not sparse. (In the latter case, we will not save on memory or operations.) As we discuss below, some MATLAB operations may accept sparse operands but under certain conditions return non-sparse results; it is often important to confirm with `issparse` that a matrix which is intended to be sparse is indeed (declared) sparse.

Now we display $K$. We observe directly the sparse storage format described in the previous section: MATLAB displays effectively $(I(m), J(m), V_A(m))$ triplets (MATLAB does not display $m$, which as we indicated is in any event an arbitrary label).

The MATLAB built-in function `nnz(M)` returns the number of nonzero entries in a matrix `M`. The MATLAB built-in function `spy(M)` displays the `n1` $\times$ `n2` matrix `M` as a rectangular grid with (only) the nonzero entries displayed as blue filled circles — Figure 28.1 displays `spy(K)`. In short, `nnz` and `spy` permit us to quantify the sparsity and structure, respectively, of a matrix `M`.

The MATLAB built-in functions `full` and `sparse` create a full matrix from a (declared) sparse matrix and a (declared) sparse matrix from a full matrix respectively. Note however, that it is better to initialize a sparse matrix with `spalloc` rather than simply create a full matrix and then invoke `sparse`; the latter will require at least temporarily (but sometimes fatally) much more memory than the former.

There are many other sparse MATLAB built-in functions for performing various operations.

### Operations

This section is very brief: once a matrix `A` is declared sparse, then the MATLAB statement `w = A*v` will invoke the efficient sparse matrix vector product described above. In short, the (matrix) multiplication operator `*` recognizes that `A` is a (declared) sparse matrix object and then automatically invokes the correct/efficient "method" of interpretation and evaluation. Note in the most common application and in our case most relevant application the matrix `A` will be declared sparse, the vector `v` will be full (i.e., not declared sparse), and the output vector `w` will be full (i.e., not declared sparse).[1] We emphasize that if the matrix `A` is mathematically sparse but *not* declared sparse then the MATLAB `*` operand will invoke the standard full matrix-vector multiply and we not realize the potential computational savings.

[1] Note if the vector `v` is also declared sparse then the result `w` will be declared sparse as well.

## 28.2 Sparse Gaussian Elimination

This section is also very brief. As we already described in Unit III, in order to solve the matrix system $Au = f$ in MATLAB we need only write `u = A \ f` — the famous backslash operator. We can now reveal, armed with the material from the current unit, that the backslash operator in fact performs Gaussian elimination (except for overdetermined systems, in which case the least-squares problem is solved by a QR algorithm). The backslash will automatically perform partial pivoting — permutations of rows to choose the maximum-magnitude available pivot — to ensure for a non-singular matrix $K$ that a zero pivot is never encountered and that furthermore amplification of numerical round-off errors (in finite precision) is minimized.

The sparse case is similarly streamlined. If $A$ is a mathematically sparse matrix and we wish to solve $Au = f$ by sparse Gaussian elimination as described in the previous chapter, we need only make sure that `A` is declared sparse and then write `u = A \ f` . (As for the matrix vector product, `f` need not be declared sparse and the result `u` will not be sparse.) In this case the backslash does more than simply eliminate unnecessary calculations with zero operands: the backslash will permute columns (a reordering) in order to minimize fill-in during the elimination procedure. (As for the non-sparse case, row permutations will also be pursued, for purposes of numerical stability.)

The case of $A$ SPD is noteworthy. As already indicated, in this case the Gaussian elimination process is numerically stable without any row permutations. For an SPD matrix, the backslash operator will thus permute the rows in a similar fashion to the columns; the columns, as before, are permuted to minimize fill-in, as described in the previous chapter. A particular variant of Gaussian elimination, the Cholesky factorization, is pursued in the SPD case.

# 2.086 Numerical Computation for Mechanical Engineers

Spring 2013