8
Matrix Computing and N-D Newton Raphson

Physical systems are often modeled by systems of simultaneous equations. As the models are made more realistic, the matrices may become very large, and what with matrix manipulations intrinsically involving the continued repetition of a small number of simple instructions, computer become an excellent tool for these models. Because the steps are so predictable and straightforward, the codes can be made to run extraordinarily quickly by using clever techniques and by *tuning* them to a particular machine's architecture (see Chap. 13 for a discussion of computer architectures).

For the above reasons, it has become increasingly important for the computational scientist to use industrial-strength matrix subroutines from well-established scientific libraries. These subroutines are usually an order of magnitude or more faster than the elementary methods found in linear algebra texts[1], are usually designed to minimize the roundoff error, and are often "robust," that is, have a high chance of being successful for a broad class of problems. For these reasons we recommend that you *do not write your own matrix subroutines*, but, instead, get them from a library. An additional value of the library routine is that you can run the same program on a workstation and a supercomputer and automatically have the numerically most intensive parts of it adapted to the RISC, parallel, or vector architecture of the individual computer.

The thoughtful and pensive reader may be wondering when a matrix is "large" enough to be worth the effort of using a library routine. Basically, if the summed sizes of all your matrices are a good fraction of your computer's RAM, if virtual memory is needed to run your program, or if you have to wait minutes or hours for your job to finish, then it is a good bet that your matrices are "large."

Now that you have heard the sales pitch, you may be asking, "What's the cost?" In the later part of this chapter we pay the costs of having to find what libraries are available, of having to find the name of the routine in that library,

[1] Although we prize the book [9] and what it has accomplished, we cannot recommend taking subroutines from it. They are neither optimized nor documented for easy and stand-alone use. The subroutine libraries recommended in this chapter are both.

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

ISBN: 978-3-527-40626-5

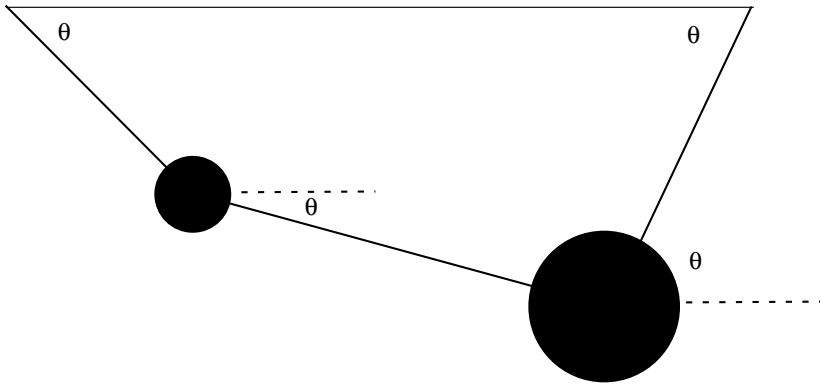

Fig. 8.1 Two masses connected by three pieces of string, and suspended from a horizontal bar of length L. The angles and the tensions in the strings are unknown.

of having to find the names of the subroutines your routine calls, and then of having to figure out how to call all these routines. And because many of the library routines are in Fortran, if you are a C programmer you will also be taxed by having to call a Fortran routine from your C program!

8.1 Two Masses on a String (Problem II)

Two masses with weights $(W_1, W_2) = (10, 20)$ N are hung from two pieces of string and a horizontal bar of lengths $(L, L_1, L_2, L_3,) = (8, 3, 4, 4)$ cm (Fig. 8.1). The **problem** is to find the angles made by the strings and the tensions exerted by the strings.

In spite of the fact that this is a simple problem requiring no more than first-year physics to formulate, the coupled transcendental equations that result cannot be solved by hand, and a computer is needed. However, even the computer cannot solve this directly, but rather must solve it by trail and error. For larger problems, like bridges, only a computed solution is possible.

In the sections to follow we solve the two-mass problem. Your **problem** is to test this solution for a variety of weights and lengths, and, for the more adventurous, to try extending it to the three-mass problem. In either case, check the physical reasonableness of your solution; the deduced tensions should be positive and of similar magnitude to the weights of the spheres, and the deduced angles should correspond to a physically realizable geometry, as confirmed with a sketch. Some of the **Exploration** you should do is see at what point your initial guess gets so bad that the computer is unable to find a physical solution.

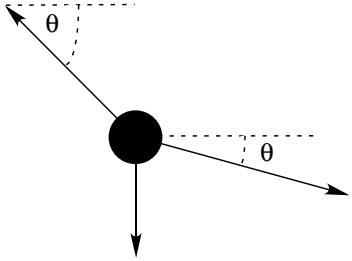

Fig. 8.2 A free body diagram for one weight in equilibrium.

8.1.1 Statics (Theory)

This physics problem is easy to convert to equations using the laws of statics, yet inhumanely painful in yielding an analytic solution. We start by writing down the geometric constraints that the horizontal length of the structure is L, and that the strings begin and end at the same height:

$$L_1 \cos\theta_1 + L_2 \cos\theta_2 + L_3 \cos\theta_3 = L \tag{8.1}$$
$$L_1 \sin\theta_1 + L_2 \sin\theta_2 - L_3 \sin\theta_3 = 0 \tag{8.2}$$
$$\sin^2\theta_1 + \cos^2\theta_1 = 1 \tag{8.3}$$
$$\sin^2\theta_2 + \cos^2\theta_2 = 1 \tag{8.4}$$
$$\sin^2\theta_3 + \cos^2\theta_3 = 1 \tag{8.5}$$

Observe that the last three equations include trigonometric identities as independent equations because we are treating $\sin\theta$ and $\cos\theta$ as independent variables; this makes the search procedure easier to implement. The basics physics says that since there are no accelerations, the sum of the forces in the horizontal and vertical directions must equal zero:

$$T_1 \sin\theta_1 - T_2 \sin\theta_2 - W_1 = 0 \tag{8.6}$$
$$T_1 \cos\theta_1 - T_2 \cos\theta_2 = 0 \tag{8.7}$$
$$T_2 \sin\theta_2 + T_3 \sin\theta_3 - W_2 = 0 \tag{8.8}$$
$$T_2 \cos\theta_2 - T_3 \cos\theta_3 = 0 \tag{8.9}$$

Here W_i is the weight of mass i, T_i is the tension in string i, and the geometry is given in Fig. 8.2. Note that since we do not have a rigid structure, we cannot assume that the torques are in equilibrium.

8.1.2 Multidimensional Newton–Raphson Searching

Equations (8.1)–(8.9) are nine simultaneous nonlinear equations. While linear equations can be solved directly, nonlinear equations cannot [10]. You can use the computer to search for a solution, but there is no guarantee that it will find one. However, if you use your physical intuition to come up with a good initial guess for the solution, the computer is more likely to succeed.

We apply the Newton–Raphson algorithm to solve this set of equations, much like we did for a single equation. We start by renaming the nine unknown angles and tensions as the subscripted variable y_i, and placing the variable together as a vector:

$$\mathbf{y} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} \sin\theta_1 \\ \sin\theta_2 \\ \sin\theta_3 \\ \cos\theta_1 \\ \cos\theta_2 \\ \cos\theta_3 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix} \tag{8.10}$$

The nine equations to be solved are written in a general form with zeros on the right-hand sides, and also placed in vector form:

$$f_i(x_1, x_2, ..., x_N) = 0 \qquad i = 1, N \tag{8.11}$$

$$\mathbf{f}(\mathbf{y}) = \begin{pmatrix} f_1(\mathbf{y}) \\ f_2(\mathbf{y}) \\ f_3(\mathbf{y}) \\ f_4(\mathbf{y}) \\ f_5(\mathbf{y}) \\ f_6(\mathbf{y}) \\ f_7(\mathbf{y}) \\ f_8(\mathbf{y}) \\ f_9(\mathbf{y}) \end{pmatrix} = \begin{pmatrix} 3x_4 + 4x_5 + 4x_6 - 8 \\ 3x_1 + 4x_2 - 4x_3 \\ x_7x_1 - x_8x_2 - 10 \\ x_7x_4 - x_8x_5 \\ x_8x_2 + x_9x_3 - 20 \\ x_8x_5 - x_9x_6 \\ x_1^2 + x_4^2 - 1 \\ x_2^2 + x_5^2 - 1 \\ x_3^2 + x_6^2 - 1 \end{pmatrix} = \mathbf{0} \tag{8.12}$$

The solution to these equations is quite a feat, a set of nine x_i values which make all nine f_i's vanish simultaneously. Although these equations are not very complicated (the physics after all is elementary), the terms quadratic in x make them nonlinear and this makes it hard or impossible to find an analytic solution. In fact, even the numerical solution is by trial and error, that is, guess work. We guess a solution, expand the nonlinear equations into linear form,

solve the linear equations, and hope that this brings us closer to the nonlinear solution. Explicitly, let the approximate solution at any one stage be the set $\{x_i\}$. We then assume that there is a set of corrections $\{\Delta x_i\}$ for which

$$f_i(x_1 + \Delta x_1, x_2 + \Delta x_2, ..., x_9 + \Delta x_9) = 0 \qquad i = 1, 9 \tag{8.13}$$

We solve for the approximate values of the Δx_i's by assuming that our previous solution is close enough to the actual solution for two terms in the Taylor series to be accurate:

$$f_i(x_1 + \Delta x_1, \ldots, x_9 + \Delta x_9) \simeq f_i(x_1, \ldots, x_9) + \sum_{j=1}^{9} \frac{\partial f_i}{\partial x_j} \Delta x_j = 0, \qquad i = 1, 9 \tag{8.14}$$

We thus have a solvable set of nine linear equations in the nine unknowns Δx_i. To make this clearer, we write them out as nine explicit equations and then as a single matrix equation:

$$\begin{aligned} f_1 + \partial f_1/\partial x_1 \Delta x_1 + \partial f_1/\partial x_2 \Delta x_2 + \cdots + \partial f_1/\partial x_9 \Delta x_9 &= 0 \\ f_2 + \partial f_2/\partial x_1 \Delta x_1 + \partial f_2/\partial x_2 \Delta x_2 + \cdots + \partial f_2/\partial x_9 \Delta x_9 &= 0 \\ &\ddots \\ f_9 + \partial f_9/\partial x_1 \Delta x_1 + \partial f_9/\partial x_2 \Delta x_2 + \cdots + \partial f_9/\partial x_9 \Delta x_9 &= 0 \end{aligned}$$

$$\text{i.e.} \quad \begin{pmatrix} f_1 \\ f_2 \\ \ddots \\ f_9 \end{pmatrix} + \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_9 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_2/\partial x_9 \\ \ddots & & & \\ \partial f_9/\partial x_1 & \partial f_9/\partial x_2 & \cdots & \partial f_9/\partial x_9 \end{pmatrix} \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \ddots \\ \Delta x_9 \end{pmatrix} = 0 \tag{8.15}$$

Note now that the derivatives and the f's are all evaluated at known values of the x_i's, so only the vector of Δx_i values is unknown.

We write this equation in matrix notation as

$$\mathbf{f} + \mathbf{F}'\boldsymbol{\Delta x} = 0 \qquad \Rightarrow \qquad \mathbf{F}'\boldsymbol{\Delta x} = -\mathbf{f} \tag{8.16}$$

$$\boldsymbol{\Delta x} = \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \ddots \\ \Delta x_9 \end{pmatrix} \quad \mathbf{f} = \begin{pmatrix} f_1 \\ f_2 \\ \ddots \\ f_9 \end{pmatrix} \quad \mathbf{F}' = \begin{pmatrix} \partial f_1/\partial x_1 & \cdots & \partial f_1/\partial x_9 \\ \partial f_2/\partial x_1 & \cdots & \partial f_2/\partial x_9 \\ \ddots & & \\ \partial f_9/\partial x_1 & \cdots & \partial f_9/\partial x_9 \end{pmatrix}$$

Here we use **bold** to emphasize the vector nature of the columns of f_i and Δx_i values, and call the matrix of the derivatives $\mathbf{F}'$ (it is also called $\mathbf{J}$ sometimes because it is the *Jacobian* matrix)

The equation $\mathbf{F}'\mathbf{\Delta x} = -\mathbf{f}$ is in the standard form for the solution of a linear equation (often written as $\mathbf{Ax} = \mathbf{b}$), where $\mathbf{\Delta x}$ is the vector of unknowns and $\mathbf{b} = -\mathbf{f}$. Matrix equations are solved using the techniques of linear algebra, and in the sections to follow we shall show how to do that on a computer. In a formal (and sometimes practical) sense, the solution of (8.16) is obtained by multiplying both sides of the equation by the inverse of the $\mathbf{F}'$ matrix:

$$\mathbf{\Delta x} = -\mathbf{F}'^{-1}\mathbf{f} \tag{8.17}$$

where the inverse must exist if there is to be a unique solution. Although we are dealing with matrices now, this solution is identical in form to that of the 1D problem, $\Delta x = -(1/f')f$. In fact, one of the reasons we use formal or abstract notations for matrices is to reveal the simplicity that lies within.

Seeing that we have gone through a good number of steps to reach this point, let us summarize. Each time we solve for the corrections Δx_i we use them to improve the existing value of the guess x_i. This is our improved guess. We then repeat the entire procedure to obtain corrections to our improved guess, which leads to an even more improved guess. The technique is repeated until the improvements are smaller than a predefined tolerance limit, or until too many iterations are made. If the initial guess is close to a solution to the nonlinear equations, then convergence is usually found after just a few iterations; if the initial guess is not close, the technique may well fail.

As we indicated for the single-equation Newton–Raphson method, while for our two-mass problem we can derive analytic expressions for the derivatives $\partial f_i/\partial x_j$, there are $9 \times 9 = 81$ such derivatives for this (small) problem, and entering them all is both time consuming and error prone. In contrast, and especially for more complicated problems, it is straightforward to program up a forward-difference approximation for the derivatives,

$$\frac{\partial f_i}{\partial x_j} \simeq \frac{f_i(x_j + \Delta x_j) - f_i(x_j)}{\delta x_j} \tag{8.18}$$

where each individual x_j is varied independently since these are partial derivatives, and δx_j are some arbitrary changes you input. While a central-difference approximation for the derivative would be more accurate, it would also require more evaluations of the f's, and once we find a solution it does not matter how accurate our algorithm for the derivative was.

As we have already discussed for the 1D Newton–Raphson method, the method can fail if the initial guess is not close enough to the zero of f (here all N of them) for the f's to be approximated as linear. The *backtracking* technique may be applied here as well, in the present case, progressively decreasing the corrections Δx_i until $|f|^2 = |f_1|^2 + |f_2|^2 + \cdots + |f_N|^2$ decreases.

8.2 Classes of Matrix Problems (Math)

It helps to remember that the rules of mathematics apply even to the world's most powerful computers. To illustrate, you *should* have problems solving equations if you have more unknowns than equations, or if your equations are not linearly independent. But do not fret, while you cannot obtain a unique solution when there are not enough equations, you may still be able to map out a space of allowable solutions. At the other extreme, if you have more equations than unknowns, you have an *overdetermined* problem, which may not have a unique solution. This overdetermined problem is sometimes treated like data fitting in which a solution to a sufficient set of equations is found, tested on the unused equations, and then improved if needed. Not surprisingly, this latter technique is known as the *linear least-squares method* because it finds the best solution "on the average."

The most basic matrix problem is the system of linear equations you have to solve for the two-mass **problem**:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \qquad \mathbf{A}_{N\times N}\mathbf{x}_{N\times 1} = \mathbf{b}_{N\times 1} \tag{8.19}$$

where $\mathbf{A}$ is a known $N \times N$ matrix, $\mathbf{x}$ is an unknown vector of length N, and $\mathbf{b}$ is a known vector of length N. The best way to solve this equation is by Gaussian elimination or LU (lower–upper) decomposition. They yields the vector $\mathbf{x}$ without explicitly calculating $\mathbf{A}^{-1}$. Another, albeit slower and less robust, method is to determine the inverse of $\mathbf{A}$, and then form the solution by multiplying both sides of (8.19) by $\mathbf{A}^{-1}$:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{8.20}$$

If you have to solve the matrix equation,

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \tag{8.21}$$

with $\mathbf{x}$ an unknown vector and λ an unknown parameter, then the direct solution (8.20) will not be of much help because the matrix $\mathbf{b} = \lambda\mathbf{x}$ contains the unknowns λ and $\mathbf{x}$. Equation (8.21) is *the eigenvalue problem*. It is harder to solve than (8.19) because solutions exist only for certain λ values (or possibly none depending on $\mathbf{A}$). We use the identity matrix to rewrite (8.21) as

$$[\mathbf{A} - \lambda\mathbf{I}]\mathbf{x} = 0 \tag{8.22}$$

we see that multiplication by $[\mathbf{A} - \lambda\mathbf{I}]^{-1}$ yields the *trivial solution*:

$$\mathbf{x} = 0 \qquad (\text{trivial solution}) \tag{8.23}$$

While the trivial solution is a bona fide solution, it is trivial. For a more interesting solution to exist, there must be something that forbids us from multiplying both sides of (8.22) by $[\mathbf{A} - \lambda\mathbf{I}]^{-1}$. That something is the nonexistence

of the inverse. If you recall that Cramer's rule for the inverse requires division by $\det[\mathbf{A} - \lambda\mathbf{I}]$, it becomes clear that the inverse fails to exist (and in this way eigenvalues *do* exist) when

$$\det[\mathbf{A} - \lambda\mathbf{I}] = 0 \tag{8.24}$$

Those values of λ that satisfy the *secular equation* (8.24) are the eigenvalues of the original equation (8.21).

If you were interested only in the eigenvalues, you would have the computer to solve (8.24). To do that, you need a subroutine to calculate the determinant of a matrix, and then a search routine to zero in the solution of (8.24). Such routines are available in the libraries. The traditional way to solve the eigenvalue problem (8.21) for both eigenvalues and eigenvectors is by *diagonalization*. This is equivalent to successive changes of basis vectors, each change leaving the eigenvalues unchanged while continually decreasing the values of the off-diagonal elements of $\mathbf{A}$. The sequence of transformations is equivalent to continually operating on the original equation with a matrix $\mathbf{U}$:

$$\mathbf{UA}(\mathbf{U}^{-1}\mathbf{U})\mathbf{x} = \lambda\mathbf{Ux} \tag{8.25}$$

$$(\mathbf{UAU}^{-1})(\mathbf{Ux}) = \lambda\mathbf{Ux} \tag{8.26}$$

until one is found for which $\mathbf{UAU}^{-1}$ is diagonal:

$$\mathbf{UAU}^{-1} = \begin{pmatrix} \lambda_1' & & \cdots & 0 \\ 0 & \lambda_2' & \cdots & 0 \\ 0 & 0 & \lambda_3' & \cdots \\ 0 & \cdots & & \lambda_N' \end{pmatrix} \tag{8.27}$$

The diagonal values of $\mathbf{UAU}^{-1}$ are the eigenvalues, with eigenvectors

$$\mathbf{x}_i = \mathbf{U}^{-1}\hat{\mathbf{e}}_i \tag{8.28}$$

that is, the eigenvectors are the columns of the matrix $\mathbf{U}^{-1}$. A number of routines of this type are found in the subroutine libraries.

8.2.1 Practical Aspects of Matrix Computing

Many scientific programming bugs arise from the improper use of arrays.[2] This may be due to the extensive use of matrices in scientific computing or the complexity of keeping track of indices and dimensions. In any case, here are some rules of thumb to observe.

[2] Even a vector $V(N)$ is called an "array," albeit a 1D one.

- **Computers are finite:** Unless you are careful, you can run out of memory or run very slowly when dealing with large matrices. As a case in point, let us say that you store data in a 4D array with each index having a *physical dimension* of 100: `A[100] [100] [100] [100]`. This array of $(100)^4$ 64-byte words occupies ≈ 1 GB (gigabyte) of memory.

- **Processing time:** Matrix operations such as inversion require on the order of N^3 steps for a square matrix of dimension N. Therefore, doubling the dimensions of a 2D square matrix (as happens when the number of integration steps are doubled) leads to an *eightfold* increase in processing time.

- **Paging:** Many computer systems have *virtual memory* in which disk space is used when a program runs out of RAM (see Chap. 13 for a discussion of how computers arrange memory). This is a slow process that requires writing a full *page* of words to the disk. If your program is near the memory limit at which paging occurs, even a slight increase in the dimensions of a matrix may lead to an order-of-magnitude increase in running time.

- **Matrix storage:** While we may think of matrices as multidimensional blocks of stored numbers, the computer stores them sequentially as linear strings of numbers. For instance, a matrix `a(3,3)` in Java and C is stored in *row-major order* (Fig. 8.3, left), while in Fortran it is stored in *column-major order* as (Fig. 8.3, right):

```
a(1,1) a(2,1) a(3,1) a(1,2) a(2,2) a(3,2) a(1,3) a(2,3) a(3,3),
```

It is important to keep this linear storage scheme in mind in order to write a proper code and to permit mixing Fortran and C programs.

- **Subscript 0:** It is standard in C and Java to have array indices begin with the value 0. While this is now permitted in Fortran, the standard has been to start indices at 1. On that account, the same matrix element will be referenced differently in the different languages:

Location	Java/C element	Fortran element
Lowest	`a[0][0]`	`a(1,1)`
	`a[0][1]`	`a(2,1)`
	`a[1][0]`	`a(3,1)`
	`a[1][1]`	`a(1,2)`
	`a[2][0]`	`a(2,2)`
Highest	`a[2][1]`	`a(3,2)`

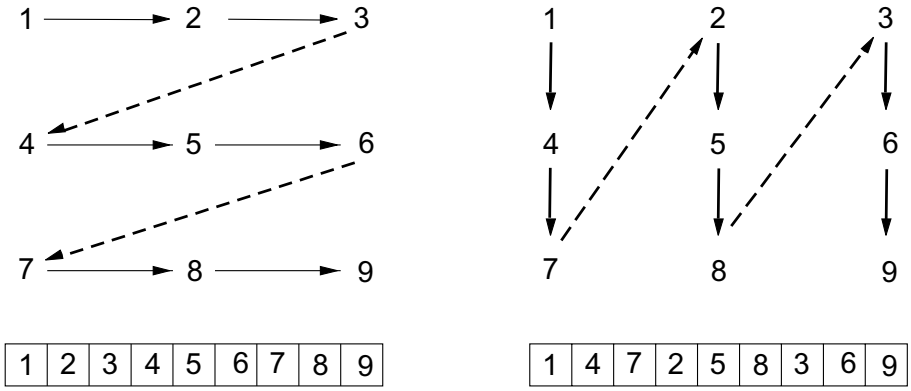

Fig. 8.3 *Left*: Row-major order used for matrix storage in C and Pascal. *Right*: column-major order used for matrix storage in Fortran. On the bottom is shown how successive matrix elements are stored in a linear fashion in memory.

Care is also needed when programming up equations in the two schemes because the implementation of the mathematics may differ. To illustrate, the Legendre polynomials are often computed via a recursion relation

$$(l+1)P_{l+1} - (2l+1)xP_l + lP_{l-1} = 0 \qquad (l = 0, 1, \ldots) \tag{8.29}$$

starting with $P_{-1} = 0$ and $P_0 = 1$. However, if the index scheme starts at $l = 1$, you would need to use the mathematically equivalent, but computationally different, relation

$$iP_i - (2i-1)xP_{i-1} + (i-1)P_{i-2} = 0 \qquad (i = 1, 2, \ldots) \tag{8.30}$$

• **Physical and logical dimensions*:** When you run a program, you issue commands such as `double a[3][3]` or `Dimension a(3,3)` that tell the compiler how much memory it needs to set aside for the array `a`. This is called *physical memory*. Sometimes you may run programs without the full complement of values declared in the declaration statements, for example, as a test case. The amount of memory you actually use to store numbers is the matrix's *logical size*.

Modern programming techniques, such as those used in Java, C, and Fortran90, permit *dynamic memory allocation*, that is, you may use variables as the dimension of your arrays and read in the values of the variables at run time. With these languages you should read in the sizes of your arrays at run time, and thus give them the same physical and logical sizes. However, Fortran77, which is the language used for many library routines, required the dimensions to be specified at compile time, and so the physical and logical sizes may well differ. To see why care is needed if the physical and logical sizes of the arrays differ, imagine that you declared `a[3][3]`, but defined elements only up to `a[2][2]`. Then the `a` in storage would look like

```
a[1][1]' a[1][2]' a[1][3] a[2][1]' a[2][2]' a[2][3] a[3][1] a[3][2] a[3][3]
```

where only the prime elements have values assigned to them. Clearly, the defined `a` values do not occupy sequential locations in memory, and so an algorithm processing this matrix cannot assume that the next element in memory is the next element in your array. This is the reason why subroutines from a library usually need to know *both* the physical and logical sizes of your arrays.

- **Passing sizes to subprograms*:** It is needed when the logical and physical dimensions of arrays differ, as is true with some library routines, but probably not with the programs you write. In cases such as using external libraries, you must also watch that the sizes of your matrices do not exceed the bounds which have been declared in the subprograms. This may occur *without* an error message, and probably will give you the wrong answers. In addition, if you are running a C program that calls a Fortran subroutine (something we discuss in Section 9.5), you need to pass *pointers* to variables and not the actual values of the variables to the Fortran subprograms (Fortran makes *reference calls*, which means it deals only with pointers as subprogram arguments). Here we have a program possibly ruining some data stored nearby:

```
Program Main                                    // In main program
  Dimension a(100), b(400)

  Subroutine Sample(a)                             // In subroutine
    Dimension a(10)                             // Smaller dimension
    a(300) = 12                    // Out of bounds, but no message
```

One way to ensure size compatibility among main programs and subroutines is to declare array sizes only in your main program and then to pass those sizes along to your subprograms as arguments.

- **Equivalence, pointers, references manipulations*:** Once upon a time computers had such limited memories that programmers would conserve memory by having different variables occupy the *same* memory location; the theory being that this would cause no harm as long as these variables were not being used at the same time. This was done by use of `Common` and `Equivalence` statements in Fortran, and by manipulations using pointers and references in other languages. These types of manipulations are obsolete (the bane of object-oriented programming) and can cause endless grief; *do not do them unless it is a matter of life or death!*

- **Say what is happening:** You decrease errors and problems by using self-explanatory labels for your indices (subscripts), telling what your variables

mean, and describing your storage schemes.

- **Clarity versus efficiency:** When dealing with matrices, you have to balance the clarity of the operations being performed against the efficiency with which the computer does them. For example, having one matrix with many indices such as `V[L,Nre,Nspin,k,kp,Z,A]` may be neat packaging, but it may require the computer to jump through large blocks of memory to get to the particular values needed (large *strides*) as you vary `k`, `kp`, and `Nre`. The solution would be to have several matrices such as `V1[Nre,Nspin,k,kp,Z,A]`, `V2[Nre,Nspin,k,kp,Z,A]`, and `V3[Nre,Nspin,k,kp,Z,A]`.

- **Tests:** Always test a library routine on a small problem whose answer you know. Then you will know if you are supplying it with the right arguments and if you have all the links working. We give exercises in Section 8.2.3 that are useful for this purpose.

8.2.2 Implementation: Scientific Libraries, WWW

Some major scientific and mathematical libraries available include (search the Web for them):

NETLIB	A WWW metalib of free math libraries	ScaLAPACK	Distributed Memory LAPACK
LAPACK	Linear Algebra Pack	JLAPACK	LAPACK library in Java
SLATEC	Comprehensive Math & Statistical Pack	ESSL	Engr & Sci Subroutine Lib (IBM)
IMSL	International Math & Statistical Libs	CERNLIB	European Cntr Nuclear Research
BLAS	Basic Linear Algebra Subprograms	JAMA	Java Matrix Lib
NAG	Numerical Algorithms Group (UK Labs)	Lapack++	Linear Algebra in C++
TNT	C++ Template Numerical Toolkit	GNUMATH	Linear Algebra in C & C++

Except for ESSL, IMSL, and NAG, all these libraries are in the public domain. However, even these proprietary ($$$) ones are frequently available on a central computer or via an institution-wide site license. General subroutine libraries are treasures to possess because they typically contain routines for almost everything you might want to do, such as

Linear algebra manipulations	Matrix operations	Interpolation, fitting
Eigensystem analysis	Signal processing	Sorting and searching
Solution of linear equations	Differential equations	Roots, zeros, & extrema
Random-number operations	Statistical functions	Numerical quadrature

LAPACK is a free, portable, modern (1990) library of Fortran 77 routines for solving the most common problems in numerical linear algebra. It is designed to be efficient on a wide range of high-performance computers, under the proviso that the hardware vendor has implemented an efficient set of BLAS (Basic Linear Algebra Subroutines). The name LAPACK is an acronym for Linear Algebra Package. In contrast to LAPACK, the SLATEC library contains general purpose mathematical and statistical Fortran routines, and is consequently more general. Nonetheless, it is not tuned to the architecture of a particular machine the way LAPACK is.

Often a subroutine library will supply only Fortran routines, and this requires the C programmer to call a Fortran routine (we describe how to do that in Section 9.5). In some cases, C-language routines may also be available, but they may not be optimized for a particular machine.

8.2.2.1 JAMA: Java Matrix Library⊙

JAMA is a basic linear algebra package for Java, developed at the US National Institute of Science [11]. We recommend it since it works well, is natural and understandable to nonexperts, is free, helps make scientific codes more universal and portable, and because not much else is available. JAMA provides object-oriented classes that construct true `Matrix` objects, that add and multiply matrices, that solve matrix equations, and prints out entire matrices in aligned row-by-row format. JAMA is intended to serve as the standard matrix class for Java.[3]

8.2.2.2 JAMA Examples

The first example deals with $\mathbf{Ax} = \mathbf{b}$ for a 3×3 $\mathbf{A}$ and 3×1 $\mathbf{x}$ and $\mathbf{b}$:

```
double[][] array = { {1.,2.,3}, {4.,5.,6.}, {7.,8.,10.} }; Matrix A
= new Matrix(array); Matrix b = Matrix.random(3,1); Matrix x =
A.solve(b); Matrix Residual = A.times(x).minus(b);
```

The vectors and matrices are declared and created as `Matrix` variables, with $\mathbf{b}$ given random values. It then solves the 3×3 linear system of equations $\mathbf{Ax} = \mathbf{b}$ with the single command `Matrix x = A.solve(b)`, and computes the residual $\mathbf{Ax} - \mathbf{b}$ with the command `Residual = A.times(x).minus(b)`.

[3] A sibling matrix package, *Jampack* [11], has also been developed at NIST and the University of Maryland, and it works for complex matrices as well.

Listing 8.1: The sample program `JamaEigen.java` uses the JAMA matrix library to solve the eigenvalue problem. Note that JAMA defines and manipulates the new data type (object) `Matrix`, which differs from an array, but can be created from one.

```
/*  JamaEigen.java: eigenvalue problem with NIST JAMA
    JAMA must be in same directory, or include JAMA in CLASSPATH
    uses Matrix.class, see Matrix.java or HTML documentation */

import Jama.*;
import java.io.*;

public class JamaEigen {

  public static void main(String[] argv) {
  double[][] I = { {2./3,-1./4,-1./4}, {-1./4,2./3,-1./4},
                                        {-1./4,-1./4,2./3}};
  Matrix MatI = new Matrix(I);          // Form Matrix from 2D arrays
  System.out.print( "Input Matrix" );
  MatI.print (10, 5);                           // Jama Matrix print

                                       // Jama eigenvalue finder
  EigenvalueDecomposition E = new EigenvalueDecomposition(MatI);
  double[] lambdaRe = E.getRealEigenvalues();          // Real eigens
  double[] lambdaIm =  E.getImagEigenvalues();         // Imag eigens
  System.out.println("Eigenvalues: \t lambda.Re[]="
                  + lambdaRe[0]+","+lambdaRe[1]+", "+lambdaRe[2]);
                                     // Get matrix of eigenvectors
  Matrix V = E.getV();
  System.out.print("\n Matrix with column eigenvectors ");
  V.print (10, 5);
  Matrix Vec = new Matrix(3,1);         // Extract single eigenvector
  Vec.set( 0, 0, V.get(0, 0) );
  Vec.set( 1, 0, V.get(1, 0) );
  Vec.set( 2, 0, V.get(2, 0) );
  System.out.print( "First Eigenvector, Vec" );
  Vec.print (10,5);
  Matrix LHS = MatI.times(Vec);           // Should get Vec as answer
  Matrix RHS = Vec.times(lambdaRe[0]);
  System.out.print( "Does LHS = RHS?" );
  LHS.print (18, 12);
  RHS.print (18, 12);
}}
```

Our second JAMA example arises in the solution for the principal-axes system for a cube, and requires us to find a coordinate system in which the inertia tensor is diagonal. This entails solving the eigenvalue problem,

$$\mathbf{I}\vec{\omega} = \lambda\vec{\omega} \tag{8.31}$$

where $\mathbf{I}$ is the original inertia matrix, $\vec{\omega}$ is an eigenvector, λ is an eigenvalue, and we are using arrows to indicate vectors. The program `JamaEigen.java` in Listing 8.1 solves for the eigenvalues and vectors, and produces output of the form

```
Input Matrix
     0.66667    -0.25000    -0.25000
    -0.25000     0.66667    -0.25000
    -0.25000    -0.25000     0.66667

Eigenvalue:  lambda.Re[] = 0.1666666666666665, 0.9166666666666666,
0.9166666666666666

 Matrix with column eigenvectors               First Eigenvector, Vec
    -0.57735    -0.70711    -0.40825                    -0.57735
    -0.57735     0.70711    -0.40825                    -0.57735
    -0.57735     0.00000     0.81650                    -0.57735

Does LHS = RHS?
     -0.096225044865     -0.096225044865
     -0.096225044865     -0.096225044865
     -0.096225044865     -0.096225044865
```

Look at `JamaEigen` and notice how we first set up the array `I` with all the elements of the inertia tensor, and then we create a matrix `MatI` with the same elements as the array. We then solve the eigenvalue problem with the creation of an eigenvalue object `E` via the JAMA command:

```
EigenDecomposition E = new EigenDecomposition(MatI);
```

Next we extract (`get`) a vector `lambdaRe` of length 3 containing the three (real) eigenvalues, `lambdaRe[0]`, `lambdaRe[1]`, `lambdaRe[2]`:

```
14 double[] lambdaRe = E.getRealEigenvalues();
```

Next we create a 3×3 matrix `V` containing the eigenvectors in the three columns of the matrix with the JAMA command:

```
Matrix V = E.getV();
```

which takes the eigenvector object `E` and gets the vectors from it. Then we form a vector `Vec` (a 3×1 `Matrix`) containing a single eigenvector by extracting the elements from `V` with a `get` method, and assigning them with a `set` method:

```
Vec.set(0,0,V.get(0,0));
```

Our final JAMA example, `JamaFit.java` in Listing 8.2, demonstrates many of the features of JAMA. It arises in the context of least-square-fitting, as discussed in Section 9.4 where we give the equations being used to fit the parabola $y(x) = b_0 + b_1 x + b_2 x^2$ to a set of N_D measured data points $(y_i, y_i \pm \sigma_i)$.

Listing 8.2: `JamaFit.java` performs a least-squares fit of a parabola to data using the JAMA matrix library to solve the set of linear equations $\mathbf{ax} = \mathbf{b}$. For illustration, the equation is solved both directly and by matrix inversion, and several techniques for assigning values to JAMA's `Matrix` are used.

```
/* JamaFit: NIST JAMA matrix libe least-squares parabola fit
    y(x) = b0 + b1 x + b2 xx JAMA libe must be in same directory as
    program, or modify CLASSPATH to include JAMA                  */

import Jama.*;
```

```
import java.io.*;

public class JamaFit {

  public static void main(String[] argv) {

    double []   x = {1., 1.05, 1.15, 1.32,  1.51, 1.68, 1.92};
    double []   y = {0.52, 0.73, 1.08, 1.44, 1.39, 1.46, 1.58};
    double [] sig = {0.1, 0.1, 0.2, 0.3, 0.2, 0.1, 0.1};
    int Nd = 7;                                  // Number of data points
    double sig2, s, sx, sxx, sy, sxxx, sxxxx, sxy, sxxy, rhl;
    double [][] Sx = new double[3][3];             // Create 3x3 array
    double [][] Sy = new double[3][1];             // Create 3x1 array

    s = sx = sxx = sy = sxxx = sxxxx = sxy = sxy = sxxy = 0;
                                                 // Generate matrix elements
    for (int i=0; i <= Nd-1; i++)  {
      sig2   = sig[i]*sig[i];
      s     += 1./sig2;
      sx    += x[i]/sig2;
      sy    += y[i]/sig2;
      rhl    = x[i]*x[i];
      sxx   += rhl/sig2;
      sxxy  += rhl*y[i]/sig2;
      sxy   += x[i]*y[i]/sig2;
      sxxx  += rhl*x[i]/sig2;
      sxxxx += rhl*rhl/sig2;
    }
                                                          // Assign arrays
    Sx[0][0] = s;
    Sx[0][1] = Sx[1][0] = sx;
    Sx[0][2] = Sx[2][0] = Sx[1][1] = sxx;
    Sx[1][2] = Sx[2][1] = sxxx;
    Sx[2][2] = sxxxx;
    Sy[0][0] = sy;
    Sy[1][0] = sxy;
    Sy[2][0] = sxxy;
                                                    // Form Jama Matrices
    Matrix MatSx = new Matrix(Sx);
    Matrix MatSy = new Matrix(3, 1);
    MatSy.set(0, 0, sy);
    MatSy.set(1, 0, sxy);
    MatSy.set(2, 0, sxxy);
                                                   // Determine inverse
    Matrix B = MatSx.inverse().times(MatSy);
    Matrix Itest = MatSx.inverse().times(MatSx);       // Test inverse
                                                     // Jama print
    System.out.print( "B Matrix via inverse" );
    B.print (16, 14);
    System.out.print( "MatSx.inverse().times(MatSx) " );
    Itest.print (16, 14);
                                                  // Direct solution too
    B = MatSx.solve(MatSy);
    System.out.print( "B Matrix via direct" );
    B.print (16,14);

              //Extract using Jama get & Print parabola coefficients
```

```
    System.out.println("FitParabola2 Final Results");
    System.out.println("\n");
    System.out.println("y(x) = b0 + b1 x + b2 x^2");
    System.out.println("\n");
    System.out.println("b0 = "+B.get(0,0));
    System.out.println("b1 = "+B.get(1,0));
    System.out.println("b2 = "+B.get(2,0));
    System.out.println("\n");
                                                        // Test fit
    for (int i=0; i <= Nd-1; i++) {
        s=B.get(0,0)+B.get(1,0)*x[i]+B.get(2,0)*x[i]*x[i];
        System.out.println
                ("i, x, y, yfit = "+i+", "+x[i]+", "+y[i]+", "+s);}
}   }
```

8.2.2.3 Other Netlib Libraries

Our example of using LAPACK has assumed that someone has been nice and placed the library on the computer. Life can be rough, however, and you may have to get the routines yourself. Probably the best place to start looking for them is Netlib, a repository of free software, documents, and databases of interest to computational scientists. Information, as well as subroutines, is available over the internet.

8.2.2.4 SLATEC Common Math Library

SLATEC (Sandia, Los Alamos, Air Force Weapons Laboratory Technical Exchange Committee) is a portable, nonproprietary, mathematical subroutine library available from Netlib. We recommend it highly and also recommend that you get more information about it from Netlib. SLATEC (CML) contains over 1400 general purpose mathematical and statistical Fortran routines. It is more general than LAPACK, which is devoted to linear algebra, yet not necessarily tuned to the particular architecture of a machine the way LAPACK is. The full library contains a guide, table of contents, and documentation via comments in the source code. The subroutines are classified by the Guide to Available Mathematical Software (GAMS) system.

For our masses on strings **Problem** we have found the needed routines:

snsq-s, dnsq-d	Find zero of *n*-variable, nonlinear function
snsqe-s, dnsqe-d	Easy-to-use snsq

If you extract these routines, you will find that they need the following:

enorm.f	j4save.f	r1mach.f	xerprn.f	fdjac1.f	r1mpyq.f
xercnt.f	xersve.f	fdump.f	qform.f	r1updt.f	xerhlt.f
xgetua.f	dogleg.f	i1mach.f	qrfac.f	snsq.f	xermsg.f

Of particular interest in these "helper" routines are *i1mach.f*, *r1mach.f*, and *d1mach.f*. They tell LAPACK the characteristic of your particular machine when

the library is first installed. Without that knowledge, LAPACK does not know when convergence is obtained or what step sizes to use.

8.2.3 Exercises for Testing Matrix Calls

Before you direct the computer to go off crunching numbers on the million elements of some matrix, it is a good idea for you to try out your procedures on a small matrix, especially one for which you know the right answer. In this way it will only take you a short time to realize how hard it is to get the calling procedure perfectly right! Here are some exercises:

1. Find the inverse of $\mathbf{A} = \begin{pmatrix} 4 & -2 & 1 \\ 3 & 6 & -4 \\ 2 & 1 & 8 \end{pmatrix}$.
   (a) As a general procedure, applicable even if you do not know the analytic answer, check your inverse in both directions; that is, check that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$.
   (b) Verify that $\mathbf{A}^{-1} = \frac{1}{263}\begin{pmatrix} 52 & 17 & 2 \\ -32 & 30 & 19 \\ -9 & -8 & 30 \end{pmatrix}$.
2. Consider the same matrix $\mathbf{A}$ as before, now used to describe three simultaneous linear equations, $\mathbf{A}\mathbf{x} = \mathbf{b}$, or explicitly,

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

   Here the vector $\mathbf{b}$ on the RHS is assumed to be known, and the problem is to solve for the vector $\mathbf{x}$. Use an appropriate subroutine to solve these equations for the three different $\mathbf{x}$ vectors appropriate to these three different $\mathbf{b}$ values on the RHS:

$$b_1 = \begin{pmatrix} +12 \\ -25 \\ +32 \end{pmatrix}, \quad b_2 = \begin{pmatrix} +4 \\ -10 \\ +22 \end{pmatrix} \quad b_3 = \begin{pmatrix} +20 \\ -30 \\ +40 \end{pmatrix}$$

   The solutions should be

$$x_1 = \begin{pmatrix} +1 \\ -2 \\ +4 \end{pmatrix} \quad x_2 = \begin{pmatrix} +0.312 \\ -0.038 \\ +2.677 \end{pmatrix} \quad x_3 = \begin{pmatrix} +2.319 \\ -2.965 \\ +4.790 \end{pmatrix}$$

3. Consider the matrix $\mathbf{A} = \begin{pmatrix} \alpha & \beta \\ -\beta & \alpha \end{pmatrix}$, where you are free to use any values you want for α and β. Use a numerical eigenproblem solver to show that the eigenvalues and eigenvectors are the complex conjugates:

$$\mathbf{x}_{1,2} = \begin{pmatrix} +1 \\ \mp i \end{pmatrix} \qquad \lambda_{1,2} = \alpha \mp i\beta$$

4. Use your eigenproblem solver to find the eigenvalues of the matrix

$$\mathbf{A} = \begin{pmatrix} -2 & +2 & -3 \\ +2 & +1 & -6 \\ -1 & -2 & +0 \end{pmatrix}$$

   (a) Verify that you obtain the eigenvalues $\lambda_1 = 5, \lambda_2 = \lambda_3 = -3$. Notice that double roots can cause problems. In particular, there is a uniqueness problem with their eigenvectors because any combinations of these eigenvectors would also be an eigenvector.

   (b) Verify that the eigenvector for $\lambda_1 = 5$ is proportional to

$$\mathbf{x}_1 = \frac{1}{\sqrt{6}} \begin{pmatrix} -1 \\ -2 \\ +1 \end{pmatrix}$$

   (c) The eigenvalue -3 corresponds to a double root. This means that the corresponding eigenvectors are degenerate, which, in turn, means that they are not unique. Two linearly independent ones are

$$\mathbf{x}_2 = \frac{1}{\sqrt{5}} \begin{pmatrix} -2 \\ +1 \\ +0 \end{pmatrix} \qquad \mathbf{x}_3 = \frac{1}{\sqrt{10}} \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix}$$

   In this case it is not clear what your eigenproblem solver will give for the eigenvectors. Try to find a relationship between your computed eigenvectors with the eigenvalue -3 to these two linearly independent ones.

5. You are investigating a physical system that you model as the $N = 100$ coupled, linear equations in N unknowns:

$$\begin{aligned} a_{11}y_1 + a_{12}y_2 + \cdots + a_{1N}y_N &= b_1 \\ a_{21}y_1 + a_{22}y_2 + \cdots + a_{2N}y_N &= b_2 \\ &\cdots \\ a_{N1}y_1 + a_{N2}y_2 + \cdots + a_{NN}y_N &= b_N \end{aligned}$$

In many cases, the a and b values are known, so your exercise is to solve for all the x values, taking **a** as the *Hilbert* matrix, and **b** as its first row:

$$[a_{ij}] = \mathbf{a} = \left[\frac{1}{i+j-1}\right] = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{100} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{101} \\ \ddots & & & & & \\ \frac{1}{100} & \frac{1}{101} & \cdots & & \cdots & \frac{1}{199} \end{pmatrix}$$

$$[b_i] = \mathbf{b} = \left[\frac{1}{i}\right] = \begin{pmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{3} \\ \ddots \\ \frac{1}{100} \end{pmatrix}$$

Compare to the analytic solution

$$\begin{pmatrix} y_1 \\ y_2 \\ \ddots \\ y_N \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \ddots \\ 0 \end{pmatrix}$$

8.2.4 Matrix Solution of Problem II

We have now set up the solution to our problem of two mass on a string. Your **problem** will be to check out the physical reasonableness of the solution for a variety of weights and lengths. You should check that the deduced tensions are positive and that the deduced angles correspond to a physical geometry (to illustrate, with a sketch). Since this is a physics-based problem, we know that the sine and cosine functions must be less than 1 in magnitude, and that the tensions should be of similar magnitude to the weights of the spheres.

8.2.5 Explorations

1. See at what point your initial guess gets so bad that the computer is unable to find a physical solution.

2. A possible problem with the formalism we have just laid out is that by incorporating the identity $\sin^2\theta_i + \cos^2\theta_i = 1$ into the equations, we may be discarding some information about the sign of $\sin\theta$ or $\cos\theta$. If you look at Fig. 8.1 you can observe that for some values of the weights and lengths, θ_2 may turn out to be negative, yet $\cos\theta$ should remain

positive. We can build this condition into our equations by replacing $f_7 - f_9$ with f's based on the form

$$f_7 = x_4 - \sqrt{1 - x_1^2} \quad f_8 = x_5 - \sqrt{1 - x_2^2} \quad f_9 = x_6 - \sqrt{1 - x_3^2} \tag{8.32}$$

See if this makes any difference in the solutions obtained.

2.* Solve the similar three-mass problem. The approach is the same, but the number of equations gets larger.