2.1 Numerical linear algebra

In this chapter we will study numerical methods for the solution of problems in linear algebra, that is to say, of problems that involve matrices or systems of linear equations. Among these we mention the following:

(a) Given an $n \times n$ matrix, calculate its determinant.

(b) Given $m$ linear algebraic equations in $n$ unknowns, find the most general solution of the system, or discover that it has none.

(c) Invert an $n \times n$ matrix, if possible.

(d) Find the eigenvalues and eigenvectors of an $n \times n$ matrix.

As usual, we will be very much concerned with the development of efficient software that will accomplish the above purposes.

We assume that the reader is familiar with the basic constructs of linear algebra: vector space, linear dependence and independence of vectors, Euclidean $n$-dimensional space, spanning sets of vectors, basis vectors. We will quickly review some additional concepts that will be helpful in our work. For a more complete discussion of linear algebra, see any of the references cited at the end of this chapter.

And now, to business. The first major concept we need is that of a linear mapping.

Let $V$ and $W$ be two vector spaces over the real numbers (we'll stick to the real numbers unless otherwise specified). We say that $T$ is a linear mapping from $V$ to $W$ if $T$ associates with every vector $\vec{v}$ of $V$ a vector $T\vec{v}$ of $W$ (so $T$ is a mapping) in such a way that

$$
T(\alpha \vec{v} + \beta \vec{v}'') = \alpha T\vec{v} + \beta T\vec{v}''
$$

for all vectors $\vec{v}'$, $\vec{v}''$ of $V$ and real numbers (or scalars) $\alpha$ and $\beta$ (i.e., $T$ is linear). Notice that the “+” signs are different on the two sides of (2.1.1). On the left we add two vectors of $V$, on the right we add two vectors of $W$.

Here are a few examples of linear mappings.

First, let $V$ and $W$ both be the same, namely the space of all polynomials of some given degree $n$. Consider the mapping that associates with a polynomial $f$ of $V$ its derivative $Tf = f'$ in $W$. It’s easy to check that this mapping is linear.

Second, suppose $V$ is Euclidean two-dimensional space (the plane) and $W$ is Euclidean three-dimensional space. Let $T$ be the mapping that carries the vector $(x, y)$ of $V$ to the vector $(3x + 2y, x - y, 4x + 5y)$ of $W$. For instance, $T(2, -1) = (4, 3, 3)$. Then $T$ is a linear mapping.

More generally, let $A$ be a given $m \times n$ matrix of real numbers, let $V$ be Euclidean $n$-dimensional space and let $W$ be Euclidean $m$-space. The mapping $T$ that carries a vector $\vec{x}$ of $V$ into $A\vec{x}$ of $W$ is a linear mapping. That is, any matrix generates a linear mapping between two appropriately chosen (to match the dimensions of the matrix) vector spaces.

The importance of studying linear mappings in general, and not just matrices, comes from the fact that a particular mapping can be represented by many different matrices. Further, it often happens that problems in linear algebra that seem to be questions about matrices, are in fact questions about linear mappings. This means that we can change to a simpler matrix that represents the same linear mapping before answering the question, secure in the knowledge that the answer will be the same. For example, if we are given a square matrix and we want its determinant, we seem to confront a problem about matrices. In fact, many matrices that represents the same mapping will
have the same determinant as the given one, and making this kind of observation and identifying simple representatives of the class of relevant matrices can be quite helpful.

To get back to the matter at hand, suppose the vector spaces $V$ and $W$ are of dimensions $m$ and $n$, respectively. Then we can choose in $V$ a basis of $m$ vectors, say $\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_m$, and in $W$ there is a basis of $n$ vectors $\vec{f}_1, \vec{f}_2, \ldots, \vec{f}_n$. Let $T$ be a linear mapping from $V$ to $W$. Then we have the situation that is sketched in figure 2.1.1 below.

**NEED FIGURE HERE**

We claim now that the action of $T$ on every vector of $V$ is known if we know only its effect on the $m$ basis vectors of $V$. Indeed, suppose we know $T\vec{e}_1, T\vec{e}_2, \ldots, T\vec{e}_m$. Then let $\vec{x}$ be any vector in $V$. Express $\vec{x}$ in terms of the basis of $V$,

$$\vec{x} = \alpha_1 \vec{e}_1 + \alpha_2 \vec{e}_2 + \cdots + \alpha_m \vec{e}_m.$$  

Now apply $T$ to both sides and use the linearity of $T$ (extended, by induction, to linear combinations of more than two vectors) to obtain

$$T\vec{x} = \alpha_1 (T\vec{e}_1) + \alpha_2 (T\vec{e}_2) + \cdots + \alpha_m (T\vec{e}_m). \tag{2.1.2}$$

The right side is known, and the claim is established.

So, to describe a linear mapping, “all we have to do” is describe its action on a set of basis vectors of $V$. If $\vec{e}_i$ is one of these, then $T\vec{e}_i$ is a vector in $W$. As such, $T\vec{e}_i$ can be written as a linear combination of the basis vectors of $W$. The coefficients of this linear combination will evidently depend on $i$, and so we write

$$T\vec{e}_i = \sum_{j=1}^{n} t_{ji} \vec{f}_j \quad i = 1, \ldots, m. \tag{2.1.3}$$

Now the $mn$ numbers $t_{ji}$, $i = 1, \ldots, m$, $j = 1, \ldots, n$, together with the given sets of basis vectors for $V$ and $W$, are enough to describe the linear operator $T$ completely. Indeed, if we know all of those numbers, then by (2.1.3) we know what $T$ does to every basis vector of $V$, and then by (2.1.2) we know the action of $T$ on every vector of $V$.

To summarize, an $n \times m$ matrix $t_{ji}$ represents a linear mapping $T$ from a vector space $V$ with a distinguished basis $E = \{\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_m\}$, to a vector space $W$ with a distinguished basis $F = \{\vec{f}_1, \vec{f}_2, \ldots, \vec{f}_n\}$, in the sense that from a knowledge of $(t, E, F)$ we know the full mapping $T$.

Next, suppose once more that $T$ is a linear mapping from $V$ to $W$. Since $T$ is linear, it is easy to see that $T$ carries the $\vec{0}$ vector of $V$ into the $\vec{0}$ vector of $W$. Consider the set of all vectors of $V$ that are mapped by $T$ into the zero vector of $W$. This set is called the kernel of $T$, and is written $\ker(T)$. Thus

$$\ker(T) = \{\vec{x} \in V \mid T\vec{x} = \vec{0}_W\}. \tag{2.1.4}$$

Now $\ker(T)$ is not just a set of vectors, it is itself a vector space, that is a vector subspace of $V$. Indeed, one sees immediately that if $\vec{x}$ and $\vec{y}$ belong to $\ker(T)$ and $\alpha$ and $\beta$ are scalars, then

$$T(\alpha\vec{x} + \beta\vec{y}) = \alpha T(\vec{x}) + \beta T(\vec{y}) = \vec{0},$$

and so $\alpha\vec{x} + \beta\vec{y}$ belongs to $\ker(T)$ also.

Since $\ker(T)$ is a vector space, we can speak of its dimension. If $\nu = \dim \ker(T)$, then $\nu$ is called the nullity of the mapping $T$.

Consider also the set of vectors $\vec{w}$ of $W$ that are of the form $\vec{w} = T\vec{v}$, for some vector $\vec{v} \in V$ (possibly many such $\vec{v}$’s exist). This set is called the image of $T$, and is written

$$\im(T) = \{\vec{w} \in W \mid \vec{w} = T\vec{v}, \ \vec{v} \in V\}. \tag{2.1.5}$$
Once again, we remark that im(\(T\)) is more than just a set of vectors, it is in fact a vector subspace of \(W\), since if \(\vec{w}^\prime\) and \(\vec{w}''\) are both in im(\(T\)), and if \(\alpha\) and \(\beta\) are scalars, then we have \(\alpha \vec{w}^\prime = T \vec{v}^\prime\) and \(\beta \vec{w}'' = T \vec{v}''\) for some \(\vec{v}^\prime, \vec{v}''\) in \(V\). Hence
\[
\alpha \vec{w}^\prime + \beta \vec{w}'' = T(\alpha \vec{v}^\prime + \beta \vec{v}'')
\]
and so \(\alpha \vec{w}^\prime + \beta \vec{w}''\) lies in im(\(T\)), too.

The dimension of the vector (sub)space im(\(T\)) is called the \textit{rank} of the mapping \(T\).

A celebrated theorem of Sylvester asserts that
\[
\text{rank}(T) + \text{nullity}(T) = \dim(V). \quad (2.1.6)
\]

By the \textit{rank of a matrix} \(A\) we mean any of the following:

(a) the maximum number of linearly independent rows that we can find in \(A\)
(b) same for columns
(c) the largest number \(r\) for which there is an \(r \times r\) nonzero sub-determinant in \(A\) (\(i.e.,\) a set of \(r\) rows and \(r\) columns, not necessarily consecutive, such that the \(r \times r\) submatrix that they determine has a nonzero determinant.

It is true that the rank of a linear mapping \(T\) from \(V\) to \(W\) is equal to the rank of any matrix \(A\) that represents \(T\) with respect to some pair of bases of \(V\) and \(W\).

It is also true that the rank of a matrix is not computable unless infinite-precision arithmetic is used. In fact, the \(2 \times 2\) matrix
\[
\begin{bmatrix}
1 & 1 \\
1 & 1 + 10^{-20}
\end{bmatrix}
\]
has rank 2, but if the \([2,2]\)-entry is changed to zero, the rank becomes 1. No computer program will be able to tell the difference between these two situations unless it is doing arithmetic to at least 21 digits of precision. Therefore, unless our programs do exact arithmetic on rational numbers, or do finite field arithmetic, or whatever, the rank will be uncomputable.

Now we’re ready to consider one of the most important problems of numerical linear algebra: the solution of simultaneous linear equations.

Let \(A\) be a given \(m \times n\) matrix, let \(\vec{b}\) be a given column vector of length \(m\), and consider the system
\[
A \vec{x} = \vec{b} \quad (2.1.7)
\]
of \(m\) linear simultaneous equations in \(n\) unknowns \(x_1, x_2, \ldots, x_n\).

Consider the set of all solution vectors \(\vec{x}\) of (2.1.7). Is it a vector space? That’s right, it isn’t, unless \(\vec{b}\) happens to be the zero vector (why?).

Suppose then that we intend to write a computer program that will in some sense present as output all solutions \(\vec{x}\) of (2.1.7). What might the output look like?

If \(\vec{b} = \vec{0}\), \(i.e.,\) if the system is \textit{homogeneous}, there is no difficulty, for in that case the solution set is ker(\(A\)), a vector space, and we can describe it by printing out a set of basis vectors of ker(\(A\)).

If the right-hand side vector \(\vec{b}\) is not \(\vec{0}\), then consider any two solutions \(\vec{x}'\) and \(\vec{x}''\) of (2.1.7). Then \(A \vec{x}' = \vec{b}\), \(A \vec{x}'' = \vec{b}\), and by subtraction, \(A(\vec{x}' - \vec{x}'') = \vec{0}\). Hence \(\vec{x}' - \vec{x}''\) belongs to ker(\(A\)), and so if \(\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_\nu\) are a basis for ker(\(A\)), then \(\vec{x}' - \vec{x}'' = \alpha_1 \vec{e}_1 + \alpha_2 \vec{e}_2 + \cdots + \alpha_\nu \vec{e}_\nu\).
If \( \vec{b} \neq \vec{0} \) then, we can describe all possible solutions of (2.1.7) by printing out one particular solution \( \vec{x}'' \), and a list of the basis vectors of \( \ker(A) \), because then all solutions are of the form

\[
\vec{x} = \vec{x}'' + \alpha_1 \vec{e}_1 + \alpha_2 \vec{e}_2 + \cdots + \alpha_\nu \vec{e}_\nu.
\] (2.1.8)

Therefore, a computer program that alleges that it solves (2.1.7) should print out a basis for the kernel of \( A \) together with, in case \( \vec{b} \neq \vec{0} \), any one particular solution of the system. We will see how to accomplish this in the next section.

**Exercises 2.1**

1. Show by examples that \( \text{rank}(A) \) is a discontinuous function of the matrix \( A \).

2. Consider the vector space \( V \) of all polynomials in \( x \) of degree at most 2. Let \( T \) be the linear mapping that sends each polynomial to its derivative.
   
   (a) What is the rank of \( T \)?
   
   (b) What is the image of \( T \)?
   
   (c) For the basis \( \{1, x, x^2\} \) of \( V \), find the \( 3 \times 3 \) matrix that represents \( T \).
   
   (d) Regard \( T \) as a mapping from \( V \) to the space \( W \) of polynomials of degree 1. Use the basis given in part (c) for \( V \), and the basis \( \{1, x - 1\} \) for \( W \), and find the \( 2 \times 3 \) matrix that represents \( T \) with respect to these bases.
   
   (e) Check that the ranks of the matrices in (c) and (d) are equal.

3. Let \( T \) be a linear mapping of Euclidean 3-dimensional space to itself. Suppose \( T \) takes the vector \((1, 1, 1)\) to \((1, 2, 3)\), and \( T \) takes \((1, 0, -1)\) to \((2, 0, 1)\) and \( T \) takes \((3, -1, 0)\) to \((1, 1, 2)\). Find \( T(1, 2, 4) \).

4. Let \( A \) be an \( n \times n \) matrix with integer entries having absolute values at most \( M \). What is the maximum number of binary digits that could be needed to represent all of the elements of \( A \)?

5. If \( T \) acts on the vector space of polynomials of degree at most \( n \) according to \( T(f) = f' - 3f \), find ker\((T)\), im\((T)\), and rank\((T)\).

6. Why isn’t the solution set of (2.1.7) a vector space if \( \vec{b} \neq \vec{0} \)?

7. Let \( a_{ij} = r_i s_j \) for \( i, j = 1, \ldots, n \). Show that the rank of \( A \) is at most 1.

8. Suppose that the matrix

\[
A = \begin{bmatrix}
0 & 1 & 1 \\
1 & 0 & -1 \\
-2 & 1 & 1
\end{bmatrix}
\]

represents a certain linear mapping from \( V \) to \( V \) with respect to the basis

\[
\{(1, 0, 0), (0, 1, 0), (1, 1, 1)\}
\]

of \( V \). Find the matrix that represents the same mapping with respect to the basis

\[
\{(0, 0, 1), (0, 1, 1), (1, 1, 1)\}.
\]

Check that the determinant is unchanged.

9. Find a system of linear equations whose solution set consists of the vector \((1, 2, 0)\) plus any linear combination of \((-1, 0, 1)\) and \((0, 1, 0)\).
10. Construct two sets of two equations in two unknowns such that
(a) their coefficient matrices differ by at most $10^{-12}$ in each entry, and
(b) their solutions differ by at least $10^{+12}$ in each entry.

2.2 Linear systems

The method that we will use for the computer solution of $m$ linear equations in $n$ unknowns will be a natural extension of the familiar process of Gaussian elimination. Let’s begin with a little example, say of the following set of two equations in three unknowns:

\[
\begin{align*}
x + y + z &= 2 \\
x - y - z &= 5.
\end{align*}
\]  

(2.2.1)

If we subtract the second equation from the first, then the two equations can be written

\[
\begin{align*}
x + y + z &= 2 \\
2y + 2z &= -3.
\end{align*}
\]

We divide the second equation by 2, and then subtract it from the first, getting

\[
\begin{align*}
x &= \frac{7}{2} \\
y + z &= -\frac{3}{2}.
\end{align*}
\]

(2.2.2)

The value of $z$ can now be chosen arbitrarily, and then $x$ and $y$ will be determined. To make this more explicit, we can rewrite the solution in the form

\[
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix} = \begin{bmatrix}
\frac{7}{2} \\
-\frac{3}{2} \\
0
\end{bmatrix} + z \begin{bmatrix}
0 \\
-1 \\
1
\end{bmatrix}.
\]

(2.2.3)

In (2.2.3) we see clearly that the general solution is the sum of a particular solution $(7/2, -3/2, 0)$, plus any multiple of the basis vector $(0, -1, 1)$ for the kernel of the coefficient matrix of (2.2.1).

The calculation can be compactified by writing the numbers in a matrix and omitting the names of the unknowns. A vertical line in the matrix will separate the left sides and the right sides of the equations. Thus, the original system (2.2.1) is

\[
\begin{bmatrix}
1 & 1 & 1 & 2 \\
1 & -1 & -1 & 5
\end{bmatrix}.
\]

Now we do $(\text{row} 2) \leftarrow (\text{row} 1) - (\text{row} 2)$ and we have

\[
\begin{bmatrix}
1 & 1 & 1 & 2 \\
0 & 2 & 2 & -3
\end{bmatrix}.
\]

Next $(\text{row} 2) \leftarrow (\text{row} 2)/2$, and then $(\text{row} 1) \leftarrow (\text{row} 1) - (\text{row} 2)$ bring us to the final form

\[
\begin{bmatrix}
1 & 0 & 0 & \frac{7}{2} \\
0 & 1 & 1 & -\frac{3}{2}
\end{bmatrix}.
\]

which is the matrix equivalent of (2.2.2).
Now we will step up to a slightly larger example, to see some of the situations that can arise. We won’t actually write the numerical values of the coefficients, but we’ll use asterisks instead. So consider the three equations in five unknowns shown below.

\[
\begin{bmatrix}
\ast & \ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast & \ast
\end{bmatrix}.
\]

(2.2.4)

The first step is to create a 1 in the extreme upper left corner, by dividing the first row through by the [1,1] element. We will assume for the moment that the various numbers that we want to divide by are not zero. Later on we will take extensive measures to assure this.

After we divide the first row by the [1,1] element, we use the 1 in the upper left-hand corner to zero out the entries below it in column 1. That is, we let \( t = a_{21} \) and then do

\[
\overrightarrow{\text{row}(2)} \leftarrow \overrightarrow{\text{row}(2)} - t \ast \overrightarrow{\text{row}(1)}.
\]

Then let \( t = a_{31} \) and do

\[
\overrightarrow{\text{row}(3)} \leftarrow \overrightarrow{\text{row}(3)} - t \ast \overrightarrow{\text{row}(1)}.
\]

The result is that we now have the matrix

\[
\begin{bmatrix}
1 & \ast & \ast & \ast & \ast \\
0 & \ast & \ast & \ast & \ast \\
0 & 1 & \ast & \ast & \ast
\end{bmatrix}.
\]

(2.2.5)

We pause for a moment to consider what we’ve done, in terms of the original set of simultaneous equations. First, to divide a row of the matrix by a number corresponds to dividing an equation through by the same number. Evidently this does not change the solutions of the system of equations. Next, to add a constant multiple of one row to another in the matrix corresponds to adding a multiple of one equation to another, and this also doesn’t affect the solutions. Finally, in terms of the linear mapping that the matrix represents, what we are doing is changing the sets of basis vectors, keeping the mapping fixed, in such a way that the matrix that represents the mapping becomes a bit more acceptable to our taste.

Now in (2.2.5) we divide through the second row by \( a_{22} \) (again blissfully assuming that \( a_{22} \) is not zero) to create a 1 in the [2,2] position. Then we use that 1 to create zeroes (just one zero in this case) below it in the second column by letting \( t = a_{32} \) and doing

\[
\overrightarrow{\text{row}(3)} \leftarrow \overrightarrow{\text{row}(3)} - t \ast \overrightarrow{\text{row}(2)}.
\]

Finally, we divide the third row by the [3,3] element to obtain

\[
\begin{bmatrix}
1 & \ast & \ast & \ast & \ast \\
0 & 1 & \ast & \ast & \ast \\
0 & 0 & 1 & \ast & \ast
\end{bmatrix}.
\]

(2.2.6)

This is the end of the so-called \textit{forward solution}, the first phase of the process of obtaining the general solution.

Again, let’s think about the system of equations that is represented here. What is special about them is that the first unknown does not appear in the second equation, and neither the first nor the second unknown appears in the third equation.

To finish the solution of such a system of equations we would use the third equation to express \( x_3 \) in terms of \( x_4 \) and \( x_5 \), then the second equation would give us \( x_2 \) in terms of \( x_4 \) and \( x_5 \), and
finally the first equation would yield $x_1$, also expressed in terms of $x_4$ and $x_5$. Hence, we would say that $x_4$ and $x_5$ are free, and that the others are determined by them. More precisely, we should say that the kernel of $A$ has a two-dimensional basis.

Let’s see how all of this will look if we were to operate directly on the matrix (2.2.6). The second phase of the solution, that we are now beginning, is called the backwards substitution, because we start with the last equation and work backwards.

First we use the 1 in the [3,3] position to create zeros in the third column above that 1. To do this we let $t = a_{23}$ and then do
\[
\overrightarrow{\text{row}(2)} \leftarrow \overrightarrow{\text{row}(2)} - t \times \overrightarrow{\text{row}(3)}.
\]
Then we let $t = a_{13}$ and set
\[
\overrightarrow{\text{row}(1)} \leftarrow \overrightarrow{\text{row}(1)} - t \times \overrightarrow{\text{row}(3)}
\]
resulting in
\[
\begin{bmatrix}
1 & 0 & * & * & * \\
0 & 1 & 1 & * & * & * \\
0 & 0 & 1 & * & * & *
\end{bmatrix}.
\]

Observe that now $x_3$ does not appear in the equations before it. Next we use the 1 in the [2,2] position to create a zero in column 2 above that 1 by letting $t = a_{12}$ and
\[
\overrightarrow{\text{row}(1)} \leftarrow \overrightarrow{\text{row}(1)} - t \times \overrightarrow{\text{row}(2)}
\]
Of course, none of our previously constructed zeros gets wrecked by this process, and we have arrived at the reduced echelon form of the original system of equations
\[
\begin{bmatrix}
1 & 0 & 0 & * & * & * \\
0 & 1 & 0 & * & * & * \\
0 & 0 & 1 & * & * & *
\end{bmatrix}.
\] (2.2.7)

We need to be more careful about the next step, so it’s time to use numbers instead of asterisks. For instance, suppose that we have now arrived at
\[
\begin{bmatrix}
1 & 0 & 0 & a_{14} & a_{15} & a_{16} \\
0 & 1 & 0 & a_{24} & a_{25} & a_{26} \\
0 & 0 & 1 & a_{34} & a_{35} & a_{36}
\end{bmatrix}.
\] (2.2.8)
Each of the unknowns is expressible in terms of $x_4$ and $x_5$:
\[
x_1 = a_{16} - a_{14}x_4 - a_{15}x_5 \\
x_2 = a_{26} - a_{24}x_4 - a_{25}x_5 \\
x_3 = a_{36} - a_{34}x_4 - a_{35}x_5 \\
x_4 = x_4 \\
x_5 = x_5.
\]

This means that we have found the general solution of the given system by finding a particular solution and a pair of basis vectors for the kernel of $A$. They are, respectively, the vector and the two columns of the matrix shown below:
\[
\begin{bmatrix}
16 \\
26 \\
36 \\
0 \\
0
\end{bmatrix} ,
\begin{bmatrix}
a_{14} & a_{15} \\
a_{24} & a_{25} \\
a_{34} & a_{35} \\
-1 & 0 \\
0 & -1
\end{bmatrix}.
\] (2.2.9)
As this shows, we find a particular solution by filling in extra zeros in the last column until its length matches the number (five in this case) of unknowns. We find a basis matrix (i.e., a matrix whose columns are a basis for the kernel of $A$) by extending the fourth and fifth columns of the reduced row echelon form of $A$ with $-I$, where $I$ is the identity matrix whose size is equal to the nullity of the system, in this case 2.

It’s time to deal with the case where one of the ∗’s that we divide by is actually a zero. In fact we will have to discuss rather carefully what we mean by zero. In numerical work on computers, in the presence of rounding errors, it is unreasonable to expect a 0 to be exactly zero. Instead we will set a certain threshold level, and numbers that are smaller than that will be declared to be zero. The hard part will be the determination of the right threshold, but let’s postpone that question for a while, and make the convention that in this and the following sections, when we speak of matrix entries being zero, we will mean that their size is below our current tolerance level.

With that understanding, suppose we are carrying out the row reduction of a certain system, and we’ve arrived at a stage like this:

\[
\begin{bmatrix}
1 & * & * & \cdots & * & * \\
0 & 1 & * & \cdots & * & * \\
0 & 0 & 0 & \cdots & * & * \\
0 & 0 & * & \cdots & * & * \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\end{bmatrix}
\]

Normally, the next step would be to divide by $a_{33}$, but it is zero. This means that $x_3$ happens not to appear in the third equation. However, $x_3$ might appear in some later equation.

If so, we can renumber the equations so that later equation becomes the third equation, and continue the process. In the matrix, this means that we would exchange two rows, so as to bring a non-zero entry into the [3,3] position, and continue.

It is possible, though, that $x_3$ does not appear in any later equation. Then all entries $a_{ij} = 0$ for $i \geq 3$. Then we could ask for some other unknown $x_j$ for $j > 3$ that does appear in some equation later than the third. In the matrix, this amounts to searching through the whole rectangle that lies “southeast” of the [3,3] position, extending up to, but not beyond, the vertical line, to find a non-zero entry, if there is one.

If $a_{ij}$ is such a non-zero entry, then we want next to bring $a_{ij}$ into the pivot position [3,3]. We can do this in two steps. First we exchange rows 3 and $i$ (interchange two equations). Second, exchange columns 3 and $j$ (interchange the numbers of the unknowns, so that $x_3$ becomes $x_j'$ and $x_j$ becomes $x_3'$). We must remember somehow that we renumbered the unknowns, so we’ll be able to recognize the answers when we see them. The calculation can now proceed from the rejuvenated pivot element in the [3,3] position.

Else, it may happen that the rectangle southeast of [3,3] consists entirely of zeros, like this:

\[
\begin{bmatrix}
1 & * & * & \cdots & * \\
0 & 1 & * & \cdots & * \\
0 & 0 & 0 & 0 & \cdots & * \\
0 & 0 & 0 & 0 & \cdots & * \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\end{bmatrix}
\]

What then? We’re finished with the forward solution. The equations from the third onwards have only zeros on their left-hand sides. If any solutions at all exist, then those equations had better have only zeros on their right-hand sides also. The last three ∗’s in the last column (and all the entries below them) must all be zeros, or the calculation halts with the announcement that the input
system was inconsistent (i.e., has no solutions). If the system is consistent, then of course we ignore the final rows of zeros, and we do the backwards solution just as in the preceding case.

It follows that in all cases, whether there are more equations than unknowns, fewer, or the same number, the backwards solution always begins with a matrix that has a diagonal of 1’s stretching from top to bottom (the bottom may have moved up, though!), with only zero entries below the 1’s on the diagonal.

Speaking in theoretical, rather than practical terms for a moment, the number of nonzero rows in the coefficient matrix at the end of the forward solution phase is the rank of the matrix. Speaking practically again, this number simply represents a number of rows beyond which we cannot do any more reductions because the matrix entries are all indistinguishable from zero. Perhaps a good name for it is pseudorank. The pseudorank should be thought of then, not as some interesting property of the matrix that we have just computed, but as the number of rows we were able to reduce before roundoff errors became overwhelming.

**Exercises 2.2**

In problems 1–5, solve each system of equations by transforming the coefficient matrix to reduced echelon form. To “solve” a system means either to show that no solution exists or to find all possible solutions. In the latter case, exhibit a particular solution and a basis for the kernel of the coefficient matrix.

1. 
   
   \[
   \begin{align*}
   2x - y + z &= 6 \\
   3x + y + 2z &= 3 \\
   7x - y + 4z &= 15 \\
   8x + y + 5z &= 12
   \end{align*}
   \]

2. 
   
   \[
   \begin{align*}
   x + y + z + q &= 0 \\
   x - y - z - q &= 0
   \end{align*}
   \]

3. 
   
   \[
   \begin{align*}
   x + y + z &= 3 \\
   3x - y - 2z &= -1 \\
   5x + y &= 7
   \end{align*}
   \]

4. 
   
   \[
   \begin{align*}
   3x + u + v + w + t &= 1 \\
   x - u + 2v + w - 3t &= 2
   \end{align*}
   \]

5. 
   
   \[
   \begin{align*}
   x + 3y - z &= 4 \\
   2x - y + 2z &= 6 \\
   x + y + z &= 6 \\
   3x - y - z &= -2
   \end{align*}
   \]

6. Construct a set of four equations in four unknowns, of rank two.

7. Construct a set of four equations in three unknowns, with a unique solution.

8. Construct a system of homogeneous equations that has a three-dimensional vector space of solutions.

9. Under precisely what conditions is the set of all solutions of the system \( Ax = b \) a vector space?

10. Given a set of \( m \) vectors in \( n \) space, describe an algorithm that will extract a maximal subset of linearly independent vectors.
11. Given a set of \( m \) vectors, and one more vector \( \vec{w} \), describe an algorithm that will decide whether or not \( \vec{w} \) is in the vector subspace spanned by the given set.

2.3 Building blocks for the linear equation solver

Let’s now try to amplify some of the points raised by the informal discussion of the procedure for solving linear equations, with a view towards the development of a formal algorithm.

First, let’s deal with the fact that a diagonal element might be zero (in the fuzzy sense defined in the previous section) at the time when we want to divide by it.

Consider the moment when we are carrying out the forward solution, we have made \( i - 1 \) ones down the diagonal, all the entries below the ones are zeros, and next we want to put a 1 into the \([i, i]\) position and use that 1 as a pivot to reduce the entries below it to zeros.

Previously we had said that this could be done by dividing through the \( i \)th row by the \([i, i]\) element, unless that element is zero, in which case we carry out a search for some nonzero element in the rectangle that lies southeast of \([i, i]\) in the matrix. After careful analysis, it turns out that an even more conservative approach is better: the best procedure consists in searching through the entire southeast rectangle, whether or not the \([i, i]\) element is zero, to find the largest matrix element in absolute value.

If this complete search is done every time, whether or not the \([i, i]\) element is zero, then it develops that the sizes of the matrix elements do not grow very much as the algorithm unfolds, and the growth of the numerical errors is also kept to a minimum.

If that largest element found in the rectangle is, say, \( a_{uv} \), then we bring \( a_{uv} \) into the pivot position \([i, i]\) by interchanging rows \( u \) and \( i \) (renumbering the equations) and interchanging columns \( v \) and \( i \) (renumbering the unknowns). Then we proceed as before.

It may seem wasteful to make a policy of carrying out a complete search of the rectangle whenever we are ready to find the next pivot element, and especially even if a non-zero element already occupies the pivot position anyway, without a search, but it turns out that the extra labor is well-rewarded with optimum numerical accuracy and stability.

If we are solving \( m \) equations in \( n \) unknowns, then we need to carry along an extra array of length \( n \). Let’s call it \( \tau_j \), \( j = 1, \ldots, n \). This array will keep a record of the column interchanges that we do as we do them, so that in the end we will be able to identify the output. Initially, we put \( \tau_j = j \) for \( j = 1, \ldots, n \). If at a certain moment we are about to interchange, say, the \( p \)th column and the \( q \)th column, then we will also interchange the entries \( \tau_p \) and \( \tau_q \). At all times then, \( \tau_j \) will hold the number of the column where the current \( j \)th column really belongs.

It must be noted that there is a fundamental difference between the interchange of rows and the interchange of columns. An interchange of two rows corresponds simply to listing the equations that we are trying to solve in a slightly different sequence, but has no effect on the solutions. On the other hand, an interchange of two columns amounts to renumbering two of the unknowns. Hence we must keep track of the column interchanges while we are doing them, so we’ll be able to tell which unknown is which at output time, but we don’t need to record row interchanges.

At the end of the calculation then, the output arrays will have to be shuffled. The reader might want to think about how do carry out that rearrangement, and we will return to it in section 2.6 under the heading of “to unscramble the eggs”.

The next item to consider is that we would like our program to be able to solve not just one system \( A\vec{x} = \vec{b} \), but several systems of simultaneous equations, each of the form \( A\vec{x} = \vec{b} \), where the left-hand sides are all the same, but the right-hand sides are different. The data for our program will therefore be an \( m \) by \( n + p \) matrix whose first \( n \) columns will contain the coefficient matrix \( A \) and whose last \( p \) columns will be the \( p \) different right-hand side vectors \( \vec{b} \).
Why are we allowing several different right sides? Some of the main customers for our program will be matrices $A$ whose inverses we want to calculate. To find, say, the first column of the inverse of $A$ we want to solve $A\vec{x} = \vec{b}$, where $\vec{b}$ is the first column of the identity matrix. For the second column of the inverse, $\vec{b}$ would be the second column of the identity matrix, and so on. Hence, to find $A^{-1}$, if $A$ is an $n \times n$ matrix, we must solve $n$ systems of simultaneous equations each having the same left-hand side $A$, but with $n$ different right-hand side vectors.

It is convenient to solve all $n$ of these systems at once because the reduction that we apply to $A$ itself to bring it into reduced echelon form is useful in solving all $n$ of these systems, and we avoid having to repeat that part of the job $n$ times. Thus, for matrix inversion, and for other purposes too, it is very handy to have the capability of solving several systems with a common left-hand side at once.

The next point concerns the linear array $\tau$ that we are going to use to keep track of the column interchanges. Instead of storing it in its own private array, it's easier to adjoin it to the matrix $A$ that we're working on, as an extra row, for then when we interchange two columns we will automatically interchange the corresponding elements of $\tau$, and thereby avoid separate programming.

This means that the full matrix that we will be working with in our program will be $(m+1) \times (n+p)$ if we are solving $p$ systems of $m$ equations in $n$ unknowns with $p$ right-hand sides. In the program itself, let's call this matrix $C$. So $C$ will be thought of as being partitioned into blocks of sizes as shown below:

$$
C = \begin{bmatrix}
A : m \times n & RHS : m \times p \\
\tau : 1 \times n & 0 : 1 \times p
\end{bmatrix}.
$$

(2.3.1)

Now a good way to begin the writing of a program such as the general-purpose matrix analysis program that we now have in mind is to consider the different procedures, or modules, into which it may be broken up. We suggest that the individual blocks that we are about to discuss should be written as separate subroutines, each with its own clearly defined input and output, each with its own documentation, and each with its own local variable names. They should then be tested one at a time, by giving them small, suitable test problems. If this is done, then the main routine won't be much more than a string of calls to the various blocks.

1. **Procedure searchmat**($C,r,s,i1,j1,i2,j2$)

   This routine is given an $r \times s$ array $C$, and two positions in the matrix, say $[i1,j1]$ and $[i2,j2]$. It then carries out a search of the rectangular submatrix of $C$ whose northwest corner is at $[i1,j1]$ and whose southeast corner is at $[i2,j2]$, inclusive, in order to find an element of largest absolute value that lives in that rectangle. The subroutine returns this element of largest magnitude, as $\text{big}$, and the row and column in which it lives, as $\text{iloc}$, $\text{jloc}$.

   Subroutine searchmat will be called in at least two different places in the main routine. First, it will do the search for the next pivot element in the southeast rectangle. Second, it can be used to determine if the equations are consistent by searching the right-hand sides of equations $r+1, \ldots, m$ ($r$ is the pseudorank) to see if they are all zero (i.e., below our tolerance level).

2. **Procedure switchrow**($C,r,s,i,j,k,l$)

   The program is given an $r \times s$ matrix $C$, and four integers $i$, $j$, $k$ and $l$. The subroutine interchanges rows $i$ and $j$ of $C$, between columns $k$ and $l$ inclusive, and returns a variable called $\text{sign}$ with a value of $-1$, unless $i = j$, in which case it does nothing to $C$ and returns a $+1$ in $\text{sign}$.

3. **Procedure switchcol**($C,r,s,i,j,k,l$)

   This subroutine is like the previous one, except it interchanges columns $i$ and $j$ of $C$, between
rows \( k \) and \( l \) inclusive. It also returns a variable called \( \text{sign} \) with a value of \(-1\), unless \( i = j \), in which case it does nothing to \( C \) and returns a \( +1 \) in \( \text{sign} \).

The subroutines \texttt{switchrow} and \texttt{switchcol} are used during the forward solution in the obvious way, and again after the back solution has been done, to unscramble the output (see procedure 5 below, and section 2.6).

4. Procedure \texttt{pivot}(\( C, r, s, i, k, u \))

Given the \( r \times s \) matrix \( C \), and three integers \( i, k \) and \( u \), the subroutine assumes that \( C_{ii} = 1 \). It then stores \( C_{ki} \) in the local variable \( t \) sets \( C_{ki} \) to zero, and reduces row \( k \) of \( C \), in columns \( u \) to \( s \), by doing the operation \( C_{kq} \leftarrow C_{kq} - t \cdot C_{iq} \) for \( q = u, \ldots, s \).

The use of the parameter \( u \) in this subroutine allows the flexibility for economical operation in both the forward and back solution. In the forward solution, we take \( u = i + 1 \) and it reduces the whole row \( k \). In the back solution we use \( u = n + 1 \), because the rest of row \( k \) will have already been reduced.

\textit{Note:} Be sure to save this routine because it will be used again in connection with linear programming, in section 2.13.

5. Procedure \texttt{scale}(\( C, r, s, i, u \))

Given an \( r \times s \) matrix \( C \) and integers \( i \) and \( u \), the routine stores \( C_{ii} \) in a variable called \( piv \). It then does \( C_{ij} \leftarrow C_{ij} / piv \) for \( j = u, \ldots, s \) and returns the value of \( piv \).

\textit{Note:} This routine will be used again in connection with linear programming, in section 2.13.

6. Procedure \texttt{scamb}(\( C, r, s, n \))

This procedure permutes the first \( n \) rows of the \( r \times s \) matrix \( C \) according to the permutation that occupies the positions \( C_{r1}, C_{r2}, \ldots, C_{rn} \) on input.

The use of this subroutine is explained in detail in section 2.6 (\textit{q.v.}). Its purpose is to rearrange the rows of the output matrix that holds a basis for the kernel, and also the rows of the output matrix that holds particular solutions of the give system(s). After rearrangement the rows will correspond to the original numbering of the unknowns, thereby compensating for the renumbering that was induced by column interchanges during the forward solution. This subroutine poses some interesting questions if we require that it should not use any additional array space beyond the input matrix itself.

7. Procedure \texttt{ident}(\( C, r, s, i, j, n, q \))

This procedure inserts \( q \) times the \( n \times n \) identity matrix into the \( n \times n \) submatrix whose Northwest corner is at position \([i, j]\) of the \( r \times s \) matrix \( C \).

Now let’s look at the assembly of these building blocks into a complete matrix analysis procedure called \texttt{matalg}(\( C, r, s, m, n, p, \text{opt}, \text{eps} \)). Input items to it are:

- An \( r \times s \) matrix \( C \) (as well as the values of \( r \) and \( s \)), whose Northwest \( m \times n \) submatrix contains the matrix of coefficients of the system(s) of equations that are about to be solved. The values of \( m \) and \( n \) must also be provided to the procedure. It is assumed that \( r = 1 + \max(m, n) \). Unless the inverse of the coefficient matrix is wanted, the Northeast \( m \times p \) submatrix of \( C \) holds \( p \) different right-hand side vectors for which we want solutions.
- The numbers \( r, s, m, n \) and \( p \).
- A parameter \texttt{option} that is equal to 1 if we want an inverse, equal to 2 if we want to see the determinant of the coefficient matrix (if square) as well as a basis for the kernel (if it is nontrivial) and a set of \( p \) particular solution vectors.
• A real parameter $\epsilon$ that is used to bound roundoff error.

Output items from the procedure `matalg` are:

• The pseudorank $r$
• The determinant $\text{det}$ if $m = n$
• An $n \times r$ matrix `basis`, whose columns are a basis for the kernel of the coefficient matrix.
• An $n \times p$ matrix `partic`, whose columns are particular solution vectors for the given systems.

In case $\text{opt} = 1$ is chosen, the procedure will fill the last $m$ columns and rows of $C$ with an $m \times m$ identity matrix, set $p = n = m$, and proceed as before, leaving the inverse matrix in the same place, on output.

Let’s remark on how the determinant is calculated. The reduction of the input matrix to echelon form in the forward solution phase entails the use of three kinds of operations. First we divide a row by a pivot element. Second, we multiply a row by a number and add it to another row. Third, we exchange a pair of rows or columns.

The first operation divides the determinant by that same pivot element. The second has no effect on the determinant. The third changes the sign of the determinant, at any rate if the rows or columns are distinct. At the end of the forward solution the matrix is upper triangular, with 1’s on the diagonal, hence its determinant is clearly 1.

What must have been the value of the determinant of the input matrix? Clearly it must have been equal to the product of all the pivot elements that were used during the reduction, together with a plus or minus sign from the row or column interchanges.

Hence, to compute the determinant, we begin by setting $\text{det}$ to 1. Then, each time a new pivot element is selected, we multiply $\text{det}$ by it. Finally, whenever a pair of different rows or columns are interchanged we reverse the sign of $\text{det}$. Then $\text{det}$ holds the determinant of the input matrix when the forward solution phase has ended.

Now we have described the basic modules out of which a general purpose program for linear equations can be constructed. In the next section we are going to discuss the vexing question of roundoff error. And how to set the tolerance level below which entries are declared to be zero. A complete formal algorithm that ties together all of these modules, with control of rounding error, is given at the end of the next section.

**Exercises 2.3**

1. Make a test problem for the major program that you’re writing by tracing through a solution the way the computer would:

   Take one of the systems that appears at the end of section 2.2. Transform it to reduced row echelon form step by step, being sure to carry out a complete search of the Southeast rectangle each time, and to interchange rows and columns to bring the largest element found in to the pivot position. Record the column interchanges in $\tau$, as described above. Record the status of the matrix $C$ after each major loop so you’ll be able to test your program thoroughly and easily.

2. Repeat problem 1 on a system of each major type: inconsistent, unique solution, many solutions.
3. Construct a formal algorithm that will invert a matrix, using no more array space than the matrix itself. The idea is that the input matrix is transformed, a column at a time, into the identity matrix, and the identity matrix is transformed, a column at a time, into the inverse. Why store all of the extra columns of the identity matrix? (Good luck!)

4. Show that a matrix $A$ is of rank one if and only if its entries are of the form $A_{ij} = f_ig_j$ for all $I$ and $j$.

5. Show that the operation $\mathbf{r}_i \leftarrow c \mathbf{r}_i + \mathbf{r}_j$ applied to a matrix $A$ has the same effect as first applying that same operation to the identity matrix $I$ to get a certain matrix $E$, and then computing $EA$.

6. Show that the operation of scaling $\mathbf{r}_i$: $a_{ik} \leftarrow \frac{a_{ik}}{a_{ii}}$, $k = 1, \ldots, n$ has the same effect as first dividing the $i$th row of the identity matrix by $a_{ii}$ to get a certain matrix $E$, and then computing $EA$.

7. Suppose we do a complete forward solution without ever searching or interchanging rows or columns. Show that the forward solution amounts to discovering a lower triangular matrix $L$ and an upper triangular matrix $U$ such that $LA = U$ (think of $L$ as a product of several matrices $E$ such as you found in the preceding two problems).

2.4 How big is zero?

The story of the linear algebra subroutine has just two pieces untold: the first concerns how small we will allow a number to be without calling it zero, and the second concerns the rearrangement of the output to compensate for interchanges of rows and columns that are done during the row-echelon reduction.

The main reduction loop begins with a search of the rectangle that lies Southeast of the pivot position $[i, i]$, in order to locate the largest element that lives there and to use it for the next pivot. If that element is zero, the forward solution halts because the remaining pivot candidates are all zero.

But “how zero” do they have to be? Certainly it would be to much to insist, when working with sixteen decimal digits, that a number should be exactly equal to zero. A little more natural would be to declare that any number that is no larger than the size of the accumulated roundoff error in the calculation should be declared to be zero, since our microscope lens would then be too clouded to tell the difference.

It is important that we should know how large roundoff error is, or might be. Indeed, if we set too small a threshold, then numbers that “really are” zero will slip through, the calculation will continue after it should have terminated because of unreliability of the computed entries, and so forth. If the threshold is too large, we will declare numbers to be zero that aren’t, and our numerical solution will terminate too quickly because the computed matrix elements will be declared to be unreliable when really they are perfectly OK.

The phenomenon of roundoff error occurs because of the finite size of a computer word. If a word consists of $d$ binary digits, the when two $d$-digit binary numbers are multiplied together, the answer that should be $2d$ bits long gets rounded off to $d$ bits when it is stored. By doing so we incur a rounding error whose size is at most 1 unit in the $(d + 1)$st place.

Then we proceed to add that answer to other numbers with errors in them, and to multiply, divide, and so forth, some large number of times. The accumulation of all of this rounding error
can be quite significant in an extended computation, particularly when a good deal of cancellation occurs from subtraction of nearly equal quantities.

The question is to determine the level of rounding error that is present, while the calculation is proceeding. Then, when we arrive at a stage where the numbers of interest are about the same size as the rounding errors that may be present in them, we had better halt the calculation.

How can we estimate, during the course of a calculation, the size of the accumulated roundoff error? There are a number of theoretical _a priori_ estimates for this error, but in any given computation these would tend to be overly conservative, and we would usually terminate the calculation too soon, thinking that the errors were worse than they actually were.

We prefer to let the computer estimate the error for us while it’s doing the calculation. True, it will have to do more work, but we would rather have it work a little harder if the result will be that we get more reliable answers.

Here is a proposal for estimating the accumulated rounding error during the progress of a computation. This method was suggested by Professors Nijenhuis and Wilf. We carry along an additional matrix of the same size as the matrix $C$, the one that has the coefficients and right-hand sides of the equations that we are solving. In this extra matrix we are going to keep estimates of the roundoff error in each of the elements of the matrix $C$.

In other words, we are going to keep two whole matrices, one of which will contain the coefficients and the right-hand sides of the equations, and the other of which will contain estimates of the roundoff error that is present in the elements of the first one.

At any time during the calculation that we want to know how reliable a certain matrix entry is, we’ll need only to look at the corresponding entry of the error matrix to find out.

Let’s call this auxiliary matrix $R$ (as in roundoff). Initially an element $R_{ij}$ might be as large as $2^{-d}|C_{ij}|$ in magnitude, and of either sign. Therefore, to initialize the $R$ matrix we choose a number uniformly at random in the interval $[-|2^{-d}C_{ij}|, |2^{-d}C_{ij}|]$, and store it in $R_{ij}$ for each $i$ and $j$. Hence, to begin with, the matrix $R$ is set to randomly chosen values in range in which the actual roundoff errors lie.

Then, as the calculation unfolds, we do arithmetic on the matrix $C$ of two kinds. We either scale a row by dividing it through by the pivot element, or we pivot a row against another row. In each case let’s look at the effect that the operation has on the corresponding roundoff error estimator in the $R$ matrix.

In the first case, consider a scaling operation, in which a certain row is divided by the element $C_{ii}$, and let $R_{ii}$ be the corresponding entry of the error matrix. Then, in view of the fact that

$$
\frac{C_{ij} + R_{ij}}{C_{ii} + R_{ii}} = \frac{C_{ij}}{C_{ii}} + \frac{R_{ij}}{C_{ii}} - \frac{R_{ii}C_{ij}}{C_{ii}^2} + \text{terms involving products of two or more errors}
$$

we see that the error entries $R_{ij}$ in the row that is being divided through by $C_{ii}$ should be computed as

$$
R_{ij} \leftarrow \frac{R_{ij}}{C_{ii}} - \frac{R_{ii}C_{ij}}{C_{ii}^2}. \tag{2.4.1}
$$

In the second case, suppose we are doing a pivoting operation on the $k$th row. Then for each column $q$ we do the operation $C_{kq} \leftarrow C_{kq} + t*C_{iq}$, where $t = C_{ki}$. Now let’s replace $C_{kq}$ by $C_{kq} + R_{kq}$, replace $C_{iq}$ by $C_{iq} + R_{iq}$ and replace $t$ by $t + t'$ (where $t' = R_{kq}$). Then substitute these expressions into the pivot operation above, and keep terms that are of first order in the errors (i.e., that do not involve products of two of the errors).
Then $C_{kq} + R_{kq}$ is replaced by
\[
C_{kq} + R_{kq} - (t + t') \ast (C_{iq} + R_{iq}) = (C_{kq} - t \ast C_{iq}) + (R_{kq} - t \ast R_{iq} - t' \ast C_{iq})
= (\text{new } C_{kq}) + (\text{new error } R_{kq}).
\]

It follows that as a result of the pivoting, the error estimator is updated as follows:
\[
R_{kq} \leftarrow R_{kq} - C_{ki} \ast R_{iq} - R_{ki} \ast C_{iq}.
\]

Equations (2.4.1) and (2.4.2) completely describe the evolution of the $R$ matrix. It begins life as random roundoff error; it gets modified along with the matrix elements whose errors are being estimated, and in return, we are supplied with good error estimates of each entry while the calculation proceeds.

Before each scaling and pivoting sequence we will need to update the $R$ matrix as described above. Then, when we search the now-famous Southeast rectangle for the new pivot element we accept it if it is larger in absolute value that its corresponding roundoff estimator, and otherwise we declare the rectangle to be identically zero and halt the forward solution.

The $R$ matrix is also used to check the consistency of the input system. At the end of the forward solution all rows of the coefficient matrix from a certain row onwards are filled with zeros, in the sense that the entries are below the level of their corresponding roundoff estimator. Then the corresponding right-hand side vector entries should also be zero in the same sense, else as far as the algorithm can tell, the input system was inconsistent. With typical ambiguity of course, this means either that the input system was “really” inconsistent, or just that rounding errors have built up so severely that we cannot decide on consistency, and continuation of the “solution” would be meaningless.

**Algorithm matalg**($C, r, s, m, n, p, \text{opt}, \text{eps}$). The algorithm operates on the matrix $C$, which is of dimension $r \times s$, where $r = \max(m, n) + 1$. It solves $p$ systems of $m$ equations in $n$ unknowns, unless $\text{opt}=1$, which means that it should calculate the inverse of the matrix in the first $m = n$ rows and columns of $C$.

matalg:=proc(C,r,s,m,n,p,opt,eps)
local R,i,j,Det,Done,ii,jj,Z,k,psrank;
# if opt = 1 that means inverse is expected
if opt=1 then ident(C,r,s,1,n+1,n,1) fi;
# initialize random error matrix
R:=matrix(r,s,(i,j)->0.000000000001*(rand()-500000000000)*eps*C[i,j]);
# set row permutation to the identity
for j from 1 to n do C[r,j]:=j od;
# begin forward solution
Det:=1; Done:=false; i:=0;
while ((i<m) and not(Done))do
# find largest in SE rectangle
Z:=searchmat(C,r,s,i+1,i+1,m,n);ii:=Z[1][1]; jj:=Z[1][2];
if abs(Z[2])>abs(R[ii,jj]) then
i:=i+1;
# switch rows
Det:=Det*switchrow(C,r,s,i,ii,i,s);
Z:=switchrow(R,r,s,i,ii,i,s);
# switch columns
Det:=Det*switchcol(C,r,s,i,jj,1,r);
Z:=switchcol(R,r,s,i,jj,1,r);
end do;
if abs(Z[2])>abs(R[ii,jj]) then
i:=i+1;
# switch rows
Det:=Det*switchrow(C,r,s,i,ii,i,s);
Z:=switchrow(R,r,s,i,ii,i,s);
# switch columns
Det:=Det*switchcol(C,r,s,i,jj,1,r);
Z:=switchcol(R,r,s,i,jj,1,r);
end do;
end proc;
# divide by pivot element
Z:=scaler(C,R,r,s,i,i);
Det:=Det*scale(C,r,s,i,i);
for k from i+1 to m do
  # reduce row k against row i
  Z:=pivotr(C,R,r,s,i,k,i+1);
  Z:=pivot(C,r,s,i,k,i+1);
  od;
else Done:=true fi;
od;
pseudorank:=i;
# end forward solution; begin consistency check
if pseudorank<n then
  Det:=0;
  for j from 1 to p do
    # check that right hand sides are 0 for i>pseudorank
    Z:=searchmat(C,r,s,pseudorank+1,n+j,m,n+j);
    if abs(Z[2])>abs(R[Z[1][1],Z[1][2]]) then
      printf("Right hand side %d is inconsistent",j);
      return;
    fi;
  fi;
# equations are consistent, do back solution
for j from pseudorank to 2 by -1 do
  for i from 1 to j-1 do
    Z:=pivotr(C,R,r,s,j,i,pseudorank+1);
    Z:=pivot(C,r,s,j,i,pseudorank+1);
    C[i,j]:=0; R[i,j]:=0;
  od;
od;
# end back solution, insert minus identity in basis
if pseudorank<n then
  # fill bottom of basis matrix with -I
  Z:=ident(C,r,s,pseudorank+1,pseudorank+1,n-pseudorank,-1);
  # fill under right-hand sides with zeroes
  for i from pseudorank+1 to n do for j from n+1 to s do C[i,j]:=0 od od;
  # fill under R matrix with zeroes
  for i from pseudorank+1 to n do for j from n-pseudorank to s do R[i,j]:=0 od od;
fi;
# permute rows prior to output
Z:=scramb(C,r,s,n);
# copy row r of C to row r of R
for j from 1 to n do R[r,j]:=C[r,j] od;
Z:=scramb(R,r,s,n);
return(Det,pseudorank,evalm(R));
end;

If the procedure terminates successfully, it returns a list containing three items: the first is the determinant (if there is one), the second is the pseudorank of the coefficient matrix, and the third is the matrix of estimated roundoff errors. The matrix C (which is called by name in the procedure, which means that the input matrix is altered by the procedure) will contain a basis for the kernel of the coefficient matrix in columns pseudorank + 1 to n, and p particular solution vectors, one for each
input right-hand side, in columns \( n + 1 \) to \( n + p \).

Two new sub-procedures are called by this procedure, namely \texttt{scaler} and \texttt{pivotr}. These are called immediately before the action of \texttt{scale} or \texttt{pivot}, respectively, and their mission is to update the \( R \) matrix in accordance with equations (2.4.1) or (2.4.2) to take account of the impending scaling or pivoting operation.

**Exercises 2.4**

1. Break off from the complete algorithm above, the forward solution process. State it formally as algorithm \texttt{forw}, list its global variables, and describe precisely its effect on them. Do the same for the backwards solution.

2. When the program runs, it gives the solutions and their roundoff error estimates. Work out an elegant way to print the answers and the error estimates. For instance, there’s no point in giving 12 digits of roundoff error estimate. That’s too much. Just print the number of digits of the answers that can be trusted. How would you do that? Write subroutine \texttt{prnt} that will carry it out.

3. Suppose you want to re-run a problem, with a different set of random numbers in the roundoff matrix initialization. How would you do that? Run one problem three or four times to see how sensitive the roundoff estimates are to the choice of random values that start them off.

4. Show your program to a person who is knowledgeable in programming, but who is not one of your classmates. Ask that person to use your program to solve some set of three simultaneous equations in five unknowns.

   Do not answer any questions verbally about how the program works or how to use it. Refer all such questions to your written documentation that accompanies the program.

   If the other person is able to run your program and understand the answers, award yourself a gold medal in documentation. Otherwise, improve your documentation and let the person try again. When successful, try again on someone else.

5. Select two vectors \( f, g \) of length 10 by choosing their elements at random. Form the \( 10 \times 10 \) matrix of rank 1 whose elements are \( f_i g_j \). Do this three times and add the resulting matrices to get a single \( 10 \times 10 \) matrix of rank three.

   Run your program on the coefficient matrix you just constructed, in order to see if the program is smart enough to recognize a matrix of rank three when it sees one, by halting the forward solution with pseudorank = 3.

   Repeat the above experiment 50 times, and tabulate the frequencies with which your program “thought” that the \( 10 \times 10 \) matrix had various ranks.

**2.5 Operation count**

With any numerical algorithm it is important to know how much work is involved in carrying it out. In this section we are going to estimate the labor involved in solving linear systems by the method of the previous sections.

Let’s recognize two kinds of labor: arithmetic operations, and other operations, both as applied to elements of the matrix. The arithmetic operations are \( +, -, \times, \div \), all lumped together, and by other operations we mean comparisons of size, movement of data, and other operations performed directly on the matrix elements. Of course there are many “other operations”, not involving the matrix elements directly, such as augmenting counters, testing for completion of loops, etc., that go on during the reduction of the matrix, but the two categories above represent a good measure
of the work done. We’re not going to include the management of the roundoff error matrix $R$ in our estimates, because its effect would be simply to double the labor involved. Hence, remember to double all of the estimates of the labor that we are about to derive if you’re using the $R$ matrix.

We consider a generic stage in the forward solution where we have been given $m$ equations in $n$ unknowns with $p$ right-hand sides, and during the forward solution phase we have just arrived at the $[i, i]$ element.

The next thing to do is to search the Southeast rectangle for the largest element, the rectangle contains about $(m - i) \times (m - i)$ elements. Hence the search requires that many comparisons.

Then we exchange two rows ($n + p - i$ operations), exchange two columns ($m$ operations) and divide a row by the pivot element ($n + p - i$ arithmetic operations).

Next, for each of the $m - i - 1$ rows below the $i$th, and for each of the $n + p - i$ elements of one of those rows, we do two arithmetic operations when we do the elementary row operation that produces a zero in the $i$th column. This requires, therefore, $2(n + p - i)(m - i - 1)$ arithmetic operations.

For the forward phase of the solution, therefore, we count

$$A_f = \sum_{i=1}^{r} \{2(n + p - i)(m - i - 1) + (n + p - i)\}$$  \hspace{1cm} (2.5.1)

arithmetic operations altogether, where $r$ is the pseudorank of the matrix, because the forward solution halts after the $r$th row with only zeros below.

The non-arithmetic operations in the forward phase amount to

$$N_f = \sum_{i=1}^{r} \{(m - i)(n - i) + (n + p - i) + m\}.$$  \hspace{1cm} (2.5.2)

Let’s leave these sums for a while, and go to the backwards phase of the solution. We do the columns in reverse order, from column $r$ back to 1, and when we have arrived at a generic column $j$, we want to create zeroes in all of the positions above the 1 in the $[j, j]$ position.

To do this we perform the elementary row operation

$$\overrightarrow{\text{row}}(i) \leftarrow \overrightarrow{\text{row}}(i) - A_{ij} \cdot \overrightarrow{\text{row}}(j)$$  \hspace{1cm} (2.5.3)

to each of the $j - 1$ rows above row $j$. Let $i$ be the number of one of these rows. Then, exactly how many elements of $\overrightarrow{\text{row}}(i)$ are acted upon by the elementary row operation above? Certainly the elements in $\overrightarrow{\text{row}}(i)$ that lie in columns 1 through $j - 1$ are unaffected, because only zero elements are in $\overrightarrow{\text{row}}(j)$ below them, thanks to the forward reduction process.

Furthermore, the elements in $\overrightarrow{\text{row}}(i)$ that lie in columns $j + 1$ through $r$ are unaffected, for a different reason. Indeed, any such entry is already zero, because it lies above an entry of 1 in some diagonal position that has already had its turn in the back solution (remember that we’re doing the columns in the sequence $r, r - 1, \ldots, 1$). Not only is such an entry zero, but it remains zero, because the entry of $\overrightarrow{\text{row}}(j)$ below it is also zero, having previously been deleted by the action of a diagonal element below it.

Hence in $\overrightarrow{\text{row}}(i)$, the elements that are affected by the elementary row operation (2.5.3) are those that lie in columns $j, n - r, \ldots, n, n + 1, \ldots, n + p$ (be sure to write [or modify!] the program so that the row reduction (2.5.3) acts only on those columns!). We have now shown that exactly $N + p + r - 1$ entries of each row above $\overrightarrow{\text{row}}(j)$ are affected (note that the number is independent of $j$ and $i$), and so

$$A_b = \sum_{j=1}^{r} (n + p - r + 1)(j - 1)$$  \hspace{1cm} (2.5.4)
arithmetic operations are done during the back solution, and no other operations.

It remains only to do the various sums, and for this purpose we recall that

\[
\sum_{i=1}^{N} i = \frac{N(N+1)}{2}
\]

\[
\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}
\]

Then it is straightforward to find the total number of arithmetic operations from \(A_f + A_b\) as

\[
\text{Arith}(m, n, p, r) = \frac{r^3}{6} - (2m + n + p - 5)\frac{r^2}{2} + ((n + p)(2m - 5/2) - m + 1/3)r
\]  \hspace{1cm} (2.5.5)

and the total of the non-arithmetic operations from \(N_f\) as

\[
\text{NonArith}(m, n, p, r) = \frac{r^3}{3} - (m + n)\frac{r^2}{2} + (6mn + 3m + 3n + 6p - 2)\frac{r}{6}.
\]  \hspace{1cm} (2.5.6)

Let’s look at a few important special cases. First, suppose we are solving one system of \(n\) equations in \(n\) unknowns that has a unique solution. Then we have \(m = n = r\) and \(p = 1\). We find that

\[
\text{Arith}(n, n, 1, n) = \frac{2}{3}n^3 + O(n^2)
\]  \hspace{1cm} (2.5.7)

where \(O(n^2)\) refers to some function of \(n\) that is bounded by a constant times \(n^2\) as \(n\) grows large. Similarly, for the non-arithmetic operations on matrix elements we find \(\frac{1}{3}n^3 + O(n^2)\) in this case.

It follows that a system of \(n\) equations can be solved for about one third of the price, in terms of arithmetic operations, of one matrix multiplication, at least if matrices are multiplied in the usual way (did you know that there is a faster way to multiply two matrices? We will see one later on).

Now what is the price of a matrix inversion by this method? Then we are solving \(n\) systems of \(n\) equations in \(n\) unknowns, all with the same left-hand side. Hence we have \(r = m = n = p\), and we find that

\[
\text{Arith}(n, n, n, n) = \frac{13}{6}n^3 + O(n^2).
\]  \hspace{1cm} (2.5.8)

Hence we can invert a matrix by this method for about the same price as solving 3.25 systems of equations! At first glance, it may seem as if the cost should be \(n\) times as great because we are solving \(n\) systems. The great economy results, of course, from the common left-hand sides.

The cost of the non-arithmetic operations remains at \(\frac{1}{3}n^3 + O(n^2)\).

If we want only the determinant of a square matrix \(A\), or want only the rank of \(A\) then we need to do only the forward solution, and we can save the cost of the back solution. We leave it to the reader to work out the cost of a determinant, or of finding the rank.

2.6 To unscramble the eggs

Now we have reached the last of the issues that needs to be discussed in order to plan a complete linear equation solving routine, and it concerns the rearrangement of the output so that it ends up in the right order.

During the operation of the forward solution algorithm we found it necessary to interchange rows and columns so that the largest element of the Southeast rectangle was brought into the pivot position. As we mentioned previously, we don’t need to keep a record of the row interchanges,
because they correspond simply to solving the equations in a different sequence. We must remember the column interchanges that occur along the way though, because each time we do one of them we are, in effect, renumbering the unknowns.

To remember the column interchanges we glue onto our array $C$ an additional row, just for bookkeeping purposes. Its elements are called $\tau_j$, $j = 1, \ldots, n$, and it is kept at the bottom of the matrix. More precisely, the elements of $\{\tau_j\}$ are the first $n$ entries of the new last row of the matrix, where the row contains $n + p$ entries altogether, the last $p$ of which are not used (refer to (2.3.1) to see the complete partitioning of the matrix $C$).

Now suppose we have arrived at the end of the back solution, and the answers to the original question are before us, except that they are scrambled. Here’s an example of the kind of situation that might result:

$$
\begin{bmatrix}
1 & 0 & 0 & a & b & c \\
0 & 1 & 0 & d & e & f \\
0 & 0 & 1 & g & h & k \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
3 & 5 & 2 & 1 & 4 & \ast
\end{bmatrix}
$$

(2.6.1)

The matrix above represents schematically the reduced row echelon form in a problem where there are five unknowns ($n = 5$), the pseudorank $r = 3$, just one right-hand side vector is given ($p = 1$), and the permutations that were carried out on the columns are recorded in the array $\tau : [3, 5, 2, 1, 4]$ shown in the last row of the matrix as it would be stored in a computation.

The question now is, how do we express the general solution of the given set of equations? To find the answer, let’s go back to the set of equations that (2.6.1) stands for. The first of these is

$$
x_3 = c - ax_1 - bx_4
$$

because the numbering of the unknowns is as shown in the $\tau$ array. The next two equations are

$$
x_5 = f - dx_1 - ex_4
$$
$$
x_2 = k - gx_1 - hx_4.
$$

If we add the two trivial equations $x_1 = x_1$ and $x_4 = x_4$, then we get the whole solution vector which, after re-ordering the equations, can be written as

$$
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5
\end{bmatrix} =
\begin{bmatrix}
0 \\
k \\
c \\
0 \\
f
\end{bmatrix} + (-x_1) \ast
\begin{bmatrix}
-1 \\
g \\
a \\
0 \\
d
\end{bmatrix} + (-x_4) \ast
\begin{bmatrix}
0 \\
h \\
b \\
-1 \\
e
\end{bmatrix}.
$$

(2.6.2)

Now we are looking at a display of the output as we would like our subroutine to give it. The three vectors on the right side of (2.6.2) are, respectively, a particular solution of the given system of equations, and the two vectors of a basis for the kernel of the coefficient matrix.

The question can now be rephrased: exactly what operations must be done to the matrix shown in (2.6.1) that represents the situation at the end of the back solution, in order to obtain the three vectors in (2.6.2)?

The first things to do are, as we have previously noted, to append the negative of a $2 \times 2$ identity matrix to the bottom of the fourth and fifth columns of (2.6.1), and to lengthen the last column on
the right by appending two more zeros. That brings us to the matrix

\[
\begin{bmatrix}
1 & 0 & 0 & a & b & c \\
0 & 1 & 0 & d & e & f \\
0 & 0 & 1 & g & h & k \\
-1 & 0 & 0 & -1 & 0 \\
|3 & 5 & 2 & 1 & 4|
\end{bmatrix}
\]

(2.6.3)

The first two of the three long columns above will be the basis for the kernel, and the last column above will be the particular solution, but only after we do the right rearrangement.

Now here is the punch line: the right rearrangement to do is to permute the rows of those three long columns as described by the permutation \( \tau \).

That means that the first row becomes the third, the second row becomes the fifth, the third row becomes the second, the fourth row is the new first, and the old fifth row is the new fourth. The reader is invited to carry out on the rows the interchanges just described, and to compare the result with what we want, namely with (2.6.2.). It will be seen that we have gotten the desired result.

The point that is just a little surprising is that to undo the column interchanges that are recorded by \( \tau \), we do row interchanges. Just roughly, the reason for this is that we begin by wanting to solve \( A\vec{x} = \vec{b} \), and instead we end up solving \( (AE)\vec{y} = \vec{b} \), where \( E \) is a matrix obtained from the identity by elementary column operations. Evidently, \( \vec{x} = E\vec{y} \), which means that we must perform row operations on \( \vec{y} \) to recover the answers in the right order.

Now we can leave the example above, and state the rule in general. We are given \( p \) systems of \( m \) simultaneous equations each, all having a common \( m \times n \) coefficient matrix \( A \), in \( n \) unknowns. At the end of the back solution we will have before us a matrix of the form

\[
\begin{bmatrix}
I(r, r) & B(r, n-r) & P(r, p)
\end{bmatrix}
\]

where \( I(r, r) \) is the \( r \times r \) identity matrix, \( r \) is the pseudorank of \( A \), and \( B \) and \( P \) are matrices of the sizes shown.

We adjoin under \( B \) the negative of the \( (n-r) \times (n-r) \) identity matrix, and under \( P \) we adjoin an \( (n-r) \times p \) block of zeros. Next, we forget the identity matrix on the left, and we consider the entire remaining \( n \times (n-r+p) \) matrix as a whole, call it \( T \), say. Now we exchange the rows of \( T \) according to the permutation array \( \tau \). Precisely, row 1 of \( T \) will be row \( \tau_1 \) of the new \( T \), row 2 will be row \( \tau_2 \), .... Conceptually, we should regard the old \( T \) and the new \( T \) as occupying different areas of storage, so that the new \( T \) is just a rearrangement of the rows of the old.

Now the first \( n-r \) columns of the new \( T \) are a basis for the kernel of \( A \), and should be output as such, and the \( j \)th one of the last \( p \) columns of the new \( T \) is a particular solution of the \( j \)th one of the input systems of equations, and should be output as such.

Although conceptually we should think of the old \( T \) and the new \( T \) as occupying distinct arrays in memory, in fact it is perfectly possible to carry out the whole row interchange procedure described above in just one array, the one that holds \( T \), without ever “stepping on our own toes”, so let’s consider that problem.

Suppose a linear array \( a = [a_1, \ldots, a_n] \) is given, along with a permutation array \( \tau = [\tau_1, \ldots, \tau_n] \). We want to rearrange the entries of the array \( a \) according to the permutation \( \tau \) without using any additional array storage. Thus the present array \( a_1 \) will end up as the output \( a_{\tau_1} \), the initial \( a_2 \) will end up as \( a_{\tau_2} \), etc.
To do this with no extra array storage, let’s first pick up the element \( a_1 \) and move it to \( a_{\tau_1} \), being careful to store the original \( a_{\tau_1} \) in a temporary location \( t \) so it won’t be destroyed. Next we move the contents of \( t \) to its destination, and so forth. After a certain number of steps (maybe only 1), we will be back to \( a_1 \).

Here’s an example to help clarify the situation. Suppose the arrays \( a \) and \( \tau \) at input time were:

\[
\begin{align*}
a &= [5, 7, 13, 9, 2, 8] \\
\tau &= [3, 4, 5, 2, 1, 6].
\end{align*}
\]

So we move the 5 in position \( a_1 \) to position \( a_3 \) (after putting the 13 into a safe place), and then the 13 goes to position \( a_5 \) (after putting the 2 into a safe place) and the 2 is moved into position \( a_1 \), and we’re back where we started. The \( a \) array now has become

\[
a = [2, 7, 5, 9, 13, 8].
\]

The job, however, is not finished. Somehow we have to recognize that the elements \( a_2, a_4 \) and \( a_6 \) haven’t yet been moved, while the others have been moved to their destinations. For this purpose we will flag the array positions. A convenient place to hang a flag is in the sign position of an entry of the array \( \tau \), since we’re sure that the entries of \( \tau \) are all supposed to be positive. Therefore, initially we’ll change the signs of all of the entries of \( \tau \) to make them negative. Then as elements are moved around in the \( a \) array we will reverse the sign of the corresponding entry of the \( \tau \) array. In that way we can always begin the next block of entries of \( a \) to move by searching \( \tau \) for a negative entry. When none exist, the job is finished.

Here’s a complete algorithm, in Maple:

```maple
cshuffle:= proc(a,tau,n)
local i,j,t,q,u,v;
#permutes the entries of a according to the permutation tau
#
# flag entries of tau with negative signs
for i from 1 to n do tau[i]:=-tau[i] od;
for i from 1 to n do
# has entry i been moved?
if tau[i]<0 then
# move the block of entries beginning at a[i]
t:=i; q:=-tau[i]; tau[i]:=q;
u:=a[i]; v:=u;
while q<t do
v:=a[q]; a[q]:=u; tau[q]:=tau[q];
u:=v; q:=tau[q] od;
    a[t]:=v;
fi;
    od;
return(1);
end;
```

The reader should carefully trace through the complete operation of this algorithm on the sample arrays shown above. In order to apply the method to the linear equation solving program, the entries \( C[r+1, i], i = 1, \ldots, n \) are interpreted as \( \tau_i \), and the array \( a \) of length \( n \) whose entries are going to be moved is one of the columns \( r+1, \ldots, n+p \) of the matrix \( C \) in rows 1, \ldots, \( n \).