

Now that you see how solving linear equations works, try another set of equations:

$$\begin{aligned}x + 2y + z &= 8 \\ 2x + 3y - 2z &= 2 \\ 3x + 5y - z &= 10\end{aligned}\tag{1.9}$$

i.e., you enter,

```
A = [1 2 1; 2 3 -2; 3 5 -1]
b = [8; 2; 10]
x = A\b
```

Whoops! You get an error message:

```
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 1.171932e-017
```

(Note that your result for RCOND might be a little different, but generally very small.)

What happened? Well, look closely at the original set of equations. The third equation is really not very useful, since it is the sum of the first two equations. It is not linearly independent of the other two. You have in effect two equations in three unknowns, which is therefore not solvable. This is seen in the structure of the matrix. The error message arises when you try to invert the matrix because it is *rank deficient*. If you look at its rank, with `rank(A)`, you get a value of 2, which is less than the full dimensionality (3) of the matrix. If you did that for the first matrix we looked at, by typing `rank(AA)`, it would return a value of 3.

Remember our friend the determinant? Try `det(A)` again. What value do you get? Zero. If the determinant of the inverse of A is the inverse of the determinant of A (get it?), then guess what happens? A matrix with a determinant of zero is said to be *singular*.

1.4.2 Singular value decomposition (SVD)

Common sense tells you to quit right there. Trying to solve two equations with three unknowns is not useful ... or is it? There are an infinite number of combinations of x , y ,

and z that satisfy the equations, but not every combination will work. Sometimes it is of value to know what the range of values is, or to obtain some solution subject to some other (as yet to be defined) criteria or conditions. We will tell you of a sure-fire technique to do this, *singular value decomposition*. Here's how it works.

You can split a scalar into an infinite number of factors. For example, you can represent the number 12 in the following ways:

- 2×6
- 3×4
- 1×12
- $2 \times 3 \times 2$
- $1 \times 6 \times 2$
- 0.5×24
- 1.5×8
- $1\frac{2}{3} \times 7.20$
- $24 \times 1 \times 2 \times 0.25$

and so on. You can even require (constrain) one or more of these factors to be even, integer, etc. Well, the same is true for matrices. This leads to a host of different techniques called *decomposition*. You'll hear of various ones, including "LU", "QR" and so on. Their main purpose is to get the matrices in a form that is useful for solving equations, or eliminating parts of the matrix. They are all used by MATLAB but the one you should know about is SVD (which is short for singular value decomposition). It is nothing magic, but allows you to break a matrix down into three very useful components. The following can be "proved" or "demonstrated" with several pages of matrix algebra, but we will just make an assertion. We assert that for any matrix (**A**) of dimension $N \times M$ (N rows, M columns), there exists a triple product of matrices:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}' \quad (1.10)$$

where:

U is column orthonormal (i.e. each column as a vector is orthogonal to the others and of "unit length" – sum of squares of elements is 1) and of size $N \times M$,

S is a diagonal matrix of dimension $M \times M$, whose diagonal elements are called *singular values*. These values may be zero if the matrix is *rank deficient* (i.e. its rank is less than the shortest dimension of the matrix),

V is an orthonormal square matrix of size $M \times M$. In linear algebra and MATLAB \mathbf{V}' means **V** *transpose* where you swap rows and columns, i.e. if

$$\mathbf{G} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad \text{then} \quad \mathbf{G}' = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \quad (1.11)$$

Note that there are two ways of defining the size of the matrices, you may come across the other in Strang's (2005) book, but the results are the same when you multiply them

out. The actual procedure for calculating the SVD is pretty long and tedious, but it always works regardless of the form of the matrix. You accomplish this for the first matrix in MATLAB in the following way:

```
[U,S,V]=svd(AA,0);
```

That's how we get more than one thing back from a MATLAB function call; you line them up inside a set of brackets separated by commas on the *left-hand side* (LHS) of the equation. You can get this information by typing `help svd`. Note also that we have included a `0` after the `AA`. This selects a special (and more useful to us) form of the SVD output. To look at any of the matrices, simply type its name. For example, let's look at `S` by typing `S`. Note that for the matrix `AA`, which we had no trouble with, all three *singular values* are non-zero.

```
S =
```

```
5.1623      0      0
      0  3.0000      0
      0      0  1.1623
```

Now try it with the other, troublesome matrix:

```
[U,S,V]=svd(A,0);
```

and after typing `S`, you can see that the lowest right-hand element is zero. This is the trouble spot!

```
S =
```

```
7.3728      0      0
      0  1.9085      0
      0      0  0.0000
```

Now, we don't need to go into the details, but it can be proven that you can construct the matrix inverse from the relation $\text{inv}(A) = V \cdot W \cdot U'$, as we would write it in MATLAB, where `W` is just `S` with the diagonal elements inverted (each element is replaced by its inverse). For a rank deficient matrix (like the one we had trouble with), at least one of the diagonal elements in `S` is zero. In fact, the number of non-zero singular values is the *rank* of the matrix. Thus if you went ahead and blindly inverted a zero element, you'd have an infinity. The trick is to replace the inverse of the zero element with zero, not infinity. Doing that allows you to compute an inverse anyway.

We can do this *inversion* in MATLAB in the following way. First replace any zero elements by 1. You convert the diagonal matrix to a single column vector containing the diagonal elements with:

```
s=diag(S)
```

(note the lower and upper case usage). Then set the zero element to 1 with:

```
s(3)=1
```

Then invert the elements with:

```
w=1./s
```

(note the decimal point, which means do the operation on each element, as an “array operation” rather than a “matrix operation”). Next, make that pesky third element *really* zero with:

```
w(3)=0;
```

Then, convert it back to a diagonal matrix with:

```
W=diag(w)
```

Note that MATLAB is smart enough to know that you are handing it a column vector and to convert it to a diagonal matrix (it did the opposite earlier on). Now you can go ahead and calculate the best-guess inverse of A with:

```
BGI=V*W*U'
```

where BGI is just a new variable name for this inverted matrix. Now, try it out with:

```
A*BGI
```

Bet you were expecting something like the identity matrix. Instead you get:

```
0.6667    -0.3333     0.3333
-0.3333     0.6667     0.3333
0.3333     0.3333     0.6667
```

Why isn't it identity? Well, the answer to that question gets to the very heart of *inverse theory*, and we'll get to that later in this book (Chapter 18). For now we just want you to note the symmetry of the answer to $A*BGI$ (i.e., the 0.6667 down the diagonal with a positive or negative 0.3333 everywhere else).

Now, let's get down to business, and get a solution for the equation set. We compute the solution with:

```
x=BGI*b
```

which is:

$$x = \begin{pmatrix} 0.7879 \\ 2.1212 \\ 2.9697 \end{pmatrix} \tag{1.12}$$

Do you believe the results? Well, try them with:

```
A*x
```

which gives you the original \mathbf{b} ! But why this solution? For example, $\mathbf{x} = [1 \ 2 \ 3]'$ works fine too (entering the numbers without the semicolons gives you a row vector, and the prime turns a row vector into a column vector, its transpose). Well, the short answer is because, of all of the possible solutions, this one has the shortest *length*. Check it out: the square root of the sum of the squares of the components of $[1 \ 2 \ 3]'$ is longer than the vector you get from $\mathbf{B} \backslash \mathbf{b}$. The reason is actually an important attribute of SVD, but more explanations will have to wait for Chapter 18.

Also, note that the singular values are arranged in order of decreasing value on page 11. This doesn't have to be the case, but the MATLAB algorithm does this to be nice to you. Also, the singular values to some extent tell you about the *structure* of the matrix.

Not all cases are as clear-cut as the two we just looked at. A matrix may be nearly singular, so that although you get an “answer,” it may be subject to considerable uncertainties and not particularly robust. The ratio of the largest to the smallest singular values is the *condition number* of the matrix. The larger it is, the worse (more singular) it is. You can get the condition number of a matrix by entering:

```
cond(A)
```

In fact, MATLAB uses SVD to calculate this number. And `RCOND` is just its reciprocal.

So what have we learned? We've learned about matrices, and how they can be used to represent and solve systems of equations. We have a technique (SVD) that allows us to calculate, under any circumstances, the inverse of a matrix. With the inverse of the matrix, we can then solve a set of simultaneous equations with a very simple step, even if there is no unique answer. But wait, there's more! Stay tuned ...