Solving general linear equations using Matlab

In this note we consider the following problem: Determine whether there is a solution \( x \in \mathbb{R}^n \) of the (set of) \( m \) linear equations \( Ax = b \), and if so, find one. To check existence of a solution is the same as checking if \( b \in \mathcal{R}(A) \). We consider here the general case, with \( A \in \mathbb{R}^{m \times n} \), with \( \text{Rank}(A) = r \). In particular, we do not assume that \( A \) is full rank.

Existence of solution via rank

A simple way to check if \( b \in \mathcal{R}(A) \) is to check the rank of \([A \ b]\), which is either \( r \) (i.e., the rank of \( A \)) if \( b \in \mathcal{R}(A) \), or \( r + 1 \), if \( b \notin \mathcal{R}(A) \). This can be checked in Matlab using

\[
\text{rank}([A \ b]) == \text{rank}(A)
\]

(or evaluating the two ranks separately and comparing them). If the two ranks above are equal, then \( Ax = b \) has a solution. But this method does not give us a solution, when one exists. This method also has a hidden catch: Matlab uses a numerical tolerance to decide on the rank of a matrix, and this tolerance might not be appropriate for your particular application.

Using the backslash and pseudo-inverse operator

In Matlab, the easiest way to determine whether \( Ax = b \) has a solution, and to find such a solution when it does, is to use the backslash operator. Exactly what \( A\backslash b \) returns is a bit complicated to describe in the most general case, but if there is a solution to \( Ax = b \), then \( A\backslash b \) returns one. A couple of warnings: First, \( A\backslash b \) returns a result in many cases when there is no solution to \( Ax = b \). For example, when \( A \) is skinny and full rank (i.e., \( m > n = r \)), \( A\backslash b \) returns the least-squares approximate solution, which in general is not a solution of \( Ax = b \) (unless we happen to have \( b \in \mathcal{R}(A) \)). Second, \( A\backslash b \) sometimes causes a warning to be issued, even when it returns a solution of \( Ax = b \). This means that you can’t just use the backslash operator: you have to check that what it returns is a solution. (In any case, it’s just good common sense to check numerical computations as you do them.) In Matlab this can be done as follows:

\[
x = A\backslash b; \quad \% \text{possibly a solution to } Ax=b\\
norm(A*x-b) \% \text{if this is zero or very small, we have a solution}
\]

If the second line yields a result that is not very small, we conclude that \( Ax = b \) does not have a solution. Note that executing the first line might cause a warning to be issued.

In contrast to the rank method described above, you decide on the numerical tolerance you’ll accept (i.e., how small \( \|Ax-b\| \) has to be before you accept \( x \) as a solution of \( Ax = b \)). A common test that works well in many applications is \( \|Ax-b\| \leq 10^{-5}\|b\| \).
You can also use the pseudo-inverse operator: \( x = \text{pinv}(A) \ast b \) is also guaranteed to solve \( Ax = b \), if \( Ax = b \) has a solution. As with the backslash operator, you have to check that the result satisfies \( Ax = b \), since in general, it doesn’t have to.

**Using the QR factorization**

While the backslash operator is a convenient way to check if \( Ax = b \) has a solution, it’s a bit opaque. Here we describe a method that is transparent, and can be fully explained and understood using material we’ve seen in the course.

We start with the full QR factorization of \( A \) with column permutations:

\[
AP = QR = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 & R_2 \\ 0 & 0 \end{bmatrix}.
\]

Here \( Q \in \mathbb{R}^{m \times m} \) is orthogonal, \( R \in \mathbb{R}^{m \times n} \) is upper triangular, and \( P \in \mathbb{R}^{n \times n} \) is a permutation matrix. The submatrices have the following dimensions: \( Q_1 \in \mathbb{R}^{m \times r} \), \( Q_2 \in \mathbb{R}^{m \times (m-r)} \), \( R_1 \in \mathbb{R}^{r \times r} \) is upper triangular with nonzero elements along its main diagonal, and \( R_2 \in \mathbb{R}^{r \times (n-r)} \). The zero submatrices in the bottom (block) row of \( R \) have \( m - r \) rows.

Using \( A = QRP^T \) we can write \( Ax = b \) as

\[
QRP^T x = QRz = b,
\]

where \( z = P^T x \). Multiplying both sides of this equation by \( Q^T \) gives the equivalent set of \( m \) equations \( Rz = Q^T b \). Expanding this into subcomponents gives

\[
Rz = \begin{bmatrix} R_1 & R_2 \\ 0 & 0 \end{bmatrix} z = \begin{bmatrix} Q_1^T b \\ Q_2^T b \end{bmatrix}.
\]

We see immediately that there is no solution of \( Ax = b \), unless we have \( Q_2^T b = 0 \), because the bottom component of \( Rz \) is always zero.

Now let’s assume that we do have \( Q_2^T b = 0 \). Then the equations reduce to

\[
R_1 z_1 + R_2 z_2 = Q_1^T b,
\]

a set \( r \) linear equations in \( n \) variables. We can find a solution of these equations by setting \( z_2 = 0 \). With this form for \( z \), the equation above becomes \( R_1 z_1 = Q_1^T b \), from which we get \( z_1 = R_1^{-1} Q_1^T b \). Now we have a \( z \) that satisfies \( Rz = Q^T b \): \( z = [z_1^T, 0]^T \). We get the corresponding \( x \) from \( x = Pz \):

\[
x = P \begin{bmatrix} R_1^{-1} Q_1^T b \\ 0 \end{bmatrix}.
\]

This \( x \) satisfies \( Ax = b \), provided we have \( Q_2^T b = 0 \). Whew.

Actually, the construction outlined above is pretty much what \( A \backslash b \) does.
In Matlab, we can carry out this construction as follows:

```
[m,n]=size(A);
[Q,R,P]=qr(A); % full QR factorization
r=rank(A); % could also get rank directly from QR factorization ...

% construct the submatrices
Q1=Q(:,1:r);
Q2=Q(:,r+1:m);
R1=R(1:r,1:r);

% check if b is in range(A)
norm(Q2'*b) % if this is zero or very small, b is in range(A)

% construct a solution
x=P*[R1\(Q1'*b); zeros(n-r,1)]; % satisfies Ax=b, if b is in range(A)

% check alleged solution (just to be sure)
norm(A*x-b)
```