Chapter 2 - *Linear Equations*

2.1 Solving Linear Equations

One of the most common problems in scientific computing is the solution of linear equations.

It is a problem in its own right, but it also occurs as a sub-problem for many other problems.

For example, as we will see, the solution of a *nonlinear* system of equations is typically approximated through a *sequence of solutions of linear systems*.

The discretization of a linear elliptic partial differential equation typically leads to the solution of a (highly structured) linear system of equations.

Here is a simple example: Three unknowns x, y, and z satisfy

$$\begin{array}{rcrcrcr} 2x & + & 4y & - & 2z & = & 2, \\ 4x & + & 9y & - & 3z & = & 8, \\ -2x & - & 3y & + & 7z & = & 10. \end{array}$$

The goal is to find the magical values of x, y, z that satisfy all 3 of the above equations.

Before we talk about strategies to solve linear systems, we start by standardizing the notation to generalize the problem to m equations in m unknowns.

Relabel the unknowns $\mathbf{x} = (x_1, x_2, \ldots, x_m)^T$, the coefficients $a_{11}, a_{21}, \ldots, a_{m1}$, $a_{21}, a_{22}, \ldots, a_{2m}$, $\ldots, a_{m1}, a_{m2}, \ldots, a_{mm}$, and the right-hand side values $\mathbf{b} = (b_1, b_2, \ldots, b_m)^T$.

$$
\begin{aligned}
a_{11}x_1 + a_{21}x_2 + \ldots + a_{1m}x_m &= b_1, \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2m}x_m &= b_2, \\
&\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mm}x_m &= b_m.
\end{aligned}
$$

In matrix notation:

$$\mathbf{A}\mathbf{x} = \mathbf{b}.$$

Formally, the solution to this problem is

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

However, in practice we *never* actually find the inverse of $\mathbf{A}$ and then multiply it by $\mathbf{b}$ to get $\mathbf{x}$!

Suppose we wish to solve the easy equation $7x_1 = 21$.

If we find the inverse of 7 and multiply it by 21 to get x_1, this requires 2 floating-point operations and suffers from roundoff error:

$$x_1 = 7^{-1} \times 21 \approx 0.142857 \times 21 \approx 2.99997.$$

The cheapest and most accurate way to solve this equation is by division:

$$x_1 = \frac{21}{7} = 3.$$

Although this example may look artificial, these principles extend to larger systems, where they become even more important!

2.2 The \ Operator

To emphasize the distinction between solving linear equations and computing inverses, MATLAB uses the backward slash (or *backslash*) operator \.

If $\mathbf{A}$ is a matrix of any size and shape and $\mathbf{b}$ is a vector with as many rows as $\mathbf{A}$, then the solution to the system of simultaneous equations $\mathbf{Ax} = \mathbf{b}$ can be computed in MATLAB by `x = A\b`.

If it helps, you can think of this as "left-dividing" both sides of the equation by the coefficient matrix A.

$\rightarrow$ Because matrix multiplication is not commutative and $\mathbf{A}$ occurs on the left in the original equation, this is left division.

Note 1. *This operator can be applied even if* $\mathbf{A}$ *is not square; i.e., the number of equations is not the same as the number of unknowns.*

However, in this course, we always assume $\mathbf{A}$ is square.

Note 2. *Solving the linear system* $\mathbf{Ax} = \mathbf{b}$ *for multiple right-hand sides* $\mathbf{b}_1$, $\mathbf{b}_2$, . . . , $\mathbf{b}_p$, *can be done all at once by defining* $\mathbf{B} = [\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_p]$ *and using the command* `X = A\B`.

The answers come out in matrix $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_p]$.

2.3 Simple Examples

The simplest kinds of linear systems to solve are ones where the unknowns are *decoupled*. For example,

$$0.3\,x_1 = 3, \qquad 5\,x_2 = 1.5, \qquad 2\,x_3 = -4.$$

In matrix form,

$$\begin{bmatrix} 0.3 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 1.5 \\ -4 \end{bmatrix}.$$

Note that the coefficient matrix is *diagonal*.

In this case, the unknowns can be solved for *independently*; e.g., in any order, or in batches if you have a parallel computer.

It is possible to reduce a non-singular matrix to diagonal form and then solve for the unknowns; this is known as *Gauss–Jordan elimination*.

In practice, Gauss–Jordan elimination is more expensive than another method (Gaussian elimination) that works just as well.

Gaussian elimination is based on a systematic[1] procedure to reduce the coefficient matrix $\mathbf{A}$ to *upper-triangular form*.

What's the advantage of upper-triangular form?

It turns out that you can solve for the unknowns in an upper-triangular system of linear equations one after the other.

We will see this in the next section; but for future reference and to appreciate the concept of equation manipulation as matrix multiplication, we first look at permutation and triangular matrices.

[1] So that it can be programmed!

2.4.1 Permutation Matrices

Sometimes you may want to interchange rows or columns of a matrix. This operation can be represented by matrix multiplication by a *permutation matrix*.

Definition 1. *A* permutation matrix $\mathbf{P}$ *is a matrix made by taking the identity matrix and interchanging its rows or columns.*

In other words, $\mathbf{P}$ has exactly one 1 in each row and column; all the other elements are 0.

Here is an example of a permutation matrix:

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

How is it related to the 4×4 identity matrix $\mathbf{I}$?

Row 4 of $\mathbf{I}$ is Row 1 of $\mathbf{P}$; Row 1 of $\mathbf{I}$ is Row 2 of $\mathbf{P}$; Row 3 of $\mathbf{I}$ is Row 3 of $\mathbf{P}$; Row 2 of $\mathbf{I}$ is Row 4 of $\mathbf{P}$.

$\Longrightarrow$ The matrix $\mathbf{PA}$ has $\mathbf{A}$'s Row 4 as its Row 1, $\mathbf{A}$'s Row 1 as its Row 2, $\mathbf{A}$'s Row 3 as its Row 3, and $\mathbf{A}$'s Row 2 as its Row 4.

Similarly,

Col 2 of $\mathbf{I}$ is Col 1 of $\mathbf{P}$; Col 4 of $\mathbf{I}$ is Col 2 of $\mathbf{P}$;
Col 3 of $\mathbf{I}$ is Col 3 of $\mathbf{P}$; Col 1 of $\mathbf{I}$ is Col 4 of $\mathbf{P}$.

$\Longrightarrow$ The matrix $\mathbf{AP}^T$ has $\mathbf{A}$'s Col 4 as its Col 1, $\mathbf{A}$'s Col 1 as its Col 2, $\mathbf{A}$'s Col 3 as its Col 3, and $\mathbf{A}$'s Col 2 as its Col 4.

Note 3. *A compact way to store* $\mathbf{P}$ *(or simply keep track of permutations) is in a vector.*

e.g., The row interchanges represented by $\mathbf{P}$ in our example can be stored as the vector `p = [4 1 3 2]`.

A quick way to create a matrix $\mathbf{B}$ from interchanging the *rows and columns* of $\mathbf{A}$ via `p` is `B = A(p,p)`.

Permutation matrices are *orthogonal matrices*; i.e., they satisfy

$$\mathbf{P}^T\mathbf{P} = \mathbf{I}, \quad \text{or equivalently} \quad \mathbf{P}^{-1} = \mathbf{P}^T.$$

2.4.2 Triangular Matrices

An *upper-triangular matrix* has all its nonzero elements on or above the main diagonal.

A *lower-triangular matrix* has all its nonzero elements on or below the main diagonal.

A *unit* lower-triangular matrix is a lower-triangular matrix with ones on the main diagonal.

Here are examples of an upper-triangular matrix $\mathbf{U}$ and a unit lower-triangular matrix $\mathbf{L}$:

$$\mathbf{U} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix}, \qquad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix}.$$

Linear equations involving triangular matrices are easy to solve.

We focus on the case for upper-triangular systems because it will be relevant in $\mathbf{LU}$ factorization.

One way to solve an $m \times m$ upper-triangular system, $\mathbf{Ux} = \mathbf{b}$ is to begin by solving the last equation for the last variable, then the next-to-last equation for the next-to-last variable, and so on, using the newly computed values of the solution along the way.

Here is some MATLAB code to do this:

```
x = zeros(m,1);
for k = m:-1:1,
  j = k+1:m;
  x(k) = (b(k) - U(k,j)*x(j))/U(k,k);
end
```

2.4.3 Banded Matrices

A matrix $\mathbf{A}$ is called a banded matrix with lower bandwidth p and upper bandwidth q if

$$a_{i,j} = 0 \qquad \text{if } i > j + p \quad \text{or} \quad j > i + q.$$

The band width of $\mathbf{A}$ is naturally defined as $p + q + 1$.

$\mathbf{A}$ has p sub-diagonals and q super-diagonals, and the band width is the total number of diagonals.

Important special cases of banded matrices are

- diagonal matrices ($p = q = 0$),
  tridiagonal matrices ($p = q = 1$),
  pentadiagonal matrices ($p = q = 2$),
  $\vdots$
  full matrices ($p = q = \lfloor m/2 \rfloor$)

- bidiagonal matrices
  (lower: $(p, q) = (1, 0)$, upper: $(p, q) = (0, 1)$)
  Hessenberg matrices (lower: $q = 1$; upper: $p = 1$)

2.5 $\mathbf{LU}$ Factorization

The $\mathbf{LU}$ factorization is the factorization of a matrix $\mathbf{A}$ into the product of a unit lower-triangular matrix and an upper-triangular matrix; i.e., $\mathbf{A} = \mathbf{LU}$.

The $\mathbf{LU}$ factorization is an interpretation of Gaussian elimination, one of the oldest numerical methods, generally named after Gauss.

It is perhaps the simplest systematic way to solve systems of linear equations by hand, and it is also the standard method for solving them on a computer.

Research between 1955 to 1965 revealed the importance of two aspects of Gaussian elimination that were not emphasized in earlier work: the search for pivots and the proper interpretation of the effect of rounding errors.

Pivoting is essential to the stability of Gaussian elimination!

The first important concept is that the familiar process of eliminating elements below the diagonal in Gaussian elimination can be viewed as matrix multiplication.

The elimination is accomplished by subtracting multiples of each row from subsequent rows.

Before we see the details, let's pretend we know how to do this in a systematic fashion.

The elimination process is equivalent to multiplying $\mathbf{A}$ by a sequence of lower-triangular matrices $\mathbf{L}_k$ on the left

$$\mathbf{L}_{m-1}\mathbf{L}_{m-2}\ldots\mathbf{L}_1\mathbf{A} = \mathbf{U},$$

where $\mathbf{L}_k$ eliminates the elements below the diagonal in row k.

In other words, we keep eliminating elements below the diagonal until $\mathbf{A}$ has been reduced to an upper-triangular matrix, which we call $\mathbf{U}$.

Setting

$$\mathbf{L}^{-1} = \mathbf{L}_{m-1}\mathbf{L}_{m-2}\dots\mathbf{L}_1,$$

or equivalently,

$$\mathbf{L} = \mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\dots\mathbf{L}_{m-1}^{-1},$$

gives an $\mathbf{LU}$ *factorization* of $\mathbf{A}$

$$\mathbf{A} = \mathbf{LU},$$

where $\mathbf{L}$ is lower-triangular and $\mathbf{U}$ is upper-triangular.

In practice, we can choose $\mathbf{L}$ to be *unit lower-triangular*.

Example: Suppose $\mathbf{A}$ is 4×4:

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{\mathbf{L}_1} \begin{bmatrix} \times & \times & \times & \times \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix}$$

$\mathbf{A}$ $\qquad\qquad$ $\mathbf{L}_1\mathbf{A}$

$$\begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \end{bmatrix} \xrightarrow{\mathbf{L}_2} \begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & 0 & * & * \\ & 0 & * & * \end{bmatrix}$$

$\mathbf{L}_1\mathbf{A}$ $\qquad\qquad$ $\mathbf{L}_2\mathbf{L}_1\mathbf{A}$

$$\begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & & \times & \times \\ & & \times & \times \end{bmatrix} \xrightarrow{\mathbf{L}_3} \begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & & \times & \times \\ & & 0 & * \end{bmatrix}$$

$\mathbf{L}_2\mathbf{L}_1\mathbf{A}$ $\qquad\qquad$ $\mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{A}$

$\diamond$ EXAMPLE

Let

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

(This particular $\mathbf{A}$ was chosen because it has a nice $\mathbf{LU}$ decomposition.)

So,

$$\mathbf{L}_1\mathbf{A} = \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix}.$$

(Subtract 2 $\times$ row 1 from row 2, 4 $\times$ row 1 from row 3, and 3 $\times$ row 1 from row 4.)

Further,

$$\mathbf{L}_2\mathbf{L}_1\mathbf{A} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -3 & 1 & \\ & -4 & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix}.$$

(Subtract 3 $\times$ row 2 from row 3 and 4 $\times$ row 2 from row 4.)

Finally,

$$\mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{A} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix} = \mathbf{U}.$$

(Subtract row 3 from row 4.)

To obtain $\mathbf{A} = \mathbf{L}\mathbf{U}$, we need to form

$$\mathbf{L} = \mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1}.$$

This turns out to be easy, thanks to two happy surprises.

$\mathbf{L}_1^{-1}$ can be obtained from $\mathbf{L}_1$ by negating the off-diagonal entries:

$$\mathbf{L}_1^{-1} = \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & & 1 & \\ 3 & & & 1 \end{bmatrix}.$$

(Similarly, the same is true for $\mathbf{L}_2$ and $\mathbf{L}_3$.)

Also, $\mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1}$ is obtained by "merging" the individual non-zero elements.

i.e.,

$$\mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix}.$$

Thus,

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix} = \mathbf{LU}.$$

See `lugui.m`

```
lugui([2 1 1 0; 4 3 3 1; 8 7 9 5; 6 7 9 8], 'diagonal')
```

Note: "diagonal" pivoting = no pivoting.

To formalize the facts that we need (without proof!):

1. In Gaussian elimination, elimination of elements below the diagonal in row k by row operations against row k (the pivot row) can be viewed as left-multiplication by a unit lower-triangular matrix with elements l_{jk} in column k with $l_{jk} = -$(element to eliminate)/(pivot).

2. $\mathbf{L}_k$ can be inverted by negating the off-diagonal elements $-l_{jk}$ (so $\mathbf{L}$ contains l_{jk}),

3. $\mathbf{L}$ can be formed by merging the entries l_{jk}.

Of course, in practice, the matrices $\mathbf{L}_k$ are never formed and multiplied explicitly.
$\rightarrow$ Only the quantities l_{jk} are computed and stored, often in the lower triangle of $\mathbf{A}$, which would otherwise contain zeros upon reduction to $\mathbf{U}$.

How does this help us solve $\mathbf{Ax} = \mathbf{b}$?

Writing this as $\mathbf{LUx} = \mathbf{b}$, this can now be solved by solving *two triangular systems*.

First solve $\mathbf{Ly} = \mathbf{b}$ for an intermediate variable $\mathbf{y}$ (forward substitution).

Then solve $\mathbf{Ux} = \mathbf{y}$ for the unknown variable $\mathbf{x}$ (back substitution).

Once we have $\mathbf{L}$ and $\mathbf{U}$, solving for the unknown $\mathbf{x}$ is quick and easy for any number of right-hand sides $\mathbf{b}$.

Pivoting

Pure Gaussian elimination is unstable.

Fortunately, this instability can be controlled by permuting the rows of $\mathbf{A}$ as we proceed.

This process is called *pivoting*.

Pivoting has been a standard feature of Gaussian elimination computations since the 1950s.

At step k of Gaussian elimination, multiples of row k are subtracted from rows $k+1, k+2, \ldots, m$ of the working matrix $\mathbf{X}$ in order to zero out the elements below the diagonal.

In this operation, row k, column k, and especially x_{kk} play special roles.

We call x_{kk} the *pivot*.

From every entry in the submatrix $\mathbf{X}(k+1:m, k:m)$, we subtract the product of a number in row k and a number in column k, divided by x_{kk}.

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & x_{kk} & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ & x_{kk} & \times & \times & \times \\ & 0 & * & * & * \\ & 0 & * & * & * \\ & 0 & * & * & * \end{bmatrix}.$$

But there is no inherent reason to eliminate against row k.

We could just as easily zero out other entries against row $i,\ k < i \leq m$.

In this case, x_{ik} would be the pivot.

e.g., $k = 2,\ i = 4$:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & x_{ik} & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ & 0 & * & * & * \\ & 0 & * & * & * \\ & x_{ik} & \times & \times & \times \\ & 0 & * & * & * \end{bmatrix}.$$

Similarly, we could zero out the entries in column j instead of column k, $k < j \leq m$.

e.g., $k = 2,\ i = 4,\ j = 3$:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & x_{ij} & \times & \times \\ & \times & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ & * & 0 & * & * \\ & * & 0 & * & * \\ & \times & x_{ij} & \times & \times \\ & * & 0 & * & * \end{bmatrix}$$

Basically, we can choose any entry of $\mathbf{X}(k:m, k:m)$ as the pivot, as long as it is not zero.

This flexibility is good because $x_{kk} = 0$ is possible even in exact arithmetic for non-singular matrices!

In a floating-point number system, x_{kk} may be "numerically" zero.

For stability, we choose as pivot the element with the largest magnitude among the pivot candidates.

Pivoting in this crazy (but smart!) fashion can be confusing.

$\rightarrow$ It is easy to lose track of what has been zeroed and what still needs to be zeroed.

Instead of leaving x_{ij} in place after it is chosen as pivot (as illustrated above) we interchange rows and columns so that x_{ij} takes the position of x_{kk}.

This interchange of rows and/or columns is what is commonly referred to as *pivoting*.

$\rightarrow$ The look of pure Gaussian elimination is maintained.

Note 4. *Elements may or may not actually be swapped in practice!*

We may only keep a list of the positions of the swapped elements.

It is actually possible to search over all the elements in the matrix to find the one with the largest magnitude and swap it into the pivot element.

(This strategy is called *complete pivoting*.)

But this would involve permutations of columns, and that means re-labelling the unknowns.

In practice, pivots of essentially the same quality can be found by searching only within the column being zeroed.
$\rightarrow$ This is known as *partial pivoting*.

Recall: The act of swapping rows can be viewed as left-multiplication by a *permutation matrix* $\mathbf{P}$.

We have seen that the elimination at step k corresponds to left-multiplication by a unit lower-triangular matrix $\mathbf{L}_k$.

So the k^{th} step of Gaussian elimination with partial pivoting can be summed up as

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & x_{ik} & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix} \xrightarrow{\mathbf{P}_k} \begin{bmatrix} \times & \times & \times & \times & \times \\ & x_{ik} & * & * & * \\ & \times & \times & \times & \times \\ & * & * & * & * \\ & \times & \times & \times & \times \end{bmatrix}$$

pivot selection row interchange

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & x_{ik} & * & * & * \\ & \times & \times & \times & \times \\ & * & * & * & * \\ & \times & \times & \times & \times \end{bmatrix} \xrightarrow{\mathbf{L}_k} \begin{bmatrix} \times & \times & \times & \times & \times \\ & x_{ik} & \times & \times & \times \\ & 0 & * & * & * \\ & 0 & * & * & * \\ & 0 & * & * & * \end{bmatrix}$$

row interchange elimination

After $m-1$ steps, $\mathbf{A}$ is transformed into an upper-triangular matrix $\mathbf{U}$:

$$\mathbf{L}_{m-1}\mathbf{P}_{m-1}\ldots\mathbf{L}_2\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1\mathbf{A}=\mathbf{U}.$$

Example: Recall our friend

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

To do Gaussian elimination with partial pivoting proceeds as follows:

Interchange the 1st and 3rd rows (left-multiplication by $\mathbf{P}_1$):

$$\begin{bmatrix} & & 1 & \\ & 1 & & \\ 1 & & & \\ & & & 1 \end{bmatrix}\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}=\begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

The first elimination step now looks like this (left-multiplication by $\mathbf{L}_1$):

$$\begin{bmatrix} 1 & & & \\ -\frac{1}{2} & 1 & & \\ -\frac{1}{4} & & 1 & \\ -\frac{3}{4} & & & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix}$$

$$= \begin{bmatrix} 8 & 7 & 9 & 5 \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \end{bmatrix}.$$

Now the 2nd and 4th rows are interchanged (multiplication by $\mathbf{P}_2$):

$$\begin{bmatrix} 1 & & & \\ & & & 1 \\ & & 1 & \\ & 1 & & \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \end{bmatrix}$$

$$= \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \end{bmatrix}.$$

The second elimination step then looks like this (multiplication by $\mathbf{L}_2$):

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & \frac{3}{7} & 1 & \\ & \frac{2}{7} & & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \end{bmatrix}$$

$$= \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{2}{7} & \frac{4}{7} \\ & & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix}.$$

Now the 3rd and 4th rows are interchanged (multiplication by $\mathbf{P}_3$)

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{2}{7} & \frac{4}{7} \\ & & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix}$$

$$= \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & -\frac{2}{7} & \frac{4}{7} \end{bmatrix}.$$

The final elimination step look like this (multiplication by $\mathbf{L}_3$):

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -\frac{1}{3} & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & -\frac{2}{7} & \frac{4}{7} \end{bmatrix}$$

$$= \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & & \frac{2}{3} \end{bmatrix}.$$

$\diamond$ $\mathbf{PA} = \mathbf{LU}$ AND A THIRD HAPPY SURPRISE

Notice that if you form $\mathbf{L}$ and then $\mathbf{LU}$ from the previous example, you don't get $\mathbf{LU} = \mathbf{A}$!

Was all of our work for nothing?

No! It turns out that we can manipulate this information to get a permuted version of $\mathbf{A}$.

(This might be understandable because we have been messing around with pivots.)

In fact,

$$\mathbf{LU} = \mathbf{PA},$$

where

$$\begin{bmatrix} 1 & & & \\ \frac{3}{4} & 1 & & \\ \frac{1}{2} & -\frac{2}{7} & 1 & \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & & \frac{2}{3} \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

Note 5. ***IMPORTANT***
The entries of $\mathbf{L}$ *all satisfy* $|l_{ij}| \leq 1$.
This is a consequence of pivoting
($\leftrightarrow$ *eliminating against* $|x_{kk}| = \max_j |x_{jk}|$*).*

But given the way the permutations were introduced, it is not obvious as to why all of them can be lumped into one big $\mathbf{P}$;

i.e.,

$$\mathbf{L}_3\mathbf{P}_3\mathbf{L}_2\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1\mathbf{A} = \mathbf{U}.$$

Here is where we use a third stroke of good luck:

These (elementary) operations can be reordered as follows:

$$\mathbf{L}_3\mathbf{P}_3\mathbf{L}_2\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1 = \mathbf{L}'_3\mathbf{L}'_2\mathbf{L}'_1\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1,$$

where $\mathbf{L}'_k$ =($\mathbf{L}_k$ with subdiagonal entries permuted).

To be precise,

$$\begin{aligned}
\mathbf{L}'_3 &= \mathbf{L}_3, \\
\mathbf{L}'_2 &= \mathbf{P}_3\mathbf{L}_2\mathbf{P}_3^{-1}, \\
\mathbf{L}'_1 &= \mathbf{P}_3\mathbf{P}_2\mathbf{L}_1\mathbf{P}_2^{-1}\mathbf{P}_3^{-1}.
\end{aligned}$$

Note 6. *Each* $\mathbf{P}_j$ *has* $j > k$ *for* $\mathbf{L}_k$.
$\Longrightarrow$ $\mathbf{L}'_k$ *has the same structure of* $\mathbf{L}_k$! (verify!)

Thus,

$$\begin{aligned}\mathbf{L}_3'\mathbf{L}_2'\mathbf{L}_1'\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1 &= \mathbf{L}_3(\mathbf{P}_3\mathbf{L}_2\mathbf{P}_3^{-1})(\mathbf{P}_3\mathbf{P}_2\mathbf{L}_1\mathbf{P}_2^{-1}\mathbf{P}_3^{-1})\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1 \\ &= \mathbf{L}_3\mathbf{P}_3\mathbf{L}_2\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1.\end{aligned}$$

In the general $m \times m$ case, Gaussian elimination with partial pivoting can be written as

$$(\mathbf{L}_{m-1}' \ldots \mathbf{L}_2'\mathbf{L}_1')(\mathbf{P}_{m-1} \ldots \mathbf{P}_2\mathbf{P}_1)\mathbf{A} = \mathbf{U},$$

where

$$\mathbf{L}_k' = \mathbf{P}_{m-1} \ldots \mathbf{P}_{k+1}\mathbf{L}_k\mathbf{P}_{k+1}^{-1} \ldots \mathbf{P}_{m-1}^{-1}.$$

Now we write

$$\mathbf{L} = (\mathbf{L}_{m-1}' \ldots \mathbf{L}_2'\mathbf{L}_1')^{-1}$$

and

$$\mathbf{P} = (\mathbf{P}_{m-1} \ldots \mathbf{P}_2\mathbf{P}_1)$$

to get

$$\mathbf{PA} = \mathbf{LU}.$$

Note 7. *In general, any square matrix (whether non-singular or not) has a factorization* $\mathbf{PA} = \mathbf{LU}$*, where* $\mathbf{P}$ *is a permutation matrix,* $\mathbf{L}$ *is unit lower-triangular and whose elements satisfy* $|l_{ij}| \leq 1$*, and* $\mathbf{U}$ *is upper-triangular.*

Despite this being a bit of an abuse, this factorization is really what is meant by the $\mathbf{LU}$ factorization of $\mathbf{A}$.

The formula $\mathbf{PA} = \mathbf{LU}$ has another great interpretation:

If you could permute $\mathbf{A}$ *ahead of time* using matrix $\mathbf{P}$, then you could apply Gaussian elimination to $\mathbf{A}$ and you would not need pivoting!

Of course, this cannot be done in practice because $\mathbf{P}$ is not known a priori.

See `lugui.m`

See also `pivotgolf.m`

Solution of Banded Systems

Banded systems arise fairly often in practice; they can be solved by a streamlined version Gaussian elimination that only operates within the bandwidth.

In fact, the streamlined solution of tridiagonal systems by (pure) Gaussian elimination is occasionally called the *Thomas algorithm*.

In general, a solver would need to be told that the input coefficient matrix $\mathbf{A}$ is banded, but MATLAB can detect bandedness automatically.

We now explore the algorithm to solve a tridiagonal linear system by Gaussian elimination with pivoting.

Note: Complete pivoting destroys the bandedness, causing *fill-in*, but partial pivoting preserves it.

Fill-in refers to the creation of non-zero elements where (structurally) zero elements used to be; it is undesirable because the newly created elements must be stored and operated on (they are generally not zero!).

Solution of Tridiagonal Systems

We begin with the augmented matrix

$$\mathbf{T} = \left[\begin{array}{cccccc|c} b_1 & c_1 & 0 & 0 & 0 & 0 & d_1 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 & d_2 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 & d_3 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & \vdots \\ 0 & 0 & 0 & a_{m-1} & b_{m-1} & c_{m-1} & d_{m-1} \\ 0 & 0 & 0 & 0 & a_m & b_m & d_m \end{array}\right].$$

We can visualize the fill-in produced by pivoting by using `lugui`:

```
m = 6;
lugui(diag(ones(m-1,1),-1) + 2* diag(ones(m,1)) ...
    + diag(ones(m-1,1),1), option)
```

where `option = 'partial'` or `'complete'`.

The Thomas Algorithm

We now present a rendition of the Thomas algorithm without pivoting (for simplicity).

The key to the algorithm is to notice that every step of Gaussian elimination works on a system like

$$\begin{aligned} b_i x_i + c_i x_{i+1} &= d_i, \\ a_{i+1} x_i + b_{i+1} x_{i+1} + c_{i+1} x_{i+2} &= d_{i+1}. \end{aligned}$$

Zeroing below the diagonal only involves one element and requires a multiplier $-a_{i+1}/b_i$, so that the next system to solve is

$$\begin{aligned} \tilde{b}_{i+1} x_{i+1} + c_{i+1} x_{i+2} &= \tilde{d}_{i+1}, \\ a_{i+2} x_{i+1} + b_{i+2} x_{i+2} + c_{i+2} x_{i+3} &= d_{i+2}, \end{aligned}$$

where $\tilde{b}_{i+1} = b_{i+1} - a_{i+1} c_i / b_i$ and $\tilde{d}_{i+1} = d_{i+1} - a_{i+1} d_i / b_i$.

The Thomas Algorithm

Thus, the new system is of the same form as the previous one, so the procedure can be repeated.

This leads to the following algorithm:

```
% forward substitution
```

$\tilde{b}_1 = b_1$; $\tilde{d}_1 = d_1$;

```
for i = 2:m,
```

$l = a_i / \tilde{b}_{i-1}$;

$\tilde{b}_i = b_i - l\, c_{i-1}$;

$\tilde{d}_i = d_i - l\, \tilde{d}_{i-1}$;

```
% backward substitution
```

$x_m = \tilde{d}_m / \tilde{b}_m$;

```
for i = m-1:-1:1,
```

$x_i = (\tilde{d}_i - c_i\, x_{i-1}) / \tilde{b}_i$;

The Thomas Algorithm

In practice, the "tilde" variables are not stored; the original variables are over-written.

To avoid additionally complicating the translation of the algorithm to code, we are not storing the multipliers l, but they could easily be stored in the lower diagonal.

See

thomasAlgorithmDemo.m

2.6 Why Is Pivoting Necessary?

Recall, the diagonal elements of $\mathbf{U}$ are called *pivots*.

Formally, the kth pivot is the coefficient of the kth variable in the kth equation at the kth step of the elimination.

Both the computation of the multipliers and the back substitution require divisions by the pivots.

So, Gaussian elimination fails if any of the pivots are zero, *but the problem may still have a perfectly well defined solution*.

It is a also bad idea to complete the computation if any of the pivots are nearly zero!

To see what bad things can happen, consider a small modification to the problem

$$\begin{bmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \\ 6 \end{bmatrix}.$$

This problem has the exact solution $\mathbf{x} = (0, -1, 1)^T$.

Change a_{22} from 2 to 2.099.

Note: Also change b_2 so that the exact solution is still $\mathbf{x} = (0, -1, 1)^T$.

$$\begin{bmatrix} 10 & -7 & 0 \\ -3 & 2.099 & 6 \\ 5 & -1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 3.901 \\ 6 \end{bmatrix}.$$

Suppose our computer does floating-point arithmetic with 5 significant digits.

The first step of the elimination produces

$$\begin{bmatrix} 10.000 & -7.0000 & 0.0000 \\ 0.0000 & -0.0010 & 6.0000 \\ 0.0000 & 2.5000 & 5.0000 \end{bmatrix} \begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \\ \hat{x}_3 \end{bmatrix} = \begin{bmatrix} 7.0000 \\ 6.0010 \\ 2.5000 \end{bmatrix}.$$

The $(2, 2)$ element is now quite small compared with the other elements in the matrix.

If we continue without pivoting, the next step requires adding 2.5000×10^3 times the second equation to the third, leading to:

$$(5.0000 + (2.5000 \times 10^3)(6))\hat{x}_3 = 2.5000 + (2.5000 \times 10^3)(6.0010).$$

Now the exact answer to the product on the right-hand side is 1.50025×10^4, which cannot be represented exactly in our hypothetical floating-point number system with 5 significant digits.

It must be rounded to 1.5003×10^4.

The result is then added to 2.5000 and rounded again.

So in our hypothetical floating-point number system, the last equation becomes

$$1.5005 \times 10^4 \hat{x}_3 = 1.5006 \times 10^4.$$

The back substitution begins with

$$\hat{x}_3 = \frac{1.5006 \times 10^4}{1.5005 \times 10^4} = 1.0001.$$

Because $x_3 = 1$, the error is not too serious.

However, $\hat{x}_2$ is determined from

$$-0.00100\hat{x}_2 + (6.0000)(1.0001) = 6.0010$$

$$\Longrightarrow \ \hat{x}_2 = \frac{0.00040}{-0.00100} = -0.40000.$$

Finally $\hat{x}_1$ is determined from

$$10.000\hat{x}_1 + (-7.0000)(-0.40000) = 7.0000 \ \Longrightarrow \ \hat{x}_1 = -0.42000.$$

Instead of the exact solution $\mathbf{x} = (0, -1, 1)^T$, we get the approximation $\hat{\mathbf{x}} = (-0.42000, -0.40000, 1.0001)^T$!

What went wrong?

- There was no accumulation of rounding error caused by doing thousands of arithmetic operations.

- The matrix is not close to singular.

The problem comes entirely from choosing a small pivot at the second step of the elimination!

As a result, the multiplier is 2.5000×10^3, and the final equation involves coefficients that are 1000 times as large as those in the original problem.

Roundoff errors that are small compared to these large coefficients are not small compared to the entries of the original matrix and the actual solution!

Exercise: Verify that if the second and third equations are interchanged, then no large multipliers are necessary and the final result is accurate.

This turns out to be true in general: *If the multipliers are all less than or equal to one in magnitude, then the computed solution can be proven to be acceptable.*

The multipliers can be guaranteed to be less than one in magnitude through the use of partial pivoting.

At the kth step of the forward elimination, the pivot is taken to be the element with the largest absolute value in the unreduced part of the kth column.

The row containing this pivot is then interchanged with the kth row to bring the pivot element into the (k, k) position.

Of course, the same interchanges must be done with the elements of $\mathbf{b}$ (otherwise we would be changing the problem!).

The unknowns in $\mathbf{x}$ are not reordered because the columns of $\mathbf{A}$ are not interchanged.

2.8 The Effect of Roundoff Errors

Rounding errors always cause the computed solution $\hat{\mathbf{x}}$ to differ from the exact solution, $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

In fact, if the elements of $\mathbf{x}$ are not floating-point numbers themselves, then $\hat{\mathbf{x}}$ cannot equal $\mathbf{x}$!

There are two common measures of the quality of $\hat{\mathbf{x}}$:

the *error*:

$$\mathbf{e} = \mathbf{x} - \hat{\mathbf{x}}$$

and the *residual*:

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}.$$

Matrix theory tells us that, because $\mathbf{A}$ is nonsingular, if one of these is $\mathbf{0}$, then so is the other.

But they may not both be "small" at the same time!

Consider the following example:

$$\begin{bmatrix} 0.780 & 0.563 \\ 0.913 & 0.659 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.217 \\ 0.254 \end{bmatrix}.$$

Suppose we carry out Gaussian elimination with partial pivoting in three-digit arithmetic with truncation.

First, interchange the rows so that 0.913 is the pivot.

The multiplier is $-0.780/0.913 = -0.854$ (to 3 digits).

The elimination yields:

$$\begin{bmatrix} 0.913 & 0.659 \\ 0 & 0.001 \end{bmatrix} \begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} = \begin{bmatrix} 0.254 \\ 0.001 \end{bmatrix}.$$

Performing the back substitution:

$$\hat{x}_2 = 0.001/0.001 = 1.00 \qquad \text{(exactly)};$$

$$\hat{x}_1 = (0.254 - 0.659)/0.913 = -0.443 \text{ (to 3 digits)}.$$

Thus the computed solution is $\hat{\mathbf{x}} = (-0.443, 1.000)^T$.

To assess the accuracy without knowing the exact answer, we compute the residuals (exactly):

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}} = (-0.000460, -0.000541)^T.$$

The residuals are less than 10^{-3}
$\rightarrow$ it is hard to expect better on a 3-digit machine!

However, it is easy to see that the exact solution to this system is $\mathbf{x} = (1.000, -1.000)^T$!

Our computed solution $\hat{\mathbf{x}}$ is so bad that the components actually have the wrong signs!

So why were our residuals so small?

Of course, this example is highly contrived: The matrix is very close to being singular (not typical of most problems in practice).

But we can still understand what happened.

If Gaussian elimination with partial pivoting is carried out for this example on a computer with 6 or more digits, the forward elimination will produce a system something like

$$\begin{bmatrix} 0.913000 & 0.659000 \\ 0 & 0.000001 \end{bmatrix} \begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} = \begin{bmatrix} 0.254000 \\ -0.000001 \end{bmatrix}.$$

Notice that the sign of b_2 differs from that obtained with 3-digit computation.

Now the back substitution produces

$$\hat{x}_2 = -0.000001/0.000001 = -1.00000,$$

$$\hat{x}_1 = (0.254 - 0.659\hat{x}_2)/0.913 = 1.00000,$$

the exact answer!

Key point: On our 3-digit machine, $\hat{x}_2$ was computed by dividing two quantities, both on the order of roundoff (and one of which did not even have the correct sign!).

Hence $\hat{x}_2$ can turn out to be almost anything.

Then this arbitrary value of $\hat{x}_2$ was substituted into the first equation to obtain $\hat{x}_1$, essentially rendering it arbitrary as well.

We can reasonably expect the residual from the first equation to be small: $\hat{x}_1$ was computed in such a way as to make this certain.

Now comes a subtle but crucial point:

We can also expect the residual from the second equation to be small, precisely because the equations are almost multiples of each other!

$\rightarrow$ Any pair $(\hat{x}_1, \hat{x}_2)$ that nearly satisfies the first equation will also nearly satisfy the second.

If the two equations really were multiples of each other, we would not need the second equation at all: any solution of the first equation would automatically satisfy the second one.

This example is admittedly on the phony side, but the conclusion is not!

According to Cleve Moler, this is probably the single most important fact that we have learned about matrix computation since the invention of the digital computer:

Gaussian elimination with partial pivoting is guaranteed to produce small residuals.

"guaranteed" $\leftrightarrow$ it is possible to prove a precise theorem that bounds the size of the residual (assuming certain technical details about floating-point arithmetic).

"small" $\leftrightarrow$ on the order of roundoff error relative to:

- the size of the elements of $\mathbf{A}$,
- the size of the elements of the matrix during the elimination process,
- the size of the elements of $\mathbf{x}$.

If any of these are "large", then the residual will not necessarily be small in an absolute sense.

Final note: *Even if the residual is small, this does not mean that the error will be small!*

The relationship between the size of the residual and the size of the error is partly determined by the *condition number* of $\mathbf{A}$ (see next section).

2.9 Norms and Condition Numbers

The coefficients defining a system of linear equations often contain errors; e.g., experimental, roundoff, etc.

Even if the system can be stored exactly, roundoff errors will be introduced during the solution process.

It can be shown that roundoff errors in Gaussian elimination have the same effect on the answer as errors in the original coefficients.

So the question becomes: if the coefficients are altered slightly, how much is the solution altered?

Put another way, if $\mathbf{Ax} = \mathbf{b}$, what is the *sensitivity* of $\mathbf{x}$ to changes in $\mathbf{A}$ and $\mathbf{b}$?

The answer to this question we have to consider the meaning of "nearly singular" matrices.

If $\mathbf{A}$ is a singular matrix, then for some vectors $\mathbf{b}$, a solution $\mathbf{x}$ will not exist, and for others it will not be unique.

So if $\mathbf{A}$ is nearly singular, we can expect small changes in $\mathbf{A}$ and $\mathbf{b}$ to cause very large changes in $\mathbf{x}$.

On the other hand, if $\mathbf{A} = \mathbf{I}$, then $\mathbf{x} = \mathbf{b}$.

So if $\mathbf{A}$ is close to $\mathbf{I}$, small changes in $\mathbf{A}$ and $\mathbf{b}$ should result in correspondingly small changes in $\mathbf{x}$.

It might appear that there is some connection between the size of the pivots encountered in Gaussian elimination with partial pivoting and nearness to singularity: if arithmetic is exact, all pivots would be nonzero if and only if the matrix is nonsingular.

To some extent, it is also true that if the "partial" pivots are small, then the matrix is close to singular.

However, in the presence of roundoff, the converse is no longer true: *a matrix might be close to singular even though none of the pivots are small*.

In order to obtain a more precise quantification of nearness to singularity, we need to introduce the concept of a norm of a vector.

In general, the p-norm of a vector $\mathbf{x}$ is denoted by $\|\mathbf{x}\|_p$; it is a scalar that measures the "size" of $\mathbf{x}$.

The family of vector norms known as l_p is defined by

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{m} |x_i|^p \right)^{1/p}, \quad 1 \leq p < \infty.$$

In practice, we only use:

- $p = 1$ (Manhattan or taxicab norm)
- $p = 2$ (Euclidean norm)
- $p \to \infty$ (Chebyshev or maximum norm)

All three norms are easy to compute, and they satisfy

$$\|\cdot\|_1 \geq \|\cdot\|_2 \geq \|\cdot\|_\infty.$$

Often, the specific value of p is not important, and we simply write $\|\mathbf{x}\|$.

All vector norms satisfy the following relations, associated with the interpretation it being a distance:

- $\|\mathbf{x}\| > 0$ with $\|\mathbf{x}\| = 0 \iff \mathbf{x} = \mathbf{0}$.
- $\|c\mathbf{x}\| = |c|\|\mathbf{x}\|$ for all scalars c.
- $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ (triangle inequality).

The first relation says that the size of a non-zero vector is positive.

The second relations says that if a vector is scaled by a factor c then its size is also scaled by c.

The triangle inequality is a generalization of the fact that the size of one side of a triangle cannot exceed the sum of the sizes of the other two sides.

A useful variant of the triangle inequality is

$$\|\mathbf{x} - \mathbf{y}\| \geq \|\mathbf{x}\| - \|\mathbf{y}\|.$$

In MATLAB, $\|\mathbf{x}\|_p$ is computed by `norm(x,p)`; `norm(x)` defaults to `norm(x,2)`.

See normDemo.m

Multiplication of a vector $\mathbf{x}$ by a matrix $\mathbf{A}$ results in a new vector $\mathbf{Ax}$ that can be very different from $\mathbf{x}$.

In particular, the change in norm is directly related to the sensitivity we want to measure.

The range of the possible change can be expressed by the quantities:

$$M = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|}, \qquad m = \min_{\mathbf{x}\neq\mathbf{0}} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|}.$$

Note: If $\mathbf{A}$ is singular, $m = 0$.

The ratio M/m is called the *condition number* of $\mathbf{A}$:

$$\kappa(\mathbf{A}) = \frac{M}{m}.$$

The precise numerical value of $\kappa(\mathbf{A})$ depends on the vector norm being used!

Normally we are only interested in order of magnitude estimates of the condition number, so the particular norm is usually not very important.

Consider the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

and a perturbed system

$$\mathbf{A}(\mathbf{x} + \boldsymbol{\delta}\mathbf{x}) = (\mathbf{b} + \boldsymbol{\delta}\mathbf{b}).$$

We think of $\boldsymbol{\delta}\mathbf{b}$ as the error in $\mathbf{b}$ and $\boldsymbol{\delta}\mathbf{x}$ as the resulting error in $\mathbf{x}$.

(We don't necessarily assume that the errors are small!)

Note: $\mathbf{A}(\boldsymbol{\delta}\mathbf{x}) = \boldsymbol{\delta}\mathbf{b}$.

The definitions of M and m imply

$$\|\mathbf{b}\| \leq M\|\mathbf{x}\|, \qquad \|\boldsymbol{\delta}\mathbf{b}\| \geq m\|\boldsymbol{\delta}\mathbf{x}\|.$$

So if $m \neq 0$,

$$\frac{\|\boldsymbol{\delta}\mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(\mathbf{A})\frac{\|\boldsymbol{\delta}\mathbf{b}\|}{\|\mathbf{b}\|}.$$

Thus, *the condition number is a relative error magnification factor*.

In other words, relative changes in $\mathbf{b}$ can cause relative changes in $\mathbf{x}$ that are $\kappa(\mathbf{A})$ times as large.

It turns out that the same is true of changes in $\mathbf{A}$.

$\kappa(\mathbf{A})$ is also a measure of nearness to singularity:
$\rightarrow$ *if $\kappa(\mathbf{A})$ is large, $\mathbf{A}$ is close to singular*.

Some basic properties of the condition number:

- $M \geq m$, so $\kappa(\mathbf{A}) \geq 1$.
- If $\mathbf{P}$ is a permutation matrix, then $\mathbf{Px}$ is simply a rearrangement of $\mathbf{x}$, so $\|\mathbf{Px}\| = \|\mathbf{x}\|$ for all $\mathbf{x}$; hence $\kappa(\mathbf{P}) = 1$. In particular, $\kappa(\mathbf{I}) = 1$.
- If $\mathbf{A}$ is multiplied by a scalar c, then M and m are both multiplied by c, so $\kappa(c\mathbf{A}) = \kappa(\mathbf{A})$.
- If $\mathbf{D}$ is a diagonal matrix, then $\kappa(\mathbf{D}) = \frac{\max |d_{ii}|}{\min |d_{ii}|}$.

These last two properties are reasons that $\kappa(\mathbf{A})$ is a better measure of nearness to singularity than $\det(\mathbf{A})$.

e.g., consider a 100×100 diagonal matrix with 0.1 on the diagonal.

Then $\det(\mathbf{A}) = 10^{-100} \rightarrow$ generally a small number!

But $\kappa(\mathbf{A}) = 1$, and in fact for linear systems of equations, such a matrix behaves more like $\mathbf{I}$ than like a singular matrix.

Example

Let

$$\mathbf{A} = \begin{bmatrix} 4.1 & 2.8 \\ 9.7 & 6.6 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 4.1 \\ 9.7 \end{bmatrix}.$$

The solution to $\mathbf{Ax} = \mathbf{b}$ is $\mathbf{x} = (1, 0)^T$.

In the 1-norm, $\|\mathbf{b}\|_1 = 13.8$, and $\|\mathbf{x}\|_1 = 1$.

If we perturb the right-hand side to $\tilde{\mathbf{b}} = (4.11, 9.70)^T$, then the solution becomes $\tilde{\mathbf{x}} = (0.34, 0.97)^T$.

$\rightarrow$ A tiny perturbation has completely changed the solution!

Defining $\boldsymbol{\delta}\mathbf{b} = \mathbf{b} - \tilde{\mathbf{b}}$ and $\boldsymbol{\delta}\mathbf{x} = \mathbf{x} - \tilde{\mathbf{x}}$, we find

$$\|\boldsymbol{\delta}\mathbf{b}\|_1 = 0.01, \quad \|\boldsymbol{\delta}\mathbf{x}\|_1 = 1.63.$$

Thus,

$$\frac{\|\boldsymbol{\delta}\mathbf{b}\|_1}{\|\mathbf{b}\|_1} = 0.0007246, \quad \frac{\|\boldsymbol{\delta}\mathbf{x}\|_1}{\|\mathbf{x}\|_1} = 1.63,$$

and so

$$\kappa(\mathbf{A}) \geq \frac{1.63}{0.0007246} = 2249.4.$$

(We have made it so that in fact `cond(A,1)`$= 2249.4$.)

Important note: This example deals with the *exact solutions to two slightly different problems*.

$\rightarrow$ The method to obtain the solutions is irrelevant!

The example is chosen so that the effect of changing $\mathbf{b}$ is quite pronounced, but similar behaviour can be expected in any problem with a large condition number.

Now we see how $\kappa(\mathbf{A})$ also plays a fundamental role in the analysis of roundoff errors during Gaussian elimination.

Assume that $\mathbf{A}$ and $\mathbf{b}$ have elements that are exact floating-point numbers.

Let $\mathbf{x}$ be the exact solution, and let $\mathbf{x}_*$ be the vector of floating-point numbers obtained from Gaussian elimination representing the computed solution.

Assume $\mathbf{A}$ is non-singular and nothing funny happened like underflows or overflows during the elimination.

Then it is possible to establish the following inequalities:

$$\frac{\|\mathbf{b} - \mathbf{A}\mathbf{x}_*\|}{\|\mathbf{A}\|\|\mathbf{x}\|} \leq \rho\epsilon_{\text{machine}}, \quad \frac{\|\mathbf{x} - \mathbf{x}_*\|}{\|\mathbf{x}_*\|} \leq \rho\kappa(\mathbf{A})\epsilon_{\text{machine}},$$

where ρ is a constant (rarely larger than about 10) and $\|\mathbf{A}\|$ is the norm of $\mathbf{A}$ (to be defined momentarily).

But first, what do these inequalities say?

The first one says that *the norm of the relative residual is about the size of roundoff error — no matter how badly conditioned the matrix is*!

(See example in the previous section.)

The second one says that *the norm of the relative error will be small if $\kappa(\mathbf{A})$ is small — but it might be quite large if $\kappa(\mathbf{A})$ is large, i.e., if $\mathbf{A}$ is nearly singular*!

Now back to defining the matrix norm.

The norm of a matrix $\mathbf{A}$ is defined to be

$$\|\mathbf{A}\| = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|}.$$

Technically this is called an *induced* matrix norm because it is defined using an underlying vector norm.

(Other matrix norms do exist, but we do not consider them in this course.)

It turns out that $\|\mathbf{A}^{-1}\| = 1/m$, so an equivalent definition of the condition number of a matrix is $\kappa(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$.

Note: The precise numerical values of $\|\mathbf{A}\|$ and $\kappa(\mathbf{A})$ depend on the underlying vector norm.

It is easy to compute the matrix norms corresponding to the ℓ_1 and ℓ_∞ vector norms:

$$\|\mathbf{A}\|_1 = \max_j \sum_{i=1}^{m} |a_{ij}|, \qquad \|\mathbf{A}\|_\infty = \max_i \sum_{j=1}^{m} |a_{ij}|.$$

In words, $\|\mathbf{A}\|_1 =$ maximum absolute column sum.

$$\|\mathbf{A}\|_\infty = \text{maximum absolute row sum.}$$

Computing the matrix norm corresponding to the ℓ_2 vector norm involves the *singular value decomposition*, which is beyond the scope of this course.

MATLAB computes matrix norms with `norm(A,p)` for `p`$= 1, 2,$ or `Inf`.

Computing $\kappa(\mathbf{A})$ requires knowing $\|\mathbf{A}^{-1}\|$.

But computing $\mathbf{A}^{-1}$ requires roughly 3 times as much work as solving a single linear system.

Fortunately, the exact value of $\kappa(\mathbf{A})$ is rarely required.
$\rightarrow$ Any good estimate is satisfactory.

MATLAB has several functions for computing or estimating condition numbers.

- `cond(A)` $(=$ `cond(A,2)`$)$ computes $\kappa_2(\mathbf{A})$. It is suitable for smaller matrices where the geometric properties of the ℓ_2 norm are important.

- `cond(A,1)` computes $\kappa_1(\mathbf{A})$. It is less work than computing `cond(A,2)`.

- `cond(A,Inf)` computes $\kappa_\infty(\mathbf{A})$. It is the same as `cond(A',1)`.

- `condest(A)` estimates $\kappa_1(\mathbf{A})$. It is especially suitable for large, sparse matrices.