2 Solving linear systems of equations

The need to solve systems of linear equations $\mathbf{Ax} = \mathbf{b}$ arises across nearly all of engineering and science, business, statistics, economics, and many other fields. In a standard undergraduate linear algebra course, we have learned how to solve this problem using Gaussian Elimination (GE). We will show here how such a procedure is equivalent to an LU factorization of the coefficient matrix $\mathbf{A}$, followed by a forward and a back substitution. To achieve stability of the factorization in computer arithmetic, a strategy called pivoting is necessary, which leads to the LU factorization with partial pivoting. This is the standard *direct method* for solving linear systems where $\mathbf{A}$ is a dense matrix.

Linear systems with large and sparse (most entries are zero) coefficient matrices arise often in numerical solution methods of differential equations, e.g. by the finite element and finite difference discretizations. State-of-the-art direct methods can nowadays efficiently solve such linear systems up to an order of a few million, using advanced strategies to keep the LU factors as sparse as possible and the factorization stable. However, problems of ever-increasing dimension need be tackled, and sparse linear systems of order tens of millions to billions have become more routine. To efficiently solve these large systems approximately, *iterative methods* such as the Conjugate Gradient (CG) method are typically used, and on sufficiently large problems, can be advantageous over direct methods.

This chapter will mainly focus on direct methods but will also discuss the CG method. 'Linear solvers' has become a vast field and is a very active research area. We aim here to provide a fundamental understanding of the basic types of solvers, but note that we are just scratching the surface, in particular for iterative methods.

2.1 Solving triangular linear systems

Consider a system of linear equations $\mathbf{Ax} = \mathbf{b}$, where the coefficient matrix $\mathbf{A}$ is square and nonsingular. Recall that the GE procedure gradually eliminates all entries in the coefficient matrix below the main diagonal by elementary row operations, until the modified coefficient matrix becomes an upper triangular matrix $\mathbf{U}$. The solution remains unchanged during the entire procedure. In this section, we consider how to solve a linear system where the coefficient matrix is upper or lower triangular. The procedure of elimination will be reviewed and explored in the new perspective of matrix factorization in the next section.

Example 3 (Back substitution for an upper triangular system)**.** *Consider the linear system* $x_1 + 2x_2 + 3x_3 = 2$, $4x_2 + 5x_3 = 3$ *and* $6x_3 = -6$. *It can be written in matrix form as*

$$\begin{pmatrix} 1 & 2 & 3 \\ & 4 & 5 \\ & & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ -6 \end{pmatrix},$$

where the coefficient matrix is upper triangular. To solve this linear system, we start from the last equation $6x_3 = -6$, *which immediately gives* $x_3 = \frac{-6}{6} = -1$.

Then, from the second equation $4x_2 + 5x_3 = 3$, *we get* $x_2 = \frac{3-5x_3}{4} = 2$. *Finally, the first equation* $x_1 + 2x_2 + 3x_3 = 2$ *leads to* $x_1 = 2 - 2x_2 - 3x_3 = 1$.

This procedure illustrates the general procedure of *back substitution*. Given an upper triangular linear system with nonzero diagonal entries

$$\mathbf{Ux} = \begin{pmatrix} u_{11} & u_{12} & & \dots & u_{1n} \\ & \ddots & & \ddots & \vdots \\ & & & & \\ & & u_{(n-1)(n-1)} & u_{(n-1)n} & \\ & & & & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ \\ b_{n-1} \\ b_n \end{pmatrix},$$

we start with the last equation and evaluate $x_n = \frac{b_n}{u_{nn}}$ directly, then substitute it into the previous equation and compute $x_{n-1} = \frac{b_{n-1} - u_{(n-1)n} x_n}{u_{(n-1)(n-1)}}$. Assume in general that we have already solved for $x_{i+1}, \dots, x_n$, then $x_i = \frac{b_i - \sum_{j=i+1}^{n} u_{ij} x_j}{u_{ii}}$ can be evaluated. We continue until the value of x_1 is found.

A MATLAB code for back substitution is given below.

```
function [x] = BackSubstitution(A,b)
% solve the upper triangular system Ax=b using back-substitution

n = length(b);
x = zeros(n,1);

for j=n:-1:1
    % Check to see if the diagonal entry is zero
    if abs(A(j,j)) < 1e-15
        error('A is singular (diagonal entries of zero)')
    end

    % Compute solution component
    x(j) = b(j) / A(j,j);

    % Update the RHS vector
    for i=1:j-1
       b(i) = b(i) - A(i,j)*x(j);
    end
end
```

Similarly, consider a lower triangular linear system

$$\begin{pmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

Here, we start from the first equation and get $x_1 = \frac{b_1}{l_{11}}$ first, then move to the second equation. At Step i, we can evaluate $x_i = \frac{b_i - \sum_{j=1}^{i-1} l_{ij} x_j}{l_{ii}}$. This procedure, called *forward substitution*, moves on until x_n is determined.

One can follow these algorithms and see that both substitutions need $\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2} \approx \frac{1}{2}n^2$ addition/subtractions and $\sum_{i=1}^{n}[(i-1)+1] = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$ multiplication/divisions, or n^2 combined. Both will be used after we complete an LU factorization of the coefficient matrix $\mathbf{A}$ to solve $\mathbf{Ax} = \mathbf{b}$.

2.2 From Gaussian elimination to LU factorization

In this section, we show that the standard GE without row swapping can be done through elementary matrix operations, which naturally leads to an LU factorization without pivoting.

Let us briefly review GE. Given a system of linear equations $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} \in \mathbb{R}^{n\times n}$ is nonsingular, and $\mathbf{b} \in \mathbb{R}^n$, GE updates the coefficient matrix and the right hand side by elementary row operations, one column at a time, eliminating all entries below the main diagonal. This produces a new linear system $\mathbf{Ux} = \mathbf{c}$, where $\mathbf{U}$ is an upper triangular matrix. Then, back substitution is used to solve $\mathbf{Ux} = \mathbf{c}$ for $\mathbf{x}$. Note that this procedure does not change the solution to $\mathbf{Ax} = \mathbf{b}$, i.e., it is the same $\mathbf{x}$ in $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{c}$.

After reviewing this process below, we will show how it can be 'saved' in an LU factorization $\mathbf{A} = \mathbf{LU}$, with $\mathbf{L}$ lower triangular and $\mathbf{U}$ upper triangular. The $\mathbf{U}$ is the upper triangular matrix that results from the GE procedure. The $\mathbf{L}$ is a matrix with ones on the diagonal, and whose entries are easily determined from the GE procedure (although the mathematical reasons why it works are a little more complicated). In your linear algebra course, the GE procedure could be performed in many different ways, so long as the Gauss rules are not violated. However, for a computer algorithm, it is necessary to avoid ambiguity and to have a more precise algorithm: we will always move left to right, and use the diagonal entry to eliminate entries below it.

Example 4 (Gaussian elimination)**.** *Apply GE to solve the system of linear equations*

$$\begin{aligned} 2x_1 + x_2 + 3x_3 &= 5, \\ 4x_1 - x_2 + 2x_3 &= -1, \\ -x_1 + 4x_2 + x_3 &= 7. \end{aligned}$$

This linear system can be written as $\mathbf{Ax} = \mathbf{b}$*, for which the augmented matrix is*

$$(\mathbf{A} \mid \mathbf{b}) = \begin{pmatrix} 2 & 1 & 3 & 5 \\ 4 & -1 & 2 & -1 \\ -1 & 4 & 1 & 7 \end{pmatrix}.$$

Step 1: *Eliminate the entries in column 1 below the main diagonal, namely, the* $(2,1)$ *entry* 4 *and the* $(3,1)$ *entry* -1*. This is done by subtracting twice of row 1 from row 2* ($R_2 \leftarrow R_2 - 2 \times R_1$)*, and adding half of row 1 to row 3* ($R_3 \leftarrow R_3 + \frac{1}{2}R_1$)*. This is described as*

$$\begin{pmatrix} 2 & 1 & 3 & 5 \\ 4 & -1 & 2 & -1 \\ -1 & 1 & 1 & 4 \end{pmatrix} \begin{array}{l} R_2 \leftarrow R_2 - 2 \times R_1 \\ R_3 \leftarrow R_3 + \frac{1}{2}R_1 \\ \longrightarrow \end{array} \begin{pmatrix} 2 & 1 & 3 & 5 \\ 0 & -3 & -4 & -11 \\ 0 & \frac{9}{2} & \frac{5}{2} & \frac{19}{2} \end{pmatrix}.$$

Let m_{ji} be the multiplier used to zero-out the (j,i) entry: $m_{21}=-2$, $m_{31}=\frac{1}{2}$.

Step 2: *Eliminate the entries in column 2 below the diagonal, which is simply the $(3,2)$ entry $\frac{9}{2}$. This is done by adding $\frac{3}{2}$ times row 2 to row 3 ($R_3 \leftarrow R_3+\frac{3}{2}R_2$).*

That is, $\begin{pmatrix} 2 & 1 & 3 & 5 \\ 0 & -3 & -4 & -11 \\ 0 & \frac{9}{2} & \frac{5}{2} & \frac{19}{2} \end{pmatrix} \overset{R_3\leftarrow R_3+\frac{3}{2}R_2}{\longrightarrow} \begin{pmatrix} 2 & 1 & 3 & 5 \\ 0 & -3 & -4 & -11 \\ 0 & 0 & -\frac{7}{2} & -7 \end{pmatrix}$

Here, $m_{32}=\frac{3}{2}$.

Step 3: *Solve the upper triangular system, bottom to top. From the last equation $-\frac{7}{2}x_3=-7$, we have $x_3=2$; then from the 2nd equation $-3x_2-4x_3=-11$, we get $x_2=1$; finally, from the 1st equation $2x_1+x_2+3x_3=5$, we find $x_1=-1$.*

If we define the matrix $\mathbf{L}$ by

$$l_{jj}=1, \quad l_{ji}=0 \text{ if } j<i, \quad l_{ji}=-m_{ji}, \text{ if } j>i,$$

then we obtain the LU factorization $\mathbf{A}=\mathbf{LU}$. One can check that

$$\mathbf{A}=\begin{pmatrix} 2 & 1 & 3 \\ 4 & -1 & 2 \\ -1 & 4 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -\frac{1}{2} & -\frac{3}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 3 \\ 0 & -3 & -4 \\ 0 & 0 & -\frac{7}{2} \end{pmatrix} = \mathbf{LU}.$$

We now discuss why $\mathbf{L}$ can be created this way. At Step i, we want to eliminate all entries in the i-th column below the diagonal. To eliminate the (j,i) entry ($j>i$), adding m_{ji} times row i to row j is equivalent to left-multiplying by the elementary matrix

$$\mathbf{E}_{ji}=\begin{pmatrix} 1 & & & \\ & \ddots & & \\ & m_{ji} & \ddots & \\ & & & 1 \end{pmatrix},$$

which differs from the identity matrix $\mathbf{I}_n$ at the (j,i) entry alone. This can be verified by looking at the j-th row of $\mathbf{E}_{ji}$ applied to the current (modified)

coefficient matrix:

$$\begin{pmatrix} 0,\ldots,0, & \underbrace{m_{ji}}_{\text{the } i\text{-th entry}} & ,0,\ldots, & \underbrace{1}_{\text{the } j\text{-th entry}} & ,\ldots,0 \end{pmatrix} \begin{pmatrix} R_1 \\ \vdots \\ R_n \end{pmatrix} = R_j + m_{ji}R_i.$$

Observe that all other rows remain intact after this matrix multiplication.

We can verify that the product of all such elementary matrices accumulated at Step i is $\mathbf{L}_i = \mathbf{E}_{ni}\mathbf{E}_{(n-1)i}\ldots\mathbf{E}_{(i+1)i} =$

$$\begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & \underbrace{1}_{(i,i)} & & \\ & & m_{(i+1)i} & \ddots & \\ & & \vdots & & \ddots \\ & & m_{ni} & & 1 \end{pmatrix},$$

which differs from the identity at the entries below the i-th diagonal entry.

Interestingly, a simple calculation shows that the inverse of $\mathbf{L}_i$ is obtained by simply negating the off-diagonal entries (check this by verifying $\mathbf{L}_i\mathbf{L}_i^{-1} = \mathbf{I}_n$):

$$\mathbf{L}_i^{-1} = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & \underbrace{1}_{(i,i)} & & \\ & & -m_{(i+1)i} & \ddots & \\ & & \vdots & & \ddots \\ & & -m_{ni} & & 1 \end{pmatrix}.$$

Another interesting fact is that the product of $\mathbf{L}_i^{-1}$ matrices creates a matrix that keeps all the nonzero entries of each matrix unchanged:

$$\mathbf{L}_i^{-1}\mathbf{L}_j^{-1} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & -m_{(i+1)i} & & \ddots & & & \\ & \vdots & & & 1 & & \\ & \vdots & & & -m_{(j+1)j} & \ddots & \\ & \vdots & & & \vdots & & \ddots \\ & -m_{ni} & & & -m_{nj} & & 1 \end{pmatrix}.$$

We can now describe the GE procedure in terms of multiplication of $\mathbf{A}$ by the $\mathbf{L}_i$ matrices: $\mathbf{U} = \mathbf{L}_{n-1}\mathbf{L}_{n-2}\cdots\mathbf{L}_1\mathbf{A}$. Thus, we can write

$$\mathbf{A} = \mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\cdots\mathbf{L}_{n-1}^{-1}\mathbf{U}.$$

From the discussion above, we can define $\mathbf{L} := \mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\cdots\mathbf{L}_{n-1}^{-1}$, which is easily calculated to be

$$\mathbf{L} := \begin{pmatrix} 1 & & & & \\ -m_{21} & 1 & & & \\ \vdots & -m_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ -m_{n1} & -m_{n2} & \dots & -m_{n(n-1)} & 1 \end{pmatrix}.$$

Hence, even though it took a bit of work to develop the representation of $\mathbf{L}$, in practice it comes down to simply placing the negative of these GE multipliers m_{ji} in their corresponding locations in $\mathbf{L}$. Thus, no extra work is needed to create $\mathbf{L}$ and $\mathbf{U}$, and this is why the terms GE and LU are used synonymously.

2.3 Pivoting

GE breaks down at Step i if the i-th diagonal entry of the current (modified) coefficient matrix, referred to as the *pivot*, is zero (or close to 0), since there is no way to eliminate a nonzero entry using a zero pivot. A zero pivot may arise

at any step of GE, making the algorithm fail, even if $\mathbf{A}$ is nonsingular and a unique solution to $\mathbf{Ax} = \mathbf{b}$ exists. Consider for example the linear equation

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}.$$

There is a unique solution ($x_1 = 2$ and $x_2 = 0$), but GE fails at the first step.

To fix this issue, if a zero pivot (current (i, i) entry) appears at Step i, we can switch the i-th row containing the zero pivot with a row below it, say the j-th ($j > i$) row where the (j, i) entry is nonzero, so that the new pivot (new (i, i) entry) is no longer zero. This strategy is called *partial pivoting*.

In exact arithmetic (no round-off errors), this would be sufficient to make sure GE proceeds to the end. However, partial pivoting must also be used if the original pivot is excessively small. Consider the example $\mathbf{A} = \begin{pmatrix} \frac{1}{2}\epsilon & -1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & \\ 2\epsilon^{-1} & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{2}\epsilon & -1 \\ & 1 + 2\epsilon^{-1} \end{pmatrix} = \mathbf{LU}$, where ϵ is the machine precision. Assume that $2\epsilon^{-1}$ can be evaluated exactly in computer arithmetic. Then, the floating point representation of the $(2, 2)$ entry of $\mathbf{U}$, namely $1 + 2\epsilon^{-1}$, is $2\epsilon^{-1}$ due to rounding in computer arithmetic. Let the computed factors be $\widehat{\mathbf{L}} = \mathbf{L}$, and $\widehat{\mathbf{U}} = \begin{pmatrix} \frac{1}{2}\epsilon & -1 \\ & 2\epsilon^{-1} \end{pmatrix}$, such that $\widehat{\mathbf{A}} = \widehat{\mathbf{L}}\widehat{\mathbf{U}} = \begin{pmatrix} \frac{1}{2}\epsilon & -1 \\ 1 & 0 \end{pmatrix}$. In other words, the computed factors $\widehat{\mathbf{L}}$ and $\widehat{\mathbf{U}}$ form the exact LU factorization of $\widehat{\mathbf{A}}$ that is *very different* from $\mathbf{A}$! This is, of course, not acceptable.

The standard partial pivoting used for GE that will address these issues is as follows. At Step i, we have eliminated all entries of $(\mathbf{A} \mid \mathbf{b})$ in the first $i - 1$ columns below the diagonal, and we need to do so for the i-th column. The

current modified coefficient matrix we have is

$$\begin{pmatrix} a_{11} & \cdots & a_{1i} & \cdots & a_{1n} \\ 0 & \ddots & \vdots & & \vdots \\ \vdots & 0 & a_{ii} & \cdots & a_{in} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a_{ni} & \cdots & a_{nn} \end{pmatrix}.$$

We compare the entries $a_{ii}, a_{(i+1)i}, \dots a_{ni}$ in absolute value and find the largest

$$|a_{ji}| = \max_{i \le k \le n} |a_{ki}|,$$

where we denote by j the row with the largest entry. Next, swap row i and row j ($i \le j$), and perform the current step of elimination as usual.

The row swapping can be done equivalently by left-multiplying by the permutation matrix $\mathbf{P}_i$, which itself is obtained by switching rows i and j of the identity matrix $\mathbf{I}$; the elimination is performed by left-multiplying by the unit lower triangular matrix $\mathbf{L}_i$ introduced before. Then we proceed to the next step, continuing until we have eliminated all entries below the diagonal and obtain the upper triangular $\mathbf{U}$. This procedure is described in matrix operations as

$$\underbrace{\mathbf{L}_{n-1}\mathbf{P}_{n-1}}_{\text{Step } (n-1)} \underbrace{\mathbf{L}_{n-2}\mathbf{P}_{n-2}}_{\text{Step } (n-2)} \cdots \underbrace{\mathbf{L}_2\mathbf{P}_2}_{\text{Step 2}} \underbrace{\mathbf{L}_1\mathbf{P}_1}_{\text{Step 1}} \mathbf{A} = \mathbf{U}.$$

To develop a deeper insight into the above relation, note that all $\mathbf{P}_i$ are symmetric permutation matrices, such that $\mathbf{P}_i\mathbf{P}_i^T = \mathbf{P}_i^2 = \mathbf{I}$. With some effort, we can show that this equation can be written as

$$(\mathbf{L}_{n-1})\,(\mathbf{P}_{n-1}\mathbf{L}_{n-2}\mathbf{P}_{n-1})\dots(\mathbf{P}_{n-1}\dots\mathbf{P}_2\mathbf{L}_1\mathbf{P}_2\dots\mathbf{P}_{n-1})\,\mathbf{P}_{n-1}\dots\mathbf{P}_2\mathbf{P}_1\mathbf{A} = \mathbf{U},$$

where each matrix in a parentheses is lower triangular with 1's on the diagonal!

This might not seem obvious at first glance, but can be shown by noting that $\mathbf{L}_i$ is a unit lower triangular with nonzeros only in the i-th column, and that it has identical nonzero pattern to $\mathbf{P}_{i+1}\mathbf{L}_i\mathbf{P}_{i+1}$ (switching rows $i+1$ and j of $\mathbf{L}_i$, then its columns $i+1$ and j, $j \ge i+1$) and $\mathbf{P}_{i+2}\mathbf{P}_{i+1}\mathbf{L}_i\mathbf{P}_{i+1}\mathbf{P}_{i+2}$, etc. In summary, we have

$$\widetilde{\mathbf{L}}_{n-1}\widetilde{\mathbf{L}}_{n-2}\dots\widetilde{\mathbf{L}}_1\mathbf{P}\mathbf{A} = \mathbf{U},$$

where each $\widetilde{\mathbf{L}}_i = \mathbf{P}_{n-1}\dots\mathbf{P}_{i+1}\mathbf{L}_i\mathbf{P}_{i+1}\dots\mathbf{P}_{n-1}$ is a lower triangular matrix with 1's on the diagonal, and thus so is $\widetilde{\mathbf{L}}_{n-1}\widetilde{\mathbf{L}}_{n-2}\dots\widetilde{\mathbf{L}}_1$, and $\mathbf{P} = \mathbf{P}_{n-1}\dots\mathbf{P}_1$ is a

permutation matrix. Taking the inverse of the lower triangular matrices on both sides, we have

$$\mathbf{PA} = \widetilde{\mathbf{L}}_1^{-1} \dots \widetilde{\mathbf{L}}_{n-2}^{-1} \widetilde{\mathbf{L}}_{n-1}^{-1} \mathbf{U} = \mathbf{LU},$$

where $\mathbf{L} = \widetilde{\mathbf{L}}_1^{-1} \dots \widetilde{\mathbf{L}}_{n-2}^{-1} \widetilde{\mathbf{L}}_{n-1}^{-1}$ is also a unit lower triangular.

The above derivation gives clear guidance on how to generate the $\mathbf{U}$ and $\mathbf{P}$ matrices. However, the $\mathbf{L}$ factor has a complicated expression, and it would be awkward and inefficient to construct it in the original formula using matrix multiplications and inverses. In light of the above analysis (given for the non-pivoting case), all the three factors can be constructed as follows. Initialize $\mathbf{P} = \mathbf{L} = \mathbf{I}$. Then at Step i ($1 \le i \le n-1$),

1. Find the largest (in absolute value) (j, i) entry in the i-th column ($1 \le i \le j \le n$) of the modified coefficient matrix, at or below the (i, i) entry;
2. Swap rows i and j in the modified coefficient matrix, the permutation matrix $\mathbf{P}$, and these two rows in columns 1 through $i-1$ in $\mathbf{L}$;
3. Eliminate all subdiagonal entries in the i-th column of the modified coefficient matrix, updating rows/columns $i+1$ through n;
4. Place in the (k, i) entry ($i < k \le n$) of $\mathbf{L}$ negative the multiplying factor of row i added to row k used in the elimination.

Upon completion, $\mathbf{U}$ is the final modified upper triangular coefficient matrix, $\mathbf{L}$ is a lower triangular matrix, $\mathbf{P}$ is a permutation matrix, and $\mathbf{PA} = \mathbf{LU}$.

Example 5 (LU with partial pivoting, or PLU)**.** *We follow the procedure above to factorize*

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & 3 \\ 4 & -1 & 2 \\ -1 & 4 & 1 \end{pmatrix}$$

into $\mathbf{PA} = \mathbf{LU}$*, with initial* $\mathbf{P} = \mathbf{I}_3$ *and* $\mathbf{L} = \mathbf{I}_3$*.*

Step 1: *Compare the entries on and below the diagonal in column* 1 *and find the largest in modulus (the* $(2, 1)$ *entry* 4*). Then we swap rows* 1 *and* 2 *of* $\mathbf{A}$ *and* $\mathbf{P}$ *and get the updated matrix* $\begin{pmatrix} 4 & -1 & 2 \\ 2 & 1 & 3 \\ -1 & 4 & 1 \end{pmatrix}$ *and* $\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$*. There is no swapping to do for* $\mathbf{L}$*. Next, we eliminate the new* $(2, 1)$ *entry by performing*

$R_2 \leftarrow R_2 - \mathbf{\frac{1}{2}}R_1$ *and the new* $(3,1)$ *entry via* $R_3 \leftarrow R_3 + \mathbf{\frac{1}{4}}R_1$. *We have the modified coefficient matrix* $\begin{pmatrix} 4 & -1 & 2 \\ 0 & \frac{3}{2} & 2 \\ 0 & \frac{15}{4} & \frac{3}{2} \end{pmatrix}$ *and the updated* $\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ -\frac{1}{4} & 0 & 1 \end{pmatrix}$.

Step 2: *Compare the entries on and below the diagonal in column* 2 *and find the largest in modulus (the* $(3,2)$ *entry* $\frac{15}{4}$*). Then we swap rows* 2 *and* 3 *of this modified matrix and of* $\mathbf{P}$*, and these two rows in column* 1 *of* $\mathbf{L}$ *to get* $\begin{pmatrix} 4 & -1 & 2 \\ 0 & \frac{15}{4} & \frac{3}{2} \\ 0 & \frac{3}{2} & 2 \end{pmatrix}$, $\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$, *and* $\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{1}{4} & 1 & 0 \\ \frac{1}{2} & 0 & 1 \end{pmatrix}$.

Next, we eliminate the $(3,2)$ *entry by performing* $R_3 \leftarrow R_3 - \mathbf{\frac{2}{5}}R_2$. *This gives* $\begin{pmatrix} 4 & -1 & 2 \\ 0 & \frac{15}{4} & \frac{3}{2} \\ 0 & 0 & \frac{7}{5} \end{pmatrix}$ *along with updated* $\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{1}{4} & 1 & 0 \\ \frac{1}{2} & \frac{2}{5} & 1 \end{pmatrix}$.

Since the matrix has become upper triangular, we have completed the algorithm. We recover $\mathbf{U} = \begin{pmatrix} 4 & -1 & 2 \\ 0 & \frac{15}{4} & \frac{3}{2} \\ 0 & 0 & \frac{7}{5} \end{pmatrix}$, *and have the final* $\mathbf{L}$ *and* $\mathbf{P}$ *as their last values in the algorithm.*

Note that the algorithm above is built into MATLAB, and can be called via `lu`:

```
>> A = [2 1 3; 4 -1 2; -1 4 1]

A =

     2     1     3
     4    -1     2
    -1     4     1

>> [L,U,P] = lu(A)
```

```
L =

    1.0000         0         0
   -0.2500    1.0000         0
    0.5000    0.4000    1.0000

U =

    4.0000   -1.0000    2.0000
         0    3.7500    1.5000
         0         0    1.4000

P =

     0     1     0
     0     0     1
     1     0     0
```

Remark 6. *If* $\mathbf{A}$ *is real, symmetric, and positive definite* ($\mathbf{x}^T\mathbf{A}\mathbf{x} > 0,\ \forall \mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$)*, the above process can be modified to produce the Cholesky factorization* $\mathbf{A} = \mathbf{L}\mathbf{L}^T$*. This factorization is more efficient, no pivoting is required, and it improves storage since only one triangular matrix needs stored.* MATLAB *has this factorization built in as well and it can be called via the* `chol` *command.*

2.3.1 Work in LU/GE

How many floating point operations (flops) are needed to perform PLU? At Step i, we need to compare $n-i+1$ entries $a_{ii}, \ldots, a_{ni}$ and choose the largest one in modulus. After pivoting, we need $n-i$ divisions to get the multiple factors used for row operations. To eliminate each subdiagonal (j,i) entry ($i < j \leq n$) in the i-th column, it takes $n-i$ scalar multiplications and $n-i$ additions to subtract a multiple of the new i-th row from the j-th row. In total, there will be $\sum_{i=1}^{n-1}(n-i+1) = \frac{1}{2}(n+2)(n-1) \approx \frac{n^2}{2}$ scalar comparisons, $\sum_{i=1}^{n-1}[(n-i)+(n-i)(n-i)] = \frac{1}{3}(n^3-n) \approx \frac{n^3}{3}$ multiplication/divisions, and $\sum_{i=1}^{n-1}(n-i)(n-i) = \frac{1}{6}n(n-1)(2n-1) \approx \frac{1}{3}n^3$ addition/subtractions. If multiplications and additions take about the same time (which is the case nowadays on many platforms), we can combine them and simply say that the arithmetic cost for LU factorization is $\frac{2}{3}n^3$ flops plus $\frac{1}{2}n^2$ comparisons.

The above estimates assume that every entry of $\mathbf{A}$ is nonzero. If $\mathbf{A}$ has certain special nonzero structures, the estimate may become smaller. For instance, if all entries of $\mathbf{A}$ below the k-th subdiagonal are zero, where k is independent of n, then the total arithmetic cost is at most $\mathcal{O}(kn^2)$. This is because at each step of

the factorization, at most $k-1$ entries need to be eliminated, and hence at most $k-1$ rows will be updated. In addition, for such a matrix $\mathbf{A}$, if all entries above the k-th superdiagonal are zero, then the total work is at most $\mathcal{O}(k^2 n)$.

Finally, assume that we have completed the LU factorization and have $\mathbf{P}$, $\mathbf{L}$, and $\mathbf{U}$ such that $\mathbf{PA} = \mathbf{LU}$. To solve the linear system $\mathbf{Ax} = \mathbf{b}$, note that it is equivalent to $\mathbf{LUx} = \mathbf{PAx} = \mathbf{Pb}$, which leads to $\mathbf{x} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{Pb}$. This can be evaluated by (a) solving the lower triangular system $\mathbf{Ly} = \mathbf{Pb}$ by forward substitution, then (b) solving the upper triangular system $\mathbf{Ux} = \mathbf{y}$ by back substitution. Note that if one needs to solve many linear systems $\mathbf{Ax}_k = \mathbf{b}_k$ $(k = 1, 2, \ldots)$ with the same matrix and different right hand sides $\mathbf{b}_1, \mathbf{b}_2, \ldots$, then at the first step one does $O(n^3)$ work to create the LU factorization, but then reuses $\mathbf{L}$ and $\mathbf{U}$ so that each additional solve needs only $O(n^2)$ work.

2.4 Direct methods for large sparse linear systems

Direct methods such as LU and Cholesky factorizations can also be used to solve large sparse linear systems. Such systems typically have mostly zeroes, and a smart algorithm should not repeatedly zero-out zeros. Typically, for such linear systems arising from discretization of partial differential equations in 2-D domains, direct methods can be quite competitive. Overall, the success of direct methods primarily depends on our ability to construct a factorization that is sufficiently sparse. In particular, what is needed is a reordering of the rows and columns so that after the factorization is created, the factor matrices (e.g. the $\mathbf{L}$ and $\mathbf{U}$) have many less non-zeros than they would have otherwise.

It is easy to construct such a factorization for a symmetric positive definite matrix. We first apply a *fill-reducing ordering* of $\mathbf{A}$, represented as a permutation matrix $\mathbf{P}$, such that the Cholesky factorization of the reordered matrix $\widehat{\mathbf{A}} = \mathbf{P}^T\mathbf{AP} = \widehat{\mathbf{L}}\widehat{\mathbf{L}}^T$ leads to a factor $\widehat{\mathbf{L}}$ sparser than the counterpart $\mathbf{L}$ for $\mathbf{A} = \mathbf{LL}^T$. The solution to $\mathbf{Ax} = \mathbf{b}$ is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{L}^{-T}\mathbf{L}^{-1}\mathbf{b}$ or $\mathbf{x} = \mathbf{P}\widehat{\mathbf{L}}^{-T}\widehat{\mathbf{L}}^{-1}\mathbf{P}^T\mathbf{b}$, but the latter corresponds to a computationally more efficient solve thanks to the lower density (fewer nonzero entries) of $\widehat{\mathbf{L}}$. The following example shows this.

```
>> A = delsq(numgrid('B',512));   b = randn(length(A),1);
>> tic; L = chol(A,'lower'); toc;
% time to factorize the original A
Elapsed time is 2.960520 seconds.
>> tic; x = L'\(L\b); toc;
% time to solve with the original L factor
Elapsed time is 0.259174 seconds.
>> tic; p = symamd(A);    toc;
% time for fill-reducing permutation
```

```
Elapsed time is 0.251770 seconds.
>> P = speye(size(A));    P = P(:,p);
>> tic; Lhat = chol(A(p,p),'lower'); toc;
% time to factorize the permutated A
Elapsed time is 0.380372 seconds.
>> tic; xhat = P*(Lhat'\(Lhat\(P'*b))); toc;
% time to solve with the sparser L factor
Elapsed time is 0.050527 seconds.
>> disp([nnz(L) nnz(Lhat)]);

     86216840      5848939

>> disp([norm(A*x-b)/norm(b) norm(A*xhat-b)/norm(b)]);

   2.6099e-14   7.0041e-15
```

Here, $\mathbf{A}$ is a matrix arising from the discrete negative Laplacian on a 2-D region, and $\widehat{\mathbf{A}}$ is a reordered version of $\mathbf{A}$, using the symmetric approximate minimum degree permutation (`symamd`). The above result shows that the Cholesky factor of $\widehat{\mathbf{A}}$ is about 15 times sparser than that of the original $\mathbf{A}$ (about 5.8×10^6 entries vs. 8.6×10^7 entries, shown by `nnz`). Direct solves using the two Cholesky factors produce numerically equivalent solutions to the linear system $\mathbf{Ax} = \mathbf{b}$, but the one with more sparse factors needs fewer arithmetic operations and less storage. Consequently, the time needed to factorize the permutated matrix $\widehat{\mathbf{A}}$ is much less than that for factorizing the original matrix $\mathbf{A}$ (0.38 vs. 2.96 seconds), and solving the linear system $\mathbf{Ax} = \mathbf{b}$ using the sparser Cholesky factor is faster than the solve using the original Cholesky factor (0.05 vs. 0.26 seconds).

Similar strategies can be done for nonsymmetric and symmetric indefinite matrices, such as using `colamd` in MATLAB. This is not always as effective, but in general can still give considerable improvement over non-reordered matrices.

Quality software packages based on the above ideas have been developed. In MATLAB, the function `lu` with five output parameters generates a diagonal scaling matrix $\mathbf{S}$, permutation matrices $\mathbf{P}$ and $\mathbf{Q}$, a unit lower triangular $\mathbf{L}$ and an upper triangular $\mathbf{U}$, such that $\mathbf{PS}^{-1}\mathbf{AQ} = \mathbf{LU}$. The solution to $\mathbf{Ax} = \mathbf{b}$ is therefore $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{QU}^{-1}\mathbf{L}^{-1}\mathbf{PS}^{-1}\mathbf{b}$. For symmetric indefinite matrices, the function `ldl` with four output parameters generates a diagonal scaling matrix $\mathbf{S}$, a permutation $\mathbf{P}$, a unit lower triangular $\mathbf{L}$ and a block diagonal $\mathbf{D}$ with 1×1 or 2×2 blocks along the diagonal, such that $\mathbf{P}^T\mathbf{SASP} = \mathbf{LDL}^T$. The solution to $\mathbf{Ax} = \mathbf{b}$ is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{SPL}^{-T}\mathbf{D}^{-1}\mathbf{L}^{-1}\mathbf{P}^T\mathbf{Sb}$. Here is an example.

```
>> n = 512^2;
>> A = gallery('neumann', n) - speye(n);   b = randn(n,1);
>> [L,U,P,Q,R] = lu(A);
>> xa = Q*(U\(L\(P*(R\b))));
```

```
>> B = (A+A')/2;     % symmetric part of A, indefinite
>> [L,D,P,S] = ldl(B);
>> xb = S*(P*(L'\(D\(L\(P'*(S*b))))));
>> disp([norm(A*xa-b)/norm(b) norm(B*xb-b)/norm(b)]);

   9.7249e-11   4.5134e-11
```

Finally, we note that MATLAB has the 'backslash' operator, which is a very efficient solver for sparse linear systems. For dense systems, it is essentially just the PLU algorithm. For sparse matrices (which need to be of sparse type in MATLAB), it uses the efficient reorderings and factorizations described above, as well as automatically multi-threading the process. If the factorization of a coefficient matrix does not need saved (for example, if this coefficient matrix appears in only one linear system), then in MATLAB, backslash is the direct solver of choice. If it does need saved (say, if we solve multiple linear systems with the same coefficient matrix), then we follow the above examples to compute the LU, LDL or Cholesky factors of $\mathbf{A}$ first, depending on the symmetry and definiteness of $\mathbf{A}$, and solve $\mathbf{Ax} = \mathbf{b}$ using the triangular factors.

2.5 Conjugate Gradient

We discuss now an important class of iterative methods for solving large sparse linear systems approximately. Given an initial approximation $\mathbf{x}_0$, iterative methods generate a sequence of approximate solutions $\mathbf{x}_1, \mathbf{x}_2, \ldots$, that gradually converge to the solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Specifically, we consider *Krylov subspace methods*, which construct subspaces $\mathcal{K}_k(\mathbf{A}, \mathbf{r}_0) = \text{span}\{\mathbf{r}_0, \mathbf{Ar}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{k-1}\mathbf{r}_0\}$ of low dimension ($k \ll n$, e.g., $k = 100$ for $n = 10^7$), where $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$, such that good approximate solutions $\mathbf{x}_k$ can be extracted from the space $\mathbf{W}_k = \mathbf{x}_0 + \mathcal{K}_k(\mathbf{A}, \mathbf{r}_0)$.

The conjugate gradient method (CG) for symmetric positive definite (SPD) linear systems is the earliest development. Subsequent milestones include the minimal residual method (MINRES) and simplified quasi-minimal residual (SQMR) for symmetric indefinite systems, and the generalized minimal residual method (GMRES), bi-conjugate gradient stabilized (BI-CGSTAB), BI-CGSTAB(ℓ)[1] and

1 BI-CGSTAB(ℓ) is described in *BiCGstab(ℓ) for linear equations involving unsymmetric matrices with complex spectrum*, G. L. G. Sleijpen and D. R. Fokkema, Elec. Trans. Numer. Anal., Vol. 1 (1993), pp. 11–32.

induced dimension reduction[2] (IDR(s)) for nonsymmetric linear systems. We will concentrate exclusively on CG for SPD systems, and note that theory for methods for non-SPD matrices (such as GMRES, BI-CGSTAB and IDR) is available but more complex.[3] The way one *uses* these other methods in MATLAB is essentially identical to how one uses CG for solving SPD systems. Another excellent book on Krylov methods and applications is recently published by Olshanskii and Tyrtyshnikov [4].

The Conjugate Gradient (CG) method for SPD $\mathbf{Ax} = \mathbf{b}$

Choose $tol > 0$**,** $\mathbf{x}_0$**, compute** $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$**, set** $\mathbf{p}_0 = \mathbf{r}_0$**.**

For $k = 0, 1, \ldots,$ **until convergence**

$$\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \quad \mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$$

If $\frac{\|\mathbf{r}_{k+1}\|_2}{\|\mathbf{b}\|_2} \leq tol$**, exit**

$$\beta_k = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}, \quad \mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$$

End For

The CG algorithm itself is fairly straightforward. In each iteration, the computational work is one matrix-vector multiplication $\mathbf{A}\mathbf{p}_k$, two inner products $\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k$ and $\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}$ ($\mathbf{r}_k^T \mathbf{r}_k$ has been evaluated in the previous iteration), and three vector updates for $\mathbf{x}_{k+1}$, $\mathbf{r}_{k+1}$ and $\mathbf{p}_{k+1}$. The dominant cost of CG is that at every iteration, one matrix-vector product is required. If $\mathbf{A}$ were dense, then this is n^2 flops. However, if $\mathbf{A}$ were sparse with at most m nonzeros ($m \ll n$) in each row, then the cost is just mn flops per iteration. Note that for matrices arising from finite element or difference methods, m is small, maybe 5 to 20, and does not grow when n grows (we will learn more about this in later chapters). In

2 IDR(s) is described in *IDR(s): a family of simple and fast algorithms for solving large nonsymmetric systems of linear equations*, P. Sonneveld and M. B. van Gijzen, SIAM J. Sci. Comput. Vol. 31 (2008), pp. 1035–1062.

3 For more details about Krylov subspace methods, see *Iterative methods for sparse linear systems (2nd edition)* by Y. Saad, SIAM, 2003.

4 M. A. Olshanskii and E. E. Tyrtyshnikov, *Iterative Methods for Linear Systems, Theory and Applications*, SIAM (Philadelphia), 2014

fact, CG is so computationally efficient that one would not need to consider an alternative Krylov subspace method for solving large SPD linear systems.

The mathematics behind CG is not quite straightforward and probably more appropriate for a graduate level course in numerical analysis. Nevertheless, we summarize the major theorems for this algorithm.

Theorem 7. *The k^{th} CG iterate $\mathbf{x}_k$ is the unique vector in $\mathbf{W}_k = \mathbf{x}_0 + \mathcal{K}_k(\mathbf{A}, \mathbf{r}_0)$ for which the $\mathbf{A}$-norm error $\|\mathbf{e}\|_A = \sqrt{(\mathbf{x} - \mathbf{x}^*)^T \mathbf{A}(\mathbf{x} - \mathbf{x}^*)}$ or $\varphi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x}$ is minimized over all $\mathbf{x} \in \mathbf{W}_k$. Here $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$ is the exact solution.*

This theorem means that at step k, $\mathbf{x}_k$ is the best solution in the affine space $\mathbf{W}_k$. Noting that $\mathbf{W}_k$ typically has dimension k, a more delicate argument (not explored here) guarantees that CG will converge to the exact solution in at most n steps, if there were no round off errors. Unfortunately, round off error is an issue with CG, as there is an implicit orthogonalization process, and such processes are round off sensitive. Hence, CG may not get the exact solution in n steps due to the problems with round off. However, CG need not get the solution exactly for the algorithm to converge, and it may get a good approximate solution in less than n steps. Moreover, with a good preconditioner, as we mention below, the number of steps needed to converge can be reduced even further.

Theorem 8. *Let $\kappa = \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}$, where $\lambda_{\max}(\mathbf{A})$ and $\lambda_{\min}(\mathbf{A})$ denote the largest and the smallest eigenvalues of $\mathbf{A}$ (both positive). After k steps of CG, the error $\mathbf{e}_k = \mathbf{x}_k - \mathbf{x}^*$ satisfies*

$$\frac{\|\mathbf{e}_k\|_A}{\|\mathbf{e}_0\|_A} \leq 2\left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k.$$

In addition, if $\mathbf{A}$ has only m distinct eigenvalues, then CG converges to the exact solution $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$ in at most m steps.

Here, $\kappa = \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}$ for a SPD matrix equals its 2-condition number, defined as $\kappa_2(\mathbf{A}) = \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2$ (the equivalence is not valid for non-SPD matrices).

Theorem 8 suggests that a smaller condition number κ of $\mathbf{A}$ (or having eigenvalues closer together), usually leads to faster convergence of CG, but a large κ does not necessarily mean the convergence is slow, especially if A has only a few distinct eigenvalues.

We give an example to illustrate the use of CG. In the last section of this chapter, we will discuss the crucial role of a technique called 'preconditioning' and provide a MATLAB code for preconditioned CG with additional examples.

```
% a discrete negative 2-D Laplacian
>> A = delsq(numgrid('L',512));
>> rng default;
```

```
>> b = randn(length(A),1); % right-hand side vector
% use CG to find an approximate solution to relative tol 1e-8
>> x = pcg(A,b,1e-8,1500);
pcg converged at iteration 1430 to a solution with relative
residual 1e-08.
```

Although we will not discuss other types of iterative solvers, we note that if $\mathbf{A}$ were nonsymmetric and one wanted to use the solver BI-CGSTAB, then one can simply replace `pcg` in the code above with `bicgstab`. Other solvers are available and have been built into MATLAB, such as GMRES and BI-CGSTAB(ℓ).

2.6 Preconditioning and ILU

In the previous sections, we used CG to solve linear systems $\mathbf{Ax} = \mathbf{b}$ approximately. This approach is called *unpreconditioned solve*, and usually suffers from slow convergence. To speed up the convergence, a crucial technique called *preconditioning* needs to be applied. Nowadays, it is widely acknowledged that unpreconditioned methods are not efficient, and it is the preconditioning that makes these methods competitive for solving large sparse linear systems.

Generally speaking, preconditioning is a linear process represented by a matrix $\mathbf{M} \approx \mathbf{A}$, which transforms the original linear system into a new one with the same solution. Consider the 'left-sided' *preconditioned linear systems*

$$\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}.$$

An approximate solution to the preconditioned system is also an approximate solution to the original system. With a good preconditioner $\mathbf{M} \approx \mathbf{A}$, the preconditioned coefficient matrix $\mathbf{M}^{-1}\mathbf{A}$ should have most eigenvalues clustered around 1, with a small number of outliers. Consequently, if such an $\mathbf{M}$ can be constructed, iterative methods generally converge to the solution in a significantly fewer number of iterations than they do for the original system.

It is important to understand that the preconditioned coefficient matrices $\mathbf{M}^{-1}\mathbf{A}$ should in practice never be formed explicitly. Instead, when a Krylov subspace method is applied, to apply the action of $\mathbf{M}^{-1}\mathbf{A}$ to a vector $\mathbf{p}_k$, first a multiplication by $\mathbf{A}$ is done, followed by a linear solve using the $\mathbf{M}$ matrix or some linear procedure to compute the action of $\mathbf{M}^{-1}$ on $\mathbf{Ap}_k$. Hence, it is important that the preconditioner $\mathbf{M}$ be cheap to perform the action $\mathbf{M}^{-1}$ on any vector, in addition to clustering the eigenvalues of $\mathbf{M}^{-1}\mathbf{A}$.

What could be a reasonably good approximation of $\mathbf{A}$ that is also relatively cheap to apply? One of the earliest developments of general-purpose preconditioners is the incomplete LU factorization (ILU). For example, let $\mathbf{PA} = \mathbf{LU}$ be

a complete LU factorization with partial pivoting, and $\widetilde{\mathbf{L}}, \widetilde{\mathbf{U}}$ be approximations to $\mathbf{L}, \mathbf{U}$ respectively with many fewer nonzero entries. Consequently, we have $\mathbf{PA} \approx \widetilde{\mathbf{L}}\widetilde{\mathbf{U}}$ and therefore $\mathbf{M}^{-1}\mathbf{u} = \widetilde{\mathbf{U}}^{-1}\widetilde{\mathbf{L}}^{-1}\mathbf{Pu}$ can define the action of preconditioning. Note that such incomplete LU factors are obtained by dropping certain entries from the triangular factors *during* the factorization; they should not be obtained by computing the complete LU factors first (if we could perform the LU factorization, then we simply solve the system directly) before dropping entries. We remark that if the matrix is symmetric, the incomplete Cholesky is a more efficient variant of ILU that produces only one factor $\mathbf{L}$, such that $\mathbf{A} \approx \mathbf{L}\mathbf{L}^T$.

Consider now an example of using a variant of ILU to precondition a linear system. Below we solve a $195,075 \times 195,075$ SPD linear system, first with unpreconditioned CG, then using incomplete Cholesky (ICHOL) as a preconditioner. Note that MATLAB's `pcg` allows for the function $\mathbf{u} \to \mathbf{Q}\mathbf{L}^{-T}\mathbf{L}^{-1}\mathbf{Q}^T\mathbf{u}$ to be input as the preconditioner, where $\mathbf{Q}$ is the approximate minimum degree permutation matrix for reducing fill-ins, and $\mathbf{L}$ is the lower triangular ICHOL factor. We also output FLAG (0 means convergence), RELRES (relative residual of the final iterate), ITER (total number of iterations performed), and RESVEC (a vector of absolute residual norms at each step).

```
A = delsq(numgrid('L',512));
b = randn(length(A),1); % right-hand side vector
tic; [x,FLAG,RELRES,ITER,RESVEC] = pcg(A,b,1e-8,2000); toc;
[FLAG,ITER]
% precondition with incomplete Cholesky (ichol since A is SPD)
q = symamd(A);
Q = speye(size(A));    Q = Q(:,q);
L = ichol(A(q,q),struct('type','ict','droptol',1e-3));
% the function implementing the action of preconditioning
mfun = @(u)Q*(L'\(L\(Q'*u)));
tic; [x,FLAGp,RELRESp,ITERp,RESVECp] = pcg(A,b,1e-8,500,mfun); toc;
[FLAGp,ITERp]
```

The output we get is

```
Elapsed time is 9.350714 seconds.
ans =
     0   1402
Elapsed time is 1.792773 seconds.
ans =
     0    53
```

The methods both converged (flags are 0), and the preconditioner allows for convergence in 53 iterations instead of 1402. In addition, unpreconditioned CG takes 9.35 seconds to converge, whereas ICHOL-preconditioned CG only needs 1.79 seconds. RESVEC and RESVECp let us see how the methods converged.

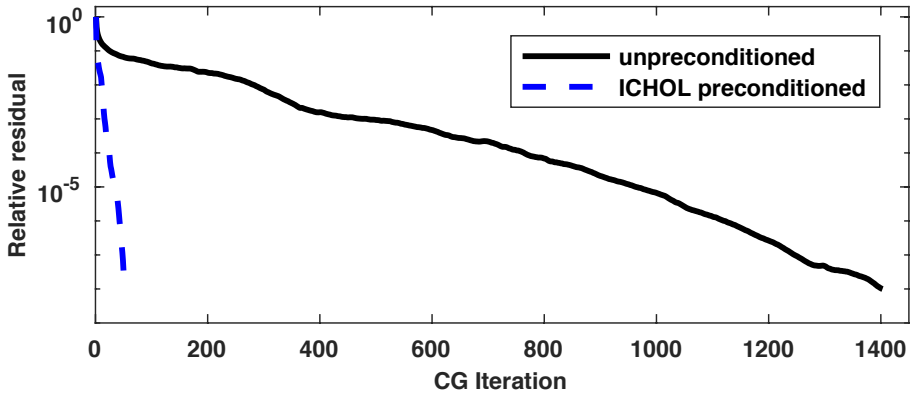

Fig. 2.1: Convergence of unpreconditioned and ICHOL-preconditioned PCG.

Finally, as mentioned above, a good preconditioner $\mathbf{M}$ should cluster the eigenvalues of $\mathbf{M}^{-1}\mathbf{A}$. Note if $\mathbf{M} = \mathbf{A}$, then all eigenvalues will be 1, and CG would converge in one step.[5] We wish we could plot all eigenvalues of $\mathbf{A}$ and $\mathbf{M}^{-1}\mathbf{A}$ to see the effect of eigenvalue clustering, but determining all the eigenvalues of a large matrix of order over tens of thousand is too expensive. Fortunately, one may see such an effect by looking at analogous smaller matrices.

Below are the eigenvalues of an analogous but much smaller $\mathbf{A}$ plotted on the complex plane, along with the eigenvalues of $\mathbf{M}^{-1}\mathbf{A}$ where $\mathbf{M} = \mathbf{L}\mathbf{L}^T$ comes from the incomplete Cholesky factorization. In fact, since CG is applicable to symmetric matrices only, the preconditioned system should be formulated as $\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T}\tilde{\mathbf{x}} = \mathbf{L}^{-1}\mathbf{b}$ (called 'split preconditioning'), where an approximate solution $\tilde{\mathbf{x}}_k$ can be used to retrieve an approximate solution $\mathbf{x}_k = \mathbf{L}^{-T}\tilde{\mathbf{x}}_k$ to the original linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$. Note that the eigenvalue distribution of the preconditioned coefficient matrix $\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T}$ (identical to those of $\mathbf{M}^{-1}\mathbf{A}$) are much closer together, exactly what preconditioning should achieve.

2.7 Accuracy in solving linear systems

An important question which we now consider is whether numerical solutions to linear systems are accurate. Some systems are very sensitive to small changes in data or roundoff error, and thus their answers are potentially inaccurate. Other

5 But in this case, the action of preconditioning $\mathbf{M}^{-1}$ applied to $\mathbf{b}$ is equivalent to the linear solve $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ itself, which is assumed to be too expensive here. A tradeoff is needed between the quality/cost of preconditioning and the number of iterations.

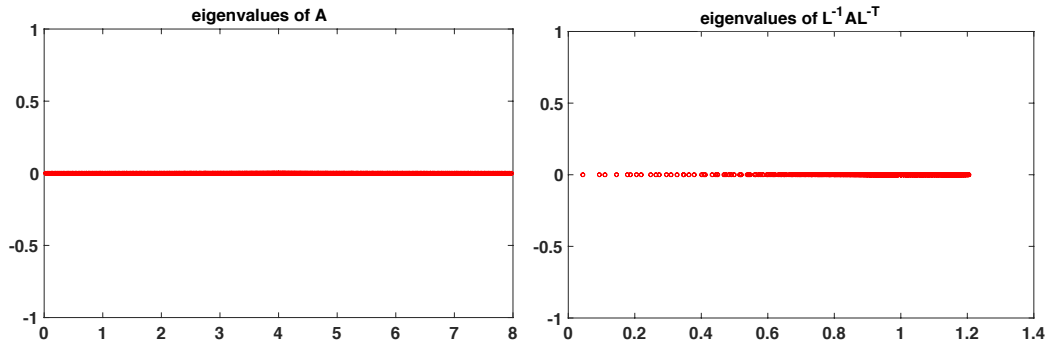

Fig. 2.2: Eigenvalues of the matrix $\mathbf{A}$ (left) and the clustered eigenvalues of the preconditioned matrix $\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T}$ (right).

systems are not sensitive, and their solutions are likely good. We will quantify the sensitivity and accuracy of systems with the notion of matrix conditioning.

There are two major sources of error that arise when solving linear systems of equations. The first comes from poor representation of the equations in the computer. This arises in the 16th digit from rounding error, but also if the equations are created from experiments, then likely there is measurement error in the fourth (or so) digit in each entry of $\mathbf{A}$ and $\mathbf{b}$. Hence although one wants to solve $\mathbf{A}\mathbf{x} = \mathbf{b}$, one is really solving $\hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}$. The question then arises, how close is $\hat{\mathbf{x}}$ to $\mathbf{x}$? Note that this type of error is not from numerical calculations, but from error in the representation of the linear system.

The second source of error comes from the calculations that produce a numerical solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$. When GE (or some variant of it) is used as the linear solver, the numerical error produced may be in the last few digits of the solution components (i.e. relative error is small). With other types of solvers such as CG, we may accept an approximate solution when the relative residual drops to 10^{-6}. We will aim to quantify this phenomenon in this chapter too.

2.7.1 Matrix and vector norms and condition number

In order to study how 'close' two vectors are, we introduce norms. Some commonly used vector norms are:

1. Infinity norm: $\|\mathbf{v}\|_\infty = \max_{1\le i\le n} |v_i|$
2. Euclidean norm (2-norm): $\|\mathbf{v}\|_2 = \left(\sum_{i=1}^{n} v_i^2\right)^{1/2}$

Although the intuitive measure of distance is the 2-norm, mathematically speaking, any function $\|\cdot\|$ that satisfies the following four properties can be considered a mathematical measure of distance, which we refer to as a norm:

1. $\|x\| \geq 0$
2. $\|x\| = 0$ if and only if $x = 0$
3. $\|\alpha x\| = |\alpha| \, \|x\|$ for all $\alpha \in \mathbb{R}$
4. $\|x + y\| \leq \|x\| + \|y\|$

The reason why we define a different way to measure the 'size' of a vector, as opposed to just using Euclidean norm for everything, is that all measures in finite dimension are equivalent (not equal, however), and so we want to pick one that is easy/cheap to perform calculations with.

To show that $\|\mathbf{v}\|_\infty$ is a norm for vectors in $\mathbb{R}^n$, we must show each of the 4 properties of a norm hold for all vectors $\mathbf{v} \in \mathbb{R}^n$.

Lemma 9. *The function* $\|\mathbf{v}\|_\infty = \max\limits_{1 \leq i \leq n} |v_i|$ *defines a norm for vectors in* $\mathbb{R}^n$.

Proof. We prove this by showing the function satisfies all 4 properties of a norm, with $\mathbf{v}$ being an arbitrary vector in $\mathbb{R}^n$.

For property 1, we can immediately see that $\|\mathbf{v}\|_\infty \geq 0$ since it is a max of non-negative numbers $|v_i|$.

Property 2 is an if-and-only-if statement, and so we must show both implications. If $\mathbf{v} = \mathbf{0}$, then each of its components is zero, and so $\|\mathbf{v}\|_\infty = \max_i |v_i| = 0$. Conversely, if $\|\mathbf{v}\|_\infty = 0$, then $\max_i |v_i| = 0$, but if the max of non-negative numbers is 0, then they must all be 0. Hence $\mathbf{v} = \mathbf{0}$.

To prove property 3, we calculate

$$\|\alpha \mathbf{v}\|_\infty = \max_{1 \leq i \leq n} |\alpha v_i| = \max_{1 \leq i \leq n} |\alpha| \, |v_i| = |\alpha| \max_{1 \leq i \leq n} |v_i| = |\alpha| \|\mathbf{v}\|_\infty,$$

where we use only properties of real numbers.

Similarly, we only need properties of real numbers to prove property 4:

$$\|\mathbf{v} + \mathbf{w}\|_\infty = \max_{1 \leq i \leq n} |v_i + w_i| \leq \max_{1 \leq i \leq n} \left(|v_i| + |w_i| \right),$$

which is upper bounded if we apply the max to both terms,

$$\max_{1 \leq i \leq n} \left(|v_i| + |w_i| \right) \leq \max_{1 \leq i \leq n} |v_i| + \max_{1 \leq i \leq n} |w_i| = \|\mathbf{v}\|_\infty + \|\mathbf{w}\|_\infty.$$

□

We will also need to discuss the 'size' of a matrix. We will consider (only) matrix norms induced by vector norms, i.e.

$$||\mathbf{A}|| = \max_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\|,$$

where $\|\cdot\|$ is a vector norm (it could be the 2 norm, ∞ norm, or other). We leave the proof that the $\|\cdot\|$ is a norm for matrices in $\mathbb{R}^{n\times n}$ as an exercise. Proving it requires proving for arbitrary $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times n}$,

1. $\|\mathbf{A}\| \geq 0$
2. $\|\mathbf{A}\| = 0$ if and only if $\mathbf{A} = \mathbf{0}$
3. $\|\alpha\mathbf{A}\| = |\alpha|\|\mathbf{A}\|$ for all $\alpha \in \mathbb{R}$
4. $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$.

Additionally, since we define matrix norms as being induced by vector norms, we also have the properties, for any $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times n}$ and $\mathbf{x} \in \mathbb{R}^n$,

5. $\|\mathbf{A}\mathbf{x}\| \leq \|\mathbf{A}\|\,\|\mathbf{x}\|$
6. $\|\mathbf{A}\mathbf{B}\| \leq \|\mathbf{A}\|\|\mathbf{B}\|$

The ∞- matrix norm is simple and easy to calculate (although we omit the proof of how to derive this formula). It is the max absolute row sum of a matrix:

$$\|\mathbf{A}\|_\infty = \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}|$$

Unfortunately, no such formula exists for the 2- matrix norm, and calculating it is not cheap. For this reason, and because all norms are equivalent in finite dimensions, we will use the ∞- matrix and vector norm for the rest of this chapter. What equivalent means essentially is that although we will get different numbers with different norms, the numbers are usually relatively close and in practice, typically of the same of similar order of magnitude.

Example 10. *Find* $\|\mathbf{A}\|_\infty$ *and* $\|\mathbf{v}\|_\infty$ *for*

$$\mathbf{A} = \begin{pmatrix} 1 & -3 & 7 \\ -3 & 1 & 10 \\ -10 & 8 & -4 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 1 \\ -6 \\ 5 \end{pmatrix}.$$

For the vector norm, we calculate

$$\|\mathbf{v}\|_\infty = \max\{1,\ 6,\ 5\} = 6,$$

For the matrix norm, again we calculate

$$\|\mathbf{A}\|_\infty = \max\{11,\ 14,\ 22\} = 22.$$

2.7.2 Condition number of a matrix

Recall the concept of matrix inverses. If a square matrix $\mathbf{A}$ is nonsingular, then there exists a matrix $\mathbf{A}^{-1}$ such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ and $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. Recall from linear algebra that $\mathbf{A}^{-1}$ can be found using Gauss-Jordan elimination (like GE, but zero out above the diagonal as well) applied to all columns of the matrix. In general we do not need to actually calculate $\mathbf{A}^{-1}$; we just need to know it exists.

We now define the condition number of a matrix:

$$\text{cond}(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|.$$

The condition number satisfies $1 \leq \text{cond}(\mathbf{A}) \leq \infty$, and its value increases as the rows (or columns) of $\mathbf{A}$ get closer to being linear dependent (i.e. the matrix is getting closer to being singular). That is, if we think of the rows (or columns) of a matrix as n dimensional vectors, if they are all perpendicular to each other, the 2-condition number is 1. However, as the smallest angle made by the vectors shrinks, the 2-condition number grows in an inversely proportional manner.

As we will see in the next section, the condition number is the fundamental measure of sensitivity in solving linear systems. If the condition number of $\mathbf{A}$ is small, then we say $\mathbf{A}\mathbf{x} = \mathbf{b}$ is a **well-conditioned** system and we expect an accurate numerical solution by stable algorithms (such as GE with partial pivoting). However, if the condition number of $\mathbf{A}$ is large, then the system is called **ill-conditioned** and the numerical solution is most likely inaccurate.

2.7.3 Sensitivity in linear system solving

We begin this section with an example.

Example 11. *Consider the system of equations*

$$\mathbf{A}\mathbf{x} = \begin{pmatrix} 6 & -2 \\ 11.5 & -3.85 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 10 \\ 17 \end{pmatrix} = \mathbf{b}.$$

The solution to this system is $\mathbf{x} = [45 \;\; 130]^T$.

However, for the slightly perturbed system

$$\widetilde{\mathbf{A}}\tilde{\mathbf{x}} = \begin{pmatrix} 6 & -2 \\ 11.5 & -3.84 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 10 \\ 17 \end{pmatrix} = \mathbf{b},$$

the solution to the system is $\tilde{\mathbf{x}} = [110 \;\; 325]^T$*!*

Why is there such a difference in solutions? After all, we made just a 'small' change to one entry of $\mathbf{A}$. The reason comes down to the sensitivity of the matrix, which can be measured by its condition number. If a matrix is ill-conditioned (large condition number), then small changes to data/entries can cause large changes in solutions. Obviously, we want to be aware of such problems. This fact can be mathematically described as follows.

Theorem 12. *Assume* $\mathbf{A}$ *is nonsingular and let* $\mathbf{Ax} = \mathbf{b}$ *and* $\hat{\mathbf{A}}\hat{\mathbf{x}} = \mathbf{b}$*. Then*

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|}{\|\hat{\mathbf{x}}\|} \leq \text{cond}(\mathbf{A}) \frac{\|\mathbf{A} - \hat{\mathbf{A}}\|}{\|\mathbf{A}\|}.$$

Proof. Since $\mathbf{b} = \mathbf{Ax}$ and $\mathbf{b} = \hat{\mathbf{A}}\hat{\mathbf{x}}$, $\mathbf{Ax} = \hat{\mathbf{A}}\hat{\mathbf{x}}$, and subtracting $\mathbf{A}\hat{\mathbf{x}}$ from both sides gives

$$\mathbf{A}(\mathbf{x} - \hat{\mathbf{x}}) = (\hat{\mathbf{A}} - \mathbf{A})\hat{\mathbf{x}}.$$

Multiplying both sides by $\mathbf{A}^{-1}$ gives $(\mathbf{x} - \hat{\mathbf{x}}) = \mathbf{A}^{-1}(\hat{\mathbf{A}} - \mathbf{A})\hat{\mathbf{x}}$, and then taking norms of the vectors on both sides gives

$$\|\mathbf{x} - \hat{\mathbf{x}}\| = \|\mathbf{A}^{-1}(\hat{\mathbf{A}} - \mathbf{A})\hat{\mathbf{x}}\| \leq \|\mathbf{A}^{-1}\| \|(\hat{\mathbf{A}} - \mathbf{A})\hat{\mathbf{x}}\| \leq \|\mathbf{A}^{-1}\| \, \|\hat{\mathbf{A}} - \mathbf{A}\| \, \|\hat{\mathbf{x}}\|,$$

with the two inequalities coming from the matrix norm properties. Next, divide both sides by $\|\hat{\mathbf{x}}\|$ and multiply the right hand side by $1 = \frac{\|\mathbf{A}\|}{\|\mathbf{A}\|}$ to reveal

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|}{\|\hat{\mathbf{x}}\|} \leq \|\mathbf{A}^{-1}\| \|\hat{\mathbf{A}} - \mathbf{A}\| \frac{\|\mathbf{A}\|}{\|\mathbf{A}\|}.$$

Using the definition of $\text{cond}(\mathbf{A})$ and rearranging finishes the proof. □

The consequence of this theorem is that the relative change made to $\mathbf{A}$ (or, equivalently, the error in the representation of $\mathbf{A}$) is magnified in the solution by up to $\text{cond}(\mathbf{A})$. In fact, we can think of the theorem as saying: the percent difference between the solutions is bounded by the percent difference between the matrices times the condition number.

Example 13. *Suppose we wish to solve* $\mathbf{Ax} = \mathbf{b}$*, but we are sure of only the first 6 digits in each entry of* $\mathbf{A}$*. If the condition number of* $\mathbf{A}$ *is* 10^2 *in* ∞*-norm, how many digits of the solution do we expect will agree with the true solution?*

Here we are solving $\hat{\mathbf{A}}\hat{\mathbf{x}} = \mathbf{b}$*, since we are solving a linear system but not using* $\mathbf{A}$ *exactly. Using the infinity norm,* $\frac{\|\hat{\mathbf{A}} - \mathbf{A}\|_\infty}{\|\mathbf{A}\|_\infty} \leq 10^{-6}$ *since we are sure of only the first 6 digits of* $\mathbf{A}$ *(with the infinity norm, 'digits of agreement' and 'relative difference' are directly related in this way). The theorem shows that*

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|_\infty}{\|\hat{\mathbf{x}}\|_\infty} \leq \text{cond}_\infty(\mathbf{A}) \frac{\|\hat{\mathbf{A}} - \mathbf{A}\|_\infty}{\|\mathbf{A}\|_\infty} \leq 10^2 \cdot 10^{-6} = 10^{-4},$$

which says precisely that each entry of your solution $\hat{\mathbf{x}}$ *will agree with that of the true solution* $\mathbf{x}$ *up to at least 4 digits.*

There is a similar result for changes in the vector $\mathbf{b}$, as follows.

Theorem 14. *Let* $\mathbf{Ax} = \mathbf{b}$ *and* $\mathbf{A}\hat{\mathbf{x}} = \hat{\mathbf{b}}$*. Then*

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \text{cond}(\mathbf{A}) \frac{\left\|\mathbf{b} - \hat{\mathbf{b}}\right\|}{\|\mathbf{b}\|}.$$

In general, for a given linear system, the best accuracy we can hope for is

$$\text{Relative error in numerical solution} \approx \text{cond}(\mathbf{A}) \times \text{machine epsilon}$$

This ignores possible error in numerical methods for $\hat{\mathbf{x}}$, so error could be worse if $\mathbf{b}$ comes from data measurements. Thus, knowing $\text{cond}(\mathbf{A})$ is very important, as it will determine how much we should believe our computed solution.

Returning now to our example, we can answer the question of why the two solutions in the above example differed by so much. We calculate the condition number first, via

$$\mathbf{A}^{-1} = \begin{pmatrix} 38.5 & -20 \\ 115 & -60 \end{pmatrix},$$

$$\text{cond}(\mathbf{A}) = \|\mathbf{A}\|_\infty \left\|\mathbf{A}^{-1}\right\|_\infty = \max\{8, 15.35\} \cdot \max\{58.5, 175\} = 2686.25.$$

The relative change in $\mathbf{A}$ is $\frac{\left\|\mathbf{A} - \widetilde{\mathbf{A}}\right\|}{\|\mathbf{A}\|} \approx \frac{0.01}{15.35} = 6.515 \times 10^{-4}$. From the theorem, we can expect the relative difference between $\mathbf{x}$ and $\hat{\mathbf{x}}$ to be bounded by

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|}{\|\hat{\mathbf{x}}\|} \leq 2686.25 \cdot 6.515 \times 10^{-4} = 1.75,$$

and thus, we can't even expect accuracy in the first digit of the solution entries, which is precisely what we observe. This terrible outcome is caused directly by the condition number being large relative to the size of the perturbation.

2.7.4 Error and residual in linear system solving

Assume now that we can represent $\mathbf{A}$ and $\mathbf{b}$ exactly, and let's consider a different type of error. All numerical methods for solving $\mathbf{Ax} = \mathbf{b}$ introduce error; that is, they almost surely find $\hat{\mathbf{x}} \neq \mathbf{x}$. Unfortunately, we usually never know $\mathbf{x}$, but we

still want to have an idea of the size of the error $\mathbf{e} = \hat{\mathbf{x}} - \mathbf{x}$. What we do know, if given an approximation $\hat{\mathbf{x}}$, is the residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}$. Residual and error are different, but related:

$$\mathbf{Ae} = \mathbf{A}(\hat{\mathbf{x}} - \mathbf{x}) = \mathbf{A}\hat{\mathbf{x}} - \mathbf{Ax} = \mathbf{A}\hat{\mathbf{x}} - \mathbf{b} = \mathbf{r}.$$

Multiplying both sides of $\mathbf{Ae} = \mathbf{r}$ by $\mathbf{A}^{-1}$ gives $\mathbf{e} = \mathbf{A}^{-1}\mathbf{r}$, and then taking norms of both sides yields

$$\|\mathbf{e}\| = \|\mathbf{A}^{-1}\|\|\mathbf{r}\| \leq \|\mathbf{A}^{-1}\|\|\mathbf{r}\|,$$

where the last inequality came from a property of matrix norms. Dividing both sides by $\|\hat{\mathbf{x}}\|$, and multiplying the right hand side by $\frac{\|\mathbf{A}\|}{\|\mathbf{A}\|}$ yields

$$\frac{\|\mathbf{e}\|}{\|\hat{\mathbf{x}}\|} \leq \text{cond}(A)\frac{\|\mathbf{r}\|}{\|\mathbf{A}\|\|\hat{\mathbf{x}}\|}.$$

The left hand side is the relative error of the solution, and the right hand side is the condition number of $\mathbf{A}$ times the relative residual $\frac{\|\mathbf{r}\|}{\|\mathbf{A}\|\|\hat{\mathbf{x}}\|}$. With $\hat{\mathbf{x}}$ computed by numerically stable algorithms such as GE with partial pivoting, the relative residual $\frac{\|\mathbf{r}\|}{\|\mathbf{A}\|\|\hat{\mathbf{x}}\|}$ is on the order of machine epsilon.[6] Overall, direct solvers for sparse matrices typically produce approximations that have very small relative residuals, e.g., smaller than 10^{-12}. Iterative solvers, such as CG or GMRES, often use relative residual size as a stopping criteria, and usually on the order of 10^{-6} or 10^{-8}. Hence, if there is a large condition number compared to the relative residual, then the error $\frac{\|\hat{\mathbf{x}}-\mathbf{x}\|}{\|\hat{\mathbf{x}}\|}$ may be large!

2.8 Exercises

1. By hand, solve the triangular linear systems

$$\begin{pmatrix} 2 & 3 & 4 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 11 \\ 5 \\ 9 \end{pmatrix}$$

6 Sometimes the relative residual is also defined as $\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}$, but this quantity itself can be as large as cond($\mathbf{A}$) times machine epsilon.

and

$$\begin{pmatrix} 4 & 0 & 0 \\ -1 & 5 & 0 \\ 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -8 \\ 7 \\ 7 \end{pmatrix}.$$

2. Write MATLAB code for the forward substitution (lower triangular), and test it on the second system in Exercise 1.
3. By hand, use (a) Gaussian elimination with no pivoting, (b) LU factorization with no pivoting, and (c) LU factorization with partial pivoting, to solve the system of linear equations

$$\begin{pmatrix} 1 & -1 & -1 \\ -1 & -3 & 1 \\ 2 & 6 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -6 \\ 2 \\ -1 \end{pmatrix}.$$

4. By hand, compute the LU factorization of $\mathbf{A} = \begin{pmatrix} \frac{\epsilon}{2} & -1 \\ 1 & 1 \end{pmatrix}$ with partial pivoting, taking round-off errors into consideration. Check if the product of the *computed* $\widehat{\mathbf{L}}$ and $\widehat{\mathbf{U}}$ is close to $\mathbf{PA}$ (see the example in Section 2.3).
5. Consider a 5×5 tridiagonal matrix $\mathbf{A}$ (nonzeros appear only on the main diagonal, and the first superdiagonal and subdiagonal), and assume that LU factorization with no pivoting is stable for this $\mathbf{A}$. Where would the nonzero entries appear in the $\mathbf{L}$ and $\mathbf{U}$ factors, respectively? Generalize your observation to $n \times n$ tridiagonal matrices. How many flops are needed to obtain such an LU factorization and forward/back substitutions?
6. In MATLAB, run `A = delsq(numgrid('B',256));` and `B = A + sprandn(A);` to construct an SPD and a nonsymmetric matrix respectively. Use the techniques in Section 2.4 to solve the linear systems $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{Bx} = \mathbf{b}$, where $\mathbf{b} = [1, \ldots, 1]^T$. Check if all the relative residual norms $\frac{\|\mathbf{r}\|}{\|\mathbf{A}\|\|\hat{\mathbf{x}}\|}$, $\frac{\|\mathbf{r}\|}{\|\mathbf{B}\|\|\hat{\mathbf{x}}\|}$ and $\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}$ are sufficiently small.
7. Let us derive some preliminary properties of the conjugate gradient method. For a symmetric positive definite matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, a vector $\mathbf{b} \in \mathbb{R}^n$, and any nonzero vector $\mathbf{x} \in \mathbb{R}^n$, define $\varphi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x}$.
   (a) Given $\mathbf{x}_k, \mathbf{p}_k \in \mathbb{R}^n \setminus \{0\}$, find the scalar α_k such that $\varphi(\mathbf{x}_k + \alpha_k\mathbf{p}_k)$ is minimized. Simplify the result by introducing the residual $\mathbf{r}_k = \mathbf{b} - \mathbf{Ax}_k$.

(b) Let $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$, with α_k defined in (a), such that $\mathbf{r}_{k+1} = \mathbf{b} - \mathbf{A}\mathbf{x}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$. Show that $\mathbf{r}_{k+1} \perp \mathbf{p}_k$. Let $\beta_k = -\frac{\mathbf{r}_{k+1}^T \mathbf{A}\mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k}$ and $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$. Show that $\mathbf{p}_{k+1} \perp \mathbf{A}\mathbf{p}_k$.

(c) From $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$, write $\mathbf{r}_{k+1}$ in terms of $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$. Then, replace k with $k+1$ in $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$ so that an expression of $\mathbf{r}_{k+2}$ is obtained. Show that $\mathbf{r}_{k+1} \perp \mathbf{r}_{k+2}$, using the relations shown in (b).

8. (a) In MATLAB, run the following commands:

```
n = 32; A = delsq(numgrid('S',n+2));
b = ones(length(A),1);
x = pcg(A,b,1e-8,length(A)-1);
disp(condest(A));
```

Increase n to $64, 128, 256$ and 512, and make a table summarizing n, the number of CG iterations needed to reach the relative tolerance 10^{-8}, and the 2-condition number (estimate) of $\mathbf{A}$. Are the results consistent with Theorem 8 in Section 2.5? (Disregard the fact that we ask CG with $\mathbf{x}_0 = \mathbf{0}$ to terminate when $\frac{\|\mathbf{r}_k\|_2}{\|\mathbf{r}_0\|_2} \leq 10^{-8}$ instead of $\frac{\|\mathbf{e}_k\|_A}{\|\mathbf{e}_0\|_A} \leq 10^{-8}$)

(b) Run the following commands:

```
un = ones(256,1); D = diag([un; 2e1*un; 4e2*un; 1.6e5*un]);
rng default;
V = randn(size(D));  [V,~] = qr(V);
A = V*D*V';  A = (A+A')/2;  b = ones(length(A),1);
x = pcg(A,b,1e-8,length(A)-1);
disp(cond(A));
```

In how many iterations does CG converge? Compare this result to that in part (a), and provide an explanation of the iteration count here.

9. Consider a matrix that arises from solving the 2D Poisson problem. This matrix $\mathbf{A}$ has 4 for each diagonal entry, and -1's on each of the first upper and lower diagonals, and again on the $\sqrt{n} \pm 1$ diagonals. The rest of the matrix is all zeros. You can create such a 2500×2500 matrix in MATLAB

```
A = gallery('poisson',50);
```

Note the 50 gets squared, as it has to do with number of unknowns in each dimension. This matrix is automatically created as 'sparse' type.

Create such matrices of size 2500, 3600, 4900, ..., and vectors $\mathbf{b}$ of all ones of the commensurate lengths, respectively. Solve the linear systems $\mathbf{A}\mathbf{x} = \mathbf{b}$ two ways, and keep track of timings: First, solve using the sparse direct solver in MATLAB by solving the linear system with backslash. Next, repeat the process but using non-sparse matrices (e.g. `A2 = full(A);`). Compare timings of the sparse vs. non-sparse solve.

10. Given $\mathbf{A} = \begin{pmatrix} 2 & 4 \\ -3 & -6.001 \end{pmatrix}$ and $\mathbf{b} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$
    (a) Solve $\mathbf{Ax} = \mathbf{b}$ using backslash.
    (b) Change the second entry of $\mathbf{b}$ to 3.01, and solve for the new solution.
    (c) What is the relative difference between the solutions?
    (d) Does this agree with the Theorem 14?
11. Repeat Exercise 10, but now using the matrix $\mathbf{A} = \begin{pmatrix} 2 & 4 \\ 3 & 6.001 \end{pmatrix}$.
12. Prove that $\|\mathbf{v}\|_\infty = \max_{1 \le i \le n} |v_i|$ defines a norm for vectors in $\mathbb{R}^n$.
13. Prove properties 5 and 6 of vector-norm-induced matrix norms (assume that the associated vector norms satisfy the norm properties).
14. Prove Theorem 14.