Lecture 4: Solving linear systems: LU and iterative methods

TANA21/22: Beräkningsmatematik — *Andrew R. Winters* — *Autumn 2019*

We expand upon the work from the previous lecture and go into detail for numerical methods to solve a square linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$. In the first half of this lecture, we focus on a factorization technique that comes from our knowledge of Gauss-Jordan elimination. This involves working with triangular systems and then solving for the vectors of unknowns. This is a type of *direct* method to numerically solve the system. For the second half of this lecture, we will work with alternative *iterative* techniques to approximate the solution vector. To do so, we will revisit the concept of a fixed point iteration from Lecture 2. However, there are differences which we clarify such as what does a contraction mapping mean in the context of a system of equations.

4.1 LU factorization

From last time, we discussed a solution approach where we applied subsequent Gauss-Jordan eliminations and drove the system to an upper triangular form. The upper triangular system was then efficiently solved with a backward substitution strategy. At the end of last lecture we saw that the combined action of all the Gauss-Jordan eliminations was a *unit lower triangular matrix*, i.e. the values above the main diagonal are zero, the values along the main diagonal are all one, and the values below the main diagonal can be nonzero. In turn, this allowed us to write the matrix $\mathbf{A}$ as the multiplication of this unit lower triangular matrix and an upper triangular matrix, e.g.

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ l_{n1} & \cdots & l_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{pmatrix} = \mathbf{LU}.$$

This is a type of *factorization* strategy where we rewrite the matrix $\mathbf{A}$ into a form that is easier to solve. In particular, this is known as the **LU factorization** (also sometimes called the LU decomposition).

The LU factorization is a powerful technique to directly solve square linear systems. However, we need to dig into the details to ensure that the factorization and subsequent solution of the systems is reliable.

Recall that we interpreted Gauss-Jordan eliminations into a useful *elementary matrix* form. Theses matrices were particularly *easy to invert* and we saw that new transformations did not affect the previous ones:

Example 4.1. Let $n = 4$ and say that a transformation $\mathbf{M}_1$ has already zeroed out the first column of $\mathbf{A}$ below the main diagonal. If we apply a second transformation $\mathbf{M}_2$ we have the structure

$$\mathbf{M}_2\mathbf{M}_1\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -l_{32} & 1 & 0 \\ 0 & -l_{42} & 0 & 1 \end{pmatrix} \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix} = \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & *' & *' \\ 0 & 0 & *' & *' \end{pmatrix},$$

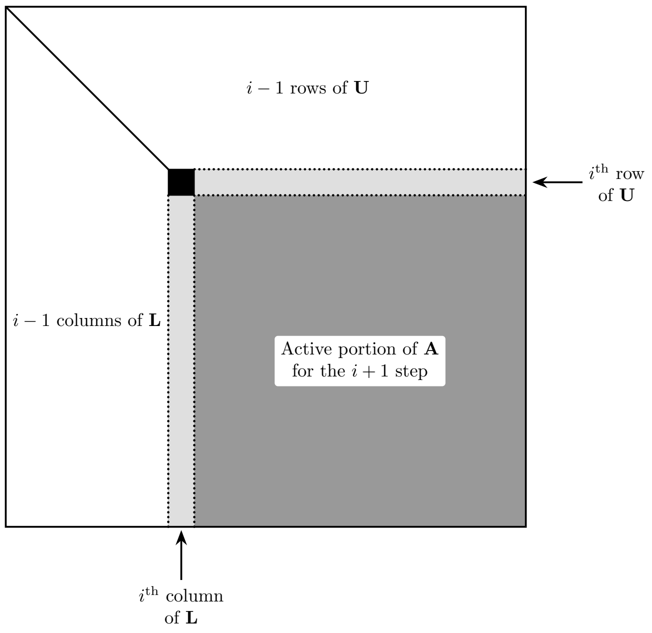
where the $*$ are placeholders for arbitrary values. We see that the application of the transformation $\mathbf{M}_2$ to $\mathbf{A}$ **does not** destroy the 0 elements a previous transformation $\mathbf{M}_1$ introduced in the first column. Also, the inverse of the transformation $\mathbf{M}_2$ is

$$\mathbf{M}_2^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & l_{32} & 1 & 0 \\ 0 & l_{42} & 0 & 1 \end{pmatrix},$$

where we simply flip the signs of the values below the main diagonal. ⧫

This example reveals something else important: There is a "static" part of the matrix that remains unchanged by new transformations and an "active" part of the matrix where values are updated/changed. This is excellent news because we can exploit this structure to save on memory and create the **LU** factorization *in place*. Therefore, the storage complexity remains $\mathcal{O}(n^2)$, but we will not store an unnecessary zero values. Before we walk through an example of this in place strategy to factorize **A**, we give a visual representation (at the right) of what the factorization strategy produces during the i^{th} step:

- Column i of the unit lower triangular matrix **L**.
- Row i of the upper triangular matrix **U**.
- Updates to the remaining "active" portion of **A** that has dimension $(n-i)\times(n-i)$.


$i-1$ rows of **U**
i^{th} row of **U**
$i-1$ columns of **L**
Active portion of **A** for the $i+1$ step
i^{th} column of **L**

Keep in mind that the i^{th} step *does not* change the already created $i-1$ columns of **L** and $i-1$ rows of **U**. Also, implicit to this storage strategy is that the matrix **L** has ones along its main diagonal. So, we don't actually store them, but we must keep in mind when we go to solve the system.

This introduces an important distinction of working with matrices and other linear algebra objects in computational methods. A matrix is a *mathematical object* whereas storing the factorization of a matrix, such as the in place **LU**, is done in an array which is a *data structure*. We will distinguish between the two as we work through an example.

Remark 4.1. For examples throughout this lecture, we will highlight within the array structure the new rows of **U** in pink, the new columns of **L** in blue, and the active part of **A** in orange. ⋈

Example 4.2. Form the in place **LU** factorization of the matrix **A** where $n=4$. We initialize the array with the same values as those in the matrix **A**:

$$\mathbf{A}=\begin{pmatrix}5&1&3&1\\10&5&12&3\\5&10&23&5\\15&6&19&7\end{pmatrix},\qquad \texttt{array}=\begin{bmatrix}5&1&3&1\\10&5&12&3\\5&10&23&5\\15&6&19&7\end{bmatrix}.$$

Now, we apply the first set of Gauss-Jordan eliminations to zero out the first column

$$\begin{pmatrix}5&1&3&1\\0&3&6&1\\0&9&20&4\\0&3&10&4\end{pmatrix}=\begin{pmatrix}1&0&0&0\\-2&1&0&0\\-1&0&1&0\\-3&0&0&1\end{pmatrix}\begin{pmatrix}5&1&3&1\\10&5&12&3\\5&10&23&5\\15&6&19&7\end{pmatrix},$$

$$\texttt{array}=\begin{bmatrix}5&1&3&1\\10&5&12&3\\5&10&23&5\\15&6&19&7\end{bmatrix}\quad\Rightarrow\quad\texttt{array}=\begin{bmatrix}5&1&3&1\\2&3&6&1\\1&9&20&4\\3&3&10&4\end{bmatrix}.$$

Note that we flip the signs when placing the values into the array structure, which is effectively taking the "inverse" of the Gauss-Jordan matrices. For the second step we zero out the second column and update the remaining portion of $\mathbf{A}$

$$\begin{pmatrix} 5 & 1 & 3 & 1 \\ 0 & 3 & 6 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 4 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -3 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 5 & 1 & 3 & 1 \\ 0 & 3 & 6 & 1 \\ 0 & 9 & 20 & 4 \\ 0 & 3 & 10 & 4 \end{pmatrix},$$

$$\texttt{array} = \begin{bmatrix} 5 & 1 & 3 & 1 \\ 2 & 3 & 6 & 1 \\ 1 & 9 & 20 & 4 \\ 3 & 3 & 10 & 4 \end{bmatrix} \quad \Rightarrow \quad \texttt{array} = \begin{bmatrix} 5 & 1 & 3 & 1 \\ 2 & 3 & 6 & 1 \\ 1 & 3 & 2 & 1 \\ 3 & 1 & 4 & 3 \end{bmatrix}.$$

We see that the first row of $\mathbf{U}$ and the first column of $\mathbf{L}$ are unaffected by this new update. Finally, we perform one more Gauss-Jordan elimination to complete the $\mathbf{LU}$ factorization

$$\begin{pmatrix} 5 & 1 & 3 & 1 \\ 0 & 3 & 6 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -2 & 1 \end{pmatrix} \begin{pmatrix} 5 & 1 & 3 & 1 \\ 0 & 3 & 6 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 4 & 3 \end{pmatrix},$$

$$\texttt{array} = \begin{bmatrix} 5 & 1 & 3 & 1 \\ 2 & 3 & 6 & 1 \\ 1 & 3 & 2 & 1 \\ 3 & 1 & 4 & 3 \end{bmatrix} \quad \Rightarrow \quad \texttt{array} = \begin{bmatrix} 5 & 1 & 3 & 1 \\ 2 & 3 & 6 & 1 \\ 1 & 3 & 2 & 1 \\ 3 & 1 & 2 & 1 \end{bmatrix}.$$

Now the array compactly stores the $\mathbf{LU}$ decomposition

$$\mathbf{A} = \begin{pmatrix} 5 & 1 & 3 & 1 \\ 10 & 5 & 12 & 3 \\ 5 & 10 & 23 & 5 \\ 15 & 6 & 19 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 1 & 3 & 1 & 0 \\ 3 & 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} 5 & 1 & 3 & 1 \\ 0 & 3 & 6 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \mathbf{LU}$$

of the matrix $\mathbf{A}$. ♦

Now we have a strategy to obtain and compactly store the $\mathbf{LU}$ factorization of the matrix $\mathbf{A}$. How do we actually solve the system? Recall we used back substitution to solve the upper triangular system $\mathbf{Ux} = \mathbf{c}$. But where does the vector $\mathbf{c}$ come from in this context? Through some algebraic manipulation we find that this is where the matrix $\mathbf{L}$ comes into play:

$$\mathbf{Ax} = \mathbf{b} \quad \Rightarrow \quad \mathbf{LUx} = \mathbf{b} \quad \Rightarrow \quad \begin{array}{c} \mathbf{Lc} = \mathbf{b} \\ \mathbf{Ux} = \mathbf{c} \end{array}$$

So, the computation of the solution vector $\mathbf{x}$ comes from a three-step process:

1. Create the $\mathbf{LU}$ factorization of $\mathbf{A}$.
2. Compute the intermediate vector $\mathbf{c}$ by solving a lower triangular system using *forward substitution*.
3. The vector $\mathbf{c}$ becomes the right hand side for the previously discussed *backward substitution* strategy.

The forward substitution technique is very similar to backward substitution apart from the direction in which direction we iterate through the information. Provided the main diagonal entries of the matrix $\mathbf{L}$ are nonzero (as is the case for the $\mathbf{LU}$ factorization) we can compute the solution for $\mathbf{Lc} = \mathbf{b}$ as

$$c_1 = \frac{b_1}{l_{11}}, \qquad c_i = \frac{1}{l_{ii}} \left(b_i - \sum_{j=1}^{i-1} l_{ij} c_j \right), \; i = 2, \ldots, n.$$

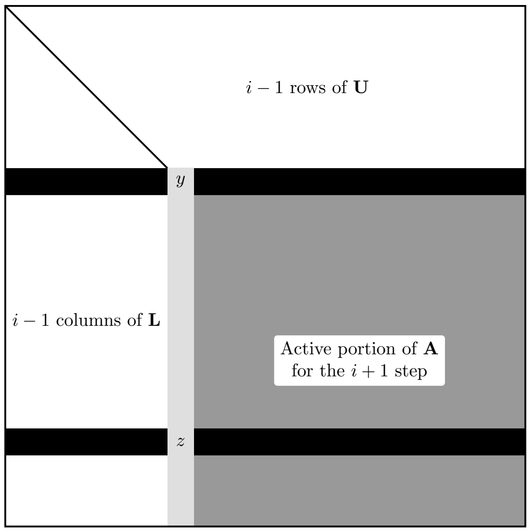
The computational complexity for the forward substitution is $\mathcal{O}(n^2)$ (similar to the backward substitution). Therefore, the overall complexity of the algorithm is dominated by the $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ work necessary to create the **LU** factorization. Pseudocode for the forward substitution process is found below.

Algorithm 4.1: *Forward substitution*: Solve a lower triangular linear system

```
Procedure Forward Substitution
Input: $\mathbf{L}$, $\mathbf{b}$, $n$ // lower triangular square matrix, right hand side vector, and size
$c_1 \leftarrow b_1/l_{11}$
for $i = 2$ to $n$ do
    for $j = 1$ to $i - 1$ do
        $b_i \leftarrow b_i - l_{ij} * c_j$
    $c_i \leftarrow b_i/l_{ii}$
Output: $\mathbf{c}$ // solution vector
End Procedure Forward Substitution
```

Remark 4.2. This pseudocode is written for a general lower triangular system. For the **LU** factorization we know that the matrix **L** is unit lower triangular. Therefore, we do not need to perform the division of the l_{ii} elements, which decreases computational complexity. ⋈

Example 4.3. Continuing the previous example we solve the system with the right hand side $\mathbf{b} = (30, 82, 101, 123)^T$ such that the forward substitution yields the intermediate vector $\mathbf{c} = (30, 22, 5, 1)^T$ and the backward substitution gives the solution vector $\mathbf{x} = (4, 3, 2, 1)^T$. ⧫


$i-1$ rows of $\mathbf{U}$
y
$i-1$ columns of $\mathbf{L}$
Active portion of $\mathbf{A}$ for the $i+1$ step
z

Next, we address the stability of the **LU** factorization. In the examples considered so far the algorithm never "broke," but what happens if the entry along the main diagonal becomes zero (or close to it)? Then dividing by a small number in the Gauss-Jordan elimination becomes a very large number and we have seen that this can lead to accuracy issues when working in finite precision. We will denote the current element that will divide in the Gauss-Jordan elimination as the **pivot element**. To avoid these issues of dividing by a small pivot element we think back to the possible elementary row operations and recall that we are allowed to exchange rows of the matrix *without* changing the solution.

Definition 4.1 (Pivoting). The interchanging of rows in order to determine a suitably large nonzero pivot element is called **pivoting**. In general, pivoting is necessary to guarantee existence and numerical stability of the **LU** factorization. ◀

There are many strategies to select the pivot element. We choose a strategy referred to as ***partial row pivoting*** where we search the first column of the active sub-matrix and select the element of *maximum magnitude*. We illustrate this process on the left where we exchange the current row that has leading element y with the row that has leading element z with the largest magnitude throughout the whole column.

Remark 4.3. Note that the partial row pivoting will also exchange rows of the lower triangular matrix **L**. ⋈

Remark 4.4. One consequence of partial row pivoting is that all the off-diagonal elements in the unit lower triangular matrix **L** will have a magnitude less than one. ⋈

Example 4.4. To demonstrate the necessity of pivoting consider a 2×2 linear system where

$$\mathbf{A} = \begin{pmatrix} -10^{-9} & 1 \\ 1 & -1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 - 10^{-9} \\ 0 \end{pmatrix}.$$

Without pivoting we use elimination to find

$$\begin{pmatrix} -10^{-9} & 1 \\ 0 & -1 + 10^{9} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 - 10^{-9} \\ -1 + 10^{9} \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \frac{(1-10^{-9})-1}{-10^{-9}} \\ 1 \end{pmatrix}.$$

The calculation of x_1 is ill-conditioned because it involves the subtraction of nearby numbers and accuracy will be lost in finite precision. However, if we pivot the two rows such that the largest value in the first column is on the main diagonal *and then* preform elimination we see

$$\begin{pmatrix} 1 & -1 \\ -10^{-9} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 - 10^{-9} \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} 1 & -1 \\ 0 & 1 - 10^{-9} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 - 10^{-9} \end{pmatrix}$$

Now the computation of the solution is well-conditioned and we correctly find that $\mathbf{x} = (1,1)^T$. ♦

Definition 4.2 (Permutation matrix). We represent the exchange of two rows of a matrix $\mathbf{A}$ using a **permutation matrix** denoted as $\mathbf{P}$. It is essentially a scrambled version of the identity matrix. For example, take $n = 3$ then the permutation of $\mathbf{A}$

$$\mathbf{PA} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} a_{31} & a_{32} & a_{33} \\ a_{21} & a_{22} & a_{23} \\ a_{11} & a_{12} & a_{13} \end{pmatrix},$$

exchanges the first and third rows. ◄

Remark 4.5. Permutation matrices are a type of *orthogonal* matrix. One consequence of this is that they are trivial to invert because $\mathbf{P}^{-1} = \mathbf{P}^T$. ⋈

Definition 4.3 (LU with partial row pivoting). If the matrix $\mathbf{A}$ is nonsingular then there exists a product of row permutations $\mathbf{P} = \mathbf{P}_{n-1} \cdots \mathbf{P}_1$, a unit lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$ such that

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

That is, partial row pivoting is equivalent to computing the $\mathbf{LU}$ factorization of a permuted version of the original system $\mathbf{A}$. ◄

Remark 4.6. This only alters the solution strategy slightly. We simply must keep track of the permutations that occur because we must apply them to the right hand side as well to ensure the correct solution is computed:

$$\mathbf{Ax} = \mathbf{b} \quad \Rightarrow \quad \mathbf{PAx} = \mathbf{Pb} \quad \Rightarrow \quad \mathbf{LUx} = \widetilde{\mathbf{b}} \quad \Rightarrow \quad \begin{array}{c} \mathbf{Lc} = \widetilde{\mathbf{b}} \\ \mathbf{Ux} = \mathbf{c} \end{array}$$

where we, again, apply a forward/backward substitution strategy to compute $\mathbf{x}$. ⋈

Example 4.5. Use partial row pivoting and find the $\mathbf{LU}$ factorization of the matrix $\mathbf{A}$ and store the decomposition in place within an array. For the first step we pivot the first and third rows such that the pivot element is maximal and then perform Gauss-Jordan elimination

$$\mathbf{A} = \begin{pmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{pmatrix}, \quad \mathbf{P}_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} 6 & 18 & -12 \\ 0 & -2 & 2 \\ 0 & 8 & 16 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{1}{3} & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{pmatrix}.$$

For the array storing the factorization in place we, as before, start by saving the original values and order of $\mathbf{A}$ in the array. We then permute the rows, compute the first column of the matrix $\mathbf{L}$ and apply the elimination to the "active" 2×2 portion of the matrix:

$$\texttt{array} = \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix} \overset{\text{permute}}{\Rightarrow} \begin{bmatrix} 6 & 18 & -12 \\ 2 & 4 & -2 \\ 3 & 17 & 10 \end{bmatrix} \overset{\text{get } \mathbf{L}_1}{\Rightarrow} \begin{bmatrix} 6 & 18 & -12 \\ \frac{1}{3} & 4 & -2 \\ \frac{1}{2} & 17 & 10 \end{bmatrix} \overset{\text{eliminate}}{\Rightarrow} \begin{bmatrix} 6 & 18 & -12 \\ \frac{1}{3} & -2 & 2 \\ \frac{1}{2} & 8 & 16 \end{bmatrix}.$$

The second step also requires a pivot to swap the second and third rows and we find

$$\begin{pmatrix} 6 & 18 & -12 \\ 0 & -2 & 2 \\ 0 & 8 & 16 \end{pmatrix}, \quad \mathbf{P}_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} 6 & 18 & -12 \\ 0 & 8 & 16 \\ 0 & 0 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{1}{4} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 6 & 18 & -12 \\ 0 & -2 & 2 \\ 0 & 8 & 16 \end{pmatrix}.$$

To update the in place storage of the factorization we have the steps

$$\texttt{array} = \begin{bmatrix} 6 & 18 & -12 \\ \frac{1}{3} & -2 & 2 \\ \frac{1}{2} & 8 & 16 \end{bmatrix} \overset{\text{permute}}{\Rightarrow} \begin{bmatrix} 6 & 18 & -12 \\ \frac{1}{2} & 8 & 16 \\ \frac{1}{3} & -2 & 2 \end{bmatrix} \overset{\text{get } \mathbf{L}_2}{\Rightarrow} \begin{bmatrix} 6 & 18 & -12 \\ \frac{1}{2} & 8 & 16 \\ \frac{1}{3} & -\frac{1}{4} & 2 \end{bmatrix} \overset{\text{eliminate}}{\Rightarrow} \begin{bmatrix} 6 & 18 & -12 \\ \frac{1}{2} & 8 & 16 \\ \frac{1}{3} & -\frac{1}{4} & 6 \end{bmatrix}.$$

For verification, we have the total permutation matrix

$$\mathbf{P} = \mathbf{P}_2\mathbf{P}_1 = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

and we see that the factorization gives

$$\mathbf{PA} = \begin{pmatrix} 6 & 18 & -12 \\ 3 & 17 & 10 \\ 2 & 4 & -2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{3} & -\frac{1}{4} & 1 \end{pmatrix} \begin{pmatrix} 6 & 18 & -12 \\ 0 & 8 & 16 \\ 0 & 0 & 6 \end{pmatrix} = \mathbf{LU},$$

as expected. ⧫

We close our discussion on the $\mathbf{LU}$ factorization with a special case of the decomposition for a *symmetric positive definite (s.p.d)* matrix. A matrix is s.p.d. provided as long as $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for any vector $\mathbf{x} \neq 0$. Symmetry ($\mathbf{A} = \mathbf{A}^T$) is easy to check, but positive definiteness is more subtle. A sufficient condition is that all the diagonal elements of $\mathbf{A}$ are positive and the largest element *is on* the diagonal

$$a_{kk} > 0, \quad k = 1, \dots, n \qquad \text{and} \qquad \max_{i,j} |a_{ij}| = \max_{k} a_{kk}.$$

Definition 4.4 (Cholesky factorization). Every symmetric positive definite matrix $\mathbf{A}$ can be factored symmetrically and *without* pivoting as $\mathbf{A} = \mathbf{C}\mathbf{C}^T$ where $\mathbf{C}$ is a lower triangular matrix. ◀

Remark 4.7. Because of the symmetry of $\mathbf{A}$ the computational complexity to compute the Cholesky factorization is halved, meaning it costs $\frac{1}{3}n^3 + \mathcal{O}(n^2)$.

Example 4.6. The matrix

$$\mathbf{A} = \begin{pmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{pmatrix}.$$

is clearly symmetric and satisfies the sufficient condition for positive definiteness. Therefore, applying the $\mathbf{LU}$ factorization technique to this matrix yields

$$\begin{pmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{3}{5} & 1 & 0 \\ -\frac{1}{5} & \frac{1}{3} & 1 \end{pmatrix} \begin{pmatrix} 25 & 15 & -5 \\ 0 & 9 & 3 \\ 0 & 0 & 9 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{pmatrix} = \begin{pmatrix} 5 & 0 & 0 \\ 3 & 3 & 0 \\ -1 & 1 & 3 \end{pmatrix} \begin{pmatrix} 5 & 3 & -1 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{pmatrix},$$

which is the Cholesky decomposition $\mathbf{A} = \mathbf{C}\mathbf{C}^T$ because $\mathbf{A}$ is a s.p.d. matrix. ⧫

4.2 Stationary iterative methods

The **LU** factorization is a powerful, general tool to solve linear systems. However, sometimes the matrix is too large to be stored in the computer memory making such a direct method difficult to use. Even more importantly, the computational cost of $\frac{2}{3}n^3$ for Gauss-Jordan elimination to create the **L** and **U** matrices is too expensive for such large systems. Instead, we *return* to iterative solution techniques that can reduce the computational cost to $\mathcal{O}(n^2)$. This is done by creating a solution algorithm that only requires matrix-vector multiplication (or something equivalently expensive) as well as vector addition/subtraction. Therefore, iterative methods belong to the BLAS2 part of the hierarchy whereas factorization methods, like **LU**, belong in the BLAS3 category.

Recall that iterative methods to solve a problem repeat a task until the solution is deemed "good enough" in some error tolerance. So, we must again discuss the questions:

1. How do we know when an iterative technique will *converge*?
2. What is a good *initial guess* to kick start the iteration?
3. What is an appropriate condition to *stop* the iteration?

A strategy we explored previously explored for nonlinear equations of a single variable to create an iterative solution process was the fixed point iteration. An analogue of this idea to linear systems is the creation of a *stationary iterative method.* Again, we want to solve the problem $\mathbf{Ax} = \mathbf{b}$ and assume that the matrix $\mathbf{A}$ is nonsingular.

Definition 4.5 (Stationary Iterative Method). Let $\mathbf{T} \in \mathbb{R}^{n\times n}, \mathbf{c} \in \mathbb{R}^n$, and an initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$ all be given. Then the operator $\mathbf{T}$ and vector $\mathbf{c}$ define a linear stationary method

$$\mathbf{x}^{(k+1)} = \mathbf{T}\mathbf{x}^{(k)} + \mathbf{c},$$

where $\mathbf{x}^{(0)}$ is the initial condition that defines a particular sequence of solutions. ◀

To define the pieces of the stationary iterative technique above we split the nonsingular matrix $\mathbf{A}$ into two matrices

$$\mathbf{A} = \mathbf{M} - \mathbf{N},$$

where $\mathbf{M}$ is also nonsingular (i.e. invertible) and $\mathbf{N}$ is an $n \times n$ matrix. Then our problem of interest is rewritten to become

$$\mathbf{Ax} = \mathbf{b} \quad \Rightarrow \quad \mathbf{Mx} = \mathbf{Nx} + \mathbf{b} \quad \Rightarrow \quad \mathbf{x} = \mathbf{M}^{-1}\mathbf{Nx} + \mathbf{M}^{-1}\mathbf{b}.$$

Thus, the *iteration matrix* for this splitting approach is $\mathbf{T} = \mathbf{M}^{-1}\mathbf{N}$ and $\mathbf{c} = \mathbf{M}^{-1}\mathbf{b}$.

Example 4.7. Show that the solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ is a fixed point of the stationary iterative method defined by the matrix splitting $\mathbf{A} = \mathbf{M} - \mathbf{N}$. We examine the right hand side of the equation to find

$$\begin{aligned}\mathbf{Tx} + \mathbf{c} = \mathbf{M}^{-1}\mathbf{Nx} + \mathbf{M}^{-1}\mathbf{b} &= \mathbf{M}^{-1}\mathbf{N}\mathbf{A}^{-1}\mathbf{b} + \mathbf{M}^{-1}\mathbf{b}\\ &= \mathbf{M}^{-1}(\mathbf{M} - \mathbf{A})\mathbf{A}^{-1}\mathbf{b} + \mathbf{M}^{-1}\mathbf{b}\\ &= \mathbf{A}^{-1}\mathbf{b} - \mathbf{M}^{-1}\mathbf{b} + \mathbf{M}^{-1}\mathbf{b}\\ &= \mathbf{A}^{-1}\mathbf{b}\\ &= \mathbf{x}.\end{aligned}$$

⧫

Remark 4.8. An equivalent way to write the stationary iterative method is in terms of the *residual* in the k^{th} iteration, $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$, where

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{M}^{-1}\mathbf{r}^{(k)}.$$

This is often referred to as the *preconditioner* form. Heuristically, this is because we select the matrix $\mathbf{M}$ such that $\mathbf{M}^{-1} \approx \mathbf{A}^{-1}$ albeit significantly cheaper to invert. ⋈

It is important to note that the components of the iterative technique, $\mathbf{T}$ and $\mathbf{c}$, ***do not*** depend on the iteration! This is a hallmark of stationary iterative techniques. Broadly, these methods (either written in the form from Definition 4.5 or in residual form) belong to the family of *Richardson* stationary methods.

Now that we have defined a fixed point iteration for the solution of a linear system, when will it work? To find out, we develop a condition to determine if the method will converge. As such, we require an equation for the error in a given iteration k, where

$$\mathbf{x}^{(k)} = \mathbf{T}\mathbf{x}^{(k-1)} + \mathbf{c}.$$

If we subtract the exact solution $\mathbf{x}$, which also satisfies the fixed point equation, we have

$$\boldsymbol{\epsilon}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x} = \mathbf{T}(\mathbf{x}^{(k-1)} - \mathbf{x}) = \mathbf{T}\boldsymbol{\epsilon}^{(k-1)}$$

where the vector $\mathbf{c}$ cancels due to its independence from the iteration. Next, we use a suitable vector norm to measure the magnitude of each side of the equation. Further, we apply the fact that the error is recursive to see

$$\|\boldsymbol{\epsilon}^{(k)}\| = \|\mathbf{T}\boldsymbol{\epsilon}^{(k-1)}\| = \|\mathbf{T}^2\boldsymbol{\epsilon}^{(k-2)}\| = \cdots = \|\mathbf{T}^k\boldsymbol{\epsilon}^{(0)}\|$$

such that the error of the k^{th} iteration depends on a power of the iteration matrix $\mathbf{T}$ and the initial error $\boldsymbol{\epsilon}^{(0)} = \mathbf{x}^{(0)} - \mathbf{x}$. Now, we apply the subordinate and submultiplicative properties of the induced matrix norm to find

$$\|\boldsymbol{\epsilon}^{(k)}\| = \|\mathbf{T}^k\boldsymbol{\epsilon}^{(0)}\| \leq \|\mathbf{T}^k\|\|\boldsymbol{\epsilon}^{(0)}\| \leq \|\mathbf{T}\|^k\|\boldsymbol{\epsilon}^{(0)}\|,$$

where the error in the stationary iteration goes to zero provided $\|\mathbf{T}\|^k \to 0$ as $k \to \infty$.

Definition 4.6 (Sufficient condition for convergence). If the norm of the iteration matrix $\mathbf{T}$ is less than unity, $\|\mathbf{T}\| < 1$, for a given nonsingular matrix $\mathbf{A}$, then the stationary iterative method converges. ◀

Remark 4.9. Note that this condition on the "magnitude" of the iteration matrix $\mathbf{T}$ is *reminiscent* of the definition for a contraction mapping in a fixed point iterative method from Lecture 2. ⋈

Remark 4.10. Keep in mind that the sufficient condition $\|\mathbf{T}\| < 1$ is rather vague because it does not specify the matrix norm we should use. However, as long as we satisfy this inequality in *some* matrix norm we know that the stationary iteration will converge, although this convergence can be quite slow. ⋈

Remark 4.11. In this course, we will only use this criterion to discuss convergence of a stationary iterative method. However, more rigorous conditions on the iteration matrix exist that are *necessary and sufficient* which rely on examining the eigenvalues of the matrix $\mathbf{T}$. ⋈

Next, we examine the computational cost of the stationary iterative method. For this, the matrix splitting construct is particularly useful where we keep in mind that the matrix $\mathbf{M}$ is nonsingular. Note, that the matrix form is a useful theoretical tool to estimate computational cost or discuss convergence but, as we will see in later examples, this is not how one implements this type of method in practice. The general algorithm for a single iteration of the stationary method built from a matrix splitting approach can be divided into four steps:

1. Solve $\mathbf{M}\mathbf{c} = \mathbf{b}$.
2. Compute the vector $\mathbf{v} = \mathbf{N}\mathbf{x}^{(k)}$.
3. Solve $\mathbf{M}\mathbf{z}^{(k)} = \mathbf{v}$.
4. Update the solution $\mathbf{x}^{(k+1)} = \mathbf{z}^{(k)} + \mathbf{c}$.

We see that the first step of the algorithm actually only has to be done *once* because it does not depend on the iteration. In contrast, the matrix solution in step three must be done in *every* iteration, which reinforces that the inversion of the matrix $\mathbf{M}$ should be *cheap.*

With this theoretical background we can now discuss two classical stationary iterative methods for the solution of linear systems. For this we consider a general, nonsingular matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}.$$

Further, we divide $\mathbf{A}$ into a matrix that contains only diagonal elements $\mathbf{D} = \operatorname{diag}(a_{11}, \ldots, a_{nn})$, a strictly lower triangular matrix $\mathbf{R}_L$, and a strictly upper triangular matrix $\mathbf{R}_U$

$$\mathbf{R}_L = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ -a_{21} & 0 & 0 & \cdots & 0 \\ -a_{31} & -a_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ -a_{n1} & -a_{n2} & \cdots & -a_{n,n-1} & 0 \end{pmatrix}, \quad \mathbf{R}_U = \begin{pmatrix} 0 & -a_{12} & -a_{13} & \cdots & -a_{1n} \\ 0 & 0 & -a_{23} & \cdots & -a_{2n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & -a_{n-1,n} \\ 0 & 0 & \cdots & 0 & 0 \end{pmatrix},$$

such that

$$\mathbf{A} = \mathbf{D} - \mathbf{R}_L - \mathbf{R}_U.$$

The signs here are useful to relate this division of the matrix $\mathbf{A}$ to the splitting form of the stationary iterative method.

Definition 4.7 (Jacobi iterative method). If we take the splitting matrices to be

$$\mathbf{M} = \mathbf{D} \quad \text{and} \quad \mathbf{N} = \mathbf{R}_L + \mathbf{R}_U,$$

we obtain iterative strategy of Jacobi

$$\mathbf{x}^{(k+1)} = \mathbf{T}\mathbf{x}^{(k)} + \mathbf{c} = \mathbf{D}^{-1}(\mathbf{R}_L + \mathbf{R}_U)\mathbf{x}^{(k)} + \mathbf{D}^{-1}\mathbf{b}.$$

◀

Example 4.8. Consider the three matrices

$$\mathbf{A}_1 = \begin{pmatrix} 9 & 1 & 1 \\ 2 & 10 & 3 \\ 3 & 4 & 11 \end{pmatrix}, \quad \mathbf{A}_2 = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix}, \quad \mathbf{A}_3 = \begin{pmatrix} 4 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 4 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 4 \end{pmatrix},$$

and determine if the Jacobi iteration will converge. The associated iteration matrices are

$$\mathbf{T}_1 = \begin{pmatrix} 0 & -\frac{1}{9} & -\frac{1}{9} \\ -\frac{2}{10} & 0 & -\frac{3}{10} \\ -\frac{3}{11} & -\frac{4}{11} & 0 \end{pmatrix}, \quad \mathbf{T}_2 = \frac{1}{2}\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \quad \mathbf{T}_3 = \frac{1}{4}\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

We can compute that $\|\mathbf{T}_1\|_\infty = \frac{7}{11}$, $\|\mathbf{T}_2\|_1 = \|\mathbf{T}_2\|_\infty = 1$, and $\|\mathbf{T}_3\|_1 = \|\mathbf{T}_3\|_\infty = \frac{1}{2}$. So, we immediately know that Jacobi's iteration will converge for $\mathbf{A}_1$ and $\mathbf{A}_3$, but it is not clear if it will work for $\mathbf{A}_2$. However, if we compute the 2-norm of this matrix we find $\|\mathbf{T}_2\|_2 = 0.923879532511287$ and, therefore, Jacobi will converge. This reinforces that the choice of norm for our sufficient convergence condition is important. ⧫

From the perspective of computational cost, we see that the preconditioning matrix $\mathbf{M}$ for Jacobi's iteration is diagonal and, therefore, trivial to invert requiring n divisions. In the general algorithmic structure above we see that the second step, which is a matrix-vector product, is the most expensive part for a given iteration. Therefore, each step of the Jacobi iteration has an estimated cost of $2n^2 + \mathcal{O}(n)$.

Such an estimate of the work is nice, but in practice this matrix form *is not* an efficient way to implement the Jacobi iteration. For one thing, storing a diagonal matrix $\mathbf{D}$ is a waste of memory because of the large number of zero entries. Instead, it is useful to look at the index form of the Jacobi iterative method

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1,i\neq j}^{n} a_{ij}x_j^{(k)}\right), \quad i = 1,\dots,n.$$

We see that the new approximation to the solution $\mathbf{x}^{(k+1)}$ is constructed from a linear combination of the right hand side information $\mathbf{b}$ and the *old* approximation $\mathbf{x}^{(k)}$. We provide pseudocode for the Jacobi iterative technique (again we set a maximum number of iterations to ensure that we do not create an infinite loop):

Algorithm 4.2: *Jacobi iteration*: Solve a nonsingular linear system with Jacobi's method

```
Procedure Jacobi Iteration
Input: $\mathbf{A}$, $\mathbf{b}$, $\mathbf{x}^{(0)}$, $n$, tol
$max_{its} \leftarrow 250$ // maximum number of iterations allowed
for $k = 1$ to $max_{its}$ do
    for $i = 1$ to $n$ do
        sum $\leftarrow 0$
        for $j = 1$ to $n$ do
            if $j \neq i$ then
                sum $\leftarrow$ sum + $a_{ij}x_j^{(0)}$
        $x_i^{(1)} \leftarrow (b_i -$ sum$)/a_{ii}$
    $\mathbf{r}^{(1)} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}^{(1)}$
    if $\|\mathbf{r}^{(1)}\|_2 <$ tol $\cdot \|\mathbf{b}\|_2$ then
        exit
    $\mathbf{x}^{(0)} \leftarrow \mathbf{x}^{(1)}$
Output: $\mathbf{x}^{(1)}$ // solution vector
End Procedure Jacobi Iteration
```

Just as we saw with fixed point iterations we require some kind of *kill condition* to stop the iteration once the solution is deemed "good enough".

Definition 4.8 (Kill condition for iterative method). There are many ways to define a condition that terminates the iteration. Two such choices, where we select the vector 2-norm, are

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_2 < \texttt{tol} \qquad \text{or} \qquad \frac{\|\mathbf{r}^{(k+1)}\|_2}{\|\mathbf{b}\|_2} < \texttt{tol}.$$

The second stopping condition is the one given in the pseudocode above. ◀

The first stopping criterion compares the difference between components of the last two iterates k and $k+1$. Thus, it is possible that this kill condition could halt the iteration before an accurate solution is reached because it *does not* check the residual. In contrast, the first stopping condition is cheaper to evaluate because it does not require a matrix-vector product. However, the second kill condition is more robust in providing an accurate iterative solution to the problem.

Recall from Remark 4.8 there is an equivalent residual form of the stationary iterative method. This is convenient because we can ***reuse*** the residual computed for the stopping criterion to update the solution in the next iteration! We provide psudocode for an alternative, residual form of the Jacobi iteration where the preconditioner matrix is $\mathbf{M} = \mathbf{D}$.

Algorithm 4.3: *Jacobi iteration (residual form)*: Equivalent form of Jacobi's method

```
Procedure Jacobi Iteration (Residual Form)
Input: $\mathbf{A}$, $\mathbf{b}$, $\mathbf{x}^{(0)}$, $n$, tol
$max_{its} \leftarrow 250$ // maximum number of iterations allowed
$\mathbf{r}^{(0)} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for $k = 1$ to $max_{its}$ do
    for $i = 1$ to $n$ do
        $x_i^{(1)} \leftarrow x_i^{(0)} + r_i^{(0)}/a_{ii}$
    $\mathbf{r}^{(1)} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}^{(1)}$
    if $\|\mathbf{r}^{(1)}\|_2 < \mathtt{tol} \cdot \|\mathbf{b}\|_2$ then
        exit
    $\mathbf{x}^{(0)} \leftarrow \mathbf{x}^{(1)}$
    $\mathbf{r}^{(0)} \leftarrow \mathbf{r}^{(1)}$
Output: $\mathbf{x}^{(1)}$ //solution vector
End Procedure Jacobi Iteration (Residual Form)
```

We see that in the Jacobi iteration we do not use the most recently available information when computing $x_i^{(k+1)}$. For example, $x_1^{(k)}$ is used in the computation of $x_2^{(k+1)}$ despite the fact that the component $x_1^{(k+1)}$ is already known. If we assume that iterative method is convergent, it would make sense to use the most up-to-date information to compute the solution within an iteration. This is the basic idea of the *Gauss-Seidel iterative method.*

Definition 4.9 (Gauss-Seidel method). If we take the splitting matrices to be

$$\mathbf{M} = \mathbf{D} - \mathbf{R}_L \quad \text{and} \quad \mathbf{N} = \mathbf{R}_U,$$

we obtain the Gauss-Seidel iterative strategy

$$\mathbf{x}^{(k+1)} = \mathbf{T}\mathbf{x}^{(k)} + \mathbf{c} = (\mathbf{D} - \mathbf{R}_L)^{-1}\mathbf{R}_U\mathbf{x}^{(k)} + (\mathbf{D} - \mathbf{R}_L)^{-1}\mathbf{b}.$$

◀

For the Gauss-Seidel method we see that the iteration matrix is $\mathbf{T} = (\mathbf{D} - \mathbf{R}_L)^{-1}\mathbf{R}_U$, so constructing $\mathbf{T}$ and determining the matrix norm to check the sufficient condition for convergence is unwieldy. Instead we define a stricter, but easier to check, condition:

Definition 4.10 (Diagonal dominance). Examining the rows of a matrix $\mathbf{A}$ where $1 \le i \le n$

$$|a_{ii}| \ge \sum_{j=1, j\neq i}^{n} |a_{ij}|,$$

then the matrix is said to be *weakly diagonally dominant.* If the equality is removed then the matrix is *strictly diagonally dominant.* ◀

Remark 4.12. If the matrix $\mathbf{A}$ is strictly diagonally dominant, then $\mathbf{A}$ is nonsingular and the associated Jacobi and Gauss-Seidel iterations converge for *any* initial guess $\mathbf{x}^{(0)}$. ⋈

Remark 4.13. For strictly diagonally dominant matrices finding the inverse of the preconditioner matrix $\mathbf{M} = \mathbf{D}$ for Jacobi or $\mathbf{M} = \mathbf{D} - \mathbf{R}_L$ for Gauss-Seidel is a fairly good, cheaper to invert approximation of the true matrix inverse $\mathbf{A}^{-1}$. ⋈

Example 4.9. From this knowledge we immediately know that Gauss-Seidel will converge for the matrices $\mathbf{A}_1$ and $\mathbf{A}_3$ from the Example 4.8 because they are strictly diagonally dominant. The matrix $\mathbf{A}_2$ is only weakly diagonally dominant, but it is symmetric positive definite. Thus, Gauss-Seidel is still guaranteed to converge for any initial guess. ⧫

What about the computational cost of the Gauss-Seidel method? From the general algorithm above we see it still involves a matrix-vector multiplication in the second step. But the third step now requires the solution of a matrix problem. However, this can be done efficiently through a forward solve which is known to cost $n^2 + \mathcal{O}(n)$. Therefore, the total estimated cost is $3n^2 + \mathcal{O}(n)$. So, on paper, we see that the Gauss-Seidel iterative method is more expensive than Jacobi's method. However, *if both methods converge*, the Gauss-Seidel method will converge ***faster*** because it uses the most up-to-date information in each iteration.

Just as before, the matrix form is useful to examine the Gauss-Seidel method theoretically but has limited use in practice. We exploit the forward substitution nature of inverting the matrix $\mathbf{M}$ and write the index form of the method

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \quad i = 1, \dots, n.$$

This reformulation and exploitation of the forward solution algorithm puts the Gauss-Seidel method into a computational form that has a very similar cost (and structure) to the Jacobi iteration, but keep in mind Gauss-Seidel will be faster. We provide pseudocode for the Gauss-Seidel iterative technique below where, to save memory consumption, the updated solution continually overwrites the previous information:

Algorithm 4.4: *Gauss-Seidel iteration*: Solve a nonsingular linear system with Gauss-Seidel's method

```
Procedure Gauss-Seidel Iteration
Input: $\mathbf{A}$, $\mathbf{b}$, $\mathbf{x}$, $n$, tol // x contains the initial guess
$max_{its} \leftarrow 250$ // maximum number of iterations allowed
for $k = 1$ to $max_{its}$ do
    for $i = 1$ to $n$ do
        sum $\leftarrow 0$
        for $j = 1$ to $n$ do
            if $j \neq i$ then
                sum $\leftarrow$ sum + $a_{ij}x_j$
        $x_i \leftarrow (b_i - \text{sum})/a_{ii}$ // update the information in the solution vector
    $\mathbf{r} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}$
    if $\|\mathbf{r}\|_2 <$ tol $\cdot \|\mathbf{b}\|_2$ then
        exit
Output: x //solution vector
End Procedure Gauss-Seidel Iteration
```

Remark 4.14. If the matrix $\mathbf{A}$ is symmetric positive definite, then the Gauss-Seidel iterative method is guaranteed to converge for any initial guess $\mathbf{x}^{(0)}$. ⋈

Example 4.10. We apply the Jacobi and Gauss-Seidel method to the three matrices given in Example 4.8. For the stopping criterion we use the residual check given in the pseudocode for Algorithms 4.2 and 4.4 with a tolerance `tol` $= 10^{-6}$. We ***manufacture*** the true solution to be $\mathbf{x}_{exact}$ to be a vector of all ones, where the right hand side $\mathbf{b}$ is set accordingly. We take the initial guess for the solution to be a vector of zeros, e.g. $\mathbf{x}^{(0)} = \mathbf{0}$. In the following table we give the number of iterations required for the Jacobi and Gauss-Seidel iterative methods to reach the set tolerance for each matrix:

Matrix	Dimension	Iterations k (Jacobi)	Iterations k (Gauss-Seidel)
$\mathbf{A}_1$	3×3	18	7
$\mathbf{A}_2$	7×7	163	81
$\mathbf{A}_3$	7×7	18	11

For these small "toy" examples we see that the number of iterations k is larger than the dimension of the system n. We can expect this because the motivation of iterative techniques was for very large linear system (i.e. n is large). In this case the iterative methods that cost $\mathcal{O}(n^2)$ scale better than direct methods that cost $\mathcal{O}(n^3)$. ⧫

Can these methods be improved and convergence toward the solution accelerated? Yes, there are many strategies to do this. In the realm of stationary iterative methods there are so-called *relaxation methods*. However, there are also more sophisticated *nonstationary* iterative methods. The main difference between the method types is that the iteration involves information that changes with each iteration. To develop such nonstationary techniques we view the solution of a linear system through a different lens and motivate a different iterative approach to solve the problem.