Chapter 2

Solutions of systems of Linear Equations

There are many problems in science and engineering that requires the solution of a linear problem of the kind

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{2.1}$$

where $\mathbf{A}$ is a square $m \times m$ matrix, and $\mathbf{x}$ and $\mathbf{b}$ are both m-long column vectors. In what follows, we will assume that the problem is non-singular, that is, that we have somehow already demonstrated that $\det(\mathbf{A}) \neq 0$.

1. Examples of applications that require the solution of $\mathbf{Ax} = \mathbf{b}$

1.1. Fitting a hyperplane to m points in $\mathbb{R}^m$

Given 2 distinct points on a plane ($\mathbb{R}^2$), there is only 1 line that passes through both points. Similarly, given 3 distinct *non-aligned* points in 3D space ($\mathbb{R}^3$), there is only 1 plane that goes through all three points. Given 4 distinct *non-coplanar* points in 4D space ($\mathbb{R}^4$), there is only 1 hyperplane that goes through all four points – and so forth! Finding the equation of this hyperplane requires solving a linear system of the kind $\mathbf{Ax} = \mathbf{b}$. Indeed, the equation for an $m-$dimensional hyperplane is the linear equation

$$a_1 x_1 + a_2 x_2 + a_3 x_3 + \cdots + a_m x_m = \text{const} \tag{2.2}$$

where $\mathbf{x} = (x_1, \ldots, x_m)^T$ are the coordinates of a point in the hyperplane, the coefficients $\{a_i\}_{i=0,\ldots,m}$ are real numbers, and the constant const is arbitrary and can be chosen to be equal to one without loss of generality. Given m points in the hyperplane denoted as $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, $\ldots$, $\mathbf{x}^{(m)}$, we therefore have the system of equations

$$\begin{aligned}
&a_1 x_1^{(1)} + a_2 x_2^{(1)} + \cdots + a_m x_m^{(1)} = 1 \\
&a_1 x_1^{(2)} + a_2 x_2^{(2)} + \cdots + a_m x_m^{(2)} = 1 \\
&\vdots \\
&a_1 x_1^{(m)} + a_2 x_2^{(m)} + \cdots + a_m x_m^{(m)} = 1
\end{aligned} \tag{2.3}$$

which can be re-cast as $\mathbf{Ax} = \mathbf{b}$ with

$$\mathbf{A} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_m^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_m^{(2)} \\ & & \vdots & \\ x_1^{(m)} & x_2^{(m)} & \dots & x_m^{(m)} \end{pmatrix}, \mathbf{x} = \begin{pmatrix} a_1 \\ a_2 \\ \dots \\ a_m \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ \dots \\ 1 \end{pmatrix} \tag{2.4}$$

The solution of this equation is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ provided $\mathbf{A}$ is nonsingular.

1.2. Solving a partial differential equation

Suppose we want to solve the PDE

$$\frac{\partial f}{\partial t} = \frac{\partial^2 f}{\partial x^2} \tag{2.5}$$

with some given initial condition $f(x, 0) = f_0(x)$. As you shall see at length in AMS 213B, a simple way of solving the problem consists in discretizing this equation both in space and time: letting space be discretized as a series of closely spaced $\{x_i\}$ so $x_{i+1} - x_i = \Delta x$, and similarly time be discretized as $\{t^{(n)}\}$ so $t^{(n+1)} - t^{(n)} = \Delta t$, we rewrite the PDE as the discrete equation

$$\frac{f_i^{(n+1)} - f_i^{(n)}}{\Delta t} = \frac{f_{i+1}^{(n)} - 2f_i^{(n)} + f_{i-1}^{(n)}}{\Delta x^2} \tag{2.6}$$

where $f_i^{(n)} \equiv f(x_i, t^{(n)})$. This is called an explicit scheme, because we can then simply write the solution at the next timestep *explicitly* as a function of the solution at the previous timestep.

$$f_i^{(n+1)} = f_i^{(n)} + \frac{\Delta t}{\Delta x^2}\left(f_{i+1}^{(n)} - 2f_i^{(n)} + f_{i-1}^{(n)}\right) \tag{2.7}$$

This can actually be cast more simply as the matrix equation $\mathbf{f}^{(n+1)} = \mathbf{C}\mathbf{f}^{(n)}$ where $\mathbf{C} = \mathbf{I} + \frac{\Delta t}{\Delta x^2}\mathbf{M}$ and

$$\mathbf{M} = \begin{pmatrix} \ddots & \ddots & \ddots & & & \\ & 1 & -2 & 1 & & \\ & & 1 & -2 & 1 & \\ & & & \ddots & \ddots & \ddots \end{pmatrix} \text{ and } \mathbf{f}^{(n)} = \begin{pmatrix} \vdots \\ f_{i-1}^{(n)} \\ f_i^{(n)} \\ f_{i+1}^{(n)} \\ \vdots \end{pmatrix} \tag{2.8}$$

In this algorithm, $\mathbf{f}^{(n+1)}$ can be obtained simply by matrix multiplication. However, this explicit algorithm is subject to satisfy a strict stability constraint (see AMS213B for more information), and better results can be obtained, both in terms of stability and accuracy, using a Crank-Nicholson scheme, in which

$$\frac{f_i^{(n+1)} - f_i^{(n)}}{\Delta t} = \frac{1}{2}\frac{f_{i+1}^{(n)} - 2f_i^{(n)} + f_{i-1}^{(n)}}{\Delta x^2} + \frac{1}{2}\frac{f_{i+1}^{(n+1)} - 2f_i^{(n+1)} + f_{i-1}^{(n+1)}}{\Delta x^2} \tag{2.9}$$

that is, by evaluating the time derivative half way between the timesteps $t^{(n)}$ and $t^{(n+1)}$. This algorithm, by contrast with the previous one, becomes $\mathbf{A}\mathbf{f}^{(n+1)} = \mathbf{B}\mathbf{f}^{(n)}$ where $\mathbf{A} = \mathbf{I} - \frac{\Delta t}{2\Delta x^2}\mathbf{M}$ and $\mathbf{B} = \mathbf{I} + \frac{\Delta t}{2\Delta x^2}\mathbf{M}$. In order to advance the solution in time, we therefore have to solve a matrix problem for $\mathbf{f}^{(n+1)}$. In principle, this can be done by finding the inverse of $\mathbf{A}$, and evaluating $\mathbf{f}^{(n+1)} = \mathbf{A}^{-1}\mathbf{B}\mathbf{f}^{(n)}$ at each timestep. While both examples require solving a matrix problem, their practical use in large-scale computations (e.g. fitting *many* hyperplanes different sets of points, or advancing the PDE in time for *many* timesteps) is quite different. In the first case, each set of points gives rise to a different matrix $\mathbf{A}$, so the problem needs to be solved from scratch every time. In the second case on the other hand, assuming that the timestep Δt and the grid spacing Δx remain constant, the matrix $\mathbf{A}$ remains the same each time, so it is worth solving for $\mathbf{A}^{-1}$ just once ahead of time, save it, and then simply multiply each new right-hand-side by $\mathbf{A}^{-1}$ to evolve the solution forward in time. Or something similar, at the very least (as we shall see, things are done a little different in practice).

2. A little aside on invariant transformations

To solve a linear system, we usually transform it step by step into one that is much easier to solve, making sure in the process that the solution remains unchanged. In practice, this can easily be done in linear algebra noting that multiplication of the equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ by any nonsingular matrix $\mathbf{M}$ on both sides, namely:

$$\mathbf{M}\mathbf{A}\mathbf{x} = \mathbf{M}\mathbf{b} \tag{2.10}$$

leaves the solution unchanged. Indeed, let $\mathbf{z}$ be the solution of Eqn. 2.10. Then

$$\mathbf{z} = (\mathbf{M}\mathbf{A})^{-1}\mathbf{M}\mathbf{b} = \mathbf{A}^{-1}\mathbf{M}^{-1}\mathbf{M}\mathbf{b} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{x}. \tag{2.11}$$

These transformations, i.e., multiplication by a non-singular matrix, are called **invariant transformations**.

2.1. Permutations

A permutation matrix $\mathbf{P}$, a square matrix having exactly one 1 in each row and column and zeros elsewhere – which is also always a nonsingular – can always be multiplied without affecting the original solution to the system. For instance,

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.12}$$

permutes $\mathbf{v}$ as

$$\mathbf{P}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} v_3 \\ v_1 \\ v_2 \end{bmatrix}. \tag{2.13}$$

□

The same permutation matrix applied to a 3×3 matrix $\mathbf{A}$ would shuffle its rows in the same way. More generally, permutation matrices are operations that shuffle the rows of a matrix.

2.2. Row scaling

Another invariant transformation exists which is called *row scaling*, an outcome of a multiplication by a diagonal matrix $\mathbf{D}$ with nonzero diagonal entries $d_{ii}, i = 1, \ldots m$. In this case, we have

$$\mathbf{DAx} = \mathbf{Db}, \tag{2.14}$$

by which each row of the transformed matrix $\mathbf{DA}$ gets to be scaled by d_{ii} from the original matrix $\mathbf{A}$. Note that the scaling factors are cancelled by the same scaling factors introduced on the right hand side vector, leaving the solution to the original system unchanged.

Note: The column scaling does not preserve the solution in general. □

3. Gaussian elimination

Chapter 2.2 of Numerical Recipes, and Chapter 20 of the textbook

Recall that in this chapter we are interested in solving a well-defined linear system given as

$$\mathbf{Ax} = \mathbf{b}, \tag{2.15}$$

where $\mathbf{A}$ is a $m \times m$ *square* matrix and $\mathbf{x}$ and $\mathbf{b}$ are m-vectors. The most standard algorithm for the solution of linear systems learned in introductory linear algebra classes is Gaussian elimination. Gaussian elimination proceeds in steps sweeping the matrix from left to right, and successively zeroing out (using clever linear operations on the rows), all the coefficients below the diagonal. The same operations are carried out on the right-hand-side, to ensure invariance of the solution.

$$\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{bmatrix} \tag{2.16}$$

The end product is an upper triangular matrix, and the associated set of linear equations can then be solved by back-substitution. Here is a sample algorithm describing the steps required for Gaussian elimination.

Algorithm: Basic Gaussian elimination:

```
do j = 1 to m − 1
    ![loop over column]
    if a_jj = 0 then
        stop
        ![stop if pivot (or divisor) is zero]
    endif
    do i = j + 1 to m
    ![sweep over rows r_i below row r_j]
        r_i = r_i − r_j a_ij / a_jj
        ![zeros terms below a_jj and transforms rest of matrix]

        b_i = b_i − b_j a_ij / a_jj
        ![carries over same operation on RHS]
    enddo
enddo
```

Note that the operation $\mathbf{r}_i = \mathbf{r}_i - \mathbf{r}_j a_{ij}/a_{jj}$ is another loop over all terms in row $\mathbf{r}_i$. It guarantees to zero out all of the elements in the column j below the diagonal, and since it only operates on rows below the diagonal, it does not affect any of the terms that have already been zeroed out.

This algorithm is however problematic for a few reasons. The first is that it stops if the diagonal term a_{jj} is zero, which may well happen even if the matrix is non-singular. The second problem is that even for non-singular, well-conditioned matrices, this algorithm is not very stable. Both problems are discussed in the next section. Finally it is also quite wasteful since it spends time calculating entries that we already know are zero. More on this later.

It is worth noting that the operations described in the previous algorithm can be written formally in terms of invariant transformations. Indeed, suppose we define the matrix $\mathbf{M}_1$ so that its action on $\mathbf{A}$ is to zero out the elements in the first column below the first row, and apply it to both the left-hand-side and right-hand-sides of $\mathbf{Ax} = \mathbf{b}$. Again, we repeat this process in the next step so that we find $\mathbf{M}_2$ such that the second column of $\mathbf{M}_2\mathbf{M}_1\mathbf{A}$ becomes zero below the second row, along with applying the equivalent multiplication on the right hand side, $\mathbf{M}_2\mathbf{M}_1\mathbf{b}$. This process is continued for each successive column until all of the subdiagonal entries of the resulting matrix have been annihilated.

If we define the final matrix $\mathbf{M} = \mathbf{M}_{n-1}\cdots\mathbf{M}_1$, the transformed linear system becomes

$$\mathbf{M}_{n-1}\cdots\mathbf{M}_1\mathbf{Ax} = \mathbf{MAx} = \mathbf{Mb} = \mathbf{M}_{n-1}\cdots\mathbf{M}_1\mathbf{b}. \tag{2.17}$$

To show that this is indeed an invariant transformation, we simply have to show that $\mathbf{M}$ is non-singular. The easiest (non-rigorous) way of showing this is to look at the structure of $\mathbf{M}$, through an example.

Example: Consider

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 10 \\ 1 \end{bmatrix} = \mathbf{b}. \tag{2.18}$$

The first question is to find a matrix $\mathbf{M}_1$ that annihilates the subdiagonal entries of the first column of $\mathbf{A}$. This can be done if we consider a matrix $\mathbf{M}_1$ that can subtract twice the first row from the second row, four times the first row from the third row, and three times the first row from the fourth row. The matrix $\mathbf{M}_1$ is then identical to the identity matrix $\mathbf{I}_4$, except for those multiplication factors in the first column:

$$\mathbf{M}_1\mathbf{A} = \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix}, \tag{2.19}$$

where we treat the blank entries to be zero entries.

The next step is to annihilate the third and fourth entries from the second column (3 and 4), which gives the next matrix $\mathbf{M}_2$ that has the form:

$$\mathbf{M}_2\mathbf{M}_1\mathbf{A} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -3 & 1 & \\ & -4 & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix}, \tag{2.20}$$

The last matrix $\mathbf{M}_3$ completes the process, resulting an upper triangular matrix $\mathbf{U}$:

$$\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1\mathbf{A} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix} = \mathbf{U}, \tag{2.21}$$

together with the right hand side:

$$\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1\mathbf{b} = \begin{bmatrix} 3 \\ 0 \\ -2 \\ -6 \end{bmatrix} = \mathbf{y}. \tag{2.22}$$

We see that the matrix $\mathbf{M}$ formed in the process is the product of three lower-triangular matrices, which is itself lower triangular (you can easily check this out!). Since lower triangular matrices are singular if and only if one of their diagonal entries is zero (which is not the case since they are all 1), $\mathbf{M}$ is non-singular. $\square$

Example: In the previous example, we see that the final transformed linear system $\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1\mathbf{A} = \mathbf{MA}$ yields $\mathbf{MAx} = \mathbf{Mb}$ which is equivalent to $\mathbf{Ux} = \mathbf{y}$, an upper triangular system which we wanted and can be solved easily by back-substitution, starting from obtaining $x_4 = -3$, followed by x_3, x_2, and x_1 in reverse order to find a complete solution

$$\mathbf{x} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ -3 \end{bmatrix}. \tag{2.23}$$

Furthermore, the full LU factorization of $\mathbf{A}$ can be established as $\mathbf{A} = \mathbf{LU}$ if we compute

$$\mathbf{L} = (\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-1} = \mathbf{M}_1^{-1}\mathbf{M}_2^{-1}\mathbf{M}_3^{-1}. \tag{2.24}$$

At first sight this looks like an expensive process as it involves inverting a series of matrices. Surprisingly, however, this turns out to be a trivial task. The inverse of $\mathbf{M}_i$, $i = 1, 2, 3$ is just itself but with each entry below the diagonal negated. Therefore, we have

$$\begin{aligned}
\mathbf{L} &= \mathbf{M}_1^{-1}\mathbf{M}_2^{-1}\mathbf{M}_3^{-1} \\
&= \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -3 & 1 & \\ & -4 & & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -1 & 1 \end{bmatrix}^{-1} \\
&= \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & & 1 & \\ 3 & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & 3 & 1 & \\ & 4 & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & 1 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix}.
\end{aligned} \tag{2.25}$$

Notice also that the matrix multiplication $\mathbf{M}_1^{-1}\mathbf{M}_2^{-1}\mathbf{M}_3^{-1}$ is also trivial and is just the unit lower triangle matrix with the nonzero subdiagonal entries of $\mathbf{M}_1^{-1}$, $\mathbf{M}_2^{-1}$, and $\mathbf{M}_3^{-1}$ inserted in the appropriate places. (Notice that the similar is not true for $\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1$.)

All together, we finally have our decomposition $\mathbf{A} = \mathbf{LU}$:

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix}. \tag{2.26}$$

□

While we have proved it only for this particular example, it is easy to see how the proof could generalize for any non-singular matrix $\mathbf{A}$, and in the process, suggest how to write down a much more compact version of the Gaussian elimination algorithm:

Algorithm:
do $j = 1$ to $m - 1$

$$\mathbf{M}_j = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & -a_{j+1,j}/a_{jj} & 1 & & & \\ & & -a_{j+2,j}/a_{jj} & & \ddots & & \\ & & \vdots & & & \ddots & \\ & & -a_{m,j}/a_{jj} & & & & 1 \end{pmatrix}$$

$\mathbf{A} = \mathbf{M}_j\mathbf{A}$
$\mathbf{b} = \mathbf{M}_j\mathbf{b}$
enddo

Note, however, that creating the matrix $\mathbf{M}_j$ and multiplying both $\mathbf{A}$ and $\mathbf{b}$ by it at every iteration is very wasteful both in memory and time, so this more compact version is never used in practical implementations of the algorithm. It is only really useful for illustration purposes, and sometimes for proving theorems.

In any case, the resulting transformed linear system is $\mathbf{MAx} = \mathbf{Ux} = \mathbf{Mb} = \mathbf{y}$, where $\mathbf{M} = \mathbf{M}_{m-1} \ldots \mathbf{M}_1$ is lower triangular and $\mathbf{U}$ is upper triangular. The equivalent problem $\mathbf{Ux} = \mathbf{y}$ can be solved by back-substitution to obtain the solution to the original linear system $\mathbf{Ax} = \mathbf{b}$. This is done simply as

---

Algorithm: Backsubstitution of $\mathbf{Ux} = \mathbf{y}$ ($\mathbf{U}$ is upper triangular):

$x_m = y_m/u_{mm}$![stop if u_{mm} is zero, singular matrix]

do $i = m - 1$ to 1
    ![loop over lines, bottom to top]
    if $u_{ii} = 0$ then
        stop ![stop if entry is zero, singular matrix]
    endif
    sum = 0.0 ![initialize to zero first]
    do $k = i + 1$ to m
        sum = sum + $u_{ik}x_k$
    enddo
    $x_i = (y_i - \text{sum})/u_{ii}$
enddo

Note:

- Some algorithms simply return the solution within the RHS vector $\mathbf{y}$ instead of returning it as a separate vector $\mathbf{x}$. To do so simply replace the last operation with $y_i = (y_i - \texttt{sum})/u_{ii}$, which gradually overwrites the entries of $\mathbf{y}$ as they are no longer needed.

- It is very easy to do the same operations at the same time on many RHS vectors $\mathbf{y}$. To do so, create a matrix $\mathbf{Y}$ formed by all the RHS column-vectors and perform the Gaussian elimination and backsubstitution on the whole matrix $\mathbf{Y}$ at the same time (see, e.g., Numerical Recipes for an example).

4. Gaussian elimination with pivoting

We obviously run into trouble when the choice of a divisor – called a **pivot** – is zero, whereby the Gaussian elimination algorithm breaks down. The solution to this singular pivot issue is fairly straightforward: if the pivot entry is zero at stage k, i.e., $a_{kk} = 0$, then we interchange row k of *both* the matrix and the right hand side vector with some subsequent row whose entry in column k is nonzero and resume the process as usual. Recall that permutation is an invariant transformation that does not alter the solution to the system.

This row interchanging is part of a process called **pivoting**, which is illustrated in the following example.

Example: Suppose that after zeroing out the subdiagonal in the first column, the next diagonal entry is zero (red line). Then simply swap it with one of the rows below that, e.g. here the blue one.

$$\begin{bmatrix} * & * & * & * \\ 0 & 0 & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \xrightarrow{\mathbf{P}} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \end{bmatrix} \tag{2.27}$$

where the permutation matrix $\mathbf{P}$ is given as

$$\mathbf{P} = \begin{bmatrix} 1 & & & \\ & & & 1 \\ & & 1 & \\ & 1 & & \end{bmatrix}. \tag{2.28}$$

□

Problems do not only occur with zero pivots, but also when the pivots are very small, i.e. close to or below machine precision ϵ_{mach}. Recall that we have $\epsilon_{\text{mach}} \approx 10^{-7}$ for single precision, and $\epsilon_{\text{mach}} \approx 10^{-16}$ for double precision.

Example: Let us consider the problem

$$\mathbf{A} = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \tag{2.29}$$

where $\epsilon < \epsilon_{\text{mach}} \approx 10^{-16}$, say, $\epsilon = 10^{-20}$. The real solution of this problem is

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1}{1-\epsilon} \\ \frac{1-2\epsilon}{1-\epsilon} \end{pmatrix} \simeq \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{2.30}$$

However, let's see what a numerical algorithm would give us. If we proceed without any pivoting (i.e., no row interchange) then the first step of the elimination algorithm gives

$$\mathbf{A} = \begin{pmatrix} \epsilon & 1 \\ 0 & 1-\epsilon^{-1} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2-\epsilon^{-1} \end{pmatrix}, \tag{2.31}$$

which is numerically equivalent to

$$\mathbf{A} = \begin{pmatrix} \epsilon & 1 \\ 0 & -\epsilon^{-1} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ -\epsilon^{-1} \end{pmatrix}, \tag{2.32}$$

in floating point arithmetic since $1/\epsilon \simeq 10^{20} \gg 1$. The solution via back-substitution is then

$$y = \frac{-\epsilon^{-1}}{-\epsilon^{-1}} = 1 \text{ and } x = \frac{1-1}{\epsilon} = 0 \tag{2.33}$$

which is very, very far from being the right answer!! We see that using a small pivot, and a correspondingly large multiplier, has caused an unrecoverable loss of information in the transformation. Also note that the original matrix here is far from being singular, it is in fact a very well-behaved matrix.

As it turns out, we can easily cure the problem by interchanging the two rows first, which gives

$$\begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \tag{2.34}$$

so this time the elimination proceeds as

$$\begin{pmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 1-2\epsilon \end{pmatrix}, \tag{2.35}$$

which is numerically equal to

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \tag{2.36}$$

in floating-point arithmetic, and whose solution is $\mathbf{x} = (1,1)^T$. This time, we actually get the correct answer within machine accuracy. $\square$

The foregoing example is rather extreme, but large errors would also occur even if we had $\epsilon_{\text{mach}} \ll \epsilon \ll 1$. This suggests a general principle in which we want to make sure to always work with the largest possible pivot, in order to produce smaller errors in floating-point arithmetic, and stabilizes an otherwise very unstable algorithm. Gaussian elimination with **partial pivoting** therefore proceeds as below. At step j, find $p = \max_{k=j,\ldots,m} |a_{kj}|$ and select it as the $j-$th pivot. Then switch the lines j and K (if $j \neq K$), where K is the value of k for which $|a_{kj}|$ is maximal before zeroing out the subdiagonal elements of column j.

$$\begin{bmatrix} * & * & * & * \\ & * & * & * \\ & a_{kj} & * & * \\ & * & * & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * \\ & a_{kj} & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * \\ & a_{kj} & * & * \\ & 0 & * & * \\ & 0 & * & * \end{bmatrix} \tag{2.37}$$

Note that it is possible to use **full pivoting**, i.e. by selecting a pivot from *all* the entries in the lower right submatrix below the element a_{jj}. In practice, however, the extra counting and swapping work required is not usually worth it.

Algorithm: Gaussian elimination with Partial Pivoting:

```
do j = 1 to m − 1
    ![loop over column]
    Find index K and pivot p such that p = |a_Kj| = max_{k=j,...,m} |a_kj|
    if K ≠ j then
        interchange rows K and j
        ![interchange rows if needed]
    endif
    if a_jj = 0 then
        stop
        ![matrix is singular]
    endif
    do i = j + 1 to m
    ![loop over rows below row j]
        r_i = r_i − a_ij r_j / a_jj
        ![transformation of remaining submatrix]
        b_i = b_i − a_ij b_j / a_jj
        ![transformation of RHS vector]
    enddo
enddo
```

Finally, note that there is a slight oddity in the algorithm in the sense that if a whole row of the matrix is multiplied by a large number, and similarly the corresponding entry in the RHS is multiplied by the same value, then this row is *guaranteed* to contain the first pivot even though the actual problem is exactly the same as the original one. If you are worried about this, you may consider using the **implicit pivoting** algorithm, where each row of the augmented matrix (i.e. matrix and RHS) is first scaled by its largest entry in absolute value (see Numerical Recipes for detail).

5. Gauss-Jordan elimination for the calculation of the inverse of a matrix.

Chapter 2.1 of Numerical Recipes

Gaussian elimination (with pivoting) works very well as long as the RHS vector(s) $\mathbf{b}$ is (are) known ahead of time, since the algorithm needs to operate on the RHS at the same time as it operates on $\mathbf{A}$. This is fine for applications such as fitting linear functions to a set of points (as in the first example above), where each new set of points gives rise to a new matrix problem that needs to be solved from scratch. On the other hand, as discussed above, the evolution of the solution of the PDE using the Crank-Nicholson algorithm from timesteps $t^{(n)}$ to $t^{(n+1)}$ requires the solution of $\mathbf{A}\mathbf{f}^{(n+1)} = \mathbf{B}\mathbf{f}^{(n)}$, where the right-hand-side vector $\mathbf{B}\mathbf{f}^{(n)}$ changes at each timestep $t^{(n)}$. In this case it is better to calculate the inverse of $\mathbf{A}$, store it, and then merely perform the matrix multiplication $\mathbf{f}^{(n+1)} = \mathbf{A}^{-1}\mathbf{B}\mathbf{f}^{(n)}$ at each timestep to advance the solution (although, see also the next section on LU factorization).

A very basic and direct method for obtaining and storing the inverse of a matrix $\mathbf{A}$ is to use the so-called **Gauss-Jordan elimination** on the **augmented matrix** formed by $\mathbf{A}$ and $\mathbf{I}$.

Definition: Let us introduce a form called **augmented matrix** of the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ which writes the $m \times m$ matrix $\mathbf{A}$ and the m-vector $\mathbf{b}$ together in a new $m \times (m+1)$ matrix form:

$$\left[\mathbf{A}\middle|\mathbf{b}\right]. \tag{2.38}$$

The use of augmented matrix allows us to write each transformation step of the linear system (i.e., both $\mathbf{A}$ and $\mathbf{b}$) in a compact way. Note that we could have done this for Gaussian elimination too.

Example: Consider the following system using Gauss-Jordan elimination without pivoting:

$$\begin{array}{llll} x_1 & +x_2 & +x_3 & = 4 \\ 2x_1 & +2x_2 & +5x_3 & = 11 \\ 4x_1 & +6x_2 & +8x_3 & = 24 \end{array}, \tag{2.39}$$

which can be put in as an augmented matrix form:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 2 & 5 & 11 \\ 4 & 6 & 8 & 24 \end{array}\right]. \tag{2.40}$$

First step is to annihilate the first column:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 2 & 5 & 11 \\ 4 & 6 & 8 & 24 \end{array}\right] \xrightarrow{\mathbf{M}_1} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 0 & 3 & 3 \\ 0 & 2 & 4 & 8 \end{array}\right], \text{ where } \mathbf{M}_1 = \begin{bmatrix} 1 & & \\ -2 & 1 & \\ -4 & & 1 \end{bmatrix}. \tag{2.41}$$

Next we permute to get rid of the zero (so there is some basic pivoting involved):

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 0 & 3 & 3 \\ 0 & 2 & 4 & 8 \end{array}\right] \xrightarrow{\mathbf{P}_1} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 2 & 4 & 8 \\ 0 & 0 & 3 & 3 \end{array}\right], \text{ where } \mathbf{P}_1 = \begin{bmatrix} 1 & & \\ & & 1 \\ & 1 & \end{bmatrix}. \tag{2.42}$$

Next row scaling by multiplying a diagonal matrix $\mathbf{D}_1$:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 2 & 4 & 8 \\ 0 & 0 & 3 & 3 \end{array}\right] \xrightarrow{\mathbf{D}_1} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 4 \\ 0 & 0 & 1 & 1 \end{array}\right], \text{ where } \mathbf{D}_1 = \begin{bmatrix} 1 & & \\ & 1/2 & \\ & & 1/3 \end{bmatrix}. \tag{2.43}$$

Next annihilate the remaining upper diagonal entries in the third column:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 4 \\ 0 & 0 & 1 & 1 \end{array}\right] \xrightarrow{\mathbf{M}_2} \left[\begin{array}{ccc|c} 1 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{array}\right], \text{ where } \mathbf{M}_2 = \begin{bmatrix} 1 & & -1 \\ & 1 & -2 \\ & & 1 \end{bmatrix}. \tag{2.44}$$

Finally, annihilate the upper diagonal entry in the second column:

$$\left[\begin{array}{ccc|c} 1 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{array}\right] \xrightarrow{\mathbf{M}_3} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{array}\right], \text{ where } \mathbf{M}_3 = \begin{bmatrix} 1 & -1 & \\ & 1 & \\ & & 1 \end{bmatrix}. \tag{2.45}$$

□

In this example the right hand side is a single m-vector. What happens if we perform the same procedure using multiple m-vectors? Then we get the augmented matrix

$$\left[\mathbf{A} \middle| \mathbf{b}_1 \sqcup \mathbf{b}_2 \sqcup \cdots \sqcup \mathbf{b}_n\right]. \tag{2.46}$$

We see that the same operation can easily be performed simultaneously on individual $\mathbf{b}_i, 1 \leq i \leq n$.

Especially, if we choose m vectors $\mathbf{b}_i = \mathbf{e}_i$ then the matrix formed by these column vectors is the identity matrix $\mathbf{I}$. In this case we see that Gauss-Jordan elimination yields the inverse of $\mathbf{A}$, that is, the solution of $\mathbf{AX} = \mathbf{I}$.

$$\left[\mathbf{A}\middle|\mathbf{e}_1 \sqcup \mathbf{e}_2 \sqcup \cdots \sqcup \mathbf{e}_n\right] = \left[\mathbf{A}\middle|\mathbf{I}\right] \to \cdots \to \left[\mathbf{I}\middle|\mathbf{A}^{-1}\right]. \tag{2.47}$$

Quick summary: The process of calculating the inverse of a matrix by Gauss-Jordan elimination can be illustrated as in the following pictorial steps:

$$\begin{aligned}
&\left[\begin{array}{cccc|cccc} * & * & * & * & 1 & 0 & 0 & 0 \\ * & * & * & * & 0 & 1 & 0 & 0 \\ * & * & * & * & 0 & 0 & 1 & 0 \\ * & * & * & * & 0 & 0 & 0 & 1 \end{array}\right] \to \left[\begin{array}{cccc|cccc} 1 & * & * & * & * & 0 & 0 & 0 \\ 0 & * & * & * & * & 1 & 0 & 0 \\ 0 & * & * & * & * & 0 & 1 & 0 \\ 0 & * & * & * & * & 0 & 0 & 1 \end{array}\right] \\
\to &\left[\begin{array}{cccc|cccc} 1 & 0 & * & * & * & * & 0 & 0 \\ 0 & 1 & * & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & * & 1 & 0 \\ 0 & 0 & * & * & * & * & 0 & 1 \end{array}\right] \to \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & * & * & * & * & 0 \\ 0 & 1 & 0 & * & * & * & * & 0 \\ 0 & 0 & 1 & * & * & * & * & 0 \\ 0 & 0 & 0 & * & * & * & * & 1 \end{array}\right] \\
\to &\left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & * & * & * & * \\ 0 & 1 & 0 & 0 & * & * & * & * \\ 0 & 0 & 1 & 0 & * & * & * & * \\ 0 & 0 & 0 & 1 & * & * & * & * \end{array}\right]
\end{aligned} \tag{2.48}$$

where the matrix on the RHS of the last iteration is $\mathbf{A}^{-1}$. $\square$

Note that pivoting is just as important for Gauss-Jordan elimination as it is for Gaussian elimination. As a result, here is a basic algorithm for Gauss-Jordan elimination with partial pivoting. This assumes that $\mathbf{C} = [\mathbf{A}|\mathbf{B}]$ where $\mathbf{A}$ is a non-singular $m \times m$ matrix, and $\mathbf{B}$ is $m \times n$ matrix consisting of all the RHS vectors. If $\mathbf{B} = \mathbf{I}$, then the algorithm returns $\mathbf{A}^{-1}$ in the place of $\mathbf{B}$. Otherwise, it simply returns the solution to $\mathbf{AX} = \mathbf{B}$ in that place.

Algorithm: Gauss-Jordan elimination with Partial Pivoting:

```
do j = 1 to m
   ![loop over column]
   Find index K and pivot p such that p = |c_Kj| = max_{k=j,...,m} |c_kj|
   if K ≠ j then
      interchange rows K and j
   endif
   if c_jj = 0 then
      stop ![matrix is singular]
   endif
   r_j = r_j / c_jj      ![scale row j so diagonal element is 1]
   do i = 1 to m
      if i ≠ j then
   ![loop over all rows except j]
         r_i = r_i − c_ij r_j
         ![transformation of remaining submatrix]
      endif
   enddo
enddo
```

6. Operation counts for basic algorithms

The two main *efficiency* concerns for numerical linear algebra algorithm are:

- time efficiency
- storage efficiency

While these concerns are mild for small problems, they can become very serious when the matrices are very large. In this section, we will rapidly look at the first one. A good way of estimating the efficiency and speed of execution of an algorithm is to count the number of floating point operations performed by the algorithm.

Example 1: Matrix multiplication $\mathbf{AB}$, where $\mathbf{A}$, $\mathbf{B}$ are $m \times m$ matrices
In computing $\mathbf{C} = \mathbf{AB}$, we have to compute the m^2 coefficients of $\mathbf{C}$, which each involves calculating

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \tag{2.49}$$

or in other words, m multiplications and m additions. The total operation count is therefore of order $2m^3$, although because multiplications and divisions are much more costly than additions and subtractions, operation counts usually ignore the latter when they are dominated by the former. So the number of operations in matrix multiplications is $\sim m^3$. $\square$

Example 2: Back-substitution of $\mathbf{Ux} = \mathbf{y}$, where $\mathbf{U}$ is an $m \times m$ matrix
Running through the algorithm we have

$$
\begin{aligned}
x_m &= \frac{y_m}{u_{mm}} \rightarrow \text{ 0 add/sub, 1 mult/div} \\
x_{m-1} &= \frac{y_{m-1} - u_{m-1,m}x_m}{u_{m-1,m-1}} \rightarrow \text{ 1 add/sub, 2 mult/div} \\
x_{m-2} &= \frac{y_{m-2} - u_{m-2,m-1}x_{m-1} - u_{m-2,m}x_m}{u_{m-2,m-2}} \rightarrow \text{ 2 add/sub, 3 mult/div} \\
&\vdots \\
x_1 &= \frac{y_1 - \sum_{k=2}^{m} u_{1,k}x_k}{u_{11}} \rightarrow m-1 \text{add/sub and } m \text{ mult/div}
\end{aligned} \tag{2.50}
$$

This gives an operation count of order $m(m+1)/2$ multiplications/divisions, and $m(m-1)/2$ additions and subtractions, so a gran total of $\sim m^2/2$ multiplication/division operations per RHS vector if m is large. If we do a backsubstitution on n vectors simultaneously, then the operation count is $m^2n/2$. $\square$

Operation counts have been computed for the algorithms discussed so far for the solution of linear systems, namely Gaussian elimination + back-substitution and Gauss-Jordan elimination. This gives, for n right-hand-sides

- Gaussian elimination + backsubstitution: of order $\frac{m^3}{3} + \frac{m^2n}{2} + \frac{m^2n}{2}$.
- Gauss-Jordan elimination : of order $\frac{m^3}{2} + \frac{m^2n}{2}$.

so the latter is about 1.5 times as expensive as the former for small m, but 33% cheaper for $m = n$. In the case of matrix inversion, however, *if we cleverly avoid computing entries that we know are zero anyway* (which requires re-writing the GE algorithm on purpose), the operation counts between GE and GJ are the same, and about m^3 in both cases.

Finally, suppose we now go back to the problem of advancing a PDE forward in time, save the inverse, and apply it by matrix multiplication to each new right-hand-side. In this case, the matrix multiplication of a single vector at each timestep takes m^2 multiplications and m^2 additions.

7. LU factorization

Chapter 20 of the textbook

Another very popular way of solving linear systems, that has the advantage of having a minimal operation count *and* has the flexibility to be used as in the PDE problem, in a deferred way with multiple RHS vectors but the same matrix $\mathbf{A}$, is the **LU algorithm**. The idea is the following. Suppose we find a way of decomposing the matrix $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{LU} \tag{2.51}$$

where $\mathbf{L}$ is lower triangular and $\mathbf{U}$ is upper triangular. In that case, the system $\mathbf{Ax} = \mathbf{b}$ is equivalent to $\mathbf{LUx} = \mathbf{b}$, which can be solved in two steps:

- Solve $\mathbf{Ly} = \mathbf{b}$
- Solve $\mathbf{Ux} = \mathbf{y}$

The second of these steps looks very familiar – in fact, it is simply the same backsubstitution step as in the Gaussian elimination algorithm. Furthermore, we saw that Gaussian elimination can be interpreted as a sequence of multiplications by lower triangular matrices to transform $\mathbf{A}$ into $\mathbf{U}$, and this can be used to construct the LU decomposition.

7.1. Relationship between Gaussian elimination and LU factorization

Ignoring pivoting for a moment, recall that transforming the matrix $\mathbf{A}$ via Gaussian elimination into an upper-triangular matrix involves writing $\mathbf{U}$ as

$$\mathbf{U} = \mathbf{MA} = \mathbf{M}_{m-1} \dots \mathbf{M}_1 \mathbf{A} \tag{2.52}$$

where

$$\mathbf{M}_j = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & -l_{j+1,j} & 1 & & & \\ & & -l_{j+2,j} & & \ddots & & \\ & & \vdots & & & \ddots & \\ & & -l_{m,j} & & & & 1 \end{pmatrix}$$

and the coefficients l_{ij} are constructed from the elements below the diagonal of the matrix $\mathbf{M}_{j-1} \dots \mathbf{M}_1 \mathbf{A}$ (i.e. the matrix obtained at the previous step of the iteration process).

It is easy to show that since all the $\mathbf{M}_j$ matrices are lower triangular, then so is their product $\mathbf{M}$. Furthermore, it is also relatively easy to show that the inverse of a lower-triangular matrix is also lower triangular, so writing $\mathbf{MA} = \mathbf{U}$ is equivalent to $\mathbf{A} = \mathbf{M}^{-1}\mathbf{U}$ which is in LU form with $\mathbf{L} = \mathbf{M}^{-1}$! All that remains to be done is to evaluate $\mathbf{M}^{-1}$. As it turns out, calculating the inverse of $\mathbf{M}$ is actually fairly trivial as long as the coefficients l_{ij} of each matrix $\mathbf{M}_j$ are known – and this can be done by Gaussian elimination.

Indeed, first note that if we define the vector $\mathbf{l}_j = (0, 0, \dots, 0, l_{j+1,j}, l_{j+2,j}, \dots, l_{m,j})^T$, then $\mathbf{M}_j$ can be written more compactly in terms of $\mathbf{l}_j$ as $\mathbf{M}_j = \mathbf{I} - \mathbf{l}_j \mathbf{e}_j^*$. We can then verify that $\mathbf{M}_j^{-1} = \mathbf{I} + \mathbf{l}_j \mathbf{e}_j^*$ simply by constucting the product

$$\mathbf{M}_j \mathbf{M}_j^{-1} = (\mathbf{I} - \mathbf{l}_j \mathbf{e}_j^*)(\mathbf{I} + \mathbf{l}_j \mathbf{e}_j^*) = \mathbf{I} - \mathbf{l}_j \mathbf{e}_j^* + \mathbf{l}_j \mathbf{e}_j^* - \mathbf{l}_j \mathbf{e}_j^* \mathbf{l}_j \mathbf{e}_j^* \tag{2.53}$$

Now it's easy to check that $\mathbf{e}_j^* \mathbf{l}_j$ is zero, proving that $\mathbf{M}_j \mathbf{M}_j^{-1} = \mathbf{I}$. In other words, finding the inverse of $\mathbf{M}_j$ merely requires negating its subdiagonal components. $\square$

Next, another remarkable property of the matrices $\mathbf{M}_j$ and their inverses is that their product $\mathbf{L} = \mathbf{M}^{-1} = \mathbf{M}_1^{-1}\mathbf{M}_2^{-1}\dots\mathbf{M}_{m-1}^{-1}$ can also very easily be calculated. Indeed,

$$\mathbf{M}_i^{-1}\mathbf{M}_j^{-1} = (\mathbf{I} + \mathbf{l}_i\mathbf{e}_i^*)(\mathbf{I} + \mathbf{l}_j\mathbf{e}_j^*) = \mathbf{I} + \mathbf{l}_i\mathbf{e}_i^* + \mathbf{l}_j\mathbf{e}_j^* + \mathbf{l}_i\mathbf{e}_i^*\mathbf{l}_j\mathbf{e}_j^* \tag{2.54}$$

The last term is zero as long as $i \leq j$, which is always the case in the construction of $\mathbf{M}^{-1}$. This is because $\mathbf{e}_i^*\mathbf{l}_j = 0$, because the product would be equal to the i-th component of the vector $\mathbf{l}_j$, which is zero as long as $i \leq j$. This shows that $\mathbf{M}_i^{-1}\mathbf{M}_j^{-1} = \mathbf{I} + \mathbf{l}_i\mathbf{e}_i^* + \mathbf{l}_j\mathbf{e}_j^*$, and so $\mathbf{L} = \mathbf{M}^{-1} = \mathbf{I} + \mathbf{l}_1\mathbf{e}_1^* + \mathbf{l}_2\mathbf{e}_2^* + \dots + \mathbf{l}_{m-1}\mathbf{e}_{m-1}^*$, which is the matrix

$$\mathbf{L} = \begin{pmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ l_{m1} & l_{m2} & \dots & l_{m,m-1} & 1 \end{pmatrix}$$

These considerations therefore suggest the following algorithm:

Algorithm: LU factorization by Gaussian elimination (without pivoting), version 1:

```
do j = 1 to m − 1
    ![loop over columns]
    if a_jj = 0 then
        stop
        ![stop if pivot (or divisor) is zero]
    endif
    do i = j + 1 to m
        l_ij = a_ij / a_jj
        ![create the non-zero coefficients of l_j]
    enddo
    A = A − l_j e_j^* A
    ![Overwrite A by M_j A]
enddo
```

□

Note that the operation $\mathbf{A} = \mathbf{A} - \mathbf{l}_j\mathbf{e}_j^*\mathbf{A}$ can either be written component-wise, or as vector operations as written here, in which case the most efficient way of doing this is to evaluate first $\mathbf{e}_j^*\mathbf{A}$ then multiply by $\mathbf{l}_j$. Also note that this algorithm doesn't discuss storage (i.e. where to put the l_{ij} coefficients, etc...), and as such is merely illustrative. At the end of this algorithm, the matrix $\mathbf{A}$ becomes the matrix $\mathbf{U}$ and the matrix $\mathbf{L}$ can be formed separately if needed by the combination of all the $\mathbf{l}_j$ vectors, as discussed above.

A common way of storing the entries l_{ij} is to actually put them in the matrix $\mathbf{A}$, gradually replacing the zeroes that would normally occur by Gaussian elimination. This is very efficient storage-wise, but then prohibits the use of the compact

form of the algorithm, requiring instead that the operation $\mathbf{A} = \mathbf{A} - \mathbf{l}_j \mathbf{e}_j^* \mathbf{A}$ be written out component-wise. The end product, after completion, are the matrix $\mathbf{L}$ and $\mathbf{U}$ stored into the matrix $\mathbf{A}$ as

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1,m-1} & u_{1m} \\ l_{21} & u_{22} & u_{23} & \dots & u_{2,m-1} & u_{2m} \\ l_{31} & l_{32} & u_{33} & \dots & u_{3,m-1} & u_{3m} \\ \vdots & \dots & \ddots & \ddots & \dots & \dots \\ l_{m1} & l_{m2} & \dots & & l_{m,m-1} & u_{mm} \end{pmatrix} \tag{2.55}$$

Algorithm: LU factorization by Gaussian elimination (without pivoting), version 2:

```
do j = 1 to m − 1
    ![loop over columns]
    if a_jj = 0 then
        stop
        ![stop if pivot (or divisor) is zero]
    endif
    do i = j + 1 to m
        a_ij = a_ij / a_jj
        ![create the l_ij and stores them in a_ij]
        do k = j + 1 to m
            a_ik = a_ik − a_ij a_jk
        enddo
    ![Updates A]
    enddo
enddo
```

□

This algorithm can now directly be implemented in Fortran as is. Also note that because the operations are *exactly the same* as for Gaussian elimination, the operation count is also exactly the same.

7.2. Pivoting for the LU algorithm

Chapter 21 of the textbook

Since LU decomposition and Gaussian elimination are essentially identical – merely different interpretations of the same matrix operations – pivoting is just as important here. However, how do we do it in practice, and how does it affect the LU decomposition? The key is to remember that in the Gaussian elimination algorithm, pivoting swaps rows of the RHS at the same time as it swaps rows of the matrix $\mathbf{A}$. In terms of matrix operations, the pivoted Gaussian elimination algorithm can be thought of as a series of operations

$$\mathbf{M}_{m-1}\mathbf{P}_{m-1} \dots \mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1\mathbf{A}\mathbf{x} \equiv \mathbf{U}\mathbf{x} = \mathbf{M}_{m-1}\mathbf{P}_{m-1} \dots \mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1\mathbf{b} \equiv \mathbf{y} \tag{2.56}$$

where the $\mathbf{M}_j$ matrices are defined as earlier, and where the $\mathbf{P}_j$ matrices are permutations matrices. We already saw an example of permutation matrix ear-

lier, but it is worth looking at them in a little more detail now. The matrix $\mathbf{P}_j$ swaps row j with a row $k \geq j$. If $k = j$, then $\mathbf{P}_j$ is simply the identity matrix, but if $k > j$ then $\mathbf{P}_j$ is the identity matrix where the row k and j have been swapped. For instance, a matrix permuting rows 3 and 5 is

$$\mathbf{P} = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 0 & & 1 & \\ & & & 1 & & \\ & & 1 & & 0 & \end{pmatrix} \tag{2.57}$$

By definition, applying $\mathbf{P}_j\mathbf{P}_j\mathbf{A} = \mathbf{A}$ which shows that $\mathbf{P}_j^{-1} = \mathbf{P}_j$. Also, it's easy to verify that the product $\mathbf{A}\mathbf{P}_j$ permutes the *columns* of $\mathbf{A}$ rather than its rows.

Let's now go back to the product $\mathbf{M}_{m-1}\mathbf{P}_{m-1}\ldots\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1$. By contrast with the unpivoted algorithm, where we were able to show that $\mathbf{M}_{m-1}\ldots\mathbf{M}_1$ is a unit lower triangular matrix, it is not clear at all that the same property applies here – in fact, it doesn't! However, what we can show is that

$$\mathbf{M}_{m-1}\mathbf{P}_{m-1}\ldots\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1 = \mathbf{M}'\mathbf{P} \tag{2.58}$$

where $\mathbf{M}'$ is also a unit lower triangular matrix (different from the unpivoted $\mathbf{M}$) and where $\mathbf{P} = \mathbf{P}_{m-1}\ldots\mathbf{P}_2\mathbf{P}_1$ is the product of all the permutation matrices applied by pivoting, which is itself a permutation matrix.

Proof: To show this, consider for simplicity a case where $m = 4$. Let's rewrite it as follows

$$\begin{aligned} &\mathbf{M}_3\mathbf{P}_3\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1 && (2.59)\\ &= \mathbf{M}_3\mathbf{P}_3\mathbf{M}_2(\mathbf{P}_3^{-1}\mathbf{P}_3)\mathbf{P}_2\mathbf{M}_1(\mathbf{P}_2^{-1}\mathbf{P}_3^{-1}\mathbf{P}_3\mathbf{P}_2)\mathbf{P}_1 && (2.60)\\ &= (\mathbf{M}_3)(\mathbf{P}_3\mathbf{M}_2\mathbf{P}_3^{-1})(\mathbf{P}_3\mathbf{P}_2\mathbf{M}_1\mathbf{P}_2^{-1}\mathbf{P}_3^{-1})(\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1) && (2.61)\\ &\equiv (\mathbf{M}_3^{'})(\mathbf{M}_2^{'})(\mathbf{M}_1^{'})\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1, && (2.62) \end{aligned}$$

whereby we can define the $\mathbf{M}_i^{'}$ matrices as

$$\begin{aligned} \mathbf{M}_3^{'} &= \mathbf{M}_3 && (2.63)\\ \mathbf{M}_2^{'} &= \mathbf{P}_3\mathbf{M}_2\mathbf{P}_3^{-1} && (2.64)\\ \mathbf{M}_1^{'} &= \mathbf{P}_3\mathbf{P}_2\mathbf{M}_1\mathbf{P}_2^{-1}\mathbf{P}_3^{-1} && (2.65) \end{aligned}$$

These matrices look complicated, but in fact they are just equal to $\mathbf{M}_i$ with the *subdiagonal entries* permuted by the pivoting (as opposed to the whole rows permuted). To see why, note that in each expression the permutation matrices operating on $\mathbf{M}_j$ always have an index *greater* than j. Let's look at an example – suppose we consider the product $\mathbf{P}_2\mathbf{M}_1\mathbf{P}_2^{-1}$, and say, for the sake of example, that $\mathbf{P}_2$ swaps rows 2 and 4. Then

$$\mathbf{P}_2\begin{pmatrix} 1 & & & \\ -l_{21} & 1 & & \\ -l_{31} & & 1 & \\ -l_{41} & & & 1 \end{pmatrix}\mathbf{P}_2^{-1} = \begin{pmatrix} 1 & & & \\ -l_{41} & 0 & & 1 \\ -l_{31} & & 1 & \\ -l_{21} & 1 & & 0 \end{pmatrix}\mathbf{P}_2 = \begin{pmatrix} 1 & & & \\ -l_{41} & 1 & & \\ -l_{31} & & 1 & \\ -l_{21} & & & 1 \end{pmatrix} \tag{2.66}$$

This easily generalizes, so we can see that the matrices $\mathbf{M}'_j$ have exactly the same properties as the matrices $\mathbf{M}_j$, implying for instance that their product $\mathbf{M}' = \mathbf{M}'_3\mathbf{M}'_2\mathbf{M}'_1$ is also unit lower triangular. We have therefore shown, as required, that $\mathbf{M}_3\mathbf{P}_3\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1 = \mathbf{M}'\mathbf{P}$ where $\mathbf{M}'$ is unit lower triangular, and where $\mathbf{P} = \mathbf{P}_3\mathbf{P}_2\mathbf{P}_1$ is a permutation matrix. □

Having shown that the pivoted Gaussian elimination algorithm equivalent to transforming the problem $\mathbf{Ax} = \mathbf{b}$ into

$$\mathbf{M}'\mathbf{PAx} = \mathbf{M}'\mathbf{Pb} \equiv \mathbf{y} \tag{2.67}$$

where $\mathbf{M}'\mathbf{PA}$ is an upper triangular matrix $\mathbf{U}$, we therefore have

$$\mathbf{PA} = \mathbf{LU} \tag{2.68}$$

where this time, $\mathbf{L} = (\mathbf{M}')^{-1}$.

What does this all mean in practice? Well, a few things.

- The first is that we can now create the LU decomposition just as before, but we need make sure to swap the lower diagonal entries of the matrix $\mathbf{L}$ as we progressively construct it. If these are stored in $\mathbf{A}$, as in version 2 of the LU algorithm, this is done trivially!
- Second, solving the problem $\mathbf{Ax} = \mathbf{b}$ is still equivalent to solving $\mathbf{Ux} = \mathbf{y}$ but now $\mathbf{y} = \mathbf{L}^{-1}\mathbf{Pb}$. So we need to record the permutation matrix $\mathbf{P}$ to compute $\mathbf{y}$. Note that it is not necessary to save the entire matrix $\mathbf{P}$ to record the permutation – this would be quite wasteful. We can simply record the integer permutation vector $\mathbf{s}$.

These two considerations give us the revised pivoted LU algorithm, together with a corresponding backsubstitution algorithm.

Algorithm: LU factorization by Gaussian elimination (with partial pivoting):

```
![Initialize permutation vector]
do j = 1 to m
   s_j = j
enddo

do j = 1 to m
   ![loop over columns]
   Find index K and pivot p such that p = |a_Kj| = max_{k=j,...,m} |a_kj|
   if K ≠ j then
      interchange rows K and j of A
      interchange K and j entries of s
      ![interchange rows and record permutation]
   endif
   if a_jj = 0 then
      stop
      ![stop if pivot (or divisor) is zero]
```

```
    endif
    do i = j + 1 to m
        a_ij = a_ij / a_jj
        ![create the l_ij and stores them in a_ij]
        do k = j + 1 to m
            a_ik = a_ik − a_ij a_jk
        enddo
    ![Updates A]
    enddo
enddo
```

Note: this time the loop goes from $j = 1$ to $j = m$. This is because we just need to record the last swapping index.

LU backsubstitution in general (i.e. with or without pivoting) is a little trickier than Gaussian elimination backsubstitution since we first have to create the vector $\mathbf{y} = \mathbf{L}^{-1}\mathbf{Pb}$. This is called a forward substitution. The texbook is remarkably silent on the topic, but the algorithm can be worked out reasonably easily if we remember that

- The vector $\mathbf{Pb}$ is just a permutation of the entries of the vector $\mathbf{b}$, and the vector $\mathbf{s}$ was constructed so that $(\mathbf{Pb})_i = b_{s_i}$.
- Although the matrix $\mathbf{L}^{-1}$ is fairly tricky to compute directly, this computation is not actually needed. Indeed, recall that $\mathbf{L}^{-1} = \mathbf{M} = \mathbf{M}_{m-1} \dots \mathbf{M}_1$, and $\mathbf{M}_j = \mathbf{I} - \mathbf{l}_j \mathbf{e}_j^*$.

This implies that we can create $\mathbf{y} = \mathbf{L}^{-1}\mathbf{Pb}$ by first letting $\mathbf{y} = \mathbf{Pb}$, and then successively applying the aglorithm $\mathbf{y} = \mathbf{M}_j\mathbf{y}$ where

$$\mathbf{M}_j\mathbf{y} = \mathbf{y} - \mathbf{l}_j\mathbf{e}_j^*\mathbf{y} = \mathbf{y} - y_j\mathbf{l}_j \tag{2.69}$$

which is now easy to express in component form since we know that $\mathbf{l}_j = (0, 0, \dots, 0, l_{j+1,j}, l_{j+2,j}, \dots, l_{mj})^T$. We therefore get

---

Algorithm: LU backsubstitution:

```
![Initialize y with Pb]
do j = 1 to m
    y_i = b_{s_i}
enddo

![Forward substitution, y = L^{-1}Pb]
do j = 1 to m − 1
![Do y = M_j y]
    do i = j + 1 to m
        y_i = y_i − y_j a_ij
    enddo
enddo
```

```
![Backward substitution, Ux = y]
do i = m to 1
    ![loop over lines, bottom to top]
    if u_ii = 0 then
        stop
        ![stop if entry is zero, singular matrix]
    endif
    sum = 0.
    do k = i + 1 to m
        sum = sum + u_ik x_k
    enddo
    x_i = (y_i − sum)/a_ii
enddo
```

---

8. Cholesky factorization for Hermitian positive definite systems

Chapter 23 of the textbook

Thus far we have assumed that the linear system has a general square non-singular matrix $\mathbf{A}$, that is otherwise unremarkable, and have learned algorithms for the solution of $\mathbf{Ax} = \mathbf{b}$ that work for any such matrix. However, in some cases we know that the matrix $\mathbf{A}$ has special properties that allow us to use more specialized algorithms, that are often faster, but restricted only to matrices that share these properties.

A well-known example of such algorithms is the Cholesky decomposition, that is only applicable to **positive definite Hermitian matrices**. Such matrices satisfy the property that

$$\mathbf{x}^*\mathbf{Ax} > 0 \tag{2.70}$$

for *any* nonzero vector $\mathbf{x}$.

The fact that $\mathbf{x}^*\mathbf{Ax}$ is real simply comes from the fact that the matrix is Hermitian. Indeed, the complex conjugate of $\mathbf{x}^*\mathbf{Ax}$ is $\mathbf{x}^*\mathbf{A}^*\mathbf{x} = \mathbf{x}^*\mathbf{Ax}$ since $\mathbf{A}^* = \mathbf{A}$. If a number is equal to its complex conjuate, this simply shows it is real. The property $\mathbf{x}^*\mathbf{Ax} > 0$ on the other hand is the defining property of **positive definite** matrices. It means that the action of a Hermitian positive definite matrix on a vector always return a vector whose projection on the original one is positive. Proving that a matrix is positive definite is not always easy, but they often arise in many linear algebra problems that derive from the solution of a PDE, for instance, and as such, are very common.

Note that if you have a **negative definite** matrix, i.e. a matrix with the property that

$$\mathbf{x}^*\mathbf{Ax} < 0 \tag{2.71}$$

for *any* $\mathbf{x}$, then it is very easy to create the positive definite matrix $\mathbf{B} = -\mathbf{A}$. Non positive definite or negative definite matrices are matrices for which $\mathbf{x}^*\mathbf{A}\mathbf{x}$ is sometimes positive, and sometimes negative, depending on the input vector $\mathbf{x}$.

If the matrix $\mathbf{A}$ is symmetric and positive definite (SPD), then an LU decomposition of $\mathbf{A}$ indicates that $\mathbf{A} = \mathbf{L}\mathbf{U}$. This then implies that

$$\mathbf{A}^* = (\mathbf{L}\mathbf{U})^* = \mathbf{U}^*\mathbf{L}^* = \mathbf{A} = \mathbf{L}\mathbf{U} \tag{2.72}$$

so we have just shown that the LU decomposition of a Hermitian matrix satisfies

$$\mathbf{U}^*\mathbf{L}^* = \mathbf{L}\mathbf{U} \tag{2.73}$$

Since the transpose of an upper-triangular matrix is a lower triangular one, and vice versa, this also shows that one should in principle be able to find a decomposition such that $\mathbf{L} = \mathbf{U}^*$, or $\mathbf{U} = \mathbf{L}^*$ so that

$$\mathbf{A} = \mathbf{L}\mathbf{U} = \mathbf{U}^*\mathbf{U} = \mathbf{L}\mathbf{L}^* \tag{2.74}$$

for any Hermitian matrix $\mathbf{A}$. In practice, this factorization only exists for positive definite matrices. To see why, note that

$$\mathbf{x}^*\mathbf{A}\mathbf{x} = \mathbf{x}^*\mathbf{U}^*\mathbf{U}\mathbf{x} = (\mathbf{U}\mathbf{x})^*(\mathbf{U}\mathbf{x}) > 0 \tag{2.75}$$

so the existence of a decomposition $\mathbf{A} = \mathbf{U}^*\mathbf{U}$ (or equivalently $\mathbf{L}\mathbf{L}^*$) does indeed depend on the positive definiteness of $\mathbf{A}$.

This decomposition is known as the *Cholesky factorization* of $\mathbf{A}$:

Definition: A *Cholesky factorization* of a Hermitian positive definite matrix $\mathbf{A}$ is given by $A = \mathbf{U}^*\mathbf{U} = \mathbf{L}\mathbf{L}^*$, where $\mathbf{U}$ is upper-triangular and $\mathbf{L}$ is lower triangular such that $u_{jj} = l_{jj} > 0$.

To prove that it exists for any positive definite Hermitian matrix, we simply need to come up with an algorithm to systematically create it. Let us consider the decomposition $\mathbf{A} = \mathbf{L}\mathbf{L}^*$, and write it out in component form:

$$a_{ij} = \sum_{k=1}^{m} (\mathbf{L})_{ik}(\mathbf{L})^*_{kj} = \sum_{k=1}^{m} l_{ik}l^*_{jk} = \sum_{k=1}^{\min(i,j)} l_{ik}l^*_{jk} \tag{2.76}$$

using successively, the definition of the Hermitian transpose, and the fact that $\mathbf{L}$ is lower triangular so its coefficients are zero if the column number is larger than the row number. Writing these out for increasing values of i we have

- $i = 1$:

$$a_{11} = l_{11}l^*_{11} \tag{2.77}$$

- $i = 2$:

$$\begin{aligned} a_{21} &= l_{21}l^*_{11} \\ a_{22} &= l_{21}l^*_{21} + l_{22}l^*_{22} \end{aligned} \tag{2.78}$$

- $i = 3$:

$$\begin{aligned} a_{31} &= l_{31} l_{11}^* \\ a_{32} &= l_{31} l_{21}^* + l_{32} l_{22}^* \\ a_{33} &= l_{31} l_{31}^* + l_{32} l_{32}^* + l_{33} l_{33}^* \end{aligned} \tag{2.79}$$

and so forth. Note that because the matrix is Hermitian, $a_{11} = a_{11}^*$ and so a_{11} must be real, implying that the equation $a_{11} = l_{11} l_{11}^* = ||l_{11}||^2$ indeed makes sense.

Let's now choose $l_{11} = \sqrt{a_{11}}$. We can then move to the next equation, and write successively

$$\begin{aligned} l_{21} &= \frac{a_{21}}{l_{11}} \\ l_{22} = l_{22}^* &= \sqrt{a_{22} - l_{21} l_{21}^*} \end{aligned} \tag{2.80}$$

then

$$\begin{aligned} l_{31} &= \frac{a_{31}}{l_{11}} \\ l_{32} &= \frac{a_{32} - l_{31} l_{21}^*}{l_{22}} \\ l_{33} = l_{33}^* &= \sqrt{a_{33} - l_{31} l_{31}^* - l_{32} l_{32}^*} \end{aligned} \tag{2.81}$$

etc...

This can always be done, which provides us with a straightforward algorithm to construct the Cholesky factorization. Note that many version of this algorithm exist, which merely reorder some of these operations in different ways (cf textbook, Numerical Recipes, etc). They appear to be roughly equivalent in terms of time and stability. The one presented below is reasonably simple to understand, and merely corresponds to working column by column, first calculating the diagonal element in each column, that is, calculating

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk} l_{jk}^*} \tag{2.82}$$

then working on the column below that element, in which

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}^*}{l_{jj}} \tag{2.83}$$

The elements of $\mathbf{L}$ are simply stored by over-writing the lower triangular elements of $\mathbf{A}$.

Algorithm: Cholesky factorization (decomposition):

```
![loop over columns]
```

```
do j = 1 to m
    ![Calculate new diagonal element]
    do k = 1 to j − 1
        a_jj = a_jj − a_jk a*_jk
    enddo
    a_jj = √a_jj
    ![Calculate elements below diagonal]
    do i = j + 1 to m
        do k = 1 to j − 1
            a_ij = a_ij − a_ik a*_jk
        enddo
        a_ij = a_ij / a_jj
    enddo
enddo
```

Note: There is a number of facts about the CF algorithm that make it very attractive and popular for symmetric positive definite matrices:

- No pivoting is required for numerical stability.
- Only about $n^3/6$ multiplications and a similar number of additions are required, which makes it about twice as fast as the standard Gaussian elimination.
- The construction requires calculating the square roots of m numbers. If the matrix $\mathbf{A}$ is Hermitian positive definite, these numbers are all positive so the square root is well-defined. On the other hand, if one of these is not positive, then this means the original matrix was not positive definite – this is often used as a test to determine the positive-definiteness of a matrix.

As in the case of LU decomposition, the Cholesky decomposition stores a partially inverted matrix $\mathbf{A}$ (so to speak), so solutions to the problem $\mathbf{Ax} = \mathbf{b}$ can be obtained quickly for different RHS $\mathbf{b}$ as they become known. To do so, we need to solve the problem $\mathbf{LL}^*\mathbf{x} = \mathbf{b}$, which can be decomposed into two steps:

- a forward substitution step of the form $\mathbf{Ly} = \mathbf{b}$, which gives the vector $\mathbf{y}$
- a backsubstitution step of the form $\mathbf{L}^*\mathbf{x} = \mathbf{y}$, which gives the vector $\mathbf{x}$.

Algorithm: Cholesky backsubstitution :

```
![Forward substitution, solving Ly = b
do i = 1 to m
    sum = b_i
    do j = 1 to i − 1
        sum = sum −y_j a_ij
```

```
    enddo
    y_i = sum/a_ii
enddo

![Backward substitution, solving L*x = y
do i = m to 1
    if a_ii = 0 then
        stop
    endif
    do k = i + 1 to m
        y_i = y_i − a*_ki x_k
    enddo
    x_i = y_i/a_ii
enddo
```

Note that this algorithm preserves **b** and returns a separate solution **x**. It is also possible to write it in such a way as to overwrite **b** with the solution on exit.

9. Some notions of stability for numerical linear algebra

Chapters 14 and 15 from the textbook

In this section, we now introduce the subtle but very important concepts of accuracy and stability in numerical linear algebra, where any algorithm put forward needs to be examined in the light of the potential errors arising from floating point arithmetic.

Let's first introduce some notations. An algorithm in linear algebra usually takes a quantity such as scalar, vector or a matrix, denoted X, and applies to it a series of functions or transformations to arrive at another quantity, scalar, vector, or matrix, denoted Y. In theory, this algorithm is exact, but its numerical implementation is not because of rounding errors and floating point arithmetic. Let's call f the exact theoretical algorithm (so $Y = f(X)$), and $\tilde{f}$ the approximate one. Let's also call $\tilde{X}$ the numerically-approximated input value (which may contain roundoff errors), and $\tilde{Y}$ is the numerically-approximated value of the exact solution (assuming the latter is somehow known).

Definition: A round-off error is defined by the difference between an exact and an approximated solutions.

We know that the relative roundoff errors are smaller or equal to machine accuracy, so by definition

$$\frac{||\tilde{X} - X||}{||X||} < \epsilon_{\text{mach}}, \tag{2.84}$$

$$\frac{||\tilde{Y} - Y||}{||Y||} < \epsilon_{\text{mach}} \tag{2.85}$$

However, except for the most trivial algorithms, there is no guarantee that

$$\tilde{f}(\tilde{X}) = \tilde{Y} \tag{2.86}$$

which means that, even though (2.85) is true, there is no guarantee that

$$\frac{||\tilde{f}(\tilde{X}) - f(X)||}{||f(X)||} < \epsilon_{\text{mach}} \tag{2.87}$$

should be. In fact, this is very rarely true! So the question is, can we nevertheless estimate under which circumstances the numerical solution $\tilde{f}(\tilde{X})$ is indeed close to $f(X)$?

Definition: A numerical algorithm $\tilde{f}$ for a problem f is called **accurate** if, for each possible input value X, we have

$$\frac{||\tilde{f}(X) - f(X)||}{||f(X)||} = O(\epsilon_{\text{mach}}) \tag{2.88}$$

for small enough ϵ_{mach}.

The symbol O is a notation that means *of order of*, and has a very strict mathematical definition which is explored in detail in AMS212B and AMS213B. For the purpose of this class, it is sufficient to interpret this to mean that there exists a positive constant C such that

$$\frac{||\tilde{f}(X) - f(X)||}{||f(X)||} < C\epsilon_{\text{mach}} \tag{2.89}$$

for small enough ϵ_{mach} and for all X. Note that if $||f(X)|| \to 0$, this can and should be re-interpreted as $||\tilde{f}(X) - f(X)|| < C\epsilon_{\text{mach}}||f(X)||$. In words, this implies that $\tilde{f}(X) - f(X)$ decays to zero *at least as fast* as $f(X)$ does. Note that nothing in this definition requires C to be of order unity; as we shall see, sometimes C can be very large, so the concept of accuracy takes a different meaning in the strict mathematical sense and in the everyday-life sense.

Accuracy is usually quite difficult to prove directly (though see later for an important theorem on the topic). In fact, algorithms applied to input matrices

X that are ill-conditioned cannot be accurate. For this reason, another more practical concept is that of stability.

Definition: An algorithm $\tilde{f}$ for a problem f is **stable** if for each possible input X,

$$\frac{||\tilde{f}(X) - f(\tilde{X})||}{||f(\tilde{X})||} = O(\epsilon_{\text{mach}}) \tag{2.90}$$

$$\text{for some } \tilde{X} \text{ with } \frac{||\tilde{X} - X||}{||X||} = O(\epsilon_{\text{mach}}) \tag{2.91}$$

As described in the book, such *a stable numerical algorithm gives nearly the right answer for nearly the right input.*

Example: The unpivoted algorithm for Gaussian elimination is a clear example of an **unstable** algorithm. In the case of the input matrix

$$\mathbf{X} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \tag{2.92}$$

there is no nearby value $\mathbf{X}'$ that would let the algorithm return *nearly the right answer.* Indeed, as long as the 0 is there in the diagonal, the algorithm fails because of division by zero, and even if we try to consider instead the matrix

$$\mathbf{X}' = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \tag{2.93}$$

with $\epsilon = O(\epsilon_{\text{mach}})$, to avoid the problem, we saw that the answer will still be $O(1)$ away from the true answer.

Directly proving the stability of an algorithm can be quite difficult but there is another concept that is both stronger (i.e. that implies stability), and often simpler to prove, that of **backward stability.**

Definition: An algorithm $\tilde{f}$ for a problem f is **backward stable** if, for each possible input X,

$$\tilde{f}(X) = f(\tilde{X}) \text{ for some } \tilde{X} \text{ with } \frac{||\tilde{X} - X||}{||X||} = O(\epsilon_{\text{mach}}). \tag{2.94}$$

To paraphrase the textbook again, *a backward stable numerical algorithm gives exactly the right answer for nearly the right input.* Note that backward stability trivially implies stability, but the converse is not true (stable algorithms are not all backward stable). Also note that, because all norms are equivalent

within a factor unity, the definitions of stable and backward stable hold regardless of the norm used – hence we can prove stability or backward stability using whichever one is the easiest to use for any given algorithm.

Backward stability is often easier to demonstrate than stability because it is relatively easy to show using some of the fundamental properties of floating point arithmetic that any of the four basic operations $+$, $-$, $\times$ and $\div$ are all backward stable – namely, given any input vector $(x, y)^T$, the numerical algorithms that compute respectively $x + y$, $x - y$, xy and x/y are backward stable (as long as the quantity is defined). For details, examples, and more, see the textbook. In particular, read the section on the proof of backward stability for the backsubstitution algorithm (Chapter 17), which should illustrate exactly how tricky and detailed stability analysis needs to be in order to be rigorous.

Because stability/instability proofs are so cumbersome, we will never attempt to prove stability in this course, but will merely use well-known results from the literature, namely:

- The backsubstitution algorithm for Gaussian elimination is backward stable.
- Gaussian elimination or LU factorization without pivoting is neither backward stable nor stable.
- Gaussian elimination or LU factorization with partial pivoting is *theoretically* backward stable, but can be prone to extremely large errors for large, poorly conditioned matrices (see Chapter 22 of the textbook and examples).
- Cholesky factorization and backsubstitution are backward stable.

So what can be said about accuracy, which was the original question posed in this section? As it turns out, the accuracy of an algorithm can be estimated as long as the algorithm has already been shown to be backward stable, with the following theorem:

Theorem: Suppose a backward stable algorithm $\tilde{f}$ is applied to solve a problem $f(X) = Y$ with a relative condition number $\kappa(X)$ that is known. Then the relative errors satisfy

$$\frac{||\tilde{f}(X) - f(X)||}{||f(X)||} = O(\kappa(X)\epsilon_{\text{mach}}) \tag{2.95}$$

To prove the theorem, notice that since $\tilde{f}$ is backward stable, $\tilde{f}(X) = f(\tilde{X})$ for some $\tilde{X}$ such that

$$\frac{||\delta X||}{||X||} = O(\epsilon_{\text{mach}}), \tag{2.96}$$

where $\delta X = \tilde{X} - X$. If we let $\delta f = f(\delta X)$, the relative errors can be written as

$$\frac{||\tilde{f}(X) - f(X)||}{||f(X)||} = \frac{||f(\tilde{X}) - f(X)||}{||f(X)||} = \frac{||\delta f||}{||f(X)||} \tag{2.97}$$

as long as Eq.(2.96) is true. Using the definition of $\kappa(X)$, we get

$$\frac{||\delta f||}{||f(X)||} \frac{||X||}{||\delta X||} \leq \kappa(X), \tag{2.98}$$

which yields, together with Eq. (2.96),

$$\frac{||\delta f||}{||f(X)||} \leq \kappa(X) \frac{||\delta X||}{||X||} \leq C\kappa(X)\epsilon_{\text{mach}}, \tag{2.99}$$

for some $C > 0$. This prove the theorem. $\square$

Hence, as long as $\kappa(X)$ is bounded for all X, the theorem guarantees accuracy of the algorithm, and provides upper bounds on the relative error. If $\kappa(X)$ is not globally bounded, the theorem can nevertheless be used to estimate the accuracy of the algorithm for a particular input X. This shows once again that, regardless of the quality of an algorithm, little can be done to get an accurate answer if the original problem itself is ill-conditioned.

Finally, we saw that for both matrix multiplication $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ and for the solution of the problem $\mathbf{A}\mathbf{x} = \mathbf{b}$ (i.e. $f(\mathbf{b}) = \mathbf{A}^{-1}\mathbf{b}$, given $\mathbf{b}$), the relative condition number of the problem was bounded by the condition number of the matrix $\mathbf{A}$. This then implies that the accuracy of a backward stable algorithm can be estimated with

$$\frac{||\tilde{f}(\tilde{X}) - f(X)||}{||f(X)||} \leq O(\kappa(\mathbf{A})\epsilon_{\text{mach}}) \tag{2.100}$$

This bound therefore applies to any matrix multiplication $\mathbf{A}\mathbf{x} = \mathbf{b}$ to given an estimate of the error on $\mathbf{b}$, and to applications of LU factorization and Gaussian elimination for the solution of linear systems, as well as to the use of the Cholesky factorization for the same purpose, to give an estimate of the error on the solution $\mathbf{x}$.