CHAPTER 8

MATRIX I: LINEAR ALGEBRAIC EQUATIONS

So far our focus has been on numerical methods for calculus. Linear algebra is another major mathematical component in physics, where vectors and matrices are main players. It plays an essential role particularly in quantum mechanics. The numerical methods of linear algebra are also used in other numerical methods in later chapters when we solve multivariate root finding/minimization problems, data fitting, and partial differential equations. It is highly desirable to develop efficient and accurate numerical methods for linear algebra. Despite that the problem looks very basic, numerical methods to solve it is not trivial. In this chapter, various numerical methods for solving linear systems are introduced.

The size of the matrix can be very large in real world applications. Writing a code for large matrices is often complicated. Fortunately, standard libraries such as LAPACK[1] are available for most computer languages and we utilize them. However, it is dangerous to use such black-box routines without knowing how they work. In this chapter, basic ideas are introduced using small matrices, mostly 3-by-3. Once we understand the ideas, we can use the black-box routines with confidence and when they fail we will find alternative methods.

In particular, we are interested in linear algebraic equations. We often encounter a set of simultaneous equations like

$$\begin{aligned} 3x - y + 4z &= 2 && \text{(8.1a)} \\ 2x - z &= -1 && \text{(8.1b)} \\ 3y + 2z &= 3 && \text{(8.1c)} \end{aligned}$$

Writing it in a matrix form, the set of equations are expressed in a single equation $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{bmatrix} 3 & -1 & 4 \\ 2 & 0 & -1 \\ 0 & 3 & 2 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}. \tag{8.2}$$

To discuss more general cases, we write a system of linear equation in a matrix form

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & A_{22} & \cdots & A_{2N} \\ & & \ddots & \\ A_{N1} & A_{N2} & \cdots & A_{NN} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} \tag{8.3}$$

or simply

$$A\mathbf{x} = \mathbf{b} \tag{8.4}$$

where A is a N-by-N square matrix, and $\mathbf{x}$ and $\mathbf{b}$ are column vectors of length N. Mathematically speaking, the solution to this equation is as simple as $\mathbf{x} = A^{-1}\mathbf{b}$ where A^{-1} the inverse of A. However, finding the inverse matrix is not a trivial task as you know from the linear algebra course. Fortunately, there are smart numerical methods to solve it even without computing the actual inverse matrix. In MATLAB, simply `x=b\A` solves the problem. We can of course use it and it works in most cases. However, we always need to understand the degree of accuracy and the stability of the numerical method used inside MATLAB.

When we solve Eq. (8.1) by hand a common method is to eliminate variables one by one (the method of variable elimination). Numerical methods essentially do the same. First, we discuss a trivial case. If the matrix A is either upper or lower triangular, *forward substitution* and *back substitution* solve the problem right away. For general cases, we introduce numerical methods known as *Gaussian elimination* and *LU* decomposition, which transform general matrix problems to triangular matrix problems.

8.1 Triangular Matrices

We first discuss a special kind of matrices: lower triangular matrix

$$L = \begin{bmatrix} L_{11} & 0 & 0 & \cdots & 0 \\ L_{21} & L_{22} & 0 & \cdots & 0 \\ L_{31} & L_{32} & L_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ L_{N1} & L_{N2} & L_{N3} & \cdots & L_{NN} \end{bmatrix}, \tag{8.5}$$

and upper triangular matrix

$$U = \begin{bmatrix} U_{11} & U_{12} & U_{13} & \cdots & U_{1N} \\ 0 & U_{22} & U_{23} & \cdots & U_{2N} \\ 0 & 0 & U_{33} & \cdots & U_{3N} \\ 0 & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & U_{NN} \end{bmatrix}. \tag{8.6}$$

The triangular matrices have several nice properties such as

- The product of two same type of triangular matrices is again the same type of a triangular matrix.
- The inverse of a triangular matrix is the same type of triangular matrix as the original one.
- The determinant of a triangular matrix is just a product of all diagonal elements.

EXAMPLE 8.1

To familiarize ourselves with triangular matrices, we verify the above properties numerically. Since we have not learned how to evaluate matrix inverse and determinant, we use MATLAB built-in functions, `inv()` and `det()`. Numerical methods to compute them will be discussed in this chapter. Let us verify the three properties using the following lower triangular matrix,

$$A = \begin{bmatrix} 2 & 0 & 0 \\ -1 & 1 & 0 \\ 3 & 2 & -1 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 4 & 0 \\ -1 & -2 & 3 \end{bmatrix}. \tag{8.7}$$

Program 8.1 computes the product of A and B, inverse and determinant of A. Here is the outputs.

```
Mutilication: A*B
  2   0   0
  1   4   0
  8  10  -3

Inverse of A
     5.0000e-01     -5.5511e-17      5.5511e-17
     5.0000e-01      1.0000e+00      0.0000e+00
     2.5000e+00      2.0000e+00     -1.0000e+00

Products of the diagonal elements = -2
Determinant by MATLAB = -2.000000e+00
```

The product is again a lower triangular matrix. The inverse is not exactly a lower triangular matrix since the upper triangle elements are not exactly zero. They are numerical errors caused mostly by round-off error and practically small enough to be ignored. Finally, the product of the diagonal elements matches to the determinant obtained by the built-in function.

8.1.1 Forward/Back Substitutions

First, we will solve a simple linear equation

$$L\mathbf{x} = \mathbf{b} \tag{8.8}$$

where the matrix L is lower triangular. For simplicity, we consider 3-by-3 matrices but the method will work for any size of matrices.

Writing Eq. (8.8) explicitly, the corresponding equations of the system is

$$L_{11}x_1 = b_1 \tag{8.9a}$$

$$L_{21}x_1 + L_{22}x_2 = b_2 \tag{8.9b}$$

$$L_{31}x_1 + L_{32}x_2 + L_{33}x_3 = b_3 \tag{8.9c}$$

It is trivial to solve this equation. From the first equation, $x_1 = b_1/L_{11}$. Solving the second equation for x_2, we obtain $x_2 = (b_2 - L_{21}x_1)/L_{22} = (b_2 - b_1 L_{21}/L_{11})/L_{22}$. x_3 can be obtained in the same way. For general cases, the solution is given by

$$x_i = \frac{1}{L_{ii}} \left(b_i - \sum_{j=1}^{i-1} L_{ij} x_j \right) \tag{8.10}$$

In order to find x_i, we must know $x_1, x_2, \cdots, x_{i-1}$. In other words, you must evaluate this equation in the forward order from $i = 1$ to N. That is why this method is called *forward substitution.*

Similarly for the upper triangular matrix, $U\mathbf{x} = \mathbf{b}$ can be solved easily by back substitution

$$x_i = \frac{1}{U_{ii}} \left(b_i - \sum_{j=1+1}^{N} U_{ij} x_j \right) \tag{8.11}$$

which must be evaluated backward from $i = N$ to $i = 1$ and thus this method is known as *back substituion.*

EXAMPLE 8.2

We solve the following equation.

$$3x - y + 4z = -1 \tag{8.12a}$$
$$2y - z = -2 \tag{8.12b}$$
$$2z = 4 \tag{8.12c}$$

First, we write it in matrix form

$$\begin{bmatrix} 3 & -1 & 4 \\ 0 & 2 & -1 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -1 \\ -2 \\ 4 \end{bmatrix}. \tag{8.13}$$

Since the matrix is upper triangular, we use the back substitution method. Program 8.2 carries out back substitution and the solution is

```
x=-3.0, y=0.0, z=2.0
```

8.2 Gaussian Elimination

Solving linear equations of triangular matrix is almost trivial. Is there a similar formula for general matrix problems? The answer is NO. The problem is much harder. However, almost any general matrix problem can be transformed to an equivalent triangular matrix problem as long as A is not singular. Since a non-trivial problem becomes a trivial problem, the transformation procedure must be non-trivial (due to the law of the conservation of difficulty). The procedure is actually the same as what we do when we solve the equation manually by hand. That is the method of variable elimination which is commonly known as *Gaussian elimination.*

8.2.1 Elmination Procedures

First, let us solve (8.1) by hand. Diagram (8.14) shows it. First, we eliminate x in the second equation using the first equation. From the first equation $x = \frac{1}{3}(y - 4z + 2)$. Substituting it to the second equation, x in the second equation is eliminated. Next, we do the same for the third equation. This time, we will eliminate y. The third equation contains only z and thus we solve the problem. Notice that the final expression is upper triangle. The second equation can be simplified by multiplying 3 to both sides. However, that will change the properties of the matrix, namely the determinant. So, we keep the rather messy expression.

$$\begin{array}{rcrcr}
3x - y + 4z = 2 & & 3x - y + 4z = 2 & & 3x - y + 4z = 2 \\
2x - z = -1 & \xrightarrow[\text{elimination}]{\text{1 st}} & \frac{2}{3}y - \frac{11}{3}z = -\frac{7}{3} & \xrightarrow[\text{elimination}]{\text{2 nd}} & \frac{2}{3}y - \frac{11}{3}z = -\frac{7}{3} \\
3y + 2z = 3 & & 3y + 2z = 3 & & \frac{37}{2}z = \frac{27}{2}
\end{array} \quad \begin{array}{r}(8.14a)\\(8.14b)\\(8.14c)\end{array}$$

Now we write this procedure in a matrix form,

$$A\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad M^{(1)}A\mathbf{x} = M^{(1)}\mathbf{b} \quad \Rightarrow \quad M^{(2)}M^{(1)}A\mathbf{x} = M^{(2)}M^{(1)}\mathbf{b}. \tag{8.15}$$

where the transformation matrix $M^{(i)}$ applies the i-th step of the forward Gaussian elimination. For the above example, the transformation matrices are

$$M^{(1)} = \begin{bmatrix} 1 & 0 & 0 \\ -2/3 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad M^{(2)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -9/2 & 1 \end{bmatrix}. \tag{8.16}$$

Notice that these matrices are lower triangular matrices and only one element differs from the identity matrix.

In general, the product of $N-1$ transformation matrices transforms a general linear equation to an upper triangular equation.

$$M^{(N-1)}M^{(N-2)}\cdots M^{(2)}M^{(1)}A\mathbf{x} = M^{(N-1)}M^{(N-2)}\cdots M^{(2)}M^{(1)}\mathbf{b} \longrightarrow U\mathbf{x} = \mathbf{b}' \tag{8.17}$$

Note that the transformation matrix is applied to the both sides of the equation. In other words, we are modifying $\mathbf{b}$ as well as A, which is a weak point of the Gaussian elimination method which we will discuss later. Algorithm 9.1 shows the summary of the Gaussian forward elimination procedure.

Algorithm 8.1 Gaussian forward elimination

1. Consider a recursive equation $A^{(n+1)} = M^{(n)}A^{(n)}$ and $\mathbf{b}^{(n+1)} = M^{(n)}\mathbf{b}^{(n)}$, starting with the original equation $A^{(1)}\mathbf{x} = \mathbf{b}^{(1)}$ where $A^{(1)}$ is a N-by-N matrix.
2. $M^{(n)}$ is the same as the identity matrix except for the n-th column, $M^{(n)}_{kn} = -A^{(n)}_{kn}/A^{(n)}_{nn}$ where $k = n+1, \cdots, N$.
3. Apply the transformation to both $A^{(n)}$ and $\mathbf{b}^{(n)}$. Note that the transformation affect only the rows from $n+1$ to N of $A^{(n)}$ and $\mathbf{b}^{(n)}$.
4. Increment n and repeat from step 2 until $n = N$.

Although the method is simple and works fine for many cases, it fails when the matrix is close to singular. There are better methods. We introduced the Gaussian elimination method for a pedagogical purpose since similar ideas are used in other methods. More practical methods will be discussed later.

EXAMPLE 8.3

We solve Eq. (8.1) using the Gaussian elimination followed by the back substitution. Program 8.3 implements Algorithm 9.1. The following output shows the linear equation after Gaussian elimination is applied. The matrix A is transformed to a upper triangular form and b is also transformed accordingly. Then, the solution x is obtained from the transformed equation by the back substitution, which is in agreement with the exact solution $x = -\frac{5}{37}$, $y = \frac{19}{37}$, $z = \frac{27}{37}$.

```
A=
 3.00000  -1.00000   4.00000
 0.00000   0.66667  -3.66667
 0.00000   0.00000  18.50000

b=
 2.00000
-2.33333
13.50000

x=
-0.13514
 0.51351
 0.72973
```

8.2.2 Pivoting

The Gaussian elimination method suffers from round-off errors, sometimes severely. To see the source of the error, apply the Gaussian elimination to the following problem:

$$\begin{aligned} \epsilon x + y + z &= 1 \\ x + y &= 2 \\ x + z &= 3 \end{aligned} \quad \xrightarrow[\text{elimination}]{\text{1st}} \quad \begin{aligned} \epsilon x + y + z &= 1 \\ \left(1 - \frac{1}{\epsilon}\right) y - \frac{1}{\epsilon} z &= 2 - \frac{1}{\epsilon} \\ -\frac{1}{\epsilon} y + \left(1 - \frac{1}{\epsilon}\right) z &= 3 - \frac{1}{\epsilon} \end{aligned} \quad \xrightarrow[\text{round-off}]{\epsilon \to 0} \quad \begin{aligned} \epsilon x + y + z &= 1 \\ -\frac{1}{\epsilon} y - \frac{1}{\epsilon} z &= -\frac{1}{\epsilon} \\ -\frac{1}{\epsilon} y - \frac{1}{\epsilon} z &= -\frac{1}{\epsilon} \end{aligned}$$

where $\epsilon \ll 1$. Using the first equation, we eliminate x from the second and third equations. As $\epsilon \to 0$, $\frac{1}{\epsilon}$ becomes so large that computers cannot distinguish $1 - \frac{1}{\epsilon}$ and $-\frac{1}{\epsilon}$ due to round-off. Now the second and third equations are identical and thus there is no unique solution. On the other hand, when $\epsilon = 0$, the solution does exist and it is $x = 2, y = 0, z = 1$. This example demonstrates the failure of the Gaussian elimination.

Fortunately, there is a way to avoid such errors. We did not have to use the first equation to eliminate x. Instead, use the second equation to eliminate x in two other equations. After swapping the first and second rows, we apply the regular Gaussian elimination.

$$\begin{array}{r} x+y=2 \\ \epsilon x+y+z=1 \\ x+z=3 \end{array} \xrightarrow[\text{elimination}]{\text{1st}} \begin{array}{c} x+y=2 \\ (1-\epsilon)y+z=1-2\epsilon \\ -y+z=1 \end{array} \xrightarrow[\text{elimination}]{\text{2nd}} \begin{array}{c} x+y=2 \\ (1-\epsilon)y+z=1-2\epsilon \\ \dfrac{2-\epsilon}{1-\epsilon}z=\dfrac{2-3\epsilon}{1-\epsilon} \end{array}$$

$$\xrightarrow[\text{round-off}]{\epsilon\to 0} \begin{array}{rcl} x+y &=& 2 \\ y+z &=& 1 \\ 2z &=& 2 \end{array}$$

After the first step, we don't see any extreme value. Now, we eliminate y using the second equation. If the new coefficient to y happened to be very small, we need to swap the second and third equations to avoid the round-off error. Since the coefficient to y is not small in this example, we don't need to worry about it. We now go ahead and eliminate y. The final expression takes an upper triangular form. When $\epsilon = 0$, we obtain the correct solution by the back substitution.

This algorithm of avoiding the round-off errors by rearranging the equations is known as pivoting. The above example swapped two rows. This is known as partial pivoting. In some cases, interchanging both rows and columns may be needed to achieve a desired accuracy. This is known as complete pivoting. We must recall that when the matrix A is singular (the determinant of A is zero) Eq. (8.4) does not have a unique solution. When A is near singular (the determinant is close to zero) the Gaussian elimination in general fails even with pivoting. Then, we must resort to other method such as singular value decomposition (SVD)[2].

Algorithm 8.2 summarizes the so-called scaled partial pivoting method. The basic idea is that when we eliminate a variable x_n, we look for the row which has the largest coefficient to x_n (the pivot element). However, the absolute magnitude of the coefficients does not have significant meaning since each row can be scaled by multiplying a constant without changing the solution. So, we normalize each row by the largest coefficient in the row.

Algorithm 8.2 Scaled Partial Pivoting

1. Find a scale factor for each row. $S_i = \max_j(|A_{ij}|)$.
2. Staring with $n = 1$, repeat the following procedure up to $n = N - 1$.
3. Assume that $n - 1$ variables are already eliminated and the first $n - 1$ rows are already upper triangular. The remaining submatrix is still not triangular. (See Fig. 8.1.) Now, we eliminate x_n.
4. Find a row $j \geq n$ such that $|A_{jn}|/S_j \geq |A_{kn}|/S_k, \quad (\forall k \geq n)$. This is the pivot row.
5. Move j-th row to the top of the submatrix. (Pivoting)
6. Apply the forward elimination to the row below the pivot row. After that, we have a new A and b.
7. If $n = N - 1$, the elimination is completed. Otherwise, increment n and go to step 3.

EXAMPLE 8.4

We solve Eq. (8.1) again but using partial pivoting this time. Program 8.4 will do it. The permutation matrix P indicates which rows are swapped. The results show that the first elimination swapped the first and second rows. Then, the second elimination interchanged the second and third rows. The final triangular form of A is totally different from the one obtained without pivoting (see example 8.3). However, the final solution $\mathbf{x}$ agrees with the previous example. Note that the final triangular matrix in Example 8.3 contains a large element 18.5. However, the pivoting avoided the appearance of such a large element.

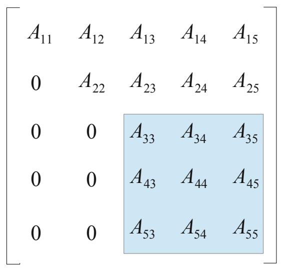


Figure 8.1: After two steps of forward elimination, 3-by-3 submatrix remains non-triangular. To find the next pivot, find the maximum of A_{33}/S_3, A_{43}/S_4, and A_{53}/S_5. The row carrying the maximum goes to the top of the submatrix.

```
A=
 2.00000   0.00000  -1.00000
 0.00000   3.00000   2.00000
 0.00000   0.00000   6.16667

b=
-1.00000
 3.00000
 4.50000

P=
0  1  0
1  0  1
0  1  0

x=
-0.13514
 0.51351
 0.72973
```

8.2.3 Determinant

The determinant of a triangular matrix is simple. Just the product of the all diagonal is the determinant. For example, an upper triangular matrix U has the determinant:

$$|U| = \prod_{i}^{N} U_{ii}. \tag{8.20}$$

Now, we show that the Gaussian elimination preserves the determinant. Using the transform matrices defined in Eq. (8.17), the final upper triangular form U is given by

$$U = M^{(N-1)} M^{(N-2)} \cdots M^{(2)} M^{(1)} A \tag{8.21}$$

and its determinant

$$|U| = |M^{(N-1)}M^{(N-2)}\cdots M^{(2)}M^{(1)}A| = |M^{(N-1)}||M^{(N-2)}|\cdots|M^{(2)}||M^{(1)}||A| \tag{8.22}$$

Noting that $M^{(n)}$ is a lower triangular matrix with a unit diagonal, its determinant is 1. Hence, $|U| = |A|$. When pivoting is used,

$$|U| = |M^{(N-1)}||P^{(N-1)}||M^{(N-2)}||P^{(N-2)}|\cdots|M^{(1)}||P^{(1)}||A| = (-1)^p|A| \tag{8.23}$$

where p is the number of pivoting. The proof is simple. If two rows are swapped $|P^{(n)}| = -1$ and otherwise it is $|P^{(n)}| = +1$.

EXAMPLE 8.5

We calculate the determinant of the matrix in Eq. (8.2). By using the rule of Sarrus, its determinant is 37. Now, look at Eq. (8.14). The corresponding matrix take a upper triangular form:

$$\begin{bmatrix} 3 & -1 & 4 \\ 0 & 2/3 & -11/3 \\ 0 & 0 & 37/2 \end{bmatrix} \tag{8.24}$$

The product of the all diagonal elements is $3 \times 2/3 \times 37/2 = 37$ which is the determinant.

8.2.4 Matrix Inversion

The Gaussian elimination cleverly solves equation $A\mathbf{x} = \mathbf{b}$ without deriving A^{-1}. However, since we are able to calculate $\mathbf{x}$, there must be a way to find A^{-1}. Indeed, the Gaussian elimination method can be used to get the inverse. Consider N sets of the linear equations with unit vectors as $\mathbf{b}$.

$$\begin{bmatrix} & & \\ & A_{ij} & \\ & & \end{bmatrix}\begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} & & \\ & A_{ij} & \\ & & \end{bmatrix}\begin{bmatrix} x_{12} \\ x_{22} \\ x_{32} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} & & \\ & A_{ij} & \\ & & \end{bmatrix}\begin{bmatrix} x_{13} \\ x_{23} \\ x_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \tag{8.25}$$

Each equation can be solved by the Gaussian elimination. Therefore, we have x_{ij}. Now, we can write the set of equations in a single matrix equation

$$\begin{bmatrix} & & \\ & A_{ij} & \\ & & \end{bmatrix}\begin{bmatrix} & & \\ & x_{ij} & \\ & & \end{bmatrix} = \begin{bmatrix} & & \\ & I_{ij} & \\ & & \end{bmatrix} \tag{8.26}$$

where I_{ij} is an identity matrix. Therefore, the matrix x is the inverse of A. By performing Gaussian elimination N times, we can find the inverse of a matrix. This method is known as Gauss-Jordan elimination. Since this is based on the Gaussian elimination method, it may suffer from round-off error. Other methods are usually used in practical applications. However, it is very useful to know the basic idea of Gauss-Jordan method in order to develop other methods.

EXAMPLE 8.6

We calculate the inverse of matrix A in Eq (8.2)) using the Gaussian elimination. It is trivial to modify the code in Example 8.4. Program 8.5 calculates the inverse and check the answer by calculating $A\,A^{-1}$. The output show that we recover the identity matrix and thus the inverse is accurate.

```
Invers of A=
 0.08108  -0.10811   0.16216
 0.37838   0.16216  -0.24324
 0.02703   0.29730   0.05405

A A^(-1)=
 1.00000   0.00000   0.00000
 0.00000   1.00000  -0.00000
-0.00000   0.00000   1.00000
```

8.3 LU Decomposition

While it is simple, the Gaussian elimination has various weakness. We reduced the chance of round-off error by pivoting. Another issue arises when we want to solve the equation many times with the same A but different $\mathbf{b}$. We have to carry out the elimination for every different $\mathbf{b}$ even with the same A. For a large system, that is annoying. Fortunately, there are better ways. LU decomposition[2] is one of them.

8.3.1 Decomposition Algorithm

Looking at Eq. (8.21) again. $U = MA$ where $M = M^{(N-1)}M^{(N-2)}\cdots M^{(1)}$. Recall that U is upper triangular and each $M^{(n)}$ is lower triangular. Now, using the properties of triangular matrices: (1) the product of triangular matrices is again the same kind of triangular matrix. (2) the inverse of a triangular matrix is again the same kind of triangular matrix. Hence, M is lower triangular and so is M^{-1}. Let $L = M^{-1}$, we conclude that

$$A = LU \tag{8.27}$$

which is called LU decomposition or LU factorization of A. If pivoting is used, the rows are shuffled. Therefore,

$$PA = LU \tag{8.28}$$

where P is a permutation matrix. Using the property of permutation, namely $P^{-1} = P$, we obtain a more popular expression

$$A = PLU \tag{8.29}$$

This is just another way to express the Gaussian elimination. However, this decomposition does not depend of the right hand side $\mathbf{b}$. You need to carry out the decomposition only once for A. This saves computer time significantly if $Ax = b$ has to be solved many times with different $\mathbf{b}$.

MATLAB has a built-in function `lu()` to compute LU decomposition (See Example 8.7.) For other languages, LAPACK includes LU decomposition routines. Actually MATLAB internally calls LAPACK routines. Be reminded that any numerical algorithm has weakness. A blind use of "canned" routines is dangerous. We should carefully check possible pitfalls whenever we use canned routines.

8.3.2 Linear equations

Now, we solve $A\mathbf{x} = \mathbf{b}$ using the LU decomposition. The equation is now $PLU\mathbf{x} = \mathbf{b}$ or equivalently $LU\mathbf{x} = P\mathbf{b}$, which can be divided to two equations, $L\mathbf{y} = P\mathbf{b}$ and $U\mathbf{x} = \mathbf{y}$ where $\mathbf{y}$ is an auxiliary vector. The former equation can be solved for $\mathbf{y}$ easily by forward substitution. Then, solve the latter for $\mathbf{x}$ with back substitution. Once L and U are computed for A, we can use them for different $\mathbf{b}$ with the same L and U. That is a huge advantage.

EXAMPLE 8.7

Here is another attempt to solve Eq. (8.1). This time we use LU decomposition (built-in function in MATLAB). The permutation matrix indicated that the second and third rows are swapped during the decomposition procedure. The product PLU recovers the original A. The results agree perfectly with the analytic answers.

```
L (Lower Triangular Matrix)
 1.00000   0.00000   0.00000
 0.00000   1.00000   0.00000
 0.66667   0.22222   1.00000

U (Upper Triangular Matrix)
 3.00000  -1.00000   4.00000
 0.00000   3.00000   2.00000
 0.00000   0.00000  -4.11111

P (Permutation Matrix)
1   0   0
0   0   1
0   1   0

P*L*U
 3.00000  -1.00000   4.00000
 2.00000   0.00000  -1.00000
 0.00000   3.00000   2.00000

x=-0.13514,  y=0.51351, z=0.72973
```

8.3.3 Matrix Inverse

It is straight forward to find the inverse of a matrix using LU decomposition. The idea is exactly the same as Gauss-Jordin elimination. Substituting $A = PLU$ and using $P^{-1} = P$, Eq. (8.26) becomes

$$\begin{bmatrix} \\ (LU)_{ij} \\ \\ \end{bmatrix} \begin{bmatrix} \\ x_{ij} \\ \\ \end{bmatrix} = \begin{bmatrix} \\ P_{ij} \\ \\ \end{bmatrix} \tag{8.30}$$

If Gaussian elimination is used, we have to repeat the elimination N times for a $N \times N$ matrix. With LU decomposition, we use the elimination only once and we need to repeat only forward/back substitution.

8.3.4 Determinant

Calculation of the determinant is also straight forward. $|A| = |P\,L\,U| = |P|\,|U|\,|L| = (-1)^p |U|\,|L|$. Here we used $|P| = (-1)^p$ where p is the number of pivoting (the number of permutations). The determinant of U and L are just the product of all diagonal elements. Thus,

$$|A| = (-1)^p \prod_{i=1}^{N} U_{ii}\, L_{ii} \tag{8.31}$$

EXAMPLE 8.8

Using the L and U obtained in Example 8.7, we find the determinant $|A| = |PLU| = (-1)*3*3*(-4.11111) = 36.99999$ which is in agreement with exact value 37.

8.4 Tridiagonal Matrices

A tridiagonal matrix is defined by a sparse matrix

$$\begin{bmatrix} d_1 & u_1 & & & 0 \\ \ell_2 & d_2 & u_2 & & \\ & \ell_3 & d_3 & \ddots & \\ & & \ddots & \ddots & u_{N-1} \\ 0 & & & \ell_N & d_N \end{bmatrix} \tag{8.32}$$

which is a popular matrix expression of one-dimensional Laplace operator. In Chapter 2, we evaluated the second order derivative of a function $f(x)$ at a single point x. Suppose that we want evaluate the second order derivative at all points on a grid $x_i = x_0 + ih, \quad i = 1, \cdots, N$ where h is a step length. Using the standard method (2.13),

$$f''(x_i) = \frac{f(x_{i+1}) + f(x_{i-1}) - 2f(x_i)}{h^2}, \qquad i = 1, \cdots, N \tag{8.33}$$

where we assume that $x_0 = x_{N+1} = 0$. We can express it simultaneously for all points in a matrix form.

$$\begin{bmatrix} f_1'' \\ f_2'' \\ f_2'' \\ \vdots \\ f_N'' \end{bmatrix} = \frac{1}{h^2} \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & -2 & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -2 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_N \end{bmatrix} \tag{8.34}$$

where $f_i = f(x_i)$ and $f_i'' = f''(x_i)$. This indicates that the second-order derivative is an tridiagonal matrix acting on a column vector $\mathbf{f}$.

8.4.1 Linear Equations

If A in the linear equations eq8.4 is tridiagonal, the Gaussian elimination becomes rather simple. Since the most of matrix elements are zero, the use of regular Gaussian elimination programs is not efficient. We can actually write down the elimination process explicitly. Here is the backward elimination procedure:

$$\xi_{N-1} = \frac{-\ell_N}{d_N}, \quad \xi_{i-1} = \frac{-\ell_i}{d_i + u_i \xi_i}, \quad i = N-1, \cdots, 2 \tag{8.35}$$

$$\zeta_{N-1} = \frac{b_N}{d_N}, \quad \zeta_{i-1} = \frac{b_i - u_i \zeta_i}{d_i + u_i \xi_i}, \quad i = N-1, \cdots, 2 \tag{8.36}$$

If the denominator is close to zero, pivoting is necessary. Now, the equation is lower triangular and the solution is obtained by forward substitution:

$$x_1 = \frac{b_1 - u_1 \zeta_1}{d_1 + u_1 \xi_1}, \quad x_{i+1} = \xi_i x_i + \zeta_i, \quad i = 1, \cdots, N-1. \tag{8.37}$$

8.4.2 Determinant and Inverse

We can also write down the explicit procedure for determinant and inverse. The following recursive equation

$$D_n = d_n D_{n-1} - \ell_{n-1} u_{n-1} D_{n-2}, \qquad D_0 = 1 \text{ and } D_{-1} = 0 \tag{8.38}$$

converges to the determinant D_N.

For the inverse of tridiagonal matrix, first we compute the following recursive equations,

$$\eta_n = \ell_n \eta_{n-1} - d_{n-1} u_{n-1} \eta_{n-2}, \quad n = 1, 2, \cdots, N \tag{8.39}$$

starting with $\eta_0 = 1$, $\eta_{-1} = 0$ and compute

$$\xi_n = a_n \xi_{n+1} - d_n u_n \xi_{n+2}, \quad n = N, N-1, \cdots, 1 \tag{8.40}$$

backward starting with $\xi_{N+1} = 1$ and $\xi_{N+2} = 0$. Then, the elements of the inverse matrix is given by

$$(A^{-1})_{ij} = \begin{cases} (-1)^{i+j} d_i \cdots d_{j-1} \eta_{i-1} \xi_{j+1} / \eta_N & i \le j \\ (-1)^{i+j} u_j \cdots u_{j-1} \eta_{j-1} \xi_{i+1} / \eta_N & i > j \end{cases} \tag{8.41}$$

EXAMPLE 8.9

We want to solve the following equation.

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 2 & 1 & 2 & 0 \\ 0 & 3 & 1 & 3 \\ 0 & 0 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 1 \\ 3 \end{bmatrix} \tag{8.42}$$

Program 8.7 first checks if this is not a singular problem by computing the determinant. If the determinant is not zero, it solves the equation and check the numerical errors. Since the determinant is rather large compared with the matrix elements, it is safe to ignore pivoting. The error of all solutions is quite small.

```
Determinant -18
x= 1.000000 -0.500000 0.500000 0.833333
Eerror= 4.440892e-16 4.440892e-16 0.000000e+00 0.000000e+00
```

8.5 Solving Linear Equations by Minimization

The methods discussed above are strictly for the linear equations (8.4). There are quite different approaches to solve the same linear equations. Although they are not very efficient for linear problems, they can be extended to non-linear equations. Therefore, we introduce them here for the pedagogical purpose.

Consider a multivariate function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}} A\,\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{x} \tag{8.43}$$

where A is a $N \times N$ positive definite symmetric matrix and $\mathbf{b}$ and $\mathbf{x}$ are vectors of N dimension. The superscript T represents transpose. Recall that $\mathbf{a}^{\mathrm{T}}\mathbf{b}$ is inner product between $\mathbf{a}$ and $\mathbf{b}$. The function has a unique minimum at $\mathbf{x}$ determined by

$$\nabla f(\mathbf{x}) = A\mathbf{x} - \mathbf{b} = 0 \tag{8.44}$$

which is nothing but a linear equation.

Let us take it inversely. If we want to solve the linear equation (8.44), we just minimize Eq. (8.43) with respect to $\mathbf{x}$. If A is not symmetric nor positive definite, minimize the following function:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}} A^{\mathrm{T}} A\,\mathbf{x} - A^{\mathrm{T}}\mathbf{b}^{\mathrm{T}}\mathbf{x} \tag{8.45}$$

This expression is essentially the same as Eq. (8.43) since it has a positive definite symmetric matrix $A^{\mathrm{T}}A$ and constant vector $A^{\mathrm{T}}\mathbf{b}$. Furthermore, this function has a minimum at

$$A^{\mathrm{T}}(A\mathbf{x} - \mathbf{b}) = 0 \tag{8.46}$$

which is equivalent to Eq. (8.4) since A is not singular.

There are many ways to minimize such a function. In the following we will discuss the *steepest descent* and *conjugate gradient* methods.

8.5.1 Steepest Descent Method

To minimize the function value starting from an initial point $\mathbf{x}_0$, we need to find the direction in which the function value decreases. Recalling that the gradient of the function gives the direction of the highest slope,

$$\mathbf{g}_0 \equiv -\nabla f(\mathbf{x}_0) = \mathbf{b} - A\mathbf{x}_0 \tag{8.47}$$

provides the direction of the steepest descent. Do not miss the minus sign in front of the nabla operator. Now, we move down the slope along the line specified by the steepest descent until we hit the bottom along the line. This process is called *line minimization*. The new point is written as

$$\mathbf{x}_1 = \mathbf{x}_0 + \lambda\mathbf{g}_0. \tag{8.48}$$

where λ is a constant to be determined. The new gradient $\mathbf{g}_1$ at $\mathbf{x}_1$ must be orthogonal to the previous gradient because we are already at the minimum in the direction of $\mathbf{g}_0$. Hence,

$$\mathbf{g}_1^{\mathrm{T}}\mathbf{g}_0 = (\mathbf{b}^{\mathrm{T}} - \mathbf{x}_1^{\mathrm{T}}A^{\mathrm{T}})\mathbf{g}_0 = [\mathbf{b}^{\mathrm{T}} - (\mathbf{x}_0^{\mathrm{T}} + \lambda\mathbf{g}_0^{\mathrm{T}})A^{\mathrm{T}}]\mathbf{g}_0 = (\mathbf{g}_0^{\mathrm{T}} - \lambda\mathbf{g}_0^{\mathrm{T}}A^{\mathrm{T}})\mathbf{g}_0 = 0 \tag{8.49}$$

Solving this equation for λ, we find

$$\lambda = \frac{\mathbf{g}_0^{\mathrm{T}}\mathbf{g}_0}{\mathbf{g}_0^{\mathrm{T}} A \mathbf{g}_0} \tag{8.50}$$

where we used the symmetric property $A^{\mathrm{T}} = A$. Now the line minimization is completed. The new point $\mathbf{x}_1$ is just a minimum on the line and not the minimum of the function yet. However, if the procedure is repeated with $\mathbf{x}_1$ as a new stating point, the function value keeps decreasing and reaches the global minimum of the function within a tolerance. The summary of the procedure is gicen in Algorithm 8.3.

Algorithm 8.3 Steepest Descent Minimization

```
1. Starting with $n = 0$, repeat the following recursive process.
2. Evaluate the gradient vector $\mathbf{g}_n = \mathbf{b} - A\mathbf{x}_n$.
3. If $|\mathbf{g}_n| <$ tolerance, $\mathbf{x}_n$ is the solution. Otherwise continue.
4. Calculate the step length $\lambda = \dfrac{\mathbf{g}_n^{\mathrm{T}}\mathbf{g}_n}{\mathbf{g}_n^{\mathrm{T}} A \mathbf{g}_n}$.
5. Jump to a new point $\mathbf{x}_{n+1} = \mathbf{x}_n + \lambda \mathbf{g}_n$.
6. Increment $n$ and go to step 2.
```

EXAMPLE 8.10

We solve a simple two-dimensional problem of $A\mathbf{x} = \mathbf{b}$ where

$$A = \begin{bmatrix} 4 & 1 \\ 1 & 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{8.51}$$

Note that A is symmetric. The analytic solution is $x_1 = 1/11$ and $x_2 = 7/11$. We solve this problem iteratively using the steepest descent minimization. The corresponding function to be minimized is

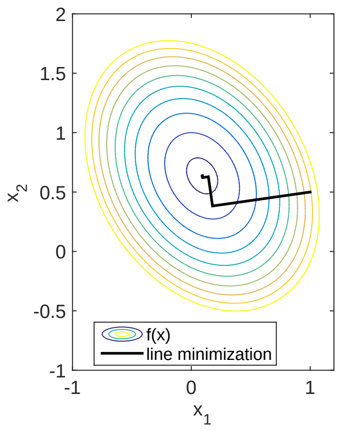
$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}} A \mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{b} \tag{8.52}$$

Program 8.9 minimizes it with the steepest descent method. The contour plot in Fig. 8.2(a) shows the function near the minimum. The trajectory of the steepest descent plotted in Fig 8.2(a) shows that after a few line minimization, it is already very close to the minimum. In fact, it took only nine steps even with a small tolerance 1×10^{-8}. The result agrees well with the analytic solution.

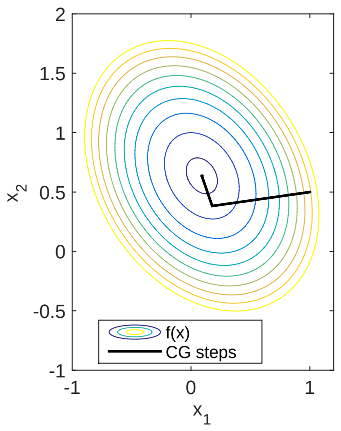
```
Solution=(0.09092,0.63636)
```

8.5.2 Conjugate Gradient Method

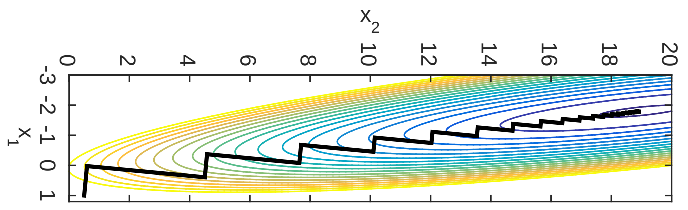
The steepest descent method becomes inefficient when the trajectory is trapped in a narrow valley as shown in Fig. 8.2(c). For a quadratic system (8.43), there is an algorithm called *conjugate gradient* method[3] which find the solution



(a) The steepest descent minimization of the 2D quadratic system (Example 8.10). Fore steps are visible. More steps (not visible in the plot) are needed to get sufficient accuracy.



(b) The conjugate gradient minimization of the 2D quadratic system (Example 8.11. By construction, it needs only two steps to find the solution.



(c) When the function has a narrow valley like this case, the steepest descent method takes a long zig-zag path, making it very inefficient.

Figure 8.2: Illustration of steepest descent/conjugate gradient methods.

in exactly N iterations for N dimensional quadratic system. We again assume that A is symmetric and positive definite. The conjugate gradient method utilizes the geometry of quadratic system as summarized in Algorithm 8.4.

Algorithm 8.4 Conjugate Gradient Method

```
1. Start with an initial guess $\mathbf{x}_0$ and $n = 0$.
2. Set the initial residual vector: $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
3. Set the initial conjugate direction: $\mathbf{p}_0 = \mathbf{r}_0$.
4. Evaluate the step size: $\alpha = \frac{\mathbf{r}_n^T \mathbf{r}_n}{\mathbf{p}_n^T A \mathbf{p}_n}$.
5. Update the point: $\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha \mathbf{p}_n$.
6. Update the residual vector: $\mathbf{r}_{n+1} = \mathbf{r}_n - \alpha A \mathbf{p}_n$.
7. If $|\mathbf{r}_{n+1}| <$ tolerance, $\mathbf{x}_{n+1}$ is the solution. Otherwise continue.
8. Evaluate the other step size: $\beta = \frac{\mathbf{r}_{n+1}^T \mathbf{r}_{n+1}}{\mathbf{r}_n^T \mathbf{r}_n}$.
9. Update the conjugate vector: $\mathbf{p}_{n+1} = \mathbf{r}_{n+1} + \beta \mathbf{p}_n$.
10. Increment $n$ and go to step 4.
```

EXAMPLE 8.11

We solve the same problem as Example 8.10 again but with the conjugate gradient method. Program 8.10 implements the above Algorithm and solve the problem. Figure 8.2(b) shows that two steps of line minimization hits the minimum as expected. The solution is in a good agreement with the exact one.

```
Solution=(0.09089,0.63638)
```

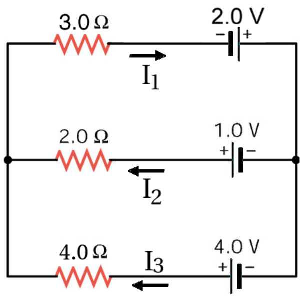
8.6 Applications in Physics

8.6.1 Multiloop circute: Kirchhoff rules

Find currents I_i, $i = 1, 2, 3$ in the circuit shown in Figure.

Applying the Kirchhoff rules, we find a set of equations for the currents

$$\begin{aligned} I_1 - I_2 - I_3 &= 0 \\ 3I_1 + 2I_2 &= 3 \\ -2I_2 + 4I_3 &= 3 \end{aligned}$$



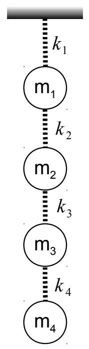
or in a matrix form $A\mathbf{I} = \mathbf{b}$ with

$$A = \begin{bmatrix} 1 & -1 & -1 \\ 3 & 2 & 0 \\ 0 & -2 & 4 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 0 \\ 3 \\ 3 \end{bmatrix}$$

The code used in Example 8.4 can be used. The answer is $I_1 = 0.92308$ A, $I_2 = 0.11538$ A, $I_3 = 0.80769$ A.

8.6.2 Coupled Harmonic Oscillators in a Uniform Gravity

Four particles of mas m_i $(i = 1, \cdots, 4)$ are linked by four springs and the whole system is hanged from the ceiling as shown in Figure. The natural length of the springs is $\ell_1 = \ell_3 = 0.1\,m$, $\ell_2 = \ell_4 = 0.2$ and their spring constants are $k_1 = k_4 = 100\,N/m$ and $k_2 = k_3 = 150\,N/m$. The mass of particles is $m_1 = m_2 = 0.15\,kg$, $m_3 = m_4 = 0.30\,kg$ We want to know the distance between particles when the system is at a mechanical equilibrium. The force on each particles are given by



$$\begin{aligned} F_1 &= k_1 x_1 - k_2 x_2 - m_1 g & (8.53a) \\ F_2 &= k_2 x_2 - k_3 x_3 - m_2 g & (8.53b) \\ F_3 &= k_3 x_3 - k_4 x_4 - m_3 g & (8.53c) \\ F_4 &= k_4 x_4 - m_2 g\,. & (8.53d) \end{aligned}$$

where x_i is the stretch of each spring from its natural length. At the mechanical equilibrium, the force on each particle must vanish. Hence, $F_i = 0$ for all i. Writing it in matrix form $A\mathbf{x} = \mathbf{b}$,

$$A = \begin{bmatrix} k_1 & -k_2 & 0 & 0 \\ 0 & k_2 & -k_3 & 0 \\ 0 & 0 & k_3 & -k_4 \\ 0 & 0 & 0 & k4 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} m_1 g \\ m_2 g \\ m_3 g \\ m_4 g \end{bmatrix} \tag{8.54}$$

The matrix is already upper triangular and thus we can solve it immediately using back substitution. Taking into account the natural length, the distance between particles $i-1$ and i is $d_{i-1,i} = \ell_i + x_i$. We can use Program 8.1 to solve this problem. The answer is $d_{12} = 0.249000$, $d_{23} = 0.139200$, $d_{34} = 0.229400$.

8.6.3 Determinant of Tree Graphs: Graham-Pollack theorem

Consider a graph shown in Fig 8.3. This graph consist of $n = 10$ vertices and 9 edges. When there is no loop in it, the graph is called tree. A tree of n vertices has $n-1$ edges. We consider distance between vertices. The distance between vertices v_2 and v_9 is 3 since there are three edges between them. The distance of between v_i and v_j forms a

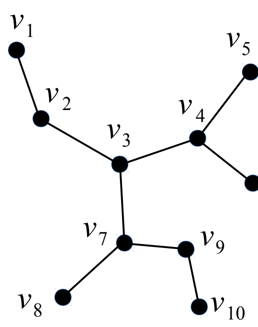


Figure 8.3: A small example of tree graph. It has 10 vertices and 9 edges.However, there is no loop.

distance matrix D_{ij}. The example graph shown in Fig 8.3 has a distance matrix

$$D = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 4 & 3 & 4 & 4 & 5 \\ 1 & 0 & 1 & 2 & 3 & 3 & 2 & 3 & 3 & 4 \\ 2 & 1 & 0 & 1 & 2 & 2 & 1 & 2 & 2 & 3 \\ 3 & 2 & 1 & 0 & 1 & 1 & 2 & 3 & 3 & 4 \\ 4 & 3 & 2 & 1 & 0 & 2 & 3 & 4 & 4 & 5 \\ 4 & 3 & 2 & 1 & 2 & 0 & 3 & 4 & 4 & 5 \\ 3 & 2 & 1 & 2 & 3 & 3 & 0 & 1 & 1 & 2 \\ 4 & 3 & 2 & 3 & 4 & 4 & 1 & 0 & 2 & 3 \\ 4 & 3 & 2 & 3 & 4 & 4 & 1 & 2 & 0 & 1 \\ 5 & 4 & 3 & 4 & 5 & 5 & 2 & 3 & 1 & 0 \end{bmatrix} \tag{8.55}$$

In 1971, Graham and Pollak obtained a remarkable formula[4, 5]

$$\det(D) = -(n-1)(-2)^{n-2} \tag{8.56}$$

which is independent of the structure of the tree. For the above distance matrix, $\det(D) = -9 \cdot (-2)^8 = -2304$. We now check it by numerical calculation. Program 8.8 calculates the determinant of the distance matrix using Gaussian elimination with partial pivoting. The output is

```
Gaussian Elimination: -2304.000000
       Graham-Pollak: -2304.000000
```

The formula works!

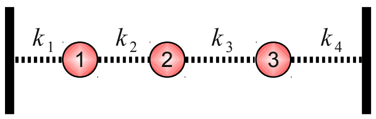
8.7 Problems

8.1 Orthogoinal matrices

A rotation matrix R rotates points in a coordinate space. Accordingly, any vector $\mathbf{a}$ is rotated as $\mathbf{b} = R\mathbf{a}$. Since rotation does not changes the norm of vector, $b^{\mathrm{T}}b = a^{\mathrm{T}}R^{\mathrm{T}}Ra = a^{\mathrm{T}}a$ where T indicates transpose. Hence, $R^{\mathrm{T}}R = I$ where I is the identity matrix. This means the rotation matrix is an orthogonal matrix defined by $R^{-1} = R^{\mathrm{T}}$. A rotation of $-74°$ around the axes (-1/3,2/3,2/3) is given by a rotation matrix

$$R = \begin{bmatrix} 0.36 & 0.48 & -0.80 \\ -0.80 & 0.60 & 0 \\ 0.48 & 0.64 & 0.60 \end{bmatrix}. \tag{8.57}$$

Show that this matrix is indeed orthogonal by comparing R^{-1} and R^{T}.



8.2 Three particles are chained by four springs as shown in Fig. Two springs at the ends are fixed to the walls. The walls are separated by distance $d = 8$. The natural length of the springs are $\ell_1 = \ell_3 = 1$, $\ell_2 = \ell_4 = 2$ and their spring constants are $k_1 = k_4 = 2$ and $k_2 = k_3 = 4$. Find the length of the springs at mechanical equilibrium.
Let d_i be the length of the i-th spring. The energy of the system is $U = \sum_i \frac{1}{2}k_i(d_i - \ell_i)^2$. Letting the position of the particles $x_i, i = 1, \cdots, 4$.

$$U = \frac{k_1}{2}(x_1 - \ell_1)^2 + \frac{k_2}{2}(x_2 - x_1 - \ell_2)^2 + \frac{k_3}{2}(x_3 - x_2 - \ell_3)^2 + \frac{k_4}{2}(d - x_3 - \ell_4)^2 \tag{8.58}$$

MATLAB Source Codes

Program 8.1

```
%**********************************************************************
%*     Example  8.1                                                   *
%*     filename: ch08pr01.m                                           *
%*     program listing number: 8.1                                    *
%*                                                                    *
%*     This program checks the properties of triangular matrices.     *
%*                                                                    *
%*     Programed by Ryoichi Kawai for Computational Physics Course.   *
%*     Last modification:  01/31/2015.                                *
%**********************************************************************
clear all;

% define matrices A and B
A=[[2, 0, 0];[-1,1,0];[3,2,-1]]
B=[[1, 0, 0];[2,4,0];[-1,-2,3]]

C=A*B;
fprintf('Mutilication: A*B\n')
% MATLAB print column first. Thus you need to print its transpose
fprintf('%3d %3d %3d\n',C')
fprintf('\nInverse of A\n')
D=inv(A);
fprintf('%15.4e %15.4e %15.4e\n',D')

E1=A(1,1)*A(2,2)*A(3,3);
E2=det(A);
fprintf('\nProducts of the diagonal elements = %d\n',E1)
fprintf('Determinant by MATLAB = %d\n',E2)
```

▲▲▲

Program 8.2

```
%**********************************************************************
%*     Example  8.2                                                   *
%*     filename: ch08pr02.m                                           *
%*     program listing number: 8.2                                    *
%*                                                                    *
%*     This program solves a upper-triangular linear equation with    *
%*     the backsubstitution method.                                   *
%*                                                                    *
%*     Programed by Ryoichi Kawai for Computational Physics Course.   *
%*     Last modification:  10/13/2013.                                *
%**********************************************************************
clear all;

% define matrix A and vector b
A=[[3, -1, 4];[0,2,-1];[0,0,2]];
b=[-1;-2;4];

% backsubstitution
for i=3:-1:1
    Ax=0;
    for j=i+1:3
        Ax = Ax+A(i,j)*x(j);
    end
    x(i) = (b(i)-Ax)/A(i,i);
end

fprintf('x=%.1f, y=%.1f, z=%.1f\n',x)
```

▲▲▲

Program 8.3

```
%*********************************************************************
%*     Example  8.3                                                  *
%*     filename: ch08pr03.m                                          *
%*     program listing number: 8.3                                   *
%*                                                                   *
%*     This program solves a simple linear equation with the Gaussian     *
%*     eliminationa and backsubstitution methods.                    *
%*                                                                   *
%*     Programed by Ryoichi Kawai for Computational Physics Course.  *
%*     Last modification:  10/13/2013.                               *
%*********************************************************************
clear all;

% Set a linear equation
N=3;
A=[[3,-1,4];[2,0,-1];[0,3,2]];
b=[2;-1;3];

%forward elimination
for n=1:N-1
    for i=n+1:N
        M=-A(i,n)/A(n,n);
        A(i,n+1:N)=M*A(n,n+1:N)+A(i,n+1:N);
        b(i)=M*b(n)+b(i);
    end
    A(n+1,n)=0;
end

% backsubstitution
for i=3:-1:1
    Ax=0;
    for j=i+1:3
        Ax = Ax+A(i,j)*x(j);
    end
    x(i) = (b(i)-Ax)/A(i,i);
end

% result
fprintf('\nA=\n')
fprintf('%8.5f  %8.5f  %8.5f\n',A')
fprintf('\nb=\n')
fprintf('%8.5f\n',b)
fprintf('\nx=\n')
fprintf('%8.5f\n',x)
```

▲▲▲

Program 8.4

```
%*********************************************************************
%*     Example  8.4                                                  *
%*     filename: ch08pr04.m                                          *
%*     program listing number: 8.4                                   *
%*                                                                     *
%*     This program solves a simple linear equation with the Gaussian     *
%*     elimination with partial pivoting and backsubstitution methods.    *
%*                                                                     *
%*     Programed by Ryoichi Kawai for Computational Physics Course.       *
%*     Last modification:  10/13/2013.                                    *
%**********************************************************************
clear all;

% Set a linear equation
N=3;
A=[[3,-1,4];[2,0,-1];[0,3,2]];
b=[2;-1;3];
P=eye(N,N);  % permutation matrix is initially an identity matrix

% Find scale factors
for i=1:N
    S(i)=max(A(i,:));
end

for n=1:N-1
    % Look for the pivot row
    j=n;
    Amax=abs(A(n,n)/S(n));
    for i=n:N
        AS=abs(A(i,n)/S(i));
        if AS > Amax
            j=i;
            Amax = AS;
        end
    end
    % Carry out pivoting
    if j ~= n
        for i=n:N
            TMP=A(n,i);
            A(n,i)=A(j,i);
            A(j,i)=TMP;
        end
        TMP=b(n);
        b(n)=b(j);
        b(j)=TMP;
        % Record the permutation
        P(n,n)=0; P(j,j)=0;
        P(n,j)=1; P(j,n)=1;
    end
    % Gaussian elimination
    for i=n+1:N
        M=-A(i,n)/A(n,n);
        A(i,n+1:N)=M*A(n,n+1:N)+A(i,n+1:N);
        b(i)=M*b(n)+b(i);
    end
    A(n+1,n)=0;
end

% backsubstitution
for i=3:-1:1
    Ax=0;
    for j=i+1:3
        Ax = Ax+A(i,j)*x(j);
    end
    x(i) = (b(i)-Ax)/A(i,i);
```

```
end

% result
fprintf('\nA=\n')
fprintf('%8.5f  %8.5f  %8.5f\n',A')
fprintf('\nb=\n')
fprintf('%8.5f\n',b)
fprintf('\nP=\n')
fprintf('%i  %i  %i\n',P')
fprintf('\nx=\n')
fprintf('%8.5f\n',x)
```

Program 8.5

```
%**************************************************************************
%*     Example  8.6                                                       *
%*     filename: ch08pr05.m                                               *
%*     program listing number: 8.5                                        *
%*                                                                        *
%*     This program calculates the inverse of a given matrix using       *
%*     Gaussi-Jordin methods.                                             *
%*                                                                        *
%*     Programed by Ryoichi Kawai for Computational Physics Course.       *
%*     Last modification:  10/13/2013.                                    *
%**************************************************************************
clear all;

% Set a linear equation
N=3;
A0=[[3,-1,4];[2,0,-1];[0,3,2]];
A=A0; % keep the original matrix
b=eye(N,N);

% scale factors
for i=1:N
    S(i)=max(A(i,:));
end

for n=1:N-1
    % Look for the pivot row
    j=n;
    Amax=abs(A(n,n)/S(n));
    for i=n:N
        AS=abs(A(i,n)/S(i));
        if AS > Amax
            j=i;
            Amax = AS;
        end
    end
    % Carry out pivoting
    if j ~= n
        for i=n:N
            TMP=A(n,i);
            A(n,i)=A(j,i);
            A(j,i)=TMP;
        end
        TMP2(1:N) = b(n,:);
        b(n,:)=b(j,:);
        b(j,:)=TMP2(1:N);
    end
    % Gaussian elimination
    for i=n+1:N
        M=-A(i,n)/A(n,n);
        A(i,n+1:N)=M*A(n,n+1:N)+A(i,n+1:N);
        b(i,:)=M*b(n,:)+b(i,:);
    end
    A(n+1,n)=0;
end

% backsubstitution
for i=3:-1:1
    Ax(1:N)=0;
    for j=i+1:3
        for k=1:N
            Ax(k) = Ax(k)+A(i,j)*x(j,k);
        end
    end
    for j=1:N
        x(i,j) = (b(i,j)-Ax(j))/A(i,i);
    end
end

% result
fprintf('\nInvers of A=\n')
fprintf('%8.5f  %8.5f  %8.5f\n',x)
fprintf('\nA A^(-1)=\n')
fprintf('%8.5f  %8.5f  %8.5f\n',A0*x)
```

▲▲▲

Program 8.6

```
%**************************************************************************
%*     Example  8.7                                                       *
%*     filename: ch08pr06.m                                               *
%*     program listing number: 8.6                                        *
%*                                                                        *
%*     This program solves a simple linear equation with LU decomposition.*
%*     MATLAB function lu() is used.                                      *
%*                                                                        *
%*     Programed by Ryoichi Kawai for Computational Physics Course.       *
%*     Last modification:  10/13/2013.                                    *
%**************************************************************************
clear all;

% Define a matrix
A=[[3, -1, 4];[2, 0, -1];[0, 3, 2]];
b=[2;-1;3];

% LU dcomposition
[L U P]=lu(A);

% Rcover the original matrix
S=P*L*U;

% Show the results
fprintf('\nA (Original Matrix)\n')
fprintf('%8.5f  %8.5f  %8.5f\n',A')
fprintf('\nL (Lower Triangular Matrix)\n')
fprintf('%8.5f  %8.5f  %8.5f\n',L')
fprintf('\nU (Upper Triangular Matrix)\n')
fprintf('%8.5f  %8.5f  %8.5f\n',U')
fprintf('\nP (Permutation Matrix)\n')
fprintf('%i  %i  %i\n',P')
fprintf('\nP*L*U\n')
fprintf('%8.5f  %8.5f  %8.5f\n',S')

b = P*b;
% forward substition
for i=1:3
    Ly=0;
    for j=1:i-1
        Ly = Ly+L(i,j)*y(j);
    end
    y(i) = (b(i)-Ly)/L(i,i);
end

% backsubstitution
for i=3:-1:1
    Ux=0;
    for j=i+1:3
        Ux = Ux+U(i,j)*x(j);
    end
    x(i) = (y(i)-Ux)/U(i,i);
end

fprintf('\nx=%.5f,  y=%.5f, z=%.5f\n',x)
```

Program 8.7

```
%***********************************************************************
%*     Example  8.9                                                    *
%*     filename: ch08pr07.m                                            *
%*     program listing number: 8.7                                     *
%*                                                                     *
%*     This program solves a tridiagonal system with backward elimination.*
%*     Then, find solution by forward substitution.                    *
%*                                                                     *
%*     Programed by Ryoichi Kawai for Computational Physics Course.    *
%*     Last modification:  02/01/2015.                                 *
%***********************************************************************
clear all

% Define matrices. No need to use the full matrix.
d=[2,3,4,3]; % diagonal elements
u=[2,3,3,0]; % above diagonal
l=[0,2,3,3]; % below diagonal
b=[1,2,3,4]; % right hand side

% Calculation of determinant
D(1)=d(1);
D(2)=d(2)*D(1)-l(1)*u(1);
for i=3:4
    D(i)=d(i)*D(i-1)-l(i-1)*u(i-1)*D(i-2);
end

fprintf('Determinant %d\n',D(4))
if D(4) == 0
    fprintf('Singular')
    stop
end
```

```
% Decomposition by backword elimination
Y(3)=-l(4)/d(4);
Z(3)= b(4)/d(4);
for i=3:-1:2
    Y(i-1)=-l(i)/(d(i)+u(i)*Y(i));
    Z(i-1)=(b(i)-u(i)*Z(i))/(d(i)+u(i)*Y(i));
end

% Forward substitution
x(1)=(b(1)-u(1)*Z(1))/(d(1)+u(1)*Y(1));
for i=1:3
    x(i+1)=Y(i)*x(i)+Z(i);
end

% Answer
fprintf('x= %f %f %f %f\n',x)

% Check the errors.
s(1)=d(1)*x(1)+u(1)*x(2)-b(1);
s(2)=l(2)*x(1)+d(2)*x(2)+u(2)*x(3)-b(2);
s(3)=l(3)*x(2)+d(3)*x(3)+u(3)*x(4)-b(3);
s(4)=l(4)*x(3)+d(4)*x(4)-b(4);
fprintf('Error= %e %e %e %e\n',s)
```

Program 8.8

```
%**************************************************************************
%*     Section  8.6.3                                                     *
%*     filename: ch08pr08.m                                               *
%*     program listing number: 8.8                                        *
%*                                                                        *
%*     This program calculate the determinant of distance matrix for      *
%*     a tree graph using Gaussian elimination with partial pivoting.     *
%*                                                                        *
%*     Programed by Ryoichi Kawai for Computational Physics Course.       *
%*     Last modification:  02/01/2015.                                    *
%**************************************************************************
clear all;

% Set a linear equation
N=10;
A=[[0 , 1 , 2 , 3 , 4 , 4 , 3 , 4 , 4 , 5 ];...
   [1 , 0 , 1 , 2 , 3 , 3 , 2 , 3 , 3 , 4 ];...
   [2 , 1 , 0 , 1 , 2 , 2 , 1 , 2 , 2 , 3 ];...
   [3 , 2 , 1 , 0 , 1 , 1 , 2 , 3 , 3 , 4 ];...
   [4 , 3 , 2 , 1 , 0 , 2 , 3 , 4 , 4 , 5 ];...
   [4 , 3 , 2 , 1 , 2 , 0 , 3 , 4 , 4 , 5 ];...
   [3 , 2 , 1 , 2 , 3 , 3 , 0 , 1 , 1 , 2 ];...
   [4 , 3 , 2 , 3 , 4 , 4 , 1 , 0 , 2 , 3 ];...
   [4 , 3 , 2 , 3 , 4 , 4 , 1 , 2 , 0 , 1 ];...
   [5 , 4 , 3 , 4 , 5 , 5 , 2 , 3 , 1 , 0 ]];

P=eye(N,N);  % permutation matrix is initially an identity matrix

% Find scale factors
for i=1:N
    S(i)=max(A(i,:));
end

for n=1:N-1
```

```
    % Look for the pivot row
    j=n;
    Amax=abs(A(n,n)/S(n));
    for i=n:N
        AS=abs(A(i,n)/S(i));
        if AS > Amax
            j=i;
            Amax = AS;
        end
    end
    % Carry out pivoting
    if j ~= n
        for i=n:N
            TMP=A(n,i);
            A(n,i)=A(j,i);
            A(j,i)=TMP;
        end
        % Record the permutation
        P(n,n)=0; P(j,j)=0;
        P(n,j)=1; P(j,n)=1;
    end
    % Gaussian elimination
    for i=n+1:N
        M=-A(i,n)/A(n,n);
        A(i,n+1:N)=M*A(n,n+1:N)+A(i,n+1:N);
    end
    A(n+1,n)=0;
end

p=sum(sum(P))
D=(-1)^p;
for i=1:10
    D=D*A(i,i);
end
D_GP=-(N-1)*(-2)^(N-2);
fprintf('Gaussian Elimination: %f\n',D)
fprintf('       Graham-Pollak: %f\n',D_GP)
```

Program 8.9

```
%***************************************************************************
%*     Example  8.10                                                       *
%*     filename: ch08pr09.m                                                *
%*     program listing number: 8.9                                         *
%*                                                                         *
%*     This program solves a 2x2 linear equation by the steepest descent  *
%*     minimization.                                                       *
%*                                                                         *
%*     Programed by Ryoichi Kawai for Computational Physics Course.       *
%*     Last modification:  10/13/2013.                                     *
%***************************************************************************
clear all;
A=[[4,1];[1,3]];
b=[1;2];
c=[-0.65,  -0.5, -0.3,  -0.1, 0.1,  0.3, 0.5, 0.7, 0.9, 1.1];
tol = 1e-8;

% contour plot of the cost function
x=linspace(-1,1.2);
y=linspace(-1,2);
```

```
[X,Y]=meshgrid(x,y);
N=size(X,2);
M=size(Y,2);
for i=1:N
    for j=1:M
        Z(i,j) = 0.5*(X(i,j)^2*A(1,1)+(A(1,2)+A(2,1))*X(i,j)*Y(i,j)...
                +A(2,2)*Y(i,j)^2) - X(i,j)*b(1)-Y(i,j)*b(2);
    end
end

contour(X,Y,Z,c);
hold on

% steepest descent with line minimization
n=1;
x=[1;0.5]; % starting point
u(1)=x(1);
v(1)=x(2);
g = A*x-b;
gg=g'*g;
while abs(gg)>tol
    n=n+1;
    lambda = gg/(g'*A*g);
    x = x - lambda * g;
    u(n)=x(1);
    v(n)=x(2);
    g = A*x-b;
    gg = g'*g;
end

fprintf('Solution=(%.5f,%.5f)\n',x)

p=plot(u,v);
set(p,'linewidth',2,'color','black')
axis equal tight
xlabel(texlabel('x_1'),'fontsize',14)
ylabel(texlabel('x_2'),'fontsize',14)

hold off
```

▲▲▲

Program 8.10

```
%***************************************************************************
%*     Example  8.11                                                       *
%*     filename: ch08pr10.m                                                *
%*     program listing number: 8.10                                        *
%*                                                                         *
%*     This program solves a 2x2 linear equation by the conjugate          *
%*     gradient minimization.                                              *
%*                                                                         *
%*     Programed by Ryoichi Kawai for Computational Physics Course.        *
%*     Last modification:  10/13/2013.                                     *
%***************************************************************************
clear all;
A=[[4,1];[1,3]];
b=[1;2];
c=[-0.65,  -0.5, -0.3,  -0.1, 0.1,  0.3, 0.5, 0.7, 0.9, 1.1];
tol = 1e-8;

% contour plot of the cost function
```

```
x=linspace(-1,1.2);
y=linspace(-1,2);
[X,Y]=meshgrid(x,y);
N=size(X,2);
M=size(Y,2);
for i=1:N
    for j=1:M
        Z(i,j) = 0.5*(X(i,j)^2*A(1,1)+(A(1,2)+A(2,1))*X(i,j)*Y(i,j)...
                 +A(2,2)*Y(i,j)^2) - X(i,j)*b(1)-Y(i,j)*b(2);
    end
end

contour(X,Y,Z,c);
hold on

% conjugate gradient method
n=1;
x=[1;0.5]; % starting point
u(1)=x(1);
v(1)=x(2);
r=b- A*x;
p=r;
rr=r'*r;
while abs(rr)>tol
    n=n+1;
    alpha = rr/(r'*A*r);
    x = x + alpha*p;
    u(n)=x(1);
    v(n)=x(2);
    r1=b-A*x;
    rr1=r1'*r1;
    beta=rr1/rr;
    r=r1;
    rr=rr1;
    p=r+beta*p;
end

fprintf('Solution=(%.5f,%.5f)\n',x)

p=plot(u,v);
set(p,'linewidth',2,'color','black')
axis equal tight
xlabel(texlabel('x_1'),'fontsize',14)
ylabel(texlabel('x_2'),'fontsize',14)
legend('f(x)','CG steps');
hold off
```

▲▲▲

Python Source Codes

Program 8.1

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%***************************************************************************
```

```
%*     Example  8.1                                                    *
%*     filename: ch08pr01.m                                            *
%*     program listing number: 8.1                                     *
%*                                                                     *
%*     This program checks the properties of triangular matrices.      *
%*                                                                     *
%*     Programed by Ryoichi Kawai for Computational Physics Course.    *
%*     Last modification:  01/31/2015.                                 *
%***********************************************************************
"""
import numpy as np

# define matrices A and B (do not use array)
A=np.matrix([[2, 0, 0],[-1,1,0],[3,2,-1]])
B=np.matrix([[1, 0, 0],[2,4,0],[-1,-2,3]])

print("A*B")
print(A*B)

print('\nInverse of A')
print(np.linalg.inv(A))

print("\nDeterminant of A={0:7.5f}".format(np.linalg.det(A)))
```

Program 8.2

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%***********************************************************************
%*     Example  8.2                                                    *
%*     filename: ch08pr02.py                                           *
%*     program listing number: 8.2                                     *
%*                                                                     *
%*     This program solves a upper-triangular linear equation with     *
%*     the backsubstitution method.                                    *
%*                                                                     *
%*     Programed by Ryoichi Kawai for Computational Physics Course.    *
%*     Last modification:  02/06/2017.                                 *
%***********************************************************************
"""
import numpy as np

# define matrix A and vector b
A=np.matrix([[3, -1, 4],[0,2,-1],[0,0,2]])
b=np.matrix([[-1],[-2],[4]])
x=b
# backsubstitution
for i in range(2,-1,-1):
    Ax=0.0
    for j in range(i+1,3):
        Ax = Ax+A[i,j]*x[j]

    x[i] = (b[i]-Ax)/A[i,i]

print(x)
```

Program 8.3

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*********************************************************************
%*     Example  8.3                                                  *
%*     filename: ch08pr03.py                                         *
%*     program listing number: 8.3                                   *
%*                                                                   *
%*     This program solves a simple linear equation with the Gaussian *
%*     eliminationa and backsubstitution methods.                    *
%*                                                                   *
%*     Programed by Ryoichi Kawai for Computational Physics Course.  *
%*     Last modification:  02/06/2017.                               *
%*********************************************************************
"""
import numpy as np

# Set a linear equation
N=3;
A=np.matrix([[3.,-1.,4.],[2.,0.,-1.],[0.,3.,2.]])
b=np.matrix.transpose(np.matrix([2.,-1.,3.]))
x=np.matrix.transpose(np.matrix(np.zeros(N)))

#forward elimination
for n in range(0,N-1):

    for i in range(n+1,N):
        M=-A[i,n]/A[n,n]
        A[i,n+1:N]=M*A[n,n+1:N]+A[i,n+1:N]
        b[i]=M*b[n]+b[i]

    A[n+1,n]=0.0


# backsubstitution
for i in range(2,-1,-1):
    Ax=0.0
    for j in range(i+1,3):
        Ax = Ax+A[i,j]*x[j]

    x[i] = (b[i]-Ax)/A[i,i]


# result
print('\nA=\n')
print(A)
print('\nb=\n')
print(b)
print('\nx=\n')
print(x)
```

Program 8.4

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*********************************************************************
%*     Example  8.4                                                  *
%*     filename: ch08pr04.pu                                         *
%*     program listing number: 8.4                                   *
%*                                                                      *
%*     This program solves a simple linear equation with the Gaussian     *
%*     elimination with poartial pivoting and backsubstitution methods.   *
%*                                                                      *
%*     Programed by Ryoichi Kawai for Computational Physics Course.        *
%*     Last modification:  02/08/2017.                                    *
%***********************************************************************
"""

# Set a linear equation
N=3;
A=np.matrix([[3.,-1.,4.],[2.,0.,-1.],[0.,3.,2.]])
b=np.matrix.transpose(np.matrix([2.,-1.,3.]))
x=np.matrix.transpose(np.matrix(np.zeros(N)))
# permutation matrix must be initially an identity matrix
P=np.matrix(np.identity(3,dtype=int))
S=np.zeros(3)

# Find scale factors
for i in range(0,N):
    S[i]=A[i,:].max()


for n in range(0,N-1):
    # Look for the pivot row
    j=n
    Amax=abs(A[n,n]/S[n])
    for i in range(n,N):
        AS=abs(A[i,n]/S[i])
        if AS > Amax:
            j=i
            Amax = AS

    # Carry out pivoting
    if j != n :
        for i in range(n,N) :
            TMP=A.item(n,i)
            A[n,i]=A.item(j,i)
            A[j,i]=TMP

        TMP=b.item(n)
        b[n]=b.item(j)
        b[j]=TMP
        # Record the permutation
        P[n,n]=P[j,j]=0
        P[n,j]=P[j,n]=1

    # Gaussian elimination
    for i in range(n+1,N):
        M=-A[i,n]/A[n,n]
        A[i,n+1:N]=M*A[n,n+1:N]+A[i,n+1:N]
        b[i]=M*b[n]+b[i]

    A[n+1,n]=0.0


# backsubstitution
for i in range(2,-1,-1):
    Ax=0.0
    for j in range(i+1,3):
        Ax = Ax+A[i,j]*x[j]
    x[i] = (b[i]-Ax)/A[i,i]

# result
print('\nA=\n')
print(A)
print('\nb=\n')
print(b)
print('\nP=\n')
print(P)
print('\nx=\n')
print(x)
```

Program 8.5

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%***************************************************************************
%*     Example  8.5                                                        *
%*     filename: ch08pr05.py                                               *
%*     program listing number: 8.5                                         *
%*                                                                         *
%*     This program calculates the inverse of a given matrix using        *
%*     Gaussi-Jordin methods.                                              *
%*                                                                         *
%*     Programed by Ryoichi Kawai for Computational Physics Course.       *
%*     Last modification:  02/08/2017.                                     *
%***************************************************************************
"""
import numpy as np

# Set a linear equation
N=3
A0=np.matrix([[3.,-1.,4.],[2.,0.,-1.],[0.,3.,2.]])
A=np.identity(N)
A[:,:]=A0[:,:]
b=np.matrix(np.identity(N))  # permutation matrix is initially an identity matrix
x=np.matrix(np.identity(N))
S=np.zeros(N)
TMP2=np.zeros(N)

# scale factors
for i in range(0,N):
    S[i]=A[i,:].max()


for n in range(0,N-1):
    # Look for the pivot row
    j=n
    Amax=abs(A[n,n]/S[n])
    for i in range(n,N):
        AS=abs(A[i,n]/S[i])
        if AS > Amax:
            j=i
            Amax = AS

    # Carry out pivoting
    if j != n:
        for i in range(n,N):
            TMP=A[n,i]
            A[n,i]=A[j,i]
            A[j,i]=TMP

        TMP2[:]=b[n,:]
        b[n,:]=b[j,:]
        b[j,:]=TMP2[:]

    # Gaussian elimination
    for i in range(n+1,N):
        M=-A[i,n]/A[n,n]
        A[i,n+1:N]=M*A[n,n+1:N]+A[i,n+1:N]
        b[i,:]=M*b[n,:]+b[i,:]

    A[n+1,n]=0.0

# backsubstitution
Ax=np.zeros(N)
for i in range(N-1,-1,-1):
    Ax=np.zeros(N)
    for j in range(i+1,N):
        for k in range(0,N):
            Ax[k] = Ax[k]+A.item(i,j)*x[j,k]

    x[i,:] = (b[i,:]-Ax[:])/A[i,i]

# result
print('\nInvers of A=')
print(x)
print('\nA A^(-1)=')
print(A0*x)
```

Program 8.6

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*********************************************************************
%*     Example  8.6                                                  *
%*     filename: ch08pr06.py                                         *
%*     program listing number: 8.6                                   *
%*                                                                   *
%*     This program solves a simple linear equation with LU decomposition.*
%*     MATLAB function lu() is used.                                 *
%*                                                                   *
%*     Programed by Ryoichi Kawai for Computational Physics Course.  *
%*     Last modification:  02/08/2017.                               *
%*********************************************************************
"""

import numpy as np
import scipy.linalg as la

# Define a matrix
A=np.matrix([[3., -1., 4.],[2., 0., -1.],[0., 3., 2.]])
b=np.matrix([2.,-1.,3.]).transpose()
x=np.matrix(np.zeros(3)).transpose()
y=np.matrix(np.zeros(3)).transpose()

# LU dcomposition
```

```
P, L, U = la.lu(A)
P=np.matrix(P)
U=np.matrix(U)
L=np.matrix(L)

# Rcover the original matrix
S=P*L*U
# Show the results
print('\nA (Original Matrix)')
print(A)
print('\nL (Lower Triangular Matrix)')
print(L)
print('\nU (Upper Triangular Matrix)')
print(U)
print('\nP (Permutation Matrix)')
print(P)
print('\nP*L*U')
print(S)

b = P*b
# forward substition
for i in range(0,3):
    Ly=0.0
    for j in range(0,i):
        Ly = Ly+L[i,j]*y[j]

    y[i] = (b[i]-Ly)/L[i,i]

# backsubstitution
for i in range(2,-1,-1):
    Ux=0.0
    for j in range(i+1,3):
        Ux = Ux+U[i,j]*x[j]

    x[i] = (y[i]-Ux)/U[i,i]

print('\nSolution x')
print(x)
```

Program 8.7

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*************************************************************************
%*     Example  8.7                                                      *
%*     filename: ch08pr07.py                                             *
%*     program listing number: 8.7                                       *
%*                                                                       *
%*     This program solves a tridiagonal system with backward elimination.*
%*     Then, find solution by forward substitution.                      *
%*                                                                       *
%*     Programed by Ryoichi Kawai for Computational Physics Course.      *
%*     Last modification:  02/08/2017.                                   *
%*************************************************************************
"""
import numpy as np

# Define matrices. No need to use the full matrix.
d=[2,3,4,3] # diagonal elements
```

```
u=[2,3,3,0] # above diagonal
l=[0,2,3,3] # below diagonal
b=[1,2,3,4] # right hand side
D = np.zeros(4)
Y = np.zeros(4)
Z = np.zeros(4)
s = np.zeros(4)
x = np.zeros(4)
# Calculation of determinant
D[0]=d[0]
D[1]=d[1]*D[0]-l[0]*u[0]
for i in range(2,4):
    D[i]=d[i]*D[i-1]-l[i-1]*u[i-1]*D[i-2]


print('Determinant {0:8.3f}='.format(D[3]))
if D[3] == 0:
    exit('Singular')

# Decomposition by backword elimination
Y[2]=-l[3]/d[3]
Z[2]= b[3]/d[3]
for i in range(2,0,-1):
    Y[i-1]=-l[i]/(d[i]+u[i]*Y[i])
    Z[i-1]=(b[i]-u[i]*Z[i])/(d[i]+u[i]*Y[i])


# Forward substitution
x[0]=(b[0]-u[0]*Z[0])/(d[0]+u[0]*Y[0])
for i in range(0,3):
    x[i+1]=Y[i]*x[i]+Z[i]

# Answer
print('Solution x')
print(x)

# Check the errors.
s[0]=          d[0]*x[0]+u[0]*x[1]-b[0]
s[1]=l[1]*x[0]+d[1]*x[1]+u[1]*x[2]-b[1]
s[2]=l[2]*x[1]+d[2]*x[2]+u[2]*x[3]-b[2];
s[3]=l[3]*x[2]+d[3]*x[3]-b[3]
print('Error')
print(s)
```

Program 8.8

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*********************************************************************
%*     Section  8.6.3                                                *
%*     filename: ch08pr08.py                                         *
%*     program listing number: 8.8                                   *
%*                                                                   *
%*     This program calculate the determinant of distance matrix for *
%*     a tree graph using Gaussian elimination with partial pivoting. *
%*                                                                   *
%*     Programed by Ryoichi Kawai for Computational Physics Course.  *
%*     Last modification:  02/08/2017.                               *
%*********************************************************************
```

```
"""
import numpy as np

# Set a linear equation
N=10
A=[[0. , 1. , 2. , 3. , 4. , 4. , 3. , 4. , 4. , 5. ],
   [1. , 0. , 1. , 2. , 3. , 3. , 2. , 3. , 3. , 4. ],
   [2. , 1. , 0. , 1. , 2. , 2. , 1. , 2. , 2. , 3. ],
   [3. , 2. , 1. , 0. , 1. , 1. , 2. , 3. , 3. , 4. ],
   [4. , 3. , 2. , 1. , 0. , 2. , 3. , 4. , 4. , 5. ],
   [4. , 3. , 2. , 1. , 2. , 0. , 3. , 4. , 4. , 5. ],
   [3. , 2. , 1. , 2. , 3. , 3. , 0. , 1. , 1. , 2. ],
   [4. , 3. , 2. , 3. , 4. , 4. , 1. , 0. , 2. , 3. ],
   [4. , 3. , 2. , 3. , 4. , 4. , 1. , 2. , 0. , 1. ],
   [5. , 4. , 3. , 4. , 5. , 5. , 2. , 3. , 1. , 0. ]]
A=np.matrix(A)
P=np.matrix(np.identity(N),dtype=int)   # permutation matrix
b=np.matrix(np.zeros(N)).transpose()

S=np.zeros(N)
# Find scale factors
for i in range(0,N):
    S[i]=A[i,:].max()


for n in range(0,N-1):
    # Look for the pivot row
    j=n
    Amax=abs(A[n,n]/S[n])
    for i in range(n,N):
        AS=abs(A[i,n]/S[i])
        if AS > Amax:
            j=i
            Amax = AS

    # Carry out pivoting
    if j != n :
        for i in range(n,N):
            TMP=A[n,i]
            A[n,i]=A[j,i]
            A[j,i]=TMP

        TMP=np.asscalar(b[n])
        b[n]=b[j]
        b[j]=TMP

        # Record the permutation
        P[n,n]=P[j,j]=0
        P[n,j]=P[j,n]=1

    # Gaussian elimination
    for i in range(n+1,N):
        M=-A[i,n]/A[n,n]
        A[i,n+1:N]=M*A[n,n+1:N]+A[i,n+1:N]
        b[i]=M*b[n]+b[i]

    A[n+1,n]=0.0

p=P.sum()
D=(-1)**p
for i in range(0,N):
    D=D*A[i,i]

D_GP=-(N-1)*(-2)**(N-2);
print('Gaussian Elimination: {0:f}'.format(D))
print('       Graham-Pollak: {0:d}'.format(D_GP))
```

▲▲▲

Program 8.9

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%***************************************************************************
%*     Example  8.10                                                       *
%*     filename: ch08pr09.py                                               *
%*     program listing number: 8.9                                         *
%*                                                                         *
%*     This program solves a 2x2 linear equation by the steepest descent  *
%*     minimization.                                                       *
%*                                                                         *
%*     Programed by Ryoichi Kawai for Computational Physics Course.        *
%*     Last modification:  02/08/2017.                                     *
%***************************************************************************
"""
import numpy as np
import matplotlib.pyplot as plt

A=np.matrix([[4.,1.],[1.,3.]])
b=np.matrix([1.,2.]).transpose()



# steepest descent with line minimization

kmax=1000
tol = 1e-8
x=np.matrix([1,0.5]).transpose() # starting point
u=np.zeros(kmax)
v=np.zeros(kmax)
u[0]=np.asscalar(x[0])   # In numpy, a column vector must be
v[0]=np.asscalar(x[1])   # treated as matrix of (Nx1).

g = A*x-b;
gg=np.asscalar(g.transpose()*g)

n=0
while abs(gg)>tol and n<kmax:
    n+=1
    lam = gg/np.asscalar(g.transpose()*A*g)
    x = x - lam * g
    u[n]=np.asscalar(x[0])
    v[n]=np.asscalar(x[1])
    g = A*x-b
    gg = np.asscalar(g.transpose()*g)

print('\nSolution=({0:f},{1:f})'.format(x.item(0),x.item(1)))

# contour plot of the cost function
plt.figure(figsize=(5,6))
delta = 0.025
x = np.arange(-1.0, 1.2, delta)
```

```
y = np.arange(-1.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
c=np.array([-0.65,  -0.5, -0.3,  -0.1, 0.1,  0.3, 0.5, 0.7, 0.9, 1.1])

N=x.size
M=y.size
Z=np.zeros((M,N))

for i in range(0,M):
    for j in range(0,N):

        Z[i,j] = 0.5*(X[i,j]**2*A[0,0]+(A[0,1]+A[1,0])*X[i,j]*Y[i,j]
                 +A[1,1]*Y[i,j]**2) - X[i,j]*b[0]-Y[i,j]*b[1]

CS = plt.contour(X, Y, Z, c)
plt.clabel(CS, inline=1, fontsize=10)
plt.xlim(-1.0,1.2)
plt.ylim(-1.0,2.0)
plt.axes().set_aspect('equal', 'datalim')

# plot the trajectory
plt.plot(u[0:n+1],v[0:n+1],'-r',linewidth=2)
plt.xlabel(r'$x_1$',fontsize=14)
plt.ylabel(r'$x_2$',fontsize=14)
plt.title('Steepest Descent Minimization')

plt.show()
```

Program 8.10

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
%*********************************************************************
%*     Example  8.11                                                 *
%*     filename: ch08pr10.py                                         *
%*     program listing number: 8.10                                  *
%*                                                                   *
%*     This program solves a 2x2 linear equation by the conjugate    *
%*     gradient minimization.                                        *
%*                                                                   *
%*     Programed by Ryoichi Kawai for Computational Physics Course.  *
%*     Last modification:  02/08/2017.                               *
%*********************************************************************
"""
import numpy as np
import matplotlib.pyplot as plt

A=np.matrix([[4.,1.],[1.,3.]])
b=np.matrix([1.,2.]).transpose()
p=np.matrix([0.,0.]).transpose()
r=np.matrix([0.,0.]).transpose()
r1=np.matrix([0.,0.]).transpose()

tol = 1e-8;

# conjugate gradient method
kmax=1000
tol = 1e-8
x=np.matrix([1,0.5]).transpose() # starting point
```

```
u=np.zeros(kmax)
v=np.zeros(kmax)
u[0]=x[0,0]    # In numpy, a column vector must be
v[0]=x[1,0]    # treated as matrix of (Nx1).

r=b-A*x
p[:]=r[:]
rr=np.asscalar(r.transpose()*r)
n=0
while abs(rr)>tol and n<kmax:
    n+=1
    alpha = rr/np.asscalar(r.transpose()*A*r)
    x = x + alpha*p
    u[n]=x.item(0)    # In numpy, a column vector must be
    v[n]=x.item(1)    # treated as matrix of (Nx1).
    r1=b-A*x
    rr1=np.asscalar(r1.transpose()*r1)
    beta=rr1/rr
    r[:]=r1[:]
    rr=rr1
    p[:]=r[:]+beta*p[:]


print('\nSolution=({0:f},{1:f})'.format(x.item(0),x.item(1)))

# contour plot of the cost function
plt.figure(figsize=(5,6))
delta = 0.025
x = np.arange(-1.0, 1.2, delta)
y = np.arange(-1.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
c=np.array([-0.65,  -0.5, -0.3,  -0.1, 0.1,  0.3, 0.5, 0.7, 0.9, 1.1])

N=x.size
M=y.size
Z=np.zeros((M,N))

for i in range(0,M):
    for j in range(0,N):

        Z[i,j] = 0.5*(X[i,j]**2*A[0,0]+(A[0,1]+A[1,0])*X[i,j]*Y[i,j]
                 +A[1,1]*Y[i,j]**2) - X[i,j]*b[0]-Y[i,j]*b[1]

CS = plt.contour(X, Y, Z, c)
plt.clabel(CS, inline=1, fontsize=10)
plt.xlim(-1.0,1.2)
plt.ylim(-1.0,2.0)
plt.axes().set_aspect('equal', 'datalim')

# plot the trajectory
plt.plot(u[0:n+1],v[0:n+1],'-r',linewidth=2)
plt.xlabel(r'$x_1$',fontsize=14)
plt.ylabel(r'$x_2$',fontsize=14)
plt.title('Conjugate Gradient Minimization')

plt.show()
```

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 4th edition, 2012.

[3] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. This is unpublished document. Use Google to find it., 1994.

[4] R. L. Graham and H. O. Polak. On the addressing problem for loop switching. *Bell System Tech. J.*, 50:2495–2519, 1971.

[5] Wigen Yan and Teong-Nan Yeh. A simple proof of graham and pollak's theorem. *Journal of Combinatorial Theory, Series A*, 113:892–893, 2006.