Lecture 12

Linear systems of equations II

1 Introduction

We have looked at Gauss-Jordan elimination and Gaussian elimination as ways to solve a linear system $\mathbf{Ax}=\mathbf{b}$. We now turn to the *LU decomposition*, which is arguably "the best" way to solve a linear system. We will then see how to use the *backslash* operator that is built in to Scilab/Matlab.

2 LU decomposition

Generally speaking, a square matrix **A**, for example

$$\mathbf{A}=\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \tag{1}$$

can be "decomposed" or factored as

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

where **L** and **U** are *unit-lower-triangular* and *upper-triangular* matrices of the form

$$\mathbf{L}=\begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \qquad \mathbf{U}=\begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix} \tag{2}$$

One way to see this is to carry out the multiplication.

$$\mathbf{LU}=\begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ u_{11}l_{21} & u_{22}+u_{12}l_{21} & u_{23}+u_{13}l_{21} & u_{24}+u_{14}l_{21} \\ u_{11}l_{31} & u_{22}l_{32}+u_{12}l_{31} & u_{33}+u_{23}l_{32}+u_{13}l_{31} & u_{34}+u_{24}l_{32}+u_{14}l_{31} \\ u_{11}l_{41} & u_{22}l_{42}+u_{12}l_{41} & u_{33}l_{43}+u_{23}l_{42}+u_{13}l_{41} & u_{44}+u_{23}l_{43}+u_{24}l_{42}+u_{14}l_{41} \end{pmatrix} \tag{3}$$

Comparing (1) and (3) we see immediately from the first row that

$$u_{11}=a_{11}\ ,\ u_{12}=a_{12}\ ,\ u_{13}=a_{13}\ ,\ u_{14}=a_{14} \tag{4}$$

Then from the first column we have

$$l_{21}=a_{21}/u_{11}\ ,\ l_{31}=a_{31}/u_{11}\ ,\ l_{41}=a_{41}/u_{11} \tag{5}$$

Now the second row gives us

$$u_{22}=a_{22}-u_{12}l_{21}\ ,\ u_{23}=a_{23}-u_{13}l_{21}\ ,\ u_{24}=a_{24}-u_{14}l_{21} \tag{6}$$

Then from the second column we have

$$l_{32}=(a_{32}-u_{12}l_{31})/u_{22}\ ,\ l_{42}=(a_{42}-u_{12}l_{41})/u_{22} \tag{7}$$

The third row now gives us

$$u_{33}=a_{33}-(u_{23}l_{32}+u_{13}l_{31})\ ,\ u_{34}=a_{34}-(u_{24}l_{32}+u_{14}l_{31}) \tag{8}$$

The third column gives us

$$l_{43}=a_{43}-(u_{23}l_{42}+u_{13}l_{41})/u_{33} \tag{9}$$

and, finally, from the fourth row

$$u_{44}=a_{44}-(u_{23}l_{43}+u24l_{42}+u_{14}l_{41}) \tag{10}$$

This can be generalized to a matrix **A** of size *n*-by-*n* in which case it forms *Doolittle's algorithm*.

LU decomposition can be conveniently done "in place," meaning we don't need to actually create the **L** and **U** matrices. Instead, we can replace the elements of **A** by the corresponding element of **L** or **U** as they are calculated. The result is that **A** is replaced with

$$\mathbf{A}\leftarrow\begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21} & u_{22} & u_{23} & u_{24} \\ l_{31} & l_{32} & u_{33} & u_{34} \\ l_{41} & l_{42} & l_{43} & u_{44} \end{pmatrix}$$

The diagonal values of **L** (all 1's) are understood. The algorithm to do this can be stated very concisely as

LU decomposition algorithm

for $k=1,2,\ldots,n-1$
  for $i=k+1,k+2,\ldots,n$
    $a_{ik}\leftarrow a_{ik}/a_{kk}$
    for $j=k+1,k+2,\ldots,n$
      $a_{ij}\leftarrow a_{ij}-a_{ik}a_{kj}$

A Scilab implementation of this appears in the Appendix as `linearLU`. Note the three nested for loops, implying that this is an $O(n^3)$ process.

2.1 Solving a linear system using LU decomposition

LU decomposition replaces the original system $\mathbf{A}\,\mathbf{x}=\mathbf{b}$ with a factored or "decomposed" system $\mathbf{LU}\,\mathbf{x}=\mathbf{b}$. We get the solution **x** by a two-step process. First we define a vector $\mathbf{y}=\mathbf{U}\,\mathbf{x}$ and write the system in the form $\mathbf{L}\,\mathbf{y}=\mathbf{b}$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}=\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

From the first row we see that the value of y_1 is simply

$$y_1 = b_1$$

The second row now gives us

$$l_{21} y_1 + y_2 = b_2 \rightarrow y_2 = b_2 - l_{21} y_1$$

and the i^{th} row will provide the solution

$$y_i = b_i - \sum_{k=1}^{i-1} l_{ik} y_k$$

This process used to solve for **y** is called *forward substitution*. The complete algorithm is

Forward-substitution algorithm

The solution of $\mathbf{Ly} = \mathbf{b}$ where **L** is unit-lower-triangular is

for $i = 1, 2, \ldots, n$

$$y_i = b_i - \sum_{j=1}^{i-1} l_{ij} y_j$$ (for $i = 1$ there are no terms in the sum)

With two nested loops this is an $O(n^2)$ process.

Now that we have **y**, in the next step we solve $\mathbf{U}\,\mathbf{x} = \mathbf{y}$

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}$$

This upper-triangular system can be solved by the backward-substitution algorithm we developed in the previous lecture. We restate it here for completeness.

Back-substitution algorithm

The solution of $\mathbf{Ux} = \mathbf{y}$ where **U** is upper-triangular is

for $i = n, n-1, n-2, \ldots, 1$

$$x_i = \frac{1}{u_{ii}} \left(y_i - \sum_{j=i+1}^{n} u_{i,j} x_j \right)$$ (summation is zero when $i = n$)

A Scilab implementation of forward- and back-substitution is given as `linearLUsubstitute` in the Appendix.

Back-substitution is also an $O(n^2)$ process. Therefore in solving $\mathbf{Ax} = \mathbf{b}$ using LU decomposition followed by forward- and back-substitution, the bulk of the computation is in performing the LU decomposition. A detailed analysis shows that while Gauss-Jordan elimination, Gaussian elimination and LU decomposition are all $O(n^3)$ processes, Gauss-Jordan elimination takes about 3 times as many, and Gaussian elimination about 1.5 times as many operations as LU decomposition. Hence, LU decomposition is preferable.

Moreover, in a matrix-vector equation of the form $\mathbf{Ax}=\mathbf{b}$, typically **A** represents the geometry, structure and/or physical properties of the system being analyzed, **x** is the response of the system (displacements, temperature, etc.) and **b** represents the "inputs" (such as forces or heat flows). It is not uncommon to want to calculate the system response for several different inputs (different **b** vectors). The beauty of the LU decomposition is that it only needs to be performed once at a "computational cost" of $O(n^3)$. The solution **x** for a new input **b** then simply requires application of the forward-substitution and back-substitution algorithms with are only $O(n^2)$. This is a tremendous benefit over repeatedly solving $\mathbf{Ax}=\mathbf{b}$ for each **b**.

2.2 Partial pivoting

In the LU decomposition algorithm we divide by the pivots a_{kk}. Obviously this fails if one of these terms is zero, and it produces poor results if one of these terms is very small. As with Gaussian elimination, the solution is to perform partial (or row) pivoting.

In the previous lecture we pointed out that swapping rows of the augmented matrix $\tilde{\mathbf{A}}=(\mathbf{A},\mathbf{b})$ does not change the solution vector **x**. This is because the **b** values are swapped along with the **A** values. LU decomposition, however, does not consider the **b** vector. Rather it factors **A** into a form in which we can quickly solve for **x** given *any* **b** vector. Therefore we need to keep track of these row swaps so that we can apply them to the vector **b** before performing forward- and back-substitution. The result is sometimes called LUP decomposition. We write

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

where **P** is a *permutation matrix*. As an example, consider

```
-->A = [1,2,3;4,5,6;7,8,9]
 A  =

    1.    2.    3.
    4.    5.    6.
    7.    8.    9.

-->P = [0,1,0;0,0,1;1,0,0]
 P  =

    0.    1.    0.
    0.    0.    1.
    1.    0.    0.

-->P*A
 ans  =

    4.    5.    6.
    7.    8.    9.
    1.    2.    3.
```

P is an identity matrix in which the rows have been swapped. In this case the identity matrix rows 1,2,3 are reordered 2,3,1. The product **PA** then simply reorders the rows of **A** in the same manner. With row pivoting, we actually compute the LU decomposition of **PA**. Multiplying both sides of $\mathbf{Ax}=\mathbf{b}$ by **P** we have $\mathbf{PAx}=\mathbf{Pb}$, and then

$$\mathbf{LUx}=\mathbf{Pb}$$

This shows that we need only apply the permutation to a **b** vector and then we can obtain the

correct **x** using forward- and back-substitution. Instead of generating the matrix **P** we can form a permutation vector **p** which lists the reordering of the rows of **A**, for example

$$\mathbf{p}=\begin{pmatrix}2\\3\\1\end{pmatrix}$$

This tells us to perform the reordering

$$\mathbf{b}\leftarrow\begin{pmatrix}b_2\\b_3\\b_1\end{pmatrix}$$

before performing forward- and back-substitution. In Scilab/Matlab, if b is a vector and p is a permutation vector then $b(p)$ is the permuted version of b. A Scilab implementation of the LUP decomposition algorithm is given in the Appendix as `linearLUP`. An example of using this functions to solve a linear system is

```
-->A = [1,2,3;4,5,-6;7,-8,-9];
-->b = [1;2;3];
-->[LU,p] = linearLUP(A);
-->x = linearLUsubstitute(LU,b(p));
-->disp(x);
```

which produces output

```
    0.6078431
    0.0392157
    0.1045752
```

As a check

```
-->A*x
 ans  =

    1.
    2.
    3.
```

The same result is obtained with

```
-->LU = linearLU(A);
-->x = linearLUsubstitute(LU,b);
```

3 Matrix determinant

In your linear algebra course you learned about the determinant of a square matrix, which we will write as $\det\mathbf{A}$. The determinant is very important theoretically. Numerically it does not find much application, but on occasion you may want to compute it. We know from the properties of the determinant that if $\mathbf{A}=\mathbf{LUP}$ then

$$\det\mathbf{A}=\det\mathbf{L}\det\mathbf{U}\det\mathbf{P}$$

The determinant of a permutation matrix is ±1 with the sign depending on whether **P** represents an even or odd number of row swaps. The determinant of a triangular matrix is simply the

product of its diagonal elements. Therefore $\det \mathbf{L} = 1$ and

$$\det \mathbf{A} = \det \mathbf{L} \det \mathbf{U} \det \mathbf{P} = \pm u_{11} u_{22} u_{33} \cdots u_{nn}$$

This is the "best" way to numerically calculate the determinant of a square matrix **A**. In Scilab/Matlab the determinant can be calculated using the command `det(A)`.

4 The backslash operator \

In both Scilab and Matlab the most common way to solve the square linear system $\mathbf{Ax} = \mathbf{b}$ is using the "backslash operator"

```
x = A\b
```

You can think of `A\b` as "A inverse times b," or "A left-divided into b." If **A** is non-singular, Scilab uses LUP decomposition with partial pivoting and backward and forward substitution to solve for **x**. If **A** is singular or "close" to singular ("poorly conditioned"), or is not square, then a "least-squares" solution is computed. We will study least squares in a future lecture.

In Scilab/Matlab, "built-in" functions run *much* faster than anything we can code and run ourselves. This is because built-in functions are written in a compiled language (such as C), compiled and then called at run time. They run with the much greater speed of compiled code whereas anything we write must run in the Scilab/Matlab interpreter environment. It is possible for us to write compiled functions in C and have Scilab/Matlab call them, but this is an advanced topic.

Repeating our previous example

```
-->A = [1,2,3;4,5,-6;7,-8,-9];
-->b = [1;2;3];
-->x = A\b
 x  =

    0.6078431
    0.0392157
    0.1045752
```

5 Singular matrices

Even with pivoting, all of the methods we have studied may fail to solve $\mathbf{Ax} = \mathbf{b}$ because *there may be no solution!* In fact we know that if $\det \mathbf{A} = 0$ then **A** is a *singular matrix*, $\mathbf{A}^{-1}$ fails to exist, and it is therefore impossible to calculate $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Consider the following example

```
-->A = [1,2,3;4,5,9;6,7,13]
 A  =

    1.    2.    3.
    4.    5.    9.
    6.    7.    13.

-->[A,p] = linearLUP(A)
 p  =

    3.
    1.
    2.
```

```
 A  =

    6.          7.          13.
    0.1666667   0.8333333   0.8333333
    0.6666667   0.4         0.
```

Notice that $a_{33}=u_{33}=0$, even with pivoting. It follows that $\det \mathbf{A}=0$ and the matrix is singular. If we try to perform forward- and back-substitution we will fail when we come to the step where we are supposed to divide by u_{33}.

The problem with this **A** is that the third column is the sum of the first two columns; the three columns are *linearly dependent*, and the matrix is singular. There is no solution to $\mathbf{Ax}=\mathbf{b}$. If we ask Scilab to find one we are told

```
-->x = A\[1;2;3]
Warning :
matrix is close to singular or badly scaled. rcond =    0.0000D+00
computing least squares solution. (see lsq).

 x  =

    0.
    0.7105263
  - 0.1578947

-->A*x
 ans  =

    0.9473684
    2.1315789
    2.9210526
```

We're warned that the matrix is "close to singular" and given a "least squares solution." Testing **Ax** we get something close to, but no equal to **b**. There is no true solution, but Scilab gives us "the best that can be achieved" in its place.

6 References

1. Golub, G.H. and C.F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, 1983, ISBN: 0-8018-3011-7.

7 Appendix – Scilab code

7.1 LU decomposition

```
0001  ////////////////////////////////////////////////////////////////////
0002  // linearLU.sci
0003  // 2014-06-24, Scott Hudson, for pedagogic purposes
0004  // Given an n-by-n matrix A, calculate the LU decomposition
0005  // A = LU where U is upper triangular and L is unit lower triangular.
0006  // A is replaced by the elements of L and U.
0007  // No pivoting is performed.
0008  ////////////////////////////////////////////////////////////////////
0009  function A=linearLU(A)
0010    n = size(A,1); //A is n-by-n
0011    for k=1:n-1
0012      for i=k+1:n
0013        A(i,k) = A(i,k)/A(k,k);
0014        for j=k+1:n
0015          A(i,j) = A(i,j)-A(i,k)*A(k,j);
0016        end
0017      end
0018    end
0019  endfunction
```

7.2 Forward and backward substitution

```
0001  ////////////////////////////////////////////////////////////////////
0002  // linearLUsubstitute.sci
0003  // 2014-06-24, Scott Hudson, for pedagogic purposes
0004  // A has been replaced by its LU decomposition. This function applies
0005  // forward- and back-substitution to solve Ax=b. Note that if LUP
0006  // decomposition was performed then b(p) should be used as the b
0007  // argument where p is the permutation vector.
0008  ////////////////////////////////////////////////////////////////////
0009  function x=linearLUsubstitute(A, b)
0010    n = length(b);
0011    y = zeros(b);
0012    for i=1:n //forward-substitution
0013      y(i) = b(i);
0014      for j=1:i-1
0015        y(i) = y(i)-A(i,j)*y(j); //a(i,j) = l(i,j) for j<i
0016      end
0017    end
0018    x = zeros(b);
0019    for i=n:-1:1 //back-substitution
0020      x(i) = y(i);
0021      for j=i+1:n
0022        x(i) = x(i)-A(i,j)*x(j); //a(i,j) = u(i,j) for j>i
0023      end
0024      x(i) = x(i)/A(i,i);
0025    end
0026  endfunction
```

7.3 LU decomposition with partial pivoting

```
0001  ////////////////////////////////////////////////////////////////////
0002  // linearLUP.sci
0003  // 2014-06-24, Scott Hudson, for pedagogic purposes
0004  // Given an n-by-n matrix A, calculate the LU decomposition
0005  // A = LU where U is upper triangular and L is unit lower triangular.
0006  // A is overwritten by LU. Partial pivoting is performed.
0007  // Vector p lists the rearrangement of the rows of A. Given a
0008  // vector b, the solution to A*x=b is the solution to L*U*x=b(p).
0009  ////////////////////////////////////////////////////////////////////
0010  function [A, p]=linearLUP(A)
0011    n = size(A,1); //a is n-by-n
0012    //Replace A with its LU decomposition
0013    p = [1:n]';
0014    for k=1:n-1 //k indexes the pivot row
0015      //pivoting - find largest abs() in column k
0016      amax = abs(A(k,k));
0017      imax = k;
0018      for i=k+1:n
0019        if abs(A(i,k))>amax
0020          amax = abs(A(i,k));
0021          imax = i;
0022        end
0023      end
0024      if (imax~=k) //we found a larger pivot
0025        w = A(k,:); //copy row k
0026        A(k,:) = A(imax,:); //replace it with row imax
0027        A(imax,:) = w; //replace row imax with original row k
0028        t = p(k); //perform same swap of elements of p
0029        p(k) = p(imax);
0030        p(imax) = t;
0031      end
0032      //pivoting complete, continue with LU decomposition
0033      for i=k+1:n
0034        A(i,k) = A(i,k)/A(k,k);
0035        for j=k+1:n
0036          A(i,j) = A(i,j)-A(i,k)*A(k,j);
0037        end
0038      end
0039    end
0040  endfunction
```

8 Appendix – Matlab code

8.1 LU decomposition

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% linearLU.m
%% 2014-06-24, Scott Hudson, for pedagogic purposes
%% Given an n-by-n matrix A, calculate the LU decomposition
%% A = LU where U is upper triangular and L is unit lower triangular.
%% A is replaced by the elements of L and U.
%% No pivoting is performed.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function A = linearLU(A)
  n = size(A,1); %%A is n-by-n
  for k=1:n-1
    for i=k+1:n
      A(i,k) = A(i,k)/A(k,k);
      for j=k+1:n
        A(i,j) = A(i,j)-A(i,k)*A(k,j);
      end
    end
  end
end
```

8.2 Forward and backward substitution

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% linearLUsubstitute.m
%% 2014-06-24, Scott Hudson, for pedagogic purposes
%% A has been replaced by its LU decomposition. This function applies
%% forward- and back-substitution to solve Ax=b. Note that if LUP
%% decomposition was performed then b(p) should be used as the b
%% argument where p is the permutation vector.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function x = linearLUsubstitute(A,b)
  n = length(b);
  y = zeros(size(b));
  for i=1:n %%forward-substitution
    y(i) = b(i);
    for j=1:i-1
      y(i) = y(i)-A(i,j)*y(j); %%a(i,j) = l(i,j) for j<i
    end
  end
  x = zeros(size(b));
  for i=n:-1:1 %%back-substitution
    x(i) = y(i);
    for j=i+1:n
      x(i) = x(i)-A(i,j)*x(j); %%a(i,j) = u(i,j) for j>i
    end
    x(i) = x(i)/A(i,i);
  end
end
```

8.3 LU decomposition with partial pivoting

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% linearLUP.m
%% 2014-06-24, Scott Hudson, for pedagogic purposes
%% Given an n-by-n matrix A, calculate the LU decomposition
%% A = LU where U is upper triangular and L is unit lower triangular.
%% A is overwritten by LU. Partial pivoting is performed.
%% Vector p lists the rearrangement of the rows of A. Given a
%% vector b, the solution to A*x=b is the solution to L*U*x=b(p).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [A,p] = linearLUP(A)
  n = size(A,1); %%a is n-by-n
  %%Replace A with its LU decomposition
  p = [1:n]';
  for k=1:n-1 %%k indexes the pivot row
    %%pivoting - find largest abs() in column k
    amax = abs(A(k,k));
    imax = k;
    for i=k+1:n
      if abs(A(i,k))>amax
        amax = abs(A(i,k));
        imax = i;
      end
    end
    if (imax~=k) %%we found a larger pivot
      w = A(k,:); %%copy row k
      A(k,:) = A(imax,:); %%replace it with row imax
      A(imax,:) = w; %%replace row imax with original row k
      t = p(k); %%perform same swap of elements of p
      p(k) = p(imax);
      p(imax) = t;
    end
    %%pivoting complete, continue with LU decomposition
    for i=k+1:n
      A(i,k) = A(i,k)/A(k,k);
      for j=k+1:n
        A(i,j) = A(i,j)-A(i,k)*A(k,j);
      end
    end
  end
end
```