Lecture 11

Linear systems of equations I

1 Introduction

In this lecture we consider ways to solve a linear system $\mathbf{Ax}=\mathbf{b}$ for $\mathbf{x}$ when given $\mathbf{A}$ and $\mathbf{b}$. Writing out the components our system has the form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \tag{1}$$

Each row of $\mathbf{A}$ corresponds to a single linear equation in the unknown x values. The i^{th} equation is

$$a_{i1}x_1+a_{i2}x_2+\cdots+a_{in}x_n=b_i$$

and the j^{th} equation is

$$a_{j1}x_1+a_{j2}x_2+\cdots+a_{jn}x_n=b_j$$

We will use the following facts to transform the system (1) into a form in which the solution is trivially apparent or at least can be easily calculated.

Fact 1: We can scale any equation by a non-zero constant ($c\neq 0$) without changing the solution

$$a_{i1}x_1+a_{i2}x_2+\cdots+a_{in}x_n=b_i \quad \leftarrow \quad ca_{i1}x_1+ca_{i2}x_2+\cdots+ca_{in}x_n=cb_i$$

Fact 2: We can replace an equation by its sum or difference with another equation without changing the solution

$$a_{i1}x_1+a_{i2}x_2+\cdots+a_{in}x_n=b_i \quad \leftarrow \quad (a_{i1}+a_{j1})x_1+(a_{i2}+a_{j2})x_2+\cdots+(a_{in}+a_{jn})x_n=b_i+b_j$$

2 Gauss-Jordan elimination

If $\mathbf{A}$ was the identity matrix

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \tag{2}$$

the solution would be trivial

$$x_k=b_k$$

Gauss-Jordan elimination is a process to convert an arbitrary system (1) into the trivial system (2). Since we want to end up with $a_{11}=1$, we use Fact 1 to multiply the first row of **A** and **b** by $1/a_{11}$

$$a_{1j} \leftarrow a_{1j}/a_{11},\ j=1,2,\ldots,n\ ,\ b_1 \leftarrow b_1/a_{11}$$

to obtain the form

$$\begin{pmatrix} 1 & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

(note a_{12}, a_{13} etc. will have changed values). Now we use Fact 2 to eliminate the elements $a_{21}, a_{31}, \ldots, a_{n1}$ in the first column by subtracting a_{i1} times the first row from the i^{th} row

$$a_{ij} \leftarrow a_{ij} - a_{i1}a_{1j},\ j=1,2,\ldots,n\ ,\ b_i \leftarrow b_i - a_{i1}b_1$$

for $i=2,3,\ldots,n$ resulting in

$$\begin{pmatrix} 1 & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

The first column is now in the desired form. Let's move to the second column. Since we want to end up with $a_{22}=1$, use Fact 1 to multiply the second row by $1/a_{22}$

$$a_{2j} \leftarrow a_{2j}/a_{22},\ j=2,3,\ldots,n\ ,\ b_2 \leftarrow b_2/a_{22}$$

Note that we don't bother with $j=1$ since $a_{21}=0$. Then we use Fact 2 to eliminate all elements a_{i2}, $i \neq 2$

$$a_{ij} \leftarrow a_{ij} - a_{i2}a_{2j},\ j=2,3,\ldots,n\ ,\ b_i \leftarrow b_i - a_{i2}b_2$$

Again we don't bother with $j=1$ since $a_{21}=0$. We end up with

$$\begin{pmatrix} 1 & 0 & a_{13} & \cdots & a_{1n} \\ 0 & 1 & a_{23} & \cdots & a_{2n} \\ 0 & 0 & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & a_{n3} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

We now move on to the third column and so on, continuing until we have the form shown in (2).

Notice that the element b_i is transformed in the same manner as the elements a_{ij}. This suggests that we form an *augmented matrix* as **A** plus **b** as an extra column and then simply transform the *a* values as a whole. Let's call the augmented matrix $\widetilde{\mathbf{A}}$. Then

$$\widetilde{\mathbf{A}}=\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} & b_2 \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} & b_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} & b_n \end{pmatrix}=\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & a_{1\,n+1} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} & a_{2\,n+1} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} & a_{3\,n+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} & a_{n\,n+1} \end{pmatrix}$$

We simply replace $a_{ij}\leftarrow a_{ij}-a_{i2}a_{2j},\ j=2,3,\ldots,n$ by $a_{ij}\leftarrow a_{ij}-a_{i2}a_{2j},\ j=2,3,\ldots,n+1$ and the *b* values are automatically taken care of. Our algorithm can now be stated as

Gauss-Jordan elimination algorithm

Form the augmented matrix $\widetilde{\mathbf{A}}=(\mathbf{A}\,,\mathbf{b})$

```
for k = 1,2,...,n
    for j = n+1, n, n-1,...,k
        a_kj ← a_kj / a_kk
    for i = 1,2,...,n
        if i ≠ k  (we don't want a row to "eliminate" itself)
            for j = n+1, n, n-1,...,k
                a_ij ← a_ij - a_ik a_kj
```

When the loops are complete, the last column of $\widetilde{\mathbf{A}}$ will be the solution vector **x**.

Example 1: Consider the system **Ax=b** *with*

$$\mathbf{A}=\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix},\ \mathbf{b}=\begin{pmatrix} 4 \\ 10 \end{pmatrix}$$

We form the augmented matrix

$$\widetilde{\mathbf{A}}=\begin{pmatrix} 1 & 2 & 4 \\ 3 & 4 & 10 \end{pmatrix}$$

The first pivot, a_{11}*, is already 1. We now subtract 3 times row 1 from row 2*

$$\widetilde{\mathbf{A}}=\begin{pmatrix} 1 & 2 & 4 \\ 0 & -2 & -2 \end{pmatrix}$$

The second pivot is $a_{22}=-2$*. We divide row 2 by this to get*

$$\widetilde{\mathbf{A}}=\begin{pmatrix} 1 & 2 & 4 \\ 0 & 1 & 1 \end{pmatrix}$$

We now subtract 2 times row 2 from row 1 to obtain

$$\widetilde{\mathbf{A}}=\begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix}$$

The last column is the solution **x**.

The function `linearGaussJordan` shown in the Appendix is a Scilab implementation of this algorithm.

Now, suppose we solve the following problems one at a time using this method

$$\mathbf{A}\,\mathbf{x}_1=\begin{pmatrix}1\\0\\0\\\vdots\\0\end{pmatrix}\ ,\ \mathbf{A}\,\mathbf{x}_2=\begin{pmatrix}0\\1\\0\\\vdots\\0\end{pmatrix}\ ,\ \mathbf{A}\,\mathbf{x}_3=\begin{pmatrix}0\\0\\1\\\vdots\\0\end{pmatrix}\ ,\ \ldots\ ,\ \mathbf{A}\,\mathbf{x}_n=\begin{pmatrix}0\\0\\0\\\vdots\\1\end{pmatrix}$$

Then

$$\mathbf{A}\left(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3,\ldots,\mathbf{x}_n\right)=\begin{pmatrix}1&0&0&\cdots&0\\0&1&0&\cdots&0\\0&0&1&\cdots&0\\\vdots&\vdots&\vdots&\ddots&0\\0&0&0&\cdots&1\end{pmatrix}$$

or $\mathbf{AX}=\mathbf{I}$ where $\mathbf{X}$ is the matrix with column vectors $\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n$. It follows that $\mathbf{X}$ is the inverse of $\mathbf{A}$. We don't have to solve n problems one at a time, however. The operations we perform with the a_{ij} values will be the same regardless of the right-hand vector $\mathbf{b}$. Therefore, we can simply form the augmented matrix

$$\widetilde{\mathbf{A}}=\begin{pmatrix}a_{11}&a_{12}&a_{13}&\cdots&a_{1n}&1&0&0&\cdots&0\\a_{21}&a_{22}&a_{23}&\cdots&a_{2n}&0&1&0&\cdots&0\\a_{31}&a_{32}&a_{33}&\cdots&a_{3n}&0&0&1&\cdots&0\\\vdots&\vdots&\vdots&\ddots&\vdots&\vdots&\vdots&\vdots&\ddots&\vdots\\a_{n1}&a_{n2}&a_{n3}&\cdots&a_{nn}&0&0&0&\cdots&1\end{pmatrix}$$

and perform Gauss-Jordan elimination on $\widetilde{\mathbf{A}}$ to effectively solve these n problems "in parallel." When complete, the last n columns of $\widetilde{\mathbf{A}}$ will be the inverse $\mathbf{A}^{-1}$. In the Appendix, program `linearGaussJordanInverse` gives a Scilab implementation of this algorithm. Note the slight differences between this and `linearGaussJordan`.

Notice that the Gauss-Jordan elimination algorithm consists of three nested levels of for loops. Each (runs roughly speaking) over on order of n values. It follows that the total number of operations is on the order of $n\cdot n\cdot n=n^3$. This gives a measure of the computational complexity and therefore of the amount of cpu time required for the algorithm. For this reason you will often see statements of the form "matrix inversion is an $O(n^3)$ operation, where the "big O" represents "order of."

3 Gaussian elimination

Gauss-Jordan elimination is a logical way to solve $\mathbf{Ax}=\mathbf{b}$ or to find $\mathbf{A}^{-1}$. However, there are faster and more robust methods. In particular we can solve $\mathbf{Ax}=\mathbf{b}$ with only about 1/3 the number of operations using the so-called LU decomposition that we will develop in the next lecture. The price we pay is that getting the solution is more convoluted than it is for Gauss-

Jordan elimination; it involves various "substitution" operations. Here we'll introduce this idea by considering the so-called Gaussian elimination algorithm.

In Gauss-Jordan elimination we "zero-out" all elements of **A** except those on the diagonal, and we normalize the diagonal elements to 1. In Gaussian elimination we don't bother with the elements above the diagonal; we only zero-out the elements below the diagonal. We also don't bother to normalize the diagonal elements to 1. The result is a system in the form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & 0 & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \tag{3}$$

The matrix is in *upper-triangular* form; all elements below the diagonal are zero. The algorithm to achieve this is an obvious variation of Gauss-Jordan elimination.

Gaussian elimination algorithm

Form the augmented matrix $\widetilde{\mathbf{A}} = (\mathbf{A}, \mathbf{b})$

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

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

    for $j = n+1, n, n-1, \ldots, k$

      $a_{ij} \leftarrow a_{ij} - a_{kj} a_{ik} / a_{kk}$

Since it has three nested for loops, each running over on the order of n values, this is also an $O(n^3)$ process. A problem with (3) is that the solution is not obvious (as it is for (2)) *except* for the last row which gives us

$$a_{nn} x_n = b_n \rightarrow x_n = b_n / a_{nn}$$

So, x_n is easy to get. The next-to-last equation is

$$a_{n-1,n-1} x_{n-1} + a_{n-1,n} x_n = b_{n-1}$$

But, we already know x_n, so we can solve for

$$x_{n-1} = \frac{1}{a_{n-1,n-1}} \left(b_{n-1} - a_{n-1,n} x_n \right)$$

The second-to-last equation is

$$a_{n-2,n-2} x_{n-2} + a_{n-2,n-1} x_{n-1} + a_{n-2,n} x_n = b_{n-2}$$

and since we've already solved for x_{n-1}, x_n we can solve for the single remaining unknown

$$x_{n-2} = \frac{1}{a_{n-2,n-2}} \left(b_{n-2} - \left[a_{n-2,n-1} x_{n-1} + a_{n-2,n} x_n \right] \right)$$

Continuing to work our way up row-by-row we have the following algorithm.

Back-substitution algorithm

If **A** is upper-triangular, the solution to $\mathbf{Ax}=\mathbf{b}$ is given by

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

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

A Scilab implementation of Gaussian elimination followed by back-substitution appears in the Appendix as `linearGaussian`. This algorithm has two nested for loops (*i* and *j*) and is therefore an $O(n^2)$ process. This tells us that, at least for large matrices (large *n* values), back-substitution is much faster than Gaussian elimination or Gauss-Jordan elimination. The fact that we've had to add this step, therefore, is not of much concern regarding cpu time.

4 The need for "pivoting"

Looking over the Gauss-Jordan and Gaussian elimination algorithms it is clear that the only way they can "go wrong" is if $a_{kk}=0$ for some value of *k* since we would then be trying to divide by zero. Note that when we get to this point in either algorithm, in general a_{kk} *will not* have the same value it had in the original matrix **A**, but will have been modified by various subtractions performed in the algorithm. Therefore, even if none of the diagonal elements of **A** are zero, we can still end up with $a_{kk}=0$ at some stage in the algorithm. Furthermore, even if a_{kk} is non-zero but very small this will still cause problems by amplifying accumulated round-off error by a large factor of $1/a_{kk}$.

The diagonal elements a_{kk} are called *pivots*, and the way to avoid the problem of a small or zero pivot is through *pivoting* – swapping two rows or two columns of the augmented matrix. Pivoting is a critical requirements for a robust algorithm, and without it the code given in the Appendix is not generally reliable.

Let's again consider the general problem $\mathbf{Ax}=\mathbf{b}$ written out in component form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}=\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

Each row represents a single linear equation in the unknowns. In what way will the solution change if we swap, say, the first and third rows of both **A** and **b**?

$$\begin{pmatrix} a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}=\begin{pmatrix} b_3 \\ b_2 \\ b_1 \\ \vdots \\ b_n \end{pmatrix}$$

The answer is that it won't. We've just rearranged the same *n* equations in *n* unknowns. It doesn't

matter in what order we write them; the solution will remain the same.

Fact 3: Any two rows of the augmented matrix $\widetilde{\mathbf{A}}$ can be swapped without changing the solution vector **x**.

What about swapping columns? Say we swap the first and third columns of **A**.

$$\begin{pmatrix} a_{13} & a_{12} & a_{11} & \cdots & a_{1n} \\ a_{23} & a_{22} & a_{21} & \cdots & a_{2n} \\ a_{33} & a_{32} & a_{31} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n3} & a_{n2} & a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

Thinking of **Ax** as a linear combination of the columns of **A**, the i^{th} column of **A** gets multiplied by x_i. Swapping the columns is equivalent to relabeling the two corresponding components of **x**. In other words if we write the system as

$$\begin{pmatrix} a_{13} & a_{12} & a_{11} & \cdots & a_{1n} \\ a_{23} & a_{22} & a_{21} & \cdots & a_{2n} \\ a_{33} & a_{32} & a_{31} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n3} & a_{n2} & a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_3 \\ x_2 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

it is just a rearrangement of our original system. Provided we remember to swap x_3, x_1 at the end this system will give us the same solution as the original system.

Fact 4: Any two columns of the matrix **A** can be swapped without changing the solution vector **x**, *except* for a reordering of its elements.

The idea of "full pivoting" is that when we come to the place in our algorithm where we will be dividing by a_{kk}, we examine all the ways we can swap rows and columns so as to replace a_{kk} by the largest magnitude value possible. Provided we do the bookkeeping correctly, this won't change our solution but it will make it numerically robust.

The difficultly with full pivoting is that for each loop there are a lot of possible row-plus-column swaps, and finding the best involves many absolute-value tests in each loop. Moreover, if we swap columns we need to keep track of this in order to "untangle" the **x** values at the end. However, if we limit ourselves to a single row swap, which is called "partial pivoting" or "row pivoting," then we need only check the magnitudes of the elements in the k^{th} column, and we don't have to keep track of the swaps. Experience shows that partial pivoting is sufficient to produce reliable algorithms. Almost always in practice, therefore, only partial pivoting is used. From here on when we talk about "pivoting" we mean "partial pivoting" – swapping rows only. Adding partial pivoting to the `linearGaussian` code results in the function `linearGaussianPivot` given in the Appendix. This is a serviceable routine.

5 References

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

6 Appendix – Scilab code

6.1 Gauss-Jordan elimination

```
0001  ////////////////////////////////////////////////////////////////////
0002  // linearGaussJordan.sci
0003  // 2014-06-23, Scott Hudson, for pedagogic purposes
0004  // Solves Ax=b for x using Gauss-Jordan elimination.
0005  // No pivoting is performed.
0006  ////////////////////////////////////////////////////////////////////
0007  function x=linearGaussJordan(A, b)
0008    n = length(b);
0009    A = [A,b]; //form augmented matrix
0010    for k=1:n //A(k,k) is the pivot
0011      for j=n+1:-1:k //normalize row k so A(k,k)=1
0012        A(k,j) = A(k,j)/A(k,k);
0013      end
0014      for i=1:n //eliminate a(i,k) for all i~=k
0015        if (i~=k) //a Pivot does not eliminate itself
0016          for j=n+1:-1:k
0017            A(i,j) = A(i,j)-A(k,j)*A(i,k);
0018          end
0019        end
0020      end //i loop
0021    end //k loop
0022    x = A(:,n+1); //last column of augmented matrix is now x
0023  endfunction
```

6.2 Matrix inverse using Gauss-Jordan elimination

```
0001  ////////////////////////////////////////////////////////////////////
0002  // linearGaussJordanInverse.sci
0003  // 2014-06-23, Scott Hudson, for pedagogic purposes
0004  // Forms inverse of matrix A using Gauss-Jordan elimination.
0005  // No pivoting is performed.
0006  ////////////////////////////////////////////////////////////////////
0007  function Ainv=linearGaussJordanInverse(A)
0008    n = size(A,'r');
0009    A = [A,eye(A)]; //form augmented matrix
0010    for k=1:n //A(k,k) is the pivot
0011      for j=2*n:-1:k //normalize row k so A(k,k)=1
0012        A(k,j) = A(k,j)/A(k,k);
0013      end
0014      for i=1:n //eliminate a(i,k) for all i~=k
0015        if (i~=k) //a Pivot does not eliminate itself
0016          for j=2*n:-1:k
0017            A(i,j) = A(i,j)-A(k,j)*A(i,k);
0018          end
0019        end
0020      end //i loop
0021    end //k loop
0022    Ainv = A(:,n+1:2*n); //last column of augmented matrix is now x
0023  endfunction
```

6.3 Gaussian elimination

```
0001  ////////////////////////////////////////////////////////////////////
0002  // linearGaussian.sci
0003  // 2014-06-25, Scott Hudson, for pedagogic purposes
0004  // Solves Ax=b for x using Gaussian elimination and backsubstitution.
0005  // No pivoting is performed.
0006  ////////////////////////////////////////////////////////////////////
0007  function x=linearGaussian(A, b)
0008    n = length(b);
0009    A = [A,b]; //form augmented matrix
0010    //Gaussian elimination loop
0011    for k=1:n-1 //A(k,k) is the pivot
0012      for i=k+1:n //eliminate a(i,k) for all i>k
0013        for j=n+1:-1:k
0014          A(i,j) = A(i,j)-A(k,j)*A(i,k)/A(k,k);
0015        end
0016      end //i loop
0017    end //k loop
0018    //Backsubstitution loop
0019    x = zeros(n,1);
0020    x(n) = A(n,n+1)/A(n,n);
0021    for i=n-1:-1:1
0022      x(i) = A(i,n+1);
0023      for j=i+1:n
0024        x(i) = x(i)-A(i,j)*x(j);
0025      end
0026      x(i) = x(i)/A(i,i);
0027    end
0028  endfunction
```

6.4 Gaussian elimination with partial pivoting

```
0001  ////////////////////////////////////////////////////////////////////
0002  // linearGaussianPivot.sci
0003  // 2014-06-25, Scott Hudson, for pedagogic purposes
0004  // Solves Ax=b for x using Gaussian elimination and backsubstitution.
0005  // Partial pivoting is performed.
0006  ////////////////////////////////////////////////////////////////////
0007  function x=linearGaussianPivot(A, b)
0008    n = length(b);
0009    A = [A,b]; //form augmented matrix
0010    //Gaussian elimination loop
0011    for k=1:n-1 //A(k,k) is the pivot
0012      //see if there is a larger pivot below this in the kth column
0013      Amax = abs(A(k,k));
0014      imax = k;
0015      for i=k+1:n
0016        if abs(A(i,k))>Amax
0017          Amax = abs(A(i,k));
0018          imax = i;
0019        end
0020      end
0021      if (imax~=k) //we found a larger pivot, swap rows
0022        w = A(k,:); //copy the kth row
0023        A(k,:) = A(imax,:); //replace it with the imax row
0024        A(imax,:) = w; //replace the imax row with the original kth row
0025      end
0026      //pivoting complete
0027      for i=k+1:n //eliminate a(i,k) for all i>k
0028        for j=n+1:-1:k
0029          A(i,j) = A(i,j)-A(k,j)*A(i,k)/A(k,k);
0030        end
0031      end //i loop
0032    end //k loop
0033    //Backsubstitution loop
0034    x = zeros(n,1);
0035    x(n) = A(n,n+1)/A(n,n);
0036    for i=n-1:-1:1
0037      x(i) = A(i,n+1);
0038      for j=i+1:n
0039        x(i) = x(i)-A(i,j)*x(j);
0040      end
0041      x(i) = x(i)/A(i,i);
0042    end
0043  endfunction
```

7 Appendix – Matlab code

7.1 Gauss-Jordan elimination

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% linearGaussJordan.sci
%% 2014-06-23, Scott Hudson, for pedagogic purposes
%% Solves Ax=b for x using Gauss-Jordan elimination.
%% No pivoting is performed.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function x = linearGaussJordan(A,b)
  n = length(b);
  A = [A,b]; %%form augmented matrix
  for k=1:n %%A(k,k) is the pivot
    for j=n+1:-1:k %%normalize row k so A(k,k)=1
      A(k,j) = A(k,j)/A(k,k);
    end
    for i=1:n %%eliminate a(i,k) for all i~=k
      if (i~=k) %%a Pivot does not eliminate itself
        for j=n+1:-1:k
          A(i,j) = A(i,j)-A(k,j)*A(i,k);
        end
      end
    end %%i loop
  end %%k loop
  x = A(:,n+1); %%last column of augmented matrix is now x
end
```

7.2 Matrix inverse using Gauss-Jordan elimination

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% linearGaussJordanInverse.sci
%% 2014-06-23, Scott Hudson, for pedagogic purposes
%% Forms inverse of matrix A using Gauss-Jordan elimination.
%% No pivoting is performed.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function Ainv = linearGaussJordanInverse(A)
  n = size(A,1);
  A = [A,eye(size(A))]; %%form augmented matrix
  for k=1:n %%A(k,k) is the pivot
    for j=2*n:-1:k %%normalize row k so A(k,k)=1
      A(k,j) = A(k,j)/A(k,k);
    end
    for i=1:n %%eliminate a(i,k) for all i~=k
      if (i~=k) %%a Pivot does not eliminate itself
        for j=2*n:-1:k
          A(i,j) = A(i,j)-A(k,j)*A(i,k);
        end
      end
    end %%i loop
  end %%k loop
  Ainv = A(:,n+1:2*n); %%last column of augmented matrix is now x
end
```

7.3 Gaussian elimination

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% linearGaussian.sci
%% 2014-06-25, Scott Hudson, for pedagogic purposes
%% Solves Ax=b for x using Gaussian elimination and backsubstitution.
%% No pivoting is performed.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function x = linearGaussian(A,b)
  n = length(b);
  A = [A,b]; %%form augmented matrix
  %%Gaussian elimination loop
  for k=1:n-1 %%A(k,k) is the pivot
    for i=k+1:n %%eliminate a(i,k) for all i>k
      for j=n+1:-1:k
        A(i,j) = A(i,j)-A(k,j)*A(i,k)/A(k,k);
      end
    end %%i loop
  end %%k loop
  %%Backsubstitution loop
  x = zeros(n,1);
  x(n) = A(n,n+1)/A(n,n);
  for i=n-1:-1:1
    x(i) = A(i,n+1);
    for j=i+1:n
      x(i) = x(i)-A(i,j)*x(j);
    end
    x(i) = x(i)/A(i,i);
  end
end
```

7.4 Gaussian elimination with partial pivoting

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% linearGaussianPivot.sci
%% 2014-06-25, Scott Hudson, for pedagogic purposes
%% Solves Ax=b for x using Gaussian elimination and backsubstitution.
%% Partial pivoting is performed.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function x = linearGaussianPivot(A,b)
  n = length(b);
  A = [A,b]; %%form augmented matrix
  %%Gaussian elimination loop
  for k=1:n-1 %%A(k,k) is the pivot
    %%see if there is a larger pivot below this in the kth column
    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, swap rows
      w = A(k,:); %%copy the kth row
      A(k,:) = A(imax,:); %%replace it with the imax row
      A(imax,:) = w; %%replace the imax row with the original kth row
    end
    %%pivoting complete
    for i=k+1:n %%eliminate a(i,k) for all i>k
      for j=n+1:-1:k
        A(i,j) = A(i,j)-A(k,j)*A(i,k)/A(k,k);
      end
    end %%i loop
  end %%k loop
  %%Backsubstitution loop
  x = zeros(n,1);
  x(n) = A(n,n+1)/A(n,n);
  for i=n-1:-1:1
    x(i) = A(i,n+1);
    for j=i+1:n
      x(i) = x(i)-A(i,j)*x(j);
    end
    x(i) = x(i)/A(i,i);
  end
end
```