Rozdział 1

dd

This page intentionally left blank.

Rozdział 2

dd

This page intentionally left blank.

Rozdział 3

dd

This page intentionally left blank.

Rozdział 4

dd

This page intentionally left blank.

Rozdział 5

dd

This page intentionally left blank.

Rozdział 6

dd

This page intentionally left blank.

Rozdział 7

Matrices and Linear Systems

7.1. DD

7.2. DD

7.3. DD

7.4. DD

7.5. Gaussian Elimination, Pivoting, and LU Factorization

The reader can check that U \ b will give the same result.

A lower triangular system $Lx = b$ can be solved with an analogous algorithm called **forward substitution**. Here we start with the first equation to get x_1, then plug this result into the second equation and solve for x_2, and so on.

EXERCISE FORTHEREADER 7.18: (a) Write a function M-file called `x=fwdsubst (L, b)` , that will input a lower triangular matrix L, a column vector b of the same dimension, and will output the column vector x solution of the system $Lx = b$ solved by forward substitution, (b) Use this program to solve the system $Lx = b$, where $L = U'$, and U and b are as in Example 7.11.

We now introduce the **three elementary row operations (EROs)** that can be performed on augmented matrices.

(i) Multiply a row by a nonzero constant.
(ii) Switch two rows.
(iii) Add a multiple of one row to a different row.

EXAMPLE 7.12: (a) Consider the following augmented matrix:
$Ab = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$. Perform ERO (iii) on this matrix by adding the multiple
-2 times row 1 to row 2
(b) Perform this same ERO on I_3 , the 3×3 identity matrix, to obtain a matrix M, and multiply this matrix M on the left of Ab. What do you get?

SOLUTION: Part (a): -2 times row 1 of Ab is-2[1 2- 3 5]= [-2 -4 6 -10]. Adding this row vector to row 2 of Ab, produces the new matrix:

$$\begin{bmatrix} 1 & 2 & -3 & 5 \\ 0 & 2 & 7 & -11 \\ 0 & 4 & 7 & 8 \end{bmatrix}.$$

Part (b): Performing this same ERO on I_3 produces the matrix $M = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$,

which when multiplied by Ab gives

$$M(Ab) = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & -3 & 5 \\ 2 & 6 & 1 & -1 \\ 0 & 4 & 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 2 & -3 & 5 \\ 0 & 2 & 7 & -11 \\ 0 & 4 & 7 & 8 \end{bmatrix},$$

and we are left with the same matrix that we obtained in part (a). This is no coincidence, as the following theorem shows.

THEOREM 7.4: *(Elementary Matrices)* Let A be any $n \times m$ matrix and $I = I_n$ denote the identity matrix. If B is the matrix obtained from A by performing any particular elementary row operation, and M is the matrix obtained from I by performing this same elementary row operation, then $B = MA$. Also, the matrix M is invertible, and its inverse is the matrix that results from I by performing the inverse elementary row operation on it (i.e., the elementary row operation that will transform M back into I).

Such a matrix M is called an **elementary matrix**. This result is not hard to prove; we refer the reader to any good linear algebra textbook, such as those mentioned in the last section

It is easy to see that any of these EROs, when applied to the augmented matrix of a linear system, will not alter the solution of the system. Indeed, the first ERO corresponds to simply multiplying the first equation by a nonzero constant (the equation still represents the same line, plane, hyperplane, etc.). The second ERO merely changes the order in which the equations are written; this has no effect on the (joint) solution of the system. To see why the third ERO does not alter solutions of the system is a bit more involved, but not difficult. Indeed, suppose for definiteness that a multiple (say, 2) of the first row is added to the second. This corresponds to a new system where all the equations are the same, except for the second, which equals the old second equation plus twice the first equation. Certainly if we have all of $x_1, x_2, \ldots, x_n$ satisfying the original system, then they will satisfy the new system. Conversely, if all of the equations of the new system are solved by $x_1, x_2, \ldots, x_n$, then this already gives all but the second equation of the original system. But the second equation of the old system is gotten by subtracting twice the first equation of the new system from the second equation (of the new system) and so must also hold.

Each of the EROs is easily programmed into an M-file. We do one of them and leave the other two as exercises.

PROGRAM 7.5:

Function M-file for elementary row operation (ii): switching two rows.

```
function B=rowswitch(A,i,j)
->Imputs: a matrix A, and row indicos i and j
->outputs: the matrix gotton from A by interchanging row i and row j
[m,n]=size(A);
if i<l|i>m|j<l|j>m
  error('Invalid index')
end
B=A;
if i==j
  return
end
B(i,:)=A(j,:);
B(j,:)=A(i,:);
```

It may seem redundant to have included that if-branch for detecting invalid indices, since it would seem that no one in their right mind would use the program with, say, an index equaling 10 with an 8×8 matrix. This program, however, might get used to build more elaborate programs, and in such a program it may not always be crystal clear whether some variable expression is a valid index.

EXERCISE FOR THE READER 7.19: Write similar function M-files `B=rowmult (A,i,c)` for ERO (i) and `B=rowcomb (A,i,j,c)` for ERO (iii). The first program will produce the matrix B resulting from A by multiplying the ith row of the latter by c the second program should replace the ith row of A by c times the jth row plus the ith row.

We next illustrate the Gaussian elimination algorithm, the partial pivoting feature, as well as the *LU* decomposition, by means of a simple example. This will give the reader a feel for the main concepts of this section. Afterward we will develop the general algorithms and comment on some consequences of floating point arithmetic. Remember, the goal of Gaussian elimination is to transform, using only EROs, a (nonsingular) system into an equivalent one that is upper triangular; the latter can then be solved by back substitution.

EXAMPLE 7.13: We will solve the following linear system $Ax = b$ using Gaussian elimination (without partial pivoting):

$$\begin{cases} x_1 + 3x_2 - x_3 = 2 \\ 2x_1 + 5x_2 - 2x_3 = 3 \\ 3x_1 + 6x_2 + 9x_3 = 39 \end{cases} .$$

SOLUTION: For convenience, we will work instead with the corresponding augmented matrix Ab for this system $Ax = b$:

$$Ab = \left[\begin{array}{ccc|c} 1 & 3 & -1 & 2 \\ 2 & 5 & -2 & 3 \\ 3 & 6 & 9 & 39 \end{array}\right].$$

For notational convenience, we denote the entries of this or any future augmented matrix as a_{ij}. In computer codes, what is usually done is that at each step the new matrix overwrites the old one to save on memory allocation (not to mention having to invent new variables for unnecessary older matrices). Gaussian elimination starts with the first column, clears out (makes zeros) everything below the main diagonal entry, then proceeds to the second column, and so on.

We begin by zeroing out the entry $a_{2l} = 2$: `Ab=rowcomb (Ab, 1 , 2, -2) .`

$$Ab \to M_1(Ab) \left[\begin{array}{ccc} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \left[\begin{array}{ccc|c} 1 & 3 & -1 & 2 \\ 2 & 5 & -2 & 3 \\ 3 & 6 & 9 & 39 \end{array}\right] = \left[\begin{array}{ccc|c} 1 & 3 & -1 & 2 \\ 0 & -1 & 0 & -1 \\ 3 & 6 & 9 & 39 \end{array}\right],$$

where M_1, is the corresponding elementary matrix as in Theorem 7.4. In the same fashion, we next zero out the next (and last) entry $a_{31} = 3$ of the first column: `Ab=rowcomb(Ab,1,3,-3) .`

$$Ab \to M_2(Ab) \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 0 & 1 \end{array}\right] \left[\begin{array}{ccc|c} 1 & 3 & -1 & 2 \\ 0 & -1 & 0 & -1 \\ 3 & 6 & 9 & 39 \end{array}\right] = \left[\begin{array}{ccc|c} 1 & 3 & -1 & 2 \\ 0 & -1 & 0 & -1 \\ 0 & -3 & 12 & 33 \end{array}\right],$$

We move on to the second column. Here only one entry needs clearing, namely $a_{32} = -3$. We will always use the row with the corresponding diagonal entry to clear out entries below it. Thus, to use the second row to clear out the entry $a_{32} = -3$ in the third row, we should multiply the second row by -3 since $-3 \cdot a_{22} = -3 \cdot (-1) = 3$ added to $a_{32} = -3$ would give zero: `Ab=rowcomb(Ab,2,3,-3)`

$$Ab \to M_3(Ab) \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{array}\right] \left[\begin{array}{ccc|c} 1 & 3 & -1 & 2 \\ 0 & -1 & 0 & -1 \\ 0 & -3 & 12 & 33 \end{array}\right] = \left[\begin{array}{ccc|c} 1 & 3 & -1 & 2 \\ 0 & -1 & 0 & -1 \\ 0 & 0 & 12 & 36 \end{array}\right].$$

We now have an augmented matrix representing an equivalent upper triangular system. This system (and hence our original system) can now be solved by the back substitution algorithm:

```
>> U=Ab(:,1:3); b=Ab(:,4);
>> x=backsubst(U,b)
-> x =    2
     1
     3
```

NOTE: One could have gone further and similarly cleared out the above diagonal entries and then used the first ERO to scale the diagonal entries to each equal one. This is how one gets to the reduced row echelon form.

To obtain the resulting LU decomposition, we form the product of all the elementary matrices that were used in the above Gaussian elimination: $M = M_3M_2M_l$. From what was done, we have that $MA = U$, and hence

$$A = M^{-1}(MA) = M^{-1}U = (M_3M_2M_1)^{-1}U = M_1^{-1}M_2^{-1}M_3^{-1}U \equiv LU,$$

where we have defined $L = M_1^{-1}M_2^{-1}M_3^{1}$.We have used, in the second-to-last equality, the fact that the inverse of a product of invertible matrices is the product of the inverses in the reverse order (see Exercise 7). From Theorem 7.4, each of the inverses M_1^{-1}, M_2^{-1}, and $M_3^{-1}$is also an elementary matrix corresponding to the inverse elementary row operation of the corresponding original elementary matrix; and furthermore Theorem 7.4 tells us how to multiply such matrices to obtain

$$L = M_1^{-1}M_2^{-1}M_3^{-1} \left[\begin{array}{ccc} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \cdot \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 0 & 1 \end{array}\right] \cdot \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{array}\right] = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -3 & 1 \end{array}\right].$$

We now have a factorization of the coefficient matrix A as a product LU of a lower triangular and an upper triangular matrix:

$$A = Ab = \left[\begin{array}{ccc} 1 & 3 & -1 \\ 2 & 5 & -2 \\ 3 & 6 & 9 \end{array}\right] = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 3 & 1 \end{array}\right] \cdot \left[\begin{array}{ccc} 1 & 3 & -1 \\ 0 & -1 & 0 \\ 0 & 0 & 12 \end{array}\right] = LU.$$

This factorization, which easily came from the Gaussian elimination algorithm, is a preliminary form of what is known as the LU factorization of A. Once such a factorization is known, any other (nonsingular) system $Ax = c$, having the same coefficient matrix A, can be easily solved in two steps. To see this, rewrite the system as $LUx = c$. First solve $Ly = c$ by forward substitution (works since L is lower triangular), then solve $Ux = y$ by back substitution (works since U is upper triangular). Then x will be the desired solution (Proof: $Ax = (LU)x = L(Ux) = Ly = c$).

We make some observations. Notice that we used only one of the three EROs to perform Gaussian elimination in the above example. Part of the reason for this is that none of the diagonal entries encountered was zero. If this had happened we would have needed to use the `rowswitch` ERO in order to have nonzero diagonal entries. (This is always possible if the matrix A is nonsingular.) In Gaussian elimination, the diagonal entries that are used to clear out the entries below (using `rowcomb`) are known as **pivots**. The **partial pivoting** feature, which is often implemented in Gaussian elimination, goes a bit further to assure (by switching the row with the pivot with a lower row, if necessary) that the pivot is as large as possible in absolute value. In exact arithmetic, this partial pivoting has no effect whatsoever, but in floating point arithmetic it can most certainly cut back on errors. The reason for this is that if a pivot turned out to be nonzero, but very small, then its row would need to be multiplied by very large numbers to clear out moderately sized numbers below the pivot. This may cause other numbers in the pivot's row to get multiplied into very large numbers that, when mixed with much smaller numbers, can lead to floating point errors. We will soon give an example to demonstrate this phenomenon.

EXAMPLE 7.14: Solve the linear system $Ax = b$ of Example 7.13 using Gaussian elimination with partial pivoting.

SOLUTION: The first step would be to switch rows 1 and 3 (to make $|a_{11}|$ as large as possible):`Ab=rowswitch (Ab, 1,3).`

$$Ab \to P_1(Ab) \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \left[\begin{array}{ccc|c} 1 & 3 & -1 & 2 \\ 2 & 5 & -2 & 3 \\ 3 & 6 & 9 & 39 \end{array}\right] = \left[\begin{array}{ccc|c} 3 & 6 & 9 & 39 \\ 2 & 5 & -2 & 3 \\ 1 & 3 & -1 & 2 \end{array}\right].$$

(We will denote elementary matrices resulting from the `rowswitch` ERO by $P_i's$) Next we pivot on the $a_{11} = 3$ entry to clear out the entries below it. To clear out $a_{21} = 2$, we will do `Ab=rowcomb (Ab, 1,2,-2/3 )` (i.e., to clear out $a_{21} = 2$ we multiply row 1 by $-a_{21}/a_{11} = -2/\ 3$ and add this to row 2) Similarly,to clear out $a_{31} = 1$ we will do `»Ab=rowcomb (Ab, 1,3,-1/3)`.Combining both ofthese elementary matrices into a single matrix M_1, we may now write the result of these two EROs as follows:

$$Ab \to M_1(Ab) \begin{bmatrix} 1 & 0 & 0 \\ -2/3 & 1 & 0 \\ -1/3 & 0 & 1 \end{bmatrix} \cdot \left[\begin{array}{ccc|c} 3 & 6 & 9 & 39 \\ 2 & 5 & -2 & 3 \\ 1 & 3 & -1 & 2 \end{array}\right] = \left[\begin{array}{ccc|c} 3 & 6 & 9 & 39 \\ 0 & 1 & -8 & -23 \\ 1 & 1 & -4 & -11 \end{array}\right].$$

The pivot $a_{22} = 1$ is already as large as possible so we need not switch rows and can clear out the entry $a_{32} = 1$ by doing `Ab=rowcomb (Ab, 2,3,-1):`

$$Ab \to M_2(Ab) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \cdot \left[\begin{array}{ccc|c} 3 & 6 & 9 & 39 \\ 0 & 1 & -8 & -23 \\ 0 & 1 & -4 & -11 \end{array}\right] = \left[\begin{array}{ccc|c} 3 & 6 & 9 & 39 \\ 0 & 1 & -8 & -23 \\ 0 & 0 & 4 & 12 \end{array}\right],$$

and with this the elimination is complete.

Solving this (equivalent) upper triangular system will again yield the above solution. We note that this produces a slightly different factorization: From $M_2M_1PA = U$, where U is the left part of the final augmented matrix, we proceed as in the previous example to get $PA = (M_2^{-1}M_1^{-1})U \equiv LU$,i.e.,

$$PA = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 3 & -1 \\ 2 & 5 & -2 \\ 3 & 6 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ 1/3 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 6 & 9 \\ 0 & 1 & -8 \\ 0 & 0 & 4 \end{bmatrix} = LU.$$

This is the LU factorization of the matrix A. We now explain the general algorithm of **Gaussian elimination with partial pivoting** and the LU factorization. In terms of EROs and the back substitution, the resulting algorithm is quite compact.

Algorithm for Gaussian Elimination with Partial Pivoting: Given a linear system $Ax = b$ with A an $n \times n$ nonsinguiar matrix, this algorithm will solve for the solution vector x. The algorithm works on the $n \times (n+1)$ augmented matrix $\left[\begin{array}{c|c} A & B \end{array}\right]$, which we denote by Ab, but whose entries we still denote by a_{ij}.

```
if $|a_{kk}| = 0$, exit program with message "A is singular".
for $i = k + 1$ to $n$
$m_{ik} = a_{ik}/a_{kk}$
$A = rowcomb(A, k, i, -m_{ik})$
end i
end k
if $|a_{nn}| = 0$, exit program with message' Apply the back substitution algorithm triangular) to get solution to the system.
```

Without the interchanging rows step (unless to avoid a zero pivot), this is **Gaussian elimination** without partial pivoting. From now on, we follow the standard convention of referring to Gaussian elimination with partial pivoting simply as "Gaussian elimination," since it has become the standard algorithm for solving linear systems.

The algorithm can be recast into a matrix factorization algorithm for A. Indeed, at the kth iteration we will, in general, have an elementary matrix P_k corresponding to a row switch or permutation, followed by a matrix M_k that consists of the product of each of the elementary matrices corresponding to the "rowcomb" ERO used to clear out entries below a_{kk} . Letting U denote the upper triangular matrix left at the end of the algorithm, we thus have:

$$M_{n-1}P_{n-1}\ldots M_2P_2M_1P_1A = U.$$

The **LU factorization** (or the **LU decomposition**) of A, in general, has the form (see Section 4.4 of [GoVL-83]):

$$PA = LU, \tag{7.1}$$

Where

$$P = P_{n-1}P_{n-2}\ldots P_2P_1 \quad and \quad L = P(M_{n-1}P_{n-1}\ldots M_1P_1)^{-1}, \tag{7.2}$$

and L is lower triangular.[1] Also, by Theorem 7.4, the matrix PA corresponds to sequentially switching the rows of the matrix A, first corresponding to P_1, next by P_2, and so on. Thus the LU factorization A, once known, leads to a quick and practical way to solve any linear system $Ax = b$. First, permute the order of the equations as dictated by the permutation matrix P (do this on the augmented matrix so that $b's$ entries get permuted as well), relabel the system as $Ax = b$, and rewrite it as $LUx = b$. First solve $Ly = c$ by forward substitution (works since $L$is lower triangular), then solve $Ux = y$ by back substitution (works since U is upper triangular). Then x will be the desired solution (Proof:

$$PA = LU \Rightarrow Ax = P^{-1}LUx = P^{-1}L(Ux) = P^{-1}L(y) = p^{-1}Pb = b.) \tag{7.3}$$

This approach is useful if it is needed to solve a lot of linear systems with the same coefficient matrix A. For such situations, we mention that MATLAB has a built-in function `lu` to compute the LU factorization of a nonsingular matrix A. The syntax is as follows:

$[L,U,P] = lu(A)$ →	For a square singular matrix A, this command will output the lower triangular matrix L, the upper triangular matrix `U`, and the permutation matrix `P` of the *LU* factorization (29) and (30) of the matrix A

For example, applying this command to the matrix A of Example 7.14 gives:

```
>>A=[ 1 3 -1 ; 2 5 -2 ; 3 6 9] ; format rat , [L, U, P]=lu(A)
->L=       1 0 0
        2/3  1 0
        1/3  1 1

U=       3 6 9
       0 1  -8
       0 0 4

P=       0 0 1
       0 1 0
       1 0 0
```

We now wish to translate the above algorithm for Gaussian elimination into a MATLAB program. Before we do this, we make one remark about MATLAB's built-in function max, which we have encountered previously in its default format (the first syntax below):

max(v)→	For a vector v, this command will give the maximum of its components.
[max, index]=max(v) →	With an optional second output variable (that must be declared), max(v) will also give the first index at which this maximum value occurs.

[1] The permutation matrix P in (29) cannot, in general, be dispensed with; see Exercise 6.

Here, v can be either a row or a column vector. A simple example will illustrate this functionality.

```
>>v=[1 -3 5 -7 9 -11];
>>max(v)                   ->ans = 9
>>[max, index ] = max(v)       ->max = 9, index = 5
>>[max, index ] = max (abs (v))      ->max = 11, index = 6
```

Another useful tool for programming M-files is the `error` command:

error('message')→	If this command is encountered with an execution of any M-file, the M-file stops running immediately and displays the message .

PROGRAM 7.6: Function M-file for Gaussian elimination (with partial pivoting) to solve the linear system Ax = b> where A is a square nonsingular matrix. This program calls on the previous programs backsubst, rowswitch, and rowcomb

```
function x=gausselim(A,b)
%Inputs: Square matrix A, and column vector b of same dimension
%Output: Column vector solution x of linear system Ax - b obtained
%by Gaussian elimination with partial pivoting, provided coefficient.
%matrix A is nonsingular.
[n,n]=size(A);
Ab=[A';b']'; form augmented matrix for system

for k=1:n
  [biggest, occured] = max(abs(Ab(k:n,k)));
  if biggest == 0
  error('the coefficient mattix is numerically singular')
  end
  m=k+occured-1;
  Ab=rowswitch(Ab,k, m);
  for j=k+1:n
    Ab=rowcomb(Ab,k,j,-Ab(j,k)/Ab(k, k) ) ;
  end
end
% BACK SUBSTITUTION
x=backsubst (Ab(:,1:n ),Ab(:,n+1) );
```

EXERCISE FOR THE READER 7.20: Use the program gausseli m to resolve the Hubert systems of Example 7.9 and Exercise for the Reader 7.16, and compare to the results of the left divide method that proved most successful in those examples. Apart from additional error messages (regarding condition numbers), how do the results of the above algorithm compare with those of MATLAB's default system solver?

We next give a simple example that will demonstrate the advantages of partial pivoting

EXAMPLE 7.15:

Consider the following linear system: $\begin{bmatrix} 10^{-10} & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$, whose exact solution (starts) to look like $\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 1.0020... \\ .9989... \end{bmatrix}$.

(a) Using floating point arithmetic with three significant digits and chopped arithmetic, solve the system using Gaussian elimination with partial pivoting.
(b) Repeat part (a) in the same arithmetic, except without partial pivoting.

SOLUTION: For each part we show the chain of augmented matrices. Recall, after each individual computation, answers are chopped to three significant digits before any subsequent computations. Part (a): (with partial pivoting)

$$\left[\begin{array}{cc|c} .001 & 1 & 1 \\ 1 & 2 & 3 \end{array}\right] \underset{rowswitch(A,1,2)}{\longrightarrow} \left[\begin{array}{cc|c} 1 & 2 & 3 \\ .001 & 1 & 1 \end{array}\right] \underset{rowcomb(A,1,2,-.001)}{\longrightarrow} \left[\begin{array}{cc|c} 1 & 2 & 3 \\ 0 & .998 & .997 \end{array}\right].$$

Now we use back substitution: $x_2 = .997/.998 = .998, x_1 = (3-2x_2)/1 = (3-1.99)/1 = 1.01$. Our computed answer is correct to three decimals in x_2, but has a relative error of about 0.0798% in x_1.
Part (b): (without partial pivoting)

$$\left[\begin{array}{cc|c} .001 & 1 & 1 \\ 1 & 2 & 3 \end{array}\right] \underset{rowcomb(A,1,2,-1000)}{\longrightarrow} \left[\begin{array}{cc|c} 1 & 1 & 1 \\ 0 & -997 & -997 \end{array}\right]$$

Back substitution now gives $x_2 = -997/-998 = .998, x_1 = (1-1\cdot x_2)/1 = (1-.998)/1 = .002$. The relative error here is unacceptably large, exceeding 100% in the second component!

EXERCISE FOR THE READER 7.21: Rework the above example using rounded arithmetic rather than chopped, and keeping all else the same.

We close this section with a bit of information on flop counts for the Gaussian elimination algorithm. Note that the partial pivoting adds no flops since permuting rows involves no arithmetic. We will assume the worst-case scenario, in that none of the entries that come up are zeros. In counting flops, we refer to the algorithm above, rather than the MATLAB program. For $k = 1$, we will need to perform $n-1$ divisions to compute the multipliers m_{il} , and for each of these multipliers, we will need to do a `rowcomb`, which will involve n multiplications and n additions/subtractions. (Note: Since the first column entry will be zero and need not be computed, we are only counting columns 2 through $n+1$ of the augmented matrix.) Thus, associated with the pivot a_{11} , we will have to do $n-1$ divisions, $(n-1)n$ multiplications, and $(n-1)n$ additions/subtractions. Grouping the divisions and multiplications together, we see that at the $k = 1$ (first) iteration, we will need $(n-1)+(n-1)n = (n-1)(n+1)$ multiplications/divisions and $(n-1)n$ additions/subtractions. In the same fashion, the calculations associated with the pivot a_{22} will involve $n-2$ divisions plus $(n-2)(n-1)$ multiplications which is $(n-2)n$ multiplications/divisions and $(n-2)(n-1)$ additions/ subtractions. Continuing in this fashion, when we get to the pivot a_{kk}, we will need to do $(n-k)(n-k+2)$ multiplications/divisions and $(n-k)(n-k+1)$ additions/ subtractions. Summing from $k = 1$ to $n-1$ gives the following:

Total multiplications/divisions $\equiv M(n) = \sum_{k=1}^{n-1}(n-k)(n-k+2)$,

Total additions/subtractions $\equiv A(n) = \sum_{k=1}^{n-1}(n-k)(n-k+1)$.

Combining these two sums and regrouping gives:

Grand total flops

$$\equiv F(n) = \sum_{k=1}^{n-1}(n-k)(2(n-k)+3) = 2\sum_{k=1}^{n-1}(n-k)^2 + 3\sum_{k=1}^{n-1}.$$

If we reindex the last two sums, by substituting $j = n-k$, then as k runs from 1 through $n-1$, so will j (except in the reverse order), so that

$$F(n) = 2\sum_{j=1}^{n-1} j^2 + 3\sum_{j=1}^{n-1} j.$$

We now invoke the power sum identities (23) and (24) to evaluate the above two sums (replace n with $n-1$ in the identities) and thus rewrite the flop count $F(n)$ as:

$$F(n) = \frac{1}{3}(n-1)n(2n-1) + \frac{3}{2}(n-1)n = \frac{2}{3}n^3 + 3\text{+lower power terms.}$$

The "lower power terms" in the above flop counts can be explicitly computed (simply multiply out the polynomial on the left), but it is the highest order term that grows the fastest and thus is most important for roughly estimating flop counts. The flop count does not include the back substitution algorithm; but a similar analysis shows flop count for the back substitution to be just n_2 (see the Exercise for the Reader 7.22), and we thus can summarize with the following result.

PROPOSITION 7.5: *(Flop Counts for Gaussian Elimination)* In general, the number of flops needed to perform Gaussian elimination to solve a nonsingular system $Ax = b$ with an $n \times n$ coefficient matrix A is

$$\frac{2}{3}n^3\text{+lower power terms.}$$

EXERCISE FOR THE READER 7.22: Show that for the back substitution algorithm, the number of multiplications/divisions will be $(n^2+n)/2$, and the number of additions/subtractions will be $(n^2-n)/2$. Hence, the grand total flops required will be n^2.

By using the natural algorithm for computing the inverse of a matrix, a similar analysis can be used to show the flop count for finding the inverse of a matrix of a nonsingular $n/timesn$ matrix to be

$$(8/3)n^3\text{+lower power terms,}$$

or, in other words, essentially four times that for a single Gaussian elimination (see Exercise 16). Actually, it is possible to modify the algorithm (of Exercise 16) to a more complicated one that can bring this flop count down to $2n^3$+ lower power terms; but this is still going to be a more expensive and error prone method than Gaussian elimination, so we reiterate: For solving a single general linear system, Gaussian elimination is the best all-around method.

The next example will give some hard evidence of the rather surprising fact that the computer time required (on MATLAB) to perform an addition/subtraction is about the same as that required to perform a multiplication/division.

EXAMPLE 7.16:

In this example, we perform a short experiment to record the time and flops required to add 100 pairs of random floating point numbers. We then do the related experiment involving the same number of divisions.

```
>>A=rand(100); B=rand(100);,
>>tic, for i=1:100 , C=A+B; end, toc
```

→ Elapsed time is 5.778000 seconds.

```
>>tic, for i=1:100, C=A./B;, end, toc
```

→ Elapsed time is 5.778000 seconds.

The times are roughly of the same magnitude and, indeed, the flop counts are identical and close to the actual number of mathematical operations performed. The reason for the discrepancy in the latter is that, as previously mentioned, a flop is "approximately" equal to one arithmetic operation on the computer; and this is the most useful way to think about a flop.

EXERCISES 7.5:

NOTE: As mentioned in the text, we take "Gaussian elimination" to mean Gaussian elimination with partial pivoting.

1. Solve each of the following linear systems $Ax = b$ using three-digit chopped arithmetic and Gaussian elimination (i) without partial pivoting and then (ii) with partial pivoting. Finally redo the problems (iii) using MATLAB's left divide operator, and then (iv) using exact arithmetic (any method).

   a) $A = \begin{bmatrix} 2 & 9 \\ 1 & 7.5 \end{bmatrix}, b = \begin{bmatrix} 2 \\ -3 \end{bmatrix}$ b) $A = \begin{bmatrix} .99 & .98 \\ 101 & 100 \end{bmatrix}, b = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

   c) $A = \begin{bmatrix} 2 & 3 & -1 \\ 4 & 2 & -1 \\ -8 & 2 & 0 \end{bmatrix}, b = \begin{bmatrix} 0 \\ 21 \\ -4 \end{bmatrix}$

2. Parts (a) through (c): Repeat all parts of Exercise 1 using two-digit rounded arithmetic.

3. For each square matrix specified, find the LU factorization of the matrix (using Gaussian elimination). Do it first using (i) three-digit chopped arithmetic, then using (ii) exact arithmetic; and finally (iii) compare these with the results using MATLAB's built-in function `lu`.
   (a) The matrix A in Exercise 1, part (a).
   (b) The matrix A in Exercise 1, part (b).
   (c) The matrix A in Exercise 1, part (c).

4. Parts (a) through (c): Repeat all parts of Exercise 3 using two-digit rounded arithmetic in (i).

5. Consider the following linear system involving the 3×3 Hubert matrix H_3 as the coefficient matrix:

$$\begin{cases} x_1 & + & \frac{1}{2}x_2 & + & \frac{1}{3}x_3 & = & 2 \\ \frac{1}{2}x_1 & + & \frac{1}{3}x_2 & + & \frac{1}{4}x_3 & = & 0 \\ \frac{1}{2}x_1 & + & \frac{1}{4}x_2 & + & \frac{1}{5}x_3 & = & -1 \end{cases}$$

   (a) Solve the system using two-digit chopped arithmetic and Gaussian elimination without partial pivoting.
   (b) Solve the system using two-digit chopped arithmetic and Gaussian elimination.
   (c) Solve the system using exact arithmetic (any method).
   (d) Find the LU decomposition of the coefficient matrix H_3 by using 2-digit chopped arithmetic and Gaussian elimination.
   (e) Find the exact LU decomposition of H_3 .

6. (a) Find the LU factorization of the matrix $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.
   (b) Is it possible to find a lower triangular matrix L and an upper triangular matrix U (not necessarily those in part (a)) such that $A - LU$? Explain why or why not.

7. Suppose that $M_1, M_2, \ldots, M_k$ are invertible matrices of the same size. Prove that their product is invertible with $(M_1 \cdot M_2 \ldots M_k)^{-1} = M_k^{-1} \ldots M_2^{-1} \cdot M_1^{-1}$.In words, "The inverse of the product is the reverse-order product of the inverses."

8. (Storage and Computational Savings in Solving Tridiagonal Systems) Just as with any (nonsingular) matrix, we can apply Gaussian elimination to solve tridiagonal systems:

$$A = \begin{bmatrix} d_1 & a_1 & & & & \\ b_2 & d_2 & a_2 & & \mathbf{0} & \\ & b_3 & d_3 & a_3 & & \\ & & b_4 & d_4 & a_4 & \\ & & & \ddots & \ddots & \ddots \\ & \mathbf{0} & & b_{n-1} & d_{n-1} & a_{n-1} \\ & & & & b_n & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ \vdots \\ r_{n-1} \\ r_n \end{bmatrix}. \tag{7.4}$$

Here, d's stand for diagonal entries, b's for below-diagonal entries, a's for above-diagonal entries, and r's for right-side entries. We can greatly cut down on storage and unnecessary mathematical operations with zero by making use of the special sparse form of the tridiagonal matrix. The main observation is that at each step of the Gaussian elimination process, we will always be left with a banded matrix with perhaps one additional band above the a's diagonal. (Think about it, and convince yourself. The only way a switch can be done in selecting a pivot is with the row immediately below the diagonal pivot entry.) Thus, we may wish to organize a special algorithm that deals only with the tridiagonal entries of the coefficient matrix. (a) Show that the Gaussian elimination algorithm, with unnecessary operations involving zeros being omitted, will require no more than $8(n-1)$ flops (multiplications, divisions, additions, subtractions), and the corresponding back substitution will require no more than $5n$ flops. Thus the total number of flops for solving such a system can be reduced to less than $13n$. (b) Write a program, `x = tridiaggauss (d,b,a,r)` that inputs the diagonal vector d (of length n) and the above and below diagonal vectors a and b (of length $n-1$) of a nonsingular tridiagonal matrix, the column vector r and will solve the tridiagonal system (31) using the Gaussian elimination algorithm but which overwrites only the four relevant diagonal vectors (described above; you need to create an $n-2$ length vector for the extra diagonal) and the vector r rather than on the whole matrix. The output should be the solution column vector x. (c) Test out your algorithm on the system (31) with $n = 2, n = 100, n - 500, andn = 1000$ using the following data in the matrices $d_i = 4, a_i = 1, b_i = 1, r = [1 - 11 - 1 \ldots]$ and compare results and flop counts with MATLAB's left divide. You should see that your algorithm is much more efficient.

Note: The upper bound $13n$ on flops indicated in part (a) is somewhat liberal; a more careful analysis will show that the coefficient 13 can actually be made a bit smaller (How small can you make it?) But even so, the savings on flops (not to mention storage) are incredible. If we compare $13n$ with the bound $2n^3/3$, for large values of w, we will see that this modified method will allow us to solve extremely large tridiagonal systems that previously would have been out of the question. For example, when $n = 10,000$, this modified method would require storage of $2n + 2(n - l) = 39,998$ entries and less than $13n = 130,000$ flops (this would take a few seconds on MATLAB even on a weak computer); whereas the ordinary Gaussian elimination would require the storage of $n^2 + n = 100,010,000$ entries and approximately $2n^3/3 = 6.66\cdots \times 10^{11}$ flops, an unmanageable task!

9. *(The Thomas Method, an Even Faster Way to Solve Tridiagonal Systems)* By making a few extra assumptions that are usually satisfied in most tridiagonal systems that arise in applications, it is possible to slightly modify the usual Gaussian elimination algorithm to solve the triadiagonal system (31) in just $8n - 7$ flops (compared to the upper bound $13n$ of the last problem). The algorithm, known as the **Thomas method,**[2] differs from the usual Gaussian elimation by scaling the diagonal entry to equal I at each pivot, and by not doing any row changes (i.e., we forgo partial pivoting, or assume the matrix is of a form that makes it unnecessary; see Exercise 10). This will mean that we will have to keep track only of the above-diagonal entries (the a's vector) and the right-side vector r. The Thomas method algorithm thus proceeds as follows:

Step 1: (Results from $\texttt{rowmult}(A, 1, 1/d_1)) : a_1 = a_1/d_1, r_1 = r_1/d_1$.
(We could also add $d_1 - 1$, but since the diagonal entries will always be scaled to equal one, we do not need to explicitly record this change.)

Steps k=2 through n-1:(Results from rowcomb$(A, k-1, k, -b_k,)$ and then $\texttt{rowscale}(A, k, 1/(d_k - b_k a_{k-1})))$:

$$a_k = a_k/(d_k - b_k a_{k-1}), r_k = (r_k - b_k r_{k-1})/(d_k - b_k a_{n-1})$$

[2] The method is named after the renowned physicist Llewellyn H. Thomas; but it was actually discovered independently by several different individuals working in mathematics and related disciplines. W.F. Ames writes in his book [Ame-77] (p. 52): "The method we describe was discovered independently by many and has been called the Thomas algorithm. Its general description first appeared in widely distributed published form in an article by Bruce et al. [BPRR-53]."

Step n: (Results from same procedure as in steps 2 through $n-1$, but there is no a_n):

$$r_n = (r_n - b_n r_{n-1})/(d_n - b_n a_{n-1}).$$

This variation of Gaussian elimination has transformed the tridiagonal system into an upper triangular system with the following special form:

$$A = \left[\begin{array}{ccccc} 1 & a_1 & & & \\ & 1 & a_2 & \mathbf{0} & \\ & & \ddots & \ddots & \\ & \mathbf{0} & & 1 & a_{n-1} \\ & & & & 1 \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{array}\right] = \left[\begin{array}{c} r_1 \\ r_2 \\ \vdots \\ r_{n-1} \\ r_n \end{array}\right].$$

for which the back substitution algorithm takes on the particularly simple form: $x_n - r_n$; then for $k = n-1, n-2, \ldots, 2, 1 : x_k = r_k - a_k x_{k+1}$.

(a) Write a MATLAB M-file, `x=thomas (d,b,a,r)` that performs the Thomas method as described above to solve the tridiagonal system (31). The inputs should be the diagonal vector d (of length n) and the above and below diagonal vectors a and b (of length $n-1$) of a nonsingular tridiagonal matrix, and the column vector r. The output should be the computed solution, as a column vector x. Write your program so that it overwrites only the vectors a and r.

(b) Test out your program on the systems of part (c) of Exercise 8, and compare results and flop counts with those for MATLAB's left divide solver. If you have done part (c) of Exercise 8, compare also with the results from the program of the previous exercise.

(c) Do a flop count on the Thomas method to show that the total number of flops needed is $8n-7$.

NOTE: Looking over the Thomas method, we see that it assumes that $d_1 \neq 0$, and $d_k \neq b_k a_{k-1}$ (for k = 2 through n). One might think that to play it safe, it may be better to just use the slightly more expensive modification of Gaussian elimination described in the previous exercise, rather than risk running into problems with the Thomas method. For at most all applications, it turns out that the requirements for the Thomas method indeed are satisified. Such triadiagonal systems come up naturally in many applications, in particular in finite difference schemes for solving differential equations. One safe approach would be to simply build in a deferral to the previous algorithm in cases where the Thomas algorithm runs into a snag.

10. We say that a square matrix $A = [a_{ij}]$ **is strictly diagonally dominant (by columns)** if for each index $k, 1 \leq k \leq\neq n$, the following condition is met:

$$|a_{kk}| > \sum_{\substack{j=1 \\ j\neq k}}^{n} |a_{kj}|. \tag{7.5}$$

This condition merely states that each diagonal entry is larger, in absolute value, than the sum of the absolute values of all of the other entries in its column.

(a) Explain why when Gaussian elimination is applied to solve a linear system $Ax = b$ whose coefficient matrix is strictly diagonally dominant by columns, then no row changes will be required.

(b) Explain why the *LU* factorization of a diagonally dominant by columns matrix A will not have any permutation matrix.

(c) Explain why the requirements for the Thomas method (Exercise 9) will always be met if the coefficient matrix is strictly diagonally dominant by columns.

(d) Which, if any, of the above facts will continue to remain true if the strict diagonal dominance condition (32) is weakened to the following?

$$|a_{kk}| > \sum_{j=k+1}^{n} |a_{kj}|.$$

(That is, we are now only assuming that each diagonal entry is larger, in absolute value, than the sum of the absolute values of the entries that lie in the same column but below it.)

11. Discuss what conditions on the industries must hold in order for the technology matrix M of the Leontief input/output model of Exercise 10 from Section 7.4 to be diagonally dominant by columns (see the preceding exercise).

12. *(Determinants Revisited: Effects of Elementary Row/Column Operations on Determinants)* Prove the following facts about determinants, some of which were previewed in Exercise 10 of Section 7.1.
(a) If the matrix B is obtained from the square matrix A by multiplying one of the latter's rows by a number c (and leaving all other rows the same, i.e., `B=rowmult (A,i,c))` , then $det(B) = cdet(A)$.
(b) If the matrix B is obtained from the square matrix A by adding a multiple of the ith row of A to the jth row $(i \neg j)$ (i.e., `B=rowcomb (A,i,j,c))`, then det(B) = det(A).
(c) If the matrix B results from the matrix A by switching two rows of the latter (i.e., `B=rowswitch (A,i,j))`, then det(B) = -det(A).
(d) If two rows of a square matrix are the same, then det(A) = 0.
(e) If B is the transpose of A, then det(B) = det(A).
Note: In light of the result of part (e), each of the statements in the other parts regarding the effect of a row operation on a determinant has a valid counterpart for the effect of the corresponding column operation on the determinant.
Suggestions: You should make use of identity (20) $det(a) = det(,4)det(5)$, as well as Proposition 7.3 and Theorem 7.4. The results of (a), (b), and (c) can then be proved by calculating determinants of certain elementary matrices. The only difficult thing is for part (c) to show that the determinant of a permutation matrix gotten from the identity matrix by switching two rows equals -1 . One way this can be done is by an appropriate (but not at all obvious) matrix factorization. Here is one way to do it for the (only) 2×2 permutation matrix:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

(Check this!) All of the matrix factors on the right are triangular so the determinants of each are easily computed by multiplying diagonal entries (Proposition 7.3), so using (20), we get

$$\det\left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\right) = 1 \cdot 1 \cdot (-1) \cdot 1 = -1.$$

In general, this argument can be made to work for any permutation matrix (obtained by switching two rows of the identity matrix), by carefully generalizing the factorization. For example, here is how the factorization would generalize for a certain 3×3 permutation matrix:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Part (d) can be proved easily from part (c); for part (e) use mathematical induction and cofactor expansion.

13. (Determinants Revisited: A Better Way to Compute Them) The Gaussian elimination algorithm provides us with an efficient way to compute determinants. Previously, the only method we gave to compute them was by cofactor expansion, which we introduced in Chapter 4 . But we saw that this was an extremely expensive way to compute determinants. A new idea is to use the Gaussian climination algorithm to transform a square matrix A into an upper triangular matrix. From the previous exercise, each time a `rowcomb` is done, there will be no effect on the determinant, but each time a `rowswitch` is done, the determinant is negated. By Proposition 7.3, the determinant of the diagonal matrix is just the product of the diagonal entries. Of course, in the Gaussian elimination algorithm, the column vector b can be removed (if all we are interested in is the determinant). Also, if a singularity is detected, the algorithm should exit and assign $\det(A) = 0$.
(a) Create a function M-file, called `y= gaussdet` (A), that inputs a square matrix M and outputs the determinant using this algorithm.
(b) Test your program out by computing the determinants of matrices with random integer entries from -9 to 9 of sizes $3 \times 3, 8 \times 8, 20 \times 20$, and 80×80 (you need not print the last two matrices) that you can construct using the M-file randint of Exercise for the Reader 7.2. Compare the results, computing times and (if you have Version 5) flop counts with those for MATLAB's builtin det function applied to the same matrices.
(c) Go through an analysis similar to that done at the end of the section to prove a result similar to that of Proposition 7.5 that will give an estimate of the total flop counts for this algorithm, with the highest-order term being accurate.
(d) Obtain a similar flop count for the cofactor expansion method and compare with the answer you got in (c). (The highest-order term will involve factorials rather than powers.)
(e) Use your answer in (c) to obtain a flop count for the amount of flops needed to apply Cramer's rule to solve a nonsingular linear system Ax = b with A being an n x n nonsingular matrix.

14. (a) Prove Proposition 7.3.
(b) Prove an analogous formula for the determinant of a square matrix that is upper-left triangular in the sense that

all entries above the off-main diagonal are zeros. More precisely, prove that any matrix of the following form,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & & \\ \vdots & & / & & \\ a_{n-1,1} & a_{n-1,2} & & \mathbf{0} & \\ a_{n1} & & & & \end{bmatrix},$$

has determinant given by $\det(A) = (-1)^k a_{1n} \cdot a_{2,n-1} \cdot a_{3,n-2} \cdots a_{n-1,2} \cdot a_{n1}$ where $n = 2k + i (i = 0, 1)$.
Suggestion: Proceed by induction on n, where A is an $n \times n$ matrix. Use cofactor expansion along an appropriate row (or column).

15. (a) Write a function M-file, call it `[L, U, P]= mylu (A)`, that will compute the LU factorization of an inputted nonsingular matrix A.
(b) Apply this function to each of the three coefficient matrices in Exercise 1 as well as the Hilbert matrix H_3, and compare the results (and flop counts) to those with MATLAB's built-in function `lu`Ḟrom these comparisons, does your program seem to be as efficient as MATLAB's?

16. (a) Write a function M-file, call it `B=myinv(A)`, that will compute the inverse of an inputted nonsingular matrix A, and otherwise will output the error message: "Matrix detected as numerically singular." Your algorithm should be based on the following fact (which follows from the way that matrix multiplication works). To find an inverse of an $n \times n$ nonsingular matrix A, it is sufficient to solve the following n linear equations:

$$Ax^1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, Ax^2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, Ax^3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \cdots Ax^n = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix},$$

where the column vectors on the right sides of these equations are precisely the columns of the $n \times n$ identity matrix. It then would follow that $A = \left[\, x^1 \;\vdots\; x^2 \;\vdots\; x^3 \;\vdots\; \cdots \;\vdots\; x^n \,\right] = I$, so that the desired inverse of A is the matrix $A_{-1} = \left[\, x^1 \;\vdots\; x^2 \;\vdots\; x^3 \;\vdots\; \cdots \;\vdots\; x^n \,\right]$. Your algorithm should be based on the LU decomposition, so it gets computed once, rather than doing a complete Gaussian elimination for each of the n equations.
(b) Apply this function to each of the three coefficient matrices in Exercise 1 as well as the Hilbert matrix H_4, and compare the results `inv`. From these comparisons, does your program seem to be as efficient as MATLAB's?
(c) Do a flop count similar to the one done for Proposition $(andflopcounts)tothosewithMATLAB'sbuilt-infunction7.5$ for this algorithm. Note: For part (a), feel free to use MATLAB's built-in function 1u; see the comments in the text about how to use the LU factorization to solve linear systems.

7.6. VECTOR AND MATRIX NORMS, ERROR ANALYSIS, AND EIGENDATA

In the last section we introduced the Gaussian elimination (with partial pivoting) algorithm for solving a nonsingular linear system

$$Ax = b \tag{7.6}$$

where $A = [a_{ij}]$ is an $n \times n$ coefficient matrix, b is an $n \times 1$ column vector, and x is the $n \times 1$ column vector of variables whose solution is sought. This algorithm is the best all-around general numerical method for solving the linear system (33), but its performance can vary depending on the coefficient matrix A. In this section we will present some practical estimates for the error of the computed solution that will allow us to put some quality control guarantee on the answers that we obtain from (numerical) Gaussian elimination. We need to begin with a practical way to measure the "sizes" of vectors and matrices. We have already used the Euclidean length of a vector v to measure its size, and norms will be a generalization of this concept. We will introduce norms for vectors and matrices in this section, as well as the so-called condition numbers for square matrices. Shortly we will use norms and condition numbers to give precise estimates for the error of the computed solution of (33) (using Gaussian elimination). We will also explain some ideas to try when a system to be solved is poorly conditioned. The theory on modified algorithms that can deal with poorly conditioned systems contains an assortment of algorithms that can perform well if the (poorly conditioned) matrix takes on a special form. If one has the Student Version of MATLAB (or has the Symbolic Toolbox) there is always the option of working in exact arithmetic or with a fixed but greater number of significant digits. The main (and only) disadvantage of working in such arithmetic is that computations move a lot slower, so we will present some concrete criteria that will help us to decide when such a route might be needed. The whole subject of error analysis and refinements for numerically solving linear systems is quite vast and we will not be delving too deeply into it. For more details and additional results, the interested reader is advised to consult one of the following references (listed in order of increasing mathematical sophistication):

[Atk-89], [Ort-90], [GoVL-83].

The Euclidean "length" of an n-dimensional vector $X = [x_1\ X_2\ \ldots\ x_n]$ is defined by:

$$\mathrm{len}(x) = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}. \tag{7.7}$$

For this definition it is immaterial whether x is a row or column vector. For example, if we are working in two dimensions and if the vector is drawn in the $xy-$ plane from its tail at $(x,y) = (0,0)$ to its tip $(x,y) = (x_1,x_2)$, then len (x) is (in most cases) the hypotenuse of a right triangle with legs having length $|x_1|$ and $|x_2|$, and so the formula (34) becomes the Pythagorean theorem. In the remaining cases where one of x_1 or x_2 is zero, then len (x) is simply the absolute value of the other coordinate (in this case also the length of the vector x that will lie on either the x-or y-axis.) From what we know about plane geometry, we can deduce that len (x) has the following properties:

$$\text{len } (x) \geq 0, \text{ and len } (x) = 0 \text{ if and only if } x = 0 \text{ (vector)}, \tag{7.8a}$$

$$\text{len } (cx) = |c| \text{ len } (x) \text{ for any scalar } c, \tag{7.8b}$$

$$\text{len } (x+y) \leq \text{len}(x) + \text{len}(y) \text{ (Triangle Inequality)}. \tag{7.8c}$$

Property (35A) is clear (even in n dimensions). Property (35B) corresponds to the geometric fact that when a vector is multiplied by a scalar, the length gets multiplied by the absolute value of the scalar (we learned this early in the chapter). The triangle inequality (35C) corresponds to the geometric fact (in two dimensions) that the length of any side of any triangle can never exceed the sum of the lengths of the other two sides. These properties remain true for general n-dimensional vectors (see Exercise 11 for a more general result).

A vector norm for n-dimensional (row or column) vectors $x = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$ is a way to associate a nonnegative number (default notation: $\|x\|$) with the vector x such that the following three properties hold:

$$\|x\| \geq 0, \|x\| = 0 \text{ if and only if } x = 0 \text{ (vector)}, \tag{7.9a}$$

$$\|cx\| = |c|\|x\| \text{ for any scalar } c \tag{7.9b}$$

$$\|x+y\| \leq \|x\| + \|y\| \text{ (Triangle Inequality)} \tag{7.9c}$$

We have merely transcribed the properties (35) to obtain these three axioms (36) for a norm. It turns out that there is an assortment of useful norms, the aforementioned Euclidean norm being one of them. The one we will use most in this section is the so-called **max norm** (also known as the **infinity norm**) and this is defined as follows:

$$\|x\| = \|x\|_\infty = \max\{|x_i|, 1 \leq i \leq n\} = \max\{|x_1|, |x_2|, \cdots, |x_n|\}. \tag{7.10}$$

The proper mathematical notation for this vector norm is $\|x\|_\infty$, but since it will be our default vector norm we will often denote it by $\|x\|$ for convenience. The max norm is the simplest of all vector norms, so working with it will allow the complicated general concepts from error analysis to be understood in the simplest possible setting. The price paid for this simplicity will be that some of the resulting error estimates that we obtain using the max norm may be somewhat more liberal than those obtained with other, more complicated norms. Both the max and Euclidean norms are easy to compute on MATLAB (e.g., for the max norm of x we could simply type max `(abs(x))`, but (of course) MATLAB has built-in functions for both of these vector norms and many others.

norm $(x) \rightarrow$	Computes the length norm len (x) of a (row or column) vector x.
norm $(x,\ \text{inf}\ ) \rightarrow$	Computes the max norm $\mid x \mid$ of a (row or column) vector x.

EXAMPLE 7.17: For the two four-dimensional vectors $x = [1,0,-4,6]$ and $y = [3,-4,1,-3]$ find the following:
(a) $\mathrm{len}(x), \mathrm{len}(y), \mathrm{len}(x+y)$
(b) $\|x\|, \|y\|, \|x+y\|$

SOLUTION: First we do these computations by hand, and then redo them using MATLAB.
Part (a): Using (34) and since $x = [4,-4,-3,3]$ we get that $\mathrm{len}(x) = \sqrt{1^2+0^2+(-4)^2+6^2} = \sqrt{53} = 7.2801\ldots, \mathrm{len}(y) = \sqrt{3^2+(-4)^2+1^2+(-3)^2} = \sqrt{35} = 5.9160\ldots$, and len $(x+y) = \sqrt{4^2+(-4)^2+(-3)^2+3^2} = \sqrt{50} = 7.0710\ldots$
Part (b): Using (37), we compute: $\|x\| = \max\{|1|,|0|,|-4|,|6|\} = 6, \|y\| = \max\{|3|,|-4|,|1|,|-3|\} = 4$, and $\|x+y\| = \max\{|4|,|-4|,|-3|,|3|\} = 4$.

These computations give experimental evidence of the validity of the triangle inequality in this special case. We now repeat these same computations using MATLAB:

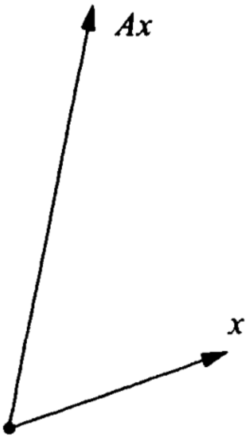
```
>>x=[1 0 -4 6] ; y=[3 -4 1 -3] ;
>>norm(x), norm(y), norm(x+y)    ->ans = 7.2801 5.9161 7.0711
>>norm(x, inf) , norm(y,inf) , norm (x+y, inf)     ->ans = 6 4 4
```

EXERCISE FOR THE READER 7.23: Show that the max norm as defined by (37) is indeed a vector norm by verifying the three vector norm axioms of (36).

Given any vector norm, we define an **associated matrix norm** by the following:

$$\|A\| \equiv \max\left\{\frac{\|Ax\|}{\|x\|}, x \neq 0 \text{ (vector)}\right\}. \tag{7.11}$$

For any nonzero vector x, its norm $\|x\|$ will be a positive number (by (36 A)); the transformed vector Ax will be another vector and so will have a norm $\|Ax\|$. The norm of the matrix A can be thought of as the maximum magnification factor by which the transformed vector Ax' s norm will have changed from the original vector x 's norm; see Figure 7.35.



Rysunek 7.1: Graphic for the matrix norm definition (38). The matrix A will transform x into another vector Ax (of same dimension if A is square). The norm of A is the maximum magnification that the transformed vector Ax norm will have in terms of the norm of x.

It is interesting that matrix norms, despite the daunting definition (38), are often easily computed from other formulas. For the max vector norm, it can be shown that the corresponding matrix norm (38), often called the **infinity matrix norm**, is given by the following formula:

$$\|A\| = \max_{1\le i\le n}\left\{\sum_{j=1}^{n}|a_{ij}|\right\} = \max_{1\le i\le n}\left\{|a_{i1}|+|a_{i2}|+\cdots+|a_{in}|\right\}. \tag{7.12}$$

This more practical definition is simple to compute: We take the sum of each of the absolute values of the entries in each row of the matrix A, and $\|A\|$ will equal the maximum of these "row sums." MATLAB has a command that will do this computation for us:

`norm(A,inf) ->`	Computes the infinity norm $\|A\|$ of a matrix A.

One simple but very important consequence of the definition (38) is the following inequality:

$$\|Ax\| \le \|A\|\|x\| \text{ (for any matrix } A \text{ and vector } x \text{ of compatible size).} \tag{7.13}$$

To see why (40) is true is easy: First if x is the zero vector, then so is Ax and so both sides of (40) are equal to zero. If x is not the zero vector then by (38) we have $\|Ax\|/\|x\| \le \|A\|$, so we can multiply both sides of this inequality by the positive number $\|x\|$ to produce (40).

EXAMPLE 7.18: Let $A = \begin{bmatrix} 1 & 2 & -1 \\ 0 & 3 & -1 \\ 5 & -1 & 1 \end{bmatrix}$ and $x = \begin{bmatrix} 1 \\ 0 \\ -2 \end{bmatrix}$. Compute $\|x\|$, $\|Ax\|$, and $\|A\|$ and check the validity of (40).

SOLUTION: Since $Ax = \begin{bmatrix} 3 \\ 2 \\ 3 \end{bmatrix}$, we obtain: $\|x\| = 2, \|Ax\| = 3$, and using (39), $\|A\| = \max\{1+2+|-1|, 0+3+|-1|, 5+|-1|+1\} = \max\{4,4,7\} = 7$. Certainly $\|Ax\| \le \|A\|\|x\|$ holds here $(3 \le 7\cdot 2)$.

EXERCISE FOR THE READER 7.24: Prove the following two facts about matrix norms: For two $n\times n$ matrices A and B :

(a) $\|AB\| \le \|A\|\|B\|$.

(b) If A is nonsingular, then $\left\|A^{-1}\right\| = \left(\min_{x\neq 0}\frac{\|Ax\|^{-1}}{\|x\|}\right)^{-1}$.

With matrix norms introduced, we are now in a position to define the condition number of a nonsingular (square) matrix. For such a matrix A, the condition number of A, denoted by $\kappa(A)$, is the product of the norm of A and the norm of A^{-1}, i.e.,

$$\kappa(A) = \text{ condition number of } A \equiv \|A\|\|A^{-1}\|. \tag{7.14}$$

By convention, for a singular matrix A, we define $\kappa(A) = \infty$,[3] Unlike the determinant, a large condition number is a reliable indicator that a square matrix is **nearly singular** (or **poorly conditioned**); and condition numbers will be a cornerstone in many of the error estimates for linear systems that we give later in this section. Of course, the condition number depends on the vector norm that is being used (which determines the matrix norm), but unless explicitly stated otherwise, we will always use the infinity vector norm (and the associated matrix norm and condition numbers). To compute the condition number directly is an expensive computation in general, since it involves computing the inverse A^{-1}. There are good algorithms to estimate condition numbers relatively quickly to any degree of accuracy. We will forgo presenting such algorithms, but will take the liberty of using the following MATLAB built-in function for computing condition numbers:

`cond (A, inf) ->`	Computes and outputs the condition number (with respect to the infinity vector norm) of the square matrix A.

The condition number has the following general properties (actually valid for condition numbers arising from any vector norm):

$$\kappa(A) \geq 1, \text{ for any square matrix } A. \tag{7.15}$$

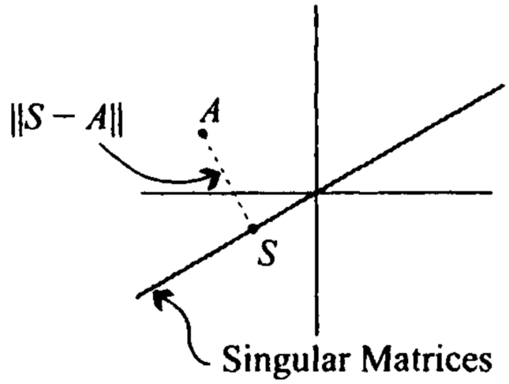
If D is a diagonal matrix with nonzero diagonal entries: $d_1, d_2, \cdots, d_n$, then

$$\kappa(D) = \frac{\max\{|d_i|\}}{\min\{|d_i|\}}. \tag{7.16}$$

If A is a square matrix and c a nonzero scalar, then

$$\text{then } \kappa(cA) = \kappa(A). \tag{7.17}$$

In particular, from (43) it follows that $\kappa(I) = 1$. The proofs of these identities will be left to the exercises. Before giving our error analysis results (for linear systems), we state here a theorem that shows, quite quantitatively, that nonsingular matrices with large condition numbers are truly very close to being singular. Recall that the singular square matrices are precisely those whose determinant is zero. For a given $n \times n$ nonsingular matrix A, we think of the distance from A to the set of all $n \times n$ singular matrices to be $\min\{\|S - A\| : \det(S) = 0\}$. (Just as with absolute values, the norm of a difference of matrices is taken to be the distance between the matrices.) We point out that $\min_{\det(S)=0} \|S - A\|$ can be thought of as the distance from A to the set of singular matrices. (See Figure 7.36.)



Rysunek 7.2: Heuristic diagram showing the distance from a nonsingular matrix A to the set of all singular matrices (line).

THEOREM 7.6: (*Geometric Characterization of Condition Numbers*) If A is any $n \times n$ nonsingular matrix, then we have:

$$\frac{1}{\kappa(A)} = \frac{1}{\|A\|} \cdot \min_{\mathrm{de}(S)=0} \|S - A\| = \frac{1}{\|A\|}. \tag{7.18}$$

Like all of the results we state involving matrix norms and condition numbers, this one is true, in general, for whichever matrix norm (and resulting condition number) we would like to use. A proof can be found in the paper [Kah-66]. This theorem suggests some of the difficulties in trying to numerically solve systems having large condition numbers. Gaussian elimination involves many computations and each time we modify our matrix, because of roundoff errors, we are actually dealing with matrices that are close to but not the same as the actual (mathematically exact) matrices. The theorem shows that for poorly conditioned matrices (i.e., ones with large condition numbers), this process is extremely sensitive since even a small change in a poorly conditioned matrix could result in one that is singular!

We close with an example that will review some of the concepts about norms and condition numbers that have been introduced.

[3] Sometimes this condition number is denoted $\kappa_\infty(A)$ to emphasize that it derives from the infinity vector and matrix norm. Since this will be the only condition number that we use, no ambiguity should arise by our adopting this abbreviated notation.

EXAMPLE 7.19: Consider the matrix: $A = \begin{bmatrix} 7 & -4 \\ -5 & 3 \end{bmatrix}$.
(a) Is there a (2×1) vector x such that: $\|Ax\| > 8\|x\|$? If yes, find one; otherwise explain why one does not exist.
(b) Is there a nonzero vector x such that $\|Ax\| \geq 12\|x\|$? If so, find one; otherwise explain why one does not exist.
(c) Is there a singular matrix $S = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ (i.e., $ad - bc = 0$) such that $\|S - A\| \leq 0.2$? If so, find one; otherwise explain why one does not exist.
(d) Is there a singular matrix $S = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ (i.e., $ad - bc = 0$) such that $\|S - A\| \leq 0.05$? If so, find one; otherwise explain why one does not exist.

SOLUTION: Parts (a) and (b): Since $\|A\| = 7 + 4 = 11$, it follows from (38) that there exist (nonzero) vectors x with $\|Ax\|/\|x\| = 11$ or, put differently (multiply by $\|x\|$)$\|Ax\| = 11\|x\|$, but there will not be any nonzero vectors x that will make this equation work if 11 gets replaced by any larger number. (The maximum amount that matrix multiplication by A can magnify the norm of any nonzero vector x is 11 times.) Thus part (a) will have a vector solution but part (b) will not. To find an explicit vector x that will solve part (a), we will actually do more and find one that undergoes the maximum possible magnification $\|Ax\| = 11\|x\|$. The procedure is quite simple (and general). The vector x will have entries being either 1 or -1. To find such an appropriate vector x, we simply identify the row of A that gives rise to its norm being 11; this would be the first row (in general if more than one row gives the norm, we can choose either one). We simply choose the signs of the x-entries so that when they are multiplied in order by the corresponding entries in the just-identified row of A, all products are positive. In other words, if an entry in the special row of A is positive, take the corresponding component of x to be 1 ; if the special row entry of A is negative, take the corresponding component of x to be -1. In our case the special row of A is (the first row) $[7 - 4]$, and so in accordance we take $x = [1 - 1]$ '. The first entry of the vector Ax is $\begin{bmatrix} 7 & -4 \end{bmatrix} \cdot [1 - 1]' = 7(1) - 4(-1) = 7 + 4 = 11$, so $\|x\| = 1$ and $\|Ax\| = 11$ (actually, this shows only $\|Ax\| \geq 11$ since we have not yet computed the other component of Ax, but from what was already said, we know $\|Ax\| \leq 11$, so that indeed $\|Ax\| = 11$). This procedure easily extends to any matrices of any size.

Parts (c) and (d): We rewrite equation (45) to isolate the distance from A to the singular matrices (simply multiply both sides by $\|A\|$):

$$\min_{\det(S)=0} \|S - A\| = \frac{\|A\|}{\kappa(A)}$$

Appealing to MATLAB to compute the right side (and hence the distance from A to singulars):

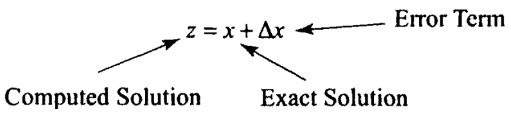
```
>>A=[7  -4 ;-5  3]
>>norm(A,inf)cond(A,inf)
->ans =         0.0833
```

Since this distance is less than 0.2, the theorem tells us that there is a singular matrix satisfying the requirement of part (c) (the theorem unfortunately does not help us to find one), but there is no singular matrix satisfying the more stringent requirements of part (d). We use *ad hoc* methods to find a specific matrix S that satisfies the requirements of part (c). Note that $\det A = 7 \cdot 3 - (-5) \cdot (-4) = 1$. We will try to tweak the entries of A into those of a singular matrix $S = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ with determinant $ad - bc = 0$. The requirement of the distance being less than 0.2 means that our perturbations in each row must add up (in absolute value) to at most 0.2. Let's try tweaking 7 to $a = 6.9$ and 3 to $d = 2.9$ (motive for this move: right now A has $ad = 21$, which is one more than $bc = 20$; we need to tweak things so that ad is brought down a bit and bc is brought up to meet it). Now we have $ad = 20.01$, and we still have a perturabtion allowance of 0.1 for both entries b and c and we need only bring bc up from its current value of 20 to 20.01. This is easythere are many ways to do it. For example, keep $c = -5$ and solve $bc = 20.01$, which gives $c = 20.01/-5 = -4.002$ (well within the remaining perturbation allowance). In summary, the matrix $S = \begin{bmatrix} 6.9 & -4.002 \\ -5 & 2.9 \end{bmatrix}$ meets the requirements that were asked for in part (c). Indeed S is singular (its determinant was arranged to be zero), and the distance from this matrix to A is

$$\|S - A\| = \left\| \begin{bmatrix} 6.9 & -4.002 \\ -5 & 2.9 \end{bmatrix} - \begin{bmatrix} 7 & -4 \\ -5 & 3 \end{bmatrix} \right\| = \left\| \begin{bmatrix} -.1 & -.002 \\ 0 & -.1 \end{bmatrix} \right\| = .102 < 0.2$$

NOTE: The matrix S that we found was actually quite a bit closer to A than what was asked for. Of course, the closer that we wish to find a singular matrix to the ultimate distance, the harder we will have to work with such *ad hoc* methods. Also, the idea used to construct the "extremal" vector x can be modified to give a proof of identity (39); this task will be left to the interested reader as Exercise 10.

When we use Gaussian elimination to solve a nonsingular linear system (33): $Ax = b$, we will get a **computed solution** vector z that will, in general, differ from the **exact (mathematical) solution** x by the **error term** Δx :



The main goal for the error analysis is to derive estimates for the size of the error (vector) term: $||\Delta x||$. Such estimates will give us quality control on the computed solution z to the linear system.

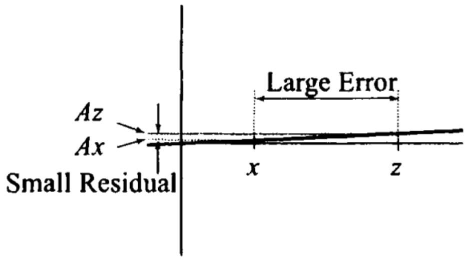
Caution: It may seem that a good way to measure the quality of the computed solution is to look at the size (norm) of the so-called residual vector:

$$r = \text{ residual vector } \equiv b - Az. \tag{7.19}$$

Indeed, if z were the exact solution x then the residual would equal the zero vector. We note the following different ways to write the residual vector:

$$r \equiv b - Az = Ax - Az = A(x-z) = A(x-(x+\Delta x)) = A(-\Delta x) = -A(\Delta x) \tag{7.20}$$

in particular, the residual is simply the (negative of) the matrix A multiplied by the error term vector. The matrix A may distort a large error term into a much smaller vector thus making the residual much smaller than the actual error term. The following example illustrates this phenomenon (see Figure 7.37).



Rysunek 7.3: Heuristic illustration showing the unreliability of the residual as a gauge to measure the error. This phenomenon is a special case of the general principle that a function having a very small derivative can have very close outputs resulting from different inputs spread far apart.

EXAMPLE 7.20: Consider the following linear system $Ax = b$:

$$\begin{bmatrix} 1 & 2 \\ 1.0001 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 3.0001 \end{bmatrix}.$$

This system has (unique) exact solution $x = [1,1]'$. Let's consider the (poor) approximation $z = [3,0]'$. The (norm of the) error of this approximation is $||x-z|| = ||[-2,1]'|| = 2$, but the residual vector,

$$r = b - Az = \begin{bmatrix} 3 \\ 3.0001 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 1.0001 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 3.0001 \end{bmatrix} - \begin{bmatrix} 3 \\ 3.0003 \end{bmatrix} = \begin{bmatrix} 0 \\ -.0002 \end{bmatrix}$$

has a much smaller norm of only 0.0002. This phenomenon is also depicted in Figure 7.37.

Despite this drawback about the residual itself, it can be manipulated to indeed give us a useful error estimate. Indeed, from (47), we may multiply left sides by A^{-1} to obtain:

$$r = A(-\Delta x) \Rightarrow -\Delta x = A^{-1}r.$$

If we now take norms of both sides and apply (40) , we can conclude that:

$$||\Delta x|| = ||-\Delta x|| = ||A^{-1}r|| \le ||A^{-1}||\,||r||.$$

(We have used the fact that $||-v|| = ||v||$ for any vector v; this follows from norm axiom (36B) using $c = -1$.) We now summarize this simple yet important result in the following theorem.

THEOREM 7.7: (Error Bound via Residual) If z is an approximate solution to the exact solution x of the linear system (33) $Ax = b$, with A nonsingular, and $r = b - Az$ is the residual vector, then

$$\text{error } \equiv ||x-z|| \le ||A^{-1}||\,||r||.. = \tag{7.21}$$

REMARK: Using the condition number $\kappa(A) = ||A||A^{-1}||$, Theorem 7.7 can be reformulated as

$$||x-z|| \le \kappa(A)\frac{||r||}{||A||} \tag{7.22}$$

EXAMPLE 7.21: Consider once again the linear system $Ax = b$ of the preceding example:

$$\begin{bmatrix} 1 & 2 \\ 1.0001 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 3.0001 \end{bmatrix}.$$

If we again use the vector $z = [3,0]'$ as the approximate solution, then, as we saw in the last example, the error $= 2$ and the residual is $r = [0, -0.0002]'$. The estimate for the error provided in Theorem 9.2, is (with MATLAB's help) found to be 4 :

```
>>A=[ 1 2; 1.0001 2] ; r=[0 -.0002];
>>norm(inv(A), inf) *norm(r,inf)      ->ans = 4.0000
```

Although this estimate for the error is about as far off from the actual error as the approximation z is from the actual solution, as far as an estimate for the error is concerned, it is considered a decent estimate. An estimate for the error is considered good if it has approximately the same order of magnitude (power of 10 in scientific notation) as the actual error.

Using the previous theorem on the error, we obtain the following analogous result for the relative error.

THEOREM 7.8: (Relative Error Bound via Residual) If z is an approximate solution to the exact solution x of the linear system (33) $Ax = b$, with A nonsingular, $b \neq 0$ (vector), and $r = b - Az$ is the residual vector, then

$$\text{relative error} \equiv \frac{\|x - z\|}{\|x\|} \leq \frac{\|A\| \, \|A^{-1}\|}{\|b\|} \|r\|. \tag{7.23}$$

REMARK: In terms of condition numbers, Theorem 7.8 takes on the more appealing form:

$$\frac{\|x - z\|}{\|x\|} \leq \kappa(A) \frac{\|r\|}{\|b\|} \tag{7.24}$$

Proof of Theorem 7.8: We first point out that $x \neq 0$ since $b \neq 0$ (and $Ax = b$ with A nonsingular). Using identity (40), we deduce that:

$$\|b\| = \|Ax\| \leq \|A\| \|x\| \Rightarrow \frac{1}{\|x\|} \leq \frac{\|A\|}{\|b\|}.$$

We need only multiply both sides of this latter inequality by $\|x - z\|$ and then apply (48) to arrive at the desired inequality:

$$\frac{\|x - z\|}{\|x\|} \leq \frac{\|A\|}{\|b\|} \cdot \|x - z\| \leq \frac{\|A\|}{\|b\|} \cdot \|A^{-1}\| \, \|r\|$$

EXAMPLE 7.21: (cont.) Using MATLAB to compute the right side of (50),

```
>>cond(A,inf) *norm(r, inf)/norm ([3 3.0001]',inf )
->ans = 4.0000
```

Once again, this compares favorably with the true value of the relative error whose explicit value is $\|x - z\| / \|x\| = 2/1 = 2$ (see Example 7.20).

EXAMPLE 7.22: Consider the following (large) linear system Ax=b, with

$$A = \begin{bmatrix} 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & \cdots & & 0 \\ 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & \ddots & & 0 \\ -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & \ddots & & 0 \\ 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & \ddots & . & 0 \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & & \\ 0 & \vdots & & & & & & & & & \\ 0 & & & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & & \ddots & -1 & 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & \cdots & & 0 & -1 & 0 & 0 & -1 & 4 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ \vdots \\ 798 \\ 799 \\ 800 \end{bmatrix}.$$

The 800×800 coefficient matrix A is diagonally banded with a string of 4 's down the main diagonal, a string of -1 's down each of the diagonals 4 below and 4 above the main diagonal, and each of the diagonals directly above and below the main diagonal consist of the vector that starts off with $[-1 - 1 - 1]$, and repeatedly tacks the sequence $[0 - 1 - 1 - 1]$ onto this until the diagonal fills. Such banded coefficient matrices are very common in finite difference methods for solving

(ordinary and partial) differential equations. Despite the intimidating size of this system, MATLAB's "left divide" can take advantage of its special structure and will produce solutions with very decent accuracy as we will see below:
(a) Compute the condition number of the matrix A.
(b) Use the left divide (Gaussian elimination) to solve this system, and call the computed solution z. Use Theorem 7.7 to estimate its error and Theorem 7.8 to estimate the relative error. Do not print out the vector z !
(c) Obtain a second numerical solution $z2$, this time by left multiplying the equation by the inverse A^{-1}. Use Theorem 7.7 to estimate its error and Theorem 7.8 to estimate the relative error. Do not print out the vector $z2$!

SOLUTION: We first enter the matrix A, making use of MATLAB's useful diag function:

```
>>A=diag(4*ones(1,800));
>>al=[- 1 - 1 -1] ; vrep=[0 - 1 - 1 -1] ;
>>for i=1:199, al=[al, vrep] ; end %this is level +1/-1 diagonal
>>v4 = -1*ones(1,796); %this is level +4/-4 diagonal
>>A=A+diag(al,1)+diag(al,-1)+diag(v4,4)+diag(v4,-4) ;
>>A(1:8,1:8 ) %we make a quick check to see how A looks
->ans=    4 -3  0  0 -1  0  0  0
         -3  4 -3  0  0 -1  0  0
          0 -3  4 -3  0  0 -1  0
          0  0 -3  4  0  0  0 -1
         -1  0  0  0  4 -3  0  0
          0 -1  0  0 -3  4 -3  0
          0  0 -1  0  0 -3  4 -3
          0  0  0 -1  0  0 -3  4
```

The matrix A looks as it should. The vector b is, of course, easily constructed.

```
>>b=1:800; b=b'; %needed to take transpose to make b a column vector
```

Part (a):

```
>>c= cond(A,inf) -> c = 2.6257e + 003
```

With a condition number under 3000 , considering its size, the matrix A is rather well conditioned.

Part (b): Here and in part (c), we use the condition number formulations (49) and (51) for the error estimates of Theorems 7.7 and 7.8.

```
>> z=A\b; r=b-A*z; errest=c*norm(r,inf)/norm(A,inf)
-> errest = 2.4875e-010

>> relerrest=c*norm(r,inf)/norm(b,inf)
-> relerrest = 3.7313e-012
```

Part (c):

```
>> z2 = inv(A)*b; r2=b-A*z2; errest2=c*norm(r2,inf)/norm (A,inf)
-> errest2 = 6.8656e-009

>> relerrest2=c*norm(r2,inf)/norm(b,inf)
-> relerrest2 = 1 0298e-010
```

Both methods have produced solutions of very decent accuracy. All of the computations here were done with lightning speed. Thus even larger such systems (that are decently conditioned) can be dealt with safely with MATLAB's "left divide." The matrix in the above problem had a very high percentage of its entries being zeros. Such matrices are called sparse matrices, and MATLAB has efficient ways to store and manipulate such matrices. We will discuss this topic in the next section.

For (even moderately sized) poorly conditioned linear systems, quality control of computed solutions becomes a serious issue. The estimates provided in Theorems 7.7 and 7.8 are just that, estimates that give a guarantee of the closeness of the computed solution to the actual solution. The actual errors may be a lot smaller than the estimates that are provided. Another more insidious problem is that computation of the error bounds of these theorems is expensive, since it involves either the norm of A^{-1} directly or the condition number of A (which implicitly requires computing the norm of A^{-1}). Computer errors can lead to inaccurate computation of these error bounds that we would like to use to give us confidence in our numerical solutions. The next example will demonstrate and attempt to put into perspective some of these difficulties. The example will involve the very poorly conditioned Hilbert matrix that we introduced in Section 7.4. We will solve the system exactly (using MATLAB's symbolic toolbox),[4] and thus be able to compare estimated errors (using Theorems 7.7 and 7.8) with the actual errors. We warn the reader that some of the results of this example may be shocking, but we hasten to add that the Hilbert matrix is notorious for being extremely poorly conditioned.

[4] For more on the Symbolic Toolbox, see Appendix A. This toolbox may or may not be in the version of MATLAB that you are using. A reduced version of it comes with the student edition. It is not necessary to have it to understand this example.

EXAMPLE 7.23: Consider the linear system $Ax = b$ with

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{48} & \frac{1}{49} & \frac{1}{50} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{49} & \frac{1}{50} & \frac{1}{51} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{50} & \frac{1}{51} & \frac{1}{52} \\ \vdots & & & \ddots & & \vdots & \vdots \\ \frac{1}{48} & \frac{1}{49} & \frac{1}{50} & \cdots & \frac{1}{96} & \frac{1}{97} & \frac{1}{98} \\ \frac{1}{49} & \frac{1}{50} & \frac{1}{51} & \cdots & \frac{1}{97} & \frac{1}{98} & \frac{1}{99} \\ \frac{1}{50} & \frac{1}{51} & \frac{1}{52} & \cdots & \frac{1}{98} & \frac{1}{99} & \frac{1}{100} \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ 48 \\ 49 \\ 50 \end{bmatrix}.$$

Using MATLAB, perform the following computations.
(a) Compute the condition number of the 50×50 Hilbert matrix A (on MATLAB using the usual floating point arithmetic).
(b) Compute the same condition number using symbolic (exact) arithmetic on MATLAB.
(c) Use MATLAB's left divide to solve this system and label the computed solution as z, then use Theorem 7.7 to estimate the error. (Do not actually print the solution.)
(d) Solve the system by (numerically) left multiplying both sides by A^{-1} and label the computed solution as z2; then use Theorem 7.7 to estimate the error. (Do not actually print the solution.)
(e) Solve the system exactly using MATLAB's symbolic capabilities, label this exact solution as x, and compute the norm of this solution vector. Then use this exact solution to compute the exact errors of the two approximate solutions in parts (a) and (b).

SOLUTION: Since MATLAB has a built-in function for generating Hilbert matrices, we may very quickly enter the data A and b :

```
>> A=hilb(50); b=l:50;b=b';
```

Part (a): We invoke MATLAB's built-in function for computing condition numbers:

```
>> cl=cond(A,inf)
->Warning: Matrix is close to singular or badly scaled.
  Results may be inaccurate. RCOND = 3.615845e-020.
> In C:\MATLABR11\toolbox\matlab\matfun\cond.m at line 44
->c1 =5.9243e+019
```

This is certainly very large, but it came with a warning that it may be an inaccurate answer due to the poor conditioning of the Hilbert matrix. Let's now see what happens when we use exact arithmetic.

Part (b): Several of MATLAB's built-in functions are not defined for symbolic objects; and this is true for the norm and condition number functions. The way around this is to work directly with the definition (41) of the condition number: $\kappa(A) = \|A\| \left\|A^{-1}\right\|$, compute the norm of A directly (no computational difficulties here), and compute the norm of A^{-1} by first computing A^{-1} in exact arithmetic, then using the double command to put the answer from "symbolic" form into floating point form, so we can take its norm as usual (the computational difficulty is in computing the inverse, not in finding the norm).

```
>>c=norm(double(inv(sym(A))),inf)*norm(A, inf) % "sym" declares A as
a symbolic variable, so inv is calculated exactly; double switches
the symbolic answer back into floating point form.
->c1 = 4.33036 + 074
```

The difference here is astounding! This condition number means that, although the Hilbert matrix A has its largest entry being 1 (and smallest being $1/99$), the inverse matrix will have some entries having absolute values at least $4.33 \times 10^{74}/50 = 8.66 \times 10^{72}$ (why?). With floating point arithmetic, however, MATLAB's computed inverse has all entries less than 10^{20} in absolute value, so that MATLAB's inverse is totally out in left field!

Part (c):

```
>> z=A\b; r=b-A*z;
-> Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 3.615845e - 020.
```

As expected, we get a flag about our poorly conditioned matrix.

```
>> norm(r,inf) ->ans = 6.2943e - 005
```

Thus the residual of the computed solution is somewhat small. But the extremely large condition number of the matrix will overpower this residual to render the following useless error estimate (see (49)):

```
>> errest=cl*norm(r, inf)/norm(A, inf) ->errest = 9.5437e+014
```

Siece

```
>> norm(z, inf) ->ans = 5.0466e+012
```

this error estimate is over 100 times as large as the largest component of the numerical solution. Things get even worse (if you can believe this is possible) with the inverse multiplication method that we look at next.

Part (d):

```
>> z2=inv(A)*b; r2=b-A*z2;
->Warning: Matrix is close to singular or badly scaled.
  Results may be inaccurate. RCOND = 3.615845e - 020.
>>norm(r2,inf)      ->ans = 1.6189e + 004
```

Here, even the norm of the residual is unacceptably large.

```
>> errest2=cl*norm(r2,inf)/norm(A, inf)    ->errest = 2.2078e+023
```

Part (e):

```
>> S=sym(A); %declares A as a symbolic matrix
>> x=S\b; %Computes exact solution of system
>> x=double (x); %Converts x back to a floating point vector
>> norm (x, inf)  ->ans = 7.4601e + 04
```

We see that the solution vector has some extremely large entries.

```
>> norm (x-z, inf)  ->ans = 7.4601e + 040
>> norm (x-z2,inf)  ->ans = 7.4601e + 040
>> norm (z-z2, inf) ->ans = 3.8429e + 004
```

Comparing all of these norms, we see that the two approximations are closer to each other than to the exact solution (by far). The errors certainly met the estimates provided for in the theorem, but not by much. The (exact arithmetic) computation of JC took only a few seconds for MATLAB to do. Some comments are in order. The reader may wonder why one should not always work using exact arithmetic, since it is so much more reliable. The reasons are that it is often not necessary to do this floating point arithmetic usually provides acceptable (and usually decent) accuracy, and exact arithmetic is much more expensive. However, when we get such a warning from MATLAB about near singularity of a matrix we must discard the answers, or at least do some further analysis. Another option (again using the Symbolic Toolbox of MATLAB) would be to use variable precision arithmetic rather than exact arithmetic. This is less expensive than exact arithmetic and allows us to declare how many significant digits with which we would like to compute. We will give some examples of this arithmetic in a few rare cases where MATLAB's floating point arithmetic is not sufficient to attain the desired accuracy (see also Appendix A).

EXERCISE FOR THE READER 7.25: Repeat all parts of the previous example to the following linear system $Ax = b$, with:

$$A = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 & 1 \\ 2^{11} & 2^{10} & 2^{9} & \cdots & 4 & 2 & 1 \\ 3^{11} & 3^{10} & 3^{9} & \cdots & 9 & 3 & 1 \\ \vdots & & & \ddots & & \vdots & \vdots \\ 10^{11} & 10^{10} & 10^{9} & \cdots & 100 & 10 & 1 \\ 11^{11} & 11^{10} & 11^{9} & \cdots & 121 & 11 & 1 \\ 12^{11} & 12^{10} & 12^{9} & \cdots & 144 & 12 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ -2 \\ 3 \\ \vdots \\ -10 \\ 11 \\ -12 \end{bmatrix}.$$

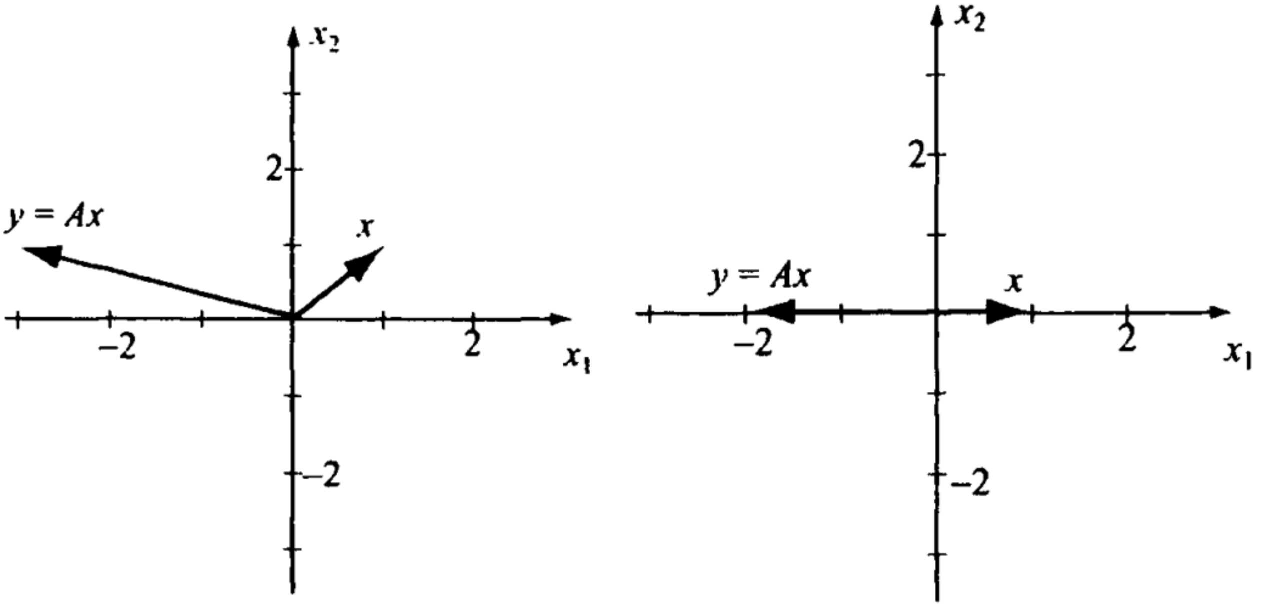
This coefficient matrix is the 12×12 Vandermonde matrix that was introduced in Section 7.4 with polynomial interpolation.

We next move on to the concepts of eigenvalues and eigenvectors of a matrix. These concepts are most easily motivated geometrically in two dimensions, so let us begin with a 2×2 matrix $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, and a nonzero column vector $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \neq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. We view A (as in Section 7.1) as a linear transformation acting on the vector x. The vector x will have a positive length given by the Pythagorean formula:

$$\text{len}(x) = \sqrt{x_1^2 + x_2^2} \tag{7.25}$$

Thus A transforms the two-dimensional vector x into another vector $y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$ by matrix multiplication $y = Ax$. We consider the case where y is also not the zero vector; this will always happen if A is nonsingular. In general, when we graph the vectors x and y together in the same plane, they can have different lengths as well as different directions (see Figure 7.38a).

Sometimes, however, there will exist vectors x for which y will be parallel to x (meaning that y will point in either the same direction or the opposite direction as x; see Figure 7.38b). In symbols, this would mean that we could write $y = \lambda x$ for some scalar (number) λ. Such a vector x is called an eigenvector for the matrix A, and the number λ is called an associated eigenvalue. Note that if λ is positive, then x and $y=\lambda x$ point in the same direction and len $(y)=\lambda\cdot\text{len}(x)$, so that λ acts as a magnification factor. If λ is negative, then (as in Figure 7.38b) y points in the opposite direction as x, and finally if $\lambda=0$, then y must be the zero vector (so has no direction). By convention, the zero vector is parallel to any vector. This is permissible, as long as x is not the zero vector. This definition generalizes to square matrices of any size.



Rysunek 7.4: Actions of the matrix $A=\begin{bmatrix}-2 & -1\\ 0 & 1\end{bmatrix}$ on a pair of vectors: (a) (left) The (shorter) vector $x=\begin{bmatrix}1\\1\end{bmatrix}$ of length $\sqrt{2}$ gets transformed to the vector $y=Ax=\begin{bmatrix}-3\\1\end{bmatrix}$ of length $\sqrt{10}$. Since the two vectors are not parallel, x is not an eigenvector of A. (b) (right) The (shorter) unit vector $x=\begin{bmatrix}1\\0\end{bmatrix}$ gets transformed to the vector $y=Ax=\begin{bmatrix}-2\\0\end{bmatrix}$ (red) of length 2 , which is parallel to x, therefore x is an eigenvector for A.

DEFINITION: Let A be an $n\times n$ matrix. An $n\times 1$ nonzero column vector x is called an **eigenvector** for the matrix A if for some scalar λ we have

$$Ax=\lambda x \tag{7.26}$$

The scalar λ is called the **eigenvalue** associated with the eigenvector x .

Finding all eigenvalues and associated eigenvectors for a given square matrix is an important problem that has been extensively studied and there are numerous algorithms devoted to this and related problems. It turns out to be useful to know this **eigendata** for a matrix A for an assortment of applications. It is actually quite easy to look at eigendata in a different way that will give an immediate method for finding it. We can rewrite equation (53) as follows:

$$Ax=\lambda x\Leftrightarrow Ax=\lambda Ix\Leftrightarrow \lambda Ix-Ax=0\Leftrightarrow(\lambda I-A)x=0.$$

Thus, using what we know about solving linear equations, we can restate the eigenvalue definition in several equivalent ways as follows:
λ is an eigenvalue of $A\Leftrightarrow(\lambda I-A)x=0$ has a nonzero solution x (which will be an eigenvector)

$$\Leftrightarrow\quad \det(\lambda I-A)=0$$

Thus the eigenvalues of A are precisely the roots λ of the equation $\det(\lambda I-A)=0$. If we write out this determinant with a bit more detail,

$$\det(\lambda I-A)=\det\begin{bmatrix}\lambda-a_{11} & -a_{12} & -a_{13} & \cdots & -a_{1,n-1} & -a_{1n}\\ -a_{21} & \lambda-a_{22} & -a_{23} & & & \\ -a_{31} & -a_{32} & \lambda-a_{33} & & & \vdots\\ \vdots & & & \ddots & & \vdots\\ -a_{n-1,1} & -a_{n-1,2} & -a_{n-1,3} & \cdots & \lambda-a_{n-1,n-1} & -a_{n-1,n}\\ -a_{n1} & -a_{n2} & -a_{n3} & \cdots & -a_{n,n-1} & \lambda-a_{nn}\end{bmatrix},$$

it can be seen that this expression will always be a polynomial of degree n in the variable λ, for any particular matrix of numbers $A=[a_{ij}]$ (see Exercise 30).This polynomial, because of its importance for the matrix A, is called the **characteristic polynomial** of the matrix A, and will be denoted as $p_A(\lambda)$. Thus $p_A(\lambda)=\det(\lambda I-A)$, and in summary:

The eigenvalues of a matrix A are the roots of the characteristic polynomial i.e., the solutions of the equation $p_A(\lambda) = 0$.

Finding eigenvalues is an algebraic problem that can be easily solved with MATLAB; more on this shortly. Each eigenvalue will always have associated eignvectors. Indeed, the matrix equation $(\lambda I - A)x = 0$ has a singular coefficient matrix when λ is an eigenvalue (since its determinant is zero). We know from our work on solving linear equations that a singular $(n \times n)$ linear system of form $Cx = 0$ can have either no solutions or infinitely many solutions, but $x = 0$ is (obviously) always a solution of such a linear system, and consequently such a singular system must have infinitely many solutions. To find eigenvectors associated with a particular eigenvalue λ, we could compute them numerically by applying `rref` rather than Gaussian elimination to the augmented matrix $[\lambda I - A|0]$. Although theoretically sound, this approach is not a very effective numerical method. Soon we will describe MATLAB's relevant built-in functions that are based on more sophisticated and effective numerical methods.

EXAMPLE 7.24: For the matrix $A = \begin{bmatrix} -2 & -1 \\ 1 & 1 \end{bmatrix}$, do the following:
(a) Find the characteristic polynomial $p_\lambda(\lambda)$.
(b) Find all roots of the characteristic polynomial (i.e., the eigenvalues of A).
(c) For each eigenvalue, find an associated eigenvector.
SOLUTION: Part (a): $p_A(\lambda) = \det(\lambda I - A) =$

$$\det\left(\begin{bmatrix} \lambda+2 & 1 \\ -1 & \lambda-1 \end{bmatrix}\right) = (\lambda+2)(\lambda-1)+1 = \lambda^2+\lambda-1$$

. Part (b): The roots of $p_A(\lambda) = 0$ are easily obtained from the quadratic formula:

$$\lambda = \frac{-1 \pm \sqrt{1-4\cdot 1\cdot(-1)}}{2} = \frac{-1 \pm \sqrt{5}}{2} \approx -1.6180, .6180.$$

Part (c): For each of these two eigenvalues, let's use `rref` to find (all) associated eigenvectors.
Case $\lambda = (-1-\sqrt{5})/2$:

```
>> A=[-2 -1; 1 1] ; lambda=(-1-sqrt(5))/2 ;
>> C=lambda*eye(2)-A; C(:,3)=zeros(2,1) ;
>> rref(C)
          1.0000 2.6180   0
->ans=         0      0   0
```

From the last matrix, we can read off the general solution of the system $(\lambda I - A)x = 0$ (written out to four decimals):

$$\begin{cases} x_1 + 2.6180x_2 = 0 \\ (= \text{ any number }) \end{cases} \Rightarrow \begin{cases} x_1 = -2.6180t \\ x_2 = t \end{cases}, t = \text{ any number}$$

These give, for all choices of the parameter t except $t = 0$, all of the associated eigenvectors. For a specific example, if we take $t = 1$, this will give us the eigenvector $x = \begin{bmatrix} -2.6180 \\ 1 \end{bmatrix}$. We can verify this geometrically by plotting the vector x along with the vector $y = Ax$ to see that they are parallel.

Case $\lambda = (-1+\sqrt{5})/2$:

```
>> lambda=(-1+sqrt(5))/2 ; C=lambda*eye(2)-A; rref(C)
->ans= 1.0000   0.3820   0
          0        0     0
```

As in the preceding case, we can get all of the associated eigenvectors. We consider the eigenvector $x = \begin{bmatrix} -.3820 \\ 1 \end{bmatrix}$ for this second eigenvalue. Since the eigenvalue is postive, x and Ax will point in the same directions, as can be checked. Of course, each of these eigenvectors has been written in format `short`; if we wanted we could have displayed them in format long and thus written our eigenvectors with more significant figures, up to about 15 (MATLAB's accuracy limit).

Before discussing MATLAB's relevant built-in functions for eigendata, we state a theorem detailing some useful facts about eigendata of a matrix. First we give a definition. Since an eigenvalue λ of a matrix A is a root of the characteristic polynomial $p_A(x)$, we know that $(x-\lambda)$ must be a factor of $p_A(x)$. Recall that the **algebraic multiplicity** of the root λ is the highest exponent m such that $(x-\lambda)^m$ is still a factor of $p_A(x)$, i.e., $p_A(x) = (x-\lambda)^m q(x)$ where $q(x)$ is a polynomial (of degree $n-m$) such that $q(\lambda) \neq 0$.

THEOREM 7.9: (Facts about Eigenvalues and Eigenvectors): Let $A = [a_{ij}]$ be an $n \times n$ matrix.
(i) The matrix A has at most n (real) eigenvalues, and their algebraic multiplicities add up to at most n.
(ii) If u, w are both eigenvectors of A corresponding to the same eigenvalue λ, then $u+w$ (if nonzero) is also an eigenvector

of A corresponding to λ, and if c is a nonzero constant, then cu is also an eigenvector of A.[5] The set of all such eigenvectors associated with λ, together with the zero vector, is called the **eigenspace** of A **associated with the eigenvalue** λ.
(iii) The dimension of the eigenspace of A associated with the eigenvalue λ, called the **geometric multiplicity of the eigenvalue** λ, is always less than or equal to the algebraic multiplicity of the eigenvalue λ.
(iv) In general a matrix A need not have any (real) eigenvalues,[6] but if A is a **symmetric matrix** (meaning: A coincides with its transpose matrix), then A will always have a full set of n real eigenvalues, provided each eigenvalue is repeated according to its geometric multiplicity.

The proofs of (i) and (ii) are rather easy; they will be left as exercises. The proofs of (iii) and (iv) are more difficult; we refer the interested reader to a good linear algebra textbook, such as [HoKu-71], [Kol-99] or [Ant-00]. There is an extensive theory and several factorizations associated with eigenvalue problems. We should also point out a couple of more advanced texts. The book [GoVL-83] has become the standard reference for numerical analysis of matrix computations. The book [Wil-88] is a massive treatise entirely dedicated to the eigenvalue problem; it remains the standard reference on the subject. Due to space limitations, we will not be getting into comprehensive developments of eigenalgorithms; we will merely give a few more examples to showcase MATLAB's relevant built-in functions.

EXAMPLE 7.25: Find a matrix A that has no real eigenvalues (and hence no eigenvectors), as indicated in part (iv) of Theorem 7.9.

SOLUTION: We should begin to look for a 2×2 matrix A. We need to find one for which its characteristic polynomial $p_A(\lambda)$ has no real root. One approach would be to take a simple second-degree polynomial, that we know does not have any real roots, like $\lambda^2 + 1$, and try to build a matrix $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ which has this as its characteristic polynomial. Thus we want to choose a, b, c, and d such that:

$$\det \begin{pmatrix} \lambda - a & b \\ c & \lambda - d \end{pmatrix} = \lambda^2 + 1$$

If we put $a = d = 0$, and compute the determinant we get $\lambda^2 - bc = \lambda^2 + 1$, so we are okay if $bc = -1$. For example, if we take $b = 1$ and $c = -1$, we get that the matrix $A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ has no real eigenvalues.

MATLAB has the built-in function eig that can find eigenvalues and eigenvectors of a matrix. The two possible syntaxes of this function are as follows:

norm $(x) \to$	If A is a square matrix, this command produces a vector containing the eigenvalues of A. Both real and complex eigenvalues are given. x.
norm $(x,\ \inf) \to$	If A is an $n \times n$ matrix, this command will create two nxn matrices. D is a diagonal matrix whose diagonal entries are the eigenvalues of A, and V is matrix whose columns are corresponding eigenvectors for A. For complex eigenvalues, the corresponding eigenvectors will also be complex.

Since, by Theorem 7.9 (ii), any nonzero scalar multiple of an eigenvector is again an eigenvector, MATLAB's eig chooses its eigenvectors to have length = 1.

For example, let's use these commands to find eigendata for the matrix of Example 7.24:

```
>> [V,D]=eig([- 2 -1;1 1])
-> V=     -0.9342    0.3568
           0.3568   -0.9342

->D=      -1.6180       0
               0     0.6180
```

The diagonal entries in the matrix D are indeed the eigenvalues (in short format) that we found in the example. The corresponding eigenvectors (from the columns of) $V \begin{bmatrix} -0.9342 \\ 0.3568 \end{bmatrix}$ and $\begin{bmatrix} 0.3568 \\ -0.9342 \end{bmatrix}$ are different from the two we gave in that example, but can be obtained from the general form of eigenvectors that was found in that example. Also, unlike those that we gave in the example, it can be checked that these two eigenvectors have length equal to 1 .

[5] Thus when we throw all eigenvectors associated with a particular eigenvalue λ of a matrix A together with the zero vector, we get a set of vectors that is closed under the two linear operations: vector additon and scalar multiplication. Readers who have studied linear algebra will recognize such a set as a vector space; this one is called the eigenspace associated with the eigenvalue λ of the matrix A. Geometrically the eigenspace will be either a line through the origin (one-dimensional), a plane through the origin (two-dimensional), or in general, any k-dimensional hyperplane through the origin $(k \leq n)$.

[6] This is reasonable since, as we have seen before, a polynomial need not have any real roots (e.g., $x_2 + 1$). If complex numbers are considered, however, a polynomial will always have a complex root (this is the so-called "Fundamental Theorem of Algebra") and so any matrix will always have at least a complex eigenvalue. Apart from this fact, the theory for complex eigenvalues and eigenvectors parallels that for real eigendata.

In the case of an eigenvalue with geometric multiplicity greater than 1 , eig will find (whenever possible) corresponding eigenvectors that are linearly independent.[7] Watch what happens when we apply the eig function to the matrix that we constructed in Example 7.25:

```
>> [v=D]=eig((0 1;-1 0])
->V =     0.7071          0.7071
          0 + 0.7071i     0-0.7071i

->D=      0 + 1.0000i     0
          0               0-1.0000i
```

We get the eigenvalues (from D) to be the complex numbers $\pm i$ (where $i=\sqrt{-1}$) and the two corresponding eigenvectors also have complex numbers in them. Since we are interested only in real eigendata, we would simply conclude from such an output that the matrix has no real eigenvalues.

If you are interested in finding the characteristic polynomial $p_A(\lambda)=a_n\lambda^n+ +a_{n-1}\lambda^{n-1}\cdots+a_1\lambda+a_0$ of an $n\times n$ matrix A, MATLAB has a function `poly` that works as follows:

poly(A) →	For an nxn matrix A, this command will produce the vector v = $[a_na_{n-1}\ldots a_1a_0]$of the n + 1 coefficients of the nth-degree characteristic polynomial $p_A(\lambda)=\det(\lambda I-A)=a_n\lambda^n+a_{n-1}\lambda^{n-1}+\cdots+a_1\lambda+a_0$ of the matrix A.

For example, for the matrix we constructed in Example 7.25, we could use this command to check its characteristic polynomial:

```
>> poly([ 0 1;- 1 0])      ->ans = 1   0   1
```

which translates to the polynomial $1\cdot\lambda^2+0\cdot\lambda+1=\lambda^2+1$, as was desired. Of course, this command is particularly useful for larger matrices where computation of determinants by hand is not feasible. The MATLAB function `roots` will find the roots of any polynomial:

roots (v) →	For a vector $\begin{bmatrix} a_n & a_{n-1} & \cdots & a_1 & a_0 \end{bmatrix}$ of the $n+1$ coefficients of the nth-degree polynomial $p(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$ this comman will produce a vector of length n containing all of the roots of $p(x)$ (real and complex) repeated according to algebraic multiplicity.

Thus, another way to get the eigenvalues of the matrix of Example 7.24 would be as followsThus, another way to get the eigenvalues of the matrix of Example 7.24 would be as follows:[8]

```
>> roots(poly([- 2 -1; 1 1]))
-> ans      -1.6180
             0.6180
```

EXERCISE FOR THE READER 7.26: For the matrix $A=\begin{bmatrix} 2 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, do the following:

(a) By hand, compute $p_A(\lambda)$, the characteristic polynomial of A, in factored form, and find the eigenvalues and the algebraic multiplicity of each one.

(b) Either by hand or using MATLAB, find all the corresponding eigenvectors for each eigenvalue and find the geometric multiplicity of each eigenvalue.

(c) After doing part (a), can you figure out a general rule for finding the eigenvalues of an upper triangular matrix?

EXERCISES 7.6:

1. For each of the following vectors x, find len(x) and find $\|x\|$.

$$x=[2,-6,0,3]$$
$$x=[\cos(n),\sin(n),3^n]\ (n=\text{ a positive integer})$$
$$x=[1,-1,1,-1,\cdots,1,-1]\ \text{(vector has } 2n \text{ components)}$$

2. For each of the following matrices A , find the infinity norm $\|A\|$.

[7] "Linear independence is a concept from linear algebra. What is relevant for the concept at hand is that any other eigenvector associated with the same eigenvalue will be expressible as a linear combination of eigenvectors that MATLAB produces. In the parlance of linear algebra, MATLAB will produce eigenvectors that form a basis of the corresponding eigenspaces.

[8] We mention, as an examination of the M-file will show (enter `type roots`), that the roots of a polynomial are found using the `eig` command on an associated matrix.

a) $A = \begin{bmatrix} 2 & -3 \\ 1 & 6 \end{bmatrix}$ b) $A = \begin{bmatrix} 4 & -5 & -2 \\ 1 & 2 & 3 \\ -2 & -4 & -6 \end{bmatrix}$

c) $A = \begin{bmatrix} \cos(\pi/4) & -\sin(\pi/4) & 0 \\ \sin(\pi/4) & \cos(\pi/4) & 0 \\ 0 & 0 & 1 \end{bmatrix}$ d) $A = H_n$ the $n \times n$ Hilbert matrix (introduced and defined in Example 7.8)

3. For each of the matrices A (parts (a) through (d)) of Exercise 2, find a nonzero vector x such that $\|Ax\| = \|A\|\|x\|$.

4. For the matrix $A = \begin{bmatrix} 2 & -3 \\ -2 & 4 \end{bmatrix}$, calculate by hand the following: $\|A\|, \left\|A^{-1}\right\|, \kappa(A)$, and then verify your calculations with MATLAB. If possible, find a singular matrix S such that $\|S-A\| = 1/3$.

5. For the matrix $A = \begin{bmatrix} 1 & -1 \\ -1 & 1.001 \end{bmatrix}$, calculate by hand the following: $\|A\|, \left\|A^{-1}\right\|, \kappa(A)$, and then verify your calculations with MATLAB. Is there a singular matrix S such that $\|S-A\| = 1/1000$? Explain.

6. Consider the matrix $B = \begin{bmatrix} 2.6 & 0 & -3.2 \\ 3 & -8 & -4 \\ 1 & 2 & -1 \end{bmatrix}$.
(a) Is there a nonzero (3×1) vector x such that $\|Bx\| \geq 13\|x\|$? If so, find one; otherwise explain why one does not exist.
(b) Is there a singular 3×3 matrix S such that $\|S-B\| \leq 0.2$? If so, find one; otherwise explain why one does not exist.

7. Consider the matrices: $A = \begin{bmatrix} 2 & -6 \\ 11 & -5 \end{bmatrix}, B = \begin{bmatrix} 7 & 1 & -4 \\ 5 & -8 & -5 \\ 4 & 4 & 4 \end{bmatrix}$.
(a) Is there a (2×1) vector X such that: $\|AX\| > 12\|X\|$?
(b) Is there a nonzero vector X such that $\|AX\| \geq 16\|X\|$? If so, find one; otherwise explain why one does not exist.
(c) Is there a nonzero (3×1) vector X such that $\|BX\| \geq 20\|X\|$? If so, find one; otherwise explain why one does not exist.
(d) Is there a singular matrix $S = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ (i.e., $ad - bc = 0$) such that $\|S-A\| \leq 4.5$? If yes, find one; otherwise explain why one does not exist.
(c) Is there a singular 3×3 matrix S such that $\|S-B\| \leq 2.25$? If so find one; otherwise explain why one does not exist.

8. Prove identities (42), (43), and (44) for condition numbers.
Suggestion: The identity that was established in Exercise for the Reader 7.24 can be helpful.

9. *(True/False)* For each statement below, either explain why it is always true or provide a counterexample of a single situation where it is false:
(a) If A is a square matrix with $\|A\| = 0$, then $A = 0$ (matrix), i.e., all the entries of A are zero.
(b) If A is any square matrix, then $|\det(A)| \leq \kappa(A)$.
(c) If A is a nonsingular square matrix, then $\kappa\left(A^{-1}\right) = \kappa(A)$.
(d) If A is a square matrix, then $\kappa(A') = \kappa(A)$.
(e) If A and B are same-sized square matrices, then $\|AB\| \leq \|A\|\|B\|$.
Suggestion: As is always recommended, unless you are sure about any of these identities, run a bunch of experiments on MATLAB (using randomly generated matrices).

10. Prove identity (39). Suggestion: Reread Example 7.19 and the note that follows it for a useful idea.

11. (A General Class of Norms) For any real number $p \geq 1$, the p-norm $\|\cdot\|_p$ defined for an n dimensional vector x by the equation

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p} = \left(|x_1|^p + |x_2|^p + \cdots + |x_n|^p\right)^{1/p}$$

turns out to satisfy the norm axioms $(36A-C)$. In the general setting, the proof of the triangle inequality is a bit involved (we refer to one of the more advanced texts on error analysis cited in the section for details).
(a) Show that len $(x) = \|x\|_2$ (this is why the length norm is sometimes called the 2− norm).
(b) Verify the norm axioms $(36A-C)$ for the 1 -norm $\|\cdot\|_1$.
(c) For a vector x, is it always true that $\|x\|_\infty \leq \|x\|_1$? Either prove it is always true or give a counterexample of an instance of a certain vector x for which it fails.
(d) For a vector x, is it always true that $\|x\|_\infty \leq \|x\|_2$? Either prove it is always true or give a counterexample of an instance of a certain vector x for which it fails.

(e) How are the norms $\|\cdot\|_1$ and $\|\cdot\|_1$ related? Does one always seem to be at least as large as the other? Do (lots of) experiments with some randomly generated vectors of different sizes.
Note: Experiments will probably convince you of a relationship (and inequality) for part (c), but it might be difficult to prove, depending on your background; the interested reader can find a proof in one of the more advanced references listed in the section. The reason that infinity norm got this name is that for any fixed vector x, we have

$$\lim_{p\to\infty}\|x\|_p = \|x\|_\infty.$$

As the careful reader might have predicted, the MATLAB built-in function for the p-norm of a vector x is norm (x,p).

12. Let $A = \begin{bmatrix} -11 & 3 \\ 4 & -1 \end{bmatrix}, b = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \quad z = \begin{bmatrix} 1.2 \\ 7.8 \end{bmatrix}$
(a) Let z be an approximate solution of the system $Ax = b$; fond the residual vector r.
(b) Use Theorem 7.7 to give an estimate for the error of the approximation in part (a).
(c) Give an estimate for the relative error of the approximation in part (a).
(d) Find the norm of the exact error $\|\Delta x\|$ of the approximate solution in part (a).

13. Repeat all parts of Exercise 12 for the following matrices:

$$A = \begin{bmatrix} -0.1 & -9 \\ 11 & 1000 \end{bmatrix}, \quad b = \begin{bmatrix} -0.1 \\ 10 \end{bmatrix} \quad z = \begin{bmatrix} 11 \\ -0.2 \end{bmatrix}$$

14. Let $A = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 1 & 1 \\ -1 & -1 & 1 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$
(a) Use MATLAB's "left divide" to solve the system $Ax = b$.
(b) Use Theorems 7.7 and 7.8 to estimate the error and relative error of the solution obtained in part (a).
(c) Use MATLAB to solve the system $Ax = b$ by left multiplying the equation by the inv (A).
(d) Use Theorems 7.7 and 7.8 to estimate the error and relative error of the solution obtained in part (c).
(e) Solve the system using MATLAB's symbolic capabilities and compute the actual errors of the solutions obtained in parts (a) and (c).

15. Let A be the 60×60 matrix whose entries are 1 's across the main diagonal and the last column, -1 's below the main diagonal, and whose remaining entries (above the main diagonal) are zeros, and let $b = \begin{bmatrix} 1 & 2 & 3 & \cdots & 58 & 59 & 60 \end{bmatrix}'$.
(a) Use MATLAB's left divide to solve the system $Ax = b$ and label this computed solution as z. Print out only $z(37)$.
(b) Use Theorems 7.7 and 7.8 to estimate the error and relative error of the solution obtained in part (a).
(c) Use MATLAB to solve the system $Ax = b$ by left multiplying the equation by the inv (A) and label this computed solution as $z2$. Print out only $z2(37)$.
(d) Use Theorems 7.7 and 7.8 to estimate the error and relative error of the solution obtained in part (c).
(e) Solve the system using MATLAB's symbolic capabilities, and print out $\times(37)$ of the exact solution. Then compute the norms of the actual errors of the solutions obtained in parts (a) and (c).

16. *(Iterative Refinement)* Let z_0 be the computer solution of Exercise $15(\mathrm{a})$, and let r_0 denote the corresponding residual vector. Now use the Gaussian program to solve the system $Ax = r_0$, and call the computer solution z_1, and the corresponding residual vector r_1. Next use the Gaussian program once again to solve $Ax = r_1$, and let z_2 and r_2 denote the corresponding approximate solution and residual vector. Now let

$$z = z_0, \quad z' = z + z_1, \text{ and } z'' = z' + z_2.$$

Viewing these three vectors as solutions of the original system $Ax = b$ (of Exercise 15), use the error estimate Theorem 7.8 to estimate the relative error of each of these three vectors. Then compute the norm of the actual errors by comparing with the exact solution of the system as obtained in part (e) of Exercise 15. See the n

17. In theory, the iterative technique of the previous exercise can be useful to improving accuracy of approximate solutions in certain circumstances. In practice, however, roundoff errors and poorly conditioned matrices can lead to unimpressive results. This exercise explores the effect that additional digits of precision can have on this scheme.
(a) Using variable precision arithmetic (see Appendix A) with 30 digits of accuracy, redo the previous exercise, starting with the computed solution of Exercise 15(a) done in MATLAB's default floating point arithmetic.
(b) Using the same arithmetic of part (a), solve the original system using MATLAB's left divide.
(c) Compare the norms of the actual errors of the three approximate solutions of part (a) and the one of part (b) by using symbolic arithmetic to get MATLAB to compute the exact solution of the system.
Note: We will learn about different iterative methods in the next section.

18. This exercise will examine the benefits of variable precision arithmetic over MATLAB's default floating point arithmetic and over MATLAB's more costly symbolic arithmetic. As in Section 7.4, we let $H_n = [1/(i+j-1)]$

denote the $n \times n$ Hilbert matrix. Recall that it can be generated in MATLAB using the command `hilb (n)`.
(a) For the values $n = 5, 10, 15, \cdots, 100$ create the corresponding Hilbert matrices H_n in MATLAB as symbolic matrices and compute symbolically the inverses of each. Use `tic/toc` to record the computation times (these times will be machine dependent; see Chapter 4). Go as far as you can until your cumulative MATLAB computation time exceeds one hour. Next compute the corresponding condition numbers of each of these Hilbert matrices.
(b) Starting with MATLAB's default floating point arithmetic (which is roughly 15 digits of variable precision arithmetic), and then using variable precision arithmetic starting with 20 digits and then moving up in increments of $5(25, 30, 35, \ldots)$, continue to compute the inverses of each of the Hilbert matrices of part (a) until you get a computed inverse whose norm differs from the norm of the exact inverse in part (a) by no more than 0.000001. Record (using `tic/toc` the computation time for the final variable precision arithmetically computed inverse, along with the number of digits used, and compare it to the corresponding computation time for the exact inverse that was done in part (a).

19. Prove the following inequality $||Ax|| \geq ||x||/||A^{-1}||$, where A is any invertible $n \times n$ matrix and x is any column vector with n entries.

20. Suppose that A is a 2×2 matrix with norm $||A|| = 0.5$ and x and y are 2×1 vectors with $||x - y|| \leq 0.8$. Show that: $||A^2x - A^2y|| \leq 0.2$.

21. *(Another Error Bound for Computed Solutions of Linear Systems)* For a nonsingular matrix A and a computed inverse matrix C for A^{-1}, we define the resulting **residual matrix** as $R = I - CA$. If z is an approximate solution to $Ax = b$, and as usual $r = b - Az$ is the residual vector, show that

$$\text{error } \equiv ||x - z|| \leq \frac{||CR||}{1 - ||R||}$$

provided that $||R|| < 1$.
Hint: For part (a), first use the equation $I - R = CA$ to get that $(I - R)^{-1} = A^{-1}C^{-1}$ and so $A^{-1} = (I - R)^{-1}C$. (Recall that the inverse of a product of invertible matrices equals the reverse order product of the inverses.)

22. For each of the matrices A below, find the following:
(a) The characteristic polynomial $p_A(\lambda)$.
(b) All eigenvalues and all of their associated eigenvectors.
(c) The algebraic and geometric multiplicity of each eigenvalue.

$$\text{(i) } A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \quad \text{(ii) } A = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} \quad \text{(iii) } A = \begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix} \quad \text{(iv) } A = \begin{bmatrix} -1 & 0 \\ 2 & 3 \end{bmatrix}$$

23. Repeat all parts of Exercise 22 for the following matrices.

$$\text{(i) } A = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix} \qquad \text{(ii) } A = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 2 \\ 0 & 2 & 1 \end{bmatrix}$$

$$\text{(iii) } A = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{(iv) } A = \begin{bmatrix} 1 & 2 & 2 \\ -2 & 1 & 2 \\ -2 & -2 & 1 \end{bmatrix}$$

24. Consider the matrix $A = \begin{bmatrix} 11 & 11 & 4 \\ 7 & 7 & -4 \\ -7 & -11 & 0 \end{bmatrix}$.
(a) Find all eigenvalues of A, and for each find just one eigenvector (give your eigenvectors as many integer components as possible).
(b) For each of the eigenvectors x that you found in part (a), evaluate $y = (2A)x$. Is it possible to write $y = \lambda x$ for some scalar λ ? In other words, is x also an eigenvector of the matrix $2A$?
(c) Find all eigenvalues of $2A$. How are these related to those of the matrix A ?
(d) For each of your eigenvectors x from part (a), evaluate $y = (-5A)x$. Is it possible to write $y = \lambda x$ for some scalar λ ? In other words, is x also an eigenvector of the matrix $-5A$?
(e) Find all eigenvalues of $-5A$; how are these related to those of the matrix A ?
(f) Based on your work in these above examples, without picking up your pencil or typing anything on the computer, what do you think the eigenvalues of the matrix 23 A would be? Could you guess also some associated eigenvectors for each eigenvalue? Check your conclusions on MATLAB.

25. Consider the matrix $A = \begin{bmatrix} 2 & 0 & 1 \\ 1 & -4 & 1 \\ 1 & 0 & 2 \end{bmatrix}$.
(a) Find all eigenvalues of A, and for each find just one eigenvector (give your eigenvectors as many integer components as possible).

(b) For each of the eigenvectors x that you found in part (a), evaluate $y = A^2x$. Is it possible to write $y = \lambda x$ for some scalar λ ? In other words, is x also an eigenvector of the matrix A^2 ?
(c) Find all eigenvalues of A^2; how are these related to those of the matrix A ?
(d) For each of your eigenvectors x from part (a), evaluate $y = A^3x$. Is it possible to write $y = \lambda x$ for some scalar λ? In other words, is x also an eigenvector of the matrix A^3 ?
(e) Find all eigenvalues of A^3; how are these related to those of the matrix A ?
(f) Based on your work in the above examples, without picking up your pencil or typing anything on the computer, what do you think the eigenvalues of the matrix A^8 would be? Could you guess also some associated eigenvectors for each eigenvalue? Check your conclusions on MATLAB.

26. Find the characteristic polynomial (factored form is okay) as well as all eigenvalues for the $n \times n$ identity matrix I. What are (all) of the corresponding eigenvectors (for each eigenvalue)?

27. Consider the matrix $A = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 2 & 4 \\ 3 & 4 & 2 \end{bmatrix}$.
(a) Find all eigenvalues of A, and for each find just one eigenvector (give your eigenvectors as many integer components as possible).
(b) For each of the eigenvectors x that you found in part (a), evaluate $y = \left(A^2 + 2A\right)x$. Is it possible to write $y = \lambda x$ for some scalar λ? In other words, is x also an eigenvector of the matrix $A^2 + 2A$?
(c) Find all eigenvalues of $A^2 + 2A$; how are these related to those of the matrix A ?
(d) For each of your eigenvectors x from Part (a), evaluate $y = \left(A^3 - 4A^2 + I\right)x$. Is it possible to write $y = \lambda x$ for some scalar λ ? In other words, is x also an eigenvector of the matrix $A^3 - 4A^2 + I$?
(e) Find all eigenvalues of $A^3 - 4A^2 + I$; how are these related to those of the matrix A ?
(f) Based on your work in the above examples, without picking up your pencil or typing anything on the computer, what do you think the eigenvalues of the matrix $A^5 - 4A^3 + 2A - 4I$ would be? Could you guess also some associated eigenvectors for each eigenvalue? Check your conclusions on MATLAB.

NOTE: The spectrum of a matrix A, denoted $\sigma(A)$, is the set of all eigenvalues of the matrix A. The next exercise generalizes some of the results discovered in the previous four exercises.

28. For a square matrix A and any polynomial $p(x) = a_m x^m + a_{m-1}x^{m-1} + \cdots + a_1 x + a_0$, we define a new matrix $p(A)$ as follows:

$$p(A) = a_m A^m + a_{m-1}A^{m-1} + \cdots + a_1 A + a_0 I.$$

(We simply substituted A for x in the formula for the polynomial; we also had to replace the constant term a_0 by this constant times the identity matrix–the matrix analogue of the number 1.) Prove the following appealing formula;

$$\sigma(p(A)) = p(\sigma(A))$$

which states that the spectrum of the matrix $p(A)$ equals the set

$$\{p(\lambda) : \lambda \text{is an eigenvalue of A}\}.$$

29. Prove parts (i) and (ii) of Theorem 7.9.

30. Show that the characteristic polynomial of any $n \times n$ matrix is always a polynomial of degree n in the variable λ. **Suggestion:** Use induction and cofactor expansion.

31. (a) Use the basic Gaussian elimination algorithm (Program 7.6) to solve the linear systems of Exercise for the Reader 7.16, and compare with results obtained therein.
(b) Use the Symbolic Toolbox to compute the condition numbers of the Hilbert matrices that came up in part (a). Are the estimates provided by Theorem 7.8 accurate or useful?
(c) Explain why the algorithm performs so well with this problem despite the large condition numbers of A. **Suggestion:** For part (c), examine what happens after the first pivot operation.

7.7. ITERATIVE METHODS

As mentioned earlier in this chapter, Gaussian elimination is the best all-around solver for nonsingular linear systems $Ax = b$. Being a universal method, however, there are often more economical methods that can be used for particular forms of the coefficient matrix. We have already seen the tremendous savings, both in storage and in computations that can be realized in case A is tridiagonal, by using the Thomas method. All methods considered thus far have been direct methods in that, mathematically, they compute the exact solution and the only errors that arise are numerical. In this section we will introduce a very different type of method called an **iterative method**. Iterative methods begin with an initial guess at the (vector) solution $x^{(0)}$, and produce a sequence of vectors, $x^{(1)}, x^{(2)}, x^{(3)}, \cdots$, which, under certain circumstances, will converge to the exact solution. Of course, in any floating point arithmetic system, a solution from an iterative method (if the method converges) can be made just as accurate as that of a direct method.

In solving differential equations with so-called finite difference methods, the key numerical step will be to solve a linear system $Ax = b$ having a large and sparse coefficient matrix A (a small percentage of nonzero entries) that will have a special form. The large size of the matrix will often make Gaussian elimination too slow. On the other hand, the special structure and sparsity of A can make the system amenable to a much more efficient iterative method. We have seen that in general Gaussian elimination for solving an n variable linear system performs in $O\left(n^3\right)$-time. We take this as the ceiling performance time for any linear system solver. The Thomas method, on the other hand, for the very special triadiagonal systems, performed in only $O(n)$-time. Since just solving n independent linear equations (i.e., with A being a diagonal matrix) will also take this amount, this is the theoretical floor performance time for any linear system solver. Most of iterative methods today perform theoretically in $O\left(n^2\right)$-time, but in practice can perform in times closer to the theoretical floor $O(n)$-time. In recent years, iterative methods have become increasingly important and have a promising future, as increasing computer performance will make the improvements over Gaussian elimination more and more dramatic.

We will describe three common iterative methods: Jacobi, Gauss-Seidel, and SOR iteration. After giving some simple examples showing the sensitivity of these methods to the particular form of A, we give some theoretical results that will guarantee convergence. We then make some comparisons among these three methods in flop counts and computation times for larger systems and then with Gaussian elimination. The theory of iterative methods is a very exciting and interesting area of numerical analysis. The Jacobi, Gauss-Seidel, and SOR iterative methods are quite intuitive and easy to develop. Some of the more stateof-the-art methods such as conjugate gradient methods and GMRES (generalized minimum residual method) are more advanced and would take a lot more work to develop and understand, so we refer the interested reader to some references for more details on this interesting subject: [Gre-97], [TrBa-97], and [GoVL-83]. MATLAB, however, does have some built-in functions for performing such more advanced iterative methods. We introduce these MATLAB functions and do some performance comparisons involving some (very large and) typical coefficient matrices that arise in finite difference schemes.

We begin with a nonsingular linear system:

$$Ax = b \tag{7.27}$$

In scalar form, it looks like this:

$$a_{i1}x_1 + a_{i2}x_2 + \cdots a_{in}x_n = b_i \quad (1 \le i \le n) \tag{7.28}$$

Now, if we assume that each of the diagonal entries of A are nonzero, then each of the equations in (55) can be solved for x_i to arrive at:

$$x_i = \frac{1}{a_{ii}}\left[b_i - a_{i1}x_1 - a_{i2}x_2 - \cdots - a_{i,i-1}x_{i-1} - a_{i,i+1}x_{i+1} - \cdots a_{in}x_n\right](1 \le i \le n). \tag{7.29}$$

The Jacobi iteration scheme is obtained from using formula (56) with the values of current iteration vector $x^{(k)}$ on the right to create, on the left, the values of the next iteration vector $x^{(k+1)}$. We record the simple formula:

Jacobi Iteration:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{\substack{j=1,\\ j\neq i}}^{n} a_{ij}x_j^{(k)}\right](1 \le i \le n) \tag{7.30}$$

Let us give a (very) simple example illustrating this scheme on a small linear system and compare with the exact solution.

EXAMPLE 7.26: Consider the following linear system:

$$\begin{aligned} 3x_1 + x_2 - x_3 &= -3 \\ 4x_1 - 10x_2 + x_3 &= 28. \\ 2x_1 + x_2 + 5x_3 &= 20 \end{aligned}$$

(a) Starting with the vector $x^{(0)} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$ ', apply the Jacobi iteration scheme with up to 30 iterations until (if ever) the 2 -norm of the differences $x^{(k+1)} - x^{(k)}$ is less than 10^{-6}. Plot the norms of these differences as a function of the iteration. If convergence occurs, record the number of iterations and the actual 2-norm error of the final iterant with the exact solution.
(b) Repeat part (a) on the equivalent system obtained by switching the first two equations.

SOLUTION: Part (a): The Jacobi iteration scheme (57) becomes:

$$\begin{aligned} x_1^{(k+1)} &= \left(-3 - x_2^{(k)} + x_3^{(k)}\right)/3 \\ x_2^{(k+1)} &= \left(28 - 4x_1^{(k)} - x_3^{(k)}\right)/(-10). \\ x_3^{(k+1)} &= \left(20 - 2x_1^{(k)} - x_2^{(k)}\right)/5 \end{aligned}$$

The following MATLAB code will perform the required tasks:

```
xold = [0 0 0]' ; xnew=xold;
for k=1:30
  xnew(1)=(-3-xold(2)+xold(3))/3;
  xnew(2) = (28-4*xold(1)-xold(3))/(-10) ;
  xnew(3)=(20-2*xold(1)-xold(2))/5;
  diff(k)=norm(xnew-xold, 2) ;
  if diff(k)<1e-6
    fprintf('Jacobi iteration has converged in %d iterations', k)
    return
  end
  xold=xnew;
end
-> Jacobi iteration has converged in 26 iterations
```

The exact solution is easily seen to be $\begin{bmatrix} 1 & -2 & 4 \end{bmatrix}'$. The exact 2-norm error is thus given by:

```
>>norm(xnew-[1 -2 4]',2) -> ans = 3.9913e-007
```

which compares favorably with the norm of the last difference of the iterates (i.e., the actual error is smaller):

```
>> diff(k) ->ans = 8.9241e-007
```

We will see later in this section that finite difference methods typically exhibit linear convergence (if they indeed converge); the quality of convergence will thus depend on the asymptotic error constant (see Section 6.5 for the terminology). Due to this speed of the decay of errors, an ordinary plot will not be so useful (as the reader should verify), so we use a log scale on the *y*-axis. This is accomplished by the following MATLAB command:

semilogy(x , y) →	If x and y are two vectors of the same size, this will produce a plot where the .y-axis numbers are logarithmically spaced rather than equally spaced as with `plot(x, y)`.*x*.
semilogy(x , y) →	Works as the above command, but now the x-axis numbers are logarithmically spaced.

The required plot is now created with the following command and the result is shown in Figure 7.39(a).

```
>>semilogy(1:k,diff(1:k))
```

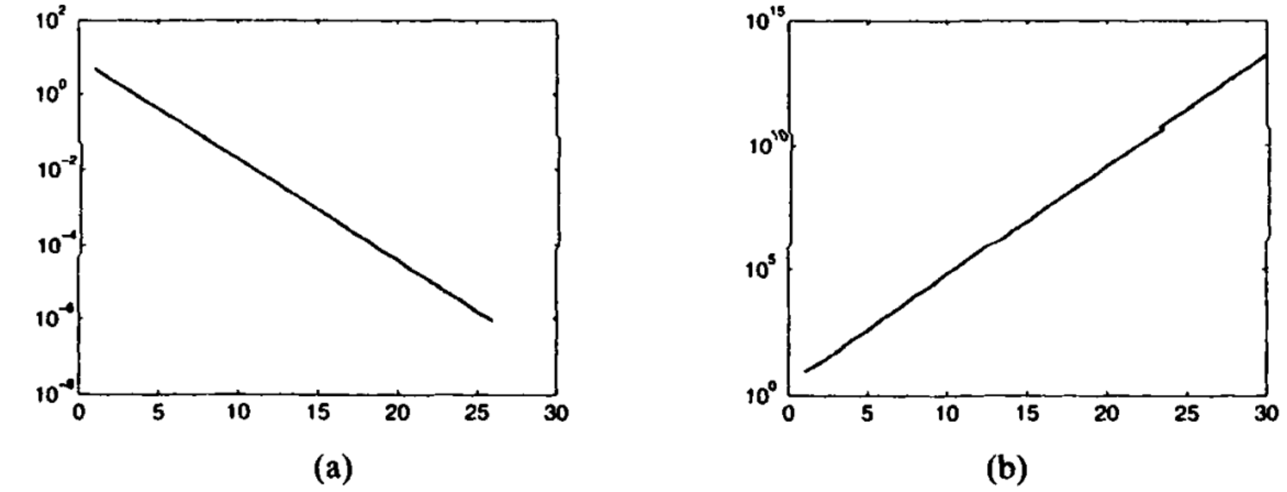
Part (b): Switching the first two equations of the given system leads to the following modified Jacobi iteration scheme:

$$\begin{aligned} x_1^{(k+1)} &= \left(28 + 10x_2^{(k)} - x_3^{(k)}\right)/4 \\ x_2^{(k+1)} &= -3 - 3x_1^{(k)} + x_3^{(k)} \qquad . \\ x_3^{(k+1)} &= \left(20 - 2x_1^{(k)} - x_2^{(k)}\right)/5 \end{aligned}$$

In the above MATLAB code, we need only change the two lines for xnew(54) and xnew(55) accordingly:

```
xnew(1)=(28+10*xold(2)-xold(3))/4;
xnew(2)=-3-3*xold(1)+xold(3);
```

Running the code, we see that this time we do not get convergence. In fact, a semilog plot will show that quite the opposite is true, the iterates badly diverge. The plot, obtained just as before, is shown in Figure 7.39(b). We will soon show how such sensitivities of iterative methods depend on the form of the coefficient matrix.



Rysunek 7.5: (a) (left) Plots of the 2-norms of the differences of successive iterates in the Jacobi scheme for the linear system of Example 7.26, using the zero vector as the initial iterate. The convergence is exponential, (b) (right) The corresponding errors when the same scheme is applied to the equivalent linear system with the first two equations being permuted. The sequence now badly diverges, showing the sensitivity of iterative methods to the particular form of the coefficient matrix.

The code given in the above example can be easily generalized into a MATLAB M-file for performing the Jacobi iteration on a general system. This task will be delegated to the following exercise for the reader.

EXERCISE FOR THE READER 7.27: (a) Write a function M-file, `[x,k,diff]=jacobi(A,b,x0,tol,kmax`, that performs the Jacobi iteration on the linear system $Ax = b$. The inputs are the coefficient matrix A, the inhomogeneity (column) vector b, the seed (column) vector $x0$ for the iteration process, the tolerance `tol`, which will cause the iteration to stop if the 2-norms of successive iterates become smaller than `tol`, and `kmax`, the maximum number of iterations to perform. The outputs are the final iterate x, the number of iterations performed k, and a vector `diff` that records the 2-norms of successive differences of iterates. If the last three input variables are not specified, default values of `x0` = the zero column vector, `tol` $= 1e-10$, and kmax $= 100$ are used.
(b) Apply the program to recover the data obtained in part (a) of Example 7.26. If we reset the tolerance for accuracy to $1e-10$ in that example, how many iterations would the Jacobi iteration need to converge?

If we compute the values of $x^{(k+1)}$ in order, it seems reasonable to update the values used on the right side of (57) sequentially, as they become available. This modification in the scheme gives the Gauss-Seidel iteration. Notice that the Gauss-Seidel scheme can be implemented so as to roughly cut in half the storage requirements for the iterates of the solution vector x. Although the M-file we present below does not take advantage of such a scheme, the interested reader can easily modify it to do so. Futhermore, as we shall see, the Gauss-Seidel scheme almost always outperforms the Jacobi scheme.

Gauss-Seidel Iteration:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right] (1 \le i \le n). \tag{7.31}$$

We proceed to write an M-file that will apply the Gauss-Seidel scheme to solving the nonsingular linear system (54).

PROGRAM 7.7: A function M-file

`[x,k,diff]=gaussseidel(A,b,x0,tol,kmax)`

that performs the Gauss-Seidel iteration on the linear system $Ax = b$. The inputs are the coefficient matrix A, the inhomogeneity (column) vector b, the seed (column) vector $x0$ for the iteration process, the tolerance `tol`, which will cause the iteration to stop if the 2 norms of successive iterates become smaller than `tol`, and `kmax`, the maximum number of iterations to perform. The outputs are the final iterate x, the number of iterations performed k, and a vector `diff` that records the 2-norms of successive differences of iterates. If the last two input variables are not specified, default values of `tol` $= 1e-10$ and kmax $= 100$ are used.

```
function [x, k, diff] = gaussseidel(A,b,x0,tol,kmax)
%performs the Gauss-Seidel iteration on the linear system Ax = b.
%Inputs: the coefficient matrix 'A', the inhomogeneity (column)
%vector 'b',the seed (column) vector 'x0' for the iteration process,
%the tolerance 'tol' which will cause the iteration to stop if the
```

```
%2-norms of differences of. successive iterates becomes smaller than
%'tol', and 'kmax' that is the maximum number of iterations to
%perform.
%Outputs: the final iterate 'x', the number of iterations performed
%'k' and a vector 'diff' that records the 2-norms of successive
%differences of iterates.
%if either of the lase three input variables are not specified,
%default values of x0-zero column vector, tol-le-10 and kmax=100
%are used.

%assign default input variables, as necessary
if nargin<3, xO=zeros (size (b) ) ; end
if nargin<4, tol=le-10; end
if nargin<5, kmax=100; end

if min(abs(diag(A)))<eps
  error('Coefficient matrix has zero diagonal entries, iteration
                cannet be performed.\r')
end

[n m]=size(A);
x=x0;
k=l; diff=[];

while k<=kmax
  norm=0;
  for i=l:n
    oldxi=x(i); x(i)=b(i);
    for j=[l:i-l i+l:n]
      x(i)=x(i)-A(i,j)*x(j);
    end
    x(i)-x(i)/A(i,i);
    norm=norm+(oldxi-x(i))^2;
  end
  diff(k)=sqrt(norm);
  if diff(k)<tol
      fprintf('Gauss-Seidel iteration has converged in %d
                iterations/r,k)
      return
  end
  k=k+1;
end
 fprintf('Gauss-Seidel iteration failed to converge./r')
```

EXAMPLE 7.27: For the linear system of the last example, apply Gauss-Seidel iteration with initial iterate being the zero vector and the same tolerance as that used for the last example. Find the number of iterations that are now required for convergence and compare the absolute 2-norm error of the final iterate with that for the last example.

SOLUTION: Reentering, if necessary, the data from the last example, create corresponding data for the Gauss-Seidel iteration using the preceding M-file:

```
>>[xGS, kGS, diffGS] = gaussseidel(A,b,zeros(size(b)),le-6);
->Gauss-Seidel iteration has converged in 17 iterations
```

Thus with the same amount of work per iteration, Gauss-Seidel has done the job in only 17 versus 26 iterations for Jacobi.

Looking at the absolute error of the Gauss-Seidel approximation,

```
>>norm(xGS-[l -2 4]' , 2)      ->ans = 1.4177e - 007
```

we see it certainly meets our tolerance goal of le-6 (and, in fact, is smaller than that for the Jacobi iteration).

The Gauss-Seidel scheme can be extended to include a new parameter, ω, that will allow the next iterate $x^{(k+1)}$ to be expressed as a linear combination of the current iterate $x^{(k)}$ and the Gauss-Seidel values given by (58). This gives a family of iteration schemes, collectively known as SOR (successive over relaxation) whose iteration schemes are given by the following formula:

SOR Iteration:

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right] + (1-\omega)x_i^{(k)} \quad (1 \le i \le n) \tag{7.32}$$

The parameter ω, called the **relaxation parameter**, controls the proportion of the Gauss-Seidel update versus the current iterate to use in forming the next iterate. We will soon see that for SOR to converge, we will need the relaxation parameter to satisfy $0 < \omega < 2$. Notice that when $\omega = 1$, SOR reduces to Gauss-Seidel. For certain values of ω, SOR can accelerate the convergence realized in Gauss-Seidel.

With a few changes to the Program 7.7, a corresponding M-file for SOR is easily created. We leave this for the next exercise for the reader.

With a few changes to the Program 7.7, a corresponding M-file for SOR is easily created. We leave this for the next exercise for the reader.

EXERCISE FOR THE READER 7.28: (a) Write a function M-file, $[x,k,$ `diff` $] =$ `sorit` $(A,b,$ `omega`, $x0$, `tol, kmax`), that performs the SOR iteration on the linear system $Ax = b$. The inputs are the coefficient matrix A, the inhomogeneity (column) vector b, the relaxation parameter omega, the seed (column) vector $\times 0$ for the iteration process, the tolerance `tol`, which will cause the iteration to stop if the 2-norms of successive iterates become smaller than `tol`, and `kmax`, the maximum number of iterations to perform. The outputs are the final iterate x, the number of iterations performed k, and a vector `diff` that records the 2-norms of successive differences of iterates. If the last three input variables are not specified, default values of $\times 0 =$ the zero column vector, `tol` = $1e-10$, and kmax $= 100$ are used.
(b) Apply the program to recover the solution obtained in Example 7.27.
(c) If we use $\omega = 0.9$, how many iterations would the SOR iteration need to converge?

EXAMPLE 7.28: Run a set of SOR iterations by letting the relaxation parameter run from 0.05 to 1.95 in increments of 0.5. Use a tolerance for error $= 1e-6$, but set $k$max $= 1000$. Record the number of iterations needed for convergence (if there is convergence) for each value of the ω (up to 1000) and plot this number as a function of ω.

SOLUTION: We can use the M-file sorit of Exercise for the Reader 7.28 in conjunction with a loop to easily obtain the needed data.

```
>> omega=0.05:.05:1.95;
>> length (omega) ->ans = 39
>> for i=l:39
[xSOR, kSOR(i), diffSOR] = sorit(A,b,omega(i),zeros(size(b)),.. .
                        le-6,1000);
end
```

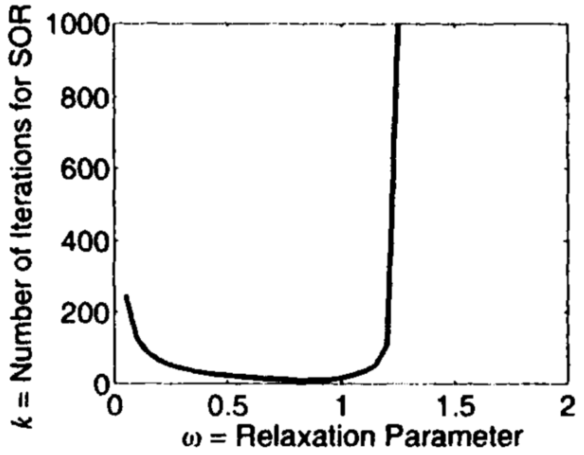
The above loop has overwritten all but the iteration counters, which were recorded as a vector. We use this vector to locate the best value (from among those in our vector omega) to use in SOR.

```
>> [mink ind]=min(kSOR)          ->mink = 9, ind = 18
>> omega (18)         ->ans = 0.9000
```

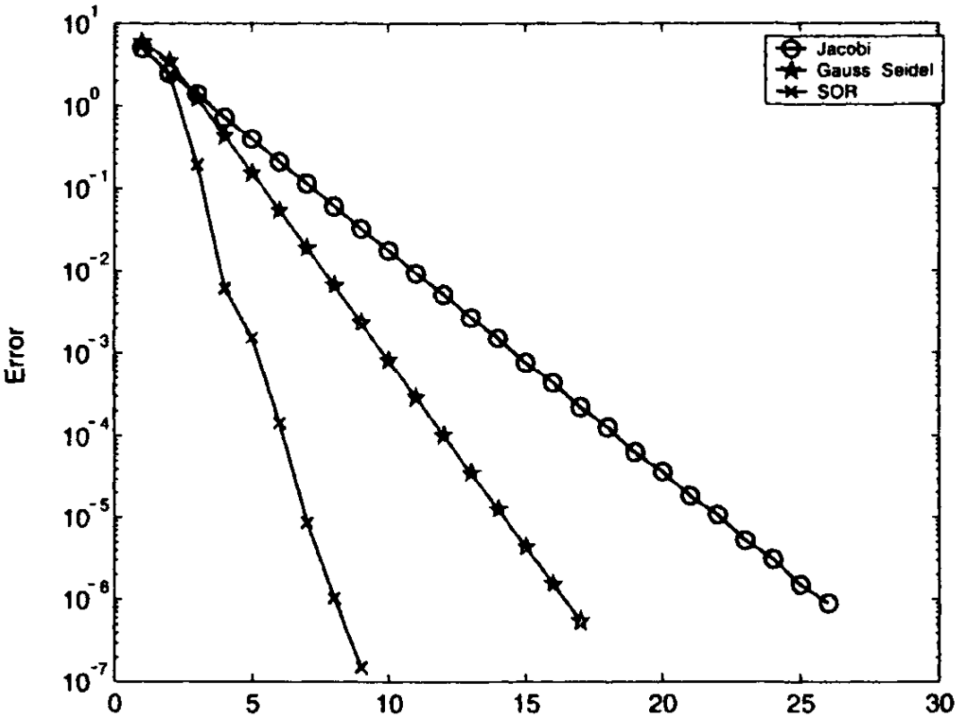
Thus we see that the best value of ω to use (from those we tested) is $\omega = 0.9$, which requires only nine iterations in the SOR scheme, nearly a 50% savings over Gauss-Seidel. The next two commands will produce the desired plot of the required number of iterations needed in SOR versus the value of the parameter ω. The resulting plot is shown in Figure 7.40.

```
>> plot(omega, kSOR),
>> axis([0 2 0 1000])
```

Figure 7.41 gives a plot that compares the convergences of three methods: Jacobi, Gauss-Seidel, and SOR (with our pseudo-optimal value of ω). The next exercise for the reader will ask to reproduce this plot.



Rysunek 7.6: Graph of the number of iterations required for convergence (to a tolerance of $(e-6$) using SOR iteration as a function of the relaxation parameter ω. The k-values are truncated at 1000 . Notice from the graph that the convergence for GaussSeidel $(\omega = 1)$ can be improved.



Rysunek 7.7: Comparison of the errors versus the number of iterations for each of the three iteration methods: Jacobi (o), Gauss-Seidel (*), and SOR (x).

EXERCISE FOR THE READER 7.29: Use MATLAB to reproduce the plot of Figure 7.41. The key in the upper-right corner can be obtained by using the "Data Statistics" tool from the "Tools" menu of the MATLAB graphics window once the three plots are created.

Of course, even though the last example has shown that SOR can converge faster than Gauss-Seidel, the amount of work required to locate a good value of the parameter greatly exceeded the actual savings in solving the linear system of Example 7.26. In the SOR iteration the value of $\omega = 0.9$ was used as the relaxation parameter.

There is some interesting research involved in determining the optimal value of ω to use based on the form of the coefficient matrix. What is needed to prove such results is to get a nice formula for the eigenvalues of the matrix (in general an impossible problem, but for special types of matrices one can get lucky) and then compute the value of ω for which the corresponding maximum absolute value of the eigenvalues is as small as possible. A good survey of the SOR method is given in [You-71]. A sample result will be given a bit later in this section (see Proposition 7.14).

We now present a general way to view iteration schemes in matrix form. From this point of view it will be a simple matter to specialize to the three forms we gave above. More importantly, the matrix notation will allow a much more natural way to perform error analysis and other important theoretical tasks.

To cast iteration schemes into matrix form, we begin by breaking the coefficient matrix A into three pieces:

$$A = D - L - U \tag{7.33}$$

where D is the diagonal part of A, L is (strictly) lower triangular and U is the (strictly) upper triangular. In long form this (60) looks like:

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

This decomposition is actually quite simple. Just take D to be the diagonal matrix with the diagonal entries equal to those of A, and take L/U to be, respectively, the strictly lower/upper triangular matrix whose nonzero entries are the opposites of the corresponding entries of A.

Next, we will examine the following general (matrix form) iteration scheme for solving the system (54) $Ax = b$:

$$Bx^{(k+1)} = (B-A)x^{(k)} + b, \tag{7.34}$$

where B is an invertible matrix that is to be determined. Notice that if B is chosen so that this iteration scheme produces a convergent sequence of iterates: $x^{(k)} \to \tilde{x}$, then the limiting vector $\tilde{x}$ must solve (54). (Proof: Take the limit in (61) as $k \to \infty$ to get $B\tilde{x} = (B-A)\tilde{x} + b = B\tilde{x} - A\tilde{x} + b \Rightarrow A\tilde{x} = b$.) The matrix B should be chosen so that the linear system is easy to solve for $x^{(k+1)}$ (in fact, much easier than our original system $Ax = b$ lest this iterative scheme would be of little value) and so that the convergence is fast.

To get some idea of what sort of matrix B we should be looking for, we perform the following error analysis on the iterative scheme (61). We mathematically solve (61) for $x^{(k+1)}$ by left multiplying by B^{-1} to obtain:

$$x^{(k+1)} = B^{-1}(B-A)x^{(k)} + B^{-1}b = \left(I - B^{-1}A\right)x^{(k)} + B^{-1}b.$$

Let x denote the exact solution of $Ax = b$ and $e^{(k)} = x^{(k)} - x$ denote the error vector of the k th iterate. Note that $-\left(I - B^{-1}A\right)x = -x + B^{-1}b$. Using this in conjunction with the last equation, we can write:

$$\begin{aligned} e^{(k+1)} = x^{(k+1)} - x &= \left(I - B^{-1}A\right)x^{(k)} + B^{-1}b - x \\ &= \left(I - B^{-1}A\right)x^{(k)} - \left(I - B^{-1}A\right)x \\ &= \left(I - B^{-1}A\right)\left(x^{(k)} - x\right) \\ &= \left(I - B^{-1}A\right)e^{(k)} \end{aligned}$$

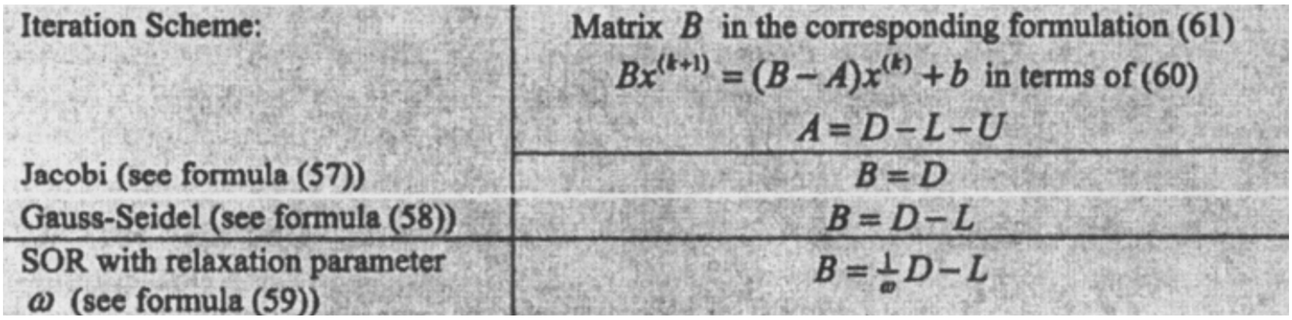
We summarize this important error estimate:

$$e^{(k+1)} = \left(I - B^{-1}A\right)e^{(k)} \tag{7.35}$$

From (62), we see that if the matrix $\left(I - B^{-1}A\right)$ is "small" (in some matrix norm), then the errors will decay as the iterations progress.[9] But this matrix will be small if $B^{-1}A$ is "close" to I, which in turn will happen if B^{-1} is "close" to A^{-1} and this translates to B being close to A.

Table 7.1 summarizes the form of the matrix B for each of our three iteration schemes introduced earlier. We leave it as an exercise to show that with the matrices given in Table 7.1, (61) indeed is equivalent to each of the three iteration schemes presented earlier (Exercise 20).

TABLE 7.1: Summary of matrix formulations of each of the three iteration schemes: Jacobi, Gauss-Seidel, and SOR.



Iteration Scheme:	Matrix B in the corresponding formulation (61) $Bx^{(k+1)} = (B-A)x^{(k)} + b$ in terms of (60) $A = D - L - U$
Jacobi (see formula (57))	$B = D$
Gauss-Seidel (see formula (58))	$B = D - L$
SOR with relaxation parameter ω (see formula (59))	$B = \frac{1}{\omega}D - L$

Thus far we have done only experiments with iterations. Now we turn to some of the theory.

THEOREM 7.10: (Convergence Theorem) Assume that A is a nonsingular (square) matrix, and that B is any nonsingular matrix of the same size as A. The (real and complex) eigenvalues of the matrix $I - B^{-1}A$ all have absolute value less than one if and only if the iteration scheme (61) converges (to the solution of $Ax = b$) for any initial seed vector $x^{(0)}$.

For a proof of this and the subsequent theorems in this section, we refer the interested reader to the references [Atk-89] or to [GoVL-83]. MATLAB's eig function is designed to produce the eigenvalues of a matrix. Since, as we have pointed out (and seen in examples), the Gauss-Seidel iteration usually converges faster than the Jacobi iteration, it is not surprising that there are examples where the former will converge but not the latter (see Exercise 8). It turns out

[9] It is helpful to think of the one-dimensional case, where everything in (62) is a number. If $\left(I - B^{-1}A\right)$ is less than one in absolute value, then we have exponential decay, and furthermore, the decay is faster when absolute values of the matrix are smaller. This idea can be made to carry over to matrix situations. The corresponding needed fact is that all of the eigenvalues (real or complex) of the matrix $\left(I - B^{-1}A\right)$ are less than one in absolute value. In this case, it can be shown that we have exponential decay also for the iterative scheme, regardless of the initial iterate. For complete details, we refer to [Atk-89] or to [GoVL-83].

that there are examples where the Jacobi iteration will converge even though the GaussSeidel iteration will fail to converge.

EXAMPLE 7.29: Using MATLAB's `eig` function for finding eigenvalues of a matrix, apply Theorem 7.10 to check to see if it tells us that the linear system of Example 7.26 will always lead to a convergent iteration method with each of the three schemes: Jacobi, Gauss-Seidel, and SOR with $\omega = 0.9$. Then create a plot of the maximum absolute value of the eigenvalues of $\left(I - B^{-1}A\right)$ for the SOR method as ω ranges from 0.05 to 1.95 in increments of 0.5, and interpret.

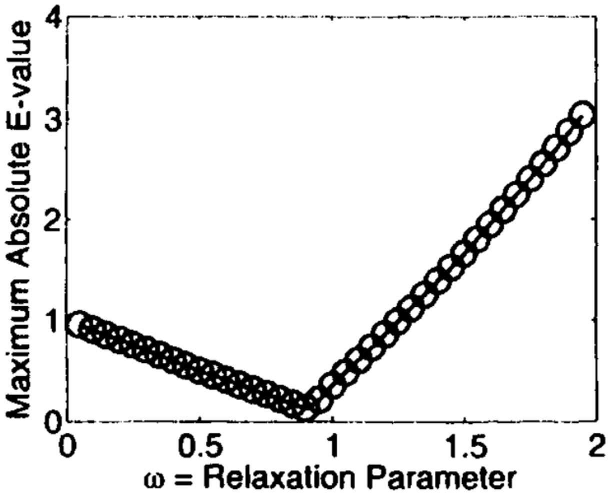
SOLUTION: We enter the relevant matrices into a MATLAB session:

```
>> A=[3 1 -1; 4 -10 1;2 1 5] ; D=diag(diag(A));
>> L=[0 0 0;- 4 0 0;- 2 - 1 0J; U=D-L-A; I = eye(3) ;
>> % Jacobi
>> max (abs (eig (I-inv (D) *A))) ->ans = 0.5374
>> % Gauss Seide l
>> max (abs (eig (I-inv (D-L) *A) ) ) ->ans = 0.3513
>> % SOR onega - 0.9>
>> max (abs (eig (I-inv (D/. 9-L) *A) ) ) -.ans = 0.1301
```

The three computations give the maximum absolute values of the eigenvalues of $\left(I - B^{-1}A\right)$ for each of the three iteration methods. Each maximum is less than one, so the theorem tells us that, whatever initial iteration vector we choose for any of the three schemes, the iterations will always converge to the solution of $Ax = b$. Note that the faster converging methods tend to correspond to smaller maximum absolute values of the eigenvalues of $\left(I - B^{-1}A\right)$; we will see a corroboration of this in the next part of this solution.

The next set of commands produces a plot of the maximum value of the eigenvalues of $\left(I - B^{-1}A\right)$ for the various values of ω, which is shown in Figure 7.42.

```
>> omega=0.05:.05:1.95;
>> for i=l:length(omega)
  rad(i)=max(abs(eig(I-...
  inv(D/omega(i)-L)*A)));
end
>> plot(omega, rad,'0-')
```



Rysunek 7.8: Illustration of the maximum absolute value of the eigenvalues of the matrix $I - B^{-1}A$ of Theorem 7.10 for the SOR method (see Table 7.1) for various values of the relaxation parameter ω. Compare with the corresponding number of iterations needed for convergence (Figure 7.40).

The above theorem is quite universal in that it applies to all situations. The drawback is that it relies on the determination of eigenvalues of a matrix. This eigenvalue problem can be quite a difficult numerical problem, especially for very large matrices (the type that we would want to apply iteration methods to). MATLAB's ei g function may perform unacceptably slowly in such cases and/or may produce inaccurate results. Thus, the theorem has limited practical value. We next give a more practical result that gives a sufficient condition for convergence of both the Jacobi and Gauss-Seidel iterative methods.

Recall that an $n \times n$ matrix A is **strictly diagonally dominant (by rows)** if the absolute value of each diagonal entry is greater than the sum of the absolute values of all other entries in the same row, i.e.,

$$|a_{ii}| > \sum_{\substack{j=i \\ j \neq i}}^{n} \left|a_{ij}\right|, \text{ for } i = 1, 2, \ldots, n.$$

THEOREM 7.11: (Jacobi and Gauss-Seidel Convergence Theorem) Assume that A is a nonsingular (square matrix). If A is strictly diagonally dominant, then the Jacobi and Gauss-Seidel iterations will converge (to the solution of $Ax = b$) for any initial seed vector $x^{(0)}$.

Usually, the more diagonally dominant A is, the faster the rate of convergence will be. Note that the coefficient matrix of Example 7.26 is strictly diagonally dominant, so that Theorem 7.11 tells us that no matter what initial seed vector $x^{(0)}$ we started with, both the Jacobi and Gauss-Seidel iteration schemes would produce a sequence that converges to the solution. Although we knew this already from Theorem 7.10, note that, unlike the eigenvalue condition, the strict diagonal dominance was trivial to verify (by inspection). There are matrices A that are not strictly diagonally dominant but for which both Jacobi and Gauss-Seidel schemes will always converge. For an outline of a proof of the Jacobi part of the above result, see Exercise 23 .

For the SOR method (for other values of ω than 1) there does not seem to be such a simple useful criterion. There is, however, another equivalent condition in the case of a symmetric coefficient matrix A with positive diagonal entries. Recall that an $n \times n$ matrix A is symmetric if $A = A'$; also A is positive definite provided that $x'Ax > 0$ for any nonzero $n \times 1$ vector x.

THEOREM 7.12: (SOR Convergence Theorem) Assume that A is a nonsingular (square matrix). Assume that A is symmetric and has positive diagonal entries. For any choice of relaxation parameter $0 < \omega < 2$, the SOR iteration will converge (to the solution of $Ax = b$) for any initial seed vector $x^{(0)}$, if and only if A is positive definite.

Matrices that satisfy the hypotheses of Theorems 7.11 and 7.12 are actually quite common in numerical solutions of differential equations. We give a typical example of such a matrix shortly. For reference, we collect in the following theorem two equivalent formulations for a symmetric matrix to be positive definite, along with some necessary conditions for a matrix to be positive definite. Proofs can be found in [Str-88], p. 331 (for the equivalences) and [BuFa-01], p. 401 (for the necessary conditions).

THEOREM 7.13: (Positive Definite Matrices) Suppose that A is a symmetric $\boldsymbol{n} \times \boldsymbol{n}$ matrix.

(a) The following two conditions are each equivalent to A being positive definite:

(i) All eigenvalues of A are positive, or

(ii) The determinants of all upper-left submatrices of A have positive determinants.

(b) If A is positive definite, then each of the following conditions must hold:

(i) A is nonsingular.

(ii) $a_{ii} > 0$ for $i = 1, 2, \ldots, n$.

(iii) $a_{ii}a_{ij} > a_{ij}^2$ whenever $i \neq j$.

We will be working next with a certain class of sparse matrices, which is typical of those that arise in finite difference methods for solving partial differential equations. We study such problems and concepts in detail in [Sta-05]; here we only very briefly outline the connection.

The matrix we will analyze arises in solving the so-called Poisson boundary value problem on the two-dimensional unit square $\{(x,y) : 0 \leq x,y \leq 1\}$, which asks for the determination of a function $u = u(x,y)$ that satisfies the following partial differential equation and boundary conditions:

$$\begin{cases} -\Delta u = f(x,y), \text{ inside the square: } 0 < x,y < 1 \\ u(x,y) = 0, \text{ on the boundary: } x = 0,1 \text{ or } y = 0,1 \end{cases}$$

Here Δu denotes the Laplace differential operator $\Delta u = u_{xx} + u_{yy}$. The finite difference method "discretizes" the problem into a linear system. If we use the same number N of grid points both on the x - and the y-axis, the linear system $Ax = b$ that arises will have the $N^2 \times N^2$ coefficient matrix shown in (63).

In our notation, the partition lines break the $N^2 \times N^2$ matrix up into smaller $N \times N$ block matrices $\left(N^2\right.$ of them). The only entries indicated are the nonzero entries that occur on the five diagonals shown. Because of their importance in applications, matrices such as the one in (63) have been extensively studied in the context of iterative methods. For example, the following result contains some very practical and interesting results about this matrix.

$$\left[\begin{array}{c|c|c|c}
\begin{matrix}4&-1&&\\-1&4&\ddots&\\&\ddots&\ddots&-1\\&&-1&4\end{matrix} & \begin{matrix}-1&&&\\&-1&&\\&&\ddots&\\&&&-1\end{matrix} & & \\ \hline
\begin{matrix}1&&&\\&-1&&\\&&\ddots&\\&&&-1\end{matrix} & \begin{matrix}4&-1&&\\-1&4&\ddots&\\&\ddots&\ddots&-1\\&&-1&4\end{matrix} & \begin{matrix}\ddots&&&\\&\ddots&&\\&&\ddots&\\&&&\ddots\end{matrix} & \\ \hline
 & \begin{matrix}\ddots&&&\\&\ddots&&\\&&\ddots&\\&&&\ddots\end{matrix} & \begin{matrix}\ddots&&&\\&\ddots&&\\&&\ddots&\\&&&\ddots\end{matrix} & \begin{matrix}-1&&&\\&-1&&\\&&\ddots&\\&&&-1\end{matrix} \\ \hline
 & & \begin{matrix}-1&&&\\&-1&&\\&&\ddots&\\&&&-1\end{matrix} & \begin{matrix}4&-1&&\\-1&4&\ddots&\\&\ddots&\ddots&-1\\&&-1&4\end{matrix}
\end{array}\right] \tag{7.36}$$

PROPOSITION 7.14: Let A be the $N^2 \times N^2$ matrix (63).
(a) A is positive definite (so SOR will converge by Theorem 7.12) and the optimal relaxation parameter ω for an SOR iteration scheme for a linear system $Ax = b$ is as follows:

$$\omega = \frac{2}{1 + \sin\left(\frac{\pi}{N+1}\right)}$$

(b) With this optimal relaxation parameter, the SOR iteration scheme works on order of N times as fast as either the Jacobi or Gauss-Seidel iteration schemes. More precisely, the following quantities $R_f, R_{Gs}, R_{\text{soe}}$ indicate the approximate number of iterations that each of these three schemes would need, respectively, to reduce the error by a factor of $1/10$:

$$R_J \approx 0.467(N+1)^2, R_{GS} = \frac{1}{2}R_J \approx 0.234(N+1)^2, \text{ and } R_{\text{SOR}} \approx 0.367(N+1)$$

In our next example we compare the different methods by solving a very large fictitious linear system $Ax = b$ involving the matrix (63). This will allow us to make exact error comparisons with the true solution.

A proof of the above proposition, as well as other related results, can be found in Section 8.4 of [StBu-93]. Note that since A is not (quite) strictly diagonally dominant, the Jacobi/Gauss-Seidel convergence theorem (Theorem 7.11) does not apply. It turns out that the Jacobi iteration scheme indeed converges, along with SOR (and, in particular, the Gauss-Seidel method); see Section 8.4 of [StBu-93].

Consider the matrix A shown in (63) with $N = 50$. The matrix A has size 2500×2500 so it has 6.25 million entries. But of these only about $5N^2 = 12,500$ are nonzero. This is about 0.2% of the entries, so A is quite sparse. EXERCISE FOR THE READER 7.30: Consider the problem of multiplying the matrix A in (63) (using $N = 50$) by the vector $x = \left[\begin{array}{cccccccccc} 1 & 2 & 1 & 2 & 1 & 2 & \cdots & 1 & 2 \end{array}\right]'$.
(a) Compute (by hand) the vector $b \equiv Ax$ by noticing the patterns present in the multiplication.
(b) Get MATLAB to compute $b = Ax$ by first creating and storing the matrices A and x and performing a usual matrix multiplication. Use `tic/toc` to time the parts of this computation.
(c) Store only the five nonzero diagonals of (as column vectors): `d, a1, aN b1, bN(` d stands for main diagonal, a for above-main diagonal, b for below-main diagonal). Recompute b by suitably manipulating these 5 vectors in conjunction with x Use `tic/toc` to time the computation and compare with that in part (b).
(d) Compare all three answers. What happens to the three methods if we bump N up to 100 ?

Shortly we will give a general development on the approach hinted at in part (c) of the above Exercise for the Reader 7.30.

As long as the coefficient matrix A is not too large to be stored in a session, MATLAB's left divide is quite an intelligent linear system solver. It has special more advanced algorithms to deal with positive definite coefficient matrices, as well as with other special types of matrices. It can numerically solve systems about as large as can be stored; but the accuracy of the numerical solutions obtained depends on the condition number of the matrix, as explained earlier in this chapter.[10] The next example shows that even with all that we know about the optimal relaxation parameter for the special matrix (63), MATLAB's powerful left divide will still work more efficiently for a very large linear system than our SOR program. After the example we will remedy the situation by modifying the SOR program to make it more efficient for such banded sparse matrices.

EXAMPLE 7.30: In this example we do some comparisons in some trial runs of solving a linear system $Ax = b$ where A is the matrix of (63) with $N = 50$, and the vectors x and b are as in the preceding Exercise for the Reader 7.30. Having the exact solution will allow us to look at the exact errors resulting from any of the methods.
(a) Solve the linear system by using MATLAB's left divide (Gaussian elimination). Record the computation time and error of the computed solution.
(b) Solve the system using the Gauss-Seidel program gaussseidel (Program 7.7) with the default number of iterations and initial vector. Record the computation time and error. Repeat using 200 iterations.
(c) Solve again using the SOR program sorit (from Exercise for the Reader 7.28) with the optimal relaxation parameter ω given in Proposition 7.14. Record the computation time and error. Repeat using 200 iterations.
(d) Reconcile the data of parts (b) and (c) with the results of part (c) of Proposition 7.14.

SOLUTION: We first create and store the relevant matrices and vectors:

```
>> x=ones(2500,1); x(2:2:2500,1)=2;
>> A=4*eye(2500);
>> vl=-l*ones (49,1) ; vl=[vl;0]; %seed vector for sub/super diagonals
>> secdiag=vl;
>> for i = 1:49
  if i<49
    secdiag=[secdiag;vl] ;
  else
    secdiag=[secdiag;vl(1:49)] ;
  end
end
>> A=A+diag(secdiag,1)+diag(secdiag,-1)-diag(ones(2450,1),50)...
-diag(ones(2450,l),-50);
>> b=A*x;
```

Part (a):

```
>> tic , xMATLAB=A\b; toc ->elapsed_time = 9.2180
>> max(xMATLAB-x) ->ans = 6.2172e - 015
```

Part (b):

```
>> tic , [xGS, k, diffJ=gaussseidel(A,b); toc
->Gauss-Seidel iteration failed to converge.->elapsed time = 181.6090
>> max(abs(xGS-x)) ->ans = 1.4353

>> tic , [xGS2, k, diff] = gaussseidel(A,b,zeros(size(b)), le-10,200);
toe
->Gauss-Seidel iteration failed to converge.->elapsed_time = 374.5780
>> max (abs (xGS2-x) )->ans = 1.1027
```

Part (c):

```
>> tic , [xSORr k, diff]=sorit(A,b , 2/(1+sin(pi/51))) ; toc
->SOR iteration failed to converge. ->elapsed_time = 186.7340
>> max(abs(xSOR-x)) ->ans = 0.0031

>> tic , [xSOR2, k, diff]=sorit(A,b , 2/(1+sin(pi/51)),.. .
         zeros(size(b)) , le-10,200) ; toc

->SOR iteration failed to converge, ->elapsed_time = 375.2650
>> max(abs(xSOR2-x)) ->ans = 1.1885e - 008
```

Part (d): The above data shows that our iteration programs pale in performance when compared with MATLAB's left divide (both in time and accuracy). The attentive reader will realize that both iteration programs do not take advantage of

[10] Depending on the power of the computer on which you are running MATLAB's as well as the other processes being run, computation times and storage capacities can vary. At the time of writing this section on the author's 1.6MHz, 256MB RAM Pentium IV PC, some typical limits, for random (dense) matrices, are as follows: The basic Gaussian elimination (Program 7.6) starts taking too long (toward an hour) when the size of the coefficient matrix gets larger than 600×600; for it to take less. than about one minute the size should be less than about 250×250. Before memory runs out, on the other hand, matrices of sizes up to about 6000×6000 can be stored, and MATLAB's left divide can usually (numerically) solve them in a reasonable amount of time (provided that the condition number is moderate). To avoid redundant storage problems, MATLAB does have capabilities of storing sparse matrices. Such functionality introduced at the end of this section. Taking advantage of the structure of sparse banded matrices (which are the most important ones in numerical differential equations) will enable us to solve many such linear systems that are quite large, say up to about $50,000 \times 50,000$. Such large systems often come up naturally in numerical differential equations.

the sparseness of the matrix (63). They basically run through all of the entries in this large matrix for each iteration. One other thing that should be pointed out is that the above comparisons are somewhat unfair because MATAB's left divide is a compiled code (built-into the system), whereas the other programs were interpreted codes (created as external programs-M-files). After this example, we will show a way to make these programs perform more efficiently by taking advantage of the special banded structure of such matrices. The resulting modified programs will then perform more efficiently than MATLAB's left divide, at least for the linear system of this example.

Using $N = 50$ in part (b) of Proposition 7.14, we see that in order to cut errors by a factor of 10 with the Gauss-Seidel method, we need approximately $0.234 \cdot 51^2 \approx 609$ additional iterations, but for the SOR with optimal relaxation parameter the corresponding number is only $.367.51 \approx 18.7$. This corroborates well with the experimental data in parts (b) and (c) above. For Gauss-Seidel, we first used 100 iterations and then used 200 . The theory tells us we would need over 600 more iterations to reduce the error by 90%. Using 100 more iterations resulted in a reduction of error by about 23%. On the other hand, with SOR, the 100 additional iterations gave us approximately $100/18.7 \approx 5.3$ reductions in the errors each by factors of $1/10$, which corresponds nicely to the (exact) error shrinking from about $3\mathrm{e}-3$ to $1\mathrm{e}-8$ (literally, about 5.3 decimal places!).

In order to take advantage of sparsely banded matrices in our iteration algorithms, we next record here some elementary observations regarding multiplications of such matrices by vectors. MATLAB is enabled with features to easily manipulate and store such matrices. We will now explore some of the underlying concepts and show how exploiting the special structure of some sparse matrices can greatly expand the sizes of linear systems that can be effectively solve. MATLAB has its own capability for storing and manipulating general sparse matrices; at the end of the section we will discuss how this works.

The following (nonstandard) mathematical notations will be convenient for the present purpose: For two vectors v, w that are both either row-or columnvectors, we let $v \otimes w$ denote their juxtaposition. For the pointwise product of two vectors of the same size, we use the notation $v \odot w = [v_i w_i]$. So, for example, if $v = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}$, and $w = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}$ are both 3×1 row vectors, then $v \otimes w$ is the 6×1 row vector $\begin{bmatrix} v_1 & v_2 & v_3 & w_1 & w_2 & w_3 \end{bmatrix}$ and $v \odot w$ is the 3×1 row vector $[v_1 w_1 v_2 w_2 v_3 w_3]$. We also use the notation 0_n to denote the zero vector with n components. We will be using this last notation only in the context of juxtapositions so that whether 0_n is meant to be a row vector or a column vector will be clear from the context.[11]

LEMMA 7.15: (a) Let S be an $n \times n$ matrix whose k th superdiagonal ($k \geq 1$) is made of the entries (in order) of the $(n-k) \times 1$ vector v, i.e.,

$$S = \begin{bmatrix} 0 & 0 & \cdots & 0 & v_1 & 0 & & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & v_2 & 0 & & \vdots \\ \vdots & \vdots & 0 & \ddots & \cdots & 0 & v_3 & \ddots & 0 \\ & & \vdots & 0 & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \ddots & & & 0 & v_{n-k} \\ \vdots & & & & & \ddots & & & 0 \\ 0 & \vdots & \vdots & & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 & 0 \end{bmatrix},$$

where $v = \begin{bmatrix} v_1 & v_2 & \cdots & v_{n-k} \end{bmatrix}$. (In MATLAB's notation, the matrix S could be entered as diag (v,k), once the vector v has been stored.) Let $x = \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n \end{bmatrix}'$ be any $n \times 1$ column vector. The following relation then holds:

$$Sx = (v \otimes 0_k) \odot \left(\begin{bmatrix} x_{k+1} & x_{k+2} & \cdots & x_n \end{bmatrix} \otimes 0_k \right) \tag{7.37}$$

(b) Analogously, if S is an $n \times n$ matrix whose k th subdiagonal ($k \geq 1$) is made of the entries (in order) of the $(n-k) \times 1$

[11] We point out that these notations are not standard. The symbol $\otimes$ is usually reserved for the socalled tensor product.

vector v, i.e.,

$$S=\begin{bmatrix} 0 & 0 & \cdots & & & & \cdots & 0 & 0 \\ \vdots & 0 & 0 & \cdots & & & & & 0 \\ 0 & \vdots & 0 & 0 & \cdots & & & & \vdots \\ v_1 & 0 & \vdots & 0 & 0 & \cdots & & & \\ 0 & v_2 & 0 & & \ddots & \ddots & & & \\ \vdots & 0 & v_3 & 0 & & \ddots & & & \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \vdots & \vdots & 0 & v_{n-k-1} & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & v_{n-k} & 0 & \cdots & 0 \end{bmatrix},$$

where $v=\begin{bmatrix} v_1 & v_2 & \cdots & v_{n-k} \end{bmatrix}$ and $x=\begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n \end{bmatrix}'$ is any $n\times 1$ column vector, then

$$Sx=(0_k\otimes v)\odot\left(0_k\otimes\begin{bmatrix} x_1 & x_2 & \cdots & x_k \end{bmatrix}\right) \tag{7.38}$$

The proof of the lemma is left as Exercise 21 . The lemma can be easily applied to greatly streamline all of our iteration programs for sparse banded matrices. The next exercise for the reader will ask the reader to perform this task for the SOR iteration scheme.

EXERCISE FOR THE READER 7.31: (a) Write a function M-file with the following syntax:

```
[x,k,diff] = sorsparsediag(diags, inds, b, omega, x0, tol, kmax)
```

that will perform the SOR iteration to solve a nonsingular linear system $Ax=b$ in which the coefficient matrix A has entries only on a sparse set of diagonals. The first two input variables are `diags`, an $n\times j$ matrix where each column consists of the entries of A on one of its diagonals (with extra entries at the end of the column being zeros), and `inds`, a $1\times j$ vector of the corresponding set of indices for the diagonals (index zero corresponds to the main diagonal and should be first). The remaining input and output variables will be exactly as in the M-file `sorit` of Exercise for the Reader 7.28. The program should function just like the `sorit` M-file, with the only exceptions being that the stopping criterion for the norms of the difference of successive iterates should now be determined by the infinity norm[12] and the default number of iterations is now 1000 . The algorithm should, of course, be based on formula (59) for the SOR iteration, but the sum need only be computed over the index set (`inds`) of the nonzero diagonals. To this end, the above lemma should be used in creating this M-file so that it will avoid unnecessary computations (with zero multiplications) as well as storage problems with large matrices. (b) Apply the program to redo part (c) of Example 7.30.

EXAMPLE 7.31: (a) Invoke the M-file `sorsparsediag` of the preceding exercise for the reader to obtain SOR numerical solutions of the linear system of Example 7.30 with error goal $5e-15$ (roughly.MATLAB's machine epsilon) and compare the necessary runtime with that of MATLAB's left divide, which was recorded in that example. (b) Next use the program to solve the linear system $Ax=b$ with A as in (63) with $N=300$ and $b=\begin{bmatrix} 1 & 2 & 1 & 2 & \ldots & 1 & 2 \end{bmatrix}'$. Use the default tolerance $1e-10$, then, looking at the last norm difference (estimate for the actual error) use Proposition 7.14 to help to see how much to increase the maximum number of iterations to ensure convergence of the method. Record the runtimes. The size of A is $90,000\times 90,000$ and so it has over 8 billion entries. Storage of such a matrix would require a supercomputer.

SOLUTION: Part (a): We first need to store the appropriate data for the matrix A. Assuming the variables created in the last example are still in our workspace, this can be accomplished as follows:

```
>> diags=zeros(2500,5);
>> diags (:,1)=4*ones(2500,1);
>> diags(1:2499,2:3)=[secdiag secdiag];
>> diags(1:24 50,4:5)=-ones(2450,2);
>> inds=[0 1 -1 50 -50];
```

```
>> tic, [xSOR, k, diff]=sorsparsediag(diags, inds,b,...
        2/(1+sin(pi/51)) , zeros(size(b)), 5e-15); toc
->SOR iteration has converged in 308 iterations   ->elapsed_time = 1.3600
>> max(abs(xSOR-x))   ->ans = 2.3537e - 014
```

Our answer is quite close to machine precision (there were roundoff errors) and the answer obtained by MATLAB's left divide. The runtime of the modified SOR program is now, however, significantly smaller than that of the MATLAB solver. We will see later, however, that when we store the matrix A as a sparse matrix (in MATLAB's syntax), the left divide

[12] The infinity norm of a vector x is simply, in MATLAB's notation, max(abs (x)). This is a rather superficial change in the M-file, merely to allow easier performance comparisons with MATLAB's left divide system solver.

method will work at comparable speed to our modified SOR program.

Part (b): We first create the input data by suitably modifying the code in part (a):

```
>> b=ones(90000, 1); b(2:2:90000,1)=2;
>> vl=-l*ones(299, 1); vl=[vl;0]; %seed vector for sub/super diagonals
>> secdiag=vl;
>> for i=l:299
  if i<299
    secdiag=[secdiag; vl];
  else
    secdiag=[secdiag;vl(1:299)];
  end
end
>> diags=zeros(90000,5);
>> diags(:,l)=4*ones(90000,1) ;
>> diags(1:89999,2:3)=[secdiag secdiag];
>> diags(l:89700,4:5)= [-ones(89700,1) -ones(89700,1)];
>> inds=[0 1 -1 300 -300];
>> tic, [xSORbig, k, diff] =sorsparsediag (diags, inds,b, ...
    2/(l+sin(pi/301))) ; toc
->SOR iteration failed to converge, ->elapsed_time = 167.0320
>> diff(k-l ) ->ans = 1.3845e-005
```

We need to reduce the current error by a factor of $1\mathrm{e}-5$. By Proposition 7.14, this means that we should bump up the number of iterations by a bit more than $5R_{\mathrm{SOR}} \approx 5 \cdot 0.367 \cdot 301 \approx 552$. Resetting the default number of iterations to be 1750 (from 1000) should be sufficient. Here is what transpires:

```
>> tic, [xSORbig, k, diff]=sorsparsediag(diags, inds,b,...
        2/(l+sin(pi/301)), zeros(size(b)), le-10, 1750); toc
->SOR iteration has converged in 1620 iterations ->elapsed_time =
   290.1710
>> diff(k -1) ->ans = 1.0550e - 010
```

We have thus solved this extremely large linear system, and it only took about three minutes!

As promised, we now give a brief synopsis of some of MATLAB's built-in, state-of-the-art iterative solvers for linear systems $Ax = b$. The methods are based on more advanced concepts that we briefly indicated and referenced earlier in the section. Mathematical explanations of how these methods work would lie outside the focus of this book. We do, however, outline the basic concept of **preconditioning**. As seen early in this section, iterative methods are very sensitive to the particular form of the coefficient matrix (we gave an example where simply switching two rows of A resulted in the iterative method diverging when it originally converged). An invertible matrix (usually positive definite) M is used to precondition our linear system when we apply the iterative method instead to the equivalent system: $M^{-1}Ax = M^{-1}b$. Often, preconditioning a system can make it more suitable for iterative methods. For details on the practice and theory of preconditioning, we refer to Part II of [Gre-97], which includes, in particular, preconditioning techniques appropriate for matrices that arise in solving numerical PDEs. See also Part IV of [TrBa-97].

Here is a detailed description of MATLAB's function for the so-called p**reconditioned conjugate gradient method**, which assumes that the coefficient matrix A is symmetric positive definite.

`x=pcg(A,b,tol,kmax,Ml,M2 , xO)` →	Performs the preconditioned gradient method to solve the linear system $Ax = b$, where the $N \times N$ coefficient matrix A must be symmetric positive definite and the preconditioner $M = M1 * M2$. Only the first two input variables are required; any tail sequence of input variables can be omitted. The default values of the optional variables are as follows: `tol`=le-6, $\mathrm{kmax} = \min(N, 20), M1 = M2 = I$ (identity matrix), and $x0 =$ the zero vector. Setting any of these optional input variables equal to [ ] gives them their default values.

`[x,flag]` `=pcg(A,b,tol,kmax,Ml,M2,xO)` →	Works as above but returns additional output `flag`: `flag = 0` means `pcg` converged to the desired tolerance `tol` within `kmax` iterations; `flag = 1` means `pcg` iterated `kmax` times but did not converge. For a detailed explanation of other flag values, type `help peg`.

We point out that with the default values $M1 = M2 = I$, there is no conditioning and the method is called the conjugate gradient method.

Another powerful and and more versatile method is the **generalized minimum residual method (GMRES)**. This method works well for general (nonsymmetric) linear systems. MATLAB's syntax for this function is similar to the above, but there is one additional (optional) input variable:

`x=gmres(A,b,restart,tol,` `kmax,Ml,M2,xO)` →	Performs the generalized minimum residual method to solve the linear system $Ax = b$, with preconditioner `M = M1*M2`. Only the first two input variables are required; any tail sequence of input variables can be omitted. The default values of the optional variables are as follows: restart = [ ] (unrestarted method) tol = le-6, kmax = min(N,20), `M1 = M2 = I` (identity matrix), and JCO = the zero vector. Setting any of these optional input variables equal to [ ], gives them their default values. An optional second output variable `flag` will function in a similar fashion as with `pcg`.

EXAMPLE 7.32: (a) Use pcg to resolve the linear system of Example 7.30, with the default settings and flag. Repeat by resetting the tolerance at $1e-15$ and the maximum number of iterations to be 100 and then 200 . Record the runtimes and compare these and the errors to the results for the SOR program of the previous example.
(b) Repeat part (a) with gmres.

SOLUTION: Assume that the matrices and vectors remain in our workspace (or recreate them now if necessary). We need only follow the above syntax instructions for `pcg`:

Part (a):

```
>> tic , [xpeg, flagpcg]=pcg(A,b) ; toc   ->elapsed_time = 3.2810
>> max(abs(xpeg-x))   ->ans = 1.5007
>> flagpcg    ->flagpcg = 1
```

The flag being = 1 means after 20 iterations, pcg did not converge within tolerance (le-5), a fact that we knew from the exact error estimate.

```
>> tic , [xpeg, flagpcg]=pcg (A,b,5e-15, 100); toc    ->elapsed_time =
   15.8900
>> max(abs(xpcg-x))   ->ans = 4.5816e - 006
>> flagpcg    ->flagpcg = 1

>> tic, [xpeg, flagpcg)=pcg(A,b,5e-15, 200); toc    ->elapsedjime =
   29.7970
>> flagpcg    ->flagpcg = 0
>> max (abs(xpeg-x))    ->ans = 3.2419 - 014
```

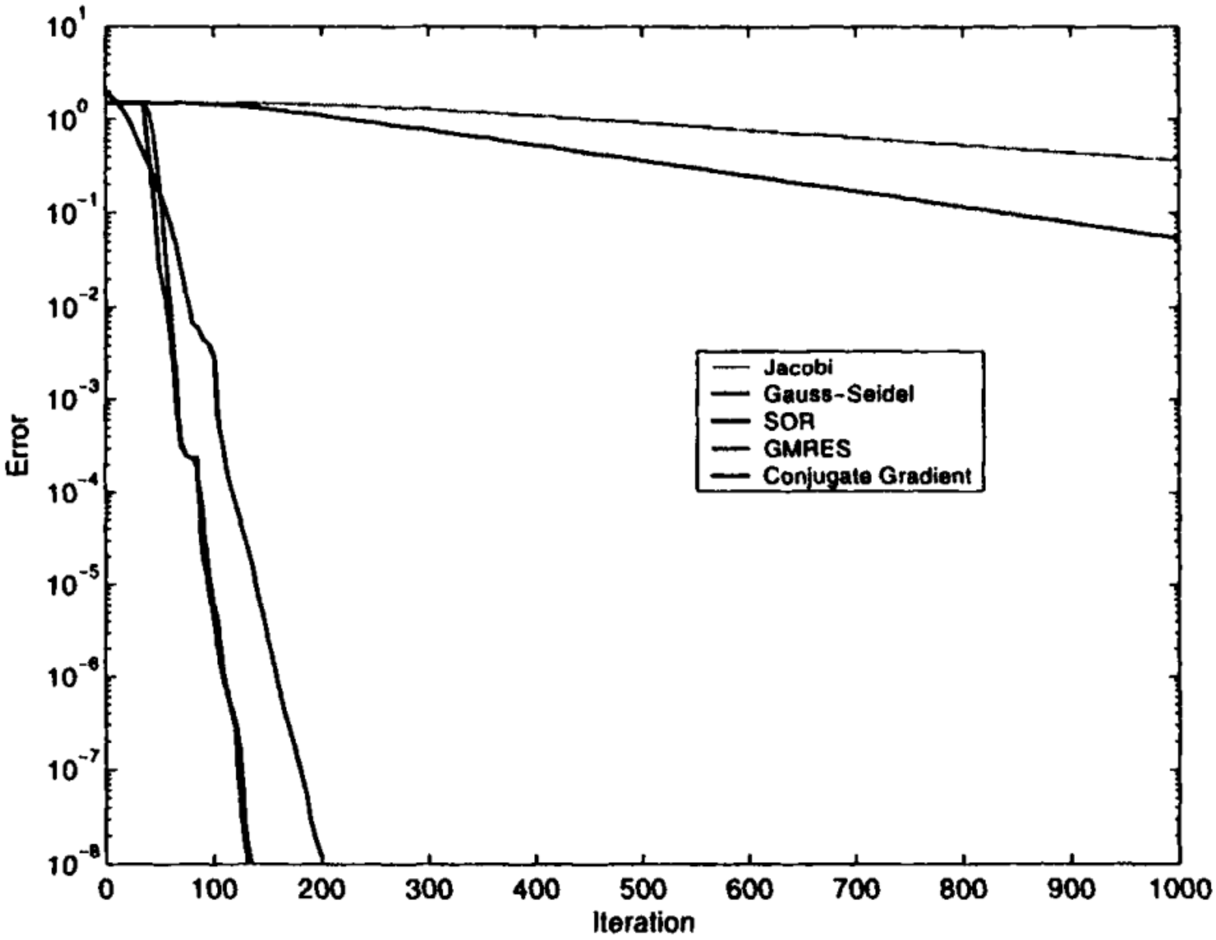
The flag being = 0 in this last run shows we have convergence. The max norm is a different one from the 2 -norm used in the M-file; hence the slight discrepancy. Notice the unconditioned conjugate gradient method converged in fewer iterations than did the optimal SOR method, and in much less time than the original sorit program. The more efficient `sorsparsediag` program, however, got the solution in by far the shortest amount of real time. Later, we will get a more equitable comparison when we store A as a sparse matrix.
Part (b):

```
>> tic , (xgmres, flaggmres]=gmres(A,b, [],[] , 200); toc
>> tic , [xgmres, f lagcfmres] =gmres (A,b) ; toc   ->elapsed_time =
   2.3280
>> max(abs(xgmres-x))   ->ans = 1.5002
>>flaggmres   ->flaggmres = 1

>> tic, [xgmres, flaggmres]=gmres(A,b, [],5e-15 , 100); toe
->elapsed_time = 37.1250
>> max (abs (xgmres-x))   ->ans = 9.2037 - 013
```

The results for GMRES compare well with those for the preconditioned conjugate gradient method. The former method converges a bit more slowly in this situation. We remind the reader that the conjugate gradient method is ideally suited for positive definite matrices, like the one we are dealing with.

Figure 7.43 gives a nice graphical comparison of the relative speeds of convergence of the five iteration methods that have been introduced in this section. An exercise will ask the reader to reconstruct this MATLAB graphic.



Rysunek 7.9: Comparison of the convergence speed of the various iteration methods in the solution of the linear system $Ax = b$ of Example 7.30 where the matrix A is the matrix (63) of size 2500×2500. In the SOR method the optimal relaxation parameter of Proposition 7.14 was used. Since we did not invoke any conditioning, the preconditioned conjugate gradient method is simply referred to as the conjugate gradient method. The errors were measured in the infinity norm.

In the construction of the above data, the program `sorsparsediag` was used to get the SOR data and, despite the larger number of iterations than GMRES and the conjugate gradient method, the SOR data was computed more quickly. The `sorsparsediag` program is easily modified to construct similar programs for the Jacobi and Gauss-Seidel iterations (of course Gauss-Seidel could simply be done by setting $\omega = 0$ in the SOR program), and such programs were used to get the data for these iterations. Note that the GMRES and conjugate gradient methods take several iterations before errors start to decrease, unlike the SOR method, but they soon catch up. Note also the comparable efficiencies between the GMRES and conjugate gradient methods.

We close this chapter with a brief discussion of how to store and manipulate sparse matrices directly with MATLAB. Sparse matrices in MATLAB can be stored using three vectors: one for the nonzero entries, the other two for the corresponding row and column indices. Since in many applications sparse matrices will be banded, we will explain only a few commands useful for the creation and storage of such sparse matrices. Enter `help sparse` for more detailed information. To this end, suppose that we have an $n \times n$ banded matrix A and we wish to store it as a sparse matrix. Let the indices corresponding to the nonzero bands (diagonals) of A have numbers stored in a vector d (so the size of d is the number of bands, 0 corresponds to the main diagonal, positive numbers mean above the main diagonal, negative numbers mean below). Letting p denote the length of the vector d we form a corresponding $n \times p$ matrix, `Diags`, containing as its columns the corresponding bands (diagonals) of A. When columns are longer than the bands they replace (this will be the case except for main diagonal), super (above) diagonals should be put on the lower portion of `Diags` and sub (below) diagonals on the upper portion of `Diags`, with remaining entries on the column being set to zero.

`S=spdiags(Diags,d,n,n)` →	This command creates a sparse matrix data type S, of size $n \times n$ provided that `d` is a vector of diagonal indices (say there are p), and `Diags` is the corresponding $n \times p$ matrix whose columns are the diagonals of the matrix (arranged as explained above).
`full(S)` →	Converts a sparse matrix data type back to its usual "full" form. This command is rarely used in dealing with sparse matrices as it defeats their purpose.

A simple example will help shed some light on how MATLAB deals with sparse data types. Consider the matrix $A =$

$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 4 & 0 & 2 & 0 \\ 0 & 5 & 0 & 3 \\ 0 & 0 & 6 & 0 \end{bmatrix}$. The following MATLAB commands will store A as a sparse matrix:

```
>> d=[-l 1] ; Diags=[4 5 6 0; 0 1 2 3]';
>> S=spdiags(Diags,d , 4, 4)
->=(2,1)    4    (2,3)    2
   (1.2)    1    (4,3)    6
   (3,2)    5    (3,4)    3
```

The display shows the storage scheme. Let's compare with the usual form:

```
>> full(S)
-> ans =  0 1 0 0
          4 0 2 0
          0 5 0 3
          0 0 6 0
```

The key advantage of sparse matrix storage in MATLAB is that if A is stored as a sparse matrix S, then to solve a linear system $Ax = b$, MATLAB's left divide operation textttx=S \ b takes advantage of sparsity and can greatly increase the size of (sparse) problems we can solve. In fact, at most all of MATLAB's matrix functions are able to operate on sparse matrix data types. This includes MATLAB's iterative solvers textttpcg, gmres, etc. We invite the interested reader to perform some experiments and discover the additional speed and capacity that taking advantage of sparsity can afford. We end with an example of a rematch of MATLAB's left divide against our `sorsparsediag` program, this time allowing the left divide method to accept a sparse matrix. The results will be quite illuminating.

EXAMPLE 7.33: We examine the large $(10{,}000 \times 10{,}000)$ system $Ax = b$, where A is given by (63) with $N = 100$, and $x = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \end{pmatrix}'$. By examining the matrix multiplication we see that

$$b = Ax = \begin{pmatrix} 2 & 1 & 1 & \cdots & 2 | 1 & 0 & 0 & \cdots & 0 & 1 & \cdots & 1 & 0 & 0 & \cdots & 0 & 1 | & 2 & 1 & 1 & \cdots & 1 & 2 \end{pmatrix}'.$$

We thus have a linear system with which we can easily obtain the exact error of any approximate solution.
(a) Solve this system using MATLAB's left divide and by storing A as a sparse matrix. Use `tic/toc` to track the computation time (on your computer); compute the error as measured by the infinity norm (i.e., as the maximum difference of any component of the computed solution with the exact solution).
(b) Solve the system using the `sorsparsediag` M-file of Exercise for the Reader 7.31. Compute the time and errors as in part (a) and compare.

SOLUTION: Part (a): We begin by entering the parameters (for (63)), creating the needed inputs for `spdiags`, and then using the latter to store A as a sparse matrix.

```
>> N=200; n=N^2; d=[-N -1 0 1 N];, dia=4*ones(1,n);
>> seed1=-1*ones(1,N-1); v1=[seed1 0];
for i=1:N-1, if i<N-1, v1 = (v1 [seed1 0]];, else, v1 = [vl seed1];
>> end, end
>> b1=[v1 0]; a1=(0 v1]; %below/above 1 unit diagonals
>> %Next here are the below/above N unit diagonals
>> bN=[-ones(1,n-N) zeros(1,N)];
>> aN=[zeros(1,N) -ones(1,n-N) ];
>> %Now we can form the n by 5 Diags matrix.
>> Diags=[bN; b1; dia; a1; aN]';
>> S=spdiags(Diags,d,n,n); %S is the sparsely stored matrix A
>> %We use a simple iteration to contruct the inhomogeneity
>> %vector b.
>> bseedl=ones(1,N);, bseedl([1 N]) = [2 2]; % 2 end pieces
>> bseed2=bseedl-ones (1,N) ; %N-2 middle pieces
>> b=bseedl; for k=l:N-2, b=[b bseed2];, end, b=[b bseedl];
>> b=b';
>> tic, xLD=s\b;, toc     ->Elapsed time is 0.250000 seconds.
>> x=ones(size(xLD));
>> max(x-xLD)      -> ans =1.0947e-013 (Exact Error)
```

Part (b): The syntax and creation of input variables is just as we did in Example 7.31.

```
>> d=(0 -N N -1 1];, diags=zeros(n,5);
>> diags(:,l)=dia; diags(:,2:3)=[bN bN']; diags(:,4:5)=[b1' b1' ];
>> tic, [xSOR, k, diff]=sorsparsediag(diags, d,b,...
               2/(l+sin(pi/101))) ; toc
->Elapsed time is 8.734000 seconds.
>> max(x-xSOR)            s-> 3.9l02e-0l2 (ExactError)
```

Thus, now that the inputted data structures are similar, MATLAB's left divide has transcended our sorsparsediags program both in performance time and in accuracy. The reader is invited to perform further experiments with sparse matrices and

MATLAB's iterative solvers.

EXERCISES 7.6:

1. For each of the following data for a linear systern $Ax = b$, perform the following iterations using the zero vector as the initial vector.
(a) Use Jacobi iteration until the error (as measured by the infinity norm of the difference of successive iterates) is less than le-10, if this is possible. In cases where the iteration does not converge, try rearranging the rows of the matrices to attain convergence (through all n ! rearrangements, if necessary). Find the norm of the exact error (use MATLAB's left divide to get the "exact" solutions of these small systems).
(b) Repeat part (a) with the Gauss-Seidel iteration.

$$\text{(i) } A = \begin{bmatrix} 6 & -1 \\ -1 & 6 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \qquad \text{(ii) } A = \begin{bmatrix} 6 & -1 & 0 \\ -1 & 6 & -1 \\ 0 & -1 & 6 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}.$$

$$\text{(iii) (iii) } A = \begin{bmatrix} -2 & 5 & 4 \\ 6 & 2 & -3 \\ 1 & 1 & -1 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}. \qquad \text{(iv) (iv) } A = \begin{bmatrix} 2 & 1 & 0 & 0 \\ 2 & 4 & 1 & 0 \\ 0 & 4 & 8 & 2 \\ 0 & 0 & 8 & 16 \end{bmatrix}, b = \begin{bmatrix} 4 \\ -2 \\ 1 \\ 3 \end{bmatrix}.$$

2. For each of the following data for a linear system $Ax = b$, perform the following iterations using the zero vector as the initial vector. Determine if the Jacobi and Gauss-Seidel iterations converge. In cases of convergence, produce a graph of the errors (as measured by the infinity norm of the difference of successive iterates) versus the number of iterations, that contains both the Jacobi iteration data as well as the Gauss-Seidel data. Let the errors go down to 10^{-10}.

$$\text{(i) } A = \begin{bmatrix} 5 & 0 \\ 2 & 4 \end{bmatrix}, \quad b = \begin{bmatrix} -1 \\ 3 \end{bmatrix}. \qquad \text{(ii) } A = \begin{bmatrix} 10 & 2 & -1 \\ 2 & 10 & 2 \\ -1 & 2 & 10 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

$$\text{(iii) } A = \begin{bmatrix} 7 & 5 & 4 \\ 3 & 2 & 1 \\ 2 & 8 & 21 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \\ 5 \end{bmatrix}. \qquad \text{(iv) } A = \begin{bmatrix} 3 & 1 & 0 & 0 \\ 1 & 9 & 1 & 0 \\ 0 & 1 & 27 & 1 \\ 0 & 0 & 1 & 81 \end{bmatrix}, \quad b = \begin{bmatrix} 4 \\ 3 \\ 2 \\ 1 \end{bmatrix}.$$

3. (a) For each of the linear systems specified in Exercise 1, run a set of SOR iterations with initial vector the zero vector by letting the relaxation parameter run form 0.05 to 1.95 in increments of 0.5. Use a tolerance of $1e-6$, but a maximum of 1000 iterations. Plot the number of iterations versus the relaxation parameter ω.
(b) Using MATLAB's eig function, let the relaxation parameter ω run through the same range 0.05 to 1.95 in increments of 0.5, and compute the maximum absolute value of the eigenvalues of the matrix $I - B^{-1}A$ where the matrix B is as in Table 7.1 (for the SOR iteration). Create a plot of this maximum versus ω, compare and comment on the relationship with the plot of part (a) and Theorem 7.10.

4. Repeat both parts (a) and (b) for each of the linear systems A x=b of Exercise 2 .

5. For the linear system specified in Exercise 2 (iv), produce graphs of the exact errors of each component of the solution: x_1, x_2, x_3, x_4 as a function of the iteration. Use the zero vector as the initial iterate. Measure the errors as the absolute values of the differences with the corresponding components of the exact solution as determined using MATLAB's left divide. Continue with iterations until the errors are all less than 10^{-10}. Point out any observations.

6. (a) For which of the linear systems specified in Exercise 1(i)-(iv) will the Jacobi iteration converge for all initial iterates?
(b) For which of the linear systems specified in Exercise 1(i)-(iv) will the Gauss-Seidel iteration converge for all initial iterates?

7. (a) For which of the linear systems specified in Exercise 2 (i)-(iv) will the Jacobi iteration converge for all initial iterates?
(b) For which of the linear systems specified in Exercise 2(i)-(iv) will the Gauss-Seidel iteration converge for all initial iterates?

8. *(An Example Where Gauss-Seidel Jteration Converges, but Jacobi Diverges)* Consider the following linear system:

$$\begin{bmatrix} 5 & 3 & 4 \\ 3 & 6 & 4 \\ 4 & 4 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ 13 \\ 13 \end{bmatrix}.$$

(a) Show that if initial iterate $x^{(0)} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}'$, the Jacobi iteration converges to the exact solution $x = [111]'$. Show that the same holds true if we start with $x^{(0)} = \begin{bmatrix} 10 & 8 & -6 \end{bmatrix}'$.
(b) Show that if initial iterate $x^{(0)} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}'$, the Gauss-Seidel iteration will diverge. Show that the same holds true if we start with $x^{(0)} = \begin{bmatrix} 10 & 8 & -6 \end{bmatrix}'$.
(c) For what sort of general initial iterates $x^{(0)}$ do the phenomena in parts (a) and (b) continue to hold?
(d) Show that the coefficient matrix of this system is positive definite. What does the SOR convergence theorem (Theorem 7.12) allow us to conclude? Suggestion: For all parts (especially part (c)) you should first do some MATLAB experiments, and then aim to establish the assertions mathematically.

9. *(An Example Where Jacobi Iteration Converges, but Gauss-Seidel Diverges)* Consider the following linear system:

$$\begin{bmatrix} 1 & 2 & -2 \\ 1 & 1 & 1 \\ 2 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}$$

(a) Show that if initial iterate $x^{(0)} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}'$, the Jacobi iteration will converge to the exact solution $x = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$ ' in just four iterations. Show that the same holds true if we start with $x^{(0)} = \begin{bmatrix} 10 & 8-6 \end{bmatrix}'$.
(b) Show that if initial iterate $x^{(0)} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}'$, the Gauss-Seidel iteration will diverge. Show that the same holds true if we start with $x^{(0)} = \begin{bmatrix} 10 & 8-6 \end{bmatrix}'$.
(c) For what sort of general initial iterates $x^{(0)}$ do the phenomena in parts (a) and (b) continue to hold? Suggestion: For all parts (especially part (c)) you should first do some MATLAB experiments, and then aim to establish the assertions mathematically. Note: This example is due to Collatz [Col-42].

10. (a) Use the formulas of Lemma 7.15 to write a function M-file with the following syntax:

```
b = sparsediag(diags, inds, x)
```

The input variables are `diags`, an $n \times j$ matrix where each column consists of the entries of A on one of its diagonals, and `inds`, a $1 \times j$ vector of the corresponding set of indices for the diagonals (index zero corresponds to the main diagonal). The last input x is the $n \times 1$ vector to be multiplied by A. The output is the corresponding product $b = Ax$.
(b) Apply this program to check that $x = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$ ' solves the linear system of Exercise 9.
(c) Apply this program to compute the matrix products of Exercise for the Reader 7.30 and check the error against the exact solution obtained in the latter.

11. (a) Modify the program sorsparsediag of Exercise for the Reader 7.31 to construct an analogous M-file:

```
[x,k,diff] = jacbobisparsediag(diags, inds, b, x0, tol, kmax)
```

for the Jacobi method.
(b) Modify the program sorsparsedia g of Exercise for the Reader 7.31 to construct an analogous M-file:

```
(x,k,diff]=gaussseidelsparsediag(diags, inds, b, x0, tol, kmax)
```

for the Gauss-Seidel method.
(c) Apply these programs to recover the results of Examples 7.26 and 7.27.
(d) Using the *M*-files of parts (a) and (b), along with `sorsparsediag`, recreate the MATLAB graphic that is shown in Figure 7.43.

12. (a) Find a 2×2 matrix A whose optimal relaxation parameter ω appears to be greater than 1.5 (as demonstrated by a MATLAB plot like the one in Figure 7.42) resulting from the solution of some linear system $Ax = b$.
(b) Repeat part (a), but this time try to make the optimal value of ω to be less than 0.5.

13. Repeat both parts of Exercise 12, this time working with 3×3 matrices.

14. (a) Find a 2×2 matrix A whose optimal relaxation parameter ω appears to be greater than 1.5 (as demonstrated by a MATLAB plot like the one in Figure 7.42) resulting from the solution of some linear system $Ax = b$.
(b) Repeat part (a), but this time try to make the optimal value of ω to be less than 0.5.

15. *(A Program 10 Estimate the Optimal SOR Parameter ω)*(a) Write a program that will aim to find the optimal relaxation parameter ω for the SOR method in the problem of solving a certain linear system $Ax = b$ for which it is assumed that the SOR method will converge. (For example, by the SOR convergence theorem, if A is symmetric positive definite, this program is applicable.) The syntax is as follows:

```
omega = optimalomega (A,b,tol,iter)
```

Of the input and output variables, only the last two input variables need explanation. The input variable `tol` is simply the accuracy goal that we wish to approximate `omega`. The variable `iter` denotes the number of iterations to use on each trial run. The default value for`tol` is le-3 and for `iter` it is 10. (For very large matrices a larger value may be needed for `iter`, and likewise for very small matrices a smaller value should be used.) Once this tolerance is met, the program terminates. The program should work as follows: First run through a set of SOR iterations with the values of ω running from 0.05 to 1.95 in increments of 0.05. For each value of ω we run through iter iterations. For each of these we keep track of the infinity norm of the difference of the final iterate and the immediately preceding iterate. For each tested value of $\omega = a_0$ for which this norm is minimal, we next run the tests on the values of ω running from $a_b - .05$ to $a_0 + .05$ in increments of 0.005 (omit the values $\omega = 0$ or $\omega = 2$ should these occur as endpoints). In the next iteration, we single out those new values of ω for which the new error estimate is minimized. For each new corresponding value $\omega = \omega_b$ for which the norm is minimized, we will next run tests on the set of values from $\omega_0 - .005$ to $a_b + .005$ in increments of 0.0005. At each iteration, the minimizing values of $\omega = \omega_0$ should be unique; if they are not, the program should deliver an error message to this effect, and recommend to try running the program again with a larger value of `iter`. When the increment size is less than tol, the program terminates and outputs the resulting value of $\omega = \omega_b$.
(b) Apply the above program to aim to determine the optimal value of the SOR parameter ω for the linear system of Example 7.26 with default tolerances. Does the resulting output change if we change `iter` to 5 ? To 20 ?
(c) Repeat part (b), but now change the default tolerance to le-6.
(d) Run the SOR iteration on the linear system using the values of the relaxation parameter computed in parts (a) and (b) and compare the rate of convergences with each other and with that seen in the text when $\omega = 0.9$ (Figure 7.41).
(e) Is the program in part (a) practical to run on the large matrix such as the 2500×2500 matrix of Example 7.30 (perhaps using a small value for `iter`)? If yes, run the program and compare with the result of Proposition 7.14.

16. *(A Program to Estimate the Optimal SOR Parameter ω for Sparse Banded Systems)* (a) Write a program that will aim to find the optimal relaxation parameter ω for the SOR method in the problem of solving a certain linear system $Ax = b$ for which it is assumed that the SOR method will converge. The functionality of the program will be similar to that of the preceding exercise, except that now the program should be specially designed to deal with sparsely banded systems, as did the program `sorsparsediag` of Exercise for the Reader 7.31 (in fact, the present program should call on this previous program). The syntax is as follows:

```
omega = optimalomegasparsediag(diags, inds, b, tol,iter)
```

The first three input variables are as explained in Exercise for the Reader 7.31 for the program `sorsparsediag`. The remaining variables and functionality of the program are as explained in the preceding exercise.
(b) Apply the above program to aim to determine the optimal value of the SOR parameter ω for the linear system of Example 7.26 with default tolerances. Does the resulting output change if we change `iter` to 5 ? To 20?
(c) With default tolerances, run the program on the linear system of Example 7.30 and compare with the exact result of Proposition 7.14. You may need to experiment with different values of `iter` to attain a successful approximation. Run SOR on the system with this computed value for the optimal relaxation parameter, and 308 iterations. Compute the exact error and compare with the results of Example 7.31.
(d) Repeat part (c) but now with `tol` reset to le-6.

NOTE: For tridiagonal matrices that are positive definite, the following formula gives the optimal value of the relaxation parameter for the SOR iteration:

$$\omega = \frac{2}{1+\sqrt{1-\rho(D-L)^2}}, \tag{7.39}$$

where the matrices D and L are as in (60) $A = D - L - U$,[24] and $\rho(D-L)$ denotes the spectral radius of the matrix $D - L$.

17. We consider tridiagonal square $n \times n$ matrices of the following form:

$$F = \begin{bmatrix} 2 & a & & & & \\ a & 2 & a & & 0 & \\ & a & 2 & a & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & a \\ & & & & a & 2 \end{bmatrix}$$

(a) With $a = -1$ and $n = 10$, show that F is positive definite.
(b) What does formula (17) give for the optimal SOR parameter for the linear system?
(c) Run the SOR iteration with the value of ω obtained in part (b) for the linear system $F_x = b$ where the exact solution is $x = \begin{bmatrix} 1 & 2 & 1 & 2 & \dots & 1 \end{bmatrix}$.How many iterations are needed to get the exact error to be less than le-10?
(d) Create a graph comparing the performance of the SOR of part (c) along with the corresponding Jacobi and Gauss-Seidel iterations.

18. Repeat all parts of Exercise 17, but change a to -0.5.

19. Repeat all parts of Exercise 17 , but change n to 100 . Can you prove that with $a=-1$, the matrix F in Exercise 17 is always positive definite? If you cannot, do some MATLAB experiments (with different values of n) and conjecture whether you think this is a true statement.

20. (a) Show that the Jacobi iteration scheme is represented in matrix form (61) by the matrix B indicated in Table 7.1.
(b) Repeat part (a) for the Gauss-Seidel iteration.
(c) Repeat part (a) for the SOR iteration.

21. Prove Lemma 7.15.

22. (a) Given a nonsingular matrix A, find a corresponding matrix T so that the Jacobi iteration can be expressed in the form $x^{(k+1)}=x^{(k)}+\mathrm{Tr}^{(k)}$, where $r^{(k)}=b-Ax^{(k)}$ is the residual vector for the k th iterate.
(b) Repeat part (a) for the Gauss-Seidel iteration.
(c) Can the result of part (b) be generalized for the SOR iteration?

23. *(Proof of Jacobi Convergence Theorem)* Complete the following outline for a proof of the Jacobi Convergence Theorem (part of Theorem 7.11): As in the text, we let $e^{(k)}=x^{(k)}-x$ denote the error vector of the k th iterate $x^{(k)}$ for the Jacobi method for solving a linear system $Ax=b$, where the $n\times n$ matrix A is assumed to be strictly diagonally dominant. For each (row) index i, we let $\mu_i=\frac{1}{|a_{ii}|}\sum_{\substack{j=1\\ j\neq i}}^{n}|a_{ij}|\,(1\leq i\leq n)$. For any vector v, we let $\|v\|$ denote its infinity norm: $\|v\|_\infty=\max(|v_i|)$.
For each iteration index k and component index i, use the triangle inequality to show that

$$\left|e_i^{(k)}\right|\leq\frac{1}{|a_{ii}|}\sum_{\substack{j=1\\ j\neq i}}^{n}|a_{ij}|\left|e_i^{(k-1)}\right|\leq\mu_i\left\|e^{(k-1)}\right\|,$$

and conclude that

$$\left\|e^{(k)}\right\|\leq\|\mu\|\left\|e^{(k-1)}\right\|,$$

and, in turn, that the Jacobi iteration converges.

This page intentionally left blank