EMTH211-19S2 Tutorial 1

Introduction to Matlab

MATLAB is a computer package that is design to do matrix calculations. It will be used extensively in this course. Most of you will have already met MATLAB in EMTH171. If you have not used MATLAB before, first work through the three MATLAB worksheets in the Extra Study Material section on Learn.

This tutorial reviews working with matrices in MATLAB. Work through the following sections and enter all the commands. The final section is an investigation on solving tridiagonal systems of linear equations using MATLAB.

1 Matlab Commands

Entering Matrices

The best way for you to get started with MATLAB is to learn how to handle matrices. Start MATLAB and follow along with each example. You can enter matrices into MATLAB in several different ways. The most straightforward way is to enter an explicit list of elements.

Start by entering Dürer's matrix as a list of its elements. You only have to follow a few basic conventions:

- Separate the elements of a row with blanks or commas.
- Use a semicolon, `;` , to indicate the end of each row.
- Surround the entire list of elements with square brackets, `[ ]`.

To enter Dürer's matrix, simply type in the Command Window

```
A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
```

MATLAB displays the matrix you just entered.

```
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

Once you have entered the matrix, it is automatically remembered in the MATLAB workspace. You can refer to it simply as `A`. Note that MATLAB is **case sensitive**. The variable `A` is **different** to the variable a. Now that you have `A` in the workspace, take a look at what makes it so interesting. Why is it magic?

sum, transpose, and diag

You are probably already aware that the special properties of a magic square have to do with the various ways of summing its elements. If you take the sum along any row or column, or along either of the two main diagonals, you will always get the same number. Let us verify that using MATLAB. The first statement to try is

```
sum(A)
```

MATLAB replies with

```
ans =
    34    34    34    34
```

When you do not specify an output variable, MATLAB uses the variable `ans`, short for answer, to store the results of a calculation. You have computed a row vector containing the sums of the columns of `A`. Sure enough, each of the columns has the same sum, the magic sum, 34.

How about the row sums? MATLAB has a preference for working with the columns of a matrix, so the easiest way to get the row sums is to transpose the matrix, compute the column sums of the transpose, and then transpose the result. The transpose operation is denoted by an apostrophe or single quote, `'`. It flips a matrix about its main diagonal and it turns a row vector into a column vector.

So

```
 A'
```

produces

```
ans =
    16     5     9     4
     3    10     6    15
     2    11     7    14
    13     8    12     1
```

and

```
sum(A')'
```

produces a column vector containing the row sums

```
ans =
    34
    34
    34
    34
```

The sum of the elements on the main diagonal is obtained with the `sum` and the `diag` functions.

```
diag(A)
```

produces

```
ans =
    16
    10
     7
     1
```

and

```
sum(diag(A))
```

produces

```
ans =
    34
```

The other diagonal, the so-called antidiagonal, is not so important mathematically, so MATLAB does not have a ready-made function for it. But a function originally intended for use in graphics, `fliplr`, flips a matrix from left to right.

```
sum(diag(fliplr(A)))
ans =
    34
```

You have verified that Dürer matrix is indeed a magic square and, in the process, have sampled a few MATLAB matrix operations.

Subscripts

The element in row i and column j of A is denoted by `A(i,j)`. For example, `A(4,2)` is the number in the fourth row and second column. For our magic square, `A(4,2)` is 15. So to compute the sum of the elements in the fourth column of `A`, type

```
A(1,4) + A(2,4) + A(3,4) + A(4,4)
```

This produces

```
ans =
     34
```

but is not the most elegant way of summing a single column.

If you try to use the value of an element outside of the matrix, it is an error. Thus

```
t = A(4,5)
Index exceeds matrix dimensions.
```

On the other hand, if you store a value in an element outside of the matrix, the size increases to accommodate the newcomer.

```
X = A;
X(4,5) = 17

X =
    16     3     2    13     0
     5    10    11     8     0
     9     6     7    12     0
     4    15    14     1    17
```

In this example, the semicolon in command `X = A;` supresses the output. Normally the value of `X` would appear in the Command Window.

The Colon Operator

The colon, :, is one of the most important MATLAB operators. It occurs in several different forms. The expression

```
1:10
```

is a row vector containing the integers from 1 to 10

```
1     2     3     4     5     6     7     8     9    10
```

To obtain nonunit spacing, specify an increment. For example,

```
100:-7:50
```

is

```
100    93    86    79    72    65    58    51
```

and

```
0:pi/4:pi
```

is (`pi` is the name of the mathematical constant π in MATLAB)

```
0    0.7854    1.5708    2.3562    3.1416
```

Subscript expressions involving colons refer to portions of a matrix.

```
A(1:k,j)
```

is the first k elements of the jth column of `A`. So

```
sum(A(1:4,4))
```

computes the sum of the fourth column. But there is a better way. The colon by itself refers to all the elements in a row or column of a matrix and the keyword `end` refers to the last row or column. So

```
sum(A(:,end))
```

computes the sum of the elements in the last column of `A`.

```
ans =
     34
```

Why is the magic sum for a 4-by-4 square equal to 34? If the integers from 1 to 16 are sorted into four groups with equal sums, that sum must be

```
sum(1:16)/4
```

which, of course, is

```
ans =
     34
```

2 Expressions

Like most other programming languages, MATLAB provides mathematical expressions, but unlike most programming languages, these expressions involve entire matrices. The building blocks of expressions are variables, numbers and operators.

Variables

MATLAB does not require any type declarations or dimension statements. When MATLAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage. If the variable already exists, MATLAB changes its contents and, if necessary, allocates new storage. For example,

```
num_students = 25
```

creates a 1-by-1 matrix named `num_students` and stores the value 25 in its single element.

Variable names consist of a letter, followed by any number of letters, digits, or underscores. MATLAB uses only the first 31 characters of a variable name. MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. `A` and `a` are not the same variable. To view the matrix assigned to any variable, simply enter the variable name.

Numbers

MATLAB uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, for numbers. Scientific notation uses the letter `e` to specify a power-of-ten scale factor. Imaginary numbers use either `i` or `j` as a suffix. Some examples of legal numbers are

```
3              -99            0.0001
9.6397238      1.60210e-20    6.02252e23
1i             -3.14159j      3e5i
```

All numbers are stored internally using the long format specified by the IEEE floating-point standard. Floating-point numbers have a finite precision of roughly 16 significant decimal digits and a finite range of roughly 10^{-308} to 10^{+308}.

Operators

Expressions use familiar arithmetic operators and precedence rules.

- Addition +
- Subtraction –
- Multiplication *
- Division /
- Left division \
- Power ^
- Complex conjugate transpose '
- Specify evaluation order ( )

All these operations are *matrix* operations. For example, multiplication is *matrix* multiplication. Matrix division is not normally defined mathematically. These operations are discussed below in the section on solving linear systems.

3 Working with Matrices

This section introduces you to other ways of creating matrices.

MATLAB provides five functions that generate basic matrices.

- `zeros` - All zeros
- `ones` - All ones
- `eye` - A matrix with ones on the diagonal and zeros everywhere else
- `rand` - Uniformly distributed random elements
- `randn` - Normally distributed random elements

Here are some examples.

```
Z = zeros(2,4)
Z =
     0     0     0     0
     0     0     0     0

F = 5*ones(3,3)
F =
     5     5     5
     5     5     5
     5     5     5

eye(2,4)
ans =
     1     0     0     0
     0     1     0     0

N = fix(10*rand(1,10))
N =
     4     9     4     4     8     5     2     6     8     0

R = randn(4,4)
R =
    1.0668    0.2944   -0.6918   -1.4410
    0.0593   -1.3362    0.8580    0.5711
   -0.0956    0.7143    1.2540   -0.3999
   -0.8323    1.6236   -1.5937    0.6900
```

The command `fix` will round its argument towards zero.

The `diag` command is also useful in constructing matrices. If its argument is a vector (rather than a matrix) it returns a diagonal matrix with that vector on the diagonal. Thus

```
diag([1 2 3])
ans =
       1 0 0
       0 2 0
       0 0 3
```

`diag` also has an (optional) argument to give an offset. Thus `diag(ones(3,1),-1)` yields

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

and `diag(ones(4,1),1)` yields

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

This will be helpful later when working with tridiagonal matrices.

Concatenation

Concatenation is the process of joining small matrices to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, `[]`, is the concatenation operator. For an example, start with the 4-by-4 magic square, `A`, and form

```
B = [A  A+32; A+48  A+16]
```

The result is an 8-by-8 matrix, obtained by joining the four submatrices.

```
B =

    16     3     2    13    48    35    34    45
     5    10    11     8    37    42    43    40
     9     6     7    12    41    38    39    44
     4    15    14     1    36    47    46    33
    64    51    50    61    32    19    18    29
    53    58    59    56    21    26    27    24
    57    54    55    60    25    22    23    28
    52    63    62    49    20    31    30    17
```

This matrix is halfway to being another magic square. Its elements are a rearrangement of the integers 1:64. Its column sums are the correct value for an 8-by-8 magic square.

```
sum(B)

ans =
   260   260   260   260   260   260   260   260
```

But its row sums, `sum(B')'`, are not all the same. Further manipulation is necessary to make this a valid 8-by-8 magic square. Note that `A+32` has the effect of adding 32 to each entry of `A`.

Deleting Rows and Columns

You can delete rows and columns from a matrix using just a pair of square brackets. Start with

```
X = A;
```

Then, to delete the second column of `X`, use

```
X(:,2) = []
```

This changes `X` to

```
X =
    16     2    13
     5    11     8
     9     7    12
     4    14     1
```

If you delete a single element from a matrix, the result is not a matrix anymore. So, expressions like

```
X(1,2) = []
```

result in an error.

Elementary Row Operations

Elementary row operations can be easily achieved in MATLAB with the use of the `:` operator. For a given matrix `A`, we have

- Multiply row `i` by a constant `c`

  ```
  A(i,:) = c*A(i,:)
  ```

- Add `c` times row `j` to row `i`

  ```
  A(i,:) = A(i,:) + c*A(j,:)
  ```

- Interchange rows `i` and `j`

  ```
  A([i j],:) = A([j i],:)
  ```

Note that we can select multiple rows of a matrix by using square brackets in the index. Thus `A([1 2 5],:)` will select rows 1, 2 and 5 whereas `A([1 3:6],:)` will select rows 1, 3, 4, 5 and 6 (why?).

Matrix Operations

Adding a matrix to its transpose produces a symmetric matrix. Thus

```
A + A'

ans =
    32     8    11    17
     8    20    17    23
    11    17    14    26
    17    23    26     2
```

The multiplication symbol, *, denotes the matrix multiplication involving inner products between rows and columns. Multiplying the transpose of a matrix by the original matrix also produces a symmetric matrix.

```
A'*A

ans =
   378   212   206   360
   212   370   368   206
   206   368   370   212
   360   206   212   378
```

The determinant of this particular matrix happens to be zero, indicating that the matrix is singular.

```
d = det(A)

d =
     1.0871e-12
```

Here we see a potential problem with numerical computations. This number is *very small* (note you may see a different but equally small number when you do this computation). Is it really zero (and we are seeing roundoff error) or a small non-zero number? The distinction is critical. If the determinant is zero then A cannot be inverted and an associated system of equations has either no solutions or infinitely many solutions. If it is non-zero then A is invertible and an associated system has a unique solution. The reduced row echelon form of A is not the identity.

```
R = rref(A)

R =
     1     0     0     1
     0     1     0    -3
     0     0     1     3
     0     0     0     0
```

Thus A is **not** invertible and the determinant is really 0. If you try to compute the inverse with

```
X = inv(A)
```

you will get a warning message

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 9.796086e-018.
```

Roundoff error has prevented the matrix inversion algorithm from detecting exact singularity. But the value of `rcond`, which stands for reciprocal condition estimate (again you may see a different but equally small number), is on the order of the floating-point relative precision (in other words, extremely small), so the computed inverse is unlikely to be of much use.

4 Controlling Command Window Input and Output

Format function

The format function controls the numeric format of the values displayed by MATLAB. The function affects only how numbers are displayed, not how MATLAB computes or saves them. Here are the different formats, together with the resulting output produced from a vector x with components of different magnitudes.

Note that to ensure proper spacing, use a fixed-width font, such as Courier.

```
x = [4/3 1.2345e-6]

format short

   1.3333    0.0000

format short e

   1.3333e+000  1.2345e-006

format short g

   1.3333  1.2345e-006

format long

   1.33333333333333   0.00000123450000

format long e

   1.333333333333333e+000    1.234500000000000e-006

format long g

   1.33333333333333               1.2345e-006

format bank

   1.33          0.00

format rat

   4/3          1/810045
```

If the largest element of a matrix is larger than 10^3 or smaller than 10^{-3}, MATLAB applies a common scale factor for the short and long formats.

In addition to the format functions shown above

```
format compact
```

suppresses many of the blank lines that appear in the output. This lets you view more information on a screen or window.

Suppressing Output

If you simply type a statement and press Return or Enter, MATLAB automatically displays the results on screen. However, if you end the line with a semicolon, MATLAB performs the computation but does not display any output. This is particularly useful when you generate large matrices. For example,

```
B = rand(100,100);
```

Entering Long Statements

If a statement does not fit on one line, use an ellipsis (three periods), ..., followed by Return or Enter to indicate that the statement continues on the next line. For example,

```
s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...
      - 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```

Blank spaces around the $=$, $+$, and - signs are optional, but they improve readability.

Command Line Editing

Various arrow and control keys on your keyboard allow you to recall, edit, and reuse statements you have typed earlier. For example, suppose you mistakenly enter

```
rho = (1 + sq(5))/2
```

You have misspelled `sqrt`. MATLAB responds with

```
Undefined function or variable 'sq'.
```

Instead of retyping the entire line, simply press the ↑ key. The statement you typed is redisplayed. Use the ← key to move the cursor over and insert the missing `r`. Repeated use of the ↑ key recalls earlier lines. Typing a few characters and then the ↑ key finds a previous line that begins with those characters. You can also copy previously executed statements from the Command History.

Following is the list of arrow keys you can use in the Command Window.

- ↑ Recall previous line. Works only at command line.
- ↓ Recall next line. Works only at command line if you previously used the ↑.
- ← Move back one character.
- → Move forward one character.

5 Solving Linear Systems of Equations

One of the most important problems in technical computing is the solution of simultaneous linear equations. In matrix notation, this problem can be stated as follows. Given two matrices A and B, does there exist a unique matrix X so that $AX = B$ or $XA = B$? This is clearly a generalization the problem of solving $A\mathbf{x} = \mathbf{b}$ which was considered in EMTH118/119 and MATH199.

It is instructive to consider a 1-by-1 example. Does the equation

$$7x = 21$$

have a unique solution? The answer, of course, is yes. The equation has the unique solution $x = 3$. The solution is easily obtained by division.

The solution is not ordinarily obtained by computing the inverse of 7, that is $7^{-1} = 0.142857\ldots$, and then multiplying 7^{-1} by 21. This would be more work and, if 7^{-1} is represented to a finite number of digits, less accurate. Similar considerations apply to sets of linear equations with more than one unknown; MATLAB solves such equations **without** computing the inverse of the matrix.

Although it is not standard mathematical notation, MATLAB uses the division terminology familiar in the scalar case to describe the solution of a general system of simultaneous equations. The two division symbols, slash, /, and backslash, \, are used for the two situations where the unknown matrix appears on the left or right of the coefficient matrix. Thus

```
X = A\B
```

denotes the solution to the matrix equation $AX = B$ whereas

```
X = B/A
```

denotes the solution to the matrix equation $XA = B$. In particular, for a column vector `b`, the solution to the system of linear equations $A\mathbf{x} = \mathbf{b}$ is given by

```
x = A\b
```

You can think of "dividing" both sides of the equation `AX = B` or `XA = B` by `A`. The coefficient matrix `A` is always in the "denominator". The dimension compatibility conditions for `X = A\B` require the two matrices `A` and `B` to have the same number of rows. The solution `X` then has the same number of columns as `B` and its row dimension is equal to the column dimension of `A`. For `X = B/A`, the roles of rows and columns are interchanged.

In practice, linear equations of the form `AX = B` occur more frequently than those of the form `XA = B`. Consequently, backslash is used far more frequently than slash. The remainder of this section concentrates on the backslash operator; the corresponding properties of the slash operator can be inferred from the identity

```
(B/A)' = (A'\B')
```

The coefficient matrix `A` need not be square. If `A` is $m \times n$, there are three cases.

- $m = n$. Square system (exactly the same number of equations as unknowns). Seek an exact solution.

- $m > n$. Overdetermined system (more equations than unknowns). Find a least squares solution. This case will be considered later in this course.
- $m < n$. Underdetermined system (more unknowns than equations). Find a basic solution with at most m nonzero components. This case will be not be considered in this course.

The backslash operator employs different algorithms to handle different kinds of coefficient matrices. None of these algorithms rely on computing the inverse. Gaussian elimination is an example of one of these algorithms though there may be more efficient methods available.

Square Systems

The most common situation involves a square coefficient matrix `A` and a single right-hand side column vector `b`.

If the matrix `A` is nonsingular (that is, invertible), the solution, `x = A\b`, is then the same size as `b`. For example,

```
A=pascal(3)

A =

     1     1     1
     1     2     3
     1     3     6

b = [3; 1; 4];
x = A\b

x =
      10
     -12
       5
```

Note `pascal` generates a special matrix called the Pascal matrix. It can be confirmed that `A*x` is exactly equal to `b`. This example has an exact, integer solutions. This is because the coefficient matrix was chosen to be `pascal(3)`, which has a determinant equal to one.

If A is singular, the solution to `Ax = b` either does not exist, or is not unique. The backslash operator, `A\b`, issues a warning if `A` is nearly singular and raises an error condition if it detects exact singularity. For example,

```
A=[1 3 7;-1 4 4;1 10 18]

A =

     1     3     7
    -1     4     4
     1    10    18

det(A)

ans =
```

```
     0

b=[5;2;12]

b =

     5
     2
    12

A\b
Warning: Matrix is singular to working precision.

ans =

   NaN
   NaN
   NaN
```

NaN stands for *Not a Number*. It is MATLAB'S notation for, among other things, ∞. You can determine whether `Ax = b` has an exact solution by finding the row reduced echelon form of the augmented matrix `[A b]`. To do so for this example, type

```
rref([A b])

ans =

    1.0000         0    2.2857    2.0000
         0    1.0000    1.5714    1.0000
         0         0         0         0
```

We see that the last row is all zeros and so this system does have an exact solution. In fact, it has infinitely many solutions.

It is important to realize that, due to finite precision arithmetic, matrices that are singular, may appear to have unique solutions. The example of Dürer's matrix that we began with is a case in point. We have already seen that its determinant is zero but

```
A=[16 3 2 13; 5 10 11 8;9 6 7 12; 4 15 14 1];
A\[1 1 1 1]'
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 9.796086e-018.

ans =

    0.0276
    0.0349
    0.0239
    0.0313
```

MATLAB warnings are important!

6 Plotting

The natural operations in MATLAB are **matrix** operations. Thus `A^2` will give the matrix product AA. There are occasions when we might want an "elementwise" power (or product); that is, we want a matrix whose entries are given function of a equivalent entry of another matrix. An operator prefixed with a `.` will act elementwise rather than as a matrix operation. Thus, as the example shows, `A.^2` produces a matrix whose entries are the square of the entries of A.

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

     1     2     3
     4     5     6
     7     8     9
A^2
ans =

    30    36    42
    66    81    96
   102   126   150
A.^2
ans =

     1     4     9
    16    25    36
    49    64    81
```

Note that for an elementwise product, the matrices must be the *same* size.

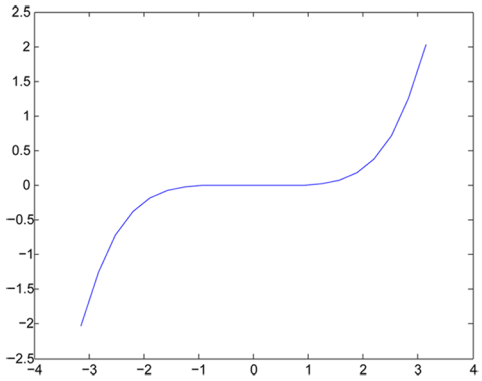
A case where these elementwise operations occur is plotting. In its simplest application, the MATLAB command `plot(x,y)` will plot the values in the vector `y` (on the vertical axis) against the values in the vector `x` (on the horizontal axis). Both vectors have to be the same length. Thus to plot the function

$$y = \sin x + \frac{x^3}{6} - x$$

on say $[-\pi, \pi]$ we first construct a vector of values between $-\pi$ and π (in the example below we have used evenly spaced points but this is not necessary) and then evaluate the function at each of these points. In this case we want elementwise operations and not matrix operations (note that, since `x` is a vector, `x^3` is not defined).

```
x=-pi:pi/10:pi;
y=sin(x) + x.^3/6 - x;
plot(x,y)
```

will produce the plot



7 M-files

Scripts (M-files) are simply files containing a sequence of MATLAB statements. The name of an M-file begins with an alphabetic character, and has a filename extension of `.m`. The M-file name, less its extension, is what MATLAB searches for when you try to use the script.

When you invoke a script, MATLAB simply executes the commands found in the file. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. For example, you can create your own matrices using M-files. Choose `New Script` or `New Live Script` from the top of the MATLAB window or another text editor[1] to create a file containing the same statements you would type at the MATLAB command line. Save the file under a name that ends in .m. For example, create a file containing these five lines.

```
A = [ ...
    16.0     3.0     2.0    13.0
     5.0    10.0    11.0     8.0
     9.0     6.0     7.0    12.0
     4.0    15.0    14.0     1.0 ];
```

Store the file under the name `magik.m`. Then the statement `magik` reads the file and creates a variable, `A`, containing our example matrix.

Any variables that a script creates remain in the workspace, to be used in subsequent computations. For example consider the script `magicrank.m`

```
% Investigate the rank of magic squares
r = zeros(1,32);
for n = 3:32
   r(n) = rank(magic(n));
end
r
bar(r)
```

Typing the statement `magicrank` causes MATLAB to execute the commands, compute the rank (a concept that is discussed in lectures) of the first 30 magic squares, and plot a bar graph of the result. After execution of the file is complete, the variables `n` and `r` remain in the workspace. The `for ... end` is a loop. It causes MATLAB to execute the enclosed statements for `n=3,4,5,...,31,32`. MATLAB will ignore anything on a line that follows a `%`. This allows us to add comments to scripts which improves readability (and is **strongly** encouraged).

Normally, the commands in M-files do not display on the screen during execution. The `echo` command allows the commands to be viewed as they execute. The use of echo is simple; echoing can be either on or off, in which case any script used is affected.

`echo on`	Turns on the echoing of commands in all script files.
`echo off`	Turns off the echoing of commands in all script files.
`echo`	Toggles the echo state.

The advantage in using scripts is that you have a way of repeating work without retyping it. This is very useful for debugging purposes and for extending work already done.

By highlighting the commands in command history windows that you wish to keep (by holding the Ctrl key while clicking on the commands or the Shift key if you want to select a range of commands) and then right

[1] If you wish to use a word processor like WORD, you must save the file as "plain text" without any formatting.

clicking, MATLAB will generate a M-file with those commands for you to save. This file can then be edited if required.

8 Output

A hardcopy of a MATLAB session can be produced by using the print command which is accessed from the `File` menu. Alternatively, we can use the `diary` command.

The diary function creates a log of keyboard input and the resulting output (except it does not include graphics). The output of diary is an ASCII file, suitable for printing or for inclusion in reports and other documents. This file can also be edited to remove extraneous output or to add comments and explanations of your work. The syntax of the command is

`diary on`	Resumes the diary. If a filename has not been specified, MATLAB creates a file called `diary`.
`diary off`	Suspends the diary.
`diary filename`	Writes to the named file, where filename is the full pathname or filename is in the current MATLAB directory.
`diary`	Toggles diary mode on and off.

If the file already exists, the output is **appended** to the end of the file. You cannot use a filename called `off` or `on`.

A third alternative is to use WINDOWS cut and paste to put MATLAB output into a file which has been opened in an text editor.

9 Solving Tridiagonal Systems

Matrices of the form

$$A = \begin{bmatrix} a & b & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ b & a & b & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & b & a & b & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & b & a & b \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & b & a \end{bmatrix}$$

occur frequently in solving partial differential equations. They are *tridiagonal* matrices with a on the main diagonal and b on the diagonals immediately above and below the main diagonal. All other elements are zero.

Let $a = 1.2$, $b = -0.1$. Consider the system

$$A\mathbf{x} = \mathbf{d}$$

where **d** is a *random* vector (of appropriate length).

1.1 Solve this system with the MATLAB backslash operator when A is a matrix of size 100×100, 1000×1000 and 5000×5000. Measure the (cpu) time it takes MATLAB to solve each of these cases (if you don't know how to do this, check `doc cputime`). Footnote: When timing a routine you should run it a number of times and then average the time taken to get a reliable estimate.

1.2 MATLAB has a `sparse` function (see `doc sparse`). The use of this function reduces storage (only non-zero elements are stored). Moreover backslash will use specialised routines to exploit the *structure* of A. Repeat your timings when A is "sparsified".

Systems of this type arise when the heat equation

$$u_t = u_{xx}$$

is solved using backward central differences. In that case $a = 1 + 2r$ and $b = -r$ where

$$r = \frac{\Delta t}{(\Delta x)^2}.$$

Δt and Δx are the time increment and spacing in the spatial grid respectively. The values of the temperature at time $t = (k+1)\,\Delta t$ is then given by

$$A\,\mathbf{u}_{k+1} = \mathbf{u}_k.$$

The moral of this example is that **any structure that A possesses should be exploited by the algorithm chosen to solve the system.**