Chapter 7

A Least Squares Filter

This ***Maple*** session is about matrices and how ***Maple*** can be made to employ matrices for attacking practical problems. There are a large number of matrix functions known to ***Maple***.

Here we see how ***Maple*** might be useful in an attempt to solve a difficult mathematical problem. We will not delve much deeper than we would without the use of a computer algebra system. However our use of ***Maple*** will speed up the learning process significantly. It is possible that you will not fully comprehend the meaning of the ***Maple*** procedures that we will use here. I hope that after you have worked through some of the work, you will understand this session better.

7.1 Matrices and Vectors

We will load all available matrix functions at once with the ***Maple*** command `with(linalg)`.

```
> with(linalg):
```

Matrices can be introduced using the `matrix` function which is part of the **linalg** package.

```
> A := matrix(3,4,(i,j)-> 1 + 2*i - 3*j );
> A, evalm(A);
```

Note that in order to evaluate a matrix, the we must use the `evalm` procedure, otherwise the matrix is merely evaluated to its name. The call to ***A*** is not a call that fully evaluates ***A***. It is easier for ***Maple*** to deal with ***A***, not with the matrix representation.

In addition to the `matrix` procedure, the **linalg** package also contains a `vector` command. The first argument of the `vector` procedure determines the number of vector components. Apparently, ***Maple*** does not distinguish row vectors from column vectors, because, as you will see below, ***Maple*** displays a vector as a list of its components, arranged in a row and separated by commas. The output of ***Maple***'s `vector` procedure should always be seen as a one-dimensional list. On the other hand, a column vector really is a 3 x 1 matrix; here two dimensions are involved. The next lines verify all this.

```
> b := vector(3,[1,2,3]); c := matrix(1,3,[1,2,3]);
> evalm(b&*A);
>  evalm(transpose(A)&*b);
```

```
>  evalm(c&*A);
```

```
>  evalm(transpose(A)&*c);
```

Note the difference between the list output and the row vector output; the former uses commas to separate elements, the latter does not. First, in `b&*A`, Maple treats the vector **b** as a 1 x 3 matrix, while in `transpose(A)&*b`, the vector **b** is suddenly seen as a 3×1 matrix! Of course, this is all for the benefit of matching dimensions in vector-matrix multiplication.

Each specific matrix element can be extracted by using a double index system, row number first. The instruction

```
>  A[2,3];
```

shows the matrix element on the intersection of the second row and the third column. An individual matrix element can be changed by assigning a new value to it.

```
>  A[2,3] := 9;
```

Now verify that this change really has taken effect.

```
>  A = evalm(A);
```

Apparently, calling just A (note that no back quotes are used around A in the left-hand side) returns A unevaluated without showing its elements. The letter **m** in `evalm` comes from *matrix* of course. The `evalm` command reveals the values of all elements of many types of matrix expressions, such as sums and products of matrices.

```
>  evalm(A + A);
```

This different property of matrix evaluation has a strange effect on assignments like

```
>  B := A;
```

Each element of B gets the same value as the corresponding element of A, which is what we expected. The surprising information is that new changes on the elements of A automatically effect B in the same way!

```
>  evalm(A), evalm(B); A[2,3] := 6; evalm(A), evalm(B);
```

To avoid this happening we have ***Maple***'s `copy` command.

```
>  C := copy(A);
```

```
>  evalm(A), evalm(C); A[3,4] := 11; evalm(A), evalm(C);
```

Observe that the change in A has no effect on the matrix C.

The elements of a matrix often show a certain regularity. In many cases such a regular pattern can be considered the result of a function prescription applied to each pair of indices (i, j). ***Maple*** enables the user to insert a prescription as part of the matrix definition. For example:

```
>  f := (i,j) -> 1/(i+j-1);
```

```
>  H := n -> matrix(n,n,f);
```

This defines the so-called Hilbert matrices $H(\ n)$.

```
>  H(4);
```

It is not absolutely necessary to introduce a matrix in the completely explicit way of giving all its elements as we did before. A symbolic matrix is obtained when elements are not explicitly given. Of course, the size of the matrix must then be specified.

```
> A := matrix(2,3); evalm(A);
```

It is also possible to instruct ***Maple*** to provide a symmetric (symbolic) matrix. However, this can not be done by means of the `matrix` command, the `array` procedure should be used instead. After introducing a symmetric matrix A in this way, it is easy to check that A is indeed symmetric.

```
> A := array(1..3,1..3,`symmetric`);
> i, j, A[i,j] - A[j,i];
```

First we have made sure that both i and j are free variables.

Other special matrices are zero matrices and identity matrices. A zero matrix is a matrix of only zero elements. It can be defined by inserting a single 0 at the position where we normally type the matrix elements, or where the element prescription goes.

```
> ZeroMatrix := (m,n) -> matrix(m,n,0);
> ZeroMatrix(2,3);
```

Identity matrices are not automatically provided by ***Maple***. You will have to create them yourself.

```
> Id1 := n -> array(1..n,1..n,`identity`);
> Id2 := n -> matrix(n,n,(i,j) -> if i=j then 1 else 0 fi);
> Id1(3), Id2(4);
```

A disadvantage of both methods is that the size has to be specified. It would be nice to have a single symbol for all identity matrices. The symbol **I** (capital i) is generally reserved for this purpose. But ***Maple*** has another use for it, namely the complex number $i = \sqrt{-1}$. Fortunately, the designers of ***Maple*** invented another symbol, or rather procedure, that produces a universal identity matrix, namely `&*()`. Assisted by ***Maple***'s `alias` function, we can now create a more natural symbol, such as **Id**, which is short for identity.

```
> alias(Id = &*());
```

Recall that Maple uses the `&*` symbol for matrix multiplication, in contrast to the single `*` which stands for scalar multiplication. One reason for this difference is that the usual multiplication symbol `*` is only used for commutative operations, and, as you know, matrix multiplication is not commutative. The distinction between commutative and non-commutative is important in connection with the automatic simplification of complicated expressions.

```
> A := matrix(7,5); evalm(A&*Id - Id&*A);
```

Note that in the second expression above the symbol **Id** is used for two different identity matrices, one of size 5×5, the other of size 7×7.

7.1.1 Computations with Matrices

In order to test hypotheses like: *matrix multiplication is non-commutative*, it is often convenient to use random matrices. The ***Maple*** procedure `randmatrix` produces such matrices.

A number of options, such as `sparse`, `symmetric`, and `anti-symmetric` are available. Recall that an anti-symmetric matrix A satisfies the condition $A^t = -A$, and that most elements (let us say at least 80%) of a sparse matrix are zero.

```
> A := randmatrix(5,5); B := randmatrix(5,5);
> equal(A&*B - B&*A,matrix(5,5,0));
```

We may conclude that, generally speaking, the matrix products AB and BA do not coincide.

With `equal(A,B)` ***Maple*** checks that the two matrices A and B are equal.

```
> A := matrix(2,3,[1,2,3,4,5,6]);
> B := matrix(2,2,[1,-1,3,-1]);
> C := (2*(B^3 + inverse(B))&*A)/5:  evalm(C);
```

On clicking the right mouse button on one of the matrices, a menu of common matrix functions pops up. Note that for the square matrix B there are more procedures available than for the other two matrices.

7.1.2 Solving a System of Linear Equations

Next let's consider solving a system of linear equations by matrix computation. Choose a matrix of coefficients A and a constant vector b.

```
> A := matrix(3,3,[1,2,3,2,5,3,1,0,8]);
> b := vector([5,3,17]);
```

You can't be surprised that ***Maple*** will solve the system $Ax = b$. Use the command `linsolve` to do it.

```
> x := linsolve(A,b);
```

Now let's take a look at the reduction process on which Gaussian elimination is based. Recall that we first create the augmented matrix:

```
> A_b := augment(A,b);
```

Then we apply row reduction to this augmented matrix. First exchange the first and third rows, then subtract the first row twice from the second row, and after that subtract the first row from the third row.

```
> swaprow(%,1,3);
> addrow(%,1,2,-2);
> addrow(%,1,3,-1);
```

The ***Maple*** commands `swaprow` and `addrow` do what you expect them to do. Now, subtract $\frac{2}{5}$ times the second row from the third row, and after that multiply the last row by 5.

```
> addrow(%,2,3,-2/5);
> R := mulrow(%,3,5);
```

Now we have obtained the row reduced form of the augmented matrix used in the Gaussian elimination process to find the solution to a system of equations. Applying `backsub` (back substitution) to the row reduced matrix **R** gives the solution.

```
>  backsub(R); equal(backsub(R),x);
```

Finally, the reduced row-echelon form can be obtained directly from the augmented matrix by means of the ***Maple*** procedure `rref`, which is short for *reduced row-echelon form.*

```
>  evalm(rref(A_b));
```

7.2 Removing Noise from a Data Set

In the remainder of this ***Maple*** session, we are going to consider how the least squares method may be applied to the process of removing, or filtering, so-called noise from a given data set.

Consider the following problem. We are given a large set of N points $P_i = (t_i, b_i)$ ($i =$ 1,..., N) in the plane, which can be viewed as the set of (inexact) observations of an event (or function) f , so that b_i is approximately equal to $f(t_i)$ for i =1,..., N. Our objective is to find a polynomial of fixed degree n that best fits the observed values, in the sense that it does better than all other polynomials of the same degree. Such a polynomial will also give us good approximations (we hope) of other, unobserved values of the function f. Moreover - and this is crucial - we hope to be able to smooth down the irregularities caused by non accurate observations. Our choice of
n

depends on the overall pattern in which the points P_i are arranged, and we shall use the least squares approximation to obtain a best fit. Therefore, the problem we set out to solve is to compute the least squares solution to the linear system

$$Ax = b$$

where $A = (a_{ij})$ is the $N \times (n+1)$ matrix with elements $a_{ij} = {t_i}^{(j-1)}$ and b is a vector with components b_i.

Let as first consider a simple example. Suppose we have the use of 100 observations of the sine function on the interval [0,2 π]. These observations contain small errors, usually referred to as noise (think of the graph of a noisy electronic signal). The noisy character of the sine data can be simulated by adding to each sine value a random value between -0.2 and 0.2. We use ***Maple***'s random generator `rand` to generate the additional noise.

```
>  t := vector(100,i -> 2*Pi*i/100):
```

```
>  b := vector(100,i -> sin(t[i]) + 0.2*(rand(101)()-50)/100):
```

Now we shall try to retrieve the true sine curve from the noisy data by computing the best cubic polynomial fit in the least squares sense.

The noisy sine curve is shown in the next figure.

```
>  plot([seq([t[i],b[i]],i=1..100)]);
```

The ***Maple*** command `leastsqrs` solves the least squares problem. Of course, we could instead use `linsolve` to solve the system of normal equations. Like so:

```
> A := matrix(100,4,(i,j) -> t[i]^(j-1)):
> a := leastsqrs(A,b);
```

Maple's `leastsqrs` gives a vector output of polynomial coefficients.

```
> i := 'i':  p := t -> sum(a[i+1]*t^i,i=0..3);
> plot({sin(x),p(x)},x=0..2*Pi);
```

In this figure the sine function is plotted, superimposed by the graph of the least squares solution. Despite the rather sizeable noise in the sine values, the retrieved curve is not bad at all, at least it is a great improvement on its noisy counterpart. We realize of course that the example may seem rather unfair, knowing that the sine function behaves very much like a cubic polynomial on any interval of length 2 π anyway.

Now assume more realistically that our data do not permit good polynomial approximations of low degree. We then could try to filter out the noise by replacing each point $P_i = (t_i, b_i)$ of the data set by another point $(t_i, p_i(t_i))$ obtained by applying the cubic polynomial least squares technique to the original point and its closest neighbors, say five to the left, and five to the right. Here p_i is the least squares cubic polynomial associated with the point P_i ; observe that p_i changes from point to point. The first five and the last five data points of our set naturally need special treatment, as there are fewer neighboring points available.

Let us first generate a 200-point data set on the interval [0,1]. We use a strongly fluctuating data function to assist us.

```
> datafunction := x -> exp(-(10*x-2)^2) + exp(-2*(10*x-4)^2)
> + exp(-4*(10*x-6)^2) + exp(-8*(10*x-8)^2);
> p1 := plot(datafunction(x),x=0..1):  p1;
```

Next we generate the data set of function values with noise.

```
> N := 200:  argset := [seq(i/N,i=1..N)]: _seed := 123456:
> funcset := [seq(evalf(datafunction(i/N) +
> 0.2*(rand(N+1)() - N/2)/N),i=1..N)]:
> dataset := zip((a,b) -> [a,b],argset,funcset):
> plot(dataset);
```

Next to the graph of the data function, the noisy data is unmistakable.

We shall use the least squares technique described in detail above to smooth down the rough edges from `dataset`. The ***Maple*** code is given below.

```
> nfuncset := [ ]:  k := 'k':
> for k to 5 do nfuncset := [op(nfuncset),funcset[k]] od:
> for k from 6 to N-6 do
> A := matrix(11,4,(i,j) -> argset[k+i-6]^(j-1)):
> b := vector(11, i -> funcset[k+i-6]):
> a := leastsqrs(A,b):
> m := 'm':  p := t -> sum(a[m+1]*t^m,m=0..3):
```

```
> nfuncset := [op(nfuncset),evalf(p(argset[k]))]  od:
> for k from N-5 to N do
> nfuncset := [op(nfuncset),funcset[k]] od:
> ndataset := [zip((a,b) -> [a,b],argset,nfuncset)]:
```

The next figure shows the graph of the original data function superimposed by that of the new data set. Except for a few points at the beginning and at the end of the interval, and those at the function's extremes, the result is quite satisfactory.

```
> p2 := plot(ndataset,color=black):
> plots[display]({p1,p2});
```

The subpackage **fit** of Maple's **stats** package also contains a least squares procedure, called **leastsquare**. This procedure is very suitable for curve-fitting in the least squares sense. We might just as well have used this procedure for the application given above. However, wishing to avoid the black-box character of the latter, we chose for a more transparent approach.