

22. Array, Table, Last Name Eval, Function Call, Conversions

22.1 Array

Just like any programming language, Maple offers one and multi-dimensional arrays. One dimensional arrays are usually called vectors. Vectors can be created from lists, as follows:

```
[> v := Vector([1, a, a+b]);
[> LinearAlgebra[Norm](v);    # see what norm is most commonly used
```

Vectors are similar to lists, except that we can apply linear algebra routines to them.

```
[> A := array(0..1,1..2,[[a, b],[c, d]]);
```

The arguments of the `array` constructor are the ranges and a list of lists.

```
[> A[0,1]; A[1,1];
```

Suppose we now wish to know the determinant of A:

```
[> LinearAlgebra[Determinant](A);
```

This results in an error message, because we must first convert to a matrix:

```
[> M := convert(A,Matrix);
[> LinearAlgebra[Determinant](M);
```

Let us generate some more examples. The (i, j) -th entry of the Hilbert matrix is given by $1/(i + j - 1)$. Let us create a 4-by-4 Hilbert matrix.

Our first method is via a nested sequence:

```
[> ha := array(1..4,1..4,[seq([seq(1/(i+j-1),i=1..4)],j=1..4)]);
```

The second method uses an index function:

```
[> h := (i,j) -> 1/(i+j-1);    # index function
[> h(3,2);                    # defines (i,j)-th entry in Hilbert matrix
[> hm := Matrix(4,4,h);
```

22.2 Table

An `array` is actually a special case of a `Table`. We encountered tables as remember tables of procedures.

```
[> diff(exp(sin(a*x)),x);
[> T := op(4,eval(diff));
```

Tables are indexed by sequences:

```
[> indices(T);
```

Here are the corresponding entries:

```
[> entries(T);
```

Recall that with every call of `diff`, the table is consulted, and, if there is no index in the table matching the call for `diff`, then a new entry is added to the table. We can also add entries:

```
[> T[b*y,y] := a;
[> eval(T);
[> diff(b*y,y);
```

22.3 Last Name Evaluation

By default, a scalar variable is always evaluated in full. With composite data types we have to force evaluation.

We illustrate this with a rotation matrix. In the command below, it is very important that we type in `matrix` with a little `m`. The newer `Matrix` type will have a different effect in what follows.

```
[> R := matrix([[cos(alpha), -sin(alpha)], [sin(alpha), cos(alpha)]]);
```

This matrix expresses the rotation about an angle α . Suppose we wish to specify α to $\pi/4$:

```
[> alpha := Pi/4;
[> print(R);
```

Giving `alpha` a value has no effect on the matrix. Instead we have to force evaluation, mapping the `eval` command

```
[> T := map(eval,R);
```

Note that `T` and `R` are two different matrices:

```
[> print(T); print(R);
[> S := T;
[> alpha := Pi/3;
[> T := map(eval,R);
[> print(S);
[> eval(S,1);
[> eval(S,2);
```

When assigning from scalar data types, the right hand side is by default fully evaluated. This is not the case with composite data structures. The command `S := T` is equivalent to `S := eval(T,1)`. We call this *last name evaluation*, opposed to the full evaluation, invoked by `eval(T)`. To get the same effect as with scalar variables, we must copy:

```
[> S := copy(T);
[> T[2,1] := 0;
[> print(S); print(T);
```

22.4 Function Call

We can store data by function calls. For example, we have already seen the `RootOf` construction. Here we see how to store the polar representation of a point in the plane:

```
[> pt := [3,4];           # point with cartesian coordinates (3,4)
[> r := sqrt(pt[1]**2 + pt[2]**2);
[> angle := arctan(pt[2]/pt[1]);
[> polpt := polar(r,angle);
[> whattype(polpt);
```

We select the operands from a polar type with the `op` command:

```
[> r := op(1,polpt); alpha := op(2,polpt);
[> x := r*cos(alpha); y := r*sin(alpha);
```

Below we show how to extend the `convert` procedure for polar/cartesian types.

```
[> 'convert/polar' := proc(pt)
>   description 'conversion from cartesian to polar':
>   return polar(sqrt(pt[1]**2+pt[2]**2),arctan(pt[2]/pt[1]));
> end proc;
```

Be aware that the above convert procedure is not well defined for points with `pt[1] = 0`.

```
[> convert(pt,polar);
```

In similar fashion, we can extend the convert procedure to enable the conversion from polar to rectangular coordinates:

```
[> 'convert/cartesian' := proc(z)
>   description 'conversion from polar to cartesian':
>   local r,a;
>   r := op(1,z);
>   a := op(2,z);
>   return [r*cos(a),r*sin(a)];
> end proc;
[> convert(z,'cartesian');
```

22.5 Conversions between Composite Data Types

Converting a list to a set and back is a way to remove duplicates from a list. In this lecture we have seen how to convert lists into vectors and lists of lists into arrays and matrices. We can convert a matrix (using the matrix `R` from above) into a lists of lists as follows:

```
[> convert(R,'listlist');
```

22.6 Assignments

1. The (i, j) -th entry of the Pascal matrix is $\binom{i+j-2}{j-1}$ (`= binomial(i+j-2,j-1)`).

Give the Maple commands to create a Pascal matrix with four rows and six columns.

2. Create the 7-by-7 matrix M where $M_{ij} = 10^{-(i^2+j^2)}$.

Use the procedure `fnormal` to set small entries of M to zero.

3. Change the remember table of `diff` so that `diff(x^3,x)`; returns $3x^2dx$.

Execute first the command `diff(x^3,x)`; and then change the table to obtain the desired effect.

Give all relevant Maple commands to achieve this.

4. Define a convert operation on polynomials, so that `convert(p,'coeffvec')` returns the coefficient vector of p . Use the `coeffs` command and make sure the the result of the conversion is of type `Vector`.

To illustrate your convert works well, do `p := randpoly(x,dense,degree=7)`; followed by `convert(p,'coeffvec')`; then you should see the coefficients of p as one column vector with 8 entries.

5. Create a table `distance` which collects the distance in miles between major cities in the world. For example, the distance between Berlin and London is 574 miles, between New York and Berlin, the distance is 3961 miles, etc... If the table is created correctly, then we can query the table as follows:

```
[> distance[Berlin,London];
"574 miles"
```