[P]. Once the production of [X] becomes limited, [Y] rises. This production of [Y] causes a decline in [Z]. Eventually, enough [X] is produced to consume the remaining [Y] and the cycle begins again.

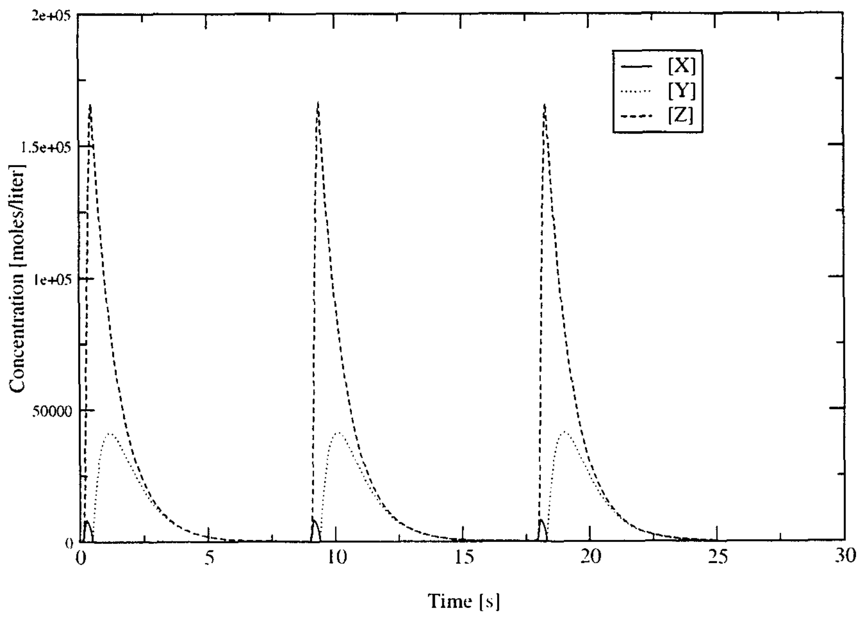

Figure 6.10. Oscillatory response from the Oregonator reaction.

6.5 LANGUAGE FUNDAMENTALS

6.5.1 Information hiding

In this chapter we have shown several uses of the `protected` keyword. If you have parameters or variables which you wish to hide from users of your model you can place them in a `protected` section. The obvious question is then, "Why would I want to hide things"?

The first reason is that internals of the model (*e.g.*, parameters and variables) contained within a `protected` section cannot be referenced externally. This allows the model developer the freedom to change some of the implementation details at some later time without fear of "breaking" any existing models that relied on the original model.

The second reason is that it is not necessary for users of the model to be aware of all of the internal details. By hiding the details of the model, the interface of the model (the publicly accessible portion) is simplified. This makes the model simpler and easier for others to use.

The drawback of making declarations `protected` is that external modifications are not possible. For example, consider a model with an internal variable whose solution is determined by a differential equation. The `start` attribute for the internal variable cannot be modified externally (*i.e.*, modifications can only be made by the model that contains the variable or by a derived model, as we saw in the `Body` model in Example 6.3). This can make it difficult to control the initial state of the entire system since that variable cannot be modified. Of course, putting the variable in a `protected` section is still a good idea if a change in implementation (*e.g.*, one that would eliminate the variable) is likely, since it prevents users of the model from relying on the presence of that variable.

6.5.2 Arrays

As shown in this chapter, arrays in Modelica can be useful in solving many kinds of problems. In addition to creating arrays of variables (as in Section 6.3.2), it is possible to declare arrays of connectors and subcomponents as well (as shown in Section 6.3.3). In this section we will review the functionality presented in this chapter and present additional details not covered by the examples.

6.5.2.1 Arrays of scalars

Arrays of scalars are the simplest example of array usage in Modelica. For example, declaring an array, `x`, of 5 `Real` variables is done as follows:

```
Real x[5];
```

In some cases we might wish to allow a `parameter` to govern the size of the array. In that case we would do something like:

```
parameter Integer x_size=5;
Real x[x_size];
```

In yet other cases, we might wish to leave the size of the array unspecified and let an initializer determine the size. In that case the array declaration would look something like:

```
model Beam
  parameter Real x[:];
  ...
end Beam;
```

Later, when an instance of a `Beam` is declared we can initialize `x` by writing:

```
Beam b(x={0.2, 0.77, 0.92});
```

Within the `Beam` model, if we wish to know how big the `x` array is, after the initialization, we can use the `size()` function as follows:

```
model Beam
  parameter Real x[:];
  ..
equation
  for i in 1:size(x,1) loop
    ...
  end for;
end Beam;
```

The first argument to the `size()` is the array we are interested in and the second argument indicates which dimension we are interested in. In this case, `x` only has one dimension.

Arrays can be initialized in several ways, as the following code fragment shows:

```
  parameter Real x[5]={0.1,0.3,0.5,0.7,0.9};
  parameter Real y[:]=x;
  parameter Real z[:]=0.1:0.2:0.9;
  parameter Integer evens[:]=2:2:10;
```

Array `x` is initialized directly from an explicit array. The size of `y` is left unspecified in the declaration but then the initialization establishes the size as 5 because the values are copied from `x`.

In the case of `z`, the array is constructed by starting with the number .1 and incrementing by .2 until the value exceeds .9 which means that `z` will have the same values as `y` and `x`. The array construction syntax also works in the same way with integers, as can be seen in the initialization of `evens` which creates the array `{2,4,6,8,10}`. In fact, this is the most common form of such constructions and is often used in conjunction with `for` loops. If no increment value is given (*i.e.*, there are only two numbers given with a semicolon in between), it is assumed that the increment is 1 for both `Integer` and `Real` cases.

An important point to make regarding array expressions is that there is no difference between:

```
x = {1, 2, 3, 4, 5}
```

and

```
x = 1:5
```

Likewise, there is no difference between:

```
for i in {1, 2, 3, 4, 5} loop
  ...
end for;
```

and

```
for i in 1:5 loop
  ...
end for;
```

Because the loop is performed over the elements of an array, loops can be constructed over non-contiguous or non-sequential indices, for example:

```
for i in {1, 3, 2, 5, 4, 7, 9, 8, 6} loop
  ...
end for;
```

6.5.2.2 Arrays and attributes

Although we have discussed how to declare and initialize an array of scalars, there is still the issue of how to initialize array attributes. For example, we can declare an array as:

```
Real x[5];
```

But, what if we would like to set the `start` attribute for each of these five elements? Just as `x` is an array, the `start` attribute is also an array.[8] Therefore, the `start` attribute could be initialized as follows:

```
Real x[5](start={0.1,0.2,0.3,0.4,0.5});
```

6.5.2.3 Arrays of components

As we have seen in several examples, the Modelica syntax allows us to declare arrays of components. Such arrays can be useful because they provide increased flexibility for applying constitutive equations to a large number of variables. In all of the examples shown with arrays of components, each component in the array was initialized using the same parameter value. This is often the case and easily accomplished. However, there are cases where it is useful to initialize each component in the array with a unique parameter value. Unfortunately, the Modelica language specification does not completely specify how this can be accomplished.[9]

6.5.2.4 Multi-dimensional arrays

Most of the examples contain arrays with only a single dimension. Such arrays are used primarily to represent mathematical vectors. Arrays with more than a single dimension (*e.g.*, representing matrices) are also possible. The `Reaction` model in Section 6.4.4 demonstrated how to create multi-

[8] These are the semantics in version 1.4 of the Modelica language semantics. However, there are some problems with this syntax and newer versions of the semantics may be slightly different.
[9] It is currently an issue being discussed by the Modelica Association.

dimensional arrays in Modelica. When declaring multi-dimensional arrays, each dimension of the array must be separated by a comma. For example:

```
Real x[5,2,7,8,12];
```

While most models use either 1, 2, or 3 dimensional arrays, there is no limit imposed by the Modelica language on the dimensionality of arrays.

In some cases, a `type` may define the dimensionality of an array. For example:

```
type Point=Real[3];
```

In such cases, an array of that type such as

```
Point particles[12];
```

creates an array with the same shape (*i.e.*, number of rows and columns) as:

```
Real particles[12,3];
```

Another issue with multi-dimensional arrays is initialization. To initialize a multi-dimensional array from a set of literal values, an array of the appropriate shape must be constructed. For example,

```
Real x[2,3] = {{1,2,3},{4,5,6}};
```

Note that the first index represents the "outer" array (*i.e.*, the rows of a two dimensional array) and the second index represents the "inner" array (*i.e.*, the columns of a two dimensional array). In this way, a three-dimensional array could be initialized as follows:

```
Real y[2,3,4] = {{{1,2,3,4},{5,6,7,8},{9,10,11,12}},
                 {{12,11,10,9},{8,7,6,5},{4,3,2,1}}};
```

As mentioned previously, arrays can be constructed by choosing an interval and an increment value. So, the following two initializations are equivalent:

```
Real x[2,3] = {{1,2,3},{4,5,6}};
Real z[2,3] = {1:3,4:6};
```

6.5.3 Looping and equations

In this chapter, we have seen how looping can be used to generate sets of equations. We discussed looping earlier in Section 5.7.5 but the focus then was on algorithms. In this section, we will focus on the special implications of using `for` within an `equation` section as opposed to an `algorithm` section. While `for` loops can be convenient in an `equation` section, they are not always necessary. For instance, as we can see in Example 6.3, it is not necessary to write explicit loops because implicit ones are generated when working with arrays.

The important thing to remember about looping in an `equation` section is that the statements contained within the loop are **equations**, not assignments. For example, the following code fragments have quite different meanings:

```
equation
  var = 0;
  for i in 1:4 loop
    var = var*x+i;
  end loop;

algorithm
  var := 0
  for i in 1:4 loop
    var := var*x+i;
  end loop;
```

When the `for` loop appears in an `equation` section, the following 5 equations are generated:

```
equation
  var = 0;
  var = var*x+1;
  var = var*x+2;
  var = var*x+3;
  var = var*x+4;
```

Note that these equations are not linearly independent (*i.e.*, they are singular). On the other hand, when the `for` loop occurs within an `algorithm` section it generates the following assignments:

```
algorithm
  var := 0;
  var := var*x+1;
  var := var*x+2;
  var := var*x+3;
  var := var*x+4;
```

the net effect of these assignments is equivalent to:

```
algorithm
  var := 5+x*(4+x*(3+x*(2+x*1)));
```

The fact that this assignment was carried out in five separate steps is no different than if it had been carried out in one.

6.5.4 Advanced array manipulation features

6.5.4.1 MATLAB compatibility

Although the examples in this chapter have focused on basic array manipulation techniques, Modelica also includes many advanced array manipulation features. Modelica shares many of the same features and, in general, the same syntax for array manipulation as MATLAB.[10]

6.5.4.2 Array construction and concatenation

For example, matrices can be created using the same syntax that is used in MATLAB, *i.e.*,

```
Real x[2,3] = [1,2,3;4,5,6];
```

The fact that the expressions are contained between the " [" and "] " characters indicates that this is a matrix construction. Within such matrix construction expressions, a " , " indicates the construction is proceeding to the next column (*i.e.*, the second dimension) and the " ; " indicates the construction is proceeding to the next row. In this way, matrices can be constructed by concatenating matrices, vectors and scalars.

6.5.4.3 Array subsets

We saw in the `CalcMultiplier` function, defined in Section 6.4.4.3, the following array shorthand:

```
m[products[:,2]] := m[products[:,2]]-products[:,1];
```

If the dimensions of each array at the location of the " : " are equal, then such equations represent relationships between subsets of matrices. For example, this equation could have been written more explicitly as:

```
for i in 1:size(products,1) loop
  m[products[i,2]] := m[products[i,2]]-products[i,1];
end for;
```

Similar types of equations can be written that specify specific elements. For example, the following is also equivalent to the previous two code fragments:

```
n = size(products,1);
m[products[1:n,2]] := m[products[1:n,2]]-products[1:n,1];
```

Remember that "`1:n`" expands to a vector containing every integer value between `1` and `n`, inclusively.

6.5.4.4 Vectorizing of functions

The semantics of Modelica are designed so that it is not necessary to create special vectorized forms of functions. Instead, the normal form of the function can still be used. For example:

[10]MATLAB is a registered trademark of The MathWorks, Inc.

```
sqrt({1, 2, 3});
```

is equivalent to:

```
{sqrt(1), sqrt(2), sqrt(3)};
```

In this way, even though `sqrt()` was defined to take a scalar argument, it can be applied element-wise to an array.

The general rule for taking advantage of this functionality is that the dimensionality of one or more of the arguments to a function can be given additional dimensions. However, all arguments that are given additional dimensions must have the same size in each additional dimension. For example, the following is legal:

```
mod({10,20,30},{4,5,6});
```

and yields:

```
{mod(10,4), mod(20,5), mod(30,6)};
```

Furthermore, this is also legal:

```
mod({10,20,30},4);
```

because only one argument was expanded and it is equivalent to:

```
{mod(10,4), mod(20,4), mod(30,4)};
```

On the other hand, this is not legal:

```
mod({10,20,30},{4,5});
```

because the additional dimensions are not the same size.

6.5.4.5 Mathematical operators

The mathematical operators such as "+" and "*" are frequently used with scalars, but can also be used with arrays. For example, the "+" and "-" operators can be used to add and subtract arrays that have the same size in each dimension. Furthermore, the "*" can be used with arrays in several ways.

The simplest example of using the "*" with arrays is the combination of multiplying a scalar by an array. Each element of the resulting array is equal to the product of the scalar and the corresponding element in the array being multiplied. Another example would be to use the "*" to take the inner product of two vectors of the same size. In other words, the following code fragment:

```
  Real u[5], v[5];
  Real s;
equation
  s = u*v;
```

is equivalent to:

```
    Real u[5], v[5];
    Real s;
  algorithm
    s := 0;
    for i in 1:size(u,1) loop
      s = s + u[i]*v[i];
    end for;
```

More complex examples are also possible. For example, the "*" can be used to represent the product of any two arrays as long as the sizes are mathematically compatible. For example, the following shorthand:

```
    Real A[5,7], u[5], v[7];
  equation
    u = A*v;
```

is equivalent to:

```
    Real A[5,7], u[5], v[7];
  algorithm
    for i in 1:5 loop
      u[i] := 0;
      for j in 1:7 loop
        u[i] = u[i] + A[i,j]*v[j];
      end for;
    end for;
```

Another example is that the following:

```
    Real A[5,7], u[5], v[7];
  equation
    v = u*A;
```

is equivalent to:

```
    Real A[5,7], u[5], v[7];
  algorithm
    for j in 1:7 loop
      v[j] := 0;
      for i in 1:5 loop
        v[j] = v[j] + u[i]*A[i,j];
      end for;
    end for;
```

Taking the matrix product of two matrices is also possible, as in:

```
    Real A[5,3], B[3,7];
    Real C[5,7];
  equation
    C = A*B;
```

which is equivalent to:

```
    Real A[5,3], B[3,7];
    Real C[5,7];
  algorithm
    C := fill(0,5,7);
    for i in 1:size(A,1) loop
      for j in 1:size(B,2) loop
        for k in 1:size(A,2) loop
          C[i,j] = C[i,j] + A[i,k]*B[k,j];
        end for;
      end for;
    end for;
```

One final trick that can be very useful (*e.g.*, in formulating transfer functions) is to compute an array containing:

$$\left\{x, \dot{x}, \ddot{x}, ..., \frac{d^n}{dt^n}x\right\} \tag{6.31}$$

We can do this with the following code fragment:

```
    Real x;
    Real dx[5];
  equation
    x = Modelica.Math.Sin(time);
    dx[1] = der(x);
    dx[2:5] = der(dx[1:4]);
```

In this way, we can construct a vector, dx, such that dx[i] represents the i^{th} derivative of x.

6.5.5 Built-in functions for arrays

Table 6.1 contains several of the built-in functions for manipulating arrays in Modelica. Full details of these functions (and others not described) can be found in the Modelica language specification.

6.6 PROBLEMS

PROBLEM 6.1 *Extend the* BinarySystem *model so that the total energy and momentum of the system is computed and make sure that it remains constant throughout the simulation.*

PROBLEM 6.2 *Using the material presented in Section 6.2, create a model of the solar system using the information provided in Table 6.2.*

PROBLEM 6.3 *Create a model to solve the hyperbolic PDE:*

$$\frac{d^2u}{dt^2} = c\frac{d^2u}{dx^2} \tag{6.32}$$

Function name	Purpose
`cross(x,y)`	Returns the cross product of the `x` and `y` vectors. The size of both vectors must be 3.
`diagonal(v)`	Generates a square matrix with the elements of `v` on the diagonal.
`fill(s,n1,n2,...)`	Generates an array of size `n1`×`n2`×`...` and fills it with the value `s`.
`identity(n)`	Returns an `nxn` identity matrix.
`linspace(x1,x2,n)`	Linearly interpolate `n` evenly spaced points along a line between `x1` and `x2`
`matrix(A)`	Similar to `vector(A)` except the size of two dimensions must be greater than 1.
`max(A)`	Returns the largest element of `A`.
`min(A)`	Returns the smallest element of `A`.
`ndims(A)`	Returns the number of dimensions `A` has.
`ones(n)`	Generates an array of length `n` and fills it with the value 1.0.
`outerProduct(v1,v2)`	Returns the outer product of `v1` and `v2`.
`product(A)`	Returns the product of all elements of `A`.
`scalar(A)`	Assuming `size(A,i)==1` for $1 \leq$ `i` $\leq$ `ndims(A)`, `scalar(A)` returns the single element of A.
`size(A)`	Returns a vector containing the size for each dimension of `A`.
`size(A,i)`	Returns the size of dimension `i` in array `A`.
`skew(x)`	Returns the 3x3 skew matrix for `x` where `size(x,1)==3`.
`sum(A)`	Returns the sum of all elements of `A`.
`symmetric(A)`	Returns a matrix where the upper triangular elements of `A` are copied to the lower triangular portion.
`transpose(A)`	Permutes the first two dimensions of `A`.
`vector(A)`	If `A` is a scalar, `vector(A)` returns a vector with `A` as the only element. If `A` is an array, it must have only one dimension with a size greater than 1 and that dimension is extracted as a vector.
`zeros(n)`	Generates an array of length `n` and fills it with the value 0.0.

Table 6.1. Built-in functions for arrays in Modelica.

Object	Mass (kg)	Distance from Sun (m)	Tangential Velocity (m/s)	Radius (m)
Sun	$1.989 \cdot 10^{30}$	0	0	$1.39 \cdot 10^{9}$
Mercury	$3.303 \cdot 10^{23}$	$69.82 \cdot 10^{9}$	$38.03 \cdot 10^{3}$	$4.879 \cdot 10^{6}$
Venus	$4.869 \cdot 10^{24}$	$108.92 \cdot 10^{9}$	$34.79 \cdot 10^{3}$	$12.10 \cdot 10^{6}$
Earth	$5.976 \cdot 10^{24}$	$152.10 \cdot 10^{9}$	$29.29 \cdot 10^{3}$	$12.74 \cdot 10^{6}$
Mars	$6.421 \cdot 10^{23}$	$249.2 \cdot 10^{9}$	$21.87 \cdot 10^{3}$	$6.780 \cdot 10^{6}$
Jupiter	$1.900 \cdot 10^{27}$	$816.4 \cdot 10^{9}$	$12.42 \cdot 10^{3}$	$139.8 \cdot 10^{6}$
Saturn	$5.688 \cdot 10^{26}$	$1.510 \cdot 10^{12}$	$9.11 \cdot 10^{3}$	$116.4 \cdot 10^{6}$
Uranus	$8.686 \cdot 10^{25}$	$3.001 \cdot 10^{12}$	$6.49 \cdot 10^{3}$	$50.72 \cdot 10^{6}$
Neptune	$1.024 \cdot 10^{26}$	$4.555 \cdot 10^{12}$	$5.38 \cdot 10^{3}$	$49.24 \cdot 10^{6}$
Pluto	$1.270 \cdot 10^{22}$	$7.358 \cdot 10^{12}$	$3.58 \cdot 10^{3}$	$2.390 \cdot 10^{6}$

Table 6.2. Solar system data.

You may choose to use the following spatial approximation:

$$\frac{d^2u}{dx^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} \qquad (6.33)$$

PROBLEM 6.4 *Create a model for a "collision force" model between two bodies such that when they are in contact (i.e., the total distance between the centers of the bodies is less than the sum of their radii) they generate a repelling force as follows:*

$$F = c(r - r_1 - r_2) \qquad (6.34)$$

where c is a large "stiffness" coefficient, r is the distance between the centers of the bodies, r_1 is the radius of body 1 and r_2 is the radius of body 2.

Next, create a "pool table" model with several billiard balls on it. Only one ball should have an initial velocity. Position the balls so that at least two collisions take place. Using Dymola, you can declare a `Sphere` *for each body so that the collisions can be animated (see Example 9.2 for an example).*