Polynomial Evaluation

Our first example will center around using functions to evaluate polynomials. This will help use understand the basics of defining and using functions.

Computing a Line

Mathematical Background

Before diving until polynomials of arbitrary order, let’s first consider how we could use a function to evaluate points on a line. Mathematically, what we’d like to define is a function that is applied as follows:

y(x,x0,y0,x1,y1)

where x is the independent variable, (x0,y0) is one point that defines the line and (x1,y1) is the other point that defines the line. Mathematically, such a function could be defined as follows:

y(x,x0,y0,x1,y1)=x(y1−y0)/(x1−x0)-(x1+x0)/(2(x1−x0))(y1−y0)/2+(y1+y0)/2

To reduce the number of arguments, let’s assume that combine x0 and y0 into a single point represented by the vector p0→ and we combine x1 and y1 into a single point represented by the vector p1→ so that the function is now invoked as:

Line(x,p0→,p1→)

Modelica Representation

The question now is how can we transform this mathematical relationship into a function that we can invoke from within a Modelica model. To do this, we must define a new Modelica function.

It turns out that a function definition is very similar (syntactically, at least) to a [*Model Definition*](http://book.xogeny.com/behavior/equations/model_def/#model-definition). Here is the definition of our Line function in Modelica:

**function** **Line** "Compute coordinates along a line"

**input** Real x "Independent variable";

**input** Real p0[2] "Coordinates for one point on the line";

**input** Real p1[2] "Coordinates for another point on the line";

**output** Real y "Value of y at the specified x";

**algorithm**

y := x\*(p1[2]-p0[2])/(p1[1]-p0[1])+

(p1[2]+p0[2]-(p1[1]+p0[1])\*(p1[2]-p0[2])/

(p1[1]-p0[1]))/2.0;

**end** **Line**;

All the arguments to the function are prefixed with the input qualifier. The result of the function has the output qualifier. The body of the function is an algorithm section. The value for the return value (y in this case) is computed by the algorithm section.

So in this case, the output value, y, is computed in terms of the input values x, p0 and p1. Note that there is no return statement in this function. Whatever the value of the output variable is at the conclusion of the algorithm section is automatically the value returned.

A couple of things to note that were discussed in previous chapters. First, note the descriptive strings on both the function itself and the arguments. These are very useful in documenting the purpose of the function and its arguments. Also note how the points use arrays to represent a two-dimensional vector and how those arrays are indexed in this example.

One troubling aspect of the Line model is the length of the expression used to compute y. It would be nice if we could break that expression down.

Intermediate Variables

In order to simplify the expression for y, we need to introduce some intermediate variables. We can already see that x, p0 and p1 are variables that we can use from within the function. We’d like to introduce additional variables, but they shouldn’t be arguments. Instead, their values should be computed “internally” to the function. To achieve this, we create a collection of variables that are protected. Such variables are assumed to be computed internally by the function. Here is an example that uses protected to declare and compute two internal variables:

**function** **LineWithProtected** "The Line function with protected variables"

**input** Real x "Independent variable";

**input** Real p0[2] "Coordinates for one point on the line";

**input** Real p1[2] "Coordinates for another point on the line";

**output** Real y "Value of y at the specified x";

**protected**

Real m = (p1[2]-p0[2])/(p1[1]-p0[1]) "Slope";

Real b = (p1[2]+p0[2]-m\*(p1[1]+p0[1]))/2.0 "Offset";

**algorithm**

y := m\*x+b;

**end** **LineWithProtected**;

This model introduces two new variables. One variable, m, represents the slope of the line and the other, b, represents the return value for the condition when x=0. Having computed these two intermediate variables, the expression to evaluate y becomes the more easily recognized form y := m\*x+b.

Computing a Polynomial

Mathematical Definition

Of course, our goal for this section is to create a function that can compute arbitrary polynomials. So now that we’ve seen a few basic functions, let us proceed with our ultimate goal. We will formulate a function that is invoked as follows:

p(x,c⃗ )

where x is again the independent variable and c⃗  is a vector of coefficients such that our polynomial is evaluated as:

p(x,c⃗ )=∑i=1…N(cix^(N−i))

where N is the number of coefficients passed to the function. There are two important things to note at this point. First, **the first element in** c⃗  **corresponds to the highest order term in the polynomial**. Second, we are using a notation that assumes that the elements in c⃗  are numbered **starting from 1** to make the transition to Modelica code (where arrays are indexed starting from 1) easier.

Note that the definition for p above is easy to read and understand. But when working with floating point numbers with finite precision, it is more efficient and more accurate to use a recursive approach for evaluating the polynomial. For a 4th order polynomial, the evaluation would be:

p(x,c⃗ )=((c1x+c2)x+c3)x+c4

This is more efficient because it relies on simple multiplication and addition operations and avoids performing exponentiation operations, which are more expensive, It is more accurate because exponentiation can easily trigger round-off or truncation errors in finite precision floating point representations.

Modelica Definition

Now that we’ve defined precisely what computations we want the function to perform, we are just left with the task of defining the function in Modelica. In this case, our polynomial evaluation function can be represented in Modelica as:

**function** **Polynomial** "Create a generic polynomial from coefficients"

**input** Real x "Independent variable";

**input** Real c[:] "Polynomial coefficients";

**output** Real y "Computed polynomial value";

**protected**

Integer n = size(c,1);

**algorithm**

y := c[1];

**for** i **in** 2:n **loop**

y := y\*x + c[i];

**end** **for**;

**end** **Polynomial**;

Again, all the arguments to the function have the input qualifier and the return value has the output qualifier. As with the previous example, we’ve defined an intermediate variable, n, as a convenient way to refer to the length of the coefficient vector. We also see how a for loop can be used to represent the recursive evaluation of our polynomial for any arbitrary order.

To verify that this function is working properly, let’s use it in a model. Consider the following Modelica model:

**model** **EvaluationTest1** "Model that evaluates a polynomial"

Real yf;

Real yp;

**equation**

yf = Polynomial(**time**, {1, -2, 2});

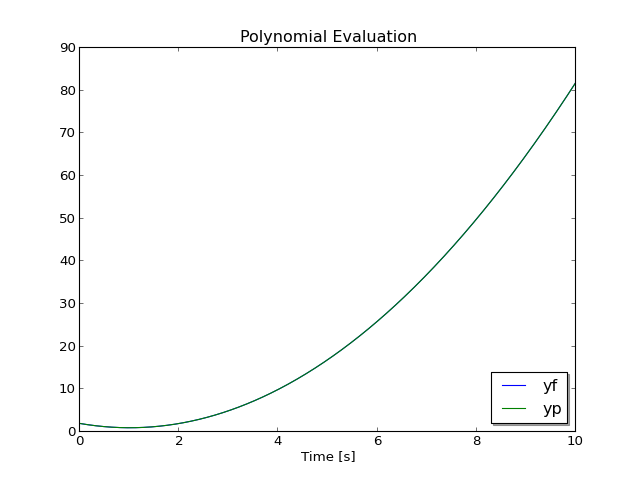
yp = **time**^2-2\***time**+2;

**end** **EvaluationTest1**;

Remember that the first element in c corresponds to the highest order term. If we compare a direct evaluation of the polynomial, yp, with one computed by our function, yf, we see they are identical:

([Source code](http://book.xogeny.com/plots/Eval1.py))

 Simulate EvaluationTest1 in your browser



Differentiation

It is completely plausible that this polynomial evaluation might be used to represent a quantity that was ultimately differentiated by the Modelica compiler. The following examples is admittedly contrived, but it demonstrates how such a polynomial might come to be differentiated in a model:

**model** **Differentiation1** "Model that differentiates a function"

Real yf;

Real yp;

Real d\_yf;

Real d\_yp;

**equation**

yf = Polynomial(**time**, {1, -2, 2});

yp = **time**^2-2\***time**+2;

d\_yf = der(yf); *// How to compute?*

d\_yp = der(yp);

**end** **Differentiation1**;

Here we have the same equations for yf, evaluated using Polynomial, and yp, evaluated directly as a polynomial. But we’ve added two additional variables,d\_yf and d\_yp representing the derivative of yf and yp, respectively. If we attempt to compile this model the compiler is very likely to throw an error related to the equation for d\_yf. The reason is that it has no way to compute the derivative of yf. This is because, unlike yp which is computed with a simple expression, we’ve hidden the details of how yf is computed behind the function Polynomial. In general, Modelica tools do not look at the implementations of functions to compute derivatives and, even if they did, determining the derivative of an arbitrary algorithm is not an easy thing to do.

So the next question is how can we deal with this situation? Won’t this make it difficult to use our functions within models? Fortunately, Modelica gives us a way to specify how to evaluate the derivative of a function. This is done by adding something called an annotation to the function definition.

**Annotations**

An annotation is a piece of metadata that doesn’t describe the behavior of the function directly (*i.e.,* it doesn’t affect the value the function returns). Instead, annotations are used by Modelica compilers to give them “hints” about how to deal with certain situations. Annotations are always “optional” information which means tools are not required to use the information when provided. The Modelica specification defines a number of standard annotations so that they are interpreted consistently across Modelica tools.

In this case, what we need is the derivative annotation because it will allow us to communicate information to the Modelica compiler on how to evaluate the derivative of our function. To do this, we define a new evaluation function, PolynomialWithDerivative, as follows:

**function** **PolynomialWithDerivative**

"Create a generic polynomial from coefficients (with derivative information)"

**input** Real x "Independent variable";

**input** Real c[:] "Polynomial coefficients";

**output** Real y "Computed polynomial value";

**protected**

Integer n = size(c,1);

**algorithm**

y := c[1];

**for** i **in** 2:n **loop**

y := y\*x + c[i];

**end** **for**;

**annotation**(derivative=PolynomialFirstDerivative);

**end** **PolynomialWithDerivative**;

Note that this function is identical except for the highlighted line. In other words, all we needed to do was add the line:

**annotation**(derivative=PolynomialFirstDerivative);

to our function in order to explain to the Modelica compiler how to evaluate the derivative of this function. What it indicates is that the functionPolynomialFirstDerivative should be used to evaluate the derivative of PolynomialWithDerivative.

Before discussing the implementation of the PolynomialFirstDerivative function, let’s first understand, mathematically, what is required. Recall our original definition of our polynomial interpolation function:

p(x,c⃗ )=∑i=1NcixN−i

Note that p takes two arguments. If we wish to differentiate p by some arbitrary variable z, we can use the chain rule to express the total derivative of pwith respect to z as:

dp(x,c⃗ )dz=∂p/∂x dx/dz+∂p/∂c⃗ dc⃗/dz

We can derive the following relations from our original definition of p. First, for the partial derivative of p with respect to x we get:

∂p/∂x=p(x,c′)

where c′ is defined as:

c′i=(N−i)ci

Second, for the partial derivative of p with respect to c⃗  we get:

∂p/∂ci=p(x,di→)

where the **vector** di→ is the ith column of an NxN identity matrix.

It turns out that for efficiency reasons, it is better for the Modelica compiler to give us dxdz and dc⃗ dz than to provide functions to evaluate ∂p∂x and ∂p∂ci. So, mathematically speaking, what the Modelica compiler needs is a new function that is invoked with the following arguments:

df(x,c⃗ ,dxdz,dc⃗ dz)

such that:

df(x,c⃗ ,dxdz,dc⃗ dz)=dfdz

For this reason, the derivative annotation should point to a function that takes the same arguments as df. In our case, that function,PolynomialFirstDerivative would be defined as follows:

**function** **PolynomialFirstDerivative**

"First derivative of the function Polynomial"

**input** Real x;

**input** Real c[:];

**input** Real x\_der;

**input** Real c\_der[size(c,1)];

**output** Real y\_der;

**protected**

Integer n = size(c,1);

Real c\_diff[n-1] = {(n-i)\*c[i] **for** i **in** 1:n-1};

**algorithm**

y\_der :=PolynomialWithDerivative(x, c\_diff)\*x\_der +

PolynomialWithDerivative(x, c\_der);

**end** **PolynomialFirstDerivative**;

Note how the arguments of our original function are repeated to create twice as many arguments (as we would expect). The second set of arguments represent the dxdz and dc⃗ dz quantities, respectively. Note that the assumption is that z is a scalar so the types of the input arguments are the same. Exploiting our knowledge about the partial derivatives of a polynomial, the calculation of the derivatives is done by leveraging our polynomial evaluation function.

We can exercise all of these functions using the following model:

**model** **Differentiation2** "Model that differentiates a function using derivative annotation"

Real yf;

Real yp;

Real d\_yf;

Real d\_yp;

**equation**

yf = PolynomialWithDerivative(**time**, {1, -2, 2});

yp = **time**^2-2\***time**+2;

d\_yf = der(yf);

d\_yp = der(yp);

**end** **Differentiation2**;

Simulating this model and comparing results, we see agreement between yf and yp as well as d\_yf and d\_yp:

([Source code](http://book.xogeny.com/plots/Diff2.py))

 Simulate Differentiation2 in your browser

