Chapter 11 Programming in Mathematica

Jennifer Voitle 2002

Contents

The power and versatility of Mathematica allow one to use many programming styles. I have found that a Mathematica program takes about one-tenth the time to write as the equivalent program in FORTRAN, BASIC or Pascal, and is probably one-tenth the size. Additionally, there is much more capability in Mathematica. The programming portion of a Master's thesis on using B-splines to approximate solutions to second-order partial differential equations was actually carried out in just six hours using Mathematica.. This feat would have been more difficult in FORTRAN as it involved setting up piecewise cubics on subintervals, multiplying, expanding, integrating and plotting as well as error analysis. On another occasion a program was written to solve the inviscid boundary layer equations in a single afternoon, including generation of plots and grid scaling. More recently, a complex Mathematica program for pricing financial derivatives including stochastic volatility and interest rates using the alternating implicit direction method was developed in only twenty hours.

One could probably get by doing all of one's programming in procedural style, but Mathematica has a lot of overhead, and you will doubtless find that your programs take longer in Mathematica than in other languages. To really take advantage of Mathematica's power, functional programming is the way to go.

In the following section, I introduce several examples and demonstrate how they might be written in both a procedural and a functional style in Mathematica. The examples will be compared, where possible, to FORTRAN and to BASIC. I shall lead up to some fairly complex problems encountered in industry, and show how Mathematica can be used to interact between other applications that you may have.

Introduction

Lists

Apply

Sequence

Optimization

Packages

Although Mathematica offers a very powerful and diverse programming language, I find that often it is not necessary to "program" at all in Mathematica: for example, in procedural languages one might need to write code (or call subroutine libraries) to compute factorials; sort lists; perform numerical quadrature, root-finding, matrix inversions or solution of systems of differential equations; perform statistical or Fourier analysis, or plot lists of data triplets. Yet these are but a few of the features in mathematica which can be carried out directly. If Mathematica does not have the function or procedure that one needs, it can easily be programmed. Of course, the best way to learn to program Mathematica is to do it. There are many excellent texts available. It is also instructive to call up Mathematica packages and see how they are written.

Let us begin with the procedural programming style. We then move to the functional style. Tips and engineering applications appear throughout the chapter. The focus is on the features that I have needed in my own application of Mathematica on industrial problems.

Lists

One of the fundamental structures of Mathematica is the list:lists provide an efficient structure for data storage. Many Mathematica functions either require lists as input or return them as output.Thus they should be mastered. We shall study lists through an application: performing linear regression on lists of data pairs.These data could be read in from a file, generated via Mathematica functions, or typed in directly.Here, we use some built-in Mathematica functions to generate our data.

Table

The Table command renders a table of data. To create a table (named data) of the first ten integers, we can type:

```
data = Table[i, {i, 1, 10}]
```

The output is:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

An alternative to Table is the Range function, which is not as powerful as Table but has its uses. To generate the integers using Range, we type

```
Range[10]
```

which produces the same answer as before:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Since this is a list, we can perform all manner of list operations on it. For example, the maximum and minimum of the list are:

```
{Max[data], Min[data]}
```

```
{10, 1}
```

The list can be sorted, reversed or rotated:

```
Sort[data]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
Reverse[data]
```

```
{10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

```
RotateLeft[data]
```

```
{2, 3, 4, 5, 6, 7, 8, 9, 10, 1}
```

We can raise each integer in the list to the power (1/integer), compute the factorials of the integers, add our list to the reciprocal of the list, or compute the ith prime numbers of the list:

```
data^(1/data)
data!
data + 1/data
Prime[data]
```

$$\{1, \sqrt{2}, 3^{1/3}, \sqrt{2}, 5^{1/5}, 6^{1/6}, 7^{1/7}, 2^{3/8}, 3^{2/9}, 10^{1/10}\}$$

$$\{1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800\}$$

$$\left\{2, \frac{5}{2}, \frac{10}{3}, \frac{17}{4}, \frac{26}{5}, \frac{37}{6}, \frac{50}{7}, \frac{65}{8}, \frac{82}{9}, \frac{101}{10}\right\}$$

$$\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$$

The number of elements in the list is obtained by evaluating the length of the list, as

```
Length[data]
```

```
10
```

There are a variety of functions that provide useful information about lists. For example, the frequency of occurrence of a certain pattern may be determined using Count; the common elements of a list are found with Union; and Complement gives the unique elements of a list. To combine two lists (with no sorting), use Join. Note that lists need not consist of similar elements.

```
list1 = {swap, MiniMe, 3 - I};
list2 = {Sqrt[a], Pi, sea*shell}
CombinedList = Join[list1, list2]
```

```
{√a, π, sea shell}
```

```
{swap, MiniMe, 3 - i, √a, π, sea shell}
```

```
Count[CombinedList, MiniMe]
```

```
1
```

```
Union[list1, list2]
```

```
{3 - i, √a, MiniMe, π, sea shell, swap}
```

The set of all elements in list1 that are not repeated in list2 are:

```
Complement[list1, list2]
```

```
{3 - i, MiniMe, swap}
```

Order matters with this function as with many others as can be seen by reversing the order of the arguments:

```
Complement[list2, list1]
```

```
{√a, π, sea shell}
```

Dot Products

Since lists are essentially vectors, vector operations may be applied to them. For example, to multiply two lists, use the dot operator:

```
{a, b, c} . {i, j, k}
```

```
a i + b j + c k
```

Checking Attributes

Mathematica objects, both intrinsic and user-defined, haave attributes which one can modify if need be. According to Skeel and Keiper, there are sixteen such attributes which include Listable and Protected. To see the attributes of an object, use the argument Attributes as follows:

```
Attributes[Log]
```

```
{Listable, NumericFunction, Protected}
```

This response means that we can wrap Log around a list and have it applied to every element of the list. This is why we were able to take the log of our list earlier. Most of Mathematica's functions such as Plus, Prime, Sin, Exp and the like are listable. The Protected attribute prevents the built-in function from being inadvertently overwritten (though you can do this on purpose, and may need to on occasion.) As an obscure example, consider the function FactorInteger, which returns the prime integer products of the integer argument. Thus, the factors of 119 are 7 and 17; while 41, while prime, factors as 1*41. FactorInteger returns no value (an empty list, or the null set) for unity.

```
FactorInteger[119]
```

```
{{7, 1}, {17, 1}}
```

```
FactorInteger[41]
```

```
{{41, 1}}
```

```
FactorInteger[1]
```

```
{}
```

Just for fun, and because we can, let us force FactorInteger to return the value of {{1,1}} for an input of 1. The attributes of the function are:

```
Attributes[FactorInteger]
```

```
{Listable, Protected}
```

So we will unprotect FactorInteger, execute our definition, and then protect it. The steps are:

```
Unprotect[FactorInteger]
```

```
{FactorInteger}
```

```
FactorInteger[1] = {{1, 1}};
```

```
Protect[FactorInteger]
```

```
{FactorInteger}
```

Test it:

```
FactorInteger[1]
```

```
{{1, 1}}
```

These attributes are not defined by default on one's own functions, however, as we can see:

```
Clear[f]
Attributes[f]
```

```
{}
```

If we apply f to a list, we get:

```
f[data]
```

```
f[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}]
```

We can force f to be applied to every element of the list by defining it to be Listable. Use the Attributes command or SetAttributes:

```
Attributes[f] = {Listable};
f[data]
```

```
{f[1], f[2], f[3], f[4], f[5], f[6], f[7], f[8], f[9], f[10]}
```

■ The Flat Attribute

For some functions, order matters. For example, the Plus operator is orderless while for Complement, division and most matrix operations order is important. Checking a few of Mathematica's functions:

```
Attributes[Plus]
```

```
{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
```

```
Attributes[Complement]
```

```
{Protected}
```

```
Attributes[Dot]
```

```
{Flat, OneIdentity, Protected}
```

What does Flat do?

```
Information["Flat", LongForm -> False]
```

```
Flat is an attribute that can be assigned to a symbol f to indicate
  that all expressions involving nested functions f should be
  flattened out. This property is accounted for in pattern matching.
```

We will have particular need of Flat very soon.

■ A Two-dimensional List

Let us simulate a dice roll which is a pair of random integers ranging from one to six. The function Random takes arguments specifying the type of value to return (Real, Integer or Complex), and the range over which to generate the number. Randome with no arguments returns a real value on [0,1]. Two numbers can be produced by Table:

```
Table[Random[Integer, {1, 6}], {2}]
```

```
{4, 4}
```

A series of ten such rolls is created by another Table:

```
TenRolls = Table[Table[Random[Integer, {1, 6}], {2}], {10}]
```

```
{{6, 2}, {3, 5}, {1, 4}, {3, 1}, {1, 5}, {4, 2}, {2, 4}, {2, 3}, {2, 6}, {1, 6}}
```

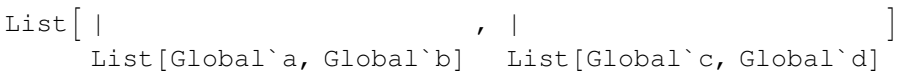
What if we wanted to compute the sum of each roll? That is, we want a function that, when given the input {{a,b},{c,d}} returns the sum of each pair {a+c,b+d}. We might think of applying Plus to the above table. However, if we do this, each pair is taken as an element and we obtain

```
Plus @@ TenRolls
```

```
{25, 38}
```

which is not what we want. If we look at the TreeForm (or FullForm) of the array {{a,b},{c,d}} we can get a clue to the solution of the problem:

```
TreeForm[{{a, b}, {c, d}}]
```



The depth of this expression is three, with the zeroth level the outer list. The first level is {{a,b},{c,d}}. We need to change the inner head from List to Plus. We can do this by specifying the level at which we wish to use Apply:

```
Information["Apply", LongForm -> False]
```

```
Apply[f, expr] or f @@ expr replaces
  the head of expr by f. Apply[f, expr, levelspec]
  replaces heads in parts of expr specified by levelspec.
```

This does what we want:

```
Apply[Plus, {{a, b}, {c, d}}, 1]
```

```
{a + b, c + d}
```

Alternatively, we can create our own function AddLists that takes two lists as input and returns the sum of each list. A first attempt:

```
Clear[AddLists]
AddLists[{x_List, y_List}] := x + y
```

```
AddLists[{{a, b}, {c, d}}]
```

```
{a + c, b + d}
```

Application: Integration by the Trapezoid Rule

To perform numerical integration,the intrinsic function Nintegrate may be used.However,the trapezoid method is illustrated as it is a good example of the improvement that can be made by using functional rather than procedural programming in Mathematica.The basic idea is:given an integrand over the interval[a,b],partition into n uniformly spaced subintervals of size

$$h = \frac{b-a}{n}$$

The function is evaluated at each point a+i*h,and the integral computed as the sum of the integrals over all subintervals.- Thus,the scheme is:

$$\int_a^b f(x)dx = \frac{h}{2}\left(f(a) + f(b) + 2\sum_{i=1}^{n-1} f(a+ih)\right)$$

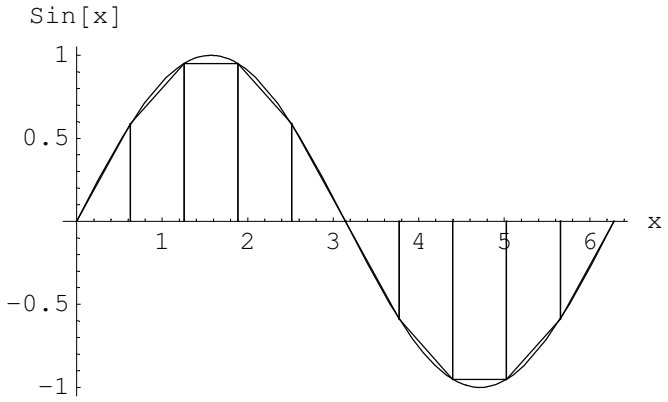
(Later, we shall treat non-uniformly spaced data.) We shall apply the rule to the sine function on [0,2π], using ten subintervals.

A plot of the composite trapezoid rule can be generated as follows:

In[12]:=

```
sinPlot = Plot[Sin[x], {x, 0, 2*Pi}, AxesLabel -> {"x", "Sin[x]"},
DisplayFunction -> Identity];
a = 0; b = 2*Pi; n = 10; h = (b - a)/n;
points = Partition[Flatten[Table[{{i, a}, {i, Sin[i]}, {i + b/n, Sin[i +
b/n]}, {i + b/n, 0}},
      {i, 0, ((n - 1)/n)*b, b/n}]], 2];
trapPlot = Show[Graphics[Line /@ Table[Take[points, {i, i + 1}], {i, 1,
Length[points] - 1}]],
    DisplayFunction -> Identity];
Show[sinPlot, trapPlot, DisplayFunction -> $DisplayFunction];
```

```
Show[- Graphics -, AspectRatio → 0.618,
 PlotRange → {{71, 321}, {0, 154.5}}, ImageSize → {250, 154.5}]
```



FORTRAN Program

As a comparison,here is how the same problem might look in FORTRAN:

```
f(x) = sin(x)
a = 0
b = 6.28319
sum = 0.
n = 10
h = (b-a)/n
Do 10 i = 1, n - 1
              sum = sum + 2*f(a+I*h)
10        continue
sum = sum + f(a) + f(b)
sum = h/2*sum
print *,'Integral approximated as: ',sum
end
```

Procedural Approach in Mathematica

The above code can be duplicated in Mathematical via constructs such as For, Do, Table and the like. We show Do here for a direct comparison with FORTRAN.

```
f[x_] = Sin[x];
a = 0;
b = 2 Pi;
sum = 0;
n = 10;
h = (b - a)/n;
Do[
  sum = sum+ 2 f[a + i h], {i,1,n-1}];
sum = sum + f[a] + f[b];
sum = h/2 sum;
```

```
Print["Integral approximated as : ",N[sum] ]
```

The main difference between Mathematical and FORTRAN here is the lack of a required End statement and Do label in the Mathematica code. Also, we were required to wrap N[] around the final result to obtain a numerical value. Unlike FORTRAN, Mathematica does not have default variable types: the variable int might as well be a string, real, complex or array as well as an integer. All of the above variables are assumed to be numeric and real except for i.

Functional Approach in Mathematica

The preceding calculation can be substantially improved if we take advantage of Mathematica's list structure. Actually, all that is really involved in the trapezoid rule is creating a list of function values,, summing them up and multiplying by the appropriate weighting factor: h if an interior point, h/2 if an end point. Then let us attack the problem from this viewpoint. First, create a list of function values using Table:

In[17]:=

```
Clear[f]
f[x_] = Sin[x];
a = 0; b = 2*Pi; n = 10; h = (b - a)/n;
DomainPoints = Table[i, {i, a, b, h}]
FunctionValues = f[DomainPoints]
```

Out[20]=

$$\{0, \frac{\text{Pi}}{5}, \frac{2\ \text{Pi}}{5}, \frac{3\ \text{Pi}}{5}, \frac{4\ \text{Pi}}{5}, \text{Pi}, \frac{6\ \text{Pi}}{5}, \frac{7\ \text{Pi}}{5}, \frac{8\ \text{Pi}}{5}, \frac{9\ \text{Pi}}{5}, 2\ \text{Pi}\}$$

Out[21]=

$$\{0, \frac{\text{Sqrt}[\frac{5 - \text{Sqrt}[5]}{2}]}{2}, \frac{\text{Sqrt}[\frac{5 + \text{Sqrt}[5]}{2}]}{2}, \frac{\text{Sqrt}[\frac{5 + \text{Sqrt}[5]}{2}]}{2}, \frac{\text{Sqrt}[\frac{5 - \text{Sqrt}[5]}{2}]}{2}$$

Now, we'll sum these values using the appropriate weighting factors. One way to accomplish this is to create a list of weighting factors {1,2,2, … ,2,2,1} and use the vector product to multiply. This list has to be the same length as f:

In[22]:=

```
myWts = Table[2, {i, 2, Length[FunctionValues] - 1}];
PrependTo[myWts, 1];
AppendTo[myWts, 1]
```

Out[24]=

```
{1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1}
```

The integral is then

In[25]:=
```
N[(h/2)*Plus @@ (myWts . FunctionValues)]
```

Out[25]=
```
0.
```

In[26]:=
```
Integrate[Sin[x], {x, 0, 2*Pi}]
```

Out[26]=
```
0
```

The above statements may be put together into a procedure which takes as its input the list of function values and returns the computed sum.

In[27]:=
```
TrapezoidRule[f_Symbol, {a_, b_}, (n_Integer)?Positive] :=
  Module[{h = (b - a)/n, FunctionValues = Table[f[i], {i, a, b, h}]},
   N[(h/2)*(f[a] + f[b]) + Take[FunctionValues, {2, -2}] /. {x___, y___}
-> x + y]]
```

As an example of how to call the function, we will integrate Sin[x] from 0 to 2Pi using n = 10:

In[28]:=
```
TrapezoidRule[Sin, {0, 2*Pi}, 10]
```

Out[28]=
```
0.
```

The time to carry out the integration is obtained by wrapping Timing[] around our function:

```
Timing[TrapezoidRule[Sin, {0, 2*Pi}, 10]]
```

```
{0. Second, 0.}
```

Non-Uniformly Spaced Data

As an application, suppose that pressures and volumes are given for a substance undergoing a compression from state 1 (superheated) to state 2 (saturated vapor). At T = 400 F, the following data are given:

v, ft3/lbm	P, psia
4.934	100
4.079	120
3.466	140

3.007	160
2.648	180
2.361	200
1.866	247.1

We define this in Mathematica and plot:

In[29]:=

```
volumes = {4.934, 4.079, 3.466, 3.007, 2.648, 2.361, 1.866};
pressures = Table[k, {k, 100, 200, 20}];
AppendTo[pressures, 247.1];
pVData = MapThread[List, {volumes, pressures}]
```

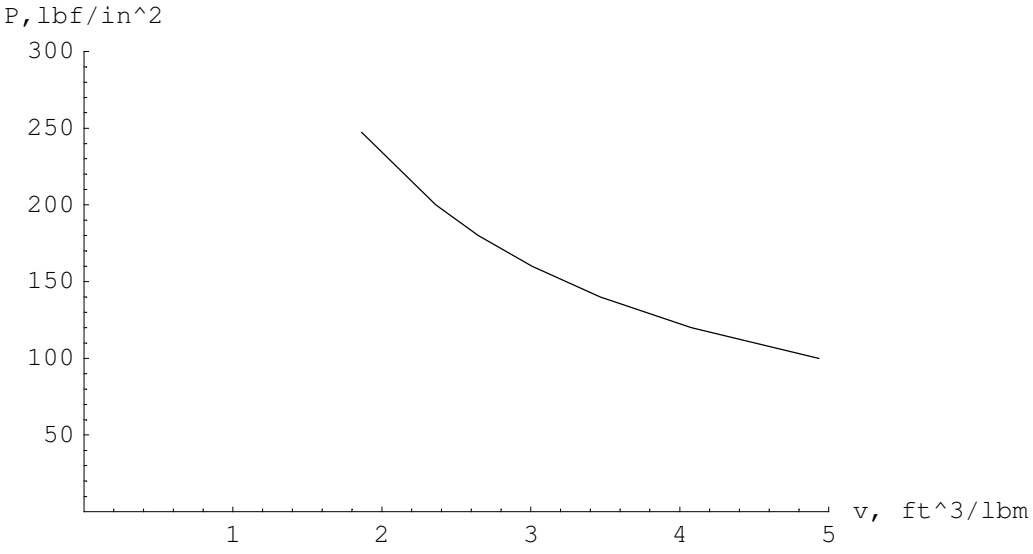
Out[32]=

```
{{4.934, 100}, {4.079, 120}, {3.466, 140}, {3.007, 160}, {2.648, 180}, {2.361,
```

In[33]:=

```
ListPlot[pVData, PlotRange -> {{0, 5}, {0, 300}}, AxesLabel -> {"v,
ft^3/lbm", "P,lbf/in^2"},
   PlotJoined -> True];
```

```
Show[- Graphics -, AspectRatio → 0.618,
 PlotRange → {{71, 321}, {0, 154.5}}, ImageSize → {250, 154.5}]
```



A generalization of the trapezoid plotting code we showed earlier is shown by the following procedure, which allows input in the form of a list.