THE PHYSICAL PENDULUM

Solomon Douglas and David Harrison
Department of Physics
60 St. George St.
University of Toronto
Toronto, Ontario, Canada M5S 1A7

Electronic: harrison@faraday.physics.utoronto.ca

This section is intended as reference for students in PHY351F – Advanced Classical Mechanics and/or PHY307F – Computational Physics, although others may find it interesting and/or useful. Students in these courses will be using MMA for required work, for which the following information is intended to provide needed information.

I. Some useful commands

Because `Pendulum` returns a possibly enormous list, it is important to be able to suppress the output. If you are not interested in seeing the output from a given *MMA* input, finish the input line with a semicolon. (Refer to § 1.4.7 of the **Mathematica** book.)

Since `Pendulum` returns a list of ordered triples, each consisting of q, p, and E, you will usually want to extract either q and p or E in order to manipulate and graph them. This can easily be done with *pure functions* and *mapping.* The syntax looks complicated, but it is worth learning because of its usefulness and versatility. An introduction to these two concepts follows.

I.A Mapping

First let us look at the idea of mapping. There are many functions in *MMA;* `IntegerQ` is an example. If you want to know whether or not `x` is an integer, you can type `IntegerQ[x]` and the result will be `True` if it is, and `False` if it isn't. What if you want to know whether each member of a list is an integer? For example [1]:

```
In[1]:= a = {1, 2, 3, 4, 4.5, 19/4, 5}
```

1. As usual, the *In[1]:=* prompt, and all other text that is not in **bold** type, are *MMA* output; don't type them in.

```
                         19
Out[1]= {1, 2, 3, 4, 4.5, --, 5}
                         4
```

Now the variable `a` contains a list of seven numbers, some of which are integers and some of which are not. If we simply say

```
In[2]:= IntegerQ[a]

Out[2]= False
```

The result was False because `a` is not an integer, it is a list. The problem is that we tested the entire list rather than testing each element. The solution is to use the *MMA* mapping operator. The notation *`f /@ expr`* applies the function *`f`* to each element in the first level of the list *`expr`* and evaluates to a list of the results. Thus:

```
In[3]:= IntegerQ /@ a

Out[3]= {True, True, True, True, False, False, True}
```

So, *`f /@ expr`* (or `Map[`*`f, expr`*`]`) returns a list of the return values of *`f,`* as applied to each element of *`expr.`* Note that square brackets (`[]`) for the call to the function `IntegerQ` are *not* used when the `/@` operator is used. They are two different ways of calling the function; if you use the mapping operator you do not need the brackets. For more information about `Map,` refer to § 2.2.4 of the **Mathematica** book.

I.B Pure Functions

Now let us examine the idea of nameless, or "pure" functions. A pure function is a function that, unlike `IntegerQ` or `Sin,` has no name and is simply defined "on the fly". This is useful if you want to use the function once and don't want to bother defining an explicit name for it.

You could define a function[2] called `f` like so:

```
In[4]:= f[x_] := (x+1)^2
```

It could then be used just as any other function in *MMA*; for example, `f[5]` would give `36.` It can also, of course, be used with `Map` the same way `IntegerQ` was used above (note again the lack of square brackets):

```
In[5]:= f /@ {1, 2, 3, foo, I}
```

2. If you don't know about defining functions in *MMA*, cf. § 1.7.1 of the book.

```
Out[5]= {4, 9, 16, (1 + foo)^2, 2 I}
```

This produces the expected result: a list of the return values of `f` as applied to the list `{1, 2, 3, foo, I}`. If we are going to use the function `f` again and again, defining a function like that is probably the best way to do it. However, what if we only want to perform that operation once or twice? That is what anonymous functions are for. An anonymous function, or a pure function, consists of the body of the function followed by an ampersand. The formal parameter to the function (if there is only one) is represented by a # symbol. For instance, `(#+1)^2 &` is the pure function equivalent to `f` as defined above, and it can be used in all the same places that the name `f` can be used. One such place is, of course, the `/@` operator; it would be used like this:

```
In[6]:= (#+1)^2 & /@ {1, 2, 3, foo, I}

Out[6]= {4, 9, 16, (1 + foo)^2, 2 I}
```

As expected, this produced the same output as the version using `f`. For more details on pure functions, cf. § 2.2.5 of the **Mathematica** book.

So, how will `Map` and `Function` be useful for use with `Pendulum`? As was mentioned earlier, `Pendulum` returns a list of triples `{q, p, E}`. If you want to extract the energy, you can use an anonymous function and a mapping to accomplish this: knowing that `Last[`*`list`*`]` means the last element in *`list`* (§ 1.8.4), then if `a` contains the list of triples, it makes sense that `Last[#]& /@ a` should be the list of energy values. That expression means "a list of the last element in each sublist in the list `a`". A similar strategy can be employed to extract the first and second elements of each sublist.

I.C First session

This complete *MMA* session illustrates the above concepts. Everything in bold text is meant to be typed in by you; everything else will be output by *MMA*. Make sure to type in everything exactly as you see it. Especially be careful that you don't miss any semicolons because they are what prevent a list of 1600 elements from scrolling by on the screen. (1600 because 800 points were requested and each point is returned twice, with the second one displaced in the `q` direction by a distance of 2π.) Also, case is significant, so enter all lowercase and uppercase letters exactly as you see them.

Note that when you call the `Pendulum` function, it prints a line onto the screen telling you what parameters the C function was invoked with; this is a diagnostic feature which is part of the C program and has nothing to do with *MMA*.

The `Pendulum` function invokes the C program, which by default implements a fourth-order Runge-Kutta integration routine. For information about this method of integration, refer to § 15.1 of *Numerical Recipes in C* by William H. Press et al; this book is available from the II Year Lab library . The C program is interfaced to the *MMA* function by means of a system supplied with *MMA* called *MathLink* . To learn about *MathLink,* read the technical report "*MathLink* ™ External Communication in *Mathematica* ®".

As you type in each input line, make sure you understand everything you are doing (especially on the lines with seemingly cryptic syntax) so that you will be able to do it on your own later .

```
Mathematica 2.2 for HP 9000 RISC
Copyright 1988-92 Wolfram Research, Inc.
 -- Motif graphics initialized --

In[1]:= Install["pendulum"]

Out[1]= LinkObject[pendulum, 1, 1]

In[2]:= ?Pendulum
Pendulum[{q0, p0}, {pts, dt}, double] generates a
   list of {q, p, E} triples.  The initial position
   and momentum are denoted by "q0" and "p0"; the
   total number of points to return is given by
   "pts"; "dt" is the timestep size.  If you want
   to return every point twice (in order to plot a
   phase portrait with -2Pi < q < 2Pi instead of
   -Pi < q < Pi) then "double" should be True
   (this is the default).

In[2]:= a = Pendulum[{Pi/2, 0}, {800, 0.02}];
pendulum(1.5708, 0, 800, 0.02, 1);

In[3]:= energy = Last[#]& /@ a;

In[4]:= ListPlot[energy - 1]

Out[4]= -Graphics-

In[5]:= points = Take[#, 2]& /@ a;

In[6]:= ListPlot[points]

Out[6]= -Graphics-

In[7]:= end
```

Some points to note about the above *MMA* session:

- The call to `Pendulum` specifies that the initial angle should be $\pi/2$ and the initial momentum should be 0; also that 800 points should be calculated and that each timestep should be 0.02. Note that doubling is implicitly requested by the absence of a third parameter.

- On line `In[2]` the list returned by `Pendulum` is stored in the variable `a`. The assignment on line `In[3]` specifies that the last element (i.e. the energy value) in each sublist in `a` is to be stored in a new list, which is called `energy`. The way the third element is extracted is as described above, using the pure function `Last[#]&` and the `/@` operator.
- Since we know that the correct energy value should be 1 (based on the initial condition), when we plot the energy (on line `In[4]`) we subtract 1 in order to see the differences from the expected value.
- The `{q, p}` pairs are stored in a list called `points` on line `In[5]`. This is done with the `Take` command: `Take[`*`list`*`, `*`n`*`]` returns a list of the first *n* elements in *list*. See § 1.8.4.
- `ListPlot` can plot either one- or two-dimensional lists. If it is a one-dimensional list (for example, `energy`) the x-axis of the plot is simply the index into the list of each element plotted. In the case of a two-dimensional list (such as `points`) one axis is used for each dimension.

I.D Some more useful commands

Now you know how to produce a single phase plot. What if you want to construct an entire phase portrait, consisting of several individual plots? One way would be to do each one individually, using the method shown above. Then you could show all the plots on one set of axes with the `Show` command: If the graphics you want to combine were produced from input lines 4, 5, 7 and 8, for instance, you could type `Show[%4, %5, %7, %8]` and all four graphs would be combined into one[3]. For more information about `Show` you are invited to read § 1.9.4 of the book *Mathematica*.

Another way of doing this would be to use some sort of iterative construct to call `Pendulum` repeatedly, with different parameters each time, then either plot each result and then combine the plots afterwards, or save the results in one big array and plot that. It would be a useful exercise to try each of these methods to make sure you know how they would be implemented.

3. Using the percent sign like this is described in § 1.2.1 of the **Mathematica** book.

I.E The *MMA* `Table` function

The iterative construct we will use is called `Table`. `Table`'s first argument is an expression that is to be evaluated again and again. The rest of its arguments are called iterators. For information about iterators, read § A.3.7 of the **Mathematica** book. `Table` returns a list of the values of its first expression with occurrences of the iterator variables in the expression replaced successively by each of their possible values. You should read § 1.8.2 of the **Mathematica** book attentively to further understand the operation of `Table`.

I.F The *MMA* `Flatten` function

Since `Table` returns a list with one dimension for each iterator , and we are iterating on the initial angles *and* the initial momenta, we will end up with a two-dimensional list (really a list of lists) of time series (and remember that each timeseries is itself a list of 3-element lists.) So, we will have a list of lists of lists of lists of numbers. We would like to reduce this to one list which contains all the points in all the timeseries, so we would like in effect to "peel off" the two outer sets of curly braces. There exists a *MMA* function for exactly this purpose, and it is called `Flatten`. `Flatten[`*`list`*`, `*`n`*`]` flattens out the outer *`n`* levels in *`list`*. See § 1.8.12.

II. Second session

This short *MMA* session simply demonstrates the use of `Table` and `Flatten` to combine several time series. It is especially important not to omit the terminal semicolons because the list returned by `Table` in this case is very very long.

```
Mathematica 2.2 for HP 9000 RISC
Copyright 1988-92 Wolfram Research, Inc.
 -- Motif graphics initialized

In[1]:= Install["pendulum"]

Out[1]= LinkObject[pendulum, 1, 1]

In[2]:= a = Table[Pendulum[{q0, p0}, {1500, 0.02}],
        {q0, 0, Pi, Pi}, {p0, -2, 2, 2/3}];
pendulum(0, -2, 1500, 0.02, 1);
pendulum(0, -1.33333, 1500, 0.02, 1);
pendulum(0, -0.666667, 1500, 0.02, 1);
pendulum(0, 0, 1500, 0.02, 1);
pendulum(0, 0.666667, 1500, 0.02, 1);
pendulum(0, 1.33333, 1500, 0.02, 1);
pendulum(0, 2, 1500, 0.02, 1);
pendulum(3.14159, -2, 1500, 0.02, 1);
pendulum(3.14159, -1.33333, 1500, 0.02, 1);
```

```
pendulum(3.14159, -0.666667, 1500, 0.02, 1);
pendulum(3.14159, 0, 1500, 0.02, 1);
pendulum(3.14159, 0.666667, 1500, 0.02, 1);
pendulum(3.14159, 1.33333, 1500, 0.02, 1);
pendulum(3.14159, 2, 1500, 0.02, 1);

In[3]:= b = Flatten[a, 2];

In[4]:= points = Take[#, 2]& /@ b;

In[5]:= ListPlot[points]

Out[5]= -Graphics-

In[6]:= end
```

Some points to note about the second *MMA* session:

- The call to `Pendulum` this time was the same as in the first session except that the initial position was specified with the variables `q0` and `p0` (any names could have been used), and that the number of points should be 1500 instead of 800. The former change was obviously so that it could be called with different initial conditions depending on where execution has reached in the loop. The latter is because some of the plots generated need more than 800 points to cover a whole period of the pendulum's motion. Trial and error was used to determine a good limit. (It seems inefficient to do more than twice the number of steps necessary for one initial condition in order to get the right amount for another initial condition; this is true, however with the implementation of `Pendulum` being used, that is practically the only choice. You are welcome to modify it in order to allow another parameter to the function that specifies a sampling period, so that, for instance, 500 points would be returned each time but they would consist of every 15th point calculated.)

- The second and third parameters to `Table` are the iterators. The first iterator , `{q0, 0, Pi, Pi}`, specifies that the variable `q0` (we must use this name since it is the one we chose in the body of the loop) should start at 0 and loop to π in steps of π. (That is, it should take on successively the values 0 and π.) The second iterator , `{p0, -2, 2, 2/3}`, states that the variable `p0` should loop from -2 to 2 in steps of 2/3. This is verified by the diagnostic information printed by the C program.

- On line *`In[3]`*, the `Flatten` function is used as described above to flatten out the two dimensions of the list returned by `Table` into one long list of points.

- The points are plotted in the same manner as in the first sample *MMA* session.