Introduction & Programming

Lecture 1

Physics 200
Laboratory

Monday, January 31st, 2011

A programming language useful to this course must provide a minimal set of components that can be used to combine numbers (computational tools), compare quantities and act on the result of that comparison (conditional control), repeat operations until a condition is met (loops), and contain functions that we can use to input data, and output results (i/o). Almost any language will suffice, but I have chosen to use `Mathematica`'s programming environment as the vehicle for this course. The reasoning is that 1. The input/output functions of `Mathematica` are easy to use, and require little additional preparation[1] 2. We will be focused on the ideas, numerical and otherwise, associated with the methods we study, and I want to draw a clear distinction between those ideas and issues of implementation. This course is not meant to teach you everything you need to know about programming – we will discuss only the bare essentials needed to implement the methods from class. Instead, we will focus on the physical motivation and tools of analysis for a variety of techniques. My hope is that the use of `Mathematica` allows us to discuss implementation in a homogeneous way, and our restriction to the basic programming structure of `Mathematica` (as opposed to the higher level functionality) allows for easy porting to the language of your choice.

Here, we will review the basic operations, rendered in `Mathematica`, falling under each category – we'll look at the arithmetic operations, comparisons, loops and i/o. In addition, we must be able to use variable names that can be assigned values, and there is a scoping for these constructions in `Mathematica` similar to C. Functions, in the sense of C, exist in `Mathematica`, and we will use a particular (safe) form, although depending

[1] There are libraries to import audio and video, for example, in C++, but the resulting internal representation can be more difficult to work with. In addition, the linking of libraries is a machine-specific detail of the programming that I do not want to address.

on the context, there are faster (and slower) ways to generate functions. In this class, we will bundle almost every set of computations into a function, and this is to mimic good coding practice that is enforced in more traditional languages (for a reason – the logic and readability one gains by breaking calculations up into named constituents cannot be overvalued). Finally, we will look at two important ideas for algorithm development: Recursion and Function pointers. Both are supported in `Mathematica`, and these are also available in almost any useful programming language.

1.1 Arithmetic Operations

All of the basic arithmetic operations are in `Mathematica` – we can add and subtract, multiply, divide, even evaluate trigonometric functions. The only occasional hiccup we will encounter is the distinction, made in `Mathematica`, between an exact quantity and a number – in Figure 1.1, we can see that when presented with `1/4`, `Mathematica` responds by leaving the ratio alone, since it is already reduced. What we are most interested in is the actual numerical value. In order to force `Mathematica` to provide real numbers, we can wrap expressions in the `N` function, as in `In[5]`, or we can specify decimal places, as in `In[6]` of Figure 1.

`Mathematica` is aware of most mathematical constants, like π and e, and when necessary, I'll tell you the name of any specific constant of interest. Finally, all angles used in trigonometric functions (and returned by arc-trig functions) are in radians.

1.2 Comparison Operations

We will use most comparison operations, and `Mathematica` outputs `True` or `False` to any of the common ones – we can determine whether a number is greater than, less than, or equal to another number using `>`, `<`, `==` (notice that equality requires a double equals sign to distinguish it from assignment). We denote "less than or equal to" with `<=`, and similarly for "greater than or equal to" (`>=`). The logical "not" operation is denoted `!`, so that inequality is tested using `!=` (not equal). Finally, we can string together the `True`/`False` output with "AND" (denoted `&&`) and "OR" (`||`). Some examples are shown on the right in Figure 1.1.

Arithmetic Operations		Logical Operations	
In[1]:=	3 + 5	In[1]:=	3 < 5
Out[1]=	8	Out[1]=	True
In[2]:=	4 * 20	In[2]:=	7 ≥ 5
Out[2]=	80	Out[2]=	True
In[3]:=	N[Pi / E]	In[3]:=	1 < 2 && 2 > 3
Out[3]=	1.15573	Out[3]=	False
In[4]:=	Sin[.1]	In[4]:=	1 < 2 || 2 > 3
Out[4]=	0.0998334	Out[4]=	True
In[5]:=	1 / 4	In[5]:=	3 == 3
Out[5]=	$\frac{1}{4}$	Out[5]=	True
In[6]:=	1.0 / 4.0	In[6]:=	2 ≠ 3
Out[6]=	0.25	Out[6]=	True

Figure 1.1: Examples of basic arithmetic input and output and logical operations.

1.3 Variables

Unlike C++, variables in `Mathematica` can be instantiated by definition, and do not require explicit typedef-ing. So setting a variable is as easy as typing `x = 5.0`. The variable has this value (subject to scoping) until it is changed, or "cleared" (closest to `delete` that exists in `Mathematica`) by typing `x = .`

Variables, for us, will be "doubles" by default, and always take numerical values. We can define tables and arrays, again by giving values to a variable name. In Figure 1.2, we see a few different ways of defining variables – first we define and set the variable `p` to have value 5 – `Mathematica` will print an output in general, and in the case of defining variables, it prints an output that reminds us of the variable's value. To suppress printing output, we use a semicolon at the end of a line – in the second example on the left in Figure 1.2, we define `q` to have value 7, and the semicolon tells `Mathematica` to just set the value without extra verbiage. As a check, if we input `q`, and allow output, we get the correct value.

Defining Variables

```
In[1]:= p = 5
Out[1]= 5

In[2]:= p
Out[2]= 5

In[3]:= q = 7;

In[4]:= x = {1.0, 2.0, 3.0, 4.0}
Out[4]= {1., 2., 3., 4.}

In[5]:= x
Out[5]= {1., 2., 3., 4.}

In[6]:= y = Table[j, {j, 1.0, 8.0, 1.0}]
Out[6]= {1., 2., 3., 4., 5., 6., 7., 8.}

In[7]:= y
Out[7]= {1., 2., 3., 4., 5., 6., 7., 8.}
```

Setting Variables

```
In[1]:= q = 9;

In[2]:= q
Out[2]= 9

In[3]:= q = 10;

In[4]:= q
Out[4]= 10

In[5]:= x = Table[j^2, {j, 1.0, 10.0, 2.0}]
Out[5]= {1., 9., 25., 49., 81.}

In[6]:= x[[2]]
Out[6]= 9.

In[7]:= x[[2]] = 3.0
Out[7]= 3.

In[8]:= x
Out[8]= {1., 3., 25., 49., 81.}
```

Figure 1.2: Examples of defining and setting variable values.

We can define tables of fixed length by specifying the appropriate values for each entry, using {...}, as in the definition of `x` on the left in Figure 1.2.

The Mathematica command Table[f[j],{j,start,end,step}] can also be used to generate tables that have values related to index number by the function f[j] – in the definition of the array variable y, we use f[j] = j for *iterator* j.

Once a variable or table has been defined by giving it a value, the value can be accessed (by typing the name of the variable as input) or changed (using the operator =) as shown on the right in Figure 1.2.

Variables can be used with the normal arithmetic operations, their value replaces the variable name internally, just as in any programming language. In Figure 1.3, we define x and y, and add them. We can perform operations on elements of arrays, or on the arrays themselves (so the final example in Figure 1.3 adds each element of the lists X and Y – note that you cannot add together lists of different size).

1.4 Control Structures

The most important tools for us will be the if-then-else, while and for operations. These can be used with logical operations to perform instructions based on certain variable values.

The if-then-else construction operates as you would expect – we perform instructions *if* a certain logical test returns True, and other instructions (*else*) if it is False. The Mathematica structure is:

```
If[test, op-if-test-true, op-if-test-false]
```

In Figure 1.4, we define and set the value of x to 4. Then we use the If statement to check the value of x – *if* x is less than or equal to 4, *then* we set x to 5, *else* we set x to −1.

Using While is similar in form – we perform instructions *while* a specified test yields True, and stop when the logical test returns False. The Mathematica command that carries out the While "loop" is

```
While[test, op-if-test-true]
```

An example in which we set x to −1 and then add one to x if its value is less than or equal to four is shown in Figure 1.4. In this example, we also

```
In[1]:= x = 2;
        y = 3;

In[3]:= x + y

Out[3]= 5

In[4]:= X = Table[Sin[j], {j, 0.0, Pi, Pi / 10}]

Out[4]= {0., 0.309017, 0.587785, 0.809017,
         0.951057, 1., 0.951057, 0.809017,
         0.587785, 0.309017, 1.22465×10^-16}

In[5]:= Y = Table[2.0 * j, {j, 1, 11, 1}]

Out[5]= {2., 4., 6., 8., 10.,
         12., 14., 16., 18., 20., 22.}

In[6]:= X[[2]] * Y[[1]]

Out[6]= 0.618034

In[7]:= X + Y

Out[7]= {2., 4.30902, 6.58779, 8.80902, 10.9511,
         13., 14.9511, 16.809, 18.5878, 20.309, 22.}
```

Figure 1.3: Using arithmetic operations with variables, list elements, and lists.

If Statement

```
In[1]:= x = 4;

In[2]:= If[x ≤ 4,
          x = 5;
          ,
          x = -1;
         ];

In[3]:= x

Out[3]= 5
```

While Statement

```
In[1]:= x = -1

Out[1]= -1

In[2]:= While[x ≤ 4,
          Print[x];
          x = x + 1;
         ]
-1
0
1
2
3
4
```

For Statement

```
In[1]:= For[x = -1, x ≤ 4, x = x + 1,
          Print[x];
         ]
-1
0
1
2
3
4
```

Figure 1.4: Using `Mathematica`'s `If`, `While` and `For`.

encounter the i/o function `Print[x]`, which prints the value of the variable `x`.

Finally, "for loops" perform instructions repeatedly while an *iterator* counts from a specified start value to a specified end value – more generally, the iterator is given some initial value, and a logical test is performed on a function of the iterator – while the logical test is true, operations are performed. We can construct a "for loop" from a While loop, so the two are, in a sense complimentary. In `Mathematica`, the syntax is:

```
For[ j = initialval, f[j], j-update, operations]
```

where `j` is the iterator, `f[j]` represents a logical test on some provided function of `j`, `j-update` is a rule for incrementing `j`, and `operations` is the set of instructions to perform while `f[j]` returns `True` – each execution of `operations` increments `j` according to `j-update`. This is easier done than said – an example of using the for loop is shown in Figure 1.4. That example produces the same results as the code in the `While` example in Figure 1.4.

1.5 Functions

Writing programs requires the ability to break, in our case, computational instructions into logically isolated blocks – this aids in reading, and debugging a program. These isolated blocks are called "functions", generically, a name for anything that takes in input and provides output. `Mathematica` provides a few different ways to define programming functions. We will focus on the `Module` form of function definition – the basic idea is:

```
functionname[input1_, input2_] := Module[ { local variables },
                                  operations;
                                  Return[value];
                                   ]
```

Examples of the `Module` in action are shown below, but morally, the important thing to remember is that we now have a function that can be called with some inputs, returns some output, and has hidden local variables that are not accessible to the "outside world".

```
In[1]:= HelloWorld[name_] :=
         Module[{localvarx, localvary},
          localvarx = 1.0;
          localvary = name;
          Print["Hello ", localvary];
          Return[localvarx];
         ]

In[2]:= X = HelloWorld["Joel"]

        Hello Joel

Out[2]= 1.

In[3]:= Y = HelloWorld[33];

        Hello 33

In[4]:= Y

Out[4]= 1.

In[5]:= localvarx

Out[5]= localvarx
```

Figure 1.5: Example of defining, and then calling, a function in `Mathematica` using `Module`.

In Figure 1.5, we define the function `HelloWorld`, that takes a single argument called `name` – the underscore in the function definition defines the argument of the function. The `Module` is set up with two local variables, one takes the value of `name` (generally, a string), and the other is set to one. The function itself prints a friendly greeting, and returns the value stored in `localvarx` (i.e. one). As a check that the variable `localvarx` really is undefined as far as the rest of the `Mathematica` "session" is concerned, the last line in Figure 1.5 calls `localvarx` – the fact that `Mathematica` returns the variable name, unevaluated, indicates that it is not currently defined.

We can use all of our arithmetic, logical, and control operations inside the function to make it do more interesting things. As an example, the two functions defined in Figure 1.6 are used to sort an array of numbers in increasing order.

```
In[1]:= Swap[inlist_, a_, b_] := Module[{holder, outlist},
          outlist = inlist;
          holder = outlist[[b]];
          outlist[[b]] = outlist[[a]];
          outlist[[a]] = holder;
          Return[outlist];
         ]

In[2]:= InsertionSort[inlist_] := Module[{outlist, indexx, indexy, curelm, Nlist},
          outlist = inlist;
          Nlist = Length[outlist];
          For[indexx = 2, indexx ≤ Nlist, indexx = indexx + 1,
           curelm = outlist[[indexx]];
           indexy = indexx - 1;
           While[indexy > 0 && outlist[[indexy]] > curelm,
            outlist = Swap[outlist, indexy, indexy + 1];
            indexy = indexy - 1;
           ];
           outlist[[indexy + 1]] = curelm;
          ];
          Return[outlist]
         ]

In[3]:= InsertionSort[{5, 2, 4, 6, 1, 3}]
Out[3]= {1, 2, 3, 4, 5, 6}

In[4]:= InsertionSort[{-4, 1, 7, 2, 3, -10}]
Out[4]= {-10, -4, 1, 2, 3, 7}
```

Figure 1.6: Definition of the function `InsertionSort` – this function takes a list and sorts the elements of the list in increasing order.

The first function we define is `Swap` – this takes a list, and two numbers as input, swaps the value of the elements of the list using the two numbers as array indices, and returns an array with the two elements interchanged. For the `InsertionSort` function, we go through the input array, and sequentially generate a sorted list of size `indexx-1`, increasing `indexx` until it is the size of the entire array. This is an inefficient, but straightforward way

to sort lists of numbers.

1.6 Input and Output

There are a wide variety of `Mathematica` functions that handle various input and output. We will introduce specific ones as we go, I just want to mention two at the start that are of interest to us. The first, we have already seen: `Print[ stuff ]` prints whatever you want, and can be used within a function to tell us what is going on inside the function.

The second output command we will make heavy use of is `ListPlot`, this function takes an array and generates a plot with the array values as heights at locations given by the array index. `ListPlot` can be used to visualize arrays of data, or function values. A few examples are shown in Figure 1.7.

1.7 Recursion

Most programming languages support a notion of "recursion" – this is the idea that a function can call itself. Recursion can be useful when designing "divide-and-conquer" algorithms. As a simple example of a recursive function, consider `DivideByTwo` defined below. This function takes a number, and, if it is possible to divide the number by two, calls itself with the input divided by two. If the number cannot be divided by two, the function returns the non-dividable-by-two input. In order to check divisibility, I have defined the function `IsDivisableByTwo` – this checks divisibility using `Mathematica`'s built-in `Round` command.

A concrete example of the function in action is shown in Figure 1.8 – we are using the `Print` command to see what value the function `DivideByTwo` gets at each call – you can see that it is called four times for the input 88.

The function `DivideByTwo` returns a concrete result when its input is not divisable by two.

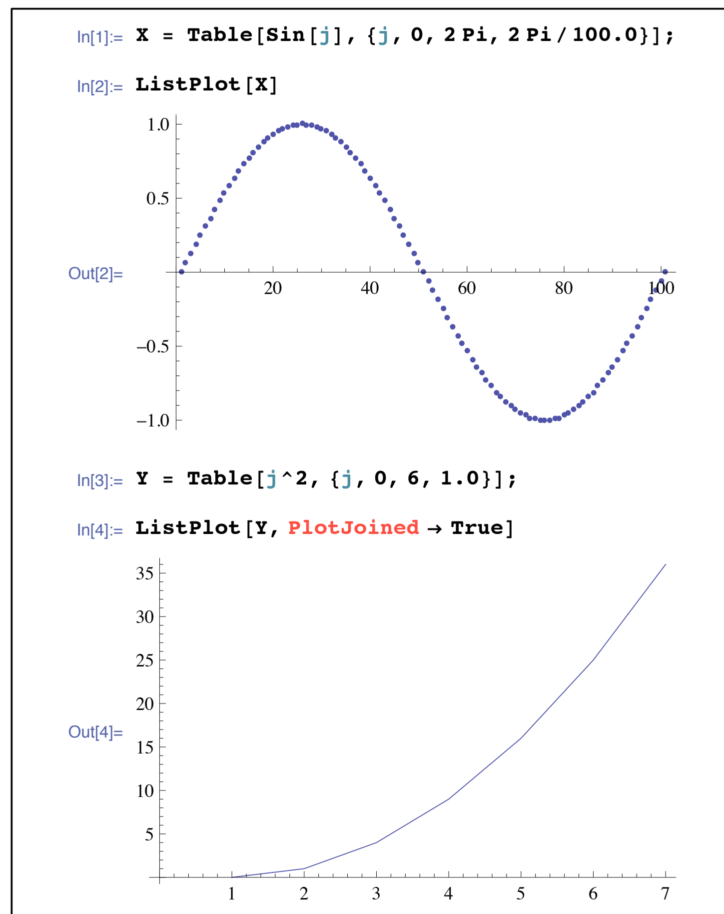


Figure 1.7: Plotting arrays of data.

```
In[1]:= IsDivisableByTwo[inx_] := Module[{div},
          div = inx / 2;
          If[div - Round[div] ≠ 0,
           Return[False];
           ,
           Return[True];
          ];
         ]

In[2]:= DivideByTwo[num_] := Module[{oput},
          Print[num];
          oput = num;
          If[IsDivisableByTwo[oput] == True,
           oput = DivideByTwo[oput / 2.0];
          ];
          Return[oput];
         ]

In[3]:= DivideByTwo[88]
       88
       44.
       22.
       11.
Out[3]= 11.
```

Figure 1.8: Example of recursive function definition – you must provide both the recursive outcome (call the function again with modified input) and the final outcome (the definition of the end point).

1.8 Function Pointers

It is important that functions be able to call other functions – we can accomplish this in a few different ways. One way to make a function you write accessible to other functions is to define it globally, and then call it. That is the preferred method if you have a "helper" function that is not meant to be called by "users". The function `Swap` in the insertion sort example Figure 1.6 is such a support function – it is not meant to be called by a user of the function `InsertionSort`, it is purely a matter of convenience for us, the programmer.

But sometimes, the user must specify a set of functions for use by a program. In this case, we don't know or care what the names of the functions supplied by the user are – they are user-specified, and hence should be part of the argument of any function we write. This "variable" function is known, in C, as a "function pointer" – a user-specifiable routine. Because of the low-key type-checking in `Mathematica`, we can pass functions as arguments to another function in the same way we pass anything. It is up to us to tell the user what constraints their function must satisfy. As an example, suppose we write a function that computes the time average of some user-specified function $f(t)$ – that is, we want to write a function that takes:

$$\text{TimeAverage}(f) = \frac{1}{T} \int_0^T f(t)\, dt. \tag{1.1}$$

We'll use `Mathematica`'s `Integrate` function for now. Our function, called `TimeAverage` below, takes as input the function $f(t)$ and the period of interest, T, and returns the right-hand side of (1.1). The code and two user test cases is shown in Figure 1.9.

1.9 `Mathematica`-Specific Array Syntax

Most programming languages have a notion of memory allocation (whether the responsibility of the programmer, compiler, or operating system) – we need a way to store values of various sorts. An array of numbers can be viewed as a vector, but we could also have an array of alphanumeric characters, called a "string", or an array of some more exotic collection of data (think of an array of arrays of numbers of length 2, for example).

`Mathematica` uses "`Table`"s to store information. The simplest tables are

```
In[1]:= TimeAverage[f_, T_] := Module[{tavg},
          tavg = (1 / T) Integrate[f[t], {t, 0, T}];
          Return[tavg];
         ]

In[2]:= userfunction1[t_] := Sin[t]^2

In[3]:= TimeAverage[userfunction1, 2 Pi]

Out[3]= 1
        -
        2

In[4]:= userfunction2[x_] := Sin[x] Cos[x]

In[5]:= TimeAverage[userfunction2, 2 Pi]

Out[5]= 0
```

Figure 1.9: The function TimeAverage takes a function name as its argument, in addition to the period over which to average.

just lists of numbers, but we can make tables of tables (of tables), and use these to store multiple pieces of information. As an example, in Figure 1.10, we see a definition of a table – each entry contains two elements, one number, and one list of two numbers. This type of table would be useful, for example, in encoding the time (the first element), and position-and-velocity (the second pair) of a particle. In the example, if we ask for the second element of our list, we get back an array consisting of one number, and an array (as expected) – we can work our way down, so that `tXV[[2,1]]` gives us back the first element of the array at location 2 of `tXV`, a "time" (if we like). Finally, we can recover the position via `tXV[[2,2,1]]`, literally "The first element of the second element of the array `tXV`.".

We can also use the built-in table command to construct such tables, and in that context, we see at the bottom of Figure 1.10 the same `tXV` table built automatically.

```
In[1]:= tXV = { {0.0, {1.0, 0.1}}, {0.1, {1.5, 0.2}}, {0.2, {2.0, 0.3}},
          {0.3, {2.5, 0.4}} };

In[2]:= tXV[[2]]
Out[2]= {0.1, {1.5, 0.2}}

In[3]:= tXV[[2, 1]]
Out[3]= 0.1

In[4]:= tXV[[2, 2]]
Out[4]= {1.5, 0.2}

In[5]:= tXV[[2, 2, 1]]
Out[5]= 1.5

In[1]:= tXV = Table[{t, {1 +  .5 (t / .1), .1 + t}}, {t, 0, .3, .1}]
Out[1]= {{0., {1., 0.1}}, {0.1, {1.5, 0.2}}, {0.2, {2., 0.3}}, {0.3, {2.5, 0.4}}}
```

Figure 1.10: We can define tables (arrays) with elements consisting of multiple data types.

1.10 Timing and Operation Counts

Finally, we will be interested in timing our methods, so we need a basic notion of counting the amount of time a particular algorithm will take to run. The rules are simple: 1. Every addition, subtraction, multiplication or division takes the "same" amount of time to execute. 2. We are interested in the timing of an algorithm, up to constants that are machine specific. While a particular computer may add in three microseconds, but take nine microseconds to multiply, these are, up to order of magnitude, the same. Our interest is generally in the scale of the computation, not the details. So, given a problem with N parameters that we know (experimentally, say) can be solved on an iPad in n seconds, our question will always be, how many seconds does it take the iPad to solve the same type of problem with M parameters?

To be concrete, a general matrix inversion problem for an $N \times N$ matrix can be solved in $T(N) = \alpha\, N^3$ seconds, where α is a machine-specific (and therefore uninteresting) constant. So we know what happens to the timing if we double the size of the matrix – we octuple the timing: $T(2\,N) = \alpha\,(2\,N)^3 = 8\,T(N)$. It is this scaling that is of interest, so clearly, constants

like α don't really matter. We use the "O" notation to indicate the scaling, so we would write: $T(N) = O(N^3)$ to suggest that the timing for a problem of size N takes time proportional to N^3, with constants of proportionality set by the machine itself (and sometimes, our cleverness of implementation – after all, $.9\,N^3$ is worse than $.01\,N^3$).

When counting operations, then, we can simply give the scaling with fundamentally interesting parameters in the problem. As a concrete example, take the matrix dot product of $\mathbf{a}$ and $\mathbf{b}$, both in $\mathbb{R}^N$. We know that:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{N} a_i\, b_i \tag{1.2}$$

and if we think of how the timing for this calculation goes with N, we have:

$$T(n) = \alpha_\times\, N + \alpha_+\, (N - 1) \tag{1.3}$$

where $\alpha_\times$ is the constant associated with multiplication, α_+ that associated with addition. But, using our O notation, we can simplify:

$$T(n) \le \max(\alpha_+, \alpha_\times)\, N = O(N) \tag{1.4}$$

so that fundamentally, the time it takes to compute the dot product scales linearly with N. This suggests that if you knew, say, $T(2)$ for your computer (in order to get the coefficient out front of $O(N)$), you could compute $T(100)$.

Given that, what is $T(N)$ for matrix-vector multiplication?

Bibliography

1. Cormen, Thomas H. & Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

Lab

In this laboratory, we will explore elements of programming and manipulating basic output. Use the provided boxes for answers, and to show any work relevant to your calculation.

Problem 1.1

Predict the output of the command `DivideByTwo[76]`:

Problem 1.2

We can view arrays in a variety of ways. In this problem, we'll make a list of particle locations. Suppose we have one hundred particles to keep track of, each has three spatial coordinates, so we could imagine generating a table with 100 entries, each entry is itself a set of three values. Using the `Table` command, construct a table that spaces our "particles" evenly along the line from $\mathbf{r}_0 = 0$ to $\mathbf{r}_f = 1\,\hat{\mathbf{x}} + 1\,\hat{\mathbf{y}} + 1\,\hat{\mathbf{z}}$ (so the first particle is at the origin, and the hundredth particle sits at $\{1, 1, 1\}$). If you call your table `Xlist`, what is the output of: `Xlist[[72,2]]`? Provide both the numerical answer to five digits, and describe your interpretation of this entry in words, below:

Problem 1.3

Suppose we make a table using the command:

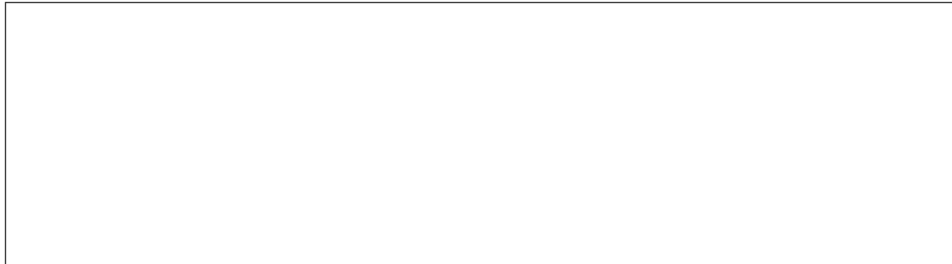
```
ret = Table[{j, Table[Table[m,{m,1,k}],{k,1,j}]},{j,1,10}]
```

What will the output of `ret[[8,2,5,3]]` be? (Predict first, then check).

Problem 1.4

Write a function that takes two vectors $\mathbf{a}$ and $\mathbf{b}$ and outputs the dot product $\mathbf{a} \cdot \mathbf{b}$ using the `Sum` command. Note that your function should fail if the two vectors are not of the same length (you can check the length of a list using `Length`) – failure should be indicated with a printed statement, and a return value that could *not* be a valid dot-product output.

There is a `Timing` command that can be used to find how many seconds it takes to perform a particular operation – the usage is: `Timing[ ops ]`, and the output is { `timing, return value of ops`} (i.e. a table with two entries). Using this command, write a function that takes, as its argument, an integer (call it `nsize`), and returns the amount of time it took to compute the dot-product of $\mathbf{a} \cdot \mathbf{b}$ for $\mathbf{a}$, $\mathbf{b} \in \mathbb{R}^{\texttt{nsize}}$. Using your function, create a table of the amount of time it takes to compute the dot-product for vectors of length $10000, 20000, \ldots, 100000$ (in steps of 10000). Use `ListPlot` with your table to estimate the amount of time it takes your computer (running `Mathematica`) to perform an arithmetic operation – describe your procedure for determining this value.



Problem 1.5

Write a function that takes input parameters `T` and `m`, and output (in a table of length two) that contains: 1. The frequency (f, not ω) of a sinusoidal wave that is zero at $t = 0$ and $t = T$ and contains m full cycles, and 2. A plot of this sinusoidal function that has time as its x-axis. Give the output of

```
yourfunc[4 Exp[1], 3]
```

in the space below:

Problem 1.6

Construct a function that takes as input the height z at which a solid body approaches a sphere of radius $R = 1$, as shown below, and outputs the angle θ at which the body "bounces" off the sphere (use angle of incidence equals angle of reflection). What is the angle θ if $z = 0.25$ (answer below)?

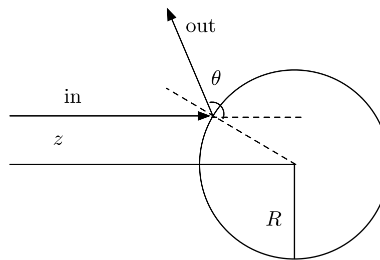


Figure 1.11: A solid body approaches a sphere at height z, and bounces off of it at angle θ.

Make a table of angles for a range of heights z (you can use the built-in command `RandomReal[{-1,1}]` to specify a random number from -1 to 1, for example) – using the `Histogram` function, generate a histogram of output angles for 100000 input heights. Sketch your histogram on the range $-\pi \longrightarrow \pi$ (this means you will need to modify the output for $z < 0$).