Introducing *Mathematica*

John H. Lowenstein

Preface

This informal introduction to *Mathematica*®(a product of Wolfram Research, Inc.) is offered as a downloadable resource for users of the textbook *Essentials of Hamiltonian Dynamics* (Cambridge University Press, 2012). The aim is to familiarize the student with the core concepts and functions of *Mathematica* programming, so that he or she can very quickly become comfortable with computational methods in dealing with the illustrative examples and exercises in the textbook. The scope of *Mathematica* obviously greatly exceeds what can be covered in these few pages, and so it is highly recommended that the student take full advantage of the excellent documentation which is included with the software (accessible via the Help menu).

1. Getting started

Mathematica conducts a dialogue with the user: you type in a mathematical expression, press Enter , and *Mathematica* evaluates the expression according to rules which are either built-in or have been prescribed by you, displaying the result as output. The input/output alternation continues until you quit the session, with all steps recorded in the cells of your notebook.

The simplest expressions involve ordinary numbers (e.g. **1**, **2**, **3**, . . . , **1/2, 2/3**, . . . , **3.14159**), and the familiar operations and relations of arithmetic and logic.

Elementary numerical operations:

+ (plus), - (minus), * (times), / (divided by), ^ (to the power)

Elementary numerical relations:

== (is equal to), != (is not equal to),

< (is less than), > (is greater than),

<= (is less than or equal to), >= (is greater than or equal to)

Elementary logical relations:

|| (or), && (and), ! (not)

For example, **2+2** evaluates to 4 , **1<2** evaluates to True , and **!(2>1)** evaluates to False . Note that if *Mathematica* cannot decide whether a relation is true or false (or more generally when there are no rules applicable to the evaluation of an input expression), it simply outputs the expression unchanged. For instance, it is meaningless to ask whether the complex number $2 + 3i$ is greater than 1 , and so evaluation of **2+3I > 1** yields the output 2+3i > 1 , together with a few words of friendly advice about he folly of trying to order the complex numbers.

Exercise 1. Evaluation of simple expressions.

Evaluate the following expressions "by hand" . Check your results using Mathematica.

1) 4 × 7 - 8 × 6
2) 4 × (7 - 8) × 6
3) 4 × (7 - 8 × 6)
4) 3 · 3 · 3 ≥ 27
5) (4 × 6 > 12 × 2) || (-5/7 - 1 > -1/7)
6) (4 < 5) && (5 < 6) && (6 < 7) && (7 < 4)
7) (3 × 9 - 35 ÷ 5) ÷ (6 × 4 + 72 ÷ 2)
8) 5/8 == 1/(1+1/(1+1/(1+1/(1+1/(1+1)))))
9) Is it possible to construct an expression involving only *real* numbers which *Mathematica* will be unable to evaluate (except by cheating or simply outputting the input)?
10) Give an example of an expression which involves all three of the listed logical operations and which evaluates to True .

In addition to numbers, *Mathematica* manipulates abstract symbols ("unknowns") which you introduce. For this purpose, you may use individual letters or strings of letters and numbers (i.e. words), avoiding capitalization. Capitalized words are used by *Mathematica* for its own built-in expressions (e.g. **Pi** for π , **E** for e , the base of natural logarithms, **I** for i , the square-root of -1 , and **Abs** for the absolute-value function).

Numbers and abstract symbols are the simplest examples of what *Mathematica* recognizes as legitimate *expressions*. They are *elementary* , in the sense that there are no rules which can be used to reduce them to simpler form. All non-elementary expressions have the form

```
h[e1,e2,...en]
```

were **h, e1, e2, ...,en** are all expressions, which may or may not be elementary. The basic *Mathematica* operation (its part in the dialogue with you) is to *evaluate* expressions, i.e. to consult its stored list of rules to replace the input expression by a well defined output expression. For example, the input expression **4*6** is actually shorthand for the more formal **Times[4,6]** . The sub-expressions **Times, 4, and 6** are all elementary, and cannot be simplified. On the other hand, *Mathematica* has built-in rules associated with expressions having **Times** as the "head", namely the rules of multiplication, which allow Mathematica to replace **Times[4,6]** by the single number 24 . In the same way, the expression **3==5** is the shorthand form of **Equal[3,5]**, which evaluates to `False`.

A *compound expression*

```
ex1;ex2;....;exn
```

is evaluated sequentially (left to right), with only the evaluated expression `exn`
displayed in the output. If `exn` is omitted, or replaced by `Null` , there is no output displayed, although the evaluation of **ex1;ex2;...;ex(n-1);** may have an effect on stored quantities. For example, the *assignment* expression

```
a=3;
```

associates the rule "Replace **a** by **3**" with the symbol **a** . Because of the semicolon, the evaluation of **a=3;** creates no output , but of course, the effect of the evaluation is non-trivial. Henceforth, every time **a** appears in an expression, it will be replaced by **3** . The value can be changed by a new assignment, or cleared by evaluating **Clear[a]** . At any time during your *Mathematica* session, you can check what rules, if any, pertain to a particular symbol. Evaluating **?a** reveals all of the rules associated with the symbol **a** .

Often it is desirable to evaluate an expression with certain values inserted for the sysmbols, but without storing those values for future use. For example, suppose we want to evaluate **Sin[a*b]** with **a** set equal to 0.75 and **b** set equal to 1.09, but without permanently assigning those numerical values to **a** and **b**. We would evaluate

```
In[1]:= Sin[a * b] /. {a → 0.75, b → 1.09}
Out[1]= 0.729438
```

Exercise 2. Assignment. Evaluation of compound expressions.

Evaluate the following expressions "by hand". Check your results using Mathematica.

1) **a=2; b=3; a*b**
2) **a + 1/a + 1/(a*a)/.a->2**
3) **x=3.1; y=2.5; z=1.9; (x^2 + y^2 +z^2)^(1/2)**
4) **a=1; a=2; b=a^3**
5) **Clear[a]; b=a**
6) **b=(a+1)/.a->1; c=b+1; a=b+c**
7) **a=2; a*b^2/.b->10**

---

2. Functions

A *function* quite generally is a rule which assigns to each member of a certain set (domain) a unique member of a certain other set (range). A function in *Mathematica* has, not surprisingly, the additional requirement that the domain and range are sets of *expressions* in the strict sense discussed above. To define a function f depending on n variable arguments we write

f[x1_,x2_,....,xn_]:= expression depending explicitly on **x1,x2,....,xn**

where the lower dashes following the argument names indicate that these are not specific expressions, but rather are dummy variables which can take on arbitrary values within their respective domains, and the *delayed assignment* symbol := indicates that all expressions appearing on the righthand side are to be evaluated only at the time that the function is evaluated. The distinction between := and = is a bit subtle, but should become clear through examination of a simple example. Define a function **g[x_]** via

```
g[x_]:= a*x^2 + 1
```

Now evaluate

```
a=3;x=9;g[2]
```

to obtain the value 13. Note that assigning the symbol **x** the value 9 has no effect on the evaluation of **g[2]** since the **x** appearing in the function definition is a dummy variable which is replaced by the argument 2 rather than being evaluated according to the stored rule **x=9** associated with the symbol **x**.

The concept of assignment is a bit tricky, and is made more mysterious by the use of the equals sign in a way quite different from ordinary mathematical usage. Consider, for instance, the expression

```
a=a^2
```

In ordinary algebra this would be an equation with only two numerical solutions, 0 and 1. But suppose we have already assigned the value 3 to the symbol **a**. What happens when Mathematica evaluates **a=a^2** ? Answer: first,

the righthand member is evaluated making use of the stored value assigned to **a;** the result (in this case, 9) is then assigned to **a** as its new value. If we now inquire about the rules associated with **a** by evaluating **?a** , we will find only one rule, namely **a=9** . This example is not a weird exceptional case. Whenever we want to describe the transformation of a quantity **x** by a function **f** (written in ordinary mathematics as $x \mapsto f(x)$) we will evaluate in *Mathematica* the expression **x=f[x]**. Whatever value is assigned to **x** is inserted into the function definition of **f** and the evaluation carried out; the result is then assigned to **x** as its new value.

Exercise 3 Function Definitions.

1) Define functions **sphereA[diam_]** and **sphereV[diam_]** which output, respectively, the area and volume of a sphere of diameter **diam** .

2) Define a function **dist[x_,y_,z_]** which, for every point **(x,y,z)** in 3-dimensional space (rectangular coordinates) calculates the distance of that point from the origin.

3) Define a function **triangleQ[s1_,s2_,s3_]** which tests for the triangle inequality. It should output **True** if each of its arguments is less than the sum of the other two, and **False** otherwise. (Note that if **s1,s2,s3** are the lengths of the sides of an actual triangle, the function will indeed evaluate to **True)**.

4) Define a function **f[a_,b_]** which outputs **True** if the cube of **a+b** is smaller than the square of **a** or larger than the 4th power of **b** , and **False** otherwise.

5) Define a function **g[a_,b_]** which assigns the value **a** to the variable **x** , assigns the value **b** to the variable **y** , and, finally, outputs the product of **x** and **y**.

---

3. Representation of numbers

Mathematica stores and manipulates various types of numbers.

Exact real and complex numbers. Integers, rational numbers (ratios of integers), and unapproximated irrationals (such as the built-in constants **Pi and E**) are handled as exact quantities. For example, if you ask *Mathematica* to evaluate 13 or 51/101 , or **Pi**, it merely outputs the entered number. Complex numbers take the form **a+b I** , where **a** and **b** are real and **I** is a built in constant representing $\sqrt{-1}$.

Arbitrary precision floating-point numbers. *Mathematica* can approximate an exact real number **r** with a specified number **digits** of significant digits. Specifically, evaluate **N[r,digits]**, or, alternatively, **SetPrecision[r,digits].** For example, **N[1/3,8]** evaluates to 0.33333333, while **N[Pi,1000]** evaluates to

```
3.14159265358979323846264338327950288419716939937510582097494459230781640\
  62862089986280348253421170679821480865132823066470938446095505822317253\
  40812848111745028410270193852110555964462294895493038196442881097566593\
  44612847564823378678316527120190914564856692346034861045432664821339360\
  26024914127372458700660631558817488152092096282925409171536436789259036\
  01133053054882046652138414695194151160943305727036575959195309218611738\
  93261179310511854807446237996274956735188575272489122793818301194912983\
  67336244065664308602139494639522473719070217986094370277053921717629317\
  75238467481846766940513200056812714526356082778577134275778960917363717\
  72146844090122495343014654958537105079227968925892354201995611212902196\
  86403441815981362977477130996051870721134999999837297804995105973173281\
  60963185950244594553469083026425223082533446850352619311881710100031378\
  75288658753320838142061717766914730359825349042875546873115956286388235\
  78759375195778185778053217122680661300192787661119590921642019893
```

Machine precision floating-point numbers. This is the most efficient way of handling real numbers in most applications. If your input number contains an explicit decimal point, *Mathematica* will interpret it as having a certain number (the *machine precision*) of significant digits. You can learn your computer's machine precision by evaluating the built-in constant **$MachinePrecision**. To approximate an exact number **r** by a machine-precision floating-point number, evaluate **N[r]**.

Minimum and maximum precision. *Mathematica* has two stored constants, **$MinPrecision** and **$MaxPrecision,** which specify the minimum and maximum precision with which approximate real numbers are represented. In the evaluation of **N[r,digits]** or **SetPrecision[r,digits]**, with **r** an exact real number, if **digits** is less than **$MinPrecision**, the assigned precision will be upgraded to **$MinPrecision**; if **digits** exceeds **$MaxPrecision**, the assigned precision will be downgraded to **$MaxPrecision**. Since normally **$MinPrecision** and **$MaxPrecision** have the values 0 and infinity, respectively, the user can safely forget about them. However, there is a useful application of these bounds which deserves mention here. If you would like to perform a calculation using *uniform* precision **n**, then you should set

```
$MaxPrecision = $MinPrecision = n;
```

Repeating a calculation with several different values of **n** is often a good way of testing whether round-off error is playing a significant role.

4. Transcendental functions

Mathematica includes a large supply of built-in mathematical functions, including those normally found on a good scientific calculator, namely **Sqrt[z_], Exp[z_], Log[z_], Log[base_,z_], Sin[z_], ArcSin[z_], Sinh[z_], ArcSinh[z_],** etc. , as well as a fairly complete collection of the special functions of mathematical physics, such as **BesselJ[z_]**, **EllipticK[m_]**, **JacobiSN[u_,m_]**, etc.
Important: a function whose arguments are all exact will evaluate to an exact (not floating-point) quantity, so, for example, **Sin[2]** evaluates to itself. To get the machine-precision numerical value, you should use either Sin[2.0] or N[Sin[2]].

5. Lists

One of the most common types of non-elementary expressions is a *list* of numbers or, more generally, expressions, written **{e1,e2,....,en}** or, more fully, **List[e1,e2,...en].** Given a list **mylist**, you can extract its kth member by evaluating **mylist[[k]]** (not to be confused with **mylist[k] !).** The number of elements in a list **mylist** is given by **Length[mylist]**. For use later on, we introduce a few of the built-in *Mathematica* functions pertaining to manipulation of lists.

- **Join and AppendTo**

The use of **Join** and **AppendTo** is obvious from the following examples:

```
In[2]:= Join[{1,2,3},{4,5,6}]
Out[2]= {1, 2, 3, 4, 5, 6}
In[3]:= alist={1,2,3};AppendTo[alist,4];alist
Out[3]= {1, 2, 3, 4}
```

- **Map**

The function **Map** is used to apply a function **f[x_]** to each element of a list, i.e. **Map[f,{e1,e2,...,en}** evaluates to **{f[e1],f[e2],...,f[en]} .**
For example, if we define

```
In[4]:= f[x_]:=x^2
```

we have

In[5]:=
```
Map[f,{1,2,3,4,5,6}]
```

Out[5]=
```
{1, 4, 9, 16, 25, 36}
```

A commonly used shorthand notation for **Map[f,{e1,e2,...,en}** is **f/@{e1,e2,....,en}**.

- **Example: compound interest**

```
ls = {1023.76, 123.98, 765.50, 86.09, 877.82, 1201.21, 576.91, 233.77,
1172.40,36.63}
```

is a list of the Jan. 1, 1998 savings account balances at the National City County Bank, earning 5% per annum compounded quarterly, and we wish to predict the Jan. 1, 1999 balances (assuming no deposits or withdrawals), we define the function

```
new[x_] := 1.0125*x
```

which gives the balance at the end of a quarter as a function of its value x at the beginning of that quarter (the interest rate is of course one-quarter of .05). We then introduce a new function which gives the cumulative effect after 4 quarters, i.e. the so-called *fourth iterate* of the function new,

```
new4[x_] := new[new[new[new[x]]]]
```

The built-in function **Nest[f_,exp_,n_]** provides a more economical alternative:

```
new4[x_]:= Nest[new,x,4]
```

.

To update all ten of the savings balances simultaneously, we evaluate

```
ls = new4 /@ ls
```

,

with output

```
{1075.92, 130.296, 804.499, 90.4759,  922.541, 1262.41, 606.301, 245.679,
1232.13, 38.4961}
```

- **Table**

Often we create lists by applying repeatedly a single rule (function). This construction is realized in *Mathematica* by the built-in function **Table,** which behaves as follows:

```
Table[g[k],{k,1,n}]
```

evaluates to **{g[1],g[2],....,g[n]}.** Within the **Table** function, the "iterator" **k** is a dummy variable which assumes the values 1,2,...,n. The list of squares obtained by using **Map** can also be obtained, more simply, using **Table**:

In[6]:=
```
Table[i^2,{i,1,6}]
```

Out[6]=
```
{1, 4, 9, 16, 25, 36}
```

In dynamics, lists are often defined recursively, with the (n+1)st member generated from the nth member by a rule $x_{n+1} = f(x_n)$. Given the initial value x_0, a 1000-step sequence can be calculated as

```
Join[{{x0}}, Table[x = f[x], {1000}]]
```

Here the argument **{1000}** is shorthand for **{k,1,1000}**.

Exercise 4. Constructing and transforming lists.

1) Using **Table**, construct a list **intlist** of the first 100 positive integers.

2) Using **Table**, construct a list **oddlist** of the first 100 odd, positive integers in reverse numerical order (largest first).

3) Use **Map** with a suitably defined transformation function to to transform **oddlist** into **evenlist** consisting of the first 100 even integers in reverse numerical order.

4) Use **Map** with a suitably defined transformation function to to transform **evenlist** into a list of the first 100 integers in reverse numerical order.

5) Use **Table** to define a function **reverse[ls_]** which takes any list **ls** of length 100 and outputs a list of length 100 in which the members of **ls** appear in reverse order. What happens if you apply this function to a list of length different from 100?

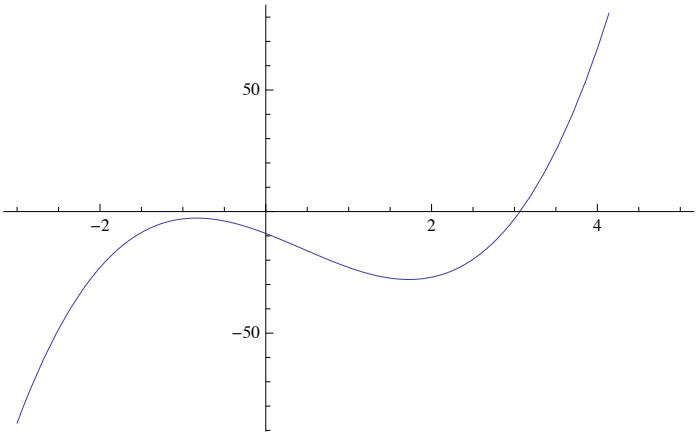
---

6. Displaying data

Mathematica has a number of built-in functions for the graphical display of information. We shall make frequent use of these functions, without worrying about how they have been programmed by experts. In this section we introduce the two most frequently used of the graphics functions, **Plot[]**, which plots the graph of a function over a prescribed range of arguments, for example

In[7]:=
```
Plot[3*x^3-4*x^2-13*x-9,{x,-3,5}]
```

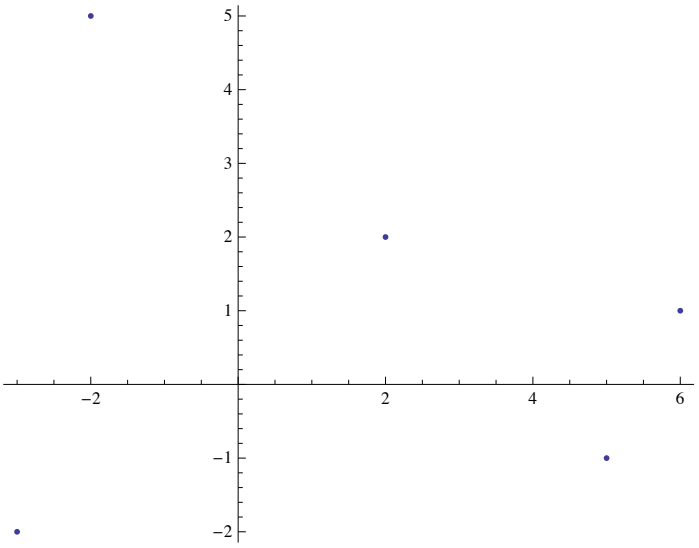
Out[7]=



and **ListPlot[]**, which plots the points in the plane specified by a list of **{x,y}** pairs, for example

In[8]:=
```
ListPlot[{{-2,5},{6,1},{-3,-2},{2,2},{5,-1}}, AspectRatio->Automatic]
```

Out[8]=



Each of the graphics functions has a large number of options. When none are listed, defaults are chosen behind the scenes. For example, if you do not specify the aspect ratio, *Mathematica* chooses it to be the inverse of the golden ratio, i.e. .618034... The choice **AspectRatio->Automatic**, on the other hand, specifies the same scale for hyorizontal and vertical coordinates, which is what we usually want to represent points in the plane. The hidden options are revealed by evaluating **Options[Plot]** and **Options[ListPlot].**

■ **Explorations**

Evaluate **Options[Plot]** and **Options[ListPlot]** to display all the default options for these functions.

In the **ListPlot** example, try out the options (one at a time)

```
AspectRatio->1/2,
AspectRatio->2,
Joined->True,
PlotRange->{{-3,4},{-3,4}},
Axes->False,
Frame->True,
PlotStyle->PointSize[.1],
PlotStyle->Red
AxesStyle->RGBColor[0,1,0]
DisplayFunction->Identity
```

Exercise 5. **Rootfinding using Plot[]**

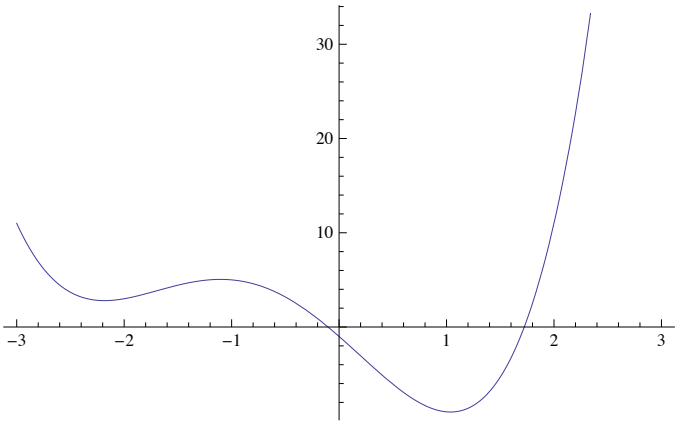
In this exercise we put *Mathematica*'s graphing power to use to solve a nontrivial mathematical problem. Define a polynomial function $p(x)$ of the real variable x by

```
p[x_] := x^4 + 3*x^3 - 2*x^2 - 10*x - 1 ;
```

We seek to find the largest real x for which $p(x) = 0$.
Very soon we shall develop a very elegant method for solving this class of problems, but for the time being we just want to see how raw computational power can solve the problem without being very clever. To get a rough idea of the solution we can make use of *Mathematica*'s basic plotting function, Plot, to obtain the graph

In[24]:= **Plot[p[x],{x,-3,3}]**

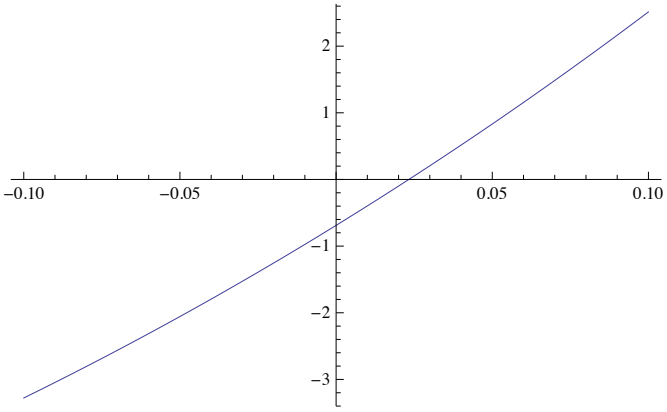
Out[24]=



The root of p(x) clearly lies between 1.6 and 1.8. To improve on our estimate, we "zoom in" and plot the function over that small range of x values, centered about its midpoint:

In[25]:= **Plot[p[1.7 + x],{x,-.1,.1}]**

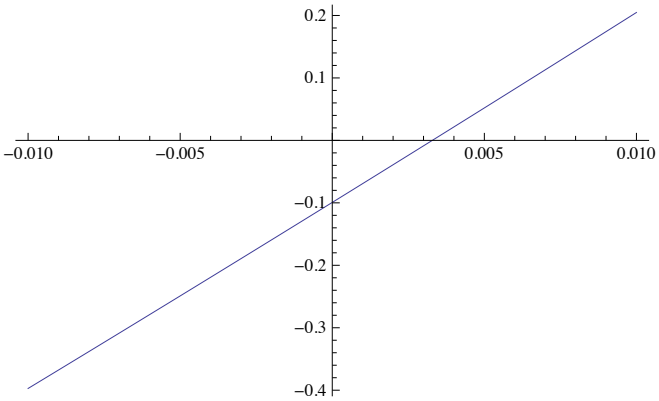
Out[25]=



Now we see that the function p(x) goes through zero between 1.71 and 1.73, and so we again use the Plot function to zoom in on the relevant interval:

In[26]:= **Plot[p[1.72 + x],{x,-.01,.01}]**

Out[26]=



Continue in this fashion until you have determined the root with 12 significant figures.
Once you get the hang of it, you should be able to gain at least one digit of accuracy with each new plot.

7. Branching and iteration

Thus far we have been using *Mathematica* as an electronic calculator capable of manipulating numbers, symbols, lists, and graphics. To obtain full programming capability, we need to add to our machinery two essential functions to allow for branching (alternative pathways) and iteration (repetition). To allow for the dependence of a sequence of operations on whether a certain condition is satisfied or not, we use the built-in function **`If[a_,b_,c_]`**, which evaluates to **`b`** if **`a`** evaluates to `True`, and to **`c`** if **`a`** evaluates to `False`. (What happens if a is neither true nor false ?)

Exercise 6. Using `If[a_,b_,c_]`.

Define the functions

```
abs[x_]:= If[x>=0,x,-x]
step[x_]:= If[x>=0,1,0]
```

Without help from *Mathematica*, sketch the graphs of the functions over the interval $-1 \leqslant x \leqslant 1$.
Then use **`Plot`** to check your results.

Mathematica has several functions which implement repeated (iterated) evaluation of an expression. We shall restrict ourselves to the simplest of these,

```
Do[ex[i],{i,min,max}]
```

which evaluates **`ex[i]`** for **`i`** equal to **`min`**, then again for **`i`** equal to **`min+1`**, etc. and finally for **`i`** equal to **`max`**. The examples below will show the power and versatility of this function.

Note that some people prefer to state the iteration range *before* the expression **`ex`**. This can be done using

```
For[i=min,i<=max,i++,ex[i]]
```

which is equivalent to **`Do[ex[i],{i,min,max}]`**. The **`For`** function and the notation **`i++`** (shorthand for **`i=i+1`**) are borrowed from the C programming language.

The **`Do`** and **`For`** functions provide a convenient way to sum over a sequence of numbers. For example, to add up the entries in a list **`ls`** of length **`length`**, we evaluate

```
sum=0; Do[sum=sum+ls[[i]],{i,1,length}]; sum
```

and to sum up the geometric series 10 + 11 + 12 + ...+ 87 + 88 + 89 + 90, we evaluate

```
In[12]:= total = 10; For[n = 10, n <= 90, n++, total = total + n]; total
Out[12]= 4060
```

Exercise 7. Products and series.

1) The factorial function, $n! = 1 \cdot 2 \cdot 3 \cdots n$, is a product of n factors, and may be calculated using the same iterative strategy used above for a sum of n terms. Define a *Mathematica* function **`fac[n_]`** which, for given n, evaluates to n!.

2) The base of natural logarithms, e , can be approximated by the following sum:

$$e(n) = \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$$

Using **Do** or **For** and **fac** , define a *Mathematica* function **e[n_]** which, for any positive integer **n** , gives a fractional approximation to e . Write this number as a decimal using *Mathematica*'s built-in function **N[x_,k_]**. Calculate $e(15)$ and compare with the result of evaluating **N[E,k]** with the precision **k** appropriately chosen.

3) The natural logarithm of 2 has the following expansion as an infinite series:

$$\ln 2 = \sum (-1)^{k+1} \frac{1}{k}$$

Define a *Mathematica* function **ln2[n_]** which calculates the first n terms of the series and displays the result as a decimal with 50 significant figures.

4) Repeat calculations (2) and (3) using the built-in function **Sum**. (Consult the documentation for the proper usage.)

Another useful application of iteration is to produce a table of values. For example, to create a table of the first 3 powers of the first 10 positive integers, we evaluate

In[13]:=
```
powers={};
Do[AppendTo[powers,{n,n^2,n^3}],{n,1,10}];
powers
```

Out[15]=
```
{{1, 1, 1}, {2, 4, 8}, {3, 9, 27}, {4, 16, 64}, {5, 25, 125},
 {6, 36, 216}, {7, 49, 343}, {8, 64, 512}, {9, 81, 729}, {10, 100, 1000}}
```

The same result is achieved more concisely using the list-making function **Table** introduced earlier:

```
powers=Table[{n,n^2,n^3},{n,1,10}]
```

To display our result in a more readable form, we can make use of *Mathematica*'s built-in function **TableForm**:

In[16]:=
```
TableForm[powers]
```

Out[16]//TableForm=

1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

Exercise 8. Constructing a table of values. Ravages of inflation.

Skeptical about the long-term survival of the National City County Bank, a winner of the state lottery, in 2012, places $1 million in gold coins in a buried safe with instructions in her will that the money be distributed to her heirs 500 years later. Assuming a uniform inflation rate of 3% per year over the entire period, construct a table of the values, in 2012 dollars, of the buried cash for the years 2012,2022,2032,...,2512. Plot the results using *Mathematica*'s built-in function **ListPlot[list_]** . To connect the points in the graph, include in **ListPlot** the argument **Joined->True**.

It is often desirable to allow for an iterated evaluation to be interrupted under specific conditions, or for a particular step to be omitted under specific conditions. These are accomplished by employing, together with **Do** and **If** , the built-in functions **Break[]** and **Continue[]**, respectively. If **Break[]** is encountered in any step, the iteration sequence is immediately halted; if **Continue[]** is encountered, evaluation of the current step is halted and the evaluation sequence proceeds to the next step. For, example, here is an expression which evaluates to the largest integer whose square is less than 1000:

```
Do[ If[(n+1)^2>=1000,Break[]]; m=n  , {n,1,50} ]; m
```

and here is one which evaluates to the sum of all elements of the list **numlist** which are greater than 5:

```
numlist={1,3,7,9,31,2,14,4,5,8};
sum=0;
For[n=1,n<=10,n++,
  If[n<=5,Continue[]];
  sum=sum+numlist[[n]]];
sum
```

Exercise 9. Using Break[] and Continue[].

1) Define a function **pos[numlist_]** which, when given a list of numbers **numlist**, outputs another list of numbers consisting of the positive members of **numlist**, in order. Hint: use **Table** to construct the new list, using **If** and **Continue** to filter out the non-positive members.

2) Define a function **search[stringlist_]** which, when applied to a list of strings **stringlist**, systematically examines each member in order and performs one of the following operations:

* if the string is "Bob", it prints "Bob found. End of search." and quits the search.
* if the string is "stop", it prints "End of search." and quits the search.
* otherwise it proceeds to the next string on the list.

3) What exactly does the following short "program" calculate?

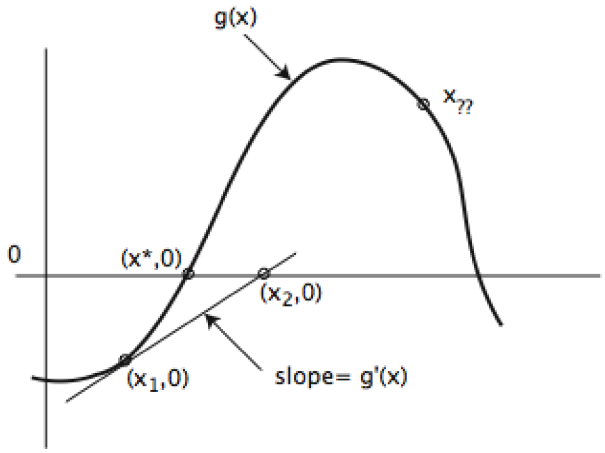
```
count=300000;
Do[If[count/100000.0>N[Pi],Break[],count=count+1],{k,1,100000}];
count/100000
```

Run the "program" and compare the result with what you get by evaluating **N[Pi,6]**. Explain any discrepancy.

4) A sheet of paper has thickness 0.1 mm. Folding the sheet doubles the thickness. Assume that it is possible to repeat the folding operation arbitrarily many times, After how many foldings does the thickness exceed the diameter of the Earth? Use *Mathematica*'s **For** function to automate the calculation of the thickness after each folding, using **Break** to stop after the right number of foldings. Estimate the answer using the approximate equality of 2^10 and 10^3 and compare with your numerical result.

8. Newton's method for rootfinding

The iterated function approach provides a powerful method for numerically solving an equation of the form $g(x)=0$, provided that we can make a good guess at the solution and can explicitly differentiate the function $g(x)$. The typical situation is shown below:



The root x* we are seeking is, in the figure, the point where the graph of $g(x)$ crosses the x axis. Suppose that we have guessed the value x_1. We can improve on this value by considering the straight line through $(x_1, g(x_1))$ which is tangent to the graph of $g(x)$ at x_1 (slope=$g'(x_1)$) and calculating the point x_2 where the straight line intersects the x axis, namely

$$x_2 = x_1 - \frac{g(x_1)}{g'(x_1)} \ .$$

The new value can be improved upon by the same procedure, and in fact we can keep repeating it until the increment in xn is below the desired accuracy threshold. It is easy to reformulate the process as an iterated mapping ("Newton's map") :

$$x_{n+1} = f(x_n) \ ,$$

where

$$f(x) = x - \frac{g(x)}{g'(x)} \ ,$$

with

$$x^* = \lim_{n\to\infty} x_n \ .$$

In typical cases, the convergence is extremely rapid, usually more rapid than other systematic rootfinding methods. The price one pays is that the method doesn't always work. For example, it is not always possible to write down explicitly the derivative of a function. And if the first guess is far from the zero of the function, the function f may map you to a point *further away* from x^* than the original point (like the point $x_{??}$ in Fig. 3.7).

In Exercise 5 we introduced the polynomial

In[17]:=
```
p[x_] := x^4 + 3 * x^3 - 2 * x^2 - 10 * x - 1
```

and used a "trial and error" method to find its largest root. Let us define explicitly Newton's map associated with this function:

In[18]:=
```
f[x_] := x - p[x] / (4 * x^3 + 9 * x^2 - 4 * x - 10)
```

Guessing the position of the largest root from Fig.2.1 to be 1.6, we apply f iteratively:

```
In[19]:= x = 1.6;
         TableForm[Table[x = f[x], {6}]]
```

```
Out[20]//TableForm=
         1.7423905489923557`
         1.723655811175423`
         1.7232888209165893`
         1.7232886816569`
         1.72328868165688`
         1.72328868165688`
```

Note that the output originally showed only 6 significant digits. The remaining digits were revealed by converting to input form (via the "Cell/Convert To/InputForm Display" menu item). We see that iteration of Newton's map have converged after 5 iterations to a result with 15-digit precision. If we want (say) 30-digit precision, we must start with a more precise initial condition:

```
In[21]:= x = SetPrecision[16 / 10, 40];
         TableForm[Table[x = f[x], {7}]]
```

```
1.7423905489923558026407227241139680333 56
1.7236558111754229885320809003757432844 4
1.7232888209165892212388635249921206882 2
1.7232886816569000795514254637311313473
1.7232886816568800317552026684964751982
1.7232886816568800317552026680809962 37
1.7232886816568800317552026680809962 37
```

As expected, one additional iteration of Newton's map is sufficient for convergence (with 37 significant digits).

Exercise 10. Newton's method I

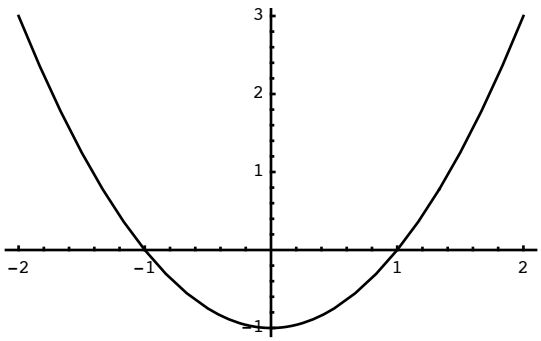
1) Find the second real root of p(x), to 18-digit precision, by the same method used above for the first root.

2) Try to determine, by reasoning and perhaps some trial and error, which points on the real line are attracted to each of the real roots. These are called the *basins of attraction* of the two roots. Can you find any points which lie outside the basins of attraction of both real roots? Are the real zeroes of p'(x) in this category? If so, are they the only ones?

To see why Newton's method works so well, let us consider a simpler example,

$$p(x) = x^2 - 1 \ .$$

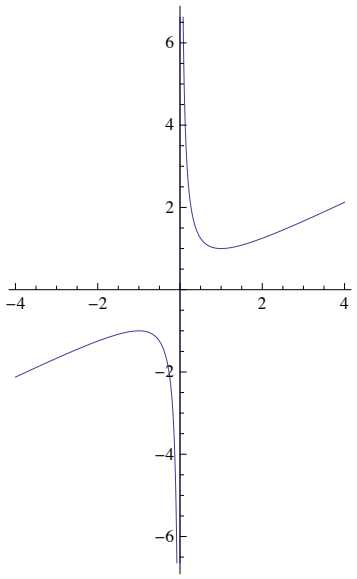
The graph of this function is shown below, where one can see immediately the positions of the two zeroes at $x = 1$ and $x = -1$.



Now let us construct Newton's map,

$$f(x) = x - \frac{p(x)}{p'(x)} = \frac{x^2+1}{2x} ,$$

which is graphed below.



You can easily differentiate $f(x)$ to get

$$f'(x) = \frac{x^2-1}{2x^2} , \qquad f''(x) = \frac{1}{x^3} .$$

from which it follows (consistent with the appearance of the graph) that the function $f(x)$ has a local maximum at $x = -1$ and a local minimum at $x = 1$. Let us zoom in on a small neighborhood of $x = 1$ to check the stability of the fixed point of $f(x)$ there. Since the function is behaving like $(x-1)^2$ inear the fixed point x=1 , $f(1.04) - 1$ is approximately $(1/2)\,(.04)^2 = .0008$, and by the same reasoning, $f(1.0008) - 1 = 0.00000032$, and $f(1.00000032) - 1 = 0.000\,000\,000\,000\,05$. It is clear that the secret of Newton's method's success lies in the vanishing of the derivative of Newton's map at the fixed point. Each time the function f is applied, the already tiny error is squared, becoming extremely tiny. That extremely tiny error is then squared, becoming absurdly tiny, etc.,etc. We can check that the desired flatness of the Newton map is almost always true:
If $f(x) = x - p(x)/p'(x)$, then $f'(x) = p(x)p''(x)/p'(x)^2$, which is equal to zero at any zero of $p(x)$, provided that it is a *simple* zero where the slope of the graph $p'(x)$ is not zero. A modified version of Newton's method can be used for higher order zeroes, but we shall not pursue that issue here.

Exercise 11. Newton's method II

1) Calculate the zero of $\sin x$ near $x = 3.0$ with 15 significant figures using Newton's method, i.e. by calculating the derivative of the function, defining Newton's mapping as a *Mathematica* function, and applying the function

iteratively starting with a value of x known to be close to the exact root. See if you can automate the process using the **Do** function, using **Print** to print out the result after each iteration. Use the built-in function **Sin[x_]**. Compare your result with **N[Pi,18]**. Plot the Newton mapping and see that it has a flat point at the expected value of x.

2) Calculate the natural logarithm of 1000 to 15 significant figures by finding the zero of the function e^x-1000. Again you are to construct Newton's map and find the result by iteration. Use **Plot** to help you choose your initial value. Use the built-in function **Exp[x_]** or the built-in constant **E**. Compare your result with **N[Log[1000],18]**. Plot the Newton mapping and see that it is flat at the expected value of x.

---

9. Integrating differential equations

It is typical in the theoretical description of a moving particle to be given its initial state (position and velocity) and a set of differential equationsof motion. Integration of the differential equations is generally not possible in exact form. However, excellent approximation methods exist which, with the help of computers, make possible the *numerical integration* of the equations. We will restrict our attention to a simple and reliable method of numerical integration, namely the Runge-Kutta scheme, which works as follows.

We consider the case of a particle restricted to the x axis, with position at time t determined by the differential equation

$$\frac{dx}{dt} = f(x) \;, \quad \text{with} \quad x(0)=x_0 \;.$$

We want to approximate the continuous orbit $x(t)$ by a sequence x_n, n=0,1,2,... of approximate positions at times $t_n = n\,dt$, where dt is small and we want the difference x_{n+1}- $x_n$to coincide with the Taylor series expansion up to term of order dt^4:

$$x_{n+1} - x_n = x(t_{n+1}) - x(t_n) = \dot{x}(t_n)\,dt + \frac{1}{2!}\ddot{x}(t_n)\,dt^2 + \frac{1}{3!}\dddot{x}(t_n)\,dt^3 + \frac{1}{4!}\ddddot{x}(t_n)\,dt^4 + O\left(dt^5\right)$$

where the dots denote time-differentiation. The Taylor coefficients can be expressed as known functions of x by applying the differential equation:

$$\dot{x} = f(x)\,,$$
$$\ddot{x} = f'(x)\,f(x)\,,$$
$$\dddot{x} = f''(x)\,(f(x))^2 + (f'(x))^2\,f(x)\,,$$
$$\ddddot{x} = f'''(x)\,f(x)^3 + 4\,f''(x)\,f'(x)\,f(x)^2 + (f'(x))^3\,f(x)\,.$$

According to the Runge-Kutta prescription, we calculate the sequence of x_n by iterating

$$x_{n+1} = \text{RK4step}[x_n]$$

with RK4step defined by

In[22]:=
```
RK4step[f_,dt_,z_]:= Module[{k1,k2,k3,k4},
                     k1=f[z]*dt;
                     k2=f[z+k1/2]*dt;
                     k3=f[z+k2/2]*dt;
                     k4=f[z+k3]*dt;
                     z+(k1+2*k2+2*k3+k4)/6]
```

Here the function **Module** is used as a "wrapper" (instead of parentheses) for the function definition. The advantage is that the variables **k1**, **k2**, **k3**, **k4** are strictly localized within the definition, so that calling **RK4step** will not disturb any assignments which you may previously have made to global variables with the same names. The same is of course true of the dummy variables **f**, **dt**, and **z**.

Mathematica has a very nice built-in function **Series** for calculating Taylor series of functions. For example, to compute the Taylor series of sin(ϵ) about the point 0, to O(ϵ^{18}), we evaluate

In[23]:=
```
Series[Sin[ϵ], {ϵ, 0, 18}]
```

Out[23]=
$$\epsilon - \frac{\epsilon^3}{6} + \frac{\epsilon^5}{120} - \frac{\epsilon^7}{5040} + \frac{\epsilon^9}{362\,880} - \frac{\epsilon^{11}}{39\,916\,800} + \frac{\epsilon^{13}}{6\,227\,020\,800} - \frac{\epsilon^{15}}{1\,307\,674\,368\,000} + \frac{\epsilon^{17}}{355\,687\,428\,096\,000} + O[\epsilon]^{19}$$

Let us test the Runge-Kutta functions to see whether they succeed in reproducing the Taylor series up to 4th order in *dt* .

In[24]:=
```
Series[RK4step[f,dt,x],{dt,0,4}]
```

Out[24]=
$$x + f[x]\,dt + \frac{1}{2} f[x]\,f'[x]\,dt^2 + \frac{1}{6}\left(f[x]\,f'[x]^2 + f[x]^2 f''[x]\right) dt^3 + \frac{1}{24}\left(f[x]\,f'[x]^3 + 4\,f[x]^2 f'[x]\,f''[x] + f[x]^3 f^{(3)}[x]\right) dt^4 + O[dt]^5$$

Success!

When we generalize the above analysis to a system of *n* differential equations, the Taylor expansions become much more complicated, involving the full zoo of partial derivatives. There is no need to change the RK4step function, however, only to reinterpret its arguments. The state variable **z** is now an *n*-dimensional array (an *n*-vector), as is the function **f** and the auxiliary variables **k1,k2,k3,k4** in the defining formula. Since the latter involves only operations (addition, multiplication by a scalar) which apply to vectors of any dimension, there is no need to alter the definition in any way. The remarkable fact is that, regardless of dimension, this relatively simple prescription continues to reproduce the terms of the Taylor expansion up to 4th order in the time step. You may want to verify this statement yourself for (say) $n = 2$.

As a relatively simple illustrative example from classical dynamics, let us consider the case of a single particle on the *x*-axis moving under the influence of a potential energy function $V(x) = \frac{5}{2}x^2 + \frac{1}{4}x^4$. The equation of motion can still be written $\frac{dz}{dt} = f(z)$, but now with *z = (x,v)* and

In[25]:=
```
f[{x_, v_}] := {v, -5 x - x^3}
```

Let us calculate numerically the approximate *x,v*-space orbit with a time step *dt*= 0.01 and initial condition *z*(0)= (1,0), from *t*=0 until *t*=10 , requiring 1000 time steps.

In[26]:=
```
dt = 0.01;   z = z0 = {1, 0};
```

In[27]:=
```
xvorbit = Join[{z0}, Table[z = RK4step[f, dt, z], {1000}]];
```

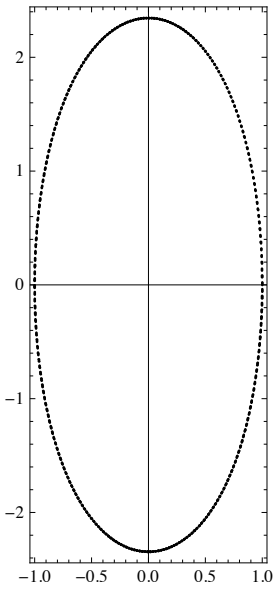
The values of *x*(*t*) and *v*(*t*) over the same time interval are given by

In[28]:=
```
xvals = Table[{(k - 1) * dt, xvorbit[[k, 1]]}, {k, 1, 1001}];
vvals = Table[{(k - 1) * dt, xvorbit[[k, 2]]}, {k, 1, 1001}];
```

We can now display these quantities using **ListPlot:**

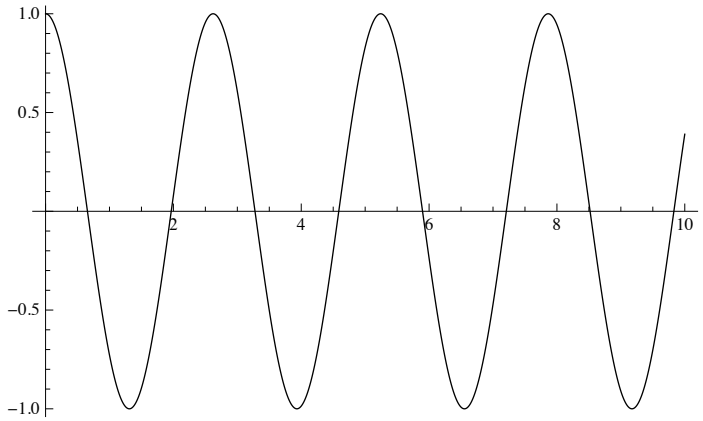
In[30]:=
```
ListPlot[xvorbit, AspectRatio → Automatic,
 PlotStyle → {Black, PointSize[.002]}, Frame → True]
```

Out[30]=



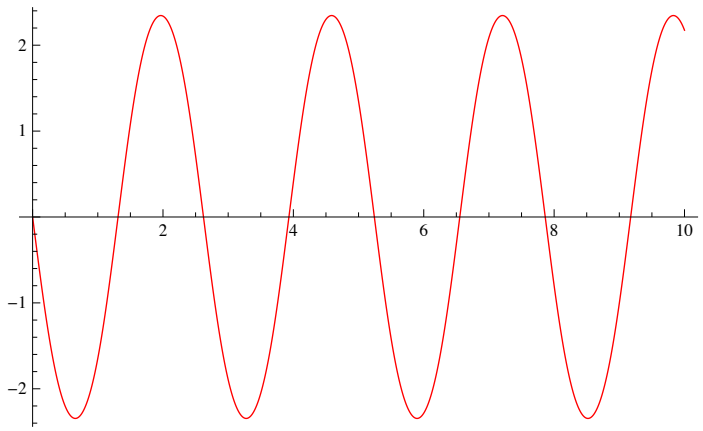
In[31]:=
```
ListPlot[xvals, Joined → True, PlotStyle → {Black, Thickness[.002]}]
```

Out[31]=



In[32]:=
```
ListPlot[vvals, Joined → True, PlotStyle → {Red, Thickness[.002]}]
```

Out[32]=



Now suppose we are interested in an accurate value for x(5). Let us calculate this using different sizes for the timestep.

In[33]:=
```
TableForm[Table[z = {1, 0}; dt = N[10^(-k - 1)];
  Do[z = RK4step[f, dt, z], {5 * 10^(k + 1)}]; z[[1]], {k, 1, 5}]]
```

```
0.8303583916750513`
0.8303584148097363`
0.8303584148119723`  (* converted to Input Form via Cell menu *)
0.8303584148119801`
0.830358414812048`
```

It seems we are running into the effects of round-off errors as we increase the number of iterations. Let us repeat the calculations with higher-precision data.

In[34]:=
```
TableForm[Table[z = {1, 0}; dt = N[10^(-k - 1), 40];
  Do[z = RK4step[f, dt, z], {5 * 10^(k + 1)}]; z[[1]], {k, 1, 5}]]
```

Out[34]//TableForm=
```
0.8303583916750530236708151454211521
0.8303584148097332778721690880249257
0.8303584148119724270345425217938646
0.8303584148119726502003824188680016
0.8303584148119726502226915062673201
```

The above numerical experiment shows that the number of significant digits in our value for $x(5)$ increases by 4 for each reduction of dt by a factor of 1/10. This is exactly as it should be for a 4th-order scheme. It allows us to conclude that our best value has an error of order 10^{-23}.

10. The functions `Manipulate` and `Animate`

One of the most useful innovations in recent versions of *Mathematica* is the inclusion of the functions **`Manipulate`** and **`Animate`** , which allow us to display very effectively the parameter dependence of our results. For example, suppose we wish to visualize the effect of changing the initial position $x(0)$ on the function $x(t)$, for $0 \leq t \leq 5$. We evaluate

In[35]:=
```
dt = 0.01; Animate[z = {a, 0};
 ListPlot[Table[z = RK4step[f, dt, z]; {k * dt, z[[1]]}, {k, 1, 500}],
  Joined → True, PlotStyle → {Black, Thickness[.002]}], {a, 0, 5}]
```

Out[35]=