

Mathematica[®] notes, Physics 545
University of North Dakota

Bill Schwalm
Fall, 2002

Introduction	1
Stepwise calculations.....	1
Symbolic expressions.....	2
Suppression of output	5
Simple plot.....	5
Lists and list operations.....	6
Getting list parts.....	7
Matrices as lists.....	8
Det, eigenvalues/vectors.....	9
Dot, Table	10
Matrix vector product	11
Matrix matrix product, Sum.....	12
Inverse.....	13
Roots.....	14
Logical expressions, list programming..	15
List operations	16
PDEs with RotateRight/Left.....	17
Accessing expressions as lists.....	18
Infix operations, substitution rules	19
Functions and programming	22
Writing functions.....	23
If, Drop, sig example	24
Map, Clear, Modules	25
makePretty example.....	26
spherint example.....	27
Green function example.....	30
Anonymous functions.....	31
Vector fields and Lie algebras example.	33
Korteweg-de Vries soliton example.....	37
Rational expression modulo polynomial	40

Mathematica is a registered trademark of Wolfram Research.

Mathematica Introduction, Phys. 545 Classical Mechanics

Start *Mathematica* by clicking or double clicking on the icon. You get several windows, usually. There is a menu bar on the top and the largest window (usually) is workspace. At first, the work space is labeled “Untitled 1.” At any time, if you want to save your work (for example as a *Mathematica* notebook) then use the *File* menu and choose *save as* to select a name for your notebook and a place to put it. After you do that, the label on the workspace will change to reflect whatever name you chose for your file. Personally, I almost never save workbooks. Usually, I save program files that define functions, or short files containing intermediate results.

These short notes will not deal much with how to operate the computer or the *Mathematica* front end. For one thing you probably know more than I do about the computer front end. Mostly I will write about how to use the language to do mathematics.

In addition, I will ask you to get hold of the big *Mathematica* book; for instance you can check one out from the desk in the mathematics lab down stairs, as long as you read it in the computer room. Of course, that’s the best way to read the book: right beside the computer with *Mathematica* running. Thus, I will not write very much about the introductory material. Rather, I will ask you to read Part 1, “A Practical Introduction to *Mathematica*,” especially the first few sections. Try to work through the examples on the computer.

Stepwise calculation protocol: Now, suppose you have started *Mathematica*, and you want to compute something. So far the workspace is blank. Here are three general ideas:

1. *Mathematica* will try to evaluate anything you enter as soon as you enter it.
2. You “enter” an expression by using both the shift and the “enter” keys together, where “Enter” is on the main part of the keyboard. Enter alone will continue on to the next line.
3. After you enter an expression, the expression you entered will get an input number, and the result that comes back will be given an output number.

So, suppose I type

$$5 + 3$$

and then press shift and enter together. (The first entry takes a few seconds for loading the kernel.) The input gets numbered [1] and an output comes back, so the screen now says

```
In[1] := 5 + 3
```

```
Out[1]= 8
```

Multiplication can be either as a star * as in Fortran, or a space between symbols, or using parentheses.

```
In[2] := 5 3
Out[2] = 15

In[3] := 5*3
Out[3] = 15

In[4] := 5(3)
Out[4] = 15
```

Division is done with a slash or soldus, as 5/3. For integers, *Mathematica* doesn't evaluate the fraction, but just treats it algebraically. For real numbers, the division evaluates to a decimal fraction (*i.e.* to a real number).

```
In[5] := 5/3
Out[5] =  $\frac{5}{3}$ 

In[6] := 5./3
Out[6] = 1.66667
```

In the latter example the decimal point in the 5. converts the 5 to a real number, so the whole expression evaluates as a real number.

The usual transcendental number π is treated algebraically, unless you convert it into a real approximation using the N function. This function evaluates an expression as real, as far as this is possible. A second argument of N will choose how many decimal places you want. In principle, *Mathematica* can handle any number of decimal places correctly.

```
In[7] := Pi
Out[7] =  $\pi$ 

In[8] := N[Pi]
Out[8] = 3.14159
```

```
In[9] := N[Pi,20]
Out[9] = 3.1415926535897932385
```

Here is a good place to digress on the different kinds of brackets and how they are used. Parentheses () are used to group terms only, thus

```
In[10] := 1 + (3 5)
Out[10] = 16

In[11] := (1 + 3) 5
Out[11] = 20
```

If you neglect to type the parentheses, the first result is obtained. The priority of operations is as in Fortran.

The square brackets [] are used for functions. Thus in the example above, the arguments of the N function are written inside single square brackets. Note too that the input and output lines appear as functions. Some other examples are

```
In[12]:= Sin[Pi]      In[13]:= Sin[23]
```

```
Out[12]= 0           Out[13]= Sin[23]
```

Notice that in the last example, *Mathematica* did not evaluate the sine, since it didn't know what you wanted. In many ways it is better to know the answer is the sine of 23, rather than to know an approximate real value. Of course, you can get a real number if you want

The built-in functions of *Mathematica* all begin with a capital letter, then they are usually spelled with lower case letters, except when there is more than one word. Examples are:

```
In[14]:= Sin[23.]      In[15]:= N[Sin[23],20]
```

```
Out[14]= -0.84622      Out[15]= -0.84622040417517063524
```

Sin, Cos, Sqrt, Exp, Log, BesselJ, LegendreP, Tan, ArcTan, PolynomialRemainder, Denominator, etc. Notice that Log is the natural logarithm. *Mathematica* also accepts Ln. Also, the letter E is reserved to mean *e* the base of natural logarithms. Hence the usual way to write Exp[x] would be E^x. (Notice that the carrot ^ is for exponentiation.)

Of course you know how to get numerical values from these by now.

```
In[16]:= Exp[-5]      In[17]:= E^(-5)
```

```
Out[16]=  $\frac{1}{e^5}$       Out[17]=  $\frac{1}{e^5}$ 
```

Now turn to symbolic expressions. You can enter an expression such as

```
x^2 (1-x^2)^5
```

and then press shift and enter simultaneously, as usual, and you get

```
In[18]:= x^2 (1-x^2)^5
```

```
Out[18]= x^2 (1 - x^2)^5
```

Now why didn't *Mathematica* expand the parenthesis? The answer is: Because you didn't tell it to. *Mathematica* has no way of knowing whether or not you want to expand the parenthesis. The expression is more compact, hence "simpler," if you don't expand. So, let's expand the previous expression using the Expand function. You could retype the expression. But there is an easier way. *Mathematica* lets you refer to the most recent

output as %. You can also refer to the specific output by its number. You can either call it %18, or else Out[18]. In either case, this uses the Out function to look up the appropriate output.

```
In[19] := Expand[%]
```

```
Out[19] = x2 - 5x4 + 10x6 - 10x8 + 5x10 - x12
```

You can go the other way using the Factor function. (Factor is the most powerful tool in computer algebra. The computer can factor expressions that are ten pages long in a few seconds to a minute.)

```
In[20] := Factor[%]
```

```
In[21] := FactorSquareFree[%19]
```

```
Out[20] = -(-1+x)5x2(1+x)5
```

```
Out[21] = -x2(-1+x2)5
```

Both Factor and FactorSquareFree can be applied to any polynomial or rational expression. Remember that a rational expression is the ratio of two polynomials.

So far we have been referring to expressions by their output numbers, or by %. But you can assign them to a variable name, just as you would in any other computer language. For instance, I can use the name K to refer to the expression on output line 19.

```
In[22] := K = Out[19]
```

```
Out[22] = x2 - 5x4 + 10x6 - 10x8 + 5x10 - x12
```

Now, whenever I refer to K the computer will operate on this expression, unless I redefine K. If you want to undefine K, that is, to put it back to just the letter K, you can use K=. or else Clear[K]. So, suppose I want to differentiate the expression K with respect to x, and then divide the result by 1+K. The result will define a new expression R.

```
In[23] := R = D[K,x] / (1+K)
```

```
Out[23] = 
$$\frac{2x - 20x^3 + 60x^5 - 80x^7 + 50x^9 - 12x^{11}}{1 + x^2 - 5x^4 + 10x^6 - 10x^8 + 5x^{10} - x^{12}}$$

```

Of course, you could now use Factor to simplify this expression, differentiate it again with respect to x. Notice that D[y,x] means $\partial y / \partial x$. It is sometimes useful to get the numerator or denominator of a rational expression. So for example

```
In[24] := Denominator[%23]
```

```
Out[24] = 1 + x2 - 5x4 + 10x6 - 10x8 + 5x10 - x12
```

Then suppose you would like to have the indefinite integral of this result.

```
In[25] := Integrate[%23, x]
```

```
Out[25] = Log[1 + x^2 - 5x^4 + 10x^6 - 10x^8 + 5x^10 - x^12]
```

Sometimes you know ahead of time that a result will be too complicated to look at. In that case, you can choose not to look at the result. For example, suppose you want to set the name M to contain $\sqrt{1+K^2}/\sqrt{1-K^2}$. This will be a bit of a mess to look at, so I will use a semicolon (;) to suppress the output. Then the output isn't written to the screen, but it is still computed.

```
In[26] := Sqrt[1+K^2]/Sqrt[1-K^2];
```

If you want to define M using the result, you can just type

```
M = %;
```

or else type

```
M = Out[26];
```

In the latter case, for instance, all the computer will show is

```
In[27] := M = Out[26];
```

Even though Out[26] or Out[27] did not print on the screen, the name M will be set equal to the result. *Mathematica* has kept track of Out[26] even though it didn't type it. If you change your mind and decide you do want to look at it, you can now type M, with shift return, and get

```
In[28] := M
```

```
Out[28] = 
$$\frac{\sqrt{1 + (x^2 - 5x^4 + 10x^6 - 10x^8 + 5x^{10} - x^{12})^2}}{\sqrt{1 - (x^2 - 5x^4 + 10x^6 - 10x^8 + 5x^{10} - x^{12})^2}}$$

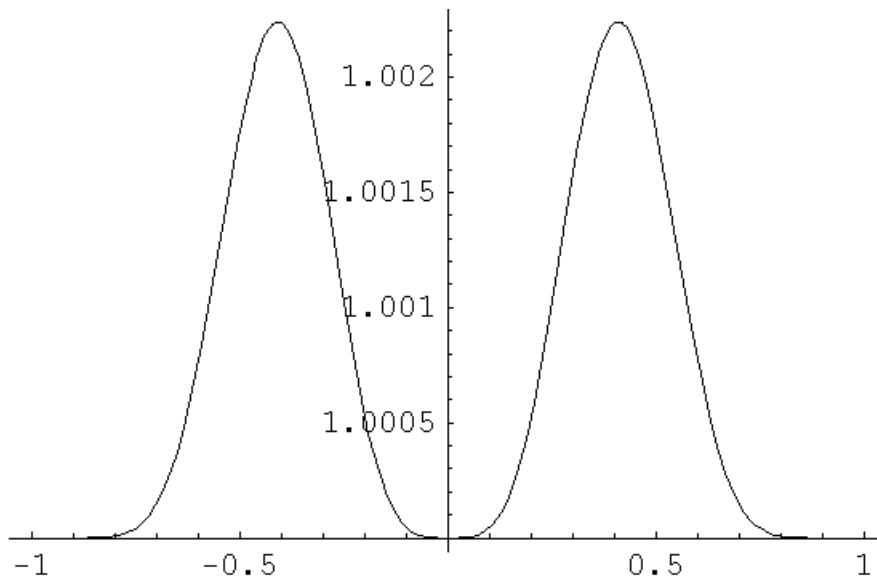
```

Next suppose I would like to plot the numerator of M from $x = -1$ to $x = 1$. First I will store the numerator in a variable that I choose to call num. Then I plot num from $x = -1$ to $x = 1$ in the following way.

```
In[29] := num = Numerator[M];
```

```
In[30] = Plot[num, {x, -1, 1}]
```

Then you get a graph on the screen, and an output that says -Graphics-. The output can be used to manipulate the graph as a graphics object. We'll see more about that later. For now, just notice the output, which should look like



```
Out[30]= -Graphics-
```

If you were to type in `Show[%30]` at some later point it would cause this graph to be redrawn. Of course there are ways to add axes labels and scales and to put a title on the graph, *etc.* However, for now, you can see that *Mathematica* will be rather useful for making graphs for your research and for course work. You should look through the *Mathematica* book at this point paying particular attention to read about graphics.

Lists and List Operations

Now we come to one of the three main points about *Mathematica*. The three main points are that *Mathematica* supports list operations (like Fortran 90 and Matlab), *Mathematica* is a term re-write system, and finally, *Mathematica* supports functional programming (like Pascal and C). Functions can call each other and even call themselves recursively, which gives you a very powerful tool for writing simple programs that do difficult work. Let us put off term re-writing and the use of functions until later on. For now, let's take a first look at lists and list operations.

From a mathematical point of view, a list is a set. We write lists enclosed in set brackets `{}`.

```
In[31]:= aa={s,p,q,r,x,y,z}
```

```
Out[31]= {s,p,q,r,x,y,z}
```

Then the variable aa contains a list of seven letters, or seven simple variables. To get the third entry from the list, you use double square brackets.

```
In[32]:= aa[[3]]
```

```
Out[32]= q
```

It is interesting that you can use a list as a subscript for a list too. This lets you pick out a certain permuted subset of a set very easily. For example,

```
In[33]:= aa
```

```
Out[33]= {s,p,q,r,x,y,z}
```

```
In[34]:= aa[{{1,2,3}}]
```

```
Out[34]= {s,p,q}
```

```
In[35]:= aa[{{1,5,3,4,5,1,2,6}}]
```

```
Out[35]= {s,x,q,r,x,s,p,y}
```

You can see that this would make an interesting way to study groups, for example, where “multiplication” can be represented by a permutation.

The next interesting thing about lists is that a list can contain lists also as elements. So for example consider

```
In[36]:= bb = {1,2,x,4,1-x^2,{1,2,3},x,{3,{x,y}}}
```

```
Out[36]= {1,2,x,4,1-x^2,{1,2,3},x,{3,{x,y}}}
```

Then, for instance, take the sixth element in the list.

```
In[37]:= bb[[6]]
```

```
Out[37]= {1,2,3}
```

This of course is the right answer, since the sixth entry in bb is the list {1,2,3}. Here is another point. Let’s take the second entry of the sixth entry of bb.

```
In[38]:= bb[[6]][[2]]
```

```
Out[38]= 2
```

Is this correct? Now, here is another way to do the same thing

```
In[39] := bb[[6,2]]
```

```
Out[39]= 2
```

The way the subscripting works is that the first index chooses the position in the outermost list, or “first level” of the list. The second subscript chooses the position in the second level, and so on, provided of course that there is a second level. If you ask of an element that does not exist, *Mathematica* will tell you. One convenient thing is that you can use negative numbers to access the back end of a list. Hence,

```
In[40] := bb[[-1]]
```

```
Out[40]= {3, {x, y}}
```

So, if I want the first entry of the second entry of the last entry of list bb, I write

```
In[41] := bb[[-1,2,1]]
```

```
Out[41]= x
```

We can get at the last entry in bb another way. That is, we find the length of bb first and then use it to get the entry itself.

```
In[42] := Length[bb]
```

```
Out[42]= 8
```

```
In[43] := bb[[8,2,1]]
```

```
Out[43]= x
```

Now you might wonder why one should spend so much time thinking about lists. There are two reasons. First, one of the most efficient computing techniques is to use list operations. For computers with the right kind of architecture, these correspond to vector or parallel operations. Rather than using “do loops” to operate sequentially on the entries of a list, you operate on the whole list all at once. This programming style is easier to understand, quicker to program and, usually, makes programs that are more efficient and run much faster. The other reason we are interested in lists is that at a fundamental level, every data structure in *Mathematica* is really just a list. Vectors are obviously lists, sets are lists, matrices are lists (lists of lists, actually) algebraic expressions are lists. Everything is really a list. When you know how to operate with lists, you know how to do a lot of things automatically.

So, suppose I want to work with a square matrix A, which I enter as

```
In[44] := A = {{1,0,0,-1},{0,-1,1,0},{0,1,-1,0},{-1,0,0,1}}
```

```
Out[44]= {{1,0,0,-1},{0,-1,1,0},{0,1,-1,0},{-1,0,0,1}}
```

This is obviously a list, so one can get entries of it as usual. For example $A[[1,2]]$ will be 0, while $A[[2,2]]$ is -1, and so on. So we have constructed a matrix by making a list of lists, where the inside lists are the rows of the matrix. *Mathematica* knows how to deal with this special type of two-subscripted list as a Matrix. For instance

```
In[45] := MatrixForm[A]
```

$$\text{Out[45]} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix}$$

Matrix form is a display form. It is good for looking at the matrix (if it isn't too big) either on the screen or for printing it. So far, A itself is still the standard list-of-lists form, which is the best way to keep it for mathematical purposes. Notice that A is symmetric, so if one were to take the transpose it would look the same.

```
In[46] := MatrixForm[Transpose[A]]
```

$$\text{Out[46]} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix}$$

The determinant is just

```
In[47] := Det[A]
```

```
Out[47] = 0
```

So we see that A is singular. Often one would like the eigenvalues, or eigenvectors.

```
In[48] := Eigenvalues[A]
```

```
Out[48] = {-2, 0, 0, 2}
```

Indeed, one is not surprised to see that the eigenvalues are real, since A is symmetric, and at least one is zero, since A is singular. The eigenvectors should be orthogonal too.

```
In[49] := Eigenvectors[A]
```

```
Out[49] = {{0, -1, 1, 0}, {1, 0, 0, 1},
           {0, 1, 1, 0}, {-1, 0, 0, 1}}
```

The eigenvectors appear as the rows of the output matrix. If you want the second eigenvector, then

```
In[50]:= V = %;
```

```
In[51]:= V[[2]]
```

```
Out[51]= {1, 0, 0, 1}
```

Now we know that, since the matrix A is symmetric (and also real, so it is Hermitian) the eigenvectors should be mutually orthogonal. If you form dot products between different eigenvectors you should get zero. In other words, the eigenvectors should be perpendicular to one another. In *Mathematica*, a dot product is done either with the Dot function or else (infix form) with a “dot” *i.e.* (.) a period.

```
In[52]:= Dot[V[[2]],V[[3]]]
```

```
In[53]:= V[[2]].V[[3]]
```

```
Out[52]= 0
```

```
Out[53]= 0
```

To check that the eigenvectors are orthogonal in pairs, lets construct a table (which will be another matrix, in this case) of dot products. This will utilize the Table function to make a list. So first, let's do an example of the use of Table. Suppose I want a table of powers of x, from the zeroth to the 10th power.

```
In[53]:= Table[x^n, {n, 0, 10}]
```

```
Out[53]= {1, x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^10}
```

The Table function has two arguments. The first tells what to put in the table, and the second is a range specification for the iterator n. A comma separates the two arguments. Just for safety sake, be sure the iterator n is “blank,” or free of any definition outside the loop.

Or, suppose you need a matrix consisting of powers of x and y, thus

```
In[54]:= Table[x^m y^n, {m, 0, 2}, {n, 0, 2}];
```

```
In[55]:= MatrixForm[%]
```

```
Out[55]= 
$$\begin{bmatrix} 1 & y & y^2 \\ x & x y & x y^2 \\ x^2 & x^2 y & x^2 y^2 \end{bmatrix}$$

```

Thus, we can check that the eigenvectors of A are orthogonal by tabulating dot products as follows:

```
In[56]:= Table[V[[m]].V[[n]],{m,1,4},{n,1,4}];
```

```
In[57]:= TableForm[%]
```

```
Out[57]=
```

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

So, yes, the eigenvectors of A are mutually orthogonal. Notice, though, that *Mathematica* doesn't normalize the eigenvectors. It keeps convenient integer coefficients wherever possible. (For a numerical matrix, one where the entries are decimal approximate real numbers, the eigenvectors are usually returned in normalized form.)

Now we have seen how Dot works for two vectors. (By the way, when you dot two complex vectors, the *Mathematica* function Dot or (.) does not do any complex conjugation. You must take care of that yourself. There are, however, many possible extensions to the Dot function. See book.) The next thing to look at is the use of Dot for matrix-times-vector and matrix-times-matrix product operations.

Remember that when you type a vector v in as a list, it appears as a row vector. Also, a matrix A appears as a set of row vectors. If you dot A into v, like (A .v.), what *Mathematica* does is to dot each row of A in turn into the row vector v, and then make a table. The table appears again as a row vector. A schematic diagram of this process is as follows

$$\begin{bmatrix} \vec{a}_1 \rightarrow \\ \vec{a}_2 \rightarrow \\ \vec{a}_3 \rightarrow \end{bmatrix} \cdot [\vec{v} \rightarrow] = [\vec{a}_1 \cdot \vec{v}, \vec{a}_2 \cdot \vec{v}, \vec{a}_3 \cdot \vec{v}]$$

And so one sees this is very much like matrix multiplication. It would be matrix multiplication if you viewed the vectors on the right and left as columns rather than rows. Thus in this particular case, if you interpret the vectors as column vectors, the schematic diagram is

$$\begin{bmatrix} \vec{a}_1 \rightarrow \\ \vec{a}_2 \rightarrow \\ \vec{a}_3 \rightarrow \end{bmatrix} \cdot \begin{bmatrix} \vec{v} \\ \downarrow \end{bmatrix} = \begin{bmatrix} \vec{a}_1 \cdot \vec{v} \\ \vec{a}_2 \cdot \vec{v} \\ \vec{a}_3 \cdot \vec{v} \end{bmatrix}$$

This is just the usual pattern of matrix-times-(column) vector multiplication.

The diagrams above indicate that the *Mathematica* recipe $u = A . v$ should correspond in matrix multiplication to $u^+ = A v^+$ where dagger represents transpose. The dagger (+) changes row vectors into column vectors.

$u = A.v$	$u^+ = A v^+$
Mathematica input form	Matrix-vector form, where u, v are row vectors

Now let us think about the general problem of matrix-matrix multiplication.

It is clear that matrix multiplication can be done using the Dot function. However the way to do this is not so clear. Thus before showing how to matrix multiply using dot products, let me show how to do it using Table and Sum. This is more like the way you matrix multiply in Fortran. It involves three loops, one to do the sum and two others to step over the rows and columns of the product.

The Sum function works as follows:

```
In[58]:= Sum[x^n, {n, 0, 5}]
```

```
Out[58]= 1 + x + x^2 + x^3 + x^4 + x^5
```

So, suppose we have two rectangular matrices

```
In[59]:= A = {{1, 0, 3, 4}, {-1, 1, 2, 1}};
```

```
In[60]:= B = {{2, 7}, {1, 0}, {0, -1}, {2, 3}};
```

```
In[61]:= MatrixForm[A]
```

```
Out[61]= 
$$\begin{bmatrix} 1 & 0 & 3 & 4 \\ -1 & 1 & 2 & 1 \end{bmatrix}$$

```

```
In[62]:= MatrixForm[B]
```

```
Out[62]= 
$$\begin{bmatrix} 2 & 7 \\ 1 & 0 \\ 0 & -1 \\ 2 & 3 \end{bmatrix}$$

```

```
In[63]:= Table[Table[Sum[A[[m, k]] B[[k, n]],  
{k, 1, 4}], {n, 1, 2}], {m, 1, 2}];
```

```
In[64]:= MatrixForm[%]
```

$$\text{Out}[64]= \begin{bmatrix} 10 & 16 \\ 1 & -6 \end{bmatrix}$$

That was the dumb way to multiply matrices. But at least the answer is correct, since it obeys the definition of matrix multiplication. Now compute

```
In[65]:= A.B;
```

```
In[66]:= MatrixForm[%]
```

$$\text{Out}[64]= \begin{bmatrix} 10 & 16 \\ 1 & -6 \end{bmatrix}$$

So you see, the *Mathematica* operation (A . B) corresponds to the simple matrix product A B. Thus, all of matrix algebra becomes rather simple in *Mathematica*. Take for

A.B

A B

Mathematica
input form

corresponds to
matrix product

example the matrix A defined as

```
In[65]:= A = {{2,3,1},{3,2,1},{1,1,2}};
```

```
In[66]:= MatrixForm[A]
```

$$\text{Out}[66]= \begin{bmatrix} 2 & 3 & 1 \\ 3 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

```
In[67]:= Ain = Inverse[A];
```

```
In[68]:= MatrixForm[Ain]
```

$$\text{Out}[68]:= \begin{bmatrix} -\frac{3}{8} & \frac{5}{8} & -\frac{1}{8} \\ \frac{5}{8} & -\frac{3}{8} & -\frac{1}{8} \\ -\frac{1}{8} & -\frac{1}{8} & \frac{5}{8} \end{bmatrix}$$

```
In[69]:= MatrixForm[Ain.A]
```

$$\text{Out}[69] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Evidently *Mathematica* is very handy for matrix calculations.

Warning: ☹ It is best not to try to work with large matrices symbolically, unless you take special care. You can use very large matrices numerically. Use the N function to be sure the numbers are converted to real.

Here is a note on complex expressions in *Mathematica*. You cannot do very much algebra without encountering complex numbers. The standard square roots of -1 in *Mathematica* is input as I, and is output as i. The fact is, *Mathematica* does not handle complex numbers very well. In particular, it is difficult to get the real and imaginary part of an expression, even though the functions Re and Im are supposed to do this. Later on we'll see some other ways to handle complex numbers. But for right now, just be aware that the letter I is usually used to input the square root of minus 1.

So *Mathematica* will complain if you try to use the letter I to contain an identity matrix. I often use id for the identity matrix. Let's compute the characteristic polynomial of the matrix A above.

```
In[70]:= id = IdentityMatrix[3];
```

```
In[71]:= MatrixForm[id]
```

$$\text{Out}[71] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
In[72]:= p = Expand[Det[x id - A]]
```

$$\text{Out}[72] = 8 + x - 6x^2 + x^3$$

Now let's find the roots of p, which are the eigenvalues of A. This gives us a chance to use the Roots function, and also to see logical expressions.

```
In[73]:= Roots[p==0, x]
```

$$\text{Out}[73] = x == \frac{1}{2}(7 - \sqrt{17}) \parallel x == \frac{1}{2}(7 + \sqrt{17}) \parallel x == -1$$

The double equal (\equiv) appearing in In[73] is a logical operator. In the output you see a logical expression. The double bars ($|$) are the logical “or” operator. The output just says that x is one or the other of three possibilities. Later on we will see how to get substitution rules back for the roots, rather than a logical statement, which will often be more useful.

So far you may feel that, while it is nice to be able to do matrix and vector operations conveniently with lists, it is not clear that there are other advantages to lists or list operations. But in fact lists provide a whole method of programming. Let's look at list programming in general.

First, if you have two lists of simple enough structure, and with the same dimensions, then you can add the lists term by term with one simple addition (Plus). For example

```
In[74] := u = {1, x, x^2, x^3}
```

```
Out[74] = {1, x, x^2, x^3}
```

```
In[75] := v = {y^3, y^2, y, 1}
```

```
Out[75] = {y^3, y^2, y, 1}
```

```
In[76] := w = u + v
```

```
Out[76] = {1+y^3, x+y^2, x^2+y, 1+x^3}
```

Thus lists add term by term in the same way vectors add component by component. This is again handy when dealing with lists as vectors. Matrices of the same shape also add term by term. But list operations also include term-by-term subtraction, multiplication and division. For vectors or matrices with the same dimensions you can do term by term operations with a single statement. For example

```
In[77] := h = u v
```

```
Out[77] = {y^3, x y^2, x^2 y, x^3}
```

```
In[78] := h/w
```

```
Out[78] = { $\frac{1+y^3}{y^3}, \frac{x+y^2}{x y^2}, \frac{x^2+y}{x^2 y}, \frac{1+x^3}{x^3}$ }
```

Moreover, many simple functions in *Mathematica* are *listable*, meaning that when you feed them a list, they act on each member of the list at once. For example

```
In[79] := Sin[h]
```

```
Out[79] = {Sin[y^3], Sin[x y^2], Sin[x^2 y], Sin[x^3]}
```

```
In[80]:= Sin[ {.1, .2, .3, .4, .5} ]
```

```
Out[80]= {0.0998334, 0.198669, 0.29552, 0.479462}
```

The possibility of doing operations like this on a whole list at once makes programming a lot easier, and the resulting programs are also easier to read, more efficient and usually run faster. Several list operations are especially convenient for programming. These include RotateRight, RotateLeft and Reverse functions.

```
In[81]:= sinelist = %
```

```
Out[81]= {0.0998334, 0.198669, 0.29552, 0.479462}
```

```
In[82]:= Reverse[sinelist]
```

```
Out[82]= {0.479462, 0.29552, 0.198669, 0.0998334}
```

```
In[83]:= RotateRight[sinelist]
```

```
Out[83]= {0.479462, 0.0998334, 0.198669, 0.29552}
```

```
In[84]:= RotateLeft[sinelist]
```

```
Out[84]= {0.198669, 0.29552, 0.479462, 0.0998334}
```

The most basic function for creating lists is the Range function

```
In[85]:= Range[5]
```

```
Out[85]= {1, 2, 3, 4, 5}
```

```
In[86]:= Range[-6, 6]
```

```
Out[86]= {-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6}
```

```
In[87]:= Range[-6, 6, 2]
```

```
Out[87]= {-6, -4, -2, 0, 2, 4, 6}
```

The exponentiation $^$ is listable. Thus one can obtain lists like

```
In[88]:= 1 + x^Range[5]
```

```
Out[88]= {1+x, 1+x2, 1+x3, 1+x4, 1+x5}
```

Notice that adding a single quantity, in this case the constant 1, to each entry in the list can also be accomplished by just one operation. It is an exception to the rule that only objects of the same dimensions (or same “shape”) can be added. *Mathematica* “broadcasts” the constant 1, adding it to each list entry. Similarly, since division and partial differentiation *D* are both listable, you can form

```
In[89] := % + D[1/%, x]
```

```
Out[89] = {1 + x -  $\frac{1}{(1+x)^2}$ , 1 + x^2 -  $\frac{2x}{(1+x^2)^2}$ , 1 + x^3 -  $\frac{3x^2}{(1+x^3)^2}$ , 1 + x^4 -  $\frac{4x^3}{(1+x^4)^2}$ }
```

If *Mathematica* were running on a platform with parallel processing, the calculation of individual terms could be divided up amongst separate, independent processors, since these computations are independent from one another. For this reason, list-processing programming methods are very compatible with vector and parallel architecture. You can probably see the advantage they offer to the programmer, since bookkeeping for the individual list entries need not be considered. The programs become easier to read and hence there are fewer programming errors.

Let’s do an example of list programming that is related to solving a nonlinear reaction/diffusion equation in one spatial dimension, namely

$$\frac{\partial u}{\partial t} - D \frac{\partial^2 u}{\partial x^2} - k u^2 = 0 .$$

The usual way to proceed is to divide both the *x*-axis and time up into equally spaced intervals Δx and Δt , where there are *N* different space intervals. One can assume periodic boundary conditions $u_{N+1} = u_1$. Then the thing to do is to start with known initial values of the concentration *u* inside each of *N* boxes and to step forward in time, step-by-step. This can be done by Euler’s method or Runge-Kutta, or by some other method. But in any event, you need some program steps that compute, at each time step, the partial time derivatives at each point x_i using the partial DE shown above.

At each time step, the time derivative at point x_i is given approximately by

$$\left(\frac{\partial u}{\partial t} \right)_i = k u_i^2 + D \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} .$$

So, suppose the concentrations are stored in a vector *u* of length *N*, with periodic boundary conditions such that $u_{N+1} = u_1$. In other words, *u* is a list with *N* entries. Then a single statement prepares a list *ut* of partial time derivatives

```
ut = k u^2 + (D/dx^2) (RotateRight[u]- 2 u + RotateLeft[u])
```

That's absolutely all there is to it. The expression evaluates the whole list of partials all at once. No loops are needed. You would embed a statement like this where needed in your integration routine. Here dx is the spacing of grid points along the x-axis. One might have thought the nonlinear term u^2 would be difficult to deal with, but the programming is trivial. Because of periodic boundary conditions, rotate right and left take care of spatial derivatives.

Strictly speaking, it is permitted to use D in this way, even though D is the name of the derivative function. But I'm superstitious. I would probably use DD or something for the diffusion constant.

One more list-related topic is the following: Mathematica expressions are all really lists. Thus you can get parts of expressions just as you get parts of lists. Suppose you have an expression like

```
In[90]:= f = Sum[(x^n+1) y^n, {n, 0, 3}]
```

```
Out[90]= 2 + (1+x) y + (1+x^2) y^2 + (1+x^3) y^3
```

Then you can access the parts as follows:

```
In[91]:= f[[-1]]
```

```
Out[91]= (1+x^3) y^3
```

```
In[92]:= f[[-1, 1, 2]]
```

```
Out[92]= x^3
```

Remember too that you can index a list with a list, so you can get a range of terms.

```
In[93]:= f[{{1, 2, 3}}]
```

```
Out[93]= 2 + (1+x) y + (1+x^2) y^2
```

```
In[94]:= f[{{1, 4}}]
```

```
Out[94]= 2 + (1+x^4) y^4
```

I wonder what would happen if I asked for $f[[0]]$. There isn't really a zeroth term. Let's see.

```
In[95]:= f[[0]]
```

```
Out[95]= Plus
```

The function named `Plus` is what *Mathematica* uses to represent a sum of terms. The zeroth entry in an expression is called the "head" of the expression, and it tells what kind

of thing the expression is, or in other words, it tells *Mathematica* how to interpret the list. Another way to examine the head of f is

```
In[96]:= Head[f]
```

```
Out[96]= Plus
```

```
In[97]:= Plus[1,x,y,z]
```

```
Out[97]= 1 + x + y + z
```

The sum shown in Out[97] is really stored inside *Mathematica* using Plus, as in In[97]. To get a feel for the internal representation of an algebraic expression, let's take a look at the "full form" of the expression f.

```
In[98]:= FullForm[f]
```

```
Out[98]=
      Plus[2, Times[Plus[1, x], y], Times[Plus[1, Power[x, 2]
      ], Power[y, 2]], Times[Plus[1, Power[x, 3]], Power[y, 3]
      ]]
```

```
In[99]:= f
```

```
Out[99]= 2 + (1+x) y + (1+x2) y2 + (1+x3) y3
```

The "tree form" provides another interesting view of the guts of Mathematica. A list can be thought of as a tree. Tree form shows the nodes and branches. The expression f has a tree that is already too large to be comprehended easily, so consider the smaller subexpression in Out[94]

```
In[100]:= %94
```

```
Out[100]= 2 + (1+x4) y4
```

```
In[101]:= FullForm[%]
```

```
Out[101]= Plus[2, Times[Plus[1, Power[x, 4]], Power[y, 4]]]
```

```
In[102]:= TreeForm[%100]
```

```
Out[102]= Plus[2, |
                Times[|
                    Plus[1, |
                        Power[x, 4]
                    ]
                ], |
                Power[y, 4]
            ]]
```

It is useful to know how to pop the head off from an expression and replace it by another. Suppose you want a list of terms in f. This is done using the Apply function, as follows:

```
In[103]:= Apply[List, f]
```

```
Out[103]= {2, (1+x) y, (1+x2) y2, (1+x3) y3}
```

The head Plus was replaced by the head List. Now you could form a product of these terms using Apply and Times.

```
In[104]:= Apply[Times, %]
```

```
Out[104]= 2 (1+x) (1+x2) (1+x3) y6
```

```
In[105]:= Apply[Times, f]
```

```
Out[105]= 2 (1+x) (1+x2) (1+x3) y6
```

What happens in each case is that one head is being replaced by another.

A function such as Apply that takes two arguments is a binary function. Dot is another binary function. All functions have a *prefix form* which is has the head of the function followed by the arguments in square brackets. Binary functions often have also an *infix form*. The infix form is a symbol written between the arguments. We already know the infix form of Dot is (.) the period. The infix form of Apply is two at signs (@@). Thus

```
In[106]:= Plus @@ 2 (1+x) (1+x2) (1+x3) y6
```

```
Out[106]= 5 + x + x2 + x3 + y6
```

It may take a minute to convince you that this is the correct result

Term re-writing, or substitutions: We are almost ready to move on to functions, and hence to start writing programs. However it is important to spend some time looking at how to make substitutions.

Consider the expression

```
In[107]:= g = Sqrt[(1+x)/(1-x)]
```

```
Out[107]=  $\sqrt{\frac{1+x}{1-x}}$ 
```

We are interested in the result when x is replaced by (s+1/s)/2 in the expression g. One possibility is just to input x = (s+1/s)/2. But this changes x everywhere. Every place you

type x it will be replaced automatically by $(s+1/s)/2$. That's probably not what you want to do. Here is a better way to make a substitution.

Type

```
G = g /. x -> (s+1/s)/2
```

with a shift and enter combination, as usual, to input. On input, the two characters - and > will merge to form an arrow. Thus you will get

```
In[108]:= G = g /. x → (s+1/s)/2
```

```
Out[108]=  $\sqrt{\frac{1 + \frac{1}{2}(\frac{1}{s} + s)}{1 + \frac{1}{2}(-\frac{1}{s} - s)}}$ 
```

In this way you've gotten G defined without messing up x or g . The process has two parts. The first is to define a *substitution rule*, which in this case is the rule

$$x \rightarrow (s+1/s)/2$$

and the second part is to apply the rule to the expression g . The general form for applying a substitution rule (or a set of rules) is

```
expression /. rule (or rules)
```

where some people say the combination $(/.)$ means "at this point" make the substitution. Thus if you replace x using $x = \dots$, the replacement is *global*. The result of making substitutions with re-write rules is *local*. In fact, the way *Mathematica* simplifies an expression is to apply and reapply a set of active substitution rules to the expression until it no longer changes. This is a thumbnail sketch of *Mathematica's* evaluation process. I understand it is what is called a *term re-write system*.

When you want to make two substitutions, one for x and one for y , for instance, there are several ways to do it. These often give different results, so you need to think about which you want. Now remember to make the arrows using - and >. Consider for example

```
In[109]:= u = x + y;
```

```
In[110]:= u /. {x → 1/y, y → 1/x}
```

```
Out[110]=  $\frac{1}{x} + \frac{1}{y}$ 
```

```
In[111]:= u /. x → 1/y /. y → 1/x
```

```
Out[111]= x + 1/x
```

The difference is, of course, that in the first case the two substitutions are done simultaneously, or in parallel, while in the second case the substitutions are done sequentially, or in series. Thus in the second case there are two steps. After the first step the result is $y + 1/y$, and then in the second step y is replaced by $1/x$ in each place it appears, resulting in $1/x + x$. But *Mathematica* rearranges the two terms to put them in a standard output order, winding up with the result Out[111].

There is more to know about substitutions, as these are the most common operations. Most algebraic calculation I do is a succession of substitutions and factoring. However, we can study substitutions further in the context of writing programs. It is possible to write *Mathematica* programs that look like Fortran or Basic, but this is a very poor use of a powerful programming tool. Programs are actually functions. Functions are the basic procedural building blocks from which all programs are built. Thus let's now move on to study functions. Then we will build up a programming style based on functions and list operations.

Functions and programming: Let's start with something rather simple. Both Sin and BesselJ are built-in functions. They represent the trigonometric sine function and the n^{th} Bessel function (for any n) respectively.

```
In[112]:= BesselJ[m,x];
```

```
In[113]:= D[%,x]
```

```
Out[113]= 1/2 (BesselJ[-1+m,x] - BesselJ[1+m,x])
```

```
In[114]:= BesselJ[0,2.]
```

```
Out[114]= 0.223891
```

Mathematica knows quite a bit about Bessel functions. It knows how to relate the derivatives to other Bessel functions, how to evaluate them, *etc.* Now, suppose we would like to define a new function. We can call it Bx. It will be the composition of the n^{th} Bessel function of the sine of the square of x . In standard algebra notation,

$$Bx(m, x) = J_m(\sin(x^2))$$

So, here's how one defines the function in *Mathematica*.

```
In[115]:= Bx[m_,x_] := BesselJ[m,Sin[x^2]]
```

Arguments
are generic

Note the deferred
assignment

Even without a semicolon termination, no output is echoed back. The left side defines the way the function will be used, including its name (Head) and some generic names for its arguments. Commas separate the arguments. Notice the attached underlines. These arguments would be called “m-blank” and “x-blank” respectively. The right side gives the recipe telling how to interpret the function, *i.e.* its definition in terms of known functions. Whenever you use the function you put two variables in as arguments. Whatever they are, they will be substituted in for m and x respectively on the right side. The special assignment := is necessary for a function definition. The basic difference between =, which is more or less like the Fortran assignment, and the deferred assignment := is the following: The expression x = y means, “go to the current definition of y, and whatever it is right now, make that also the definition of x.” Of course if y is free of definition, then the definition of x is just the letter y. On the other hand x := y means that, each time x is used, go to y and see what the current definition of y is at that time, and then use that for x. Of course, you probably do not need to know this much detail in order to know that := rather than = is used to define functions.

```
In[116]:= y = Bx[n, a z]
```

```
Out[116]= BesselJ[n, Sin[a2 z2]]
```

```
In[117]:= y /. {n → 3, a → 2, z → .5}
```

```
Out[117]= 0.0118733
```

```
In[118]:= D[y, z]
```

```
Out[118]= a2 z (BesselJ[-1 + n, Sin[a2 z2]]
            - BesselJ[1 + n, Sin[a2 z2]]) Cos[a2 z2]
```

Notice that *Mathematica* uses the chain rule and information it has about BesselJ and Sin in order to differentiate Bx correctly. You could plot y versus z provided you substitute definite numerical values for a and n.

For another example, one a little closer to a program, let's start with the *Mathematica* function Divisors. This gives a list of all exact divisors of a given integer.

```
In[119]:= Divisors[87]
```

```
Out[119]= {1, 3, 29, 87}
```

It is interesting to have a function, which we will call sig, that takes a given integer into the sum of all of its “proper” divisors. That is, the sum of all of all divisors of n that are less than n. so for 87 one has sig[87] = 1+3+29 = 33. It almost looks like you could apply Plus to the output of Divisors. But you have to drop the number itself from the list

first. For this we can use the Drop function. Moreover, it will be convenient to have $\text{sig}[0]=0$, which is a special case. Thus

```
In[120]:= sig[n_]:=If[n==0,0,Plus @@ Drop[Divisors[n],-1]]
```

```
In[121]:= sig[772]
```

```
Out[121]= 586
```

Let's look at the way this works. First off, we need to know about the If function and the Drop function.

The If function takes three arguments, If [condition, case1, case2], and returns an expression. The expression it returns depends on the value of the condition, which is a logical expression. Thus the condition in this case is $n==0$, which is true only when zero is put in for n, otherwise it is false. When it is true, the If returns whatever is calculated in case1, which for sig is just 0. When the condition is false, then the If returns whatever is calculated in case2. In general, case1 and case2 can be single statements, or they can be a succession of statements separated by semicolons (;). Thus, the three arguments of If are separated by two commas (,) while different statements within each of the cases are separated by semicolons (;). In the sig example, both cases consist of only single statements. Hence, If returns 0 if $n==0$, or else it returns a sum of divisors.

```
In[122]:= If[x<7,x^2+2,x-1] /. x -> 3
```

```
Out[122]= 11
```

```
In[123]:= If[x<7,x^2+2,x-1] /. x -> 8
```

```
Out[123]= 7
```

The Drop function acts on a list. It has two arguments. The first is the name of the list, and the second is a number K. If $K>0$, then the first K elements are dropped from the front of the list. If $K<0$, then the last |K| elements are dropped from the end of the list.

```
In[124]:= Drop[{1,2,3,4,5,6},2]
```

```
Out[124]= {3,4,5,6}
```

```
In[125]:= Drop[{1,2,3,4,5,6},-3]
```

```
Out[125]= {1,2,3}
```

Now, if you want to review the definition of sig, just type FullDefinition[sig] and see what happens.

The number 28 is special, since it equals the sum of its proper divisors. Such numbers are called "magic." Thus $\text{sig}[28] = 28$. Can you find another magic number?

For this purpose it would be handy to apply `sig` to a large list of numbers, say numbers from 1 to 1000, and see which ones come back to the same value. Unfortunately, our function `sig` is not listable! Thus we cannot just write `sig[{1,2,3,4...}]`. There are two simple ways around this. The first is to change an *attribute* of the function `sig` to make it listable. Let's not do that. The second way is the standard one. You can "map" `sig` over a list.

```
In[126]:= Map[sig, {1, 2, 3, 4, 5, 6, 7, 8, 9}]
```

```
Out[126]= {0, 1, 1, 3, 1, 6, 1, 7, 4}
```

This shows that 6 is also a magic number: $6=1+2+3$. It may be quite hard to find another magic number. Can you do it?

The infix form of the binary function `Map` is slash at (`/@`). So one could write

```
In[127]:= sig /@ {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
Out[127]= {0, 1, 1, 3, 1, 6, 1, 7, 4}
```

I read this infix form to myself as the function "mapped over" the list.

After you are done using a function you created, and you are sure you will not need it again, it is a good idea to clear it. You would clear the definition of `sig`, for example, using `Clear[sig]`. The same should be done with variable assignments. But be sure you don't need an expression any more before you de-assign it.

Modules: In order to make more involved functions--functions that consist of many expression evaluations and use private, or *local*, intermediate variables--one can use a Module function. Module creates a private environment for evaluating a sequence of intermediate results. The general form is

$$\text{Module}[\{\text{var1}, \text{var2}, \text{var3} \dots\}, \text{expr1}; \text{expr2}; \text{expr3}; \dots]$$

Thus module has two arguments separated by commas. The first argument is a list of local variables, labeled above as `var1`, `var2`, *etc.* The second argument is a sequence of statements or expressions `expr1`, `expr2` *etc.* Module evaluates the expressions in order. The expressions can involve the local variables from the list as well as global variables that do not appear in the list. The difference is that the local variables do not exist outside the module. Thus when `a` is a local variable, then even if there is a variable called `a` outside the module, the local variable `a` will be used inside the module. When the module has finished evaluating, the value of `a` outside the module will be the same as it was before.

```
In[128]:= Clear[f]
```

```
In[129]:= n = 9
```

```
Out[129]= 9
```

```
In[130]:= f[x_] := Module[{n}, Sum[x^n, {n, 1, 5}]]
```

```
In[131]:= f[s]
```

```
Out[131]= s + s^2 + s^3 + s^4 + s^5
```

```
In[132]:= n
```

```
Out[132]= 9
```

Let's make a function that will take a polynomial expression like

$$3x^2y^2 + 4(1+x+y^3)^2 - (x+y)^2,$$

expand them in powers of x, and factor all of the coefficients. So in the example given, the result would be

$$(2 - y + 2y^3)(2 + y + 2y^3) + 2(4 - y + 4y^3)x + 3(1 + y^2)x^2$$

Part of the work can be done by the built-in *Mathematica* function `CoefficientList`. The latter function will find coefficients of a given polynomial with respect to a given variable.

```
In[133]:= CoefficientList[(s+t)(s-t)+5 t^3,t]
```

```
Out[133]= {s^2, 0, 1, 5}
```

This result tells me that the polynomial expands as $s^2+t^2+5t^3$. So let's call the new function `makePretty`. The method I have in mind is to compute the coefficient list `c`, then let `m` be the length of `c`. The degree of the polynomial is `m-1`, so if `v` is a list of powers of `x` from x^0 to x^{m-1} , then the polynomial can be represented as the dot product $(c.v)$. Then all `makePretty` has to do is to factor the list `c` before taking the dot product.

We need generic arguments for the polynomial and for the variable. So we can call them `p_` and `x_`. Two local variables are needed, one each for the coefficient list and the list of powers of `x`, so call these `clis` and `xlis`.

```
In[134]:= makePretty[p_,x_] := Module[{clis,xlis},
      clis=CoefficientList[p,x];
      xlis=x^Range[0,Length[clis]-1];
      xlis.Factor[clis]]
```

```
In[135]:= makePretty[(s+t)^3-(1-s t)^2,t]
```

```
Out[135]= (-1+s)(1+s+s^2) + s(2+3 s)t - (-3+s)st^2 + t^3
```

Notice that the value returned by Module is the result of evaluating the last expression in the sequence. In the sig function, this is the dot product of the factored coefficient list, with the list $\{1, x, x^2, x^3, \dots\}$ of powers of the argument. In order to see better how the function sig works, it is a good idea for you to step through the calculation a line at a time and watch what happens at each step.

Next let's write a function that will do three-dimensional integrals in spherical polar coordinates. The main ingredient will be *Mathematica's* built-in function Integrate. You already know how to do indefinite integration. Let's take a while to go over definite integration.

```
In[136]:= Integrate[(1-x^2) x E^(-x), {x, -1, 3}]
```

```
Out[136]=  $\frac{47}{e^3} - 2e$ 
```

```
In[137]:= Integrate[(1-x^2) x E^(-x^2), {x, 0, Infinity}]
```

```
Out[137]= -5
```

It's interesting to see what happens if you put in a parameter such as

```
In[138]:= Integrate[(1-x^2) x E^(-a x), {x, -1, 3}]
```

```
Out[138]= If[Re[a]>0,  $-\frac{6}{a^4} + \frac{1}{a^2}, \int_0^\infty e^{-a x} x (1-x^2) dx$ ]
```

The output tells us that *Mathematica* cannot do the integration if the real part of a is not positive. That is clearly because the integral does not exist otherwise. Usually, one knows that the parameters are in the proper range. So here for example one would usually have a real and positive. So usually, it is not so good to have the If function in the output. You can avoid this by adding GenerateConditions -> False as a parameter in the Integrate function. I will give the integration control parameter a nickname, "nope."

```
In[139]:= nope = GenerateConditions -> False;
```

```
In[140]:= Integrate[(1-x^2) x E^(-a x), {x, -1, 3}, nope]
```

```
Out[140]=  $-\frac{6}{a^4} + \frac{1}{a^2}$ 
```

Now let's think about the project. Let's suppose the function to be integrated is an ordinary expression (not a *Mathematica* function) which we can give the generic name F with generic polar coordinates $0 < r < \infty$, $0 < t < \pi$ and $0 < p < 2\pi$. So we want to integrate

$$K = \int_{r=0}^{\infty} \int_{t=0}^{\pi} \int_{p=0}^{2\pi} F(r,t,p) r^2 \sin t \, dp \, dt \, dr$$

In class we will learn some tricks for dealing with such integrals, but let us ignore them for now. The order of integration is chosen so that the r integral (non-compact) gets as much help converging as possible from the other two integrals.

```
In[141]:= spherint[F_, r_, t_, p_] :=
           Integrate[
             Integrate[
               Integrate[F, {p, 0, 2Pi, nope}] Sin[t],
               {t, 0, Pi}, nope],
             {r, 0, Infinity}, nope]
```

```
In[142]:= test = A E^(-r^2/a^2)
```

```
Out[142]= A e-r2/a2
```

```
In[143]:= spherint[test^2, r, xx, yy]
```

```
Out[143]=  $\frac{\sqrt{a^2} A^2 \pi^{5/2}}{\sqrt{2}}$ 
```

Of course, we probably want $a > 0$, although at this point it only needs to be such that the real part of a^2 is positive. The Simplify function can straighten this out.

```
In[144]:= Simplify[%, a>0]
```

```
Out[144]=  $\frac{a A^2 \pi^{5/2}}{\sqrt{2}}$ 
```

Notice that in applying spherint I used some randomly chosen symbols for t and p. This doesn't matter since the angle variables do not enter appear in test. Now we can make a normalized Gaussian wavefunction out of test.

```
In[145]:= Solve[%%==1, A]
```

```
Out[145]= {{A -> - $\frac{2^{3/4}}{\sqrt{a} \pi^{5/4}}$ }, {A ->  $\frac{2^{3/4}}{\sqrt{a} \pi^{5/4}}$ }}
```

Solve is similar to Roots, except that it returns substitution rules. The second substitution looks better so we can apply it.

```
In[146]:= psi0 = test /. %[[2]]
```

$$\text{Out}[146] = \frac{2^{1/4} e^{-r^2/a^2}}{\sqrt{a} \pi^{5/4}}$$

Of course you can use pretty symbols to represent the wave function and the coordinates.

Suppose you want a variational approximation to the ground state energy of a particle in the potential $V = -B \exp(-k r^2)$. The expectation of kinetic energy is the integral of $(\text{grad } \psi)^* \cdot (\text{grad } \psi)/2$, in dimensionless units. The expectation of the potential energy is the integral of $\psi^* V \psi$.

A Gaussian trial function approximation

```
In[147]:= B=.; k=.;
```

```
In[148]:= V = -B E^(- k r^2);
```

```
In[149]:= gradpsi = D[psi0,r];
```

```
In[150]:= KE = (1/2) spherint[gradpsi^2,r,p,t]
```

$$\text{Out}[150] = \frac{1}{2a^2}$$

```
In[151]:= PE = shperint[psi0^2 V,r,p,t]
```

$$\text{Out}[151] = -\frac{\sqrt{2} B}{a \sqrt{\frac{2}{a^2} + k}}$$

```
In[152]:= EN = KE + PE;
```

```
In[153]:= Numerator[Factor[D[EN,a]]]
```

$$\text{Out}[153] = \sqrt{2} a B k - \left(\frac{2 + a^2 k}{a^2} \right)^{3/2}$$

The energy is minimized when a is chosen to make this zero. This is the best Gaussian approximation to the ground state. Let $a = q/a^{1/2}$, and $B = b k$. Then the above becomes an equation to solve for q (hence a) in terms of b .

```
In[154]:= Sqrt[2] b q - ((2+q^2)/q^2)^(3/2)
```

```
Out[154]=  $\sqrt{2} b q - \left(\frac{2 + q^2}{q^2}\right)^{\frac{3}{2}}$ 
```

```
In[155]:= 2 b^2 q^2 - % [[-1]]^2
```

If you follow this along, you find eventually

$$8 + 12 q^2 + 6 q^4 + q^6 - 2 b^2 q^8$$

must be zero. This lets you solve (at least numerically) for q^2 and hence for a . Putting this value of a back into the expression for EN gives you the energy estimate.

Next let's make a function that solves the following quantum problem. A system has a Hamiltonian matrix H (with respect to some discrete basis) which is known. We want the Green function, or propagator, between state n at time $t = 0$ and state m at time t . However we want the Green function in the energy domain. In standard algebra notation,

$$G_{mn}(z) = \langle m | (z I - H)^{-1} | n \rangle$$

where $z = E + I h$ for small positive h is a complex energy parameter. Here, I is the identity matrix. We assume that H is rather large, say 100 or 200 square. This Green function could be used to find the density of states or the optical response, for example. So the *Mathematica* function that computes the Green function numerically is `makeGmn`. The generic arguments for m , n , E and h are `msit_`, `nsit_`, `eng_` and `h_`. The Hamiltonian H is a large, Hermitian matrix, filled by some other function. (One that would do a lot of integrals, like the ones above.) Let the generic `Hmat_` represent the input Hamiltonian.

Let local variable z be complex energy. The first thing needed is the identity matrix. Remember that we cannot use `I`, so we use a local list `id` for the identity. The size of this is set to the size of `Hmat`.

Now since we are concerned with only one matrix element of G , it is ridiculous to actually invert the $z I - H$ matrix. Instead, we use `LinearSolve`. I have created the following function in a file named `makeGmn.m` on `a:` and loaded it using `<<` as shown.

```
In[156]:= <<a:\makeGmn.m
```

```
Out[156]= makeGmn[msit_, nsit_, eng_, Hmat_, h_] :=  
Module[{nvec, fvec, id, z},  
z = eng + I*h; id = IdentityMatrix[Length[Hmat]];  
  
nvec = Table[0, {Length[Hmat]}];  
nvec[[nsit]] = 1;  
fvec = LinearSolve[z*id - Hmat, nvec];  
Return[fvec[[msit]]]
```

Now let's imagine that we have a Hamiltonian to represent a cluster of atoms on the surface of a sample in the scanning electron microscope. The tunneling current is related to the local density of states (LDOS) which we can compute directly from a diagonal Green function. Let's do it!

First, let me tell you about *anonymous functions*. These functions have no name and are very easy to write and use. For example, the anonymous function for taking the square of its argument and then adding 5 is

```
In[157] := (#^2+5) &
```

```
Out[157]= #1^2 + 5 &
```

which we can apply to 5, for example

```
In[158] := % [5]
```

```
Out[158]= 30
```

So the format is that # specifies where the function argument is to be inserted in the definition, and an anonymous function always ends in & (ampersand). You can give it a name if you want, using `func = (#^2+5)&`. Then you could use `func[5]`. Now back to the physics problem

We'll make a fake Hamiltonian to represent an atom cluster with about 30 relevant, single-electron states. The following will simulate some interesting structure. It is quite easy to make a more realistic one for an actual substance using semi-empirical methods. Keep in mind that you could actually calculate such an H by doing a large number of integrals (like spherint) and solving a matrix Schrödinger equation.

```
In[159] := AA = Table[.5*Random[], {30}, {30}];
```

Here I made a 30 by 30 matrix of random real numbers distributed uniformly from -.5 to +.5. Notice the shorthand use of the iterators. Since I don't care exactly which random number goes where, I just use {30} rather than something like {p,1,30}. But as you know, the Hamiltonian must be Hermitian, so I compute

```
In[160] := id = IdentityMatrix[30];
```

```
In[161] := H = Transpose[AA].AA - id;
```

This forms a plausible model Hamiltonian with reasonable properties. Next, select `h = .004`, which controls the widths of the spectral peaks. The energy step size is `h/2`.

```
In[162] := energylist = Table[ee, {-1.5, 1.5, .002}];
```

```
In[163] := gg = makeGmn[5, 5, #, H, .004] & /@ energylist;
```

Now I will compute the local density of states. It is an easy quantum mechanics exercise to show that

$$\sum_{\alpha} |\langle m | \alpha \rangle|^2 \delta(E - E_{\alpha}) = -\frac{1}{\pi} \text{Im}(G_{mm}(E + ih))$$

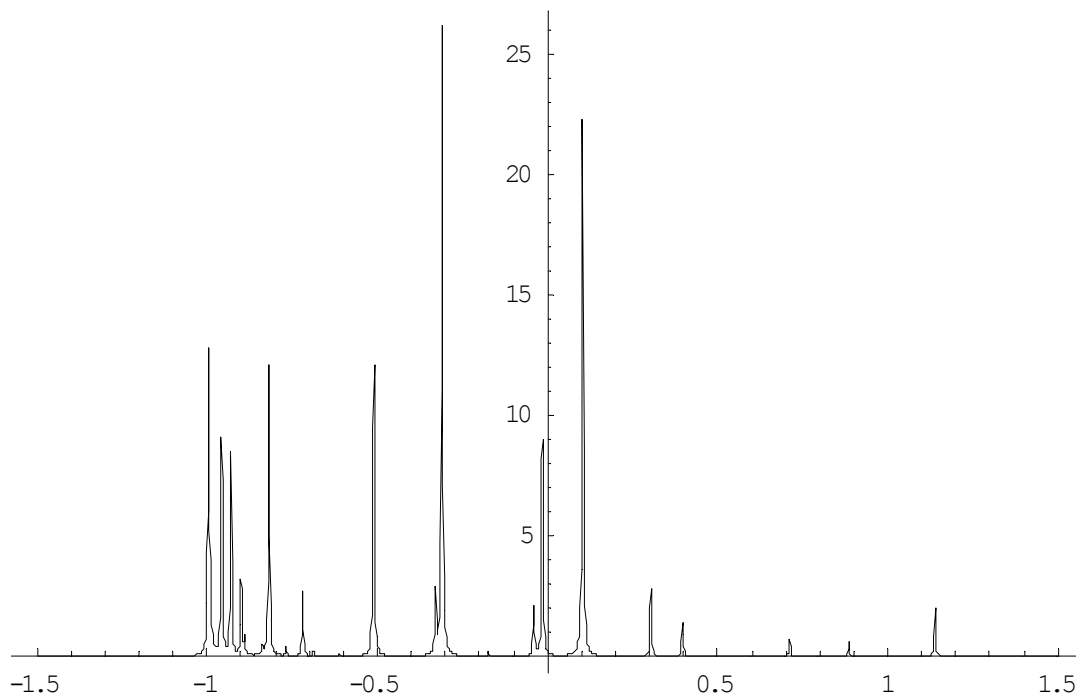
The sum on the left defines the LDOS on state m or “atom” m. The index α runs over eigenstates of H. So this is a sum over delta peaks, each at one of the eigenenergies (of width h) and the weighting of each peak is determined by how much it overlaps basis state m.

```
In[164]:= dos = -N[1/Pi Im[gg]];
```

```
In[165]:= spectrum = Transpose[{energylist,dos}];
```

Input 165 makes a list of ordered pairs like {E,D} where E is energy and D is LDOS at energy E. Then one can make a plot using *Mathematica*'s ListPlot function.

```
In[166]:= ListPlot[spectrum,PlotJoined→True,
PlotRange→All]
```



```
Out[166]= -Graphics-
```

At this point one could go on and do many numerical physics calculations relating to Green functions of this sort. However, let's stick to basics. The following demonstrates how to use *Mathematica* in a line-by-line mode. As you go along you find you want to build up short, utility functions that help you get between steps. The reality is that one does not often know where one is going, at least not several steps in advance. What to do next is often decided based on what happens now. Hence, very often when working with symbolic algebra, rather than with numerical simulations (*Mathematica*'s good at both.) you do not develop long programs but rather work along one line at a time developing what you need as you go.

Vector fields and Lie algebras As an example of line-by-line development, let's take up the topic of vector fields and Lie algebras. This is an example of working with a non-commutative, non-associative algebra in *Mathematica*.

A *vector field* in three dimensions is a set of three functions of position giving the three components of a vector at each point in space. One can write

$$\vec{v}(\vec{r}) = \{v_x(x, y, z), v_y(x, y, z), v_z(x, y, z)\}$$

or

$$\vec{v}(\vec{r}) = v_x(x, y, z)\hat{x} + v_y(x, y, z)\hat{y} + v_z(x, y, z)\hat{z}$$

But for the current purpose it will be more useful to look at vector fields as *directional derivatives*.

$$\vec{v}(\vec{r}) = v_x(x, y, z)\frac{\partial}{\partial x} + v_y(x, y, z)\frac{\partial}{\partial y} + v_z(x, y, z)\frac{\partial}{\partial z}$$

Operators such as this come up in quantum mechanics as momentum or angular momentum, and in continuum mechanics. You have seen them in a discussion of symmetry, where they are the infinitesimal generators of continuous groups. For example, the rotation group $O(3)$ is generated by operators like

$$u_z = -y\frac{\partial}{\partial x} + x\frac{\partial}{\partial y}, \quad u_y = z\frac{\partial}{\partial x} - x\frac{\partial}{\partial z}$$

and Lorentz boosts in relativity are generated by operators such as

$$b_x = \tau\frac{\partial}{\partial x} + x\frac{\partial}{\partial \tau},$$

where τ is $c t$, with c the speed of light and t is time. You have probably seen generators of more general Lie groups in classical mechanics, or things such as isospin in quantum. The association with vector fields comes from thinking of tangent vectors on curves

passing through (x,y,z). Clearly the partials can be viewed as a basis. In *Mathematica*, vector fields can be dealt with in a form {vx,vy,vz}, so that for rotation one has

```
In[167]:= uz = {-y, x, 0};          In[168]:= uy = {z, 0, -x}
```

What can one do with these directional derivative operators? Two kinds of operations are of interest primarily. First, you can *apply* a vector to a function of (x,y,z). For example,

$$u_z f = -y \frac{\partial f}{\partial x} + x \frac{\partial f}{\partial y},$$

or you can take the *commutator* [u,v] of two vector fields. (Both these operations are related to Lie derivatives.) In general, the commutator is defined by

$$[u,v] f = u(v f) - v(u f).$$

Hence in *Mathematica*,

```
In[168]:= ap[vv_, ff_, vvar_] := vv.(D[ff, #] & /@ vvar)
```

```
In[169]:= co[pp_, qq_, vs_] :=
Simplify[app[pp, qq, vs] - app[qq, pp, vs]]
```

In these two definitions, vvar and vs each represent the coordinate set. In the examples below they are {x,y,z} (three-dimensional). But the coordinates could be two dimensional, var = {x,y}, or for *prolongations* they could be var = {x,y,p}, or for relativity, var = {t,x,y,z} etc. Let's specialize in order to simplify the notation.

```
In[170]:= app[vec_, func_] := ap[vec, func, {x, y, z}]
```

```
In[171]:= com[vv1_, vv2_] := co[vv1, vv2, {x, y, z}]
```

A *Lie algebra* is a vector space over the real or complex numbers, lets say, generated by taking linear combinations of vector fields. But to be a Lie algebra, then the commutator (Lie product) of any two vectors must also belong to the same space. Generally, the properties satisfied by a Lie product are that it is linear in its first entry, it is anti symmetric, so $[a, a] = 0$ for any a , and it is *Jacobi non-associative*, meaning that for any a, b , and c , one has $[a, [b, c]] + [b, [c, a]] + [c, [a, b]] = 0$. If a Lie algebra is a finite dimensional vector space, it is called *finitely generated*. These algebras are very useful in applied mathematics and physics applications, especially relating to differential equations. An example is rotation.

The 3D-rotation group O(3) corresponds to a three-dimensional Lie algebra. Start by putting u_y and u_z in the algebra. Then the commutator $[u_y, u_z]$ has to be in the algebra too, so we call it u_x . Then we find that, commutators of any combinations of u_x, u_y and u_z are always just other combinations of these same three *generators*. The generators are basis vectors for the algebra. This is how we construct a finite dimensional Lie algebra.

You can make a table listing the generators of a finitely generated algebra, analogous to a multiplication table that shows the result of taking a commutator of any two generators. A normal multiplication table is symmetric, but this commutator table will be skew symmetric. Each commutator is a fixed linear combination of generators. The table gives the *structure* of the algebra.

A common task is the following: Given some candidate generators, see whether they really generate a finite algebra and if so find a complete set of generators forming a basis.

Suppose, in x, y, z , we are given

```
In[172] := uy = {z, 0, -x};
```

```
In[173] := uz = {-y, x, 0};
```

```
In[174] := ux = com[uy, uz]
```

```
Out[174]= {0, z, -y}
```

Then, let's introduce a dilation generator and see what happens

```
In[175] := ud = {x, y, z};
```

```
In[176] := com[ud, ux]
```

```
Out[177]= {0, 0, 0}
```

It's rather clear that dilation commutes with the other three, so the four vectors taken together form a closed set. They generate an algebra related to rotations and dilations/contractions. Now, let's include also a translation along the x axis.

```
In[178] := tx = {1, 0, 0};
```

```
In[179] := com[uy, tx]
```

```
Out[179]= {0, 0, 1}
```

```
In[180] := tz = %;
```

```
In[181] := com[tz, ux]
```

```
Out[181]= {0, 1, 0}
```

```
In[182] := ty = %;
```

By now the algebra has become a little bit more interesting. It is developing in the following way. The commutator of two rotations is another rotation, so rotations themselves are closed. So are the translations, since clearly the translations commute amongst themselves to give zero. A commutator between a translation and a rotation gives another translation. The rotations each commute with the dilation. Now the only question is: What's going to happen between the translations and the dilation? Will these commute? Will there be a new generator?

So, let's take a commutator between a translation and the dilation.

```
In[183]:= com[tx,ud]
```

```
Out[183]= {1,0,0}
```

Therefore, the commutator of any translation with ud must give back the same translation. So indeed the algebra is closed. What did we learn about its structure? Let T be translations, R be rotations and D be dilation.

$$[T, T] = 0, \quad [R, R] = R, \quad [D, D] = 0, \quad [T, R] = T, \quad [R, D] = 0, \quad [T, D] = T.$$

Now things become much more interesting if we add one more generator. Try

```
In[184]:= vx = {y^2+z^2-x^2,-2 x y,-2 x z}
```

```
Out[184]= {-x^2 + y^2 + z^2,-2 x y,-2 x z}
```

```
In[185]:= com[ux,vx]
```

```
Out[185]= {0,0,0}
```

```
In[186]:= com[uy,vx]
```

```
Out[186]= {-2 x z,-2 y z, x^2 + y^2 - z^2}
```

```
In[187]:= vz=%;
```

If you keep working you will find the algebra does close. The resulting, finitely-generated Lie algebra is the *special conformal algebra* in three dimensions. Maxwell's equations have symmetries that form a special conformal algebra in a higher dimensional space. But suppose you take away the vectors vx, vy, vz, and add instead the vector

```
In[188]:= q = {0,x^3,0};
```

Will the resulting algebra be finitely generated? What happens?

The point of this section has been to show how, in reality, one may move along doing algebra step by step. This is especially true when the calculations are symbolic, *i.e.* algebraic as opposed to numerical simulations. It is a rather common misunderstanding, especially among students, that one should always be able to plan ahead, to know where to go. This is not the case. Werner von Braun said once, "Research is what I'm doing when I don't know what I'm doing."

Notice how much we were able to learn using just the two tiny, one-line functions app and com, which are special cases of the general vector functions ap and co of input lines 169 and 169.

To keep developing the ideas of numerical simulation, on the other hand, let's now turn to a study of interacting solitons in one spatial dimension.

Numerical solution of PDEs in one space dimension: The Korteweg-de Vries (KdV) equation is a PDE describing the motion of shallow water waves. You will see it in many places in physics literature as well as in texts. Its theoretical importance is that it is the standard example of a PDE that has *soliton* solutions. A soliton is a solution of a nonlinear PDE that maintains its shape over time. In equations of the KdV type this happens because the nonlinearity, which tries to make the solution focus into a peak, balances against the dispersion which tries to make the wave spread out. The result is a compromise in which certain waveforms move along in a stable way, maintaining their shape over a long period of time. (KdV)

In dimensionless variables in one dimension the KdV equation is

$$\frac{\partial u}{\partial t} - 6u \frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3} = 0.$$

The idea in this section is to write functions that will solve this PDE or any similar one, starting from a given configuration initial $u(x,0) = f(x)$ and letting $u(x,t)$ evolve forward in tiny time increments. The pedagogical objective is to show how to do functional programming. That is, to illustrate a process of building up a program as a succession of functions that use other functions.

At a particular time, let the list u represent the solution. The individual entries in the list u are values at a set of equally spaced points along the x -axis. Let dx be the spacing. If the length of u is N , then the total length of x -axis we consider is $L = N dx$. As mentioned above, it is very handy to have periodic boundary conditions. Thus we treat the N^{th} point on the axis as a neighbor of the first. Suppose the time steps are of length dt . Stepping from one time t to the next time $t+dt$ will be done by 4th order Runge-Kutta, due to its numerical stability and ease of programming. (See Matthews and Walker. Scott Crockett's thesis is worth looking at too.) The x -derivatives will be evaluated as central differences.

Now let's consider how to deal with the central differences. The symbol δ for central difference is a difference operator.

$$\delta u_n = \frac{1}{2}(u_{n+1} - u_{n-1}) = \frac{1}{2}(e^{dx D} u_n - e^{-dx D} u_n) = \frac{1}{2}(e^{dx D} - e^{-dx D})u_n,$$

where D is the derivative, and I have expanded in a Taylor series each of the two u terms on the right. Thus, work with the operators, and solve for D in terms of δ . The result is

$$D = \frac{1}{dx} \sinh^{-1} \delta = \frac{1}{dx} \left(\delta - \frac{1}{6} \delta^3 + \frac{3}{40} \delta^5 + \dots \right)$$

So, in *Mathematica* one will need functions like

```
In[189]:= rr[what_] := RotateRight[what]
```

```
In[190]:= rl[what_] := RotateLeft[what]
```

```
In[191]:= dlt[ff_] := (rl[ff]-rr[ff])/2
```

```
In[192]:= Dx[fa_,dy_] := dlt[fa]/dy
```

Here, if necessary, we could add higher-order corrections to the derivative, which we derived above, but that may not be a good idea in view of possible instability.

```
In[193]:= Dxxx[fb_,dz_] := Dx[Dx[Dx[fb,dz],dz],dz]
```

Next, we solve KdV to get the time derivative on the left side equal to everything else on the right side. We then write the following *Mathematica* function to evaluate the time derivative at any step from a given u vector, by means of the righthand side (rhs).

```
In[194]:= rhs[uu_,ddx_] := 6 uu Dx[uu,ddx] - Dxxx[uu,ddx]
```

Very well. This is the first major ingredient for the program.

The next step is to write a function that will take u at time t and move it up to u at time t+dt via Runge-Kutta. Let's do this in such a way that we could change the PDE we want to solve, and still use more or less the same program. (See Matthews and Walker p.353, Eq.(13-38), but keep in mind that we are dealing with lists. *Mathematica* doesn't care! It does the operations on whole lists at a time. You don't even need to think about it.)

```
In[195]:= next[uold_, ddt_] := Module[{k1, k2, k3, k4},
    k1 = ddt rhs[uold, dx];
    k2 = ddt rhs[uold+k1/2, dx];
    k3 = ddt rhs[uold+k2/2, dx];
    k4 = ddt rhs[uold+k3, dx];
    Return[uold+(k1+2 k2+2 k3+k4)/6]]
```

In this function, notice that the lists uold and ddt are input arguments, the lists k1..k4 are local variables and the variable dx is global, since it is neither an argument nor in the local variable list. That means, dx will be whatever it is outside the function next.

Now the programming is almost done. All you need is a way to set up initial conditions for u, and a way to present the solution after a certain number of time steps. The numerical values of the initial u should range from -1 to 1, approximately, to set the scale. Then $dt \ll dx < 1$, in order to ensure stability.

```
In[196]:= peak[z_,a_] := Sech[z/a]^2
```

```
In[197]:= dx = .1; dt = .02 dx;
```

```
In[198]:= uo = Table[Sum[-6. peak[(nx-600-1000 k) dx,1.]
                    ,{nx,1,1000}]];
```

In the step above, the initial distribution is set to a negative peak that will decay into two separate solitons. It turns out that the integration scheme is not as stable as one would like, but it is not difficult to change it to a more stable one, and all we want to do now is to get the basic idea of the simulation.

We make a function that will move u ahead a certain number $nsteps$ of time steps, and then finally an output function.

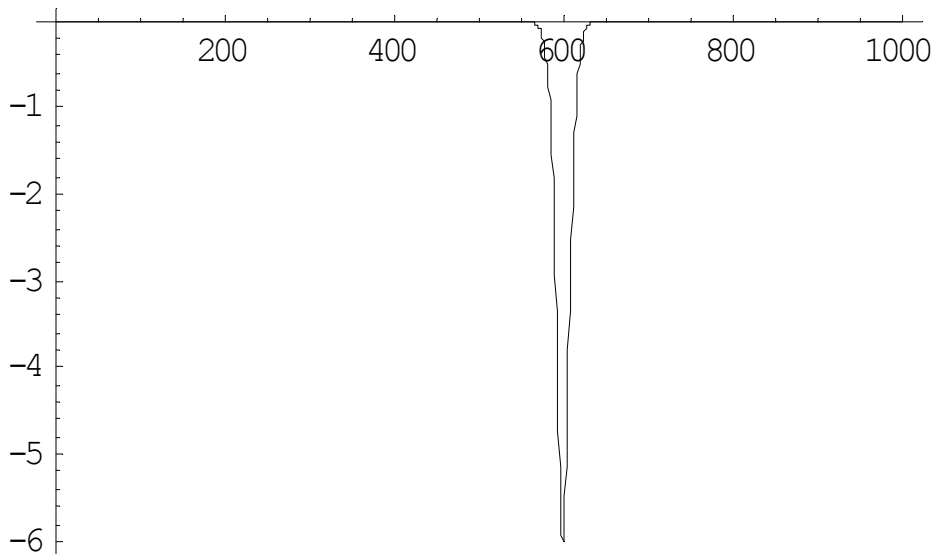
```
In[199]:= ahead[uu_,nsteps_]:=
          Nest[next[#,dt]&,uu,nsteps]
```

```
In[200]:= present[uu_]:= ListPlot[uu,PlotJoined->True,
                                   PlotRange->All]
```

The function `Nest[f,x,n]` just applies a function f of a single argument n times to x and returns the final result. Notice that I have made the function `next` into an anonymous function of a single variable so I can use `Nest`. So now we can go to work.

```
In[201]:= u = uo;
```

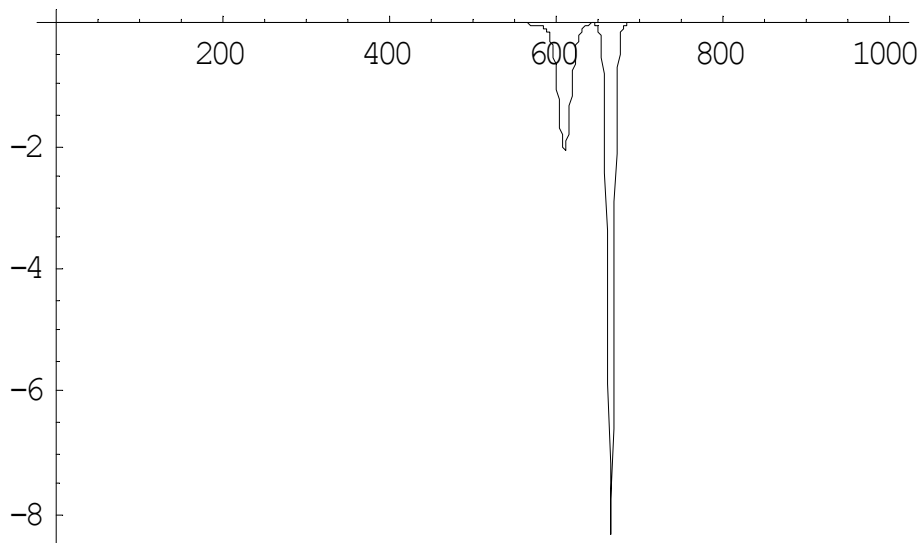
```
In[202]:= present[u]
```



```
Out[202]= -Graphics
```

```
In[203]:= u = ahead[u,200];
```

```
In[204]:= present[u]
```



```
Out[204]= -Graphics-
```

See Zabusky and Kruskal, Phys. Rev. Lett. **15**, 240-243 (1965), which is a famous computer simulation of soliton interactions.

Reducing a rational expression modulo a polynomial: Now for one final example of *Mathematica* programming, we shall go back to symbolic calculations. The following is a general utility function of great usefulness in symbolic simplifications.

Suppose you have a rational expression like

$$F = \frac{1 - x^2 + x^3 + 2x y^2}{1 + x + 2x y^2}$$

where you know that x is a root of the quadratic polynomial $p = x^2 - 2ax + 1$. Then you can simplify F “modulo $p = 0$.” That is to say, if $p = 0$, you can replace x^2 by $2ax - 1$, and you can replace x^3 by $x(2ax - 1) = 2a(2ax - 1)x - 2a$, and so on. In this way you can eventually reduce any polynomial expression in x down to one linear in x . It may be less obvious, but you can also reduce any rational expression (*i.e.* a ratio of polynomials, like F above) to an expression linear in x . This is because all higher terms in the Taylor series for F can telescope down to the form $A + Bx$. In fact

$$F = \frac{(a-1)x}{a+y^2} - \frac{1-2a-2y^2}{2(a+y^2)}$$

In general, any rational function (ratio of polynomials) in x , where x is a zero of some polynomial p of degree n , can be reduced to a polynomial expression in x of degree $n-1$ or less. (The polynomial p should be irreducible, meaning it shouldn't factor.) The coefficients of the final polynomial form of F will contain only the other symbols of F , not x . Thus, if x is a root of p , meaning $p(x) = 0$, and F is a rational expression of x and other variables, then

Remember this only holds for x such that $p(x) = 0$.

$$F = A_0 + A_1 x + A_2 x^2 + \dots + A_{n-1} x^{n-1}$$

where n is the degree of p and the coefficients A depend only on the other variables. The objective is to write a *Mathematica* function that, when given a rational function F , an irreducible polynomial p and a variable x (supposed to be a root of p), then it will return the required simplification.

The method we can use is based on the division algorithm. Let f and p be polynomials. Then p can be "divided by d " in the sense that there exist polynomials q and r such that $f = q p + r$ with the degree of r less than the degree of p . Of course the letters can stand for quotient and remainder. Suppose we evaluate this division equation at x that makes $p(x) = 0$. Then $f(x) = 0 + r(x)$. Thus the plan is as follows:

First write

$$F - A_0 + A_1 x + A_2 x^2 + \dots + A_{n-1} x^{n-1}$$

where the A s are undetermined coefficients. We want to choose the A s so as to force this to be zero. Thus, factor the expression and take the numerator, which is a polynomial. Call it f . Then divide f by p and keep only the remainder. Now choose the A s so as to make the coefficient of each power of x zero. This gives equation to solve for the coefficient A s.

Mathematica has a built-in function to do polynomial division and keep only the remainder, and this is called `PolynomialRemainder`,

```
PolynomialRemainder[f,p,x]
```

returns the remainder when f is divided by p . Thus, here is a function to do the required simplification

```
In[205]:= md[fun_,pol_,var_] :=
Factor[PolynomialRemainder[fun,pol,var]]

In[205]:= rat[ff_,pp_,vv_] := Module[{A,B,C,Q,R,h,cl,a,sz},
B=ff/.vv->sz; h=Exponent[pp,vv];
```

```

A = a /@ Range[0, h-1];
C=sz^Range[0, h-1]; Q=A.C;
B=Numerator[Factor[B-Q]];
C=md[B, pp/. vv→sz, sz];
cl = Factor[CoefficientList[C, sz]];
R=Solve[cl==0, A]; A=Factor[A/.R[[1]]];
Return[A.(vv^Range[0, h -1])]

```