A Random Walk Through Functional Programming

Richard J. Gaylord
Dept. of Materials Science & Engineering
University of Illinois
1304 W. Green St.
Urbana, IL 61801
gaylord@ux1.cso.uiuc.edu

Samuel Kamin
Computer Science Dept.
University of Illinois
1304 W. Springfield
Urbana, IL 61801
kamin@cs.uiuc.edu

August 17, 1992

Abstract

The functional approach to scientific computing is illustrated using various random walk models and the Mathematica programming language.

Keywords: functional programming, Mathematica, random walks

1 Introduction

Scientific computing has until quite recently, been defined by number crunching with one of several "procedural" programming languages: FORTRAN, introduced thirty-five years ago, is the principal language, with C, a mere twenty year old, rapidly gaining in popularity; Pascal and BASIC are also used, less often. This state of affairs in scientific computing is evidenced by the availability of "Numerical Recipes: The Art of Scientific Computing" by W.H. Pess, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, Cambridge U. Press in FORTRAN, C, Pascal and BASIC language versions.

The style of programing employed by all of these languages is characterized by the use of variables whose assigned values specifiy the state of an ongoing computation. All computation is expressed in terms of changes to this state, with "large" changes, such as the modification of an array, being accomplished by the accumulated effect of many small changes, such as the modification of a single component of the array. John Backus, the original designer of FORTRAN, in a scathing attack on procedural languages in 1978 [1], called this "the von Neumann bottleneck."

An alternative approach to programming is the "functional" style. Simply stated, this is an attempt to lift the entire programming process to the same level of convenience as using arithmetic expressions in procedural languages. For example, instead of having only integer- or real-valued expressions, we can think of matrix-valued expressions. This provides both great abstractness and conciseness, and frees the programmer from having to worry about such uninvolving, low-level details as establishing the storage size and location for a matrix. Moreover, this style erases entirely the distinction between "procedure" and "expression," by having functions be simply another type of value (like integer or matrix) which can be the value of an expression. This feature allows for great flexibility in structuring programs.

While some traditional languages, such as Lisp and APL which are nearly as old as Fortran and C, do incorporate various functional programming characteristics, they have not been widely used for scientific computing, primarily because of a lack of efficiency. However, this problem has been alleviated to a considerable extent by recent advances in computer hardware which have also permitted scientific computing to expand beyond number crunching on mainframes to symbolic manipulation and visualization on desktops.

In this paper we illustrate functional programming using the Mathematica programming language [2, 3, 4, 5], which is widely available in the scientific community. Rather than demonstrating functional programming methods in the traditional manner of presenting a language, with code fragments dealing with a number of disparate topics, we will focus on the coding of a single problem domain, the "random walk" model, in order to demonstrate the range and power of computational tools that the functional style of programming can bring to bear on a subject of current scientific interest.

2 The General Two-Dimensional Lattice Walk

The random walk model is the quintessential stochastic process [6, 7, 8]. First posed in 1905, it has since found application in many areas including the physical sciences (Einstein employed the model in his work on atomistic diffusion), biological and ecological sciences (animal territorial behavior) and economics (stock market trends).

Aside from the wide-ranging applicability of the random walk model, there is an inherent interest in the behavior of the model because as W. Feller, a doyen in the field of probability, states, "In studying random walks, we shall ... encounter ... conclusions which not only are unexpected but actually come as a shock to intuition and common sense."

The random walk model is a particularly appropriate area with which to demonstrate the functional style of programming because while there are some asymptotic analytical solutions to a number of problems related to a single random walker, and to a fewer number of problems involving multiple random walkers, in general computer simulation provides the best, and sometimes only, method for studying the properties of these systems in detail.

In this article, we will focus on a specific version of the random walk model: the two-dimensional lattice walk model which can be described as follows:

> A walker starting at some position in a plane takes unit steps in one of four directions along the x- and y-axes (North, East, South, West). The possible positions of the walker are a subset of all of the points of the plane which have integral-valued coordinates.

We begin with the simplest case of a single walker in an unbounded plane with all four neighbor sites available for a step and starting at the origin. We can program this in FORTRAN (note we consider a walk of n steps where n must be specified to execute the program):

```
1.        integer path(2,2*n)                          path(1,i) and path(2,i) are
                                                         x, y coordinates of ith step
2.        integer xdir(4), ydir(4)                     initialized below
3.        real r
4.        integer s
5.        data xdir / 0, 1, 0, -1 /                    four possible step directions,
6.        data ydir / 1, 0, -1, 0 /                    given by (xdir(i), ydir(i)), i=1,4
7.        path(1,1) = 0                                first location is (0,0)
8.        path(2,1) = 0
9.        r = 0.0                                      random number generator seed
10.       do 100 i = 2, n                              n–step walk
11.         r = rand(r)
12.         s = floor(4*r)+1                           s is a random integer in 1...4
13.         path(1,i) = path(1,i-1) + xdir(s)          add vectors: (path(1,i-1),
14.         path(2,i) = path(2,i-1) + ydir(s)            path(2,i-1)) + (xdir(s),ydir(s))
15. 100   continue
16.       print *,(path(1,i), path(2,i), i=1,n)
```

In M, the same style of programming is available, with differences, of course, in notation:

```
a.   xdir = {0, 1, 0, 1};                              analogous to line 5 above
b.   ydir = {1, 0, -1, 0};                             line 6
c.   path[1,1] = 0;                                    line 7
d.   path[2,1] = 0;                                    line 8
e.   Do[
f.     s = Random[Integer, {1, 4}];                    lines 11 and 12
g.     path[1,i] = path[1,i-1] + xdir[[s]];            line 13
h.     path[2,i] = path[2,i-1] + ydir[[s]],            line 14
i.   {i, 2, 50}];                                      line 10, together with e
j.   Do[ Print[{path[1,i], path[2,i]}], {i,n}]         line 16
```

We need to make just a few points about these programs. Random number generation is slightly different in MATHEMATICA than in Fortran; the MATHEMATICA function `Random` has some "internal memory" that allows it to remember the last number generated and return a new number on each call; the FORTRAN routine has to be passed the last number as an argument, and then generates the next number in the sequence. Aside from that, the only differences are that array variables in FORTRAN have to be declared (lines 1 and 2) and the iterator in the `Do` loop appears at the top of the loop (line 10, replaced by lines e and i).

Among the even more minor differences, note that square brackets are used in MATHEMATICA for function calls and array subscripts (parentheses are still used for grouping in expressions). Also, the splitting of MATHEMATICA programs onto lines is arbitrary (as in C and almost all modern computer languages); nor do columns have any significance (again, as in C). Finally, the use of double square brackets in lines g and h will be explained later.

The upshot of this discussion is that one can "do FORTRAN programming in MATHEMATICA." However, it is neither the most elegant and concise, nor the most efficient, way of programming in MATHEMATICA. The goal of this paper is to demonstrate an alternative programming style that MATHEMATICA supports well: functional programming.

As an example, here is a more idiomatic solution for the simple random walk:

```
FoldList[ Plus, {0, 0},
    {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}
          [[ Table[Random[Integer, {1, 4}], {n-1}] ]] ]
```

Note that this is just a single expression: there are no assignments or declarations and no Do loops. The value of this expression is essentially the same as the value of the path array after executing the previous code.

In the next section, we will explain the specific details of this code. In this section, we have shown how traditional coding techniques can be employed, *mutatis mutandis*, in MATHEMATICA, and established some notational conventions of MATHEMATICA. In the following sections, we will also frequently and explicitly demonstrate the merits of functional programming as a simpler, more natural, and more flexible style than the FORTRAN/C style for scientific computation.

3 Lists

One of the most powerful features of functional languages is dynamic memory allocation. In FORTRAN, all uses of memory—that is, arrays and variables— must be established before the program begins executing. Arrays cannot shrink or expand, nor can new arrays be created on the fly. No function can return an array. This restriction was in the original version of FORTRAN and came directly from a desire for efficiency in the use of memory. While this desire is far less compelling now, the restriction remains in the latest versions of FORTRAN. Newer languages like C allow users to allocate memory explicitly from a large area of memory reserved for this purpose, called the "heap." However, this still falls short of a fully general facility, because it doesn't account for de-allocation. If, in the course of running a program, we were to fill up memory with arrays that were no longer needed, we would run out of memory and find our computation aborted. The solution is to keep track ourselves of which arrays are no longer needed, and make their memory available to be reused, but this can be extremely complex and substantially mitigate the convenience of dynamic memory allocation.

All functional languages, MATHEMATICA included, automatically recover memory no longer needed, in a process called "garbage collection." Its chief virtue is that it makes possible the feature of "dynamically-allocated data structures." The most heavily used of these, in MATHEMATICA as well as other functional languages, is the **list**. A list is simply a sequence of arbitrary values, possibly including other lists. New lists of arbitrary length can be created at any time, and functions can return entire lists.

MATHEMATICA comes equipped with a very large collection of built-in functions to perform various common operations on lists. We will now introduce some of them, explaining along the way the code given at the end of section 2.

Before doing so, however, we need to digress momentarily, to mention what sessions with the MATHEMATICA system look like, so we can reproduce parts of sessions in the text. All that really needs to be said is that MATHEMATICA is an interactive system, that the input prompt has the form "In[*n*]:=", and that the corresponding output, if any, begins with "Out[*n*]=", where n is an integer that is automatically incremented after each input. Thus, to find $\sqrt{500}$, enter:

```
In[23]:= Sqrt[500.0]          this is the 23rd input in this session
Out[23]= 22.3607              MATHEMATICA's response
```

We will from time to time show output directly from MATHEMATICA in this form, except that, to avoid clutter, we will omit the number, and will omit the output altogether if it is of no interest to us (in which case it will usually be followed immediately by an input whose output is shown).

So, to get back to lists: the first thing to know is that the list containing n elements $x_1, \ldots, x_n$, is written in MATHEMATICA as `{x1, ..., xn}`. Thus, `{0, 0}` is the list of two zeroes; `{{0, 0}}` is a list containing one list, which in turn is a list of two zeroes; and `{{0, 1}, {1, 0}, {0, -1}, {-1, 0}}` is a list of four elements, each a list of two integers.

The latter is, in fact, a particularly important list to us; it was used in the code at the end of section 2, and will be used frequently in the rest of the paper. So, to avoid repeating it, we will give it the name `NESW` (for the compass directions) and always refer to it by that name:

```
In[]:= NESW = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}      NESW is a variable
Out[]= {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}
In[]:= NESW
Out[]= {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}
```

We can now produce our random walk in three steps:

1. Generate a list L1 of $n - 1$ random integers in the range $1 \ldots 4$. Suppose this list is

   L1 = `{4, 1, 1, 2, 3, 3, 4, ...}`.

2. Use L1 to create a list L2 of the corresponding elements of `NESW`. That is, the first element of L2 will be the fourth element of `NESW`, the second element of L2 will be the first element of `NESW`, etc. Thus:

   L2 = `{{-1, 0}, {0, 1}, {0, 1}, {1, 0}, {0, -1}, {0, -1}, {-1, 0}, ...}`.

3. Create the list L3 by starting at point `{0, 0}` and summing the prefixes of this list. That is, the first element of L3 will be `{0, 0}`, the second element will be the sum of `{0, 0}` and `{-1, 0}`, and the third will be the sum of the second element and `{0, 1}`. Thus:

   L3 = `{{0, 0}, {-1, 0}, {-1, 1}, {-1, 2}, {0, 2}, {0, 1}, {0, 0}, {-1, 0}, ...}`.

   This is the walk.

These steps can each be accomplished with a built-in MATHEMATICA function. For step 1, we use the function `Table`, which works like this: `Table[expr, {m}]` generates the list $\{\underbrace{\texttt{expr}, \ldots, \texttt{expr}}_{\texttt{m}\ times}\}$

Thus,

```
Table[Random[Integer, {1, 4}], {n-1}]]]
```

generates our list of $n-1$ random integers (L1).

For step 2, we need to know how to select elements of a list. For list {x$_1$, ..., x$_n$}, the notation {x$_1$, ..., x$_n$}[[i]] refers to x$_i$. (We can now explain the double square brackets in the MATHEMATICA program on page 3. The variables `xdir` and `ydir` in that program represented *lists* rather than arrays, so we used the double bracket notation for subscripting. The reason we made them lists was that MATHEMATICA provides no concise notation for initializing arrays.) Naturally, if the list has been given a name, say L, we can write `L[[i]]` to refer to x$_i$. For example, `{0, 0}[[1]]` is 0, `NESW[[1]]` is `{0, 1}`, and `NESW[[1]][[1]]` is 0. (The latter can also be expressed more concisely as `NESW[[1, 1]]`.) Furthermore, we can select a sublist of a list in one shot: if X is the list {x$_1$, ...x$_n$} and I is the list {i$_1$, ..., i$_k$}, then `X[[I]]` is the list {x$_{i_1}$, ..., x$_{i_k}$}. For step 2, all we need to do is to subscript the list `NESW` by the elements of the list L1. Thus, we can produce the list L2 by writing:

```
NESW[[ Table[Random[Integer, {1, 4}], {n-1}]]] ]]
```

(Alternatively, we could combine steps 1 and 2, generating random integers and using them to select from `NESW` individually: `Table[NESW[[Random[Integer, {1, 4}]]]], {n-1}]`.)

Finally, to produce L3 we need the function `FoldList`. `FoldList[f, x, {a, b, ...}]`, where `f` is a function and `x` a value, is the list `{x, f[x, a], f[f[x, a], b], ...}`. For example, `Plus` is the function, normally written as "+" in "infix" notation, that adds numbers; in MATHEMATICA, it operates on lists component-wise (e.g., `Plus[{x, y}, {u, v}]` = `{x, y} + {u, v}` = `{x+u, y+v}`). Thus, `FoldList[Plus, {0, 0}, L2]` is the list

```
{{0, 0}, {0, 0}+{-1, 0}, ({0, 0}+{-1, 0})+{0, 1}, ...,
```

which is just L3. Thus, we arrive at the expression given at the end of section 1:

```
FoldList[Plus, {0, 0},
    NESW[[ Table[Random[Integer, {1, 4}], {n-1}] ]] ]
```

What we really want is a function that will produce a walk of length n given n. The following illustrates the notation for defining functions in MATHEMATICA:

```
In[]:= walk[n_] := FoldList[Plus, {0, 0},
                      NESW[[ Table[Random[Integer, {1, 4}], {n-1}] ]] ]

In[]:= walk[8]

Out[]= {{0, 0}, {-1, 0}, {-1, 1}, {-1, 2}, {0, 2}, {0, 1}, {0, 0}, {-1, 0}}
```

The main thing to notice about the definition of `walk` is that the argument must be followed by an underline character when it appears on the left-hand side of the ":=".

We can illustrate the simple lattice walk graphically.

```
rwgraphic[n_] := Show[Graphics[Line[walk[n]] ]]
```

As this expression shows, there is a concise nested formulation of the graphics. `Line` is a graphical primitive consisting of a line joining consecutive pairs of points. The graphics can be embellished by making the lines thicker, coloring them, labelling the starting point, etc. A typical random walk is shown in Figure 1.

Figure 1

4 Analyzing a walk

We can ask a great many questions about the simple walk we've just defined, questions such as "how often does the walk intersect itself?" and "how often does some pattern occur in the walk?" Answering these will also give us the opportunity to introduce the features of **functions as arguments** (and the closely related feature of **anonymous functions**) and **pattern-matching**. Two other features discussed in adjoining boxes are **overloading** and **recursion**.

A basic item of information about any walk is the number of sites visited. This is, of course, different from the length of the walk, since some sites may be visited many times. The built-in function `Union`, given any number of lists as its arguments, returns a list containing all the items in all its arguments, *without repetitions*. In particular, `Union[path]` removes repetitions from the list `path`. As an example of its use, we have:

```
numberDistinctSites[path_] := Length[Union[path]]
```

where `Length` is the built-in function that computes the length of a list. Here, as throughout this section, we assume `path` is a list of points generated by a call `walk[n]`, for some `n`. It is one of the benefits of having lists that we can create entire walks, name them, then pass them to other functions to do the various analyses.

We will, at times, want to look at all the `x` or all the `y` values in a path separately. In terms of the list, this means we want to pick out all the first elements, or all the second elements, of the pairs in the list. To do this, we will use the `Transpose` function.

`Tranpose` turns lists "inside-out." Specifically, `Transpose[{{`x_{11}, x_{12}, ..., x_{1n}`}, ..., {`x_{m1}, x_{m2}, ..., x_{mn}`}}]` is `{{`x_{11}, x_{21}, ..., x_{m1}`}, ..., {`x_{1n}, x_{2n}, ..., x_{mn}`}}`. In particular, when $n = 2$, i.e. the argument is a list of pairs, `Transpose[{{`x_{11}, x_{12}`}, ..., {`x_{m1}, x_{m2}`}}]` is `{{`x_{11}, x_{21}, ..., x_{m1}`}, {`x_{12}, x_{22}, ..., x_{m2}`}}`; thus:

```
In[]:= xvalues[path_] := Transpose[path][[1]]

In[]:= xvalues[ {{0, 0}, {-1, 0}, {-1, 1}, {-1, 2}, {0, 2}, {0, 1}, {0, 0}, {-1, 0}} ]

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

In[]:= yvalues[path_] := Transpose[path][[2]]
```

As another example, `Transpose[[{xvalues[path], yvalues[path]}]` returns `path`.

Finally, we will find below that some analyses are more easily applied to the sequence of *step increments* taken in a walk than to the *sites visited* in the walk. By this we mean the actual elements of `NESW`, rather than their sums. (An example is the list L2 on page 5.) We can, of course, easily generate lists of steps:

```
walkSteps[n_] := NESW[[ Table[Random[Integer, {1, 4}], {n-1}] ]]
```

and, given a list of steps, we can easily obtain the list of sites:

```
stepsToWalk[steps_] := FoldList[Plus, {0, 0}, steps]
```

For example, we could obtain our previous walk as follows:

```
In[]:= steps8 = walkSteps[8]
Out[]= {{-1, 0}, {0, 1}, {0, 1}, {1, 0}, {0, -1}, {0, -1}, {-1, 0}}.
In[]:= path8 = stepsToWalk[steps8]
Out[]= {{0, 0}, {-1, 0}, {-1, 1}, {-1, 2}, {0, 2}, {0, 1}, {0, 0}, {-1, 0}}
```

From now on, when a function expects as its argument the list of sites, we will call the argument "`path`", and when it expects the list of steps, we will call it "`steps`". We will use variables `path`n and `steps`n for lists of locations or steps of length n

It is also possible, and only slightly more difficult, to obtain the list of steps from the walk, but this will be postponed to the next section.

4.1 Functions as arguments

By a "function as an argument," also known as a **downward function value**, we mean simply a function that is passed to another function as an argument. We have, in fact, already seen one, when we passed `Plus` as an argument to `FoldList`. (Moreover, readers familiar with Fortran, C, or Pascal know that it can be done in those languages as well.) An "anonymous function" is, obviously, a function that has no name. Few imperative languages permit functions to be defined other than by ordinary function declarations, which give those functions names, but in functional programming functions are so ubiquitous that it is often convenient to define them anonymously. Passing an anonymous function as an argument to another function is a very common idiom in functional programming.

In Mathematica, there are two equivalent notations for defining anonymous functions of one argument (which is all we need in this paper):

```
Function[x, ...x...x...]
(...#...#...)&
```

`Function[x, ...x...x...]` is the function that takes a value v as an argument and returns the value ...v...v...; of course, any name can be used for the argument, not just `x`, and it can appear any number of times in the body of the function. For example, `Function[x, {x, 1}]` makes a list that contains its argument and the number 1: `Function[x, {x, 1}][10]` returns `{10, 1}`. The form `(...#...#...)&` is an abbreviation for `Function[x, ...x...x...]` (assuming `x` is chosen not to occur elsewhere in `...#...#...`); so, `({x, 1})&[10]` also returns `{10, 1}`.

We can write the function `pathToSteps` that takes a path and gives the steps in one line by using an anonymous function. What we want to do is to subtract, from each step location in the walk, the preceding location. The built-in function `RotateRight` rotates a list one time to the right, so that subtracting from each element of `path` the corresponding element of `RotateRight[path]` essentially gives us the list (we will only have to discard the first element, which is the difference between the first and last elements of `path`. Furthermore, subtraction, like addition, extends to lists automatically. Thus, the code is:

```
pathToSteps[path_]:= Rest[(# - RotateRight[#])&[path]]
```

(`Rest[p]` is `p` with its first element removed.)

Perhaps the best known function that takes a function argument is `Map`. It has two arguments, a function and a list, and applies the function to each element of the list. A simple example of its use is to define the function `xvalues`, which was defined above using `Transpose`:

```
xvalues[path_] := Map[(#[[1]])&, path]
```

Another example is finding the **span** of a walk, which is the dimensions of a box that bounds the walk; in other words, it is the pair `{xspan, yspan}`, where `xspan` is the difference between the maximum and minimum x values achieved in the walk, and similarly for `yspan`. If we have a list of numbers `L`, `Max[L]` is the maximum and `Min[L]` the minimum number in the list. We can use `Transpose` to obtain the list of x values and y values in a path:

```
span[path_] := Map[(Max[#] - Min[#])&, Transpose[path]]
```

The function `(Max[#] - Min[#])&`, when applied to a list of numbers, returns the difference between the maximum and minimum numbers in the list. Mapping it over the two lists in `Transpose[path]` gives the pair `{xspan, yspan}`.

A more complicated use of `Map` is to answer the following question about a walk: how many times was each site visited? (In random walk terminology, this is an example of an *occupancy* problem.)

With no extra work, we can generalize this, asking about *any* list, how many times does each element in the list occur? A function `frequency` can be defined to answer this question:

```
In[]:= L = {1, 2, 3, 4, 3, 2, 1}

In[]:= frequency[L]

Out[]= {{1, 2}, {2, 2}, {3, 2}, {4, 1}}

In[]:= frequency[path8]

Out[]= {{{0, 0}, 2}, {{-1, 0}, 2}, {{-1, 1}, 1},
{{-1, 2}, 1}, {{0, 2}, 1}, {{0, 1}, 1}}
```

Here's how `frequency` can be defined using `Map`: Note first that there is a built-in function `Count` such that `Count[lis, x]` gives the number of times `x` occurs in `lis`. Thus, if `L` is the list just given, the expression `Count[L, 3]` returns 2, and the expression `{3, Count[L, 3]}` returns `{3, 2}`. Now suppose `M` is the list of all the values occurring in `L`, without repetitions. Then the expression

```
Map[({#, Count[L, #]})&, M]
```

Overloading

A function name is said to be **overloaded** when it can be used to apply different functions to different types of data. An example among the built-in functions of MATHEMATICA is `Plus`, which, as we have seen, can be applied to various data types, e.g. integers, floating-point numbers, and lists of numbers. It is a great convenience to be able to overload function names, since it avoids the need to invent, and remember, lots of different names.

It is easy to use this feature. Suppose, for example, that we want to define the function `walk` so that, if its argument is a floating-point number, it produces an *off-lattice* walk. An off-lattice walk is one in which the walker takes steps of unit length in a random direction. We can write:

```
walk[r_Real] := FoldList[Plus, {0, 0},
        Map[({Cos[#], Sin[#]})&, Table[Random[Real, {0, N[2Pi]}],{r}]]]
```

This code should not be very difficult to follow now, but there are a few new things: `Random[Real, {0, N[2Pi]}]` returns a random real number in the range $0 \ldots 2\pi$ (the `N[...]` is needed to ensure that `2Pi` will be evaluated to a number rather than being left in symbolic form). Taking the cosine and sine of such a number gives a vector of length one in a random direction. Now, `walk[10.0]` produces an off-lattice walk; `walk[10]` invokes the previous definition and gives a lattice walk (though it would be clearer to write our on-lattice walks as "`walk[n_Integer] := ...`").

returns the list `{{1, 2}, {2, 2}, {3, 2}, {4, 1}}`, as desired. But we have previously seen that `Union[L]` is the list of all the elements of `L` without repetitions. Thus:

```
frequency[L_] := Map[{#, Count[L, #]}&, Union[L]]
```

Once we find the frequency list of a walk, we can choose to look at whatever parts interest us, using various anonymous functions. For example, here is how we can find the sites in a walk that have been visited `t` times:

```
sitesVisitedTTimes[path_, t_] := Select[frequency[path], (#[[2]] == t)&]
```

`Select` is a built-in function that selects from its first argument (a list) all those elements for which the second argument (a function) evaluates to `True`. `==` is the equality predicate (`=` is reserved for a different use, namely for a form of assignment).

This function gives those sites visited `t` or more times:

```
sitesVisitedAtLeastTTimes[path_, t_] := Select[frequency[path], (#[[2]] >= t)&]
```

4.2 Pattern-matching

Another feature common in functional languages — and, in fact, elaborated to a far greater degree in MATHEMATICA than in most others — is **pattern-matching**. In this section, we use it to gather some further details about random walks.

Recursion

Though we've stressed the use of dynamically-allocated data structures (lists) and the use of function valuess as the essential features of functional programming, another characteristic of this style of programming is the use of **recursion** in function definitions. Unfortunately, most people find recursion a rather hard pill to swallow. This is unfortunate because many functions can be written most clearly using recursion. On the other hand, it is not needed quite as often in MATHEMATICA as in some other functional languages, because built-in functions can do so much of the work.

Recursively-defined functions are closely related to inductive proofs. For example, to define a function `f` on non-negative integers, we give the value of `f[0]` (the base case), and show how to obtain the value of `f[n]`, for arbitrary $n > 0$, assuming the values of `f[m]` are known for all $m < n$ (the inductive step).

The only new twist is that here we are most often interested in functions on *lists*, where the base case is `f[{}]` (`f` applied to the empty list) instead of `f[0]`, and for the inductive step we show how to obtain `f[p]` for arbitrary non-empty list `p`, assuming `f[q]` is known for all `q` shorter than `p`.

The simplest example is finding the length of a list. The base and inductive cases are:

Base case: `length[{}]` is zero.

Induction step: `length[p]` is one more than `length[Rest[p]]`. This is a valid inductive step, since `Rest[p]` is shorter than `p`.

Putting this together gives our definition:

```
length[p_] := If[p=={}, 0, length[Rest[p]] + 1]
```

Another simple example is `sumlist`, which adds all the elements of a list. The base and inductive cases are:

Base case: `sumlist[{}]` is zero.

Induction step: If `p` is a non-empty list, its sum is its first element plus the sum of `Rest[p]`; but the latter is, by induction, `sumlist[Rest[p]]`.

Thus, the definition is:

```
sumlist[p_] := If[p=={}, 0, p[[1]] + sumlist[Rest[p]]
```

The function `repeat` is used without definition in section 5.1. Here is a formal definition of it:

```
repeat[s, p, t] = p  if t[p] holds
                = repeat[s, Append[p, s[Last[p]]], t],  if t[p] fails to hold
```

(`Append` adds a new element at the end of a list; `Last[p]` is the last element in `p`.)

In MATHEMATICA, this can be defined very similarly to the formal definition above:

```
repeat[s_, p_, t_] := If[t[p], p, repeat[s, Append[p, s[Last[p]]], t]]
```

As our last example, a function that is defined in section 4.2 by using pattern-matching can be defined, a bit less concisely, using recursion. It is `runEncode`, which counts consecutive occurrences of identical elements in a list. For example, `runEncode[{1, 1, 2, 3, 1, 1, 1}]` returns `{{1, 2}, {2, 1}, {3, 1}, {1, 3}}`.

Base cases: Because the induction step will be applicable only when the list is of length two or greater, we need two base cases: `runEncode[{}]` = {}, and `runEncode[{x}]` = `{{x, 1}}`.

Induction step: The induction here is remarkably straightforward. Consider that we have a list L = {x_1, ..., x_n}, so `runEncode[Rest[L]]` = {{x_2, k}, ...}. How do we obtain `runEncode[L]` from this? Very simple: if $x_2 = x_1$, then `runEncode[L]` is {{x_2, $k+1$}, ...}; otherwise, it is {{x_1, 1}, {x_2, k}, ...}.

Rendering all this in MATHEMATICA, we have:

```
runEncode[L_] :=
   If[L=={}, {},
     If[Length[L]==1, {{L[[1]], 1}},
       With[{runencRest = runEncode[Rest[L]]},
         If[L[[1]]===runencRest[[1, 1]],
            Prepend[Rest[runencRest], {L[[1]], runencRest[[1, 2]]+1}],
            Prepend[runencRest, {L[[1]], 1}]]]]]
```

(`Prepend` is the same as `Append`, but adds the new element at the *end* of the list.)

It is debatable whether this code is "better" than the code in section 4.2. One thing we can say, though, is that for long lists it can run orders of magnitude faster.

Patterns are expressions, except that they have "blanks," written as sequences of one, two, or three underscores. An expression "matches" a pattern if the blanks in the pattern can be filled in so as to obtain the expression. We will consider only the case of pattern-matching in lists. Here are some patterns and explanations of what they match:

`{_, 1}`	Matched by any list with two elements, if the second element is 1
`{_, _}`	Matched by any two-element list
`{{_, _}, ___}`	Matched by any list whose first element is a list of length 2 (the "`___`" is matched by any sequence)
`{1, 2}`	Matched by the list `{1, 2}` and no other (this is the degenerate case of a pattern with no blanks)
`{{1, 2}, ___}`	Matched by any list whose first element is the list `{1, 2}`

Pattern-matching actually appears in a number of ways in MATHEMATICA. The simplest is in the built-in function `Cases`. `Cases[L, pat]` produces the sublist of `L` consisting of all those elements of `L` that match the pattern `pat`. As an example, here is one way to find those sites in a walk that were visited exactly t times, something we had previously done using `Select` (recall that `frequency` returns a list of pairs consisting of a point and a count):

```
sitesVisitedTtimes[walk_, t_] := Cases[frequency[walk], {_, t}]
```

A more complex use of patterns is in **transformation rules**. Whenever an expression is followed by the notation: "`//.` *pat* `->` *expr*", all occurrences of *pat* in the expression are transformed to *expr*; this transformation is repeated until it can no longer be applied. As an example, we can rewrite the function `sumlist` to add the elements in a list (see the "Recursion" box):

```
sumlist[p_] := (p //. {x_, y_, z___} -> {x+y, z})[[1]]
```

Here we have used *labelled patterns*, of the form "`x_`" and "`z___`". These are matched by the same expressions that match their unlabelled forms, but as side effects give the matched expressions the names `x` and `z`, respectively. After repeatedly applying the transformation to `p`, we're left with a list of one element, which is the sum of the elements in the original list.

Here is an application of transformation rules. Suppose we want to find the distribution of consecutive steps in any direction in a given random walk. That is, given the list `steps8`:

```
{{-1, 0}, {0, 1}, {0, 1}, {1, 0}, {0, -1}, {0, -1}, {-1, 0}}
```

we want to produce the list `{1, 2, 1, 2, 1}`, indicating that there is a non-recurring step in some direction, then two consecutive steps in another direction, another isolated step, and so on.

We can solve the problem in two steps:

1. Construct a list consisting of the runs of identical steps and their counts:

   ```
   {{{-1, 0}, 1}, {{0, 1}, 2}, {{1, 0}, 1}, {{0, -1}, 2}, {{-1, 0}, 1}}
   ```

   This will be done by a function called `runEncode`, which uses pattern-matching. It is shown below.

2. Make a list of all the second elements of the above lists. We have seen how to do this using `Transpose`.

Thus, our code is:

```
consecsteps[steps_] := Transpose[runEncode[steps]][[2]]
```

Obviously, the hard part of this problem is `runEncode`. We use pattern-matching and transformation rules to accomplish this. The argument to `runEncode` does not have to be a list of pairs. It can find runs in any list. For example, `runEncode[{1, 4, 4, 2, 1, 1, 1}]` will produce `{{1, 1}, {4, 2}, {2, 1}, {1, 3}}`. A well-known method in MATHEMATICA is to start by pairing each element with 1, then using the following transformation rule:

```
lis //. {u___, {v_, r_}, {v_, s_}, w___} -> {u, {v, r + s}, w}
```

In other words, repeatedly search for pairs of elements in `lis` whose first elements are the same, and add up their second elements. Putting this all together gives:

```
runEncode[L] := Map[({#, 1})&, L]//.{u___,{v_,r_},{v_,s_},w___} -> {u,{v,r+s},w}
```

Given this, we can run `consecsteps`:

```
In[]:= concsecsteps[steps8]

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

It is also interesting to know how many times there were sequences of n consecutive steps. We can use `frequency` again to find this:

```
In[]:= frequency[consecsteps[path]]

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

Finally, another kind of analysis is the study of reversal steps. The following code counts the number of "reversals". That is, it counts the number of times that a north step is immediately followed by a south step or a south step is followed by a north step or an east step is followed by a west step or a west step is followed by an east step.

We start with the the list of step increments.

```
In[]:= steps10 = walkSteps[10]

{{-1, 0}, {0, 1}, {0, -1}, {1, 0}, {0, -1}, {0, 1}, {0, -1}, {0, -1}, {0, 1}}
```

The analysis rests on the awareness that a step reversal is indicated by a step increment which is the negative of the preceding step increment (eg., ... `{-1,0},{1,0}`...`{0,1},{0,-1}`...).

It follows that if we add to each element in `steps10` the element immediately to its left (excepting the first element of the list which obviously has no such neighbor), the resulting list will indicate a reversal by the occurrence of a `{0, 0}`. This operation can be accomplished with an anonymous function which employs the built-in function `RotateRight` (see function `pathToSteps` in section 4.1):

```
In[]:= (Rest[# + RotateRight[#]])&[steps10]

Out[]= {{-1, 1}, {0, 0}, {1, -1}, {1, -1}, {0, 0}, {0, 0}, {0, -2}, {0, 0}}
```

The list indicates that the second step is followed by a reversal (a single reversal) and that the fifth step is followed by a reversal step which in turn is followed by another reversal step (a double reversal) and finally, that the eighth step is retraced by the last step (a single reversal).

We can apply `runEncode` to the preceding list:

```
In[]:= runEncode[(Rest[# + RotateRight[#]])& [steps10]]

Out[]= {{{-1, 1}, 1}, {{0, 0}, 1}, {{1, -1}, 2}, {{0, 0}, 2}, {{0, -2}, 1}, {{0, 0}, 1}}
```

and in the resulting list, the number of sequential reversals is indicated by the second component of each ordered pair whose first component is `{0,0}`.

```
In[]:= Cases[runEncode[(Rest[# + RotateRight[#]])& [steps10]], {{0,0},_}]

Out[]= {{{0, 0}, 1}, {{0, 0}, 2}, {{0, 0}, 1}
```

The various reversal runs are given by the second components of the elements of the above list. Putting this together:

```
revseq[steps_]:=
   Transpose[Cases[runEncode[(Rest[# + RotateRight[#]])& [steps10]],
                   {{0,0},_}][[2]]
```

Finally the frequency of reversal run lengths is obtained with the frequency function:

```
In[]:= revseqfreq[revseq_] := frequency[revseq]

In[]:= revseqfreq[revseq[steps10]]
Out[]= {{1, 2}, {2, 1}}
```

4.3 Random walk dimensions

A quantity of interest is the dimension or shape of a random walk. One measure of the shape is the square end-to-end separation of the walk. In our case, the random walk starts at the origin so this quantity is simply the sum of the squares of the components of the last step location of the walk.

For a particular path generated by evaluating `walk` one time the square end-to-end distance of the walk is given by

```
squaredist[path_] := path[[-1, 1]]^2 + path[[-1, 2]]^2
```

Recall that given list `L`, the expression `L[[i]]` denotes the i^{th} element of `L`, and `L[[-i]]` denotes the i^{th} element from the end. Thus, `path[[-1]]` is the last element in `path`. It is a pair of numbers. The first element of the pair is `path[[-1, 1]]`. Likewise, the y coordinate of the last step location is `path[[-1, 2]]`.

Of course, the ending location of a walk varies greatly from one randomly generated walk to another and what is most meaningful is the mean-square-end-to-end distance, realized over many executions of the walk. We can calculate this by generating a table of m paths and mapping the `squaredist` function onto the table to create the list of m square distances, adding up the elements of the list and dividing by m.

```
meansqdist[m_, n_] := Fold[Plus, 0, Table[squaredist[path[n]], {m}]]/m
```

(`Fold[...]` is equivalent to `Last[FoldList[...]]`.)

An alternative method is to use the built-in iterator function `Sum`:

```
meansqdist[m_, n_] := Sum[squaredist[path[n]], {m}]/m
```

A particularly interesting relationship exists between the mean-square-end-to-end distance and the number of steps in a random walk, `n`. To determine this we will need to generate a list `powerLawlist` = {`meansqdist[`n_1`, m]`, `meansqdist[`n_2`, m]`, ...}.

The creation of `powerLawlist` requires that `meansqdist` be evaluated for the same `m` and for different `n`. It would be useful to be able to convert meansqdist from a function which takes its arguments simultaneously to a function which is identical to meansqdist except that it takes its arguments one at a time. With this new function `powerLawlist` can be created by first supplying the value of `m` to the function and then mapping the result onto a list of `n` values.

We can define a higher-order function to do this which we will call "`curry`" (this function is well-known in the functional programming community where is sometimes called a partially applicative function).

```
curry[f_] := Function[x, Function[y, f[x, y]]]
```

We can illustrate the use of curry by changing the two-argument function, `Plus`, into the function add

```
In[]:= add = curry[Plus]

Out[]= Function[x$, Function[y$, x$ + y$]]
```

Note that `add` takes a function, `Plus`, as an argument and returns an upward function value, an anonymous function. We can illustrate the use of `add` simply :

```
In[]:= add[3]

Out[]= Function[y$, 3 + y$]

In[]:= %[4]

Out[]= 7
```

We can now create powerLawlist using `curry` and `meansqdist`. For specificity, we will consider m equal to 1000 and n equal to 50, 100, 150, and 200.

```
Map[curry[meansqdist][1000], Table[50i, {i, 1, 4}]]
```

To create `powerLawlist`, we need to create pairs consisting of the elements of the above list with the elements of `Table[50i, {i, 1, 4}]`. This can be done using `Transpose`:

```
powerLawlist = Transpose[{Map[curry[meansqdist][1000], Table[50i, {i, 1, 4}]],
                          Table[50i, {i, 1, 4}]}]
```

We can use powerLawlist to determine the power law relationship beween the mean-square-end-to-end distance and the number of steps by taking the Log of each part of each pair in the list and fitting the result to a polynomial using the built-in least-squares fitting function Fit:

```
Fit[N[Log[powerLawlist]], {1, x, x^2}, x]
```

(Log has the special property of being "listable," meaning that when applied to a list, it is automatically applied to each element of the list.)

Finally we can also obtain a log-log plot of meansqdist vs. n using the built-in graphics function LogLogListPlot:

```
LogLogListPlot[powerLawlist]
```

5 Other Walks

In this section, we describe and program some different models of random walks, namely first-passage problems:

Walk in the presence of traps. This is a walk that terminates when it hits any site in a set of sites called "traps."

Self-limiting walk. This walk terminates the first time it visits a location it has previously visited.

5.1 Walk in the presence of traps

In the simple walk model, the walk can always continue, and we choose its duration at the outset. In a walk with traps (as well as a self-limiting walk), termination of the walk is not predetermined. What is important, from a programming point of view, is that its length cannot be known in advance. While our version of walk using FoldList is fine for generating a lattice walk of known length, it does not apply in this case. We now present a different structure for that program, using the function repeat, whose MATHEMATICA definition is given in the "Recursion" box.

repeat[s, {a}, t] computes all the lists {a, s[a]}, {a, s[a], s[s[a]]}, and so on, until the first list is found for which the predicate t evaluates to true. For example, if s is the function that adds the square of an integer to its square root, and t tests whether the last element of a list is greater than 100, then repeat[s, {1}, t] produces the list {1, 2, 5.41421, 31.6406, 1006.75}:

```
In[]:= repeat[(#^2+Sqrt[#])&, {1.0}, (Last[#]>=100)&]

Out[]= {1, 2, 5.41421, 31.6406, 1006.75}
```

Here is how we can rewrite the simple random walk program using the repeat function

```
walk[n_] := repeat[(# + NESW[[Random[Integer, 1, 4]]])&,
                   {{0,0}},
                   (Length[#] == n)&]
```

The `repeat` function can be used for a lattice walk which terminates when the walk starting at the origin attempts to move out of the northeast quadrant, i.e. reaches the line $x = -1$ or the line $y = -1$, by giving as the terminating function `(Last[#][[1]] == -1 || Last[#][[2]] == -1)&`, where `||` can be read as "or".

We could also easily define a walk that terminates when it hits one of a finite set `T` of traps:

```
trapping[n_, T_] := repeat[(# + NESW[[Random[Integer, {1, 4}]])&,
                           {{0,0}},
                           (MemberQ[T, Last[#]])&]
```

(`MemberQ[L, x]` is true when `x` occurs in the list `L`.)

5.2 Self-limiting walk

You can think of the self-limiting walk as modelling a walker who plants a land-mine wherever he steps. Thus, the walk terminates as soon as the walker returns to a previously-visited site. The generation of the steps themselves is the same as in the simple walk and the trapping walk, so our code will be identical to that of `trapping`, except for the termination condition.

```
selfintersect[] := repeat[ (# + NESW[[Random[Integer, {1, 4}]]])&,
                           {{0,0}},
                           (MemberQ[Drop[#,-1], Last[#]])& ]
```

(`Drop[p, -1]` is `p` with its last element removed.) The termination function here tests whether the last element added to the walk being generated occurs among the previous elements of the walk.

6 Walks on complex terrain

From the beginning of this paper, we have passed functions into built-in functions like `Map` and `FoldList`, and the functiosn `curry` and `repeat`; in the "Recursion" box, we showed how `repeat` is defined.

In this section, we illustrate the utility of functions that return functions. This feature permits great flexibility in structuring code, and so allows us to develop a kind of "toolkit" for building random walks. This toolkit consists of a set of building blocks which can be combined in different ways to create a wide range of walks.

The domain in which our toolbox will provide solutions is that of "spatially-restricted walks," in which the terrain is filled with obstacles which limit the possible steps from any location. For example, the walker may be prevented from stepping below the x- and y-axes; or there may be a set of obstructions; or the walk may be confined to locations touched by a previous random walk (a "walk on a walk"). These sorts of walks are very typical of so-called "disordered" systems. Note that these are the converse of the trapping models; instead of terminating on certain steps, self-destructive moves are disallowed.

As an example, suppose we want a walk restricted to the northeast quadrant (this would be the converse of the trapping walk that terminates when it leaves this quadrant). We can adjust our previous (simple) walk to handle this: in choosing the next step to take, any point with negative abscissa or ordinate is excluded. Let us first define a function that randomly chooses an element from a list of arbitrary length:

```
randomChoice[lis_] := lis[[Random[Integer, {1, Length[lis]}]]]
```

and then one that returns, for any point p, the list of p's *legal* neighbors:

```
NEneighbors[p_] := Select[Map[(#+p)&, NESW],
                          (#[[1]] > -1 && #[[2]] > -1)&]
```

Now, we perform our walk exactly as before, except that we use `NEneighbors` to give the set of legal moves:

```
walkNE[n_] := repeat[(randomChoice[NEneighbors[#]])&, {{0,0}}, (Length[#] == n)&]
```

As a matter of elegance — or what computer scientists call "modularity" — we would prefer not to build the obstacles into the walk, as we've just done, but instead have one function for the walk, which accepts some sort of description of the terrain and then walks on that particular terrain. The first problem we need to confront is how to represent the terrain. Our first instinct might be to represent it as a list of the obstacles, or perhaps as a list of the free sites (i.e. sites not containing obstacles). However, either of these sets may be infinite (both are in the case of the northeast quadrant walk), so such a representation is not feasible. For our purposes, a convenient representation of the terrain is as a function taking any point to its set of neighbors:

Definition A *terrain* is a function that takes a point `{x, y}` to a subset of the set `{{x+1, y}, {x-1, y}, {x, y+1}, {x, y-1}}`.

For example, a terrain with no obstructions — the set of all grid points — corresponds to the function that takes any x, y to the set $\{x+1, y, x-1, y, x, y+1, x, y-1\}$. This can be defined in MATHEMATICA easily (the name `nn` stands for "nearest neighbors"):

```
nn[p_] = Map[(p+#)&, NESW]
```

`NEneighbors` above is another example of a terrain.

We want walks to be parameterized by terrains, so we need to make the terrain an argument to the function that creates the walks. From inspecting `walkNE`, it is clear how to do this: just pull out the terrain and make it an argument:

```
walkGen[legalnn_][n_] :=
     repeat[(randomChoice[legalnn[#]])&,
            {{0,0}},
            (Length[#] == n)&]
```

This definition bears some explanation. Like `curry` introduced earlier, `walkGen` takes one argument and produces a function that takes one argument. Another way of defining it would be:

```
walkGen[legalnn_] :=
     Function[n, repeat[(randomChoice[legalnn[#]])&,
                        {{0,0}},
                        (Length[#] == n)&]
```

or, equivalently:

```
walkGen := Function[legalnn,
     Function[n, repeat[(# + randomChoice[legalnn[#]])&,
                        {{0,0}},
                        (Length[#] == n)&]
```

Now, our first walk is identical to walkGen[nn]. That is, walkGen[nn] takes an argument n and produces a walk of n steps on the nn terrain, the one in which the walker can step in any direction at any time. Similarly, walkNE is the same as walkGen[NEneighbors].

6.1 Constructing terrains

Passing the terrain as an argument is an illustration of passing a function as an argument. Our goal in this section is to show how **upward function values** — functions created by other functions — can allow the construction of new terrains in a very elegant way.

We have already given the simplest terrain — the one with no obstructions. It was called nn. More interesting is the terrain that includes all points with a non-negative abscissa:

```
upperHalf[p_] = Select[nn[p], (#[[1]] != -1)&]
```

For example, upperHalf[{0, 0}] = {{0, 1}, {1, 0}, {-1,0}} and upperHalf[{2,2}] = {{2, 1}, {2, 3}, {1,2}, {3, 2}}. (It is also true that upperHalf[{-2, 2}] = {}, but if our walks starts at {0, 0}, we can never arrive at {-2, 2} anyway).

It is useful to generalize this to allow for a barrier at any arbitrary value of x, and to define a similar terrain containing a vertical barrier:

```
avoidXLine[x0_][p_] := Select[nn[p], (#[[1]] != x0)&]
avoidYLine[y0_][p_] := Select[nn[p], (#[[2]] != y0)&]
```

Thus, upperHalf is equivalent to avoidXLine[-1].

The benefit of this approach is that terrains can be easily combined. The most obvious combination is "superposition," i.e. forming a terrain that includes all the obstructions from either of two terrains; stated the other way around, it allows movement onto any point that is in both terrains. Thus it is the intersection of the two terrains, regarded as sets of points. It can be defined using our notion of terrains:

```
intersectTerrains[t1_, t2_][p_] := Intersect[t1[p], t2[p]]
```

where Intersect is the built-in function that takes the intersection of two lists (regarded as sets).

intersectTerrains has function arguments (t1 and t2) as well as a function result. That is, intersectTerrains[t1, t2] is a function taking any point p to a set of neighbors; i.e. it is a terrain. For example, the terrain intersectTerrains[avoidXLine[-1], avoidYLine[-1]] includes just the points in the northeast quadrant.

We can now write:

```
quadrant1 = intersectTerrains[avoidXLine[-1], avoidYLine[-1]]

walkQuad1 := walkGen[quadrant1]
```

so that walkQuad1 is equivalent to walkNE. Or we could confine the entire walk to an enclosed area. If {x0, y0} and {x1, y1} are the southwest and northeast corners of a bounding box (presumably containing the origin), then we can define a terrain that includes only the points within that box:

```
enclosingBox[{x0_, y0_}, {x1_, y1_}] :=
  intersectTerrains[avoidXLine[x0],
    intersectTerrains[avoidXLine[x1],
      intersectTerrains[avoidYLine[y0], avoidYLine[y1]] ] ]

walkInBox[swcorner_, necorner_] :=
    walkGen[enclosingBox[swcorner, necorner]]
```

Lastly, to produce a "walk on a walk," we take all the points visited on a first walk, and use those to define the terrain for a second walk. That is, the terrain will allow a step from a point to those neighboring points that were touched during the first walk. This is, in fact, very simple to define. If S is any list of points, define the following terrain:

```
allowedPoints[S_][p_] := Intersection[nn[p], S]
```

Suppose walk1 is the list of points visited in some walk. It is wise, for efficiency reasons, to remove duplicate points in walk1; this is easily done by writing Union[walk1]. This is now a set of points that can be used in selectedPoints. So, here is a walk on a walk:

```
walkOnWalk[m_] := walkGen[allowedPoints[Union[walk[m]]]]
```

A typical example is shown in Figure 2.

Figure 2

7 Conclusions

The MATHEMATICA programming language has been used here because of its widespread availability on many platforms and because of its large number of built-in mathematical quantities, its useful convention of naming built-in functions in fully spelled out English, and its relatively natural mathematical style syntax. However, it is not the only functional languages. Indeed, a great deal of effort has gone (and continues to go) into the design of functional languages, and there are currently some rather elegant and very efficient packages such as Standard ML, Scheme, Haskell and others. Thus, even if one wishes to limit one's use of the MATHEMATICA language to program prototyping (for which it is eminently well-suited by virtue of its interactive mode of operations, friendly interface, and extensive graphics capabilities), it is quite possible to use other functional languages rather than to fall back on FORTRAN or C, for full-scale program implementation. This is becoming increasingly desirable as computational tasks grow beyond number-crunching and become more multi-faceted.

References

[1] John Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Comm. ACM 21(8), Aug. 1978, 613–641.

[2] Richard J. Gaylord et al., **The Mathematica Programming Language: A Primer**, in preparation.

[3] Roman Maeder, **Programming in Mathematica, 2nd Ed.**, Addison-Wesley, 1991.

[4] Roman Maeder, Mathematica as a programming language, Dr. Dobbs Journal 187, 86–97, April 1992.

[5] Stephen Wolfram, **Mathematica: A System for Doing Mathematics by Computer, 2nd Ed.**, Addison-Wesley, 1991.

[6] M. N. Barber, B.W. Ninham, **Random and Restricted Walks: Theory and Applications**, Gordon and Breach, New York, 1970.

[7] George H. Weiss, Random walks and their applications, **American Scientist 71**, 65–71 (1983).

[8] George H. Weiss, Robert J. Rubin, RandomWalks: Theory and Selected Applications, in **Advances in Chemical Physics 52**, John Wiley & Sons, New York, 1983, pp. 363–503 .