Random Numbers

David Arnold

April 18, 2002

1 Introduction

If you are going to run simulations in mathematics, science, and engineering, then you must understand the concept of a random number. Good random number generators are extremely difficult to write. For example, consider this quote that comes directly from the manual for the `g77` compiler.

> As typically implemented (by the routine of the same name in the C library), this random number generator[1]is a very poor one, though the BSD and GNU libraries provide a much better implementation than the 'traditional' one. On a different system you almost certainly want to use something better.

Wow! They are trash talking their own random number generator.

What makes the generation of random numbers so difficult is the funny ways that seemingly random processes are not random at all. Let's begin with an example from *The Art of Computer Programming*, Volume 1, by Donald Knuth, Addison Wesley Press.

1.1 Middle Four

In the first few pages of *The Art of Computer Programming*, Knuth proposes the following idea. Take a four-digit integer, such as 5678, square it to get 32239684, then extract the middle four digits, 2396. Continue in this manner, squaring and extracting the middle four digits, to produce a sequence of seemingly random numbers.

$$5678239674088784158 6\ldots \tag{1}$$

This sequence of numbers seems to possess at least one important property of random numbers; that is, it seems equally likely that the next digit could be any one of the digits 0,1, . . . , 9. However, close inspection reveals that these numbers are not what we would really call "random."

First, the numbers are completely determined by the starting "seed." In this case, we started with the four-digit integer 5678. If we start the process a second time with the same seed, namely 5678, *we will get the exact same sequence of "random" digits.* This seems hardly random at all, and it is for this reason that we refer to these digits as *pseudo random numbers*.

Secondly, let's write a program to generate this sequence of digits, so that we can more closely inspect their eventual behavior. First, consider the function `midrand` in the module `needs`.

```
module needs
  public :: midrand
  integer, public, parameter :: eight=selected_int_kind(8)
contains
```

[1]This quote addresses the intrinsic function `rand` available via the `g77` compiler.

```
  function midrand(n) result (r)
    integer(kind=eight), intent(in) :: n
    integer(kind=eight) :: r
    r=n*n
    r=r/100
    r=modulo(r,10000)
  end function midrand
end module needs
```

This code warrants some explanation.

- With `eight=selected_int_kind(8)`, we define a kind number that guarantees enough storage for integers falling in the range $-10^8 < n < 10^8$.
- We then square the number with `r=n*n`.
- Integer division `r=r/100` "chops off" the last two digits. For example, with `n=5678`, `r=n*n=32239684`. Then `r=r/100=322396`.
- Next, division by 10000 leaves a remainder containing the last four digits. Remainders are found with Fortran's `modulo` command. Thus, `r=modulo(r,10000)=2396`.

Next, we craft a program to produce the sequence of random digits in (1). We set $n = 8$, evaluate the function `midrand` at this value, then iterate. That is, we take the value produced by the function `midrand`, 2396, then evaluate the function at this new integer, producing 7408. Continuing in this manner allows us to reproduce the string of digits in (1).

```
program mid4
  use needs
  integer, parameter :: how_many=1000
  integer :: counter
  integer(kind=eight) :: n=5678
  open(unit=1,file="random_data.txt",action="write",status="replace")
  n=midrand(n)
  do counter=1,how_many
     write(unit=1,fmt="(2i12)") counter, n
     n=midrand(n)
  end do
  close(1)
end program mid4
```

Note that the results at each step are stored in the file `random_data.txt`, along with the value of `counter`, which records the current step in the process. Running this program produces the sequence of numbers shown in Table 1. Note that the sequence of supposedly "random digits" begins to repeat in groups of four at step 26: 6100, 2100, 4100, 8100, and repeat this pattern indefinitely. Thus, a seemingly random process is not random at all. Of course, there are only so many 4-digit numbers (10^4 in all), so if we had thought a little bit more carefully, we would have conjectured that "selecting the middle four digits" has to repeat sooner or later[2]. Unfortunately, in the case where we start with a "seed-value" of `n=5678`, "sooner" occurs all too quickly.

[2]You simply run out of numbers to choose from.

step	digits	step	digits	step	digits	step	digits
1.	2396	10.	6406	19.	321	27.	2100
2.	7408	11.	368	20.	1030	28.	4100
3.	8784	12.	1354	21.	609	29.	8100
4.	1586	13.	8333	22.	3708	30.	6100
5.	5153	14.	4388	23.	7492	31.	2100
6.	5534	15.	2545	24.	1300	32.	4100
7.	6251	16.	4770	25.	6900	33.	8100
8.	750	17.	7529	26.	6100	34.	6100
9.	5625	18.	6858				

Table 1: Pseudo random numbers produced by `mid4`.

You should experiment with this code a bit before continuing. Ideally, if you can find a "good" starting value for n, then you can prolong the life of the sequence. If we can delay the repetition, then the sequence of digits can be thought of as "somewhat" random. Think of it as a contest, with the prize going to the programmer who discovers a starting value that produces the longest sequence of 4-digit numbers before repetition sets in. Post your results on the WebBoard![3]

1.2 Linear Congruential Generators

There is a large class of random number generators which is denoted by the name `Linear Congruential Generator`(LCG). There is a huge amount of literature on this topic, rich in number theory and statistics, most of which we will not have room to discuss in this activity. In general, LCG's will produce a sequence of "pseudo-random" numbers with the recurrence relation

$$x_{n+1} = (ax_n + c) \bmod m, \tag{2}$$

where a, c, and m are "magic" integers. In addition, an integer "seed" x_0 is chosen as a starting point for the sequence.

m,	the modulus;	$0 < m$
a,	the multiplier;	$0 \le a < m$
c,	the increment;	$0 \le c < m$
x_0,	the starting value (seed);	$0 \le x_0 < m$

The definition (2) of an LCG is a bit troubling, so we will break in down into manageable parts. We begin with the general recurrence relation, or recusively defined sequences.

1.2.1 Recursively Defined Sequences

A recursively defined sequence, or recurrence relation, is used to generate a sequence of numbers in a special way. One of the most common recuurence relations generates the next term of the sequence by

[3]This question actually raises a number of interesting issues in my mind, first and foremost of which is designing a technique to spot repetition in a sequence of numbers. Because the "mid4" technique is deterministic, as soon as we encounter a 4-digit number for the second time, ensuing numbers will be duplicates of the sequence of numbers occurring after the first occurrence of the 4-digit number.

I thought perhaps I could declare a logical array with `logical, dimension(10000) :: visited`. If the next number in the sequence is 5429, then I could do the following: `visited(5429)=.true.`. Then I can check the logical setting of `visited(n)` each time a number `n` is generated in the sequence.

Another thought I had was to use a linked list with pointers, but I don't know how to do that as yet.

operating with the preceding term in some manner. Consider, for example, the definition

$$x_{n+1} = x_n + 5, \qquad x_0 = 3. \tag{3}$$

This definition will generate a sequence of numbers

$$x_0, x_1, x_2, \ldots, x_n, x_{n+1}, \ldots. \tag{4}$$

The most important thing to note in the sequence (4) is the fact that x_n and x_{n+1} are *successive terms*. Indeed, x_{x+1} comes immediately after the term x_n in the sequence (4). Thus, a mathematician will look at the recurrence relation (3) and say: "The next term of the sequence, x_{n+1}, is found by adding 5 to its predecessor, x_n." In thinking like this, it is now a simple matter to see that the recurrence relation (3) generates the following sequence of numbers.

$$3, 8, 13, 18, 23, 28, 33, 38, \ldots \tag{5}$$

In a similar manner, the recurrence relation

$$x_{n+1} = 2x_n, \qquad x_0 = 3, \tag{6}$$

generates the sequence

$$3, 6, 12, 24, 48, 96, \ldots. \tag{7}$$

The key to understanding this example is to say: "The next term in the sequence, x_{n+1}, is found by multiplying its predecessor, x_n, by 2."

Finally, we combine the operations in definitions (3) and (6) in the following definition.

$$x_{n+1} = 2x_n + 5, \qquad x_0 = 3 \tag{8}$$

In this case, we say: "The next term in the sequence, x_{n+1}, is found by multiplying its predecessor x_n by 2, then we add 5 to the result." So, $x_0 = 3$. Then,

$$x_1 = 2x_0 + 5 = 2(3) + 5 = 11.$$

Next,

$$x_2 = 2x_1 + 5 = 2(11) + 5 = 27.$$

Continuing in this manner, we produce the following sequence of numbers.

$$3, 11, 27, 59, 123, 251, \ldots$$

Next, we must deal with the $\bmod$ in definition (2).

1.2.2 Modular Arithmetic

Simply put,[4] the modulus of a number is a remainder. For example, when we divide an integer by 5, the possible remainders are 0, 1, 2, 3, and 4. When we write

$$33 = 1 \bmod 8,$$

[4]Modular arithmetic is a vast study in the Theory of Numbers. We are greatly simplifying its definition and importance in this section. If you'd like to lean more about modular arithmetic, a good course in number theory can be fun, and Niven is a good author of number theory texts to read.

we are really saying that the remainder is 1 when we divide 33 by 8. If we divide 15 by 4, the remainder is 3, so we would write

$$15 = 3 \bmod 4.$$

Now, let's return to definition (2). Suppose that we set $a = 5$, $c = 1$, and $m = 16$, in definition (2). Then we get the following LCG.

$$x_{n+1} = (5x + 1) \bmod 16 \tag{9}$$

Let's use a starting or "seed" value, $x_0 = 1$. Then,

$$x_1 = (5x_0 + 1) \bmod 16 = (5(1) + 1) \bmod 16 = 6.$$

Note that $5(1) + 1 = 6$, and the remainder when we divide 6 by 16 is 6. Next,

$$x_2 = (5x_1 + 1) \bmod 16 = (5(6) + 1) \bmod 16 = 15.$$

Note that $5(6) + 1 = 31$, and the remainder when we divide 31 by 16 is 15. Continuing in this manner, we generate the following sequence of integers.

$$1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, \ldots \tag{10}$$

Now, think carefully and observe. We are dividing by 16, so the only possible remainders are 0, 1, 2, . . . , 15. This is only 16 possible remainders. So the sequence (10) produced by the recurrence relation (9) must repeat, and do so with 16 places! Of course, the repetition could (and does) occur sooner in some cases, but in the case (10) we have period 16. The 17th term in the sequence is 1, which was the original seed or starting value.

On the other hand, consider the recurrence relation

$$x_{n+1} = (5x_n + 1) \bmod 15, \qquad x_0 = 1. \tag{11}$$

A little effort shows that this definition produces the following sequence of integers, which for obvious reasons, will not provide a good sequence of "random" integers.

$$1, 6, 1, 6, 1, 6, 1, 6, \ldots$$

So, the game lies in picking "magic" numbers for a, c and m in definition (2). The hope is that appropriate choices of these parameters will provide a long sequence of apparently random integers. At this point, let me state that the theory behind the selection of "magic" numbers is rich in the Theory of Numbers, and so difficult as to be beyond the scope of this activity. If you'd like to learn more about this topic, the definitive work is *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, by Knuth.

1.2.3 A Simple LCG

In this section we will write and test an LCG function. The program is designed to work with large integers, because it is our intent to use this with some very large integers in place of the parameters a, c, and m in definition (2). We begin by writing a module containing our LCG function.

```
module lcg_needs
  public :: lcg
  integer, public, parameter :: long=selected_int_kind(15)
contains
```

```
  function lcg(a,c,m,x) result (res)
    integer(kind=long), intent(in) :: a, c, m, x
    integer(kind=long) :: res
    res=modulo(a*x+c,m)
  end function lcg
end module lcg_needs
```

We make a few remarks on this code.

- Declaring `lcg` as public allows access to this function in all program units that "use" this module.
- We let the compiler compute a kind number `long` that guarantess enough storage to hold an integer in the range $-10^{15} < n < 10^{15}$.
- All input to the function is declared to have `intent(in)` as we have no intention of changing any of these values, only using them to calculate the next term in the linear congruential sequence. The output `res` is also declared to have integer type having kind number equalling `long`.
- The intrinsic command `modulo(a*x+c,m)` computes the remainder when `a*x+c` is divided by `m`.

The program `lcg_test` will drive the function `lcg`. Because we use $a = 5$, $c = 1$, and $m = 2^4 = 16$, the program should produce the sequence (10).

```
program lcg_test
  use lcg_needs
  integer(kind=long), parameter :: a=5_long, &
       & c=1_long, m=2_long**4
  integer(kind=long) :: x=1, i
  do i=1,16
     print *, x
     x=lcg(a,c,m,x)
  end do
end program lcg_test
```

A few comments are in order here.

- We "seed" the generator with $x_0 = 1$. This is the first term in sequence (10).
- The `do` loop runs from 1 to 16, because we know that's the period of sequence (10).
- Note the code `x=lcg(a,c,m,x)`. This takes `x`, sends it to the LCG, then stores the result in the variable `x`. In this way, the variable `x` always contains the current term of the sequence.

It is comforting that this program produces the correct output; that is, the terms in sequence (10).

```
 1
 6
 15
 12
 13
 2
 11
```

```
 8
 9
 14
 7
 4
 5
 10
 3
 0
```

1.2.4 Magic Numbers

At this point, we are simply going to give you some "magic numbers" that we will use to generate a long sequence of pseudo-random integers, using the LCG function written in subsection 1.2.3.

$$x_{n+1} = (1664525x_n + 1013904223) \bmod 2^{32} \tag{12}$$

This requires only that we change our constants in our program `lcg_test`.

```
  integer(kind=long), parameter :: a=1664525_long, &
       & c=1013904223_long, m=2_long**32
```

We will start the sequence with the seed $x_0 = 0$.

```
  integer(kind=long) :: x=0, i
```

For purposes of checking, let's only write out the first 6 terms of the sequence to the screen.

```
  do i=1,6
```

Here is a complete listing of the changes.

```
program lcg_test
  use lcg_needs
  integer(kind=long), parameter :: a=1664525_long, &
       & c=1013904223_long, m=2_long**32
  integer(kind=long) :: x=0, i
  do i=1,6
     print *, x
     x=lcg(a,c,m,x)
  end do
end program lcg_test
```

These changes should produce the following sequence of integers.

```
 0
 1013904223
 1196435762
 3519870697
 2868466484
 1649599747
```

1.2.5 A Visual Test

In this section we will use a visual test to argue that our sequence of pseudorandom integers developed in subsection 1.2.4 appears to be random. We will do this by saving 1000 random pairs to a file, after which we will plot the pairs and inspect the result.

First, note that the recurrence relation

$$x_{n+1} = (ax_n + c) \bmod m \tag{13}$$

must always return a remainder between 0 and $m - 1$, inclusive. That is, the remainder must me one of the integers 0, 1, 2, ..., $m - 1$. Consequently, if we divide each of these remainders by m, we get a number that lies between 0 and 1, possibly including zero, but not 1.

We begin by adapting our `lcg_test` program from subsection 1.2.4.

```
program lcg_file
  use lcg_needs
  integer(kind=long), parameter :: a=1664525_long, &
       & c=1013904223_long, m=2_long**32
  integer, parameter :: num_pts=1000
  integer(kind=long) :: n=0, i
  real :: x, y
  open(unit=1,file="points.txt",action="write",status="replace")
  do i=1,num_pts
     n=lcg(a,c,m,n)
     x=real(n)/real(m)
     n=lcg(a,c,m,n)
     y=real(n)/real(m)
     write(unit=1,fmt="(2f15.6)") x, y
  end do
  close(1)
end program lcg_file
```

Some comments are in order.

- We use the same magic numbers as in subsection 1.2.4.
- A channel is opened to the file `points.txt` in the usual manner.
- We've changed our seed value and use `n=0` to start our sequence.
- Each time we call the function `lcg`, we divide the returned integer by m, thus obtaining a real number between 0 and 1. The first time we call `lcg`, the result is returned in `n`, divided by `m`, then stored in `x`. Then we iterate again, this time storing the result in `y`. Then we write the values in `x` and `y` to the file.
- When the `do` loop completes, we close the file, which now contains 1000 pairs (points), each coordinate a random number between 0 and 1.

Here are the first few entries of the file `points.txt`.

```
0.236068        0.278567
0.819534        0.667867
0.384077        0.621808
0.343726        0.640036
0.507778        0.581798
0.796726        0.737566
0.613829        0.885116
```

Note that each line represents an ordered pair (x, y), with $0 \leq x, y < 1$, in our thinking. What we would like to do now is plot each of the 1000 points in the file `points.txt` and see if they are randomly distributed on the rectangle R, defined by

$$R = \{(x, y) : 0 \leq x, y < 1\}. \tag{14}$$

However, just how are we going to do this? One possible method is to use Matlab.

1.2.6 Matlab

Matlab 6.1 is installed on the computers in the physical sciences building at College of the Redwoods. Here is how to access your data in preparation for plotting.

1. Open Matlab.
2. select Select Import Data ... from the File menu. Browse in the normal way until you find your file `points.txt`, select the file, then click the `Open` button to begin the process and open the Import Wizard.
3. Our data columns in `points.txt` are separated by spaces, so check the "spaces" column separator radio button in the Import Wizard. Note that previews of your data are available.
4. Click Next, then accept the default variable name (in my case this is `points`) and click the Finish button. The Import Wizard closes and you are returned to Matlab's command window.

If you now type

```
>> points
```

at the Matlab prompt, you will see the data in `points.txt` has been loaded into the variable `points`. We can plot the data in the second column of `points` versus its first column with the follwoing command.

```
>> plot(points(:,1),points(:,2),'.')
```

The syntax `points(:,1)` is read "every row, first column" of the matrix `points`. The syntax `'.'` is the marker style and causes points to be plotted as shown in Figure 1. Note that the points appear to be randomly distributed in the region $R = \{(x, y) : 0 \leq x, y < 1\}$.

This is a pretty nice technique, but not everyone can afford to buy the Matlab software for home use. Is there an alternative? You bet, and it's called Gnuplot.

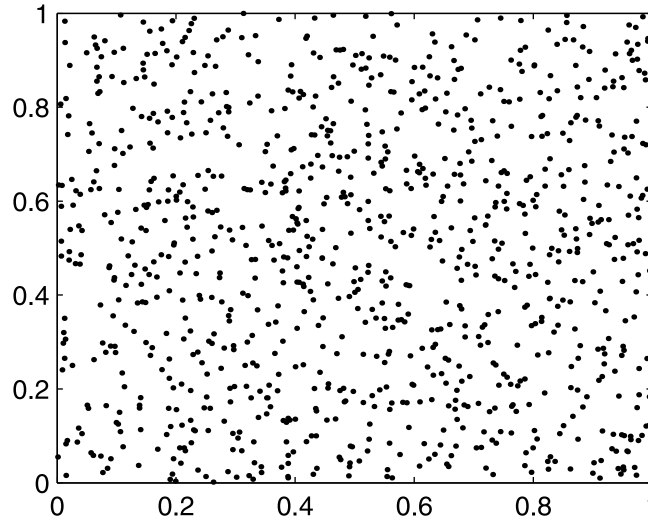

Figure 1: A random distribution of points.

1.2.7 Gnuplot

The home page for Gnuplot, called Gnuplot Central is located at the following url.

`www.gnuplot.info`

There are a number of mirrors[5] for Gnuplot, but we will use the following ftp site.

`ftp://ftp.gnuplot.info/pub/gnuplot/`

Once you reach the ftp site defined by the above url, you want to download the file `gp371w32.zip`. This is accomplished by right-clicking the file with your mouse and select "Copy to Folder" from the popup menu. Once you've completed the download, it's an easy install. Simply open the file `gp371w32.zip` in WinZip and select Extract. When the extract dialog comes up, select your c: drive for "Extract to," then check "All Files" and "Use Folder Names." This will create a directory called `gp371w32` on your c: drive. You will probably want to create a shortcut on your desktop that points to the file `c:/gp371w32/wgnupl32.exe`.

1. Start Gnuplot.

2. You must either change directories to the directory containing `points.txt`, or you must include the path as in the following Gnuplot command.

   ```
   gnuplot> plot "c:/classes/fortran/activities/RandomNumbers/points.txt" using 1:2
   ```

3. It's a bit easier if your first change directories with this command.

   ```
   gnuplot> cd "d:/classes/fortran/activities/RandomNumbers/points.txt"
   ```

4. Then you need only enter the following command to get the plot of the random data points in `points.txt`.

   ```
   gnuplot> plot "points.txt" using 1:2
   ```

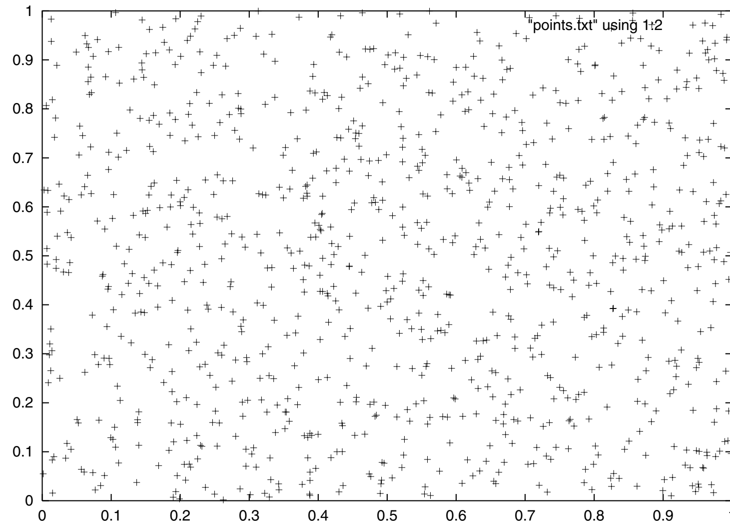

Figure 2: Plotting the random points with Gnuplot.

This last command provides the plot in Figure 2. Again, this seems satisfactory, with the points seemingly distributed in a random manner throughout R, as shown in Figure 2.

1.2.8 Fortran's Random Number Generators

Fortran has its own built-in random number generator. Here is the definition from *The F Programming Language*, Metcalf and Reid, Oxford University Press.

call random_number (harvest) returns a pseudorandom number from the uniform distribution over the range $0 \leq x < 1$ or an array of such numbers. `harvest` has intent `out`, may be a scalar or an array, and must be of type real.

We will now follow the lead of sections 1.2.6 and 1.2.7 and plot 1000 pairs of random numbers generated by `random_number`.

```
program builtin
  integer, parameter :: num_pts=1000
  integer :: i
  real :: x, y
  open(unit=1,file="builtin.txt",action="write",status="replace")
  do i=1,num_pts
     call random_number(x)
     call random_number(y)
     write(unit=1,fmt="(2f15.6)") x, y
  end do
  close(1)
end program builtin
```

There is nothing new to add here other than the call to Fortran's built-in random number generator with code `call random_number(x)`. This call invokes the `random_number` subroutine, and when complete,

[5]Alternate web and ftp sites that contain the same software and documentation files.

a random number between 0 and 1, possibly including 0, is stored in the real variable `x`. A call to Gnuplot produces the image in Figure 3. Again, the points in Figure 3 seem to be distributed randomly

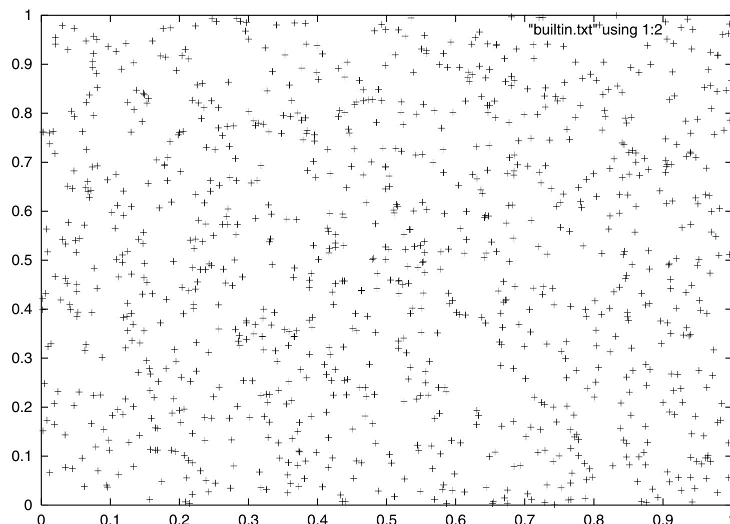

Figure 3: Random points built with the `random_number` routine.

throughout the region $R = \{(x, y) : 0 \leq x, y < 1\}$.

1.2.9 Other Random Numbers

It is important to note that the procedure `random_number` always returns a real number r such that $0 \leq r < 1$. What does the programmer do when random numbers are needed from other intervals? For example, what if the programmer needs a random number x such that $1 \leq x < 5$ and a second random number y such that $2 \leq x < 7$?

The answer is simple, but still a bit tricky. One need only manipulate the basic inequality $0 \leq r < 1$. For example, if we multiply all three members of the standard inequality by 4, we get

$$0 \leq 4r < 4 \tag{15}$$

Now, consider the following code fragment.

```
call random_number(r)
r=4*r
```

The call `call random_number(r)` returns to r a random number between 0 and 1. Then, the assignment `r=4*r` multiplies r by 4 and stores the result in r. By the inequality (15), this insures that r now contains a random number between 0 and 4.

Now, suppose we *shift* the random number by subtracting 2 from each member of the inequality (15).

$$-2 \leq 4r - 2 < 2 \tag{16}$$

Thus, the following code fragment produces a random between -2 and 2.

```
call random\_number(r)
r=4*r-2
```

In general, how might we produce a random number in the interval $a \leq r < b$? First, we multiply the standard inequality by the length of the interval, which is found by subtracting the left endpoint from the right; that is, $b - a$.

$$0 \leq r < 1$$
$$0 \leq r(b-a) < b-a$$

Next, shift the random number by adding the left endpoint a to each member of the inequality.

$$a \leq a + r(b-a) < b \tag{17}$$

The following program `interval` takes advantage of this idea to produce random numbers in the interval $1 \leq r < 5$.

```
program interval
  integer, parameter :: how_many=10
  integer :: i
  real :: x
  real, parameter :: a=1, b=5
  do i=1,how_many
     call random_number(x)
     x=a+(b-a)*x
     print *, x
  end do
end program interval
```

The key line in the program is the statement `x=a+(b-a)*x`, which uses the inequality (17) to produce random numbers between a and b. Here is the output from one run of program `interval`.

```
   2.9389486
   4.9100370
   4.9897566
   4.8346214
   1.4780432
   3.4721363
   2.1945326
   1.5104076
   3.4198890
   4.0753384
```

Note that each random number falls between 1 and 5, as desired. A second run provides a different set of random numbers.

```
   4.5174022
   1.9773008
   2.4935579
   3.2278280
   4.1055536
   4.0366716
   2.3409092
   1.6619644
   2.6362038
   4.6773634
```

This result is somewhat unexpected. What about the concept of the "seed" to begin the random number sequence that was discussed in the development of LCG's? Again, we quote from *The F Programming Language*.

call random_seed([size] [,put] [,get]) has the following arguments

- **size** has intent `out` and is a scalar default integer that the processor sets to the size n of the seed array.
- **put** has intent `in` and is a default integer array of rank one and size n that is used by the processor to reset the seed. A processor may set the same see value for more than one value of `put`.
- **get** has intent `out` and is a defult integer array of rank one and size n that the processor sets to the current value of the seed.

2 Monte Carlo Integration

Suppose that we wish to evaluate the definite integral

$$\int_0^1 x^2\,dx = \frac{1}{3}x^3\bigg|_0^1 = \frac{1}{3}(1)^3 - \frac{1}{3}(0)^3 = \frac{1}{3}. \tag{18}$$

We will proceed in a most unusual manner. First, we draw the graph of $f(x) = x^2$ on the interval $[0,1]$. Then we surround the graph with a rectangular box, again restricting our drawing to the interval $[0,1]$, as shown in Figure 4. Think of the box containing the graph of $f(x) = x^2$ as a dart board. We now

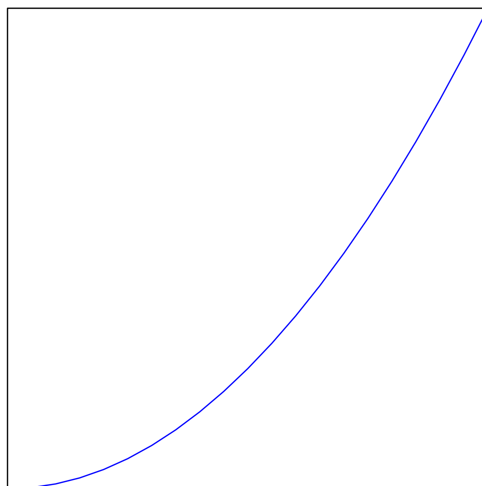

Figure 4: The graph of $f(x) = x^2$ on the interval $[0,1]$.

start tossing darts at the board in a random manner. That is, we throw the darts at the board with no semblance of skill or aiming technique. If a dart doesn't hit the board it is not counted. We've recorded our dart tosses as random dots on the board, as shown in Figure 5. Now, suppose that 100 darts hit the board randomly, as shown in Figure 5. Further, suppose that 30 of these 100 darts fall below the curve $f(x) = x^2$. Then, if the pattern is truly random[6], an approximation of the area under the curve is 30/100, or 0.30. Note that this is a fair approximation of the actual area under the curve, which is

[6]This is why we want "good" random number generators.

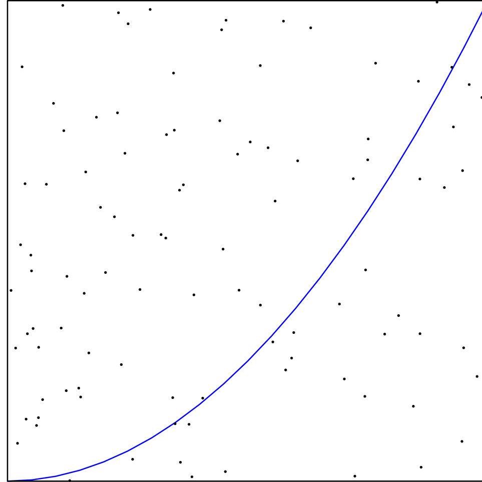

Figure 5: Tossing random darts at the region containing $f(x) = x^2$.

1/3. This technique of estimating the area under the curve via random dart tosses is called a *Monte Carlo* simulation.

Your assignment is to duplicate this simulation with a Fortran 90 program. Here are the requirements.

- First, let me add two additional definite integrals

$$\int_0^1 (x^2 + 2)\,dx = \frac{1}{3}x^3 + 2x\Big|_0^1 = \left(\frac{1}{3}(1)^2 + 2(1)\right) - \left(\frac{1}{3}(0)^3 + 2(0)\right) = \frac{7}{3}, \tag{19}$$

  and

$$\int_1^5 x\,dx = \frac{1}{2}x^2\Big|_1^5 = \frac{1}{2}(5)^2 - \frac{1}{2}(1)^2 = 12. \tag{20}$$

- Write three separate programs, `monte1.f90`, `monte2.f90`, and `monte3.f90`, one to approximate each of the integrals (18), (19), and (20) using the Monte Carlo technique. I want three hardcopy printouts, one of each of your programs.

- Each program is to write the random pairs representing each dart toss to a file. Name these files `monte1.txt`, `monte2.txt`, and `monte3.txt`.

- Using either Matlab or Gnuplot, obtain a printout of your results for each integration. Each plot should contain a plot of the function integrand. Superimpose the plot of the random points in the associated data file for the integral. Use the appropriate command for the technology you are working with (Matlab or Gnuplot) to label the plot with the area approximation reported by your program for that integral.

3 The Grading Rubric

The following rules will apply for this program, after which we will discuss and adjust the rubric during class.

1. (30 points) Will be awarded for adequate comments. Comments should include:

(a) A description of the program's purpose.
(b) Name, date, version or revision number.
(c) A complete dictionary of all variables and parameters used in the program.
(d) Interprogram comments should proceed any code snippets explained by the comments. These should be adequately sprinkled throughout your code.

2. (50 points) Will be awarded if the program works and does what it was asked to do.

3. (10 points) Will be awarded for good program style. This includes good indentation practices, etc.

4. (10 points) Will be awarded for creativity and extra effort. Did you just do the bare minimum? Or did you stretch and reach a little higher? Did you put something cute or clever into your program that nobody else seemed to think of?

4 Penalties

Each program that is assigned during the term will have a due date. On that date, the program must be on the instructor's desk before the start of class. Penalties will be assessed as follows.

1. (10 points) There will be a 10 point deduction for any program that is handed in after the class has begun.

2. (20 points) There will be a 20 point deduction per class period. That is, if you hand the program in one class period late, there is an automatic 20 point deduction. Two class periods warrants a 40 point deduction, etc. To be clear, if the program is in the instructor's hands before the beginning of the next class, that is a 20 point deduction. If the program is in the instructor's hands before the start of the second class period past the due date, that is a 40 point deduction, etc.

5 Managing Files and Folders

Each of you has been given personal space on the sci-math server to store your work. Typically, this space is mapped to the drive letter H. If you open the Windows Explorer (the file manager, not the internet browser), you can see that the drive letter has been mapped to your login name.

In this folder, create a new folder call FortranPrograms. Note that you must **never** use spaces in filenames. In the Windows operating system, filenames are not case-sensitive, which is exactly opposite what happens in Unix and Linux, where filenames are case-sensitive.

In your FortranPrograms folder, create another folder called Program10. It is in this folder that you are to place the source code and executables for this current project. Please name your program `program10.f90`. When you receive your next project, create a new folder called Program11 to hold that project, etc.

If you work at home, I still want you to place copies of your work in the space reserved for you on our system. Simply copy your home files onto a floppy disk and bring them with you to school. Use the Windows Explorer to copy the files on your disk into the proper folder (H:/FortranPrograms/Program10). Do not copy executables to the school drive. Rather, copy your source, then compile on the school computer to produce the executable.

If everyone follows these simple rules, I can easily access your work from my office machine for purposes of assigning a grade.

6 Caveat

On this project, if you stop by my office with hardcopy of your program before the due date of this assignment, I will give a quick glance and critique of your source code. Somewhat like receiving a grade on a draft before submitting your final draft for assessment.