An investigation of `rand`

Kenneth Chen

Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188

January 14, 1997

Abstract. Randomization has found applications in virtually all areas of computer science such as in cryptography and distributed computing. Many programmers trust and use the routines provided to them in ANSI C, without giving any thought to their correctness or reliability. Given that many applications require good random numbers, we perform a simple statistical test of the random number generator `rand` provided by the Standard C library.

Key words. `rand`, randomness

1 Introduction

In [1], Harbison and Steele give the following description of the random number generator `rand` provided by the Standard C library:

> Successive calls to `rand` return integer values in the range 0 to the largest representable positive value of type `int` (inclusive) that are the successive results of a pesudo-random-number generator. In ANSI C the facility is defined in `stdlib.h`, and the upper bound of the range of `rand` is given by `RAND_MAX`, which will be at least 32767.
>
> The function `srand` may be used to initialize the pseudo-random-number generator that is used to generate successive values for calls to `rand`. After a call to `srand`, successive calls to `rand` will produce a certain series of pesudo-random-numbers. If `srand` is called again with the same argument, then after that point successive calls to `rand` will produce the same series of pesudo-random numbers. Successive calls made to `rand` before `srand` is ever called in a user program will produce the same series of pesudo-random numbers that would be produced after `srand` is called with argument 1.
>
> C library routines should not call `rand` or `srand` in a way that affects the programmer's observed sequence of pseudo-random numbers.

It is of interest to know just how "random" the random numbers generated by `rand` are. Many programmers instinctively trust and use the routines provided to them in ANSI C without giving a moment's thought to their correctness or reliability. Given that many applications such as cryptography and Monte Carlo simulations require good random numbers and given the popularity of `rand`, its reliability should be studied.

In this note, we apply a naive statistical test to sequences of numbers generated by `rand` to investigate just how "random" they are. We also apply the test to the more sophisticated `drand48`.

2 The Statistical Test

Given a natural number n, we use `rand` and `drand48` (seeded with 1) to generate a sequence of n 0's and 1's, say $x_1, x_2, \ldots, x_n$ and compute:

$$w_1 = \frac{\sum_{i=1}^{n} x_i \; - \; n/2}{\sqrt{n/4}}$$

We then generate another sequence of n 0's and 1's (using seed 2), say $x_{n+1}, x_{n+2}, \ldots, x_{2n}$ and compute:

$$w_2 = \frac{\sum_{i=n+1}^{2n} x_i \; - \; n/2}{\sqrt{n/4}}$$

We do this a total of n times, computing $w_1, w_2, \ldots, w_n$, where:

$$w_i = \frac{\sum_{i=(i-1)n+1}^{in} x_i \; - \; n/2}{\sqrt{n/4}}$$

We plot histograms of the w_i for various n. If the x_i are "random", then we would expect that the w_i to be normally distributed.

3 Implementation

Here is our `main()` routine:

```
main( int argc, char *argv[] )
{
    int i, n, seed;

    if( argc < 3 )
    {
        explain_arguments();
        return;
    }
    else
        n = atoi(argv[1]);

    if( argc == 3 )
```

```
            for( i = 1; i <= atoi(argv[2]); i++ )
                printf( "%f\n", (makernd( n, i ) - (n/2.0) )/sqrt(n/4.0) );
        else if( argc == 4 )
            for( i = atoi(argv[2]); i <= atoi(argv[3]); i++ )
                printf( "%f\n", (makernd( n, i ) - (n/2.0) )/sqrt(n/4.0) );
        else if( argc == 5 )
            for ( i = atoi(argv[2]); i <= atoi(argv[3]); i += atoi(argv[4]) )
                printf( "%f\n", (makernd( n, i ) - (n/2.0) )/sqrt(n/4.0) );
        else
        {
            explain_arguments();
            return;
        }
    }
```

At the command line, we can type

```
$ a.out 10 10
```

and the program that tests `rand` will return

```
-0.632456
1.264911
-0.632456
-1.264911
1.264911
-1.264911
0.000000
0.000000
-1.897367
0.632456
```

As can be seen from our `main()` routine, our program has a slightly more sophisticated command-line interface. The commands

```
$ a.out 10 1 10
```

and

```
$ a.out 10 1 10 1
```

will also return the same result. The program generates 10 random sequences of 0's and 1's of length 10 with each sequence seeded with $1, 2, \ldots, 10$ respectively, and then outputs the values of $w_1, w_2, \ldots, w_{10}$.

Using an optional third and fourth argument, we can change the seeding of the sequences. If we enter

```
$ a.out <n> <p> <q>
```

then the seeds are the integers from p to q. That is, $q-p+1$ sequences of 0's and 1's of length n are generated and the respective w values outputted. If we enter

```
$ a.out <n> <p> <q> <r>
```

then the seeds are the integers from p to q, in steps of r.

Here is our routine to test `rand`:

```
#define characteristic(x) ((x) < 0.5 ? 0 : 1)

int makernd( long int n, long int seed )
{
    int i, sum = 0;

    srand(seed);

    for( i = 1; i <= n; i++)
    {
        double tmp = rand()*1.0/RAND_MAX; /* RAND_MAX: see p.252 of K&R */
        int rand_bit = characteristic(tmp);

        /* could print actual values returned by rand here & other info */

        sum += rand_bit;
    }

    return( sum );
}
```

The routine is pretty straightforward. It takes two arguments n and *seed*, seeds `rand` with *seed*, generates n random numbers (mapped to $\{0,1\}$ using the `characteristic` macro) and returns their sum.

Here is our routine to test `drand48`:

```
#define characteristic(x) ((x) < 0.5 ? 0 : 1)

int makernd( long int n, long int seed )
{
    int i, sum = 0;

    srand48(seed);

    for( i = 1; i <= n; i++)
    {
        double tmp = drand48();
        int rand_bit = characteristic(tmp);

        /* could print actual values returned by rand here & other info */

        sum += rand_bit;
    }

    return( sum );
}
```

This routine is similar to our routine that tests `rand`.

4 Results

We put our routine to test `rand` into an executable called `test_rand` and our routine to test `drand48` into `test_drand48`, and then executed the following commands:

```
$ test_rand 100 100 > rand100.data
$ test_rand 1000 1000 > rand1000.data
$ test_rand 5000 5000 > rand5000.data
$ test_rand 10000 10000 > rand10000.data
$ test_drand48 100 100 > drand48.100.data
$ test_drand48 1000 1000 > drand48.1000.data
$ test_drand48 5000 5000 > drand48.5000.data
$ test_drand48 10000 10000 > drand48.10000.data
```

In order to analyze the resulting data, we executed the following commands for each of the above `.data` files

```
$ sort -n <data file name> | uniq -c |awk -f 2math
```

The `sort -n` command sorts the numbers in a `.data` file in numerical order. The `uniq -c` command then takes the sorted numbers and gives us the number of occurrences of each "w" value. For example,

```
$ test_rand 10 10 | sort -n | uniq -c
```

will return

```
1 -1.897367
2 -1.264911
2 -0.632456
2 0.000000
1 0.632456
2 1.264911
```

We then wrote the following script using the AWK programming language [2], which we called `2math` above:

```
    { x[NR] = $1; y[NR] = $2 }
END { printf( "ListPlot[{\n" )

      for( i = 1; i < NR; i++ )
          printf( "{%f,%d},\n", y[i], x[i] )

      printf( "{%f,%d}}, AxesOrigin->{%f,0.0}\n]\n", y[NR], x[NR], y[1] )
    }
```

From the output of the `uniq` command, the above script produces Mathematica code that can be used to generate a histogram.

The resulting output from Mathematica is at the end of this document. In all of the plots, the horizontal axis represents values of w and the vertical axis represents the number of occurrences.

5 Conclusion

From the histograms shown, the frequency of the various values of w appear to be normally distributed and so `rand` and `drand48` pass our naive statistical test, but unfortunately, this does not prove that they are good random number generators.

References

[1] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual.* Prentice Hall, 3rd edition, 1991.

[2] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The Awk Programming Language.* Addison-Wesley Publishing Company, 1988.

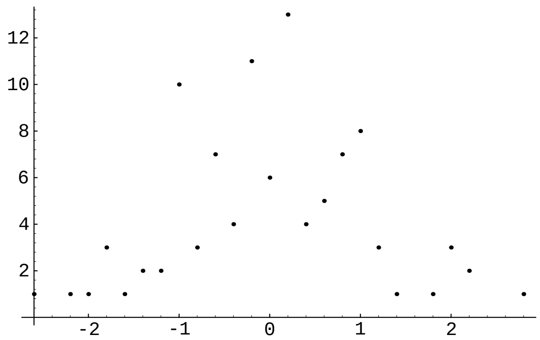

Figure 1: `rand`, $n = 100$

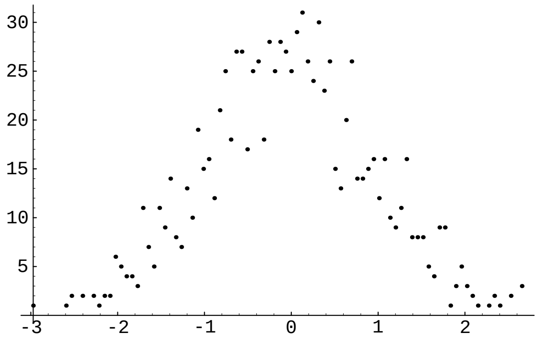

Figure 2: `rand`, $n = 1000$

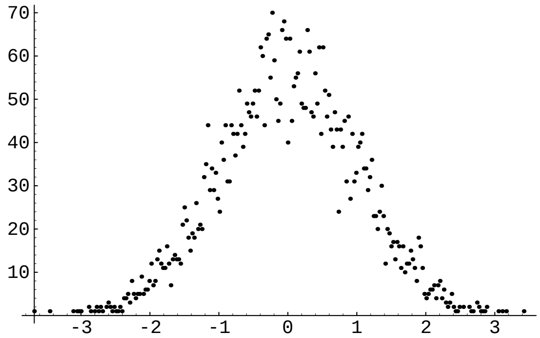

Figure 3: `rand`, $n = 5000$

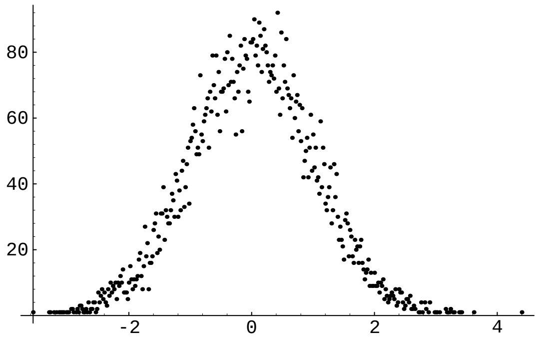

Figure 4: `rand`, $n = 10000$

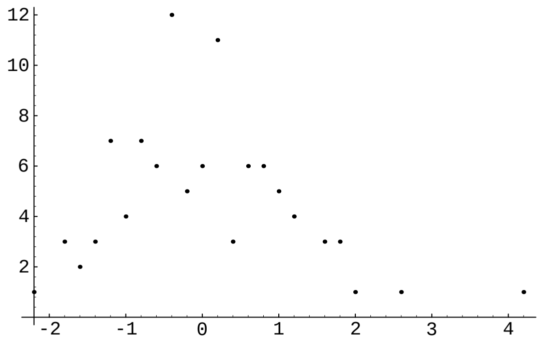

Figure 5: `drand48`, $n = 100$

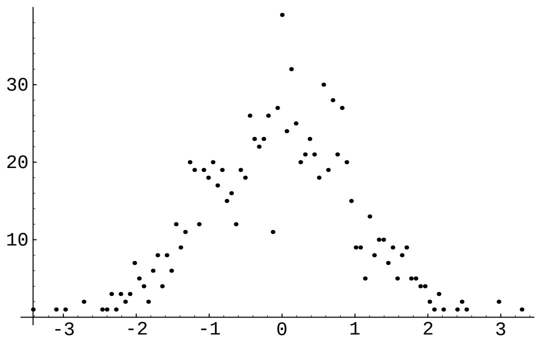

Figure 6: `drand48`, $n = 1000$

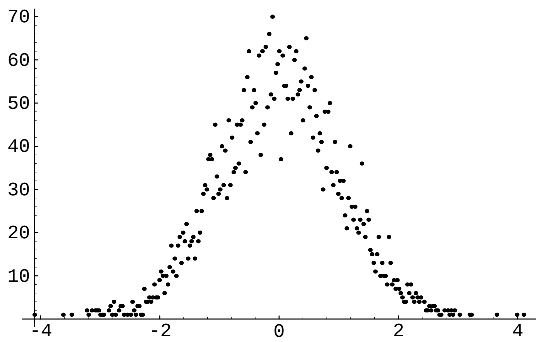

Figure 7: `drand48`, $n = 5000$

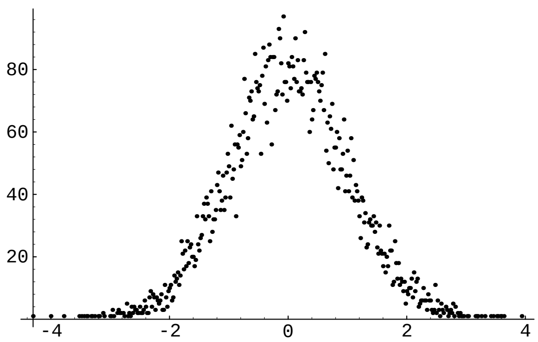

Figure 8: `drand48`, $n = 10000$