TME – Assignment: testing random number generators

The "random number generators" we find in all programming languages and libraries are in fact "pseudo-random number generators", based on one or another algorithm. These generators can be tested to check the quality of the sequences they provide.

In this exercise you will implement such a test.

- Read chapter 7 of "Numerical recipes":

- https://www.cec.uchile.cl/cinetica/pcordero/MC_libros/NumericalRecipesinC.pdf
- https://websites.pmc.ucsc.edu/ fnimmo/eart290c_17/NumericalRecipesinF77.pdf

- Choose a random number generator to work with: look for the details of its implementation (algorithm, etc.) and then implement a "bad one" (as discussed in the book).

- Select also a "good" random number generator from your programming language. Find out from the language documentation how this generator has been implemented and document it.

- Implement four tests of the generators to compare and validate them. The following references are suggested to select the tests:

- https://www.random.org/analysis/
- https://csrc.nist.gov/Projects/Random-Bit-Generation/Documentation-and-Software/Guide-to-the-Statistical-Tests
- https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf
- https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software
- http://webhome.phy.duke.edu/ rgb/General/dieharder.php
- http://simul.iro.umontreal.ca/testu01/tu01.html

- Justify the choice and then discuss the results and the comparison of the "good" and "bad" generators.

- Can you imagine a situation in a simulation/experiment where the faults of the "bad" generator could affect the outcome?

Deliver in this page a report describing the above activities; include any program you have used. Pack all the deliverables in a single zip file.

1 Creating random generators:

I've chosen to try 2 different pseudo-random number generators from Chapter 7 of Numerical recipes [1].

First I'll try the one from page 276, which is a straight forward linear congruential generator with $a = 1103515245$, $c = 12345$ and $m = 2^{32}$, which are not particularly good choices, but neither gross embarrassments (period $\approx 2^{30} \approx 10^9$):

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>


unsigned long next=1;

int ran0(void){
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

void sran0(unsigned int seed){
  next=seed;
}


int main(){

  sran0(8989743);  //We set the seed here
  int N =100000;
  double ran[N];

  for(int i = 0; i<N; i++){
    ran[i]= static_cast <float> (ran0()) / static_cast <float> (RAND_MAX);
  }

  FILE* fichero;
  fichero = fopen("Random_numbers1.txt", "w" );
  for ( int i = 0; i < N; i++){
    fprintf(fichero, "%f \n", ran[i]);
  }

  return 0;

}
```

then I'll try this same implementation but with very bad values, such as the ones IBM used in it's RANDU routine $a = 65539$, $c = 0$ and $m = 2^{31}$, which later should give bad results in the tests.

And finally I also want to try from page 279 a Minimal standard generator that applies Schrage's algorithm, with $a = 7^5 = 16807$, $m = 2^{31} - 1 = 2147483647$, $q = 127773$ and $r = 2836$, which should be better than the first one (period $\approx 2^{31} - 2 \approx 2.1\ 10^9$) :

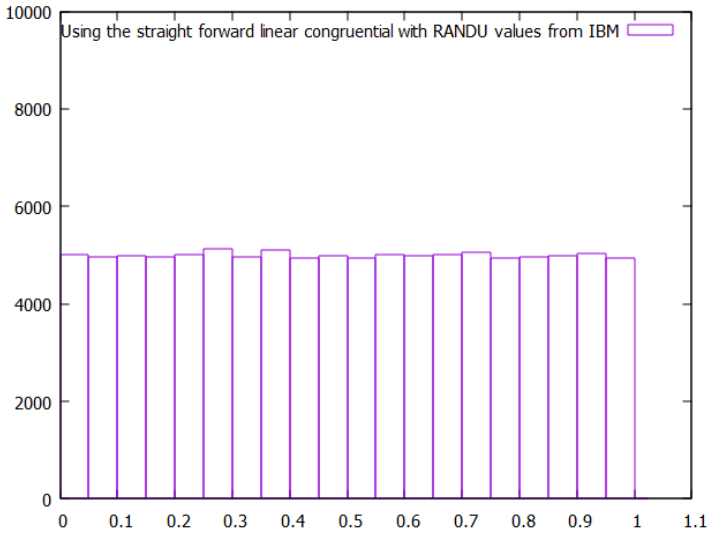
```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define IA 16807
#define IQ 127773
```

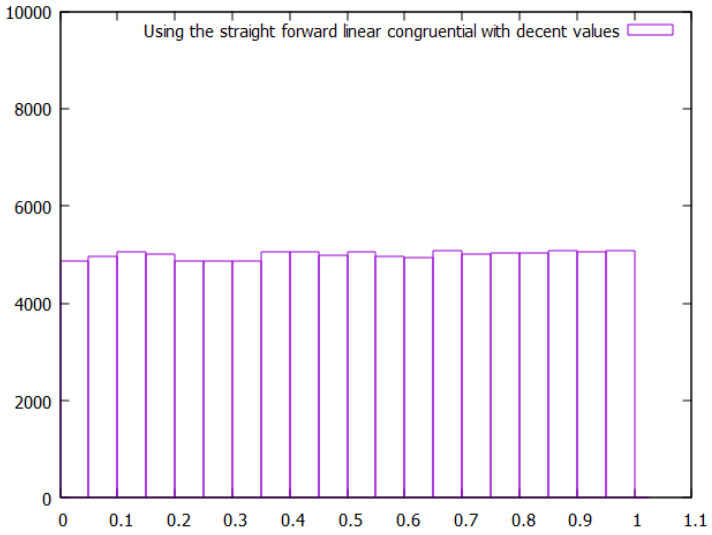
```
#define IM 2147483647
#define IR 2836
#define AM (1.0/IM)


unsigned long next=1;

void sran0(unsigned int seed){
  next=seed;
}

float ran0(void){
  long k;
  float ans;

  k=next/IQ;
  next=IA*(next-k*IQ)-IR*k;
  if (next <0 ){
    next += IM;
  }
  if (next>IM){
    next -= IM;
  }
  ans=AM*next;
  return ans;

}



int main(){

  sran0(8989743);  //We set the seed here
  int N =100000;
  double ran[N];

  for(int i = 0; i<N; i++){
    ran[i]= ran0();
  }


  FILE* fichero;
  fichero = fopen("Random_numbers.txt", "w" );
  for ( int i = 0; i < N; i++){
    fprintf(fichero, "%f \n", ran[i]);
  }

  return 0;

}
```

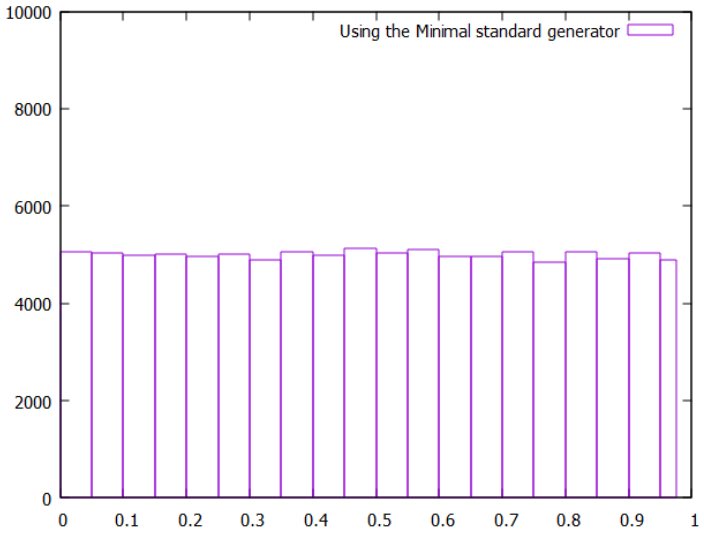
Which using this gnuplot code to plot the obtained data:

```
reset
cd 'C:\CARRERAS\MASTER UB\1r Semestre\Metodes Estadistics i matematics\
    Excercices\Random Numbers'

set yrange [0:10000]
binwidth=0.05
bin(x,width)=width*floor(x/width)

plot 'Random_numbers0.txt' using (bin($1,binwidth))+binwidth/2:(1.0) smooth
     freq with boxes
```

We get the following distributions for N=100.000 random numbers and a chosen seed = 8989743:





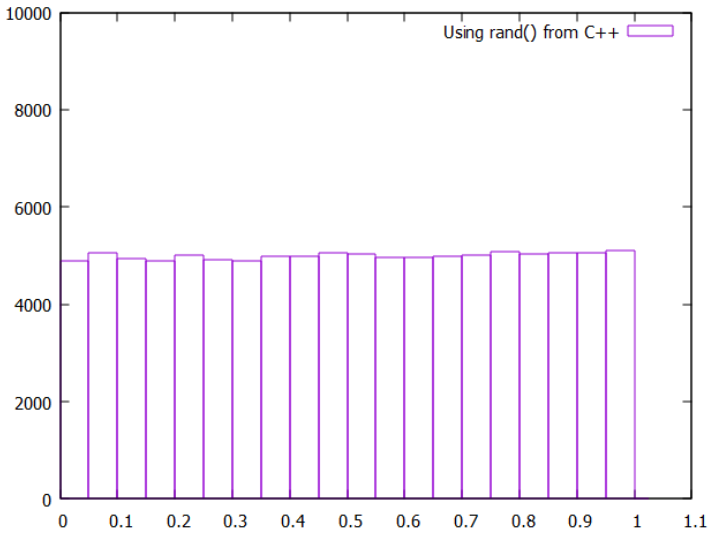


2 Choosing a "good" random number generator:

Now, we are going to choose the function rand(), imported from " #include <stdlib.h>", to do so, this time we don't need to define our random function, which is given, but instead we need to normalize it to give values from 0 to 1:

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>


int main(){

  srand(8989743);  //We set the seed here
  int N =100000;
  double ran[N];

  for(int i = 0; i<N; i++){
    ran[i]= static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
  }

  FILE* fichero;
  fichero = fopen("Random_numbers2.txt", "w" );
  for ( int i = 0; i < N; i++){
    fprintf(fichero, "%f \n", ran[i]);
  }

  return 0;
}
```

Again, for the same values of N and the seed as before, this gives a distribution:



So, now we need to find it's implementation, which after some research, we find that is given by:

```
/* Return a random integer between 0 and RAND_MAX.  */
int
rand (void)
{
  return (int) __random ();
}
```

Which calls the function _random(), that at the same time calls a function _random()_r:

```
/* If we are using the trivial TYPE_0 R.N.G., just do the old linear
   congruential bit. Otherwise, we do our fancy trinomial stuff, which is
   the same in all the other cases due to all the global variables that
   have been set up. The basic operation is to add the number at the rear
   pointer into the one at the front pointer. Then both pointers are
   advanced to the next location cyclically in the table. The value
   returned is the sum generated, reduced to 31 bits by throwing away the
   "least random" low bit.

   Note: The code takes advantage of the fact that both the front and rear
   pointers can't wrap on the same call by not testing the rear pointer if
    the front one has wrapped. Returns a 31-bit random number.  */

int
__random_r (buf, result)
     struct random_data *buf;
     int32_t *result;
{
  int32_t *state;

  if (buf == NULL || result == NULL)
    goto fail;

  state = buf->state;

  if (buf->rand_type == TYPE_0)
    {
      int32_t val = state[0];
      val = ((state[0] * 1103515245) + 12345) & 0x7fffffff;
      state[0] = val;
      *result = val;
    }
  else
    {
      int32_t *fptr = buf->fptr;
      int32_t *rptr = buf->rptr;
      int32_t *end_ptr = buf->end_ptr;
      int32_t val;

      val = *fptr += *rptr;
      /* Chucking least random bit.  */
      *result = (val >> 1) & 0x7fffffff;
      ++fptr;
      if (fptr >= end_ptr)
    {
      fptr = state;
      ++rptr;
    }
      else
    {
      ++rptr;
      if (rptr >= end_ptr)
        rptr = state;
    }
      buf->fptr = fptr;
      buf->rptr = rptr;
    }
  return 0;

 fail:
  __set_errno (EINVAL);
  return -1;
}
```

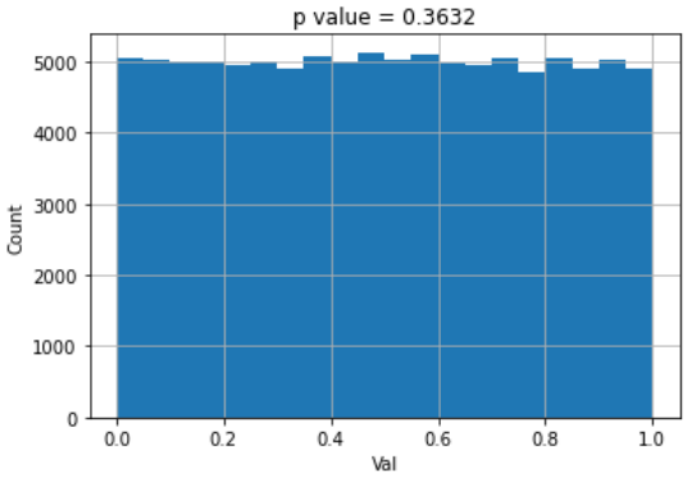
Which at the end, we see is another linear congruential generator with $a = 1103515245$ and $c = 12345$ such as the first one we choose, that plays more with cycling pointers and the size of the values, to be more efficient and give a better pseudo-random generator.

3 Implement four tests of the generators

- Looking at the provided references a Chi-square test, to test for uniformity is very recommended, given it is mentioned in various of the references.

To do this I'll use python, so first I'll pass the .txt obtained from the C++ pseudo-random number generator, to a .csv, and then I'll use pandas and numpy to pass it into an array in Python, from which I can run the Chi-square test, as shown in class:

```
1
2  import scipy.stats as scp
3  import matplotlib.pylab as plt
4  import numpy as np
5  import pandas as pd
6
7  df=pd.read_csv('C:\CARRERAS\MASTER UB\1r Semestre\Metodes Estadistics i
       matematics\Excercices\Random Numbers\Random_numbers.csv',  header =
       None, names=['random'])
8  x=df.to_numpy()
9
10 # Plot histogram; we use the hist function to recover the counts per bin
11 h= plt.hist(x,20) # h[0] will be the counts per bin
12 plt.grid(True)
13 plt.xlabel('Val')
14 plt.ylabel('Count')
15
16 # Run test
17 c2_stat, p_val = scp.chisquare(h[0])
18
19 t = plt.title("p value = {:.4f}".format(p_val))
20 plt.show()
21
22
```



from where we obtain p-values, for each of our 4 generators:

- Decent straight forward linear congruential: $p = 0.9367$
- Bad straight forward linear congruential (RANDU): $p = 0.2891$
- Minimal standard generator: $p = 0.3632$
- Rand() from C++: $p = 0.5869$

seeing that all the generators passed this test with a significance level of $\alpha = 0.25$ for each case, they also pass it for 5 more different seeds, which I tested.

- The next test we are going to do are tests of run above and bellow the median, also mentioned in various references as adequate ones, for this, we are going to add to our original C++ codes of each pseudo-random number generator this peace of code:

```
int counter=0;
for (int i=0; i<N ;i++){
  if (ran[i]<0.5){
    counter+=1;
  }
}
printf("%d", counter);
```

For which we obtain the quantity of numbers that are in the wrong side of the expected median for each case, doing this for diverse initial seeds, and applying the stadistic:

$$z = \frac{(x \pm 0.5) - \mu_x}{\sigma_x}$$

we need that $|z| < 2$ to pass the test with significance level $\alpha = 0.05$, and we obtain:

- Decent straight forward linear congruential: $z = -0.121$
- Bad straight forward linear congruential (RANDU): $z = 0.373$
- Minimal standard generator: $z = 0.157$
- Rand() from C++: $z = -0.105$

where we see that again, all test give satisfactory results, passing the test for significance level $\alpha = 0.05$, with great margin.

- Now, the following test is gonna be the Reverse Arrangements test, which will let us check if there is any correlation between the increase of the random generated numbers and the increase inside the computation time, or the opposite:

$$\sum_{i=1}^{N} \sum_{j=1(<i)}^{N} h_{ij} \qquad \text{where } h_{ij} = 1 \text{ if } rand[i] > rand[j] \text{ or } 0 \text{ else}$$

this is implemented as:

```
int counter=0;
for (int i=0; i<N ;i++){
  for (int j=0; j<N; j++){
    if (i=j+1){
      if (ran[i]<ran[j]){
        counter+=1;
      }
    }
  }
}
printf("%d",counter);
```

for which we wanna obtain a result of $N/2$, for the generation to not be correlated (in this case N=100000). The obtained results are:

- Decent straight forward linear congruential: 49967

- Bad straight forward linear congruential (RANDU): 49983
- Minimal standard generator: 49918
- Rand() from C++: 49984

from where we see that all the number generators have good results, maybe the Minimal standard generator, have them a bit lower than the rest.

- And the final one will be the Overlapping Sums test, where we will sum the square difference deviation from the uniform distribution in each bin of the histogram:

$$\chi^2 = \sum_{i=1}^{Nbins} \frac{(O_i - E_i)^2}{E_i}$$

to compute this we will use:

```
counter=0
for i in range(len(h[0])):
    print(h[0][i])
    counter+=(h[0][i]-5000)**2/5000

print(counter)
```

For which we obtain, for each case a χ^2:

- Decent straight forward linear congruential: $\chi^2 = 10.59$
- Bad straight forward linear congruential (RANDU): $\chi^2 = 21.90$
- Minimal standard generator: $\chi^2 = 20.53$
- Rand() from C++: $\chi^2 = 17.04$

checking the tables, for our $Nbins - 1 = 19$, and $\alpha = 0.05$ we get that χ must fullfill:

$$\chi^2 < 30.144$$

which all of our test fullfilled!

4 Discussion

The comparison, hasn't given much insight in which generators are better, maybe this is related with the fact, that I was using a laptop and the maximum length of the arrays of random numbers was of the order of 10^5, which isn't near the periodicity of any of the generators method.

But even with the little differences, we can see in the Overlapping Sums test, that the decent straight forward linear congruential and the rand() from C++, where the two that performed better, when taking into account the square of the difference. Maybe if we did even more experiments, we could see a clear trend in favor of the "better" ones, but with the experiments we have used, it is not enough.

If we needed arrays of the order of 10^{10} random numbers, I imagine, the results would be pretty different, because in the bad ones, we would make the errors bigger each "cycle" when we repeat them, and the Overlapping Sums test, for example would notice that pretty quick.
But even if the test, wasn't strong enough, or we could have performed them better, the selection of this concrete test seem pretty good, we tested different things with each, that gave us an idea of how good the random generators are. The main ideas that we tested were:

- If the bins of a histogram follow a uniform distribution.
- If only if, half of the values are passed the 0.5 median value.
- If the values tend to get bigger or smaller in some period of the simulation.

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, Second. Cambridge, USA: Cambridge University Press, 1992.