Homework 1

Davide Berasi

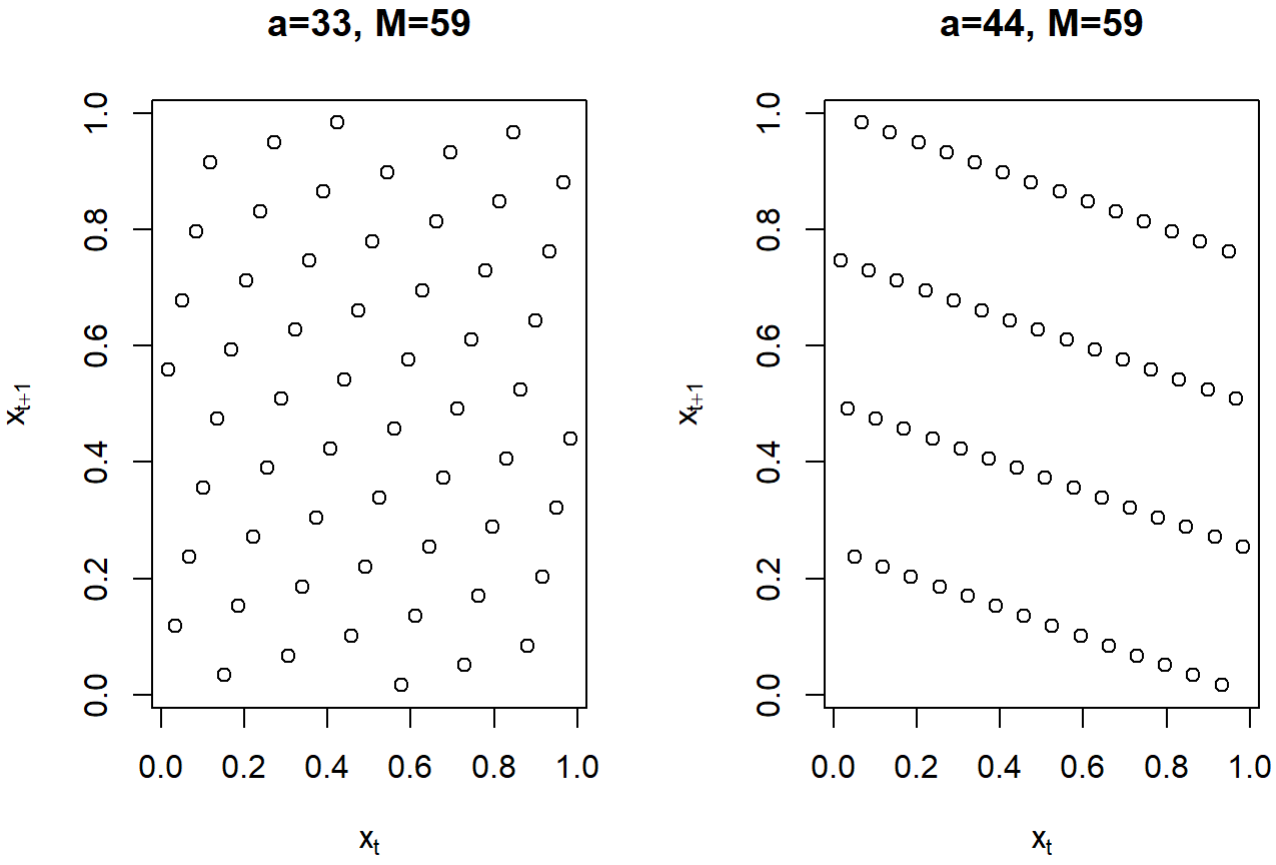
2023-05-13

Exercise 1

A Linear Congruential Generator (LCG) is given by

$$
\begin{aligned}
x_i &= a_0 + a_1 x_{i-1} \mod M \\
u_i &= x_i / M
\end{aligned}
$$

and when $a_0 = 0$ yields the Multiplicative Congruential Generator (MCG). @Marsaglia1968 showed that the consecutive tuples $(u_i, \dots, u_{i+k-1})$ from an MCG have a lattice structure. The figure below shows some examples for $k = 2$, using M small enough for us to see the structure



A lattice is an infinite set of the form

$$
\mathcal{L} = \left\{ \sum_{j=1}^{k} \alpha_j v_j | \alpha_j \in \mathbb{Z} \right\} ,
$$

where v_j are lineraly independent basis vectors in $\mathbb{R}^k$. The tuples from the MCG are the intersection of the infinite set $\mathcal{L}$ with the unit cube $[0, 1)^k$.

The lattice points in k dimensions lie within sets of parallel $k-1$ dimensional planes. Lattices where those planes are far apart miss much of the space, and so are poor approximations to the desired uniform distribution. For an MCG with period P, @Marsaglia1968 shows that there is always a system of $(k!P)^{1/k}$ or fewer parallel planes that contain all of the k-tuples.

- For each of the two MCG described above
  - Redo the figures above.
  - Find one basis (v_1, v_2) and report them in the figure.
  - Find the distance among the parallel lines.
- The lattice on the right of the figure above is not the worst one for $M = 59$. - Find another value of a for > which the period of $x_i = ax_{i-1} \mod 59$, starting with $x_1 = 1$ equals 59, but the 59 points u_i, u_{i+1} > for $u_i = x_i/59$ lie on parallel lines more widely separated than those with $a = 44$.
  - Plot the points and compute the separation between those lines;
  - Hint: identify a lattice point on one of the lines, and drop a perpendicular from it to a line defined by two > points on another of the lines.

First of all we implement the MC generator:

```
MCG <- function(n, seed=1, a=33, M=59) {
  # n is the number of values we want to generate, seed excluded.
  u <- rep(0, (n+1))
  u[1] <- seed
  for (i in 2:(n+1)) {
    u[i] <- (a * u[i-1]) %% M
  }
  u <- u[-1] / M
  return(u)
}
```

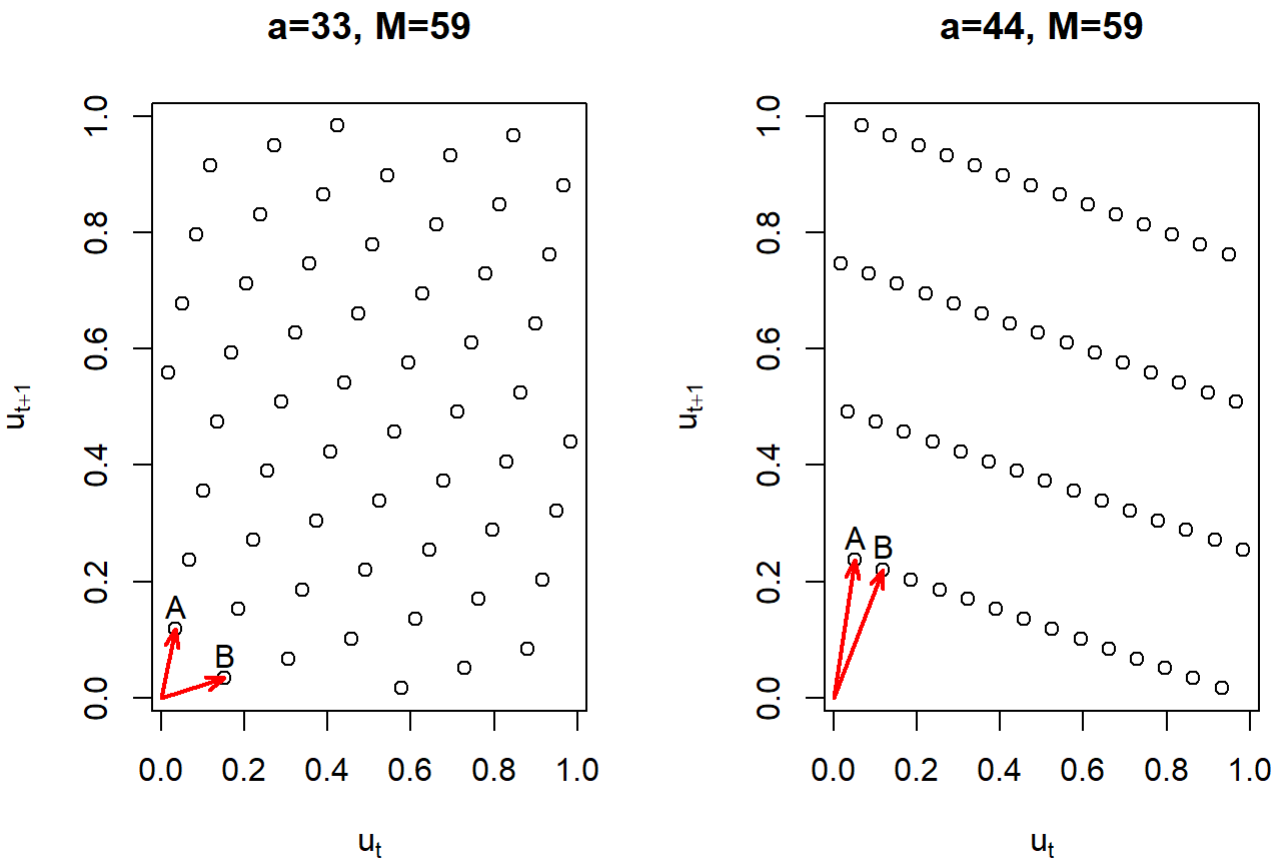
We want to compute a basis of the lattice.

Notice that the origin $O = (0, 0)$ belongs to the lattice. On the lattice, let A be the closest point to O and let B be the closest point to O among the points linearly independent with A. Then $\overrightarrow{OA}$ and $\overrightarrow{OB}$ give a basis.

We implement a function `get_basis` that takes a vector u that defines the lattice and returns A, B.

```
norm2 <- function(x) sqrt(sum(x^2))
dot <- function(v, w) return(sum(v * w))
are_LI <- function(v, w, eps=1e-6){
  cos_vw <- dot(v, w) / (norm2(v)*norm2(w))
  return(abs(abs(cos_vw)-1) > eps)
}

get_basis = function(u){
  coords <- cbind(u[-59], u[-1])
  dist <- apply(coords, MARGIN=1, FUN=norm2)
  ordered_indices <- order(dist)  # Returns the indices.
  i <- ordered_indices[1]; A <- u[i:(i+1)]
  for (j in ordered_indices[-1]){
    if (are_LI(A, u[j:(j+1)])){
      B <- u[j:(j+1)]; break
    }
  }
  return(list(A, B))
}
```

With it we can compute the bases for the two cases and plot them.



Now we have to compute the distance between parallel lines that contain the lattice.

Suppose we have two points P, Q on a line and a point M. If $\vec{n}$ is the vector orthogonal to PQ, normalized, then the distance between the point M and the line is the length of the projection of $\overrightarrow{PM}$ onto $\vec{n}$, which is $|\overrightarrow{PM} \circ \vec{n}|$.

We write a function `linePoint_distance` that does it:

```
linePoint_distance <- function(P, Q, M){
  # P and Q on the same line, M on the other line.
  PQ <- Q - P
  n <- c(PQ[2], -PQ[1]); n <- n / norm2(n)
  dist <- abs(dot(P - M, n))
  return(dist)
}
```

The points O, A, B give us three sets of parallel lines: one parallel to $\overrightarrow{OA}$, one to $\overrightarrow{OB}$ and one to $\overrightarrow{AB}$. For each set we can easily compute the distance between the lines using the previous function `linePoint_distance`. We have to pass to it two points which are on the same line and a third point which is on a close line. In particular we are interested in the biggest distance among the three.

For the case plotted on the left we have:

```
dist_par_AB <- linePoint_distance(A1, B1, O)
dist_par_OA <- linePoint_distance(O, A1, B1)
dist_par_OB <- linePoint_distance(O, B1, A1)

dist1 <- max(dist_par_AB, dist_par_OA, dist_par_OB)
dist1
```

```
## [1] 0.1373606
```

For the case plotted on the right we have:

```
dist_par_AB <- linePoint_distance(A2, B2, O)
dist_par_OA <- linePoint_distance(O, A2, B2)
dist_par_OB <- linePoint_distance(O, B2, A2)

dist2 <- max(dist_par_AB, dist_par_OA, dist_par_OB)
dist2
```

```
## [1] 0.2425356
```

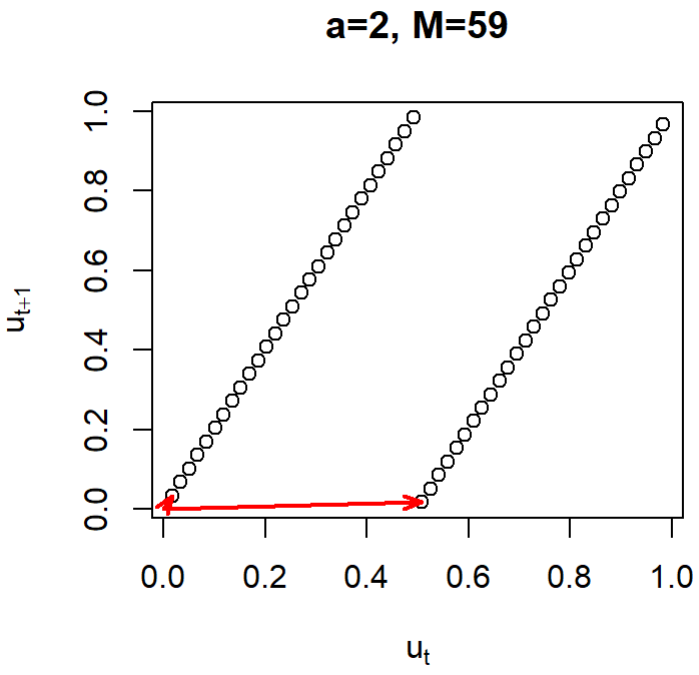
Now, among the values of a for which the generator has period 58, we want to find the “worst” one. That is the one for which the distance between parallel lines that contain the lattice is maximal.

To do it, for each $a \in \{2, \dots, 58\}$, we compute the period and the maximal distance among parallel lines. In `a_worst` we store the researched value of a.

```
a_worst <- 44; dist_worst <- dist2
for (a in 2:58){
  u <- MCG(n=59, a=a, M=59)
  period <- length(unique(trunc(u * 59)))  # 'trunc' ensures u * 59 contains integers
  if (period == 58){
    basis <- get_basis(u)
    A <- basis[[1]]; B <- basis[[2]]

    dist_par_AB <- linePoint_distance(A, B, O)
    dist_par_OA <- linePoint_distance(O, A, B)
    dist_par_OB <- linePoint_distance(O, B, A)

    dist_a <- max(dist_par_AB, dist_par_OA, dist_par_OB)
    if (dist_a > dist_worst){
      a_worst <- a; dist_worst <- dist_a
    }
  }
}

a_worst
```

```
## [1] 2
```

We obtain a = 2. Let's plot the corresponding lattice and compute the separation between parallel lines.



The distance between the lines is: 0.4472136

Exercise 2

Suppose that we are using an MCG with period $P \leq 2^{32}$.

- Evaluate Marsaglia's upper bound on the number of planes which will contain all consecutive $k = 10$ tuples from > the MCG.
- Repeat the previous part, but assume now a much larger bound $P \leq 2^{64}$.
- Repeat the previous two parts for $k = 20$ and again for $k = 100$.
- Hint: Read carefully the previous exercise!

We define a function `Marsiglia_ub` that computes Marsiglia's upper bound for the number of planes:

```
Marsiglia_ub <- function(k, P){
  return ((factorial(k) * P) ^ (1/k))
  }
```

Now we compute the Marsiglia upper bound for each combination of $k \in \{10, 20, 100\}$ and $P \in \{2^{32}, 2^{64}\}$.

We show the results in a matrix where each row correspond to a value of k and each column to a value of P.

```
k = c(10, 20, 100)
P = c(2^32, 2^64)
Mars_ub = outer(k, P, FUN=Marsiglia_ub)
rownames(Mars_ub) <- k; colnames(Mars_ub) <- c('2^32', '2^64')
Mars_ub
```

```
##         2^32      2^64
## 10  41.61715 382.44437
## 20  25.17412  76.31365
## 100 47.42743  59.20512
```

Exercise 3

Suppose that an MCG becomes available with period $2^{19937} - 1$.

What is Marsaglia's upper bound on the number of planes in $[0, 1)^{10}$ that will contain all 10-tuples from such a generator?

Now we have to compute Marsaglia's upper bound for $P = 2^{19937} - 1$ and $k = 10$.

We cannot compute it directly because the value of P exceeds the range of representable values for the class integer in R.

We can overcome this problem with simple algebraic manipulations and a small approximation. Indeed, if we call $M = (k!P)^{\frac{1}{k}}$ the Marsaglia's upper bound, then:

$$\log_2(M) = \frac{\log_2(k!) + \log_2(P)}{k}$$

In our case $\log_2(P) = \log_2(2^{19937} - 1) \approx 19937$, so we are able to approximate $\log_2(M)$ with R:

```
k = 10; log2_P <- 19937
log2_M <- (log2(factorial(k))+log2_P) / k
log2_M
```

```
## [1] 1995.879
```

That is $M \approx 2^{1995.879}$

Exercise 4

A relatively new and quite different generator type is the Inversive Congruential Generator, ICG. For a prime number M the ICG update is

$$x_i = \begin{cases} a_0 + a_1 x_{i-1}^{-1} \mod M & x_{i-1} \neq 0\,, \\ a_0 & x_{i-1} = 0\,. \end{cases}$$

When $x \neq 0$, then x^{-1} is the unique number in $\{0, 1, \ldots, M-1\}$ with $xx^{-1} = 1 \mod M$. The ICG behaves as if it uses the convention $0^{-1} = 0$. These methods produce a sequence of integer values $x_i \in \{0, 1, \ldots, M-1\}$, that is, integers modulo M. With good choices for the constants a_j and M, the x_i can simulate independent random integers modulo M. See also the wikipedia page en.wikipedia.org/wiki/Inversive_congruential_generator (https://en.wikipedia.org/wiki/Inversive_congruential_generator).

Consider the ICG for $M = 59$, $a_0 = 1$ and $a_1 = 17$.

- What is the period of this generator?
- Plot the consecutive pairs (u_i, u_{i+1}) where $u_i = x_i/59$.

In order to implement the IC generator we have to compute the inverse x^{-1} of a given $x \in \{1, \ldots, M-1\}$ in $\mathbb{Z}/M\mathbb{Z}$. One way to do it could be by checking every value in $\{1, \ldots, M-1\}$. Otherwise there is a more efficient algorithm based on the extended Euclidean algorithm.

Actually, if M is prime, we can compute the inverse with a single operation (in theory). Indeed, if M is prime, for Fermat's little Theorem, $x^{M-1} \equiv 1 \mod M$ for every $x \in \{1, \ldots, M-1\}$, that is $x^{-1} = x^{M-2}$.

The problem with this last approach is that x^{M-2} could be a very big number and so we would need a function that computes powers in modular arithmetic efficiently. Instead of importing libraries I prefer to use the first method, which works fine for values of M that are not to big.

```
ICG <- function(n, seed=1, a0=1, a1=17, M=59) {
  # n is the number of values we want to generate, seed excluded.
  u <- rep(0, (n+1))
  u[1] <- seed
  for (i in 2:(n+1)) {
    if (u[i-1]==0) u[i] <- a0
    else{
      for (inverse in 1:(M-1)){
        if (((u[i-1] * inverse) %% M) == 1) break
      }
      u[i] <- (a0 + a1 * inverse) %% M
    }
  }
  u <- u[-1] / M
  return(u)
}
```

Now we consider the ICG with $M = 59, a_0 = 1, a_1 = 17$.

Its period is:

```
u <- ICG(n=59, seed=1, a0=1, a1=17, M=59)
period <- length(unique(trunc(u * 59))) # 'trunc' ensures u * 59 contains integers
period
```

```
## [1] 59
```

From the plot of the consecutive pairs (u_i, u_{i+1}), where $u_i = x_i/59$, we can see that the distribution of the points is “more random” with respect to the two generators we analysed in Exercise 1.