Exercise 1

Random Number Generation

Olesia Galynskaia 12321492

2025

Fix the random seed for reproducibility

```
set.seed(12321492)
```

Task 1: Uniform random numbers, summaries & plots

Here we define a custom linear congruential generator (LCG) in the form

$$x_i = (a + b \cdot x_{i-1}) \bmod m.$$

The function creates a numeric sequence scaled to the interval $[0, 1)$ that we will later analyze graphically.

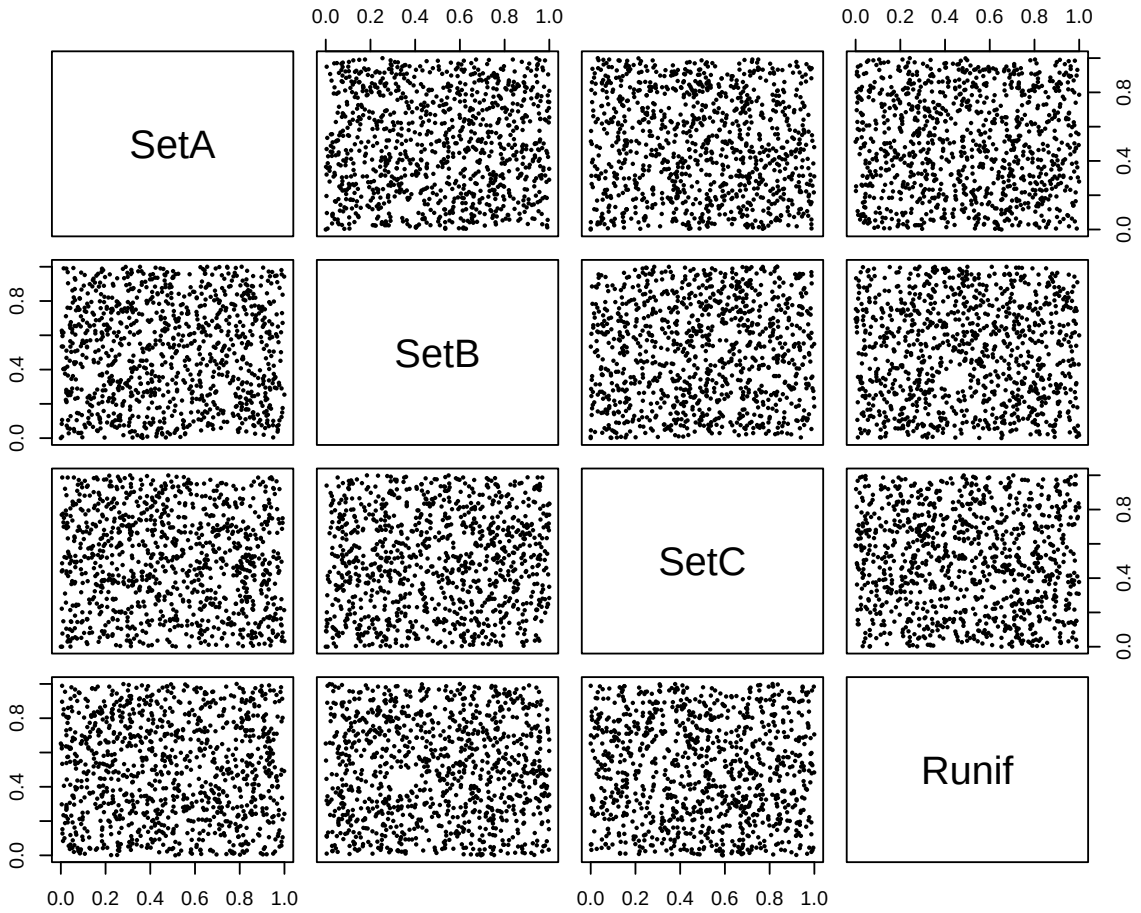
```
congruential_random <- function(n, a, b, m, x0) {
  x <- numeric(n)
  x[1L] <- x0 %% m
  if (n >= 2L) {
    for (i in 2L:n) {
      x[i] <- (a + b * x[i - 1L]) %% m
    }
  }
  x / m
}
```

In this step we generate four random sequences:
- **Set A** — the parameters required by the assignment,
- **Set B** — a weak set with a small modulus,
- **Set C** — a stronger set with a large prime-like modulus,
- **Set D** — R's built-in `runif()` sequence.

Then we combine them and create pairwise scatter plots to visually compare the generators.
Here we create lag plots (u_t, u_{t+1}) for all four sequences to reveal possible lattice structures or dependencies between consecutive values.
A uniform scatter without visible stripes indicates good randomness.

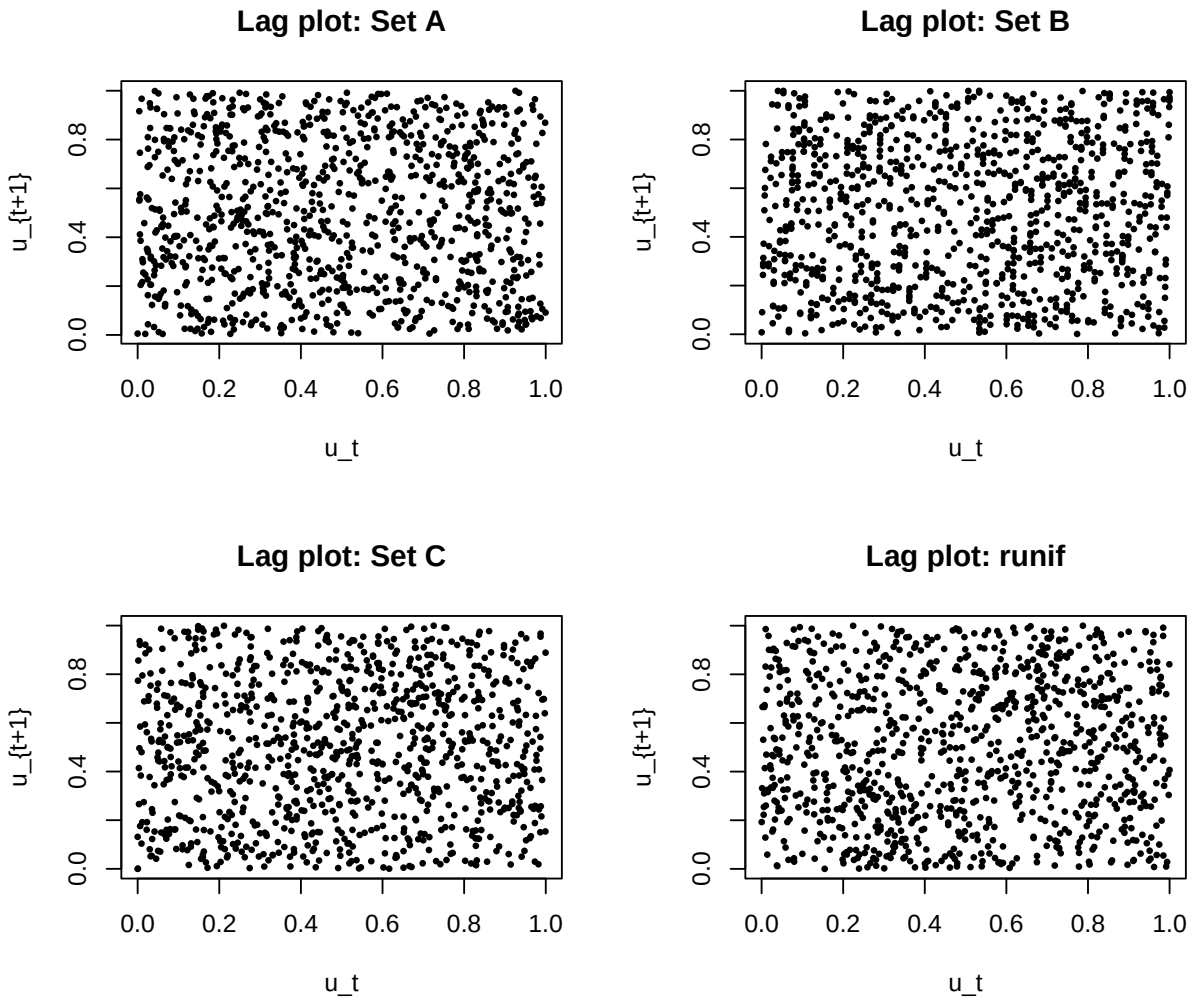
```
n <- 1000

u_A <- congruential_random(n = n, a = 23606797, b = 69069, m = 2^32, x0 = 1)
```

```r
u_B <- congruential_random(n = n, a = 0, b = 75, m = 2^16 + 1, x0 = 7)

u_C <- congruential_random(n = n, a = 0, b = 16807, m = 2^31 - 1, x0 = 1)

u_R <- runif(n)

mat <- cbind(SetA = u_A, SetB = u_B, SetC = u_C, Runif = u_R)
pairs(mat, main = "Pairwise scatter of 4 sequences (3 LCGs + runif)",
      pch = 16, cex = 0.5)
```




```r
par(mfrow = c(2, 2))
plot(u_A[-n], u_A[-1], pch = 16, cex = 0.6,
     main = "Lag plot: Set A", xlab = "u_t", ylab = "u_{t+1}")
plot(u_B[-n], u_B[-1], pch = 16, cex = 0.6,
     main = "Lag plot: Set B", xlab = "u_t", ylab = "u_{t+1}")
plot(u_C[-n], u_C[-1], pch = 16, cex = 0.6,
     main = "Lag plot: Set C", xlab = "u_t", ylab = "u_{t+1}")
plot(u_R[-n], u_R[-1], pch = 16, cex = 0.6,
```

```
     main = "Lag plot: runif", xlab = "u_t", ylab = "u_{t+1}")
```




```
par(mfrow = c(1, 1))
```

Observations

Each lag plot shows pairs of consecutive values (u_t, u_{t+1}).
For a *good* random number generator, the points should fill the square [0,1]x[0,1] uniformly, without visible lines or grids.
In other words, there should be **no linear dependence** between successive numbers.

Set A (b = 69069, a = 23606797, m = 2^{32})
- These are the parameters given in the task.
- The plot looks fairly uniform — no strong diagonal or vertical patterns.
- This generator has a *long but not full* period: since $m = 2^{32}$ is not prime and a, b do not meet all full-period conditions, the sequence can eventually repeat after fewer than m steps.
- Still, for small n (like 1000) the results are quite acceptable and visually close to `runif()`.

Set B (b = 75, a = 0, m = $2^{16} + 1$)
- Here m is small, so the possible states of x_i are limited.
- A good LCG should satisfy:
1. a and m are coprime (gcd(a,m)=1),
2. $b - 1$ divisible by all prime factors of m,
3. if m is multiple of 4, then $b - 1$ is also multiple of 4.
- These rules (the *full-period conditions*) ensure the generator uses every value before repeating.
- This set violates them, giving a **short period** and clear lattice structure — thin stripes across the plot.
- You can see these small diagonal clusters in the lag plot, showing strong dependence between consecutive values.

Set C (b = 16807, a = 0, m = $2^{31} - 1$)
- This is the famous *Park-Miller minimal standard* LCG.
- The modulus $m = 2^{31} - 1$ is prime, and b is chosen carefully so the generator achieves the **maximum possible period (m-1)**.
- Because the parameters meet all full-period criteria, the sequence covers the space evenly and shows no visible structure.
- The lag plot looks very similar to the one from `runif()`.

`runif()` (R's built-in generator)
- Uses a modern combined generator that mixes multiple sequences and passes statistical randomness tests.
- The lag plot is completely uniform, showing no dependencies at all.
- This is the *reference standard* for good pseudorandom behavior.

Summary:
- The visual differences reflect how the *mathematical conditions* on a, b, m control the quality of the generator.
- Poor choices (like Set B) break the full-period rules → visible stripes, shorter cycles.
- Good choices (like Set C) satisfy them → full period, uniform scatter.
- Built-in RNGs like `runif()` go beyond simple LCGs to eliminate any structure entirely.
- Building your own LCG helps you understand *why* these theoretical rules matter and how they show up in practice. - Even though lag plots are a simple visual test, they only reveal linear structure between pairs of values.
More advanced randomness tests (such as autocorrelation checks, spectral tests, and chi-square uniformity tests) could detect hidden dependencies that are not visible here.
For this exercise, the lag plots are sufficient to illustrate the main idea — how different parameter choices in an LCG affect the apparent randomness of the generated numbers.

Task 2: Uniform random numbers, summaries & plots

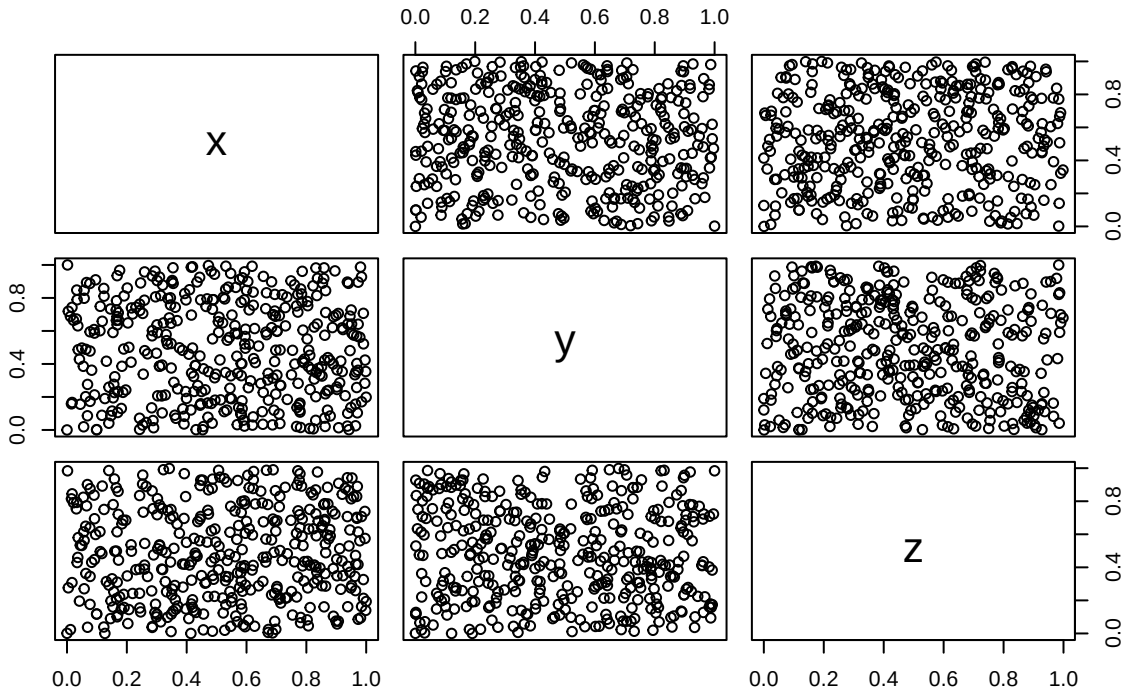
Explore the R data set `randu` graphically

In this part we load the historical `randu` dataset from the **datasets** package, convert it to a numeric matrix, and explore its structure using pairwise scatter plots.
These plots help us detect patterns that indicate correlations between coordinates.

Next we draw a simple three-dimensional static scatter plot of the RANDU points using the **scatterplot3d** package.
This gives a quick visual impression of how the points are arranged in 3-D space.

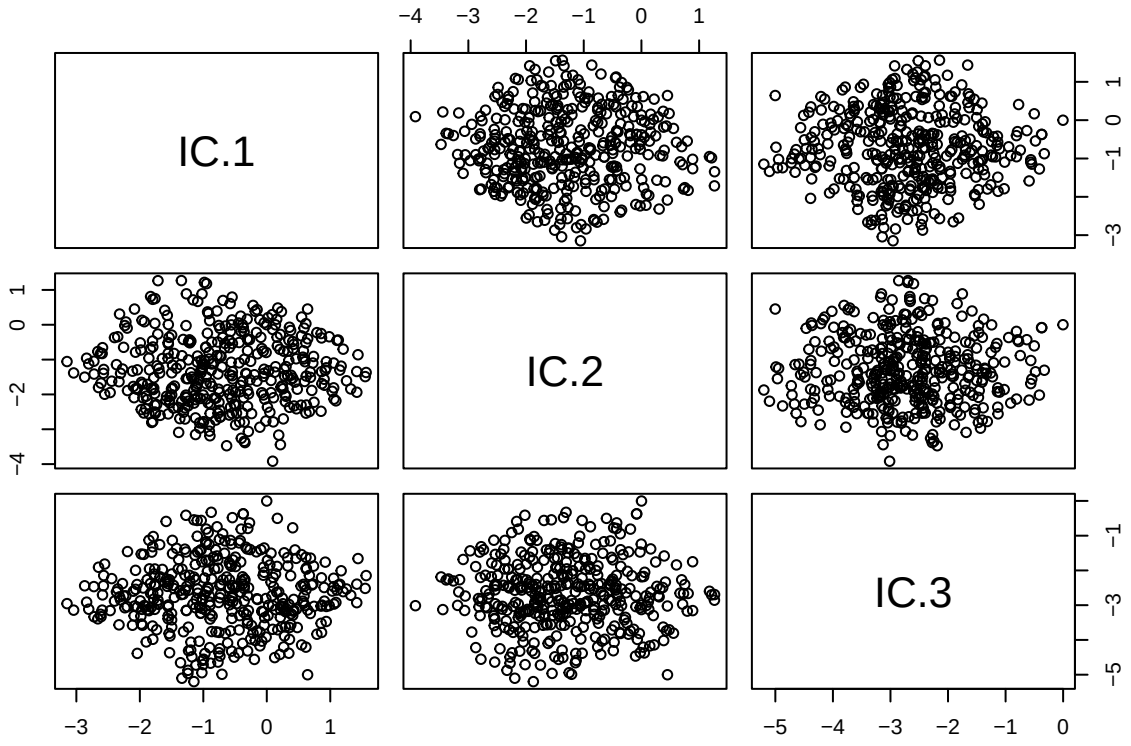
We apply **ICS (Invariant Coordinate Selection)** to the RANDU data to reveal hidden linear structures. The custom function `Vrandu()` defines a scatter-type covariance transformation that uses pairwise differences. We run ICS with two power parameters (`pow = 1` and `pow = 4`) to see how increasing the weighting factor emphasizes non-random alignments in the data.

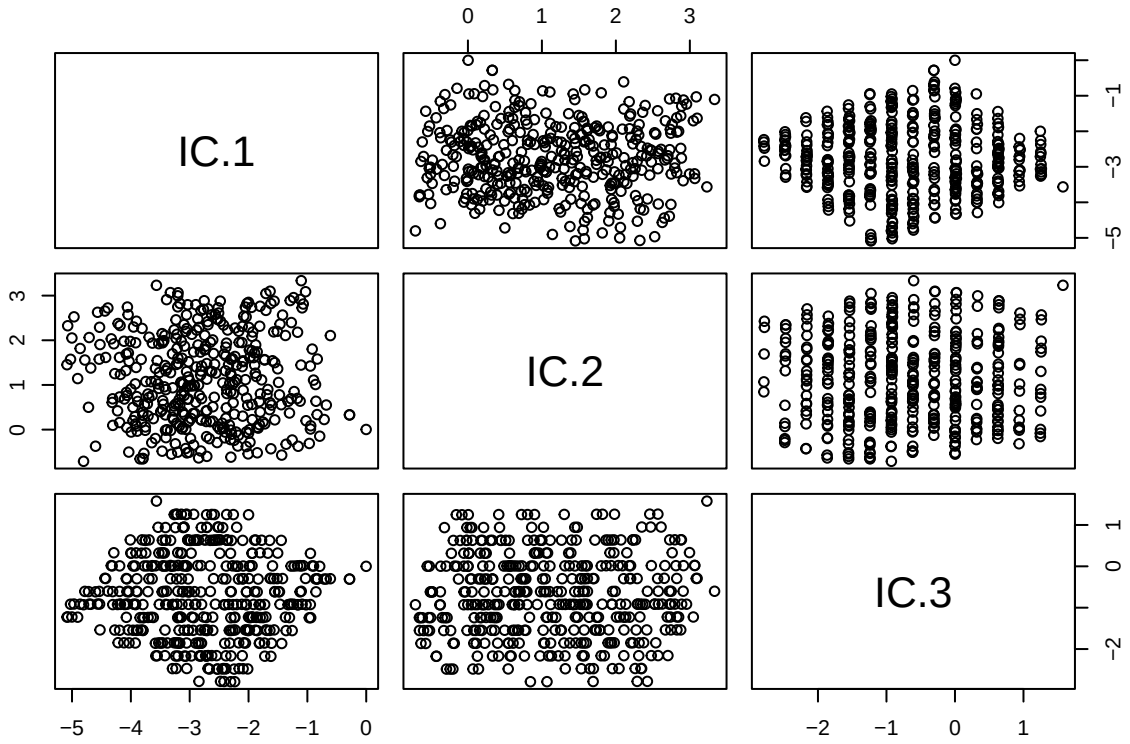
```r
data("randu", package = "datasets")

X <- as.matrix(randu)
stopifnot(ncol(X) == 3)

pairs(X, main = "R dataset `randu`: pairwise scatter")
```




```r
if (requireNamespace("scatterplot3d", quietly = TRUE)) {
  scatterplot3d::scatterplot3d(X, pch = 16, main = "randu (static 3D view)")
}

ok_ics   <- requireNamespace("ICS",   quietly = TRUE)
ok_icsnp <- requireNamespace("ICSNP", quietly = TRUE)

if (ok_ics && ok_icsnp) {
  library(ICS)
  library(ICSNP)
  Vrandu <- function(x, pow = 2) {
    Sn     <- cov(x)
    xdiffs <- pair.diff(x)                                  # all pairwise differences
    weights <- (mahalanobis(xdiffs, center = FALSE, cov = Sn))^(pow/4)
    wxdiffs <- xdiffs / weights
    V <- 2 * crossprod(wxdiffs) / nrow(xdiffs)
    V
  }
```

```
  plot(ics(as.matrix(randu), cov, Vrandu, S2args = list(pow = 1)))

  plot(ics(as.matrix(randu), cov, Vrandu, S2args = list(pow = 4)))
} else {
  message("Packages ICS and/or ICSNP are not available. Skipping ICS plots.")
}
```





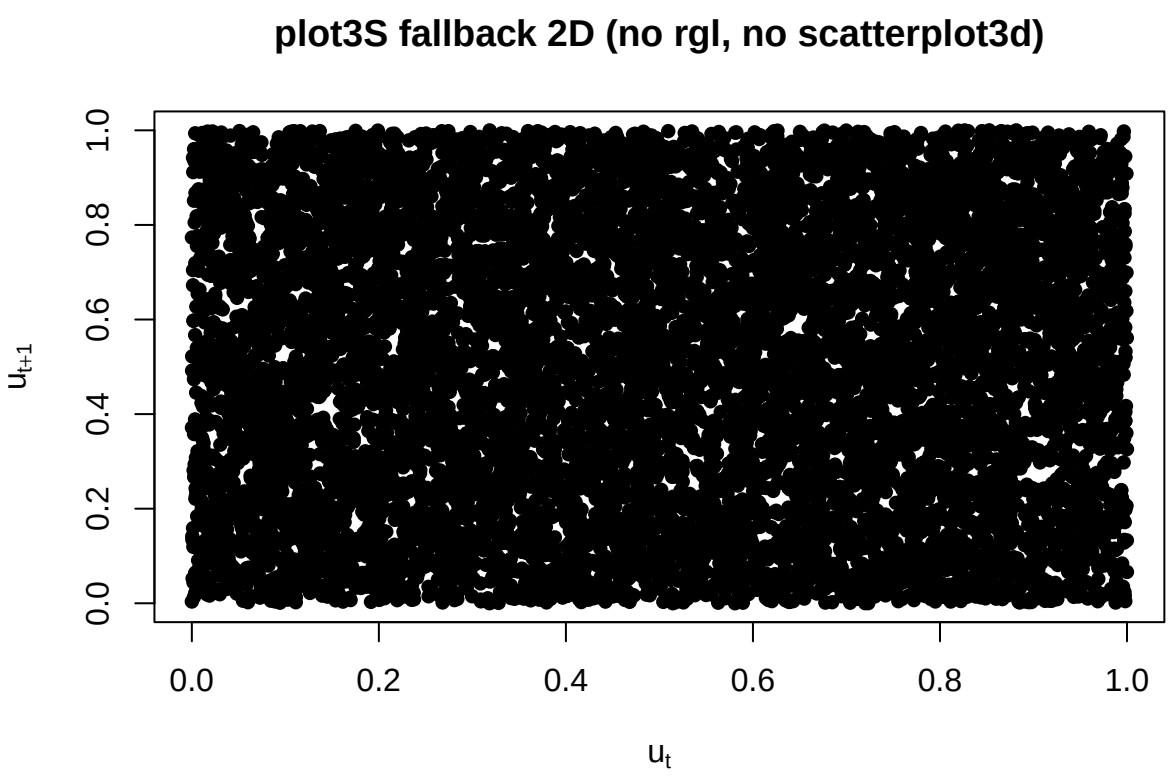
Define plot3S(), generate 10,000 values (LCG vs runif), plot with rgl

Now we define the helper function `plot3S()` that plots successive triples of values (u_t, u_{t+1}, u_{t+2}) in three dimensions.
If **rgl** is available, it creates an interactive 3-D scatter plot; otherwise it falls back to a static 2-D or pseudo-3-D version.

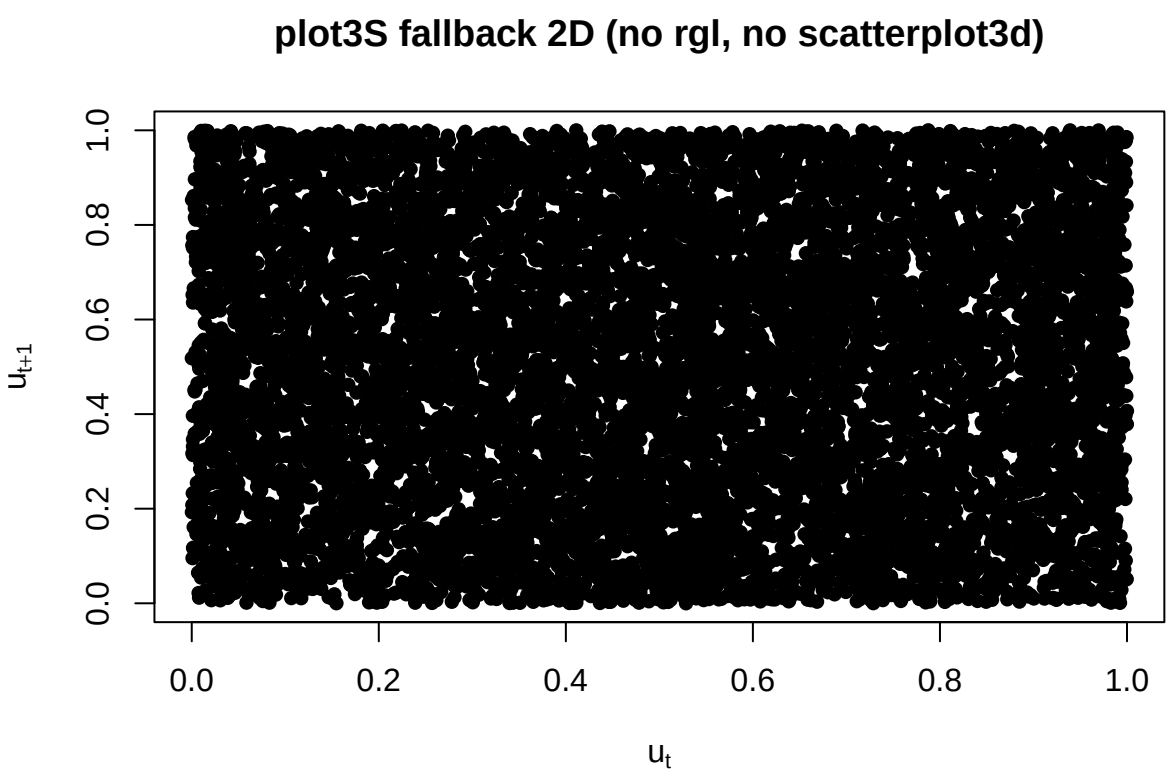
We generate two samples of length 10 000: - one using our LCG with parameters $a = 0, b = 65539, m = 2^{31}, x_0 = 123$, - and one using R's built-in `runif()` generator.
We then visualize both sequences with `plot3S()` to compare their spatial structure and randomness.

```
ok_rgl <- requireNamespace("rgl", quietly = TRUE)
if (ok_rgl) library(rgl)

plot3S <- function(rsample) {
  n <- length(rsample)
  if (n < 3) stop("Need at least 3 values.")
  x1 <- rsample[1:(n-2)]
  x2 <- rsample[2:(n-1)]
  x3 <- rsample[3:n]
  if (ok_rgl) {
    open3d()
    plot3d(x1, x2, x3, size = 3)
    play3d(spin3d())
  } else {
```

```
    if (requireNamespace("scatterplot3d", quietly = TRUE)) {
      scatterplot3d::scatterplot3d(x1, x2, x3, pch = 16, main = "plot3S fallback (no rgl)")
    } else {
      plot(x1, x2, pch = 16, main = "plot3S fallback 2D (no rgl, no scatterplot3d)",
           xlab = expression(u[t]), ylab = expression(u[t+1]))
    }
  }
}

congruential_random <- function(n, a, b, m, x0) {
  x <- numeric(n)
  x[1L] <- x0 %% m
  if (n >= 2L) {
    for (i in 2L:n) {
      x[i] <- (a + b * x[i - 1L]) %% m
    }
  }
  x / m
}

set.seed(12321492)

n <- 10000
u_lcg   <- congruential_random(n = n, a = 0, b = 65539, m = 2^31, x0 = 123)
u_runif <- runif(n)

plot3S(u_lcg)
```




```
plot3S(u_runif)
```



Observations

In this task we analyzed how different random number generators behave in higher dimensions.
We compared three sources of random values:

1. The historical `randu` dataset (an early built-in LCG in FORTRAN).
2. Our own linear congruential generator (LCG) with parameters $a = 0$, $b = 65539$, $m = 2^{31}$, $x_0 = 123$.
3. The modern built-in generator in R used by the `runif()` function.

We used several tools: pairwise scatter plots, ICS/ICSNP invariant coordinate analysis, and the 3D function `plot3S()` that visualizes triples (u_t, u_{t+1}, u_{t+2}).

RANDU dataset

The pairwise scatter plots of the `randu` data show **clear diagonal stripes and parallel planes**.
Instead of filling the square [0,1]×[0,1] uniformly, the points concentrate along linear directions.
This reveals that the values are **not independent** but strongly correlated.

The ICS/ICSNP method helps to highlight this dependency.
It transforms the data into invariant coordinates (directions where non-random structure appears).
With `pow = 1`, the plots already show elongated clusters; with `pow = 4`, the same clusters become even sharper and more aligned.

This means that the structure in `randu` is systematic and increases with the weighting power, proving that its "random" values lie on a few planes.

The reason is purely mathematical:
RANDU is generated by

$$x_{i+1} = (65539\, x_i) \bmod 2^{31}.$$

Here, $m = 2^{31}$ (a power of 2) and $b = 65539 = 2^{16} + 3$.
These parameters **violate the full-period conditions** of LCGs, which require that: 1. a and m are coprime;
2. $b - 1$ is divisible by all prime factors of m;
3. if m is a multiple of 4, then $b - 1$ must also be a multiple of 4.
Because these rules are not met, RANDU cycles through a small subset of possible values and creates visible linear patterns.

Our LCG ($a = 0, b = 65539, m = 2^{31}$)

The `plot3S(u_lcg)` visualization shows the same behavior as RANDU:
the points form **diagonal planes and grid-like stripes**, instead of filling the cube evenly.
This happens because our generator uses almost identical parameters, so it inherits the same structural defects.
Although the sequence might look uniform in one dimension (a histogram of values would look flat), it clearly fails in two or three dimensions, where dependence between consecutive values becomes visible.

This demonstrates an important principle:
> A generator can look "random" in one dimension but still fail completely in higher-dimensional tests.

R's built-in `runif()`

The 3D plot `plot3S(u_runif)` shows a completely **uniform cloud of points** with no visible planes, stripes, or gaps.
The numbers fill the cube evenly in all directions.
R's internal RNG is based on a **modern combined multiple recursive generator (MRG32k3a)**.
This algorithm mixes several sequences with large, coprime moduli, ensuring long periods, good independence, and that all full-period conditions are satisfied.
It passes standard statistical tests of randomness, which explains the smooth, structure-free result.

Overall conclusion

- Both **RANDU** and our **LCG ($\mathbf{a = 0, b = 65539, m = 2^{31}}$)** fail to produce independent pseudo-random numbers.
  Their output forms **visible planes and linear bands**, a clear signature of poor parameter choice and short period length.
- The **ICS/ICSNP** analysis confirms this: as the power parameter increases, the dependency structure becomes more obvious.
  The invariant coordinate plots reveal preferred linear directions — proof that the numbers are not uniformly distributed.
- The `runif()` generator, in contrast, shows no such structure.
  Its points are evenly distributed and independent, representing high-quality pseudorandom behavior.

- These results illustrate a key theoretical fact:
  The statistical quality of an LCG depends entirely on the parameters a, b, m.
  Violating the full-period conditions leads to correlated and predictable sequences.
  Satisfying them (or using improved generators like `runif()`) yields uniform, independent results suitable for simulation and statistical computation.

In summary, this experiment clearly demonstrates how a poor choice of parameters can make an LCG appear random in one dimension but fail badly in multidimensional space — and why modern random number generators are designed to avoid exactly these problems.