Műszaki Tudományos Közlemények vol. 18. (2023) 37–44.
DOI: Hungarian: https://doi.org/10.33895/mtk-2023.18.07
English: https://doi.org/10.33894/mtk-2023.18.07

RANDOM NUMBER GENERATOR

Katalin HARANGUS,[1] András KAKUCS[2]

[1] *Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences, Târgu-Mureș, Romania, katalin@ms.sapientia.ro*

[2] *Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences, Târgu-Mureș, Romania, kakucs2@ms.sapientia.ro*

Abstract

Illustration plays an important role during education: The Galton board is a suitable tool for illustrating random processes and explaining probability distributions. We have created this tool in a virtual version, which facilitates data collection for statistical processing of experimental data and also enables the study of non-symmetrical distributions. The random processes on the device are simulated, which requires a random number generator. Since there were some doubts about the software-generated pseudo-random numbers, we created a true random number generator based on the input noise of microcontrollers.

Keywords*: random number generator, Galton board.*

1. Introduction

During the education of university students, we would like to provide them with fundamental knowledge from the field of probability theory and statistics. At least as much as clarifies the nature of random phenomena and introduces practical, numerical relationships that describe them. Since this field belongs to mathematics, its thorough understanding and in-depth study require a mathematical way of thinking. Hence comes the biggest challenge in accomplishing this task: the engineering way of thinking is not the same as that of a mathematician. While the latter usually does not tie concepts to the real world, the engineer practically lives and breathes it. Experience shows that even at a basic level, the theoretical math knowledge of engineering students is incomplete, and they do not really see the need to fill this gap. The situation is even worse for students in non-engineering majors, where there is no math education at all. Therefore, in teaching subject areas that require abstract thinking, the use of illustrative tools that connect abstract concepts to reality plays a crucial role **[1, 2]**. For example, demonstrating the operation of various algorithms using software and hardware tools. Based on this, the idea arose that in clarifying the nature of random processes, which is followed by establishing quantitative relationships, we could "rely" on the Galton board.

So a virtual Galton board was developed. The original version of the Galton board is a tool used to illustrate normal distribution. The virtual version simulates the occurring natural phenomenon, and the user can intervene in it, allowing for different distributions to be visualized. The simulation is done using computer tools, which not only makes the phenomenon observable but also allows for data to be collected "on the go", which can later be processed and analyzed by computer program.

The virtual Galton board simulates random phenomena, so a random number generator is needed. Previous experience has shown that working with pseudo-random numbers generated by algorithms doesn't always lead to the expected results. For example, even when a few hundred consecutive random numbers are generated, they don't always distribute uniformly. To overcome this problem, the aim was to produce true random numbers using hardware, which didn't incur additional costs in building the virtual Galton board, as it was achieved through a microcontroller that controlled it. This article describes the implementation of this idea.

The didactic tool itself was presented and published in our presentation "Didactic Tool for Intuitive Random Process Visualization" at the "25th International Conference on Interactive Collaborative Learning" held in Vienna in late September 2022. The second part of this article provides a brief summary of what was discussed at the conference.

2. Generating random numbers

Generating random numbers is usually simple in most programming environments because there is typically a function [3], specifically designed for this purpose, such as:

- in C (and therefore when programming with Arduino) and C++, the *rand()* function returns an integer between 0 and *RAND_MAX*, where the upper limit depends on the programming environment but is guaranteed to be at least 32767;
- in Visual Basic the *Rnd()*, function represents a random number from the [0, 1) interval;
- in Excel the *RAND()* function returns a random number from the [0, 1) interval, or the *RANDBETWEEN(bottom, top)* function returns a random integer in the [*bottom*, *top*] interval.

These functions use some algorithm to determine the next value of the nominally uniformly distributed random number, which is why they are computed rather than truly random. They are called pseudorandom numbers. In practice, consecutive numbers appear random, but they always follow the same sequence. To reduce the likelihood of repetition, there is usually an option to set the starting point of the sequence. In C, there is the *randomSeed(x)* function, while in Visual Basic, there is a *Randomize(x)* statement, both of which set the beginning of the sequence to the number given as the parameter *x*. Of course, if the same parameter is used every time, the sequence will always start in the same place, so starting the sequence from a location determined by the computer's clock, for example, can make it more random (the parameter could be the number of seconds elapsed since midnight, for instance).

If we need true random numbers, we can generate them using some physical device [3]. This can be, for example, dice or a hardware random number generator. Their operation is not based on an algorithm, but on some physically random process. The RPG100 integrated circuit available on the market generates 16-bit random numbers based on semiconductor noise, but this circuit needs to be connected to a computer.

Looking for a simpler and cheaper solution, we built our own version based on the noise of the analog input of microcontrollers.

Although it is slower than the circuit mentioned as an example, theoretically we can generate random numbers with any resolution using it. The implementation is based on the easily programmable and computer-connectable microcontroller Arduino and ESP development boards available to anyone: we tested the Arduino Mega [4] and the ESP32-es [5] versions. The main difference between the two is the speed, with the ESP32 being significantly faster.

The operating principle is based on sampling the randomly fluctuating voltage of the microcontroller's "floating", unconnected inputs. To clarify the input state, it is usually connected to the ground with a larger resistor, or in the case of digital inputs, to the positive supply voltage. These resistors ("pull down", "pull up") are sometimes built-in elements of the microcontroller, which can be enabled through software, otherwise we have to connect them to the circuit ourselves. To make the input state random, we don't need to connect anything to it, and we have to bypass the internal pull down/pull up resistor (if there is one).

If we sample a floating digital input, its state will randomly alternate between LOW and HIGH. If we sample an analog input, the sampled voltage value will randomly fluctuate within the range of the smallest and largest possible values.

In both cases, the obtained signal depends on the noise voltage of the corresponding input. There are several explanations for what the source of this noise could be, probably due to the combined effects of these sources. One source is the thermal noise of the semiconductors (and conductors and dielectrics), which is also the basis of the RPG100 integrated circuit mentioned above. Another important component is "atmospheric noise", which can be heard as radio static from an untuned station. The latter has several components, stemming from natural phenomena and human activity. Some of the components from human activity can show regular repetition, such as the 50 Hz noise of the electrical network, so it does not develop completely randomly.

The third source of input noise in microcontrollers is, according to our experience, the microcontroller itself and the surrounding circuit. Thus, the 32nd input of the Wemos Lolin32 clone we tested (which is a development board built on an ESP32) is particularly noisy, much noisier than

the other inputs, which is probably due to some sort of design flaw.

The Arduino samples the voltage of the analog input with 10-bit resolution, while the ESP32 with 12-bit resolution, which can be read as an integer using the *analogRead()* function.

If nothing is connected to an analog input, the length of the wires that act as an antenna for environmental, atmospheric noise will be minimal. In this case, the sampled voltage changes chaotically, depending on the noise of the development board components, so there is little regularity in it. According to experience, the noise voltage changes too slowly, so there is no significant difference between the magnitude of two consecutive signals. If we used this sampled voltage to generate a random number, there would not be a significant difference between consecutive numbers.

To generate true random numbers with our random number generator, we utilize a flaw in the sampling circuit (the analog-to-digital converter) where the last one or two bits of the number representing the signal magnitude obtained by sampling even in the case of a stable voltage measurement, are fluctuating. This helps to eliminate any regularly changing components of the noise.

The operating principle can be algorithmically described as follows:

– we sample the floating analog input;
– we determine the last one or two bits of the obtained digital signal;
– we add these bits to a sequence of bits;
– when this sequence is long enough, we interpret it as a random number (e.g. with 16 bits, we get a random number between 0 and 65535).

In our program written in C and applied to Arduino and ESP32, with keeping only the last bit, it looks like this **[6]**:

```
B = byte(analogRead(Pin0) &
0b00000001) |
(byte(analogRead(Pin1) &
0b00000001) << 1) |
...
(byte(analogRead(Pin7) &
0b00000001) << 7);
```

where *analogRead(PinX)* is the value read from the analog input, *0b00000001* is a mask (with only the last bit being 1). The & operation is used to apply this mask (bitwise "and"), which results in only keeping the last bit of the read value (the rest is zeroed out). This bit is shifted left by << *n* (where n determines where the kept bit should go), and then the result is converted to a one-byte number (*byte(...)* – this is necessary because the value returned by the *analogRead() function is not a one-byte data)* –, then with the | operation (bitwise "or"), it is inserted into the already created sequence. The result will be one byte. If a two-byte number is needed, it can be obtained from two bytes with the $B1 + 256 \times B2$ operation.

To speed up the generation of random numbers, we sample eight different analog inputs simultaneously to create the eight random bits.

If we only need a single random "yes/no" value (such as in building a virtual Galton board), then the process is even simpler: we just need to observe whether the integer value read from the analog input is even or odd, and the generated random number is the remainder of dividing this value by two, either 0 or 1.

3. Testing the hardware random number generator

During testing, the program generates two one-byte data in each cycle, from which a random integer between 0 and 65,535 is calculated. Experience shows that after sampling, a little time needs to elapse for the circuits to return to their initial state, otherwise the obtained quantities do not behave randomly. If we were to solve the random number generation with a single input sampling, the process would be very slow due to the waiting time between determining two consecutive bits. Therefore, we sample eight different inputs in sequence without waiting, hoping that they do not affect each other. We briefly interrupt the program execution between the determination of the bytes (8-8 bits). From these numbers, we generated sequences, patterns of 16,384 elements. By sorting these patterns in increasing order, we obtained the graphs shown in **Figure 1.**

We found that the length of the waiting time between consecutive samplings affects the quality of the acquired dataset. Without waiting, the datasets contain a large number of zero values. A waiting time of 2 ms already improves the situation, but both the ESP32 and the Arduino Mega show strongly nonlinear behavior.

A 10 ms wait already significantly improves the situation and seems to be sufficient for the Arduino. The clock frequency of the Arduino is an order of magnitude lower than that of the ESP, so the delay due to the longer cycle time is also added to the 10 ms.

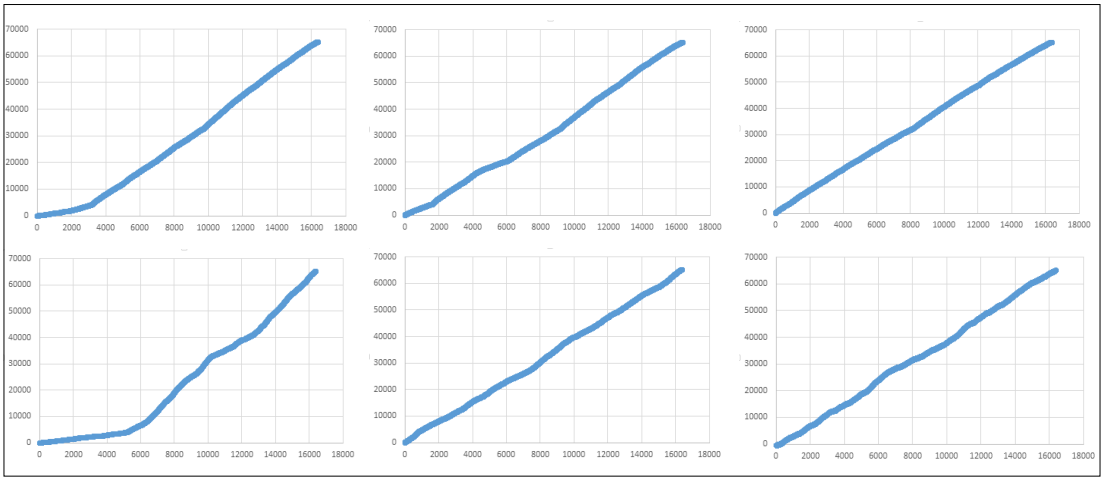

Fig. 1. *N Increasingly ordered samples.*
Top: ESP32, Bottom: Arduino Mega. From left to right: 2ms, 10ms, and 25ms delay between two consecutive samples. The horizontal axis shows the index of the generated random number, while the vertical axis shows its value.

A 25 ms wait leads to a linear approximation for the ESP32 as well.

In the case of the Arduino Mega, we obtained a better quality data set than the one sampled with a 25 ms wait presented in **Figure 1** (even better than the example shown with a 10 ms wait), which we analysed further for two reasons:

– because it belongs to the weaker quality samples obtained;
– because it was easier to build our demonstration circuit with this circuit.

During the analysis, we calculated the empirical average and standard deviation of the sample: $m = 32518{,}35$, $\sigma = 19234{,}06$. Since the obtained numbers should have a uniform distribution between 0 and 65,535, the theoretical average would be 32,767.50, and the standard deviation would be 18,918.32. We observe that these are relatively close to each other.

We calculated the empirical distribution function of the sample, as well as the value of the theoretical distribution function calculated for the sample members. Based on these two sets of numbers, we performed a goodness-of-fit test using the χ^2 test (*CHISQ.TEST*) in Excel, and the returned value was 1, indicating perfect fit (even though we can see that it is not exactly perfect), so we can say with a high degree of confidence that the generated random numbers are uniformly distributed.

Due to the suspiciously good fit, we also performed another non-parametric test, the Kolmogorov–Smirnov test. This test decides whether the empirical and theoretical distribution functions can be considered identical or not based on the largest difference between them. The change in the absolute value of D_i is shown in **Figure 2.** The maximum value of D_i must be compared with a critical value, which depends on the level of confidence and the number of samples: if it exceeds the critical value, then it cannot be accepted at the given level of confidence that the sample has the specified theoretical distribution

The highest value of D_i should be compared with a critical D_{cr}, value dependent on the level of reliability and the number of samples: if it exceeds the critical value, then it cannot be accepted at the given level of reliability that the sample follows the theoretical distribution.

In the case of our sample, which consisted of 16,384 numbers of not very good quality, the K-S test led to a result that contradicted the χ^2 test at the usual levels of confidence. Therefore, we could not prove that the distribution is uniform. However, if we perform the K-S test not with the distribution function determined by the elements of the sample, but with the histogram obtained by grouping them into intervals, then a positive result can be obtained as well.

As a check, we performed the same calculations with pseudorandom number sequences generated in Excel, and the results were similarly good or even better.

However, there was a criticism regarding the software-generated sequences, that if they are not long enough, they may show a distribution far from uniform. Therefore, we divided our hardware-generated number sequence into smaller subsamples, shorter number sequences, in their order of creation, and examined how their empirical mean and variance behave. In **Figure 3** we can see the case of our number sequence divided into 256 subsamples of 128 elements, where the empirical mean varies between 28,307.41 and 36,125.93.

The greatest deviation from the mean of the entire sample is shown by the value of 28,307.41 (in the 92nd subset). The graph of the ordered numbers in this subset can be seen in **Figure 4.** According to the chi-squared test and the K-S test (which was performed with 128 numbers and not using a histogram approximation) conducted on this subset, the hypothesis of a uniform distribution can still be accepted with very high probability.

Additional tools are often used to examine the uniformity of possible repetitions and the distribution of consecutive (i.e., non-ordered) numbers.

A commonly used solution is the visual representation of number sequences bit by bit, where the 0 bits are black and the 1 bits encode a white pixel in a black and white bitmap. If the distribution is truly uniform, then this image should be uniformly grey and should not show any repetition or pattern. We applied this principle a little differently by assigning not one bit but one byte to each pixel, so the pixels in the resulting image can have 256 different colours. This image should also be free of repetitions and patterns and should be uniformly grey, so there should be no red or blue shaded spots on it, for example. For the described sample, we obtained a 256×128

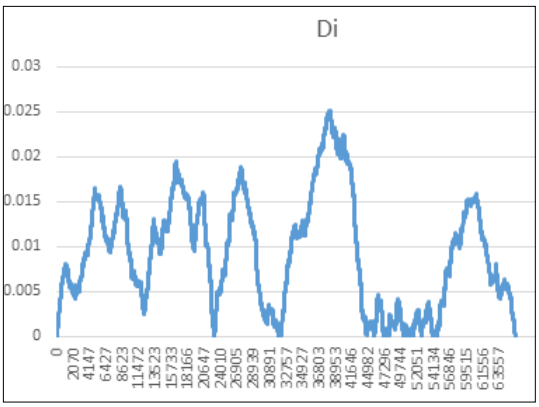

Fig. 2. *The absolute magnitude of the difference between the empirical and theoretical density functions.*

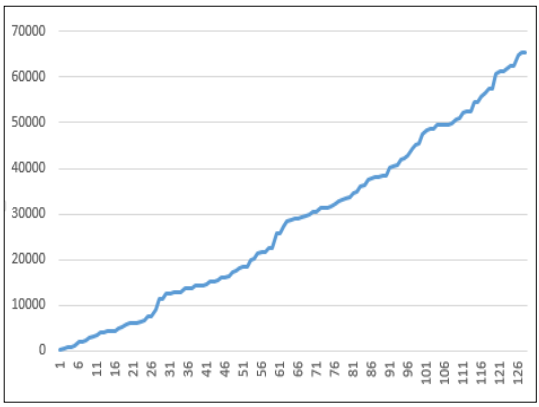

Fig. 4. *Graph of the elements of the sub-sample showing the largest deviation from the average.*

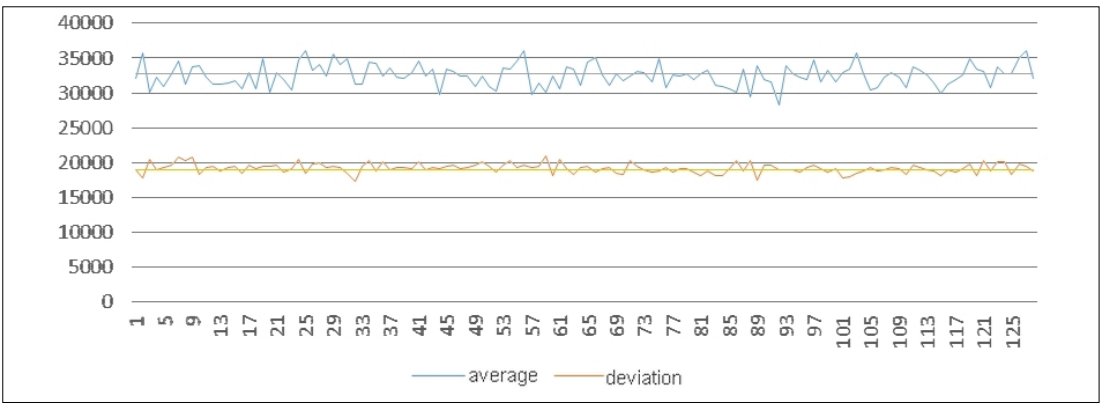

Fig. 3. *The variation of empirical mean and standard deviation around the theoretical values for the sample divided into 128-element strings.*

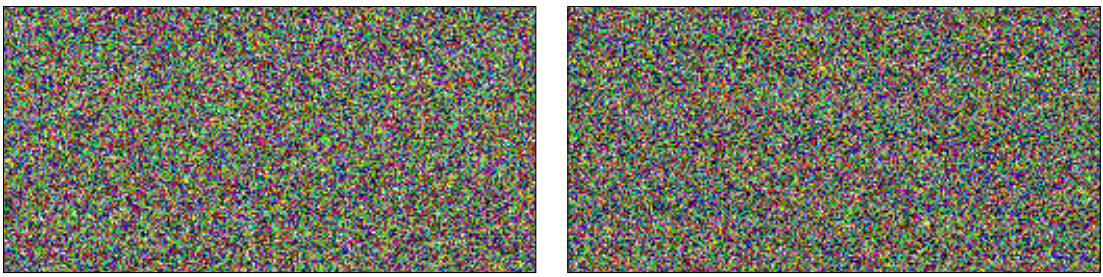

Fig. 5. *Bitmap representation of random numbers (left: Arduino Mega, right: ESP32).*

pixel sized, bitmap-format image on the left side of **Figure 5.** On the same figure, the rectangle on the right shows the result obtained with the number sequence generated by ESP32 with a waiting time of 25 ms. By examining the images, we can be sure that the obtained number sequences are indeed uniformly distributed.

4. The Galton Board as a didactic tool

In 1889, Francis Galton, an English polymath, built a device known as the Galton board to illustrate random processes intuitively. The Galton board consists of a sloping board with nails arranged in a chessboard-like pattern at equal distances from each other. At the bottom of the board, compartments are formed where the disks or balls launched from the same point above collect.

As the disk slides downward, it collides with the first nail, which prevents it from freely sliding further. Here, the disk bounces once and randomly, with probability p or q, it will slide to the right or left side of the nail, respectively, and continue downward until it reaches the next row of nails. In the "classic" version $p = q = 0.5$. Therefore, if we slide enough disks down, approximately half of them will end up on the first nail of the second row, and half will fall on the second nail.

In the second row, the left or right deviation is repeated. The probability of deviation to the right or left is the same, so the probability of further progress along any possible path is halved. However, the middle nail in the second row can be reached by the sliding disk in two different ways, so the probability of reaching it is the sum of the two appropriate probabilities (**Figure 6**).

The situation becomes more complicated in the third and subsequent rows. The calculations necessary for explanation can be made more manageable by constructing the Pascal triangle in Excel (**Figure 7**).

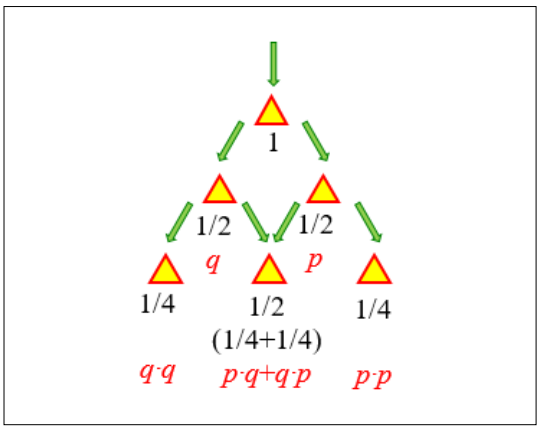

Fig. 6. *Collision with the second nail.*

5. The virtual Galton board

Our visualization tool is a virtual Galton board (**Figure 8**). In reality, it is a tangible device that simulates the phenomena that occur on the actual board. The device's "heart" is an Arduino Mega 2560 development board, to which we connected a 32×32 RGB LED matrix. The "disk" is a pixel (a coloured LED on the board), which falls downward at a constant (user-modifiable) speed. If it hits a "nail" (which is also a pixel) during its descent, we use a random number generator to determine which direction it should deviate (0 – left or 1 – right).

The probability of the disks deviating left or right on the original Galton board can be 0.5, but in our upgraded version, we can also adjust it to match the tilt of the stand, allowing us to study the general binomial distribution, not just the symmetric case corresponding to $p = 0{,}5$. For example, if we raise the left side of the device, the disks are more likely to fall to the right, and their distribution becomes skew.

The angle of deviation α of the stand from the vertical position is measured by an analog MMA7261QT accelerometer sensor.

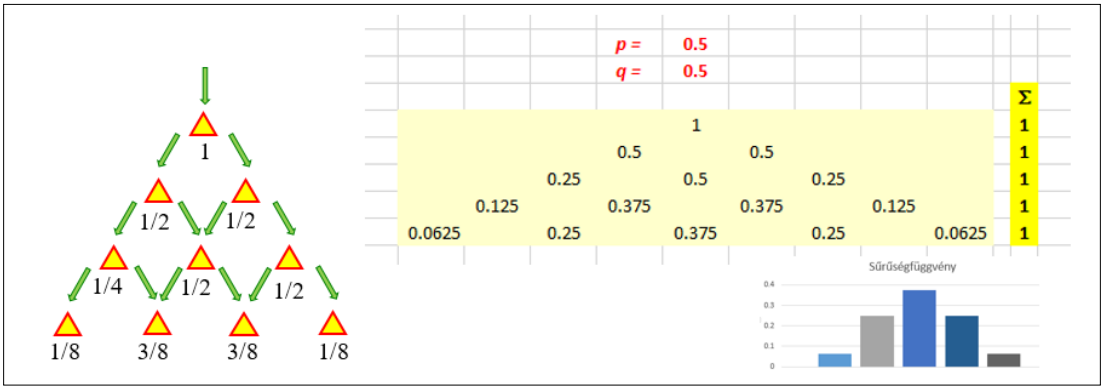

Fig. 7. *Collision with further nails.*

If the Galton board is not tilted, the disks bounce to the right or left with equal probability. If we tilt the board, this probability changes to:

$p = 0{,}5 + \sin \alpha$.

This formula only makes sense if α is in the [–30°, +30°] interval; if $\alpha < -30°$, then $p = 0$, and if $\alpha > +30°$, then $p = 1$.

Only a limited number of disks can fit in the holder, so if it gets full, we need to scale down the column heights. Therefore, after a while, the number of disks no longer corresponds to the number of illuminated pixels.

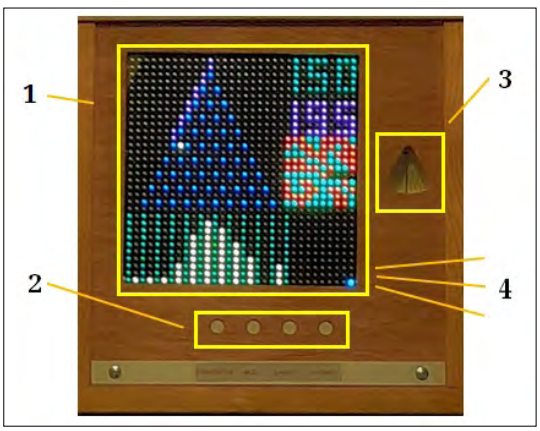

Fig. 8. *The virtual Galton board*
1- 32×32 LED panel
2- Control buttons, in order: start/stop, mode selection, animation slowdown, animation speedup;
3- Inclinometer showing the probability p calculated from the sine of the tilt angle;
4- Mode indicator:
Top LED: true random numbers,
Middle LED: pseudorandom numbers,
Bottom LED: pseudorandom numbers influenced by the tilt angle.

After every hundredth rolling ball, the Arduino sends the number of collected balls in the holder through the serial port, which can be copied from the "Serial monitor" window of the Arduino IDE program for further analysis. We can draw histograms of the frequencies, calculate the empirical mean and variance of the distribution, and perform statistical tests.

6. Conclusions

Discussions on easily constructible hardware random number generators often refer to the noise of floating (unconnected) inputs of microcontrollers, from which randomly varying data can be obtained through sampling. If we need random numbers in programming a microcontroller system, we don't really need separate devices, just the code that generates the random numbers. However, based on our experience, the state of an input, whether it's analogue or digital, changes randomly but too slowly, so the distribution of consecutive random data will not be satisfactory. Therefore, we further developed the basic idea by working with the least significant bit of randomly sampled analogue input values, and if we need random numbers from a certain range, rather than simple "yes/no" outputs, we combine the resulting sequence of bits.

Experience has shown that if the analogue inputs are sampled too frequently, the result obtained will not be as desired: most likely, the sampling circuit will not have enough time to return to its initial state. If we want to generate a long sequence of random numbers, reading the random bits from a single analogue input would be a time-consuming process due to the insertion of gaps between samples. Therefore, to determine a byte of a random number, we read the state of

not one, but eight inputs at the same time (actually one after the other, without inserted interruptions).

The numbers obtained in this way have shown a uniform distribution. Based on these, normally distributed numbers can be generated using the Box-Müller method.

References

[1] Mcloughlin C., Krakowski K.: T*echnological Tools for Visual Thinking: What Does the Research Tell Us?* Paper presented at the Apple. University Consortium Academic and Developers Conference, James Cook, 13.1–13.12 (2001)

[2] Khasanovna U. N.: *The Importance of Didactic Tools in the Teaching of Educational Science*. Academicia Globe: Inderscience Research, 2. (2021) 76–79.

[3] Johnston D.: *Random Number Generators—Principles and Practices: A Guide for Engineers and Programmers.* Walter de Gruyter GmbH & Co KG, 2018

[4] *** Arduino Mega Rev. 3. Documentation, https://docs.arduino.cc/hardware/mega-2560

[5] *** ESP32 Reference, https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/index.html

[6] *** Arduino Language Reference, https://www.arduino.cc/reference/en/