How to Easily Create a Matrix of Random Numbers in R

Authored by

stats writer

November 28, 2025

Generating a matrix populated with **random numbers** is a fundamental task in statistical computing and data simulation using R. This process typically involves combining two essential base R functions: `runif()`, which handles the generation of uniform random values, and `matrix()`, which structures these values into the desired two-dimensional data structure. Understanding how to integrate these functions allows users to quickly create large, randomized datasets for testing algorithms, bootstrapping procedures, or simulating complex scenarios.

The primary mechanism for this generation relies on the `runif()` function. This function is designed to draw samples from a uniform distribution, requiring parameters to define the number of values needed (`n`), the lower bound (`min`), and the upper bound (`max`). These generated values, which are by default floating-point numbers, are then passed into the `matrix()` constructor, along with specifications for the number of rows or columns, to finalize the structure of the resulting random array.

To effectively generate a matrix of **random numbers** in R, two distinct, highly useful methods are generally employed, depending on whether the required output should consist of continuous floating-point values or discrete integer values. We will explore both techniques in detail, focusing on practical implementation and critical function usage.

The Core Functions: `runif()` and `matrix()`

The versatility of R stems greatly from its robust suite of statistical generation functions. When working with uniform distributions, the `runif()` function is the established standard. It requires explicit declaration of the range boundaries (`min` and `max`) and the total quantity of numbers (`n`) to be produced. Crucially, the **random numbers** generated by `runif()` are inherently continuous, meaning they include decimals unless explicitly modified.

The generated vector of values must then be shaped into a two-dimensional matrix. This is where the `matrix()` function steps in. The `matrix()` function takes the vector of data as its first argument and structural parameters such as `nrow` (number of rows) or `ncol` (number of columns). It organizes the input data column-wise by default, unless the `byrow=TRUE` argument is specified. Combining `runif()` inside `matrix()` is the most efficient way to achieve the desired outcome in a single line of code.

Method 1: Creating a Matrix with Random Floating-Point Values

The most direct method for generating a randomized matrix involves relying solely on the combination of `matrix()` and `runif()`. This technique is ideal when the simulation requires continuous values, often used in complex mathematical modeling or Monte Carlo simulations where fractional precision is necessary. We must first specify how many total values (`n`) are

needed, followed by the desired lower and upper boundaries of the distribution.

For instance, if the goal is to create a matrix containing 10 random values uniformly distributed between 1 and 20, and structure them into 5 rows, the parameters are set as `n=10`, `min=1`, and `max=20`, and `nrow=5`. Since the total count (10) must be divisible by the number of rows (5), the resulting matrix will automatically possess 2 columns (10 / 5 = 2). This ensures that the generated vector completely fills the declared matrix structure.

The following abbreviated code snippet illustrates the syntax for generating a 5x2 matrix containing 10 **random numbers**:

```
#create matrix of 10 random values between 1 and 20
random_matrix <- matrix(runif(n=10, min=1, max=20), nrow=5)
```

Ensuring Reproducibility with `set.seed()`

When generating **random numbers**, especially in analytical or research contexts, it is critical that the results are reproducible. Without a mechanism to control the random number generator, running the same code twice will produce two different matrices, which severely hinders debugging and verification. The standard way to address this issue in R is by utilizing the `set.seed()` function.

The `set.seed()` function initializes R's internal random number generator state. By providing an arbitrary integer value (the "seed") to this function, subsequent calls to functions like `runif()` will always generate the exact same sequence of pseudo-random values. This is essential for ensuring that code examples shared with others, or tests run across different systems, yield identical numerical outcomes, thereby validating the statistical procedure being performed.

Detailed Example: Generating a Matrix of Continuous Random Values

To demonstrate Method 1 in a reproducible setting, we execute the full code block below. We utilize `set.seed(1)` to fix the sequence of values. We are requesting 10 total values (`n=10`) within the range , and instructing the `matrix()` function to organize them into 5 rows (`nrow=5`). The resulting structure will inherently be a 5x2 matrix.

```
#make this example reproducible
set.seed(1)

#create matrix with 10 random numbers between 1 and 20
random_matrix <- matrix(runif(n=10, min=1, max=20), nrow=5)
```

```
#view matrix
random_matrix
```

```
6.044665 18.069404
8.070354 18.948830
11.884214 13.555158
18.255948 12.953167
4.831957 2.173939
```

As demonstrated by the output, the resulting structure is indeed a 5-row, 2-column matrix. Every value contained within this matrix is a **random number** sampled from the continuous uniform distribution defined by the interval . Since `runif()` generates floating-point numbers, the matrix elements exhibit high precision.

Method 2: Generating Matrices with Random Integers

There are many instances, particularly in discrete simulation or introductory statistics, where only whole numbers (integers) are required for the random generation process. Since `runif()` produces continuous floating-point numbers, an additional step is necessary to transform these results into discrete integers before passing them to the `matrix()` function.

The standard method for achieving this transformation is by wrapping the `runif()` call within the `round()` function. By setting the second argument of `round()` to `0`, we ensure that the generated floating-point numbers are rounded to the nearest whole number. This combined approach guarantees that the resulting matrix contains strictly integer values, adhering to the specified range constraints.

The syntax below demonstrates the inclusion of the `round()` function to generate 10 random integers between 1 and 20, structured into a 5-row matrix:

```
#create matrix of 10 random integers between 1 and 20
random_matrix <- matrix(round(runif(n=10, min=1, max=20), 0), nrow=5)
```

Detailed Example: Generating a Matrix of Discrete Random Integers

We now apply Method 2 to generate a 5x2 matrix where all 10 elements are integers ranging between 1 and 50. In this example, we increase the maximum range to illustrate the variability better. As before, we invoke `set.seed(1)` to ensure the output remains consistent across execution environments in R. Note how the `runif()` function is contained within `round(..., 0)`, guaranteeing only whole numbers are passed to the `matrix()` constructor.

#make this example reproducible
set.seed(1)

#create matrix with 10 random integers between 1 and 50
random_matrix <- matrix(round(runif(n=10, min=1, max=50), 0), nrow=5)

#view matrix
random_matrix

14 45
19 47
29 33
46 32
11 4

The final output is a clearly defined 5-row, 2-column matrix. Since the values were rounded, every element is a discrete integer that falls within the established boundaries of 1 and 50. This confirms the successful transformation of the continuous output of `runif()` into the required discrete form using `round()`.

Important Considerations for Uniform Random Generation

When working with generated **random numbers** from the `runif()` function, particularly when defining the range, users must understand the mathematical nature of the uniform distribution in R. The `runif()` function, by definition, samples from a continuous distribution over the closed interval . This means that the resulting values can potentially include the exact minimum and maximum bounds specified in the function call.

For instance, in the integer example where the range was , it is mathematically possible, though statistically rare, for the generated matrix to contain the value 1 or the value 50, or both, after the rounding operation is applied. This inclusiveness is a key feature of the function and should be accounted for when defining experimental constraints. If strictly exclusive boundaries are needed (e.g., excluding 1 and 50), the user must adjust the `min` and `max` parameters slightly inward to compensate for the rounding or floating-point precision.

Furthermore, it is important to remember that `runif()` samples with replacement from the theoretical distribution. Consequently, generating a large sample of random values, especially when rounding them to integers, makes it highly probable that the same number will appear multiple times within the resulting vector or matrix. If the simulation requires unique random values (sampling without replacement), the `sample()` function should be used instead of `runif()`, although `sample()` is typically used for generating discrete integers, not continuous floats.