typst Package

suiji

User's Manual

Version **0.4.0**
April 2025

Guangxi Liu

Contents

1 Introduction

The package `suiji`[1] is a high efficient random number generator in Typst. Partial algorithm is inherited from GSL[2] and most APIs are similar to NumPy Random Generator[3]. It provides pure function implementation and does not rely on any global state variables, resulting in better performance and independency.

It has the following features:

- All functions are immutable, which means results of random are completely deterministic.
- Core random engine chooses *maximally equidistributed combined Tausworthe generator*.
- Generate random integers or floats from various distribution.
- Randomly shuffle an array of objects.
- Randomly sample from an array of objects.
- Accelerate random number generation based on the WebAssembly plugin.

To use it, import the latest version of this package with:

```
#import "@preview/suiji:0.4.0": *
```

This line will be omitted in the examples codes that follows.

In the following sections, the use of the corresponding random functions are described in detail.

[1] https://typst.app/universe/package/suiji
[2] https://www.gnu.org/software/gsl
[3] https://numpy.org/doc/stable/reference/random/generator.html

2 Guide

2.1 Random Number Algorithm

The algorithm implemented in this package is actually a *pseudorandom number generator* (PRNG). The PRNG-generated sequence is not truly random, because it is completely determined by an initial value, called the PRNG's seed (which may include truly random values).

By balancing the complexity of the implementation and the quality of the results, the PRNG chooses *maximally equidistributed combined Tausworthe generator*. The sequence is

$$x_n = s_n^1 \oplus s_n^2 \oplus s_n^3$$

where

$$\begin{aligned}
s_{n+1}^1 &= ((s_n^1 \,\&\, 4294967294) \ll 12) \oplus (((s_n^1 \ll 13) \oplus s_n^1) \gg 19) \\
s_{n+1}^2 &= ((s_n^2 \,\&\, 4294967288) \ll 4) \oplus (((s_n^2 \ll 2) \oplus s_n^2) \gg 25) \\
s_{n+1}^3 &= ((s_n^3 \,\&\, 4294967280) \ll 17) \oplus (((s_n^3 \ll 3) \oplus s_n^3) \gg 11)
\end{aligned}$$

computed modulo 2^{32}. In the formulas above $\oplus$ denotes *exclusive-or*.

The period of this generator is 2^{88} (about 10^{26}). It uses 3 words (32-bit unsigned integers) of state per generator.

The generator only accept 32-bit seed, with higher values being reduced modulo 2^{32}. A new random integer from $[0, 2^{32})$ is obtained by updating the state. Samples from each distribution implemented here can be obtained using the generator as an underlying source of randomness.

2.2 Design Considerations

As mentioned earlier, the process of generating random numbers requires state preservation. When implemented in Typst, the `state` function can be used to preserve the state values of random numbers. However, Typst's state management system is not very efficient. A purely functional style is preferred. This package chooses the latter.

Function purity means that for the same arguments, they always return the same result. They cannot "remember" things to produce another value when they are called a second time. To follow this rule, a special variable `rng` is chosen as the input and output parameters of the random number generation function. The variable `rng` is actually the 3 state words of the underlying random generator. Within the function, its value is updated and output.

Therefore, the basic usage of the random number generation function is as follows.

- The original `rng` should be created by function `gen-rng`, with an integer as the argument of seed.
- Call other random number generation functions as needed, using `rng` as input and output parameters.

As long as the value of the initialized `rng` does not change, the result of the random number output is stable and reproducible.

2.3 WASM plugin

Random number generation is a compute-intensive task. And this task could be accelerated based on the WebAssembly (WASM) plugin.

In addition to the computation itself, there is an additional protocol overhead for calling the plugin. To minimize this overhead, the following optimization is used.

- Input and output parameters are packaged into an array type.
- Input and output array is serialized as CBOR[4] format and transmitted in the protocol buffers.
- The validity of the parameters is checked on the Typst wrapper function.

The benchmark (see Section 4) shows that the acceleration is noticeable with plugin, especially for batch random number generation.

The functions implemented based on the plugin ensure that the interface and functionality are completely consistent with the original Typst script versions. The only exception is that the results of normally distributed random numbers exhibit extremely minor differences, due to the different internal floating-point calculation methods.

The set of functions based on the plugin have the suffix `-f` in their names. For example, `gen-rng-f` and `integers-f` are the fast versions of `gen-rng` and `integers`, respectively. It is recommended to always use the function version accelerated by the plugin.

[4] https://cbor.io

3 Reference

3.1 `gen-rng` / `gen-rng-f`

Construct a new random number generator with a seed.

Note: The arguments and functions of `gen-rng` and `gen-rng-f` are the same.

Parameters

```
gen-rng(
  int,
) -> object
```

```
gen-rng-f(
  int,
) -> object
```

seed `int` *Required* *Positional*

The value of seed, effective value is an integer from $[0, 2^{32} - 1]$.

Note: Actually, any integer is acceptable because it performs modulo 2^{32} operations internally.

rng `object` *Returned*

Generated object of random number generator, which is transparent to the users.

3.2 `integers` / `integers-f`

Return random integers from low to high. The interval and the size of array can be customized.

Define *gap* for the sample interval is `high` - `low` if `endpoint` is `false`, otherwise `high` - `low` + 1. And the valid range of *gap* is $[1, 2^{32}]$.

Note: The arguments and functions of `integers` and `integers-f` are the same.

Parameters

```
integers(
  object,
  low: int,
  high: int,
  size: none int,
  endpoint: bool,
) -> (object, int array)
```

```
integers-f(
  object,
```

```
  low: int,
  high: int,
  size: none int,
  endpoint: bool,
) -> (object, int array)
```

rng `object` *Required* *Positional*

The object of random number generator.

low `int` *Settable*

The lowest (signed) integers to be drawn from the distribution.

Default: `0`

high `int` *Settable*

One above the largest (signed) integer to be drawn from the distribution.

Default: `100`

size `none` or `int` *Settable*

The returned array size, must be `none` or non-negative integer. Here `none` means return single random number (i.e. `size` is 1).

Default: `none`

endpoint `bool` *Settable*

if `true`, sample from the interval [`low`, `high`] instead of the default [`low`, `high`).

Default: `false`

(rng, arr) `array` *Returned*

- `rng` `object`: Returned updated object of random number generator.
- `arr` `int` `array`: Returned single or array of random numbers.

3.3 `random` / `random-f`

Return random floats in the half-open interval [0.0, 1.0). The size of array can be customized.

Note: The arguments and functions of `random` and `random-f` are the same.

Parameters

```
random(
  object,
```

```
  size: none int ,
  endpoint: bool ,
) -> ( object , float array )
```

```
random-f(
  object ,
  size: none int ,
  endpoint: bool ,
) -> ( object , float array )
```

rng `object` *Required Positional*

The object of random number generator.

size `none` or `int` *Settable*

The returned array size, must be none or non-negative integer. Here none means return single random number (i.e. `size` is 1).

Default: none

(rng, arr) `array` *Returned*

- `rng` `object` : Returned updated object of random number generator.
- `arr` `float` `array` : Returned single or array of random numbers.

3.4 `uniform` / `uniform-f`

Draw samples from a uniform distribution of half-open interval [low, high) (includes low, but excludes high). The interval and the size of array can be customized.

Note: The arguments and functions of `uniform` and `uniform-f` are the same.

Parameters

```
uniform(
  object ,
  low: int float ,
  high: int float ,
  size: none int ,
) -> ( object , float array )
```

```
uniform-f(
  object ,
  low: int float ,
  high: int float ,
```

```
  size: none int ,
) -> ( object , float array )
```

rng `object` *Required* *Positional*

The object of random number generator.

low `int` or `float` *Settable*

The lower boundary of the output interval.

Default: `0.0`

high `int` or `float` *Settable*

The upper boundary of the output interval.

Default: `1.0`

size `none` or `int` *Settable*

The returned array size, must be none or non-negative integer. Here none means return single random number (i.e. `size` is 1).

Default: none

(rng, arr) `array` *Returned*

- `rng` `object` : Returned updated object of random number generator.
- `arr` `float` `array` : Returned single or array of random numbers.

3.5 `normal` / `normal-f`

Draw random samples from a normal (Gaussian) distribution. The mean / standard deviation and the size of array can be customized.

Note: The arguments and functions of `normal` and `normal-f` are the same.

Parameters

```
normal(
  object ,
  loc: int float ,
  scale: int float ,
  size: none int ,
) -> ( object , float array )
```

```
normal-f(
  object ,
```

```
  loc: int float,
  scale: int float,
  size: none int,
) -> (object, float array)
```

rng `object` *Required* *Positional*

The object of random number generator.

loc `int` or `float` *Settable*

The mean (centre) of the distribution.

Default: `0.0`

scale `int` or `float` *Settable*

The standard deviation (spread or width) of the distribution, must be non-negative.

Default: `1.0`

size `none` or `int` *Settable*

The returned array size, must be none or non-negative integer. Here none means return single random number (i.e. `size` is 1).

Default: none

(rng, arr) `array` *Returned*

- `rng` `object`: Returned updated object of random number generator.
- `arr` `float` `array`: Returned single or array of random numbers.

3.6 `discrete-preproc` / `discrete-preproc-f`

Preprocess the given probalilities of the discrete events and return a object that contains the lookup table for the discrete random number generator. The returned object can only be used in functions of `discrete` or `discrete-f`.

Note: The arguments and functions of `discrete-preproc` and `discrete-preproc-f` are the same.

Parameters

```
discrete-preproc(
  array,
) -> dictionary
```

```
discrete-preproc-f(
  array,
) -> dictionary
```

p `array` *Required Positional*

The array of probalilities of the discrete events, probalilities must be non-negative.

g `dictionary` *Returned*

Generated object that contains the lookup table, which is transparent to the users.

3.7 `discrete` / `discrete-f`

Return random indices from the given probalilities of the discrete events. Require preprocessed probalilities of the discrete events.

Note: The arguments and functions of `discrete` and `discrete-f` are the same.

Parameters

```
discrete(
  object,
  dictionary,
  size: none int,
) -> (object, int array)
```

```
discrete-f(
  object,
  dictionary,
  size: none int,
) -> (object, int array)
```

rng `object` *Required Positional*

The object of random number generator.

g `dictionary` *Required Positional*

Generated object that contains the lookup table by `discrete-preproc` or `discrete-preproc-f` function.

size `none` or `int` *Settable*

The returned array size, must be none or non-negative integer. Here none means return single random number (i.e. `size` is 1).

Default: none

(rng, arr) `array` *Returned*

- `rng` `object` : Returned updated object of random number generator.
- `arr` `int` `array` : Returned single or array of random indices.

3.8 `shuffle` / `shuffle-f`

Randomly shuffle a given array.

Note: The arguments and functions of `shuffle` and `shuffle-f` are the same.

Parameters

```
shuffle(
  object,
  array,
) -> (object, array)
```

```
shuffle-f(
  object,
  array,
) -> (object, array)
```

rng `object` *Required* *Positional*

The object of random number generator.

arr `array` *Required* *Positional*

The array to be shuffled.

(rng, arr) `array` *Returned*

- `rng` `object` : Returned updated object of random number generator.
- `arr` `array` : Returned shuffled array.

3.9 `choice` / `choice-f`

Generate random samples from a given array. The sample assumes a uniform distribution over all entries in the array.

Note: The arguments and functions of `choice` and `choice-f` are the same.

Parameters

```
choice(
  object,
  array,
  size: none int,
```

```
  replacement: bool ,
  permutation: bool ,
) -> ( object , any array )
```

```
choice-f(
  object ,
  array ,
  size: none int ,
  replacement: bool ,
  permutation: bool ,
) -> ( object , any array )
```

rng `object` *Required Positional*

The object of random number generator.

arr `array` *Required Positional*

The array to be sampled.

size `none` or `int` *Settable*

The returned array size, must be none or non-negative integer. Here none means return single random sample (i.e. `size` is 1).

Default: none

replacement `bool` *Settable*

Whether the sample is with or without replacement. `true` meaning that a value of `arr` can be selected multiple times.

Default: `true`

permutation `bool` *Settable*

Whether the sample is permuted when sampling without replacement. `false` provides a speedup.

Default: `true`

(rng, arr) `array` *Returned*

- `rng` `object` : Returned updated object of random number generator.
- `arr` `any` `array` : Returned single or array of random samples.

4 Benchmarks

4.1 Cases

Various test cases are used to compare the execution efficiency of the two types of functions in different scenarios.

Random Integers

Generate random integers with different array size. The test codes are as follows.

- **bench-integers-1**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 100000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = integers(rng, low: 0, high: 10000, size: n-sz)
  }
}
```

- **bench-integers-f-1**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 100000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = integers-f(rng, low: 0, high: 10000, size: n-sz)
  }
}
```

- **bench-integers-2**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 1000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = integers(rng, low: 0, high: 10000, size: n-sz)
  }
}
```

▶ **bench-integers-f-2**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 1000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = integers-f(rng, low: 0, high: 10000, size: n-sz)
  }
}
```

▶ **bench-integers-3**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 1
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = integers(rng, low: 0, high: 10000, size: n-sz)
  }
}
```

▶ **bench-integers-f-3**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 1
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = integers-f(rng, low: 0, high: 10000, size: n-sz)
  }
}
```

Uniform Distribution Random Numbers

Generate random numbers from a uniform distribution with different array size. The test codes are as follows.

▶ **bench-uniform-1**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 100000
  let n-loop = int(n-tot / n-sz)
```

```
  let rng = gen-rng(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = uniform(rng, low: -10.0, high: 10.0, size: n-sz)
  }
}
```

▶ bench-uniform-f-1

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 100000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = uniform-f(rng, low: -10.0, high: 10.0, size: n-sz)
  }
}
```

▶ bench-uniform-2

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 1000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = uniform(rng, low: -10.0, high: 10.0, size: n-sz)
  }
}
```

▶ bench-uniform-f-2

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 1000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = uniform-f(rng, low: -10.0, high: 10.0, size: n-sz)
  }
}
```

▶ bench-uniform-3

```
#{
  let seed = 0
```

```
  let n-tot = 100000
  let n-sz = 1
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = uniform(rng, low: -10.0, high: 10.0, size: n-sz)
  }
}
```

▶ **bench-uniform-f-3**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 1
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = uniform-f(rng, low: -10.0, high: 10.0, size: n-sz)
  }
}
```

Normal Distribution Random Numbers

Generate random numbers from a normal distribution with different array size. The test codes are as follows.

▶ **bench-normal-1**

```
#{
  let seed = 0
  let n-tot = 50000
  let n-sz = 50000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = normal(rng, loc: 5.0, scale: 10.0, size: n-sz)
  }
}
```

▶ **bench-normal-f-1**

```
#{
  let seed = 0
  let n-tot = 50000
  let n-sz = 50000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = ()
  for _ in range(n-loop) {
```

```
    (rng, data) = normal-f(rng, loc: 5.0, scale: 10.0, size: n-sz)
  }
}
```

▸ bench-normal-2

```
#{
  let seed = 0
  let n-tot = 50000
  let n-sz = 1000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = normal(rng, loc: 5.0, scale: 10.0, size: n-sz)
  }
}
```

▸ bench-normal-f-2

```
#{
  let seed = 0
  let n-tot = 50000
  let n-sz = 1000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = normal-f(rng, loc: 5.0, scale: 10.0, size: n-sz)
  }
}
```

▸ bench-normal-3

```
#{
  let seed = 0
  let n-tot = 50000
  let n-sz = 1
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = normal(rng, loc: 5.0, scale: 10.0, size: n-sz)
  }
}
```

▸ bench-normal-f-3

```
#{
  let seed = 0
  let n-tot = 50000
  let n-sz = 1
  let n-loop = int(n-tot / n-sz)
```

```
  let rng = gen-rng-f(seed)
  let data = ()
  for _ in range(n-loop) {
    (rng, data) = normal-f(rng, loc: 5.0, scale: 10.0, size: n-sz)
  }
}
```

Random Shuffle

Randomly shuffle a given array with different array size. The test codes are as follows.

- **bench-shuffle-1**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 100000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = range(n-sz)
  for _ in range(n-loop) {
    (rng, data) = shuffle(rng, data)
  }
}
```

- **bench-shuffle-f-1**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 100000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = range(n-sz)
  for _ in range(n-loop) {
    (rng, data) = shuffle-f(rng, data)
  }
}
```

- **bench-shuffle-2**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 1000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = range(n-sz)
  for _ in range(n-loop) {
    (rng, data) = shuffle(rng, data)
  }
}
```

- **bench-shuffle-f-2**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 1000
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = range(n-sz)
  for _ in range(n-loop) {
    (rng, data) = shuffle-f(rng, data)
  }
}
```

- **bench-shuffle-3**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 10
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng(seed)
  let data = range(n-sz)
  for _ in range(n-loop) {
    (rng, data) = shuffle(rng, data)
  }
}
```

- **bench-shuffle-f-3**

```
#{
  let seed = 0
  let n-tot = 100000
  let n-sz = 10
  let n-loop = int(n-tot / n-sz)

  let rng = gen-rng-f(seed)
  let data = range(n-sz)
  for _ in range(n-loop) {
    (rng, data) = shuffle-f(rng, data)
  }
}
```

4.2 Results

The command-line tool `hyperfine`[5] is used for benchmarking. For each case, it runs the benchmark on a warm cache and performs 10 benchmarking runs.

The list of benchmark results is below.

[5] https://github.com/sharkdp/hyperfine

Table 1 Benchmark results

Case Name	Average Time (mean ± σ)
bench-integers-1	**8.321 s** ± 0.154 s
bench-integers-f-1	**429.3 ms** ± 25.9 ms
bench-integers-2	**8.341 s** ± 0.197 s
bench-integers-f-2	**418.8 ms** ± 11.7 ms
bench-integers-3	**11.123 s** ± 0.369 s
bench-integers-f-3	**8.049 s** ± 0.504 s
bench-uniform-1	**7.747 s** ± 0.252 s
bench-uniform-f-1	**439.7 ms** ± 10.9 ms
bench-uniform-2	**7.824 s** ± 0.133 s
bench-uniform-f-2	**446.3 ms** ± 10.7 ms
bench-uniform-3	**9.766 s** ± 0.065 s
bench-uniform-f-3	**8.026 s** ± 0.643 s
bench-normal-1	**9.786 s** ± 0.151 s
bench-normal-f-1	**417.2 ms** ± 11.7 ms
bench-normal-2	**9.858 s** ± 0.130 s
bench-normal-f-2	**426.0 ms** ± 14.1 ms
bench-normal-3	**11.110 s** ± 0.270 s
bench-normal-f-3	**4.112 s** ± 0.130 s
bench-shuffle-1	**8.558 s** ± 0.132 s
bench-shuffle-f-1	**882.1 ms** ± 20.2 ms
bench-shuffle-2	**8.489 s** ± 0.060 s
bench-shuffle-f-2	**866.7 ms** ± 13.1 ms
bench-shuffle-3	**7.936 s** ± 0.109 s
bench-shuffle-f-3	**1.522 s** ± 0.027 s

Note that the total length of data processed in each set of test cases is the same. The following results can be observed:

- The more data processed in each round, the more noticeable the acceleration becomes. Especially in the normal tests, the plugin version (bench-normal-f-1) is about 23 times faster than original one (bench-normal-1).
- If the amount of data processed per round is very small (for example, only one), there is only a minimal performance improvement. The overhead of the function call becomes more apparent. But more efficient in the normal and shuffle tests, as the internal calculations are more complex for a single process.
- The original version also becomes slower if the amount of data processed per round is very small. So in practice, as many random numbers as possible should be generated at once.

5 Examples

5.1 Basic Functions

Here the same seed is specified to two generators so that we have reproducible results.

```
#let print-arr(arr) = {
  if type(arr) != array {
    [#raw(str(arr) + " ")]
  } else {
    [#raw(arr.map(it => str(it)).join(" "))]
  }
}

#{
  let seed = 1
  let rng1 = gen-rng-f(seed)
  let rng2 = gen-rng-f(seed)
  let (_, arr1) = random-f(rng1, size: 3)
  let (_, arr2) = random-f(rng2, size: 3)
  print-arr(arr1); parbreak()
  print-arr(arr2); parbreak()
}
```

```
0.18691460322588682 0.9510397978592664 0.5454357711132616
0.18691460322588682 0.9510397978592664 0.5454357711132616
```

For random number generator type function like `integers-f`, it doesn't matter if you call the function once or multiple times, you end up with the same array value.

```
#let print-arr(arr) = {
  if type(arr) != array {
    [#raw(str(arr) + " ")]
  } else {
    [#raw(arr.map(it => str(it)).join(" "))]
  }
}

#{
  let seed = 1

  let rng = gen-rng-f(seed)
  let arr1 = ()
  let (rng, arr1) = integers-f(rng, low: 0, high: 100, size: 20)
  print-arr(arr1); parbreak()

  let rng = gen-rng-f(seed)
  let arr2 = ()
  let val
  for _ in range(4) {
    (rng, val) = integers-f(rng, low: 0, high: 100, size: 5)
    arr2.push(val)
  }
  arr2 = arr2.flatten()
  print-arr(arr2); parbreak()
```

```
  let rng = gen-rng-f(seed)
  let arr3 = ()
  let val
  for _ in range(20) {
    (rng, val) = integers-f(rng, low: 0, high: 100, size: none)
    arr3.push(val)
  }
  print-arr(arr3); parbreak()
}
```

```
18 95 54 7 22 52 9 51 33 79 0 34 29 93 20 2 25 68 98 44
18 95 54 7 22 52 9 51 33 79 0 34 29 93 20 2 25 68 98 44
18 95 54 7 22 52 9 51 33 79 0 34 29 93 20 2 25 68 98 44
```

Here we generate 10000 random samples from a normal distribution with $N(5.0, 10.0^2)$, and calculate their mean and standard deviation.

```
#{
  let rng = gen-rng-f(42)
  let arr = ()
  let n = 10000
  (_, arr) = normal-f(rng, loc: 5.0, scale: 10.0, size: n)
  let a-mean = arr.sum() / n
  let a-std = calc.sqrt(arr.map(x => calc.pow(x - a-mean, 2)).sum() / n)
  [#raw("mean = " + str(a-mean)) \ #raw("std = " + str(a-std))]
}
```

```
mean = 4.824476371409619
std = 10.078050891075048
```

Here we shuffle the sequence of the English letters A-Z.

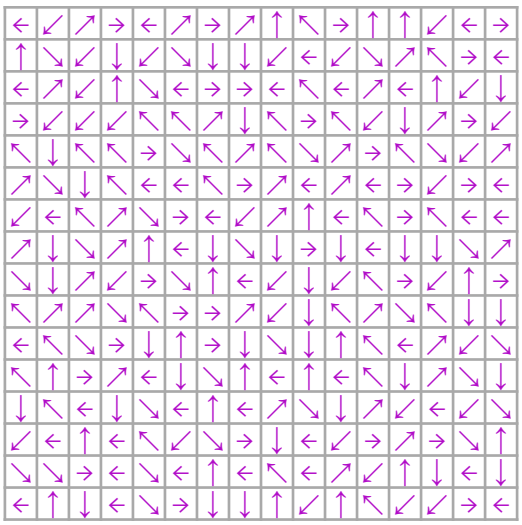
```
#{
  let rng = gen-rng-f(1)
  let arr = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".clusters()
  let (_, arr-s) = shuffle-f(rng, arr)
  raw(arr-s.join())
}
```

```
USQTIODHRLPCMZVAYGJWKFBNXE
```

Here, we randomly select 256 arrows from 8 different sets of arrows and arrange them in a 16 by 16 grid.

```
#{
  set text(size: 10pt, fill: purple)
  let rng = gen-rng-f(1)
  let arr = (sym.arrow.r, sym.arrow.l, sym.arrow.t, sym.arrow.b, sym.arrow.tr,
sym.arrow.br, sym.arrow.tl, sym.arrow.bl)
  let (_, arr-c) = choice-f(rng, arr, size: 256)
  grid(
    align: horizon + center,
    columns: (12pt,)*16, rows: 12pt,
    stroke: gray + 1pt,
    ..arr-c
```

```
  )
}
```



Here we simulate a biased dice throw, with the occurrence probability of the six sides is 1/21, 2/21, 3/21, 4/21, 5/21 and 6/21, respectively.

```
#let print-arr(arr) = {
  if type(arr) != array {
    [#raw(str(arr) + " ")]
  } else {
    [#raw(arr.map(it => str(it)).join(" "))]
  }
}

#{
  let rng = gen-rng-f(1)
  let p = (1/21, 2/21, 3/21, 4/21, 5/21, 6/21)
  let g = discrete-preproc-f(p)
  let (_, arr) = discrete-f(rng, g, size: 300)
  let points = arr.map(x => x + 1)
  print-arr(points)
}
```

```
2 5 4 6 2 4 6 4 3 5 1 3 6 5 2 1 2 5 5 3 5 5 6 3 5 2 5 4 4 4 1 2 2 4 5 6 5 2 6 6 2 4 2 4 2
6 5 3 1 5 5 5 5 4 4 3 4 6 5 6 3 4 1 5 6 3 4 6 6 5 4 4 6 5 4 6 6 4 6 4 4 4 3 6 2 2 6 1 4 2
4 6 6 6 6 2 6 6 4 4 5 1 6 5 4 6 6 4 6 5 2 2 4 3 5 4 3 2 3 5 3 1 6 2 3 3 4 4 5 3 4 5 1 5 3
2 6 3 5 5 6 6 6 6 5 4 4 4 5 6 6 4 5 6 5 4 4 5 3 6 2 5 5 6 3 6 1 3 5 3 6 5 2 4 5 5 6 2 6 4
2 3 5 2 6 6 2 5 3 4 5 3 3 5 2 3 5 2 6 6 4 5 3 4 5 2 5 5 5 2 6 2 4 6 5 6 3 6 5 6 4 1 5 3 4
5 6 6 4 2 6 2 5 6 4 5 6 3 2 3 6 3 3 2 5 1 3 6 6 5 6 6 5 5 6 2 4 5 5 4 3 4 5 3 2 6 3 5 5 2
5 1 4 5 4 5 3 5 4 5 6 6 5 3 3 5 5 6 1 6 6 6 4 6 5 3 4 2 6 6
```

5.2 Graphics with Randomness

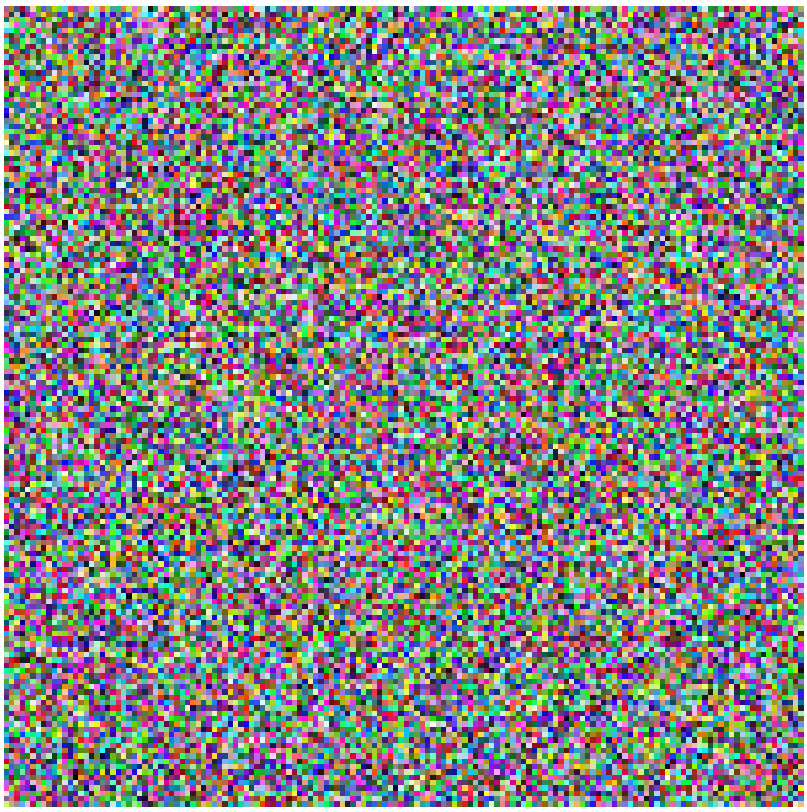
In data visualization or graphing, randomness may sometimes be introduced. Various random functions in the package can help with this.

Below is a random pixel map of 150 by 150. The RGB component of each pixel is a random number.

```
#{
  let seed = 123
  let unit = 2pt
  let (width, height) = (150, 150)

  let rng = gen-rng-f(seed)
  let (_, data) = integers-f(rng, low: 0, high: 256, size: width * height * 3)

  grid(columns: (unit,)*width, rows: unit,
    ..data.chunks(3).map(((r,g,b)) => grid.cell(fill: rgb(r,g,b))[])
  )
}
```



The example below creates a trajectory of a 2D random walk.

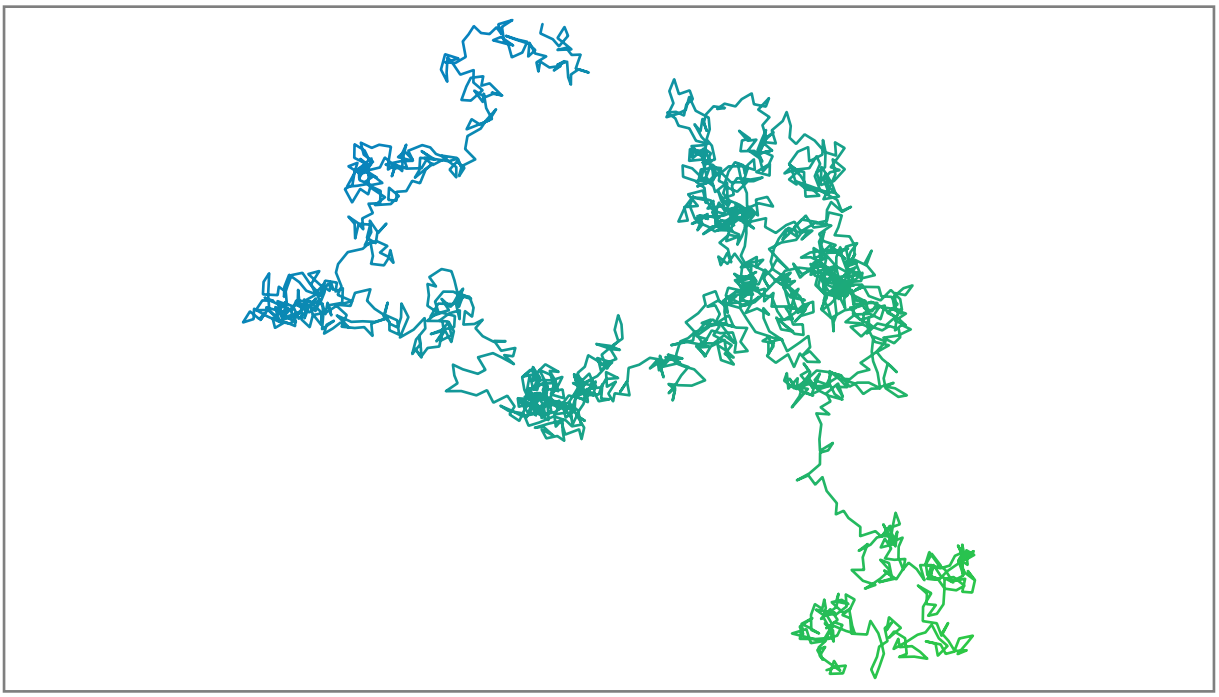
```
#{
  let seed = 9
  let n = 2000
  let step-size = 6
  let curve-stroke = stroke(paint: gradient.linear(blue, green, angle: 45deg), thickness:
1pt, cap: "round", join: "round")

  let rng = gen-rng-f(seed)
  let (rng, va) = uniform-f(rng, low: 0, high: 2*calc.pi, size: n)
  let (rng, vl) = uniform-f(rng, low: 0.5, high: 1.0, size: n)

  let a = 0
  let (x-min, x-max, y-min, y-max) = (0, 0, 0, 0)
  let (x, y) = (0, 0)
  let (dx, dy) = (0, 0)
  let cmd = ()
```

```
  for i in range(n) {
    a += va.at(i)
    dx = calc.cos(a) * vl.at(i) * step-size
    dy = -calc.sin(a) * vl.at(i) * step-size
    x += dx
    y += dy
    x-min = calc.min(x-min, x)
    x-max = calc.max(x-max, x)
    y-min = calc.min(y-min, y)
    y-max = calc.max(y-max, y)
    cmd.push(std.curve.line((dx * 1pt, dy * 1pt), relative: true))
  }
  cmd.insert(0, std.curve.move((-x-min * 1pt, -y-min * 1pt)))

  let width = (x-max - x-min) * 1pt
  let height = (y-max - y-min) * 1pt
  box(
    width: width, height: height,
    place(std.curve(stroke: curve-stroke, ..cmd))
  )
}
```



Here is a common form of the Truchet tiles[6]. Decorate each tile with two quarter-circles connecting the midpoints of adjacent sides. Each such tile has two possible orientations.

```
#{
  let seed = 42
  let unit-size = 10
  let (x-unit, y-unit) = (30, 30)
  let curve-stroke = 2pt + gradient.linear((purple.lighten(50%), 0%), (blue.lighten(50%),
100%), angle: 45deg)

  let c = 4/3*(calc.sqrt(2)-1)
```

[6] https://en.wikipedia.org/wiki/Truchet_tiles

```
  let tot-unit = x-unit * y-unit
  let s = unit-size * 1pt
  let t = s / 2
  let cell(dx, dy, k) = if k == 0 {(
    std.curve.move((dx, dy)),
    std.curve.move((t, 0pt), relative: true),
    std.curve.cubic((0pt, t*c), (-t*(1-c), t), (-t, t), relative: true),
    std.curve.move((2*t, 0pt), relative: true),
    std.curve.cubic((-t*c, 0pt), (-t, t*(1-c)), (-t, t), relative: true)
  )} else {(
    std.curve.move((dx, dy)),
    std.curve.move((t, 0pt), relative: true),
    std.curve.cubic((0pt, t*c), (t*(1-c), t), (t, t), relative: true),
    std.curve.move((-2*t, 0pt), relative: true),
    std.curve.cubic((t*c, 0pt), (t, t*(1-c)), (t, t), relative: true)
  )}

  let rng-f = gen-rng-f(seed)
  let (_, vk) = integers-f(rng-f, high: 2, size: tot-unit)
  let cmd = range(tot-unit).map(i => cell(calc.rem(i, x-unit) * s, calc.floor(i / x-unit)
* s, vk.at(i))).join()

  box(
    width: s * x-unit, height: s * y-unit,
    std.curve(stroke: curve-stroke, ..cmd)
  )
}
```