Proyecto: Parte I
Inteligencia de Enjambre

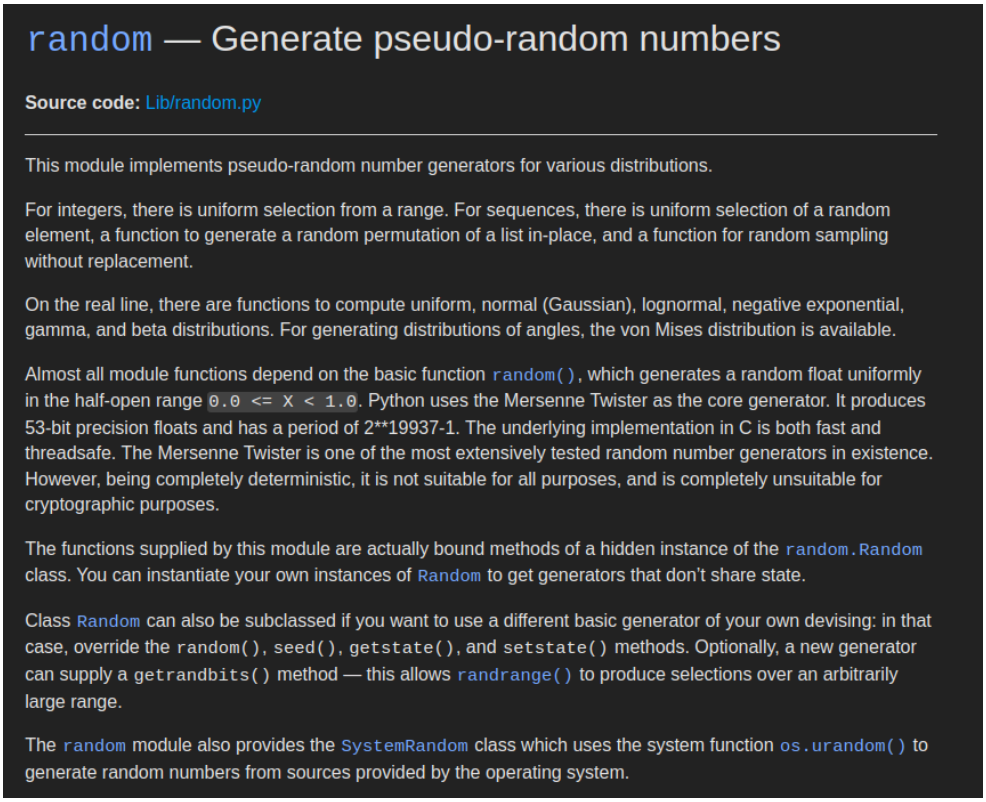
Parte I. Selecciona el lenguaje de programación de tu elección.
En nuestro caso decidimos por utilizar Python.
a) Investiga el método predeterminado para generar números pseudoaleatorios.
El método predeterminado es a través de random, los números son generados como sigue:

```
import random
#Establecemos una semilla
random.seed(88)
#Generamos el número aleatorio
num_aleatorio = random.random()
print("Número ALeatorio:", num_aleatorio)
```

Dicho segmento de código tiene como salida el siguiente resultado:

```
Número ALeatorio: 0.3974888769814575
```

b)Método que emplea:
De acuerdo a la documentación el método empleado es el método de Mersenne Twister:



`random` — Generate pseudo-random numbers

Source code: Lib/random.py

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the half-open range `0.0 <= X < 1.0`. Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the `random.Random` class. You can instantiate your own instances of `Random` to get generators that don't share state.

Class `Random` can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the `random()`, `seed()`, `getstate()`, and `setstate()` methods. Optionally, a new generator can supply a `getrandbits()` method — this allows `randrange()` to produce selections over an arbitrarily large range.

The `random` module also provides the `SystemRandom` class which uses the system function `os.urandom()` to generate random numbers from sources provided by the operating system.

c)Propiedades estadísticas:
En cuanto a las propiedades estadísticas que encontramos referentes al método se encuentran el que posee un periodo de 2^19937 – 1 tal como establece la documentación, esto quiere decir que puede generar hasta 2^19937 – 1 antes de comenzar a realizar instancias repetidas de números, de igual manera encontramos que este método posee una baja autocorrelación, esto quiere decir que números aleatorios generados consecutivamente son prácticamente independientes entre sí a diferencia de otros métodos como el congruencial, de igual manera es un método muy sensible a la elección de una semilla inicial lo que puede resultar en que las propiedades de las cuales se beneficia se vean deterioradas en caso de hacer uso de una semilla poco apropiada.
En cuanto a las desventajas que encontramos podemos destacar el que es un método totalmente determinista y por esta misma razón se excluye de implementaciones de herramientas criptográficas.
c) Investiga bibliotecas externas que permitan generar números pseudoaleatorios con mejores propiedades que el método predeterminado del lenguaje.
Entre las bibliotecas que encontramos podemos enlistar en primer lugar a TensorFlow, en particular esta biblioteca nos ofrece la opción de elegir entre dos métodos, ThreeFry y Phylox:

> Algoritmos
>
> General
>
> Tanto la clase `tf.random.Generator` como las funciones `stateless` admiten el algoritmo Philox (escrito como `"philox"` o `tf.random.Algorithm.PHILOX`) en todos los dispositivos.
>
> Diferentes dispositivos generarán los mismos números enteros, si usan el mismo algoritmo y comienzan desde el mismo estado. También generarán "casi los mismos" números de punto flotante, aunque puede haber pequeñas discrepancias numéricas causadas por las diferentes formas en que los dispositivos realizan el cálculo del punto flotante (por ejemplo, orden de reducción).
>
> dispositivos XLA
>
> En dispositivos controlados por XLA (como TPU y también CPU/GPU cuando XLA está habilitado), también se admite el algoritmo ThreeFry (escrito como `"threefry"` o `tf.random.Algorithm.THREEFRY`). Este algoritmo es rápido en TPU pero lento en CPU/GPU en comparación con Philox.
>
> Consulte el documento 'Números aleatorios paralelos: tan fácil como 1, 2, 3' para obtener más detalles sobre estos algoritmos.

En nuestro caso nos centraremos en el método Philox dado que no poseemos un dispositivo controlado por XLA, procedemos a generar un número aleatorio como sigue:

```
import tensorflow as tf
#Instanciamos la clase generador aleatorio junto con el algoritmo
rng_1 = tf.random.Generator.from_seed(423,alg="philox")
#Elegimos la distribución unimforme
random_float = rng_1.uniform(shape=())
#Generamos un número aleatorio
print("Número Aleatorio:", random_float.numpy())
```

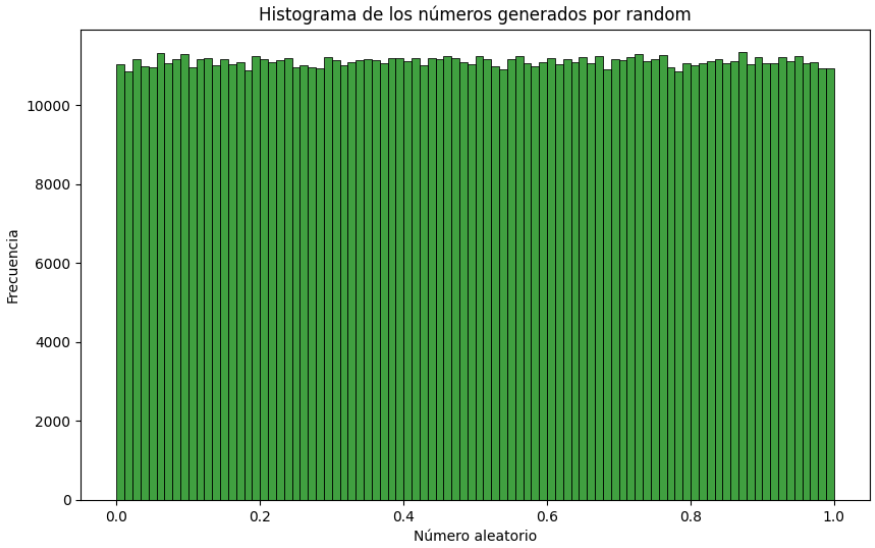
Y obtenemos la siguiente salida:

```
Número Aleatorio: 0.98145294
```

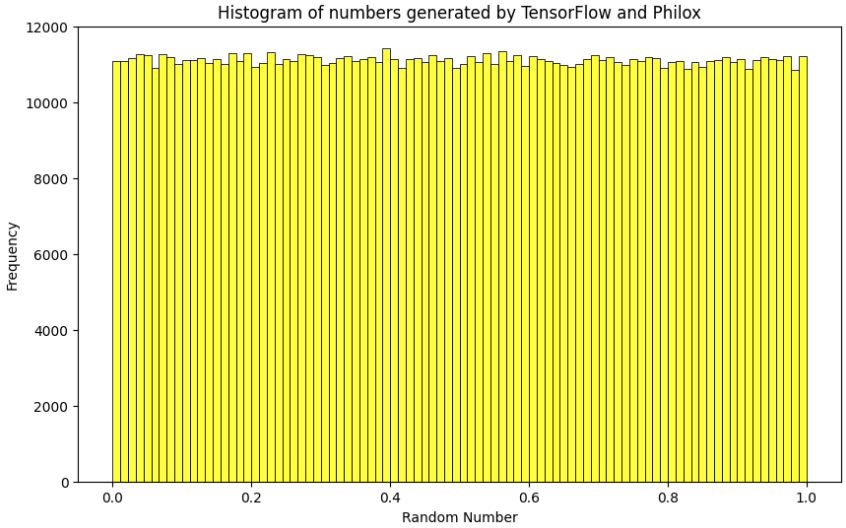
En cuanto a las características estadísticas de este método podemos agregar lo siguiente, tiene un ciclo de aproximadamente 2^64 * 14581110107779764567 * 15183679468541472403 lo cual es un número extremadamente grande, intentamos calcular esa cantidad de números aleatorios en nuestros equipos pero en todas las ocasiones el kernel detenia la tarea al no disponer de memoria suficiente, realizamos lo mismo para el método de Mersenne Twister obteniendo los mismos resultados, otra característica que encontramos fue el hecho de que requiere 2 números primos como argumentos y es extremadamente sensible a que tan bien son elegidos estos argumentos y que también es usado principalmente para entornos paralelizados dado que es el método mas eficiente bajo dicho hardware.

c) Genera 1000000 de números aleatorios y gráficalos para ver su distribución.

Para el caso del método por defecto obtuvimos el siguiente histograma con 90 bins:



Notemos que dicho histograma se asemeja relativamente al de una distribución uniforma, por otra parte respecto a los números generados con TensorFlow obtuvimos el siguiente histograma:



De igual manera se asemeja al de una distribución uniforme como era de esperarse, la diferencia entre las frecuencias de ambas gráficas es realmente poco notoria, sin embargo debemos tener en mente que el método philox como establece la bibliografía es más apropiado para las herramientas del ecosistema de TensorFlow.

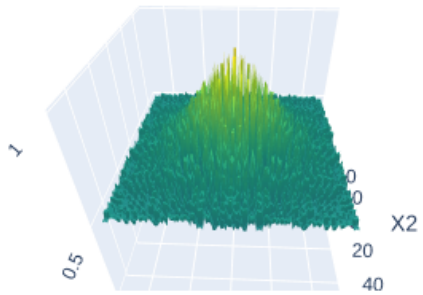
Parte II:

Elegimos las siguientes funciones a graficar, usamos Plotly dado que presenta gráficas interactivas:

- **Schaffer:**
  Esta función se ve dada por la siguiente relación:
  .

$$f(\mathbf{x}) = 0.5 + \frac{\sin^2(x_1^2 - x_2^2) - 0.5}{[1 + 0.001(x_1^2 + x_2^2)]^2}$$

  Decidimos graficarla en [-50,50] X [-50,50] obteniendo los siguientes resultados:

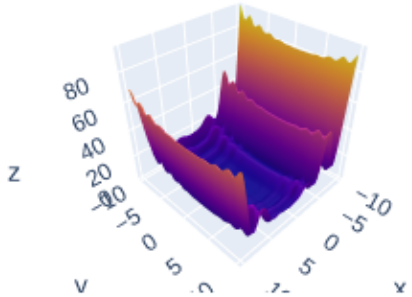


- **Levy:**
  Esta función se ve dada por la siguiente relación:

$$f(\mathbf{x}) = \sin^2(\pi w_1) + \sum_{i=1}^{d-1} (w_i - 1)^2 \left[1 + 10\sin^2(\pi w_i + 1)\right] + (w_d - 1)^2 \left[1 + \sin^2(2\pi w_d)\right], \text{ where}$$

$$w_i = 1 + \frac{x_i - 1}{4}, \text{ for all } i = 1, \ldots, d$$

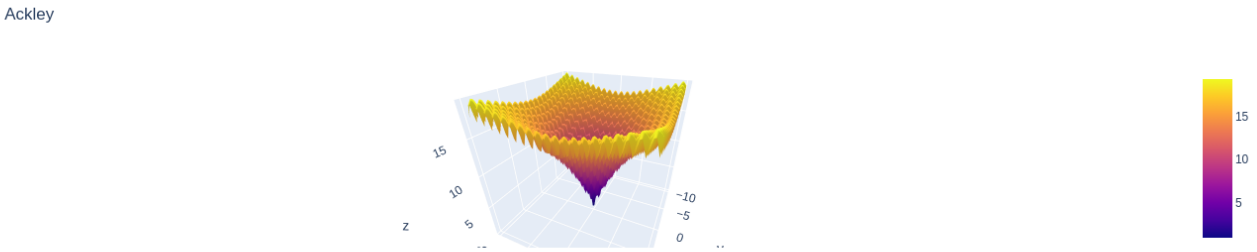
La graficamos en el conjunto [-10,10] X [-10,10] obteniendo los siguientes resultados:



- **Ackley:**
  Esta función se ve dada por la siguiente relación:

$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Al graficarla en el conjunto [-10,10] X [-10,10] obteniendo los siguientes resultados:



.

Referencias Bibliográficas:

- Almlof, T. (2022). Pseudorandom Numbers.
- Marsaglia, G., & Tsang, W. W. (1998). The Monty Python method for generating random variables. *ACM Transactions on Mathematical Software (TOMS)*, *24*(3), 341-350.
- Singh, P., Manure, A., Singh, P., & Manure, A. (2020). Introduction to tensorflow 2.0. *Learn TensorFlow 2.0: Implement Machine Learning and Deep Learning Models with Python*, 1-24.
- Van Rossum, G., & Drake, F. L. (1995). Python library reference.