**Random Number Generation**

Random number generation is a fundamental concept in computer science and is used in a wide range of applications, from cryptography to simulations to gaming. Let us dive into how random number generation works, processes to make it more random, and some common algorithms used for this purpose.

**How Random Number Generation (RNG) Works:**

**Pseudo-random number generators (PRNGs):** Most programming languages provide built-in functions to generate random numbers. These generators are not truly random but produce a sequence of numbers that appears random. They typically start with a seed value and use mathematical formulas to generate subsequent numbers in the sequence.

**True random number generators (TRNGs):** These generators produce truly random numbers based on physical processes, such as atmospheric noise or radioactive decay. TRNGs are often more complex and may require specialized hardware.

**Existing Techniques:**

**Linear Congruential Generator** is one of the oldest and simplest pseudo-random number generators. It generates a sequence of numbers using a linear recurrence equation of the form **Xn = (aXn + c)**, where **aXn** is the nth generated number, **a** is a multiplier, **c** is an increment, and **m** is the modulus. The algorithm is simple and fast. However, has periodic behaviour with a maximum period of m if certain conditions are met and is also prone to poor statistical properties, especially with poorly chosen parameters.

**Mersenne Twister** is a highly regarded pseudo-random number generator known for its long period (**219937 - 1**) and good statistical properties. It uses a twisted generalized feedback shift register to generate numbers. Having a large period ensures a vast number of distinct sequences, it also provides high-quality randomness suitable for most applications. However, it is relatively slower compared to simpler algorithms like LCG.

**Xorshift** is a family of pseudo-random number generators that use bitwise operations (**XOR** and **shift**) to generate numbers. It is known for its simplicity, efficiency and good statistical properties. This makes it useful for applications that require speed to perform while still passing many randomness tests and has simple implementation with few arithmetic operations.

**Cryptographically Secure PRNGs** are pseudo-random number generators designed to withstand cryptographic attacks and provide high-quality randomness suitable for security-sensitive applications. Designed to pass strict randomness tests and resist cryptanalysis. Typically, **CGSPRNGs** are slower than non-cryptographic PRNGs due to increased complexity and require a secure entropy source for seeding.

**Improving Randomness in generation:**

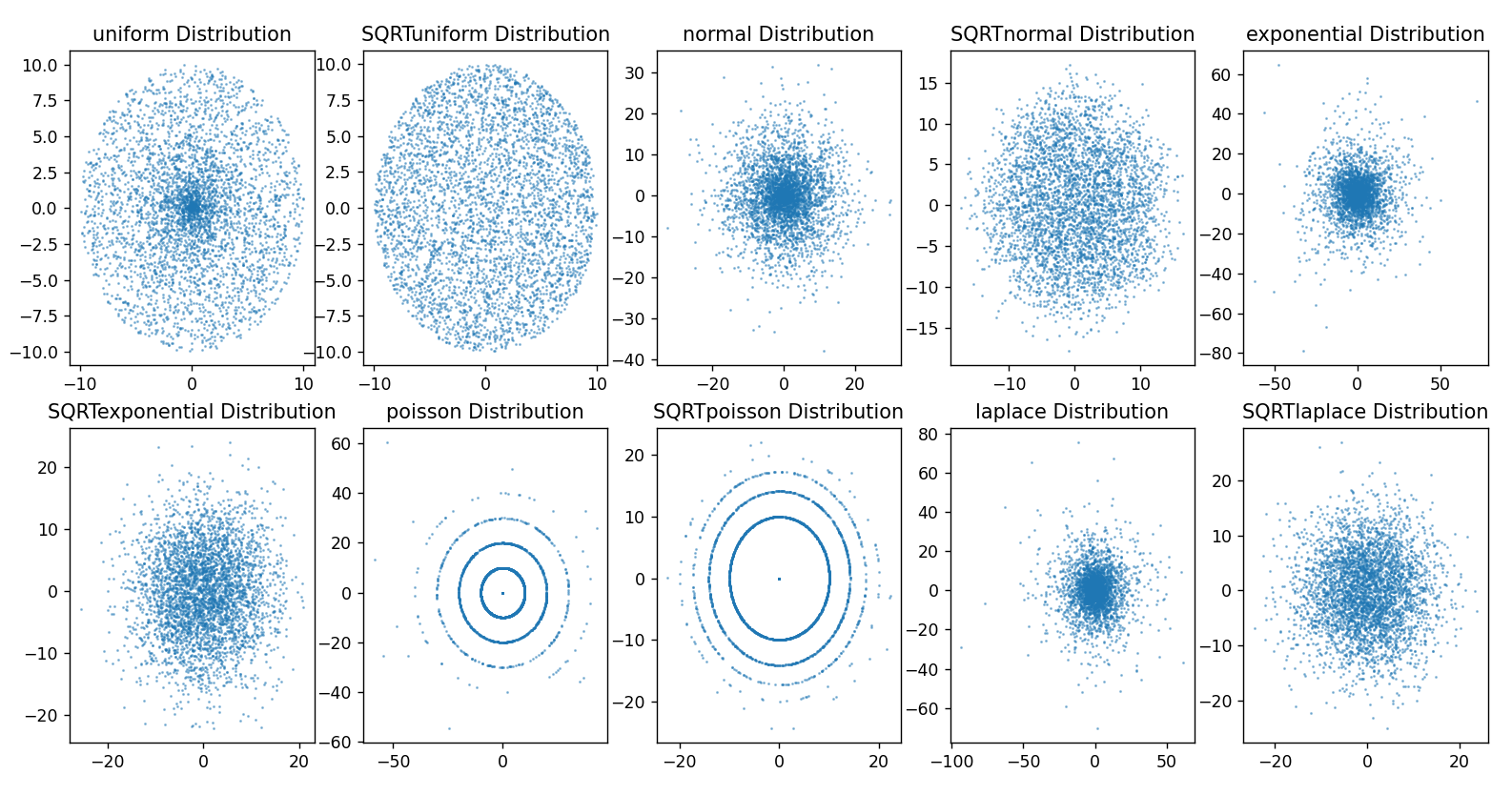
**Seeding:** PRNGs require an initial seed value to start generating the sequence of pseudo-random numbers. Providing a different seed value will result in a different sequence. Therefore, adding multiple seed values to the initial generation of the sequence will increase randomness.

**Entropy:** Increasing entropy in the system can improve randomness. Entropy refers to the unpredictability or disorder of a system. Assessing the quality of entropy is crucial. Entropy estimation techniques, such as entropy measurement tests, evaluate the randomness of collected data to ensure its suitability for generating random numbers.

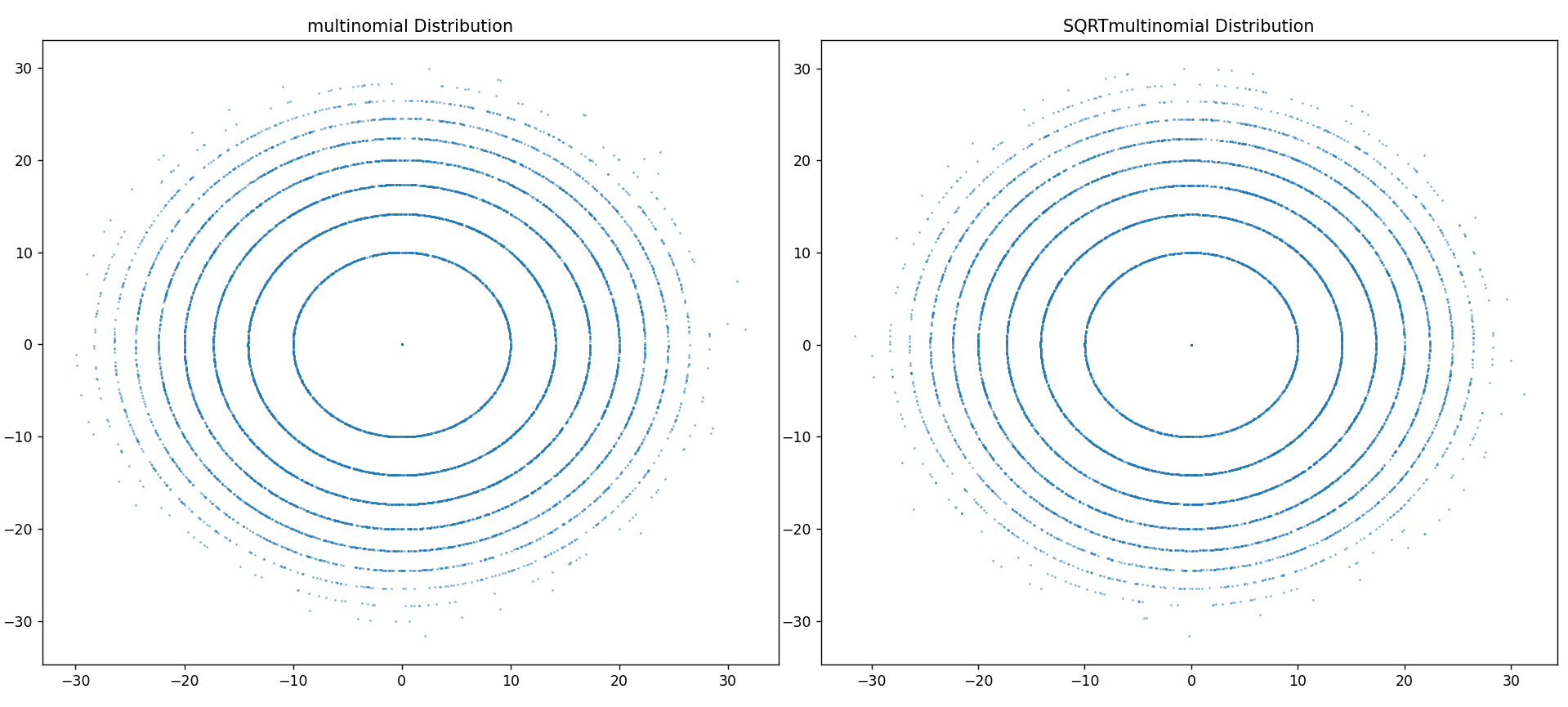
**Randomization Techniques:** Randomization techniques are methods used to introduce randomness into various processes, such as data structures, algorithms, and simulations. They are essential for creating unpredictable and diverse outcomes in computer programs. These techniques include shuffling, random sampling, and Monte Carlo methods.

**Coding**

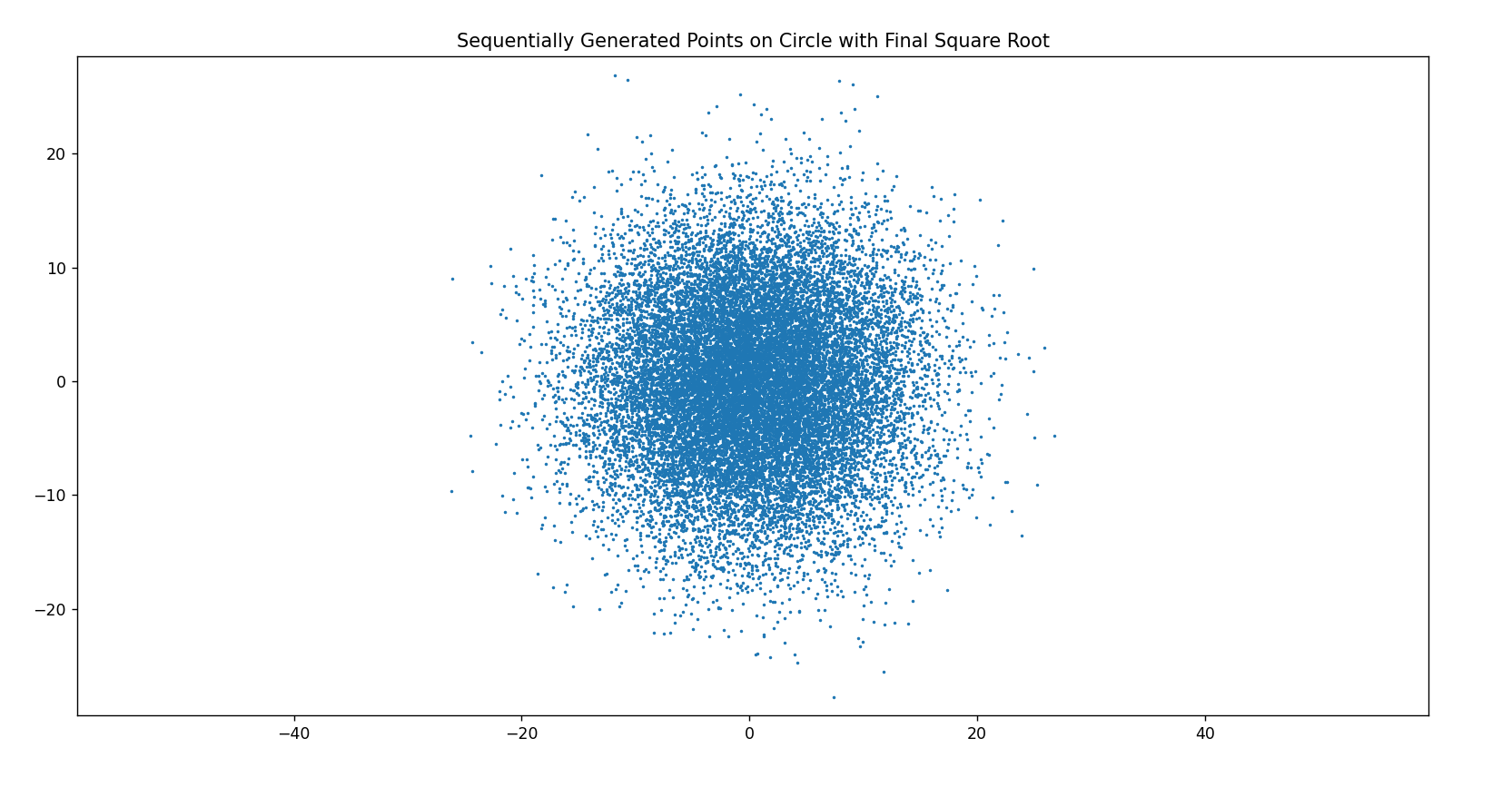
Initially when coding, on the project post about random number generation, there were a few examples of distributions that could be used already given, these were the ones I investigated first. When writing the code, using matplotlib to project my findings into the shell made it extremely easy to visualize the randomness of my generated pools.

The first code generates a series of 4000 points and runs them through 12 different formulae, plotting the outcome graphically. Included techniques; ‘uniform', 'normal', 'exponential', 'poisson', 'laplace' and 'multinomial' distribution, and their square rooted counter parts.

As we can see from the generated graphs, square rooting the distribution effectively increases the distribution amongst the plotted points. However also through doing this it shows that poisson and multinomial distribution are not effective for creating randomness in this situation, generating rings instead of a large spread of data.



Stacking the remaining techniques on top of each other we can increase the entropy of the data, then by adding a further square root, making the data more isentropic. In this iteration of the code, the data pool was increased to 20000 as the figure was bigger so more was visualized.



The final code explores the techniques spoken about in the beginning, plotting six graphs with 5000 points each.

