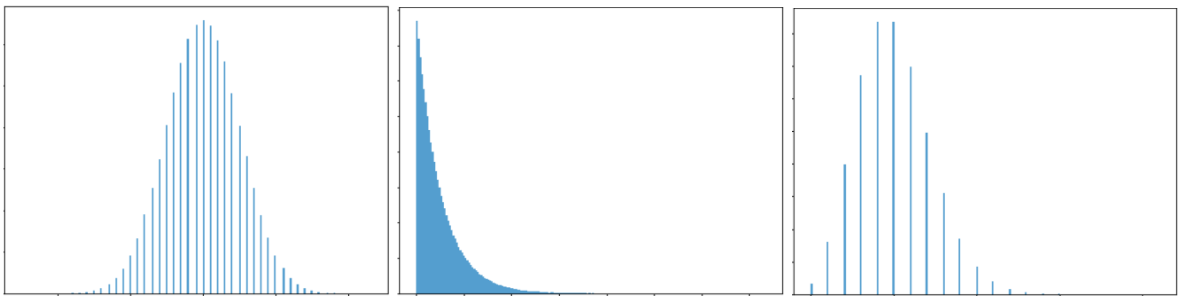
Projet CCA

Neural Networks Performance
for Generating Standard Random Distributions



Samy Horchani (28706765)
Assia Mastor (28602563)

Supervised by
Prof. Turki Lokmane Abbas
Mr. Grenard Aurélien

May 2024
Department of Computer Sciences
Sorbonne University FSI M1 CCA

Contents

Introduction

In the realm of computational science, the generation of random variables following specific probability distributions is a fundamental task with applications spanning various fields, including physics, finance, and machine learning. Our goal is to explore and understand the performance of different methodologies for generating standard random distributions, ranging from traditional library-based approaches to cutting-edge neural network techniques.

We aim to compare the performance of neural network-based approaches with traditional library-based methods, such as NumPy and PyTorch, across a range of standard probability distributions, including the Poisson distribution, non-central chi-squared distribution, and the compound Poisson distribution.

Through a series of experiments, we will investigate the efficiency and accuracy of different methodologies for random variable generation. We will explore techniques such as inverse distribution, rejection sampling, and the Box-Muller transform, evaluating their performance on both CPUs and GPUs. Additionally, we will delve into the realm of generative adversarial networks (GANs) and assess their capability to generate distributions.

To structure our analysis, we will begin by providing a comprehensive explanation of how random variables are generated mathematically. We will cover the theoretical underpinnings of key methods such as inverse transform sampling, which involves using the cumulative distribution function to generate random variables, or also the rejection sampling.

Following this theoretical groundwork, we will implement these methodologies in practical simulations. We will start with traditional approaches using well-established libraries in Python, like NumPy and PyTorch, and in C, focusing on their performance on standard CPUs. This will provide a baseline against which we can measure the newer, more advanced techniques. Next, we will transition to utilizing GPUs, leveraging their parallel processing capabilities to accelerate random variable generation. These libraries are designed to take full advantage of modern hardware architectures, offering significant speedups for many computational tasks.

Finally, we will explore the potential of generative adversarial networks (GANs) to generate random variables. We will investigate whether GANs can be effectively used to produce standard probability distributions, comparing their performance to traditional methods.

1 How to simulate random variables

1.1 What is a random variables ?

Definition 1.1

A real random variable (resp. a random vector) on Ω is a application $X : \Omega \to \mathbb{R}$ (resp. $\Omega \to \mathbb{R}^m$) such that for any open interval I of $\mathbb{R}$ (resp. an open box in $\mathbb{R}^m$) $X^{-1}(I)$ is a event (so in $\mathcal{B}$). [3]

In understanding the concept of random variables and their properties, it is essential to delve into the mechanics of sequential random number generation, which forms the backbone of many probabilistic simulations and statistical analyses.

1.2 Sequential Random Number Generation

To generate a random number, we employ a random number generator (RNG). This function operates deterministically, meaning that when applied to a specific number, it generates the subsequent number in the sequence. This sequence appears to be statistically independent and uniformly distributed, although the RGN is in reality a pseudo-random number generator. The user specifies an initial value called "seed" to initialize the generator.The same seed gives the same sequence. This leads us to the following definition :

Definition 1.2

A (pseudo) random number generator is a structure $\mathcal{G} = (S, s_0, T, U, G)$ where S is a finite set of states, $s_0 \in S$ is the initial state (or seed), the mapping $T : S \longrightarrow S$ is the transition function, U is a finite set of output symbols. [4]

1.3 Requirement for Sequential Random Number Generators

Ideally a pseudo-random number generator would satisfy some requirement such as producing an array of numbers that are uniformly distributed and satisfy any statistical test for randomness. The sequence of numbers generated must have a long period. However, as we will discuss, for practical applications, it is essential that this period of repetition is significantly longer than the total number of pseudo-random numbers potentially used in any given application. Additionally, any correlations among the numbers should be minimal to ensure they don't impact the result of computations. RNGs must operate quickly, relying on a small set of straightforward operations. This need for speed can compromise the quality of the generated randomness so finding the right balance between efficiency and the quality of randomness is crucial. Because generators operate on deterministic algorithms, the numbers generated cannot be entirely uncorrelated and reproducibility of each generation means cannot be considered truly random. An ideal pseudo-random generator should also be portable, meaning it operates consistently across different computers, and capable of being divided into several independent subsequences.

1.4 Linear Congruential Generators (LCG)

1.4.1 How it works

The most widely used is the *Linear Congruential Generators* (**LCG**).

$$X_n = (aX_{n-1} + c) \bmod m \tag{1}$$

where a is the multiplier, m the modulus, c an additive constant that can be set to 0 (when this is the case, this generator is called a *Multiplicative Linear Congruential Generator* (**MLCG**)). The period's length is limited by the modulus size. Generally, the maximum period for a LCG is m. In the case of a MLCG, the period cannot exceed $m - 1$ since $x_n = 0$ is an absorbing state that must be avoided. The parameters (a, c, m) must be chosen carefully for a long period, good uniformity and randomness properties.

1.4.2 Defects

LCGs perform well in most applications, but they are known to exhibit some significant flaws. The primary issue is that the least significant bits of the generated numbers tend to be correlated. Another concern arises from the fact that many commonly used LCGs utilize a modulus m that is a power of 2, as it is efficient and convenient for computer implementation. However, this approach results in highly correlated lower-order bits and long-range correlations for intervals that are powers of 2. To mitigate these issues, it is preferable to use a modulus that is prime rather than a power of 2. Furthermore, enhancing the precision of the generators, such as by employing 64-bit numbers instead of 32-bit ones, can alleviate the impact of these regularities. However, there are practical constraints to this approach. If the modulus exceeds the machine precision, significantly slower multi-precision arithmetic becomes necessary. Hence, in practice, the precision should not exceed the machine precision.

1.4.3 Generalization of LCGs

Multiple Recursive Generatiors (MRG) generalize LCGs with well-chosen parameters by using a recurrence of the form :

$$X_n = (a_1 X_{n-1} + a_2 X_{n-2} + \ldots + a_k X_{n-k} + b) \bmod m \tag{2}$$

Now we can generalize LCGs and MRGs by considering the linear recurrences for vectors with matrix coefficients

$$X_n = (A_1 X_{n-1} + \ldots + A_k X_{n-k}) \bmod m \tag{3}$$

Where $A_1, \ldots, A_k$ are $L \times L$ matrices and each X_n is an L-dimentional vector of elements of $\mathbb{Z}_m$ The recurrence can also be written as a matrix LCG of the form $\mathbf{X_n} = \mathbf{AX_{n-1}}$ **mod m**.

$$\mathbf{A} = \begin{pmatrix} 0 & I & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & I \\ A_k & A_{k-1} & \ldots & A_1 \end{pmatrix} \quad \text{and} \quad \mathbf{X} = \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_L \end{pmatrix}$$

Here a matrix $\mathbf{A}$ of dimension $kL \times kL$ and a vector $\mathbf{X}$ of dimension kL. I is the $L \times L$ identity matrix and this matrix notation applies to the MRG as well, with L = 1.

1.5 Different methods we use to generate basics law

1.5.1 Inverse distribution

One common approch to generating random variables following a specific probability distribution is the inverse distribution method. This method relies on the cumulative distribution function (CDF) of the desired distribution. It works as follows :

1. **Calculate the Cumultative Distribution Function (CDF):** Begin by obtaining the cumulative distribution function (CDF) of the target probability distribution. The CDF represents the cumulative probability of observing a value less than or equal to a given point.

2. **Find the Inverse of the CDF:** Determine the inverse of the CDF function. This involves solving for the value of the random variable that corresponds to a given cumulative probability.

3. **Generate Uniform Random Numbers:** Generate uniformly distributed random numbers between 0 and 1.

4. **Apply the Inverse CDF:** Use the inverse CDF function to transform the uniformly distributed random numbers into numbers distributed according to the desired distribution.

5. **Repeat as Needed:** Repeat the process as necessary to generate a sequence of random numbers following the desired distribution.

The inverse distribution method is particularly useful for distributions with easily invertible CDFs, such as the *exponential* or *normal distributions.* However, it may not always be feasible for distributions with complex or non-invertible CDFs. When this is the case, we use some advanced methods that we will present to you.

1.5.2 Rejection sampling

Rejection sampling is a Monte Carlo method used to generate random samples from a probability distribution that may be difficult to sample directly. It works by sampling from a simpler distribution that encloses the target distribution, and then rejecting samples that fall outside the target distribution.

It works as follows:

1. **Sample from a Proposal Distribution:** Begin by sampling from a proposal distribution that encloses the target distribution. This proposal distribution should be easy to sample from and should enclose the entire support of the target distribution.

2. **Acceptance-Rejection Step:** For each sample drawn from the proposal distribution, accept it with a probability proportional to the ratio of the target distribution to the proposal distribution. This acceptance probability ensures that samples are drawn more frequently from regions where the target distribution is higher.

3. **Repeat Sampling:** Repeat the process until a sufficient number of samples have been accepted. The accepted samples represent random samples drawn from the target distribution.

Rejection sampling is particularly useful when the target distribution is high-dimensional or non-standard, and when other methods like inverse transform sampling are not applicable. However, it can be computationally expensive and inefficient for distributions with low acceptance rates or complex shapes.

1.5.3 Box-Muller transform

The Box-Muller transform is a method used to generate random samples from a standard normal distribution (mean = 0, standard deviation = 1) using random samples from a uniform distribution.

It works as follows:

1. **Generate Uniform Random Numbers:** Begin by generating pairs of independent random numbers U_1 and U_2 that are uniformly distributed between 0 and 1.

2. **Compute Polar Coordinates:** Compute the polar coordinates R and Θ from the generated uniform random numbers using the following equations:
$$R = \sqrt{-2\ln(U_1)}$$
$$\Theta = 2\pi U_2$$

3. **Convert to Cartesian Coordinates:** Convert the polar coordinates (R, Θ) to Cartesian coordinates (X, Y) using the following equations:
$$X = R\cos(\Theta)$$
$$Y = R\sin(\Theta)$$

4. **Generate Normally Distributed Numbers:** The generated numbers X and Y are independent standard normal random variables. Therefore, they can be used as two independent samples from a standard normal distribution.

The Box-Muller transform provides an efficient method for generating normally distributed random numbers without relying on the inverse transform of the normal distribution's cumulative distribution function.

2 Librairies and parallelization of CPU simulation in Python + C

In this chapter, we delve into the utilization of libraries in Python and C for the simulation of statistical distributions. We highlight the performance differences between built-in random number generators and our custom implementations. The libraries employed include NumPy, PyTorch and rand() function in C for simulating various distributions with $n\_sample = 1,000,000$. The performance results are presented in the following tables. All our experiments and time measurements were conducted on the GPU-4 of the PPTI. It has 40 cores and 1152 GB of RAM. For information, on CPU, NumPy takes 0.01455 seconds to generate 1,000,000 uniform random numbers, and PyTorch takes 0.00719 seconds. Using rand() function on C, takes 0.020797 seconds for 1,000,000 uniform random number.

2.1 Using Python

2.1.1 Functions on CPU included in the library

When we compare the columns for NumPy with a 'for' loop and NumPy without a 'for' loop, it is evident that using a 'for' loop externally and calling the function for each individual sample is significantly slower. This is because this approach introduces substantial overhead from repeatedly invoking the function call, which is inherently less efficient due to the interpreted nature of Python. Each function call handle arguments, and process the return values and these operations are relatively costly when executed repeatedly for a large number of samples. For instance, the simulation of a Bernoulli distribution with NumPy with a 'for' loop takes 0.8 seconds, whereas NumPy without a for loop reduces this time significantly to just 0.02 seconds. This disparity underscores the inefficiency of external for loops in Python, and its substantial overhead.

$n\_sample = 1,000,000$ $p = 0.5$ Distribution simulated with built-in functions	Numpy with a 'for' loop	Numpy without a 'for' loop	Pytorch
Bernoulli	0.80000s	0.02087s	0.02651s
Binomial ($n = 100$)	0.81285s	0.08303s	0.22444s
Exponential ($\lambda = 2$)	0.60388s	0.01727s	0.06846s
Poisson rejection ($\mu = 5$)	0.8195s	0.13516s	0.19474s
Chi-squared (d = 10)	0.68824s	0.16115s	0.23700s
Normal	0.85456s	0.05965s	0.01545s

Table 1: Time measurements of python function already included in Python library on CPU

Indeed, everaging NumPy's vectorized operations (i.e., without a 'for' loop) allows the underlying C implementation to perform the computations in a more optimized manner. By performing the computations in bulk, NumPy reduces the number of function calls and minimizes the overhead, leading to much faster execution times.

When comparing the performance of PyTorch and NumPy on a CPU, the results are generally equivalent. For instance, in simulations involving a Bernoulli distribution, both frameworks exhibit similar speeds. While PyTorch is designed for high-performance computing and excels in optimizing and executing operations rapidly, NumPy's implementation can match its performance for certain distributions.

2.1.2 Using our custom generators on CPU

We have also implemented the distributions ourselves from uniform distributions, using *np.random.uniform(0, 1)* for NumPy and *torch.rand* for PyTorch. This approach involves generating random numbers from a uniform distribution and then transforming them to follow the desired statistical distributions.

$n\_sample = 1,000,000$ $p = 0.5$ Distribution simulated	Custom generators using NumPy	Custom generator using PyTorch
Bernoulli ($n = 100$)	0.02893s	0.05091s
Binomiale with Bernoulli ($n = 100$)	1.22517s	1.46297s
Binomiale without Bernoulli	1.87530s	
Exponentielle ($\lambda = 2$)	3.62260s	0.10872s
Poisson rejection ($\mu = 5$)	11.88620s	3.66399s
Chi-squared (d = 10)	0.16001s	1.19594s
Compound Poisson	3.48699s	2.73748s
Noncentral chi-squared	8.63048s	

Table 2: Time measurements of our custom generators in Python on CPU

When comparing the performance of PyTorch and NumPy on CPU using our custom generators, the results are mixed. For the Bernoulli distribution, NumPy is faster with 0.02893 seconds compared to PyTorch's 0.05091 seconds. In the case of the binomial distribution with Bernoulli trials, NumPy completes in 1.22517 seconds while PyTorch takes 1.46297 seconds. For the exponential distribution, PyTorch is significantly faster at 0.10872 seconds compared to NumPy's 3.62260 seconds. The Poisson distribution also shows a notable advantage for PyTorch at 3.66399 seconds versus 11.88620 seconds for NumPy, which can be attributed to the more complex implementation involving rejection sampling in NumPy.

2.2 Using C

To ensure consistency and a fair comparison between random number generators, we used an RNG provided by Prof. Turki, which we then adapted to our code in C++ for the CPU and in CUDA for the GPU. This approach allows us to directly compare the performance of both architectures using exactly the same algorithm. For the CPU version of our RNG, generating 1,000,000 uniform random numbers takes 7.9473 seconds. We then parallelized this process using OpenMP, reducing the time to 0.1127 seconds. We used this parallelized version for the C CPU code. We first generate the uniform random numbers outside of the functions and pass them as parameters. The generation of uniform numbers is, of course, included in the measurement. This approach is used to avoid passing the six state vectors as parameters to our functions, except for the Poisson function using the rejection method, which requires generating i uniforms depending on its result.

$n\_sample = 1,000,000$ $p = 0.5$ Distribution simulated	using built-in rand() function in C	using our RNG in C++
Bernoulli	0.08834s	0.09479s
Binomial with Bernoulli	2.11896s	0.06306s
Binomial without Bernoulli	2.75088s	1.30529s
Exponential ($\lambda = 2$)	0.05505s	0.08671s
Poisson rejection ($\lambda = 5$)	0.266195s	0.01859s
Chi-squared	0.79669s	0.10384s

Table 3: Time measurements of our custom generators using built-in rand() function in C and our RNG in C++

We see that our custom RNG implementations generally outperformed the standard rand function, particularly for more complex distributions like the binomial and Poisson distributions. The optimized algorithms and efficient handling of random samples were key factors in achieving these performance improvements. However, for certain distributions like the exponential distribution, the standard library's implementation proved to be highly efficient, indicating that specific optimizations are necessary to achieve comparable performance.

3 Librairies and parallelization of GPU in Python + C

In this chapter, we analyze the use of libraries in Python and C for simulating statistical distributions on GPUs. We leverage the parallel processing capabilities of GPUs to accelerate the generation of random variables. The libraries used include PyTorch and CUDA, each optimized for high-performance computation on GPUs. By utilizing these libraries, we aim to achieve significant speedups in generating various distributions compared to CPU implementations.

What is particularly interesting here is the comparison between our CPU and GPU RNG. As mentioned earlier, our generator on both CPU and GPU is exactly the same. Regarding the comparison with PyTorch's RNG, after reviewing the documentation, it appears that PyTorch also utilizes CUDA, so we do not expect significantly different results. Generating 1,000,000 uniform random numbers with our CUDA RNG takes 0.001056 seconds and generating 1,000,000 uniform random numbers with torch.rand() on GPU takes 0.000195 seconds.

$n\_sample = 1,000,000$ $p = 0.5$ Distribution simulated	in Python using PyTorch	our RNG in C using CUDA
Bernoulli	0.00054s	0.000703s
Binomial with Bernoulli	0.00052s	0.000582s
Exponential	0.00036	0.000058s
Chi-squared with Box-Muller	0.00568s	0.000099s
Poisson distribution with Rejection sampling	0.01825s	0.000202
Compound Poisson Distribution	0.019852s	

Table 4: Time measurements of our custom generators using PyTorch in Python and our RNG in CUDA

For the Bernoulli and Binomial with Bernoulli trials distributions, PyTorch is slightly faster than our RNG. This indicates that PyTorch has effective optimizations in place, even with the overhead introduced by Python. Despite this overhead, PyTorch's use of CUDA allows it to perform these simpler distributions efficiently. In contrast, for the Exponential, Chi-squared with Box-Muller, and Poisson with Rejection Sampling distributions, our RNG in CUDA significantly outperforms PyTorch. This demonstrates CUDA's high efficiency and superior performance in handling more complex and computationally intensive calculations.

In conclusion, CUDA in C exhibits superior performance for generating distributions that require complex calculations. PyTorch, on the other hand, offers competitive performance for simpler distributions, highlighting the benefits of CUDA-optimized libraries used by PyTorch. This comparison underscores the advantages of direct CUDA implementation for achieving extreme performance requirements.

4 GANs neural networks for random variable generation

In this section, we explore the application of Generative Adversarial Networks (GANs) for generating random variables. GANs are a class of machine learning models that consist of two neural networks: a generator and a discriminator, which are trained simultaneously in an adversarial manner. The generator aims to produce realistic data samples, while the discriminator attempts to distinguish between real and generated samples. Through this process, GANs can learn to generate samples that closely mimic the distribution of the training data.

We will compare the effectiveness of GANs in generating random variables following chi-squared, Poisson, and compound Poisson distributions. By examining different parameter settings, we assess how efficiently GANs can generate these distributions compared to traditional methods on CPU and GPU.

4.1 Chi-squared distribution

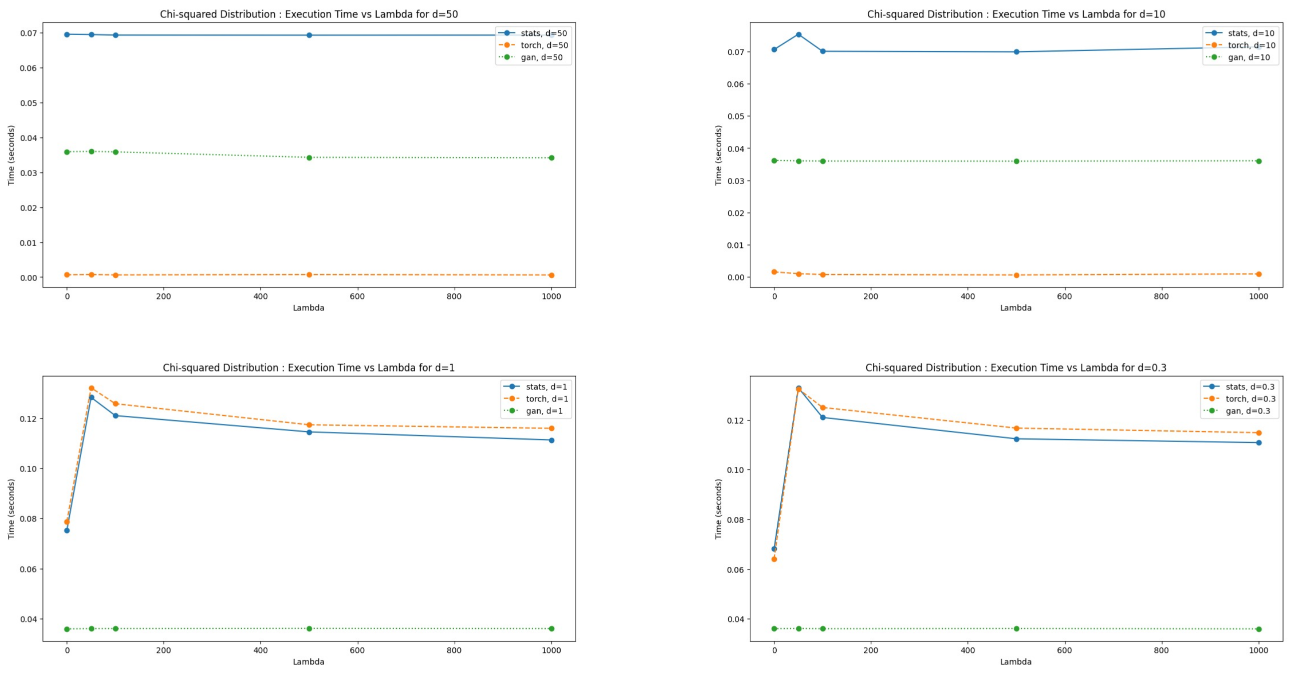


Figure 1: Comparison of Execution Times for Generating Chi-Squared - Distributed Random Variables Using GANs with Varying Degrees of Freedom

We begin here by comparing the different execution times when varying the degrees of freedom of the chi-squared distribution. By analyzing the graphs and the times, we observe that when the degrees of freedom are less than or equal to 1, GANs are faster than Stats, which in turn is faster than PyTorch. This difference in performance can be explained by the fact that PyTorch and Stats are executed on the CPU when the degrees of freedom is a real number, whereas GANs can take advantage of the parallelization capabilities of the GPU. Thus, the result is quite clear: GPUs are faster

When the degrees of freedom exceed 1, PyTorch shows a significant improvement in execution time compared to GANs, which remain faster than Stats. This improvement with PyTorch can be attributed to its optimization for matrix and tensor computations on the GPU, which become increasingly efficient with higher degrees of freedom. For example, for degrees of freedom $d = 50$ and $\lambda = 1000$, the execution times are as follows:

- PyTorch CPU time: 0.06935 seconds
- PyTorch GPU time: 0.00061 seconds
- GAN time: 0.03419 seconds

These results clearly demonstrate the efficiency of PyTorch on the GPU for higher degrees of freedom, highlighting the importance of choosing the right execution platform based on the distribution parameters.

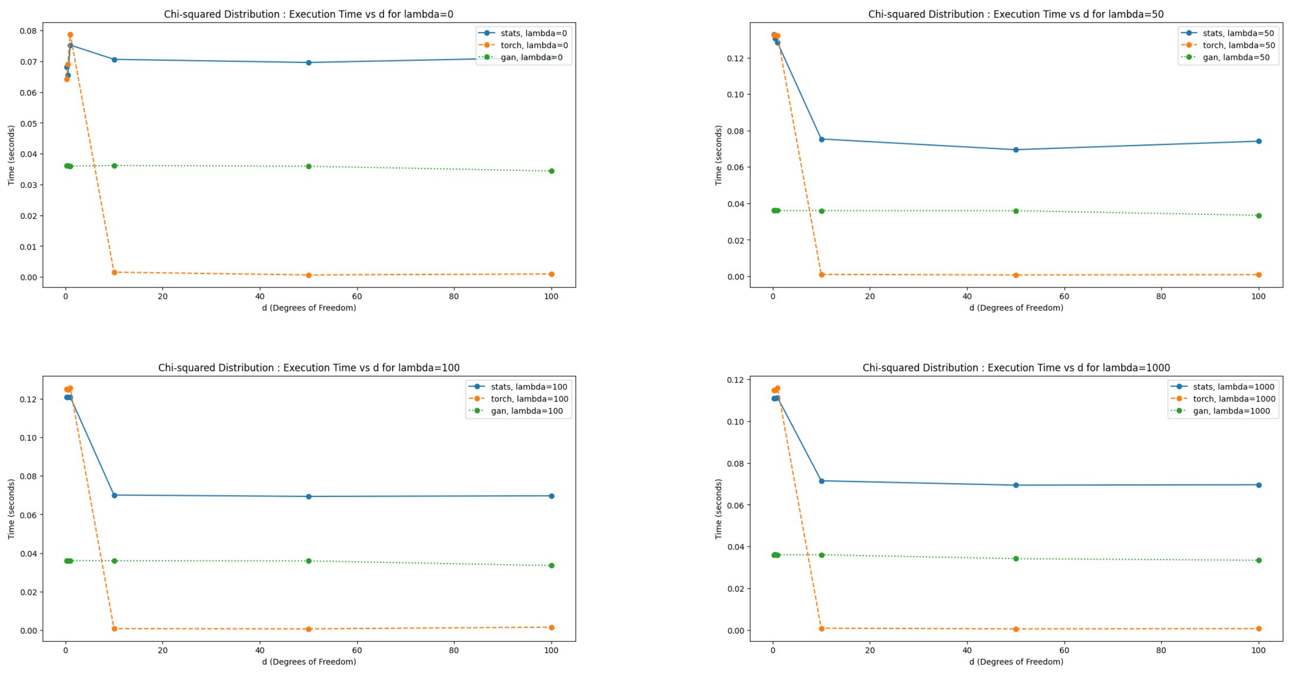


Figure 2: Comparison of Execution Times for Generating Chi-Squared - Distributed Random Variables Using GANs with Varying Lambda

After comparing the execution times by varying the degrees of freedom, we deemed it pertinent to examine whether the value of lambda (λ) had any impact on the execution times. As observed in Figure 3 and across the four graphs, the value of lambda has only a minimal effect on the results, if not negligible. This can be explained by the fact that the computational complexity of generating chi-squared distributed random variables is primarily influenced by the degrees of freedom rather than the scale parameter lambda.

The degrees of freedom determine the number of underlying normal distributions that need to be summed to generate the chi-squared distribution. The lambda parameter merely scales the resultant chi-squared values and does not add significant computational overhead. Therefore, variations in lambda do not substantially alter the execution times.

4.2 Compound Poisson distribution

Now, we turn our attention to the compound Poisson distribution. In this comparison, we examine two different implementations: one derived from normal distributions and the other from uniform distributions. By varying the parameter MU, we measure and compare the execution times for generating these distributions using PyTorch and GANs.

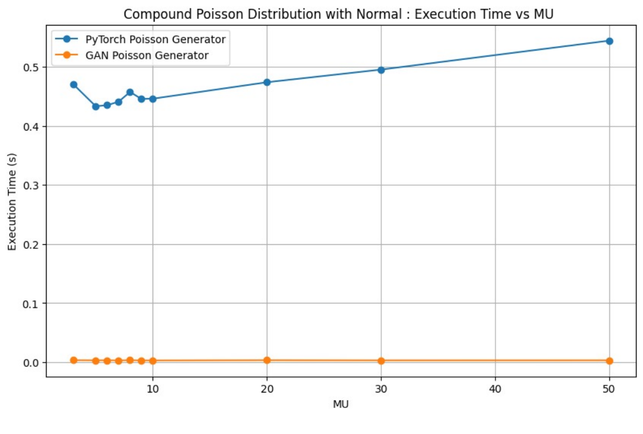


Figure 3: Comparison of Execution Times for Generating Compound Poisson distribution using Normal

In analyzing the graph in Figure 3, it is clearly observed that PyTorch exhibits significantly higher execution times compared to GANs, with execution time increasing steadily as MU increases. The GAN method consistently demonstrates significantly lower execution times compared to the PyTorch method, regardless of the MU value. As MU increases from 10 to 50, the execution time for the PyTorch method rises slightly, while the GAN method maintains a near-constant low execution time. This indicates that GANs are highly efficient in generating compound Poisson distributions with normal components, leveraging their ability to learn and replicate the distribution effectively. The increase in execution time for the PyTorch method with higher MU values suggests that it might be less optimized for handling larger parameters or more complex computations in this context. It is important to note that the PyTorch code used in this comparison was only minimally parallelized, and the results should be interpreted with caution, as we are comparing CPU-based PyTorch with GPU-optimized GANs. This naturally gives GANs a significant advantage due to the inherent parallel processing capabilities of GPUs.

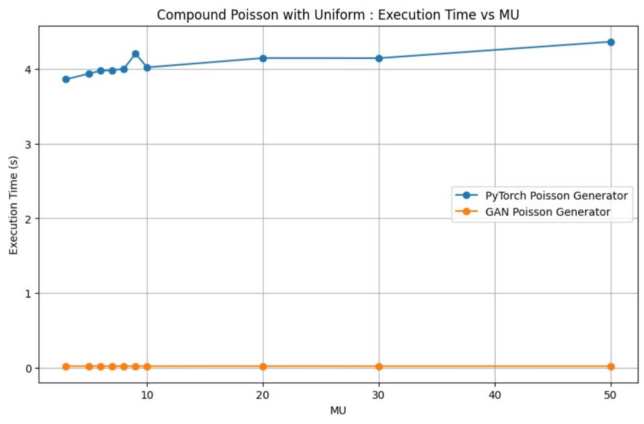


Figure 4: Comparison of Execution Times for Generating
Compound Poisson distribution using Uniform

Similarly, for the compound Poisson distribution with uniform components, GANs are significantly faster than PyTorch, which shows a gradual increase in execution time as MU increases. However, there is a notable difference in execution times compared to the compound Poisson with normal components. While the execution time for PyTorch with normal components hovers around 0.5 seconds, it averages around 4 seconds for the uniform components. This significant increase in execution time for uniform components could be due to the additional computational complexity involved in handling uniform distributions in PyTorch. The uniform distribution may require more intricate processing steps, leading to longer execution times, whereas GANs, with their parallel processing capabilities, manage to handle this complexity more efficiently.

4.3 Poisson distribution

Finally, we will analyze the Poisson distribution and compare the performance of Stats and PyTorch with two different types of GANs: one using a Poisson generator and the other using a uniform generator. The Poisson GAN is designed to directly generate samples from the Poisson distribution, potentially offering high efficiency and accuracy. In contrast, the uniform GAN generates samples from a uniform distribution and then transforms these samples into Poisson-distributed values, which may introduce additional computational steps.

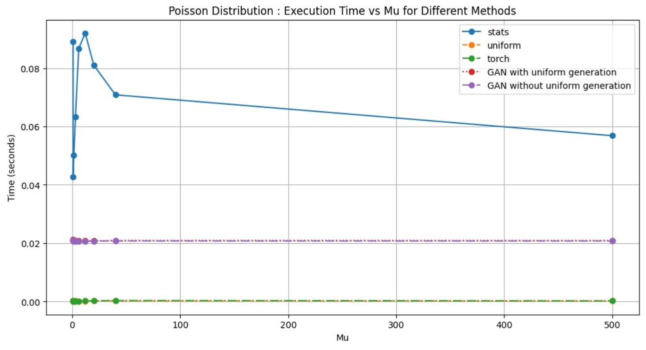


Figure 5: Comparison of Execution Times for Generating
Poisson distribution using various methods

In Figure 5, we observe that PyTorch is the fastest method for generating Poisson-distributed samples. Both

GAN methods, whether using a Poisson generator or a uniform generator, are slower, with execution times around 0.2 seconds. This is because GANs, although powerful in generating complex distributions, involve more computational overhead during the sampling process. The Poisson GAN directly generates samples from the Poisson distribution, while the uniform GAN first generates uniform samples and then transforms them into Poisson-distributed values. Despite this additional step, the performance of the two GAN methods is quite similar, making it difficult to determine which is more efficient based on the available data.

Stats, on the other hand, shows a gradual decrease in execution time as MU increases. This trend could be due to the statistical libraries optimizing their performance for larger parameter values, where repetitive calculations become more streamlined.

Overall, while PyTorch excels in speed due to its highly optimized framework, the GAN methods offer consistent performance, and Stats shows improvement with higher MU values.

Conclusion

In this project, we explored various methodologies for generating random variables following specific probability distributions, evaluating their performance on both CPU and GPU platforms. Our primary focus was on comparing traditional methods using libraries like NumPy and PyTorch with modern neural network-based approaches, specifically Generative Adversarial Networks (GANs).

Our findings indicate that traditional methods like NumPy and PyTorch are highly effective for generating random variables, particularly for simpler distributions such as Bernoulli and Binomial. PyTorch generally outperformed NumPy due to its optimized handling of tensor operations and parallel processing capabilities. However, when we implemented custom random number generators in C on CPU, we observed significant performance improvements, especially for more complex distributions. Using parallelization techniques such as OpenMP further enhanced these improvements.

Utilizing GPU acceleration, both PyTorch and our custom CUDA implementations showed substantial performance gains over CPU implementations. The speed of generating random variables was markedly increased, demonstrating the effectiveness of leveraging parallel processing capabilities. Notably, our custom CUDA implementations outperformed PyTorch for some complex and computationally intensive tasks.

When exploring GANs for generating random variables, we found that GANs were highly efficient in generating distributions such as the chi-squared and compound Poisson distributions. However, PyTorch emerged as the fastest method overall, surpassing GAN performance. The direct generation approach of the Poisson GAN and the transformation approach of the uniform GAN resulted in similar performance, albeit with higher execution times than PyTorch.

Overall, this project highlights the strengths and weaknesses of various methods for generating random variables. While traditional methods like PyTorch and NumPy are effective and efficient, particularly when optimized with parallel processing, GANs offer a powerful alternative for complex distributions, leveraging their ability to learn and replicate intricate patterns. The choice of method depends on the specific requirements of the task, such as the complexity of the distribution and the available computational resources.

References

[1] Schmidt Bertil, Gonzalez-Dominguez Jorge, Hundt Christian, Schlarb Moritz, and Bertil Schmidt. *Parallel programming : concepts and practice / Bertil Schmidt,... Jorge Gonzalez-Dominguez,... Christian Hundt,... Moritz Schlarb,...* Elsevier MK, Morgan Kaufmann Publishers, an imprint of Elsevier, Cambridge (Mass.), 2018.

[2] Gülin Kaşkara. Parallel random number generators. 2003. Department of Computer Engineering, Middle East Technical University.

[3] Raphaël Krikorian. Simulation de phénomène aléatoire - LM 364 : Partie 1, 2014.

[4] Pierre L'Ecuyer. Random number generation. In Jerry Banks, editor, *Handbook on Simulation*, chapter 4, pages 35–70. Wiley, New York, 1998.

[5] Simon J. A. Malham and Anke Wiese. Chi-square simulation of the cir process and the heston model, 2012.

[6] NumPy Contributors. *NumPy Documentation.* NumPy, 2024.

[7] NVIDIA. curand library, 2023.

[8] Gilles Pagès. *Numerical Probability: An Introduction with Applications to Finance*. Springer Nature, Netherlands, 2018.

[9] Simon J.D. Prince. *Understanding Deep Learning.* MIT Press Ltd, 2023.

[10] PyTorch Contributors. *PyTorch Documentation.* PyTorch, 2024.