The Data Science of Randomness

Course Project Topics and Specifications

Department of Computer Science
Florida State University

Academic Year 2025–2026

Course: The Data Science of Randomness **Credit Hours:** 3 Semester Hours
Level: Upper-Level Undergraduate / Beginning Graduate
Project Weight: Primary basis for course grade

Overview: This document describes substantial research and implementation projects for the course. Each project requires a working implementation, rigorous experimental analysis, and a written technical report and oral presentation at the end of the term.

General Grading Expectations

All projects will be evaluated on five equally weighted dimensions:

Component	Weight	Description
Implementation	20%	Correctness, code quality, documentation, reproducibility
Experimental Design	10%	Rigor of experiments, choice of baselines, parameter sweeps
Analysis & Results	20%	Depth of analysis, visualization quality, statistical validity
Written Report	25%	Clarity, organization, related work, conclusions
Oral Presentation	25%	20-minute final presentation with live demo

All code must be submitted in a **reproducible repository** (GitHub or equivalent) with a `README.md` that contains installation instructions and commands to reproduce every figure and table in the report.

Contents

1. Statistical Testing and Validation

1.1. Implementation and Comparative Analysis of the NIST SP 800-22 Test Suite

Difficulty: Intermediate **Primary Tools:** Python, C (optional), NumPy, SciPy **Recommended Background:** Probability theory, hypothesis testing

Overview: NIST Special Publication 800-22 defines 15 statistical tests for evaluating the quality of random and pseudorandom number generators for cryptographic applications. In this project, you will implement all 15 tests from mathematical first principles in Python, validate your results against the official NIST C implementation on identical bit streams, and conduct a comparative study across at least six generators spanning the quality spectrum from LCG to ChaCha20.

Objectives:

- Implement each of the 15 NIST tests from the mathematical specification, *not* by wrapping an existing library
- Validate each implementation against the NIST reference C code on the provided example data
- Design a multi-sequence testing framework (at least 100 sequences per generator) with proportion and uniformity analysis
- Apply your suite to at least six generators: LCG, MT19937, PCG64, xoshiro256**, Philox4x32, and ChaCha20-CSPRNG
- Identify which tests catch which generators and explain the mathematical reason for each failure
- Investigate the statistical power of each test as a function of sequence length

Deliverables:

- Complete Python implementation of all 15 tests with unit tests for each
- Comparative results table across all six generators
- Statistical power curves for at least five tests
- 15–20 page technical report with full mathematical derivations of at least five tests

Extension (for graduate students): Implement the second-level uniformity test for p-value distributions and compare your empirical false-positive rates to the theoretical predictions at $\alpha = 0.01$.

1.2. Head-to-Head Comparison: NIST SP 800-22, TestU01 BigCrush, and Dieharder

Difficulty: Intermediate–Advanced **Primary Tools:** Python, C, R or Julia for visualization **Recommended Background:** Statistical testing, C programming

Overview: The three major statistical test batteries — NIST SP 800-22, TestU01 (SmallCrush, Crush, BigCrush), and Dieharder — are frequently cited in the PRNG literature but rarely compared systematically. This project conducts a rigorous empirical study of which defects each suite detects, at what sequence length, and with what false-positive and false-negative rates.

Objectives:

- Install and configure all three test suites; document build procedures for reproducibility
- Select at least eight generators with known properties across the quality spectrum

- Design experiments to measure: detection rate, false positive rate, sequence length needed for detection, and overlap between suites
- For generators that fail one suite but pass another, identify the specific tests responsible and explain mathematically why the failure is suite-specific
- Produce a definitive comparison matrix: generator × suite × test outcome
- Analyze computational cost (wall time, memory) of each suite

Deliverables:

- Reproducible scripts running all three suites on all eight generators
- Comprehensive comparison matrix with p-value heatmaps
- Detection power curves as a function of sequence length for each suite
- 18–22 page report with a recommendation guide: when to use which suite

1.3. Design and Validation of a Novel Statistical Test Battery

Difficulty: Advanced **Primary Tools:** Python, SciPy, matplotlib **Recommended Background:** Statistical theory, information theory

Overview: You will design, implement, and rigorously validate a battery of at least eight original statistical tests for random number generators, motivated by gaps in existing suites. Tests must be grounded in statistical theory, have analytically derived null distributions, and be validated by simulation under H_0 before being applied to real generators.

Objectives:

- Survey existing test batteries to identify structural gaps (e.g., tests for high-dimensional uniformity, inter-stream independence, or specific application domains)
- Design at least eight tests with: null hypothesis, test statistic, null distribution (analytically derived or justified by simulation), and decision rule
- Validate each test: confirm nominal false-positive rate under a true random source (hardware RNG or AES-CTR)
- Measure statistical power of each test against at least four known-bad generators
- Compare your battery's sensitivity to NIST SP 800-22 on the same generators

Deliverables:

- Complete Python implementation with full documentation
- Null distribution validation report for each test
- Power comparison table vs. NIST SP 800-22
- 18–25 page report including test derivations and a discussion of design tradeoffs

2. Generator Implementation and Analysis

2.1. Comprehensive Implementation and Analysis of the PCG Family

Difficulty: Intermediate **Primary Tools:** Python (from scratch), C for benchmarking **Recommended Background:** Modular arithmetic, bit manipulation

Overview: The PCG (Permuted Congruential Generator) family, designed by Melissa O'Neill (2014), represents the state of the art in fast non-cryptographic PRNGs. You will implement PCG32, PCG64, PCG-XSH-RR, PCG-RXS-M-XS, and the PCG128 variant from the mathematical specification; analyze the permutation output functions; benchmark throughput; and rigorously test statistical quality.

Objectives:

- Implement at least four PCG variants from O'Neill's technical report in Python without using the reference C library
- Validate output against the official C reference implementation on identical seeds
- Derive the period, equidistribution properties, and spectral test scores analytically for each variant
- Benchmark throughput against MT19937, xoshiro256**, and Philox4x32
- Run full TestU01 BigCrush on each variant; analyze any failures in the context of the permutation function design
- Implement the PCG "streams" feature; test inter-stream independence using correlation tests

Deliverables:

- Python implementations with validation test suite
- Throughput benchmarks with statistical confidence intervals
- BigCrush results for all variants
- 15–20 page report including analysis of the output permutation design choices

2.2. Reverse-Engineering a Black-Box PRNG from Output Samples

Difficulty: Advanced **Primary Tools:** Python, NumPy, SciPy, Z3 solver (optional) **Recommended Background:** Linear algebra, cryptanalysis basics

Overview: Given a stream of output from an unknown PRNG (drawn from a fixed set of candidates), you will design and implement a generator identification and parameter recovery system using statistical fingerprinting, spectral analysis, and algebraic attacks. This project bridges statistical analysis and cryptanalytic thinking.

Objectives:

- Build a fingerprinting database for at least ten generators using distinctive statistical signatures (autocorrelation profile, spectral peaks, lattice structure, birthday-spacing distribution)
- Implement an LCG parameter recovery attack: recover (a, c, m) from as few outputs as possible; characterize the minimum number of outputs required
- Demonstrate MT19937 state recovery from 624 consecutive 32-bit outputs; implement the untemper function
- For truncated outputs (fewer bits per call), characterize how truncation depth affects recovery difficulty
- Design an identification pipeline: given an unknown bitstream, rank the candidate generators by likelihood
- Evaluate identification accuracy as a function of sample size across all ten generators

Deliverables:

- Working generator identification system with documented API
- LCG and MT19937 attack implementations with step-by-step documentation
- Identification accuracy matrix (generator × sample size)
- 20–25 page report including a discussion of what makes a generator "identifiable"

2.3. Spectral Test Theory, Implementation, and Empirical Validation

Difficulty: Intermediate–Advanced **Primary Tools:** Python, NumPy (FFT), matplotlib 3D **Recommended Background:** Linear algebra, number theory

Overview: The spectral test, described in Knuth's *The Art of Computer Programming* Vol. 2, measures the lattice structure of LCGs by computing the shortest vector in the dual lattice. You will implement the spectral test in 2D through 6D, reproduce Knuth's lattice structure visualizations, apply the test to a wide range of LCG parameters, and develop an interactive visualization tool for the lattice structure.

Objectives:

- Implement the spectral test for LCGs in dimensions 2 through 6 using the Lenstra–Lenstra–Lovász (LLL) basis reduction algorithm or Knuth's direct method
- Reproduce Knuth's "good" and "bad" LCG lattice visualizations in 2D and 3D using matplotlib
- Analyze at least 50 LCG parameter sets; rank them by spectral score and correlate with TestU01 results
- Apply the spectral test to non-LCG generators (PCG, xoshiro) and explain the results
- Build an interactive 3D visualization tool for the lattice structure
- Derive theoretically the relationship between spectral score and the probability of k-dimensional equidistribution failure

Deliverables:

- Python implementation of the spectral test with LLL reduction
- Interactive visualization tool (Plotly or matplotlib with widgets)
- Parameter analysis database for 50+ LCG configurations
- 15–20 page report reproducing and extending Knuth's analysis

3. Parallel and GPU Computing

3.1. GPU-Accelerated Monte Carlo Simulation with Counter-Based RNGs

Difficulty: Advanced **Primary Tools:** CUDA C/C++, Python (CuPy or PyTorch), Random123 **Recommended Background:** GPU programming, Monte Carlo methods

Overview: Counter-based RNGs (CBRNGs) such as Philox and Threefry are designed for massively parallel generation. You will implement a complete GPU-accelerated Monte Carlo simulation framework using Random123, demonstrate the parallelism and reproducibility properties of CBRNGs, and benchmark against cuRAND and CPU-based approaches on three distinct application problems.

Objectives:

- Implement a CUDA kernel framework using Random123 (Philox4x32 and Threefry4x32) that supports both key-splitting and counter-partitioning parallelism strategies
- Apply the framework to three Monte Carlo problems: (1) high-dimensional integration, (2) European option pricing under Black-Scholes, (3) a particle transport simulation
- Benchmark throughput (samples/second) vs. cuRAND XORWOW, cuRAND MT, and CPU MT19937 across grid sizes from 2^{10} to 2^{28} threads
- Demonstrate and measure reproducibility: identical results across different thread counts, block sizes, and GPU models
- Implement and test both key-splitting and counter-partitioning; compare their inter-stream independence using statistical tests
- Analyze variance of Monte Carlo estimates as a function of sample count and compare to theoretical $O(1/\sqrt{N})$ convergence

Deliverables:

- Complete CUDA framework with Python wrapper
- Benchmark results across all configurations with throughput plots
- Reproducibility demonstration across GPU configurations
- 18–22 page report with CUDA kernel documentation and performance analysis

3.2. Parallel Stream Independence: Theory, Testing, and Failure Modes

Difficulty: Intermediate–Advanced **Primary Tools:** Python, NumPy, multiprocessing, Random123 **Recommended Background:** Statistical testing, parallel computing

Overview: When a simulation generates thousands of parallel random streams, the *independence* of those streams is as important as the quality of individual streams. You will design and implement a comprehensive inter-stream independence testing framework, apply it to five parallelization strategies (leapfrogging, block-splitting, key-splitting, subsequence, and per-thread seeding), and characterize the failure modes of each strategy.

Objectives:

- Implement all five parallelization strategies for MT19937 and Philox4x32
- Design inter-stream independence tests: cross-correlation, birthday-spacing across streams, and a joint uniformity test in $2k$-dimensional space for pairs of streams
- Run tests on 100, 1 000, and 10 000 parallel streams; characterize how failure probability scales with stream count
- Demonstrate a concrete simulation failure caused by correlated streams (e.g., biased Monte Carlo estimate from correlated leapfrogged MT streams)
- Characterize the minimum safe block length for block-splitting as a function of generator period
- Compare results to theoretical predictions from the Parallel Mersenne Twister literature

Deliverables:

- Inter-stream testing framework with documented API
- Failure characterization study across all five strategies
- Concrete simulation failure demonstration with quantified bias
- 15–20 page report with recommendations for each parallelism strategy

4. Cryptographic Randomness

4.1. Entropy Harvesting, Estimation, and Accumulator Design

Difficulty: Advanced **Primary Tools:** Python, C (for hardware access), Linux `/dev/` interfaces **Recommended Background:** Information theory, operating systems basics

Overview: The security of all CSPRNGs depends on the quality of the entropy that seeds them. You will build a complete entropy harvesting and estimation system, implement the NIST SP 800-90B entropy estimation suite, and conduct a comparative study of entropy sources available on a modern Linux system.

Objectives:

- Harvest entropy from at least six sources: keystroke timing, mouse movement, disk I/O timing, network packet arrival, CPU jitter (RDTSC), and `/dev/urandom`
- Implement all ten entropy estimators from NIST SP 800-90B: most-common-value, collision, Markov, compression, t-tuple, LRS, multi-MMC, lag predictor, MultiMCW, and LZ78Y
- Apply all estimators to all six entropy sources; compare min-entropy estimates across estimators for each source
- Implement a simplified Fortuna entropy accumulator: pool management, reseeding schedule, and generator
- Demonstrate a low-entropy attack: seed a CSPRNG with only 20 bits of entropy; show how quickly exhaustive search recovers the state
- Measure how quickly each entropy source saturates to full entropy after a reboot

Deliverables:

- Complete entropy harvesting and estimation system
- SP 800-90B compliance analysis for each entropy source
- Fortuna accumulator implementation with reseeding demonstration
- 20–25 page report including theoretical discussion of entropy estimation bias

4.2. The Dual_EC_DRBG Backdoor: Implementation, Analysis, and Demonstration

Difficulty: Advanced **Primary Tools:** Python, SageMath (elliptic curve arithmetic) **Recommended Background:** Elliptic curve cryptography, number theory

Overview: Dual_EC_DRBG, standardized by NIST in SP 800-90A (2006), was later revealed to contain an NSA-designed backdoor exploiting a secret relationship between its two elliptic curve points P and Q. This project implements Dual_EC_DRBG, derives the backdoor mathematically, demonstrates a concrete prediction attack, and quantifies the output bias introduced by the truncation step.

Objectives:

- Implement Dual_EC_DRBG over the NIST P-256 curve using SageMath
- Derive mathematically how knowledge of e where $Q = eP$ allows prediction of all future outputs from 32 bytes of observed output

- Implement the Shumow–Ferguson prediction attack; demonstrate complete state recovery from observed output
- Generate "honest" P, Q pairs (via nothing-up-my-sleeve construction) and show the backdoor disappears
- Analyze the output bias introduced by the 16-byte truncation; apply NIST SP 800-22 to detect any bias
- Compare the throughput and statistical quality of Dual_EC_DRBG to Hash-DRBG and HMAC-DRBG

Deliverables:

- Working Dual_EC_DRBG implementation with backdoor demonstration
- Step-by-step mathematical derivation of the prediction attack
- Statistical comparison with Hash-DRBG and HMAC-DRBG
- 20–25 page report discussing the cryptographic policy implications

4.3. Password Hashing: Randomness, Salting, and GPU Cracking Resistance

Difficulty: Intermediate **Primary Tools:** Python, hashlib, passlib, Hashcat (for benchmarking) **Recommended Background:** Cryptographic hashing, basic security

Overview: Password hashing is one of the most practically important applications of cryptographic randomness. You will implement PBKDF2, bcrypt, scrypt, and Argon2 from specification (or thin wrappers with full parameter control), analyze the role of the random salt in each, and conduct a systematic study of cracking resistance as a function of work factor and hardware.

Objectives:

- Implement or deeply instrument PBKDF2-HMAC-SHA256, bcrypt, scrypt, and Argon2id; expose all tuning parameters
- Demonstrate a rainbow table attack on unsalted MD5 hashes; measure speedup vs. brute force
- Quantify the entropy contribution of salts: derive analytically and verify empirically the expected number of hashes required with and without a salt for a given password distribution
- Benchmark throughput on CPU and GPU (using Hashcat) for all four schemes across a range of work factors
- Develop a work-factor selection guide: given a target cracking cost (in GPU-hours), derive the minimum work factor for each scheme
- Analyze the Argon2 memory-hardness parameter; derive how it affects GPU parallelism

Deliverables:

- Instrumented implementations of all four password hashing schemes
- Rainbow table attack demonstration on unsalted MD5
- Work-factor selection guide with supporting data
- 15–20 page report including a threat model and deployment recommendation

5. Monte Carlo Methods and Applications

5.1. Variance Reduction Techniques: A Systematic Empirical Study

Difficulty: Intermediate **Primary Tools:** Python, NumPy, SciPy, matplotlib **Recommended Background:** Probability theory, calculus, basic statistics

Overview: Variance reduction techniques (VRTs) can reduce the sample count required for a given Monte Carlo accuracy by orders of magnitude. You will implement and rigorously compare five VRTs — crude Monte Carlo, antithetic variates, control variates, importance sampling, and stratified sampling — across at least five target integrands of varying smoothness and dimensionality.

Objectives:

- Implement all five VRTs for general integrands; expose all tuning parameters (control variate coefficient, importance distribution, stratum count)
- Select five test integrands: a smooth low-dimensional function, a discontinuous function, a high-dimensional Gaussian, a rare-event probability, and a financial pricing problem
- For each integrand and VRT, measure: variance, bias, mean-squared error, and wall-clock time as a function of sample count N
- Verify theoretical convergence rates ($O(1/N)$ for crude MC, $O(1/N^{2/d})$ for stratified) empirically
- Study the interaction of VRTs with RNG quality: compare results using MT19937, PCG64, and a Sobol quasi-Monte Carlo sequence
- Develop a decision guide: for a given integrand type, which VRT is preferred and why?

Deliverables:

- General VRT framework with API documentation
- Variance and MSE tables across all integrand/VRT combinations
- Convergence rate plots with theoretical overlays
- 15–20 page report with a practical VRT selection guide

5.2. Quasi-Monte Carlo vs. Pseudo-Monte Carlo: A High-Dimensional Study

Difficulty: Intermediate–Advanced **Primary Tools:** Python, SciPy (QMC module), matplotlib **Recommended Background:** Numerical analysis, linear algebra

Overview: Quasi-Monte Carlo (QMC) methods use low-discrepancy sequences instead of pseudo-random numbers to achieve faster convergence. You will implement and compare the Halton, Faure, Sobol', and Niederreiter sequences against MT19937 and PCG64, characterize convergence rates in 2 through 20 dimensions, and study the breakdown of QMC in very high dimensions.

Objectives:

- Implement the Halton and Faure sequences from mathematical specification; use `scipy.stats.qmc` for Sobol' and Niederreiter
- Define and implement the star discrepancy measure D_N^* in 2D; apply it to all sequences
- Compare convergence rates on the Genz test function suite in dimensions 2, 4, 8, 12, and 20
- Identify the crossover dimension at which QMC no longer outperforms PMC for each sequence family
- Implement randomized QMC (scrambled Sobol') and measure variance reduction vs. standard QMC

- Apply both approaches to a high-dimensional option pricing problem (basket option, 20 assets)

Deliverables:

- Implementations of Halton and Faure sequences with validation
- Discrepancy analysis for all sequences vs. dimension
- Convergence rate plots on Genz functions
- 18–22 page report including a theoretical derivation of QMC convergence bounds

5.3. RNG Quality and Its Impact on Scientific Simulation Accuracy

Difficulty: Intermediate **Primary Tools:** Python, NumPy, matplotlib, NetworkX **Recommended Background:** Simulation, basic statistics

Overview: Most scientists use Monte Carlo simulations without carefully considering the effect of RNG quality on their results. You will conduct a systematic study of how RNG quality affects the accuracy and reproducibility of three scientifically relevant simulations: epidemic spreading (SIR model), random graph generation (Erdős–Rényi and Barabási–Albert), and Markov Chain Monte Carlo (Metropolis-Hastings).

Objectives:

- Implement all three simulations with a pluggable RNG interface
- Run each simulation with at least six generators from a range of qualities (LCG with poor parameters, MT19937, PCG64, Philox4x32, ChaCha20, and a deliberately broken generator)
- For each simulation, define precise quality metrics: SIR final size distribution, graph degree distribution KL-divergence, MCMC mixing time
- Identify which simulation metrics are sensitive to RNG quality and which are robust
- Study ensemble variance as a function of sample size for each generator; identify generators that converge to biased estimates
- Design a minimal diagnostic test specific to each simulation to detect RNG-induced bias

Deliverables:

- Simulation framework with pluggable RNG interface
- Sensitivity analysis across all three simulations and six generators
- Simulation-specific diagnostic tests
- 15–20 page report with recommendations for simulation practitioners

6. True and Physical Randomness

6.1. Camera Sensor Noise as an Entropy Source: Characterization and Extraction

Difficulty: Intermediate **Primary Tools:** Python, OpenCV, NumPy, NIST SP 800-22 **Recommended Background:** Basic signal processing, information theory

Overview: Thermal noise in camera image sensors is a physical entropy source suitable for random number generation. You will build a complete pipeline from raw image acquisition to statistical-quality bit extraction, characterize the entropy per pixel as a function of sensor, temperature, and

ISO setting, and evaluate the extracted bits against NIST SP 800-22.

Objectives:

- Capture dark-frame images using a webcam or smartphone camera under controlled conditions (covered lens, varying ISO, varying temperature)
- Implement at least three bit extraction methods: LSB extraction, hash-based extraction, and von Neumann debiasing; compare the entropy rate of each
- Estimate bits-per-pixel entropy using NIST SP 800-90B estimators; measure how ISO setting, temperature, and sensor model affect entropy rate
- Apply NIST SP 800-22 to the extracted bits from each extraction method
- Characterize spatial and temporal correlations in the sensor noise; implement a decorrelation step
- Compare throughput and entropy rate to `/dev/urandom` and RDRAND

Deliverables:

- Complete image-to-bits pipeline with three extraction methods
- SP 800-90B entropy characterization as a function of sensor parameters
- SP 800-22 results for all extraction methods
- 15–20 page report including a threat model for the entropy source

6.2. Keystroke and Mouse Timing as an Entropy Source

Difficulty: Intermediate **Primary Tools:** Python, `pynput`, NIST SP 800-90B tools **Recommended Background:** Information theory, operating systems basics

Overview: Human behavioral timing — keystrokes, mouse movements, and scrolling — is a primary entropy source in operating systems such as Linux (`/dev/random`) and Windows CNG. You will build a behavioral entropy collector, characterize the entropy rate of each source, apply the NIST SP 800-90B estimation suite, and study how the entropy rate varies with user activity, system load, and timing resolution.

Objectives:

- Build a cross-platform entropy collector capturing: keystroke inter-arrival time, mouse movement delta, scroll wheel events, and window focus changes
- Collect at least 100,000 events per source from at least five distinct users performing standardized tasks (typing, web browsing, gaming)
- Apply all ten NIST SP 800-90B estimators to each source; report per-event min-entropy
- Study entropy rate as a function of user activity type, system load (idle vs. compiling), and timer resolution (1 ms vs. 100 ns)
- Implement a simple entropy accumulator; measure how long it takes to accumulate 256 bits of min-entropy from each source
- Compare across operating systems (Windows, Linux, macOS) if hardware is available

Deliverables:

- Cross-platform entropy collector
- SP 800-90B analysis across all sources and user profiles

- Entropy accumulation time study
- 15–20 page report with recommendations for OS entropy pool design

7. Data Science and Machine Learning Applications

7.1. Generator Identification via Machine Learning

Difficulty: Advanced **Primary Tools:** Python, PyTorch or TensorFlow, scikit-learn **Recommended Background:** Machine learning, deep learning basics

Overview: Statistical test batteries compute hand-designed features to distinguish generators. Can a neural network learn to distinguish generators without hand-designed features? You will train classifiers to identify the source generator from raw bitstreams, study what features the network learns, and compare classification accuracy to a hand-designed feature baseline.

Objectives:

- Build a dataset of bitstreams from at least ten generators (LCG variants, MT19937, PCG64, xoshiro256**, Philox4x32, ChaCha20, hardware noise); generate at least 100,000 sequences of length 20,000 bits each
- Train and evaluate at least three classifier architectures: (1) MLP on hand-designed statistical features, (2) 1D CNN on raw bits, (3) LSTM/Transformer on raw bits
- Measure classification accuracy as a function of sequence length from 100 to 100,000 bits
- Apply explainability tools (SHAP, Grad-CAM) to identify which bit positions or statistical patterns drive classification
- Study transfer learning: train on a subset of generators; evaluate generalization to held-out generators
- Compare neural classification accuracy to NIST SP 800-22 and TestU01 detection rates on the same generators

Deliverables:

- Complete dataset pipeline and three classifier implementations
- Accuracy vs. sequence-length curves for all classifiers
- Explainability analysis comparing learned vs. hand-designed features
- 20–25 page report analyzing what the models learn about generator structure

7.2. Randomness in Deep Learning: Initialization, Dropout, and Data Augmentation

Difficulty: Intermediate–Advanced **Primary Tools:** Python, PyTorch, NumPy **Recommended Background:** Deep learning, optimization

Overview: Deep learning depends on randomness at every stage — weight initialization, mini-batch sampling, dropout, and data augmentation. You will conduct a systematic study of how RNG quality and seeding strategy affect model training dynamics, final accuracy, and reproducibility across at least three architectures and two datasets.

Objectives:

- Implement pluggable RNG support in PyTorch training pipelines; swap between MT19937 (PyTorch default), PCG64, Philox (via NumPy), and a deliberately poor LCG
- Study weight initialization: measure the variance in final test accuracy across 50 training runs with different seeds for each RNG; quantify how much accuracy variance is attributable to the RNG vs. optimization stochasticity
- Study dropout: compare the effective information dropout rate and gradient variance for each RNG on a ResNet trained on CIFAR-10
- Study reproducibility: quantify the conditions under which fully deterministic training is achievable (CUDA determinism flags, fixed seeds); measure the reproducibility gap between CPU and GPU training
- Measure mini-batch sampling quality: test whether batch compositions differ statistically from uniform random sampling for each RNG

Deliverables:

- PyTorch training framework with pluggable RNG
- Accuracy variance study across 50 runs per RNG
- Reproducibility analysis across CPU/GPU configurations
- 18–22 page report with recommendations for reproducible deep learning

8. Theory and Mathematical Analysis

8.1. Linear Complexity Analysis and the Berlekamp–Massey Algorithm

Difficulty: Advanced **Primary Tools:** Python, SageMath (for GF arithmetic) **Recommended Background:** Abstract algebra, linear algebra over finite fields

Overview: The linear complexity of a binary sequence — the length of the shortest LFSR that generates it — is a fundamental measure of unpredictability. You will implement the Berlekamp–Massey algorithm over GF(2), derive theoretical linear complexity profiles for several generator families, and use linear complexity as a forensic tool to characterize and compare generators.

Objectives:

- Implement the Berlekamp–Massey algorithm over GF(2) from scratch; validate against SageMath's reference implementation
- Compute the linear complexity profile (LC vs. sequence length) for: LCG, MT19937, xoshiro256**, PCG64, Philox4x32, and a hardware noise source
- Derive theoretically the expected linear complexity profile for a random sequence of length N; overlay on empirical profiles
- Demonstrate that MT19937 has a linear complexity of approximately $N/2$ (matching theory for a linearly recursive generator)
- Show that Philox and ChaCha20 have linear complexity profiles indistinguishable from random
- Implement the Berlekamp–Massey attack on a stream cipher with a short LFSR; recover the connection polynomial

Deliverables:

- Complete Berlekamp–Massey implementation with validation

- Linear complexity profiles for all generators
- LFSR attack demonstration
- 18–22 page report including theoretical derivations of expected profiles

8.2. Kolmogorov Complexity and Algorithmic Randomness: Theory and Approximation

Difficulty: Advanced **Primary Tools:** Python, gzip/bzip2/zstd/lzma (via stdlib) **Recommended Background:** Theory of computation, information theory

Overview: Kolmogorov complexity provides the deepest formal definition of randomness but is uncomputable. Compression algorithms provide computable upper bounds. You will implement a comprehensive Kolmogorov complexity estimation framework using multiple compressors, validate it against sequences of known complexity, apply it to classify RNG outputs, and study the relationship between compression ratio and statistical test outcomes.

Objectives:

- Implement a compression-based complexity estimator using gzip, bzip2, zstd, lzma, and a custom LZ77 implementation; compare their sensitivity
- Design and validate a complexity classification framework: given a bitstream, classify it as "random," "structured," or "adversarially structured" based on compression ratio
- Compute normalized compression distance (NCD) between generator outputs; use hierarchical clustering to group generators by complexity profile
- Study the relationship between compression ratio and each of the 15 NIST SP 800-22 test outcomes for 10+ generators
- Analyze adversarial sequences: construct bitstreams that fool compressors (appear random under compression) but fail statistical tests, and vice versa
- Derive bounds on the compression ratio achievable for PRNG outputs as a function of the generator's state size

Deliverables:

- Complexity estimation framework with five compressors
- NCD-based generator clustering dendrogram
- Correlation analysis: compression ratio vs. NIST test outcomes
- 20–25 page report with theoretical discussion of computability limits

Appendix: Suggested Tools and Resources

Software Libraries

- **Random123**: `github.com/DEShawResearch/random123` — Counter-based RNG library (C/C++/CUDA)
- **TestU01**: `simul.iro.umontreal.ca/testu01` — Statistical test batteries (C)
- **NIST SP 800-22**: `csrc.nist.gov/publications/detail/sp/800-22` — 15-test statistical suite (C)
- **Dieharder**: `webhome.phy.duke.edu/~rgb/General/dieharder.php` — Extended Diehard battery (C)
- **PCG**: `pcg-random.org` — O'Neill's PCG family (C/C++)

- **PractRand**: `pracrand.sourceforge.net` — Continuous testing RNG framework
- **SageMath**: `sagemath.org` — Computer algebra for finite field arithmetic

Key References

- Knuth, D.E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, 1997.
- L'Ecuyer & Simard. "TestU01: A C Library for Empirical Testing of RNGs." *ACM TOMS*, 33(4), 2007.
- Salmon, Moraes, Dror, Shaw. "Parallel Random Numbers: As Easy as 1, 2, 3." *SC'11*, 2011.
- Rukhin et al. *NIST SP 800-22 Rev. 1a: A Statistical Test Suite for RNGs.* NIST, 2010.
- O'Neill, M.E. "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation." Harvey Mudd Technical Report, 2014.
- Bernstein, D.J. "ChaCha, a variant of Salsa20." Workshop Record of SASC, 2008.
- Barker & Kelsey. *NIST SP 800-90A Rev. 1: Recommendation for Random Number Generation Using DRBGs.* NIST, 2015.

Florida State University — Department of Computer Science
The Data Science of Randomness — Project Specifications