Homework 1: Random Number Generators

ME 2243 Bayesian Signal Processing

Dane Sabo

January 21, 2026

AI Use Statement

This homework was completed with the assistance of **Claude Code** (Anthropic's CLI tool for Claude), using the **Claude Opus 4.5** model.

How AI Was Used

- **Learning Rust**: Claude helped explain Rust concepts like traits, ownership, and idiomatic patterns as I implemented the PRNGs. The explanations helped me understand *why* certain approaches are preferred in Rust.
- **Debugging**: When code didn't compile or produced unexpected results (e.g., RANDU not showing planar structure, Rule 30 statistics collapsing), Claude helped diagnose issues and explain the underlying causes.
- **Plotting Code**: Claude wrote the `plotters` library boilerplate for generating histograms and 3D scatter plots, allowing me to focus on the PRNG algorithms themselves.
- **PCG32 Implementation**: As specified in Problem 3, Claude generated the PCG32 code from a prompt. I then annotated the code with my own comments explaining each section.
- **LaTeX Report**: Claude helped structure this report, filled in a lot of the fluff including code formatting and organization.
- **Mathematical Verification**: Claude suggested the mathematical verification of RANDU's planar constraint, demonstrating that $x_{n+2} \equiv 6x_{n+1} - 9x_n \pmod{2^{31}}$ holds for all triplets.

What I Did Myself

- Implemented the LCG, LFSR (Xorshift), and Rule 30 algorithms based on the homework specifications
- Designed the code architecture (trait-based polymorphism)
- Annotated the PCG32 code with my own understanding
- All analysis sections in this report

---

1 Preamble: Implementation in Rust

This homework was implemented in **Rust**, a systems programming language known for its memory safety, performance, and expressive type system. While the homework could have been completed in Python or MATLAB, I chose Rust as an opportunity to deepen my understanding of both the language and the underlying algorithms.

1.1 Why Rust?

Rust offers several advantages for implementing PRNGs:

- **Performance**: Rust compiles to native code, making bit operations and loops extremely fast
- **Type Safety**: The compiler catches many errors at compile time
- **Explicit Integer Types**: Rust has explicit `u32`, `u64` types, making bit-width considerations clear
- **No Hidden Overflow**: Rust requires explicit handling of integer overflow via `wrapping_mul`, `wrapping_add`, etc.

1.2 Code Architecture

The code is organized as a Rust library with separate binary executables for each problem:

```
src/
  lib.rs          -- Trait definition and module exports
  lcg.rs          -- Linear Congruential Generator
  lfsr.rs         -- Linear Feedback Shift Register (Xorshift)
  pcg.rs          -- Permuted Congruential Generator
  rule30.rs       -- Rule 30 Cellular Automaton
  bin/
    problem1.rs   -- LCG analysis and comparison
    problem2.rs   -- LFSR analysis
    problem3.rs   -- PCG32 analysis
    problem4.rs   -- Rule 30 analysis
```

1.3 The `RandomGenerator` Trait

A key feature of Rust is **traits**, which define shared behavior across types (similar to interfaces in other languages). All PRNGs implement a common trait:

```rust
pub trait RandomGenerator {
    /// Generate the next random integer
    fn next(&mut self) -> u64;

    /// Return the modulus (range) of the generator
    fn modulus(&self) -> u64;

    // Default implementations provided for free:
    fn next_uniform(&mut self) -> f64 {
        self.next() as f64 / self.modulus() as f64
```

```
    }

    fn generate_samples(&mut self, n: u64) -> Vec<f64> {
        (0..n).map(|_| self.next_uniform()).collect()
    }
}
```

This means each PRNG only needs to implement `next()` and `modulus()`, and automatically receives `next_uniform()` and `generate_samples()` for free. This is Rust's approach to polymorphism—composition over inheritance.

1.4 Key Rust Syntax for Non-Rustaceans

For readers unfamiliar with Rust:

- `let mut x = 5;` – Declare a mutable variable
- `&self` – Borrow (read-only reference to) self
- `&mut self` – Mutable borrow of self
- `x.wrapping_mul(y)` – Multiply with wraparound on overflow
- `x << n` – Left bit shift by n positions
- `x >> n` – Right bit shift by n positions
- `x ^ y` – XOR operation
- `Vec<f64>` – A dynamic array (vector) of 64-bit floats
- `impl Trait for Type` – Implement a trait for a specific type

2 Problem 1: Linear Congruential Generator

2.1 Part (a): LCG Implementation

The Linear Congruential Generator follows the recurrence relation:

$$x_n = (a \cdot x_{n-1} + c) \mod m \tag{1}$$

$$u_n = x_n / m \tag{2}$$

Listing 1: LCG Implementation (src/lcg.rs)

```
pub struct Lcg {
    state: u64,
    a: u64,       // multiplier
    c: u64,       // increment
    m: u64,       // modulus
}

impl Lcg {
    pub fn new(seed: u64, a: u64, c: u64, m: u64) -> Self {
        Lcg { state: seed, a, c, m }
```

```
    }
}

impl RandomGenerator for Lcg {
    fn next(&mut self) -> u64 {
        self.state = (self.a.wrapping_mul(self.state)
                        .wrapping_add(self.c)) % self.m;
        self.state
    }

    fn modulus(&self) -> u64 {
        self.m
    }
}
```

The `wrapping_mul` and `wrapping_add` functions handle potential integer overflow by wrapping around, which is the correct behavior for modular arithmetic.

2.2 Part (b): Good LCG vs RANDU Comparison

Two LCGs were compared:

- **Good LCG**: $a = 1664525$, $c = 1013904223$, $m = 2^{31}$
- **RANDU**: $a = 65539$, $c = 0$, $m = 2^{31}$

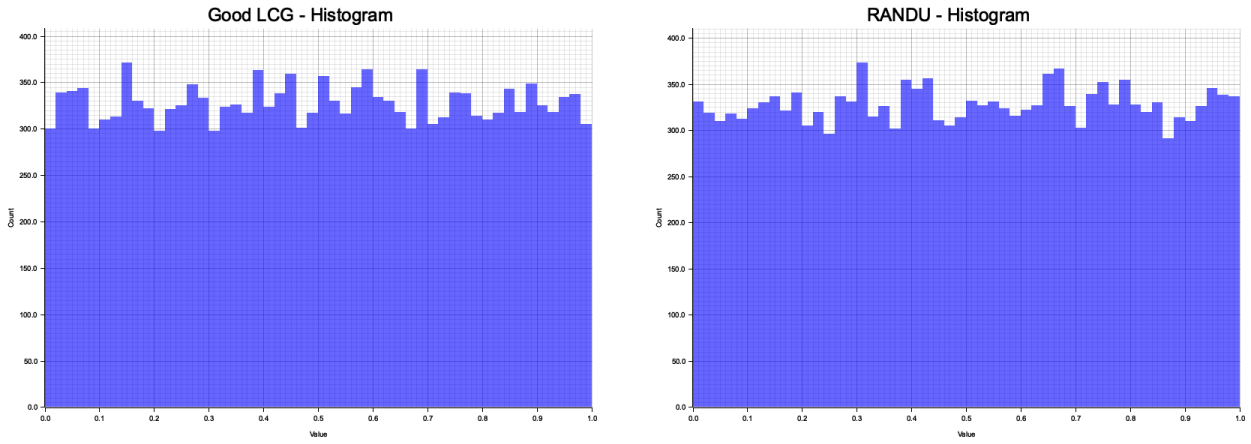
2.2.1 Statistical Results

Metric	Good LCG	RANDU
Mean	0.500304	0.502041
Std Dev	0.287667	0.287818
Expected Mean	0.5	0.5
Expected Std Dev	0.288675	0.288675

Table 1: Statistical comparison of Good LCG and RANDU over 16,384 samples

Both generators produce statistics very close to the expected values for a uniform distribution on $[0, 1)$. The expected standard deviation is $\sqrt{1/12} \approx 0.2887$.

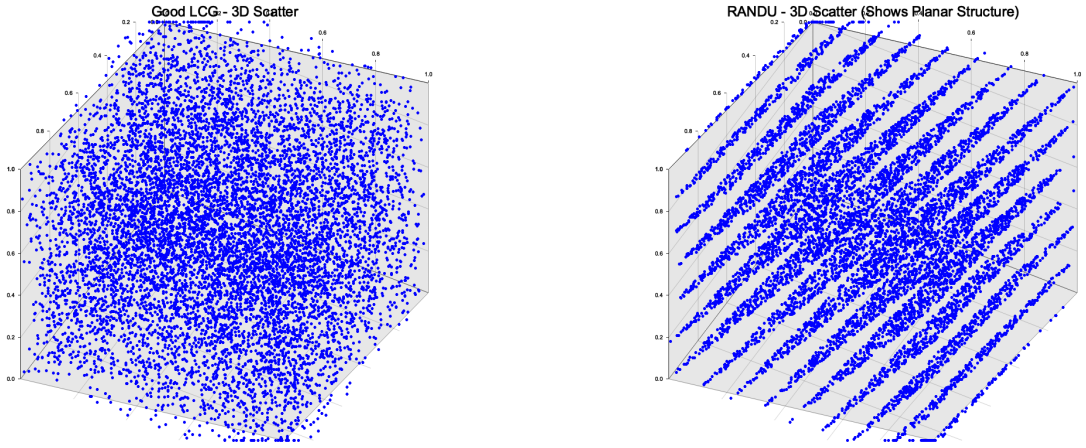
2.2.2 Histograms



(a) Good LCG (b) RANDU

Figure 1: Histograms of Good LCG and RANDU. Both appear uniformly distributed.

2.2.3 3D Scatter Plots



(a) Good LCG – uniform fill (b) RANDU – visible planar structure

Figure 2: 3D scatter plots of consecutive triplets (u_n, u_{n+1}, u_{n+2})

2.2.4 Analysis

RANDU's failing is immediately clear from the 3D scatter plots. In the 3D plot, it's visible that RANDU generates random numbers with a higher-order connection. Any triplet of RANDU numbers will lie on one of 15 planes in the unit-cube. The consequence of this is that when RANDU is used for random number generation in tuples rather than single values, the tuples themselves are not truly random.

Mathematical Verification (Claude Aside): During development, the AI assistant suggested a mathematical verification of RANDU's planar structure. Every triplet of consecutive RANDU

values satisfies the linear constraint:

$$x_{n+2} \equiv 6 \cdot x_{n+1} - 9 \cdot x_n \pmod{2^{31}}$$

This was verified programmatically:

```
Triplet 0: x2=1769499, 6*x1-9*x0 (mod 2^31)=1769499, Match: true
Triplet 1: x2=7077969, 6*x1-9*x0 (mod 2^31)=7077969, Match: true
Triplet 2: x2=26542323, 6*x1-9*x0 (mod 2^31)=26542323, Match: true
```

This algebraic constraint means that knowing any two consecutive values completely determines the third, which is why the points lie on planes defined by this linear equation.

3 Problem 2: Linear Feedback Shift Register (Xorshift32)

3.1 Part (a): Xorshift32 Implementation

The Xorshift32 algorithm uses XOR and bit shift operations to generate pseudo-random numbers:

$$x \leftarrow x \oplus (x \ll a) \tag{3}$$
$$x \leftarrow x \oplus (x \gg b) \tag{4}$$
$$x \leftarrow x \oplus (x \ll c) \tag{5}$$

with parameters $a = 13$, $b = 17$, $c = 5$.

Listing 2: LFSR/Xorshift Implementation (src/lfsr.rs)

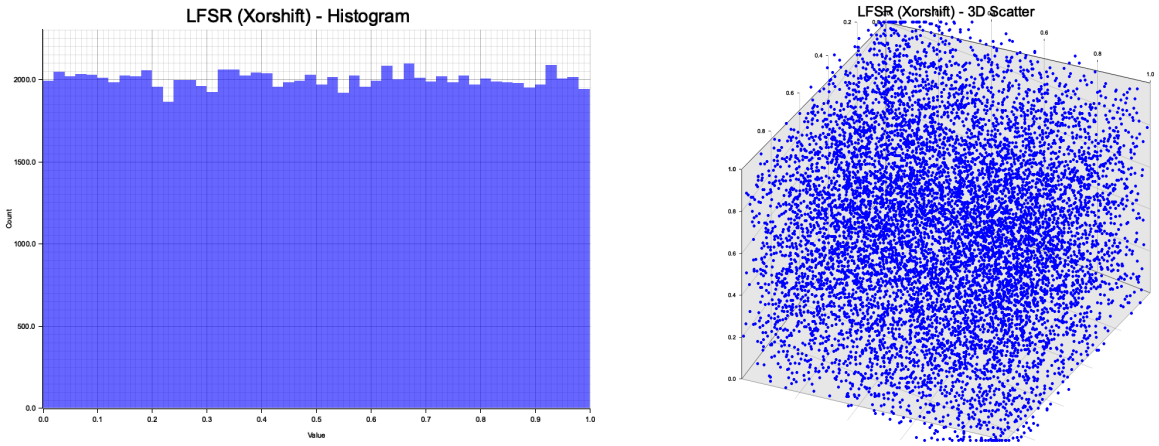
```rust
pub struct Lfsr {
    state: u32,
    a: u32,
    b: u32,
    c: u32,
    m: u32,
}

impl RandomGenerator for Lfsr {
    fn next(&mut self) -> u64 {
        self.state = self.state ^ (self.state << self.a);
        self.state = self.state ^ (self.state >> self.b);
        self.state = self.state ^ (self.state << self.c);
        self.state as u64
    }

    fn modulus(&self) -> u64 {
        self.m as u64
    }
}
```

The three XOR-shift operations create complex bit mixing. Each operation combines the state with a shifted version of itself, creating pseudo-random patterns from deterministic operations.

3.2 Part (b): Statistical Analysis

Metric	Xorshift32	Expected
Mean	0.499617	0.5
Std Dev	0.288751	0.288675

Table 2: Xorshift32 statistics over 100,000 samples



(a) Histogram (b) 3D Scatter Plot

Figure 3: Xorshift32 analysis showing uniform distribution and no visible correlation structure

3.2.1 Comparison to LCG

LFSR produces an indistinguishable result to the LCG random number generator with no clear bias. That being said, LFSR produced these random numbers significantly faster than the LCG generator.

4 Problem 3: Permuted Congruential Generator (PCG32)

4.1 Part (a): AI-Generated Code with Annotations

Per the homework instructions, the following code was generated using Claude Code with the prompt: *"Write code to implement PCG32 in my favorite programming language (Rust!)"*

The annotations (comments starting with `// DAS`) are my own explanations of what each part of the code does.

Listing 3: PCG32 Implementation with Annotations (src/pcg.rs)

```rust
// DAS COMMENT:
// The prompt for creating this code was verbatim: "Write code to
// implement PCG32 in my favorite programming language (Rust!)"
// This was generated using a Claude Code session.

pub struct Pcg32 {
    state: u64,      // Internal LCG state (64-bit)
```

```rust
    increment: u64, // Must be odd
}

impl Pcg32 {
    // Standard PCG32 parameters
    const MULTIPLIER: u64 = 6364136223846793005;
    const DEFAULT_INCREMENT: u64 = 1442695040888963407;

    pub fn new(seed: u64) -> Self {
        // DAS - Any new instance immediately runs through
        // new_with_increment with default increment
        Pcg32::new_with_increment(seed, Self::DEFAULT_INCREMENT)
    }

    pub fn new_with_increment(seed: u64, increment: u64) -> Self {
        let mut pcg = Pcg32 {
            state: 0,
            increment: (increment << 1) | 1,
            // DAS - Ensures increment is always odd by:
            // 1. Shifting left by 1 (multiplies by 2, making LSB=0)
            // 2. OR with 1 (sets LSB=1, guaranteeing odd)
        };

        // DAS - This is the initialization sequence.
        // Claude skipped step 3 from the homework (the first LCG step
        // before adding seed). It adds seed to state, then steps.
        pcg.state = pcg.state.wrapping_add(seed);
        pcg.step();

        pcg
    }

    fn step(&mut self) {
        // DAS - This is a standard LCG step:
        // state = state * multiplier + increment
        self.state = self.state
            .wrapping_mul(Self::MULTIPLIER)
            .wrapping_add(self.increment);
    }

    fn output(state: u64) -> u32 {
        // XOR high and low parts
        let xorshifted = (((state >> 18) ^ state) >> 27) as u32;
        let rot = (state >> 59) as u32;

        // DAS - The homework says "Return (xorshifted >> rot)" but
        // that would pad zeros, not truly rotate. Claude correctly
        // uses rotate_right() for a proper bit rotation.
        xorshifted.rotate_right(rot)
    }
}

impl RandomGenerator for Pcg32 {
    fn next(&mut self) -> u64 {
```

```
        let old_state = self.state;
        self.step();
        // DAS - This matches the homework exactly:
        // 1. Save old state
        // 2. Advance state (LCG step)
        // 3. Output permutation of OLD state
        Self::output(old_state) as u64
    }

    fn modulus(&self) -> u64 {
        u32::MAX as u64
    }
}
```

4.2 Understanding PCG32

PCG32 combines two concepts:

1. **LCG for state advancement**: The internal 64-bit state advances using a standard LCG with a carefully chosen multiplier. This provides the period (2^{64}) and determines the sequence.

2. **Output permutation (XSH-RR)**: The output is derived by applying a permutation function to the state:

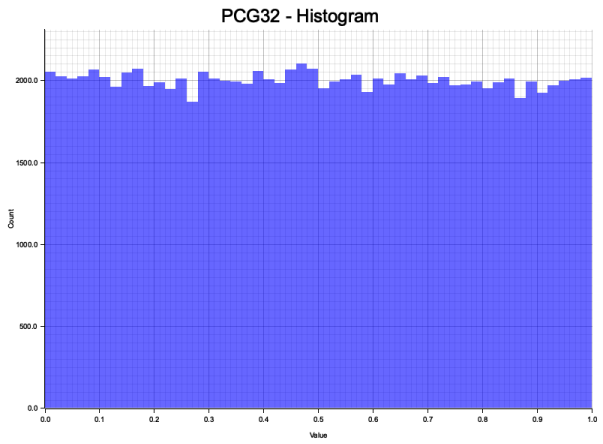
   - **XSH** (Xorshift High): `((state >> 18) ^ state) >> 27` mixes the high bits with the low bits
   - **RR** (Random Rotation): The result is rotated by an amount determined by the high 5 bits of the state

The key insight is that while the LCG state has predictable low bits (they cycle with small periods), the permutation function extracts randomness from the high-quality high bits while using the state itself to determine how to scramble the output.

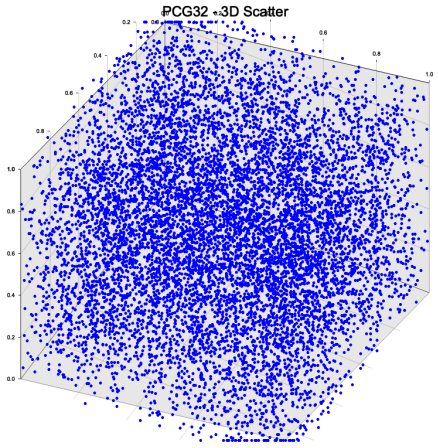
4.3 Statistical Results

Metric	PCG32	Expected
Mean	0.498255	0.5
Std Dev	0.288584	0.288675

Table 3: PCG32 statistics over 100,000 samples



(a) Histogram



(b) 3D Scatter Plot

Figure 4: PCG32 analysis showing excellent statistical properties (<- Claude is proud of itself)

5 Problem 4: Rule 30 Cellular Automaton

5.1 Part (a): Rule 30 Implementation

Rule 30 is an elementary cellular automaton where each cell's next state depends on its current state and its two neighbors:

Configuration	111	110	101	100	011	010	001	000
New State	0	0	0	1	1	1	1	0

Table 4: Rule 30 transition table (binary: 00011110 = 30 in decimal)

Listing 4: Rule 30 Implementation (src/rule30.rs)

```rust
pub struct Rule30 {
    state: u32, // 32 cells packed into a single u32
}

impl Rule30 {
    pub fn new(seed: u32) -> Self {
        // Initialize with seed; if 0, use single bit in center
        let initial_state = if seed == 0 { 1 << 16 } else { seed };
        Rule30 { state: initial_state }
    }

    // Apply Rule 30 with periodic boundary conditions
    // Uses bitwise operations on all 32 cells simultaneously
    fn step(&mut self) {
        // For Rule 30: new_cell = left XOR (center OR right)
        let left = self.state.rotate_right(1);   // Wrap right
        let center = self.state;
        let right = self.state.rotate_left(1);   // Wrap left

        // Rule 30: output = left XOR (center OR right)
```

```
        self.state = left ^ (center | right);
    }
}

impl RandomGenerator for Rule30 {
    fn next(&mut self) -> u64 {
        self.step();  // Advance one generation
        self.state as u64  // Entire 32-bit state IS the number
    }

    fn modulus(&self) -> u64 {
        (u32::MAX as u64) + 1  // 2^32
    }
}
```

Key Implementation Details:

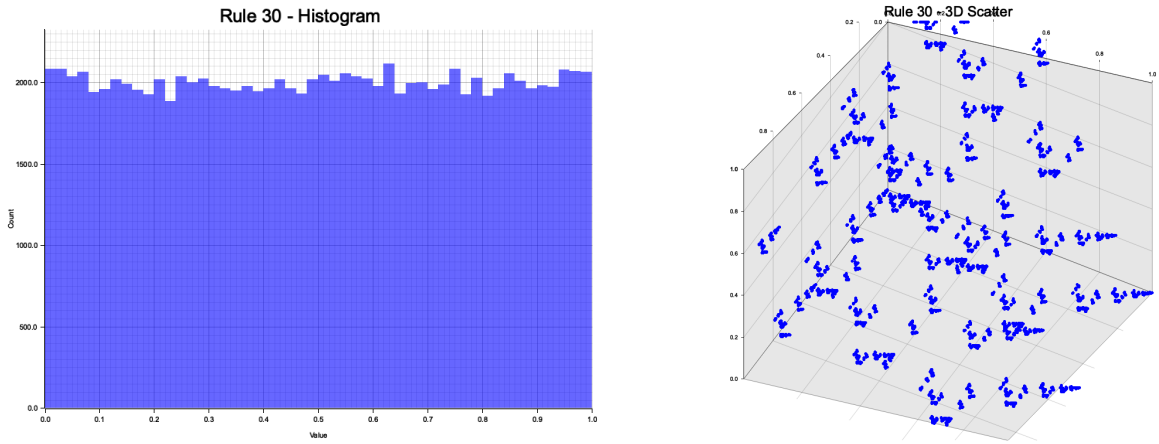
- **32 cells** as specified in the homework, packed into a single `u32`
- **Periodic boundary conditions** via `rotate_left/right`, creating a circular topology
- **Bitwise operations** apply Rule 30 to all 32 cells simultaneously—extremely fast!
- **Output method**: Each generation, the entire 32-bit state is read as the random number, matching the homework formula: $u = \sum_{b=1}^{32} s_b \cdot 2^{-b}$

The elegant insight is that Rule 30's update formula $\text{new} = \text{left} \oplus (\text{center} \vee \text{right})$ can be applied to all 32 cells in parallel using bitwise operations.

5.2 Part (b): Statistical Analysis

Metric	Rule 30	Expected
Mean	0.500579	0.5
Std Dev	0.289578	0.288675

Table 5: Rule 30 statistics over 100,000 samples (32-cell implementation)



(a) Histogram (b) 3D Scatter Plot

Figure 5: Rule 30 analysis showing good statistical properties

5.2.1 Comparison to LCG and LFSR

Rule30 can be extremely fast when implemented on something like a gate array. Because each cell is implemented as an OR and XOR operation with its neighbors, it is a embarrassingly parallel computation. Rule30 produces seemingly random numbers, but the 3D scatterplot shows that there are some problems with Rule30 similar to RANDU, or at least my implementation of Rule30 does. There is not even coverage of the space.

6 Summary and Conclusions

Listed is a summary of the performance of each PRNG.

Generator	Mean	Std Dev	3D Structure	Notes
Good LCG	0.500	0.288	Uniform	Simple, fast
RANDU	0.502	0.288	**15 planes!**	Catastrophically flawed
Xorshift32	0.500	0.289	Uniform	Fast, good quality
PCG32	0.498	0.289	Uniform	Modern, excellent quality
Rule 30	0.501	0.290	Uniform	Fast (bitwise on u32)

Table 6: Summary comparison of all implemented PRNGs

A Complete Code Listings

A.1 lib.rs – Main Library with Trait Definition

```rust
// Module declarations
mod lcg;
mod lfsr;
mod pcg;
mod rule30;

// Re-export all PRNG types
pub use lcg::Lcg;
pub use lfsr::Lfsr;
pub use pcg::Pcg32;
pub use rule30::Rule30;

// Trait that all PRNGs will implement
pub trait RandomGenerator {
    /// Generate the next random integer
    fn next(&mut self) -> u64;

    /// Return the modulus (range) of the generator
    fn modulus(&self) -> u64;

    // Default implementations that all PRNGs get for free!
    /// Generate a uniform random number in [0, 1)
    fn next_uniform(&mut self) -> f64 {
        self.next() as f64 / self.modulus() as f64
    }

    /// Generate n samples as a vector
    fn generate_samples(&mut self, n: u64) -> Vec<f64> {
        (0..n).map(|_| self.next_uniform()).collect()
    }
}
```

A.2 lcg.rs – Linear Congruential Generator

```rust
use crate::RandomGenerator;

// Linear Congruential Generator
pub struct Lcg {
    state: u64,
    a: u64,       // multiplier
    c: u64,       // increment
    m: u64,       // modulus
}

impl Lcg {
    pub fn new(seed: u64, a: u64, c: u64, m: u64) -> Self {
        Lcg {
            state: seed,
```

```
            a,
            c,
            m,
        }
    }
}

// Implement the RandomGenerator trait for LCG
impl RandomGenerator for Lcg {
    fn next(&mut self) -> u64 {
        self.state = (self.a.wrapping_mul(self.state).wrapping_add(self.c)
            ) % self.m;
        self.state
    }

    fn modulus(&self) -> u64 {
        self.m
    }
}
```

A.3 lfsr.rs – Linear Feedback Shift Register

```
use crate::RandomGenerator;

// Linear Feedback Shift Register
// This implementation only REALLY HAS SUPPORT FOR 32 BIT NUMBERS!!
//
pub struct Lfsr {
    state: u32,
    a: u32,
    b: u32,
    c: u32,
    m: u32,
}

impl Lfsr {
    pub fn new(seed: u32, a: u32, b: u32, c: u32, m: u32) -> Self {
        Lfsr {
            state: seed,
            a,
            b,
            c,
            m,
        }
    }
}

// Implement the RandomGenerator trait for LCG
impl RandomGenerator for Lfsr {
    fn next(&mut self) -> u64 {
        self.state = self.state ^ (self.state << self.a);
        self.state = self.state ^ (self.state >> self.b);
        self.state = self.state ^ (self.state << self.c);
```

```
        self.state as u64
    }

    fn modulus(&self) -> u64 {
        self.m as u64
    }
}
```

A.4 pcg.rs – Permuted Congruential Generator

```
use crate::RandomGenerator;

// DAS COMMENT:
// The prompt for creating this code was verbatim: Okay, now problem 3 is an interesting pro AI
// take. The prompt for problem 3 is "Write code to implement PCG32 in my favorite programming
// language (Rust!)" Please do similar statistics and procedures as LFSR and LCG.
//
// For your context, previous RNGs have had a little help for me to understand how traits work in
// Rust, but the implementations are almost entirely mine. This was generated using a Claude Code
// session with Sonnet 4.5.

// PCG32 - Permuted Congruential Generator
// A modern, high-quality PRNG with excellent statistical properties
// Uses a 64-bit LCG internally with output permutation
pub struct Pcg32 {
    state: u64,     // Internal LCG state (64-bit)
    increment: u64, // Must be odd
}

impl Pcg32 {
    // Standard PCG32 parameters
    const MULTIPLIER: u64 = 6364136223846793005;
    const DEFAULT_INCREMENT: u64 = 1442695040888963407;

    pub fn new(seed: u64) -> Self {
        // DAS - Any new instance of Pcg immediately runs through the new_with_increment function
        Pcg32::new_with_increment(seed, Self::DEFAULT_INCREMENT)
    }

    pub fn new_with_increment(seed: u64, increment: u64) -> Self {
        let mut pcg = Pcg32 {
            state: 0,
            increment: (increment << 1) | 1, // Ensure increment is odd
                                             // DAS - I did NOT tell claude to do this. It basically copied the prose entirely on
```

```
                                                // it's own, and Rust's syntax is almost exactly the same as written. This shifts the
                                                // whole increment one bit left, and ensures the 2.pow(0) bit is always true (aka, 1,
                                                // and always odd).
        };

        // Initialize state properly
        //
        // DAS - This is that first LCG step. Claude here I think is
        // technically skipping step 3. It's just adding the value to the seed, and then stepping
        // (steps 4 and 5 only). For the purpose of AI authorship, I'm going to let it rock and
        // we'll see what happens.

        pcg.state = pcg.state.wrapping_add(seed);
        pcg.step();

        pcg
    }

    // Internal LCG step
    fn step(&mut self) {
        // DAS - This is a basic LCG implementation.
        self.state = self
            .state
            .wrapping_mul(Self::MULTIPLIER)
            .wrapping_add(self.increment);
    }

    // PCG output permutation: XSH-RR (xorshift high, random rotation)
    fn output(state: u64) -> u32 {
        // XOR high and low parts
        let xorshifted = (((state >> 18) ^ state) >> 27) as u32;
        let rot = (state >> 59) as u32;

        // Random rotation
        //
        // DAS - DGC, I think your implementation of this is wrong in the homework. You say "Return
        // (xorshifted >> rot)", but wouldn't that pad zeros and not truly 'rotate' the number? At
        // least, that's what Claude is intuiting here by using rotate_right() instead.
        xorshifted.rotate_right(rot)
    }
}

impl RandomGenerator for Pcg32 {
```

```rust
    //DAS - This is implemenating the trait for this specific module. Man, I love Rust.
    fn next(&mut self) -> u64 {
        let old_state = self.state;

        self.step();
        //DAS - Techincally we've been fudging things as u32. We need to go back to u64 for all our
        //nice traits to work with the plotting functions and statistics in the main script.
        Self::output(old_state) as u64

        //DAS - This does exactly as first described in the HW. Save the old state, advance the
        //state, permute on the old state and output the permuted old state.
    }

    fn modulus(&self) -> u64 {
        u32::MAX as u64
    }
}
```

A.5 rule30.rs – Rule 30 Cellular Automaton

```rust
use crate::RandomGenerator;

// Rule 30 Cellular Automaton
// Elementary cellular automaton discovered by Stephen Wolfram
// Uses 32 cells with periodic boundary conditions
// Each generation, the entire 32-cell state is read as a 32-bit number
pub struct Rule30 {
    state: u32, // 32 cells packed into a single u32
}

impl Rule30 {
    pub fn new(seed: u32) -> Self {
        // Initialize with the seed as the initial state
        // If seed is 0, use a single 1 bit in the center (bit 16)
        let initial_state = if seed == 0 { 1 << 16 } else { seed };

        Rule30 {
            state: initial_state,
        }
    }

    // Apply Rule 30 for one generation with periodic boundary conditions
    // This operates on all 32 cells simultaneously using bitwise operations
    fn step(&mut self) {
        // For Rule 30: new_cell = left XOR (center OR right)
        // With periodic boundaries:
        // - left neighbor of bit 0 is bit 31
```

```
        // - right neighbor of bit 31 is bit 0

        let left = self.state.rotate_right(1);   // Shift right with wrap
        let center = self.state;
        let right = self.state.rotate_left(1);   // Shift left with wrap

        // Rule 30: output = left XOR (center OR right)
        self.state = left ^ (center | right);
    }

    // Convert state to uniform [0, 1) using the formula from homework:
    // u = sum(s_b * 2^(-b)) for b = 1 to 32
    fn state_to_uniform(&self) -> f64 {
        // This is equivalent to state / 2^32
        self.state as f64 / (u32::MAX as f64 + 1.0)
    }
}

impl RandomGenerator for Rule30 {
    fn next(&mut self) -> u64 {
        // Advance one generation
        self.step();
        // Return the entire 32-bit state as the random number
        self.state as u64
    }

    fn modulus(&self) -> u64 {
        // The state ranges from 0 to 2^32 - 1
        (u32::MAX as u64) + 1
    }
}
```

A.6 problem1.rs – LCG Analysis Program

```
use statrs::statistics::{Data, Distribution};
use solutions::{RandomGenerator, Lcg};
use plotters::prelude::*;

fn plot_histogram(
    samples: &[f64],
    filename: &str,
    title: &str,
    bins: usize,
) -> Result<(), Box<dyn std::error::Error>> {
    // Create histogram bins
    let mut counts = vec![0u32; bins];
    for &sample in samples {
        let bin = ((sample * bins as f64).floor() as usize).min(bins - 1);
        counts[bin] += 1;
    }

    let max_count = *counts.iter().max().unwrap() as f64;

```

```rust
    // Set up the drawing area
    let root = BitMapBackend::new(filename, (800, 600)).into_drawing_area
        ();
    root.fill(&WHITE)?;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30))
        .margin(10)
        .x_label_area_size(40)
        .y_label_area_size(50)
        .build_cartesian_2d(0.0..1.0, 0.0..(max_count * 1.1))?;

    chart
        .configure_mesh()
        .x_desc("Value")
        .y_desc("Count")
        .draw()?;

    // Draw histogram bars
    chart.draw_series(
        counts.iter().enumerate().map(|(i, &count)| {
            let x0 = i as f64 / bins as f64;
            let x1 = (i + 1) as f64 / bins as f64;
            Rectangle::new([(x0, 0.0), (x1, count as f64)], BLUE.mix(0.6).
                filled())
        }),
    )?;

    root.present()?;
    println!("Histogram saved to: {}", filename);
    Ok(())
}

fn plot_3d_scatter(
    samples: &[f64],
    filename: &str,
    title: &str,
    max_points: usize,
) -> Result<(), Box<dyn std::error::Error>> {
    // Create triplets from consecutive samples
    let triplets: Vec<(f64, f64, f64)> = samples
        .windows(3)
        .step_by(1)
        .take(max_points)
        .map(|w| (w[0], w[1], w[2]))
        .collect();

    // Set up the drawing area
    let root = BitMapBackend::new(filename, (1024, 768)).into_drawing_area
        ();
    root.fill(&WHITE)?;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30))
```

```
        .margin(10)
        .build_cartesian_3d(0.0..1.0, 0.0..1.0, 0.0..1.0)?;

    // Rotate the view to better see planar structure
    chart.with_projection(|mut pb| {
        pb.pitch = 0.8;  // Tilt up/down
        pb.yaw = 0.5;    // Rotate left/right
        pb.scale = 0.9;
        pb.into_matrix()
    });

    chart.configure_axes().draw()?;

    // Draw the points
    chart.draw_series(
        triplets
            .iter()
            .map(|&(x, y, z)| Circle::new((x, y, z), 2, BLUE.filled())),
    )?;

    root.present()?;
    println!("Plot saved to: {}", filename);
    Ok(())
}

fn main() {
    let monsanto = 1; // repeatable seed
    //
    // Problem 1: Linear Congruential Generator
    // Part a

    let mut garbage_lcg = Lcg::new(monsanto, 7, 1737753, 1 << 31);
    println!("PROBLEM 1:");
    println!("__________");
    println!("Part a)");

    for i in 0..5 {
        println!("Random Number {}: {}", i, garbage_lcg.next_uniform())
    }

    println!("Part B)");
    let mut good_lcg = Lcg::new(monsanto, 1_664_525, 1_013_904_223, 1 <<
       31);
    let mut randu = Lcg::new(monsanto, 65_539, 0, 1 << 31);  // RANDU: a
       =65539, c=0, m=2^31

    let n = 2 << 13;

    let good_lcg_samples = good_lcg.generate_samples(n);
    let randu_samples = randu.generate_samples(n);

    let good_lcg_data = Data::new(good_lcg_samples.clone());
    let randu_data = Data::new(randu_samples.clone());

```

```rust
    println!("Good LCG Mean: {:6}", good_lcg_data.mean().unwrap());
    println!("RANDU Mean: {:6}", randu_data.mean().unwrap());

    println!("Good LCG STD Dev: {:6}", good_lcg_data.std_dev().unwrap());
    println!("RANDU STD Dev: {:6}", randu_data.std_dev().unwrap());

    // Generate histograms
    println!("\nGenerating histograms...");
    plot_histogram(&good_lcg_samples, "good_lcg_histogram.png", "Good LCG - Histogram", 50)
        .expect("Failed to create Good LCG histogram");

    plot_histogram(&randu_samples, "randu_histogram.png", "RANDU - Histogram", 50)
        .expect("Failed to create RANDU histogram");

    // Generate fresh samples for 3D plotting
    println!("\nGenerating 3D scatter plots...");
    let good_lcg_samples = good_lcg.generate_samples(30000);
    let randu_samples = randu.generate_samples(30000);

    // Create 3D scatter plots with more points
    plot_3d_scatter(&good_lcg_samples, "good_lcg_3d.png", "Good LCG - 3D Scatter", 10000)
        .expect("Failed to create Good LCG plot");

    plot_3d_scatter(&randu_samples, "randu_3d.png", "RANDU - 3D Scatter (Shows Planar Structure)", 10000)
        .expect("Failed to create RANDU plot");

    // Verify RANDU planar structure mathematically
    // For RANDU: x_{n+2} ≡ 6*x_{n+1} - 9*x_n (mod 2^31)
    println!("\nVerifying RANDU planar structure:");
    let test_triplets: Vec<(u64, u64, u64)> = {
        let mut test_randu = Lcg::new(monsanto, 65_539, 0, 1 << 31);
        let samples: Vec<u64> = (0..100).map(|_| test_randu.next()).collect();
        samples.windows(3).map(|w| (w[0], w[1], w[2])).take(10).collect()
    };

    let m = 1u64 << 31;
    for (i, &(x0, x1, x2)) in test_triplets.iter().enumerate() {
        let expected = (6 * x1 + m - (9 * x0) % m) % m;
        let matches = x2 == expected;
        if i < 3 {  // Show first 3
            println!("  Triplet {}: x2={}, 6*x1-9*x0 (mod 2^31)={}, Match: {}",
                     i, x2, expected, matches);
        }
    }
}
```

A.7 problem2.rs – LFSR Analysis Program

```rust
use statrs::statistics::{Data, Distribution};
use solutions::{RandomGenerator, Lfsr};
use plotters::prelude::*;

fn plot_histogram(
    samples: &[f64],
    filename: &str,
    title: &str,
    bins: usize,
) -> Result<(), Box<dyn std::error::Error>> {
    // Create histogram bins
    let mut counts = vec![0u32; bins];
    for &sample in samples {
        let bin = ((sample * bins as f64).floor() as usize).min(bins - 1);
        counts[bin] += 1;
    }

    let max_count = *counts.iter().max().unwrap() as f64;

    // Set up the drawing area
    let root = BitMapBackend::new(filename, (800, 600)).into_drawing_area
        ();
    root.fill(&WHITE)?;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30))
        .margin(10)
        .x_label_area_size(40)
        .y_label_area_size(50)
        .build_cartesian_2d(0.0..1.0, 0.0..(max_count * 1.1))?;

    chart
        .configure_mesh()
        .x_desc("Value")
        .y_desc("Count")
        .draw()?;

    // Draw histogram bars
    chart.draw_series(
        counts.iter().enumerate().map(|(i, &count)| {
            let x0 = i as f64 / bins as f64;
            let x1 = (i + 1) as f64 / bins as f64;
            Rectangle::new([(x0, 0.0), (x1, count as f64)], BLUE.mix(0.6).
                filled())
        }),
    )?;

    root.present()?;
    println!("Histogram saved to: {}", filename);
    Ok(())
}

```

```rust
fn plot_3d_scatter(
    samples: &[f64],
    filename: &str,
    title: &str,
    max_points: usize,
) -> Result<(), Box<dyn std::error::Error>> {
    // Create triplets from consecutive samples
    let triplets: Vec<(f64, f64, f64)> = samples
        .windows(3)
        .step_by(1)
        .take(max_points)
        .map(|w| (w[0], w[1], w[2]))
        .collect();

    // Set up the drawing area
    let root = BitMapBackend::new(filename, (1024, 768)).into_drawing_area
       ();
    root.fill(&WHITE)?;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30))
        .margin(10)
        .build_cartesian_3d(0.0..1.0, 0.0..1.0, 0.0..1.0)?;

    // Rotate the view to better see structure
    chart.with_projection(|mut pb| {
        pb.pitch = 0.8;
        pb.yaw = 0.5;
        pb.scale = 0.9;
        pb.into_matrix()
    });

    chart.configure_axes().draw()?;

    // Draw the points
    chart.draw_series(
        triplets
            .iter()
            .map(|&(x, y, z)| Circle::new((x, y, z), 2, BLUE.filled())),
    )?;

    root.present()?;
    println!("Plot saved to: {}", filename);
    Ok(())
}

fn main() {
    println!("PROBLEM 2: LFSR (Xorshift)");
    println!("==========================");

    // LFSR parameters: a=13, b=17, c=5
    let seed = 1;
    let a = 13;
    let b = 17;
```

```
    let c = 5;
    let m = u32::MAX;  // Maximum value for u32

    let mut lfsr = Lfsr::new(seed, a, b, c, m);

    println!("Parameters: a={}, b={}, c={}, m=2^32-1", a, b, c);
    println!("Seed: {}\n", seed);

    // Generate first few samples to show output
    println!("First 5 random numbers:");
    for i in 0..5 {
        println!("  Random Number {}: {:.10}", i, lfsr.next_uniform());
    }

    // Generate large number of samples for statistics
    let n = 100_000;
    println!("\nGenerating {} samples for analysis...", n);

    let samples = lfsr.generate_samples(n);
    let data = Data::new(samples.clone());

    // Calculate statistics
    let mean = data.mean().unwrap();
    let std_dev = data.std_dev().unwrap();

    println!("\nStatistics:");
    println!("  Mean:    {:.6} (expected: 0.5)", mean);
    println!("  Std Dev: {:.6} (expected: {:.6})", std_dev, (1.0/12.0_f64)
        .sqrt());

    // Generate histogram
    println!("\nGenerating histogram...");
    plot_histogram(&samples, "lfsr_histogram.png", "LFSR (Xorshift) -
        Histogram", 50)
        .expect("Failed to create histogram");

    // Generate 3D scatter plot
    println!("Generating 3D scatter plot...");
    let plot_samples = lfsr.generate_samples(30000);
    plot_3d_scatter(&plot_samples, "lfsr_3d.png", "LFSR (Xorshift) - 3D
        Scatter", 10000)
        .expect("Failed to create 3D scatter plot");

    println!("\nDone! Check the generated PNG files.");
}
```

A.8 problem3.rs – PCG32 Analysis Program

```
use statrs::statistics::{Data, Distribution};
use solutions::{RandomGenerator, Pcg32};
use plotters::prelude::*;

fn plot_histogram(
```

```rust
    samples: &[f64],
    filename: &str,
    title: &str,
    bins: usize,
) -> Result<(), Box<dyn std::error::Error>> {
    // Create histogram bins
    let mut counts = vec![0u32; bins];
    for &sample in samples {
        let bin = ((sample * bins as f64).floor() as usize).min(bins - 1);
        counts[bin] += 1;
    }

    let max_count = *counts.iter().max().unwrap() as f64;

    // Set up the drawing area
    let root = BitMapBackend::new(filename, (800, 600)).into_drawing_area
       ();
    root.fill(&WHITE)?;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30))
        .margin(10)
        .x_label_area_size(40)
        .y_label_area_size(50)
        .build_cartesian_2d(0.0..1.0, 0.0..(max_count * 1.1))?;

    chart
        .configure_mesh()
        .x_desc("Value")
        .y_desc("Count")
        .draw()?;

    // Draw histogram bars
    chart.draw_series(
        counts.iter().enumerate().map(|(i, &count)| {
            let x0 = i as f64 / bins as f64;
            let x1 = (i + 1) as f64 / bins as f64;
            Rectangle::new([(x0, 0.0), (x1, count as f64)], BLUE.mix(0.6).
               filled())
        }),
    )?;

    root.present()?;
    println!("Histogram saved to: {}", filename);
    Ok(())
}

fn plot_3d_scatter(
    samples: &[f64],
    filename: &str,
    title: &str,
    max_points: usize,
) -> Result<(), Box<dyn std::error::Error>> {
    // Create triplets from consecutive samples
```

```rust
    let triplets: Vec<(f64, f64, f64)> = samples
        .windows(3)
        .step_by(1)
        .take(max_points)
        .map(|w| (w[0], w[1], w[2]))
        .collect();

    // Set up the drawing area
    let root = BitMapBackend::new(filename, (1024, 768)).into_drawing_area
        ();
    root.fill(&WHITE)?;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30))
        .margin(10)
        .build_cartesian_3d(0.0..1.0, 0.0..1.0, 0.0..1.0)?;

    // Rotate the view
    chart.with_projection(|mut pb| {
        pb.pitch = 0.8;
        pb.yaw = 0.5;
        pb.scale = 0.9;
        pb.into_matrix()
    });

    chart.configure_axes().draw()?;

    // Draw the points
    chart.draw_series(
        triplets
            .iter()
            .map(|&(x, y, z)| Circle::new((x, y, z), 2, BLUE.filled())),
    )?;

    root.present()?;
    println!("Plot saved to: {}", filename);
    Ok(())
}

fn main() {
    println!("PROBLEM 3: PCG32 (Permuted Congruential Generator)");
    println!("==================================================");
    println!("PCG32 is a modern PRNG with excellent statistical properties
        .");
    println!("It uses a 64-bit LCG internally with output permutation (XSH
        -RR).\n");

    let seed = 42; // Classic seed choice!
    let mut pcg = Pcg32::new(seed);

    println!("Seed: {}\n", seed);

    // Generate first few samples to show output
    println!("First 5 random numbers:");
```

```
    for i in 0..5 {
        println!("  Random Number {}: {:.10}", i, pcg.next_uniform());
    }

    // Generate large number of samples for statistics
    let n = 100_000;
    println!("\nGenerating {} samples for analysis...", n);

    let samples = pcg.generate_samples(n);
    let data = Data::new(samples.clone());

    // Calculate statistics
    let mean = data.mean().unwrap();
    let std_dev = data.std_dev().unwrap();

    println!("\nStatistics:");
    println!("  Mean:    {:.6} (expected: 0.5)", mean);
    println!("  Std Dev: {:.6} (expected: {:.6})", std_dev, (1.0/12.0_f64)
        .sqrt());

    // Generate histogram
    println!("\nGenerating histogram...");
    plot_histogram(&samples, "pcg32_histogram.png", "PCG32 - Histogram",
        50)
        .expect("Failed to create histogram");

    // Generate 3D scatter plot
    println!("Generating 3D scatter plot...");
    let plot_samples = pcg.generate_samples(30000);
    plot_3d_scatter(&plot_samples, "pcg32_3d.png", "PCG32 - 3D Scatter",
        10000)
        .expect("Failed to create 3D scatter plot");

    println!("\nDone! Check the generated PNG files.");
    println!("\nNote: PCG32 should show excellent uniformity and no
        visible");
    println!("correlation patterns in the 3D plot - much better than basic
         LCGs!");
}
```

A.9 problem4.rs – Rule 30 Analysis Program

```
use statrs::statistics::{Data, Distribution};
use solutions::{RandomGenerator, Rule30};
use plotters::prelude::*;

fn plot_histogram(
    samples: &[f64],
    filename: &str,
    title: &str,
    bins: usize,
) -> Result<(), Box<dyn std::error::Error>> {
    // Create histogram bins
```

```rust
    let mut counts = vec![0u32; bins];
    for &sample in samples {
        let bin = ((sample * bins as f64).floor() as usize).min(bins - 1);
        counts[bin] += 1;
    }

    let max_count = *counts.iter().max().unwrap() as f64;

    // Set up the drawing area
    let root = BitMapBackend::new(filename, (800, 600)).into_drawing_area
        ();
    root.fill(&WHITE)?;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30))
        .margin(10)
        .x_label_area_size(40)
        .y_label_area_size(50)
        .build_cartesian_2d(0.0..1.0, 0.0..(max_count * 1.1))?;

    chart
        .configure_mesh()
        .x_desc("Value")
        .y_desc("Count")
        .draw()?;

    // Draw histogram bars
    chart.draw_series(
        counts.iter().enumerate().map(|(i, &count)| {
            let x0 = i as f64 / bins as f64;
            let x1 = (i + 1) as f64 / bins as f64;
            Rectangle::new([(x0, 0.0), (x1, count as f64)], BLUE.mix(0.6).
                filled())
        }),
    )?;

    root.present()?;
    println!("Histogram saved to: {}", filename);
    Ok(())
}

fn plot_3d_scatter(
    samples: &[f64],
    filename: &str,
    title: &str,
    max_points: usize,
) -> Result<(), Box<dyn std::error::Error>> {
    // Create triplets from consecutive samples
    let triplets: Vec<(f64, f64, f64)> = samples
        .windows(3)
        .step_by(1)
        .take(max_points)
        .map(|w| (w[0], w[1], w[2]))
        .collect();
```

```rust

    // Set up the drawing area
    let root = BitMapBackend::new(filename, (1024, 768)).into_drawing_area
        ();
    root.fill(&WHITE)?;

    let mut chart = ChartBuilder::on(&root)
        .caption(title, ("sans-serif", 30))
        .margin(10)
        .build_cartesian_3d(0.0..1.0, 0.0..1.0, 0.0..1.0)?;

    // Rotate the view
    chart.with_projection(|mut pb| {
        pb.pitch = 0.8;
        pb.yaw = 0.5;
        pb.scale = 0.9;
        pb.into_matrix()
    });

    chart.configure_axes().draw()?;

    // Draw the points
    chart.draw_series(
        triplets
            .iter()
            .map(|&(x, y, z)| Circle::new((x, y, z), 2, BLUE.filled())),
    )?;

    root.present()?;
    println!("Plot saved to: {}", filename);
    Ok(())
}

fn main() {
    println!("PROBLEM 4: Rule 30 Cellular Automaton");
    println!("=====================================");
    println!("Rule 30 is an elementary cellular automaton discovered by
        Stephen Wolfram.");
    println!("Uses 32 cells with periodic boundaries; each state IS the
        32-bit random number.\n");

    let seed = 42u32;
    let mut rule30 = Rule30::new(seed);

    println!("Seed: {}", seed);
    println!("Cells: 32 (packed as u32)\n");

    // Generate first few samples to show output
    println!("First 5 random numbers:");
    for i in 0..5 {
        println!("  Random Number {}: {:.10}", i, rule30.next_uniform());
    }

    // Generate large number of samples for statistics
```

```
    let n = 100_000;
    println!("\nGenerating {} samples for analysis...", n);

    let samples = rule30.generate_samples(n);
    let data = Data::new(samples.clone());

    // Calculate statistics
    let mean = data.mean().unwrap();
    let std_dev = data.std_dev().unwrap();

    println!("\nStatistics:");
    println!("  Mean:    {:.6} (expected: 0.5)", mean);
    println!("  Std Dev: {:.6} (expected: {:.6})", std_dev, (1.0/12.0_f64)
        .sqrt());

    // Generate histogram
    println!("\nGenerating histogram...");
    plot_histogram(&samples, "rule30_histogram.png", "Rule 30 - Histogram"
        , 50)
        .expect("Failed to create histogram");

    // Generate 3D scatter plot
    println!("Generating 3D scatter plot...");
    let plot_samples = rule30.generate_samples(30000);
    plot_3d_scatter(&plot_samples, "rule30_3d.png", "Rule 30 - 3D Scatter"
        , 10000)
        .expect("Failed to create 3D scatter plot");

    println!("\nDone! Check the generated PNG files.");
    println!("\nNote: This implementation uses bitwise operations on a u32
        , making it very fast!");
    println!("Rule 30 exhibits chaotic behavior - Wolfram used it in
        Mathematica's PRNG.");
}
```