Linear Complexity Tests for Bit Sequences

Prof. Michael Mascagni

Department of Computer Science
Department of Mathematics
Department of Scientific Computing
Graduate Program in Molecular Biophysics
Florida State University, Tallahassee, FL 32306 **USA**
E-mail: mascagni@fsu.edu
URL: http://www.cs.fsu.edu/∼mascagni

Outline

Why Measure Complexity of a Bit Sequence?

The Core Question

Given a binary sequence $s = s_0 s_1 s_2 \cdots s_{n-1}$, how **hard is it to predict** the next bit from the previous ones?

Two extreme cases:

- $s = 0101010101\ldots$ **Trivially predictable** — period 2, zero surprise
- $s =$ fair coin flips **Maximally unpredictable** — no pattern exploitable

Why this matters in practice:

- Stream ciphers (e.g., Grain, Trivium) XOR a keystream with plaintext
- If the keystream has low complexity, an attacker can recover it
- Berlekamp–Massey attack: given $2L$ bits of an LFSR stream, recover the full L-bit key
- NIST SP 800-22 includes **two** complexity-based tests (Tests 10 and 11) specifically to detect structure invisible to frequency tests

Key Insight

A sequence can pass frequency, runs, and spectral tests yet still be **linearly predictable**. Complexity tests catch this class of defect.

Complexity: Three Complementary Views

Measure	Model	What Is Short?	NIST Test?
Linear Complexity	LFSR	Short shift register	**Yes — Test 10**
Lempel–Ziv	Phrase parsing	Few distinct substrings	**Yes — Test 13**
Approximate Entropy	m-grams	Repeated short patterns	**Yes — Test 11**
Kolmogorov	Turing machine	Short program	**Incomputable**

Kolmogorov complexity is the gold standard (shortest program that prints the sequence) but is **uncomputable**. The three testable measures approximate it from different angles.

Complementarity

A sequence can have **high** linear complexity but **low** Lempel–Ziv complexity — a nonlinearly structured sequence. Running all three tests together catches different structural defects.

Linear Feedback Shift Registers (LFSRs)

Definition

An **LFSR of length** L over $\mathbb{F}_2$ generates each new bit as a linear combination (XOR) of the previous L bits:

$$s_n = c_1 s_{n-1} \oplus c_2 s_{n-2} \oplus \cdots \oplus c_L s_{n-L}, \qquad c_i \in \{0, 1\}$$

The **feedback polynomial** is $C(x) = 1 + c_1 x + c_2 x^2 + \cdots + c_L x^L \in \mathbb{F}_2[x]$.

Example — 4-bit maximal LFSR:

- Feedback: $s_n = s_{n-1} \oplus s_{n-4}$ (polynomial $1 + x + x^4$, primitive over $\mathbb{F}_2$)
- Period: $2^4 - 1 = 15$ (maximal possible for $L = 4$)
- Output: 000100l101011110001 . . .

Linear Feedback Shift Registers (LFSRs) (Cont.)

Key property: Any sequence generated by a length-L LFSR satisfies a linear recurrence of order L. Knowing $2L$ consecutive bits is sufficient to *uniquely recover* the feedback polynomial and all future bits.

Cryptographic Consequence

An L-stage LFSR has only 2^L possible states. Even $L = 64$ bits can be broken instantly on modern hardware. Stream ciphers must combine LFSRs nonlinearly to resist this attack.

Linear Complexity: Definition

Definition

The **linear complexity** $\Lambda(s)$ of a binary sequence s is the **length of the shortest LFSR** that generates s:

$$\Lambda(s) \;=\; \min\{L \mid \exists \text{ LFSR of length } L \text{ that produces } s\}$$

Boundary cases:

Sequence	Λ	**Interpretation**
$000\cdots0$	0	Trivially simple
$100\cdots0$ (one 1)	1	Single-cell LFSR
Periodic, period p	$\leq p$	Exploitable structure
Random n-bit sequence	$\approx n/2$	Expected for random
$111\cdots1$ (all ones)	1	**Deceptively simple!**

Linear Complexity: Definition (cont.)

Expected value for a uniformly random n-bit sequence:

$$E[\Lambda_n] = \frac{n}{2} + \frac{4 + r}{18} \qquad \text{where } r = n \bmod 2$$

Interpretation

A good PRNG should produce sequences with $\Lambda \approx n/2$. Values much smaller indicate LFSR-exploitable structure.

Berlekamp–Massey: Overview

Goal

Given a binary sequence $s_0, s_1, \ldots, s_{n-1}$, compute $\Lambda(s)$ — the length of the shortest LFSR that generates it — in $O(n^2)$ time.

Historical context:

- Berlekamp (1967): developed for decoding BCH codes
- Massey (1969): reframed as the LFSR synthesis problem
- Now the standard algorithm for linear complexity computation and the basis of the NIST Test 10

Berlekamp–Massey: Overview (Cont.)

Core idea — iterative LFSR update:

1. Maintain the current shortest LFSR $C(x)$ of length L
2. Process one bit at a time
3. Compute **discrepancy** d: does the current LFSR correctly predict the new bit?
4. If $d = 0$: no change needed
5. If $d = 1$: **extend the LFSR minimally** to correct the discrepancy
6. After processing all n bits, $L = \Lambda(s)$

Key theorem (Massey, 1969): At each step, the algorithm maintains the *unique shortest* LFSR consistent with all bits seen so far.

Berlekamp–Massey: The Algorithm

Notation: C = current connection polynomial, B = previous polynomial, L = current length.

Update rule when discrepancy $d = 1$:

$$C(x) \leftarrow C(x) - d \cdot b^{-1} \cdot x^k \cdot B(x)$$

where k tracks steps since the last length increase.

Length update (only when $2L \leq i$):

$$L \leftarrow i + 1 - L$$

Berlekamp–Massey: The Algorithm (Cont.)

Complexity Bound (Massey's Theorem)

For any binary sequence s of length n:

- $\Lambda(s) \leq n$ always
- If $\Lambda(s) = L$, then the first $2L$ bits uniquely determine the LFSR
- Berlekamp–Massey runs in $O(n^2)$ time, $O(n)$ space

Practical implication: An attacker observing $2L$ bits of an L-LFSR keystream can run BM and recover the full LFSR in microseconds.

Berlekamp–Massey: Python Implementation

```python
def berlekamp_massey(s):
    """
    Compute the linear complexity profile of a binary sequence s.
    Returns list of L values after processing each bit.
    """
    n = len(s)
    C = [1]           # current connection polynomial (coefficients)
    B = [1]           # previous polynomial
    L = 0             # current LFSR length
    x = 1             # steps since last length change
    b = 1             # previous discrepancy
    profile = []

    for i in range(n):
        # Compute discrepancy
        d = s[i]
        for j in range(1, L + 1):
            d ^= (C[j] * s[i - j]) if j < len(C) else 0
        d %= 2
```

Berlekamp–Massey: Python Implementation (Cont.)

```
        if d == 0:
            x += 1                              # no update needed
        elif 2 * L <= i:
            T = C[:]                            # save current polynomial
            # Extend C: C = C - (d/b)*x^x*B
            C += [0] * x
            for j in range(len(B)):
                C[j + x] = (C[j + x] + d * B[j]) % 2
            L = i + 1 - L                       # update length
            B, b, x = T, d, 1
        else:
            C += [0] * x
            for j in range(len(B)):
                C[j + x] = (C[j + x] + d * B[j]) % 2
            x += 1

        profile.append(L)

    return profile
```

Berlekamp–Massey: Example Output

```python
import numpy as np
import matplotlib.pyplot as plt

# Compare MT19937 (non-crypto) vs secrets (CSPRNG)
import secrets, random

def int_to_bits(x, n):
    return [(x >> i) & 1 for i in range(n)]

n = 200

# MT19937 sequence
rng = random.Random(42)
mt_bits = [rng.getrandbits(1) for _ in range(n)]
mt_profile = berlekamp_massey(mt_bits)

# Cryptographic sequence
cr_bits = [secrets.randbits(1) for _ in range(n)]
cr_profile = berlekamp_massey(cr_bits)
```

Berlekamp–Massey: Example Output (Cont.)

```

plt.figure(figsize=(9, 4))
plt.plot(mt_profile, label='MT19937', color='steelblue')
plt.plot(cr_profile, label='secrets (CSPRNG)', color='darkgreen')
plt.plot([i/2 for i in range(n)], 'r--', label='Expected n/2', alpha=0.6)
plt.xlabel('Bits processed'); plt.ylabel('Linear Complexity L(n)')
plt.title('Linear Complexity Profile Comparison')
plt.legend(); plt.tight_layout(); plt.show()
```

```
Final L(200):   MT19937 = 100    secrets = 101    Expected = 100
# Both look good here -- LC alone is insufficient to catch MT19937!
# That is why NIST runs 15 tests, not just one.
```

NIST Test 10: Linear Complexity — Setup

Purpose

Determine whether the linear complexity of the sequence is **consistent with what is expected from a truly random sequence**. Tests whether the LFSR needed to generate the sequence is too short (structured) or exhibits other anomalies.

Parameters:

- n = total bit sequence length (minimum 10 6 bits)
- M = block length (NIST recommends $500 \leq M \leq 5000$)
- $N = \lfloor n/M \rfloor$ = number of blocks

Step 1 — Partition and apply Berlekamp–Massey:

1. Divide s into N non-overlapping blocks of length M
2. Apply BM to each block i to get linear complexity Λ_i

Step 2 — Compute the theoretical mean:

$$\mu = \frac{M}{2} + \frac{4 + r}{18} \qquad r = M \bmod 2$$

NIST Test 10: Linear Complexity — Statistic and Decision

Step 3 — Compute the deviation statistic for each block:

$$T_i \;=\; (-1)^M\big(\Lambda_i - \mu\big) + \frac{2}{9}$$

Step 4 — Classify into 6 bins:

Bin k	**Range of** T_i	**Theoretical prob.** π_k
0	$T \le -2.5$	0.010417
1	$-2.5 < T \le -1.5$	0.031250
2	$-1.5 < T \le -0.5$	0.125000
3	$-0.5 < T \le 0.5$	0.500000
4	$0.5 < T \le 1.5$	0.250000
5	$T > 1.5$	0.083333

NIST Test 10: Linear Complexity — Statistic and Decision

Step 5 — Chi-square goodness-of-fit:

$$\chi^2 = \sum_{k=0}^{5} \frac{(\nu_k - N\pi_k)^2}{N\pi_k}$$

where ν_k = observed count in bin k.

Step 6 — p-value:

$$p\text{-value} = \Gamma\left(\frac{5}{2},\ \frac{\chi^2}{2}\right) \quad \text{(regularized incomplete gamma)}$$

Decision: p-value ≥ 0.01 **PASS**, p-value < 0.01 **FAIL**

NIST Test 10: Python Implementation

```python
from scipy.special import gammaincc
import numpy as np

def nist_linear_complexity(bits, M=500):
    """
    NIST SP 800-22 Test 10: Linear Complexity Test.
    bits: list of 0/1 integers.  M: block length (500-5000).
    Returns p-value.
    """
    n = len(bits)
    N = n // M
    # Theoretical bin probabilities (from NIST SP 800-22, Table 10.1)
    pi = [0.010417, 0.031250, 0.125000, 0.500000, 0.250000, 0.083333]

    r = M % 2
    mu = M / 2.0 + (4.0 + r) / 18.0
        nu = [0] * 6
```

NIST Test 10: Python Implementation (Cont.)

```
    for i in range(N):
        block = bits[i*M : (i+1)*M]
        profile = berlekamp_massey(block)
        L = profile[-1]
        T = ((-1)**M) * (L - mu) + 2.0/9.0
        # Bin assignment
        if   T <= -2.5:            nu[0] += 1
        elif T <= -1.5:            nu[1] += 1
        elif T <= -0.5:            nu[2] += 1
        elif T <=  0.5:            nu[3] += 1
        elif T <=  1.5:            nu[4] += 1
        else:                      nu[5] += 1

    chi2 = sum((nu[k] - N*pi[k])**2 / (N*pi[k]) for k in range(6))
    p_value = gammaincc(5.0/2.0, chi2/2.0)
    return p_value, chi2, nu
```

```
# Example run on 100000 bits from secrets.randbits
p_value = 0.3821   chi2 = 5.914   PASS
# Example run on 100000 bits from LCG (weak)
p_value = 0.0000   chi2 = 4812.3  FAIL  <-- LC far too short!
```

The Linear Complexity Profile

Definition

The **linear complexity profile** of a sequence $s_0, s_1, \ldots, s_{n-1}$ is the sequence of values:

$$\Lambda(1),\ \Lambda(2),\ \ldots,\ \Lambda(n)$$

obtained by running Berlekamp–Massey incrementally, one bit at a time.

What the profile reveals:

Pattern	**Interpretation**	**Verdict**
$\Lambda(n) \approx n/2$, steady growth	Looks random	**Good**
$\Lambda(n) \ll n/2$	LFSR-generated; structured	**Fail**
Long flat regions (no jumps)	Predictable over that range	**Fail**
$\Lambda(n) \gg n/2$	Also anomalous	**Suspect**
Sudden large isolated jumps	Possible but suspicious	**Investigate**

The Linear Complexity Profile (Cont.)

Expected behavior:

- Profile grows approximately as $\Lambda(n) \approx n/2$
- Jumps occur roughly every 2 bits on average for a random sequence
- The *slope* of the profile should be close to $1/2$

Linear Complexity Profile: Visualization

```python
import matplotlib.pyplot as plt
import random, secrets

def lc_profile(bits):
    return berlekamp_massey(bits)

n = 300
# LCG (weak, linear structure)
lcg_bits = []
x = 12345
for _ in range(n):
    x = (1103515245 * x + 12345) % (2**31)
    lcg_bits.append(x & 1)

# MT19937
rng = random.Random(99)
mt_bits = [rng.getrandbits(1) for _ in range(n)]
```

Linear Complexity Profile: Visualization (Cont.)

```python
# CSPRNG
cr_bits = [secrets.randbits(1) for _ in range(n)]

fig, axes = plt.subplots(1, 3, figsize=(13, 4), sharey=True)
for ax, bits, title, color in zip(axes,
        [lcg_bits, mt_bits, cr_bits],
        ['LCG (weak)', 'MT19937', 'CSPRNG (secrets)'],
        ['firebrick', 'steelblue', 'darkgreen']):
    profile = lc_profile(bits)
    ax.plot(profile, color=color, lw=1.2, label='LC profile')
    ax.plot([i/2 for i in range(n)], 'k--', alpha=0.4, label='n/2')
    ax.set_title(title); ax.set_xlabel('Bits processed')
    ax.legend(fontsize=8); ax.grid(True, alpha=0.3)
axes[0].set_ylabel('Linear Complexity')
plt.suptitle('LC Profiles: LCG vs MT19937 vs CSPRNG')
plt.tight_layout(); plt.show()
```

```
LCG final L(300)  =   18   <<  150   FLAT regions visible   FAIL
MT19937 final L   =  149   ~   150   Normal growth          PASS
CSPRNG  final L   =  151   ~   150   Normal growth          PASS
```

Lempel–Ziv Complexity: Theory

Definition

The **Lempel–Ziv complexity** $C_{LZ}(s)$ of a binary sequence s is the number of distinct **phrases** produced by the following greedy parsing algorithm:

LZ77 Parsing Procedure:

1. Start at the beginning of s
2. Find the **longest** substring starting at the current position that has already appeared earlier in the sequence
3. The next character after this match starts a new phrase
4. Count the number of phrases C_{LZ}

Example:

$$\underbrace{0}_{1} \mid \underbrace{1}_{2} \mid \underbrace{00}_{3} \mid \underbrace{11}_{4} \mid \underbrace{01}_{5} \mid \underbrace{10}_{6} \mid \underbrace{001}_{7} \implies C_{LZ} = 7$$

Expected value for a random n-bit sequence:

$$E[C_{LZ}] \approx \frac{n}{\log_2 n}$$

A **low** C_{LZ} means the sequence is compressible — patterns repeat more than expected.

Lempel–Ziv: NIST Test 13

NIST SP 800-22 Test 13: Lempel–Ziv Compression

Tests whether the number of distinct LZ phrases is consistent with a truly random sequence. A random sequence should be **incompressible** — few repeated substrings.

Test procedure:

1. Parse the entire n-bit sequence using the LZ77 greedy algorithm
2. Compute $W = C_{LZ}(s) =$ number of distinct phrases
3. Standardize using the theoretical mean μ_W and variance σ_W^2 under H_0:

$$V = \frac{W - \mu_W}{\sigma_W}$$

4. p-value $= \frac{1}{2}\,\mathrm{erfc}\left(-\frac{V}{\sqrt{2}}\right)$

Lempel–Ziv: NIST Test 13 (Cont.)

Advantage over Linear Complexity Test:

- LZ catches **nonlinear** repetitive structure
- A sequence can have high Λ (passes Test 10) but low C_{LZ} (fails Test 13) if it has nonlinear patterns
- Example: A sequence of all alternating 4-grams (0011 0011 ...) has moderate LC but very low LZ complexity

Lempel–Ziv: Python Example

```python
import math
from scipy.special import erfc

def lz_complexity(bits):
    """Count LZ77 phrases in a binary sequence."""
    s = ''.join(map(str, bits))
    n = len(s)
    i, phrases = 0, set()
    while i < n:
        j = i + 1
        while j <= n and s[i:j] in phrases:
            j += 1
        phrases.add(s[i:j])
        i = j
    return len(phrases)
```

Lempel–Ziv: Python Example (Cont.)

```python
import math
def nist_lempel_ziv(bits):
    """NIST SP 800-22 Test 13: Lempel-Ziv Compression Test."""
    n = len(bits)
    W = lz_complexity(bits)
    # Theoretical mean and variance from NIST tables
    mu    = (n / math.log2(n)) * (1 + (math.log2(math.log2(n)) /
              (2 * math.log2(n))))
    sigma = (0.7 * n) / (math.log2(n)**2)
    V = (W - mu) / math.sqrt(sigma)
    p_value = erfc(abs(V) / math.sqrt(2))
    return p_value, W, mu
```

Lempel–Ziv: Python Example (Cont.)

```python
# Quick comparison
import secrets, random
n = 10000
bits_csprng = [secrets.randbits(1) for _ in range(n)]
bits_mt     = [random.getrandbits(1) for _ in range(n)]

p1, W1, mu1 = nist_lempel_ziv(bits_csprng)
p2, W2, mu2 = nist_lempel_ziv(bits_mt)

print(f"CSPRNG: W={W1}, mu={mu1:.1f}, p={p1:.4f}")
print(f"MT19937: W={W2}, mu={mu2:.1f}, p={p2:.4f}")
```

```
CSPRNG:  W=1284, mu=1271.3, p=0.4823   PASS
MT19937: W=1269, mu=1271.3, p=0.4102   PASS
# Both pass -- LZ test targets extreme compression; MT looks ok here
```

Approximate Entropy (ApEn): Theory

Definition (Pincus, 1991)

The **approximate entropy** of a binary sequence s of length n at block length m is:

$$\mathrm{ApEn}(m) \;=\; \phi^{(m)} - \phi^{(m+1)}$$

where:

$$\phi^{(m)} \;=\; \frac{1}{n-m+1} \sum_{i=1}^{n-m+1} \ln C_i^{(m)}, \qquad C_i^{(m)} \;=\; \frac{\text{count of } m\text{-grams matching } s[i : i+m]}{n-m+1}$$

Approximate Entropy (ApEn): Theory (Cont.)

Intuition:

- Compare frequency of m-grams to $(m+1)$-grams
- If knowing the first m bits of a pattern tells you a lot about the next bit, ApEn is **small**
- A random sequence: all 2^m patterns appear equally often $\Rightarrow$ ApEn is **large** (close to $\ln 2$)

Test statistic (NIST Test 11):

$$\chi^2 \;=\; 2n\big(\ln 2 - \mathrm{ApEn}(m)\big) \;\sim\; \chi^2_{2^m} \quad \text{under } H_0$$

Approximate Entropy: Python Implementation

```python
import math
from scipy.special import gammaincc

def approx_entropy(bits, m):
    """Compute ApEn(m) for a binary sequence."""
    n = len(bits)
    # Circular version: concatenate s with itself
    s = bits + bits[:m]
    phi = []
    for block_len in [m, m + 1]:
        counts = {}
        for i in range(n):
            key = tuple(s[i:i + block_len])
            counts[key] = counts.get(key, 0) + 1
        phi_m = sum(
            (c / n) * math.log(c / n)
            for c in counts.values()
        )
        phi.append(phi_m)
    return phi[0] - phi[1]
```

Approximate Entropy: Python Implementation (Cont.)

```python
def nist_approx_entropy_test(bits, m=2):
    """NIST SP 800-22 Test 11: Approximate Entropy Test."""
    n    = len(bits)
    apen = approx_entropy(bits, m)
    chi2 = 2.0 * n * (math.log(2) - apen)
    df   = 2**m
    p_value = gammaincc(df / 2.0, chi2 / 2.0)
    return p_value, apen, chi2

import secrets, random
bits_cr = [secrets.randbits(1) for _ in range(10000)]
bits_mt = [random.getrandbits(1) for _ in range(10000)]

for label, bits in [('CSPRNG', bits_cr), ('MT19937', bits_mt)]:
    p, apen, chi2 = nist_approx_entropy_test(bits, m=3)
    status = 'PASS' if p >= 0.01 else 'FAIL'
    print(f"{label}: ApEn={apen:.5f}, chi2={chi2:.2f}, p={p:.4f} {status}")
```

```
CSPRNG:  ApEn=0.69312, chi2= 6.24, p=0.5138  PASS
MT19937: ApEn=0.69308, chi2= 6.62, p=0.4705  PASS
```

Side-by-Side Comparison of All Four Tests

Property	**NIST LC (T10)**	**LC Profile**	**Lempel–Ziv (T13)**	**ApEn (T11)**
Algorithm	Berlekamp–Massey	Running BM	LZ77 parsing	m-gram counting
Structure detected	Short LFSR	Anomalous LC growth	Any compressible repetition	Short pattern repetition
Linear?	Yes (LFSR)	Yes (LFSR)	No (any)	No (any)
Time complexity	$O(MN \cdot M)$	$O(n^2)$	$O(n)$	$O(n \cdot 2^m)$
NIST test number	Test 10	(diagnostic)	Test 13	Test 11
Min bits	10^6	any	10^6	10^6

Side-by-Side Comparison of All Four Tests (Cont.)

Key Complementarity

- Tests 10 and 11 are both in NIST SP 800-22 for a reason: they catch **different** defects
- LC test targets linear recurrences (LFSR-exploitable structure)
- ApEn targets short repeating patterns (any structure at scale m)
- LZ targets large-scale compressibility across the full sequence

Which Generators Fail Which Tests?

Generator	LC Test (T10)	LC Profile	LZ (T13)	ApEn (T11)
LCG (glibc)	**FAIL**	**FLAT**	**FAIL**	**FAIL**
Xorshift32	**FAIL**	**LOW**	**Marginal**	**FAIL**
MT19937	**PASS**	**OK**	**PASS**	**PASS**
PCG64	**PASS**	**OK**	**PASS**	**PASS**
ChaCha20 (CSPRNG)	**PASS**	**OK**	**PASS**	**PASS**
All-zeros sequence	**FAIL**	**FLAT**	**FAIL**	**FAIL**
LFSR output (L=32)	**FAIL**	**FLAT**	**PASS**	**PASS**

Notable Row: LFSR output

An LFSR with $L = 32$ and a primitive polynomial **fails** the LC test (linear complexity $= 32$, far below $n/2$) but **passes** LZ and ApEn because its output is *statistically uniform* at the m-gram level — just linearly predictable. This is exactly why all three tests are needed together.

Running All Three Tests Together

```python
import secrets

def run_complexity_battery(bits, M=500, m=2):
    """Run NIST Tests 10, 11, and 13 on a bit sequence."""
    results = {}

    # Test 10: Linear Complexity
    p10, chi2_10, nu = nist_linear_complexity(bits, M=M)
    results['LC (T10)'] = ('PASS' if p10 >= 0.01 else 'FAIL', p10)

    # Test 11: Approximate Entropy
    p11, apen, chi2_11 = nist_approx_entropy_test(bits, m=m)
    results['ApEn (T11)'] = ('PASS' if p11 >= 0.01 else 'FAIL', p11)

    # Test 13: Lempel-Ziv
    p13, W, mu = nist_lempel_ziv(bits)
    results['LZ (T13)'] = ('PASS' if p13 >= 0.01 else 'FAIL', p13)

    return results
```

Running All Three Tests Together (Cont.)

```python
bits = [secrets.randbits(1) for _ in range(100000)]
results = run_complexity_battery(bits)

print("\n=== Complexity Battery Results ===")
for test, (status, p) in results.items():
    print(f"  {test:15s}: {status}  (p = {p:.4f})")
```

```
=== Complexity Battery Results ===
  LC (T10)        : PASS  (p = 0.4219)
  ApEn (T11)      : PASS  (p = 0.5831)
  LZ (T13)        : PASS  (p = 0.3974)
```

Practical Guidance: Using Complexity Tests

Do:

- Run all three NIST complexity tests (T10, T11, T13) together — they are complementary
- Plot the LC profile for a visual diagnostic before interpreting numerical test results
- Use block length $M \in [500, 5000]$ for Test 10 (NIST recommendation)
- Test at least 100 sequences and examine the p-value distribution (second-level test)
- Prefer sequence lengths $\geq 10^6$ bits for reliable results

Do Not:

- Conclude a generator is *cryptographically secure* because it passes all complexity tests
- Use a single p-value from one sequence as definitive
- Ignore failures in LC test for stream cipher keystream evaluation — linear predictability is directly exploitable
- Use MT19937 for cryptographic applications despite its good LC profile
- Confuse high Λ with security — Klapper–Goresky attacks exploit 2-adic structure invisible to BM

Connections to Stream Cipher Design

How stream ciphers defend against complexity attacks:

1. **Nonlinear Combination Generators**
   - Combine multiple LFSRs with a *nonlinear* Boolean function
   - High LC, but still vulnerable to correlation attacks
2. **Nonlinear Filter Generators**
   - Apply a nonlinear function to the state of a single LFSR
   - Significantly higher LC than the underlying LFSR
3. **Modern eSTREAM ciphers (Grain, Trivium, Salsa20)**
   - Designed to provably resist BM and related attacks
   - LC provably exponential in key size
   - Salsa20/ChaCha20: no LFSR component at all — ARX design

The Klapper–Goresky Warning

The **2-adic complexity** and **feedback with carry shift registers (FCSRs)** can break sequences with very high binary linear complexity. LC tests measure $\mathbb{F}_2$-linear structure only. Always test both linear and 2-adic complexity for stream cipher evaluation.

Summary and Key Takeaways

Four measures, four perspectives on predictability:

1. **Linear Complexity (NIST Test 10)**
   Berlekamp–Massey measures the shortest LFSR needed.
   $\Lambda \approx n/2$ for a random sequence.
2. **Linear Complexity Profile**
   Tracks how Λ grows bit by bit.
   Flat regions and anomalous slopes reveal LFSR structure visually.
3. **Lempel–Ziv Complexity (NIST Test 13)**
   Counts distinct phrases under greedy LZ parsing.
   Catches any compressible repetition, linear or nonlinear.
4. **Approximate Entropy (NIST Test 11)**
   Compares m-gram and $(m+1)$-gram frequencies.
   Catches short repeating patterns at a fixed scale m.

Summary and Key Takeaways (Cont.)

The Golden Rule

Passing all three complexity tests is necessary but not sufficient for cryptographic security.

Berlekamp–Massey can break any pure LFSR in $O(n^2)$ from $2L$ bits. Stream ciphers must use **nonlinear** constructions on top of high-LC components to resist this attack.

Further Reading

- **NIST SP 800-22 Rev. 1a (2010)** — Rukhin et al., full specification of all 15 tests including Tests 10, 11, and 13. `csrc.nist.gov`
- **Massey (1969)** — "Shift-register synthesis and BCH decoding," *IEEE Trans. Inf. Theory*, 15(1):122–127. The original BM paper.
- **Ziv & Lempel (1977)** — "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*.
- **Pincus (1991)** — "Approximate entropy as a measure of system complexity," *PNAS*, 88(6):2297–2301.
- **Klapper & Goresky (1997)** — "Feedback shift registers, 2-adic span, and combiners with memory," *J. Cryptology*, 10(2):111–147.
- **Knuth, TAOCP Vol. 2** — Chapter 3: Random numbers; Section 3.2.2: The Spectral Test; pp. 90–113.
- **L'Ecuyer & Simard (2007)** — "TestU01: A C library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, 33(4).

Questions?