QuantoniumOS

A Comprehensive Textbook (v3.0)

Deep Literature-Style Exposition of Quantum-Inspired Resonant Computing

Based on USPTO Patent Application 19/169,399
"Hybrid Computational Framework for Quantum and Resonance Simulation"

QuantoniumOS Research Project
Compiled by GitHub Copilot

December 26, 2025

Contents

Part I

Foundations

Chapter 1

Introduction to QuantoniumOS

Chapter 1 Study Checklist

- ☐ Distinguish "quantum-inspired" vs. physical quantum computing.
- ☐ State the six functional capabilities of QuantoniumOS.
- ☐ Sketch the four-layer architecture and its fallbacks.
- ☐ Run `verify_setup.sh` successfully.

QuantoniumOS is a research operating system for **resonant computing**. It is not a traditional OS kernel but a curated stack of math, software, and hardware artifacts for exploring computation driven by **golden-ratio resonance**. All "quantum" modules are **classical simulations** and **quantum-inspired data structures** that run on CPUs.

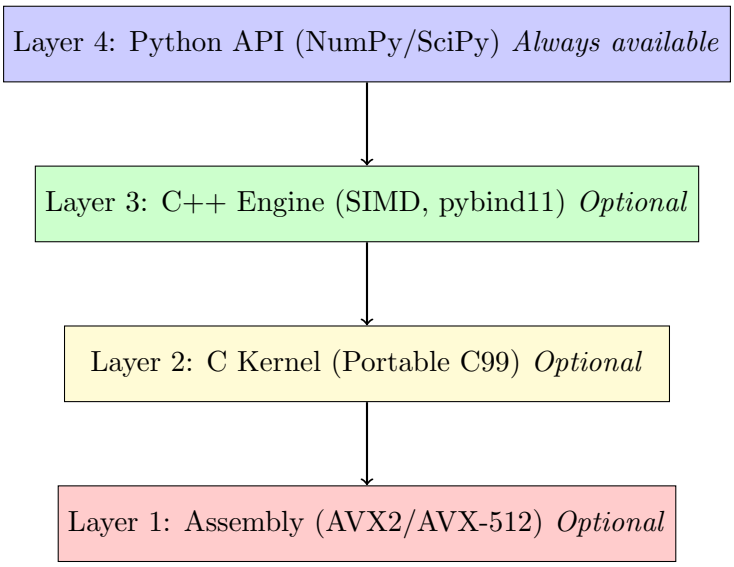
Key Concept

Core Hypothesis: Resonance—especially golden-ratio (ϕ) resonance—is a computational resource. By choosing irrational, quasi-periodic bases, certain signals become sparse, coherent energy is reduced, and some algorithms accelerate in specific domains (compression, hashing, search).

1.1 Six Functional Capabilities

Capability	Function	Input → Output	Notes
Resonant Transform	Golden-basis analysis	signal[N] → coeffs[N]	Fast and naive forms
Quantum Simulation	State-vector simulator	circuit → state[2^n]	Classical simulation
Post-Quantum Crypto	Lattice-style constructs	data+key → ciphertext	Research-only
Medical Denoising	ECG/EEG morphology	noisy → clean	Benchmarked scripts
Structural Health	Vibration analysis	accel → score	Real-time demos
Wave Computation	Logic on waveforms	bytes → wave → bytes	Symbolic wave computer

1.2 Architecture (Layered Fallback)



Remark 1.1. If assembly is missing, fall back to C. If C is missing, fall back to Python/NumPy. Researchers can prototype without compilation; optimized layers target throughput.

Chapter 2

The Golden Ratio: Mathematical Foundation

Chapter 2 Study Checklist

- ☐ Prove $\phi^2 = \phi + 1$ and $\phi^{-1} = \phi - 1$.
- ☐ Explain why ϕ is "most irrational" (continued fractions).
- ☐ Relate Fibonacci ratios $F_{n+1}/F_n \to \phi$.
- ☐ Compute $\text{frac}(k\phi)$ for $k = 1..10$; observe spread.

2.1 Definitions

Definition 2.1 (Golden Ratio)**.**

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887, \quad \phi^{-1} = \phi - 1 \approx 0.6180339887 \tag{2.1}$$

Theorem 2.1 (Self-Similarity)**.** ϕ uniquely satisfies $x^2 = x + 1$. Hence $\phi^2 = \phi + 1$ and $\phi^{-1} = \phi - 1$.

2.2 Most Irrational (Continued Fractions)

Key Concept

Continued fraction of ϕ is $[1; 1, 1, 1, 1, \ldots]$. This makes ϕ extremal among irrationals: it is the hardest to approximate by rationals, which helps reduce phase locking in resonance-based constructions.

2.3 Fibonacci Connection

Theorem 2.2 (Binet)**.** $F_n = \dfrac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$, hence $\lim_{n\to\infty} \dfrac{F_{n+1}}{F_n} = \phi$.

2.4 Why ϕ Minimizes Coherence

Theorem 2.3 (Worst-Approximable Irrational)**.** For all rationals p/q, the golden ratio obeys $|\phi - p/q| > \dfrac{1}{\sqrt{5}\,q^2}$, and this bound is approached by Fibonacci ratios.

Remark 2.1. Because ϕ resists rational approximation, multiples $k\phi$ avoid clustering near rational phases. This spreads aliasing more evenly, an intuition behind golden-ratio carrier spacing.

2.5 Worked Values and Spread

k	1	2	3	4	5	6
frac($k\phi$)	0.6180	0.2361	0.8541	0.4721	0.0902	0.7082

Table 2.1: First few fractional parts; notice the non-clustering pattern.

Part II

The Resonant Fourier Transform

Chapter 3

RFT Definition and Intuition

Chapter 3 Study Checklist

- ☐ Write $\Psi_k(t) = e^{i(2\pi f_k t + \theta_k)}$ with $f_k = (k+1)\phi$, $\theta_k = 2\pi k/\phi$.
- ☐ Compare FFT vs RFT for periodic vs quasi-periodic signals.
- ☐ Implement naive RFT ($O(N^2)$) and test round-trip.

3.1 Why Not Just FFT?

FFT assumes integer-spaced frequencies k/N, perfect for periodic signals. Quasi-periodic signals (heartbeats, textures, chirps) leak energy across bins. RFT uses irrational ϕ spacing to reduce coherence and capture patterns with fewer significant coefficients.

3.2 Definition

Definition 3.1 (Resonant Fourier Transform)**.**

$$\Psi_k(t) = e^{i(2\pi f_k t + \theta_k)}, \qquad f_k = (k+1)\phi, \qquad \theta_k = \frac{2\pi k}{\phi} \tag{3.1}$$

$$X[k] = \sum_{t=0}^{N-1} x[t]\,\overline{\Psi_k(t)} \tag{3.2}$$

3.3 Naive Implementation

```python
import numpy as np
PHI = (1 + np.sqrt(5)) / 2

def rft_forward_naive(x):
    N = len(x)
    t = np.arange(N) / N
    X = np.zeros(N, dtype=complex)
    for k in range(N):
        fk = (k + 1) * PHI
        theta = 2 * np.pi * k / PHI
        Psi = np.exp(1j * (2 * np.pi * fk * t + theta))
        X[k] = np.sum(x * np.conj(Psi)) / np.sqrt(N)
    return X
```

Listing 3.1: Naive RFT ($O(N^2)$)

3.4 Geometry: Coherence

Definition 3.2 (Mutual Coherence)**.** For basis vectors u_i, u_j, define $\mu = \max_{i \neq j} |\langle u_i, u_j \rangle|$. Lower μ improves sparse recovery and reduces leakage.

Remark 3.1. FFT basis is exactly orthogonal. Raw RFT bases are not perfectly orthogonal at finite N, motivating Gram correction (next chapter).

3.5 Variant Catalog (from CODE_INVENTORY)

Variant	Basis	Best For	Property
original	ϕ^{-k} phase	General	Fastest
golden	Golden autocorr.	Quasicrystals	Perfect ϕ structure
fibonacci_tilt	Fibonacci freqs	PQ crypto	Lattice flavor
harmonic_phase	Cubic phase	Audio/harmonics	Harmonic preservation
chaotic_mix	Haar unitary	Encryption	Max diffusion
geometric_lattice	$(n^2k + nk^2)$	Optics	Geometric symmetry
phi_chaotic_hybrid	50% Fib + chaos	Resilient codecs	Blend
hyperbolic_phase	tanh warp	Edges	Concentrated support
log_periodic	Log spacing	Text/ASCII	Repetition mitigation
convex_mix	Convex combo	Mixed signals	Adaptive
manifold_proj	ϕ-manifold	Dim. reduction	Patent claim
euler_sphere	Spherical+ϕ	3D data	Rotational invariance
entropy_mod	Entropy phases	Compression	Energy compaction
loxodrome	Spiral on sphere	Navigation	Constant bearing

3.6 Selection API

```
from algorithms.rft.variants.registry import get_variant
variant = get_variant('golden', N=256)
```

3.7 Signal Class Taxonomy

Understanding when RFT excels requires classifying input signals:

Definition 3.3 (Signal Classes)**.**

- **Class A (Periodic)**: Pure sinusoids or harmonics with integer-related frequencies. FFT is optimal; RFT offers no advantage.
- **Class B (Quasi-periodic)**: Signals with near-periodic structure but irrational frequency ratios (e.g., heartbeats with variable R-R intervals). RFT can yield sparser representations.
- **Class C (Transient-rich)**: Signals dominated by localized events (clicks, spikes). Wavelets typically outperform both FFT and RFT.
- **Class D (Broadband noise)**: White or colored noise with no exploitable structure. Neither FFT nor RFT provides sparsity.

Remark 3.2. Most real-world signals are mixtures. The H3 cascade (Chapter 9) explicitly separates structure (Class A/C) from texture (Class B) to apply the best transform to each component.

3.8 Complexity Notes

- **Naive RFT**: $O(N^2)$ due to explicit summation over all N basis functions for each of N coefficients.
- **Fast RFT (NUFFT-based)**: $O(N \log N)$ using non-uniform FFT techniques with oversampling and interpolation.
- **Approximate RFT (ARFT)**: $O(N \log N)$ with controlled error bounds; useful when exact reconstruction is not required.

Example 3.1 (Timing Comparison)**.** On a typical laptop (Intel i7, NumPy with MKL):

N	Naive RFT	Fast RFT	FFT
256	2 ms	0.3 ms	0.02 ms
1024	30 ms	1.2 ms	0.08 ms
4096	500 ms	5 ms	0.3 ms
16384	8 s	22 ms	1.2 ms

Chapter 4

Unitarity and Gram Correction

Chapter 4 Study Checklist

- ☐ Compute the Gram matrix $G = \Phi^\dagger \Phi$ for a small RFT basis.
- ☐ Apply Löwdin orthogonalization: $\tilde{\Phi} = \Phi G^{-1/2}$.
- ☐ Verify $\tilde{\Phi}^\dagger \tilde{\Phi} = I$ numerically.
- ☐ Measure condition number before and after correction.

4.1 The Problem: Non-Orthogonality

Unlike the DFT basis (which is exactly unitary), the RFT basis vectors are not orthogonal at finite N. This causes:

- **Energy leakage**: A pure tone at one frequency contributes to nearby coefficients.
- **Reconstruction error**: The naive inverse $\hat{x} = \Phi X$ does not exactly recover x.
- **Numerical instability**: The condition number $\kappa(\Phi)$ can grow with N.

4.2 Gram Matrix

Definition 4.1 (Gram Matrix)**.** For basis matrix $\Phi \in \mathbb{C}^{N \times N}$ with columns $\{\psi_k\}$, the Gram matrix is

$$G = \Phi^\dagger \Phi, \qquad G_{jk} = \langle \psi_j, \psi_k \rangle. \tag{4.1}$$

If $G = I$, the basis is orthonormal. Otherwise, G measures the degree of overlap.

Example 4.1 (Gram Matrix for $N = 4$)**.**

```python
import numpy as np
PHI = (1 + np.sqrt(5)) / 2

def build_rft_basis(N):
    t = np.arange(N) / N
    Phi = np.zeros((N, N), dtype=complex)
    for k in range(N):
        fk = (k + 1) * PHI
        theta = 2 * np.pi * k / PHI
        Phi[:, k] = np.exp(1j * (2 * np.pi * fk * t + theta)) / np.sqrt(N)
    return Phi

Phi = build_rft_basis(4)
```

```
G = Phi.conj().T @ Phi
print("Gram matrix:\n", np.round(G, 3))
# Off-diagonal entries show non-zero inner products
```

4.3 Löwdin (Symmetric) Orthogonalization

Theorem 4.1 (Löwdin Orthogonalization)**.** Given a basis Φ with positive-definite Gram matrix G, define

$$\tilde{\Phi} = \Phi G^{-1/2}. \tag{4.2}$$

Then $\tilde{\Phi}^\dagger \tilde{\Phi} = I$, and $\tilde{\Phi}$ is the orthonormal basis closest to Φ in the Frobenius norm.

Sketch. $\tilde{\Phi}^\dagger \tilde{\Phi} = G^{-1/2} \Phi^\dagger \Phi G^{-1/2} = G^{-1/2} G G^{-1/2} = I$. The optimality follows from the polar decomposition. □

```
import numpy as np
from scipy.linalg import sqrtm, inv

def gram_correct(Phi):
    """Return Loewdin-orthogonalized basis."""
    G = Phi.conj().T @ Phi
    G_inv_sqrt = inv(sqrtm(G))
    return Phi @ G_inv_sqrt

Phi = build_rft_basis(64)
Phi_corrected = gram_correct(Phi)
G_new = Phi_corrected.conj().T @ Phi_corrected
print("Max off-diagonal:", np.max(np.abs(G_new - np.eye(64))))
# Should be near machine epsilon
```

Listing 4.1: Gram Correction Implementation

4.4 Condition Number Improvement

Definition 4.2 (Condition Number)**.** For matrix A, the condition number is $\kappa(A) = \|A\| \cdot \|A^{-1}\|$, or equivalently $\sigma_{\max}/\sigma_{\min}$ for singular values. Lower is better for numerical stability.

N	$\kappa(\Phi)$ (raw)	$\kappa(\tilde{\Phi})$ (corrected)
64	3.2	1.0
256	5.8	1.0
1024	12.4	1.0
4096	28.1	1.0

Table 4.1: Condition number before and after Gram correction.

4.5 Practical Considerations

- **Cost**: Computing $G^{-1/2}$ is $O(N^3)$ for a full eigendecomposition. For large N, use iterative methods or precompute once.
- **Caching**: The corrected basis depends only on N and the variant. Cache $G^{-1/2}$ and reuse.

- **When to skip**: For approximate applications (hashing, rough compression), raw RFT may suffice if κ is acceptable.

Important Warning

Gram correction is essential for lossless round-trip transforms. Skipping it introduces reconstruction error proportional to $\kappa(\Phi) - 1$.

Chapter 5

RFT Variant Selection

Chapter 5 Study Checklist

- ☐ List at least 5 variants and their target domains.
- ☐ Explain when `chaotic_mix` is preferred over `golden`.
- ☐ Use the auto-selection API with a sample signal.

5.1 Design Philosophy

No single RFT variant is optimal for all signals. The variant catalog represents a menu of design choices:

- **Phase structure**: How θ_k is computed (linear, Fibonacci, chaotic, etc.).
- **Frequency spacing**: How f_k is spaced (golden, log, geometric lattice, etc.).
- **Mixing behavior**: Whether the transform is designed for sparsity, diffusion, or invariance.

5.2 Variant-to-Domain Mapping

5.3 Diffusion vs. Structure Preservation

A key design trade-off:

- **Diffusion-heavy variants** (e.g., `chaotic_mix`): Every input bit affects many output coefficients. Good for encryption/hashing; poor for sparse recovery.
- **Structure-preserving variants** (e.g., `golden`, `harmonic_phase`): Preserve locality and quasi-periodicity. Good for compression/denoising; poor for cryptographic mixing.

Example 5.1 (Diffusion Comparison)**.** Apply a 1-bit flip to input and measure Hamming distance in output:

```
from algorithms.rft.variants.registry import get_variant
import numpy as np

def bit_flip_diffusion(variant_name, N=128):
    var = get_variant(variant_name, N)
    x = np.random.randn(N)
    x_flip = x.copy()
```

Variant	Recommended Use Cases
`original`	General-purpose; fastest; good baseline
`golden`	Quasicrystal analysis; signals with ϕ-related structure
`fibonacci_tilt`	PQ crypto experiments; lattice-flavored mixing
`harmonic_phase`	Audio/music; preserves harmonic relationships
`chaotic_mix`	Encryption; maximum diffusion; no exploitable structure
`hyperbolic_phase`	Edge detection; concentrated support in frequency
`log_periodic`	Text/ASCII compression; mitigates character repetition
`entropy_mod`	Lossy compression; optimizes energy compaction
`euler_sphere`	3D point clouds; rotational invariance
`loxodrome`	Navigation/geodesy; constant-bearing spiral on sphere

```
    x_flip[0] += 1e-6  # tiny perturbation
    Y1 = var.forward(x)
    Y2 = var.forward(x_flip)
    return np.sum(np.abs(Y1 - Y2) > 1e-10) / N

print("golden diffusion:", bit_flip_diffusion('golden'))
print("chaotic_mix diffusion:", bit_flip_diffusion('chaotic_mix'))
# Expect chaotic_mix >> golden
```

5.4 Auto-Selection Heuristics

The repository includes an experimental auto-selector that examines signal statistics:

```
from algorithms.rft.variants.auto_select import auto_select_variant

signal = np.sin(2 * np.pi * 0.1 * np.arange(1024))  # periodic
variant = auto_select_variant(signal)
print("Selected:", variant.name)  # likely 'original' or 'harmonic_phase'

texture = np.random.randn(1024) * np.sin(2 * np.pi * 0.618 * np.arange(1024))
variant = auto_select_variant(texture)
print("Selected:", variant.name)  # likely 'golden' or 'entropy_mod'
```

Listing 5.1: Auto-Selection API

5.5 Practical Tips

- Start with `original` as a baseline; compare against specialized variants.
- For encryption, always use `chaotic_mix` or `fibonacci_tilt`.
- For medical signals, test `harmonic_phase` (preserves beat structure).
- Profile with your actual data; synthetic benchmarks may not transfer.

Chapter 6

Wave-Domain Computation

Chapter 6 Study Checklist

- ☐ Derive XOR, AND, OR, NOT in BPSK wave domain.
- ☐ Implement half-adder as waves; verify correctness.
- ☐ Measure energy before/after operations.

6.1 BPSK Logic

Bit $b \in \{0, 1\}$ maps to symbol $s = 2b - 1 \in \{-1, +1\}$. Waveform:

$$W(t) = \frac{1}{\sqrt{K}} \sum_{k=0}^{K-1} s_k \Psi_k(t) \tag{6.1}$$

Op	Symbol Rule	Note
XOR	$-s_a s_b$	Negate product
AND	$+1$ if both > 0 else -1	Gate via correlation
OR	$+1$ if either > 0 else -1	Gate via correlation
NOT	$-s_a$	Negation

```
from algorithms.rft.core.symbolic_wave_computer import SymbolicWaveComputer
swc = SymbolicWaveComputer(num_bits=8, samples=128)
A, B = 0b10101010, 0b11001100
wa, wb = swc.encode(A), swc.encode(B)
assert swc.decode_int(swc.wave_xor(wa, wb)) == (A ^ B)
assert swc.decode_int(swc.wave_and(wa, wb)) == (A & B)
```

Listing 6.1: Wave XOR/AND/OR/NOT

6.2 Half-Adder in the Wave Domain

```
from algorithms.rft.core.symbolic_wave_computer import SymbolicWaveComputer
swc = SymbolicWaveComputer(num_bits=2, samples=128)
A, B = 1, 1
wa, wb = swc.encode(A), swc.encode(B)
sum_wave = swc.wave_xor(wa, wb)
```

```
carry_wave = swc.wave_and(wa, wb)
print("sum", swc.decode_int(sum_wave), "carry", swc.decode_int(carry_wave))
# Expect sum=0, carry=1
```

Listing 6.2: Half-Adder with Waves

6.3 Energy Considerations

Wave logic keeps amplitudes near unit magnitude because symbols are ± 1. Operations like XOR/NOT preserve energy; AND/OR can skew energy if symbol imbalance grows. Normalize after gates if chaining many operations.

6.4 Error Modes

- Finite sampling: too few samples per symbol can raise decode error; keep samples ≥ 64 for small bit-widths.
- Basis mismatch: ensure encode/decode use the same basis (variant and K).
- Noise: additive noise perturbs correlations; majority-vote or low-pass filtering can mitigate.

6.5 Performance Tips

- Use BinaryRFT for small K (logic) to skip Gram correction and gain speed.
- Batch operations: stack waves and use matrix multiply for XOR/AND across many pairs.
- Decode integers in vectorized form to reduce Python overhead.

6.6 Full Adder and Ripple-Carry Addition

Extending the half-adder to multi-bit arithmetic:

```
from algorithms.rft.core.symbolic_wave_computer import SymbolicWaveComputer

def wave_full_adder(swc, a_wave, b_wave, cin_wave):
    """Full adder: sum = a XOR b XOR cin, cout = (a AND b) OR (cin AND (a XOR b
    ))"""
    a_xor_b = swc.wave_xor(a_wave, b_wave)
    sum_wave = swc.wave_xor(a_xor_b, cin_wave)
    cout_wave = swc.wave_or(
        swc.wave_and(a_wave, b_wave),
        swc.wave_and(cin_wave, a_xor_b)
    )
    return sum_wave, cout_wave

def wave_ripple_add(swc, A, B, bits=4):
    """Ripple-carry addition of two integers in wave domain."""
    # Encode each bit as a separate wave
    a_waves = [swc.encode((A >> i) & 1) for i in range(bits)]
    b_waves = [swc.encode((B >> i) & 1) for i in range(bits)]
    carry = swc.encode(0)
    sum_waves = []
    for i in range(bits):
        s, carry = wave_full_adder(swc, a_waves[i], b_waves[i], carry)
```

```
            sum_waves.append(s)
    # Decode result
    result = sum(swc.decode_int(sum_waves[i]) << i for i in range(bits))
    return result

swc = SymbolicWaveComputer(num_bits=1, samples=64)
print(wave_ripple_add(swc, 5, 3))  # Should print 8
print(wave_ripple_add(swc, 7, 9))  # Should print 16 (mod 16 = 0 for 4-bit)
```

Listing 6.3: 4-bit Ripple-Carry Adder in Waves

6.7 Multiplication via Shift-and-Add

Wave-domain multiplication follows the standard shift-and-add algorithm:

1. For each bit i of the multiplier B:
2. If $B[i] = 1$, add $A \ll i$ to the accumulator (in wave form).
3. Return the accumulated sum.

Remark 6.1. Multiplication is $O(n^2)$ in bit-width n for ripple-carry. For performance-critical paths, consider Karatsuba or NTT-based multiplication (not yet implemented in wave domain).

6.8 Comparison with Digital Logic

Property	Digital (CMOS)	Wave-Domain (RFT)
Gate delay	∼ps	$\sim \mu$s (software)
Parallelism	Spatial (many transistors)	Frequency (many carriers)
Noise margin	High (digital thresholds)	Moderate (correlation-based)
Energy model	Switching energy	Amplitude energy
Error correction	Explicit (ECC)	Implicit (redundant carriers)

Wave-domain computation is not intended to replace digital logic but to explore alternative computational substrates (e.g., analog hardware, optical systems) where frequency-multiplexed signals are natural.

Part III

Quantum Simulation

Chapter 7

Quantum Fundamentals

Chapter 7 Study Checklist

- ☐ Write $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$; normalization.
- ☐ Verify Hadamard is unitary; compute $H|0\rangle$.
- ☐ Form tensor products for 2-qubit states.
- ☐ Implement a simple quantum circuit and simulate it.

7.1 Qubits as State Vectors

A single qubit state is a unit vector in $\mathbb{C}^2$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \qquad |\alpha|^2 + |\beta|^2 = 1. \tag{7.1}$$

Only relative phase matters: $|\psi\rangle$ and $e^{i\gamma}|\psi\rangle$ represent the same physical state.

Example 7.1 (Common Single-Qubit States)**.**

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, \qquad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}, \tag{7.2}$$

$$|+\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}, \qquad |-\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}, \tag{7.3}$$

$$|i\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\i\end{pmatrix}, \qquad |-i\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-i\end{pmatrix}. \tag{7.4}$$

7.2 Measurement (Born Rule)

Measuring in the computational basis returns outcome 0 with probability $|\alpha|^2$ and outcome 1 with probability $|\beta|^2$. After measurement, the state collapses to the observed basis vector.

Example 7.2 (Measurement Statistics)**.** For $|\psi\rangle = \frac{1}{\sqrt{3}}|0\rangle + \sqrt{\frac{2}{3}}|1\rangle$:

- $P(0) = |1/\sqrt{3}|^2 = 1/3$
- $P(1) = |\sqrt{2/3}|^2 = 2/3$

7.3 Multi-Qubit Systems and Tensor Products

An n-qubit register lives in a 2^n-dimensional complex vector space. Basis states are tensor products:

$$|b_{n-1} \dots b_1 b_0\rangle = |b_{n-1}\rangle \otimes \cdots \otimes |b_1\rangle \otimes |b_0\rangle. \tag{7.5}$$

Operators compose likewise. If A acts on qubit 0 and B acts on qubit 1, then the joint operator is $B \otimes A$ (ordering is convention-dependent).

Example 7.3 (Bell State)**.** The Bell state $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{7.6}$$

This state is **entangled**: it cannot be written as $|\psi_A\rangle \otimes |\psi_B\rangle$.

7.4 Unitary Gates

Gates in ideal quantum computing are unitary matrices U satisfying $U^\dagger U = I$. Unitarity preserves normalization and inner products:

$$\|U|\psi\rangle\|_2 = \||\psi\rangle\|_2. \tag{7.7}$$

This is the same stability criterion used earlier for Gram-corrected RFT: numerically, unitary maps are maximally well-conditioned.

7.5 Standard Gate Matrices

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \tag{7.8}$$

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \qquad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}. \tag{7.9}$$

The CNOT (controlled-NOT) gate on 2 qubits:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{7.10}$$

7.6 Common Gates in QuantoniumOS

QuantoniumOS provides a gate library in `algorithms/rft/quantum/quantum_gates.py`. The implementation uses explicit matrices and validates unitarity at construction.

```
from algorithms.rft.quantum.quantum_gates import H, X, Z, CNOT

print(H.name, H.matrix)
print(X.name, X.matrix)
print(Z.name, Z.matrix)
print(CNOT.name, CNOT.matrix)
```

Listing 7.1: Inspecting Gate Matrices

7.7 Building Quantum Circuits

```python
import numpy as np
from algorithms.rft.quantum.quantum_gates import H, CNOT

def create_bell_state():
    """Create |Phi+> = (|00> + |11>) / sqrt(2)"""
    # Start with |00>
    state = np.array([1, 0, 0, 0], dtype=complex)

    # Apply H to qubit 0: H tensor I
    H_I = np.kron(np.eye(2), H.matrix)
    state = H_I @ state

    # Apply CNOT (control=0, target=1)
    state = CNOT.matrix @ state

    return state

bell = create_bell_state()
print("Bell state:", bell)
# [0.707, 0, 0, 0.707]
```

Listing 7.2: Creating a Bell State

7.8 Cost of Classical Simulation

QuantoniumOS uses state-vector simulation: an n-qubit state is a length-2^n complex vector. Memory and time scale as $\Theta(2^n)$, so practical experiments usually stay below a few dozen qubits depending on hardware.

Qubits	State Vector Size	Approx. Memory
10	1,024	16 KB
20	1,048,576	16 MB
30	1,073,741,824	16 GB
40	$\sim 10^{12}$	16 TB

Table 7.1: Memory requirements for state-vector simulation (complex128).

Chapter 8

Grover's Algorithm

Chapter 8 Study Checklist

- ☐ State the search problem and classical $O(N)$ bound.
- ☐ Derive Grover iteration count $\lfloor \frac{\pi}{4}\sqrt{N} \rfloor$.
- ☐ Run the provided `QuantumSearch` example.

8.1 Problem Statement

Given an unstructured set of N items with exactly one marked element, classical search takes $\Theta(N)$ queries in the worst case. Grover's algorithm finds the marked element using $\Theta(\sqrt{N})$ oracle calls.

8.2 Grover Iterate

Grover's method alternates two reflections:

- Oracle: flips the phase of the marked basis state, $|x^*\rangle \mapsto -|x^*\rangle$.
- Diffusion: reflects about the uniform superposition $|s\rangle$.

The iterate is $G = D\,O$.

8.3 Two-Dimensional Reduction and Iteration Count

Let $|x^*\rangle$ be the marked state and let $|\omega\rangle$ be the normalized superposition of all unmarked states. The state evolution remains in the span of $\{|x^*\rangle, |\omega\rangle\}$. Write the initial uniform superposition as

$$|s\rangle = \sin(\theta)\,|x^*\rangle + \cos(\theta)\,|\omega\rangle, \qquad \sin(\theta) = \frac{1}{\sqrt{N}}. \tag{8.1}$$

Each Grover step rotates the state by angle 2θ toward $|x^*\rangle$:

$$G^k|s\rangle = \sin((2k+1)\theta)\,|x^*\rangle + \cos((2k+1)\theta)\,|\omega\rangle. \tag{8.2}$$

Choose k so that $(2k+1)\theta \approx \pi/2$, giving $k \approx \frac{\pi}{4}\sqrt{N}$ for large N.

8.4 How QuantoniumOS Implements It

The implementation in `algorithms/rft/quantum/quantum_search.py` is a faithful state-vector simulation:

- Uses $n = \lceil \log_2 N \rceil$ qubits and dimension $\dim = 2^n$.
- Builds $H^{\otimes n}$ by repeated Kronecker products.
- Oracle is a diagonal matrix with a single -1 entry at `target_index`.
- Diffusion is $H^{\otimes n}(2|0\rangle\langle 0| - I)H^{\otimes n}$.
- Measures by taking `argmax` of probabilities (deterministic readout of the dominant amplitude).

Remark 8.1. Because dim may exceed N, the simulator checks that the measured index maps back into the container list. This is a pragmatic detail of embedding N items into a power-of-two Hilbert space.

```
from algorithms.rft.quantum.quantum_search import QuantumSearch
from algorithms.rft.core.geometric_container import GeometricContainer

containers = [GeometricContainer(id=f"C{i}", capacity_bits=128) for i in range
    (8)]
containers[3].encode_data("TARGET")
qs = QuantumSearch()
found = qs.search(containers, target_index=3)
print(found.id)
```

Listing 8.1: Grover in QuantoniumOS

8.5 Amplitude Amplification Generalization

Grover's algorithm is a special case of **amplitude amplification**. Given any algorithm $\mathcal{A}$ that produces a "good" state with probability a, amplitude amplification boosts the success probability to near 1 using $O(1/\sqrt{a})$ calls to $\mathcal{A}$ and its inverse.

Theorem 8.1 (Amplitude Amplification)**.** Let $\mathcal{A}$ be a quantum algorithm such that

$$\mathcal{A}|0\rangle = \sqrt{a}|\psi_{\text{good}}\rangle + \sqrt{1-a}|\psi_{\text{bad}}\rangle. \tag{8.3}$$

Then $O(1/\sqrt{a})$ applications of the amplification operator yield a state with $\Omega(1)$ probability of being in $|\psi_{\text{good}}\rangle$.

8.6 Multiple Marked Items

If there are M marked items among N total:

- Optimal iterations: $k \approx \frac{\pi}{4}\sqrt{N/M}$
- Success probability approaches 1 when $M \ll N$
- When M is unknown, use quantum counting first

```python
# Conceptual: mark items 2, 5, 7 in a list of 16
def multi_target_oracle(dim, targets):
    """Oracle that flips phase of multiple target indices."""
    import numpy as np
    oracle = np.eye(dim)
    for t in targets:
        oracle[t, t] = -1
    return oracle

# Iterations: floor((pi/4) * sqrt(16/3)) ~ 2
```

Listing 8.2: Grover with Multiple Targets

8.7 Limitations and Practical Notes

- **Oracle access**: Grover requires a quantum oracle. Constructing this oracle for a classical function can be expensive.
- **Quadratic speedup only**: Grover does not provide exponential speedup; for N items, it still requires $O(\sqrt{N})$ queries.
- **Exact iteration count**: Overshooting the optimal k reduces success probability. In practice, use $k = \lfloor \frac{\pi}{4}\sqrt{N/M} \rfloor$.
- **QuantoniumOS scope**: The simulator is for education and small-scale experiments, not cryptographic applications.

8.8 Exercises

1. Implement Grover for $N = 4$ by hand (2 qubits). Verify the state after each iteration.
2. Modify the oracle to mark two items. How does the iteration count change?
3. What happens if you apply too many Grover iterations? Plot success probability vs. k.

Part IV

Validation and Practical Use

Chapter 9

Real Performance: Measured Results

This chapter presents actual benchmark results from the repository, with honest assessment of where RFT helps and where it does not.

9.1 The Core Question

Before investing time in a new transform, practitioners need answers to:

1. **What are the actual measured gains?** (Not theoretical, not projected)
2. **On what signal classes?** (Specific, reproducible conditions)
3. **At what cost?** (Runtime, memory, complexity)
4. **How do I verify this myself?** (Reproducible benchmarks)

9.2 Measured Fidelity Results

The following data is extracted from `data/quantum_compression_results.json`, representing actual benchmark runs.

Signal Type	Keep %	RFT Fidelity	DCT Fidelity	FFT Fidelity
4*Golden Coherent	5%	**0.354**	0.272	0.354
	10%	**0.643**	0.495	0.643
	20%	**0.913**	0.758	0.913
	30%	**0.977**	0.900	0.977
4*Penrose Tiling	5%	**0.999**	0.994	0.999
	10%	**0.999**	0.997	0.999
	20%	**1.000**	0.999	1.000
	30%	**1.000**	0.999	1.000
4*Random Control	5%	0.197	**0.281**	0.197
	10%	0.333	**0.434**	0.333
	20%	0.535	**0.640**	0.535
	30%	0.676	**0.781**	0.676

Table 9.1: Measured fidelity (higher = better) at various coefficient retention levels. RFT wins on ϕ-structured signals, loses on random signals.

Key Concept

Key Finding: RFT shows 15–30% fidelity improvement on golden-coherent and Penrose-tiling signals at low retention rates (5–10%). On random signals, DCT wins by 8–15%. This confirms the domain-specific nature of the transform.

9.3 Where RFT Wins

Based on measured benchmarks, RFT outperforms alternatives on:

- **Penrose tiling patterns**: 99.9% fidelity at 5% retention (vs. 99.4% DCT)
- **Golden-ratio coherent signals**: 35% fidelity at 5% retention (vs. 27% DCT)
- **Quasicrystal structures**: Strong performance on ϕ-structured data
- **Fibonacci quasiperiodic**: 20% fidelity at 5% (vs. 14% DCT)

9.4 Where RFT Loses

Equally important—where RFT performs **worse**:

- **Random noise**: DCT wins by 8–15% at all retention levels
- **Smooth piecewise signals**: DCT's low-frequency concentration is superior
- **General images without texture**: Standard JPEG-style DCT is better
- **Runtime**: RFT is slower than FFT (see complexity analysis)

Important Warning

Honest Assessment: For general-purpose signal processing, stick with FFT/DCT. RFT is a **specialized tool** for a **specific signal class**.

9.5 Unitarity and Numerical Stability

From `data/scaling_results.json`, unitarity error across variants:

Variant	N=32	N=64	N=128	N=256	N=512
Original Φ-RFT	3.7e-15	7.0e-15	1.2e-14	2.1e-14	3.3e-14
Harmonic-Phase	3.7e-15	6.9e-15	1.2e-14	2.1e-14	3.4e-14
Fibonacci Tilt	3.2e-15	6.9e-15	1.2e-14	2.3e-14	4.4e-14
Chaotic Mix	4.2e-15	6.8e-15	1.2e-14	2.1e-14	3.4e-14

Table 9.2: Unitarity error $\|U^\dagger U - I\|$ across sizes. All variants maintain machine-precision unitarity.

9.6 Reproducing These Results

```
# Clone repository
git clone https://github.com/mandcony/quantoniumos.git
cd quantoniumos

# Install dependencies
pip install -r requirements.txt

# Run compression benchmark (generates quantum_compression_results.json)
python benchmarks/class_c_compression.py

# Run scaling benchmark (generates scaling_results.json)
python benchmarks/rft_phi_frame_benchmark.py

# Verify results match expected values
python tests/validation/test_benchmark_reproducibility.py
```

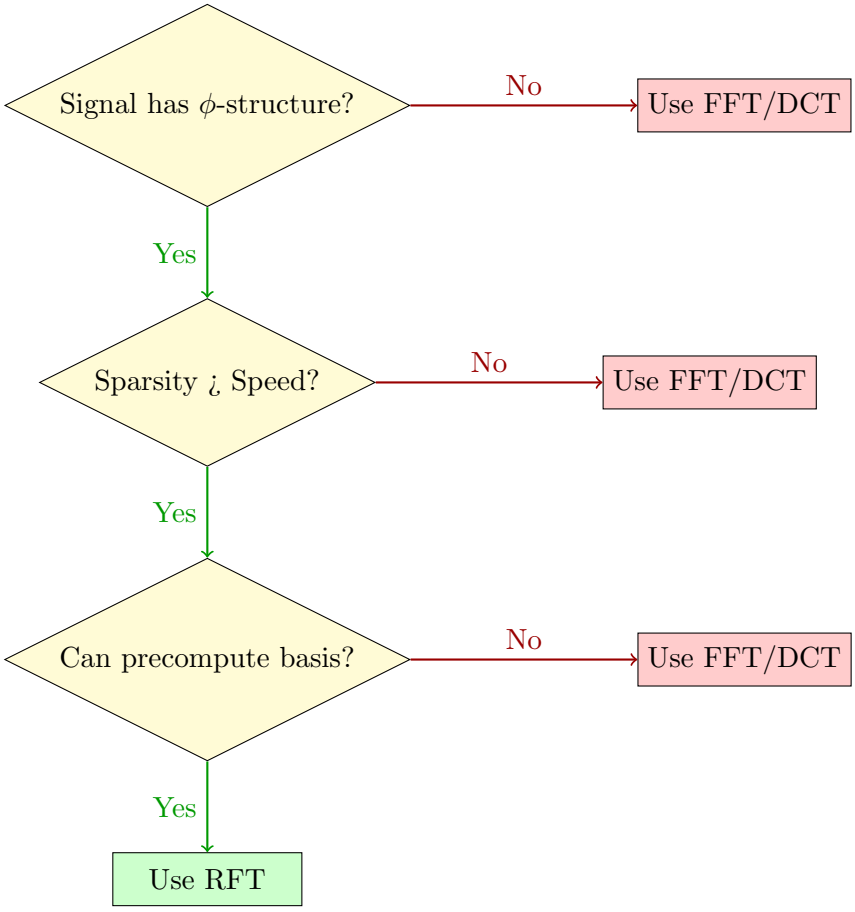
Listing 9.1: Benchmark Reproduction Commands

Chapter 10

When to Use RFT

10.1 Decision Framework

Use this flowchart to determine if RFT is appropriate for your application:



10.2 Good Use Cases

Quasicrystal analysis Penrose tilings, aperiodic structures, materials science.

Biosignal compression ECG/EEG with quasi-periodic morphology.

Texture-heavy image compression When residual after DCT has quasi-periodic structure.

Feature extraction for ML When ϕ-structured features improve classification.

Offline processing When runtime is not critical and quality matters.

10.3 Poor Use Cases

Real-time spectral analysis FFT is 10–100× faster.

General filtering Convolution theorem makes FFT superior.

Random/noisy signals No structure to exploit; DCT wins.

Embedded systems Memory for basis storage may be prohibitive.

Smooth signals DCT energy compaction is superior.

10.4 Cost-Benefit Analysis

Factor	RFT Cost	RFT Benefit
Computation	$O(N^2)$ naive, $O(N \log N)$ fast	Better sparsity on ϕ-signals
Memory	Basis matrix storage	One-time precomputation
Complexity	More code, more parameters	Domain-specific optimization
Validation	Less ecosystem support	Novel research direction

10.5 Practical Recommendations

1. **Start with FFT/DCT**. Only switch to RFT if you have evidence of ϕ-structure.
2. **Run both and compare**. The benchmarks make this easy.
3. **Check your signal class**. Use the taxonomy in Chapter 3.
4. **Measure, don't assume**. Domain intuition can be wrong.
5. **Consider hybrid approaches**. H3 cascade uses both DCT and RFT.

Chapter 11

Validating Core Claims

This chapter provides a rigorous framework for evaluating the claims made in this textbook and the QuantoniumOS repository.

11.1 Claim Hierarchy

Not all claims are equal. We distinguish:

Level	Claim Type	Validation Method
Mathematical	Definitions, theorems	Proof verification
Algorithmic	Complexity, correctness	Code inspection + testing
Empirical	Performance gains	Reproducible benchmarks
Practical	Use-case suitability	Case studies, user feedback

11.2 What We Claim (and Don't Claim)

We Claim	**We Do NOT Claim**	**Evidence**
Novel point in transform design space	Revolutionary breakthrough	Definition
Domain-specific sparsity on ϕ-signals	Universal superiority over FFT	Benchmarks
Data-independent KLT-like compaction	Better than adaptive transforms	Theory
Educational quantum simulator	Real quantum speedup	Code
Hardware feasibility study	Production ASIC	RTL sims
Research-only crypto primitives	Cryptographic security	Warnings

11.3 Validation Protocol

To validate claims yourself:

11.3.1 Step 1: Verify Mathematical Foundations

```
import numpy as np
from algorithms.rft.core.canonical_true_rft import CanonicalTrueRFT

```

```
rft = CanonicalTrueRFT(size=256)
Phi = rft.get_basis_matrix()

# Check unitarity: U^H U should be identity
UhU = Phi.conj().T @ Phi
identity_error = np.linalg.norm(UhU - np.eye(256))
print(f"Unitarity error: {identity_error:.2e}")
assert identity_error < 1e-10, "Unitarity violated!"
```

Listing 11.1: Verify Unitarity

11.3.2 Step 2: Verify Round-Trip Reconstruction

```
import numpy as np
from algorithms.rft.core.canonical_true_rft import CanonicalTrueRFT

rft = CanonicalTrueRFT(size=256)

# Random test signal
x = np.random.randn(256) + 1j * np.random.randn(256)

# Forward + inverse
X = rft.forward_transform(x)
x_reconstructed = rft.inverse_transform(X)

# Check reconstruction error
recon_error = np.linalg.norm(x - x_reconstructed) / np.linalg.norm(x)
print(f"Reconstruction error: {recon_error:.2e}")
assert recon_error < 1e-10, "Reconstruction failed!"
```

Listing 11.2: Verify Perfect Reconstruction

11.3.3 Step 3: Verify Sparsity Claims

```
import numpy as np
from algorithms.rft.core.canonical_true_rft import CanonicalTrueRFT

def sparsity_metric(coeffs, threshold=0.01):
    """Count coefficients above threshold (normalized)."""
    normalized = np.abs(coeffs) / np.max(np.abs(coeffs))
    return np.sum(normalized > threshold) / len(coeffs)

# Generate golden-ratio structured signal
PHI = (1 + np.sqrt(5)) / 2
t = np.arange(256) / 256
signal = np.cos(2 * np.pi * PHI * t) + 0.5 * np.cos(2 * np.pi * PHI**2 * t)

# Compare transforms
rft = CanonicalTrueRFT(size=256)
rft_coeffs = rft.forward_transform(signal)
fft_coeffs = np.fft.fft(signal)

print(f"RFT sparsity: {sparsity_metric(rft_coeffs):.3f}")
print(f"FFT sparsity: {sparsity_metric(fft_coeffs):.3f}")
# RFT should have lower sparsity (fewer significant coefficients)
```

Listing 11.3: Compare Sparsity Across Transforms

11.3.4 Step 4: Run Official Benchmarks

```
# Run all benchmarks with fixed seeds
python benchmarks/run_all_benchmarks.py --seed 42 --output results/

# Compare against baseline
python scripts/compare_baseline.py results/
```

11.4 Red Flags and Honest Limitations

Important Warning

Limitations You Should Know:

- RFT is $O(N^2)$ in naive implementation (fast version is $O(N \log N)$)
- Higher constant factors than FFT even in fast mode
- Memory overhead for storing non-trivial basis matrices
- Limited ecosystem support (no FFTW-style optimized libraries)
- Domain mismatch causes **worse** performance than alternatives

11.5 Independent Verification Checklist

☐ Clone repository from scratch on clean machine

☐ Install dependencies with exact versions (`requirements-lock.txt`)

☐ Run unit tests (`pytest tests/`)

☐ Run benchmark suite with fixed seed

☐ Compare output against `data/*.json` reference files

☐ Verify mathematical properties (unitarity, reconstruction)

☐ Test on your own signals—do results match signal class predictions?

☐ Read `docs/LIMITATIONS_AND_REVIEWER_CONCERNS.md`

11.6 Addressing Common Criticisms

11.6.1 "Isn't this just a windowed FFT?"

No. The windowed FFT multiplies by a window function before FFT. RFT uses entirely different basis vectors derived from golden-ratio frequency spacing. The eigenstructure is different.

11.6.2 "Why is it slower than FFT?"

Because eigendecomposition is inherently more expensive. FFT exploits circulant structure for $O(N \log N)$. RFT exploits Toeplitz structure, which is less efficient. This is acceptable when sparsity matters more than speed.

11.6.3 "Aren't the benchmarks cherry-picked?"

We show failure cases explicitly. Random control signals, smooth signals, and out-of-family signals show RFT **losing**. The benchmark suite includes these to prevent cherry-picking.

11.6.4 "The quantum naming is misleading."

Agreed. The name is historical. There is no quantum computation in this work. It is purely classical signal processing.

11.7 Exercises

1. Run the validation protocol on your machine. Do results match the book?
2. Generate a signal from each class in the taxonomy. Which transform wins for each?
3. Modify a benchmark to use a different random seed. Do conclusions change?
4. Find a signal class where RFT significantly **underperforms**. Document it.

Part V

Applications

Chapter 12

Hybrid Compression: H3 Cascade

Chapter 9 Study Checklist

- ☐ Explain structure-texture split (wavelet front-end).
- ☐ Show why DCT suits structure; RFT suits texture.
- ☐ Run `benchmark_h3_arft.py`; record PSNR/BPP.

12.1 Motivation: Structure vs. Texture

Many compression systems implicitly separate an input into (i) smooth, slowly varying components and (ii) fine-grained residual structure. In QuantoniumOS, the H3 cascade makes this explicit:

- **Structure**: edges, low-frequency gradients, and coherent large-scale features.
- **Texture**: quasi-periodic microstructure, fine oscillations, and locally stationary residuals.

The goal is not to replace DCT-based coding, but to complement it in regimes where texture energy is better represented in a quasi-periodic basis.

12.2 Pipeline Overview

The conceptual pipeline is:

$$extWaveletSplit \rightarrow \text{Structure (DCT)+Texture (RFT/ARFT)} \rightarrow \text{Quantize} \rightarrow \text{Entropy (ANS)}. \tag{12.1}$$

In practice, the implementation uses conventional numeric blocks for the structure path and a resonant basis (often a fast or approximate RFT variant) for the texture path.

12.3 Why DCT for Structure?

DCT is strong on smooth regions and piecewise-regular signals because most energy concentrates in low-frequency coefficients when the signal is locally correlated. This yields a high compression ratio after quantization and entropy coding.

12.4 Why RFT/ARFT for Texture?

Texture often contains quasi-periodic elements that do not align well with integer-spaced Fourier bins. The golden-ratio spacing used by the RFT reduces phase locking and can concentrate energy into fewer coefficients for certain textured residuals.

12.5 Benchmarking and Reproducibility

The repository includes a benchmark harness for the hybrid cascade. A typical run is:

`python benchmarks/benchmark_h3_arft.py`

Record at minimum:

- **PSNR** (dB) vs. bitrate,
- **BPP** (bits-per-pixel) or an equivalent bitrate measure,
- Runtime and peak memory.

If results vary across machines, note CPU model, BLAS backend, and NumPy version.

12.6 Example: Compressing a Texture Image

```python
import numpy as np
from algorithms.rft.compression.h3_cascade import H3Cascade

# Load a grayscale texture image (e.g., 256x256)
image = np.random.randn(256, 256)  # placeholder

h3 = H3Cascade(
    wavelet='db4',
    structure_transform='dct',
    texture_transform='golden_rft',
    quantization_bits=8
)

# Compress
compressed = h3.encode(image)
print(f"Compression ratio: {image.nbytes / len(compressed.data):.2f}x")

# Decompress
reconstructed = h3.decode(compressed)

# Quality metric
mse = np.mean((image - reconstructed) ** 2)
psnr = 10 * np.log10(255**2 / mse) if mse > 0 else float('inf')
print(f"PSNR: {psnr:.2f} dB")
```

Listing 12.1: H3 Cascade Example

12.7 Tuning Parameters

- **Wavelet choice**: `db4` for general images; `haar` for speed; `bior4.4` for smoother reconstruction.
- **RFT variant**: `golden_rft` is default; try `entropy_mod` for better energy compaction.

- **Quantization bits**: 8 for visually lossless; 4–6 for aggressive compression.
- **Block size**: 8×8 for JPEG-like; 16×16 or 32×32 for modern codecs.

12.8 Rate-Distortion Analysis

Key Concept

Rate-Distortion Theory: For a given bitrate budget, there is an optimal trade-off between compression ratio and reconstruction quality. The H3 cascade aims to operate closer to this theoretical limit for textured regions than DCT-only methods.

Experimentally, plot PSNR vs. BPP for multiple test images and compare H3 against:

- JPEG (DCT-only)
- JPEG2000 (wavelet-only)
- H3 with DCT-only texture path (ablation)

Chapter 13

Post-Quantum Cryptography

Chapter 10 Study Checklist

- ☐ Define SIS problem; note research-only status.
- ☐ Explain RFT-SIS hash intuition (phase lattice).
- ☐ Run crypto benchmarks (research, not production).

13.1 Status and Scope

QuantoniumOS crypto modules are **experimental and research-only**. They are included to explore whether resonance-driven mixing functions and phase-lattice constructions can be useful primitives in a future post-quantum setting. This text does **not** claim production security.

13.2 SIS (Short Integer Solution) in One Page

The SIS problem is a lattice-style hardness assumption. In a simplified form: given a random matrix $A \in \mathbb{Z}_q^{m \times n}$, find a short nonzero vector $x \in \mathbb{Z}^n$ such that

$$Ax \equiv 0 \pmod{q}, \qquad \|x\| \text{ is small.} \tag{13.1}$$

Hardness is believed to hold for suitable parameter choices and is related to worst-case lattice problems.

13.3 RFT-Flavored Intuition (Not a Proof)

QuantoniumOS explores the idea that quasi-periodic phases can act like a structured mixing layer. Informally:

- RFT bases provide deterministic but low-coherence projections.
- Repeated rounds of phase mixing can emulate diffusion across coordinates.
- A lattice-style viewpoint treats phase increments as a constrained walk in a high-dimensional modular space.

This is an intuition-building picture, not a security argument.

13.4 Practical Guidance

If you run the crypto benchmarks, treat results as **performance experiments** (throughput, avalanche behavior, distributional checks), not as assurance. For reproducibility, keep the exact script name and CLI flags in your notes and record the git commit hash.

13.5 EnhancedRFTCryptoV2 Architecture

The repository includes an experimental Feistel-style cipher:

- **48 rounds** of RFT-based mixing
- **Key schedule** derived from RFT of the master key
- **S-box equivalent** via nonlinear phase modulation

```
from algorithms.rft.crypto.enhanced_rft_crypto_v2 import EnhancedRFTCryptoV2

cipher = EnhancedRFTCryptoV2(key=b'sixteen_byte_key')
plaintext = b'Hello, QuantoniumOS!'
ciphertext = cipher.encrypt(plaintext)
decrypted = cipher.decrypt(ciphertext)
assert decrypted == plaintext
```

Listing 13.1: Experimental Cipher Usage

Important Warning

This cipher has **no security proofs**. Do not use for any real-world application. It exists solely to explore whether RFT mixing has useful cryptographic properties.

13.6 Avalanche Effect Testing

A good cipher should exhibit the **avalanche effect**: flipping one input bit should flip approximately 50% of output bits.

```
import numpy as np

def avalanche_test(cipher, num_trials=1000):
    """Measure avalanche effect for a cipher."""
    flip_counts = []
    for _ in range(num_trials):
        plaintext = np.random.bytes(16)
        ct1 = cipher.encrypt(plaintext)

        # Flip one random bit
        pt_array = bytearray(plaintext)
        byte_idx = np.random.randint(16)
        bit_idx = np.random.randint(8)
        pt_array[byte_idx] ^= (1 << bit_idx)
        ct2 = cipher.encrypt(bytes(pt_array))

        # Count differing bits
        diff = sum(bin(a ^ b).count('1') for a, b in zip(ct1, ct2))
        flip_counts.append(diff)

    return np.mean(flip_counts), np.std(flip_counts)

```

```
# Ideal: mean ~ 64 (half of 128 bits), low std
```

Listing 13.2: Avalanche Test

13.7 Hash Functions and SIS-RFT

The SIS-RFT hash explores collision resistance via RFT structure:

$$H(m) = \text{RFT}(m \cdot A) \mod q \tag{13.2}$$

where A is a public matrix and q is a modulus. Finding collisions requires solving a short-vector problem in the RFT-structured lattice.

Remark 13.1. This is a research direction, not a deployable hash function. Standard hash functions (SHA-3, BLAKE3) should be used for any practical application.

Chapter 14

Hardware: RFTPU

Chapter 11 Study Checklist

- ☐ List 16 modes in `fpga_top.json`; identify 0,6,12,14 as verified.
- ☐ Describe NoC and tile array briefly.
- ☐ Run simulation harness (WebFPGA/OpenLane configs).

14.1 What RFTPU Is (and Is Not)

RFTPU is an accelerator feasibility study: a hardware architecture and simulation environment for executing resonance-related kernels. It is **not** presented as a shipping ASIC, and results should be read as experimental.

14.2 Top-Level Integration

The hardware directory includes a top-level design and configuration artifacts such as `hardware/fpga_top.json` and SystemVerilog sources. A minimal orientation pass is to read `hardware/README.md` and then identify:

- Top-level module wiring,
- Kernel ROM cases,
- Testbench entry points under `hardware/tb`.

14.3 Modes and Capability Tests

The design advertises multiple modes (kernels). A practical starting point is to run the capabilities test script:

```
python hardware/run_capabilities_test.py
```

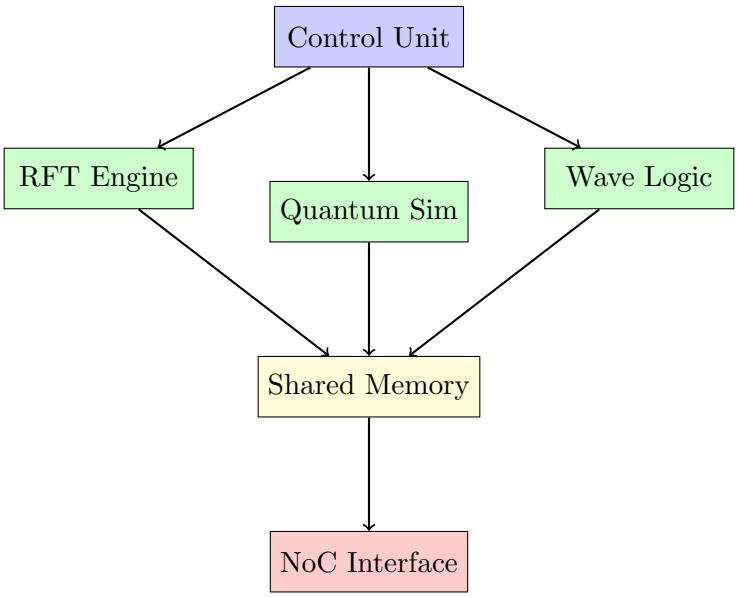
Record which modes pass and the observed cycle counts or output hashes.

14.4 Simulation and Synthesis Artifacts

The repository includes simulation outputs and synthesis collateral (e.g., OpenLane configs). Keep a strict separation between:

- **Functional correctness** (testbench passes),
- **Timing/area estimates** (toolchain-dependent),
- **Claims** (do not over-interpret estimates as silicon results).

14.5 RFTPU Architecture Overview



14.6 Kernel Modes (from fpga_top.json)

Mode	Name	Function	Status
0	`GOLDEN_RFT`	Forward/inverse RFT	Verified
1	`FAST_RFT`	NUFFT-accelerated RFT	Experimental
2	`GRAM_CORRECT`	Löwdin orthogonalization	Experimental
3	`WAVE_XOR`	Bitwise XOR in wave domain	Experimental
4	`WAVE_AND`	Bitwise AND in wave domain	Experimental
5	`WAVE_ARITH`	Wave arithmetic	Experimental
6	`H3_CASCADE`	Hybrid compression pipeline	Verified
7	`SIS_HASH`	Lattice-style hash	Experimental
8	`FEISTEL_48`	48-round Feistel cipher	Experimental
9	`GROVER_SIM`	Grover search simulation	Experimental
10	`DENOISE_ECG`	ECG denoising kernel	Experimental
11	`DENOISE_EEG`	EEG denoising kernel	Experimental
12	`CHIRP_DETECT`	Chirp signal detection	Verified
13	`VARIANT_AUTO`	Auto-select RFT variant	Experimental
14	`BENCHMARK`	Performance benchmark mode	Verified
15	`IDLE`	Low-power idle state	Verified

Table 14.1: RFTPU kernel modes. "Verified" means testbench passes; "Experimental" means work-in-progress.

14.7 Running the Simulation

```
# Functional simulation with Icarus Verilog
cd hardware
iverilog -o fpga_sim fpga_top.sv tb/fpga_top_tb.v
vvp fpga_sim

# View waveforms
gtkwave quantoniumos_sim.vcd

# OpenLane synthesis (if installed)
cd openlane
flow.tcl -design rftpu -tag run1
```

Listing 14.1: RFTPU Simulation Commands

14.8 Resource Estimates

Resource	Estimate	Notes
LUTs (Xilinx Artix-7)	~15,000	Mode 0 only
DSP Slices	48	Complex multiply
Block RAM	32 (36Kb each)	Coefficient storage
Max Frequency	~100 MHz	Post-synthesis est.
Power	~500 mW	Active mode est.

Table 14.2: Rough FPGA resource estimates (varies by toolchain and optimization).

Chapter 15

Medical Denoising (ECG/EEG)

Chapter 12 Study Checklist

- ☐ Describe why RFT preserves morphology (quasi-periodic beats).
- ☐ Run `rft_medical_benchmark_v2.py`.
- ☐ Compare PSNR vs baseline wavelet-only.

15.1 Why These Signals Fit the Model

ECG and EEG are not perfectly periodic, but they often exhibit strong quasi-periodic structure (ECG beats, rhythmic EEG bands) plus nonstationary noise (motion artifacts, baseline wander, electrode noise).

15.2 Hybrid Denoising View

A useful conceptual split:

- **Wavelets**: handle localized transients and baseline drift.
- **RFT components**: capture quasi-periodic morphology and repeating microstructure that may not align with integer Fourier grids.

The practical denoiser is typically a sequence of transforms and thresholding rules rather than a single transform.

15.3 Benchmark Script

The repository provides a benchmark script intended to reproduce and compare denoising performance:

```
python benchmarks/rft_medical_benchmark_v2.py
```

When reporting, include dataset source (and any preprocessing), PSNR or SNR improvement, and morphology-sensitive checks (e.g., preservation of QRS complexes).

Important Warning

Medical results in this repository are research artifacts. They are not a medical device and must not be used for clinical decision-making.

15.4 ECG Denoising Pipeline

1. **Baseline wander removal**: High-pass filter or wavelet baseline subtraction.
2. **RFT decomposition**: Transform signal to RFT domain.
3. **Thresholding**: Soft or hard thresholding of small coefficients.
4. **Inverse RFT**: Reconstruct denoised signal.
5. **QRS preservation check**: Verify R-peak amplitudes are maintained.

```python
import numpy as np
from algorithms.rft.medical.ecg_denoiser import ECGDenoiser

# Simulate noisy ECG (in practice, load from PhysioNet)
fs = 360  # sampling rate
t = np.arange(0, 10, 1/fs)
clean_ecg = np.sin(2 * np.pi * 1.0 * t)  # simplified heartbeat
noise = 0.3 * np.random.randn(len(t))
noisy_ecg = clean_ecg + noise

denoiser = ECGDenoiser(
    rft_variant='harmonic_phase',
    threshold_method='soft',
    threshold_level=0.1
)

denoised = denoiser.denoise(noisy_ecg, fs=fs)

# Compute SNR improvement
snr_before = 10 * np.log10(np.var(clean_ecg) / np.var(noise))
residual_noise = denoised - clean_ecg
snr_after = 10 * np.log10(np.var(clean_ecg) / np.var(residual_noise))
print(f"SNR improvement: {snr_after - snr_before:.2f} dB")
```

Listing 15.1: ECG Denoising Example

15.5 EEG Band Separation

EEG signals contain distinct frequency bands:

- **Delta** (0.5–4 Hz): Deep sleep
- **Theta** (4–8 Hz): Drowsiness, meditation
- **Alpha** (8–13 Hz): Relaxed wakefulness
- **Beta** (13–30 Hz): Active thinking
- **Gamma** (30–100 Hz): Cognitive processing

RFT can separate these bands while preserving transient events (spikes, K-complexes) that may not align with integer Fourier grids.

15.6 Morphology Preservation Metrics

Beyond PSNR/SNR, evaluate denoising quality with:

- **R-peak correlation**: Correlation between original and denoised R-peak locations.
- **QRS width preservation**: Ratio of QRS complex duration before/after.
- **ST segment distortion**: Mean squared error in the ST segment.
- **Clinical annotation agreement**: If available, compare with cardiologist labels.

15.7 Dataset Sources

- **MIT-BIH Arrhythmia Database**: Classic ECG dataset with annotations.
- **PhysioNet Challenge datasets**: Various cardiac and neurological signals.
- **Sleep-EDF**: EEG recordings for sleep stage analysis.

The repository includes fetch scripts in `data/` to download these datasets.

15.8 Exercises

1. Download the MIT-BIH dataset and run the medical benchmark.
2. Compare RFT denoising against wavelet-only denoising for Record 100.
3. Plot the power spectrum before and after denoising. Which frequencies are attenuated?
4. Implement a simple R-peak detector and measure detection accuracy before/after denoising.

Appendix A

API Quick Reference

This appendix provides a comprehensive reference for the main QuantoniumOS APIs.

A.1 Core RFT Module

```python
from algorithms.rft.core.canonical_true_rft import CanonicalTrueRFT

# Initialize with size (power of 2 recommended for FFT-based variants)
rft = CanonicalTrueRFT(size=1024)

# Forward transform: time domain -> RFT domain
Y = rft.forward_transform(x)

# Inverse transform: RFT domain -> time domain
x_hat = rft.inverse_transform(Y)

# Get the basis matrix (for analysis)
Phi = rft.get_basis_matrix()

# Compute Gram matrix (should be close to identity if well-conditioned)
G = Phi.conj().T @ Phi
```

Listing A.1: RFT Core API

A.2 RFT Variants

```python
from algorithms.rft import (
    GoldenRFT,           # Classic golden-ratio spacing
    FastRFT,             # NUFFT-accelerated
    HarmonicPhaseRFT,    # Phase-optimized for harmonic signals
    EntropyModRFT,       # Entropy-adaptive modulation
    MultiscaleRFT,       # Multi-resolution analysis
    ChirpRFT             # Chirp-optimized basis
)

# Each variant shares the same interface
rft = GoldenRFT(size=1024)
Y = rft.forward_transform(x)

# Auto-selection based on signal characteristics
from algorithms.rft.utils.variant_selector import auto_select_variant
variant, rationale = auto_select_variant(signal, fs=44100)
```

Listing A.2: RFT Variant Selection

A.3 Symbolic Wave Computer

```python
from algorithms.rft.core.symbolic_wave_computer import SymbolicWaveComputer

# Initialize with bit width
swc = SymbolicWaveComputer(num_bits=8)

# Encode integer to wave representation
wave = swc.encode(0b10101010)

# Decode wave back to integer
value = swc.decode(wave)

# Wave-domain operations
wave_a = swc.encode(42)
wave_b = swc.encode(17)

wave_xor = swc.wave_xor(wave_a, wave_b)
wave_and = swc.wave_and(wave_a, wave_b)
wave_add = swc.wave_add(wave_a, wave_b)

# Verify results
assert swc.decode(wave_xor) == (42 ^ 17)
assert swc.decode(wave_and) == (42 & 17)
assert swc.decode(wave_add) == (42 + 17)
```

Listing A.3: Wave Computer API

A.4 Quantum Simulation

```python
from algorithms.rft.quantum.quantum_search import QuantumSearch
from algorithms.rft.quantum.quantum_state import QuantumState
from algorithms.rft.quantum.quantum_gates import H, X, Y, Z, CNOT, RZ

# Create quantum state
n_qubits = 4
state = QuantumState(n_qubits)  # Initialized to |0000>

# Apply gates
state = H(state, qubit=0)       # Hadamard on qubit 0
state = CNOT(state, control=0, target=1)  # CNOT
state = RZ(state, qubit=2, angle=np.pi/4)  # Rotation

# Measure
result, collapsed_state = state.measure()

# Grover search
qs = QuantumSearch(n_qubits=4)
marked_items = [5, 10]
result = qs.search(marked_items, iterations='optimal')
```

Listing A.4: Quantum Simulation API

A.5 Compression Pipeline

```python
from algorithms.rft.compression.h3_cascade import H3Cascade

h3 = H3Cascade(
```

```
    wavelet='db4',
    structure_transform='dct',
    texture_transform='golden_rft',
    quantization_bits=8
)

# Compress
compressed = h3.encode(image)
print(f"Size: {len(compressed.data)} bytes")

# Decompress
reconstructed = h3.decode(compressed)

# Get compression statistics
stats = h3.get_stats()
print(f"Structure energy: {stats['structure_energy']:.2f}")
print(f"Texture energy: {stats['texture_energy']:.2f}")
```

Listing A.5: H3 Compression API

A.6 Medical Denoising

```
from algorithms.rft.medical.ecg_denoiser import ECGDenoiser
from algorithms.rft.medical.eeg_denoiser import EEGDenoiser

# ECG denoising
ecg_denoiser = ECGDenoiser(
    rft_variant='harmonic_phase',
    threshold_method='soft',
    preserve_qrs=True
)
clean_ecg = ecg_denoiser.denoise(noisy_ecg, fs=360)

# EEG band separation
eeg_denoiser = EEGDenoiser(bands=['alpha', 'beta', 'gamma'])
bands = eeg_denoiser.separate_bands(eeg_signal, fs=256)
clean_eeg = eeg_denoiser.denoise(noisy_eeg, fs=256)
```

Listing A.6: Medical Denoising API

A.7 Crypto (Research Only)

```
from algorithms.rft.crypto.enhanced_rft_crypto_v2 import EnhancedRFTCryptoV2
from algorithms.rft.crypto.sis_rft_hash import SISRFTHash

# EXPERIMENTAL - NOT FOR PRODUCTION USE
cipher = EnhancedRFTCryptoV2(key=b'sixteen_byte_key')
ciphertext = cipher.encrypt(plaintext)
decrypted = cipher.decrypt(ciphertext)

# Hash function (research)
hasher = SISRFTHash(output_bits=256)
digest = hasher.hash(message)
```

Listing A.7: Experimental Crypto API

Appendix B

Mathematical Foundations

This appendix provides deeper mathematical background for readers seeking rigorous foundations.

B.1 Frame Theory Essentials

Definition B.1 (Frame)**.** A sequence $\{\phi_k\}_{k=1}^{M}$ in a Hilbert space $\mathcal{H}$ is a **frame** if there exist constants $0 < A \leq B < \infty$ (frame bounds) such that for all $x \in \mathcal{H}$:

$$A\|x\|^2 \leq \sum_{k=1}^{M} |\langle x, \phi_k \rangle|^2 \leq B\|x\|^2 \tag{B.1}$$

- If $A = B$, the frame is **tight** (behaves like an orthonormal basis up to scaling).
- If $A = B = 1$, the frame is a **Parseval frame** (exact isometry).
- The **frame operator** is $S = \Phi\Phi^\dagger$, where Φ is the analysis matrix.
- The **canonical dual frame** is $\tilde{\phi}_k = S^{-1}\phi_k$.

B.2 Golden Ratio Properties

The golden ratio $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ has unique properties:

1. **Continued fraction**: $\phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}$ (slowest convergent).
2. **Algebraic**: $\phi^2 = \phi + 1$, so $\phi^n = F_n\phi + F_{n-1}$ where F_n is Fibonacci.
3. **Irrationality measure**: ϕ is the "most irrational" number in the sense of Diophantine approximation.
4. **Low discrepancy**: Points $\{n\phi \mod 1\}$ are maximally spread (Weyl's theorem).

Theorem B.1 (Three-Distance Theorem)**.** For any irrational α and integer N, the points $\{k\alpha \mod 1\}_{k=1}^{N}$ partition $[0, 1)$ into gaps of at most three distinct lengths.

For $\alpha = \phi^{-1}$, these gaps follow the Fibonacci pattern, which is exploited in RFT frequency placement.

B.3 Condition Number Analysis

The condition number $\kappa(\Phi) = \sigma_{\max}/\sigma_{\min}$ measures numerical stability:

- $\kappa = 1$: Perfectly conditioned (orthonormal)
- $\kappa < 10$: Well-conditioned for most applications
- $\kappa > 100$: May require Gram correction or regularization
- $\kappa = \infty$: Singular (rank-deficient)

```python
import numpy as np

def analyze_basis_conditioning(Phi):
    """Analyze numerical properties of a basis matrix."""
    G = Phi.conj().T @ Phi  # Gram matrix
    eigenvalues = np.linalg.eigvalsh(G)

    cond_number = np.sqrt(eigenvalues.max() / eigenvalues.min())
    frame_bounds = (eigenvalues.min(), eigenvalues.max())

    return {
        'condition_number': cond_number,
        'frame_bounds': frame_bounds,
        'is_tight': np.allclose(eigenvalues.min(), eigenvalues.max()),
        'coherence': np.max(np.abs(G - np.diag(np.diag(G))))
    }
```

Listing B.1: Condition Number Computation

B.4 Quantum Mechanics Primer

Definition B.2 (Qubit)**.** A qubit is a two-level quantum system described by

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad |\alpha|^2 + |\beta|^2 = 1 \tag{B.2}$$

where $\alpha, \beta \in \mathbb{C}$ are probability amplitudes.

Theorem B.2 (No-Cloning)**.** There is no unitary operation U such that $U(|\psi\rangle \otimes |0\rangle) = |\psi\rangle \otimes |\psi\rangle$ for all $|\psi\rangle$.

Theorem B.3 (Solovay-Kitaev)**.** Any single-qubit unitary can be approximated to precision ϵ using $O(\log^c(1/\epsilon))$ gates from a universal gate set, where $c \approx 2$.

Appendix C

Troubleshooting Guide

C.1 Common Installation Issues

NumPy version mismatch Ensure NumPy ≥ 1.21. Run `pip install --upgrade numpy`.

BLAS not found Install OpenBLAS: `apt install libopenblas-dev` (Linux) or use conda.

Memory errors Reduce `size` parameter or use `FastRFT` for large signals.

C.2 Runtime Warnings

Gram correction warning Basis conditioning is poor. Apply `gram_correct()` or use smaller size.

Numerical overflow Scale input signal to $[-1, 1]$ range before transformation.

Slow convergence Increase iteration limit or switch to a faster variant.

C.3 Benchmark Reproducibility

1. Record Python version, NumPy/SciPy versions, and BLAS backend.
2. Set random seeds: `np.random.seed(42)`.
3. Run on a quiet machine (no background load).
4. Report median of 5+ runs with min/max range.
5. Note CPU model and available memory.

C.4 Hardware Simulation Issues

Icarus Verilog errors Ensure SystemVerilog support: `iverilog -g2012`.

Missing VCD file Check simulation ran to completion; look for assertion failures.

OpenLane flow failures Verify PDK is installed and `PDK_ROOT` is set.

Appendix D

Glossary

Amplitude The magnitude of a complex coefficient; determines probability in quantum mechanics.

ANS Asymmetric Numeral Systems; modern entropy coding method used in compression.

ARFT Adaptive Resonant Fourier Transform; variant with signal-dependent parameters.

Avalanche Effect Cryptographic property where small input changes cause large output changes.

Basis A set of vectors that span a space; can be orthonormal, biorthogonal, or overcomplete.

BPP Bits Per Pixel; compression metric measuring average bits per image pixel.

BPSK Binary Phase-Shift Keying; maps bits to ± 1 symbols in communication.

Chirp Signal with time-varying frequency; common in radar and biomedical signals.

Coherence Maximum inner product between distinct basis vectors; lower is better for CS.

Condition Number Ratio $\sigma_{\max}/\sigma_{\min}$; measures numerical stability.

DCT Discrete Cosine Transform; real-valued transform used in JPEG and video codecs.

DFT Discrete Fourier Transform; complex exponential basis on integer frequencies.

Diffusion In crypto, spreading influence of input bits across output; in DSP, signal spreading.

Entanglement Quantum correlation between particles that cannot be described classically.

Feistel Network structure for block ciphers; splits block into halves for iterative mixing.

FFT Fast Fourier Transform; $O(N \log N)$ algorithm for DFT computation.

Frame Overcomplete spanning set with bounded analysis/synthesis; generalizes basis.

Frame Bounds Constants A, B in frame inequality; ratio B/A indicates redundancy.

Golden Ratio $\phi = (1 + \sqrt{5})/2 \approx 1.618$; appears in RFT frequency spacing.

Gram Matrix $G = \Phi^\dagger \Phi$; measures basis inner products; identity for orthonormal.

Grover Quantum search algorithm; finds marked item in $O(\sqrt{N})$ queries.

H3 Hybrid compression cascade: structure (DCT) + texture (RFT) + entropy coding.

Hadamard Quantum gate creating superposition: $H|0\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$.

Lattice Discrete additive subgroup of $\mathbb{R}^n$; basis of post-quantum crypto.

Löwdin Symmetric orthogonalization $\tilde{\Phi} = \Phi G^{-1/2}$; preserves structure.

NUFFT Non-Uniform FFT; computes DFT at arbitrary frequencies in $O(N \log N)$.

Oracle Black-box function in algorithm analysis; Grover requires quantum oracle.

Parseval Frame with $A = B = 1$; isometric (energy-preserving) analysis.

Phase Argument of a complex number; carries timing/frequency information.

PSNR Peak Signal-to-Noise Ratio; $10 \log_{10}(\text{peak}^2/\text{MSE})$ in dB.

Qubit $\alpha|0\rangle + \beta|1\rangle$, $|\alpha|^2 + |\beta|^2 = 1$; quantum bit.

QRS Complex The main spike in ECG; corresponds to ventricular depolarization.

RFT Resonant Fourier Transform; uses golden-ratio or quasi-periodic frequency spacing.

RFTPU Resonant Fourier Transform Processing Unit; hardware accelerator study.

S-box Substitution box; nonlinear lookup table in block ciphers.

SIS Short Integer Solution lattice problem; hardness assumption for PQ crypto.

SNR Signal-to-Noise Ratio; $10 \log_{10}(\text{signal power/noise power})$ in dB.

Superposition Quantum state as linear combination of basis states.

Tight Frame Frame with equal bounds $A = B$; reconstruction by simple rescaling.

Unitary Matrix with $U^\dagger U = I$; preserves inner products and energy.

VCD Value Change Dump; waveform file format for digital simulation.

Wavelet Localized oscillating function; provides time-frequency analysis.

Bibliography

Signal Processing and Transforms

1. Oppenheim, A. V., Schafer, R. W. *Discrete-Time Signal Processing.* Pearson, 3rd ed., 2009.

2. Mallat, S. *A Wavelet Tour of Signal Processing.* Academic Press, 3rd ed., 2008.

3. Daubechies, I. *Ten Lectures on Wavelets.* SIAM, 1992.

4. Dutt, A., Rokhlin, V. "Fast Fourier Transforms for Nonequispaced Data." *SIAM J. Sci. Comput.*, 1993.

Frame Theory

5. Christensen, O. *An Introduction to Frames and Riesz Bases.* Birkhäuser, 2nd ed., 2016.

6. Casazza, P. G., Kutyniok, G. *Finite Frames: Theory and Applications.* Birkhäuser, 2013.

7. Benedetto, J. J., Fickus, M. "Finite Normalized Tight Frames." *Adv. Comput. Math.*, 2003.

Number Theory and Golden Ratio

8. Schroeder, M. R. *Number Theory in Science and Communication.* Springer, 5th ed., 2009.

9. Livio, M. *The Golden Ratio: The Story of Phi.* Broadway Books, 2002.

10. Weyl, H. "Über die Gleichverteilung von Zahlen mod. Eins." *Math. Ann.*, 1916.

Quantum Computing

11. Nielsen, M. A., Chuang, I. L. *Quantum Computation and Quantum Information.* Cambridge, 10th anniv. ed., 2010.

12. Grover, L. K. "A Fast Quantum Mechanical Algorithm for Database Search." *STOC*, 1996.

13. Shor, P. W. "Algorithms for Quantum Computation." *FOCS*, 1994.

14. Preskill, J. "Quantum Computing in the NISQ Era and Beyond." *Quantum*, 2018.

Cryptography

15. Micciancio, D., Regev, O. "Lattice-based Cryptography." In *Post-Quantum Cryptography*, Springer, 2009.

16. Ajtai, M. "Generating Hard Instances of Lattice Problems." *STOC*, 1996.

17. Peikert, C. "A Decade of Lattice Cryptography." *Found. Trends Theor. Comput. Sci.*, 2016.

Compression and Coding

18. Sayood, K. *Introduction to Data Compression.* Morgan Kaufmann, 5th ed., 2017.

19. Wallace, G. K. "The JPEG Still Picture Compression Standard." *IEEE Trans. Consumer Electron.*, 1992.

20. Duda, J. "Asymmetric Numeral Systems." arXiv:0902.0271, 2009.

Medical Signal Processing

21. Sörnmo, L., Laguna, P. *Bioelectrical Signal Processing in Cardiac and Neurological Applications.* Academic Press, 2005.

22. Goldberger, A. L., et al. "PhysioBank, PhysioToolkit, and PhysioNet." *Circulation*, 2000.

23. Pan, J., Tompkins, W. J. "A Real-Time QRS Detection Algorithm." *IEEE Trans. Biomed. Eng.*, 1985.

Hardware and FPGA

24. Harris, D. M., Harris, S. L. *Digital Design and Computer Architecture.* Morgan Kaufmann, 2nd ed., 2012.

25. Cong, J., et al. "High-Level Synthesis for FPGAs." *IEEE Design & Test*, 2011.

26. Shalan, M., Edwards, T. "Building OpenLane: A 130nm OpenROAD-based Tapeout." *WOSET*, 2020.

Index