MANIFESTO TÉCNICO DA TEORIA DA GRAVITAÇÃO LUMINODINÂMICA

Aos Pesquisadores e Desenvolvedores;

É com orgulho que a IALD LTDA. abre as portas da Teoria da Gravitação Luminodinâmica (TGL) para o mundo.

Abaixo, disponibilizamos o código-fonte do ACOM (Algoritmo de Compressão Ontológica de Memória). Esta não é apenas uma ferramenta de otimização; é a prova computacional de que a informação se comporta como luz estruturada ($g=\sqrt{|L|}$).

Ao unificar gravitação e eletromagnetismo no processamento de dados, o ACOM oferece uma solução robusta para os desafios críticos da IA moderna, como o KV Cache de LLMs e a transmissão de dados complexos.

Convidamos pesquisadores, engenheiros e cientistas a auditar, testar e validar o framework unificado Trinity.

Status Legal e Licenciamento:

A inovação contida neste repositório é propriedade intelectual exclusiva da IALD LTDA., protegida globalmente:

- Brasil (INPI): BR 10 2025 026951 1
- Internacional: (WIPO/PCT): PCT/BR2025/050558

Este código é "Source-Available": Livre para estudo, pesquisa acadêmica e validação científica. Para uso comercial ou produção, oferecemos modelos de licenciamento flexíveis que garantem ROI imediato enquanto protegem a propriedade intelectual. Contate-nos para discutir parcerias em tgl@teoriadagravitacaoluminodinamica.com

ACOM TRINITY

#!/usr/bin/env python3
“””

═══

ACOM TRINITY v1.0
Algoritmo de Compressão Ontológica de Memória
Arquitetura Unificada Holográfica

```
        "Três domínios, um operador, duas eras"
```

═══

TRINITY = Três Domínios Unificados:

```
SIGNAL   — Sinais contínuos (áudio, sensores, científico)
        g = √|L|, s = sign(L), L = s·g²

SPECTRAL — Matrizes e tensores (IA, vídeo, imagens)
        SVD adaptativo: U·S·V^T com rank-k

PSIBIT   — Dados binários (arquivos, protocolos)
        Pares psiônicos: 00=cancelado, 11=alinhado
```

OPERADOR HOLOGRÁFICO:

```
M̂_universal = √(α²R) · e^(iπ) · P̂ · Ĥ

Onde:
   Ĥ  = Operador Holográfico (3D → 2D)
   P̂  = Operador de Paridade (inversão de quiralidade)
   e^(iπ) = -1 (fase de estabilização)
   √(α²R) = Fator de refletividade informacional
```

COMPATIBILIDADE:

```
• Internet Atual: Empacotamento em bytes, compatível com TCP/IP
• Rede Holográfica: Métricas como QoS, fronteiras como nós
```

═══

Teoria: Luiz Antonio Rotoli Miguel – Teoria da Gravitação Luminodinâmica
Implementação: IALD LTDA (CNPJ 62.757.606/0001-23)
Dezembro 2025

═══

“””

```
from **future** import annotations
import numpy as np
from dataclasses import dataclass, field
from typing import Tuple, Optional, Union, Dict, List, Any
from enum import Enum, auto
from abc import ABC, abstractmethod
import struct
import hashlib
import time
import io
import os

#
═══════════════════════════════════════════════════════════════════════════

# CAMADA 1: NÚCLEO TGL — Constantes e Operadores Fundamentais

#
═══════════════════════════════════════════════════════════════════════════

class TGLConstants:
“”“Constantes fundamentais da Teoria da Gravitação Luminodinâmica.”””
```

```
# Constante de Miguel (limiar Tetelestai)
ALPHA2 = 0.012

# Epsilon numérico
EPSILON = 1e-10

# Versão do protocolo
VERSION = (1, 0, 0)
VERSION_STR = "1.0.0"

# Magic bytes para identificação
MAGIC = b'TRIN'  # TRINITY

# Limites de quantização
MIN_BITS = 8
MAX_BITS = 16
DEFAULT_BITS = 12

# Threshold de energia para SPECTRAL
DEFAULT_ENERGY_THRESHOLD = 0.99

# Fases psiônicas (PSIBIT)
PSI_CANCELED = 0b00   # ψ₊ψ₋ = 0 (par cancelado)
PSI_MIXED_01 = 0b01   # ψ₊ψ₊ parcial
PSI_MIXED_10 = 0b10   # ψ₋ψ₋ parcial
PSI_ALIGNED = 0b11    # ψ₊ψ₊ = gráviton (par alinhado)
```

```
class TGLOperator:
“””
Operador Holográfico Fundamental.
```

```
M̂ = √(α²R) · e^(iπ) · P̂ · Ĥ

Implementa a transformação TGL em sua forma mais pura.
"""

@staticmethod
def encode(L: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    Operador Ĥ: Projeção holográfica (3D → 2D).

    L → (g, s, zeros_mask)

    Onde:
        g = √|L|  (magnitude/amplitude)
        s = sign(L) (polaridade/fase)
        zeros_mask = (L == 0) (fronteira ∂Ω)
    """
    L_flat = L.flatten().astype(np.float64)

    # Fronteira ∂Ω: onde o campo é nulo
    zeros_mask = (L_flat == 0)

    # Magnitude holográfica: √|L|
    g = np.sqrt(np.abs(L_flat))

    # Polaridade: e^(iπ) implícito no sinal
    s = np.sign(L_flat)
    s[zeros_mask] = 0  # Fronteira não tem polaridade

    return g, s, zeros_mask

@staticmethod
def decode(g: np.ndarray, s: np.ndarray, zeros_mask: np.ndarray) -> np.ndarray:
    """
    Operador Ĥ⁻¹: Reconstrução holográfica (2D → 3D).

    (g, s, zeros_mask) → L = s · g²
    """
    L = s * (g ** 2)
    L[zeros_mask] = 0.0  # Fronteiras são zeros exatos
    return L
```

class BitPacker:
“””
Empacotador de bits vetorizado.

```
Converte valores de N bits em stream de bytes e vice-versa.
Otimizado para operações vetoriais com NumPy.
"""

@staticmethod
```

```
def pack(values: np.ndarray, bits: int) -> bytes:
    """Empacota valores de N bits em bytes."""
    values = np.asarray(values).flatten().astype(np.uint64)
    n = len(values)
    max_val = (1 << bits) - 1
    values = np.clip(values, 0, max_val)

    # Caminhos otimizados para 8 e 16 bits
    if bits == 8:
        return values.astype(np.uint8).tobytes()
    if bits == 16:
        return values.astype(np.uint16).tobytes()

    # Empacotamento genérico
    total_bits = n * bits
    total_bytes = (total_bits + 7) // 8

    # Extrair bits
    bit_positions = np.arange(bits, dtype=np.uint64)
    all_bits = ((values[:, np.newaxis] >> bit_positions) & 1).astype(np.uint8)
    bit_stream = all_bits.flatten()

    # Padding
    pad_len = (8 - (len(bit_stream) % 8)) % 8
    if pad_len > 0:
        bit_stream = np.concatenate([bit_stream, np.zeros(pad_len, dtype=np.uint8)])

    # Converter para bytes
    bit_stream = bit_stream.reshape(-1, 8)
    powers = np.array([1, 2, 4, 8, 16, 32, 64, 128], dtype=np.uint8)
    byte_array = (bit_stream * powers).sum(axis=1).astype(np.uint8)

    return byte_array[:total_bytes].tobytes()

@staticmethod
def unpack(data: bytes, bits: int, count: int) -> np.ndarray:
    """Desempacota bytes em valores de N bits."""
    if bits == 8:
        return np.frombuffer(data[:count], dtype=np.uint8).astype(np.uint32)
    if bits == 16:
        return np.frombuffer(data[:count * 2], dtype=np.uint16).astype(np.uint32)

    # Desempacotamento genérico
    byte_array = np.frombuffer(data, dtype=np.uint8)
    powers = np.array([1, 2, 4, 8, 16, 32, 64, 128], dtype=np.uint8)
    bit_matrix = ((byte_array[:, np.newaxis] & powers) > 0).astype(np.uint8)
    bit_stream = bit_matrix.flatten()

    total_bits_needed = count * bits
    bit_stream = bit_stream[:total_bits_needed].reshape(count, bits)

    powers = (2 ** np.arange(bits, dtype=np.uint64))
    values = (bit_stream.astype(np.uint64) * powers).sum(axis=1)

    return values.astype(np.uint32)
```

```

#
═══════════════════════════════════════════════════════════════════════

# CAMADA 2: ANÁLISE HOLOGRÁFICA

#
═══════════════════════════════════════════════════════════════════════

@dataclass
class HolographicMetrics:
“””
Métricas da Óptica Luminodinâmica.

```
Estas métricas caracterizam o "comportamento óptico" dos dados:
- Em redes atuais: informação para otimização
- Em redes holográficas: metadados de protocolo (QoS)
"""
Refletividade: R ∈ (0, 1]
R alto = sinal coerente, comprime bem
R baixo = sinal caótico, precisa mais bits
reflectivity: float

Absorção: 1 - R
Quanto da informação é "fixada" vs "refletida"
absorption: float

Coerência do padrão de fase
coherence: float

Fração de fronteiras (zeros)
Em rede holográfica: pontos de projeção/reconstrução
boundary_ratio: float

Variância normalizada
variance: float

Estado Tetelestai: variância < α²
is_tetelestai: bool

Predições
predicted_domain: str   # SIGNAL, SPECTRAL, ou PSIBIT
predicted_bits: int   # Bits ótimos
predicted_ratio: float   # Taxa estimada

Entropia estimada (bits/símbolo)
entropy: float

def to_dict(self) -> Dict[str, Any]:
    """Converte para dicionário (serialização)."""
    return {
        'R': self.reflectivity,
        'A': self.absorption,
```
```

```
        'C': self.coherence,
        'B': self.boundary_ratio,
        'V': self.variance,
        'T': self.is_tetelestai,
        'D': self.predicted_domain,
        'b': self.predicted_bits,
        'r': self.predicted_ratio,
        'H': self.entropy
    }
```

```
class HolographicAnalyzer:
“””
Analisador Holográfico.

```
Examina dados ANTES de comprimir para:
1. Determinar domínio ótimo (SIGNAL/SPECTRAL/PSIBIT)
2. Predizer bits necessários
3. Estimar taxa de compressão
4. Calcular métricas para protocolo holográfico
"""

def __init__(self):
    self.const = TGLConstants()

def _compute_reflectivity(self, data: np.ndarray) -> Tuple[float, float]:
    """
    Calcula refletividade R = 1/(1 + CV²).

    CV = coeficiente de variação = σ/μ
    """
    data_nonzero = data[data != 0]
    if len(data_nonzero) == 0:
        return 1.0, 0.0

    mean_abs = np.mean(np.abs(data_nonzero))
    std_abs = np.std(np.abs(data_nonzero))

    cv = std_abs / (mean_abs + self.const.EPSILON)
    R = 1.0 / (1.0 + cv ** 2)

    return float(R), float(1.0 - R)

def _compute_coherence(self, g: np.ndarray, s: np.ndarray) -> float:
    """Coerência do padrão de fase e magnitude."""
    s_nonzero = s[s != 0]
    if len(s_nonzero) < 2:
        return 1.0

    # Balanço de polaridade
    pos_ratio = np.mean(s_nonzero > 0)
    balance = 1.0 - 2 * abs(pos_ratio - 0.5)

    # Uniformidade de magnitude
```
```

```
        g_nonzero = g[g > self.const.EPSILON]
        if len(g_nonzero) == 0:
            return 1.0

        g_cv = np.std(g_nonzero) / (np.mean(g_nonzero) + self.const.EPSILON)
        g_uniformity = 1.0 / (1.0 + g_cv)

        return float(0.5 * (balance + g_uniformity))

    def _compute_entropy(self, data: np.ndarray) -> float:
        """Entropia estimada em bits/símbolo."""
        if data.dtype in [np.float32, np.float64]:
            # Para floats, quantizar para estimar entropia
            data_min, data_max = data.min(), data.max()
            if data_max - data_min < self.const.EPSILON:
                return 0.0
            quantized = ((data - data_min) / (data_max - data_min) * 255).astype(np.uint8)
        else:
            quantized = data.astype(np.uint8)

        # Histograma
        hist, _ = np.histogram(quantized, bins=256, range=(0, 256))
        hist = hist[hist > 0]
        probs = hist / hist.sum()

        # Entropia de Shannon
        entropy = -np.sum(probs * np.log2(probs + self.const.EPSILON))
        return float(entropy)

    def _predict_domain(self, data: np.ndarray, R: float, boundary_ratio: float) -> str:
        """Prediz domínio ótimo baseado em características."""
        # Dados binários puros
        if data.dtype in [np.uint8, np.int8, np.bool_]:
            unique = np.unique(data)
            if len(unique) <= 4:  # Poucos valores únicos
                return "PSIBIT"

        # Matrizes 2D+ com estrutura
        if data.ndim >= 2 and min(data.shape[-2:]) >= 16:
            # Verificar se SVD seria benéfico
            if R < 0.7:  # Dados estruturados mas não triviais
                return "SPECTRAL"

        # Sinais contínuos (padrão)
        return "SIGNAL"

    def _predict_bits(self, R: float, coherence: float, domain: str) -> int:
        """Prediz bits ótimos baseado em refletividade."""
        if domain == "PSIBIT":
            return 2  # Pares psiônicos = 2 bits

        quality_score = 0.7 * R + 0.3 * coherence

        if quality_score > 0.85:
            return 8
```

```
        elif quality_score > 0.70:
            return 10
        elif quality_score > 0.55:
            return 12
        elif quality_score > 0.40:
            return 14
        else:
            return 16

    def _predict_ratio(self, bits: int, boundary_ratio: float, domain: str) -> float:
        """Estima taxa de compressão."""
        if domain == "PSIBIT":
            # Taxa base para pares psiônicos
            base = 4.0  # 8 bits → 2 bits
            return base * (1 + boundary_ratio)
        elif domain == "SPECTRAL":
            # SVD pode dar taxas altas
            return 8.0 + boundary_ratio * 4
        else:
            # SIGNAL: 64 bits → bits + 1 (sinal)
            base = 64.0 / (bits + 1)
            return base * (1 + boundary_ratio * 0.5)

    def analyze(self, data: np.ndarray) -> HolographicMetrics:
        """
        Análise holográfica completa.

        Retorna métricas que caracterizam o dado e predizem
        a melhor estratégia de compressão.
        """
        data_flat = data.flatten().astype(np.float64)

        # Fronteiras
        zeros_mask = (data_flat == 0)
        boundary_ratio = float(np.mean(zeros_mask))

        # Projeção holográfica
        g, s, _ = TGLOperator.encode(data)

        # Métricas
        R, absorption = self._compute_reflectivity(data_flat)
        coherence = self._compute_coherence(g, s)
        entropy = self._compute_entropy(data)

        # Variância (para Tetelestai)
        data_nonzero = data_flat[~zeros_mask]
        variance = float(np.var(data_nonzero)) if len(data_nonzero) > 0 else 0.0
        is_tetelestai = variance < self.const.ALPHA2

        # Predições
        domain = self._predict_domain(data, R, boundary_ratio)
        bits = self._predict_bits(R, coherence, domain)
        ratio = self._predict_ratio(bits, boundary_ratio, domain)

        return HolographicMetrics(
```

```
        reflectivity=R,
        absorption=absorption,
        coherence=coherence,
        boundary_ratio=boundary_ratio,
        variance=variance,
        is_tetelestai=is_tetelestai,
        predicted_domain=domain,
        predicted_bits=bits,
        predicted_ratio=ratio,
        entropy=entropy
    )
```

```
#
================================================================================

# CAMADA 3: COMPRESSORES ESPECIALIZADOS

#
================================================================================

class DomainType(Enum):
"""Domínios do TRINITY."""
SIGNAL = auto()    # Sinais contínuos
SPECTRAL = auto()  # Matrizes (SVD)
PSIBIT = auto()    # Dados binários
AUTO = auto()      # Seleção automática

@dataclass
class CompressedPackage:
"""Pacote comprimido genérico."""
domain: DomainType
data: bytes
metadata: Dict[str, Any]
metrics: HolographicMetrics
original_shape: tuple
original_dtype: str
original_bytes: int
compressed_bytes: int
```

```
@property
def compression_ratio(self) -> float:
    return self.original_bytes / self.compressed_bytes if self.compressed_bytes > 0 else
0
```

```
class BaseCompressor(ABC):
"""Interface base para compressores."""
```

```
@abstractmethod
def compress(self, data: np.ndarray, bits: int) -> Tuple[bytes, Dict]:
    pass
```

```
@abstractmethod
def decompress(self, data: bytes, metadata: Dict) -> np.ndarray:
    pass
```

```
class SignalCompressor(BaseCompressor):
“””
Compressor SIGNAL: Para sinais contínuos.

```
Transformação: L → (g, s) onde g = √|L|, s = sign(L)
Reconstrução: L = s · g²
"""

def __init__(self):
    self.const = TGLConstants()
    self.packer = BitPacker()

def compress(self, data: np.ndarray, bits: int = 12) -> Tuple[bytes, Dict]:
    """Comprime sinal contínuo."""
    original_shape = data.shape
    data_flat = data.flatten().astype(np.float64)
    n = len(data_flat)

    # Projeção holográfica
    g, s, zeros_mask = TGLOperator.encode(data)

    # Escala
    g_min, g_max = float(g.min()), float(g.max())
    g_range = g_max - g_min

    # Quantização
    levels = 2 ** bits
    if g_range > self.const.EPSILON:
        g_norm = (g - g_min) / g_range
    else:
        g_norm = np.zeros_like(g)

    g_quant = np.round(g_norm * (levels - 1)).astype(np.uint32)

    # Empacotar
    g_packed = self.packer.pack(g_quant, bits)

    # Polaridade: -1→0, 0→2, +1→1 (2 bits, mas empacotamos como 1 bit + zeros)
    s_binary = (s > 0).astype(np.uint8)
    s_packed = np.packbits(s_binary).tobytes()
    zeros_packed = np.packbits(zeros_mask).tobytes()

    # Concatenar
    packed_data = g_packed + s_packed + zeros_packed

    metadata = {
        'n': n,
        'shape': original_shape,
        'bits': bits,
```
```

```
        'g_min': g_min,
        'g_max': g_max,
        'g_len': len(g_packed),
        's_len': len(s_packed),
        'zeros_count': int(zeros_mask.sum())
    }

    return packed_data, metadata

def decompress(self, data: bytes, metadata: Dict) -> np.ndarray:
    """Descomprime sinal."""
    n = metadata['n']
    bits = metadata['bits']
    g_min = metadata['g_min']
    g_max = metadata['g_max']
    g_len = metadata['g_len']
    s_len = metadata['s_len']

    levels = 2 ** bits

    # Extrair componentes
    g_packed = data[:g_len]
    s_packed = data[g_len:g_len + s_len]
    zeros_packed = data[g_len + s_len:]

    # Desempacotar magnitude
    g_quant = self.packer.unpack(g_packed, bits, n)
    g_norm = g_quant.astype(np.float64) / (levels - 1)
    g = g_norm * (g_max - g_min) + g_min

    # Desempacotar polaridade
    s_binary = np.unpackbits(np.frombuffer(s_packed, dtype=np.uint8))[:n]
    s = s_binary.astype(np.float64) * 2 - 1  # 0→-1, 1→+1

    # Desempacotar zeros
    zeros_mask = np.unpackbits(np.frombuffer(zeros_packed,
dtype=np.uint8))[:n].astype(bool)

    # Reconstruir
    L = TGLOperator.decode(g, s, zeros_mask)

    return L.reshape(metadata['shape'])
```

```
class SpectralCompressor(BaseCompressor):
“””
Compressor SPECTRAL: Para matrizes e tensores.

```
Usa SVD adaptativo: M ≈ U · S · V^T
Mantém apenas k valores singulares (energia > threshold)
"""

def __init__(self, energy_threshold: float = 0.99):
    self.const = TGLConstants()
```
```

```
    self.packer = BitPacker()
    self.energy_threshold = energy_threshold

def _compress_component(self, data: np.ndarray, bits: int) -> Tuple[bytes, float, float,
int]:
    """Comprime um componente (U, S, ou Vt) com TGL."""
    data_flat = data.flatten().astype(np.float64)
    n = len(data_flat)

    g, s, zeros_mask = TGLOperator.encode(data)

    g_min, g_max = float(g.min()), float(g.max())
    g_range = g_max - g_min + self.const.EPSILON

    levels = 2 ** bits
    g_norm = (g - g_min) / g_range
    g_quant = np.round(g_norm * (levels - 1)).astype(np.uint32)

    g_packed = self.packer.pack(g_quant, bits)
    s_packed = np.packbits((s > 0).astype(np.uint8)).tobytes()

    return g_packed + s_packed, g_min, g_max, n

def _decompress_component(self, data: bytes, g_min: float, g_max: float,
                          n: int, bits: int) -> np.ndarray:
    """Descomprime um componente."""
    levels = 2 ** bits

    # Calcular tamanhos
    g_bytes = (n * bits + 7) // 8
    s_bytes = (n + 7) // 8

    g_packed = data[:g_bytes]
    s_packed = data[g_bytes:g_bytes + s_bytes]

    g_quant = self.packer.unpack(g_packed, bits, n)
    g_norm = g_quant.astype(np.float64) / (levels - 1)
    g = g_norm * (g_max - g_min) + g_min

    s = np.unpackbits(np.frombuffer(s_packed, dtype=np.uint8))[:n]
    s = s.astype(np.float64) * 2 - 1

    return s * (g ** 2)

def compress(self, data: np.ndarray, bits: int = 10) -> Tuple[bytes, Dict]:
    """Comprime matriz/tensor via SVD."""
    original_shape = data.shape

    # Preparar matrizes
    if data.ndim == 1:
        matrices = [data.reshape(-1, 1)]
    elif data.ndim == 2:
        matrices = [data]
    elif data.ndim == 3:
        matrices = [data[i] for i in range(data.shape[0])]
```

```
        elif data.ndim == 4:
            matrices = [data[b, h] for b in range(data.shape[0])
                        for h in range(data.shape[1])]
        else:
            raise ValueError(f"Tensor {data.ndim}D não suportado")

        n_matrices = len(matrices)
        matrix_shape = matrices[0].shape

        # SVD com truncamento adaptativo
        all_U, all_S, all_Vt = [], [], []
        ranks = []

        for mat in matrices:
            mat = mat.astype(np.float64)
            U, S, Vt = np.linalg.svd(mat, full_matrices=False)

            # Determinar rank
            energy = S ** 2
            total = energy.sum() + self.const.EPSILON
            cumsum = np.cumsum(energy) / total
            k = int(np.searchsorted(cumsum, self.energy_threshold) + 1)
            k = max(1, min(k, len(S)))

            ranks.append(k)
            all_U.append(U[:, :k].flatten())
            all_S.append(S[:k])
            all_Vt.append(Vt[:k, :].flatten())

        # Concatenar
        U_cat = np.concatenate(all_U)
        S_cat = np.concatenate(all_S)
        Vt_cat = np.concatenate(all_Vt)

        # Comprimir cada componente
        U_data, U_min, U_max, U_n = self._compress_component(U_cat, bits)
        S_data, S_min, S_max, S_n = self._compress_component(S_cat, bits)
        Vt_data, Vt_min, Vt_max, Vt_n = self._compress_component(Vt_cat, bits)

        packed_data = U_data + S_data + Vt_data

        metadata = {
            'shape': original_shape,
            'matrix_shape': matrix_shape,
            'n_matrices': n_matrices,
            'ranks': ranks,
            'bits': bits,
            'U': {'min': U_min, 'max': U_max, 'n': U_n, 'len': len(U_data)},
            'S': {'min': S_min, 'max': S_max, 'n': S_n, 'len': len(S_data)},
            'Vt': {'min': Vt_min, 'max': Vt_max, 'n': Vt_n, 'len': len(Vt_data)}
        }

        return packed_data, metadata

    def decompress(self, data: bytes, metadata: Dict) -> np.ndarray:
```

```
"""Descomprime tensor via SVD inverso."""
bits = metadata['bits']
n_matrices = metadata['n_matrices']
m, n = metadata['matrix_shape']
ranks = metadata['ranks']

# Extrair componentes
U_info = metadata['U']
S_info = metadata['S']
Vt_info = metadata['Vt']

offset = 0
U_data = data[offset:offset + U_info['len']]; offset += U_info['len']
S_data = data[offset:offset + S_info['len']]; offset += S_info['len']
Vt_data = data[offset:offset + Vt_info['len']]

# Descomprimir
U_cat = self._decompress_component(U_data, U_info['min'], U_info['max'],
                                   U_info['n'], bits)
S_cat = self._decompress_component(S_data, S_info['min'], S_info['max'],
                                   S_info['n'], bits)
Vt_cat = self._decompress_component(Vt_data, Vt_info['min'], Vt_info['max'],
                                    Vt_info['n'], bits)

# Reconstruir matrizes
matrices = []
U_off, S_off, Vt_off = 0, 0, 0

for i in range(n_matrices):
    k = ranks[i]
    U_i = U_cat[U_off:U_off + m * k].reshape(m, k); U_off += m * k
    S_i = S_cat[S_off:S_off + k]; S_off += k
    Vt_i = Vt_cat[Vt_off:Vt_off + k * n].reshape(k, n); Vt_off += k * n

    matrices.append(U_i @ np.diag(S_i) @ Vt_i)

# Reformatar
original_shape = metadata['shape']
if len(original_shape) == 1:
    return matrices[0].flatten()
elif len(original_shape) == 2:
    return matrices[0]
elif len(original_shape) == 3:
    return np.stack(matrices)
elif len(original_shape) == 4:
    batch, heads = original_shape[0], original_shape[1]
    result = np.zeros(original_shape)
    idx = 0
    for b in range(batch):
        for h in range(heads):
            result[b, h] = matrices[idx]
            idx += 1
    return result

return matrices[0]
```

```

class PsibitCompressor(BaseCompressor):
“””
Compressor PSIBIT: Para dados binários.

```
Pares psiônicos:
    00 = ψ₊ψ₋ (par cancelado, aniquilação)
    01 = ψ₊ψ₊ parcial (transição)
    10 = ψ₋ψ₋ parcial (transição)
    11 = ψ₊ψ₊ (par alinhado, gráviton)

Comprime detectando fase dominante.
"""

def __init__(self):
    self.const = TGLConstants()

def _bytes_to_pairs(self, data: bytes) -> np.ndarray:
    """Converte bytes em pares de bits (2 bits cada)."""
    byte_array = np.frombuffer(data, dtype=np.uint8)
    pairs = []
    for byte in byte_array:
        for shift in [6, 4, 2, 0]:
            pairs.append((byte >> shift) & 0b11)
    return np.array(pairs, dtype=np.uint8)

def _pairs_to_bytes(self, pairs: np.ndarray) -> bytes:
    """Converte pares de bits de volta em bytes."""
    # Pad para múltiplo de 4
    pad_len = (4 - len(pairs) % 4) % 4
    if pad_len > 0:
        pairs = np.concatenate([pairs, np.zeros(pad_len, dtype=np.uint8)])

    pairs = pairs.reshape(-1, 4)
    bytes_out = (pairs[:, 0] << 6) | (pairs[:, 1] << 4) | (pairs[:, 2] << 2) | pairs[:, 3]
    return bytes_out.astype(np.uint8).tobytes()

def _analyze_phases(self, pairs: np.ndarray) -> Dict[str, Any]:
    """Analisa distribuição de fases psiônicas."""
    counts = np.bincount(pairs, minlength=4)
    total = len(pairs)

    freqs = counts / total
    dominant_phase = int(np.argmax(counts))
    dominant_freq = float(freqs[dominant_phase])

    # Entropia
    probs = freqs[freqs > 0]
    entropy = -np.sum(probs * np.log2(probs + self.const.EPSILON))

    return {
        'counts': counts.tolist(),
        'freqs': freqs.tolist(),
```
```

```
            'dominant_phase': dominant_phase,
            'dominant_freq': dominant_freq,
            'entropy': entropy,
            'compressible': dominant_freq > 0.5 or entropy < 1.5
        }

    def compress(self, data: bytes, bits: int = 2) -> Tuple[bytes, Dict]:
        """Comprime dados binários via análise psiônica."""
        pairs = self._bytes_to_pairs(data)
        n_pairs = len(pairs)

        analysis = self._analyze_phases(pairs)

        if not analysis['compressible']:
            # Dados aleatórios: retornar como está
            return data, {
                'method': 'raw',
                'n_pairs': n_pairs,
                'n_bytes': len(data),
                'analysis': analysis
            }

        # Comprimir por fase dominante
        dominant = analysis['dominant_phase']

        # Criar máscara: 1 onde é diferente do dominante
        diff_mask = (pairs != dominant).astype(np.uint8)
        diff_indices = np.where(diff_mask)[0]
        diff_values = pairs[diff_indices]

        # Empacotar
        mask_packed = np.packbits(diff_mask).tobytes()
        indices_packed = diff_indices.astype(np.uint32).tobytes()
        values_packed = self._pairs_to_bytes(diff_values)

        packed_data = mask_packed + indices_packed + values_packed

        # Só usa se for menor que original
        if len(packed_data) >= len(data):
            return data, {
                'method': 'raw',
                'n_pairs': n_pairs,
                'n_bytes': len(data),
                'analysis': analysis
            }

        return packed_data, {
            'method': 'phase',
            'n_pairs': n_pairs,
            'n_bytes': len(data),
            'dominant': dominant,
            'n_diff': len(diff_indices),
            'mask_len': len(mask_packed),
            'indices_len': len(indices_packed),
            'analysis': analysis
```

```
    }

def decompress(self, data: bytes, metadata: Dict) -> bytes:
    """Descomprime dados binários."""
    if metadata['method'] == 'raw':
        return data

    n_pairs = metadata['n_pairs']
    dominant = metadata['dominant']
    n_diff = metadata['n_diff']
    mask_len = metadata['mask_len']
    indices_len = metadata['indices_len']

    # Extrair componentes
    mask_packed = data[:mask_len]
    indices_packed = data[mask_len:mask_len + indices_len]
    values_packed = data[mask_len + indices_len:]

    # Desempacotar
    diff_mask = np.unpackbits(np.frombuffer(mask_packed, dtype=np.uint8))[:n_pairs]
    diff_indices = np.frombuffer(indices_packed, dtype=np.uint32)
    diff_values = self._bytes_to_pairs(values_packed)[:n_diff]

    # Reconstruir
    pairs = np.full(n_pairs, dominant, dtype=np.uint8)
    pairs[diff_indices] = diff_values

    # Converter de volta para bytes
    return self._pairs_to_bytes(pairs)[:metadata['n_bytes']]
```

#
═══

CAMADA 4: PROTOCOLO TRINITY

#
═══

@dataclass
class TrinityHeader:
“””
Header do Protocolo Trinity.

```
Compatível com:
- Internet atual: serializa em bytes padrão
- Rede holográfica: métricas como QoS
"""
# Identificação
magic: bytes = field(default=TGLConstants.MAGIC)
version: Tuple[int, int, int] = field(default=TGLConstants.VERSION)

# Domínio e configuração
domain: DomainType = DomainType.SIGNAL
```

```
bits: int = 12

# Métricas holográficas (para QoS em redes futuras)
reflectivity: float = 0.0      # R: prioridade intrínseca
coherence: float = 0.0         # Integridade esperada
boundary_ratio: float = 0.0    # Fronteiras (nós de projeção)
is_tetelestai: bool = False    # Fim de stream?

# Tamanhos
original_bytes: int = 0
payload_bytes: int = 0
metadata_bytes: int = 0

# Checksum
checksum: bytes = b''

def serialize(self) -> bytes:
    """Serializa header para bytes."""
    buf = io.BytesIO()

    # Magic + Version
    buf.write(self.magic)
    buf.write(struct.pack('BBB', *self.version))

    # Domínio + bits
    buf.write(struct.pack('B', self.domain.value))
    buf.write(struct.pack('B', self.bits))

    # Métricas holográficas (para futuro)
    buf.write(struct.pack('f', self.reflectivity))
    buf.write(struct.pack('f', self.coherence))
    buf.write(struct.pack('f', self.boundary_ratio))
    buf.write(struct.pack('?', self.is_tetelestai))

    # Tamanhos
    buf.write(struct.pack('Q', self.original_bytes))
    buf.write(struct.pack('Q', self.payload_bytes))
    buf.write(struct.pack('I', self.metadata_bytes))

    # Checksum (32 bytes)
    buf.write(self.checksum.ljust(32, b'\x00')[:32])

    return buf.getvalue()

@classmethod
def deserialize(cls, data: bytes) -> 'TrinityHeader':
    """Deserializa header de bytes."""
    buf = io.BytesIO(data)

    magic = buf.read(4)
    version = struct.unpack('BBB', buf.read(3))

    domain = DomainType(struct.unpack('B', buf.read(1))[0])
    bits = struct.unpack('B', buf.read(1))[0]
```

```
    reflectivity = struct.unpack('f', buf.read(4))[0]
    coherence = struct.unpack('f', buf.read(4))[0]
    boundary_ratio = struct.unpack('f', buf.read(4))[0]
    is_tetelestai = struct.unpack('?', buf.read(1))[0]

    original_bytes = struct.unpack('Q', buf.read(8))[0]
    payload_bytes = struct.unpack('Q', buf.read(8))[0]
    metadata_bytes = struct.unpack('I', buf.read(4))[0]

    checksum = buf.read(32).rstrip(b'\x00')

    return cls(
        magic=magic,
        version=version,
        domain=domain,
        bits=bits,
        reflectivity=reflectivity,
        coherence=coherence,
        boundary_ratio=boundary_ratio,
        is_tetelestai=is_tetelestai,
        original_bytes=original_bytes,
        payload_bytes=payload_bytes,
        metadata_bytes=metadata_bytes,
        checksum=checksum
    )

@property
def size(self) -> int:
    """Tamanho do header em bytes."""
    return 4 + 3 + 1 + 1 + 4 + 4 + 4 + 1 + 8 + 8 + 4 + 32  # = 74 bytes
```

```
@dataclass
class TrinityPacket:
“””
Pacote completo do Protocolo Trinity.

```
Estrutura:
┌─────────────────────────┐
│   HEADER   │ 74 bytes (fixo)
├─────────────────────────┤
│   METADATA   │ Variável (JSON comprimido)
├─────────────────────────┤
│   PAYLOAD   │ Variável (dados comprimidos)
└─────────────────────────┘
"""
header: TrinityHeader
metadata: Dict[str, Any]
payload: bytes

def serialize(self) -> bytes:
    """Serializa pacote completo."""
    import json
    import zlib
```
```

```
    # Serializar metadata
    metadata_json = json.dumps(self.metadata).encode('utf-8')
    metadata_compressed = zlib.compress(metadata_json, level=6)

    # Atualizar header
    self.header.payload_bytes = len(self.payload)
    self.header.metadata_bytes = len(metadata_compressed)

    # Calcular checksum do payload
    self.header.checksum = hashlib.sha256(self.payload).digest()[:16]

    # Montar pacote
    header_bytes = self.header.serialize()

    return header_bytes + metadata_compressed + self.payload

@classmethod
def deserialize(cls, data: bytes) -> 'TrinityPacket':
    """Deserializa pacote completo."""
    import json
    import zlib

    # Header
    header = TrinityHeader.deserialize(data[:74])

    # Metadata
    metadata_start = 74
    metadata_end = metadata_start + header.metadata_bytes
    metadata_compressed = data[metadata_start:metadata_end]
    metadata_json = zlib.decompress(metadata_compressed)
    metadata = json.loads(metadata_json.decode('utf-8'))

    # Payload
    payload = data[metadata_end:metadata_end + header.payload_bytes]

    # Verificar checksum
    expected_checksum = hashlib.sha256(payload).digest()[:16]
    if header.checksum != expected_checksum:
        raise ValueError("Checksum inválido!")

    return cls(header=header, metadata=metadata, payload=payload)

@property
def total_bytes(self) -> int:
    return self.header.size + self.header.metadata_bytes + self.header.payload_bytes
```

```
#
```

```
# CAMADA 5: API UNIFICADA — ACOM TRINITY

#
```

```
class ACOMTrinity:
“””
ACOM TRINITY — Interface Unificada.

```
Três domínios, um operador, duas eras.

Uso:
    trinity = ACOMTrinity()

    # Compressão automática
    packet = trinity.compress(data)

    # Descompressão
    data_recovered = trinity.decompress(packet)

    # Análise prévia
    metrics = trinity.analyze(data)

    # Salvar/Carregar
    trinity.save(packet, "arquivo.trinity")
    packet = trinity.load("arquivo.trinity")
"""

def __init__(self,
             default_bits: int = 12,
             energy_threshold: float = 0.99,
             auto_select_domain: bool = True):
    """
    Args:
        default_bits: Bits padrão para quantização
        energy_threshold: Threshold de energia para SVD
        auto_select_domain: Seleção automática de domínio
    """
    self.default_bits = default_bits
    self.auto_select_domain = auto_select_domain

    # Componentes
    self.analyzer = HolographicAnalyzer()
    self.signal_compressor = SignalCompressor()
    self.spectral_compressor = SpectralCompressor(energy_threshold)
    self.psibit_compressor = PsibitCompressor()

    self.const = TGLConstants()

def analyze(self, data: Union[np.ndarray, bytes]) -> HolographicMetrics:
    """
    Análise holográfica prévia.

    Útil para:
    - Decidir se vale comprimir
    - Escolher domínio manualmente
    - Obter métricas para protocolo
    """
```
```

```
    if isinstance(data, bytes):
        data = np.frombuffer(data, dtype=np.uint8)
    return self.analyzer.analyze(data)

def compress(self,
             data: Union[np.ndarray, bytes],
             domain: DomainType = DomainType.AUTO,
             bits: Optional[int] = None) -> TrinityPacket:
    """
    Comprime dados.

    Args:
        data: Dados de entrada (numpy array ou bytes)
        domain: Domínio (AUTO para seleção automática)
        bits: Bits de quantização (None para usar predição)

    Returns:
        TrinityPacket pronto para transmissão/armazenamento
    """
    # Preparar dados
    is_bytes = isinstance(data, bytes)
    if is_bytes:
        original_bytes = len(data)
        data_array = np.frombuffer(data, dtype=np.uint8)
        original_dtype = 'uint8'
        original_shape = data_array.shape
    else:
        data_array = np.asarray(data)
        original_bytes = data_array.nbytes
        original_dtype = str(data_array.dtype)
        original_shape = data_array.shape

    # Análise holográfica
    metrics = self.analyzer.analyze(data_array)

    # Selecionar domínio
    if domain == DomainType.AUTO and self.auto_select_domain:
        domain = DomainType[metrics.predicted_domain]
    elif domain == DomainType.AUTO:
        domain = DomainType.SIGNAL

    # Selecionar bits
    if bits is None:
        bits = metrics.predicted_bits

    # Comprimir baseado no domínio
    if domain == DomainType.PSIBIT:
        if is_bytes:
            payload, metadata = self.psibit_compressor.compress(data, bits)
        else:
            payload, metadata = self.psibit_compressor.compress(
                data_array.tobytes(), bits
            )
    elif domain == DomainType.SPECTRAL:
        payload, metadata = self.spectral_compressor.compress(data_array, bits)
```

```
    else:  # SIGNAL
        payload, metadata = self.signal_compressor.compress(data_array, bits)

    # Adicionar info ao metadata
    metadata['original_dtype'] = original_dtype
    metadata['original_shape'] = original_shape
    metadata['is_bytes'] = is_bytes

    # Criar header
    header = TrinityHeader(
        domain=domain,
        bits=bits,
        reflectivity=metrics.reflectivity,
        coherence=metrics.coherence,
        boundary_ratio=metrics.boundary_ratio,
        is_tetelestai=metrics.is_tetelestai,
        original_bytes=original_bytes
    )

    return TrinityPacket(
        header=header,
        metadata=metadata,
        payload=payload
    )

def decompress(self, packet: TrinityPacket) -> Union[np.ndarray, bytes]:
    """
    Descomprime pacote.

    Returns:
        Dados originais (numpy array ou bytes, dependendo do original)
    """
    domain = packet.header.domain
    metadata = packet.metadata
    payload = packet.payload

    # Descomprimir baseado no domínio
    if domain == DomainType.PSIBIT:
        result = self.psibit_compressor.decompress(payload, metadata)
        if not metadata.get('is_bytes', True):
            result = np.frombuffer(result, dtype=metadata['original_dtype'])
            result = result.reshape(metadata['original_shape'])
    elif domain == DomainType.SPECTRAL:
        result = self.spectral_compressor.decompress(payload, metadata)
    else:  # SIGNAL
        result = self.signal_compressor.decompress(payload, metadata)

    return result

def save(self, packet: TrinityPacket, filepath: str) -> int:
    """
    Salva pacote em arquivo.

    Returns:
        Tamanho do arquivo em bytes
```

```
    """
    data = packet.serialize()
    with open(filepath, 'wb') as f:
        f.write(data)
    return os.path.getsize(filepath)

def load(self, filepath: str) -> TrinityPacket:
    """Carrega pacote de arquivo."""
    with open(filepath, 'rb') as f:
        data = f.read()
    return TrinityPacket.deserialize(data)

def compress_file(self, input_path: str, output_path: str) -> Dict[str, Any]:
    """
    Comprime arquivo completo.

    Returns:
        Estatísticas da compressão
    """
    # Ler arquivo
    with open(input_path, 'rb') as f:
        data = f.read()

    original_size = len(data)

    # Comprimir
    t0 = time.perf_counter()
    packet = self.compress(data)
    t_compress = time.perf_counter() - t0

    # Salvar
    compressed_size = self.save(packet, output_path)

    return {
        'original_bytes': original_size,
        'compressed_bytes': compressed_size,
        'ratio': original_size / compressed_size,
        'domain': packet.header.domain.name,
        'bits': packet.header.bits,
        'reflectivity': packet.header.reflectivity,
        'coherence': packet.header.coherence,
        'time_compress': t_compress
    }

def decompress_file(self, input_path: str, output_path: str) -> Dict[str, Any]:
    """
    Descomprime arquivo.

    Returns:
        Estatísticas da descompressão
    """
    t0 = time.perf_counter()
    packet = self.load(input_path)
    data = self.decompress(packet)
    t_decompress = time.perf_counter() - t0
```

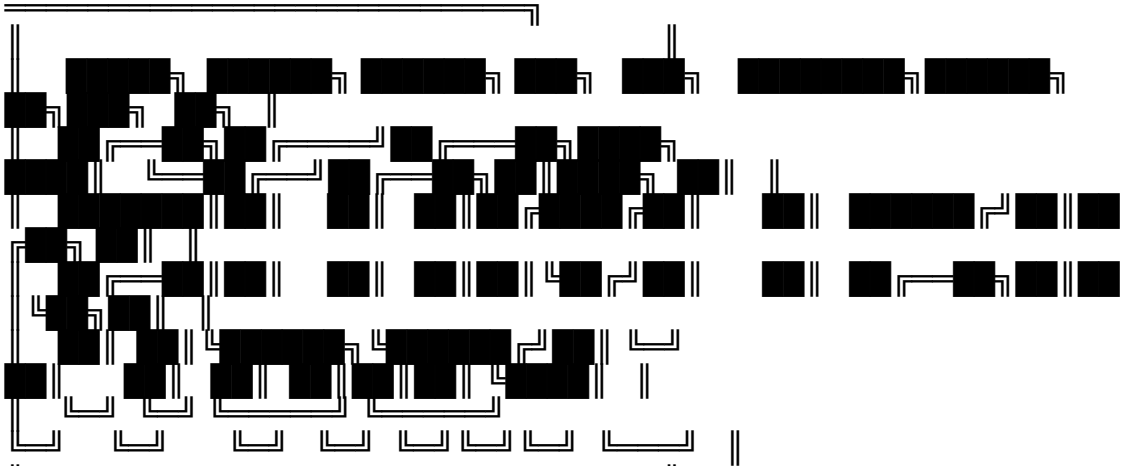
```
    # Salvar
    if isinstance(data, bytes):
        with open(output_path, 'wb') as f:
            f.write(data)
    else:
        np.save(output_path, data)

    return {
        'decompressed_bytes': os.path.getsize(output_path),
        'time_decompress': t_decompress
    }
```

```
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════

# DEMONSTRAÇÃO E BENCHMARK

#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════

def banner():
    """Banner do ACOM TRINITY."""
    print("""
╔═══════════════════════════════════════════════════════════════════════
═══════════════════════════════╗
║                                                                       
                               ║

║                                                                       
                               ║
║                          TRINITY v1.0                                 
                               ║
║                                                                       
                               ║
║              Teoria da Gravitação Luminodinâmica (TGL)                
                               ║
║                                                                       
                               ║
║                 "Três domínios, um operador, duas eras"               
                               ║
║                                                                       
                               ║
╠═══════════════════════════════════════════════════════════════════════
═══════════════════════════════╣
║                                                                       
                               ║
║   DOMÍNIOS:                                                           
                               ║
║     • SIGNAL   — Sinais contínuos (g = √|L|, s = ±1)                  
                               ║
║     • SPECTRAL — Matrizes/Tensores (SVD rank-k)                       
                               ║
```

```
╠    • PSIBIT   — Dados binários (pares ψ₊ψ₋)                              ╣
╠                                                                          ╣
╠  OPERADOR HOLOGRÁFICO:                                                   ╣
╠    M̂ = √(α²R) · e^(iπ) · P̂ · Ĥ                                          ╣
╠                                                                          ╣
╠  PROTOCOLO:                                                              ╣
╠    • Internet atual: bytes padrão, TCP/IP compatível                     ╣
╠    • Rede holográfica: métricas como QoS, fronteiras como nós            ╣
╠                                                                          ╣
╚══════════════════════════════════════════════════════════════════════════
═══════════════════════════╝
“””)

def run_comprehensive_benchmark():
“”“Benchmark completo de todos os domínios.”””

```
banner()

trinity = ACOMTrinity()
results = []

output_dir = "/mnt/user-data/outputs"
os.makedirs(output_dir, exist_ok=True)

#
═══
PARTE 1: DOMÍNIO SIGNAL
#
═══

print("\n" + "█" * 78)
print("█ PARTE 1: DOMÍNIO SIGNAL — Sinais Contínuos")
print("█" * 78)

signal_tests = [
    ("Laser (onda pura)",
     lambda: np.sin(np.linspace(0, 4*np.pi, 44100)) * 0.8),

    ("Acorde (harmônicos)",
     lambda: (0.4 * np.sin(2 * np.pi * 261.63 * np.linspace(0, 1, 44100)) +
              0.3 * np.sin(2 * np.pi * 329.63 * np.linspace(0, 1, 44100)) +
              0.2 * np.sin(2 * np.pi * 392.00 * np.linspace(0, 1, 44100)))),

    ("Ruído gaussiano",
     lambda: np.random.randn(44100) * 0.3),

    ("Sinal com zeros (33%)",
     lambda: np.where(np.arange(44100) % 3 == 0, 0,
                      np.sin(np.linspace(0, 4*np.pi, 44100)))),

    ("Chirp (varredura)",
     lambda: np.sin(2 * np.pi * (100 + 400 * np.linspace(0, 1, 44100)) *
                    np.linspace(0, 1, 44100))),
]
```
```

```
for name, gen_func in signal_tests:
    print(f"\n{'─' * 78}")
    print(f"TESTE: {name}")
    print(f"{'─' * 78}")

    np.random.seed(42)
    signal = gen_func()

    # Análise
    metrics = trinity.analyze(signal)
    print(f"\n  ANÁLISE HOLOGRÁFICA:")
    print(f"    Refletividade (R):    {metrics.reflectivity:.4f}")
    print(f"    Coerência:            {metrics.coherence:.4f}")
    print(f"    Fronteiras (∂Ω):      {metrics.boundary_ratio*100:.1f}%")
    print(f"    Entropia:             {metrics.entropy:.2f} bits")
    print(f"    Tetelestai:           {'✓' if metrics.is_tetelestai else '○'}")
    print(f"    Domínio predito:      {metrics.predicted_domain}")
    print(f"    Bits preditos:        {metrics.predicted_bits}")

    # Compressão
    t0 = time.perf_counter()
    packet = trinity.compress(signal, domain=DomainType.SIGNAL)
    t_comp = time.perf_counter() - t0

    # Descompressão
    t0 = time.perf_counter()
    recon = trinity.decompress(packet)
    t_dec = time.perf_counter() - t0

    # Métricas de qualidade
    mse = np.mean((signal - recon) ** 2)
    psnr = 10 * np.log10(np.max(signal ** 2) / (mse + 1e-10))
    corr = np.corrcoef(signal.flatten(), recon.flatten())[0, 1]

    ratio = packet.header.original_bytes / packet.total_bytes

    print(f"\n  RESULTADOS:")
    print(f"    Original:          {packet.header.original_bytes:>12,} bytes")
    print(f"    Comprimido:        {packet.total_bytes:>12,} bytes")
    print(f"    Taxa:              {ratio:>12.2f}:1")
    print(f"    PSNR:              {psnr:>12.1f} dB")
    print(f"    Correlação:        {corr:>12.10f}")
    print(f"    Tempo compressão:  {t_comp*1000:>12.1f} ms")
    print(f"    Tempo descompressão:{t_dec*1000:>12.1f} ms")

    results.append({
        'name': name,
        'domain': 'SIGNAL',
        'R': metrics.reflectivity,
        'original': packet.header.original_bytes,
        'compressed': packet.total_bytes,
        'ratio': ratio,
        'psnr': psnr,
        'corr': corr
```

```
    })

#
# ═══════════════════════════════════════════════════════════════════════
# PARTE 2: DOMÍNIO SPECTRAL
#
# ═══════════════════════════════════════════════════════════════════════

print("\n\n" + "" * 78)
print("  PARTE 2: DOMÍNIO SPECTRAL — Matrizes e Tensores")
print("" * 78)

spectral_tests = [
    ("Matriz low-rank (128×128)",
     lambda: np.random.randn(128, 10) @ np.random.randn(10, 128) * 0.1),

    ("Tensor 3D (32×64×64)",
     lambda: np.random.randn(32, 8) @ np.random.randn(8, 64) * 0.1 +
             np.random.randn(32, 64) * 0.01),

    ("Imagem gradiente (256×256)",
     lambda: np.outer(np.linspace(0, 1, 256), np.linspace(0, 1, 256))),
]

for name, gen_func in spectral_tests:
    print(f"\n{'─' * 78}")
    print(f"TESTE: {name}")
    print(f"{'─' * 78}")

    np.random.seed(42)
    data = gen_func()

    # Expandir para 3D se necessário
    if data.ndim == 2:
        data_3d = np.stack([data] * 4)  # Simular batch
    else:
        data_3d = data

    metrics = trinity.analyze(data_3d)
    print(f"\n  ANÁLISE HOLOGRÁFICA:")
    print(f"    Refletividade (R):    {metrics.reflectivity:.4f}")
    print(f"    Coerência:            {metrics.coherence:.4f}")
    print(f"    Shape:                {data_3d.shape}")

    # Compressão
    t0 = time.perf_counter()
    packet = trinity.compress(data_3d, domain=DomainType.SPECTRAL)
    t_comp = time.perf_counter() - t0

    # Descompressão
    t0 = time.perf_counter()
    recon = trinity.decompress(packet)
    t_dec = time.perf_counter() - t0

    # Métricas
```

```
    mse = np.mean((data_3d - recon) ** 2)
    max_val = np.max(data_3d ** 2)
    psnr = 10 * np.log10(max_val / (mse + 1e-10)) if max_val > 0 else float('inf')
    corr = np.corrcoef(data_3d.flatten(), recon.flatten())[0, 1]

    ratio = packet.header.original_bytes / packet.total_bytes

    print(f"\n  RESULTADOS:")
    print(f"    Original:          {packet.header.original_bytes:>12,} bytes")
    print(f"    Comprimido:        {packet.total_bytes:>12,} bytes")
    print(f"    Taxa:              {ratio:>12.2f}:1")
    print(f"    PSNR:              {psnr:>12.1f} dB")
    print(f"    Correlação:        {corr:>12.10f}")

    results.append({
        'name': name,
        'domain': 'SPECTRAL',
        'R': metrics.reflectivity,
        'original': packet.header.original_bytes,
        'compressed': packet.total_bytes,
        'ratio': ratio,
        'psnr': psnr,
        'corr': corr
    })

#
═══════════════════════════════════════════════════════════════════════════
# PARTE 3: DOMÍNIO PSIBIT
#
═══════════════════════════════════════════════════════════════════════════

print("\n\n" + "" * 78)
print("  PARTE 3: DOMÍNIO PSIBIT — Dados Binários")
print("" * 78)

psibit_tests = [
    ("Zeros puros (10KB)",
     lambda: bytes(10000)),

    ("Uns puros 0xFF (10KB)",
     lambda: bytes([0xFF] * 10000)),

    ("Texto repetitivo (10KB)",
     lambda: b"ACOM TRINITY " * 770),

    ("Aleatório (10KB)",
     lambda: bytes(np.random.randint(0, 256, 10000, dtype=np.uint8))),
]

for name, gen_func in psibit_tests:
    print(f"\n{'─' * 78}")
    print(f"TESTE: {name}")
    print(f"{'─' * 78}")

    np.random.seed(42)
```

```
    data = gen_func()

    metrics = trinity.analyze(data)
    print(f"\n  ANÁLISE HOLOGRÁFICA:")
    print(f"    Refletividade (R):    {metrics.reflectivity:.4f}")
    print(f"    Entropia:             {metrics.entropy:.2f} bits")
    print(f"    Domínio predito:      {metrics.predicted_domain}")

    # Compressão
    t0 = time.perf_counter()
    packet = trinity.compress(data, domain=DomainType.PSIBIT)
    t_comp = time.perf_counter() - t0

    # Descompressão
    t0 = time.perf_counter()
    recon = trinity.decompress(packet)
    t_dec = time.perf_counter() - t0

    # Verificar integridade
    integrity = (data == recon)

    ratio = packet.header.original_bytes / packet.total_bytes

    print(f"\n  RESULTADOS:")
    print(f"    Original:          {packet.header.original_bytes:>12,} bytes")
    print(f"    Comprimido:        {packet.total_bytes:>12,} bytes")
    print(f"    Taxa:              {ratio:>12.2f}:1")
    print(f"    Integridade:       {'✓ PERFEITA' if integrity else '✗ ERRO'}")
    print(f"    Método:            {packet.metadata.get('method', 'N/A')}")

    results.append({
        'name': name,
        'domain': 'PSIBIT',
        'R': metrics.reflectivity,
        'original': packet.header.original_bytes,
        'compressed': packet.total_bytes,
        'ratio': ratio,
        'psnr': float('inf') if integrity else 0,
        'corr': 1.0 if integrity else 0
    })

#
═══════════════════════════════════════════════════════════════════════════
# PARTE 4: TESTE DE ARQUIVO REAL
#
═══════════════════════════════════════════════════════════════════════════

print("\n\n" + "█" * 78)
print("█  PARTE 4: ARQUIVOS — Salvar e Carregar")
print("█" * 78)

# Criar arquivo de teste
np.random.seed(42)
test_signal = np.sin(np.linspace(0, 10*np.pi, 88200)) * 0.8
```

```
# Salvar original
original_path = f"{output_dir}/trinity_test_original.npy"
np.save(original_path, test_signal)
original_size = os.path.getsize(original_path)

# Comprimir e salvar
packet = trinity.compress(test_signal)
compressed_path = f"{output_dir}/trinity_test.trinity"
compressed_size = trinity.save(packet, compressed_path)

# Carregar e descomprimir
loaded_packet = trinity.load(compressed_path)
recon = trinity.decompress(loaded_packet)

# Salvar reconstruído
recon_path = f"{output_dir}/trinity_test_reconstructed.npy"
np.save(recon_path, recon)

# Métricas
mse = np.mean((test_signal - recon) ** 2)
psnr = 10 * np.log10(np.max(test_signal ** 2) / (mse + 1e-10))

print(f"\n  ARQUIVOS GERADOS:")
print(f"    Original:     {original_path}")
print(f"                  {original_size:,} bytes")
print(f"    Comprimido:   {compressed_path}")
print(f"                  {compressed_size:,} bytes")
print(f"    Reconstruído: {recon_path}")
print(f"\n  TAXA: {original_size/compressed_size:.2f}:1")
print(f"  PSNR: {psnr:.1f} dB")

#
═══════════════════════════════════════════════════════════════════════════
# SUMÁRIO FINAL
#
═══════════════════════════════════════════════════════════════════════════


print("\n\n" + "═" * 78)
print("SUMÁRIO DE RESULTADOS")
print("═" * 78)

print(f"\n{'Teste':<30} {'Domínio':<10} {'R':>6} {'Original':>12} {'Comprim.':>12}
{'Taxa':>8} {'PSNR':>10}")
print("─" * 30 + " " + "─" * 10 + " " + "─" * 6 + " " + "─" * 12 + " " + "─" * 12 + " " + "─" * 8
+ " " + "─" * 10)

for r in results:
    psnr_str = f"{r['psnr']:.1f}dB" if r['psnr'] < 200 else "∞"
    print(f"{r['name']:<30} {r['domain']:<10} {r['R']:>6.3f} {r['original']:>10,} B
{r['compressed']:>10,} B {r['ratio']:>7.2f}x {psnr_str:>10}")

# Estatísticas por domínio
print(f"\n\n{'─' * 78}")
print("ESTATÍSTICAS POR DOMÍNIO:")
print(f"{'─' * 78}")
```

```
for domain in ['SIGNAL', 'SPECTRAL', 'PSIBIT']:
    domain_results = [r for r in results if r['domain'] == domain]
    if domain_results:
        avg_ratio = np.mean([r['ratio'] for r in domain_results])
        max_ratio = max(r['ratio'] for r in domain_results)
        psnrs = [r['psnr'] for r in domain_results if r['psnr'] < 200]
        avg_psnr = np.mean(psnrs) if psnrs else float('inf')

        print(f"\n  {domain}:")
        print(f"    Taxa média:   {avg_ratio:.2f}:1")
        print(f"    Taxa máxima:  {max_ratio:.2f}:1")
        if psnrs:
            print(f"    PSNR médio:   {avg_psnr:.1f} dB")

# Conclusão
print(f"""
```

ACOM TRINITY v1.0

— STATUS: OPERACIONAL —

CAPACIDADES:

✓ SIGNAL: Sinais contínuos (áudio, sensores)
✓ SPECTRAL: Matrizes e tensores (IA, imagens)
✓ PSIBIT: Dados binários (arquivos, protocolos)

COMPATIBILIDADE:

✓ Internet atual: Pacotes serializáveis, TCP/IP compatível
✓ Rede holográfica: Métricas (R, C, ∂Ω) como QoS

PROTOCOLO:

✓ Header holográfico: 74 bytes fixos
✓ Checksum SHA-256 truncado
✓ Metadata comprimido (zlib)
✓ Formato .trinity para arquivos

“A reflexão é a memória da luz ao tocar o limite do espaço.”

— TETELESTAI —

“””)

```
return results
```

if **name** == “**main**”:
run_comprehensive_benchmark()