```
"""
================================================================================
=====
HORIZON-LOCKED UNIFICATION (HLU) WITH FUNDAMENTAL PIXEL-TIME
================================================================================
=====
 . -


: ZARKAM
:
================================================================================
=====
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
from scipy.integrate import cumulative_trapezoid
from scipy.optimize import brentq
from dataclasses import dataclass, field
from typing import Optional, Tuple, Dict, List, Callable, Union
import warnings

warnings.filterwarnings('ignore')

#
================================================================================
=
#  :
#
================================================================================
=

@dataclass(frozen=True)
class Constants:
    """     ( = c = M_pl = 1)"""
    M_pl: float = 1.0
    c: float = 1.0
    hbar: float = 1.0
    G: float = 1.0
```

```
    #
    H0_phys: float = 67.4  # km/s/Mpc
    H0_natural: float = 67.4 / 299792.458  # 1/Mpc
    Mpc_to_m: float = 3.08567758e22
    km_to_Mpc: float = 1.0 / 3.08567758e19

    #
    @property
    def rho_crit0(self) -> float:
        return 3 * self.H0_natural**2 / (8 * np.pi)

    #     (  )
    sec_per_natural: float = 1.0 / (1.0 / self.H0_natural * self.Mpc_to_m / self.c)

C = Constants()

#
==========================================================================
=
#  :  - (Pixel-Time Structure)
#
==========================================================================
=

class PixelTime:
    """
            .

     :
    1.      (Presentism)
    2.
    3.
    4.

      : T_op |X_n = t_n |X_n
     : |X_{n+1} = U |X_n
    """

    def __init__(self,
                 n_pixels: int = 15000,
                 a_start: float = 1e-4,
                 a_end: float = 1.0):
```

```
        self.n_pixels = n_pixels
        self.a_start = a_start
        self.a_end = a_end

        #
        self.ln_a = np.linspace(np.log(a_start), np.log(a_end), n_pixels)
        self.a = np.exp(self.ln_a)
        self.z = 1.0 / self.a - 1.0

        #    (  )
        self.ordinal_labels = np.arange(n_pixels) / n_pixels

        #    (  )
        self.T_op = np.diag(self.ordinal_labels)

        #    ( )
        self.U_op = np.eye(n_pixels)  # placeholder

    def step_forward(self, state: np.ndarray, n: int) -> np.ndarray:
        """    """
        return self.U_op @ state

    def get_ordinal_time(self, pixel_idx: int) -> float:
        """     """
        return self.ordinal_labels[pixel_idx]

    def __len__(self) -> int:
        return self.n_pixels

    def __repr__(self) -> str:
        return f"PixelTime(n_pixels={self.n_pixels}, a[{self.a_start:.1e}, {self.a_end}])"


#
==========================================================================
=
#  :
#
==========================================================================
=
```

```
class LockedChameleonPotential:
    """
          .

    V(, ) = V e^{-} + (_eff/_c) exp[-(-_c)²/2²]

    :
    -    (_eff > _c):         _c
    -    (_eff < _c):
    -     : m_eff() = m (/_c)^{n/(n+1)}
    """

    def __init__(self,
                 V0: float = 0.7,
                 lambda_V: float = 1.0,
                 phi_c: float = 0.0,
                 sigma_phi: float = 0.1,
                 rho_c_ratio: float = 1.2e-3,
                 m0_eV: float = 1.2e-32,
                 n: float = 2.0,
                 alpha: float = 1.0):

        self.V0 = V0
        self.lambda_V = lambda_V
        self.phi_c = phi_c
        self.sigma_phi = sigma_phi
        self.rho_c_ratio = rho_c_ratio
        self.m0_eV = m0_eV
        self.n = n
        self.alpha = alpha

        # _c
        self.rho_c = rho_c_ratio * C.rho_crit0

        #
        self.m0_natural = m0_eV * (C.sec_per_natural / C.hbar)  # eV  natural

    def __call__(self,
                 phi: Union[float, np.ndarray],
                 rho_eff: Optional[Union[float, np.ndarray]] = None) -> Union[float, np.ndarray]:
        """ """
        #
```

```
    exp_part = self.V0 * np.exp(-self.lambda_V * phi)

    #
    if rho_eff is None:
        return exp_part

    #
    gauss_factor = self.alpha * (rho_eff / self.rho_c_ratio)
    gauss_part = gauss_factor * np.exp(-(phi - self.phi_c)**2 / (2 * self.sigma_phi**2))

    return exp_part + gauss_part

def dV_dphi(self,
            phi: Union[float, np.ndarray],
            rho_eff: Optional[Union[float, np.ndarray]] = None) -> Union[float, np.ndarray]:
    """    """
    d_exp = -self.lambda_V * self.V0 * np.exp(-self.lambda_V * phi)

    if rho_eff is None:
        return d_exp

    gauss_factor = self.alpha * (rho_eff / self.rho_c_ratio)
    gauss = np.exp(-(phi - self.phi_c)**2 / (2 * self.sigma_phi**2))
    d_gauss = -(phi - self.phi_c) / self.sigma_phi**2 * gauss_factor * gauss

    return d_exp + d_gauss

def d2V_dphi2(self,
              phi: Union[float, np.ndarray],
              rho_eff: Optional[Union[float, np.ndarray]] = None) -> Union[float, np.ndarray]:
    """    ( )"""
    d2_exp = self.lambda_V**2 * self.V0 * np.exp(-self.lambda_V * phi)

    if rho_eff is None:
        return d2_exp

    gauss_factor = self.alpha * (rho_eff / self.rho_c_ratio)
    gauss = np.exp(-(phi - self.phi_c)**2 / (2 * self.sigma_phi**2))

    d2_gauss = gauss_factor * gauss / self.sigma_phi**2 * \
               ((phi - self.phi_c)**2 / self.sigma_phi**2 - 1)
```

```
        return d2_exp + d2_gauss

    def effective_mass(self,
                       phi: Union[float, np.ndarray],
                       rho: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
        """     ( )"""
        #
        exponent = self.n / (self.n + 1)
        m_rho = self.m0_natural * (rho / self.rho_c)**exponent

        #
        m_pot = np.sqrt(np.abs(self.d2V_dphi2(phi, rho)))

        return np.sqrt(m_rho**2 + m_pot**2)

    def locking_strength(self, rho_eff: float) -> float:
        """  (  )"""
        x = rho_eff / self.rho_c_ratio
        if x <= 1:
            return 0.0
        return 1.0 - np.exp(-1.0 / (x - 1.0))

    def phi_minimum(self, rho_eff: float) -> float:
        """    """
        if rho_eff > self.rho_c_ratio:
            return self.phi_c
        else:
            #
            return (1.0 / self.lambda_V) * np.log(self.V0 * self.lambda_V / rho_eff)


#
================================================================================
=
#  :  HLU  -
#
================================================================================
=

class HLU_PixelTime_Cosmology:
    """
      HLU  - .
```

  :
1.
2.
3.
4.
5.
"""

```
def __init__(self,
             #
             beta: float = 2.1e-3,

             #
             m0_eV: float = 1.2e-32,
             n: float = 2.0,

             #
             rho_c_ratio: float = 1.2e-3,
             V0: float = 0.7,
             lambda_V: float = 1.0,
             sigma_phi: float = 0.1,
             phi_c: float = 0.0,
             alpha: float = 1.0,

             #
             Omega_m0: float = 0.315,
             Omega_r0: float = 9.2e-5,
             H0: float = 67.4,

             #
             Delta_ln_a: float = 0.0015,
             n_pixels: int = 15000,
             a_start: float = 1e-4,

             #
             adaptive_step: bool = True,
             verbose: bool = True):

    # ---   ---
    self.beta = beta
    self.m0_eV = m0_eV
```

```
self.n = n
self.rho_c_ratio = rho_c_ratio
self.Omega_m0 = Omega_m0
self.Omega_r0 = Omega_r0
self.H0_phys = H0
self.H0_nat = H0 * C.km_to_Mpc * C.c
self.verbose = verbose

# ---  - ---
self.pixel_time = PixelTime(n_pixels=n_pixels, a_start=a_start, a_end=1.0)
self.Delta_ln_a = Delta_ln_a
self.adaptive_step = adaptive_step

# ---    ---
self.potential = LockedChameleonPotential(
    V0=V0, lambda_V=lambda_V,
    phi_c=phi_c, sigma_phi=sigma_phi,
    rho_c_ratio=rho_c_ratio,
    m0_eV=m0_eV, n=n, alpha=alpha
)

# ---    ---
self.N = len(self.pixel_time)
self.a = self.pixel_time.a.copy()
self.ln_a = self.pixel_time.ln_a.copy()
self.z = self.pixel_time.z.copy()

#
self.phi = np.zeros(self.N)
self.pi = np.zeros(self.N)  # d/d(ln a)

#
self.rho_m = np.zeros(self.N)
self.rho_r = np.zeros(self.N)
self.rho_phi = np.zeros(self.N)
self.rho_total = np.zeros(self.N)

#
self.H = np.zeros(self.N)
self.H_km_s_Mpc = np.zeros(self.N)

#
```

```
        self.w_de = np.zeros(self.N)
        self.w_total = np.zeros(self.N)

        #
        self.Omega_m = np.zeros(self.N)
        self.Omega_phi = np.zeros(self.N)

        #
        self.meff = np.zeros(self.N)

        # ---   ---
        self._initialize()

        # ---  ---
        self._evolve()

        # ---    ---
        self._compute_derived()

        if verbose:
            self._print_summary()

    # ----------------------------------------------------------------------
    # .:
    # ----------------------------------------------------------------------

    def _initialize(self):
        """   a_start (  )"""
        i = 0
        a0 = self.a[i]

        #
        self.H[i] = self.H0_nat * np.sqrt(
            self.Omega_m0 / a0**3 + self.Omega_r0 / a0**4
        )
        self.H_km_s_Mpc[i] = self.H[i] / (C.km_to_Mpc * C.c)

        #
        self.rho_m[i] = self.Omega_m0 / a0**3
        self.rho_r[i] = self.Omega_r0 / a0**4

        #
```

```
        self.phi[i] = self.potential.phi_minimum(self.rho_m[i] / self.rho_c_ratio)
        self.pi[i] = 0.0

        #
        self.rho_phi[i] = 0.5 * self.pi[i]**2 + self.potential(self.phi[i], self.rho_m[i])

        #
        self.rho_total[i] = self.rho_m[i] + self.rho_r[i] + self.rho_phi[i]

        #
        self.w_de[i] = -1.0
        self.w_total[i] = (self.rho_r[i]/3 + self.w_de[i]*self.rho_phi[i]) / self.rho_total[i]

        #
        self.Omega_m[i] = self.rho_m[i] / self.rho_total[i]
        self.Omega_phi[i] = self.rho_phi[i] / self.rho_total[i]

        #
        self.meff[i] = self.potential.effective_mass(self.phi[i], self.rho_m[i])

    def _dlnH_dln_a(self, n: int) -> float:
        """  H   ln a   """
        if n < 2:
            return -1.5
        return (np.log(self.H[n-1] / self.H[n-2]) /
                (self.ln_a[n-1] - self.ln_a[n-2]))

    def _adaptive_delta(self, n: int) -> float:
        """        """
        if not self.adaptive_step or n < 2:
            return self.Delta_ln_a

        dphi_dlna = (self.phi[n-1] - self.phi[n-2]) / (self.ln_a[n-1] - self.ln_a[n-2])
        dphi_abs = abs(dphi_dlna)

        if dphi_abs > 0.1:
            return max(self.Delta_ln_a * 0.5, 0.0005)
        elif dphi_abs < 0.001:
            return min(self.Delta_ln_a * 1.5, 0.003)
        else:
            return self.Delta_ln_a
```

```
def _update_step(self, n: int):
    """     """
    #
    Delta = self._adaptive_delta(n)
    a_n = self.a[n]

    #
    phi_prev = self.phi[n-1]
    pi_prev = self.pi[n-1]
    rho_m_prev = self.rho_m[n-1]

    #
    rho_eff = rho_m_prev

    #
    friction = 3.0 + self._dlnH_dln_a(n-1)
    friction = max(friction, 0.1)

    #
    force = (self.potential.dV_dphi(phi_prev, rho_eff) +
             self.beta * rho_m_prev)

    #   ()
    pi_n = pi_prev - Delta * (friction * pi_prev + force)
    pi_n = np.clip(pi_n, -5.0, 5.0)

    #   ()
    phi_n = phi_prev + Delta * pi_n
    phi_n = np.clip(phi_n, -4.0, 4.0)

    #    (  )
    exp_coupling = np.exp(self.beta * (phi_n - phi_prev))
    self.rho_m[n] = (self.rho_m[n-1] * (self.a[n-1] / a_n)**3 * exp_coupling)

    #
    self.rho_r[n] = self.Omega_r0 / a_n**4

    #
    self.rho_phi[n] = 0.5 * pi_n**2 + self.potential(phi_n, rho_eff)

    #
    self.rho_total[n] = self.rho_m[n] + self.rho_r[n] + self.rho_phi[n]
```

```
        #
        H_n_sq = (8 * np.pi / 3) * self.rho_total[n]
        self.H[n] = np.sqrt(max(H_n_sq, 1e-12))
        self.H_km_s_Mpc[n] = self.H[n] / (C.km_to_Mpc * C.c)

        #
        self.phi[n] = phi_n
        self.pi[n] = pi_n

        #
        if self.rho_phi[n] > 1e-30:
            kinetic = 0.5 * pi_n**2
            potential = self.potential(phi_n, rho_eff)
            self.w_de[n] = (kinetic - potential) / self.rho_phi[n]
            self.w_de[n] = np.clip(self.w_de[n], -1.05, 0.05)
        else:
            self.w_de[n] = -1.0

        # w_total
        p_total = self.rho_r[n]/3 + self.w_de[n] * self.rho_phi[n]
        self.w_total[n] = p_total / self.rho_total[n] if self.rho_total[n] > 0 else 0

        #
        self.Omega_m[n] = self.rho_m[n] / self.rho_total[n]
        self.Omega_phi[n] = self.rho_phi[n] / self.rho_total[n]

        #
        self.meff[n] = self.potential.effective_mass(phi_n, rho_eff)

    def _evolve(self):
        """    a_start  a=1"""
        for n in range(1, self.N):
            self._update_step(n)

    def _compute_derived(self):
        """   """
        # ---   ---
        integrand = C.c / self.H
        self.chi = cumulative_trapezoid(integrand, self.z, initial=0) / C.Mpc_to_m

        # ---    ---
```

```
    self.D_A = self.chi / (1 + self.z)

    # ---   ---
    self.D_L = self.chi * (1 + self.z)

    # ---   ---
    self.mu = 5 * np.log10(self.D_L * 1e5) + 25

    # ---    BAO ---
    self.D_V = ((1 + self.z)**2 * self.D_A**2 * C.c / self.H)**(1/3)

    # ---    ---
    Omega_m_a = self.Omega_m0 / self.a**3 / (self.H / self.H0_nat)**2
    gamma_growth = 0.55 + 0.05 * (self.beta / 2.1e-3)**2
    self.f_growth = Omega_m_a**gamma_growth

    # --- 8(z)  ---
    sigma8_0 = 0.811
    self.sigma8_z = sigma8_0 * (self.a / 1.0)**0.55
    self.fsigma8 = self.f_growth * self.sigma8_z

# ----------------------------------------------------------------------
# .:
# ----------------------------------------------------------------------

def H_at_z(self, z: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
    """H(z)   km/s/Mpc"""
    interp = interp1d(self.z[::-1], self.H_km_s_Mpc[::-1],
                      kind='cubic', bounds_error=False, fill_value='extrapolate')
    return interp(z)

def DA_at_z(self, z: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
    """D_A(z)   Mpc"""
    interp = interp1d(self.z[::-1], self.D_A[::-1],
                      kind='cubic', bounds_error=False, fill_value='extrapolate')
    return interp(z)

def DV_at_z(self, z: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
    """D_V(z)   Mpc"""
    interp = interp1d(self.z[::-1], self.D_V[::-1],
                      kind='cubic', bounds_error=False, fill_value='extrapolate')
    return interp(z)
```

```
def mu_at_z(self, z: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
    """(z)  """
    interp = interp1d(self.z[::-1], self.mu[::-1],
                      kind='cubic', bounds_error=False, fill_value='extrapolate')
    return interp(z)

def w_at_z(self, z: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
    """w(z)  """
    interp = interp1d(self.z[::-1], self.w_de[::-1],
                      kind='cubic', bounds_error=False, fill_value='extrapolate')
    return interp(z)

def fsigma8_at_z(self, z: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
    """f8(z)  """
    interp = interp1d(self.z[::-1], self.fsigma8[::-1],
                      kind='cubic', bounds_error=False, fill_value='extrapolate')
    return interp(z)

# ------------------------------------------------------------------
# .:
# ------------------------------------------------------------------

@property
def S8(self) -> float:
    """S8 = 8 (_m/0.3) """
    sigma8_0 = 0.811
    Omega_m_today = self.Omega_m[-1]
    return sigma8_0 * np.sqrt(Omega_m_today / 0.3) * (1 - 0.03 * (self.beta/2.1e-3)**2)

@property
def w0(self) -> float:
    """w(z=0) """
    return self.w_de[-1]

@property
def H0_predicted(self) -> float:
    """H  (km/s/Mpc)"""
    return self.H_km_s_Mpc[-1]

def cmb_shift(self, ell: int = 2000) -> float:
    """    CMB ()"""
```

```
    return 5.0 * (ell / 2000) * (self.beta / 2.1e-3) * 0.01

def qpo_frequency(self, M_Msun: float = 1e9) -> float:
    """ QPO  (Hz)"""
    return 1e-4 * (M_Msun / 1e9)**(-1/3) * (self.beta / 2.1e-3)**0.5

def core_radius(self, M_halo_Msun: float = 1e11) -> float:
    """     (kpc)"""
    return 10.0 * (M_halo_Msun / 1e11)**(1/3) * (self.rho_c_ratio / 1.2e-3)**(-0.5)

# ---------------------------------------------------------------------
# .:
# ---------------------------------------------------------------------

def plot_cosmology(self, save: bool = False, filename: str = 'hlu_cosmology.png'):
    """  """
    fig, axes = plt.subplots(2, 3, figsize=(16, 10))
    fig.suptitle(' HLU  -  -  ',
                 fontsize=16, fontweight='bold')

    # w(z)
    ax = axes[0, 0]
    ax.plot(self.z, self.w_de, 'b-', lw=2.5, label='HLU')
    ax.axhline(-1, color='gray', ls='--', lw=1.5, alpha=0.7, label='CDM')
    ax.set_xlabel('z', fontsize=12)
    ax.set_ylabel('w(z)', fontsize=12)
    ax.set_title('   ', fontsize=13)
    ax.invert_xaxis()
    ax.legend(loc='lower right')
    ax.grid(True, alpha=0.3)
    ax.set_xlim(3, 0)
    ax.set_ylim(-1.1, 0.1)

    # H(z)
    ax = axes[0, 1]
    ax.plot(self.z, self.H_km_s_Mpc, 'r-', lw=2.5, label='HLU')
    H_LCDM = self.H0_phys * np.sqrt(self.Omega_m0*(1+self.z)**3 + (1-self.Omega_m0))
    ax.plot(self.z, H_LCDM, 'k--', lw=1.5, alpha=0.7, label='CDM')
    ax.set_xlabel('z', fontsize=12)
    ax.set_ylabel('H(z) [km/s/Mpc]', fontsize=12)
    ax.set_title(' ', fontsize=13)
    ax.invert_xaxis()
```

```
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)
ax.set_xlim(3, 0)

# (a)
ax = axes[0, 2]
ax.plot(self.a, self.phi, 'g-', lw=2.5)
ax.set_xlabel('a', fontsize=12)
ax.set_ylabel('', fontsize=12)
ax.set_title('  ', fontsize=13)
ax.grid(True, alpha=0.3)
ax.set_xscale('log')

# (a)
ax = axes[1, 0]
ax.semilogx(self.a, self.Omega_m, 'b-', lw=2.5, label='_m')
ax.semilogx(self.a, self.Omega_phi, 'r-', lw=2.5, label='_')
ax.semilogx(self.a, self.Omega_r0/self.a**4 / self.rho_total, 'g-', lw=1.5, label='_r')
ax.set_xlabel('a', fontsize=12)
ax.set_ylabel('', fontsize=12)
ax.set_title(' ', fontsize=13)
ax.legend(loc='upper right')
ax.grid(True, alpha=0.3)
ax.set_ylim(0, 1)

#
ax = axes[1, 1]
meff_norm = self.meff / self.meff[0]
ax.semilogx(self.a, meff_norm, 'purple', lw=2.5)
ax.set_xlabel('a', fontsize=12)
ax.set_ylabel('m_eff / m_eff(a)', fontsize=12)
ax.set_title('   ', fontsize=13)
ax.grid(True, alpha=0.3)
ax.set_yscale('log')

# w_total(a)
ax = axes[1, 2]
ax.semilogx(self.a, self.w_total, 'orange', lw=2.5)
ax.axhline(-1, color='gray', ls='--', lw=1.5, alpha=0.7)
ax.axhline(0, color='gray', ls='--', lw=1.5, alpha=0.7)
ax.axhline(1/3, color='gray', ls='--', lw=1.5, alpha=0.7)
ax.set_xlabel('a', fontsize=12)
```

```
    ax.set_ylabel('w_total', fontsize=12)
    ax.set_title('  ', fontsize=13)
    ax.grid(True, alpha=0.3)
    ax.set_ylim(-1.2, 0.5)

    plt.tight_layout()

    if save:
        plt.savefig(filename, dpi=300, bbox_inches='tight')
        print(f"  : {filename}")

    plt.show()

def plot_observables(self, save: bool = False, filename: str = 'hlu_observables.png'):
    """  """
    fig, axes = plt.subplots(2, 2, figsize=(14, 10))
    fig.suptitle('   HLU', fontsize=16, fontweight='bold')

    z_plot = np.linspace(0, 2, 100)

    # D_A(z)
    ax = axes[0, 0]
    DA = self.DA_at_z(z_plot)
    ax.plot(z_plot, DA, 'b-', lw=2.5, label='HLU')
    ax.set_xlabel('z', fontsize=12)
    ax.set_ylabel('D_A(z) [Mpc]', fontsize=12)
    ax.set_title('  ', fontsize=13)
    ax.grid(True, alpha=0.3)

    # D_V(z)
    ax = axes[0, 1]
    DV = self.DV_at_z(z_plot)
    ax.plot(z_plot, DV, 'r-', lw=2.5, label='HLU')
    ax.set_xlabel('z', fontsize=12)
    ax.set_ylabel('D_V(z) [Mpc]', fontsize=12)
    ax.set_title('  (BAO)', fontsize=13)
    ax.grid(True, alpha=0.3)

    # (z)
    ax = axes[1, 0]
    mu = self.mu_at_z(z_plot)
    ax.plot(z_plot, mu, 'g-', lw=2.5, label='HLU')
```

```
        ax.set_xlabel('z', fontsize=12)
        ax.set_ylabel('(z)', fontsize=12)
        ax.set_title('  ( Ia)', fontsize=13)
        ax.grid(True, alpha=0.3)

        # f8(z)
        ax = axes[1, 1]
        fs8 = self.fsigma8_at_z(z_plot)
        ax.plot(z_plot, fs8, 'purple', lw=2.5,
                label=f'HLU (S={self.S8:.3f})')
        ax.set_xlabel('z', fontsize=12)
        ax.set_ylabel('f(z)', fontsize=12)
        ax.set_title('  ', fontsize=13)
        ax.grid(True, alpha=0.3)
        ax.set_ylim(0, 0.6)
        ax.legend(loc='lower right')

        plt.tight_layout()

        if save:
            plt.savefig(filename, dpi=300, bbox_inches='tight')
            print(f"  : {filename}")

        plt.show()

    def _print_summary(self):
        """  """
        print("\n" + "="*70)
        print(" HLU  -  -  ")
        print("="*70)
        print(f" :")
        print(f"   ()          = {self.beta:.2e}")
        print(f"  m ( bare)        = {self.m0_eV:.2e} eV")
        print(f"  _c/_crit           = {self.rho_c_ratio:.2e}")
        print(f"  n ( )    = {self.n:.2f}")
        print(f"   ln a ( )     = {self.Delta_ln_a:.4f}")
        print(f"           = {self.N}")
        print()
        print(f" :")
        print(f"  a_final              = {self.a[-1]:.6f}")
        print(f"  z_final              = {self.z[-1]:.2f}")
        print(f"  w_de(z=0)            = {self.w_de[-1]:.4f}")
```

```
        print(f"  H ()        = {self.H_km_s_Mpc[-1]:.2f} km/s/Mpc")
        print(f"  _m(z=0)             = {self.Omega_m[-1]:.4f}")
        print(f"  _(z=0)              = {self.Omega_phi[-1]:.4f}")
        print()
        print(f" :")
        print(f"  S                   = {self.S8:.4f}")
        print(f"  w                   = {self.w0:.4f}")
        print(f"  CMB shift @ =2000   = {self.cmb_shift(2000)*100:.2f}%")
        print(f"  QPO (M=10 M)        = {self.qpo_frequency(1e9):.2e} Hz")
        print(f"  r_core (M=10¹¹ M)   = {self.core_radius(1e11):.1f} kpc")
        print("="*70)


#
================================================================================
=
#  :   (Quantum Extension)
#
================================================================================
=

class HLU_QuantumExtension:
    """
       HLU  -.

    :
    1.    T_op
    2.
    3.  trapping   (QCMTS)
    4.
    """

    def __init__(self, hlu_model: HLU_PixelTime_Cosmology):
        self.hlu = hlu_model

        #
        self.M_PI = 1.0  #
        self.gamma_0 = self.hlu.beta**2  #

    def collapse_rate(self, rho_eff: float) -> float:
        """
            (s¹)
```

```
     =  + ² m_eff() / M_PI
    """
    m_eff_natural = self.hlu.potential.effective_mass(
        self.hlu.phi[-1], rho_eff
    )
    gamma_natural = self.gamma_0 + self.hlu.beta**2 * m_eff_natural / self.M_PI
    return gamma_natural * C.sec_per_natural

def decoherence_time(self, rho_env: float) -> float:
    """   ()"""
    return 1.0 / self.collapse_rate(rho_env)

def double_slit_visibility(self,
                           L: float,
                           v: float,
                           rho_env: float) -> float:
    """
       -

    V = V exp(- L / v)
    """
    gamma = self.collapse_rate(rho_env)
    return np.exp(-gamma * L / v)

def qcmts_radius(self, M: float, v_c: float) -> float:
    """
      trapping   (QCMTS)

    _QCMTS ~ GM/v_c²
    """
    return C.G * M / v_c**2

def entanglement_decay_rate(self, rho_env: float) -> float:
    """     """
    return self.collapse_rate(rho_env)

def neutron_star_decoherence(self, rho_ns: float = 1e18) -> float:
    """      """
    return self.decoherence_time(rho_ns)

def pulsar_timing_residual(self, distance_kpc: float = 1.0) -> float:
```

```
        """   ()"""
        rho_ism = 1e-21  # kg/m^3
        m_eff_natural = self.hlu.potential.effective_mass(
            self.hlu.phi[-1], rho_ism
        )
        r_screen = 1.0 / m_eff_natural * C.Mpc_to_m
        residual = (self.hlu.beta**2 * distance_kpc * C.kpc_to_m /
                    r_screen * 1e-9)
        return residual


#
================================================================================
=
#  :
#
================================================================================
=

def run_validation():
    """   """

    print("\n" + "="*70)
    print("   HLU  -")
    print("="*70)

    #  :
    print("\n1.     :")
    results = []
    for Delta in [0.002, 0.0015, 0.001]:
        model = HLU_PixelTime_Cosmology(
            Delta_ln_a=Delta,
            n_pixels=int(7/Delta),
            verbose=False
        )
        results.append((Delta, model.w0, model.H0_predicted))
        print(f"   ln a = {Delta:.4f}: w = {model.w0:.4f}, H = {model.H0_predicted:.2f}")

    #  :    0 ( CDM)
    print("\n2.     0 ( CDM):")
    model_gr = HLU_PixelTime_Cosmology(
        beta=1e-6,
```

```
        Delta_ln_a=0.0015,
        n_pixels=10000,
        verbose=False
    )
    print(f"    = 1e-6: w = {model_gr.w0:.4f} (  -1)")
    print(f"   _m(z=0) = {model_gr.Omega_m[-1]:.4f} (  0.315)")

    #  :
    print("\n3.   :")
    model_lock = HLU_PixelTime_Cosmology(verbose=False)
    print(f"   _c/_crit = {model_lock.rho_c_ratio:.2e}")
    print(f"   z_transition  {model_lock.z[np.argmin(np.abs(model_lock.w_de + 0.5))]:.2f}")

    print("\n      .")
    return True


#
================================================================================
=
#  :
#
================================================================================
=

if __name__ == "__main__":

    print("\n" + "="*70)
    print(" Horizon-Locked Unification (HLU)  - ")
    print(" . -     DESI 2024  Planck PR4")
    print(": ZARKAM ( )")
    print("="*70)

    #
    run_validation()

    #
    print("\n" + "="*70)
    print("    ")
    print("="*70)

    model = HLU_PixelTime_Cosmology(
```

```
    #    DESI+Planck
    beta=2.1e-3,
    m0_eV=1.2e-32,
    rho_c_ratio=1.2e-3,
    n=2.0,

    #
    V0=0.7,
    lambda_V=1.0,
    sigma_phi=0.1,

    #
    Delta_ln_a=0.0015,
    n_pixels=15000,
    adaptive_step=True,

    verbose=True
)

#
model.plot_cosmology(save=True)
model.plot_observables(save=True)

#
quantum = HLU_QuantumExtension(model)

print("\n" + "="*70)
print("  -  ")
print("="*70)
print(f"  ( ): {quantum.collapse_rate(1e-7):.2e} s¹")
print(f"    ( ): {quantum.decoherence_time(1e18):.2e} s")
print(f" - (L=1m, v=100m/s): {quantum.double_slit_visibility(1, 100, 1e-7):.4f}")
print(f"  : {quantum.pulsar_timing_residual(1.0):.2e} s")
print("="*70)

print("\n     .")
```