# V1_6_mass_geodesic

July 28, 2025

## -1 Intro

## 0 Table of Contents

## 1 Project Initialization

```python
# In V1_5_mass_geodesic.ipynb, Cell ID 33f07e1c (UPDATED)
import os
import shutil
import sympy as sp
import nrpy.c_function as cfc
import nrpy.c_codegen as ccg
import nrpy.params as par
import nrpy.indexedexp as ixp
import nrpy.infrastructures.BHaH.BHaH_defines_h as Bdefines_h
import nrpy.infrastructures.BHaH.Makefile_helpers as Makefile
from nrpy.infrastructures.BHaH import cmdline_input_and_parfiles
import nrpy.helpers.generic as gh
import nrpy.infrastructures.BHaH.CodeParameters as CPs

project_name = "mass_integrator"
project_dir = os.path.join("project", project_name)
shutil.rmtree(project_dir, ignore_errors=True)

par.set_parval_from_str("Infrastructure", "BHaH")

# --- Physical Parameters ---
M_scale = par.CodeParameter("REAL", __name__, "M_scale", 1.0, commondata=True, add_to_parfile=True, add_to_set_CodeParameters_h=True)
a_spin = par.CodeParameter("REAL", __name__, "a_spin", 0.9, commondata=True, add_to_parfile=True, add_to_set_CodeParameters_h=True)

metric_choice = par.CodeParameter(
    "int", __name__, "metric_choice", 0,
    commondata=True, add_to_parfile=True
```

```python
)

# --- Integration & Termination Parameters ---
t_max_integration = par.CodeParameter("REAL", __name__, "t_max_integration", 2000.0, commondata=True, add_to_parfile=True)
flatness_threshold = par.CodeParameter("REAL", __name__, "flatness_threshold", 1e-2, commondata=True, add_to_parfile=True)
r_escape = par.CodeParameter("REAL", __name__, "r_escape", 1500.0, commondata=True, add_to_parfile=True)
ut_max = par.CodeParameter("REAL", __name__, "ut_max", 1e3, commondata=True, add_to_parfile=True)

# --- Debugging & Validation Parameters ---
perform_conservation_check = par.CodeParameter("bool", __name__, "perform_conservation_check", True, commondata=True,
 ↪add_to_parfile=True)
run_in_debug_mode = par.CodeParameter("bool", __name__, "run_in_debug_mode", True, commondata=True, add_to_parfile=True)

# --- Disk Parameters ---
disk_lambda_rest_at_r_min = par.CodeParameter("REAL", __name__, "disk_lambda_rest_at_r_min", 656.3, commondata=True,
 ↪add_to_parfile=True)
disk_num_r= par.CodeParameter("int", __name__, "disk_num_r", 100, commondata=True, add_to_parfile=True)
disk_num_phi= par.CodeParameter("int", __name__, "disk_num_phi", 200, commondata=True, add_to_parfile=True)
disk_r_min = par.CodeParameter("REAL", __name__, "disk_r_min", 6.0, commondata=True, add_to_parfile=True)
disk_r_max = par.CodeParameter("REAL", __name__, "disk_r_max", 25.0, commondata=True, add_to_parfile=True)
snapshot_every_t = par.CodeParameter("REAL", __name__, "snapshot_every_t", 10.0, commondata=True, add_to_parfile=True)
t_final = par.CodeParameter("REAL", __name__, "t_final", 2000.0, commondata=True, add_to_parfile=True)
output_folder = par.CodeParameter("char[100]", __name__, "output_folder", "output", commondata=True, add_to_parfile=True)

# --- : Barred Flocculent Spiral Galaxy Shape Parameters ---
print("-> Registering CodeParameters for barred flocculent spiral galaxy geometry...")
bar_length = par.CodeParameter("REAL", __name__, "bar_length", 5.0, commondata=True, add_to_parfile=True)
bar_aspect_ratio = par.CodeParameter("REAL", __name__, "bar_aspect_ratio", 0.25, commondata=True, add_to_parfile=True) # width / length
bulge_radius = par.CodeParameter("REAL", __name__, "bulge_radius", 1.5, commondata=True, add_to_parfile=True)
arm_particle_density = par.CodeParameter("REAL", __name__, "arm_particle_density", 0.3, commondata=True, add_to_parfile=True) # Base
 ↪probability of a particle being in an arm
arm_clumpiness_factor = par.CodeParameter("REAL", __name__, "arm_clumpiness_factor", 8.0, commondata=True, add_to_parfile=True) # How
 ↪many clumps per arm rotation
arm_clump_size = par.CodeParameter("REAL", __name__, "arm_clump_size", 0.5, commondata=True, add_to_parfile=True) # How tight the
 ↪clumps are
bar_density_factor = par.CodeParameter("REAL", __name__, "bar_density_factor", 2.0, commondata=True, add_to_parfile=True) # Bar is 2x
 ↪denser than arms
bulge_density_factor = par.CodeParameter("REAL", __name__, "bulge_density_factor", 3.0, commondata=True, add_to_parfile=True) # Bulge
 ↪is 3x denser

# --- : Spiral Galaxy Shape Parameters ---
print("-> Registering CodeParameters for spiral galaxy geometry...")
spiral_galaxy_num_arms = par.CodeParameter(
    "int", __name__, "spiral_galaxy_num_arms", 2,
```

```
        commondata=True, add_to_parfile=True
)
spiral_galaxy_arm_tightness = par.CodeParameter(
    "REAL", __name__, "spiral_galaxy_arm_tightness", 0.2,
    commondata=True, add_to_parfile=True
)

# --- : Initial Conditions Type Selector ---
print("-> Registering CodeParameter for selecting initial conditions type...")
initial_conditions_type = par.CodeParameter("char[100]", __name__, "initial_conditions_type", "KeplerianDisk",commondata=True,␣
 ↪add_to_parfile=True)
```

```
-> Registering CodeParameters for barred flocculent spiral galaxy geometry...
-> Registering CodeParameters for spiral galaxy geometry...
-> Registering CodeParameter for selecting initial conditions type...
```

# Step 3: The Symbolic Core - Foundational Math

This section defines the pure mathematical logic of our problem using Python's `sympy` library. Each function in this section is a "blueprint" for a physical calculation. These functions take symbolic `sympy` objects as input and return new symbolic expressions as output. They have no knowledge of C code; they are concerned only with mathematics and will be called later to generate the "recipes" for our C code engines.

### 3.a: Metric Tensor Derivatives

The first step in calculating the Christoffel symbols is to compute the partial derivatives of the metric tensor, $g_{\mu\nu}$. This function, `derivative_g4DD`, takes the symbolic 4x4 metric tensor `g4DD` and a list of the four coordinate symbols `xx` as input.

The function iterates through all components to symbolically calculate the partial derivative of each metric component with respect to each coordinate. The resulting quantity, which we can denote using comma notation as $g_{\mu\nu,\alpha}$, is defined as:

$$g_{\mu\nu,\alpha} \equiv \frac{\partial g_{\mu\nu}}{\partial x^\alpha}$$

The nested `for` loops in the code directly correspond to the spacetime indices `μ`, `ν`, `α` in the physics equation. `sympy`'s built-in `sp.diff()` function is used to perform the symbolic differentiation, and the final result is returned as a rank-3 symbolic tensor.

**1.0.1  nrpy Functions Used in this Cell:**

- `nrpy.indexedexp.zerorank3(dimension):`
  - **Source File**: `nrpy/indexedexp.py`
  - **Description**: This function creates a symbolic rank-3 tensor (a Python list of lists of lists) of a specified dimension, with all elements initialized to the `sympy` integer 0. It is used here to create a container for the derivative results.

```
[2]: def derivative_g4DD(g4DD, xx):
         """Computes the symbolic first derivatives of the metric tensor."""
         g4DD_dD = ixp.zerorank3(dimension=4)
         for nu in range(4):
             for mu in range(4):
```

```
            for alpha in range(4):
                g4DD_dD[nu][mu][alpha] = sp.diff(g4DD[nu][mu], xx[alpha])
    return g4DD_dD
```

### 3.b: Christoffel Symbol Calculation

This function implements the core formula for the Christoffel symbols of the second kind, $\Gamma^{\delta}_{\mu\nu}$. It takes the symbolic metric tensor `g4DD` ($g_{\mu\nu}$) and its derivatives `g4DD_dD` ($g_{\mu\nu,\alpha}$) as input. The calculation requires the inverse metric, $g^{\mu\nu}$, which is computed using another `nrpy` helper function.

The function then applies the well-known formula for the Christoffel symbols. Using the comma notation for partial derivatives, the formula is:

$$\Gamma^{\delta}_{\mu\nu} = \frac{1}{2} g^{\delta\alpha} \left( g_{\nu\alpha,\mu} + g_{\mu\alpha,\nu} - g_{\mu\nu,\alpha} \right)$$

The Python `for` loops iterate over the spacetime indices $\delta$, $\mu$, $\nu$, $\alpha$ to construct each component of the Christoffel symbol tensor. After the summation is complete, the `sp.trigsimp()` function is used to simplify the resulting expression. This trigonometric simplification is highly effective and much faster than a general `sp.simplify()` for the Kerr-Schild metric, which contains trigonometric functions of the coordinates.

**1.0.2 nrpy Functions Used in this Cell:**

- `nrpy.indexedexp.zerorank3(dimension)`: Previously introduced. Used to initialize the Christoffel symbol tensor.
- `nrpy.indexedexp.symm_matrix_inverter4x4(g4DD)`:
    - **Source File**: `nrpy/indexedexp.py`
    - **Description**: This function takes a symbolic 4x4 symmetric matrix and analytically computes its inverse. It is highly optimized for this specific task, returning both the inverse matrix ($g^{\mu\nu}$) and its determinant.

```
[3]: def four_connections(g4DD, g4DD_dD):
         """
         Computes and simplifies Christoffel symbols from the metric and its derivatives.

         This version uses sp.trigsimp() which is highly effective and much faster
         than sp.simplify() for the Kerr-Schild metric.
         """
         Gamma4UDD = ixp.zerorank3(dimension=4)
         g4UU, _ = ixp.symm_matrix_inverter4x4(g4DD)

         for mu in range(4):
             for nu in range(4):
                 for delta in range(4):
                     # Calculate the Christoffel symbol component using the standard formula
                     for alpha in range(4):
                         Gamma4UDD[delta][mu][nu] += sp.Rational(1, 2) * g4UU[delta][alpha] * \
                             (g4DD_dD[nu][alpha][mu] + g4DD_dD[mu][alpha][nu] - g4DD_dD[mu][nu][alpha])

                     # Use sp.trigsimp() to simplify the resulting expression.
                     # This is the key to speeding up the symbolic calculation.
```

```
            Gamma4UDD[delta][mu][nu] = sp.trigsimp(Gamma4UDD[delta][mu][nu])

    return Gamma4UDD
```

### 3.c: Geodesic Momentum RHS

This function defines the symbolic right-hand side (RHS) for the evolution of the **reverse-time momentum**, $p^\alpha$. As established in the introduction, this is the second of our three first-order ODEs:

$$\frac{dp^\alpha}{d\kappa} = -\Gamma^\alpha_{\mu\nu} p^\mu p^\nu$$

The function `geodesic_mom_rhs` takes the symbolic Christoffel symbols $\Gamma^\alpha_{\mu\nu}$ as its input. It then defines the symbolic momentum vector `pU` using `sympy`'s `sp.symbols()` function. A key `nrpy` technique is used here: the symbols are created with names that are already valid C array syntax (e.g., `"y[4]"`). This "direct naming" simplifies the final C code generation by eliminating the need for string substitutions.

The core of this function constructs the symbolic expression for the RHS by performing the Einstein summation $-\Gamma^\alpha_{\mu\nu} p^\mu p^\nu$. A direct implementation would involve a double loop over both $\mu$ and $\nu$ from 0 to 3, resulting in $4 \times 4 = 16$ terms for each component of $\alpha$, which is computationally inefficient.

However, we can significantly optimize this calculation by exploiting symmetry. The term $p^\mu p^\nu$ is symmetric with respect to the interchange of the indices $\mu$ and $\nu$. The Christoffel symbols $\Gamma^\alpha_{\mu\nu}$ are also symmetric in their lower two indices. Therefore, the full sum can be split into diagonal ($\mu = \nu$) and off-diagonal ($\mu \neq \nu$) terms:

$$\sum_{\mu,\nu} \Gamma^\alpha_{\mu\nu} p^\mu p^\nu = \sum_{\mu=0}^{3} \Gamma^\alpha_{\mu\mu} (p^\mu)^2 + \sum_{\mu \neq \nu} \Gamma^\alpha_{\mu\nu} p^\mu p^\nu$$

The second sum over $\mu \neq \nu$ contains pairs of identical terms (e.g., the $\mu = 1, \nu = 2$ term is the same as the $\mu = 2, \nu = 1$ term). We can combine all such pairs by summing over only one of the cases (e.g., $\mu < \nu$) and multiplying by two:

$$\sum_{\mu,\nu} \Gamma^\alpha_{\mu\nu} p^\mu p^\nu = \sum_{\mu=0}^{3} \Gamma^\alpha_{\mu\mu} (p^\mu)^2 + 2 \sum_{\mu < \nu} \Gamma^\alpha_{\mu\nu} p^\mu p^\nu$$

The Python code implements this optimized version, ensuring that each component of the RHS is computed with the minimum number of floating point operations, leading to more efficient C code.

**1.0.3   nrpy Functions Used in this Cell:**

- `nrpy.indexedexp.zerorank1(dimension)`:
    - **Source File**: `nrpy/indexedexp.py`
    - **Description**: Creates a symbolic rank-1 tensor (a Python list) of a specified dimension, with all elements initialized to the `sympy` integer 0. It is used here to create a container for the four components of the momentum RHS.

```python
[4]: def geodesic_vel_rhs_massive():
        """
        Symbolic RHS for massive particle velocity ODE: du^a/dτ = -Γ^a_μν u^μ u^ν.
        u is the 4-velocity, y[4]...y[7].
        """
        Gamma4UDD = ixp.declarerank3("conn->Gamma4UDD",dimension= 4,sym="sym12")
        ut,ux,uy,uz = sp.symbols("y[4] y[5] y[6] y[7]", Real=True)
        uU = [ut,ux,uy,uz]
```

```
        geodesic_rhs = ixp.zerorank1(dimension=4)
        for alpha in range(4):
            for mu in range(4):
                geodesic_rhs[alpha] += Gamma4UDD[alpha][mu][mu] * uU[mu] * uU[mu]
                for nu in range(mu + 1, 4):
                    geodesic_rhs[alpha] += 2 * Gamma4UDD[alpha][mu][nu] * uU[mu] * uU[nu]
            geodesic_rhs[alpha] = -geodesic_rhs[alpha]
        return geodesic_rhs
```

### 3.d: Geodesic Position RHS

This function defines the symbolic right-hand side (RHS) for the evolution of the position coordinates, $x^\alpha$. As derived in the introduction, this is the first of our three first-order ODEs:

$$\frac{dx^\alpha}{d\kappa} = p^\alpha$$

The Python function `geodesic_pos_rhs` is straightforward. It defines the components of the reverse-time momentum vector, `pU`, using `sympy`'s `sp.symbols()` function with the "direct naming" convention (`y[4]`, `y[5]`, etc.). It then simply returns a list containing these momentum components. This list of four symbolic expressions will serve as the first four components of the complete 9-component RHS vector that our C code will solve.

```
[5]: def geodesic_pos_rhs_massive():
        """
        Symbolic RHS for position ODE: dx^a/dτ = u^a.
        u is the 4-velocity, y[4]...y[7].
        """
        ut,ux,uy,uz = sp.symbols("y[4] y[5] y[6] y[7]", Real=True)
        uU = [ut,ux,uy,uz]
        return uU
```

### 3.f: Symbolic Calculation of u^0

```
[6]: def ut_massive():
        """
        Symbolically derives u^t for a MASSIVE particle from the 4-velocity.
        The derivation comes from solving the timelike normalization condition g_μν u^μ u^ν = -1.
        """
        # The symbolic recipe will use the standard variable names u0, u1, etc.
        # The C-generating function will map y[4], y[5], etc. to these.
        u0,u1,u2,u3 = sp.symbols("u0 u1 u2 u3", Real=True)
        uU=[u0,u1,u2,u3]

        # The recipe uses the standard name "metric" for the struct.
        g4DD = ixp.declarerank2("metric->g", sym="sym01", dimension=4)

        # This is the quadratic equation for u^0, derived from g_μν u^μ u^ν = -1
```

```
# g_00(u^0)^2 + 2g_0i u^0 u^i + g_ij u^i u^j = -1
# We solve for u^0.
sum_g0i_ui = sp.sympify(0)
for i in range(1,4):
    sum_g0i_ui += g4DD[0][i]*uU[i]

sum_gij_ui_uj = sp.sympify(0)
for i in range(1,4):
    sum_gij_ui_uj += g4DD[i][i]*uU[i]*uU[i]
    for j in range(i+1,4):
        sum_gij_ui_uj += 2*g4DD[i][j]*uU[i]*uU[j]

# The discriminant of the quadratic formula for u^0
# CORRECTED: This now includes the "+1" term from g_μν u^μ u^ν = -1
discriminant = sum_g0i_ui**2 - g4DD[0][0]*(sum_gij_ui_uj + 1)

# We choose the positive root for a forward-in-time particle outside the horizon.
# Note: Your choice of the minus sign was correct for the final expression.
answer = (-sum_g0i_ui - sp.sqrt(discriminant)) / g4DD[0][0]
return answer
```

## 2 Symbolic g_tt,g_tphi,g_phiphi values for numerically stable u$^{t}$,u$^{x}$,u^y on xy plane.

```
[7]: # In V1_5_mass_geodesic.ipynb, add as a NEW CELL

def symbolic_kerr_metric_equatorial_boyer_lindquist():
    """
    Provides the symbolic components of the Kerr metric in Boyer-Lindquist
    coordinates, specialized to the equatorial plane (theta = pi/2).

    This is a necessary helper for the numerically stable initial condition calculation.
    """
    # Define symbolic variables for Boyer-Lindquist coordinates and parameters
    r, M, a = sp.symbols("r M_scale a_spin", real=True)

    # In the equatorial plane, sin(theta) = 1 and cos(theta) = 0.
    # The Kerr metric components simplify considerably.

    # g_tt = -(1 - 2M/r)
    g_tt = -(1 - 2*M/r)

    # g_tphi = -2*a*M/r
    g_tphi = -2*a*M/r
```

```
    # g_phiphi = (r^2 + a^2 + 2*M*a^2/r)
    g_phiphi = r**2 + a**2 + (2*M*a**2)/r

    return g_tt, g_tphi, g_phiphi
```

# 3   Symbolic ut,uphi

```
[8]:  # In V1_5_mass_geodesic.ipynb, replace cells [ceff5579] and [cbc7fdf2]
      # with this single cell.

      def symbolic_ut_uphi_from_r_stable():
          """
          Symbolically derives u^t and u^phi for a circular, equatorial orbit
          using a NUMERICALLY STABLE three-step method.

          This definitive version constructs all expressions using the final C-level
          parameter names directly, avoiding the need for .subs() and adhering to
          the nrpy design pattern.

          Returns a list containing the symbolic expressions for [u^t, u^phi].
          """
          # --- Step 0: Define symbolic variables with final C names ---
          # These symbols directly correspond to the variables that will be
          # available in the C function that uses this recipe.
          r, M, a = sp.symbols("r_initial M_scale a_spin", real=True)

          # --- Step 1: Calculate the stable angular velocity Omega = d(phi)/dt ---
          Omega = sp.sqrt(M) / (r**sp.Rational(3, 2) + a * sp.sqrt(M))

          # --- Step 2: Define the required metric components using the same symbols ---
          # These are the Kerr metric components in Boyer-Lindquist coordinates,
          # specialized to the equatorial plane (theta = pi/2).
          g_tt = -(1 - 2*M/r)
          g_tphi = -2*a*M/r
          g_phiphi = r**2 + a**2 + (2*M*a**2)/r

          # --- Step 3: Solve for u^t using the 4-velocity normalization condition ---
          # (u^t)^2 = -1 / (g_tt + 2*g_tphi*Omega + g_phiphi*Omega^2)
          ut_squared_inv_denom = g_tt + 2*g_tphi*Omega + g_phiphi*Omega**2
          ut_squared = -1 / ut_squared_inv_denom
          ut = sp.sqrt(ut_squared)
```

```
    # --- Step 4: Calculate u^phi ---
    # u^phi = Omega * u^t
    uphi = Omega * ut

    return [ut, uphi]
```

## 4  Markdown for conserved Energy

```
[9]: def symbolic_energy():
        """
        Computes the symbolic expression for conserved energy E = -p_t.
        E = -g_{t,mu} p^mu
        """
        # Define the 4-momentum components using the y[4]...y[7] convention
        pt, px, py, pz = sp.symbols("y[4] y[5] y[6] y[7]", real=True)
        pU = [pt, px, py, pz]

        # Define an abstract metric tensor to be filled by a C struct at runtime
        g4DD = ixp.declarerank2("metric->g", sym="sym01", dimension=4)

        # Calculate p_t = g_{t,mu} p^mu
        p_t = sp.sympify(0)
        for mu in range(4):
            p_t += g4DD[0][mu] * pU[mu]

        return -p_t
```

## 5  Markdown for conserved L

```
[10]: def symbolic_L_components_cart():
        """
        Computes the symbolic expressions for the three components of angular momentum,
        correctly accounting for the symmetry of the metric tensor.
        """
        # Define coordinate and 4-momentum components
        t, x, y, z = sp.symbols("y[0] y[1] y[2] y[3]", real=True)
        pt, px, py, pz = sp.symbols("y[4] y[5] y[6] y[7]", real=True)
        pU = [pt, px, py, pz]

        # Define an abstract metric tensor
        g4DD = ixp.declarerank2("metric->g", sym="sym01", dimension=4)
```

```
        # --- THIS IS THE CORE FIX ---
        # Calculate covariant momentum components p_k = g_{k,mu} p^mu,
        # correctly exploiting the metric's symmetry g_mu,nu = g_nu,mu.
        p_down = ixp.zerorank1(dimension=4)
        for k in range(1, 4): # We only need p_x, p_y, p_z for L_i
            # Sum over mu
            for mu in range(4):
                # Use g4DD[k][mu] if k <= mu, otherwise use g4DD[mu][k]
                if k <= mu:
                    p_down[k] += g4DD[k][mu] * pU[mu]
                else: # k > mu
                    p_down[k] += g4DD[mu][k] * pU[mu]

        p_x, p_y, p_z = p_down[1], p_down[2], p_down[3]


        # Calculate angular momentum components
        L_x = y*p_z - z*p_y
        L_y = z*p_x - x*p_z
        L_z = x*p_y - y*p_x


        return [L_x, L_y, L_z]
```

# 6 Markdown for Carter Constant

```
[11]:  # In file: V10_Python_to_C_via_NRPy.ipynb
       # In the "Symbolic Recipes" cell (Final, Corrected symbolic_carter_constant_Q_final)

       # symbolic_energy() and symbolic_L_components_cart() remain correct.

       def symbolic_carter_constant_Q():
           """
           Computes the symbolic expression for the Carter Constant Q using a
           verified formula, robustly handling the axial singularity.
           """
           # Define all necessary symbolic variables
           t, x, y, z = sp.symbols("y[0] y[1] y[2] y[3]", real=True)
           pt, px, py, pz = sp.symbols("y[4] y[5] y[6] y[7]", real=True)
           pU = [pt, px, py, pz]
           a = sp.Symbol("a_spin", real=True)
           g4DD = ixp.declarerank2("metric->g", sym="sym01", dimension=4)

           # --- Step 1: Compute intermediate quantities E, Lz, and p_i ---
           E = symbolic_energy()
```

```
    _, _, Lz = symbolic_L_components_cart()

    p_down = ixp.zerorank1(dimension=4)
    for k in range(1, 4):
        for mu in range(4):
            if k <= mu: p_down[k] += g4DD[k][mu] * pU[mu]
            else: p_down[k] += g4DD[mu][k] * pU[mu]
    p_x, p_y, p_z = p_down[1], p_down[2], p_down[3]

    # --- Step 2: Compute geometric terms ---
    r_sq = x**2 + y**2 + z**2
    rho_sq = x**2 + y**2

    # --- Step 3: Compute p_theta^2 directly in Cartesian components ---
    # This avoids square roots and potential complex number issues in sympy.
    # p_theta^2 = r^2 * p_z^2 + cot^2(theta) * (x*p_x + y*p_y)^2 - 2*r*p_z*cot(theta)*(x*p_x+y*p_y)
    # where cot(theta) = z / rho

    # This term is (x*p_x + y*p_y)
    xpx_plus_ypy = x*p_x + y*p_y

    # This is p_theta^2, constructed to avoid dividing by rho before squaring.
    # It is equivalent to (z*xpx_plus_ypy/rho - rho*p_z)^2
    p_theta_sq = (z**2 * xpx_plus_ypy**2 / rho_sq) - (2 * z * p_z * xpx_plus_ypy) + (rho_sq * p_z**2)

    # --- Step 4: Assemble the final formula for Q ---
    # Q = p_theta^2 + cos^2(theta) * (-a^2*E^2 + L_z^2/sin^2(theta))
    # where cos^2(theta) = z^2/r^2 and sin^2(theta) = rho^2/r^2

    # This is the second term in the Q formula
    second_term = (z**2 / r_sq) * (-a**2 * E**2 + Lz**2 * (r_sq / rho_sq))

    Q_formula = p_theta_sq + second_term

    # --- Step 5: Handle the axial singularity ---
    # For motion on the z-axis (rho_sq -> 0), Lz=0 and p_theta=0, so Q=0.
    Q_final = sp.Piecewise(
        (0, rho_sq < 1e-12),
        (Q_formula, True)
    )

    return Q_final

print("Final symbolic recipes for conserved quantities defined (Carter Constant re-derived).")
```

Final symbolic recipes for conserved quantities defined (Carter Constant re-derived).

# Step 4: Spacetime Definition in Kerr-Schild Coordinates

This section defines the specific spacetime geometry in which the geodesics will be integrated. Instead of defining separate metrics for Schwarzschild (non-rotating) and Kerr (rotating) black holes, we use a single, powerful coordinate system: **Cartesian Kerr-Schild coordinates**. This system has a major advantage over more common coordinate systems like Boyer-Lindquist: it is regular everywhere, including at the event horizon. This means the metric components and their derivatives do not diverge, allowing the numerical integrator to trace a photon's path seamlessly across the horizon without encountering coordinate singularities.

The Kerr-Schild metric $g_{\mu\nu}$ is constructed by adding a correction term to the flat Minkowski metric $\eta_{\mu\nu}$:

$$g_{\mu\nu} = \eta_{\mu\nu} + 2Hl_\mu l_\nu$$

where $\eta_{\mu\nu}$ is the Minkowski metric `diag(-1, 1, 1, 1)`, $l_\mu$ is a special null vector, and $H$ is a scalar function that depends on the black hole's mass $M$ and spin $a$.

The function `define_kerr_metric_Cartesian_Kerr_Schild()` implements this formula symbolically. It defines the coordinates (`t, x, y, z`), the mass `M`, and the spin `a` as `sympy` symbols. It then constructs the components of the null vector $l_\mu$ and the scalar function $H$. Finally, it assembles the full metric tensor $g_{\mu\nu}$.

A key feature of this formulation is that if the spin parameter `a` is set to zero, the metric automatically and exactly reduces to the Schwarzschild metric in Cartesian coordinates. This allows a single set of symbolic expressions and a single set of C functions to handle both spacetimes, with the specific behavior controlled by the runtime value of the `a_spin` parameter.

### 6.0.1   nrpy Functions Used in this Cell:

- `nrpy.indexedexp.zerorank1(dimension)`: Previously introduced. Used to initialize the null vector $l_\mu$.
- `nrpy.indexedexp.zerorank2(dimension)`: Previously introduced. Used to initialize the metric tensor $g_{\mu\nu}$.

```
[12]: def define_kerr_metric_Cartesian_Kerr_Schild():
          """
          Defines the Kerr metric tensor in Cartesian Kerr-Schild coordinates.

          This function is the new, unified source for both Kerr (a != 0) and
          Schwarzschild (a = 0) spacetimes. The coordinates are (t, x, y, z).

          Returns:
              A tuple (g4DD, xx), where g4DD is the symbolic 4x4 metric tensor
              and xx is the list of symbolic coordinate variables.
          """
          # Define the symbolic coordinates using the 'y[i]' convention for the integrator
          t, x, y, z = sp.symbols("y[0] y[1] y[2] y[3]", real=True)
          xx = [t, x, y, z]

          # Access the symbolic versions of the mass and spin parameters
          M = M_scale.symbol
          a = a_spin.symbol

          # Define intermediate quantities
          r2 = x**2 + y**2 + z**2
```

```python
    r = sp.sqrt(r2)

    # Define the Kerr-Schild null vector l_μ
    l_down = ixp.zerorank1(dimension=4)
    l_down[0] = 1
    l_down[1] = (r*x + a*y) / (r2 + a**2)
    l_down[2] = (r*y - a*x) / (r2 + a**2)
    l_down[3] = z/r

    # Define the scalar function H
    H = (M * r**3) / (r**4 + a**2 * z**2)

    # The Kerr-Schild metric is g_μν = η_μν + 2H * l_μ * l_ν
    # where η_μν is the Minkowski metric diag(-1, 1, 1, 1)
    g4DD = ixp.zerorank2(dimension=4)
    for mu in range(4):
        for nu in range(4):
            eta_mu_nu = 0
            if mu == nu:
                eta_mu_nu = 1
            if mu == 0 and nu == 0:
                eta_mu_nu = -1

            g4DD[mu][nu] = eta_mu_nu + 2 * H * l_down[mu] * l_down[nu]

    return g4DD, xx
```

# 7 Markdown for Schwarzschild

```python
[13]: # In file: V11_0_Python_to_C_via_NRPy.ipynb
      # In the NEW CELL after define_kerr_metric_Cartesian_Kerr_Schild

      def define_schwarzschild_metric_cartesian():
          """
          Defines the Schwarzschild metric tensor directly in Cartesian coordinates.

          This version uses the standard textbook formula and ensures all components
          are sympy objects to prevent C-generation errors.

          Returns:
              A tuple (g4DD, xx), where g4DD is the symbolic 4x4 metric tensor
              and xx is the list of symbolic coordinate variables.
          """
```

```
# Define Cartesian coordinates
t, x, y, z = sp.symbols("y[0] y[1] y[2] y[3]", real=True)
xx = [t, x, y, z]

# Access the symbolic mass parameter
M = M_scale.symbol

# Define r in terms of Cartesian coordinates
r = sp.sqrt(x**2 + y**2 + z**2)

# Define the Cartesian Schwarzschild metric components directly
g4DD = ixp.zerorank2(dimension=4)

# g_tt
g4DD[0][0] = -(1 - 2*M/r)

# Spatial components g_ij = δ_ij + (2M/r) * (x_i * x_j / r^2)
x_i = [x, y, z]
for i in range(3):
    for j in range(3):
        # --- CORRECTED: Use sp.sympify() for the kronecker delta ---
        delta_ij = sp.sympify(0)
        if i == j:
            delta_ij = sp.sympify(1)

        # The indices for g4DD are off by 1 from the spatial indices
        g4DD[i+1][j+1] = delta_ij + (2*M/r) * (x_i[i] * x_i[j] / (r**2))

# --- CORRECTED: Ensure time-space components are sympy objects ---
g4DD[0][1] = g4DD[1][0] = sp.sympify(0)
g4DD[0][2] = g4DD[2][0] = sp.sympify(0)
g4DD[0][3] = g4DD[3][0] = sp.sympify(0)

return g4DD, xx
```

# 8 Symbolic equations for $u^{t}, u$phi on xy plane (Designed to keep sig figs for extreme a_spin and r_initial values)

# Step 5: Symbolic Workflow Execution

This cell acts as the central hub for the symbolic portion of our project. In the preceding cells, we *defined* a series of Python functions that perform individual mathematical tasks. Here, we *execute* those functions in the correct sequence to generate all the final symbolic expressions that will serve as "recipes" for our C code generators.

This "symbolic-first" approach is a core `nrpy` principle and offers significant advantages: 1. **Efficiency**: The complex symbolic calculations, such as inverting the metric tensor and deriving the Christoffel symbols, are performed **only once** when this notebook is run. The results are stored in global Python variables, preventing redundant and time-consuming recalculations. This is especially important for the Kerr metric, whose Christoffel symbols can take several minutes to compute. 2. **Modularity**: This workflow creates a clean separation between the *specific solution* for a metric (e.g., the explicit formulas for the Kerr-Schild Christoffels) and the *generic form* of the equations of motion (which are valid for any metric).

This cell produces two key sets of symbolic expressions that are stored in global variables for later use: * `Gamma4UDD_kerr`: The explicit symbolic formulas for the Christoffel symbols of the unified Kerr-Schild metric. * `all_rhs_expressions`: A Python list containing the 9 symbolic expressions for the right-hand-sides of our generic ODE system. To achieve this generality, we create a symbolic **placeholder** for the Christoffel symbols using `ixp.declarerank3("conn->Gamma4UDD", ...)`. This placeholder is passed to `geodesic_mom_rhs()` to construct the geodesic equation in its abstract form. This elegant technique embeds the final C variable name (`conn->Gamma4UDD...`) directly into the symbolic expression, which dramatically simplifies the C code generation step for the `calculate_ode_rhs()` engine.

### 8.0.1 nrpy Functions Used in this Cell:

- `nrpy.indexedexp.declarerank3(name, dimension)`: Previously introduced. Used here to create a symbolic placeholder for the Christoffel symbols that will be passed to the generic RHS engine.

```
[14]:   # In V1_5_mass_geodesic.ipynb, Cell ID 5fbfe0b5 (UPDATED)


        # --- 1. Define the Kerr-Schild metric and get its derivatives ---
        print(" -> Computing Kerr-Schild metric and Christoffel symbols...")
        g4DD_kerr, xx_kerr = define_kerr_metric_Cartesian_Kerr_Schild()
        g4DD_dD_kerr = derivative_g4DD(g4DD_kerr, xx_kerr)
        Gamma4UDD_kerr = four_connections(g4DD_kerr, g4DD_dD_kerr)
        print("     ... Done.")


        # --- 2. Define the Standard Schwarzschild metric in Cartesian and get its derivatives ---
        print(" -> Computing Standard Schwarzschild (Cartesian) metric and Christoffel symbols...")
        g4DD_schw_cart, xx_schw_cart = define_schwarzschild_metric_cartesian()
        g4DD_dD_schw_cart = derivative_g4DD(g4DD_schw_cart, xx_schw_cart)
        Gamma4UDD_schw_cart = four_connections(g4DD_schw_cart, g4DD_dD_schw_cart)
        print("     ... Done.")


        # --- 3. Generate GENERIC symbolic RHS expressions for MASSIVE geodesics ---
        rhs_pos_massive = geodesic_pos_rhs_massive()
        rhs_vel_massive = geodesic_vel_rhs_massive()
        all_rhs_expressions_massive = rhs_pos_massive + rhs_vel_massive


        # --- 4. Generate symbolic recipes using the STABLE and original methods ---
        print(" -> Generating symbolic recipes for initial conditions and conserved quantities...")
        ut_expr_from_vel = ut_massive() # Keep original for the general C initializer
        # *** THE FIX IS HERE: Call the new, numerically stable function ***
        ut_expr, uphi_expr = symbolic_ut_uphi_from_r_stable()
        print("     ... Initial condition recipes generated.")
```

```
# --- 5. Generate symbolic recipes for conserved quantities ---
E_expr = symbolic_energy()
Lx_expr, Ly_expr, Lz_expr = symbolic_L_components_cart()
Q_expr_kerr = symbolic_carter_constant_Q()
Q_expr_schw = Lx_expr**2 + Ly_expr**2 + Lz_expr**2
list_of_expressions_kerr = [E_expr, Lx_expr, Ly_expr, Lz_expr, Q_expr_kerr]
list_of_expressions_schw = [E_expr, Lx_expr, Ly_expr, Lz_expr, Q_expr_schw]
print("    ... Conservation recipes generated.")
```

```
 -> Computing Kerr-Schild metric and Christoffel symbols...
     ... Done.
 -> Computing Standard Schwarzschild (Cartesian) metric and Christoffel
symbols...
     ... Done.
 -> Generating symbolic recipes for initial conditions and conserved
quantities...
     ... Initial condition recipes generated.
     ... Conservation recipes generated.
```

# Step 6: C Code Generation - Physics "Engines" and "Workers"

This section marks our transition from pure symbolic mathematics to C code generation. The Python functions defined here are "meta-functions": their job is not to perform calculations themselves, but to **generate the C code** that will perform the calculations in the final compiled program.

We distinguish between two types of generated functions: * **Workers**: These are specialized functions that implement the physics for a *specific metric*. For example, `con_kerr_schild()` is a worker that only knows how to compute Christoffel symbols for the Kerr-Schild metric. * **Engines**: These are generic functions that implement physics equations valid for *any metric*. For example, `calculate_ode_rhs()` is an engine that can compute the geodesic equations for any metric, as long as the Christoffel symbols are provided to it.

This design pattern allows for maximum code reuse and extensibility.

## 9 Schwarzschild Metric

```
[15]: def g4DD_schwarzschild_cartesian():
          """
          Generates and registers the C function to compute the Schwarzschild
          metric components in standard Cartesian coordinates.
          """
          print(" -> Generating C worker function: g4DD_schwarzschild_cartesian()...")

          # Use the globally defined g4DD_schw_cart from the symbolic execution step
          list_of_g4DD_syms = []
          for i in range(4):
              for j in range(i, 4):
                  list_of_g4DD_syms.append(g4DD_schw_cart[i][j])
```

```
    list_of_g4DD_C_vars = []
    for i in range(4):
        for j in range(i, 4):
            list_of_g4DD_C_vars.append(f"metric->g{i}{j}")

    includes = ["BHaH_defines.h"]
    desc = r"""@brief Computes the 10 unique components of the Schwarzschild metric in Cartesian coords."""
    name = "g4DD_schwarzschild_cartesian"
    params = "const commondata_struct *restrict commondata, const params_struct *restrict params, const double y[4], metric_struct␣
↪*restrict metric"

    body = ccg.c_codegen(list_of_g4DD_syms, list_of_g4DD_C_vars, enable_cse=True)

    cfc.register_CFunction(
        includes=includes, desc=desc, name=name, params=params, body=body,
        include_CodeParameters_h=True
    )
    print("    ... g4DD_schwarzschild_cartesian() registration complete.")
```

### 6.a: `g4DD_kerr_schild()` Worker

This Python function generates the C **worker** function `g4DD_kerr_schild()`, whose only job is to compute the 10 unique components of the Kerr-Schild metric tensor, $g_{\mu\nu}$, at a given point in spacetime.

The generation process is as follows: 1. **Access Symbolic Recipe:** It accesses the global `g4DD_kerr` variable, which holds the symbolic `sympy` expression for the Kerr-Schild metric tensor, generated in Step 5. 2. **Define C Assignment:** It creates two Python lists: one containing the 10 unique symbolic metric expressions (`list_of_g4DD_syms`) and another containing the corresponding C variable names for the members of the `metric_struct` (e.g., `metric->g00`, `metric->g01`, etc.) in `list_of_g4DD_C_vars`. 3. **Generate C Code:** It passes these two lists to `nrpy.c_codegen.c_codegen`. This powerful `nrpy` function converts the symbolic math into highly optimized C code, including performing Common Subexpression Elimination (CSE). 4. **Register C Function:** Finally, it bundles the generated C code with its metadata (description, parameters, etc.) and registers the complete function with `nrpy.c_function.register_CFunction`. Crucially, it sets `include_CodeParameters_h=True` to automatically handle access to both the `M_scale` and `a_spin` parameters via the "Triple-Lock" system.

#### 9.0.1 `nrpy` Functions Used in this Cell:

- `nrpy.c_codegen.c_codegen(sympy_expressions, C_variable_names, **kwargs)`:
  - **Source File**: `nrpy/c_codegen.py`
  - **Description**: The core symbolic-to-C translation engine. It takes a list of `sympy` expressions and a corresponding list of C variable names and generates optimized C code to perform the assignments.
  - **Key Inputs**:
    * `sympy_expressions`: A Python list of symbolic expressions to be converted to C code.
    * `C_variable_names`: A Python list of strings for the C variables that will store the results.
  - **Key Keyword Arguments (kwargs)**:
    * `enable_cse=True`: Enables Common Subexpression Elimination, which finds repeated mathematical operations, assigns them to temporary variables, and reuses those variables to reduce redundant calculations. This is essential for performance.
- `nrpy.c_function.register_CFunction(name, params, body, **kwargs)`:
  - **Source File**: `nrpy/c_function.py`

- **Description**: This is the workhorse for defining a C function. It takes all necessary metadata and stores it in a global dictionary, `cfc.CFunction_dict`. The final build system uses this dictionary to write all the `.c` source files.
  - **Key Inputs**:
    * `name`: The name of the C function.
    * `params`: A string defining the function's parameters (e.g., `"const double y[4], ..."`).
    * `body`: A string containing the C code for the function's body.
  - **Key Keyword Arguments (kwargs)**:
    * `include_CodeParameters_h=True`: Enables the "Triple-Lock" system for this function, automatically including `set_CodeParameters.h` at the top of the function body.

```python
[16]: def g4DD_kerr_schild():
          """
          Generates and registers the C function to compute the Kerr-Schild
          metric components in Cartesian coordinates. This is the new unified worker.
          """
          print(" -> Generating C worker function: g4DD_kerr_schild()...")

          # We use the globally defined g4DD_kerr from the symbolic execution step
          list_of_g4DD_syms = []
          for i in range(4):
              for j in range(i, 4):
                  list_of_g4DD_syms.append(g4DD_kerr[i][j])

          list_of_g4DD_C_vars = []
          for i in range(4):
              for j in range(i, 4):
                  list_of_g4DD_C_vars.append(f"metric->g{i}{j}")

          includes = ["BHaH_defines.h"]
          desc = r"""@brief Computes the 10 unique components of the Kerr metric in Cartesian Kerr-Schild coords."""
          name = "g4DD_kerr_schild"
          # The state vector y now contains (t, x, y, z)
          params = "const commondata_struct *restrict commondata, const params_struct *restrict params, const double y[4], metric_struct
      *restrict metric"

          body = ccg.c_codegen(list_of_g4DD_syms, list_of_g4DD_C_vars, enable_cse=True)

          cfc.register_CFunction(
              includes=includes, desc=desc, name=name, params=params, body=body,
              include_CodeParameters_h=True
          )
          print("    ... g4DD_kerr_schild() registration complete.")
```

### 6.b: `con_kerr_schild()` Worker

This function is structured identically to the `g4DD_kerr_schild` worker. It generates the C **worker** function `con_kerr_schild()`, whose only job is to compute the

40 unique Christoffel symbols for the unified Kerr-Schild metric.

The process is as follows: 1. **Access Symbolic Recipe:** It accesses the pre-computed symbolic Christoffel formulas from the global `Gamma4UDD_kerr` variable, which was generated in Step 5. 2. **Define C Assignment:** It creates a list of the 40 unique symbolic expressions and a corresponding list of the C variable names for the members of the `connection_struct` (e.g., `conn->Gamma4UDD012`). 3. **Generate C Code:** It uses `nrpy.c_codegen.c_codegen` to convert these highly complex symbolic expressions into optimized C code. The Common Subexpression Elimination (CSE) performed by `c_codegen` is absolutely essential here, as it reduces what would be thousands of floating-point operations into a much more manageable and efficient set of calculations. 4. **Register C Function:** Like the other workers, it registers the function using `nrpy.c_function.register_CFunction` and sets `include_CodeParameters_h=True` to handle its dependency on both `M_scale` and `a_spin`.

### 9.0.2 nrpy Functions Used in this Cell:

- `nrpy.indexedexp.declarerank3(name, dimension)`: Previously introduced. Used here to programmatically generate the C variable names for the Christoffel symbols that will be stored in the `connection_struct`.
- `nrpy.c_codegen.c_codegen(...)`: Previously introduced.
- `nrpy.c_function.register_CFunction(...)`: Previously introduced.

```python
[17]: def con_kerr_schild():
          """
          Generates and registers the C function to compute the Kerr-Schild Christoffel symbols.
          This is the new unified worker.
          """
          print(" -> Generating C worker function: con_kerr_schild()...")

          # We use the globally defined Gamma4UDD_kerr from the symbolic execution step
          list_of_Gamma_syms = []
          for i in range(4):
              for j in range(4):
                  for k in range(j, 4):
                      list_of_Gamma_syms.append(Gamma4UDD_kerr[i][j][k])

          conn_Gamma4UDD = ixp.declarerank3("conn->Gamma4UDD", dimension=4)
          list_of_Gamma_C_vars = []
          for i in range(4):
              for j in range(4):
                  for k in range(j, 4):
                      list_of_Gamma_C_vars.append(str(conn_Gamma4UDD[i][j][k]))

          includes = ["BHaH_defines.h"]
          desc = r"""@brief Computes the 40 unique Christoffel symbols for the Kerr metric in Kerr-Schild coords."""
          name = "con_kerr_schild"
          # The state vector y now contains (t, x, y, z)
          params = "const commondata_struct *restrict commondata, const params_struct *restrict params, const double y[4], connection_struct
      *restrict conn"
```

```
    body = ccg.c_codegen(list_of_Gamma_syms, list_of_Gamma_C_vars, enable_cse=True)

    cfc.register_CFunction(
        includes=includes, desc=desc, name=name, params=params, body=body,
        include_CodeParameters_h=True
    )
    print("    ... con_kerr_schild() registration complete.")
```

## 10  Con Schwarzschild

```
[18]: def con_schwarzschild_cartesian():
          """
          Generates and registers the C function to compute the Schwarzschild Christoffel symbols
          in standard Cartesian coordinates.
          """
          print(" -> Generating C worker function: con_schwarzschild_cartesian()...")

          # Use the globally defined Gamma4UDD_schw_cart
          list_of_Gamma_syms = []
          for i in range(4):
              for j in range(4):
                  for k in range(j, 4):
                      list_of_Gamma_syms.append(Gamma4UDD_schw_cart[i][j][k])

          conn_Gamma4UDD = ixp.declarerank3("conn->Gamma4UDD", dimension=4)
          list_of_Gamma_C_vars = []
          for i in range(4):
              for j in range(4):
                  for k in range(j, 4):
                      list_of_Gamma_C_vars.append(str(conn_Gamma4UDD[i][j][k]))

          includes = ["BHaH_defines.h"]
          desc = r"""@brief Computes the unique Christoffel symbols for the Schwarzschild metric in Cartesian coords."""
          name = "con_schwarzschild_cartesian"
          params = "const commondata_struct *restrict commondata, const params_struct *restrict params, const double y[4], connection_struct
      →*restrict conn"

          body = ccg.c_codegen(list_of_Gamma_syms, list_of_Gamma_C_vars, enable_cse=True)

          cfc.register_CFunction(
              includes=includes, desc=desc, name=name, params=params, body=body,
              include_CodeParameters_h=True
          )
```

```
print("    ... con_schwarzschild_cartesian() registration complete.")
```

### 6.d: `calculate_ode_rhs()` Engine

This function generates the core "engine" of our ODE solver: the C function `calculate_ode_rhs()`. Its single responsibility is to calculate the right-hand sides for our entire system of 9 ODEs. It is completely generic and has no knowledge of any specific metric; it only knows how to compute the geodesic equations given a set of Christoffel symbols and the spatial metric components.

The generation process is straightforward: 1. **Access Generic Recipe:** It accesses the global `all_rhs_expressions` list. This list contains the generic symbolic form of the ODEs for position, momentum, and proper length that we derived in Step 5. 2. **Generate C Code:** It passes this list directly to `nrpy.c_codegen.c_codegen`. The symbols used to build `all_rhs_expressions` were already created with their final C syntax (e.g., `y[5]` for the momentum, `conn->Gamma4UDD...` for the Christoffel placeholder, and `metric->g...` for the metric placeholder). Therefore, no further symbolic manipulation is needed. `nrpy` simply translates the expressions into optimized C code. 3. **Register C Function:** The generated C code body is bundled with its metadata and registered. This function does not require the `include_CodeParameters_h` flag because it is physically generic and receives all necessary information through its arguments.

#### 10.0.1 `nrpy` Functions Used in this Cell:

- `nrpy.c_codegen.c_codegen(...)`: Previously introduced.
- `nrpy.c_function.register_CFunction(...)`: Previously introduced.

```python
[19]: def calculate_ode_rhs_massive():
          """
          Generates the C engine to calculate the RHS of the 8 massive particle ODEs.
          """
          includes = ["BHaH_defines.h"]
          desc = r"""@brief Calculates the right-hand sides (RHS) of the 8 massive particle geodesic ODEs.

          This is a generic engine that implements the geodesic equation using pre-computed
          Christoffel symbols from the connection_struct.

          @param[in]  y        The 8-component state vector [t, x, y, z, u^t, u^x, u^y, u^z].
          @param[in]  conn     A pointer to the connection_struct holding the Christoffel symbols.
          @param[out] rhs_out  A pointer to the 8-component output array for the RHS results."""
          name = "calculate_ode_rhs_massive"
          params = "const double y[8], const connection_struct *restrict conn, double rhs_out[8]"

          rhs_output_vars = [f"rhs_out[{i}]" for i in range(8)]
          body = ccg.c_codegen(all_rhs_expressions_massive, rhs_output_vars)

          cfc.register_CFunction(
              includes=includes,
              desc=desc,
              name=name,
              params=params,
              body=body
          )
```

# 11 Markdown for check_conservation

```python
[20]: def check_conservation_massive():
          """
          Generates the C function `check_conservation_massive`.
          """
          print(" -> Generating C engine: check_conservation_massive()...")

          output_vars_kerr = ["*E", "*Lx", "*Ly", "*Lz", "*Q"]
          output_vars_schw = ["*E", "*Lx", "*Ly", "*Lz", "*Q"]

          body_C_code_kerr = ccg.c_codegen(list_of_expressions_kerr, output_vars_kerr, enable_cse=True, include_braces=False)
          body_C_code_schw = ccg.c_codegen(list_of_expressions_schw, output_vars_schw, enable_cse=True, include_braces=False)

          includes = ["BHaH_defines.h", "BHaH_function_prototypes.h"]
          desc = r"""@brief Computes conserved quantities (E, L_i, Q/L^2) for a given massive particle state vector."""
          name = "check_conservation_massive"
          params = """const commondata_struct *restrict commondata,
              const params_struct *restrict params,
              const metric_params *restrict metric_params_in,
              const double y[8],
              double *E, double *Lx, double *Ly, double *Lz, double *Q"""

          body = r"""
      // Unpack parameters from commondata struct that are needed symbolically
      const REAL a_spin = commondata->a_spin;

      metric_struct* metric = (metric_struct*)malloc(sizeof(metric_struct));
      g4DD_metric(commondata, params, metric_params_in, y, metric);

      if (metric_params_in->type == Kerr) {
          """ + body_C_code_kerr + r"""
      } else { // Both Schwarzschild types are now Cartesian
          """ + body_C_code_schw + r"""
      }

      free(metric);
      """

          cfc.register_CFunction(
              includes=includes, desc=desc, cfunc_type="void",
              name=name, params=params, body=body
          )
          print(f"    ... {name}() registration complete.")
```

# Step 7: C Code Generation - Orchestrators and Dispatchers

With the low-level "engine" and "worker" functions defined in the previous step, we now generate the higher-level C functions that manage the simulation. These functions are responsible for dispatching to the correct worker based on runtime parameters and for orchestrating the overall program flow.

- **Dispatchers** are functions that contain a `switch` statement to select the correct "worker" function based on the chosen metric (e.g., `Schwarzschild` vs. `Kerr`).
- **Orchestrators** are functions that execute a sequence of calls to other engines, workers, and dispatchers to perform a complex task, like setting up initial conditions or running the main integration loop.

### 7.a: `g4DD_metric()` Dispatcher

This Python function generates the C function `g4DD_metric()`, which serves as a high-level **dispatcher.** Its role is to select and call the correct worker function to compute the components of the metric tensor, $g_{\mu\nu}$.

The generated C code uses a `switch` statement that reads the `metric->type` member of the `metric_params` struct. In this project, both the Schwarzschild and Kerr spacetimes are handled by the unified `g4DD_kerr_schild()` worker function. The dispatcher calls this single worker, and the specific metric returned by the worker depends on the runtime value of the `a_spin` parameter (if `a_spin` is 0, the Schwarzschild metric is computed).

This modular approach cleanly separates the control flow (deciding *which* metric to use) from the physics implementation (the worker functions that know *how* to compute a specific metric). This makes the project easy to extend with new spacetimes in the future by adding new cases to the `switch` statement and new worker functions.

#### 11.0.1 nrpy Functions Used in this Cell:

- `nrpy.c_function.register_CFunction(...)`: Previously introduced. Used to register the manually written C code for the dispatcher function.

```
[21]:  # In file: V11_0_Python_to_C_via_NRPy.ipynb
       # In cell [65702cb7]


       def g4DD_metric():
           """
           Generates and registers the C function g4DD_metric(), which serves as a
           dispatcher to call the appropriate metric-specific worker function.
           """
           print(" -> Generating C dispatcher function: g4DD_metric()...")

           includes = ["BHaH_defines.h", "BHaH_function_prototypes.h"]
           desc = r"""@brief Dispatcher to compute the 4-metric g_munu for the chosen metric."""
           name = "g4DD_metric"
           # The signature is now coordinate-aware, but the y vector is always Cartesian here.
           params = "const commondata_struct *restrict commondata, const params_struct *restrict params, const metric_params *restrict metric,
       →const double y[8], metric_struct *restrict metric_out"

           body = r"""
           // The state vector y_pos contains only the position coordinates.
           const double y_pos[4] = {y[0], y[1], y[2], y[3]};

           // This switch statement chooses which "worker" function to call
```

```
        // based on the metric type provided.
        switch(metric->type) {
            case Schwarzschild:
            case Kerr:
                // For Kerr or Schwarzschild in KS coords, call the unified Kerr-Schild C function.
                g4DD_kerr_schild(commondata, params, y_pos, metric_out);
                break;
            // <-- MODIFIED: Call the new Cartesian worker
            case Schwarzschild_Standard:
                g4DD_schwarzschild_cartesian(commondata, params, y_pos, metric_out);
                break;
            case Numerical:
                printf("Error: Numerical metric not supported yet.\n");
                exit(1);
                break;
            default:
                printf("Error: MetricType %d not supported in g4DD_metric() yet.\n", metric->type);
                exit(1);
                break;
        }
    """

    cfc.register_CFunction(includes=includes, desc=desc, name=name, params=params, body=body)
    print("    ... g4DD_metric() registration complete.")
```

### 7.b: `connections()` Dispatcher

This Python function generates the C function `connections()`, which acts as a second **dispatcher.** Its sole responsibility is to select and call the correct metric-specific worker function (like `con_kerr_schild()`) to compute the Christoffel symbols.

Like the `g4DD_metric()` dispatcher, the generated C code uses a `switch` statement based on the `metric->type`. It dispatches the call to the appropriate specialized worker, which in this case is the unified `con_kerr_schild()` function for both Kerr and Schwarzschild spacetimes. This design is highly extensible: adding a new metric simply requires writing a new worker function for its Christoffel symbols and adding a new `case` to this `switch` statement.

This function demonstrates how `nrpy` allows for the seamless integration of developer-written control flow with the automatically generated worker functions.

**11.0.2   `nrpy` Functions Used in this Cell:**

- `nrpy.c_function.register_CFunction(...)`: Previously introduced.

```
[22]: # In file: V11_0_Python_to_C_via_NRPy.ipynb
      # In cell [b92b7851]


      def connections():
          """
          Generates and registers the C dispatcher for Christoffel symbols.
          """
```

```python
    print(" -> Generating C dispatcher: connections()...")

    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "stdio.h", "stdlib.h"]
    desc = r"""@brief Dispatcher to compute Christoffel symbols for the chosen metric."""

    name = "connections"
    cfunc_type = "void"
    params = "const commondata_struct *restrict commondata, const params_struct *restrict params, const metric_params *restrict metric,␣
↪const double y[8], connection_struct *restrict conn"

    body = r"""
    // The state vector y_pos contains only the position coordinates.
    const double y_pos[4] = {y[0], y[1], y[2], y[3]};

    // This switch statement chooses which "worker" function to call
    // based on the metric type provided.
    switch(metric->type) {
        case Schwarzschild:
        case Kerr:
            con_kerr_schild(commondata, params, y_pos, conn);
            break;
        // <-- MODIFIED: Call the new Cartesian worker
        case Schwarzschild_Standard:
            con_schwarzschild_cartesian(commondata, params, y_pos, conn);
            break;
        case Numerical:
            printf("Error: Numerical metric not supported yet.\n");
            exit(1);
            break;
        default:
            printf("Error: MetricType %d not supported yet.\n", metric->type);
            exit(1);
            break;
    }
"""

    cfc.register_CFunction(
        includes=includes, desc=desc, cfunc_type=cfunc_type,
        name=name, params=params, body=body
    )
    print("    ... connections() registration complete.")
```

## 12 Calculate ut and uphi values

```
[23]:  # In V1_5_mass_geodesic.ipynb, Cell ID bb00ac33 (UPDATED)

       def calculate_ut_uphi_from_r():
           """
           Generates a C helper function to compute u^t and u^phi from a radius
           using a numerically stable recipe.
           """
           print(" -> Generating C engine: calculate_ut_uphi_from_r() [STABLE VERSION]...")

           includes = ["BHaH_defines.h"]
           desc = r"""@brief Computes u^t and u^phi for a circular orbit at a given radius using a numerically stable method."""
           name = "calculate_ut_uphi_from_r"
           params = "const double r_initial, const commondata_struct *restrict commondata, const params_struct *restrict params, double *ut,
       ↪double *uphi"

           # The global ut_expr and uphi_expr are now the stable versions
           body = ccg.c_codegen(
               [ut_expr, uphi_expr],
               ["*ut", "*uphi"],
               enable_cse=True
           )

           cfc.register_CFunction(
               includes=includes,
               desc=desc,
               name=name,
               params=params,
               body=body,
               include_CodeParameters_h=True
           )
           print("    ... calculate_ut_uphi_from_r() registration complete.")
```

## 13 set_initial_conditions_massive

```
[24]:  # In file: V1_5_mass_geodesic (Copy).ipynb
       # In cell [bfd43b53] (DEFINITIVE STABLE VERSION)

       def set_initial_conditions_massive():
           """
           Generates the C engine to set the full initial 8-component state vector.
```

```python
VERSION 2: This version implements the NUMERICALLY STABLE method for
calculating u^t, as detailed in the Gemini report. It computes the
Boyer-Lindquist metric components and Omega internally to avoid
catastrophic cancellation.
"""
print(" -> Generating C engine: set_initial_conditions_massive() [STABLE VERSION]...")

includes = ["BHaH_defines.h", "<math.h>"]
desc = r"""@brief Sets the initial 8-component state vector for a massive particle using a numerically stable method."""
name = "set_initial_conditions_massive"
params = "const particle_initial_state_t *restrict initial_state, const commondata_struct *restrict commondata, double y_out[8]"

body = r"""
// Unpack parameters for clarity
const double M = commondata->M_scale;
const double a = commondata->a_spin;

// Unpack initial position
const double x = initial_state->pos[1];
const double y = initial_state->pos[2];
const double z = initial_state->pos[3];
const double r = sqrt(x*x + y*y + z*z);

// --- Numerically Stable Method to find u^t and u^phi ---

// Step 1: Calculate Omega = d(phi)/dt
const double Omega = sqrt(M) / (pow(r, 1.5) + a * sqrt(M));

// Step 2: Calculate Boyer-Lindquist metric components in the equatorial plane
const double g_tt = -(1.0 - 2.0*M/r);
const double g_tphi = -2.0*a*M/r;
const double g_phiphi = r*r + a*a + (2.0*M*a*a)/r;

// Step 3: Solve for u^t using the normalization condition
const double ut_inv_denom = g_tt + 2.0*g_tphi*Omega + g_phiphi*Omega*Omega;
if (ut_inv_denom >= 0) {
    // This indicates an unstable or invalid orbit. Set state to NaN.
    for(int i=0; i<8; i++) y_out[i] = NAN;
    return;
}
const double ut = sqrt(-1.0 / ut_inv_denom);
const double uphi = Omega * ut;

// --- Assemble the final 8-component state vector ---
```

```
        y_out[0] = initial_state->pos[0]; // t
        y_out[1] = x;
        y_out[2] = y;
        y_out[3] = z;

        y_out[4] = ut;
        // Transform u^phi to Cartesian u^x, u^y
        y_out[5] = -y * uphi;
        y_out[6] =  x * uphi;
        y_out[7] = 0.0; // u^z is zero for equatorial orbits
        """

    cfc.register_CFunction(
        includes=includes,
        desc=desc,
        name=name,
        params=params,
        body=body
    )
```

# 14 Disk Initial Conditions

```python
[25]: def generate_disk_initial_conditions():
    """
    Generates a C function that programmatically creates the initial conditions
    for a Keplerian disk.
    """
    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h"]
    desc = r"""@brief Generates the complete y[8] initial state for all particles in a Keplerian disk."""
    cfunc_type = "int"
    name = "generate_disk_initial_conditions"

    params = "const commondata_struct *restrict commondata, const params_struct *restrict params, double *restrict y_initial_states"


    body = r"""
    int particle_count = 0;
    const double dr = (commondata->disk_num_r > 1) ? (commondata->disk_r_max - commondata->disk_r_min) / (commondata->disk_num_r - 1) :␣
    ↪0;

    for (int i = 0; i < commondata->disk_num_r; i++) {
        const double r = commondata->disk_r_min + i * dr;
```

```
        const int num_phi_at_r = (commondata->disk_num_phi > 1) ? (int)(commondata->disk_num_phi * (r / commondata->disk_r_max)) : 1;
        if (num_phi_at_r == 0) continue;
        const double dphi = 2.0 * M_PI / num_phi_at_r;

        double ut_at_r, uphi_at_r;
        // The call is now valid because 'params' is available in this function's scope.
        calculate_ut_uphi_from_r(r, commondata, params, &ut_at_r, &uphi_at_r);

        for (int j = 0; j < num_phi_at_r; j++) {
            const double phi = j * dphi;
            const double cos_phi = cos(phi);
            const double sin_phi = sin(phi);

            double *y = &y_initial_states[particle_count * 8];

            y[0] = 0.0;
            y[1] = r * cos_phi;
            y[2] = r * sin_phi;
            y[3] = 0.0;

            y[4] = ut_at_r;
            y[5] = -(r * sin_phi) * uphi_at_r;
            y[6] =  (r * cos_phi) * uphi_at_r;
            y[7] = 0.0;

            particle_count++;
        }
    }
}
return particle_count;
"""


cfc.register_CFunction(
    includes=includes,
    desc=desc,
    cfunc_type=cfunc_type,
    name=name,
    params=params,
    body=body
)
```

## 15  Spiral Galaxy initail conditions

```python
[26]: def generate_spiral_galaxy_initial_conditions():
          """
          Generates a C function that programmatically creates the initial conditions
          for a spiral galaxy disk.

          UPDATED to use runtime parameters from the commondata struct to control
          the number and tightness of the spiral arms.
          """
          print(" -> Generating C function: generate_spiral_galaxy_initial_conditions()...")

          includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "<math.h>", "<stdlib.h>"] # Added stdlib.h for rand()
          desc = r"""@brief Generates the complete y[8] initial state for all particles in a spiral galaxy disk."""
          cfunc_type = "int"
          name = "generate_spiral_galaxy_initial_conditions"

          params = "const commondata_struct *restrict commondata, const params_struct *restrict params, double *restrict y_initial_states"

          body = r"""
          int particle_count = 0;
          const int num_particles_total = commondata->disk_num_r * commondata->disk_num_phi;

          // --- Spiral galaxy parameters are now read from the commondata struct ---
          const int num_arms = commondata->spiral_galaxy_num_arms;
          const double arm_tightness = commondata->spiral_galaxy_arm_tightness;

          // Seed the random number generator for reproducibility if needed.
          // For true randomness on each run, you could seed with time(NULL).
          srand(42);

          for (int i = 0; i < num_particles_total; i++) {
              // --- Particle Placement Logic (Unchanged, but now uses parameters) ---

              const double r = commondata->disk_r_min + (commondata->disk_r_max - commondata->disk_r_min) * sqrt((double)rand() / RAND_MAX);

              // The base angle for this radius from the logarithmic spiral formula.
              const double theta_base = (1.0 / arm_tightness) * log(r / commondata->disk_r_min);

              const int arm_index = rand() % num_arms;
              const double arm_offset = (2.0 * M_PI / num_arms) * arm_index;

              const double phi_spread = (M_PI / num_arms) * 0.2 * ((double)rand() / RAND_MAX - 0.5);
```

```
        const double phi = theta_base + arm_offset + phi_spread;

        // --- The rest of this logic is IDENTICAL to the original ---

        const double cos_phi = cos(phi);
        const double sin_phi = sin(phi);

        double ut_at_r, uphi_at_r;
        calculate_ut_uphi_from_r(r, commondata, params, &ut_at_r, &uphi_at_r);

        double *y = &y_initial_states[particle_count * 8];

        y[0] = 0.0;
        y[1] = r * cos_phi;
        y[2] = r * sin_phi;
        y[3] = 0.0;
        y[4] = ut_at_r;
        y[5] = -(r * sin_phi) * uphi_at_r;
        y[6] =  (r * cos_phi) * uphi_at_r;
        y[7] = 0.0;

        particle_count++;
    }
    return particle_count;
    """

    cfc.register_CFunction(
        includes=includes,
        desc=desc,
        cfunc_type=cfunc_type,
        name=name,
        params=params,
        body=body
    )
    print("    ... generate_spiral_galaxy_initial_conditions() registration complete.")
```

# 16   Generate barred flocculent spiral ic

```
[27]: def generate_barred_flocculent_spiral_ic():
        """
        Generates a C function that programmatically creates the initial conditions
        for a realistic barred spiral galaxy with clumpy, flocculent arms.
```

```python
    This function uses rejection sampling to place particles in one of three regions:
    a central bulge, a rectangular bar, or flocculent spiral arms.

    Despite the complex geometry, all particles are placed on stable, circular
    Keplerian orbits, satisfying the project constraints.
    """
    print(" -> Generating C function: generate_barred_flocculent_spiral_ic()...")

    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "<math.h>", "<stdlib.h>", "<time.h>"]
    desc = r"""@brief Generates the initial state for all particles in a barred flocculent spiral galaxy."""
    cfunc_type = "int"
    name = "generate_barred_flocculent_spiral_ic"

    params = "const commondata_struct *restrict commondata, const params_struct *restrict params, double *restrict y_initial_states"


    body = r"""
        // Seed the random number generator. A unique seed is needed for each thread in parallel.
        // We will use the particle index + a base seed.
        unsigned int seed = 42;

        int particle_count = 0;
        const int num_particles_total = commondata->disk_num_r * commondata->disk_num_phi;

        // Unpack geometry parameters from commondata for clarity
        const double r_min = commondata->disk_r_min;
        const double r_max = commondata->disk_r_max;
        const double bar_len = commondata->bar_length;
        const double bar_width = commondata->bar_length * commondata->bar_aspect_ratio;
        const double bulge_rad = commondata->bulge_radius;
        const int num_arms = commondata->spiral_galaxy_num_arms;
        const double arm_tightness = commondata->spiral_galaxy_arm_tightness;

        for (int i = 0; i < num_particles_total; i++) {
            double r, phi, x, y;

            // Use the thread-safe random number generator
            seed += i;

            // --- REJECTION SAMPLING LOOP ---
            while (1) {
                // Generate a trial particle position uniformly in the disk annulus
                double random_val = (double)rand_r(&seed) / RAND_MAX;
```

```c
        r = sqrt(random_val * (r_max*r_max - r_min*r_min) + r_min*r_min);
        phi = 2.0 * M_PI * ((double)rand_r(&seed) / RAND_MAX);
        x = r * cos(phi);
        y = r * sin(phi);

        double acceptance_prob = 0.0;

        // Region 1: Central Bulge
        if (r < bulge_rad) {
            acceptance_prob = commondata->bulge_density_factor * commondata->arm_particle_density;
        }
        // Region 2: Central Bar
        else if (fabs(x) < bar_len / 2.0 && fabs(y) < bar_width / 2.0) {
            acceptance_prob = commondata->bar_density_factor * commondata->arm_particle_density;
        }
        // Region 3: Flocculent Spiral Arms
        else {
            // Optimization: Do a cheap check first. If the base probability fails,
            // no need to do expensive log/exp/cos calls.
            if (((double)rand_r(&seed) / RAND_MAX) < commondata->arm_particle_density) {
                // --- CORRECTED ARM PROFILE LOGIC ---
                double theta_base = (1.0 / arm_tightness) * log(r / r_min);
                double delta_phi_raw = phi - theta_base;
                double angle_between_arms = 2.0 * M_PI / num_arms;
                double delta_phi_folded = fmod(delta_phi_raw, angle_between_arms);

                if (delta_phi_folded > 0.5 * angle_between_arms) {
                    delta_phi_folded -= angle_between_arms;
                } else if (delta_phi_folded < -0.5 * angle_between_arms) {
                    delta_phi_folded += angle_between_arms;
                }

                double arm_profile = exp(-0.5 * SQR(delta_phi_folded / (arm_tightness * 0.5)));

                // Flocculent/clumpy modulation
                double clump_modulation = cos(commondata->arm_clumpiness_factor * (theta_base - phi));
                clump_modulation *= cos(num_arms * phi * commondata->arm_clump_size);

                acceptance_prob = arm_profile + 0.5 * clump_modulation;
            }
        }

        if (((double)rand_r(&seed) / RAND_MAX) < acceptance_prob) {
            break; // Accept this particle's position
```

```
            }
        } // End of rejection sampling while loop

        // --- Velocity Assignment ---
        double ut_at_r, uphi_at_r;
        calculate_ut_uphi_from_r(r, commondata, params, &ut_at_r, &uphi_at_r);

        // --- Store the Full 8-Component State Vector ---
        double *y_state = &y_initial_states[particle_count * 8];

        y_state[0] = 0.0;
        y_state[1] = x;
        y_state[2] = y;
        y_state[3] = 0.0;
        y_state[4] = ut_at_r;
        y_state[5] = -y * uphi_at_r;
        y_state[6] =  x * uphi_at_r;
        y_state[7] = 0.0;

        particle_count++;
    }

    return particle_count;
"""

    cfc.register_CFunction(
        includes=includes,
        desc=desc,
        cfunc_type=cfunc_type,
        name=name,
        params=params,
        body=body
    )
```

### 7.d: The GSL Wrapper Function (Redo)

```
[28]: def ode_gsl_wrapper_massive():
          """
          Generates the C function that acts as a bridge between the GSL ODE
          solver and our project-specific physics functions.
          """
          includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "gsl/gsl_errno.h"]
          desc = r"""@brief GSL wrapper for the massive particle geodesic ODEs.

          This function matches the signature required by the GSL ODE solver. It unpacks
```

```
    the gsl_params carrier struct and calls the dispatchers for the metric and
    Christoffel symbols, before finally calling the RHS engine.

    @param[in]  t      The current value of the independent variable (proper time τ). Unused.
    @param[in]  y      The current 8-component state vector.
    @param[in]  params A generic void pointer to our gsl_params carrier struct.
    @param[out] f      A pointer to the 8-component output array where GSL expects the RHS results."""
cfunc_type = "int"
name = "ode_gsl_wrapper_massive"
params = "double t, const double y[8], double f[8], void *params"


body = r"""
    (void)t; // Proper time 't' is not explicitly used in the RHS expressions.

    // Unpack the carrier struct to access simulation parameters and metric choice.
    gsl_params *gsl_parameters = (gsl_params *)params;

    // Declare structs to hold metric and connection values.
    metric_struct g4DD;
    connection_struct conn;

    // Call dispatchers to compute the metric and Christoffel symbols at the current position y.
    // Note: The y array is 8D, but these functions only need the first 4 position components.
    g4DD_metric(gsl_parameters->commondata, gsl_parameters->params, gsl_parameters->metric, y, &g4DD);
    connections(gsl_parameters->commondata, gsl_parameters->params, gsl_parameters->metric, y, &conn);

    // Call the engine to compute the RHS of the ODEs.
    calculate_ode_rhs_massive(y, &conn, f);

    return GSL_SUCCESS;
"""

cfc.register_CFunction(
    includes=includes,
    desc=desc,
    cfunc_type=cfunc_type,
    name=name,
    params=params,
    body=body
)
```

# 17 Run Mass integrator production

```python
[29]: def run_mass_integrator_production():
          """
          Generates the C orchestrator for the production run.

          This definitive version acts as a dispatcher. It reads the
          initial_conditions_type parameter and calls the appropriate particle
          generator function before starting the full integration and snapshotting loop.
          """
          print(" -> Generating C orchestrator: run_mass_integrator_production() [Dispatcher Version]...")

          includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "<math.h>", "<sys/stat.h>", "<string.h>", "<stdio.h>", "<stdlib.h>"]
          desc = r"""@brief Orchestrates the full production run for the mass integrator."""
          name = "run_mass_integrator_production"
          params = "const commondata_struct *restrict commondata, const params_struct *restrict params, const metric_params *restrict metric"

          body = r"""
      // Step 1: Allocate memory for the flat initial state array
      const int max_particles = commondata->disk_num_r * commondata->disk_num_phi;
      double *y_initial_states = (double *)malloc(sizeof(double) * 8 * max_particles);
      if (y_initial_states == NULL) { fprintf(stderr, "Error: Failed to allocate memory for initial states.\n"); exit(1); }
      int num_particles = 0;

      // --- Step 2: Dispatch to the correct initial conditions generator ---
      printf("Generating initial conditions of type: %s\n", commondata->initial_conditions_type);
       if (strcmp(commondata->initial_conditions_type, "SpiralGalaxy") == 0) {
          num_particles = generate_spiral_galaxy_initial_conditions(commondata, params, y_initial_states);
      } else if (strcmp(commondata->initial_conditions_type, "BarredFlocculentSpiral") == 0) {
          num_particles = generate_barred_flocculent_spiral_ic(commondata, params, y_initial_states);
      } else {// Default to KeplerianDisk
          if (strcmp(commondata->initial_conditions_type, "KeplerianDisk") != 0) {
              printf("Warning: Unrecognized initial_conditions_type '%s'. Defaulting to KeplerianDisk.\n",
  ↪commondata->initial_conditions_type);
          }
          num_particles = generate_disk_initial_conditions(commondata, params, y_initial_states);
      }
      printf("Generated %d particles.\n", num_particles);

      // Step 3: Create and populate our primary data structure, an array of mass_particle_state_t
      mass_particle_state_t *particle_states = (mass_particle_state_t *)malloc(sizeof(mass_particle_state_t) * num_particles);
      if (particle_states == NULL) { fprintf(stderr, "Error: Failed to allocate memory for particle states.\n"); exit(1); }
      for (int i=0; i<num_particles; i++) {
          double *y_start = &y_initial_states[i*8];
```

```c
        particle_states[i].id = i;
        particle_states[i].pos[0] = y_start[1];
        particle_states[i].pos[1] = y_start[2];
        particle_states[i].pos[2] = y_start[3];
        particle_states[i].u[0] = y_start[4];
        particle_states[i].u[1] = y_start[5];
        particle_states[i].u[2] = y_start[6];
        particle_states[i].u[3] = y_start[7];

        const double r = sqrt(y_start[1]*y_start[1] + y_start[2]*y_start[2]);
        particle_states[i].lambda_rest = commondata->disk_lambda_rest_at_r_min * pow(r / commondata->disk_r_min, 0.75);
        particle_states[i].j_intrinsic = (float)pow(r / commondata->disk_r_min, -3.0);
    }
    free(y_initial_states);

    // Step 4: Create the output directory
    mkdir(commondata->output_folder, 0755);

    // Step 5: Main Time Evolution and Snapshotting Loop
    int snapshot_count = 0;
    for (double current_t = 0; current_t <= commondata->t_final; current_t += commondata->snapshot_every_t) {
        char filename[200];
        snprintf(filename, 200, "%s/mass_blueprint_t_%04d.bin", commondata->output_folder, snapshot_count);
        printf("Saving snapshot: %s (t=%.2f)\n", filename, current_t);

        FILE *fp_out = fopen(filename, "wb");
        if (fp_out == NULL) { exit(1); }

        int active_particles = 0;
        for(int i=0; i<num_particles; i++) {
            if (!isnan(particle_states[i].pos[0])) active_particles++;
        }
        fwrite(&active_particles, sizeof(int), 1, fp_out);
        for (int i=0; i<num_particles; i++) {
            if (!isnan(particle_states[i].pos[0])) {
                fwrite(&particle_states[i], sizeof(mass_particle_state_t), 1, fp_out);
            }
        }
        fclose(fp_out);
        snapshot_count++;

        if (current_t >= commondata->t_final) break;

        // Evolve all particles for one snapshot interval
```

```
        const double t_next_snapshot = current_t + commondata->snapshot_every_t;
        #pragma omp parallel for
        for (int i = 0; i < num_particles; i++) {
            if (isnan(particle_states[i].pos[0])) continue;

            double y_particle[8];
            y_particle[0] = current_t;
            y_particle[1] = particle_states[i].pos[0];
            y_particle[2] = particle_states[i].pos[1];
            y_particle[3] = particle_states[i].pos[2];
            y_particle[4] = particle_states[i].u[0];
            y_particle[5] = particle_states[i].u[1];
            y_particle[6] = particle_states[i].u[2];
            y_particle[7] = particle_states[i].u[3];

            int status = integrate_single_particle(commondata, params, metric, y_particle[0], t_next_snapshot, y_particle);

            if (status != 0) {
                particle_states[i].pos[0] = NAN; // Mark particle as terminated
            } else {
                particle_states[i].pos[0] = y_particle[1];
                particle_states[i].pos[1] = y_particle[2];
                particle_states[i].pos[2] = y_particle[3];
                particle_states[i].u[0] = y_particle[4];
                particle_states[i].u[1] = y_particle[5];
                particle_states[i].u[2] = y_particle[6];
                particle_states[i].u[3] = y_particle[7];
            }
        }
    }
    free(particle_states);
    """
    cfc.register_CFunction(includes=includes, desc=desc, name=name, params=params, body=body)
```

## 18   Main integration code

```
[30]: def integrate_single_particle():
          """
          Generates the main C integration loop for a single massive particle.
          This high-performance "production" version uses the GSL driver to ensure
          the state is returned at the exact requested snapshot times.
          """
          includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "gsl/gsl_errno.h", "gsl/gsl_odeiv2.h", "<math.h>"]
```

```
desc = r"""@brief Integrates a single massive particle path between two times.

This function uses the GSL driver, which internally uses an adaptive
step-size algorithm (RKF45) to evolve the particle's state vector y_in_out
from t_start to t_end, returning the state at the precise t_end.

@param[in]      commondata  Pointer to commondata struct.
@param[in]      params      Pointer to params_struct.
@param[in]      metric      Pointer to metric_params struct.
@param[in]      t_start     The starting proper time (τ) for the integration.
@param[in]      t_end       The ending proper time (τ) for the integration.
@param[in,out]  y_in_out    The 8-component state vector. Input is the state at t_start, output is the state at t_end.

@return 0 on success, 1 on GSL failure.
"""
cfunc_type = "int"
name = "integrate_single_particle"
params = """const commondata_struct *restrict commondata,
const params_struct *restrict params,
const metric_params *restrict metric,
const double t_start, const double t_end,
double y_in_out[8]"""


body = r"""
// Define the GSL ODE system
gsl_params gsl_parameters = {commondata, params, metric};
gsl_odeiv2_system sys = {ode_gsl_wrapper_massive, NULL, 8, &gsl_parameters};

// Set up the GSL driver
gsl_odeiv2_driver *d = gsl_odeiv2_driver_alloc_y_new(
    &sys, gsl_odeiv2_step_rkf45, 1e-6, 1e-11, 1e-11);

double t = t_start;

// The driver will take internal steps to reach t_end precisely.
int status = gsl_odeiv2_driver_apply(d, &t, t_end, y_in_out);

if (status != GSL_SUCCESS) {
    // Don't print an error here, as the orchestrator will check the status.
    // Just free memory and return the failure code.
    gsl_odeiv2_driver_free(d);
    return 1; // Return failure code
}
```

```
    // Robustness check after the step
    const double r_sq = y_in_out[1]*y_in_out[1] + y_in_out[2]*y_in_out[2] + y_in_out[3]*y_in_out[3];
    const double r_horizon = commondata->M_scale * (1.0 + sqrt(1.0 - commondata->a_spin*commondata->a_spin));

    if (r_sq < r_horizon*r_horizon || r_sq > r_escape*r_escape || fabs(y_in_out[4]) > ut_max) {
        gsl_odeiv2_driver_free(d);
        return 1; // Return failure code
    }

    gsl_odeiv2_driver_free(d);
    return 0; // Return success code
    """

    cfc.register_CFunction(
        includes=includes,
        desc=desc,
        cfunc_type=cfunc_type,
        name=name,
        params=params,
        body=body,
        include_CodeParameters_h=True
    )
```

# 19  Bebugging integration (for single masslike particle)

```python
[31]: def integrate_single_particle_DEBUG():
          """
          Generates a DEBUG version of the integrator that writes the full trajectory
          of a single massive particle to a text file for validation.
          """
          includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "gsl/gsl_errno.h", "gsl/gsl_odeiv2.h"]
          desc = r"""@brief DEBUG integrator for a single massive particle.

      This function integrates the path of a single particle and writes the full
      8-component state vector at each step to 'massive_particle_path.txt'.
      It also prints progress to the console and checks for termination conditions.

      @param[in]  commondata  Pointer to the commondata_struct.
      @param[in]  params      Pointer to the params_struct.
      @param[in]  metric      Pointer to the metric_params struct.
      @param[in]  start_y     The 8-component initial state vector.
      @param[out] final_y_state The 8-component final state vector upon termination."""
          cfunc_type = "void"
```

```
name = "integrate_single_particle_DEBUG"
params = """const commondata_struct *restrict commondata,
const params_struct *restrict params,
const metric_params *restrict metric,
const double start_y[8],
double final_y_state[8]"""

body = r"""
// GSL Setup
const gsl_odeiv2_step_type * T = gsl_odeiv2_step_rkf45;
gsl_odeiv2_step * step = gsl_odeiv2_step_alloc(T, 8);
gsl_odeiv2_control * control = gsl_odeiv2_control_yp_new(1e-14, 1e-14);
gsl_odeiv2_evolve * evol = gsl_odeiv2_evolve_alloc(8);
gsl_params gsl_parameters = {commondata, params, metric};
gsl_odeiv2_system sys = {ode_gsl_wrapper_massive, NULL, 8, &gsl_parameters};

double y_c[8];
double t = 0.0, dt = 0.01; // t is proper time τ
for (int j = 0; j < 8; j++) { y_c[j] = start_y[j]; }

// Setup output file
FILE *fp = fopen("massive_particle_path.txt", "w");
if (fp == NULL) {
    fprintf(stderr, "Error: Could not open massive_particle_path.txt for writing.\n");
    exit(1);
}
fprintf(fp, "# ProperTime_tau\tCoordTime_t\tx\ty\tz\tu^t\tu^x\tu^y\tu^z\n");

printf("Starting debug trace for single massive particle...\n");
printf("Step | Proper Time (τ) | Coord Time (t) |     x      |      y      |     z      |      u^t   \n");
printf("-------------------------------------------------------------------------------------------\n");

// Main Integration Loop
for (int i = 0; i < 2000000; i++) {
    int status = gsl_odeiv2_evolve_apply(evol, control, step, &sys, &t, 1e10, &dt, y_c);

    // Write full state to file
    fprintf(fp, "%.6e\t%.6e\t%.6e\t%.6e\t%.6e\t%.6e\t%.6e\t%.6e\t%.6e\n",
            t, y_c[0], y_c[1], y_c[2], y_c[3], y_c[4], y_c[5], y_c[6], y_c[7]);

    if (i % 500 == 0) {
        printf("%4d | %15.4e | %14.4f | %10.4f | %10.4f | %10.4f | %10.4f\n",
                i, t, y_c[0], y_c[1], y_c[2], y_c[3], y_c[4]);
    }
```

```
        const double r_sq = y_c[1]*y_c[1] + y_c[2]*y_c[2] + y_c[3]*y_c[3];
        // Event horizon radius for a Kerr black hole
        const double r_horizon = commondata->M_scale * (1.0 + sqrt(1.0 - commondata->a_spin*commondata->a_spin));

        // Termination Conditions
        if (status != GSL_SUCCESS) { printf("Termination: GSL ERROR (status = %d)\n", status); break; }
        if (r_sq < r_horizon*r_horizon) { printf("Termination: Fell below event horizon (r=%.2f)\n", sqrt(r_sq)); break; }
        if (r_sq > r_escape*r_escape) { printf("Termination: Escaped to r > %.1f\n", r_escape); break; }
        if (fabs(y_c[4]) > ut_max) { printf("Termination: Runaway u^t > %.1e\n", ut_max); break; }
        if (fabs(y_c[0]) > t_max_integration) { printf("Termination: Exceeded max integration time t > %.1f\n", t_max_integration);
↪break; }
    }

    // Copy final state to output and clean up
    for(int j=0; j<8; j++) { final_y_state[j] = y_c[j]; }
    fclose(fp);
    gsl_odeiv2_evolve_free(evol);
    gsl_odeiv2_control_free(control);
    gsl_odeiv2_step_free(step);
    """

    cfc.register_CFunction(
        includes=includes,
        desc=desc,
        cfunc_type=cfunc_type,
        name=name,
        params=params,
        body=body,
        include_CodeParameters_h=True
    )
```

## 20  Main

```
[32]: def main():
    """
    Generates the main() C function.

    This final version restores the detailed parameter printout for production runs.
    The production run logic is now handled by the run_mass_integrator_production dispatcher.
    """
    print(" -> Generating C entry point: main() [Final Version]...")
```

```
includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "<string.h>", "<stdio.h>", "<stdlib.h>"]

desc = r"""@brief Main entry point for the massive particle geodesic integrator."""
cfunc_type = "int"
name = "main"
params = "int argc, const char *argv[]"

body = r"""
// Step 1: Initialize structs and parameters
commondata_struct commondata;
params_struct params; // This struct is currently unused but required by function signatures.
metric_params metric;

commondata_struct_set_to_default(&commondata);
cmdline_input_and_parfile_parser(&commondata, argc, argv);

metric.type = (commondata.a_spin == 0.0) ? Schwarzschild : Kerr;

// Step 2: Check the run mode and execute the appropriate logic
if (commondata.run_in_debug_mode) {
    /********************************/
    /*** SINGLE-PARTICLE DEBUG MODE ***/
    /********************************/
    particle_initial_state_t initial_state;
    initial_state.id = 0;

    const char *filename = "particle_debug_initial_conditions.txt";
    FILE *fp_in = fopen(filename, "r");

    // --- RESTORED FALLBACK LOGIC ---
    // If the file doesn't exist, create it with default values.
    if (fp_in == NULL) {
        printf("File '%s' not found. Creating it with default values.\n", filename);
        fp_in = fopen(filename, "w");
        if (fp_in == NULL) {
            fprintf(stderr, "Error: Could not create '%s'.\n", filename);
            return 1;
        }
        fprintf(fp_in, "# Format: t_initial pos_x pos_y pos_z u_x u_y u_z\n");
        fprintf(fp_in, "# u_i are the spatial components of the 4-velocity (dx/dτ).\n");
        fprintf(fp_in, "0.0 10.0  0.0  0.0   0.0  0.363232  0.0\n");
        fclose(fp_in);

        // Re-open the newly created file for reading
```

```c
        fp_in = fopen(filename, "r");
        if (fp_in == NULL) {
            fprintf(stderr, "Error: Could not re-open '%s' for reading.\n", filename);
            return 1;
        }
    }
    // --- END OF RESTORED LOGIC ---

    char line[256];
    while (fgets(line, sizeof(line), fp_in)) {
        if (line[0] != '#') {
            sscanf(line, "%lf %lf %lf %lf %lf %lf %lf",
                    &initial_state.pos[0], &initial_state.pos[1], &initial_state.pos[2], &initial_state.pos[3],
                    &initial_state.u_spatial[0], &initial_state.u_spatial[1], &initial_state.u_spatial[2]);
            break;
        }
    }
    fclose(fp_in);

    printf("--- Single Particle Debug Run ---\n");
    printf("  pos = (t=%.4f, x=%.4f, y=%.4f, z=%.4f)\n", initial_state.pos[0], initial_state.pos[1], initial_state.pos[2],
→initial_state.pos[3]);
    printf("  u_spatial = (%.4f, %.4f, %.4f)\n", initial_state.u_spatial[0], initial_state.u_spatial[1], initial_state.
→u_spatial[2]);

    double y_start[8], y_final[8];


    set_initial_conditions_massive(&initial_state, &commondata, y_start);


    printf("\nInitial State Vector (y_start):\n");
    printf("  t=%.2f, x=%.2f, y=%.2f, z=%.2f\n", y_start[0], y_start[1], y_start[2], y_start[3]);
    printf("  u^t=%.4f, u^x=%.4f, u^y=%.4f, u^z=%.4f\n\n", y_start[4], y_start[5], y_start[6], y_start[7]);

    if(commondata.perform_conservation_check) {
        double E_i, Lx_i, Ly_i, Lz_i, Q_i, E_f, Lx_f, Ly_f, Lz_f, Q_f;
        check_conservation_massive(&commondata, &params, &metric, y_start, &E_i, &Lx_i, &Ly_i, &Lz_i, &Q_i);
        integrate_single_particle_DEBUG(&commondata, &params, &metric, y_start, y_final);
        check_conservation_massive(&commondata, &params, &metric, y_final, &E_f, &Lx_f, &Ly_f, &Lz_f, &Q_f);
    } else {
        integrate_single_particle_DEBUG(&commondata, &params, &metric, y_start, y_final);
    }
```

```c
        printf("\nDebug run finished. Trajectory saved to 'massive_particle_path.txt'.\n");

    } else {
        /*********************************/
        /*** FULL DISK PRODUCTION MODE ***/
        /*********************************/
        printf("----------------------------------------\n");
        printf("Massive Particle Integrator\n");
        printf("----------------------------------------\n");
        printf("Metric Settings:\n");
        printf("  Metric Type              = %s (a=%.2f, M=%.2f)\n", (metric.type == Kerr) ? "Kerr" : "Schwarzschild", commondata.
↪a_spin, commondata.M_scale);
        printf("\nIntegration Settings:\n");
        printf("  Max Integration Time     = %.1f M\n", commondata.t_final);
        printf("  Snapshot Every           = %.1f M\n", commondata.snapshot_every_t);
        printf("  Escape Radius            = %.1f M\n", commondata.r_escape);
        printf("\nInitial Conditions:\n");
        printf("  Generator Type           = %s\n", commondata.initial_conditions_type);
        printf("  Num Particles (r x phi) = %d x %d\n", commondata.disk_num_r, commondata.disk_num_phi);
        printf("  Disk Radial Min/Max      = %.2f / %.2f M\n", commondata.disk_r_min, commondata.disk_r_max);
        printf("----------------------------------------\n\n");

        run_mass_integrator_production(&commondata, &params, &metric);
        printf("\nProduction run finished successfully.\n");
    }

    return 0;
    """

    cfc.register_CFunction(
        includes=includes,
        desc=desc,
        cfunc_type=cfunc_type,
        name=name,
        params=params,
        body=body
    )
```

## 21 Debugging main

```
[33]: # In V1_5_mass_geodesic.ipynb, add as a NEW TEMPORARY CELL

def main_ic_validation():
    """
    Generates a TEMPORARY main() C function for the sole purpose of validating
    the output of the new, numerically stable 'calculate_ut_uphi_from_r' function.

    This main() will loop through a pre-defined set of (a, r) pairs, call the
    C function, and print the resulting u^t and u^phi to the console.
    """
    print(" -> Registering TEMPORARY main() function for IC validation...")

    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "<stdio.h>"]
    desc = r"""@brief Temporary main() for validating initial conditions.

This program tests the numerically stable 'calculate_ut_uphi_from_r' C function
by calling it for a range of black hole spins 'a' and orbital radii 'r'.
It prints the resulting 4-velocity components u^t and u^phi to the console."""
    cfunc_type = "int"
    name = "main"
    params = "int argc, const char *argv[]"

    body = r"""
// Suppress unused parameter warnings for this temporary main
(void)argc;
(void)argv;

// Initialize the commondata struct to get default M_scale
commondata_struct commondata;
params_struct params; // Dummy struct, not used but required by function signature
commondata_struct_set_to_default(&commondata);

printf("--- Initial Condition Validation for calculate_ut_uphi_from_r() ---\n");
printf("      (Using numerically stable method)\n\n");
printf("%-10s | %-10s | %-20s | %-20s\n", "a_spin", "r_initial", "u^t (dt/d_tau)", "u^phi (d_phi/d_tau)");
printf("------------------------------------------------------------------------\n");

// Define the array of test cases: {a_spin, r_initial}
const double test_cases[][2] = {
    // Prograde, high spin (a=0.99), near ISCO (~1.23M)
    {0.99, 1.5},
    {0.99, 2.0},
```

```
        {0.99, 6.0},

        // Schwarzschild (a=0.0), near ISCO (6M)
        {0.0, 6.0},
        {0.0, 8.0},
        {0.0, 20.0},

        // Retrograde, high spin (a=-0.99), near ISCO (~8.98M)
        {-0.99, 9.0},
        {-0.99, 12.0},
        {-0.99, 50.0}
    };
    const int num_test_cases = sizeof(test_cases) / sizeof(test_cases[0]);

    for (int i = 0; i < num_test_cases; ++i) {
        commondata.a_spin = test_cases[i][0];
        const double r_initial = test_cases[i][1];

        double ut, uphi;

        // Call the C function we want to test
        calculate_ut_uphi_from_r(r_initial, &commondata, &params, &ut, &uphi);

        printf("%-10.2f | %-10.2f | %-20.10f | %-20.10f\n",
               commondata.a_spin, r_initial, ut, uphi);
    }

    printf("--------------------------------------------------------------------------\n");
    printf("Validation complete. Check for NaN/inf values and physical consistency.\n");

    return 0;
"""


# This will overwrite any existing 'main' function in the dictionary,
# which is what we want for this temporary test.
cfc.register_CFunction(
    includes=includes,
    desc=desc,
    cfunc_type=cfunc_type,
    name=name,
    params=params,
    body=body
)
```

## 22 Step 8: Project Assembly and Compilation

```python
[34]: def register_custom_structures_and_params():
          """
          Generates C code for all custom structs and enums, then registers them with BHaH.
          """
          print("Registering custom C data structures for mass integrator...")

          metric_components = [f"g{nu}{mu}" for nu in range(4) for mu in range(nu, 4)]
          metric_struct_str = "typedef struct { double " + "; double ".join(metric_components) + "; } metric_struct;"

          connection_components = [f"Gamma4UDD{i}{j}{k}" for i in range(4) for j in range(4) for k in range(j, 4)]
          connections_struct_str = "typedef struct { double " + "; double ".join(connection_components) + "; } connection_struct;"

          other_structs = r"""
typedef enum { Schwarzschild, Kerr, Numerical, Schwarzschild_Standard } Metric_t;
typedef struct { Metric_t type; } metric_params;

typedef struct {
    const commondata_struct *commondata;
    const params_struct *params;
    const metric_params *metric;
} gsl_params;

// Struct for reading initial conditions from a debug file (unchanged)
typedef struct {
    int id;
    double pos[4];
    double u_spatial[3];
} particle_initial_state_t;

// DEFINITIVE struct for a single particle in the output snapshot files.
// This is the format that will be written to disk.
typedef struct {
    int id;
    double pos[3];        // x, y, z position
    double u[4];     //u^t u^x, u^y, u^z
    double lambda_rest;    // Rest-frame emission wavelength (nm)
    float j_intrinsic;     // Rest-frame intrinsic emissivity (intensity)
} __attribute__((packed)) mass_particle_state_t;
"""
          # This line needs to be updated to include the new struct string
          Bdefines_h.register_BHaH_defines("data_structures", f"{metric_struct_str}\n{connections_struct_str}\n{other_structs}")
          print(" -> Registered all necessary data structures, including mass_particle_state_t.")
```

### 8.b: Final Build Command (Redo)

```
[35]:  # In V1_1_mass_geodesic.ipynb, Cell ID a0eb212d (Final Build Script)
       print("\nAssembling and building C project for the massive particle integrator...")
       os.makedirs(project_dir, exist_ok=True)

       # --- Step 1: Register all C-generating functions in the correct order ---
       print(" -> Registering C data structures and functions...")
       register_custom_structures_and_params()

       # Register symbolic recipes and C-generating functions for physics
       # symbolic_ut_uphi_from_r() is called by the next function
       calculate_ut_uphi_from_r() # The new, required helper function

       # Register C workers for metrics and connections
       g4DD_kerr_schild(); con_kerr_schild()
       g4DD_schwarzschild_cartesian(); con_schwarzschild_cartesian()

       # Register C dispatchers
       g4DD_metric(); connections()

       # Register C engines for the core logic
       calculate_ode_rhs_massive()
       ode_gsl_wrapper_massive()
       set_initial_conditions_massive()
       check_conservation_massive()
       integrate_single_particle()
       integrate_single_particle_DEBUG()
       run_mass_integrator_production()

       # Register the production-run orchestrators
       generate_disk_initial_conditions()
       generate_spiral_galaxy_initial_conditions()
       generate_barred_flocculent_spiral_ic()
       main()
       #main_ic_validation()


       # --- Step 2: Call BHaH infrastructure functions to generate the build system ---
       print(" -> Generating BHaH infrastructure files...")
       # Generate set_CodeParameters.h and its variants
       CPs.write_CodeParameters_h_files(project_dir=project_dir)
       # Register C functions to set parameters to default values
       CPs.register_CFunctions_params_commondata_struct_set_to_default()
       # Generate the default parameter file (mass_integrator.par)
```

```python
cmdline_input_and_parfiles.generate_default_parfile(project_dir=project_dir, project_name=project_name)

# Register the C function that parses the command line and parameter file
cmdline_input_and_parfiles.register_CFunction_cmdline_input_and_parfile_parser(
    project_name=project_name,
    cmdline_inputs=[
        'M_scale', 'a_spin', 't_max_integration', 'flatness_threshold',
        'r_escape', 'ut_max', 'perform_conservation_check', 'run_in_debug_mode',
        'initial_conditions_type',
        'spiral_galaxy_num_arms',
        'spiral_galaxy_arm_tightness',
        'bar_length', 'bar_aspect_ratio', 'bulge_radius',
        'arm_particle_density', 'arm_clumpiness_factor', 'arm_clump_size',
        'bar_density_factor', 'bulge_density_factor'
    ]
)


# --- Step 3: Generate the final C code, headers, and Makefile ---
print("\nGenerating BHaH master header file (BHaH_defines.h)...")
Bdefines_h.output_BHaH_defines_h(project_dir=project_dir)

# Note: SIMD intrinsics are not used in this project, but the helper is harmless.
print("Copying required helper files...")
gh.copy_files(
    package="nrpy.helpers",
    filenames_list=["simd_intrinsics.h"],
    project_dir=project_dir,
    subdirectory="simd",
)

print("Generating all C source files, function prototypes, and the Makefile...")
# Add required GSL and OpenMP flags to the compiler
addl_CFLAGS = ["-Wall -Wextra -g $(shell gsl-config --cflags) -fopenmp"]
addl_libraries = ["$(shell gsl-config --libs) -fopenmp"]

Makefile.output_CFunctions_function_prototypes_and_construct_Makefile(
    project_dir=project_dir,
    project_name=project_name,
    exec_or_library_name="mass_integrator", # The name of our final executable
    addl_CFLAGS=addl_CFLAGS,
    addl_libraries=addl_libraries,
)


print(f"\nFinished! A C project has been generated in '{project_dir}/'")
```

```
print(f"To build, navigate to this directory in your terminal and type 'make'.")
print(f"To run, type './mass_integrator'.")
```

Assembling and building C project for the massive particle integrator...
 -> Registering C data structures and functions...
Registering custom C data structures for mass integrator...
 -> Registered all necessary data structures, including mass_particle_state_t.
 -> Generating C engine: calculate_ut_uphi_from_r() [STABLE VERSION]...
    ... calculate_ut_uphi_from_r() registration complete.
 -> Generating C worker function: g4DD_kerr_schild()...
    ... g4DD_kerr_schild() registration complete.
 -> Generating C worker function: con_kerr_schild()...
    ... con_kerr_schild() registration complete.
 -> Generating C worker function: g4DD_schwarzschild_cartesian()...
    ... g4DD_schwarzschild_cartesian() registration complete.
 -> Generating C worker function: con_schwarzschild_cartesian()...
    ... con_schwarzschild_cartesian() registration complete.
 -> Generating C dispatcher function: g4DD_metric()...
    ... g4DD_metric() registration complete.
 -> Generating C dispatcher: connections()...
    ... connections() registration complete.
 -> Generating C engine: set_initial_conditions_massive() [STABLE VERSION]...
 -> Generating C engine: check_conservation_massive()...
    ... check_conservation_massive() registration complete.
 -> Generating C orchestrator: run_mass_integrator_production() [Dispatcher
Version]...
 -> Generating C function: generate_spiral_galaxy_initial_conditions()...
    ... generate_spiral_galaxy_initial_conditions() registration complete.
 -> Generating C function: generate_barred_flocculent_spiral_ic()...
 -> Generating C entry point: main() [Final Version]...
 -> Generating BHaH infrastructure files...

Generating BHaH master header file (BHaH_defines.h)...
Outputting non-core modules key = data_structures to BHaH_defines.h
Copying required helper files...
Generating all C source files, function prototypes, and the Makefile...

Finished! A C project has been generated in 'project/mass_integrator/'
To build, navigate to this directory in your terminal and type 'make'.
To run, type './mass_integrator'.

## 23  Making the kd trees

```python
[36]: import numpy as np
import os
import glob
from collections import import deque
import numba # Make sure to import numba


# ================================================================================
#  STEP 1: Create the JIT-compiled "kernel" for the heavy lifting.
# ================================================================================
# Numba works best with simple data types, so we pass NumPy arrays.
# The @numba.jit decorator is the key. 'nopython=True' ensures it compiles to fast machine code.
@numba.jit(nopython=True)
def build_implicit_kdtree_kernel(particle_pos_array, num_particles):
    """
    Numba-compiled kernel to perform the heavy lifting of building the implicit k-d tree.
    This function contains only the numerical logic that Numba can optimize.

    Args:
        particle_pos_array (np.ndarray): A NumPy array of shape (N, 3) with just particle positions.
        num_particles (int): The number of particles.

    Returns:
        tuple: (reordering_indices, node_metadata)
                - reordering_indices: An array that maps the old index to the new, reordered index.
                - node_metadata: The array of split axes for each node.
    """
    # These will store the final results
    reordering_indices = np.zeros(num_particles, dtype=np.int32)
    node_metadata = np.full(num_particles, -1, dtype=np.int32)

    # Numba doesn't have deque, but we can simulate a queue with a NumPy array and pointers.
    # This is a common pattern for JIT-compiling queue-based algorithms.
    # We need to pre-allocate a large enough queue. num_particles is a safe upper bound.
    q_target_idx = np.zeros(num_particles, dtype=np.int32)
    # This queue will store slices (start, end) into a temporary index array
    q_slice_start = np.zeros(num_particles, dtype=np.int32)
    q_slice_end = np.zeros(num_particles, dtype=np.int32)

    q_head = 0
    q_tail = 0


    # The array of original indices that we will sort and partition
```

```python
    indices_array = np.arange(num_particles, dtype=np.int32)

    if num_particles > 0:
        # Enqueue the first item: target index 0, and the slice representing all particles
        q_target_idx[q_tail] = 0
        q_slice_start[q_tail] = 0
        q_slice_end[q_tail] = num_particles
        q_tail += 1

    while q_head < q_tail:
        # Dequeue an item
        target_idx = q_target_idx[q_head]
        start = q_slice_start[q_head]
        end = q_slice_end[q_head]
        q_head += 1

        # --- Adaptive Splitting Logic ---
        # Note: Slicing is faster inside Numba than fancy indexing
        current_indices = indices_array[start:end]
        current_positions = particle_pos_array[current_indices]

        min_x, min_y, min_z = current_positions[0]
        max_x, max_y, max_z = current_positions[0]
        for i in range(1, len(current_positions)):
            pos = current_positions[i]
            min_x, max_x = min(min_x, pos[0]), max(max_x, pos[0])
            min_y, max_y = min(min_y, pos[1]), max(max_y, pos[1])
            min_z, max_z = min(min_z, pos[2]), max(max_z, pos[2])

        spread_x = max_x - min_x
        spread_y = max_y - min_y
        spread_z = max_z - min_z

        # Find split axis (0=x, 1=y, 2=z)
        split_axis = 0
        if spread_y > spread_x: split_axis = 1
        if spread_z > spread_y and spread_z > spread_x: split_axis = 2

        # --- Partitioning ---
        # Sort the current slice of the main index array based on the split axis
        # This is the most performance-critical part
        sorted_indices_on_axis = current_indices[np.argsort(particle_pos_array[current_indices, split_axis])]
        indices_array[start:end] = sorted_indices_on_axis
```

```python
            median_offset = len(sorted_indices_on_axis) // 2
            median_original_index = sorted_indices_on_axis[median_offset]

            # --- Place the Pivot Particle's Metadata ---
            reordering_indices[target_idx] = median_original_index
            node_metadata[target_idx] = split_axis

            # --- Prepare for Next Level (Enqueue children) ---
            left_child_idx = 2 * target_idx + 1
            right_child_idx = 2 * target_idx + 2

            # Left child points to the slice before the median
            if median_offset > 0:
                if left_child_idx < num_particles:
                    q_target_idx[q_tail] = left_child_idx
                    q_slice_start[q_tail] = start
                    q_slice_end[q_tail] = start + median_offset
                    q_tail += 1

            # Right child points to the slice after the median
            if median_offset + 1 < len(sorted_indices_on_axis):
                if right_child_idx < num_particles:
                    q_target_idx[q_tail] = right_child_idx
                    q_slice_start[q_tail] = start + median_offset + 1
                    q_slice_end[q_tail] = end
                    q_tail += 1

    return reordering_indices, node_metadata


# ==============================================================================
#  STEP 2: Create the main driver function that calls the kernel.
# ==============================================================================
def build_and_save_kdtree_snapshot_fast(raw_snapshot_file, output_dir):
    """
    Main driver function that handles file I/O and calls the fast Numba kernel.
    """
    raw_dtype = np.dtype([
        ('id', np.int32), ('pos', 'f8', (3,)), ('u', 'f8', (4,)),
        ('lambda_rest', 'f8'), ('j_intrinsic', 'f4')
    ])

    # --- Step 1: Load the Raw Data (Fast I/O) ---
    with open(raw_snapshot_file, 'rb') as f:
        num_particles = np.fromfile(f, dtype=np.int32, count=1)[0]
```

```
        if num_particles == 0:
            print("   Snapshot is empty, skipping.")
            return
        particle_data = np.fromfile(f, dtype=raw_dtype, count=num_particles)

    # --- Step 2: Call the Fast JIT-Compiled Kernel ---
    # We only pass the position data to the kernel for efficiency.
    particle_positions = np.ascontiguousarray(particle_data['pos'])
    reordering_map, node_metadata = build_implicit_kdtree_kernel(particle_positions, num_particles)

    # Use the returned map to reorder the full, original particle data array
    reordered_particles = particle_data[reordering_map]

    # --- Step 3: Save the Two Parallel Arrays (Fast I/O) ---
    output_filename = os.path.join(output_dir, os.path.basename(raw_snapshot_file).replace('.bin', '.kdtree.bin'))
    with open(output_filename, 'wb') as f:
        f.write(np.uint64(num_particles))
        f.write(np.uint64(3)) # dimensions
        f.write(node_metadata.tobytes())
        f.write(reordered_particles.tobytes())
```

```
[37]:  import os
       import glob
       import time
       from multiprocessing import Pool, cpu_count
       from functools import partial
       from tqdm import tqdm # For a nice progress bar! (pip install tqdm)

       # --- Place the TWO functions from Part 1 above this ---
       # (build_implicit_kdtree_kernel and build_and_save_kdtree_snapshot_fast)

       def run_kdtree_preprocessor_parallel():
           """
           Finds all raw snapshot .bin files and processes them in parallel
           using a pool of worker processes.
           """
           base_project_dir = "project"
           mass_project_name = "mass_integrator"
           input_folder = os.path.join(base_project_dir, mass_project_name, "output")
           processed_folder = os.path.join(base_project_dir, "processed_snapshots")

           print(f"--- Starting K-d Tree Pre-processing (Parallel) ---")
           print(f"Input directory:  '{input_folder}'")
           print(f"Shared Output directory: '{processed_folder}'")
```

```python
        os.makedirs(processed_folder, exist_ok=True)

        snapshot_files = sorted(glob.glob(os.path.join(input_folder, "mass_blueprint_t_*.bin")))

        if not snapshot_files:
            print("\nWARNING: No raw snapshot files found. Did you run the mass_integrator C code first?")
            return

        num_processes = cpu_count()
        print(f"\nFound {len(snapshot_files)} files. Processing in parallel using {num_processes} CPU cores...")

        task_function = partial(build_and_save_kdtree_snapshot_fast, output_dir=processed_folder)

        start_time = time.time()
        with Pool(processes=num_processes) as pool:
            list(tqdm(pool.imap_unordered(task_function, snapshot_files), total=len(snapshot_files)))

        end_time = time.time()

        print(f"\n--- All snapshots processed successfully in {end_time - start_time:.2f} seconds. ---")
        print(f"The query-ready .kdtree.bin files are now in the shared directory: '{processed_folder}'.")


# --- How to Run ---
# It's best practice to put the execution call inside this block
if __name__ == '__main__':
    # Make sure to call the parallel version!
    run_kdtree_preprocessor_parallel()
```

```
--- Starting K-d Tree Pre-processing (Parallel) ---
Input directory:  'project/mass_integrator/output'
Shared Output directory: 'project/processed_snapshots'

WARNING: No raw snapshot files found. Did you run the mass_integrator C code
first?
```

```python
[38]: import numpy as np
import os
import glob

def view_kdtree_snapshot(
    project_dir: str = "project/mass_integrator",
    processed_folder: str = "processed_snapshots",
    snapshot_index: int = 0, # 0 for the first, -1 for the last
```

```python
    max_nodes_to_print: int = 15
) -> None:
    """
    Reads a single, processed .kdtree.bin file and prints a detailed,
    human-readable summary to verify its contents and structure.

    UPDATED to read and display the full 4-velocity u^mu.
    """
    print("--- K-d Tree Blueprint Inspector ---")

    # --- Step 1: Find and select the snapshot file ---
    snapshot_dir = os.path.join(project_dir, processed_folder)
    if not os.path.isdir(snapshot_dir):
        print(f"ERROR: Processed snapshot directory not found at '{snapshot_dir}'")
        return

    snapshot_files = sorted(glob.glob(os.path.join(snapshot_dir, "*.kdtree.bin")))
    if not snapshot_files:
        print(f"ERROR: No .kdtree.bin files found in '{snapshot_dir}'")
        print("Please run the k-d tree pre-processor cell first.")
        return

    try:
        file_to_load = snapshot_files[snapshot_index]
    except IndexError:
        print(f"ERROR: Snapshot index {snapshot_index} is out of bounds. Only {len(snapshot_files)} files exist.")
        return

    print(f"Loading and inspecting file: '{os.path.basename(file_to_load)}'")

    # --- Step 2: Define the dtypes to read the file ---
    # MODIFICATION: This dtype must exactly match the new particle data struct in C
    particle_dtype = np.dtype([
        ('id', np.int32),
        ('pos', 'f8', (3,)),
        ('u', 'f8', (4,)), # Changed from 'u_spatial' to 'u'
        ('lambda_rest', 'f8'),
        ('j_intrinsic', 'f4')
    ])
    metadata_dtype = np.int32

    # --- Step 3: Read the binary file according to the custom format ---
    try:
        with open(file_to_load, 'rb') as f:
```

```python
                num_particles = np.fromfile(f, dtype=np.uint64, count=1)[0]
                dimensions = np.fromfile(f, dtype=np.uint64, count=1)[0]
                node_metadata = np.fromfile(f, dtype=metadata_dtype, count=num_particles)
                particle_data = np.fromfile(f, dtype=particle_dtype, count=num_particles)
    except Exception as e:
        print(f"\nERROR: Failed to read or parse the binary file. Exception: {e}")
        return

    # --- Step 4: Print a summary and verify the contents ---
    print("\n--- File Header ---")
    print(f"  Number of Particles: {num_particles}")
    print(f"  Dimensions:          {dimensions}")

    if len(node_metadata) != num_particles or len(particle_data) != num_particles:
        print("\nERROR: Mismatch between header particle count and data array lengths!")
        return

    print("\n--- Data Verification ---")

    axis_map = {0: 'X', 1: 'Y', 2: 'Z', -1: 'LEAF'}

    # MODIFICATION: Updated header to show the full 4-velocity
    header = (f"{'Index':<6} | {'Split Axis':<10} | {'Particle ID':<12} | "
              f"{'Position (x, y, z)':<25} | {'4-Velocity (ut, ux, uy, uz)':<40}")
    print(header)
    print("-" * len(header))

    for i in range(min(num_particles, max_nodes_to_print)):
        split_axis_val = node_metadata[i]
        particle = particle_data[i]

        split_axis_str = axis_map.get(split_axis_val, 'INVALID')
        pos_str = (f"({particle['pos'][0]:>6.2f}, {particle['pos'][1]:>6.2f}, "
                   f"{particle['pos'][2]:>6.2f})")

        # MODIFICATION: Format the full 4-velocity for display
        vel_str = (f"({particle['u'][0]:>6.2f}, {particle['u'][1]:>6.2f}, "
                   f"{particle['u'][2]:>6.2f}, {particle['u'][3]:>6.2f})")

        print(f"{i:<6} | {split_axis_str:<10} | {particle['id']:<12} | "
              f"{pos_str:<25} | {vel_str:<40}")

    if num_particles > max_nodes_to_print:
        print("...")
```

```
        print(f"(... and {num_particles - max_nodes_to_print} more nodes)")

    print("\n--- Verification Complete ---")
    print("Check that the 4-Velocity column contains four components and looks reasonable.")
view_kdtree_snapshot()
```

```
--- K-d Tree Blueprint Inspector ---
ERROR: Processed snapshot directory not found at
'project/mass_integrator/processed_snapshots'
```

[39]:
```python
import numpy as np
import matplotlib.pyplot as plt
import os

def plot_particle_trajectory(
    project_dir: str = "project/mass_integrator",
    input_filename: str = "massive_particle_path.txt",
    M_scale: float = 1.0,
    a_spin: float = 0.9
) -> None:
    """
    Reads the trajectory data from the C code's output file and generates
    a 3D plot of the particle's orbit around the black hole.

    Args:
        project_dir: The root directory of the C project where the output file is located.
        input_filename: The name of the trajectory data file.
        M_scale: The mass of the black hole, used to plot the event horizon.
        a_spin: The spin of the black hole, used to plot the event horizon.
    """
    print("--- Generating Particle Trajectory Plot ---")

    # --- 1. Construct the full path and load the data ---
    full_path = os.path.join(project_dir, input_filename)

    if not os.path.exists(full_path):
        print(f"ERROR: Trajectory file not found at '{full_path}'")
        print("Please ensure you have compiled and run the C code successfully.")
        return

    try:
        # Load the data, skipping the header row
        data = np.loadtxt(full_path, skiprows=1)
        # Columns: 0:τ, 1:t, 2:x, 3:y, 4:z, 5:u^t, 6:u^x, 7:u^y, 8:u^z
        x_coords = data[:, 2]
```

```python
        y_coords = data[:, 3]
        z_coords = data[:, 4]
        print(f"Successfully loaded {len(x_coords)} data points from trajectory file.")
    except Exception as e:
        print(f"ERROR: Failed to load or parse the data file '{full_path}'.")
        print(f"Exception: {e}")
        return

    # --- 2. Set up the 3D plot ---
    plt.style.use('dark_background')
    fig = plt.figure(figsize=(12, 10))
    ax = fig.add_subplot(111, projection='3d')

    # --- 3. Plot the particle's trajectory ---
    ax.plot(x_coords, y_coords, z_coords, label='Particle Orbit', color='cyan', lw=2)

    # Mark the start and end points
    ax.scatter(x_coords[0], y_coords[0], z_coords[0], color='lime', s=100, label='Start', marker='o')
    ax.scatter(x_coords[-1], y_coords[-1], z_coords[-1], color='red', s=100, label='End', marker='X')

    # --- 4. Plot the black hole's event horizon ---
    # The radius of the event horizon for a Kerr black hole
    r_horizon = M_scale * (1 + np.sqrt(1 - a_spin**2))

    # Create a sphere for the event horizon
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)
    x_bh = r_horizon * np.outer(np.cos(u), np.sin(v))
    y_bh = r_horizon * np.outer(np.sin(u), np.sin(v))
    z_bh = r_horizon * np.outer(np.ones(np.size(u)), np.cos(v))

    ax.plot_surface(x_bh, y_bh, z_bh, color='black', alpha=0.9, rstride=4, cstride=4)
    # Add a grey wireframe for better visibility
    ax.plot_wireframe(x_bh, y_bh, z_bh, color='dimgrey', alpha=0.2, rstride=10, cstride=10)

    # --- 5. Customize the plot ---
    ax.set_xlabel('X (M)', fontsize=12, labelpad=10)
    ax.set_ylabel('Y (M)', fontsize=12, labelpad=10)
    ax.set_zlabel('Z (M)', fontsize=12, labelpad=10)

    # Set equal aspect ratio
    max_range = np.array([x_coords.max()-x_coords.min(), y_coords.max()-y_coords.min(), z_coords.max()-z_coords.min()]).max() / 2.0
    mid_x = (x_coords.max()+x_coords.min()) * 0.5
    mid_y = (y_coords.max()+y_coords.min()) * 0.5
```

```python
        mid_z = (z_coords.max()+z_coords.min()) * 0.5
        ax.set_xlim(mid_x - max_range, mid_x + max_range)
        ax.set_ylim(mid_y - max_range, mid_y + max_range)
        ax.set_zlim(mid_z - max_range, mid_z + max_range)

        ax.set_title(f"Massive Particle Trajectory (M={M_scale}, a={a_spin})", fontsize=16)
        ax.legend()
        ax.view_init(elev=30., azim=45) # Set a nice viewing angle

        plt.show()
```

```python
[40]: import numpy as np
      import matplotlib.pyplot as plt
      import os


      def plot_trajectory_components(
          project_dir: str = "project/mass_integrator",
          input_filename: str = "massive_particle_path.txt"
      ) -> None:
          """
          Reads trajectory data and creates four plots:
          x vs t, y vs t, z vs t, and proper time (τ) vs t.
          """
          print("--- Generating Trajectory Component Plots ---")

          # --- 1. Load the data ---
          full_path = os.path.join(project_dir, input_filename)

          if not os.path.exists(full_path):
              print(f"ERROR: Trajectory file not found at '{full_path}'")
              return

          try:
              data = np.loadtxt(full_path, skiprows=1)
              # Columns: 0:τ, 1:t, 2:x, 3:y, 4:z, ...
              proper_time = data[:, 0]
              coord_time = data[:, 1]
              x_coords = data[:, 2]
              y_coords = data[:, 3]
              z_coords = data[:, 4]
              print(f"Successfully loaded {len(coord_time)} data points.")
          except Exception as e:
              print(f"ERROR: Failed to load or parse data file '{full_path}'. Exception: {e}")
              return
```

```python
# --- 2. Create the plots ---
plt.style.use('seaborn-v0_8-whitegrid')
fig, axes = plt.subplots(2, 2, figsize=(16, 12))
fig.suptitle('Particle Trajectory Components vs. Coordinate Time', fontsize=20)

# Plot 1: x(t)
axes[0, 0].plot(coord_time, x_coords, color='cyan')
axes[0, 0].set_title('X Coordinate vs. Time', fontsize=14)
axes[0, 0].set_xlabel('Coordinate Time (t) [M]', fontsize=12)
axes[0, 0].set_ylabel('x [M]', fontsize=12)
axes[0, 0].grid(True)

# Plot 2: y(t)
axes[0, 1].plot(coord_time, y_coords, color='magenta')
axes[0, 1].set_title('Y Coordinate vs. Time', fontsize=14)
axes[0, 1].set_xlabel('Coordinate Time (t) [M]', fontsize=12)
axes[0, 1].set_ylabel('y [M]', fontsize=12)
axes[0, 1].grid(True)

# Plot 3: z(t)
axes[1, 0].plot(coord_time, z_coords, color='lime')
axes[1, 0].set_title('Z Coordinate vs. Time', fontsize=14)
axes[1, 0].set_xlabel('Coordinate Time (t) [M]', fontsize=12)
axes[1, 0].set_ylabel('z [M]', fontsize=12)
axes[1, 0].grid(True)

# Plot 4: τ(t)
axes[1, 1].plot(coord_time, proper_time, color='gold')
axes[1, 1].set_title('Proper Time vs. Coordinate Time', fontsize=14)
axes[1, 1].set_xlabel('Coordinate Time (t) [M]', fontsize=12)
axes[1, 1].set_ylabel('Proper Time (τ) [M]', fontsize=12)
axes[1, 1].grid(True)

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

```python
[41]: import numpy as np
import matplotlib.pyplot as plt
import os

def plot_radius_vs_time(
    project_dir: str = "project/mass_integrator",
    input_filename: str = "massive_particle_path.txt"
```

```python
) -> None:
    """
    Reads trajectory data and plots the particle's radial distance (r)
    as a function of coordinate time (t) to validate circularity.
    """
    print("--- Generating Radius vs. Time Validation Plot ---")

    full_path = os.path.join(project_dir, input_filename)
    if not os.path.exists(full_path):
        print(f"ERROR: Trajectory file not found at '{full_path}'")
        return

    try:
        data = np.loadtxt(full_path, skiprows=1)
        coord_time = data[:, 1]
        x_coords = data[:, 2]
        y_coords = data[:, 3]
        z_coords = data[:, 4]
    except Exception as e:
        print(f"ERROR: Failed to load data. Exception: {e}")
        return

    # Calculate the radius at each time step
    radius = np.sqrt(x_coords**2 + y_coords**2 + z_coords**2)

    # Calculate statistics on the radius
    mean_radius = np.mean(radius)
    min_radius = np.min(radius)
    max_radius = np.max(radius)
    percent_variation = 100 * (max_radius - min_radius) / mean_radius

    print(f"Radius Statistics:")
    print(f"  Mean Radius: {mean_radius:.6f} M")
    print(f"  Min Radius:  {min_radius:.6f} M")
    print(f"  Max Radius:  {max_radius:.6f} M")
    print(f"  Total Variation: {percent_variation:.4f}%")

    # Create the plot
    plt.style.use('seaborn-v0_8-whitegrid')
    plt.figure(figsize=(12, 6))

    plt.plot(coord_time, radius, label='Particle Radius r(t)', color='cyan')

    # Add lines for mean, min, and max to visualize the variation
```

```python
    plt.axhline(mean_radius, color='lime', linestyle='--', label=f'Mean r = {mean_radius:.4f}')

    plt.title('Validation: Particle Radius vs. Coordinate Time', fontsize=16)
    plt.xlabel('Coordinate Time (t) [M]', fontsize=12)
    plt.ylabel('Radius (r) [M]', fontsize=12)
    plt.legend()
    plt.grid(True)

    # Use a "tight" y-axis to emphasize any small variations
    plt.ylim(min_radius * 0.999, max_radius * 1.001)

    plt.show()
```

```python
import numpy as np
import matplotlib.pyplot as plt
import os
from scipy.stats import linregress

def plot_precession_validation(
    project_dir: str = "project/mass_integrator",
    input_filename: str = "massive_particle_path.txt",
    M_scale: float = 1.0,
    a_spin: float = 0.9,
    r_initial: float = 10.0
) -> None:
    """
    Reads trajectory data and validates the orbital precession rate against
    the theoretical Lense-Thirring formula.
    """
    print("--- Generating Precession Validation Plot ---")

    full_path = os.path.join(project_dir, input_filename)
    if not os.path.exists(full_path):
        print(f"ERROR: Trajectory file not found at '{full_path}'")
        return

    try:
        data = np.loadtxt(full_path, skiprows=1)
        coord_time = data[:, 1]
        x_coords = data[:, 2]
        y_coords = data[:, 3]
    except Exception as e:
        print(f"ERROR: Failed to load data. Exception: {e}")
        return
```

```python
    # Calculate the azimuthal angle phi at each time step
    phi = np.arctan2(y_coords, x_coords)

    # The angle will wrap around from +pi to -pi. We need to unwrap it.
    phi_unwrapped = np.unwrap(phi)

    # --- Theoretical Calculation ---
    r = r_initial
    Omega_K = (M_scale**0.5) / (r**1.5 + a_spin * M_scale**0.5)
    Omega_LT = (2 * M_scale * a_spin) / (r**3)
    Omega_phi_theory = Omega_K + Omega_LT

    # --- Measurement from Simulation Data ---
    # Perform a linear regression to find the slope of phi(t)
    regression = linregress(coord_time, phi_unwrapped)
    Omega_phi_measured = regression.slope

    percent_error = 100 * abs(Omega_phi_measured - Omega_phi_theory) / Omega_phi_theory

    print("Precession Rate (dφ/dt) Validation:")
    print(f"  Theoretical Ω_φ: {Omega_phi_theory:.6f} rad/M")
    print(f"  Measured Ω_φ (from data): {Omega_phi_measured:.6f} rad/M")
    print(f"  Relative Error: {percent_error:.4f}%")

    # --- Create the Plot ---
    plt.style.use('seaborn-v0_8-whitegrid')
    plt.figure(figsize=(12, 6))

    plt.plot(coord_time, phi_unwrapped, label='Measured φ(t) from Simulation', color='cyan', lw=2)
    plt.plot(coord_time, Omega_phi_theory * coord_time, label=f'Theoretical φ(t) (slope={Omega_phi_theory:.4f})', color='lime',
    →linestyle='--', lw=2)

    plt.title('Validation: Orbital Precession (Frame-Dragging)', fontsize=16)
    plt.xlabel('Coordinate Time (t) [M]', fontsize=12)
    plt.ylabel('Azimuthal Angle (φ) [radians]', fontsize=12)
    plt.legend()
    plt.grid(True)

    plt.show()
```

```python
[43]: import numpy as np
      import matplotlib.pyplot as plt
      import os
```

```python
import glob

def visualize_disk_snapshot(
    project_dir: str = "project/mass_integrator",
    output_folder: str = "output",
    snapshot_index: int = -1, # -1 means the last available snapshot
    M_scale: float = 1.0,
    a_spin: float = 0.9
) -> None:
    """
    Reads a specific mass blueprint snapshot file, performs sanity checks,
    and generates a 3D plot of the particle disk.
    """
    print("--- Visualizing Mass Blueprint Snapshot ---")

    # --- 1. Find and Load the Snapshot File ---
    snapshot_dir = os.path.join(project_dir, output_folder)
    if not os.path.isdir(snapshot_dir):
        print(f"ERROR: Snapshot directory not found at '{snapshot_dir}'")
        return

    # Find all blueprint files and sort them
    snapshot_files = sorted(glob.glob(os.path.join(snapshot_dir, "mass_blueprint_t_*.bin")))

    if not snapshot_files:
        print(f"ERROR: No snapshot .bin files found in '{snapshot_dir}'")
        return

    if snapshot_index == -1:
        # Select the last file
        snapshot_to_load = snapshot_files[-1]
    elif snapshot_index < len(snapshot_files):
        snapshot_to_load = snapshot_files[snapshot_index]
    else:
        print(f"ERROR: Snapshot index {snapshot_index} is out of bounds. Only {len(snapshot_files)} snapshots exist.")
        return

    print(f"Loading snapshot file: '{snapshot_to_load}'")

    # Define the data type for a single record in the binary file
    # Format: int (id), double (x), double (y), double (z), double (ux), double (uy), double (uz)
    snapshot_dtype = np.dtype([
        ('id', np.int32),
        ('pos', 'f8', (3,)), # 3 doubles for position
```

```python
        ('u_spatial', 'f8', (3,)) # 3 doubles for spatial 4-velocity
    ])

    try:
        data = np.fromfile(snapshot_to_load, dtype=snapshot_dtype)
        num_particles = len(data)
        print(f"Successfully loaded data for {num_particles} particles.")
    except Exception as e:
        print(f"ERROR: Failed to load or parse the data file '{snapshot_to_load}'.")
        print(f"Exception: {e}")
        return

    # --- 2. Perform Sanity Checks ---
    positions = data['pos']
    velocities = data['u_spatial']

    radii = np.sqrt(positions[:, 0]**2 + positions[:, 1]**2)
    speeds = np.sqrt(velocities[:, 0]**2 + velocities[:, 1]**2 + velocities[:, 2]**2)

    print("\n--- Data Sanity Checks ---")
    print(f"  Particle count: {num_particles}")
    print(f"  Mean radius: {np.mean(radii):.3f} M (should be between disk_r_min and disk_r_max)")
    print(f"  Mean spatial 4-velocity magnitude: {np.mean(speeds):.3f}")
    print(f"  Max z-coordinate: {np.max(np.abs(positions[:, 2])):.2e} (should be close to zero)")

    if np.any(np.isnan(positions)) or np.any(np.isnan(velocities)):
        nan_count = np.count_nonzero(np.isnan(data['pos'][:,0]))
        print(f"  WARNING: Found {nan_count} terminated (NaN) particles in this snapshot.")

    # --- 3. Create the 3D Plot ---
    plt.style.use('dark_background')
    fig = plt.figure(figsize=(12, 10))
    ax = fig.add_subplot(111, projection='3d')

    # Plot a subset of particles to avoid cluttering the plot
    num_to_plot = min(num_particles, 2000)
    plot_indices = np.random.choice(num_particles, num_to_plot, replace=False)

    # Use color to represent the radial position of the particles
    colors = radii[plot_indices]
    sc = ax.scatter(positions[plot_indices, 0], positions[plot_indices, 1], positions[plot_indices, 2],
                    c=colors, cmap='plasma', s=5, label='Disk Particles')

    # --- Plot the black hole's event horizon ---
```

```
    r_horizon = M_scale * (1 + np.sqrt(1 - a_spin**2))
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)
    x_bh = r_horizon * np.outer(np.cos(u), np.sin(v))
    y_bh = r_horizon * np.outer(np.sin(u), np.sin(v))
    z_bh = r_horizon * np.outer(np.ones(np.size(u)), np.cos(v))
    ax.plot_surface(x_bh, y_bh, z_bh, color='black', alpha=0.9, rstride=4, cstride=4)
    ax.plot_wireframe(x_bh, y_bh, z_bh, color='dimgrey', alpha=0.2, rstride=10, cstride=10)

    # --- Customize the plot ---
    ax.set_xlabel('X (M)', fontsize=12, labelpad=10)
    ax.set_ylabel('Y (M)', fontsize=12, labelpad=10)
    ax.set_zlabel('Z (M)', fontsize=12, labelpad=10)

    max_radius_plot = np.max(radii) * 1.1
    ax.set_xlim(-max_radius_plot, max_radius_plot)
    ax.set_ylim(-max_radius_plot, max_radius_plot)
    ax.set_zlim(-max_radius_plot/2, max_radius_plot/2) # Exaggerate z-axis if needed

    ax.set_title(f"Accretion Disk Snapshot from {os.path.basename(snapshot_to_load)}", fontsize=16)
    fig.colorbar(sc, ax=ax, shrink=0.6, aspect=10, label='Particle Radius (M)')
    ax.view_init(elev=45., azim=45)

    plt.show()
```

```
[44]: import numpy as np
import matplotlib.pyplot as plt
import os
import glob


def visualize_disk_snapshot(
    project_dir: str = "project/mass_integrator",
    output_folder: str = "output",
    snapshot_index: int = -1, # -1 means the last available snapshot
    M_scale: float = 1.0,
    a_spin: float = 0.9
) -> None:
    """
    Reads a specific mass blueprint snapshot file, performs sanity checks,
    and generates a 3D plot of the particle disk.

    UPDATED to read the new binary format with the full 4-velocity.
    """
    print("--- Visualizing Mass Blueprint Snapshot ---")
```

```python
# --- 1. Find and Load the Snapshot File ---
snapshot_dir = os.path.join(project_dir, output_folder)
if not os.path.isdir(snapshot_dir):
    print(f"ERROR: Snapshot directory not found at '{snapshot_dir}'")
    return

snapshot_files = sorted(glob.glob(os.path.join(snapshot_dir, "mass_blueprint_t_*.bin")))

if not snapshot_files:
    print(f"ERROR: No snapshot .bin files found in '{snapshot_dir}'")
    return

if snapshot_index == -1:
    snapshot_to_load = snapshot_files[-1]
elif snapshot_index < len(snapshot_files):
    snapshot_to_load = snapshot_files[snapshot_index]
else:
    print(f"ERROR: Snapshot index {snapshot_index} is out of bounds. Only {len(snapshot_files)} snapshots exist.")
    return

print(f"Loading snapshot file: '{snapshot_to_load}'")

# MODIFICATION: Define the dtype to match the new C struct with u[4].
snapshot_dtype = np.dtype([
    ('id', np.int32),
    ('pos', 'f8', (3,)),
    ('u', 'f8', (4,)),    # Changed from ('u_spatial', 'f8', (3,))
    ('lambda_rest', 'f8'),
    ('j_intrinsic', 'f4')
])

try:
    # The header is a 4-byte int, not part of the dtype
    with open(snapshot_to_load, 'rb') as f:
        num_particles = np.fromfile(f, dtype=np.int32, count=1)[0]
        data = np.fromfile(f, dtype=snapshot_dtype, count=num_particles)
    print(f"Successfully loaded data for {num_particles} particles.")
except Exception as e:
    print(f"ERROR: Failed to load or parse the data file '{snapshot_to_load}'.")
    print(f"Exception: {e}")
    return

# --- 2. Perform Sanity Checks ---
```

```python
positions = data['pos']
# MODIFICATION: Use the new 'u' field for velocities.
velocities = data['u']

radii = np.sqrt(positions[:, 0]**2 + positions[:, 1]**2)
# Calculate speed from the SPATIAL components of the 4-velocity
speeds = np.sqrt(velocities[:, 1]**2 + velocities[:, 2]**2 + velocities[:, 3]**2)

print("\n--- Data Sanity Checks ---")
print(f"  Particle count: {num_particles}")
print(f"  Mean radius: {np.mean(radii):.3f} M (should be between disk_r_min and disk_r_max)")
print(f"  Mean spatial 4-velocity magnitude: {np.mean(speeds):.3f}")
print(f"  Max z-coordinate: {np.max(np.abs(positions[:, 2])):.2e} (should be close to zero)")

if np.any(np.isnan(positions)):
    nan_count = np.count_nonzero(np.isnan(data['pos'][:,0]))
    print(f"  WARNING: Found {nan_count} terminated (NaN) particles in this snapshot.")

# --- 3. Create the 3D Plot (No changes needed here) ---
plt.style.use('dark_background')
fig = plt.figure(figsize=(12, 10))
ax = fig.add_subplot(111, projection='3d')

num_to_plot = min(num_particles, 2000)
plot_indices = np.random.choice(num_particles, num_to_plot, replace=False)

colors = radii[plot_indices]
sc = ax.scatter(positions[plot_indices, 0], positions[plot_indices, 1], positions[plot_indices, 2],
                c=colors, cmap='plasma', s=5, label='Disk Particles')

r_horizon = M_scale * (1 + np.sqrt(1 - a_spin**2))
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
x_bh = r_horizon * np.outer(np.cos(u), np.sin(v))
y_bh = r_horizon * np.outer(np.sin(u), np.sin(v))
z_bh = r_horizon * np.outer(np.ones(np.size(u)), np.cos(v))
ax.plot_surface(x_bh, y_bh, z_bh, color='black', alpha=0.9, rstride=4, cstride=4)
ax.plot_wireframe(x_bh, y_bh, z_bh, color='dimgrey', alpha=0.2, rstride=10, cstride=10)

ax.set_xlabel('X (M)', fontsize=12, labelpad=10)
ax.set_ylabel('Y (M)', fontsize=12, labelpad=10)
ax.set_zlabel('Z (M)', fontsize=12, labelpad=10)

max_radius_plot = np.max(radii) * 1.1
```

```
        ax.set_xlim(-max_radius_plot, max_radius_plot)
        ax.set_ylim(-max_radius_plot, max_radius_plot)
        ax.set_zlim(-max_radius_plot/2, max_radius_plot/2)

        ax.set_title(f"Accretion Disk Snapshot from {os.path.basename(snapshot_to_load)}", fontsize=16)
        fig.colorbar(sc, ax=ax, shrink=0.6, aspect=10, label='Particle Radius (M)')
        ax.view_init(elev=45., azim=45)

        plt.show()
```

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


def plot_apsidal_precession_validation(
    project_dir: str = "project/mass_integrator",
    input_filename: str = "massive_particle_path.txt",
    M_scale: float = 1.0,
    a_spin: float = 0.9
) -> None:
    """
    Reads trajectory data and validates the apsidal precession rate against
    the theoretical GR formula for nearly circular orbits.
    """
    print("--- Generating Apsidal Precession Validation Plot ---")

    full_path = os.path.join(project_dir, input_filename)
    if not os.path.exists(full_path):
        print(f"ERROR: Trajectory file not found at '{full_path}'")
        return

    try:
        data = np.loadtxt(full_path, skiprows=1)
        x_coords = data[:, 2]
        y_coords = data[:, 3]
    except Exception as e:
        print(f"ERROR: Failed to load data. Exception: {e}")
        return

    radius = np.sqrt(x_coords**2 + y_coords**2)
    phi = np.unwrap(np.arctan2(y_coords, x_coords))

    # Find the angles where the particle is at periapsis (minimum radius)
    # We find indices where the radius is a local minimum
```

```python
    periapsis_indices = (np.r_[True, radius[1:] < radius[:-1]] & np.r_[radius[:-1] < radius[1:], True]).nonzero()[0]

    if len(periapsis_indices) < 2:
        print("Could not find at least two periapsis points. Cannot calculate precession.")
        return

    # Calculate the measured precession angle per orbit
    delta_phi_measured = phi[periapsis_indices[1]] - phi[periapsis_indices[0]]
    precession_per_orbit_measured = delta_phi_measured - 2 * np.pi

    # --- Theoretical Calculation at the average radius of the orbit ---
    r_avg = np.mean(radius)
    M = M_scale
    a = a_spin

    Omega_phi_theory = (M**0.5) / (r_avg**1.5 + a * M**0.5)
    Omega_r_theory_sq = Omega_phi_theory**2 * (1 - (6*M)/r_avg + (8*a*M**0.5)/r_avg**1.5 - (3*a**2)/r_avg**2)

    if Omega_r_theory_sq < 0:
        print("Theoretical orbit is unstable (Ω_r^2 < 0). Cannot calculate precession.")
        return

    Omega_r_theory = np.sqrt(Omega_r_theory_sq)

    # Precession per unit time
    Omega_precession_theory = Omega_phi_theory - Omega_r_theory

    # Period of one radial oscillation
    T_r = 2 * np.pi / Omega_r_theory

    # Total precession angle over one radial period
    precession_per_orbit_theory = Omega_precession_theory * T_r

    percent_error = 100 * abs(precession_per_orbit_measured - precession_per_orbit_theory) / precession_per_orbit_theory

    print(f"Apsidal Precession Validation (at average radius r={r_avg:.3f} M):")
    print(f"  Measured precession per orbit:   {precession_per_orbit_measured:.6f} radians")
    print(f"  Theoretical precession per orbit: {precession_per_orbit_theory:.6f} radians")
    print(f"  Relative Error: {percent_error:.4f}%")

    # --- Create the Plot (Polar Plot) ---
    plt.style.use('dark_background')
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111, polar=True)
```

```
        ax.plot(phi, radius, color='cyan', label='Particle Orbit')
        ax.scatter(phi[periapsis_indices], radius[periapsis_indices], color='lime', s=100, label='Periapsis Points', zorder=5)

        ax.set_title('Validation: Apsidal Precession of a Nearly Circular Orbit', fontsize=16)
        ax.set_xlabel('Azimuthal Angle (φ)', fontsize=12)
        ax.set_ylabel('Radius (r) [M]', fontsize=12, labelpad=-50)
        ax.legend()

        plt.show()
```

```
[46]:   #plot_particle_trajectory(
        #     project_dir="project/mass_integrator",
        #     input_filename="massive_particle_path.txt",
        #     M_scale=1.0,
        #     a_spin=0.9
        #)
        # After running the C code, call this function.
        plot_trajectory_components()
        # Call this function after running your C code.
        plot_radius_vs_time()
        # --- How to run ---
        plot_precession_validation()
        # --- How to run ---
        plot_apsidal_precession_validation()
```

```
--- Generating Trajectory Component Plots ---
ERROR: Trajectory file not found at
'project/mass_integrator/massive_particle_path.txt'
--- Generating Radius vs. Time Validation Plot ---
ERROR: Trajectory file not found at
'project/mass_integrator/massive_particle_path.txt'
--- Generating Precession Validation Plot ---
ERROR: Trajectory file not found at
'project/mass_integrator/massive_particle_path.txt'
--- Generating Apsidal Precession Validation Plot ---
ERROR: Trajectory file not found at
'project/mass_integrator/massive_particle_path.txt'
```

# 24  Video of disk movement

```
[47]:  import numpy as np
       import matplotlib.pyplot as plt
       import os
       import glob
       import shutil
       from mpl_toolkits.mplot3d import Axes3D
       import nrpy.params as par

       # This is the corrected helper function.
       def _plot_single_frame(
           positions: np.ndarray,
           radii: np.ndarray,
           M_scale: float,
           a_spin: float,
           current_time: float,
           output_filename: str,
           fig_width_inches: float,
           fig_dpi: int
       ) -> None:
           """
           Plots a single frame of the disk animation and saves it to a file,
           guaranteeing the output image has even dimensions for video encoding.
           """
           # Calculate the desired pixel dimensions
           desired_width_px = fig_width_inches * fig_dpi

           # Get the aspect ratio from the data to calculate height
           x_range = np.max(positions[:, 0]) - np.min(positions[:, 0])
           y_range = np.max(positions[:, 1]) - np.min(positions[:, 1])
           aspect_ratio = y_range / x_range if x_range > 0 else 1.0
           desired_height_px = desired_width_px * aspect_ratio

           # Round down to the nearest even number
           final_width_px = int(desired_width_px // 2 * 2)
           final_height_px = int(desired_height_px // 2 * 2)

           # Recalculate figure size in inches to match the final pixel dimensions
           final_fig_width_inches = final_width_px / fig_dpi
           final_fig_height_inches = final_height_px / fig_dpi

           fig = plt.figure(figsize=(final_fig_width_inches, final_fig_height_inches))
           ax = fig.add_subplot(111, projection='3d')
```

```python
    num_particles = len(positions)
    num_to_plot = min(num_particles, 5000)
    plot_indices = np.random.choice(num_particles, num_to_plot, replace=False)

    colors = radii[plot_indices]
    sc = ax.scatter(positions[plot_indices, 0], positions[plot_indices, 1], positions[plot_indices, 2],
                    c=colors, cmap='plasma', s=5)

    r_horizon = M_scale * (1 + np.sqrt(1 - a_spin**2))
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)
    x_bh = r_horizon * np.outer(np.cos(u), np.sin(v))
    y_bh = r_horizon * np.outer(np.sin(u), np.sin(v))
    z_bh = r_horizon * np.outer(np.ones(np.size(u)), np.cos(v))
    ax.plot_surface(x_bh, y_bh, z_bh, color='black', alpha=0.9, rstride=4, cstride=4)
    ax.plot_wireframe(x_bh, y_bh, z_bh, color='dimgrey', alpha=0.2, rstride=10, cstride=10)

    ax.set_xlabel('X (M)', fontsize=12, labelpad=10)
    ax.set_ylabel('Y (M)', fontsize=12, labelpad=10)
    ax.set_zlabel('Z (M)', fontsize=12, labelpad=10)

    max_radius_plot = np.max(radii) * 1.1
    ax.set_xlim(-max_radius_plot, max_radius_plot)
    ax.set_ylim(-max_radius_plot, max_radius_plot)
    ax.set_zlim(-max_radius_plot/2, max_radius_plot/2)

    ax.set_title(f"Accretion Disk Snapshot at t = {current_time:.2f} M", fontsize=16)
    fig.colorbar(sc, ax=ax, shrink=0.6, aspect=10, label='Particle Radius (M)')
    ax.view_init(elev=45., azim=45)

    # --- THE CRITICAL FIX IS HERE ---
    # Adjust layout first, then save without the problematic argument.
    plt.tight_layout()
    plt.savefig(output_filename, dpi=fig_dpi) # REMOVED: bbox_inches='tight'
    # --- END OF FIX ---

    plt.close(fig)

# This is the main function that calls the helper above. It does not need changes,
# but is included for completeness.
def generate_animation_frames(
    project_dir: str = "project/mass_integrator",
    output_folder: str = "output_GrandDesign_ISCO",
```

```python
    frames_output_dir: str = "animation_frames",
    M_scale: float = 1.0,
    a_spin: float = 0.95,
    fig_width_inches: float = 12.0,
    fig_dpi: int = 150,
    overwrite_existing_frames: bool = False
) -> None:
    # ... (This function's body is unchanged from the previous answer) ...
    print("--- Starting Animation Frame Generation ---")
    plt.style.use('dark_background')

    snapshot_dir = os.path.join(project_dir, output_folder)
    if not os.path.isdir(snapshot_dir):
        print(f"ERROR: Snapshot directory not found at '{snapshot_dir}'")
        return

    snapshot_files = glob.glob(os.path.join(snapshot_dir, "mass_blueprint_t_*.bin"))
    if not snapshot_files:
        print(f"ERROR: No snapshot .bin files found in '{snapshot_dir}'")
        return

    snapshot_files.sort(key=lambda f: int(os.path.basename(f).split('_t_')[1].split('.bin')[0]))
    print(f"Found {len(snapshot_files)} snapshots to process.")

    full_frames_dir = os.path.join(project_dir, frames_output_dir)
    os.makedirs(full_frames_dir, exist_ok=True)

    if not overwrite_existing_frames:
        print("Overwrite is OFF. Will skip any frames that already exist.")

    snapshot_dtype = np.dtype([
        ('id', np.int32), ('pos', 'f8', (3,)), ('u', 'f8', (4,)),
        ('lambda_rest', 'f8'), ('j_intrinsic', 'f4')
    ])

    try:
        snapshot_every_t = par.parval_from_str("snapshot_every_t")
    except ValueError:
        snapshot_every_t = 1.0

    for i, snapshot_file in enumerate(snapshot_files):
        frame_filename = os.path.join(full_frames_dir, f"frame_{i:04d}.png")

        if not overwrite_existing_frames and os.path.exists(frame_filename):
```

```
            print(f"  Skipping frame {i+1}/{len(snapshot_files)}: {os.path.basename(frame_filename)} already exists.")
            continue

        print(f"  Processing frame {i+1}/{len(snapshot_files)}: {os.path.basename(snapshot_file)}")

        with open(snapshot_file, 'rb') as f:
            num_particles = np.fromfile(f, dtype=np.int32, count=1)[0]
            if num_particles == 0: continue
            data = np.fromfile(f, dtype=snapshot_dtype, count=num_particles)

        positions = data['pos']
        radii = np.sqrt(positions[:, 0]**2 + positions[:, 1]**2)

        snapshot_number = int(os.path.basename(snapshot_file).split('_t_')[1].split('.bin')[0])
        current_time = snapshot_number * snapshot_every_t

        _plot_single_frame(
            positions, radii, M_scale, a_spin, current_time, frame_filename,
            fig_width_inches=fig_width_inches, fig_dpi=fig_dpi
        )

    print("\n--- Frame generation complete. ---")
```

```python
import subprocess
import os

def encode_video_from_frames(
    image_folder: str,
    output_video_path: str,
    frame_rate: int = 30,
    crf: int = 18
) -> None:
    # ... (This function is unchanged) ...
    print(f"--- Starting Video Encoding ---")
    output_dir = os.path.dirname(output_video_path)
    if output_dir:
        os.makedirs(output_dir, exist_ok=True)
    command = [
        'ffmpeg', '-y', '-framerate', str(frame_rate),
        '-i', os.path.join(image_folder, 'frame_%04d.png'),
        '-c:v', 'libx264', '-pix_fmt', 'yuv420p',
        '-r', str(frame_rate), '-crf', str(crf),
        output_video_path
    ]
```

```python
        print(f"Running FFmpeg command:\n{' '.join(command)}")
        try:
            result = subprocess.run(command, capture_output=True, text=True, check=True)
            print(f"\n[] Video encoding successful. File saved to '{output_video_path}'")
        except FileNotFoundError:
            print("\n[!] ERROR: FFmpeg not found.")
        except subprocess.CalledProcessError as e:
            print(f"\n[!] ERROR: FFmpeg failed with exit code {e.returncode}.")
            print("\n--- FFmpeg stderr ---")
            print(e.stderr)
```

```python
[49]:  # --- HOW TO RUN THE DEFINITIVE FIX ---

       # 1. Re-generate the frames. This is necessary because the files on disk are still wrong.
       #    This will replace the 1371px-wide images with new, even-dimensioned ones.
       print("STEP 1: Re-generating all frames with corrected dimensions...")
       generate_animation_frames(
           project_dir="/home/daltonm/Documents/project/mass_integrator",
           output_folder="output_GrandDesign_ISCO",
           frames_output_dir="animation_frames_grand_design",
           M_scale=1.0,
           a_spin=0.95,
           overwrite_existing_frames=True
       )
       print("STEP 1 COMPLETE.\n")
```

STEP 1: Re-generating all frames with corrected dimensions...
--- Starting Animation Frame Generation ---
ERROR: Snapshot directory not found at
'/home/daltonm/Documents/project/mass_integrator/output_GrandDesign_ISCO'
STEP 1 COMPLETE.

```python
[50]:  # 2. Encode the video from the newly created, valid frames.
       print("STEP 2: Encoding the new frames into a video...")
       encode_video_from_frames(
           image_folder="/home/daltonm/Documents/project/mass_integrator/animation_frames_grand_design",
           output_video_path="/home/daltonm/Documents/project/mass_integrator/grand_design_disk_evolution.mp4",
           frame_rate=15
       )
       print("STEP 2 COMPLETE.")
```

STEP 2: Encoding the new frames into a video...
--- Starting Video Encoding ---
Running FFmpeg command:

```
ffmpeg -y -framerate 15 -i /home/daltonm/Documents/project/mass_integrator/anima
tion_frames_grand_design/frame_%04d.png -c:v libx264 -pix_fmt yuv420p -r 15 -crf
18
/home/daltonm/Documents/project/mass_integrator/grand_design_disk_evolution.mp4


[!] ERROR: FFmpeg failed with exit code 254.


--- FFmpeg stderr ---
ffmpeg version 6.1.1-3ubuntu5 Copyright (c) 2000-2023 the FFmpeg developers
  built with gcc 13 (Ubuntu 13.2.0-23ubuntu3)
  configuration: --prefix=/usr --extra-version=3ubuntu5 --toolchain=hardened
--libdir=/usr/lib/x86_64-linux-gnu --incdir=/usr/include/x86_64-linux-gnu
--arch=amd64 --enable-gpl --disable-stripping --disable-omx --enable-gnutls
--enable-libaom --enable-libass --enable-libbs2b --enable-libcaca --enable-
libcdio --enable-libcodec2 --enable-libdav1d --enable-libflite --enable-
libfontconfig --enable-libfreetype --enable-libfribidi --enable-libglslang
--enable-libgme --enable-libgsm --enable-libharfbuzz --enable-libmp3lame
--enable-libmysofa --enable-libopenjpeg --enable-libopenmpt --enable-libopus
--enable-librubberband --enable-libshine --enable-libsnappy --enable-libsoxr
--enable-libspeex --enable-libtheora --enable-libtwolame --enable-libvidstab
--enable-libvorbis --enable-libvpx --enable-libwebp --enable-libx265 --enable-
libxml2 --enable-libxvid --enable-libzimg --enable-openal --enable-opencl
--enable-opengl --disable-sndio --enable-libvpl --disable-libmfx --enable-
libdc1394 --enable-libdrm --enable-libiec61883 --enable-chromaprint --enable-
frei0r --enable-ladspa --enable-libbluray --enable-libjack --enable-libpulse
--enable-librabbitmq --enable-librist --enable-libsrt --enable-libssh --enable-
libsvtav1 --enable-libx264 --enable-libzmq --enable-libzvbi --enable-lv2
--enable-sdl2 --enable-libplacebo --enable-librav1e --enable-pocketsphinx
--enable-librsvg --enable-libjxl --enable-shared
  libavutil      58. 29.100 / 58. 29.100
  libavcodec     60. 31.102 / 60. 31.102
  libavformat    60. 16.100 / 60. 16.100
  libavdevice    60.  3.100 / 60.  3.100
  libavfilter     9. 12.100 /  9. 12.100
  libswscale      7.  5.100 /  7.  5.100
  libswresample   4. 12.100 /  4. 12.100
  libpostproc    57.  3.100 / 57.  3.100
[image2 @ 0x61ed38b24840] Could find no file with path '/home/daltonm/Documents/
project/mass_integrator/animation_frames_grand_design/frame_%04d.png' and index
in the range 0-4
[in#0 @ 0x61ed38b24740] Error opening input: No such file or directory
Error opening input file /home/daltonm/Documents/project/mass_integrator/animati
on_frames_grand_design/frame_%04d.png.
Error opening input files: No such file or directory
```

STEP 2 COMPLETE.

```
[51]:  # --- How to Run This Test ---
       # 1. Run your C code in PRODUCTION mode (set run_in_debug_mode = false in the .par file).
       # 2. This will create an 'output' folder with several .bin files.
       # 3. Call this function. It will automatically find and plot the LAST snapshot.
       output_folder="output_GrandDesign_ISCO"
       visualize_disk_snapshot(output_folder=output_folder)
```

--- Visualizing Mass Blueprint Snapshot ---
ERROR: Snapshot directory not found at
'project/mass_integrator/output_GrandDesign_ISCO'

## 25   Circular Check

```
[52]:  # In file: V1_5_mass_geodesic (Copy).ipynb
       # In cell [63539fd1] (DEFINITIVE STABLE VERSION)

       import numpy as np

       def get_analytical_solution(tau_values, r_initial, M_scale, a_spin):
           """
           Calculates the analytical trajectory (t,x,y,z) for a circular, equatorial
           orbit in Kerr spacetime using a NUMERICALLY STABLE method.

           This version implements the three-step process that avoids catastrophic
           cancellation for orbits near the ISCO.

           Args:
               tau_values (np.ndarray): Array of proper time values.
               r_initial (float): The constant radius of the orbit.
               M_scale (float): The mass of the black hole.
               a_spin (float): The spin parameter of the black hole.

           Returns:
               tuple: (U_t, Omega_tau, analytical_positions)
           """
           # Use shorter variable names for clarity
           r = r_initial
           M = M_scale
           a = a_spin

           # --- Step 1: Calculate the stable angular velocity Omega = d(phi)/dt ---
           sqrt_M = np.sqrt(M)
```

```python
        Omega = sqrt_M / (r**1.5 + a * sqrt_M)

        # --- Step 2: Calculate the required metric components in Boyer-Lindquist ---
        # These are specialized for the equatorial plane (theta=pi/2)
        g_tt = -(1 - 2*M/r)
        g_tphi = -2*a*M/r
        g_phiphi = r**2 + a**2 + (2*M*a**2)/r

        # --- Step 3: Solve for u^t and u^phi using the stable formulas ---
        ut_squared_inv_denom = g_tt + 2*g_tphi*Omega + g_phiphi*Omega**2

        # Check for instability. If the denominator is non-negative, the orbit is not possible.
        if ut_squared_inv_denom >= 0:
            # Return NaNs to signal that this orbit is unstable/invalid.
            nan_array = np.full((len(tau_values), 4), np.nan)
            return np.nan, np.nan, nan_array

        ut_squared = -1.0 / ut_squared_inv_denom
        U_t = np.sqrt(ut_squared)

        u_phi = Omega * U_t
        Omega_tau = u_phi # d(phi)/d(tau) is u^phi

        # --- Step 4: Calculate the trajectory over the given proper time values ---
        phi_of_tau = Omega_tau * tau_values
        t_of_tau = U_t * tau_values
        x_of_tau = r_initial * np.cos(phi_of_tau)
        y_of_tau = r_initial * np.sin(phi_of_tau)
        z_of_tau = np.zeros_like(tau_values)

        analytical_positions = np.vstack([t_of_tau, x_of_tau, y_of_tau, z_of_tau]).T

        return U_t, Omega_tau, analytical_positions
```

```python
[53]:   # In file: V1_5_mass_geodesic (Copy).ipynb
        # In cell [43df2a3a] (DEFINITIVE, ROBUST VERSION)

        import numpy as np
        import subprocess
        import os
        import matplotlib.pyplot as plt
        from typing import List, Tuple

        def verify_mass_integrator(
```

```python
    test_cases: List[Tuple[float, float]],
    M_scale: float = 1.0,
    tau_max: float = 2000.0,
    project_dir: str = "project/mass_integrator",
    executable_name: str = "mass_integrator",
    output_folder: str = "verification_plots"
) -> None:
    """
    Runs a verification suite for the massive particle integrator for specific test cases.

    VERSION 3: Includes robust NaN checking to correctly handle and report unstable orbits.
    """
    print("--- Starting Mass Integrator Verification Suite ---")

    full_output_folder = os.path.join(project_dir, output_folder)
    os.makedirs(full_output_folder, exist_ok=True)
    print(f"Plots will be saved to: {full_output_folder}")

    for a_spin, r_initial in test_cases:
        print(f"\n--- Verifying r_initial={r_initial:.2f}, a_spin={a_spin:.2f} ---")

        # --- 1. Generate Initial Conditions ---
        U_t_initial, Omega_tau_initial, _ = get_analytical_solution(np.array([0.0]), r_initial, M_scale, a_spin)

        # *** ROBUSTNESS CHECK 1: Check if analytical solution itself is unstable ***
        if np.isnan(U_t_initial):
            print("  [] ANALYTICAL FAILURE: The chosen r_initial is inside the ISCO or numerically unstable.")
            print("      Skipping this test case.")
            continue
        # *** END OF CHECK ***

        u_x_initial = 0.0
        u_y_initial = r_initial * Omega_tau_initial
        u_z_initial = 0.0

        ic_filename = os.path.join(project_dir, "particle_debug_initial_conditions.txt")
        with open(ic_filename, "w") as f:
            f.write("# Format: t_initial pos_x pos_y pos_z u_x u_y u_z\n")
            f.write(f"0.0 {r_initial:.10f} 0.0 0.0   {u_x_initial:.10f} {u_y_initial:.10f} {u_z_initial:.10f}\n")

        # --- 2. Run the C Integrator ---
        par_filename = os.path.join(project_dir, "mass_integrator.par")
        with open(par_filename, "w") as f:
            f.write(f"run_in_debug_mode = True\n")
```

```python
        f.write(f"a_spin = {a_spin:.10f}\n")
        f.write(f"M_scale = {M_scale:.10f}\n")
        f.write(f"t_max_integration = {tau_max * 2 * U_t_initial}\n")
        f.write(f"metric_choice = 0\n")

    output_path = os.path.join(project_dir, "massive_particle_path.txt")
    if os.path.exists(output_path):
        os.remove(output_path)

    try:
        subprocess.run(
            f"./{executable_name}", shell=True, capture_output=True, text=True, check=True, cwd=project_dir
        )
        print("  [] C Integrator ran successfully.")
    except subprocess.CalledProcessError as e:
        print(f"  [] ERROR: C Integrator failed for r={r_initial}, a={a_spin}.")
        print(e.stderr)
        continue

    # --- 3. Load Numerical and Generate Analytical Results ---
    try:
        numerical_data = np.loadtxt(output_path, skiprows=1)
        if numerical_data.size == 0:
            print("  [] NUMERICAL FAILURE: Output file is empty.")
            continue
        numerical_data = numerical_data[numerical_data[:, 0] <= tau_max]
        print(f"  [] Loaded {len(numerical_data)} numerical data points.")
    except Exception as e:
        print(f"  [] ERROR: Could not load or parse output file '{output_path}'. Error: {e}")
        continue

    # *** ROBUSTNESS CHECK 2: Check if numerical integrator produced NaNs ***
    if np.any(np.isnan(numerical_data)):
        print("  [] NUMERICAL FAILURE: The C integrator produced NaN values.")
        print("      This indicates the orbit was unstable as predicted. Skipping statistics.")
        continue
    # *** END OF CHECK ***

    tau_values = numerical_data[:, 0]
    _, _, analytical_data = get_analytical_solution(tau_values, r_initial, M_scale, a_spin)
    print("  [] Generated analytical ground truth.")

    # --- 4. Calculate Statistics ---
    pos_error = np.sqrt(np.sum((numerical_data[:, 2:5] - analytical_data[:, 1:4])**2, axis=1))
```

```python
    radius_error = np.abs(np.sqrt(numerical_data[:, 2]**2 + numerical_data[:, 3]**2) - r_initial)

    phi_num = np.unwrap(np.arctan2(numerical_data[:, 3], numerical_data[:, 2]))
    phi_ana = np.unwrap(np.arctan2(analytical_data[:, 2], analytical_data[:, 1]))
    phase_error = np.abs(phi_num - phi_ana)

    print("\n  --- STATISTICAL REPORT ---")
    print(f"  Final Position Error:      {pos_error[-1]:.3e} M")
    print(f"  Max Radius Error (Drift):  {np.max(radius_error):.3e} M")
    print(f"  Mean Radius Error:         {np.mean(radius_error):.3e} M")
    print(f"  Max Phase Error (Timing):  {np.max(phase_error):.3e} radians")
    print(f"  Mean Phase Error:          {np.mean(phase_error):.3e} radians")

    # *** ROBUSTNESS CHECK 3: Corrected verdict logic ***
    if np.max(radius_error) > 1e-5 or np.max(phase_error) > 1e-5:
        print("\n  --- VERDICT ---\n  FAIL: Error metrics exceed tolerance.")
    else:
        print("\n  --- VERDICT ---\n  PASS: All error metrics are within tolerance.")
    # *** END OF CHECK ***

    # --- 5. Generate and Save Plots ---
    fig, axes = plt.subplots(1, 2, figsize=(18, 8))
    fig.suptitle(f"Verification for r_initial={r_initial:.2f}, a_spin={a_spin:.2f}", fontsize=16)

    axes[0].plot(analytical_data[:, 1], analytical_data[:, 2], 'r--', label='Analytical', lw=2)
    axes[0].plot(numerical_data[:, 2], numerical_data[:, 3], 'c-', label='Numerical', lw=1, alpha=0.8)
    axes[0].set_title("Orbit Trajectory (Top-Down View)")
    axes[0].set_xlabel("x (M)"); axes[0].set_ylabel("y (M)")
    axes[0].set_aspect('equal', 'box')
    axes[0].legend(); axes[0].grid(True)

    axes[1].plot(tau_values, radius_error, label='Radius Error |r_num - r_initial|')
    axes[1].plot(tau_values, phase_error, label='Phase Error |φ_num - φ_ana|')
    axes[1].set_title("Error Growth over Proper Time")
    axes[1].set_xlabel("Proper Time (τ) [M]"); axes[1].set_ylabel("Error")
    axes[1].set_yscale('log')
    axes[1].legend(); axes[1].grid(True)

    plot_filename = f"verification_r{r_initial:.2f}_a{a_spin:.2f}.png"
    full_plot_path = os.path.join(full_output_folder, plot_filename)
    plt.savefig(full_plot_path, dpi=150, bbox_inches='tight')
    print(f"  [] Plot saved to '{full_plot_path}'")

    plt.show()
```

```
[54]: raise KeyboardInterrupt("Execution stopped here intentionally.")
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[54], line 1
----> 1 raise KeyboardInterrupt("Execution stopped here intentionally.")

KeyboardInterrupt: Execution stopped here intentionally.
```

```python
# Prograde orbits with high spin (a=0.99)
# ISCO is at approximately r = 1.232 M. Orbits below ~1.4M can be numerically sensitive.
prograde_test_cases_stable = [
    # (a_spin, r_initial)
    (0.99, 1.5),    # A stable orbit just outside the numerically sensitive region
    (0.99, 8),    # Close, stable orbit
    (0.99, 9),
    (0.99, 10),    # Photon sphere region
    (0.99, 11),
    (0.99, 6.0),    # Equivalent to Schwarzschild ISCO
    (0.99, 9.0),    # Equivalent to Retrograde ISCO
    (0.99, 15.0),    # Intermediate orbit
    (0.99, 25.0),
    (0.99, 50.0)    # Weak-field orbit
]



# Example function call:
verify_mass_integrator(
    test_cases=prograde_test_cases_stable,
    output_folder="verification_prograde_a0.99"
)
```

```python
# Retrograde orbits with high spin (a=-0.99)
# ISCO is at approximately r = 8.98 M
retrograde_test_cases = [
    # (a_spin, r_initial)
    (-0.99, 9.0),    # Just outside the ISCO
    (-0.99, 9.5),    # Near-ISCO
    (-0.99, 10.0),
    (-0.99, 12.0),
    (-0.99, 15.0),    # Intermediate orbit
    (-0.99, 20.0),
```

```
        (-0.99, 30.0),
        (-0.99, 50.0),
        (-0.99, 75.0),
        (-0.99, 100.0)    # Weak-field orbit
    ]


    # Example function call:
    verify_mass_integrator(
        test_cases=retrograde_test_cases,
        output_folder="verification_retrograde_a-0.99"
    )
```

```
[ ]:  # Zero-spin (Schwarzschild) orbits (a=0.0)
      # ISCO is at r = 6.0 M
      schwarzschild_test_cases = [
          # (a_spin, r_initial)
          (0.0, 6.0),      # At the ISCO
          (0.0, 6.5),      # Near-ISCO
          (0.0, 7.0),
          (0.0, 8.0),
          (0.0, 10.0),     # Intermediate orbit
          (0.0, 15.0),
          (0.0, 20.0),
          (0.0, 30.0),
          (0.0, 50.0),
          (0.0, 100.0)     # Weak-field orbit
      ]


      # Example function call:
      verify_mass_integrator(
          test_cases=schwarzschild_test_cases,
          output_folder="verification_schwarzschild_a0.00"
      )
```

```
[ ]:  # Prograde orbits with high spin (a=0.99)
      # ISCO is at approximately r = 1.232 M. Orbits below ~1.4M can be numerically sensitive.
      prograde_test_cases_stable_test = [
          # (a_spin, r_initial)
          (0.99, 3) # Close, stable orbit
      ]



      # Example function call:
```

```
verify_mass_integrator(
    test_cases=prograde_test_cases_stable_test,
    output_folder="verification_prograde_test_a0.99"
)
```

[ ]: 
```
plot_particle_trajectory()
```