V12_4_light_geodesic

July 30, 2025

# Step 1: Introduction, Core Physics, and Project Goals

This notebook is a self-contained tutorial that uses the `nrpy` library to construct a complete C-language project for integrating photon geodesics in curved spacetimes. The resulting C code is a flexible, high-performance ray-tracing engine capable of generating gravitationally lensed images of distant sources as seen by an observer near a black hole.

The core of the project is the numerical solution of the geodesic equation, which describes the path of a free-falling particle (or photon) through curved spacetime. The geodesic equation, as detailed on Wikipedia, is a second-order ordinary differential equation (ODE) that relates a particle's acceleration to the spacetime curvature, represented by the Christoffel symbols ($\Gamma^{\alpha}_{\mu\nu}$):

$$\frac{d^2x^\alpha}{d\lambda^2} = -\Gamma^{\alpha}_{\mu\nu}\frac{dx^\mu}{d\lambda}\frac{dx^\nu}{d\lambda}$$

Here, $x^\alpha = (t, x, y, z)$ are the spacetime coordinates, and λ is the affine parameter, which measures the proper distance along the path for a massive particle or a suitable path parameter for a photon.

-2.0.1 The Reverse Ray-Tracing Transformation

To render an image of what an observer sees, we must trace the photon's path from the observer's camera *backwards in time* to its source. While we could integrate the geodesic equation with a negative step `dλ < 0`, most ODE solvers are optimized for forward integration with a positive step. To accommodate this, we perform a change of variables on the affine parameter. We define a new parameter, κ, that increases as the original parameter, λ, decreases:

$$\kappa = -\lambda \implies d\kappa = -d\lambda \implies \frac{d}{d\lambda} = -\frac{d}{d\kappa}$$

We now substitute this transformation directly into the second-order geodesic equation:

$$\frac{d}{d\lambda}\left(\frac{dx^\alpha}{d\lambda}\right) = -\Gamma^{\alpha}_{\mu\nu}\frac{dx^\mu}{d\lambda}\frac{dx^\nu}{d\lambda}$$

Applying the chain rule, $\frac{d}{d\lambda} = -\frac{d}{d\kappa}$:

$$\left(-\frac{d}{d\kappa}\right)\left(-\frac{dx^\alpha}{d\kappa}\right) = -\Gamma^{\alpha}_{\mu\nu}\left(-\frac{dx^\mu}{d\kappa}\right)\left(-\frac{dx^\nu}{d\kappa}\right)$$

The negatives on both sides cancel, yielding the reverse-time geodesic equation:

$$\frac{d^2x^\alpha}{d\kappa^2} = -\Gamma^\alpha_{\mu\nu}\frac{dx^\mu}{d\kappa}\frac{dx^\nu}{d\kappa}$$

This equation has the same form as the original, but describes the path integrated with respect to κ. To solve it numerically, we now decompose this second-order ODE into a system of coupled first-order ODEs. We define the **reverse-time momentum**, p^α, as the 4-velocity with respect to our new parameter κ:

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

This definition immediately gives us our first ODE. We find the second by substituting p^α into the reverse-time geodesic equation:

$$\frac{d}{d\kappa}\left(\frac{dx^\alpha}{d\kappa}\right) = -\Gamma^\alpha_{\mu\nu}\left(\frac{dx^\mu}{d\kappa}\right)\left(\frac{dx^\nu}{d\kappa}\right) \implies \frac{dp^\alpha}{d\kappa} = -\Gamma^\alpha_{\mu\nu}p^\mu p^\nu$$

This gives us the final set of ODEs that our C code will solve. We also add a third ODE to track the total proper distance traveled by the photon along its spatial path, using the spatial part of the metric γ_{ij}:

1. **Position ODE**: $\frac{dx^\alpha}{d\kappa} = p^\alpha$
2. **Momentum ODE**: $\frac{dp^\alpha}{d\kappa} = -\Gamma^\alpha_{\mu\nu}p^\mu p^\nu$
3. **Path Length ODE**: $\frac{dL}{d\kappa} = \sqrt{\gamma_{ij}\frac{dx^i}{d\kappa}\frac{dx^j}{d\kappa}} = \sqrt{\gamma_{ij}p^i p^j}$

-2.0.2 Initial Conditions

The initial value of the reverse-time momentum, $p^\alpha_{\text{initial}}$, determines the starting direction of the ray traced from the camera. It is physically equivalent to the *negative* of the final momentum of a photon that started at a distant source and ended its journey at the camera. If we denote the physical, forward-time 4-velocity as $k^\alpha = dx^\alpha/d\lambda$, then:

$$p^\alpha_{\text{initial}} = \left(\frac{dx^\alpha}{d\kappa}\right)_{\text{initial}} = -\left(\frac{dx^\alpha}{d\lambda}\right)_{\text{final}} = -k^\alpha_{\text{final}}$$

This relationship is key: setting the initial conditions for our reverse-time integration is equivalent to choosing the final momentum of a physically forward-propagating photon arriving at the camera.

This notebook follows a modular, single-responsibility design pattern. It uses the `nrpy` library to first define the underlying physics symbolically, and then automatically generates a series of interoperable C functions, each with a specific job. This makes the final C project clear, efficient, and easily extensible.

Notebook Status: Validated

-1 Table of Contents

This notebook is organized into a series of logical steps that build the complete C project from the ground up. Each step focuses on a specific aspect of the architecture, from pure mathematics to the final compiled executable.

1. Step 1: Introduction, Core Physics, and Project Goals
2. Step 2: Project Initialization and Parameter Definition

3. Step 3: The Symbolic Core - Defining the Physics with nrpy and sympy
    - 3.a: Symbolic Recipe for Metric Tensor Derivatives
    - 3.b: Symbolic Recipe for Christoffel Symbols (Analytic Metrics)
    - 3.c: Symbolic Recipe for the Geodesic Momentum ODE
    - 3.d: Symbolic Recipe for the Geodesic Position ODE
    - 3.e: Symbolic Recipe for the Path Length ODE
    - 3.f: Symbolic Recipe for the Null Condition (Calculating p)
    - 3.g: Symbolic Recipes for Conserved Quantities (E, L, Q)
    - 3.h: Symbolic Recipes for Numerical Metrics
4. Step 4: Spacetime Definitions (Analytic Metrics)
5. Step 5: Symbolic Workflow Execution
6. Step 6: C Code Generation - Physics Engines and Workers
    - 6.a: C Workers for Analytic Metrics
    - 6.b: C Engines and Workers for Numerical Metrics
    - 6.c: Generic Physics and Integration Engines
7. Step 7: C Code Generation - Orchestrators and Dispatchers
8. Step 8: Project Assembly and Compilation

# Step 2: Project Initialization and Parameter Definition

This cell sets up the foundational elements for our entire project. It performs three key tasks:

1. **Import Libraries**: We import necessary modules from standard Python libraries (`os`, `shutil`, `sympy`) and the core components of `nrpy`. The `nrpy` imports provide tools for C function registration, C code generation, parameter handling, and infrastructure management.

2. **Directory Management**: A clean output directory, `project/photon_geodesic_integrator/`, is created to store the generated C code, ensuring a fresh build every time the notebook is run.

3. **Physical and Runtime Parameter Definition**: We define the many parameters that control the simulation using `nrpy`'s parameter management system. This is the central mechanism for defining a runtime parameter that will be accessible in the generated C code. The `nrpy` build system uses this registry of parameters to automatically construct C data structures, a default parameter file, and a robust command-line parser.

-1.0.1 `nrpy` Functions Used in this Cell:

- `nrpy.params.set_parval_from_str(par_name, value)`:
    - **Source File**: `nrpy/params.py`
    - **Description**: Sets the value of a core `nrpy` parameter. Here, it is used to specify that we are using the `BHaH` (BlackHoles@Home) C code generation infrastructure, which governs how files are organized and how the `Makefile` is constructed.
- `nrpy.params.register_CodeParameter(c_type, module, name, default_value, **kwargs)`:
    - **Source File**: `nrpy/params.py`
    - **Description**: This is the primary function for registering a C-level parameter. It creates a parameter object that holds all its properties and stores it in a global registry.
    - **Key Inputs**:
        * `c_type`: The data type of the parameter in the C code (e.g., `"REAL"`, `"int"`).
        * `module`: The name of the Python module where the parameter is defined (usually `__name__`).
        * `name`: The C variable name for the parameter.
        * `default_value`: The default value for the parameter.
    - **Key Keyword Arguments (`kwargs`)**:

* `commondata=True`: Specifies that the parameter is "common" to the entire simulation (e.g., black hole mass `M_scale`). It will be stored in the `commondata_struct` in the generated C code. If `False`, it's stored in the grid-specific `params_struct`.
* `add_to_parfile=True`: Instructs the build system to add an entry for this parameter to a default parameter file, making it easy to configure at runtime.
* `add_to_set_CodeParameters_h=True`: This is a crucial flag that enables the "automatic unpacking" mechanism. It tells `nrpy` to add an entry for the parameter to the `set_CodeParameters.h` convenience header. Any C function registered with `include_CodeParameters_h=True` will get a local `const REAL` variable with the same name as the parameter, making the C code clean and readable. This is handled by the `nrpy.infrastructures.BHaH.CodeParameters` module.

[ ]:
```python
import os
import shutil
import sympy as sp

# NRPy-related imports for C-code generation
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


# Set project name and clean the output directory
project_name = "photon_geodesic_integrator"
project_dir = os.path.join("project", project_name)
shutil.rmtree(project_dir, ignore_errors=True)

# Set NRPy parameters for the BHaH infrastructure
par.set_parval_from_str("Infrastructure", "BHaH")

# --- Core Simulation & Physics Parameters ---
# Register single integer and boolean parameters
_ = par.register_CodeParameter("int", __name__, "metric_choice", 0, add_to_parfile=True, commondata=True)
_ = par.register_CodeParameters(
    "bool", __name__,
    ["perform_conservation_check", "debug_mode"],
    False,  # Assign False to both
    add_to_parfile=True, commondata=True
)

# Register physical parameters for the black hole.
# Note: These need add_to_set_CodeParameters_h=True
M_scale, a_spin = par.register_CodeParameters(
    "REAL", __name__,
    ["M_scale", "a_spin"],
    [1.0, 0.0],
    add_to_parfile=True, commondata=True, add_to_set_CodeParameters_h=True
)

# --- Universal Camera System Parameters ---
_ = par.register_CodeParameters(
    "REAL", __name__,
    ["camera_pos_x", "camera_pos_y", "camera_pos_z"],
    [0.0, 0.0, 51.0],
    add_to_parfile=True, commondata=True
)
_ = par.register_CodeParameters(
    "REAL", __name__,
    ["window_center_x", "window_center_y", "window_center_z"],
    [0.0, 0.0, 50.0],
    add_to_parfile=True, commondata=True
)
_ = par.register_CodeParameters(
    "REAL", __name__,
    ["window_up_vec_x", "window_up_vec_y", "window_up_vec_z"],
    [0.0, 1.0, 0.0],
    add_to_parfile=True, commondata=True
)

# --- Independent Source Plane Definition ---
_ = par.register_CodeParameters(
    "REAL", __name__,
    ["source_plane_normal_x", "source_plane_normal_y", "source_plane_normal_z"],
    [0.0, 0.0, 1.0],
    add_to_parfile=True, commondata=True
)
# Use a single default value for all center coordinates
_ = par.register_CodeParameters(
    "REAL", __name__,
    ["source_plane_center_x", "source_plane_center_y", "source_plane_center_z"],
    0.0,
    add_to_parfile=True, commondata=True
)
_ = par.register_CodeParameters(
    "REAL", __name__,
    ["source_up_vec_x", "source_up_vec_y", "source_up_vec_z"],
    [0.0, 1.0, 0.0],
    add_to_parfile=True, commondata=True
)
_ = par.register_CodeParameters(
    "REAL", __name__,
    ["source_r_min", "source_r_max"],
    [6.0, 25.0],
    add_to_parfile=True, commondata=True
)

# --- General Ray-Tracing & Integration Parameters ---
_ = par.register_CodeParameter("int", __name__, "scan_density", 512, add_to_parfile=True, commondata=True)
_ = par.register_CodeParameters(
    "REAL", __name__,
    ["flatness_threshold", "r_escape", "t_integration_max", "t_start", "mass_snapshot_every_t", "delta_r_max"],
    [1e-2, 1500.0, 10000.0, 2000.0, 10.0, 2.0],
    add_to_parfile=True, commondata=True
)
p_t_max = par.CodeParameter(
    "REAL", __name__, "p_t_max", 1000.0,
    add_to_parfile=True, commondata=True
)
window_size = par.CodeParameter(
    "REAL", __name__, "window_size", 1.5,
    add_to_parfile=True, commondata=True, add_to_set_CodeParameters_h=True
)

# --- NEW: Time Slot Manager Parameters ---
print("-> Registering CodeParameters for the Time Slot Manager...")
slot_manager_t_min = par.CodeParameter(
    "REAL", __name__, "slot_manager_t_min", -100.0,
    add_to_parfile=True, commondata=True
)
slot_manager_delta_t = par.CodeParameter(
    "REAL", __name__, "slot_manager_delta_t", 0.1,
    add_to_parfile=True, commondata=True
)

# --- Adaptive Window Grid Parameters ---
print("-> Registering CodeParameters for adaptive window grids...")
_ = par.register_CodeParameters(
    "int", __name__,
    ["window_grid_type", "log_polar_num_r", "log_polar_num_phi"],
    [0, 512, 1024],
    commondata=True, add_to_parfile=True
```

```
)
_ = par.CodeParameter(
    "REAL", __name__, "log_polar_r_min", 0.1,
    commondata=True, add_to_parfile=True
)

# --- Bounding Box Parameters for the Accretion Disk ---
print("-> Registering CodeParameters for the disk bounding box...")
_ = par.register_CodeParameters(
    "REAL", __name__,
    ["disk_bounds_x_min", "disk_bounds_x_max", "disk_bounds_y_min", "disk_bounds_y_max", "disk_bounds_z_min", "disk_bounds_z_max"],
    [-26.0, 26.0, -26.0, 26.0, -1.0, 1.0],
    commondata=True, add_to_parfile=True
)
```

# Step 3: The Symbolic Core - Defining the Physics with `nrpy` and `sympy`

This section is the mathematical heart of the project. Here, we define the pure physics of geodesic motion not as C code, but as symbolic "recipes" using Python's `sympy` library. Each Python function in this section takes symbolic `sympy` objects as input (like a metric tensor) and returns new symbolic `sympy` expressions as output (like the Christoffel symbols).

This "symbolic-first" approach is a core principle of the `nrpy` framework. It offers several major advantages: 1. **Correctness**: By writing the physics in high-level symbolic math, we are much less likely to make subtle programming errors than if we were writing complex C code by hand. The computer handles the tedious algebra. 2. **Efficiency**: Complex calculations (like inverting a 4x4 matrix) are performed symbolically *once* when this notebook is run. The resulting simplified formula is then used to generate highly efficient C code. 3. **Modularity & Reusability**: We create generic recipes that are not tied to a specific spacetime. For example, the recipe for the geodesic equation RHS is valid for *any* metric. We can then plug different metric tensors into this single recipe to generate C code for different spacetimes.

The functions in this section will produce global Python variables containing the final symbolic expressions. These variables will be used later in Step 6 to automatically generate the C code.

### 3.a: Symbolic Recipe for 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.

-1.0.2 `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.

```
[ ]: 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: Symbolic Recipe for Christoffel Symbols (Analytic Metrics)

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 `δ, μ, ν, α` to construct each component of the Christoffel symbol tensor. The summation over the dummy index `α` is performed explicitly. 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.

-1.0.3 `nrpy` Functions Used in this Cell:

- `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.

```
[ ]: 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: Symbolic Recipe for the Geodesic Momentum ODE

This function defines the symbolic right-hand side (RHS) for the evolution of the **reverse-time momentum**, p^α. 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 μ and ν from 0 to 3, resulting in $4 \times 4 = 16$ terms for each component of α, 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 μ and ν. 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:

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

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:

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

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.

-1.0.4 `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.

[ ]:
```
def geodesic_mom_rhs(Gamma4UDD):
    """
    Symbolic RHS for momentum ODE: dp^a/dκ = -Γ^a_μν p^μ p^ν.
    p is the reverse-momentum, y[4]...y[7].
    """
    pt,pr,pth,pph = sp.symbols("y[4] y[5] y[6] y[7]", Real=True)
    pU = [pt,pr,pth,pph]
    geodesic_rhs = ixp.zerorank1(dimension=4)
    for alpha in range(4):
        for mu in range(4):
            geodesic_rhs[alpha] += Gamma4UDD[alpha][mu][mu] * pU[mu] * pU[mu]
            for nu in range(mu + 1, 4):
                geodesic_rhs[alpha] += 2 * Gamma4UDD[alpha][mu][nu] * pU[mu] * pU[nu]
        geodesic_rhs[alpha] = -geodesic_rhs[alpha]
    return geodesic_rhs
```

### 3.d: Symbolic Recipe for the Geodesic Position ODE

This function defines the symbolic right-hand side (RHS) for the evolution of the position coordinates, x^α. 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.

[ ]:
```
def geodesic_pos_rhs():
    """
    Symbolic RHS for position ODE: dx^a/dκ = p^a.
    p is the reverse-momentum, y[4]...y[7].
    """
    pt,pr,pth,pph = sp.symbols("y[4] y[5] y[6] y[7]", Real=True)
    pU = [pt,pr,pth,pph]
    return pU
```

### 3.e: Symbolic Recipe for the Path Length ODE

This function defines the symbolic right-hand side for the evolution of the proper length, L. This is the final component of our ODE system and allows us to track the total distance the photon has traveled along its spatial path. The proper length element dL is defined by the spatial part of the metric, $\gamma_{ij} = g_{ij}$ for $i, j \in \{1, 2, 3\}$:

$$dL^2 = \gamma_{ij} dx^i dx^j$$

Dividing by $d\kappa^2$ and taking the square root gives us the rate of change of proper length with respect to our integration parameter κ:

$$\frac{dL}{d\kappa} = \sqrt{\gamma_{ij} \frac{dx^i}{d\kappa} \frac{dx^j}{d\kappa}} = \sqrt{\gamma_{ij} p^i p^j}$$

The function `proper_lengh_rhs` symbolically implements the formula under the square root, $\sqrt{\gamma_{ij}p^ip^j}$. It uses `sympy` symbols for the spatial momentum components (`pU[1]`, `pU[2]`, `pU[3]`) and programmatically constructs the optimized sum $\gamma_{ij}p^ip^j$ using the same symmetry trick as the momentum RHS to reduce the number of terms. Finally, it returns a single-element list containing the square root of this sum. This will be the 9th component (`rhs_out[8]`) of our ODE system.

-1.0.5 `nrpy` Functions Used in this Cell:

- `nrpy.indexedexp.declarerank2(name, dimension, sym)`:
  - **Source File**: `nrpy/indexedexp.py`
  - **Description**: This function creates an *abstract* symbolic rank-2 tensor. Instead of creating symbols like `g11`, `g12`, etc., it creates symbols whose names are literally `name[1][1]`, `name[1][2]`, etc. This is a powerful `nrpy` technique for creating generic symbolic "recipes" that are later filled in with runtime data from a C struct. Here, it creates a placeholder for the metric components, `metric->g`, which will be provided by a C struct at runtime.

[ ]:
```python
def proper_lengh_rhs():
    p0,p1,p2,p3,L= sp.symbols("y[4] y[5] y[6] y[7] y[8]",Real=True)
    pU=[p0,p1,p2,p3]

    g4DD=ixp.declarerank2("metric->g",dimension=4, sym="sym01")

    sum = sp.simplify(0)

    for i in range(1,4):
        sum += g4DD[i][i]*pU[i]*pU[i]

        for j in range(i+1,4):
            sum += 2*g4DD[i][j]*pU[i]*pU[j]

    sp.simplify(sum)

    return [sp.sqrt(sum)]
```

### 3.f: Symbolic Recipe for the Null Condition (Calculating p)

To complete our initial data, we must enforce the **null geodesic condition**, which states that the squared 4-momentum of a photon is zero. This is because photons travel along null paths where the spacetime interval ds^2 is zero. This condition must be satisfied by the 4-momentum of any photon. Let's write this for the **forward-in-time** photon, with physical 4-momentum q^α:

$$g_{\mu\nu}q^\mu q^\nu = 0$$

Expanding this equation into its time and space components gives us the quadratic equation for the time-component of the physical momentum, q^0:

$$g_{00}(q^0)^2 + 2\left(g_{0i}q^i\right)q^0 + \left(g_{ij}q^iq^j\right) = 0$$

For our reverse ray-tracing, we use the **reverse-time momentum**, p^α, which is related to the physical momentum by $p^\alpha = -q^\alpha$. We can substitute this relationship directly into the equation above, replacing q^0 with $-p^0$ and q^i with $-p^i$:

$$g_{00}(-p^0)^2 + 2g_{0i}(-p^i)(-p^0) + \left(g_{ij}(-p^i)(-p^j)\right) = 0$$

The negative signs in the squared terms and the cross-term cancel out: `(-p^0)^2 = (p^0)^2`, `(-p^i)(-p^j) = p^i p^j`, and `(-p^i)(-p^0) = p^i p^0`. This yields a quadratic equation for p^0 that has the exact same form as the one for q^0:

$$g_{00}(p^0)^2 + 2\left(g_{0i}p^i\right)p^0 + \left(g_{ij}p^ip^j\right) = 0$$

We now solve this equation for p^0. It is a standard quadratic equation of the form $ax^2 + bx + c = 0$, where $x = p^0$. The coefficients are: * $a = g_{00}$ * $b = 2g_{0i}p^i$ * $c = g_{ij}p^ip^j$

The solution for p^0 is given by the quadratic formula:

$$p^0 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-2\left(g_{0i}p^i\right) \pm \sqrt{\left(2g_{0i}p^i\right)^2 - 4g_{00}\left(g_{ij}p^ip^j\right)}}{2g_{00}}$$

Simplifying by dividing the numerator and denominator by 2 gives:

$$p^0 = \frac{-\left(g_{0i}p^i\right) \pm \sqrt{\left(g_{0i}p^i\right)^2 - g_{00}\left(g_{ij}p^ip^j\right)}}{g_{00}}$$

The final step is to choose the physically correct root. For the reverse-traced photon, the parameter κ increases as coordinate time `t` decreases. Therefore, the derivative $p^0 = dt/d\kappa$ must be **negative**. In a typical stationary spacetime outside a black hole, g_{00} is negative. For the fraction to be negative, the numerator must be **positive**. The square root term is always positive and its magnitude is generally larger than the first term. To guarantee a positive numerator, we must choose the **plus sign (+)** in the `±`.

This leads to the final, correct result implemented in the code:

$$p^0 = \frac{-\left(g_{0i}p^i\right) + \sqrt{\left(g_{0i}p^i\right)^2 - g_{00}\left(g_{ij}p^ip^j\right)}}{g_{00}}$$

```
[ ]: def mom_time_p0_reverse():
         """
         Solves g_μν p^μ p^ν = 0 for our reverse-time momentum p^0.
         """
         p0,p1,p2,p3 = sp.symbols("y[4] y[5] y[6] y[7]", Real=True)
         pU=[p0,p1,p2,p3]
         g4DD = ixp.declarerank2("metric->g", sym="sym01", dimension=4)
         sum_g0i_pi = sp.sympify(0)
         for i in range(1,4):
             sum_g0i_pi += g4DD[0][i]*pU[i]
         sum_gij_pi_pj = sp.sympify(0)
         for i in range(1,4):
             sum_gij_pi_pj += g4DD[i][i]*pU[i]*pU[i]
             for j in range(i+1,4):
                 sum_gij_pi_pj += 2*g4DD[i][j]*pU[i]*pU[j]
         discriminant = sum_g0i_pi*sum_g0i_pi - g4DD[0][0]*sum_gij_pi_pj
         answer = (-sum_g0i_pi + sp.sqrt(discriminant)) / g4DD[0][0]
         return answer
```

### 3.g: Symbolic Recipes for Conserved Quantities (E, L, Q)

For geodesic motion in spacetimes with symmetries, certain physical quantities are conserved along the photon's path. These conserved quantities are invaluable for validating the numerical accuracy of our integrator. If the integrator is working correctly, these quantities should remain nearly constant throughout the entire evolution.

The symmetries of a spacetime are described by **Killing vectors**. For the stationary and axisymmetric Kerr spacetime, there are two such vectors, which lead to two conserved quantities:

1. **Energy at Infinity (E):** The symmetry in time (stationarity) leads to the conservation of energy. It is defined as the projection of the 4-momentum onto the time-like Killing vector, which simplifies to:
$$E = -p_t = -g_{t\mu}p^\mu$$
where p_t is the covariant time-component of the 4-momentum.
2. **Angular Momentum Component Parallel to the Axis of Symmetry (L_z):** The symmetry in rotation about the z-axis (axisymmetry) leads to the conservation of the z-component of angular momentum. It is defined as:
$$L_z = p_\phi = g_{\phi\mu}p^\mu$$
In Cartesian coordinates, this is equivalent to the standard definition:
$$L_z = xp_y - yp_x$$
where p_x and p_y are the covariant spatial components of the 4-momentum.
3. **The Carter Constant (Q):** Remarkably, the Kerr spacetime possesses a hidden symmetry related to the separability of the Hamilton-Jacobi equation, which gives rise to a third conserved quantity known as the **Carter Constant, Q**. Its formula is more complex and is a combination of the other conserved quantities and the momentum components. For a photon (mass=0), it is given by:
$$Q = p_\theta^2 + \cos^2\theta\left(\frac{L_z^2}{\sin^2\theta} - a^2E^2\right)$$
In the case of the Schwarzschild spacetime (where the spin $a = 0$), the Carter constant simplifies to the squared total angular momentum: $Q = L_x^2 + L_y^2 + L_z^2 = L^2$.

The following cells define the symbolic recipes for these three conserved quantities, which will be used to generate a C function for monitoring the numerical error of the integrator.

```
[ ]: 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
```

```
[ ]: 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]
```

```
[ ]: 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 = a_spin
    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).")
```

### 3.h: Symbolic Recipes for Numerical Metrics

While analytic metrics like Kerr-Schild are powerful, many modern simulations in astrophysics use numerical metrics, where the spacetime components are known only as numerical values on a discrete grid. To calculate Christoffel symbols for these spacetimes, we must perform all calculations numerically, including the derivatives.

A naive approach would be to first interpolate the metric $g_{\mu\nu}$ to a point, then interpolate again at nearby points to perform a finite difference derivative. This is extremely inefficient, requiring many calls to the interpolation engine. A far superior method is the **"analytic derivative of the interpolant."**

The idea is as follows: 1. The interpolation formula (e.g., trilinear interpolation) is a simple polynomial. 2. We can take the analytical partial derivative of this polynomial formula itself. The result is a new formula for the derivative that is also a simple combination of the grid point values. 3. We can then implement both the interpolation formula and its derivative formula directly in C code. This allows us to compute both $g_{\mu\nu}$ and its derivatives $g_{\mu\nu,\delta}$ in a single, efficient C function call.

To generate the C code for this, we need a symbolic recipe for the Christoffel symbols that is built from abstract placeholders for the interpolated metric and its derivatives. The following function creates this recipe. It uses `sympy.Symbol` objects with descriptive names (e.g., `g4DD00`, `g4DDdD01_d2`) that will be matched to local C variables in the final C worker function. This avoids the "expression swell" that would occur if we tried to symbolically construct the entire interpolation and differentiation process.

```
[ ]: def symbolic_numerical_christoffel_recipe():
        """
        Generates the pure symbolic recipe for the Christoffel symbols assuming
        that the metric g_μν and its derivatives g_μν,δ are provided as inputs.

        This version manually constructs the derivative tensor with a naming
        convention that matches the C code preamble (e.g., g4DDdD01_d2).
        """
        # Step 1: Create symbolic placeholders for the 10 unique metric components.
        g4DD = ixp.declarerank2("g4DD", symmetry="sym01", dimension=4)
```

```
    # --- THIS IS THE CORRECTED LOGIC ---
    # Step 2: Manually create symbolic placeholders for the 40 unique metric derivatives
    # to enforce the g4DDdD{i}{j}_d{k} naming convention.
    g4DDdD = ixp.zerorank3(dimension=4) # Initialize with zeros
    for i in range(4):
        for j in range(i, 4): # Loop over unique metric components
            for k in range(4): # Loop over derivative directions
                # Create the symbol with the exact name we need
                symbol_name = f"g4DDdD{i}{j}_d{k}"
                g4DDdD[i][j][k] = sp.Symbol(symbol_name)
                if i != j:
                    # Enforce symmetry in the symbolic tensor
                    g4DDdD[j][i][k] = g4DDdD[i][j][k]

    # Step 3: Compute the symbolic inverse of the placeholder metric.
    g4UU, _ = ixp.symm_matrix_inverter4x4(g4DD)

    # Step 4: Initialize the output tensor for the Christoffel symbols.
    Gamma4UDD_num_recipe = ixp.zerorank3(dimension=4)

    # Step 5: Build the recipe for the 40 unique Christoffel symbols.
    for alpha in range(4):
        for mu in range(4):
            for nu in range(mu, 4):
                for delta in range(4):
                    Gamma4UDD_num_recipe[alpha][mu][nu] += sp.Rational(1, 2) * g4UU[alpha][delta] * \
                        (g4DDdD[nu][delta][mu] + g4DDdD[mu][delta][nu] - g4DDdD[mu][nu][delta])

    return Gamma4UDD_num_recipe
```

# 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_μ 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_μ 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.

```
[ ]: 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
         a = a_spin

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

[ ]:
```
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

    # 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
```

# 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 act as mathematical blueprints. 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 several key global variables containing symbolic expressions that will be used in the next step to generate the final C code.

[ ]:
```
# --- 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 the GENERIC symbolic RHS expressions for the geodesic equations ---
# This part is unchanged, as the ODEs are generic.
Gamma4UDD_placeholder = ixp.declarerank3("conn->Gamma4UDD", dimension=4)
rhs_pos = geodesic_pos_rhs()
rhs_mom = geodesic_mom_rhs(Gamma4UDD_placeholder)
rhs_length = proper_lengh_rhs()
all_rhs_expressions = rhs_pos + rhs_mom + rhs_length
print(" -> Defined generic global symbolic variable for ODE RHS: all_rhs_expressions")

# --- 4. Generate symbolic recipes for conserved quantities ---
# This is now simplified, as all calculations are Cartesian.
```

```
print(" -> Generating 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

# We now have two lists of expressions, both using Cartesian formulas.
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.")
print("Kerr expressions")
print(list_of_expressions_kerr)
print("Schwarzs experessions")
print(list_of_expressions_schw)
print("\nSymbolic setup complete. All expressions are now available globally.")
```

# 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 several 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 or numerical methods 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. * **Helpers**: These are small, utility functions that perform common tasks, such as managing memory for a data structure.

This modular design allows for maximum code reuse and extensibility.

### 6.a: C Workers for Analytic Metrics

The Python functions in this subsection generate the C "worker" functions that are specialized for a particular analytic metric. Each function takes one of the symbolic metric recipes we generated in Step 5 (e.g., `Gamma4UDD_kerr`) and translates it into an optimized C function.

The core of this process is the `nrpy.c_codegen.c_codegen` function, which converts the large `sympy` expressions into C code, automatically performing Common Subexpression Elimination (CSE) to significantly improve the performance of the final C code. The generated C code is then registered with `nrpy`'s in-memory C project manager, `nrpy.c_function.register_CFunction`.

```
[ ]: 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.")
```

[ ]:
```
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.")
```

[ ]:
```
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.")
```

[ ]:
```
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.b: C Engines and Workers for Numerical Metrics

This subsection generates the more complex C "engines" and "workers" needed to handle numerical metric data. This is where the "analytic derivative of the interpolant" method is implemented in C.

1. `generate_numerical_interpolation_and_deriv_engine()`: This Python function generates the all-in-one C engine

`interpolate_metric_and_derivatives()`. This powerful C function is the heart of the numerical pipeline. It takes the raw numerical data from the in-memory `MetricSliceCache`, performs a full 4D interpolation to find the metric `g_{\mu\nu}` at an arbitrary point, and also computes the analytical derivatives of the interpolation polynomial to find `g_{\mu\nu,\delta}`.

2. `generate_algebraic_christoffel_worker()`: This Python function generates the C worker `calculate_christoffels_from_metric_and_derivs()`. This is a lightweight algebraic function. It takes the 50 metric and derivative values computed by the interpolation engine and plugs them into the symbolic recipe for the Christoffel symbols. `nrpy`'s CSE engine then turns this into a highly efficient block of C code.

[ ]:
```
def generate_numerical_interpolation_and_deriv_engine():
    """
    Generates the all-in-one C engine that performs 4D interpolation of the
    metric and computes its analytical derivatives from the interpolant.
    """
    print(" -> Generating C engine: interpolate_metric_and_derivatives()...")

    includes = ["BHaH_defines.h", "<math.h>"]
    desc = "The core numerical engine for metric interpolation and differentiation."
    name = "interpolate_metric_and_derivatives"
    params = """const MetricSliceCache *restrict cache,
              double t, double x, double y, double z,
              metric_struct *restrict g4DD_out,
              g4DD_deriv_struct *restrict g4DDdD_out"""

    body = r"""
    // Part 1: Temporal Interpolation of Stencil Points
    const metric_slice *slice1 = &cache->slices[1];
    const double grid_x = (x - slice1->x_min) * slice1->inv_dx;
    const double grid_y = (y - slice1->y_min) * slice1->inv_dy;
    const double grid_z = (z - slice1->z_min) * slice1->inv_dz;
    const int i0 = (int)floor(grid_x);
    const int j0 = (int)floor(grid_y);
    const int k0 = (int)floor(grid_z);

    if (i0 < 0 || i0 >= slice1->N_x - 1 || j0 < 0 || j0 >= slice1->N_y - 1 || k0 < 0 || k0 >= slice1->N_z - 1) {
        fprintf(stderr, "Interpolation point out of bounds! (t,x,y,z) = (%.2f, %.2f, %.2f, %.2f)\n", t, x, y, z);
        exit(1);
    }

    const double dx = grid_x - i0;
    const double dy = grid_y - j0;
    const double dz = grid_z - k0;

    const double t0 = cache->times[0], t1 = cache->times[1], t2 = cache->times[2];
    const double L0_t = ((t - t1) * (t - t2)) / ((t0 - t1) * (t0 - t2));
    const double L1_t = ((t - t0) * (t - t2)) / ((t1 - t0) * (t1 - t2));
    const double L2_t = ((t - t0) * (t - t1)) / ((t2 - t0) * (t2 - t1));
    const double dL0_dt = (2*t - t1 - t2) / ((t0 - t1) * (t0 - t2));
    const double dL1_dt = (2*t - t0 - t2) / ((t1 - t0) * (t1 - t2));
    const double dL2_dt = (2*t - t0 - t1) / ((t2 - t0) * (t2 - t1));

    double p[10][8];
    const metric_slice *slices = cache->slices;
    const double *comp_data[10][3] = {
        {slices[0].g00_data, slices[1].g00_data, slices[2].g00_data}, {slices[0].g01_data, slices[1].g01_data, slices[2].g01_data},
        {slices[0].g02_data, slices[1].g02_data, slices[2].g02_data}, {slices[0].g03_data, slices[1].g03_data, slices[2].g03_data},
        {slices[0].g11_data, slices[1].g11_data, slices[2].g11_data}, {slices[0].g12_data, slices[1].g12_data, slices[2].g12_data},
        {slices[0].g13_data, slices[1].g13_data, slices[2].g13_data}, {slices[0].g22_data, slices[1].g22_data, slices[2].g22_data},
        {slices[0].g23_data, slices[1].g23_data, slices[2].g23_data}, {slices[0].g33_data, slices[1].g33_data, slices[2].g33_data}
    };

    for (int i_s = 0; i_s <= 1; ++i_s) {
        for (int j_s = 0; j_s <= 1; ++j_s) {
            for (int k_s = 0; k_s <= 1; ++k_s) {
                const int corner_idx = 4*k_s + 2*j_s + i_s;
                 const int grid_idx_s0 = NUMERICAL_METRIC_IDX3(&slices[0], i0+i_s, j0+j_s, k0+k_s);
                const int grid_idx_s1 = NUMERICAL_METRIC_IDX3(&slices[1], i0+i_s, j0+j_s, k0+k_s);
                const int grid_idx_s2 = NUMERICAL_METRIC_IDX3(&slices[2], i0+i_s, j0+j_s, k0+k_s);
                for(int comp=0; comp<10; ++comp) {
                    p[comp][corner_idx] = L0_t * comp_data[comp][0][grid_idx_s0] + L1_t * comp_data[comp][1][grid_idx_s1] + L2_t *␣
↪comp_data[comp][2][grid_idx_s2];
                }
            }
        }
    }


    // Part 2: Spatial Interpolation and Differentiation
    const double mdx = 1.0 - dx, mdy = 1.0 - dy, mdz = 1.0 - dz;
    double val[10], ddx[10], ddy[10], ddz[10];

    for (int i = 0; i < 10; ++i) {
        const double c000=p[i][0], c100=p[i][1], c010=p[i][2], c110=p[i][3];
        const double c001=p[i][4], c101=p[i][5], c011=p[i][6], c111=p[i][7];

        const double c00 = c000*mdx + c100*dx; const double c10 = c010*mdx + c110*dx;
        const double c01 = c001*mdx + c101*dx; const double c11 = c011*mdx + c111*dx;
        const double c0 = c00*mdy + c10*dy;   const double c1 = c01*mdy + c11*dy;
        val[i] = c0*mdz + c1*dz;
        const double d_dx_0 = (c100-c000)*mdy + (c110-c010)*dy;
        const double d_dx_1 = (c101-c001)*mdy + (c111-c011)*dy;
        ddx[i] = (d_dx_0*mdz + d_dx_1*dz) * slice1->inv_dx;

        const double d_dy_0 = (c010-c000)*mdx + (c110-c100)*dx;
        const double d_dy_1 = (c011-c001)*mdx + (c111-c101)*dx;
        ddy[i] = (d_dy_0*mdz + d_dy_1*dz) * slice1->inv_dy;

        ddz[i] = (c1 - c0) * slice1->inv_dz;
    }

    // Part 3: Temporal Derivatives
    double ddt[10];
    for (int i_s = 0; i_s <= 1; ++i_s) {
        for (int j_s = 0; j_s <= 1; ++j_s) {
            for (int k_s = 0; k_s <= 1; ++k_s) {
                const int corner_idx = 4*k_s + 2*j_s + i_s;
                const int grid_idx_s0 = NUMERICAL_METRIC_IDX3(&slices[0], i0+i_s, j0+j_s, k0+k_s);
                const int grid_idx_s1 = NUMERICAL_METRIC_IDX3(&slices[1], i0+i_s, j0+j_s, k0+k_s);
                const int grid_idx_s2 = NUMERICAL_METRIC_IDX3(&slices[2], i0+i_s, j0+j_s, k0+k_s);
                for(int comp=0; comp<10; ++comp) {
                    p[comp][corner_idx] = dL0_dt * comp_data[comp][0][grid_idx_s0] + dL1_dt * comp_data[comp][1][grid_idx_s1] + dL2_dt␣
↪* comp_data[comp][2][grid_idx_s2];
                }
            }
        }
    }
    for (int i = 0; i < 10; ++i) {
        const double c000=p[i][0], c100=p[i][1], c010=p[i][2], c110=p[i][3];
        const double c001=p[i][4], c101=p[i][5], c011=p[i][6], c111=p[i][7];
        const double c00 = c000*mdx + c100*dx; const double c10 = c010*mdx + c110*dx;
        const double c01 = c001*mdx + c101*dx; const double c11 = c011*mdx + c111*dx;
        const double c0 = c00*mdy + c10*dy;   const double c1 = c01*mdy + c11*dy;
        ddt[i] = c0*mdz + c1*dz;
    }

    // Part 4: Final Assignment
    g4DD_out->g00=val[0]; g4DD_out->g01=val[1]; g4DD_out->g02=val[2]; g4DD_out->g03=val[3];
    g4DD_out->g11=val[4]; g4DD_out->g12=val[5]; g4DD_out->g13=val[6]; g4DD_out->g22=val[7];
    g4DD_out->g23=val[8]; g4DD_out->g33=val[9];

    const double *v[4] = {ddt, ddx, ddy, ddz};
    double *d[40] = {&g4DDdD_out->g00d0, &g4DDdD_out->g00d1, &g4DDdD_out->g00d2, &g4DDdD_out->g00d3,
                     &g4DDdD_out->g01d0, &g4DDdD_out->g01d1, &g4DDdD_out->g01d2, &g4DDdD_out->g01d3,
                      &g4DDdD_out->g02d0, &g4DDdD_out->g02d1, &g4DDdD_out->g02d2, &g4DDdD_out->g02d3,
                      &g4DDdD_out->g03d0, &g4DDdD_out->g03d1, &g4DDdD_out->g03d2, &g4DDdD_out->g03d3,
                      &g4DDdD_out->g11d0, &g4DDdD_out->g11d1, &g4DDdD_out->g11d2, &g4DDdD_out->g11d3,
                      &g4DDdD_out->g12d0, &g4DDdD_out->g12d1, &g4DDdD_out->g12d2, &g4DDdD_out->g12d3,
                      &g4DDdD_out->g13d0, &g4DDdD_out->g13d1, &g4DDdD_out->g13d2, &g4DDdD_out->g13d3,
                      &g4DDdD_out->g22d0, &g4DDdD_out->g22d1, &g4DDdD_out->g22d2, &g4DDdD_out->g22d3,
                      &g4DDdD_out->g23d0, &g4DDdD_out->g23d1, &g4DDdD_out->g23d2, &g4DDdD_out->g23d3,
                      &g4DDdD_out->g33d0, &g4DDdD_out->g33d1, &g4DDdD_out->g33d2, &g4DDdD_out->g33d3};
    for(int comp=0; comp<10; ++comp) for(int dir=0; dir<4; ++dir) *d[4*comp+dir] = v[dir][comp];
"""

    cfc.register_CFunction(
        includes=includes, desc=desc, name=name, params=params, body=body
    )
    print(f"    ... Registered C engine: {name}().")
```

[ ]:
```
def generate_algebraic_christoffel_worker():
    """
    Generates the C worker that computes Christoffel symbols from pre-computed
    metric and metric derivative values. This is a pure algebraic function.
    """
    print(" -> Generating C algebraic engine: calculate_christoffels_from_metric_and_derivs()...")

    # Step 1: Get the symbolic recipe.
    Gamma4UDD_num_recipe = symbolic_numerical_christoffel_recipe()

    # Step 2: Prepare lists for C code generation.
    list_of_Gamma_C_vars = []
    list_of_Gamma_syms = []
    conn_Gamma4UDD = ixp.declarerank3("conn_out->Gamma4UDD", dimension=4)
    for alpha in range(4):
        for mu in range(4):
            for nu in range(mu, 4):
                list_of_Gamma_C_vars.append(str(conn_Gamma4UDD[alpha][mu][nu]))
                list_of_Gamma_syms.append(Gamma4UDD_num_recipe[alpha][mu][nu])

    # Step 3: Generate the C preamble to unpack structs into local variables.
    preamble = "// Unpack input structs into local variables that match symbolic recipe.\n"
    g_components = [f"g{i}{j}" for i in range(4) for j in range(i, 4)]
    for i in range(4):
        for j in range(i, 4):
            preamble += f"    const double g4DD{i}{j} = g4DD_in->g{i}{j};\n"

    for i in range(4):
        for j in range(i, 4):
            for k in range(4):
                preamble += f"    const double g4DDdD{i}{j}_d{k} = g4DDdD_in->g{i}{j}d{k};\n"

    # Step 4: Generate the computational kernel.
    kernel_C_code = ccg.c_codegen(
        list_of_Gamma_syms,
        list_of_Gamma_C_vars,
        enable_cse=True,
        cse_varprefix="num_conn_intermed"
    )

    # Step 5: Assemble and register the function.
    body = preamble + kernel_C_code

    includes = ["BHaH_defines.h"]
    desc = "Computes Christoffel symbols from pre-interpolated metric and derivative values."
    name = "calculate_christoffels_from_metric_and_derivs"
    params = """const metric_struct *restrict g4DD_in,
              const g4DD_deriv_struct *restrict g4DDdD_in,
              connection_struct *restrict conn_out"""

    cfc.register_CFunction(
        includes=includes, desc=desc, name=name, params=params, body=body
    )
    print(f"    ... Registered C algebraic engine: {name}().")
```

### 6.c: Generic Physics and Integration Engines

The Python functions in this subsection generate the generic C “engines” that are valid for *any* metric, whether analytic or numerical. They operate on abstract data structures (like `metric_struct` and `connection_struct`) and are completely decoupled from the specifics of how the values in those structs were computed.

This is a key feature of the project’s modular design. For example, the `calculate_ode_rhs()` engine doesn’t care if the Christoffel symbols it receives were calculated from the analytic Kerr-Schild formula or the numerical interpolation pipeline; it just applies the geodesic equation to whatever values it is given. This allows for maximum code reuse.

[ ]:
```
def calculate_p0_reverse():
    """
    Generates and registers the C function to compute the time component
    of the reverse 4-momentum, p^0.
    """
    print(" -> Generating C engine function: calculate_p0_reverse()...")
    # The symbolic expression uses y[4] through y[7] for the 4-momentum
    p0_expr = mom_time_p0_reverse()

    includes = ["BHaH_defines.h"]
    desc = r"""@brief Computes reverse-time p^0 from the null condition g_munu p^mu p^nu = 0."""
    name = "calculate_p0_reverse"
    c_type = "double"
    # The function now takes the full 9-element state vector y.
    params = "const metric_struct *restrict metric, const double y[9]"



    # We generate the C code directly from the original expression.
    # Since the C function takes the full y[9] vector, the array indices
    # y[4], y[5], etc., in the generated code will be correct.
    p0_C_code_lines = ccg.c_codegen(
        p0_expr, 'double p0_val', enable_cse=True, include_braces=False
    )
    body = f"{{\n{p0_C_code_lines}\nreturn p0_val;\n}}"
    cfc.register_CFunction(
        includes=includes, desc=desc, cfunc_type=c_type,
        name=name, params=params, body=body
    )
    print("    ... calculate_p0_reverse() registration complete.")
```

### Generic Engine: `check_conservation()`

This Python function generates the C engine `check_conservation()`. Its purpose is to calculate the conserved quantities (Energy `E`, the three components of angular momentum `L_i`, and the Carter Constant `Q`) for a given state vector `y`.

This function is an excellent example of a **validation tool**. During a long integration, small numerical errors will inevitably accumulate. By calling this function periodically, we can monitor how well these physical quantities, which should be perfectly constant, are actually being conserved by our numerical solver. If they drift significantly, it indicates a problem with the integration accuracy (e.g., the step size is too large or the order of the integrator is too low).

The C function is a dispatcher that operates on the *symbolic recipes* for the conserved quantities that we generated in Step 3.g. It takes the metric type as input and uses a `switch` statement to select the correct set of symbolic formulas (`list_of_expressions_kerr` or `list_of_expressions_schw`). It then calls `nrpy.c_codegen.c_codegen` to translate these high-level symbolic recipes into optimized C code on the fly.

```
[ ]: def check_conservation():
        """
        Generates the C function `check_conservation`. This version is simplified
        for a purely Cartesian pipeline.
        """
        print(" -> Generating C engine: check_conservation() [Cartesian Version]...")

        # Use the globally defined Cartesian recipes
        output_vars_kerr = ["*E", "*Lx", "*Ly", "*Lz", "*Q"]
        output_vars_schw = ["*E", "*Lx", "*Ly", "*Lz", "*Q"] # Q is L^2

        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 state vector."""
    name = "check_conservation"
    params = """const commondata_struct *restrict commondata,
        const params_struct *restrict params,
        const metric_params *restrict metric_params_in,
        const double y[9],
        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;

    // Declare a POINTER to a metric_struct and allocate memory for it.
    metric_struct* metric = (metric_struct*)malloc(sizeof(metric_struct));

    // Call the dispatcher to fill the allocated struct with metric components at the given state y.
    g4DD_metric(commondata, params, metric_params_in, y, metric);

    // --- MODIFIED: Simplified logic for Cartesian-only checks ---
    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.")
```

### Generic Engine: `radiative_transfer_engine()`

This function generates the core C physics engine `calculate_radiative_transfer()`. This function implements the relativistic radiative transfer equation for an optically thin emitter, which connects the properties of the emitting gas to the light seen by a distant observer.

It takes as input the photon's covariant 4-momentum (p_μ) at the point of emission, the fluid's covariant 4-velocity (u_μ) at that same point, and the fluid's intrinsic emissivity (j_{int}) and rest-frame emission wavelength (λ_{rest}).

The function then calculates the **redshift factor** (often denoted as `g` or `D`), which accounts for both gravitational redshift and the relativistic Doppler effect. This factor is the ratio of the energy of the photon as measured by the distant observer (E_{obs}) to the energy of the photon in the rest-frame of the emitting gas (E_{emit}):

$$g = \frac{E_{obs}}{E_{emit}} = \frac{(-p_\mu u^\mu)_{obs}}{(-p_\mu u^\mu)_{emit}}$$

For a distant, stationary observer, their 4-velocity is simply $u^\mu_{obs} = (1, 0, 0, 0)$, which simplifies the numerator to $E_{obs} = p_t$. The denominator is the full dot product of the photon's and the fluid's 4-momenta.

Finally, it computes the observed physical quantities: * **Observed Intensity:** For an optically thin source, the intensity scales as the cube of the redshift factor: $I_{obs} = j_{int} \cdot g^3$. * **Observed Wavelength:** The wavelength is directly shifted by the redshift factor: $\lambda_{obs} = \lambda_{rest}/g$.

These final values are what are used to color the pixels in the final rendered image.

[ ]:
```
def radiative_transfer_engine():
    """
    Generates the C engine for calculating the final observed intensity and
    wavelength based on the interpolated disk state and photon momentum.
    """
    print(" -> Generating C engine for radiative transfer physics...")

    includes = ["BHaH_defines.h", "<math.h>"]
    desc = r"""@brief Calculates the observed intensity and wavelength from the disk and photon state."""
    name = "calculate_radiative_transfer"
    params = r"""
    const double photon_p_mu[4], const double disk_u_mu[4],
    const float disk_j_intrinsic, const double disk_lambda_rest,
    double *stokes_I, double *lambda_observed
    """
    body = r"""
    // The observer is assumed to be at rest in the coordinate frame far away,
    // so their 4-velocity is u_obs^mu = (1, 0, 0, 0).
    // The metric is Minkowski far away, so u_obs_mu = (-1, 0, 0, 0).
    // The photon momentum is p_mu.
    // Therefore, (-p_mu u^mu)_obs = - (p_0 * -1) = p_0.
    // NOTE: The photon momentum p_mu must be covariant (lower-indexed).
    const double p_mu_u_mu_obs = photon_p_mu[0];

    // Calculate (-p_mu u^mu)_disk
    const double p_mu_u_mu_disk = - (photon_p_mu[0] * disk_u_mu[0] +
                                     photon_p_mu[1] * disk_u_mu[1] +
                                     photon_p_mu[2] * disk_u_mu[2] +
                                     photon_p_mu[3] * disk_u_mu[3]);

    // Doppler factor D = E_obs / E_disk = (-p_mu u^mu)_obs / (-p_mu u^mu)_disk
    const double doppler_factor = p_mu_u_mu_obs / p_mu_u_mu_disk;
    // Observed intensity I_obs = j_intrinsic * D^3
    *stokes_I = disk_j_intrinsic * doppler_factor * doppler_factor * doppler_factor;

    // Observed wavelength lambda_obs = lambda_rest / D
    *lambda_observed = disk_lambda_rest / doppler_factor;
"""
    cfc.register_CFunction(includes=includes, desc=desc, name=name, params=params, body=body)
    print("    ... Registered C engine: calculate_radiative_transfer.")
```

[ ]:
```
def calculate_ode_rhs():

    rhs_output_vars = [f"rhs_out[{i}]" for i in range(9)]




    includes = ["BHaH_defines.h"]

    desc = r"""@brief Calculates the right-hand sides (RHS) of the 9 geodesic ODEs.

    This function implements the generic geodesic equation using pre-computed
    Christoffel symbols. It is a pure "engine" function that does not depend
    on any specific metric's parameters (like M_scale), only on the geometric
    values passed to it via the connection struct.

    @param[in]  y         The 9-component state vector [t, r, th, ph, p^t, p^r, p^th, p^ph, L].
    @param[in]  conn      A pointer to the connection_struct holding the pre-computed Christoffel symbols.
    @param[out] rhs_out   A pointer to the 9-component output array where the RHS results are stored."""

    name = "calculate_ode_rhs"
    params = "const double y[9], const metric_struct *restrict metric, const connection_struct *restrict conn, double rhs_out[9]"

    body=ccg.c_codegen(all_rhs_expressions,rhs_output_vars)

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

### 6.e: `find_event_time_and_state()` Interpolation Engine

This Python function generates a crucial C **engine** called `find_event_time_and_state()`. Its purpose is to find the precise time and state vector of a “plane-crossing” event with high accuracy, using data from three consecutive steps of the ODE integrator. This is essential for accurately mapping where a ray hits the window and source planes.

The function implements a robust interpolation scheme: 1. **Quadratic Root Finding:** It treats the event condition (e.g., the distance to a plane, `f(y) = n_i x^i - d = 0`) as a function of the affine parameter, `f(κ)`. Given three points `(κ_prev, f_prev)`, `(κ_curr, f_curr)`, and `(κ_next, f_next)` that are known to bracket a root (i.e., the function changes sign), it fits a quadratic polynomial to these points. It then uses a numerically stable formula (similar to Muller's method) to find the root `κ_event` of this polynomial. This gives a much more accurate time for the plane crossing than simply taking the time of the closest step. 2. **Lagrange Polynomial Interpolation:** Once the precise event time `κ_event` is known, the function uses second-order Lagrange basis polynomials to interpolate each of the 9 components of the state vector `y` to that exact time.

This two-step process provides a highly accurate snapshot of the photon's state `y_event` at the exact moment it crosses a plane of interest. The C function body is written manually as a string, as its logic is algorithmic rather than symbolic, and then registered with `nrpy`.

-1.0.6 `nrpy` Functions Used in this Cell:

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

```
[ ]: def lagrange_interp_engine_generic():
         """
         Generates the generic Lagrange interpolation engine.

         This definitive version is numerically robust. It checks for small denominators
         and unstable conditions, falling back to stable linear interpolation to prevent
         NaN results in edge cases.
         """
         print(" -> Generating C engine: find_event_time_and_state() [Robust Version]...")

         includes = ["BHaH_defines.h", "<math.h>"]
         desc = r"""@brief Finds the root of a generic event using a robust, second-order interpolation."""

         name = "find_event_time_and_state"
         params = r"""const double y_prev[9], const double y_curr[9], const double y_next[9],
                     double lambda_prev, double lambda_curr, double lambda_next,
                     event_function_t event_func, void *event_params,
                     event_data_struct *restrict result"""

         body = r"""
         double t0 = lambda_prev, t1 = lambda_curr, t2 = lambda_next;
         double f0 = event_func(y_prev, event_params);
         double f1 = event_func(y_curr, event_params);
         double f2 = event_func(y_next, event_params);

         // --- Linear interpolation as a fallback ---
         // This is used if quadratic interpolation is unstable or fails.
         // It finds the root between the two points where the sign change occurs.
         double t_linear;
         if (f0 * f1 < 0.0 && fabs(f1 - f0) > 1e-12) { // Sign change is between prev and curr
```

```
        t_linear = (f1 * t0 - f0 * t1) / (f1 - f0);
    } else if (f1 * f2 < 0.0 && fabs(f2 - f1) > 1e-12) { // Sign change is between curr and next
        t_linear = (f2 * t1 - f1 * t2) / (f2 - f1);
    } else {
        // This can happen if f1 is exactly zero.
        t_linear = t1;
    }

    // --- Quadratic interpolation (Muller's method variant) ---
    double h0 = t1 - t0;
    double h1 = t2 - t1;

    // Check for degenerate intervals to prevent division by zero.
    if (fabs(h0) < 1e-15 || fabs(h1) < 1e-15 || fabs(h0 + h1) < 1e-15) {
        result->lambda_event = t_linear;
    } else {
        double delta0 = (f1 - f0) / h0;
        double delta1 = (f2 - f1) / h1;
        double a = (delta1 - delta0) / (h1 + h0);
        double b = a * h1 + delta1;
        double c = f2;
        double discriminant = b*b - 4*a*c;

        if (discriminant < 0.0 || fabs(a) < 1e-15) {
            // Discriminant is negative or equation is effectively linear.
            result->lambda_event = t_linear;
        } else {
            // Use the more stable form of the quadratic formula
            double denom = (b >= 0.0) ? (b + sqrt(discriminant)) : (b - sqrt(discriminant));
            if (fabs(denom) < 1e-15) {
                result->lambda_event = t_linear;
            } else {
                double t_quad = t2 - (2.0 * c / denom);
                // Only accept the quadratic result if it's within the bracketing interval.
                if (t_quad > fmin(t0, t2) && t_quad < fmax(t0, t2)) {
                    result->lambda_event = t_quad;
                } else {
                    result->lambda_event = t_linear;
                }
            }
        }
    }

    // --- Perform final interpolation on the state vector using the found time ---
    double t = result->lambda_event;

    // Check for degenerate intervals again before final interpolation
    if (fabs(t0 - t1) < 1e-15 || fabs(t0 - t2) < 1e-15 || fabs(t1 - t2) < 1e-15) {
        // Fallback to linear interpolation for the state vector as well
        double frac = 0.5;
        if (fabs(t2 - t1) > 1e-15) {
            frac = (t - t1) / (t2 - t1);
        }
        for (int i = 0; i < 9; i++) {
            result->y_event[i] = y_curr[i] + frac * (y_next[i] - y_curr[i]);
        }
    } else {
        // Perform full quadratic interpolation
        double L0 = ((t - t1) * (t - t2)) / ((t0 - t1) * (t0 - t2));
        double L1 = ((t - t0) * (t - t2)) / ((t1 - t0) * (t1 - t2));
        double L2 = ((t - t0) * (t - t1)) / ((t2 - t0) * (t2 - t1));
        for (int i = 0; i < 9; i++) {
            result->y_event[i] = y_prev[i] * L0 + y_curr[i] * L1 + y_next[i] * L2;
        }
    }

    result->t_event = result->y_event[0];
    result->found = true;
    """
    cfc.register_CFunction(includes=includes, desc=desc, name=name, params=params, body=body)
    print("    ... Registered C engine: find_event_time_and_state (Robust Version).")
```

# 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**: These are functions that contain a `switch` statement to select the correct "worker" function based on the chosen metric (e.g., `Schwarzschild` vs. `Kerr` vs. `Numerical`).
- **Orchestrators**: These 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.
- **Helpers**: These are small utility functions that manage resources, like loading data or sorting filenames.

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

This Python function generates the C function `g4DD_metric()`, which serves as a high-level **dispatcher** for the **analytic metrics**. 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. It contains cases for the analytic spacetimes (`Kerr`, `Schwarzschild`, etc.) and calls the appropriate worker function (e.g., `g4DD_kerr_schild()`).

This modular approach cleanly separates the control flow (deciding *which* analytic metric to use) from the physics implementation (the worker functions that know *how* to compute a specific metric). Note that this dispatcher is **not** used by the numerical metric pipeline, which has its own dedicated interpolation engine.

```
[ ]: 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[9], 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** for the **analytic metrics**. 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 for the analytic spacetimes. This design is highly extensible: adding a new analytic metric simply requires writing a new worker function for its Christoffel symbols and adding a new `case` to this `switch` statement. This function is not used by the numerical metric pipeline.

[ ]:
```
def connections():
    """
    Generates and registers the C dispatcher for Christoffel symbols.
    """
    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[9], 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.")
```

### 7.c: `set_initial_conditions_cartesian()` Orchestrator

This function generates the C **orchestrator** `set_initial_conditions_cartesian()`. This function is responsible for setting the complete initial state vector `y_out[9]` for a single light ray. It orchestrates a sequence of calculations to do this.

The process for setting the initial state `y = (t, x, y, z, p^t, p^x, p^y, p^z, L)` is as follows:

1. **Set Initial Position**: The initial spatial coordinates `(x, y, z)` are set to the camera's location, `camera_pos`. The initial time `t` and path length `L` are set to `0.0`.
2. **Calculate Aiming Vector**: It computes the aiming vector `V`, which points from the camera to a specific target pixel on the window plane: `V = target_pos - camera_pos`.
3. **Set Initial Spatial Momentum**: As derived in the introduction, the initial reverse-time spatial momentum `(p^x, p^y, p^z)` must be parallel to the aiming vector `V`. It is therefore set to the normalized aiming vector: `p^i = V^i / |V|`.
4. **Calculate Initial Time Momentum**: With the spatial components of the momentum set, the final unknown is the time component, $p^t = p^0$. This requires a call to the physics engines:
    - First, it calls the `g4DD_metric()` dispatcher to compute the metric components $g_{\mu\nu}$ at the camera's location.
    - Then, it passes these metric components and the partially-filled state vector `y_out` to the `calculate_p0_reverse()` engine, which solves the null condition $g_{\mu\nu}p^\mu p^\nu = 0$ for p^0.
    - The result is stored in `y_out[4]`, completing the initial state vector.

```
[ ]: def set_initial_conditions_cartesian():
        """
        Generates the C engine to set the initial state vector, now entirely in
        Cartesian coordinates.
        """
        print(" -> Generating C engine: set_initial_conditions_cartesian()...")

        includes = ["BHaH_defines.h", "BHaH_function_prototypes.h"]
        desc = r"""@brief Sets the full initial state for a ray in Cartesian coordinates."""

        name = "set_initial_conditions_cartesian"
        params = """const commondata_struct *restrict commondata, const params_struct *restrict params,
                    const metric_params *restrict metric,
                    const double camera_pos[3], const double target_pos[3],
                    double y_out[9]"""

        body = r"""
```

```
    // --- Step 1: Set the initial position to the camera's location ---
    y_out[0] = 0.0; // t
    y_out[1] = camera_pos[0]; // x
    y_out[2] = camera_pos[1]; // y
    y_out[3] = camera_pos[2]; // z
    y_out[8] = 0.0; // L (integrated path length)

    // --- Step 2: Calculate the aiming vector V and set spatial momentum ---
    const double V_x = target_pos[0] - camera_pos[0];
    const double V_y = target_pos[1] - camera_pos[1];
    const double V_z = target_pos[2] - camera_pos[2];
    const double mag_V = sqrt(V_x*V_x + V_y*V_y + V_z*V_z);

    // The reverse-momentum p is parallel to the aiming vector V.
    if (mag_V > 1e-12) {
        y_out[5] = V_x / mag_V; // p^x
        y_out[6] = V_y / mag_V; // p^y
        y_out[7] = V_z / mag_V; // p^z
    } else {
        // Should not happen in production, but as a fallback, set a default momentum.
        y_out[5] = 1.0; y_out[6] = 0.0; y_out[7] = 0.0;
    }

    // --- Step 3: Calculate the time component p^t using the null condition ---
    metric_struct g4DD;
    // Note: The g4DD_metric function needs the first 4 elements of y_out (the coordinates).
    g4DD_metric(commondata, params, metric, y_out, &g4DD);

    // The state vector y is (t,x,y,z, p^t,p^x,p^y,p^z, L).
    y_out[4] = calculate_p0_reverse(&g4DD, y_out);
    """

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

### 7.d: `event_detection_manager()` Orchestrator

This Python function generates the C orchestrator `event_detection_manager()`. This function is called at each step of the integration for photons that have not yet terminated. Its job is to check if the photon has crossed one of the predefined geometric surfaces: the camera's **window plane** or the fallback **source plane**.

The logic is purely geometric. For each plane, it knows the photon's position at three consecutive substeps (`y_prev`, `y_curr`, `y_next`). It determines which side of the plane the photon is on at the start and end of the full step. If the photon has changed sides, a crossing has occurred.

When a crossing is detected, this orchestrator calls the `find_event_time_and_state()` engine to perform a high-accuracy interpolation, finding the precise state of the photon at the exact moment of intersection. This event data is then stored for later processing by the finalizer. This function is a key part of the "event cascade," being called only after the higher-priority disk intersection checks have been performed.

```
[ ]: def event_detection_manager():
         """
         Generates the C event detection manager.

         This final version is a pure, stateless plane-crossing detector. It takes
         the previous state of the photon (which side of the plane it was on) and
         updates the event_data_struct if a crossing has occurred.
         """
         print(" -> Generating C event detection manager (Stateless Plane Detector Version)...")

         includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "<math.h>"]


         desc = r"""@brief Detects crossings of the window and source planes."""
         name = "event_detection_manager"
         cfunc_type = "void"
         params = r"""
             const double y_prev[9], const double y_curr[9], const double y_next[9],
             double lambda_prev, double lambda_curr, double lambda_next,
             const commondata_struct *restrict commondata,
             bool *on_positive_side_of_window_prev,
             bool *on_positive_side_of_source_prev,
             event_data_struct *restrict window_event,
             event_data_struct *restrict source_plane_event
             """

         body = r"""
         // --- Window Plane Detection ---
         // This logic is only performed if the caller has not already found the event.
         if (!window_event->found) {
             double window_plane_normal[3] = {commondata->window_center_x - commondata->camera_pos_x,
                                              commondata->window_center_y - commondata->camera_pos_y,
                                              commondata->window_center_z - commondata->camera_pos_z};
             const double mag_w_norm = sqrt(SQR(window_plane_normal[0]) + SQR(window_plane_normal[1]) + SQR(window_plane_normal[2]));
             if (mag_w_norm > 1e-12) {
                 const double inv_mag_w_norm = 1.0 / mag_w_norm;
                 for(int i=0;i<3;i++) window_plane_normal[i] *= inv_mag_w_norm;
             }
             const double window_plane_dist = commondata->window_center_x * window_plane_normal[0] +
                                              commondata->window_center_y * window_plane_normal[1] +
                                              commondata->window_center_z * window_plane_normal[2];

             plane_event_params window_params = {{window_plane_normal[0], window_plane_normal[1], window_plane_normal[2]},␣
     ↪window_plane_dist};
```

```
        bool on_positive_side_curr = (plane_event_func(y_next, &window_params) > 0);
        if (on_positive_side_curr != *on_positive_side_of_window_prev) {
            find_event_time_and_state(y_prev, y_curr, y_next, lambda_prev, lambda_curr, lambda_next,
                                      plane_event_func, &window_params, window_event);
        }
        *on_positive_side_of_window_prev = on_positive_side_curr;
    }

    // --- Source Plane Detection ---
    // This logic is only performed if the caller has not already found the event.
    if (!source_plane_event->found) {
        const double source_plane_normal[3] = {commondata->source_plane_normal_x,
                                               commondata->source_plane_normal_y,
                                               commondata->source_plane_normal_z};
        const double source_plane_dist = commondata->source_plane_center_x * source_plane_normal[0] +
                                         commondata->source_plane_center_y * source_plane_normal[1] +
                                         commondata->source_plane_center_z * source_plane_normal[2];

        plane_event_params source_params = {{source_plane_normal[0], source_plane_normal[1], source_plane_normal[2]},␣
↪source_plane_dist};
        bool on_positive_side_curr = (plane_event_func(y_next, &source_params) > 0);
        if (on_positive_side_curr != *on_positive_side_of_source_prev) {
            find_event_time_and_state(y_prev, y_curr, y_next, lambda_prev, lambda_curr, lambda_next,
                                      plane_event_func, &source_params, source_plane_event);
        }
        *on_positive_side_of_source_prev = on_positive_side_curr;
    }
    """
    cfc.register_CFunction(includes=includes, desc=desc, cfunc_type=cfunc_type, name=name, params=params, body=body)
    print("    ... Registered event_detection_manager (Stateless Plane Detector Version).")
```

### 7.e: `handle_source_plane_intersection()` Engine

This function generates the C engine that processes a fallback **source plane** intersection event. It is called by the main integration loop when the `event_detection_manager` reports that a photon has hit the source plane.

Its responsibilities are: 1. **Coordinate Transformation**: It takes the 3D Cartesian intersection point `(x, y, z)` from the event data. It then projects this point onto the source plane's own local, 2D orthonormal basis to calculate the texture coordinates `(y_s, z_s)`. 2. **Bounds Checking**: It checks if the calculated planar radius `r_s = sqrt(y_s^2 + z_s^2)` is within the user-defined active region (`source_r_min`, `source_r_max`). 3. **Populate Blueprint**: If the hit is within the valid region, it populates the relevant fields in the `blueprint_data_t` struct and returns `true`. Otherwise, it returns `false`, and the photon continues its integration.

```
[ ]: def handle_source_plane_intersection_engine():
         """
         Generates a C engine that handles a source plane intersection.
         This version is reverted to be compatible with the v11.7 termination_type_t enum.
         """
```

```
    print(" -> Generating C engine: handle_source_plane_intersection (v11.7 compatible)...")

    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "<math.h>", "<stdbool.h>"]
    desc = r"""@brief Handles a source plane intersection by checking bounds and populating the blueprint."""
    name = "handle_source_plane_intersection"
    cfunc_type = "bool"
    params = r"""
    const event_data_struct *restrict source_plane_event,
    const commondata_struct *restrict commondata,
    blueprint_data_t *restrict final_blueprint_data
    """

    body = r"""
    // --- Calculate the local (y_s, z_s) coordinates on the plane ---
    const double intersection_pos[3] = {source_plane_event->y_event[1], source_plane_event->y_event[2], source_plane_event->y_event[3]};
    const double source_plane_center[3] = {commondata->source_plane_center_x, commondata->source_plane_center_y,␣
↪commondata->source_plane_center_z};
    const double source_plane_normal[3] = {commondata->source_plane_normal_x, commondata->source_plane_normal_y,␣
↪commondata->source_plane_normal_z};
    const double source_up_vector[3] = {commondata->source_up_vec_x, commondata->source_up_vec_y, commondata->source_up_vec_z};

    // Construct orthonormal basis vectors for the source plane
    double s_z[3] = {source_plane_normal[0], source_plane_normal[1], source_plane_normal[2]};
    double s_x[3] = {source_up_vector[1]*s_z[2] - source_up_vector[2]*s_z[1],
                     source_up_vector[2]*s_z[0] - source_up_vector[0]*s_z[2],
                     source_up_vector[0]*s_z[1] - source_up_vector[1]*s_z[0]};
    double mag_s_x = sqrt(SQR(s_x[0]) + SQR(s_x[1]) + SQR(s_x[2]));

    if (mag_s_x < 1e-9) {
        double alternative_up[3] = {1.0, 0.0, 0.0};
        if (fabs(s_z[0]) > 0.999) {
            alternative_up[0] = 0.0;
            alternative_up[1] = 1.0;
        }
        s_x[0] = alternative_up[1]*s_z[2] - alternative_up[2]*s_z[1];
        s_x[1] = alternative_up[2]*s_z[0] - alternative_up[0]*s_z[2];
        s_x[2] = alternative_up[0]*s_z[1] - alternative_up[1]*s_z[0];
        mag_s_x = sqrt(SQR(s_x[0]) + SQR(s_x[1]) + SQR(s_x[2]));
    }

    for(int i=0; i<3; i++) s_x[i] /= mag_s_x;

    double s_y[3] = {s_z[1]*s_x[2] - s_z[2]*s_x[1],
                     s_z[2]*s_x[0] - s_z[0]*s_x[2],
                      s_z[0]*s_x[1] - s_z[1]*s_x[0]};

    const double vec_s[3] = {intersection_pos[0] - source_plane_center[0],
                             intersection_pos[1] - source_plane_center[1],
                             intersection_pos[2] - source_plane_center[2]};

    const double y_s = vec_s[0]*s_x[0] + vec_s[1]*s_x[1] + vec_s[2]*s_x[2];
    const double z_s = vec_s[0]*s_y[0] + vec_s[1]*s_y[1] + vec_s[2]*s_y[2];

    const double r_s_sq = SQR(y_s) + SQR(z_s);

    if (r_s_sq >= SQR(commondata->source_r_min) && r_s_sq <= SQR(commondata->source_r_max)) {
        // This is a valid hit. Populate the blueprint and return true.
        // *** REVERTED: Use the correct enum member from termination_type_t ***
        final_blueprint_data->termination_type = TERMINATION_TYPE_SOURCE_PLANE;
        final_blueprint_data->y_s = y_s;
        final_blueprint_data->z_s = z_s;
        final_blueprint_data->t_s = source_plane_event->t_event;
        final_blueprint_data->L_s = source_plane_event->y_event[8];
        return true;
    }

    // The intersection was outside the valid radial bounds. Return false.
    return false;
    """
    cfc.register_CFunction(includes=includes, desc=desc, name=name, cfunc_type=cfunc_type, params=params, body=body)
    print(f"    ... Registered C engine: {name}().")
```

### 7.f: `handle_disk_intersection()` Engine

This function generates the C engine that performs the final physics calculations for a photon that has terminated on the physical **accretion disk**. It is called by the main "finalizer" function (`calculate_and_fill_blueprint_data_universal`) after all integration is complete.

This engine orchestrates the full radiative transfer calculation: 1. **Get Metric**: It calls the `g4DD_metric` dispatcher to get the metric tensor `g_{\mu\nu}` at the photon's final intersection point. 2. **Lower Indices**: It uses the metric to lower the indices of both the photon's 4-momentum (converting p^μ to p_μ) and the disk particle's 4-velocity (converting u^μ to u_μ). This is mathematically essential for the next step. 3. **Call Radiative Transfer Engine**: It passes the covariant vectors and the intrinsic properties of the disk particle (stored in the `nearest_neighbor` struct) to the `calculate_radiative_transfer()` engine. 4. **Populate Blueprint**: The `calculate_radiative_transfer` engine computes the final observed intensity and wavelength. This function then populates the `blueprint_data_t` struct with these physical results, as well as diagnostic information like the intersection time and path length.

```
[ ]: def handle_disk_intersection_engine():
         """
         Generates the C engine for handling a disk intersection. This definitive
         version (v6.0) is repurposed as a pure physics calculator, to be called
         only during the finalization phase.
         """
```

```
print(" -> Repurposing C engine: handle_disk_intersection (v6.0)...")

includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "<math.h>"]
desc = r"""@brief Performs the final radiative transfer calculation for a disk intersection.
@details This engine is called by the finalizer. It takes the photon's final
         state and the stored nearest-neighbor data, computes the observed
         intensity and wavelength, and populates the final blueprint record.
"""
name = "handle_disk_intersection"

# *** CORRECTED: New signature for its new role as a finalizer helper. ***
params = r"""
const double final_y[9],
const MassiveParticle *restrict nearest_neighbor,
const commondata_struct *restrict commondata, const params_struct *restrict params,
const metric_params *restrict metric,
blueprint_data_t *restrict final_blueprint_data
"""

body = r"""
// This function now assumes the photon's status is already TERMINATED_DISK.

// 1. Get metric at the photon's final position (the intersection point).
metric_struct g4DD;
g4DD_metric(commondata, params, metric, final_y, &g4DD);

// 2. Lower the indices of the photon's 4-momentum and the neighbor's 4-velocity
//    using the metric at the intersection point.
const double g_munu[4][4] = {
    {g4DD.g00, g4DD.g01, g4DD.g02, g4DD.g03},
    {g4DD.g01, g4DD.g11, g4DD.g12, g4DD.g13},
    {g4DD.g02, g4DD.g12, g4DD.g22, g4DD.g23},
    {g4DD.g03, g4DD.g13, g4DD.g23, g4DD.g33}
};

double photon_p_mu[4] = {0,0,0,0};
double disk_u_mu[4] = {0,0,0,0};
for(int mu=0; mu<4; mu++) {
    for(int nu=0; nu<4; nu++) {
        photon_p_mu[mu] += g_munu[mu][nu] * final_y[nu+4];
        disk_u_mu[mu] += g_munu[mu][nu] * nearest_neighbor->u[nu];
    }
}
```

```
    // 3. Call the core radiative transfer physics engine.
    double temp_stokes_I;
    double temp_lambda_observed;
    calculate_radiative_transfer(photon_p_mu, disk_u_mu,
                                 nearest_neighbor->j_intrinsic, nearest_neighbor->lambda_rest,
                                 &temp_stokes_I, &temp_lambda_observed);

    // 4. Populate the final blueprint with the results.
    final_blueprint_data->stokes_I = temp_stokes_I;
    final_blueprint_data->lambda_observed = temp_lambda_observed;

    // 5. Populate diagnostic information from the intersection.
    final_blueprint_data->y_s = nearest_neighbor->pos[0]; // x-pos of neighbor
    final_blueprint_data->z_s = nearest_neighbor->pos[1]; // y-pos of neighbor
    final_blueprint_data->t_s = final_y[0]; // time of intersection
    final_blueprint_data->L_s = final_y[8]; // path length at intersection
    """
    cfc.register_CFunction(includes=includes, desc=desc, name=name, params=params, body=body)
    print(f"    ... Registered C engine: {name}() (Repurposed as Finalizer Helper).")
```

### 7.g: `calculate_and_fill_blueprint_data_universal()`

This Python function generates the C "finalizer" engine `calculate_and_fill_blueprint_data_universal()`. Its sole purpose is to process the raw data from a single completed ray trace and compute the final quantities that are saved to the `blueprint.bin` file. It is called once for each photon after its integration is finished.

This function acts as a high-level dispatcher based on the photon's final `status`: * If the photon hit the **window plane**, it projects the 3D intersection point to 2D window coordinates `(y_w, z_w)`. * If the photon hit the **source plane**, it calls the `handle_source_plane_intersection()` engine. * If the photon hit the **accretion disk**, it calls the `handle_disk_intersection()` engine to perform the full radiative transfer calculation. * If the photon hit the **celestial sphere**, it converts the final Cartesian position to spherical polar angles `(θ, φ)`. * If the photon **failed**, it does nothing further.

Finally, it returns the fully populated `blueprint_data_t` struct to be written to the output file.

[ ]:
```
def calculate_and_fill_blueprint_data_universal():
    """
    Generates the C finalization engine, now updated to handle disk intersections
    by calling the radiative transfer physics engines.
    """
    print(" -> Generating C finalization engine: calculate_and_fill_blueprint_data_universal (with disk physics)...")

    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h"]
    desc = r"""@brief Processes a photon's final state to compute all blueprint quantities."""
    name = "calculate_and_fill_blueprint_data_universal"
    cfunc_type = "blueprint_data_t"

    params = """const commondata_struct *restrict commondata, const params_struct *restrict params,
              const metric_params *restrict metric,
            const PhotonState *restrict photon,
            const double window_center[3], const double n_x[3], const double n_y[3]"""

    body = r"""
    // Initialize all fields to zero.
    blueprint_data_t result = {0};
    // The C enum 'termination_type_t' is compatible with the integer field in the blueprint.
    result.termination_type = photon->status;

    // Always populate window data if a crossing was found.
    if (photon->window_event_data.found) {
        const double *y_event = photon->window_event_data.y_event;
        const double pos_w_cart[3] = {y_event[1], y_event[2], y_event[3]};
        const double vec_w[3] = {pos_w_cart[0] - window_center[0], pos_w_cart[1] - window_center[1], pos_w_cart[2] - window_center[2]};
        result.y_w = vec_w[0]*n_x[0] + vec_w[1]*n_x[1] + vec_w[2]*n_x[2];
        result.z_w = vec_w[0]*n_y[0] + vec_w[1]*n_y[1] + vec_w[2]*n_y[2];
        result.L_w = y_event[8];
        result.t_w = photon->window_event_data.t_event;
    }

    // Populate remaining fields based on the specific termination type.
    if (photon->status == TERMINATION_TYPE_SOURCE_PLANE) {
        // Use a temporary blueprint struct to satisfy the handle_source_plane_intersection signature.
        // This function validates the hit and calculates geometric properties.
        blueprint_data_t temp_blueprint;
        if (handle_source_plane_intersection(&photon->source_event_data, commondata, &temp_blueprint)) {
            result.y_s = temp_blueprint.y_s;
            result.z_s = temp_blueprint.z_s;
            result.L_s = temp_blueprint.L_s;
            result.t_s = temp_blueprint.t_s;
        }
    } else if (photon->status == TERMINATION_TYPE_DISK) {
        // *** NEW LOGIC FOR DISK HITS ***
        // Call the dedicated physics engine for disk hits. This function will perform
        // the index lowering and radiative transfer calculations.
        handle_disk_intersection(
            photon->y,                      // Photon's final state vector
            &photon->nearest_neighbor,      // The stored particle data from the k-d tree hit
            commondata, params, metric,
            &result                         // The blueprint to be filled with I_obs, lambda_obs, etc.
        );
    } else if (photon->status == TERMINATION_TYPE_CELESTIAL_SPHERE) {
        const double *final_y = photon->y;
        const double x = final_y[1];
        const double y = final_y[2];
        const double z = final_y[3];
        const double r = sqrt(SQR(x) + SQR(y) + SQR(z));
        if (r > 1e-9) {
            result.final_theta = acos(z / r);
            result.final_phi = atan2(y, x);
        }
    }
    // For TERMINATION_TYPE_FAILURE, no other fields need to be set.

    return result;
    """
    cfc.register_CFunction(includes=includes, desc=desc, cfunc_type=cfunc_type, name=name, params=params, body=body,␣
↪include_CodeParameters_h=True)
    print(f"    ... Registered C finalizer: {name}() (with disk physics).")
```

### 7.h: GSL State Management Helpers

The GNU Scientific Library (GSL) ODE integrator is a powerful tool, but it requires careful memory management. To avoid repeatedly allocating and freeing memory for every single time step (which would be extremely slow), we allocate a persistent "state" for each photon at the beginning of the simulation. This state includes the GSL stepper, control function, and evolution objects.

The Python function in the next cell generates two small C helper functions to manage this process: * `gsl_photon_init(PhotonState *photon)`: This function is called once per photon at the start of the simulation. It allocates the necessary GSL objects and attaches their pointers to the photon's `PhotonState` struct. * `gsl_photon_free(PhotonState *photon)`: This function is called once per photon at the end of the simulation. It safely frees all the memory allocated by the `init` function, preventing memory leaks.

```
[ ]: def generate_gsl_photon_helpers():
    """
    Generates and registers the C helper functions for initializing and
    freeing the persistent GSL state associated with each photon.
    """
    print(" -> Generating GSL state helper functions (v5.0)...")

    # Helper 1: gsl_photon_init
    includes_init = ["BHaH_defines.h"]
    desc_init = r"""@brief Initializes the persistent GSL state for a single photon.
    @details Allocates memory for the GSL stepper, control function, and
             evolution function for one photon's trajectory and attaches the
             pointers to its PhotonState struct.
    """
    name_init = "gsl_photon_init"
    params_init = "PhotonState *photon"
    body_init = r"""
    // Allocate the stepper, using the Runge-Kutta-Fehlberg 4(5) method.
    photon->step = gsl_odeiv2_step_alloc(gsl_odeiv2_step_rkf45, 9);
    // Allocate the control function, setting error tolerances.
    photon->control = gsl_odeiv2_control_yp_new(1e-8, 1e-8);

    // Allocate the evolution function.
    photon->evolve = gsl_odeiv2_evolve_alloc(9);

    if (!photon->step || !photon->control || !photon->evolve) {
        fprintf(stderr, "Error: Failed to allocate GSL state for a photon.\n");
        exit(1);
    }
    """
    cfc.register_CFunction(
        includes=includes_init, desc=desc_init,
        name=name_init, params=params_init, body=body_init
    )
    print(f"    ... Registered C helper: {name_init}().")

    # Helper 2: gsl_photon_free
    includes_free = ["BHaH_defines.h"]
    desc_free = r"""@brief Frees the memory associated with a single photon's GSL state."""
    name_free = "gsl_photon_free"
    params_free = "PhotonState *photon"
    body_free = r"""
    if (photon->evolve) { gsl_odeiv2_evolve_free(photon->evolve); }
    if (photon->control) { gsl_odeiv2_control_free(photon->control); }
    if (photon->step) { gsl_odeiv2_step_free(photon->step); }
    """
    cfc.register_CFunction(
        includes=includes_free, desc=desc_free,
        name=name_free, params=params_free, body=body_free
    )
    print(f"    ... Registered C helper: {name_free}().")
```

### 7.i: The GSL Wrapper and Stepper Functions

The GNU Scientific Library (GSL) requires that the user provide a function that calculates the right-hand side of the ODE system, and this function must have a specific C signature. The `ode_gsl_wrapper` C function serves as this critical **adapter** or **bridge** between the generic GSL interface and our specialized project code.

The Python function in the next cell generates two C functions: 1. `ode_gsl_wrapper()`: This is the core adapter. At every GSL substep, it is called. Its job is to: * Unpack the `gsl_params` struct to get access to all necessary data (like the metric choice and the numerical metric cache). * Act as the top-level dispatcher between the **analytic** and **numerical** metric pipelines. * If the metric is analytic, it calls our `g4DD_metric` and `connections` dispatchers. * If the metric is numerical, it calls our all-in-one `interpolate_metric_and_derivatives` engine and the algebraic `calculate_christoffels_from_metric_and_derivs` worker. * Finally, it passes the computed metric and Christoffel symbols to the generic `calculate_ode_rhs` engine.

2. `calculate_and_apply_single_step()`: This is a small helper function that encapsulates the call to the main GSL evolution function,

gsl_odeiv2_evolve_apply. This keeps the main integration loop in batch_integrator cleaner.

```
[ ]: def generate_granular_gsl_engines():
         """
         Generates the C engines for GSL control. The ode_gsl_wrapper is now
         updated to be the primary dispatcher for numerical metrics.
         """
         print(" -> Generating granular GSL control engines (with numerical metric support)...")

         # --- Engine 1: calculate_and_apply_single_step (Unchanged) ---
         cfc.register_CFunction(
             includes=["BHaH_defines.h", "BHaH_function_prototypes.h"],
             desc="Applies a single adaptive step of the GSL integrator.",
             name="calculate_and_apply_single_step",
             cfunc_type="int",
             params="PhotonState *photon, gsl_odeiv2_system *sys",
             body=r"""
         int status = gsl_odeiv2_evolve_apply(
             photon->evolve, photon->control, photon->step, sys,
             &photon->lambda, 1e10, &photon->d_lambda, photon->y
         );
         return status;
     """
         )

         # --- Engine 2: ode_gsl_wrapper (UPDATED) ---
         includes_wrapper = ["BHaH_defines.h", "BHaH_function_prototypes.h"]
         desc_wrapper = r"""@brief Acts as an adapter between the GSL solver and our C functions.

     This function is the core of the ODE evolution. For numerical metrics, it
     orchestrates the call to the interpolation engine and the algebraic
     Christoffel calculator. For analytic metrics, it calls the standard dispatchers.
     """
         name_wrapper = "ode_gsl_wrapper"
         cfunc_type_wrapper = "int"
         params_wrapper = "double lambda, const double y[9], double f[9], void *params"
         body_wrapper = r"""
         (void)lambda; // Suppress unused parameter warning for lambda.

         gsl_params *gsl_parameters = (gsl_params *)params;

         if (gsl_parameters->metric->type == Numerical) {
             // --- Numerical Metric Path ---
             // Step 1: Declare local storage for interpolated values.
             metric_struct g4DD_interpolated;
```

```
        g4DD_deriv_struct g4DDdD_interpolated;

        // Step 2: Call the all-in-one engine ONCE to get all 50 metric and derivative values.
        interpolate_metric_and_derivatives(gsl_parameters->cache, y[0], y[1], y[2], y[3],
                                           &g4DD_interpolated, &g4DDdD_interpolated);

        // Step 3: Declare storage for Christoffel symbols.
        connection_struct conn;

        // Step 4: Call the lightweight algebraic worker to compute Christoffels.
        calculate_christoffels_from_metric_and_derivs(&g4DD_interpolated, &g4DDdD_interpolated, &conn);

        // Step 5: Call the RHS engine with the computed values.
        calculate_ode_rhs(y, &g4DD_interpolated, &conn, f);

    } else {
        // --- Analytic Metric Path (Unchanged) ---
        metric_struct g4DD_analytic;
        connection_struct conn_analytic;

        g4DD_metric(gsl_parameters->commondata, gsl_parameters->params, gsl_parameters->metric, y, &g4DD_analytic);
        connections(gsl_parameters->commondata, gsl_parameters->params, gsl_parameters->metric, y, &conn_analytic);
        calculate_ode_rhs(y, &g4DD_analytic, &conn_analytic, f);
    }

    return GSL_SUCCESS;
"""
    # Use pop() to ensure we are replacing any old version of this function
    cfc.CFunction_dict.pop(name_wrapper, None)
    cfc.register_CFunction(
        includes=includes_wrapper, desc=desc_wrapper, cfunc_type=cfunc_type_wrapper,
        name=name_wrapper, params=params_wrapper, body=body_wrapper
    )
    print(f"    ... Registered C engines: calculate_and_apply_single_step() and ode_gsl_wrapper() (updated).")
```

### 7.j: K-d Tree and Numerical Metric Helper Functions

The following cells generate a series of C helper functions that are essential for managing the external data required by the simulation, such as the k-d tree snapshots and the numerical metric slices. These functions handle tasks like file I/O, sorting, and memory management.

This first function, `filename_sorter`, generates a C comparison function `compare_filenames`. This small utility is required by the standard C library's `qsort` function. Its only job is to compare two snapshot filenames (e.g., `mass_blueprint_t_0100.kdtree.bin` and `mass_blueprint_t_0110.kdtree.bin`) based on their numerical timestamp, ensuring that when we load all the snapshot files, they are in the correct chronological order.

[ ]:
```python
def filename_sorter():
    """
    Generates a C helper function to be used by qsort for sorting snapshot filenames.
    """
    print(" -> Registering C helper: filename_sorter()...")
    includes = ["stdio.h", "<stdlib.h>"]
    desc = "Comparison function for qsort to sort filenames numerically."
    name = "compare_filenames"
    cfunc_type = "int"
    params = "const void *a, const void *b"
    body = r"""
    const char *str_a = *(const char **)a;
    const char *str_b = *(const char **)b;
    int num_a, num_b;
    sscanf(str_a, "mass_blueprint_t_%d.kdtree.bin", &num_a);
    sscanf(str_b, "mass_blueprint_t_%d.kdtree.bin", &num_b);
    return (num_a > num_b) - (num_a < num_b);
    """
    cfc.register_CFunction(includes=includes, desc=desc, name=name, cfunc_type=cfunc_type, params=params, body=body)
    print(f"    ... Registered C helper: {name}().")
```

### 7.k: K-d Tree Snapshot Loader and Unloader

This Python function generates two low-level C worker functions for managing the memory of a single k-d tree snapshot file.

- `load_kdtree_snapshot()`: This function is responsible for loading a single `.kdtree.bin` file into memory. To achieve maximum performance and minimize RAM usage, it uses the `mmap` (memory-map) system call. Instead of reading the entire (potentially very large) file into the heap, `mmap` tells the operating system's virtual memory manager to map the file directly into the program's address space. The data is then loaded from disk on-demand by the OS as it is accessed.
- `unload_kdtree_snapshot()`: This function calls `munmap` to release the memory mapping created by the loader, ensuring there are no resource leaks.

[ ]:
```python
def kdtree_loader_and_unloader():
    """
    Generates C functions for memory-mapping a .kdtree.bin file into memory
    and for unmapping it.
    """
    print(" -> Generating C functions for k-d tree memory mapping...")

    # Function to load a snapshot
    load_includes = ["BHaH_defines.h", "<stdio.h>", "<stdlib.h>", "<sys/mman.h>", "<sys/stat.h>", "<fcntl.h>", "<unistd.h>"]
    load_desc = r"""@brief Loads a .kdtree.bin snapshot file into memory using mmap."""
    load_name = "load_kdtree_snapshot"
    load_params = "const char *filename, CustomKDTree *tree"
    load_body = r"""
    int fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror("Error opening k-d tree file");
        return -1; // Failure
    }

    struct stat sb;
    if (fstat(fd, &sb) == -1) {
        perror("Error getting file size");
        close(fd);
        return -1;
    }
    tree->file_size = sb.st_size;

    void *mapped_mem = mmap(NULL, tree->file_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (mapped_mem == MAP_FAILED) {
        perror("Error memory-mapping the file");
        close(fd);
        return -1;
    }
    close(fd); // File descriptor no longer needed after mmap

    tree->original_mmap_ptr = mapped_mem;
    char *current_ptr = (char *)mapped_mem;

    // Read header
    tree->num_particles = *(uint64_t *)current_ptr;
    current_ptr += sizeof(uint64_t);
    tree->dimensions = *(uint64_t *)current_ptr;
    current_ptr += sizeof(uint64_t);

    // Set pointers to payloads
    tree->node_metadata = (int32_t *)current_ptr;
    current_ptr += sizeof(int32_t) * tree->num_particles;
    tree->particle_data = (MassiveParticle *)current_ptr;

    return 0; // Success
    """
    cfc.register_CFunction(includes=load_includes, desc=load_desc, name=load_name, params=load_params, body=load_body, cfunc_type="int")

    # Function to unload a snapshot
    unload_includes = ["BHaH_defines.h", "<sys/mman.h>"]
    unload_desc = r"""@brief Unloads a memory-mapped k-d tree snapshot."""
    unload_name = "unload_kdtree_snapshot"
    unload_params = "CustomKDTree *tree"
    unload_body = r"""
    if (tree->original_mmap_ptr != NULL) {
        munmap(tree->original_mmap_ptr, tree->file_size);
        tree->original_mmap_ptr = NULL;
    }
    """
    cfc.register_CFunction(includes=unload_includes, desc=unload_desc, name=unload_name, params=unload_params, body=unload_body)

    print("    ... Registered C functions: load_kdtree_snapshot, unload_kdtree_snapshot.")
```

### 7.l: K-d Tree Loader Orchestrator

This Python function generates the C orchestrator `load_all_kdtree_snapshots()`. This function is called once by `main()` at the beginning of the simulation. It is responsible for finding, sorting, and loading all available k-d tree snapshot files.

Its logic is as follows: 1. **Scan Directory**: It scans the `../processed_snapshots/` directory to find all files ending in `.kdtree.bin`. It first does a pass to count the number of files to allocate the correct amount of memory. 2. **Sort Filenames**: It stores the filenames in an array and calls the standard C library's `qsort` function, passing it our custom `compare_filenames` helper. This ensures the snapshots are sorted chronologically. 3. **Load Snapshots**: It iterates through the sorted list of filenames, calling the `load_kdtree_snapshot()` worker for each one. This populates the array of `CustomKDTree` structs. 4. **Calculate Timestamps**: It parses the timestamp from each filename and, using the `mass_snapshot_every_t` parameter, calculates the physical coordinate time for each snapshot, storing it in an array. 5. **Return Data**: It returns the total number of snapshots loaded and the pointers to the arrays containing the loaded tree data and the timestamps.

```
[ ]: def generate_kdtree_loader_orchestrator():
        """
        Generates and registers the C orchestrator function for finding, sorting,
        and loading all k-d tree snapshot files from a directory into memory.
        """
        print(" -> Generating C k-d tree loader: load_all_kdtree_snapshots()...")

        # Add <dirent.h> for directory reading and <string.h> for strstr
        includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "stdio.h", "stdlib.h", "<dirent.h>", "<string.h>"]
        desc = r"""@brief Finds, sorts, and loads all .kdtree.bin files from the snapshot directory.
        @details This function encapsulates the logic for memory-mapping the k-d tree
                 data needed for disk intersection checks.
        @param[in]  commondata        Pointer to the commondata struct with runtime parameters.
        @param[out] kdtree_snapshots  Pointer to be allocated and filled with snapshot data.
        @param[out] snapshot_times    Pointer to be allocated and filled with snapshot times.
        @return The total number of snapshots successfully loaded.
        """
        name = "load_all_kdtree_snapshots"
        cfunc_type = "int"
        params = r"""
        const commondata_struct *restrict commondata,
        CustomKDTree **kdtree_snapshots,
        double **snapshot_times
        """

        body = r"""
```

```
    const char* snapshot_dir_path = "../processed_snapshots"; // Assumes a relative path
    printf("Loading k-d tree snapshots from '%s'...\n", snapshot_dir_path);
    DIR *dir;
    struct dirent *ent;
    int num_snapshots = 0;
    if ((dir = opendir(snapshot_dir_path)) != NULL) {
        while ((ent = readdir(dir)) != NULL) {
            if (strstr(ent->d_name, ".kdtree.bin") != NULL) {
                num_snapshots++;
            }
        }
        closedir(dir);
    } else {
        perror("Could not open snapshot directory");
        exit(1);
    }

    if (num_snapshots == 0) {
        fprintf(stderr, "Warning: No .kdtree.bin snapshot files found in '%s'. Disk intersection will be disabled.\n",␣
↪snapshot_dir_path);
        *kdtree_snapshots = NULL;
        *snapshot_times = NULL;
        return 0;
    }

    char **filenames = (char **)malloc(sizeof(char *) * num_snapshots);
    if (filenames == NULL) { exit(1); }
    dir = opendir(snapshot_dir_path);
    int count = 0;
    while ((ent = readdir(dir)) != NULL) {
        if (strstr(ent->d_name, ".kdtree.bin") != NULL) {
            filenames[count] = strdup(ent->d_name);
            count++;
        }
    }
    closedir(dir);
    qsort(filenames, num_snapshots, sizeof(char *), compare_filenames);

    *kdtree_snapshots = (CustomKDTree *)malloc(sizeof(CustomKDTree) * num_snapshots);
    *snapshot_times = (double *)malloc(sizeof(double) * num_snapshots);
    if (*kdtree_snapshots == NULL || *snapshot_times == NULL) { exit(1); }

    for (int i = 0; i < num_snapshots; ++i) {
        char filepath[512];
        snprintf(filepath, sizeof(filepath), "%s/%s", snapshot_dir_path, filenames[i]);
        if (load_kdtree_snapshot(filepath, &(*kdtree_snapshots)[i]) != 0) {
            fprintf(stderr, "Error: Failed to load snapshot %s\n", filepath);
            exit(1);
        }
        int snapshot_index;
        sscanf(filenames[i], "mass_blueprint_t_%d.kdtree.bin", &snapshot_index);
        (*snapshot_times)[i] = (double)snapshot_index * commondata->mass_snapshot_every_t;
        free(filenames[i]);
    }
    free(filenames);
    printf("Successfully loaded and sorted %d snapshots.\n", num_snapshots);

    return num_snapshots;
    """
    cfc.register_CFunction(
        includes=includes, desc=desc, cfunc_type=cfunc_type,
        name=name, params=params, body=body
    )
    print(f"    ... Registered C orchestrator: {name}().")
```

### 7.m: K-d Tree Nearest Neighbor Search Engine

This function generates the high-performance C engine for performing the k-Nearest Neighbor (k-NN) search. The core of the engine is the `search_recursive` C function.

A naive implementation of a tree search can be slow due to **memory latency**. When the algorithm needs to access a child node, the data for that node might be in slow main memory (RAM) instead of the fast CPU cache, causing the CPU to stall.

To solve this, the generated C code uses a low-level compiler intrinsic called `__builtin_prefetch`. This instruction acts as a hint to the CPU, telling it to start loading the data for both the "good" and "bad" child nodes into the cache *before* they are actually needed. While the CPU is busy processing the current node, the memory controller works in the background to fetch the next required data. This technique of **hiding memory latency** is crucial for achieving high performance in pointer-heavy algorithms like a tree search.

```
[ ]: def kdtree_search_engine():
    """
    Generates the C functions that perform the recursive k-Nearest Neighbor search
    on a loaded k-d tree.

    VERSION 2: PERFORMANCE OPTIMIZED.
    This version adds __builtin_prefetch compiler intrinsics to the recursive
    search function. This is a low-level hint to the CPU to begin fetching data
    for child nodes from main memory into the cache before it is explicitly needed.
    This technique aims to hide memory latency and reduce CPU stalls caused by
    cache misses, which were identified as the primary performance bottleneck.
    """
    print(" -> Generating C engine for k-d tree nearest neighbor search [PERFORMANCE OPTIMIZED]...")

    includes = ["BHaH_defines.h", "<math.h>", "<stdio.h>"]

    prefunc = r"""
// Helper to initialize the WinnersCircle struct
static void wc_init(WinnersCircle *wc, int n_wanted) {
    wc->n_wanted = n_wanted;
    wc->count = 0;
    for (int i = 0; i < n_wanted; ++i) {
        wc->indices[i] = -1;
        wc->sq_distances[i] = 1e300; // Initialize with a very large number
    }
}

// Helper to add a candidate to the WinnersCircle, maintaining sorted order
static void wc_add(WinnersCircle *wc, int index, double sq_dist) {
    if (wc->count < wc->n_wanted) {
        wc->indices[wc->count] = index;
        wc->sq_distances[wc->count] = sq_dist;
        wc->count++;
    } else if (sq_dist < wc->sq_distances[wc->n_wanted - 1]) {
        wc->indices[wc->n_wanted - 1] = index;
        wc->sq_distances[wc->n_wanted - 1] = sq_dist;
    } else {
        return; // Not a winner
    }

    for (int i = wc->count - 1; i > 0; --i) {
        if (wc->sq_distances[i] < wc->sq_distances[i - 1]) {
            double temp_d = wc->sq_distances[i];
            int temp_i = wc->indices[i];
            wc->sq_distances[i] = wc->sq_distances[i - 1];
            wc->indices[i] = wc->indices[i - 1];
            wc->sq_distances[i - 1] = temp_d;
            wc->indices[i - 1] = temp_i;
        }
    }
}

// The recursive search function
static void search_recursive(const CustomKDTree *tree, const double query_pos[3], int current_idx, WinnersCircle *wc) {
    if (current_idx < 0 || current_idx >= (int)tree->num_particles) {
        return;
    }

    const MassiveParticle *pivot = &tree->particle_data[current_idx];
    const int split_axis = tree->node_metadata[current_idx];

    const double dx = query_pos[0] - pivot->pos[0];
    const double dy = query_pos[1] - pivot->pos[1];
    const double dz = query_pos[2] - pivot->pos[2];
    const double dist_sq = dx*dx + dy*dy + dz*dz;
    wc_add(wc, current_idx, dist_sq);

    if (split_axis == -1) { // Leaf node
        return;
    }

    const double axis_dist = query_pos[split_axis] - pivot->pos[split_axis];
    const int good_side_idx = (axis_dist < 0) ? (2 * current_idx + 1) : (2 * current_idx + 2);
    const int bad_side_idx = (axis_dist < 0) ? (2 * current_idx + 2) : (2 * current_idx + 1);

    // *** PERFORMANCE OPTIMIZATION ***
    // Issue prefetch instructions for the data of the child nodes. This hints to the
    // CPU to start loading this memory into the cache while we process the current node.
    // The '0' indicates a read operation.
    // The '1' indicates low temporal locality (we likely won't need this exact data again soon).
    if (good_side_idx < (int)tree->num_particles) {
        __builtin_prefetch(&tree->particle_data[good_side_idx], 0, 1);
    }
    if (bad_side_idx < (int)tree->num_particles) {
        __builtin_prefetch(&tree->particle_data[bad_side_idx], 0, 1);
    }
    // *** END OF OPTIMIZATION ***

    search_recursive(tree, query_pos, good_side_idx, wc);

    const double search_radius_sq = wc->sq_distances[wc->n_wanted - 1];
    if (axis_dist * axis_dist < search_radius_sq) {
        search_recursive(tree, query_pos, bad_side_idx, wc);
    }
}
"""

    desc = r"""@brief Finds the N nearest neighbors to a query point in a k-d tree."""
    name = "find_n_nearest_neighbors"
    params = "const CustomKDTree *tree, const double query_pos[3], int n_neighbors, MassiveParticle *neighbor_results"
    body = r"""
    if (n_neighbors > MAX_NEIGHBORS) {
        fprintf(stderr, "Error: Requested more neighbors than MAX_NEIGHBORS.\n");
        return;
    }

    WinnersCircle wc;
    wc_init(&wc, n_neighbors);

    // Start the recursive search from the root (index 0)
    search_recursive(tree, query_pos, 0, &wc);

    // Copy the results into the output array
    for (int i = 0; i < wc.count; ++i) {
        neighbor_results[i] = tree->particle_data[wc.indices[i]];
    }
"""
    cfc.register_CFunction(includes=includes, prefunc=prefunc, desc=desc, name=name, params=params, body=body)
    print("    ... Registered C engine: find_n_nearest_neighbors [PERFORMANCE OPTIMIZED].")
```

### 7.n: Numerical Metric Cache Management Helpers

This Python function generates a set of C helper functions that are responsible for managing the lifecycle of the `MetricSliceCache`. This modular approach isolates the details of memory and file management from the main integration loop.

- `load_metric_slice_from_file()`: This is a **placeholder** function for the low-level file I/O. A real implementation would contain the code to open a numerical data file (e.g., an HDF5 file), read the grid metadata, allocate memory for the 10 metric component arrays, and read the data from the file into those arrays.
- `initialize_metric_cache()`: This is a **placeholder** for the logic that runs once at the beginning of the simulation. It would be responsible for finding the filenames of the first three metric slices that bracket the initial time `t_start` and calling `load_metric_slice_from_file()` to populate the cache.
- `free_metric_cache()`: This is a **placeholder** for the cleanup logic. It would be responsible for calling `free_metric_slice_data()` on each slice in the cache to prevent memory leaks.

```
[ ]: def generate_numerical_metric_helpers():
         """
         Generates C helper functions for managing the MetricSliceCache, including
         initialization, freeing, and a placeholder for loading data from disk.
         """
         print(" -> Generating C helpers for numerical metric cache management...")

         # --- Helper 1: load_metric_slice_from_file ---
         cfc.register_CFunction(
             includes=["BHaH_defines.h", "<stdio.h>", "<stdlib.h>"],
             desc="Placeholder for loading a numerical metric slice from a file.",
             name="load_metric_slice_from_file",
             params="const char *filename, metric_slice *slice",
             body="""
         // THIS IS A PLACEHOLDER.
         (void)filename; (void)slice; // Suppress unused parameter warnings

         exit(1);
     """
         )

         # --- Helper 2: initialize_metric_cache ---
         cfc.register_CFunction(
             includes=["BHaH_defines.h", "BHaH_function_prototypes.h"],
             desc="Initializes and populates the metric cache for the first time.",
             name="initialize_metric_cache",
             params="const commondata_struct *restrict commondata, MetricSliceCache *restrict cache",
             body="""
         // This is a placeholder for the logic to find and load the initial 3 slices.
         (void)commondata;
         (void)cache;

     """
         )

         # --- Helper 3: free_metric_cache ---
         cfc.register_CFunction(
             includes=["BHaH_defines.h", "<stdio.h>"], # Added stdio.h for fprintf
             desc="Frees all memory allocated within the metric slice cache.",
             name="free_metric_cache",
             params="MetricSliceCache *restrict cache",
             # --- THIS IS THE CORRECTED BODY ---
             body="""
         // This is a placeholder for the logic to free data arrays (e.g., g00_data).
         (void)cache;
     """
         )
         print("    ... Registered numerical metric helper functions.")
```

### 7.o: Metric Cache Updater Helper

This function generates the C helper `update_metric_cache_if_needed()`. This is a key part of the numerical metric pipeline. It is called by the main `batch_integrator` at the beginning of each time-slot epoch.

Its single responsibility is to manage the **rolling buffer** of the `MetricSliceCache`. It checks if the current integration time has moved outside the valid temporal range of the three currently loaded metric slices. Since we integrate backwards in time, it specifically checks if the current time is now earlier than the midpoint between the two oldest cached slices.

If an update is needed, it performs the “roll”: 1. Frees the memory of the “oldest” slice (the one with the latest time). 2. Shifts the remaining two slices over in the cache array. 3. Calls the (placeholder) file loader to read the new, earlier time slice from disk into the now-vacant spot in the cache.

This ensures that the interpolation engine always has the correct three time slices it needs to perform its calculations, while keeping the in-memory footprint minimal.

[ ]:
```
# Not written yet
```

### 7.p: The Main Integration Loop Orchestrator

This function generates the `batch_integrator`, which is the C orchestrator for the entire simulation. It implements the **"Iterative Time Slotting"** algorithm, which is a highly efficient method for integrating a large number of photons in parallel.

The core idea is to group photons into "time slots" based on their current coordinate time. The algorithm then processes these slots in a chronological, backward-in-time sweep. For each slot, it: 1. **(Numerical Metrics Only)** Calls `update_metric_cache_if_needed()` to ensure the metric data for the current time is loaded. 2. Processes all photons in the current slot in parallel bundles. 3. For each photon, it takes one adaptive time step using the GSL solver. 4. It then performs the full **event cascade**: checking for hard failures, then physical disk intersections (via the bounding box and k-d tree), and finally fallback geometric plane intersections. 5. If a photon is not terminated, it is placed into its new, earlier time slot to be processed in a future epoch.

This architecture ensures that all photons being processed at any given moment are clustered in time, which maximizes data reuse for both the k-d tree snapshots and the numerical metric slices, dramatically reducing I/O and redundant computations.

[ ]:
```
def generate_batch_integrator_orchestrator():
    """
    Generates the main C orchestrator, now fully integrated with k-d tree
    logic for physical disk intersection checks.
    """
    print(" -> Generating top-level C orchestrator: batch_integrator() (with k-d tree logic)...")


    prefunc = r"""
// --- Time Slot Manager Implementation ---
typedef struct {
    long int *photons;
    long int count;
    long int capacity;
} PhotonList;
typedef struct {
    double t_min, t_max, delta_t_slot;
    int num_slots;
    PhotonList *slots;
} TimeSlotManager;
static void slot_manager_init(TimeSlotManager *tsm, double t_min, double t_max, double delta_t_slot) {
    tsm->t_min = t_min;
    tsm->t_max = t_max;
    tsm->delta_t_slot = delta_t_slot;
    tsm->num_slots = (int)ceil((t_max - t_min) / delta_t_slot);
    if (tsm->num_slots <= 0) { fprintf(stderr, "Error: Invalid TimeSlotManager dimensions.\n"); exit(1); }
    tsm->slots = (PhotonList *)malloc(sizeof(PhotonList) * tsm->num_slots);
    if (tsm->slots == NULL) { fprintf(stderr, "Error: Failed to allocate memory for time slots.\n"); exit(1); }
    for (int i = 0; i < tsm->num_slots; i++) {
        tsm->slots[i].photons = (long int *)malloc(sizeof(long int) * 16);
        if (tsm->slots[i].photons == NULL) { fprintf(stderr, "Error: Failed to allocate memory for a slot's photon list.\n"); exit(1); }
        tsm->slots[i].count = 0;
        tsm->slots[i].capacity = 16;
    }
}
static void slot_manager_free(TimeSlotManager *tsm) {
    for (int i = 0; i < tsm->num_slots; i++) { free(tsm->slots[i].photons); }
    free(tsm->slots);
}
static inline int slot_get_index(const TimeSlotManager *tsm, double t) {
    if (t < tsm->t_min || t >= tsm->t_max) return -1;
    return (int)floor((t - tsm->t_min) / tsm->delta_t_slot);
}
static void slot_add_photon(PhotonList *slot, long int photon_idx) {
    if (slot->count >= slot->capacity) {
        slot->capacity *= 2;
        slot->photons = (long int *)realloc(slot->photons, sizeof(long int) * slot->capacity);
        if (slot->photons == NULL) { fprintf(stderr, "Error: Failed to reallocate memory for a slot's photon list.\n"); exit(1); }
    }
    slot->photons[slot->count++] = photon_idx;
}
static void slot_remove_chunk(PhotonList *slot, long int *chunk_buffer, long int chunk_size) {
    for (long int i = 0; i < chunk_size; ++i) {
        chunk_buffer[i] = slot->photons[i];
    }
    for (long int i = 0; i < slot->count - chunk_size; ++i) {
        slot->photons[i] = slot->photons[i + chunk_size];
    }
    slot->count -= chunk_size;
}
"""

    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "omp.h", "<stdbool.h>"]
    desc = r"""@brief Main orchestrator for "Iterative Time Slotting" photon integration with disk intersection."""
    name = "batch_integrator"
    # UPDATED C function signature to accept k-d tree data
    params = r"""
    const commondata_struct *restrict commondata,
    const params_struct *restrict params,
    const metric_params *restrict metric,
    long int num_rays,
    int num_snapshots,
    const CustomKDTree *kdtree_snapshots,
    const double *snapshot_times,
    MetricSliceCache *restrict metric_cache,
    blueprint_data_t *results_buffer
    """
    body = r"""


    // === INITIALIZATION ===
    printf("Initializing %ld photon states for batch integration...\n", num_rays);
    PhotonState *all_photons = (PhotonState *)malloc(sizeof(PhotonState) * num_rays);
    if (all_photons == NULL) { fprintf(stderr, "Error: Failed to allocate memory for photon states.\n"); exit(1); }
    long int active_photons = num_rays;

    TimeSlotManager tsm;
    slot_manager_init(&tsm, commondata->slot_manager_t_min, commondata->t_start + 1.0, commondata->slot_manager_delta_t);

    const double camera_pos[3] = {commondata->camera_pos_x, commondata->camera_pos_y, commondata->camera_pos_z};
    const double window_center[3] = {commondata->window_center_x, commondata->window_center_y, commondata->window_center_z};
    double n_z[3] = {window_center[0] - camera_pos[0], window_center[1] - camera_pos[1], window_center[2] - camera_pos[2]};
    double mag_n_z = sqrt(SQR(n_z[0]) + SQR(n_z[1]) + SQR(n_z[2]));
    for(int i=0; i<3; i++) n_z[i] /= mag_n_z;
    const double guide_up[3] = {commondata->window_up_vec_x, commondata->window_up_vec_y, commondata->window_up_vec_z};
    double n_x[3] = {n_z[1]*guide_up[2] - n_z[2]*guide_up[1], n_z[2]*guide_up[0] - n_z[0]*guide_up[2], n_z[0]*guide_up[1] -␣
↪n_z[1]*guide_up[0]};
    double mag_n_x = sqrt(SQR(n_x[0]) + SQR(n_x[1]) + SQR(n_x[2]));
    if (mag_n_x < 1e-9) {
        double alternative_up[3] = {0.0, 1.0, 0.0};
        if (fabs(n_z[1]) > 0.999) { alternative_up[1] = 0.0; alternative_up[2] = 1.0; }
        n_x[0] = alternative_up[1]*n_z[2] - alternative_up[2]*n_z[1];
        n_x[1] = alternative_up[2]*n_z[0] - alternative_up[0]*n_z[2];
        n_x[2] = alternative_up[0]*n_z[1] - alternative_up[1]*n_z[0];
        mag_n_x = sqrt(SQR(n_x[0]) + SQR(n_x[1]) + SQR(n_x[2]));
    }
    for(int i=0; i<3; i++) n_x[i] /= mag_n_x;
    double n_y[3] = {n_z[1]*n_x[2] - n_z[2]*n_x[1], n_z[2]*n_x[0] - n_z[0]*n_x[2], n_z[0]*n_x[1] - n_z[1]*n_x[0]};

    #pragma omp parallel for
    for (long int i = 0; i < num_rays; i++) {
        const int j = i / commondata->scan_density;
        const int k = i % commondata->scan_density;
        const double x_pix = -commondata->window_size/2.0 + (k + 0.5) * (commondata->window_size / commondata->scan_density);
        const double y_pix = -commondata->window_size/2.0 + (j + 0.5) * (commondata->window_size / commondata->scan_density);
        double target_pos[3] = {window_center[0] + x_pix*n_x[0] + y_pix*n_y[0],
                                window_center[1] + x_pix*n_x[1] + y_pix*n_y[1],
                                window_center[2] + x_pix*n_x[2] + y_pix*n_y[2]};
        set_initial_conditions_cartesian(commondata, params, metric, camera_pos, target_pos, all_photons[i].y);
        all_photons[i].y[0] += commondata->t_start;
        all_photons[i].lambda = 0.0;
        all_photons[i].d_lambda = 1.0;
        all_photons[i].status = ACTIVE;
        for(int ii=0; ii<9; ++ii) { all_photons[i].y_p[ii] = all_photons[i].y[ii]; all_photons[i].y_p_p[ii] = all_photons[i].y[ii]; }
        all_photons[i].lambda_p = all_photons[i].lambda; all_photons[i].lambda_p_p = all_photons[i].lambda;
        plane_event_params window_params = {{n_z[0], n_z[1], n_z[2]}, n_z[0]*window_center[0] + n_z[1]*window_center[1] +␣
↪n_z[2]*window_center[2]};
        all_photons[i].on_positive_side_of_window_prev = (plane_event_func(all_photons[i].y, &window_params) > 0);
        plane_event_params source_params = {{commondata->source_plane_normal_x, commondata->source_plane_normal_y,␣
↪commondata->source_plane_normal_z},
                                            commondata->source_plane_center_x*commondata->source_plane_normal_x +␣
↪commondata->source_plane_center_y*commondata->source_plane_normal_y +␣
↪commondata->source_plane_center_z*commondata->source_plane_normal_z};
        all_photons[i].on_positive_side_of_source_prev = (plane_event_func(all_photons[i].y, &source_params) > 0);
        all_photons[i].source_event_data.found = false;
        all_photons[i].window_event_data.found = false;
        gsl_photon_init(&all_photons[i]);
    }

    int initial_slot_idx = slot_get_index(&tsm, commondata->t_start);
    if(initial_slot_idx != -1) {
        for(long int i=0; i<num_rays; ++i) { slot_add_photon(&tsm.slots[initial_slot_idx], i); }
    } else {
        fprintf(stderr, "Error: Initial t_start is outside the defined time slot manager domain.\n");
        exit(1);
    }

    printf("Starting batch integration loop (Iterative Time Slotting)...\n");

    gsl_params gsl_parameters = {commondata, params, metric, metric_cache};
    gsl_odeiv2_system sys = {ode_gsl_wrapper, NULL, 9, &gsl_parameters};

    long int *bundle_photons = (long int *)malloc(sizeof(long int) * BUNDLE_CAPACITY);
    if (bundle_photons == NULL) { exit(1); }

    double start_time = omp_get_wtime();
    long int initial_active_photons = active_photons;

    // === MAIN EPOCH LOOP ===
    for (int i = initial_slot_idx; i >= 0 && active_photons > 0; i--) {
        int current_slot_idx = i;
        if (tsm.slots[current_slot_idx].count == 0) continue;

        while (tsm.slots[current_slot_idx].count > 0 && active_photons > 0) {
            long int bundle_size = MIN(tsm.slots[current_slot_idx].count, BUNDLE_CAPACITY);
            slot_remove_chunk(&tsm.slots[current_slot_idx], bundle_photons, bundle_size);

            #pragma omp parallel for
            for (long int j = 0; j < bundle_size; j++) {
                long int photon_idx = bundle_photons[j];
                if (all_photons[photon_idx].status != ACTIVE) continue;

                for(int ii=0; ii<9; ++ii) { all_photons[photon_idx].y_p_p[ii] = all_photons[photon_idx].y_p[ii];␣
↪all_photons[photon_idx].y_p[ii] = all_photons[photon_idx].y[ii]; }
                all_photons[photon_idx].lambda_p_p = all_photons[photon_idx].lambda_p; all_photons[photon_idx].lambda_p =␣
↪all_photons[photon_idx].lambda;

                int status = calculate_and_apply_single_step(&all_photons[photon_idx], &sys);

                const double p_t = all_photons[photon_idx].y[4];
                const double r_sq = SQR(all_photons[photon_idx].y[1]) + SQR(all_photons[photon_idx].y[2]) + SQR(all_photons[photon_idx].
↪y[3]);

                if (fabs(p_t) > commondata->p_t_max || status != GSL_SUCCESS || fabs(all_photons[photon_idx].y[0]) >␣
↪commondata->t_integration_max) {
                    all_photons[photon_idx].status = TERMINATION_TYPE_FAILURE;
                } else if (r_sq > SQR(commondata->r_escape)) {
                    all_photons[photon_idx].status = TERMINATION_TYPE_CELESTIAL_SPHERE;
                } else {
                    // *** NEW K-D TREE LOGIC INSERTED HERE ***
                    const double x = all_photons[photon_idx].y[1];
                    const double y = all_photons[photon_idx].y[2];
                    const double z = all_photons[photon_idx].y[3];
                    if (num_snapshots > 0 &&
                        x >= commondata->disk_bounds_x_min && x <= commondata->disk_bounds_x_max &&
                        y >= commondata->disk_bounds_y_min && y <= commondata->disk_bounds_y_max &&
                        z >= commondata->disk_bounds_z_min && z <= commondata->disk_bounds_z_max) {

                        // Find nearest snapshot in time
                        double min_dt = 1e100;
                        int best_snapshot_idx = -1;
                        for(int snap_i=0; snap_i<num_snapshots; ++snap_i) {
                            double dt = fabs(all_photons[photon_idx].y[0] - snapshot_times[snap_i]);
                            if (dt < min_dt) {
                                min_dt = dt;
                                best_snapshot_idx = snap_i;
                            }
                        }

                        // K-d tree proximity search
                        MassiveParticle neighbor;
                        find_n_nearest_neighbors(&kdtree_snapshots[best_snapshot_idx], &all_photons[photon_idx].y[1], 1, &neighbor);

                        const double dist_sq = SQR(x - neighbor.pos[0]) + SQR(y - neighbor.pos[1]) + SQR(z - neighbor.pos[2]);
                        if (dist_sq < SQR(commondata->delta_r_max)) {
                            all_photons[photon_idx].status = TERMINATION_TYPE_DISK;
                            all_photons[photon_idx].nearest_neighbor = neighbor;
                        }
                    }

                    if (all_photons[photon_idx].status == ACTIVE) {
                        event_detection_manager(all_photons[photon_idx].y_p_p, all_photons[photon_idx].y_p, all_photons[photon_idx].y,
                                                all_photons[photon_idx].lambda_p_p, all_photons[photon_idx].lambda_p,␣
↪all_photons[photon_idx].lambda,
                                                commondata, &all_photons[photon_idx].on_positive_side_of_window_prev,␣
↪&all_photons[photon_idx].on_positive_side_of_source_prev,
                                                &all_photons[photon_idx].window_event_data, &all_photons[photon_idx].source_event_data);

                        if (all_photons[photon_idx].source_event_data.found) {
                            blueprint_data_t temp_blueprint;
                            if (handle_source_plane_intersection(&all_photons[photon_idx].source_event_data, commondata,␣
↪&temp_blueprint)) {
                                all_photons[photon_idx].status = TERMINATION_TYPE_SOURCE_PLANE;
                            } else {
                                all_photons[photon_idx].source_event_data.found = false;
                            }
                        }
                    }
                }

                if (all_photons[photon_idx].status != ACTIVE) {
                    #pragma omp atomic
                    active_photons--;
                }
            } // End of omp for loop

            for (long int j = 0; j < bundle_size; j++) {
                long int photon_idx = bundle_photons[j];
                if (all_photons[photon_idx].status == ACTIVE) {
                    int new_slot_idx = slot_get_index(&tsm, all_photons[photon_idx].y[0]);
                    if (new_slot_idx != -1) {
                        slot_add_photon(&tsm.slots[new_slot_idx], photon_idx);
                    } else {
                        all_photons[photon_idx].status = TERMINATION_TYPE_FAILURE;
                        active_photons--;
                    }
                }
            }
        } // End inner while loop

        #pragma omp master
        {
            double current_time = omp_get_wtime();
            double elapsed_time = current_time - start_time;
            long int photons_terminated = initial_active_photons - active_photons;
            double rays_per_sec = (elapsed_time > 1e-9) ? (double)photons_terminated / elapsed_time : 0.0;
            double percent_done = (double)photons_terminated / initial_active_photons * 100.0;

            printf("\rEpoch (Slot %d), Active Photons: %ld (%.1f%% done, %.1f rays/sec) ", i, active_photons, percent_done,␣
↪rays_per_sec);
            fflush(stdout);
        }
    }

    printf("\nBatch integration finished.\n");
    free(bundle_photons);

    // === FINALIZATION ===
    printf("Processing final states and populating blueprint buffer...\n");
    #pragma omp parallel for
    for (long int i = 0; i < num_rays; i++) {
        results_buffer[i] = calculate_and_fill_blueprint_data_universal(
            commondata, params, metric, &all_photons[i],
            window_center, n_x, n_y
        );
    }
    printf("Cleaning up GSL states...\n");
    #pragma omp parallel for
    for (long int i = 0; i < num_rays; i++) {
        gsl_photon_free(&all_photons[i]);
    }
    slot_manager_free(&tsm);
    free(all_photons);
    """
    cfc.register_CFunction(includes=includes, prefunc=prefunc, desc=desc, name=name, params=params, body=body)
    print(f"    ... Registered C orchestrator: {name}() (with k-d tree logic).")
```

### 7.q: The `main()` C Function Entry Point

This function registers the C `main()` function, which serves as the entry point for the entire executable program. In our final architecture, `main()` is a pure **orchestrator**; it contains no physics logic itself. Instead, it calls other functions to manage the entire lifecycle of the simulation.

The `main` function performs the following sequence of operations: 1. **Initialize Parameters**: It first calls `commondata_struct_set_to_default()` to set compiled-in defaults, then calls `cmdline_input_and_parfile_parser()` to override them with user-provided values. 2. **Load Global Data**: It calls `load_all_kdtree_snapshots()` to load the entire accretion disk model into memory. If the user has selected a numerical metric, it also calls `initialize_metric_cache()` to load the initial set of metric data files. 3. **Print Banner**: It prints a summary of the simulation parameters to the console. 4. **Execute Integration**: It calls the main `batch_integrator()` orchestrator to run the full simulation. 5. **Save Results**: After the integrator finishes, it saves the final `results_buffer` to the `light_blueprint.bin` file. 6. **Cleanup**: It calls the appropriate helper functions (`unload_kdtree_snapshot`, `free_metric_cache`, etc.) to safely free all allocated memory before exiting.

[ ]:
```
def main():
    """
    Re-registers the main() C function to orchestrate the full simulation,
    including conditional loading and unloading of numerical metric data.
    """
    print(" -> Updating main() to manage numerical metric cache lifecycle...")

    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "stdio.h", "stdlib.h"]
    desc = r"""@brief Main entry point for the batch integrator with conditional numerical metric support."""
    cfunc_type = "int"
    name = "main"
    params = "int argc, const char *argv[]"

    body = r"""
    // --- Step 1: Initialize Core Data Structures ---
    commondata_struct commondata;
    params_struct params = {0};
    metric_params metric;

    // --- Step 2: Set Default Parameters and Parse User Input ---
    commondata_struct_set_to_default(&commondata);
    cmdline_input_and_parfile_parser(&commondata, argc, argv);

    // --- Step 3: Set Metric Type Based on Parameters ---
    if (commondata.metric_choice == 0) {
        metric.type = (commondata.a_spin == 0.0) ? Schwarzschild : Kerr;
    } else if (commondata.metric_choice == 1) {
        metric.type = Schwarzschild_Standard;
    } else if (commondata.metric_choice == 2) {
        metric.type = Numerical;
    } else {
        fprintf(stderr, "Error: Unknown metric_choice = %d\n", commondata.metric_choice);
        exit(1);
    }

    // --- Step 4: Load all k-d tree snapshot files into memory ---
    CustomKDTree *kdtree_snapshots = NULL;
    double *snapshot_times = NULL;
    int num_snapshots = 0;
    num_snapshots = load_all_kdtree_snapshots(&commondata, &kdtree_snapshots, &snapshot_times);

    // --- Step 5: Conditionally Load Numerical Metric Data ---
    MetricSliceCache *metric_cache = NULL; // Initialize to NULL
    if (metric.type == Numerical) {
        printf("Numerical metric selected. Initializing metric slice cache...\n");
        metric_cache = (MetricSliceCache*)malloc(sizeof(MetricSliceCache));
        if(metric_cache == NULL) { fprintf(stderr, "Error: Failed to allocate memory for MetricSliceCache.\n"); exit(1); }
        initialize_metric_cache(&commondata, metric_cache);
    }

    // --- Step 6: Print Simulation Banner ---
    printf("============================================\n");
    printf("  Photon Geodesic Integrator (Batch Mode)  \n");
    printf("============================================\n");
    if (metric.type == Numerical) {
        printf("Metric: Numerical\n");
    } else {
        printf("Metric: %s (a=%.3f)\n", (metric.type == Kerr) ? "Kerr" : "Schwarzschild", commondata.a_spin);
    }
    printf("Scan Resolution: %d x %d\n", commondata.scan_density, commondata.scan_density);
    if (num_snapshots > 0) {
        printf("Accretion Disk: ENABLED (%d snapshots loaded)\n", num_snapshots);
    } else {
        printf("Accretion Disk: DISABLED (no snapshots found)\n");
    }

    // --- Step 7: Main Logic ---
    if (commondata.debug_mode) {
        printf("\n>>> RUNNING IN SINGLE-RAY DEBUG MODE <<<\n");
        printf("Debug mode is not supported in this batching main() function.\n");
    } else {
        long int num_rays = (long int)commondata.scan_density * commondata.scan_density;
        blueprint_data_t *results_buffer = (blueprint_data_t *)malloc(sizeof(blueprint_data_t) * num_rays);
        if (results_buffer == NULL) { exit(1); }

        batch_integrator(&commondata, &params, &metric, num_rays,
                         num_snapshots, kdtree_snapshots, snapshot_times,
                         metric_cache, // Pass the (possibly NULL) cache pointer
                         results_buffer);

        printf("Scan finished. Writing %ld ray results to light_blueprint.bin...\n", num_rays);
        FILE *fp_blueprint = fopen("light_blueprint.bin", "wb");
        if (fp_blueprint == NULL) { perror("Error opening blueprint file"); exit(1); }
        fwrite(results_buffer, sizeof(blueprint_data_t), num_rays, fp_blueprint);
        fclose(fp_blueprint);
        free(results_buffer);
    }

    // --- Step 8: Cleanup ---
    printf("Unloading k-d tree snapshots...\n");
    if (kdtree_snapshots != NULL) {
        for (int i = 0; i < num_snapshots; ++i) {
            unload_kdtree_snapshot(&kdtree_snapshots[i]);
        }
        free(kdtree_snapshots);
    }
    if (snapshot_times != NULL) { free(snapshot_times); }

    if (metric_cache != NULL) {
        printf("Freeing metric slice cache...\n");
        free_metric_cache(metric_cache);
        free(metric_cache);
    }

    printf("\nRun complete.\n");
    return 0;
    """
    cfc.CFunction_dict.pop(name, None)
    cfc.register_CFunction(includes=includes, desc=desc, cfunc_type=cfunc_type, name=name, params=params, body=body)
    print("    ... main() has been updated successfully to manage the numerical metric cache.")
```

# Step 8: Project Assembly and Compilation

This is the final phase of the notebook for C code generation. It brings all the previously defined pieces together to construct the complete, compilable C project. The Python functions in this section do not generate any new physics code; instead, they manage the `nrpy` build system itself.

### 8.a: Registering Core C Data Structures

This function, `register_core_data_structures`, is one of the most critical in the entire notebook. Its job is to generate the C `typedef`s for all the custom data structures (`struct`s and `enum`s) used by the project.

It consolidates all type definitions into a single, large C code string. This is crucial because the C compiler requires that a type be defined before it can be used as a member in another struct. By defining everything in one place, we have full control over the declaration order, ensuring that dependencies are met (e.g., `MassiveParticle` is defined before it is used in `PhotonState`).

The entire string of C code is then registered with the `BHaH` infrastructure using a single call.

-1.0.7 `nrpy` Functions Used in this Cell:

- `nrpy.infrastructures.BHaH.BHaH_defines_h.register_BHaH_defines(name, C_code_string)`:
  - **Source File**: `nrpy/infrastructures/BHaH/BHaH_defines_h.py`
  - **Description**: This function adds a given C-code string to a global registry. This registry is later used by `output_BHaH_defines_h()` to generate the master `BHaH_defines.h` header file. We register our consolidated struct definitions under the key `"after_general"` to ensure they are placed after standard C library includes but before any function prototypes.

[ ]:
```python
# In file: V12_1_Python_to_C_via_NRPy.ipynb
# This cell REPLACES the existing register_core_data_structures().

def register_core_data_structures():
    """
    Generates and registers all core C data structures, now including support for
    numerical metrics, a derivative struct, and the metric slice cache.
    """
    print(" -> Registering all core C data structures (Numerical Metric Version)...")

    # --- Step 1: Define all C structs and enums as Python strings ---
    # The order of definition within this large string is critical.

    # Define metric, connection, and the NEW derivative structs.
    metric_components = [f"g{i}{j}" for i in range(4) for j in range(i, 4)]
    metric_struct_str = "typedef struct { 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 " + ", ".join(connection_components) + "; } connection_struct;"

    # NEW: Struct to hold the 40 unique metric derivatives g_μν,δ
    deriv_components = [f"g{i}{j}d{k}" for i in range(4) for j in range(i, 4) for k in range(4)]
    deriv_struct_str = "typedef struct { double " + ", ".join(deriv_components) + "; } g4DD_deriv_struct;"

    consolidated_structs_c_code = rf"""
// ============================================================================
// K-d Tree and Particle Data Structures (Unchanged)
// ============================================================================
typedef struct {{
    int id; double pos[3]; double u[4]; double lambda_rest; float j_intrinsic;
}} __attribute__((packed)) MassiveParticle;
typedef struct {{
    int32_t* node_metadata; MassiveParticle* particle_data; uint64_t num_particles;
    uint64_t dimensions; void* original_mmap_ptr; size_t file_size;
}} CustomKDTree;
#define MAX_NEIGHBORS 10
typedef struct {{
    int indices[MAX_NEIGHBORS]; double sq_distances[MAX_NEIGHBORS]; int count; int n_wanted;
}} WinnersCircle;

// ============================================================================
// Numerical Metric Data Structures
// ============================================================================
// Holds the raw grid data for a single time slice of the numerical metric.
typedef struct {{
    double t; // Timestamp of this slice
    int N_x, N_y, N_z; // Grid dimensions
    double x_min, y_min, z_min; // Grid boundaries
    double inv_dx, inv_dy, inv_dz; // Inverse grid spacing
    // Pointers to the raw data for each of the 10 metric components
    const double *g00_data, *g01_data, *g02_data, *g03_data, *g11_data;
    const double *g12_data, *g13_data, *g22_data, *g23_data, *g33_data;
}} metric_slice;

// The in-memory cache holding the 3 time slices needed for temporal interpolation.
typedef struct {{
    metric_slice slices[3];
    double times[3];
}} MetricSliceCache;


// --- SELF-CONTAINED INDEXING MACRO ---
// This macro takes a pointer to a metric_slice to get the grid dimensions,
// making it independent of the BHaH gridfunction infrastructure.
#define NUMERICAL_METRIC_IDX3(slice_ptr, i, j, k) ((i) + (slice_ptr)->N_x * ((j) + (slice_ptr)->N_y * (k)))


// ============================================================================
// Core Metric and GSL Parameter Structs
// ============================================================================
#include <gsl/gsl_errno.h>
```

```
#include <gsl/gsl_odeiv2.h>

{metric_struct_str}
{connections_struct_str}
{deriv_struct_str}

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;
    MetricSliceCache *cache; // Must be non-const to be updatable
}} gsl_params;


// ================================================================================
// Event Detection and Plane Crossing Helpers
// ================================================================================
// Struct to hold parameters for a plane-crossing event
typedef struct {{
    double n[3]; // Plane normal vector
    double d;    // Plane distance from origin
}} plane_event_params;

// Event function pointer type
typedef double (*event_function_t)(const double y[9], void *event_params);

// Event function for a generic plane crossing.
static inline double plane_event_func(const double y[9], void *event_params) {{
    plane_event_params *params = (plane_event_params *)event_params;
    // Event is defined as distance to plane = 0
    return y[1]*params->n[0] + y[2]*params->n[1] + y[3]*params->n[2] - params->d;
}}

// ================================================================================
// Batch Integration and Output Structs (with DISK termination)
// ================================================================================
#define CACHE_LINE_SIZE 64
#define BUNDLE_CAPACITY 16384

// 1. Define the termination_type_t enum.
typedef enum {{
    TERMINATION_TYPE_FAILURE,         // Value = 0
    TERMINATION_TYPE_DISK,            // Value = 1 (A valid, physical disk hit)
    TERMINATION_TYPE_SOURCE_PLANE,    // Value = 2
    TERMINATION_TYPE_CELESTIAL_SPHERE,// Value = 3
    ACTIVE                            // Value = 4 (for internal logic only)
}} termination_type_t;

// 2. Define the blueprint_data_t struct, which uses termination_type_t.
typedef struct {{
    termination_type_t termination_type;
    double y_w, z_w;
    double stokes_I, lambda_observed;
    double y_s, z_s;
    double final_theta, final_phi;
    double L_w, t_w, L_s, t_s;
}} __attribute__((packed)) blueprint_data_t;

// 3. Define the event_data_struct for geometric plane crossings.
typedef struct {{
    bool found;
    double lambda_event, t_event;
    double y_event[9];
}} event_data_struct;

// 4. Define the PhotonState struct, which depends on MassiveParticle.
typedef struct {{
    double y[9];
    double lambda, d_lambda;
    termination_type_t status;
    double y_p_p[9], y_p[9];
    double lambda_p_p, lambda_p;
    bool on_positive_side_of_window_prev;
    bool on_positive_side_of_source_prev;
    event_data_struct source_event_data;
    event_data_struct window_event_data;
    MassiveParticle nearest_neighbor;
    gsl_odeiv2_step *step;
    gsl_odeiv2_control *control;
    gsl_odeiv2_evolve *evolve;
    // CORRECTED padding calculation for the new struct size
    char _padding[CACHE_LINE_SIZE - ( \
        sizeof(double)*31 + \
        sizeof(termination_type_t) + \
        sizeof(bool)*2 + \
        sizeof(event_data_struct)*2 + \
        sizeof(MassiveParticle) + \
        sizeof(void*)*3 \
    ) % CACHE_LINE_SIZE];
}} __attribute__((aligned(CACHE_LINE_SIZE))) PhotonState;
"""
    # --- Step 2: Register the entire block under a single key ---
    # We use "after_general" to ensure it's placed after basic C headers but before
    # any function prototypes in BHaH_defines.h.
    Bdefines_h.register_BHaH_defines("after_general", consolidated_structs_c_code)
    print("    ... Registered all core data structures (Consolidated).")
```

### 8.b: Final Build Command

This is the main execution block of the notebook. It brings all the previously defined Python functions together and calls them in a precise sequence to generate every file needed for the final, compilable C project.

The sequence of operations is critical, as later steps depend on the files and registrations created by earlier ones:

1. **Register All Components**: It calls all the C-generating Python functions that we have defined throughout the notebook. This populates `nrpy`'s internal library (`cfc.CFunction_dict`) with the complete definitions for all our custom C functions. At this stage, no files have been written yet; everything exists only in memory.

2. **Generate Parameter Handling Files**: It calls the necessary functions from the BHaH infrastructure to set up the parameter system:

   - `CPs.write_CodeParameters_h_files()`: Generates `set_CodeParameters.h` and its variants.
   - `CPs.register_CFunctions_params_commondata_struct_set_to_default()`: Registers the C functions that initialize the parameter structs with their compiled-in default values.
   - `cmdline_input_and_parfiles.generate_default_parfile()`: Creates the `project_name.par` file.
   - `cmdline_input_and_parfiles.register_CFunction_cmdline_input_and_parfile_parser()`: Registers the C function that reads the `.par` file and command-line arguments at runtime.

3. **Generate `BHaH_defines.h`**: It calls `Bdefines_h.output_BHaH_defines_h()`. This function scans `nrpy`'s internal library for all registered data structures and writes them into the master C header file, `BHaH_defines.h`.

4. **Copy Helper Files**: It calls `gh.copy_files()` to copy any necessary dependency files from the `nrpy` library installation into our project directory.

5. **Generate C Source, Prototypes, and Makefile**: It calls the final, most important build function, `Makefile.output_CFunctions_function_prototypes_and_constr`
   This powerful function performs three tasks at once:

   - It iterates through every C function registered with `nrpy` and writes each one into its own `.c` file.
   - It generates `BHaH_function_prototypes.h`, a header file containing the declarations (prototypes) for all the generated `.c` files.
   - It constructs the `Makefile`, which contains the compilation and linking instructions needed to build the final executable program, including linking against GSL and OpenMP.

After this cell is run, a complete, self-contained, and ready-to-compile C project will exist in the output directory.

```
[ ]: # In file: V12_1_Python_to_C_via_NRPy.ipynb
     # This is the final build cell for the project, with all numerical metric
     # functionality included.
```

```python
import os
import nrpy.helpers.generic as gh

print("\nAssembling and building C project with numerical metric support...")
os.makedirs(project_dir, exist_ok=True)

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

# 1a. Register ALL data structures in a single, consolidated call.
register_core_data_structures()

# 1b. Register all k-d tree related C functions.
kdtree_loader_and_unloader()
filename_sorter()
generate_kdtree_loader_orchestrator()
kdtree_search_engine()

# 1c. Register all numerical metric C engines and helpers.
generate_numerical_interpolation_and_deriv_engine()
generate_algebraic_christoffel_worker()
generate_numerical_metric_helpers()
#generate_metric_cache_updater()

# 1d. Register all remaining C functions for physics and integration.
generate_gsl_photon_helpers()
# Analytic physics workers and dispatchers
g4DD_kerr_schild()
con_kerr_schild()
g4DD_schwarzschild_cartesian()
con_schwarzschild_cartesian()
g4DD_metric()
connections()
# Core physics and integration engines
calculate_ode_rhs()
calculate_p0_reverse()
set_initial_conditions_cartesian()
check_conservation()
lagrange_interp_engine_generic()
event_detection_manager()
radiative_transfer_engine()
handle_disk_intersection_engine()
handle_source_plane_intersection_engine()
calculate_and_fill_blueprint_data_universal()
```

```python
# Batching engines and orchestrator
generate_granular_gsl_engines()
generate_batch_integrator_orchestrator()
# Final entry point
main()

# --- Step 2: Call BHaH infrastructure functions to generate the build system ---
print(" -> Generating BHaH infrastructure files...")
CPs.write_CodeParameters_h_files(project_dir=project_dir)
CPs.register_CFunctions_params_commondata_struct_set_to_default()
cmdline_input_and_parfiles.generate_default_parfile(project_dir=project_dir, project_name=project_name)

# Define the complete list of parameters for the command-line parser.
cmdline_inputs_list = [
    'M_scale', 'a_spin', 'metric_choice',
    'camera_pos_x', 'camera_pos_y', 'camera_pos_z',
    'window_center_x', 'window_center_y', 'window_center_z',
    'window_up_vec_x', 'window_up_vec_y', 'window_up_vec_z',
    'source_plane_normal_x', 'source_plane_normal_y', 'source_plane_normal_z',
    'source_plane_center_x', 'source_plane_center_y', 'source_plane_center_z',
    'source_up_vec_x', 'source_up_vec_y', 'source_up_vec_z',
    'source_r_min', 'source_r_max',
    'scan_density', 'window_size',
    'r_escape', 't_integration_max', 't_start', 'p_t_max',
    'debug_mode', 'perform_conservation_check',
    'slot_manager_t_min', 'slot_manager_delta_t',
    'mass_snapshot_every_t', 'delta_r_max',
    'disk_bounds_x_min', 'disk_bounds_x_max',
    'disk_bounds_y_min', 'disk_bounds_y_max',
    'disk_bounds_z_min', 'disk_bounds_z_max'
]

cmdline_input_and_parfiles.register_CFunction_cmdline_input_and_parfile_parser(
    project_name=project_name,
    cmdline_inputs=cmdline_inputs_list
)

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

print("Copying required helper files...")
gh.copy_files(
    package="nrpy.helpers",
    filenames_list=["simd_intrinsics.h"],
    project_dir=project_dir,
    subdirectory="simd",
)

print("Generating C source files, prototypes, and Makefile...")
# Add GSL and OpenMP flags for compilation and linking.
# If using HDF5 for numerical data, you would add "$(shell h5cc -show LDFLAGS)" here.
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=project_name,
    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 './{project_name}'.")
```