V11_8_Python_to_C_via_NRPy

July 28, 2025

# Step 1: Introduction and Core Physics

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, with each core Python function encapsulated in its own cell. This modular design enhances readability and maintainability.

# Step 2: Project Initialization

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.params.CodeParameter`. This is the central object for defining a runtime parameter that will be accessible in the generated C code. It registers each parameter's name, C type, default value, and properties in a global dictionary, which `nrpy`'s build system then uses to construct the C interface.

-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.
- `nrpy.params.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.
  - **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, also known as the "Triple-Lock" system. 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
# Cell ID: 33f07e1c (Replacement)

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")


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

# --- Universal Camera System Parameters ---
camera_pos_x = par.CodeParameter("REAL", __name__, "camera_pos_x", 0.0, add_to_parfile=True, commondata=True)
camera_pos_y = par.CodeParameter("REAL", __name__, "camera_pos_y", 0.0,  add_to_parfile=True, commondata=True)
camera_pos_z = par.CodeParameter("REAL", __name__, "camera_pos_z", 51.0,  add_to_parfile=True, commondata=True)

window_center_x = par.CodeParameter("REAL", __name__, "window_center_x", 0.0, add_to_parfile=True, commondata=True)
window_center_y = par.CodeParameter("REAL", __name__, "window_center_y", 0.0,  add_to_parfile=True, commondata=True)
window_center_z = par.CodeParameter("REAL", __name__, "window_center_z", 50.0,  add_to_parfile=True, commondata=True)


window_up_vec_x = par.CodeParameter("REAL", __name__, "window_up_vec_x", 0.0, add_to_parfile=True, commondata=True)
window_up_vec_y = par.CodeParameter("REAL", __name__, "window_up_vec_y", 1.0, add_to_parfile=True, commondata=True)
window_up_vec_z = par.CodeParameter("REAL", __name__, "window_up_vec_z", 0.0, add_to_parfile=True, commondata=True)

# --- Independent Source Plane Definition ---
source_plane_normal_x = par.CodeParameter("REAL", __name__, "source_plane_normal_x", 0.0, add_to_parfile=True, commondata=True)
source_plane_normal_y = par.CodeParameter("REAL", __name__, "source_plane_normal_y", 0.0, add_to_parfile=True, commondata=True)
source_plane_normal_z = par.CodeParameter("REAL", __name__, "source_plane_normal_z", 1.0, add_to_parfile=True, commondata=True)

source_plane_center_x = par.CodeParameter("REAL", __name__, "source_plane_center_x", 0.0, add_to_parfile=True, commondata=True)
source_plane_center_y = par.CodeParameter("REAL", __name__, "source_plane_center_y", 0.0, add_to_parfile=True, commondata=True)
source_plane_center_z = par.CodeParameter("REAL", __name__, "source_plane_center_z", 0.0, add_to_parfile=True, commondata=True)


source_up_vec_x = par.CodeParameter("REAL", __name__, "source_up_vec_x", 0.0, add_to_parfile=True, commondata=True)
source_up_vec_y = par.CodeParameter("REAL", __name__, "source_up_vec_y", 1.0, add_to_parfile=True, commondata=True)
source_up_vec_z = par.CodeParameter("REAL", __name__, "source_up_vec_z", 0.0, add_to_parfile=True, commondata=True)
```

```
source_r_min = par.CodeParameter("REAL", __name__, "source_r_min", 6.0, add_to_parfile=True, commondata=True)
source_r_max = par.CodeParameter("REAL", __name__, "source_r_max", 25.0, add_to_parfile=True, commondata=True)

# --- General Ray-Tracing Parameters ---
scan_density = par.CodeParameter("int", __name__, "scan_density", 512, add_to_parfile=True, commondata=True)
flatness_threshold = par.CodeParameter("REAL", __name__, "flatness_threshold", 1e-2, add_to_parfile=True, commondata=True)
r_escape = par.CodeParameter("REAL", __name__, "r_escape", 1500.0, add_to_parfile=True, commondata=True)
# --- NEW: Maximum Integration Time Parameter ---
t_integration_max = par.CodeParameter("REAL", __name__, "t_integration_max", 10000.0, commondata=True, add_to_parfile=True)



window_size = par.CodeParameter(
    "REAL", __name__, "window_size", 1.5,
    add_to_parfile=True, commondata=True, add_to_set_CodeParameters_h=True
)

# --- NEW: Adaptive Window Grid Parameters ---
print("-> Registering CodeParameters for adaptive window grids...")

# Master switch for the grid type
window_grid_type = par.CodeParameter(
    "int", __name__, "window_grid_type", 0,
    commondata=True, add_to_parfile=True
)

# Parameters for Log-Polar grids (Types 1 and 2)
log_polar_num_r = par.CodeParameter(
    "int", __name__, "log_polar_num_r", 512,
    commondata=True, add_to_parfile=True
)
log_polar_num_phi = par.CodeParameter(
    "int", __name__, "log_polar_num_phi", 1024,
    commondata=True, add_to_parfile=True
)

# Parameter for the central exclusion zone (Type 2)
log_polar_r_min = par.CodeParameter(
    "REAL", __name__, "log_polar_r_min", 0.1,
    commondata=True, add_to_parfile=True
)

# --- Physical Parameters ---
```

```
M_scale = par.CodeParameter(
    "REAL", __name__, "M_scale", 1.0,
    add_to_parfile=True, commondata=True, add_to_set_CodeParameters_h=True
)
a_spin = par.CodeParameter(
    "REAL", __name__, "a_spin", 0.0,
    add_to_parfile=True, commondata=True, add_to_set_CodeParameters_h=True
)
print("-> Registering CodeParameter for the initial integration time...")
t_start = par.CodeParameter("REAL", __name__, "t_start", 2000.0, commondata=True, add_to_parfile=True)

# --- Debugging Parameters ---
perform_conservation_check = par.CodeParameter(
    "bool", __name__, "perform_conservation_check", False,
    add_to_parfile=True, commondata=True
)

debug_mode = par.CodeParameter("bool", __name__, "debug_mode", False, add_to_parfile=True, commondata=True)

mass_snapshot_every_t = par.CodeParameter("REAL", __name__, "mass_snapshot_every_t", 10.0, commondata=True, add_to_parfile=True)

# --- NEW: Proximity Detection Parameter for Radiative Transfer ---
print("-> Registering CodeParameter for the disk proximity check...")
delta_r_max = par.CodeParameter("REAL", __name__, "delta_r_max", 2.0, commondata=True, add_to_parfile=True)

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

# Step 3: The Symbolic Core - Foundational Math

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

### 3.a: Metric Tensor Derivatives

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

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

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

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

-1.0.2 `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: Christoffel Symbol Calculation

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

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

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

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

-1.0.3 `nrpy` Functions Used in this Cell:

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

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

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

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

                # Use sp.trigsimp() to simplify the resulting expression.
                # This is the key to speeding up the symbolic calculation.
                Gamma4UDD[delta][mu][nu] = sp.trigsimp(Gamma4UDD[delta][mu][nu])

    return Gamma4UDD
```

### 3.c: Geodesic Momentum RHS

This function defines the symbolic right-hand side (RHS) for the evolution of the **reverse-time momentum**, p^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: Geodesic Position RHS

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: Proper Length ODE RHS

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, $\sqrt{\gamma_{ij} p^i p^j}$. It uses `sympy` symbols for the spatial momentum components (`pU[1]`, `pU[2]`, `pU[3]`) and programmatically constructs the optimized sum $\gamma_{ij} p^i p^j$ 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 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.

[ ]:
```
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 Calculation of 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:

$$g_{00}(q^0)^2 + 2\left(\sum_{i=1}^{3} g_{0i}q^i\right) q^0 + \left(\sum_{i,j=1}^{3} g_{ij}q^i q^j\right) = 0$$

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 $-p^i$:

$$g_{00}(-p^0)^2 + 2\left(\sum_{i=1}^{3} g_{0i}(-p^i)\right)(-p^0) + \left(\sum_{i,j=1}^{3} g_{ij}(-p^i)(-p^j)\right) = 0$$

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:

$$g_{00}(p^0)^2 + 2\left(\sum_{i=1}^{3} g_{0i}p^i\right) p^0 + \left(\sum_{i,j=1}^{3} g_{ij}p^i p^j\right) = 0$$

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: * $a = g_{00}$ * $b = 2\sum_{i=1}^{3} g_{0i}p^i$ * $c = \sum_{i,j=1}^{3} g_{ij}p^i p^j$

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

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

Simplifying by dividing the numerator and denominator by 2 gives:

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

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:

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

-1.0.6 nrpy Functions Used in this Cell:

- `nrpy.indexedexp.declarerank2(name, dimension, sym)`: Previously introduced. Used here to create an abstract symbolic tensor for the metric components.

[ ]:
```
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("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
```

0 Markdown for conserved Energy

[ ]:
```
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
```

1 Markdown for conserved L

[ ]:
```python
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]
```

2 Markdown for Carter Constant

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

# symbolic_energy() and symbolic_L_components_cart() remain correct.
```

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

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

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

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

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

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

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

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

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

    Q_formula = p_theta_sq + second_term

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

    return Q_final

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

# 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 $\eta_{\mu\nu}$:

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

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.

2.0.1 `nrpy` Functions Used in this Cell:

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

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

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

         Returns:
```

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

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

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

3 Markdown for Schwarzschild

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

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

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

    Returns:
        A tuple (g4DD, xx), where g4DD is the symbolic 4x4 metric tensor
        and xx is the list of symbolic coordinate variables.
    """
    # Define Cartesian coordinates
    t, x, y, z = sp.symbols("y[0] y[1] y[2] y[3]", real=True)
    xx = [t, x, y, z]

    # Access the symbolic mass parameter
    M = M_scale.symbol

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

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

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

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

            # The indices for g4DD are off by 1 from the spatial indices
            g4DD[i+1][j+1] = delta_ij + (2*M/r) * (x_i[i] * x_i[j] / (r**2))
    # --- CORRECTED: Ensure time-space components are sympy objects ---
    g4DD[0][1] = g4DD[1][0] = sp.sympify(0)
    g4DD[0][2] = g4DD[2][0] = sp.sympify(0)
    g4DD[0][3] = g4DD[3][0] = sp.sympify(0)

    return g4DD, xx
```

# Step 5: Symbolic Workflow Execution

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

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

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

3.0.1 `nrpy` Functions Used in this Cell:

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

[ ]:
```
# In file: V11_0_Python_to_C_via_NRPy.ipynb
# In cell [f9e18b56]

# --- 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.")
```

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

This design pattern allows for maximum code reuse and extensibility.

4 Schwarzschild Metric

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

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

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

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

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

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

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

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

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

4.0.1 `nrpy` Functions Used in this Cell:

- `nrpy.c_codegen.c_codegen(sympy_expressions, C_variable_names, **kwargs)`:
  - **Source File**: `nrpy/c_codegen.py`
  - **Description**: The core symbolic-to-C translation engine. It takes a list of `sympy` expressions and a corresponding list of C variable names and generates optimized C code to perform the assignments.
  - **Key Inputs**:
    - `sympy_expressions`: A Python list of symbolic expressions to be converted to C code.
    - `C_variable_names`: A Python list of strings for the C variables that will store the results.
  - **Key Keyword Arguments** (`kwargs`):
    - `enable_cse=True`: Enables Common Subexpression Elimination, which finds repeated mathematical operations, assigns them to temporary variables, and reuses those variables to reduce redundant calculations. This is essential for performance.
- `nrpy.c_function.register_CFunction(name, params, body, **kwargs)`:
  - **Source File**: `nrpy/c_function.py`
  - **Description**: This is the workhorse for defining a C function. It takes all necessary metadata and stores it in a global dictionary, `cfc.CFunction_dict`. The final build system uses this dictionary to write all the `.c` source files.
  - **Key Inputs**:
    - `name`: The name of the C function.
    - `params`: A string defining the function's parameters (e.g., `"const double y[4], ..."`).
    - `body`: A string containing the C code for the function's body.
  - **Key Keyword Arguments** (`kwargs`):
    - `include_CodeParameters_h=True`: Enables the "Triple-Lock" system for this function, automatically including `set_CodeParameters.h` at the top of the function body.

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

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

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

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

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

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

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

This function is structured identically to the `g4DD_kerr_schild` worker. It generates the C **worker** function `con_kerr_schild()`, whose only job is to compute the 40 unique Christoffel symbols for the unified Kerr-Schild metric.

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

4.0.2 `nrpy` Functions Used in this Cell:

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

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

5 Con Schwarzschild

```
[ ]: 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.c: `calculate_p0_reverse()` Engine

This Python function generates the C **engine** `calculate_p0_reverse()`, which implements the general formula for the time-component of the reverse-time momentum, p^0, derived from the null geodesic condition $g_{\mu\nu}p^\mu p^\nu = 0$. This function is a prime example of a reusable component, as its logic is valid for any metric for which the components $g_{\mu\nu}$ are known.

The code generation follows a pattern that is both robust and highly automated, showcasing a powerful `nrpy` technique called the **Preamble Pattern**:

1. **Symbolic Recipe:** It calls our pure-math `mom_time_p0_reverse()` function (from Step 3.f) to get the complete symbolic expression for p^0. This expression is built from abstract `sympy` symbols (e.g., `g00`, `g01`, etc.).
2. **Preamble Generation:** The function programmatically generates a C code "preamble." This preamble consists of a series of `const double` declarations that unpack the numerical values from the input `metric_struct` pointer and assign them to local C variables that have the *exact same names* as our abstract `sympy` symbols (e.g., `const double g00 = metric->g00;`).
3. **C Code Generation:** It calls `nrpy.c_codegen.c_codegen` to convert the symbolic `p0_expr` into an optimized C expression, assigning it to a temporary variable `p0_val`. This works seamlessly because the symbols in the expression (`g00`, etc.) now match the local C variables created by the preamble. This avoids the need for brittle string substitutions.
4. **Return Value:** The final C function body is constructed by combining the preamble, the CSE-optimized calculation, and a `return p0_val;` statement. This creates a complete, efficient, and readable C function.

5.0.1 `nrpy` Functions Used in this Cell:

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

```
[ ]: 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]"

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

    # 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{preamble}\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.")
```

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

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

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

5.0.2 `nrpy` Functions Used in this Cell:

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

[ ]:
```
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`.

5.0.3 `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.

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

6 Markdown for check_conservation

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

7 Markdown for event detection manager

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

8 kd tree loaer and unloader

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

9 kd tree search engine

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

10 Radiative transfer engine

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

11 Source plane intersection engine

[ ]:
```
# In file: V11_8_Python_to_C_via_NRPy.ipynb
# In cell [ea1786c7]

def handle_source_plane_intersection_engine():
    """
    Generates a C engine that handles a source plane intersection.
    This version (v2.9.1) uses the correct, consolidated PhotonStatus enum.
    """
    print(" -> Generating C engine: handle_source_plane_intersection (v2.9.1)...")

    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.
        // *** CORRECTED: Use the correct enum member from PhotonStatus ***
        final_blueprint_data->termination_type = TERMINATED_SOURCE;
        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}().")
```

12 Source disk intersection

[ ]:
```
def handle_disk_intersection_engine():
    """
    Generates the C engine to handle a disk intersection event.

    UPDATED for the "Nearest Neighbor" model. This function no longer interpolates.
    It takes the single closest massive particle and directly uses its properties
    for the radiative transfer calculation.
    """
    print(" -> Generating C engine: handle_disk_intersection (Nearest Neighbor Version)...")
```

```
includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "<math.h>"]
desc = r"""@brief Handles the physics calculations for a disk intersection event using the nearest neighbor."""
name = "handle_disk_intersection"
# The signature is now much simpler.
params = r"""
const double current_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"""
// The photon's state is current_y. The disk's state is taken directly
// from the provided nearest_neighbor particle.

// 1. Get metric at the photon's current position
metric_struct g4DD;
g4DD_metric(commondata, params, metric, current_y, &g4DD);

// 2. Lower indices of photon momentum and the neighbor's 4-velocity
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}; // This is now the neighbor's u_mu
for(int mu=0; mu<4; mu++) {
    for(int nu=0; nu<4; nu++) {
        photon_p_mu[mu] += g_munu[mu][nu] * current_y[nu+4];
        disk_u_mu[mu] += g_munu[mu][nu] * nearest_neighbor->u[nu];
    }
}

// 3. Call the 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;
final_blueprint_data->termination_type = TERMINATION_TYPE_DISK;

// Store diagnostic information
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 = current_y[0]; // time of intersection
final_blueprint_data->L_s = current_y[8]; // path length at intersection
"""
cfc.register_CFunction(includes=includes, desc=desc, name=name, params=params, body=body)
print("    ... Registered C engine: handle_disk_intersection (Nearest Neighbor 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** are functions that contain a `switch` statement to select the correct "worker" function based on the chosen metric (e.g., `Schwarzschild` vs. `Kerr`).
- **Orchestrators** are functions that execute a sequence of calls to other engines, workers, and dispatchers to perform a complex task, like setting up initial conditions or running the main integration loop.

13 Filename_sorter

[ ]:
```
def filename_sorter():
    """
    Generates a C helper function to be used by qsort for sorting snapshot filenames.
    """
    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)
```

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

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

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

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

13.0.1 `nrpy` Functions Used in this Cell:

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

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

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

    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h"]
    desc = r"""@brief Dispatcher to compute the 4-metric g_munu for the chosen metric."""
    name = "g4DD_metric"
    # The signature is now coordinate-aware, but the y vector is always Cartesian here.
    params = "const commondata_struct *restrict commondata, const params_struct *restrict params, const metric_params *restrict metric,␣
↪const double y[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.** Its sole responsibility is to select and call the correct metric-specific worker function (like `con_kerr_schild()`) to compute the Christoffel symbols.

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

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

13.0.2 `nrpy` Functions Used in this Cell:

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

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

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.

13.0.3 `nrpy` Functions Used in this Cell:

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

[ ]:
```
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)
```

14 Batch integrator orchestrator

[ ]:
```
def generate_batch_integrator_orchestrator():
    """
    Generates the main C orchestrator, batch_integrator(), with all logic,
    event detection, and finalization fully and correctly implemented.
    This version (v2.9.2) adds the necessary GSL headers.
    """
    print(" -> Generating top-level C orchestrator: batch_integrator() (v2.9.2)...")

    # *** CORRECTED: Added GSL headers to the includes list. ***
    includes = [
        "BHaH_defines.h",
        "BHaH_function_prototypes.h",
        "omp.h",
        "<stdbool.h>"
    ]
    desc = r"""@brief Main orchestrator for time-bundled photon integration."""
    name = "batch_integrator"
    params = r"""
    const commondata_struct *restrict commondata,
    const params_struct *restrict params,
    const metric_params *restrict metric,
    long int num_rays,
    blueprint_data_t *results_buffer
    """
    # The body of this function remains correct from the previous approved version.
    # Only the includes list needed to be changed.
    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;

    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 = 10.0;
        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;
    }

    PriorityQueue pq;
    heap_init(&pq, num_rays);
    for (long int i = 0; i < num_rays; i++) { heap_push(&pq, -all_photons[i].y[0], i); }

    FILE *log_fp = fopen("batch_stats.txt", "w");
    if (log_fp) fprintf(log_fp, "# Iteration\tActivePhotons\tBundleSize\tBundleTime\tBundleTimeRange\n");
    int iteration = 0;
    printf("Starting batch integration loop...\n");

    while (active_photons > 0) {
        if (pq.size == 0) break;
        HeapNode earliest = heap_pop(&pq);
        double bundle_target_time = -earliest.time;
        double time_tolerance = fmax(all_photons[earliest.photon_index].d_lambda * 0.05, 1e-2);
        int bundle_indices[BUNDLE_CAPACITY];
        bundle_indices[0] = earliest.photon_index;
        int bundle_size = 1;
        while (pq.size > 0 && -pq.nodes[0].time > bundle_target_time - time_tolerance && bundle_size < BUNDLE_CAPACITY) {
            bundle_indices[bundle_size++] = heap_pop(&pq).photon_index;
        }

        #pragma omp parallel
        {
            gsl_odeiv2_step *step = gsl_odeiv2_step_alloc(gsl_odeiv2_step_rkf45, 9);
            gsl_odeiv2_control *control = gsl_odeiv2_control_yp_new(1e-8, 1e-8);
            gsl_odeiv2_evolve *evolve = gsl_odeiv2_evolve_alloc(9);
            gsl_params gsl_parameters = {commondata, params, metric};
            gsl_odeiv2_system sys = {ode_gsl_wrapper, NULL, 9, &gsl_parameters};

            #pragma omp for
            for (int i = 0; i < bundle_size; i++) {
                int photon_idx = bundle_indices[i];
                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, step, control, evolve);

                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 (status != GSL_SUCCESS || r_sq > SQR(commondata->r_escape) || fabs(all_photons[photon_idx].y[0]) >␣
↪commondata->t_integration_max) {
                    all_photons[photon_idx].status = (r_sq > SQR(commondata->r_escape)) ? TERMINATED_SPHERE : TERMINATED_FAILURE;
                    #pragma omp atomic
                    active_photons--;
                } else {
                    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 = TERMINATED_SOURCE;
                            #pragma omp atomic
                            active_photons--;
                        } else {
                            all_photons[photon_idx].source_event_data.found = false;
                        }
                    }
                }
            }
            gsl_odeiv2_evolve_free(evolve); gsl_odeiv2_control_free(control); gsl_odeiv2_step_free(step);
        }

        double min_bundle_time = bundle_target_time, max_bundle_time = bundle_target_time;
        for (int i = 0; i < bundle_size; i++) {
            int photon_idx = bundle_indices[i];
            if (all_photons[photon_idx].status == ACTIVE) {
                heap_push(&pq, -all_photons[photon_idx].y[0], photon_idx);
                if (all_photons[photon_idx].y[0] < max_bundle_time) max_bundle_time = all_photons[photon_idx].y[0];
            }
        }

        if (log_fp) fprintf(log_fp, "%d\t%ld\t%d\t%.6e\t%.6e\n", iteration, active_photons, bundle_size, bundle_target_time,␣
↪bundle_target_time - max_bundle_time);
        if (iteration % 100 == 0) {
            printf("\rIteration: %d, Active Photons: %ld, Last Bundle Size: %d, Current Time: %.4f M", iteration, active_photons,␣
↪bundle_size, bundle_target_time);
            fflush(stdout);
        }
        iteration++;
    }

    printf("\nBatch integration finished after %d iterations.\n", iteration);
    if (log_fp) fclose(log_fp);

    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,
            all_photons[i].status,
            &all_photons[i].window_event_data,
            &all_photons[i].source_event_data,
            all_photons[i].y,
            window_center, n_x, n_y,
            (const double[3]){commondata->source_plane_center_x, commondata->source_plane_center_y, commondata->source_plane_center_z},
            (const double[3]){commondata->source_plane_normal_x, commondata->source_plane_normal_y, commondata->source_plane_normal_z},
            (const double[3]){commondata->source_up_vec_x, commondata->source_up_vec_y, commondata->source_up_vec_z}
        );
    }

    heap_free(&pq);
    free(all_photons);
    """
    cfc.register_CFunction(includes=includes, desc=desc, name=name, params=params, body=body)
    print(f"    ... Registered C orchestrator: {name}().")
```

### 7.d: The GSL Wrapper Function

The GNU Scientific Library (GSL) provides powerful, general-purpose routines for solving systems of Ordinary Differential Equations (ODEs). To use them, we must provide a C function that calculates the RHS of our ODE system and conforms to a specific function pointer signature required by the library. The `ode_gsl_wrapper` C function serves as this critical **adapter** or **bridge** between the generic GSL interface and our specialized project code.

This Python function registers the C function that performs these steps in order every time the GSL solver takes a time step:

1. **Unpack Parameters**: It receives a generic `void *params` pointer from the GSL solver. Its first action is to cast this pointer back to its true type, `gsl_params *`, which is our custom "carrier" struct. This gives the function access to the `commondata`, `params`, and `metric` structs needed by our physics routines.
2. **Call Metric and Connection Dispatchers**: It declares empty `metric_struct` and `connection_struct` containers on the stack. It then calls our high-level dispatchers (`g4DD_metric()` and `connections()`) to fill these structs with the correct physical values for the current point in spacetime `y`.
3. **Call `calculate_ode_rhs()` Engine**: It then passes the current state vector `y` and the now-filled `g4DD` and `conn` structs to our RHS engine. This engine computes the derivatives and stores them in the output array `f`, which is the array GSL uses for the RHS values.
4. **Return Success**: Finally, it returns `GSL_SUCCESS`, signaling to the GSL solver that the RHS calculation was completed correctly.

14.0.1 `nrpy` Functions Used in this Cell:

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

```
[ ]: def generate_granular_gsl_engines():
         """
         Generates the C engines for GSL control. This version (v2.9.2) adds the
         necessary GSL headers to the includes list of functions that use GSL types.
         """
         print(" -> Generating granular GSL control engines (v2.9.2)...")

         # Engine 1: calculate_and_apply_single_step
         # *** CORRECTED: Added GSL headers to the includes list. ***
         includes_step = [
             "BHaH_defines.h",
             "BHaH_function_prototypes.h"
         ]
         desc_step = r"""@brief Applies a single adaptive step of the GSL integrator."""
         name_step = "calculate_and_apply_single_step"
         cfunc_type_step = "int"
         params_step = r"""
         PhotonState *photon,
         gsl_odeiv2_system *sys,
         gsl_odeiv2_step *step,
         gsl_odeiv2_control *control,
         gsl_odeiv2_evolve *evolve
         """
         body_step = r"""
         int status = gsl_odeiv2_evolve_apply(
             evolve, control, step, sys,
             &photon->lambda,
             1e10,
             &photon->d_lambda,
             photon->y
         );
         return status;
         """
```

```
    cfc.register_CFunction(
        includes=includes_step, desc=desc_step, cfunc_type=cfunc_type_step,
        name=name_step, params=params_step, body=body_step
    )
    print(f"    ... Registered C engine: {name_step}().")

    # Engine 2: ode_gsl_wrapper (NO Caching for Phase 1)
    # This function also needs the GSL headers.
    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."""
    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;

    gsl_params *gsl_parameters = (gsl_params *)params;
    metric_struct g4DD;
    connection_struct conn;

    g4DD_metric(gsl_parameters->commondata, gsl_parameters->params, gsl_parameters->metric, y, &g4DD);
    connections(gsl_parameters->commondata, gsl_parameters->params, gsl_parameters->metric, y, &conn);
    calculate_ode_rhs(y, &g4DD, &conn, f);

    return GSL_SUCCESS;
    """
    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 engine: {name_wrapper}() (Phase 1, No Caching).")
```

### 7.f: Data Processing Engine

This Python function generates the C engine `calculate_and_fill_blueprint_data_universal()`. Its sole purpose is to process the raw event data from a single completed ray trace and compute the final quantities that are saved to the `blueprint.bin` file. It acts as a translator, converting the 3D intersection points from the integration into a 2D coordinate system on both the window and source planes.

The function orchestrates the following calculations:

1. **Window Plane Projection**: It takes the 3D Cartesian intersection point on the window plane and projects it onto the local 2D orthonormal basis vectors of the window (`n_x`, `n_y`) to get the final `(y_w, z_w)` coordinates.

2. **Source Plane Projection**: It performs a similar calculation for the source plane, projecting the 3D intersection point onto the source plane's local 2D basis to get the `(y_s, z_s)` coordinates. This basis is constructed from the source plane's normal vector and its "up" vector.
3. **Redshift Calculation**: It computes the gravitational redshift factor by calling the `g4DD_metric` dispatcher at both the window and source intersection points to get the g_{00} component of the metric. The redshift is then given by $\sqrt{g_{00}(\text{window})/g_{00}(\text{source})}$.
4. **Return Struct**: It populates and returns a `blueprint_data_t` struct containing all these calculated values, along with a `found` flag that is set to `true` only if all calculations are successful and finite.

14.0.2 `nrpy` Functions Used in this Cell:

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

[ ]:
```
# In file: V11_8_Python_to_C_via_N NRPy.ipynb
# In cell [cee235eb]

def calculate_and_fill_blueprint_data_universal():
    """
    Generates the C engine to process event data. This version (v2.9.1) uses
    the correct, consolidated PhotonStatus enum type.
    """
    print(" -> Generating C finalization engine: calculate_and_fill_blueprint_data_universal (v2.9.1)...")

    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h"]
    desc = r"""@brief Processes raw event data to compute final blueprint quantities based on termination type."""
    name = "calculate_and_fill_blueprint_data_universal"
    cfunc_type = "blueprint_data_t"

    # *** CORRECTED: Changed 'termination_type_t' to 'PhotonStatus' ***
    params = """const commondata_struct *restrict commondata, const params_struct *restrict params,
                const PhotonStatus term_status,
                const event_data_struct *restrict window_event,
                const event_data_struct *restrict source_event,
                const double final_y_state[9],
                const double window_center[3], const double n_x[3], const double n_y[3],
                const double source_plane_center[3], const double source_plane_normal[3],
                const double source_up_vector[3]"""

    body = r"""
    blueprint_data_t result = {0};
    // *** CORRECTED: Use the correct enum member from PhotonStatus ***
    result.termination_type = term_status;

    if (window_event->found) {
        const double pos_w_cart[3] = {window_event->y_event[1], window_event->y_event[2], window_event->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 = window_event->y_event[8];
        result.t_w = window_event->t_event;
    }

    if (term_status == TERMINATED_SOURCE) {
        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 temp_up[3] = {1.0, 0.0, 0.0};
            if (fabs(s_z[0]) > 0.999) { temp_up[0] = 0.0; temp_up[1] = 1.0; temp_up[2] = 0.0; }
            s_x[0] = temp_up[1]*s_z[2] - temp_up[2]*s_z[1];
            s_x[1] = temp_up[2]*s_z[0] - temp_up[0]*s_z[2];
            s_x[2] = temp_up[0]*s_z[1] - temp_up[1]*s_z[0];
            mag_s_x = sqrt(SQR(s_x[0]) + SQR(s_x[1]) + SQR(s_x[2]));
        }
        const double inv_mag_s_x = 1.0 / mag_s_x;
        s_x[0] *= inv_mag_s_x; s_x[1] *= inv_mag_s_x; s_x[2] *= inv_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 pos_s_cart[3] = {source_event->y_event[1], source_event->y_event[2], source_event->y_event[3]};
        const double vec_s[3] = {pos_s_cart[0] - source_plane_center[0], pos_s_cart[1] - source_plane_center[1], pos_s_cart[2] -␣
↪source_plane_center[2]};
        result.y_s = vec_s[0]*s_x[0] + vec_s[1]*s_x[1] + vec_s[2]*s_x[2];
        result.z_s = vec_s[0]*s_y[0] + vec_s[1]*s_y[1] + vec_s[2]*s_y[2];
        result.L_s = source_event->y_event[8];
        result.t_s = source_event->t_event;
    } else if (term_status == TERMINATED_SPHERE) {
        const double x = final_y_state[1];
        const double y = final_y_state[2];
        const double z = final_y_state[3];
        const double r = sqrt(x*x + y*y + z*z);
        if (r > 1e-9) {
            result.final_theta = acos(z / r);
            result.final_phi = atan2(y, x);
        }
    }

    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 engine: {name}().")
```

### 7.h: 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 set up the simulation, run the integration, and handle the output.

The `main` function performs the full sequence of operations required for a complete ray-tracing run:

1. **Declare Data Structures**: It declares instances of all necessary structs (`commondata_struct`, `params_struct`, `metric_params`).
2. **Initialize Parameters**: It initializes the parameters in a two-step process that ensures runtime flexibility:
   - First, it calls `commondata_struct_set_to_default()` to populate the `commondata` struct with the default values that were compiled into the executable (e.g., `M_scale = 1.0`).
   - Next, it calls `cmdline_input_and_parfile_parser()`. This function reads the project's `.par` file and any command-line arguments, and it will **override** the compiled-in defaults with any values provided by the user at runtime.
3. **Print Simulation Parameters**: It prints a detailed summary of all the final simulation parameters to the console. This is crucial for verification and for creating a record of the exact settings used for a given run. It also suggests a descriptive output filename and prints the full command-line arguments needed to reproduce the run exactly.
4. **Run Scanner Loop**: It calls the main orchestrator, `run_scan_and_save_blueprint_universal()`, to execute the full ray-tracing scan.
5. **Cleanup**: It prints a final status message and returns 0, indicating successful execution.

14.0.3 `nrpy` Functions Used in this Cell:

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

[ ]:
```
def main():
    """
    Re-registers the main() C function with all syntax and logical errors corrected.
    This is the definitive version (v2.8).
    """
    print(" -> Updating main() to call batch integrator (v2.8)...")

    includes = ["BHaH_defines.h", "BHaH_function_prototypes.h", "stdio.h", "stdlib.h"]
    desc = r"""@brief Main entry point for the batch integrator."""
    cfunc_type = "int"
    name = "main"
    params = "int argc, const char *argv[]"
    # *** CORRECTED AND COMPLETE BODY ***
    body = r"""
    commondata_struct commondata;
    params_struct params;
    metric_params metric;

    commondata_struct_set_to_default(&commondata);
    // This project is gridless and does not use the params_struct.
    // The call to params_struct_set_to_default is therefore unnecessary and removed
    // to prevent a segmentation fault from dereferencing a NULL griddata pointer.
    cmdline_input_and_parfile_parser(&commondata, argc, argv);

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

    printf("=============================================\n");
    printf("  Photon Geodesic Integrator (Batch Mode)  \n");
    printf("=============================================\n");
    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);
    printf("Initial Time (t_start): %.2f M\n", commondata.t_start);

    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) {
            fprintf(stderr, "Fatal Error: Could not allocate memory for blueprint results buffer.\n");
            exit(1);
        }


        batch_integrator(&commondata, &params, &metric, num_rays, 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 for writing");
            exit(1);
        }
        fwrite(results_buffer, sizeof(blueprint_data_t), num_rays, fp_blueprint);
        fclose(fp_blueprint);
        free(results_buffer);
    }

    printf("\nRun complete.\n");
    return 0;
    """
    # De-register any old version before registering the new one.
    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.")
```

# 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.

### 8.a: Custom Data Structures

This function defines all the necessary C `struct` and `enum` types for the project. It then registers them with the BHaH infrastructure, which makes these custom data types available to all other C files via the master header `BHaH_defines.h`.

The function `register_custom_structures_and_params` performs the following actions: 1. **Generates `connection_struct`**: It programmatically creates the C `typedef` for the `connection_struct`. This struct contains 40 `double` members to hold the unique Christoffel symbols ($\Gamma^{\alpha}_{\mu\nu}$). 2. **Generates `metric_struct`**: It follows the same programmatic pattern to create the `metric_struct`, which contains 10 `double` members to hold the unique components of the metric tensor ($g_{\mu\nu}$). 3. **Defines Other Structs**: It defines the C `typedef`s for all other data structures (`Metric_t` enum, `metric_params`, the GSL "carrier" struct `gsl_params`, the `event_data_struct` for interpolation results, and the final `blueprint_data_t` for output) as Python strings. 4. **Registers with `BHaH`**: For each `struct` or `enum`, it calls `Bdefines_h.register_BHaH_defines()`. This function adds the C code string to a global registry. When `Bdefines_h.output_BHaH_defines_h()` is called later in the build process, it will automatically find and include all these registered definitions in the final header file.

14.0.4 `nrpy` Functions Used in this Cell:

- `nrpy.indexedexp.declarerank3(...)`: Previously introduced. Used here to programmatically generate the list of C variable names for the members of the `connection_struct`.
- `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 (which should define a `struct`, `enum`, or other C-level type) to a global registry, associated with a given name. This registry is later used to generate the master `BHaH_defines.h` header file.

[ ]:
```python
# In file: V11_8_Python_to_C_via_NRPy.ipynb
# This function REPLACES the previous version of register_custom_structures_and_params().

def register_custom_structures_and_params():
    """
    Generates and registers all core C data structures for the project in a
    single, ordered block. This definitive version (v2.10) uses a more
    compact, readable format for the metric and connection structs.
    """
    print(" -> Registering all core C data structures (v2.10)...")

    # Programmatically generate the compact list of members for the structs.
    # This is robust and avoids manual typos.
    metric_components = [f"g{nu}{mu}" for nu in range(4) for mu in range(nu, 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;"

    # This single string contains all necessary definitions in the correct order.
    core_structs_c_code = rf"""
```

```
// ==============================================================================
// Core Metric and GSL Parameter Structs
// ==============================================================================
#include <gsl/gsl_errno.h>
#include <gsl/gsl_odeiv2.h>

// Compact struct definitions, generated programmatically for correctness.
{metric_struct_str}
{connections_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; }} gsl_params;
typedef double (*event_function_t)(const double y[9], void *event_params);

// ==============================================================================
// Batch Integration and Output Structs
// ==============================================================================
#define CACHE_LINE_SIZE 64
#define BUNDLE_CAPACITY 20000

// 1. Define the PhotonStatus enum FIRST.
typedef enum {{
    ACTIVE,
    TERMINATED_DISK,
    TERMINATED_SOURCE,
    TERMINATED_SPHERE,
    TERMINATED_FAILURE
}} PhotonStatus;

// 2. Define the blueprint_data_t struct, which USES PhotonStatus.
typedef struct {{
    PhotonStatus 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.
typedef struct {{
    bool found;
    double lambda_event, t_event;
    double y_event[9];
}} event_data_struct;

// 4. Define the PhotonState struct, which USES PhotonStatus and event_data_struct.
typedef struct {{
    double y[9];
    double lambda, d_lambda;
    PhotonStatus 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;
    char _padding[CACHE_LINE_SIZE - (sizeof(double)*53 + sizeof(PhotonStatus) + sizeof(bool)*4) % CACHE_LINE_SIZE];
}} __attribute__((aligned(CACHE_LINE_SIZE))) PhotonState;

// --- Min-Heap (Priority Queue) Implementation ---
typedef struct {{ double time; int photon_index; }} HeapNode;
typedef struct {{ HeapNode *nodes; int size; int capacity; }} PriorityQueue;
static inline void heap_swap(HeapNode *a, HeapNode *b) {{ HeapNode temp = *a; *a = *b; *b = temp; }}
static inline void heap_init(PriorityQueue *pq, int capacity) {{
    pq->nodes = (HeapNode *)malloc(sizeof(HeapNode) * capacity);
    if (pq->nodes == NULL) {{ fprintf(stderr, "Error: Failed to allocate memory for priority queue.\n"); exit(1); }}
    pq->size = 0;
    pq->capacity = capacity;
}}
static inline void heap_free(PriorityQueue *pq) {{ if (pq->nodes != NULL) {{ free(pq->nodes); pq->nodes = NULL; }} }}
static inline void heapify_up(PriorityQueue *pq, int index) {{
    while (index > 0) {{
        int parent_index = (index - 1) / 2;
        if (pq->nodes[index].time < pq->nodes[parent_index].time) {{
            heap_swap(&pq->nodes[index], &pq->nodes[parent_index]);
            index = parent_index;
        }} else {{ break; }}
    }}
}}
static inline void heap_push(PriorityQueue *pq, double time, int photon_index) {{
    if (pq->size >= pq->capacity) {{ fprintf(stderr, "Error: Priority queue overflow.\n"); return; }}
    pq->nodes[pq->size].time = time;
    pq->nodes[pq->size].photon_index = photon_index;
    heapify_up(pq, pq->size);
    pq->size++;
}}
```

```
static inline void heapify_down(PriorityQueue *pq, int index) {{
    int min_index = index;
    while (1) {{
        int left_child = 2 * index + 1;
        int right_child = 2 * index + 2;
        if (left_child < pq->size && pq->nodes[left_child].time < pq->nodes[min_index].time) min_index = left_child;
        if (right_child < pq->size && pq->nodes[right_child].time < pq->nodes[min_index].time) min_index = right_child;
        if (min_index == index) break;
        heap_swap(&pq->nodes[index], &pq->nodes[min_index]);
        index = min_index;
    }}
}}
static inline HeapNode heap_pop(PriorityQueue *pq) {{
    if (pq->size == 0) {{
        HeapNode empty_node = {{-1.0, -1}};
        fprintf(stderr, "Error: Attempted to pop from an empty priority queue.\n");
        return empty_node;
    }}
    HeapNode min_node = pq->nodes[0];
    pq->nodes[0] = pq->nodes[pq->size - 1];
    pq->size--;
    if (pq->size > 0) heapify_down(pq, 0);
    return min_node;
}}
"""
    # Use the 'after_general' key to ensure these definitions appear early in BHaH_defines.h
    Bdefines_h.register_BHaH_defines("after_general", core_structs_c_code)
    print("    ... Registered all core data structures in a single, ordered block.")
```

15 Kdtree structs

[ ]:
```
def register_kdtree_c_structs():
    """
    Generates and registers the C structs needed for loading and querying
    the pre-processed k-d tree snapshot files.
    """
    print(" -> Registering C data structures for k-d tree handling...")

    # This C struct must EXACTLY match the Python numpy.dtype and the
    # mass_particle_state_t struct from the mass_integrator.
    # It represents the data for a single particle in the .kdtree.bin file.
    massive_particle_struct_str = r"""
    typedef struct {
        int id;
        double pos[3];
        double u[4];
        double lambda_rest;
        float j_intrinsic;
    } __attribute__((packed)) MassiveParticle;
    """

    # This C struct acts as a "handle" to a memory-mapped k-d tree file.
    # It holds pointers to the key data arrays within the mapped memory region.
    kdtree_handle_struct_str = r"""
    typedef struct {
        // Pointers to the data within the memory-mapped file
        int32_t*       node_metadata;   // Points to the start of Payload 1 (split axes)
        MassiveParticle* particle_data;   // Points to the start of Payload 2 (reordered particles)

        // Information from the header
        uint64_t num_particles;
        uint64_t dimensions;

        // For cleanup
        void*    original_mmap_ptr; // The original pointer returned by mmap
        size_t   file_size;         // The total size of the mapped file
    } CustomKDTree;
    """

    # This struct will be used to manage the "Winners' Circle" during the search.
    winners_circle_struct_str = r"""
#define MAX_NEIGHBORS 10 // A compile-time max for the nearest-neighbor search
    typedef struct {
        int indices[MAX_NEIGHBORS];
        double sq_distances[MAX_NEIGHBORS];
        int count;
        int n_wanted;
    } WinnersCircle;
    """

    # Register all structs in a single block
    Bdefines_h.register_BHaH_defines(
        "kdtree_structs",
        f"{massive_particle_struct_str}\n{kdtree_handle_struct_str}\n{winners_circle_struct_str}"
    )
    print("    ... Registered C structs: MassiveParticle, CustomKDTree, WinnersCircle.")
```

16 Register plane event helpers

[ ]:
```
def register_plane_event_helpers():
    """
    Generates and registers the C struct and helper function for detecting
    generic plane-crossing events. By registering them here, they are made
    globally available in BHaH_defines.h.
    """
    print(" -> Registering global plane event detection helpers...")

    plane_event_helpers_str = r"""
// 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 for a generic plane crossing.
// This function is now globally visible.
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;
}
"""
    # Register these definitions to be placed in BHaH_defines.h
    Bdefines_h.register_BHaH_defines("plane_event_helpers", plane_event_helpers_str)
    print("    ... Registered plane_event_params struct and plane_event_func.")
```

### 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 (`register_custom_structures_and_params`, `g4DD_kerr_schild`, `main_production`, etc.). This populates `nrpy`'s internal library (`cfc.CFunction_dict`) with the complete definitions for all our custom C data structures and 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 (like `metric_struct` and `event_data_struct`) 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 (like `simd_intrinsics.h`) 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.c_function.register_CFunction` and writes each one into its own `.c` file (e.g., `main.c`, `connections.c`).
   - It generates `BHaH_function_prototypes.h`, a header file containing the function declarations (prototypes) for all the generated `.c` files. This is crucial as it allows the different C files to call functions defined in one another.
   - It constructs the `Makefile`, which contains the compilation and linking instructions needed to build the final executable program. It is also configured to automatically link against the required GSL and OpenMP libraries.

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

16.0.1 `nrpy` Functions Used in this Cell:

- `nrpy.infrastructures.BHaH.CodeParameters.write_CodeParameters_h_files(project_dir)`:
  - **Source File**: `nrpy/infrastructures/BHaH/CodeParameters.py`
  - **Description**: Generates the `set_CodeParameters.h` header files, which contain the C code for unpacking parameters into local variables (the "Triple-Lock" system).
- `nrpy.infrastructures.BHaH.CodeParameters.register_CFunctions_params_commondata_struct_set_to_default()`:
  - **Source File**: `nrpy/infrastructures/BHaH/CodeParameters.py`
  - **Description**: Registers the C functions that initialize the `params_struct` and `commondata_struct` with their compiled-in default values.
- `nrpy.infrastructures.BHaH.cmdline_input_and_parfiles.generate_default_parfile(project_dir, project_name)`:
  - **Source File**: `nrpy/infrastructures/BHaH/cmdline_input_and_parfiles.py`
  - **Description**: Creates the `project_name.par` file, populated with all parameters that have `add_to_parfile=True`.
- `nrpy.infrastructures.BHaH.cmdline_input_and_parfiles.register_CFunction_cmdline_input_and_parfile_parser(project_name, cmdline_inputs)`:
  - **Source File**: `nrpy/infrastructures/BHaH/cmdline_input_and_parfiles.py`
  - **Description**: Registers the C function that reads the `.par` file and command-line arguments at runtime. The `cmdline_inputs` list is critical, as it defines the exact order of expected positional command-line arguments.
- `nrpy.infrastructures.BHaH.BHaH_defines_h.output_BHaH_defines_h(project_dir)`:
  - **Source File**: `nrpy/infrastructures/BHaH/BHaH_defines_h.py`
  - **Description**: Scans `nrpy`'s internal library for all registered data structures and writes them into the master C header file, `BHaH_defines.h`.
- `nrpy.helpers.generic.copy_files(...)`:
  - **Source File**: `nrpy/helpers/generic.py`
  - **Description**: A utility function to copy files from the `nrpy` installation to the project directory.
- `nrpy.infrastructures.BHaH.Makefile_helpers.output_CFunctions_function_prototypes_and_construct_Makefile(...)`:
  - **Source File**: `nrpy/infrastructures/BHaH/Makefile_helpers.py`
  - **Description**: The final build function that generates all `.c` files, the function prototypes header, and the `Makefile`.

[ ]:
```
# This is the final, correct build cell (v2.9.1).
import os
import nrpy.helpers.generic as gh

print("\nAssembling and building C project for Time-Bundled Batch Integrator (v2.9.1)...")
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...")
# Corrected order and content
register_custom_structures_and_params()
register_plane_event_helpers()
# Physics workers and dispatchers
g4DD_kerr_schild()
con_kerr_schild()
g4DD_schwarzschild_cartesian()
con_schwarzschild_cartesian()
g4DD_metric()
connections()
# Physics and integration engines
calculate_ode_rhs()
calculate_p0_reverse()
set_initial_conditions_cartesian()
check_conservation()
lagrange_interp_engine_generic()
event_detection_manager()
handle_source_plane_intersection_engine()
calculate_and_fill_blueprint_data_universal() # Now uses correct enum
generate_granular_gsl_engines() # Now adds correct headers
generate_batch_integrator_orchestrator() # Now adds correct headers
main() # This should now be the typo-free version if you've updated that cell

# --- 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)

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',
    'debug_mode', 'perform_conservation_check'
]

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...")
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}'.")
```

# Step 9: Visualization and Analysis

This final section of the notebook is dedicated to visualizing the results produced by our C code. It contains Python code cells that use standard libraries like `numpy` and `matplotlib`/`Pillow` to process and plot the output data. These cells are for analysis and are not part of the C code generation process. They are designed to be run *after* the C code has been compiled and executed, and has produced a `blueprint.bin` file.

17 Define the blueprint

```
[ ]: import numpy as np
from PIL import Image
from typing import Union, Optional, List, Tuple
import os

# Define the exact structure of a record in the new light_blueprint.bin file.
# This must match the final C struct 'blueprint_data_t'.
BLUEPRINT_DTYPE = np.dtype([
    ('termination_type', np.int32),
    ('y_w', 'f8'),
    ('z_w', 'f8'),
    # Fields for DISK hits
    ('stokes_I', 'f8'),
    ('lambda_observed', 'f8'),
    # Fields for SOURCE PLANE hits
    ('y_s', 'f8'),
    ('z_s', 'f8'),
    # Fields for CELESTIAL SPHERE hits
    ('final_theta', 'f8'),
    ('final_phi', 'f8'),
    # Diagnostic fields
    ('L_w', 'f8'),
    ('t_w', 'f8'),
    ('L_s', 'f8'),
    ('t_s', 'f8'),
], align=False)

print("Libraries and new blueprint data type (BLUEPRINT_DTYPE) defined.")
```

```
[ ]: import numpy as np
import matplotlib.pyplot as plt
import os
from mpl_toolkits.mplot3d import Axes3D

def plot_photon_trajectory_from_debug(
    project_dir: str = "project/photon_geodesic_integrator",
    input_filename: str = "photon_path.txt"
) -> None:
    """
    Reads trajectory data from the C code's debug output and generates a
    simple 3D plot of the photon's path with a sphere at the origin.
    """
    print("--- Generating Photon Trajectory Plot from Debug File ---")

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

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

    try:
        # Load the data, skipping the header row.
        # Set invalid_raise=False to handle potential trailing empty lines.
        data = np.loadtxt(full_path, skiprows=1, ndmin=2)
        if data.shape[0] == 0:
            print("ERROR: Trajectory file is empty.")
            return

        # Columns: 0:lambda, 1:t, 2:x, 3:y, 4:z, ...
        x_coords = data[:, 2]
        y_coords = data[:, 3]
        z_coords = data[:, 4]
        print(f"Successfully loaded {len(x_coords)} data points from trajectory file.")
    except Exception as e:
        print(f"ERROR: Failed to load or parse the data file '{full_path}'.")
        print(f"Exception: {e}")
        return

    # --- 2. Set up the 3D plot ---
    plt.style.use('dark_background')
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    # --- 3. Plot the photon's trajectory ---
    ax.plot(x_coords, y_coords, z_coords, label='Photon Path', color='cyan', lw=2)

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

    # --- 4. Plot a simple sphere at the origin ---
    radius = 2.0 # Represents r=2M, the Schwarzschild event horizon
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)
    x_bh = radius * np.outer(np.cos(u), np.sin(v))
    y_bh = radius * np.outer(np.sin(u), np.sin(v))
    z_bh = radius * np.outer(np.ones(np.size(u)), np.cos(v))
    ax.plot_surface(x_bh, y_bh, z_bh, color='grey', alpha=0.5, rstride=5, cstride=5)

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

    # Set aspect ratio to be equal to avoid distortion
    max_range = np.array([x_coords.max()-x_coords.min(), y_coords.max()-y_coords.min(), z_coords.max()-z_coords.min()]).max() / 2.0
    if max_range == 0: max_range = np.max(np.abs(data[:, 2:5]))
    mid_x = (x_coords.max()+x_coords.min()) * 0.5
    mid_y = (y_coords.max()+y_coords.min()) * 0.5
    mid_z = (z_coords.max()+z_coords.min()) * 0.5
    ax.set_xlim(mid_x - max_range, mid_x + max_range)
    ax.set_ylim(mid_y - max_range, mid_y + max_range)
    ax.set_zlim(mid_z - max_range, mid_z + max_range)

    ax.set_title("Photon Trajectory (Debug Run)", fontsize=16)
    ax.legend()
    ax.view_init(elev=-90, azim=60)

    plt.show()
```

[ ]:
```
# --- How to Run ---
# After running the C code in debug mode, call this function.
plot_photon_trajectory_from_debug()
```

18 Start of Blueprint Stats (3 cells)

Cell 1 of 3

[ ]:
```
import numpy as np
import os

def view_binary_blueprint(
    blueprint_filename="project/photon_geodesic_integrator/light_blueprint.bin",
    max_rays_to_print=20
):
    """
    Reads the new binary blueprint file and prints its raw contents in a
    human-readable, context-aware format.
    """
    if not os.path.exists(blueprint_filename):
        print(f"Error: Blueprint file not found at '{blueprint_filename}'")
        return

    # This function now uses the global BLUEPRINT_DTYPE defined in the previous cell.
    data = np.fromfile(blueprint_filename, dtype=BLUEPRINT_DTYPE)

    print(f"--- Raw Blueprint Data Inspector (Radiative Transfer Version) ---")
    print(f"Total records read from file: {len(data)}\n")
    print("Printing a sample of records...")
    print("Enum Mapping: 0=FAILURE, 1=DISK, 2=SOURCE_PLANE, 3=SPHERE")

    # Updated header for new data fields
    header = f"{'Ray#':<8} | {'TermType':<12} | {'y_w':>8} | {'z_w':>8} | {'Result 1':>12} | {'Result 2':>12}"
    print(header)
    print("-" * len(header))

    if len(data) > max_rays_to_print:
        indices_to_print = np.linspace(0, len(data) - 1, max_rays_to_print, dtype=int)
    else:
        indices_to_print = np.arange(len(data))

    for i in indices_to_print:
        rec = data[i]
        term_type = int(rec['termination_type'])
        term_str_map = {0: "FAILURE", 1: "DISK", 2: "SOURCE_PLANE", 3: "SPHERE"}
        term_str = term_str_map.get(term_type, "UNKNOWN")

        res1_str, res2_str = "N/A", "N/A"
        if term_type == 1: # DISK
            res1_str = f"I={rec['stokes_I']:.3e}"
            res2_str = f"λ={rec['lambda_observed']:.2f}"
        elif term_type == 2: # SOURCE_PLANE
            res1_str = f"y_s={rec['y_s']:.3f}"
            res2_str = f"z_s={rec['z_s']:.3f}"
        elif term_type == 3: # CELESTIAL_SPHERE
            res1_str = f"θ={rec['final_theta']:.3f}"
            res2_str = f"φ={rec['final_phi']:.3f}"

        print(f"{i:<8} | {term_str:<12} | {rec['y_w']:>8.2f} | {rec['z_w']:>8.2f} | {res1_str:>12} | {res2_str:>12}")
```

Cell 2 of 3

```
[ ]: import numpy as np
     import matplotlib.pyplot as plt
     from matplotlib.colors import LogNorm
     import os

     def analyze_blueprint(blueprint_filename="project/photon_geodesic_integrator/light_blueprint.bin"):
         """
         Reads the new radiative transfer blueprint file and generates a comprehensive
         statistical analysis and a full suite of plots, including a plot of
         observed wavelength vs. the ray's position on the window plane (r_w).
         """
         # --- 1. Load Data ---
         if not os.path.exists(blueprint_filename):
             print(f"Error: Blueprint file not found at '{blueprint_filename}'")
             return

         try:
             data = np.fromfile(blueprint_filename, dtype=BLUEPRINT_DTYPE)
         except NameError:
             print("ERROR: BLUEPRINT_DTYPE is not defined. Please run the cell that defines it first.")
             return

         if len(data) == 0:
             print("Blueprint file is empty. No analysis to perform.")
             return

         # --- 2. Segregate Data by Termination Type ---
         failure_hits = data[data['termination_type'] == 0]
         disk_hits = data[data['termination_type'] == 1]
         source_plane_hits = data[data['termination_type'] == 2]
         sphere_hits = data[data['termination_type'] == 3]

         num_total_rays = len(data)
         num_failure = len(failure_hits)
         num_disk = len(disk_hits)
         num_source_plane = len(source_plane_hits)
         num_sphere = len(sphere_hits)

         print("--- Blueprint File Analysis (Radiative Transfer Architecture) ---")
         print(f"Total rays in scan: {num_total_rays}")
         print(f"  Rays that hit the DISK:             {num_disk} ({100.0 * num_disk / num_total_rays:.2f}%)")
         print(f"  Rays that hit the SOURCE_PLANE:     {num_source_plane} ({100.0 * num_source_plane / num_total_rays:.2f}%)")
         print(f"  Rays that hit the CELESTIAL_SPHERE:  {num_sphere} ({100.0 * num_sphere / num_total_rays:.2f}%)")
```

```
    print(f"  Rays that FAILED (e.g., shadow):     {num_failure} ({100.0 * num_failure / num_total_rays:.2f}%)")

    if num_failure > 0:
        r_w_failure = np.sqrt(failure_hits['y_w']**2 + failure_hits['z_w']**2)
        if len(r_w_failure) > 0 and np.any(np.isfinite(r_w_failure)):
            print(f"    -> Apparent radius of black hole shadow on window: {np.nanmax(r_w_failure):.4f} M")

    print("-" * 60)

    # --- 3. Detailed Statistical Analysis (All sections restored) ---
    if num_disk > 0:
        valid_disk_hits = disk_hits[np.isfinite(disk_hits['stokes_I']) & np.isfinite(disk_hits['lambda_observed'])]
        if len(valid_disk_hits) > 0:
            intensities = valid_disk_hits['stokes_I']
            wavelengths = valid_disk_hits['lambda_observed']
            print("\n--- Detailed Disk Hit Statistics ---")
            print("\n  Observed Intensity (Stokes I):")
            print(f"    Min / Max:    {np.min(intensities):.3e} / {np.max(intensities):.3e}")
            print(f"    Mean / Median:  {np.mean(intensities):.3e} / {np.median(intensities):.3e}")
            percentiles_I = np.percentile(intensities, [10, 25, 75, 90])
            print(f"    Percentiles:  10th={percentiles_I[0]:.3e}, 25th={percentiles_I[1]:.3e}, 75th={percentiles_I[2]:.3e},␣
↪90th={percentiles_I[3]:.3e}")

            print("\n  Observed Wavelength (lambda_obs) in nm:")
            print(f"    Min / Max:    {np.min(wavelengths):.2f} / {np.max(wavelengths):.2f}")
            print(f"    Mean / Median:  {np.mean(wavelengths):.2f} / {np.median(wavelengths):.2f}")
            percentiles_L = np.percentile(wavelengths, [10, 25, 75, 90])
            print(f"    Percentiles:  10th={percentiles_L[0]:.2f}, 25th={percentiles_L[1]:.2f}, 75th={percentiles_L[2]:.2f},␣
↪90th={percentiles_L[3]:.2f}")

    if num_source_plane > 0:
        valid_plane_hits = source_plane_hits[np.isfinite(source_plane_hits['y_s']) & np.isfinite(source_plane_hits['z_s'])]
        if len(valid_plane_hits) > 0:
            r_s = np.sqrt(valid_plane_hits['y_s']**2 + valid_plane_hits['z_s']**2)
            print("\n--- Source Plane Hit Statistics ---")
            print(f"  Planar Radius (r_s) for these hits: min={np.min(r_s):.4f}, max={np.max(r_s):.4f}, mean={np.mean(r_s):.4f}")
            print(f"  Intersection Time (t_s) for these hits: min={np.min(valid_plane_hits['t_s']):.2f}, max={np.
↪max(valid_plane_hits['t_s']):.2f}, mean={np.mean(valid_plane_hits['t_s']):.2f}")

    if num_sphere > 0:
        valid_sphere_hits = sphere_hits[np.isfinite(sphere_hits['final_theta']) & np.isfinite(sphere_hits['final_phi'])]
        if len(valid_sphere_hits) > 0:
            print("\n--- Celestial Sphere Hit Statistics ---")
            print(f"  Final Angle (theta) for these hits: min={np.min(valid_sphere_hits['final_theta']):.4f}, max={np.
↪max(valid_sphere_hits['final_theta']):.4f}, mean={np.mean(valid_sphere_hits['final_theta']):.4f}")
            print(f"  Final Angle (phi)   for these hits: min={np.min(valid_sphere_hits['final_phi']):.4f}, max={np.
↪max(valid_sphere_hits['final_phi']):.4f}, mean={np.mean(valid_sphere_hits['final_phi']):.4f}")

    # --- 4. Generate Expanded Plots (3x2 Grid) ---
    fig, axes = plt.subplots(3, 2, figsize=(22, 27))
    fig.suptitle("Blueprint Data Visualization (Radiative Transfer Architecture)", fontsize=20)
    ax = axes.ravel()

    # Plot 0: Observed Intensity (Stokes I) on Window
    if num_disk > 0 and 'valid_disk_hits' in locals() and len(valid_disk_hits) > 0:
        hist = ax[0].hist2d(valid_disk_hits['y_w'], valid_disk_hits['z_w'],
                            bins=256, cmap='inferno', norm=LogNorm(),
                            weights=valid_disk_hits['stokes_I'])
        ax[0].set_title("Observed Intensity (Stokes I) on Window Plane")
        ax[0].set_xlabel("y_w (M)"); ax[0].set_ylabel("z_w (M)")
        ax[0].set_aspect('equal', 'box')
        fig.colorbar(hist[3], ax=ax[0], label="Intensity")
    else:
        ax[0].text(0.5, 0.5, "No Disk Hits", ha='center', va='center', fontsize=16)
        ax[0].set_title("Observed Intensity (Stokes I)")

    # Plot 1: Observed Wavelength on Window
    if num_disk > 0 and 'valid_disk_hits' in locals() and len(valid_disk_hits) > 0:
        lambda_min = np.min(valid_disk_hits['lambda_observed'])
        lambda_max = np.max(valid_disk_hits['lambda_observed'])
        sc = ax[1].scatter(valid_disk_hits['y_w'], valid_disk_hits['z_w'],
                           c=valid_disk_hits['lambda_observed'], cmap='jet', s=2, vmin=lambda_min, vmax=lambda_max)
        ax[1].set_title("Observed Wavelength (λ_obs) on Window Plane")
        ax[1].set_xlabel("y_w (M)"); ax[1].set_ylabel("z_w (M)")
        ax[1].set_aspect('equal', 'box')
        ax[1].set_facecolor('black')
        fig.colorbar(sc, ax=ax[1], label="Wavelength (nm)")
    else:
        ax[1].text(0.5, 0.5, "No Disk Hits", ha='center', va='center', fontsize=16)
        ax[1].set_title("Observed Wavelength (λ_obs)")

    # PLOT 2 (CORRECTED): Observed Wavelength vs. WINDOW Radius
    if num_disk > 0 and 'valid_disk_hits' in locals() and len(valid_disk_hits) > 0:
        r_w = np.sqrt(valid_disk_hits['y_w']**2 + valid_disk_hits['z_w']**2)
        hist2 = ax[2].hist2d(r_w, valid_disk_hits['lambda_observed'],
                             bins=(200, 200), cmap='magma', norm=LogNorm())
        ax[2].set_title("Observed Wavelength vs. Apparent Radius on Window")
        ax[2].set_xlabel("Apparent Radius on Window (r_w) [M]")
        ax[2].set_ylabel("Observed Wavelength (λ_obs) [nm]")
        fig.colorbar(hist2[3], ax=ax[2], label="Number of Rays")
        ax[2].grid(True, linestyle='--', alpha=0.5)
    else:
        ax[2].text(0.5, 0.5, "No Disk Hits", ha='center', va='center', fontsize=16)
        ax[2].set_title("Observed Wavelength vs. Window Radius")

    # Plot 3: Intensity Distribution
    if num_disk > 0 and 'valid_disk_hits' in locals() and len(valid_disk_hits) > 0:
        ax[3].hist(valid_disk_hits['stokes_I'], bins=100, color='cyan', log=True)
        ax[3].set_title("Distribution of Observed Intensities")
        ax[3].set_xlabel("Observed Intensity (Stokes I)")
        ax[3].set_ylabel("Number of Rays (Log Scale)")
        ax[3].grid(True, linestyle='--', alpha=0.5)
    else:
        ax[3].text(0.5, 0.5, "No Disk Hits", ha='center', va='center', fontsize=16)
        ax[3].set_title("Distribution of Observed Intensities")

    # Plot 4: Termination Type Distribution
    labels = ['Failure', 'Disk Hits', 'Source Plane', 'Celestial Hits']
    sizes = [num_failure, num_disk, num_source_plane, num_sphere]
    colors = ['#2F2F2F', '#FFC300', '#581845', '#C70039']
    explode = (0.1, 0, 0, 0)
    plot_labels = [l for i, l in enumerate(labels) if sizes[i] > 0]
    plot_sizes = [s for s in sizes if s > 0]
    plot_colors = [c for i, c in enumerate(colors) if sizes[i] > 0]
    plot_explode = [e for i, e in enumerate(explode) if sizes[i] > 0]
    if plot_sizes:
        ax[4].pie(plot_sizes, explode=plot_explode, labels=plot_labels, colors=plot_colors,
                  autopct='%1.1f%%', shadow=True, startangle=140)
        ax[4].set_title("Distribution of Ray Termination Types")
        ax[4].axis('equal')
    else:
        ax[4].text(0.5, 0.5, "No Data to Plot", ha='center', va='center', fontsize=16)
        ax[4].set_title("Distribution of Ray Termination Types")

    # Plot 5: Photon Outcome by Window Radius
    if num_total_rays > 0:
        valid_data = data[np.isfinite(data['y_w'])]
        r_w = np.sqrt(valid_data['y_w']**2 + valid_data['z_w']**2)
        r_w_failure = r_w[valid_data['termination_type'] == 0]
        r_w_disk    = r_w[valid_data['termination_type'] == 1]
        r_w_source  = r_w[valid_data['termination_type'] == 2]
        r_w_sphere  = r_w[valid_data['termination_type'] == 3]
        x_data = [r_w_failure, r_w_disk, r_w_source, r_w_sphere]
        hist_labels = [f'FAILURE ({len(r_w_failure)})', f'DISK ({len(r_w_disk)})', f'SOURCE ({len(r_w_source)})', f'SPHERE␣
↪({len(r_w_sphere)})']
        hist_colors = ['black', 'gold', 'purple', 'deepskyblue']
        ax[5].hist(x_data, bins=100, stacked=True, label=hist_labels, color=hist_colors, edgecolor='dimgray')
        ax[5].set_title("Photon Outcome by Initial Window Radius")
        ax[5].set_xlabel("Initial Radial Distance on Window (r_w)")
        ax[5].set_ylabel("Number of Photons")
        ax[5].legend()

    plt.tight_layout(rect=[0, 0.03, 1, 0.97])
    plt.show()
```

Cell 3 of 3

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

     def plot_stacked_radial_histogram(
         blueprint_filename: str = "project/photon_geodesic_integrator/light_blueprint.bin",
         bin_width: float = 0.05
     ):
         """
         Reads the new radiative transfer blueprint file and creates a stacked
         histogram showing the outcome of photons as a function of their
         initial radial distance on the camera's window plane.

         Args:
             blueprint_filename: The path to the light_blueprint.bin file.
             bin_width: The width of each radial bin for the histogram.
         """
         print(f"--- Generating stacked radial histogram for '{blueprint_filename}' ---")

         # --- Load Data ---
         if not os.path.exists(blueprint_filename):
             print(f"Error: Blueprint file not found at '{blueprint_filename}'")
             return

         # This function uses the global BLUEPRINT_DTYPE defined in a previous cell.
         # It must be executed after the cell that defines BLUEPRINT_DTYPE.
         try:
             data = np.fromfile(blueprint_filename, dtype=BLUEPRINT_DTYPE)
         except NameError:
```

```
        print("ERROR: BLUEPRINT_DTYPE is not defined. Please run the cell that defines it first.")
        return

    if len(data) == 0:
        print("Blueprint file is empty. Cannot generate plot.")
        return

    # --- Calculate r_w for all rays ---
    # We must filter out non-finite values that can corrupt statistics
    valid_data = data[np.isfinite(data['y_w']) & np.isfinite(data['z_w'])]
    r_w = np.sqrt(valid_data['y_w']**2 + valid_data['z_w']**2)

    # --- Separate r_w values based on NEW termination types ---
    mask_failure = (valid_data['termination_type'] == 0)
    mask_disk    = (valid_data['termination_type'] == 1)
    mask_source  = (valid_data['termination_type'] == 2)
    mask_sphere  = (valid_data['termination_type'] == 3)

    r_w_failure = r_w[mask_failure]
    r_w_disk    = r_w[mask_disk]
    r_w_source  = r_w[mask_source]
    r_w_sphere  = r_w[mask_sphere]

    # --- Create the Bins for the Histogram ---
    if len(r_w) == 0:
        print("No valid ray data with finite window coordinates found.")
        return
    max_radius = r_w.max()
    bins = np.arange(0, max_radius + bin_width, bin_width)

    # --- Create the Plot ---
    plt.style.use('seaborn-v0_8-whitegrid')
    fig, ax = plt.subplots(figsize=(14, 7))

    # Data and labels for the stacked histogram
    x_data = [r_w_failure, r_w_disk, r_w_source, r_w_sphere]
    labels = [
        f'FAILURE (Shadow): {len(r_w_failure)}',
        f'DISK HIT: {len(r_w_disk)}',
        f'SOURCE PLANE: {len(r_w_source)}',
        f'SPHERE: {len(r_w_sphere)}'
    ]
    colors = ['black', 'gold', 'purple', 'deepskyblue']
    # Create the stacked histogram
    ax.hist(x_data, bins=bins, stacked=True, label=labels, color=colors, edgecolor='dimgray')

    # --- Add Labels and Title ---
    title = f"Photon Outcome by Initial Radial Distance (Bin Width: {bin_width})\nTotal Rays: {len(data)}"
    ax.set_title(title, fontsize=16)
    ax.set_xlabel('Initial Radial Distance on Window ($r_w$)', fontsize=12)
    ax.set_ylabel('Number of Photons (Count)', fontsize=12)
    ax.legend(title='Termination Type')
    ax.grid(True, which='both', linestyle='--', linewidth=0.5)

    plt.show()
```

19 Wavelength to rbg

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

     def wavelength_to_rgb(wavelength_nm, min_vis_wl=400.0, max_vis_wl=700.0):
         """
         Converts an array of wavelengths in nanometers to an array of RGB colors.

         This is a fully vectorized, high-performance version that uses numpy
         operations to avoid slow Python loops.
         """
         # Ensure input is a numpy array
         wavelengths = np.asarray(wavelength_nm)

         # Normalize the wavelengths to the range [0, 1]
         norm_wl = (wavelengths - min_vis_wl) / (max_vis_wl - min_vis_wl)
         norm_wl = np.clip(norm_wl, 0.0, 1.0)

         # Use a perceptually uniform colormap
         colormap = plt.get_cmap('jet')

         # The colormap function can be applied to the entire array at once.
         # It returns an (N, 4) RGBA array.
         colors_rgba = colormap(norm_wl)

         # We only need the first three RGB components.
         return colors_rgba[:, :3]
     print("Advanced, false-color `wavelength_to_rgb` function defined.")
```

```
[ ]: import numpy as np
     import matplotlib.pyplot as plt
     import os
     from mpl_toolkits.mplot3d import Axes3D

     def plot_photon_trajectory_simple(
         project_dir: str = "project/photon_geodesic_integrator",
         input_filename: str = "photon_path.txt"
     ) -> None:
         """
         Reads trajectory data from the C code's debug output and generates a
         simple 3D plot of the photon's path with a sphere at the origin.
         """
         print("--- Generating Simple Photon Trajectory Plot ---")

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

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

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

         # --- 2. Set up the 3D plot ---
         plt.style.use('dark_background')
         fig = plt.figure(figsize=(10, 8))
         ax = fig.add_subplot(111, projection='3d')

         # --- 3. Plot the photon's trajectory ---
         ax.plot(x_coords, y_coords, z_coords, label='Photon Path', color='cyan', lw=2)

         # Mark the start (camera) and end points
```

```
    ax.scatter(x_coords[0], y_coords[0], z_coords[0], color='lime', s=100, label='Start (Camera)', marker='o', depthshade=False)
    ax.scatter(x_coords[-1], y_coords[-1], z_coords[-1], color='red', s=100, label='End Point', marker='X', depthshade=False)

    # --- 4. Plot a simple sphere at the origin ---
    radius = 2.0 # Represents r=2M, the Schwarzschild event horizon
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)
    x_bh = radius * np.outer(np.cos(u), np.sin(v))
    y_bh = radius * np.outer(np.sin(u), np.sin(v))
    z_bh = radius * np.outer(np.ones(np.size(u)), np.cos(v))
    ax.plot_surface(x_bh, y_bh, z_bh, color='grey', alpha=0.5, rstride=5, cstride=5, label='Black Hole (r=2M)')

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

    # Set aspect ratio to be equal
    max_range = np.array([x_coords.max()-x_coords.min(), y_coords.max()-y_coords.min(), z_coords.max()-z_coords.min()]).max() / 2.0
    mid_x = (x_coords.max()+x_coords.min()) * 0.5
    mid_y = (y_coords.max()+y_coords.min()) * 0.5
    mid_z = (z_coords.max()+z_coords.min()) * 0.5
    ax.set_xlim(mid_x - max_range, mid_x + max_range)
    ax.set_ylim(mid_y - max_range, mid_y + max_range)
    ax.set_zlim(mid_z - max_range, mid_z + max_range)

    ax.set_title("Photon Trajectory", fontsize=16)
    ax.legend()
    ax.view_init(elev=30., azim=-60)

    plt.show()

# --- How to Run ---
# After running the C code in debug mode, call this function.

plot_photon_trajectory_simple()
```

20 Start of Visual function definitions

21 Texture Loading Helper Function

[ ]:
```python
# In file: V11_0_Python_to_C_via_NRPy.ipynb
# In the cell defining _load_texture

def _load_texture(image_input: Union[str, np.ndarray]) -> np.ndarray:
    """
    Helper function to load an image or use a pre-loaded numpy array,
    ensuring the output is always a float64 array with values in [0.0, 1.0].
    """
    if isinstance(image_input, str):
        if not os.path.exists(image_input):
            raise FileNotFoundError(f"Texture file not found: {image_input}")
        with Image.open(image_input) as img:
            # Convert to RGB and normalize to [0.0, 1.0] floats
            return np.array(img.convert("RGB")).astype(np.float64) / 255.0
    elif isinstance(image_input, np.ndarray):
        # --- CORRECTED LOGIC ---
        # 1. Ensure the array is float64 for calculations.
        texture_array = image_input.astype(np.float64)
        # 2. Check if the values are in the [0, 255] range. If so, normalize them.
        if np.max(texture_array) > 1.0:
            texture_array /= 255.0
        return texture_array
    else:
        raise TypeError("Image input must be a file path (str) or a NumPy array.")

print("Helper function `_load_texture` (Corrected) defined.")
```

# Procedural Disk Generators (2 cells) First cell

[ ]:
```python
# Cell for Visualizing/Generating the Unlensed Source Disk (UPDATED with Anti-Aliasing)

import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def generate_source_disk_array(
    pixel_width=512,
    disk_physical_width=40.0,
    disk_inner_radius=6.0,
    disk_outer_radius=20.0,
    disk_temp_power_law=-0.75,
    colormap='hot',
    display_image=True
):
    """
    Generates an anti-aliased NumPy array of an accretion disk image.
    """
    # --- 1. Create a Coordinate Grid ---
    half_width = disk_physical_width / 2.0
    y_coords = np.linspace(-half_width, half_width, pixel_width)
    z_coords = np.linspace(-half_width, half_width, pixel_width)
    yy, zz = np.meshgrid(y_coords, z_coords)

    # --- 2. Calculate Physical Properties for Each Pixel ---
    radii = np.sqrt(yy**2 + zz**2)

    # --- 3. Apply the Disk Model with a Smooth Falloff ---
    # Instead of a sharp mask, we'll calculate temperature for all points
    # and then smoothly fade it to zero outside the disk bounds.

    # Calculate temperature based on the power law everywhere.
    # Add a small epsilon to radii to avoid division by zero at the center.
    temperature = (radii / disk_inner_radius)**disk_temp_power_law

    # Create a smooth falloff mask using numpy.clip
    # This will create a smooth transition from 1 (inside the disk) to 0 (outside)
    # over a small number of pixels. Let's define a transition width.
    transition_width = 2.0 * (disk_physical_width / pixel_width) # Width of 2 pixels

    # Inner edge falloff
    inner_falloff = np.clip((radii - (disk_inner_radius - transition_width)) / transition_width, 0, 1)

    # Outer edge falloff
    outer_falloff = 1.0 - np.clip((radii - disk_outer_radius) / transition_width, 0, 1)

    # Combine the masks and apply to the temperature
    smooth_mask = inner_falloff * outer_falloff
    temperature *= smooth_mask

    # --- 4. Map Temperature to Color and Create Image Array ---
    colormap_func = plt.colormaps[colormap]
    colors = colormap_func(temperature / np.max(temperature)) # Normalize to ensure max is 1
    image_array = (colors[:, :, :3] * 255).astype(np.uint8)
    # --- 5. Optionally Display the Image ---
    if display_image:
        print(f"Displaying the unlensed source disk (with anti-aliasing):")
        img = Image.fromarray(image_array)
        plt.figure(figsize=(8, 8))
        plt.imshow(img)
        plt.title("Unlensed Source Accretion Disk (Anti-Aliased)")
        plt.show()

    # --- 6. Return the NumPy array ---
    return image_array
```

Second Cell

[ ]:
```
# In file: V11_0_Python_to_C_via_NRPy.ipynb
# In the cell defining generate_advanced_disk_array

import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def generate_advanced_disk_array(
    # --- Basic Settings ---
    pixel_width=1024,
    disk_physical_width=30.0,
    colormap='afmhot',

    # --- Base Radial Profile ---
    disk_inner_radius=6.0,
    disk_outer_radius=25.0,
    disk_temp_power_law=-2.0,

    # --- Concentric Rings / Gaps ---
    ring_num=5,
    ring_contrast=0.7,
    ring_log_spacing=True,

    # --- Doppler Beaming (Asymmetric Brightness) ---
    doppler_factor=0.8,
    doppler_power=3,

    # --- Optional Features (set to 0 to disable) ---
    hotspot_num=0,
    hotspot_amplitude=0.15,
    hotspot_radius_center=9.0,
    hotspot_radius_width=3.0,
    shape_num_lobes=0,
    shape_inner_amplitude=0.0,
    shape_outer_amplitude=0.0,

    # --- Display Control ---
    display_image=True
):
    """
    Generates an EHT-style disk, combining advanced features with robust NaN handling.
    """
    print("--- Generating EHT-Style Accretion Disk Texture (Corrected v3) ---")

    # 1. Create Coordinate Grid & Polar Coordinates
    half_width = disk_physical_width / 2.0
    y_coords = np.linspace(-half_width, half_width, pixel_width)
    z_coords = np.linspace(-half_width, half_width, pixel_width)
    yy, zz = np.meshgrid(y_coords, z_coords)
    radii = np.sqrt(yy**2 + zz**2)
    phi = np.arctan2(zz, yy)

    # 2. Calculate Base Temperature
    # Add a small epsilon to radii to avoid division by zero at the center.
    temperature = (radii / (disk_inner_radius + 1e-12))**disk_temp_power_law

    # --- CORRECTED: Explicitly handle the NaN at the center ---
    # This is the crucial fix from the older, working code.
    temperature[np.isnan(temperature)] = 0

    # 3. Apply Modulations (Rings, Doppler, etc.)
    if ring_num > 0:
        if ring_log_spacing:
            radial_coord = np.log(radii / disk_inner_radius + 1e-9)
            max_log_rad = np.log(disk_outer_radius / disk_inner_radius)
            ring_mod = 0.5 * (1 + np.cos(ring_num * 2 * np.pi * radial_coord / max_log_rad))
        else:
            radial_coord = radii
            ring_mod = 0.5 * (1 + np.cos(ring_num * 2 * np.pi * (radial_coord - disk_inner_radius) / (disk_outer_radius -␣
↪disk_inner_radius)))
        ring_mod = 1.0 - ring_contrast * (1.0 - ring_mod)
        temperature *= ring_mod

    if doppler_factor > 0:
        doppler_mod = (1 + doppler_factor * (-np.cos(phi)))**doppler_power
        temperature *= doppler_mod

    # ... (hotspot logic would go here) ...

    # 4. Apply Final Radial Mask
    # This uses the smooth falloff logic from the older, working code.
    transition_width = 2.0 * (disk_physical_width / pixel_width)
    r_inner_mod = disk_inner_radius + shape_inner_amplitude * np.cos(shape_num_lobes * phi)
    r_outer_mod = disk_outer_radius + shape_outer_amplitude * np.cos(shape_num_lobes * phi)
    inner_falloff = np.clip((radii - (r_inner_mod - transition_width)) / transition_width, 0, 1)
    outer_falloff = 1.0 - np.clip((radii - r_outer_mod) / transition_width, 0, 1)
    smooth_mask = inner_falloff * outer_falloff
    temperature *= smooth_mask

    # 5. Map to Color using robust normalization
    max_temp = np.max(temperature)
    if max_temp > 0:
        norm_temperature = temperature / max_temp
    else:
        norm_temperature = temperature # Avoid division by zero if all temps are zero

    colormap_func = plt.colormaps[colormap]
    colors = colormap_func(norm_temperature)
    image_array = (colors[:, :, :3] * 255).astype(np.uint8)

    # 6. Display
    if display_image:
        print("Displaying the unlensed advanced source disk:")
        img = Image.fromarray(image_array)
        plt.figure(figsize=(8, 8))
        plt.imshow(img, extent=[-half_width, half_width, -half_width, half_width])
        plt.title("Advanced, EHT-Style Source Disk (Corrected)")
        plt.xlabel("y (M)")
        plt.ylabel("z (M)")
        plt.gca().invert_yaxis()
        plt.show()

    return image_array
```

22 Static lensed image

[ ]:
```python
def generate_static_lensed_image(
    output_filename: str,
    output_pixel_width: int,
    source_image_width: float,
    sphere_image: Union[str, np.ndarray],
    source_image: Union[str, np.ndarray],
    intensity_scale: float = 1.0,
    lambda_min_nm: Optional[float] = None,
    lambda_max_nm: Optional[float] = None,
    gamma: float = 2.2,
    blueprint_filename: str = "project/photon_geodesic_integrator/light_blueprint.bin",
    window_width: Optional[float] = None,
    zoom_region: Optional[Union[List[float], Tuple[float, float, float, float]]] = None
) -> None:
    """
    Generates a lensed image using a multi-layer composite with selective tone mapping.

    This definitive version applies non-linear gamma correction only to the
    high-dynamic-range disk light, preserving the linear color of background
    sources, and then additively blends them for a physically realistic image.
    """
    print(f"--- Generating Static Lensed Image (Selective Tone Mapping): '{output_filename}' ---")

    # --- Phase 1: Initialization and Data Loading ---
    if not os.path.exists(blueprint_filename):
        raise FileNotFoundError(f"Blueprint file not found: {blueprint_filename}")
    blueprint_data = np.fromfile(blueprint_filename, dtype=BLUEPRINT_DTYPE)
    if zoom_region:
        y_w_min, y_w_max, z_w_min, z_w_max = zoom_region
    elif window_width:
        half_w = window_width / 2.0
        y_w_min, y_w_max = -half_w, half_w
        z_w_min, z_w_max = -half_w, half_w
    else:
        raise ValueError("Either 'window_width' or 'zoom_region' must be provided.")
    window_y_range = y_w_max - y_w_min
    window_z_range = z_w_max - z_w_min
    aspect_ratio = window_z_range / window_y_range
    output_pixel_height = int(output_pixel_width * aspect_ratio)
    source_texture = _load_texture(source_image)
    sphere_texture = _load_texture(sphere_image)
```

```
# Separate accumulators for disk (foreground) and other (background) layers
disk_pixel_accumulator = np.zeros((output_pixel_height, output_pixel_width, 3), dtype=np.float64)
background_pixel_accumulator = np.zeros((output_pixel_height, output_pixel_width, 3), dtype=np.float64)
# A single counter for all successful rays per pixel
count_accumulator = np.zeros((output_pixel_height, output_pixel_width), dtype=np.int32)

# --- Phase 2: Vectorized Ray Processing ---
mask_in_view = (
    (blueprint_data['y_w'] >= y_w_min) & (blueprint_data['y_w'] < y_w_max) &
    (blueprint_data['z_w'] >= z_w_min) & (blueprint_data['z_w'] < z_w_max)
)
rays_in_view = blueprint_data[mask_in_view]

if len(rays_in_view) > 0:
    px_float = (rays_in_view['y_w'] - y_w_min) / window_y_range * output_pixel_width
    py_float = (z_w_max - rays_in_view['z_w']) / window_z_range * output_pixel_height
    px = np.clip(px_float, 0, output_pixel_width - 1).astype(np.int32)
    py = np.clip(py_float, 0, output_pixel_height - 1).astype(np.int32)

    # --- Process each termination type into its correct layer ---

    # 1. DISK HITS -> Disk Layer
    is_disk = rays_in_view['termination_type'] == 1
    if np.any(is_disk):
        disk_hits = rays_in_view[is_disk]
        valid_disk_hits = disk_hits[np.isfinite(disk_hits['lambda_observed'])]
        if len(valid_disk_hits) > 0:
            lambda_min = lambda_min_nm if lambda_min_nm is not None else np.min(valid_disk_hits['lambda_observed'])
            lambda_max = lambda_max_nm if lambda_max_nm is not None else np.max(valid_disk_hits['lambda_observed'])
            base_colors = wavelength_to_rgb(disk_hits['lambda_observed'], min_vis_wl=lambda_min, max_vis_wl=lambda_max)
            intensities = disk_hits['stokes_I'][:, np.newaxis]
            disk_colors = base_colors * intensities * intensity_scale
            np.add.at(disk_pixel_accumulator, (py[is_disk], px[is_disk]), disk_colors)

    # 2. SOURCE PLANE HITS -> Background Layer
    is_source_plane = rays_in_view['termination_type'] == 2
    if np.any(is_source_plane):
        source_plane_hits = rays_in_view[is_source_plane]
        source_pixel_height, source_pixel_width, _ = source_texture.shape
        half_sw = source_image_width / 2.0
        norm_y = (source_plane_hits['y_s'] + half_sw) / source_image_width
        norm_z = (source_plane_hits['z_s'] + half_sw) / source_image_width
        px_s = norm_y * (source_pixel_width - 1)
        py_s = (1.0 - norm_z) * (source_pixel_height - 1)
```

```
            px_s_int = np.clip(px_s, 0, source_pixel_width - 1).astype(np.int32)
            py_s_int = np.clip(py_s, 0, source_pixel_height - 1).astype(np.int32)
            source_colors = source_texture[py_s_int, px_s_int]
            np.add.at(background_pixel_accumulator, (py[is_source_plane], px[is_source_plane]), source_colors)

        # 3. CELESTIAL SPHERE HITS -> Background Layer
        is_sphere = rays_in_view['termination_type'] == 3
        if np.any(is_sphere):
            sphere_hits = rays_in_view[is_sphere]
            sphere_pixel_height, sphere_pixel_width, _ = sphere_texture.shape
            norm_phi = (sphere_hits['final_phi'] + np.pi) / (2 * np.pi)
            norm_theta = sphere_hits['final_theta'] / np.pi
            px_sph = norm_phi * (sphere_pixel_width - 1)
            py_sph = norm_theta * (sphere_pixel_height - 1)
            px_sph_int = np.clip(px_sph, 0, sphere_pixel_width - 1).astype(np.int32)
            py_sph_int = np.clip(py_sph, 0, sphere_pixel_height - 1).astype(np.int32)
            sphere_colors = sphere_texture[py_sph_int, px_sph_int]
            np.add.at(background_pixel_accumulator, (py[is_sphere], px[is_sphere]), sphere_colors)

        # Increment the count for every ray that successfully terminated
        np.add.at(count_accumulator, (py, px), 1)

    # --- Phase 3: Assembling Final Image with Selective Tone Mapping ---
    hit_pixels_mask = count_accumulator > 0

    # 1. Calculate the average color for each layer separately
    avg_disk_color = np.zeros_like(disk_pixel_accumulator)
    avg_background_color = np.zeros_like(background_pixel_accumulator)

    # Use the mask to avoid division by zero for pixels that were not hit
    avg_disk_color[hit_pixels_mask] = disk_pixel_accumulator[hit_pixels_mask] / count_accumulator[hit_pixels_mask, np.newaxis]
    avg_background_color[hit_pixels_mask] = background_pixel_accumulator[hit_pixels_mask] / count_accumulator[hit_pixels_mask, np.
↪newaxis]

    # 2. Apply Tone Mapping ONLY to the averaged disk layer
    tone_mapped_disk_color = np.zeros_like(avg_disk_color)
    disk_hit_mask = np.any(avg_disk_color > 0, axis=2) # Find pixels that actually have disk light

    # --- THIS IS THE CORRECTED BLOCK ---
    # Only perform normalization and gamma if there were actual disk hits.
    if np.any(disk_hit_mask):
        max_disk_brightness = np.max(avg_disk_color[disk_hit_mask])
        normalized_disk_color = avg_disk_color[disk_hit_mask]
        if max_disk_brightness > 0:
            normalized_disk_color /= max_disk_brightness

        scaled_disk_color = normalized_disk_color * intensity_scale

        tone_mapped_disk_color[disk_hit_mask] = scaled_disk_color ** (1.0 / gamma)
    # --- END OF CORRECTION ---


    # 3. Additively blend the tone-mapped disk with the linear background
    final_image_float = tone_mapped_disk_color + avg_background_color

    # 4. Convert to final image format
    final_image_uint8 = (np.clip(final_image_float, 0, 1) * 255).astype(np.uint8)
    img = Image.fromarray(final_image_uint8, 'RGB')

    output_dir = os.path.dirname(output_filename)
    if output_dir:
        os.makedirs(output_dir, exist_ok=True)

    img.save(output_filename)
    print(f"--- Static image saved to '{output_filename}' ---")
```

23 Rotating source image generator

```
[ ]: def generate_animated_lensed_image(
         output_filename: str,
         output_pixel_width: int,
         source_image_width: float,
         sphere_image: Union[str, np.ndarray],
         source_image: Union[str, np.ndarray],
         # --- Parameters for orbital dynamics ---
         M_scale: float,
         t_anim: float,
         prograde_disk: bool = True,
         # ---
         blueprint_filename: str = "project/photon_geodesic_integrator/blueprint.bin",
         window_width: Optional[float] = None,
         zoom_region: Optional[Union[List[float], Tuple[float, float, float, float]]] = None
     ) -> None:
         """
         Generates a lensed image of a DYNAMIC, differentially rotating Keplerian disk.

         This version uses the animation time (t_anim) and light-travel time (t_s)
```

```
    to calculate the rotational position of the disk at the moment of emission.
    """
    # The body of this function is identical to the one I provided previously,
    # as its logic for handling animation was already correct.

    # --- Phase 1: Initialization and Setup ---
    if zoom_region:
        y_w_min, y_w_max, z_w_min, z_w_max = zoom_region
    elif window_width:
        half_w = window_width / 2.0
        y_w_min, y_w_max = -half_w, half_w
        z_w_min, z_w_max = -half_w, half_w
    else:
        raise ValueError("Either 'window_width' or 'zoom_region' must be provided.")

    window_y_range = y_w_max - y_w_min
    window_z_range = z_w_max - z_w_min
    aspect_ratio = window_z_range / window_y_range
    output_pixel_height = int(output_pixel_width * aspect_ratio)

    source_texture = _load_texture(source_image)
    sphere_texture = _load_texture(sphere_image)
    source_pixel_height, source_pixel_width, _ = source_texture.shape
    sphere_pixel_height, sphere_pixel_width, _ = sphere_texture.shape

    if not os.path.exists(blueprint_filename):
        raise FileNotFoundError(f"Blueprint file not found: {blueprint_filename}")
    blueprint_data = np.fromfile(blueprint_filename, dtype=BLUEPRINT_DTYPE)

    pixel_accumulator = np.zeros((output_pixel_height, output_pixel_width, 3), dtype=np.float64)
    count_accumulator = np.zeros((output_pixel_height, output_pixel_width), dtype=np.int32)

    # --- Phase 2: Vectorized Ray Processing ---
    mask_in_view = (
        (blueprint_data['y_w'] >= y_w_min) & (blueprint_data['y_w'] < y_w_max) &
        (blueprint_data['z_w'] >= z_w_min) & (blueprint_data['z_w'] < z_w_max)
    )
    rays_in_view = blueprint_data[mask_in_view]

    if len(rays_in_view) > 0:
        px_float = (rays_in_view['y_w'] - y_w_min) / window_y_range * output_pixel_width
        py_float = (z_w_max - rays_in_view['z_w']) / window_z_range * output_pixel_height
        px = np.clip(px_float, 0, output_pixel_width - 1).astype(np.int32)
        py = np.clip(py_float, 0, output_pixel_height - 1).astype(np.int32)
        is_source = rays_in_view['termination_type'] == 1
        is_sphere = rays_in_view['termination_type'] == 2

        if np.any(is_source):
            source_hits = rays_in_view[is_source]

            y_s_intersect = source_hits['y_s']
            z_s_intersect = source_hits['z_s']
            t_s = source_hits['t_s']

            r_s = np.sqrt(y_s_intersect**2 + z_s_intersect**2)
            Omega = np.sqrt(M_scale / (r_s**3 + 1e-12))

            total_time = t_anim + t_s
            rotation_angle = Omega * total_time
            rotation_direction = -1.0 if prograde_disk else 1.0
            final_angle = rotation_direction * rotation_angle

            cos_angle = np.cos(final_angle)
            sin_angle = np.sin(final_angle)

            y_s_emit = y_s_intersect * cos_angle - z_s_intersect * sin_angle
            z_s_emit = y_s_intersect * sin_angle + z_s_intersect * cos_angle

            half_sw = source_image_width / 2.0
            norm_y = (y_s_emit + half_sw) / source_image_width
            norm_z = (z_s_emit + half_sw) / source_image_width

            px_s = norm_y * (source_pixel_width - 1)
            py_s = (1.0 - norm_z) * (source_pixel_height - 1)
            px_s_int = np.clip(px_s, 0, source_pixel_width - 1).astype(np.int32)
            py_s_int = np.clip(py_s, 0, source_pixel_height - 1).astype(np.int32)
            source_colors = source_texture[py_s_int, px_s_int]
            np.add.at(pixel_accumulator, (py[is_source], px[is_source]), source_colors)

        if np.any(is_sphere):
            sphere_hits = rays_in_view[is_sphere]
            norm_phi = (sphere_hits['final_phi'] + np.pi) / (2 * np.pi)
            norm_theta = sphere_hits['final_theta'] / np.pi
            px_sph = norm_phi * (sphere_pixel_width - 1)
            py_sph = norm_theta * (sphere_pixel_height - 1)
            px_sph_int = np.clip(px_sph, 0, sphere_pixel_width - 1).astype(np.int32)
            py_sph_int = np.clip(py_sph, 0, sphere_pixel_height - 1).astype(np.int32)
            sphere_colors = sphere_texture[py_sph_int, px_sph_int]
            np.add.at(pixel_accumulator, (py[is_sphere], px[is_sphere]), sphere_colors)

        np.add.at(count_accumulator, (py, px), 1)

    # --- Phase 3: Assembling Final Image ---
    hit_pixels_mask = count_accumulator > 0
    final_image_float = np.zeros_like(pixel_accumulator)
    final_image_float[hit_pixels_mask] = (
        pixel_accumulator[hit_pixels_mask] / count_accumulator[hit_pixels_mask, np.newaxis]
    )

    final_image_uint8 = (np.clip(final_image_float, 0, 1) * 255).astype(np.uint8)

    img = Image.fromarray(final_image_uint8, 'RGB')

    output_dir = os.path.dirname(output_filename)
    if output_dir:
        os.makedirs(output_dir, exist_ok=True)

    img.save(output_filename)
    print(f"--- Animated frame saved to '{output_filename}' ---")
```

24 The Master Animation Frame Generation Function

[ ]:
```
import os
import numpy as np
from IPython.display import display, Image as IPImage
from typing import Union

def generate_animation_frames(
    # --- Core Inputs ---
    blueprint_filename: str,
    output_folder: str,

    # --- Rendering & Physics Parameters ---
    source_texture: Union[str, np.ndarray],
    sphere_texture: Union[str, np.ndarray],
    source_physical_width: float,
    mass_of_black_hole: float,

    # --- Animation Control ---
    num_frames: int = 120,
    orbits_at_isco: float = 2.0,
    is_prograde: bool = True,

    # --- Image & Window Settings ---
    output_pixel_width: int = 400,
    window_width: float = 1.5
) -> None:
    """
    Generates a sequence of lensed image frames for creating an animation.

    This function takes a single ray-tracing blueprint and generates multiple
    frames by animating a differentially rotating Keplerian disk over time.

    Args:
        blueprint_filename: Path to the input 'blueprint.bin' file.
        output_folder: Directory where the output frames will be saved.
        source_texture: Path to the source image or a pre-loaded NumPy array.
        sphere_texture: Path to the background star map or a pre-loaded NumPy array.
        source_physical_width: The physical width (in units of M) of the source texture.
        mass_of_black_hole: The mass (M_scale) of the black hole.
        num_frames: The total number of frames to generate for the animation.
        orbits_at_isco: The total number of orbits the disk completes at the ISCO (r=6M)
                        over the full animation duration.
        is_prograde: Direction of disk rotation (True for prograde, False for retrograde).
        output_pixel_width: The width of the output images in pixels.
        window_width: The physical width of the camera's viewing window.
    """
    print(f"--- Starting Animation Generation ---")

    # --- Setup and Validation ---
    if not os.path.exists(blueprint_filename):
        raise FileNotFoundError(f"Blueprint file not found: {blueprint_filename}")

    os.makedirs(output_folder, exist_ok=True)

    # Calculate total animation time based on the desired number of orbits at the ISCO
    # ISCO for Schwarzschild is at r=6M.
    isco_radius = 6.0
    orbital_period_at_isco = 2 * np.pi * np.sqrt(isco_radius**3 / mass_of_black_hole)
    total_animation_time = orbital_period_at_isco * orbits_at_isco

    print(f"  Configuration:")
    print(f"    Total frames: {num_frames}")
    print(f"    Total animation time: {total_animation_time:.2f} M ({orbits_at_isco:.1f} orbits at r=6M)")
    print(f"    Output folder: '{output_folder}'")

    # --- Master Animation Loop ---
    for i in range(num_frames):
        # Calculate the animation time for the current frame
        t_animation = (i / (num_frames - 1)) * total_animation_time if num_frames > 1 else 0

        # Define a sequential filename for the frame
        frame_filename = os.path.join(output_folder, f"frame_{i:04d}.png")

        print(f"Rendering frame {i+1}/{num_frames} (t_anim = {t_animation:.2f} M)...")

        # Call the rendering function for a single frame
        generate_animated_lensed_image(
            output_filename=frame_filename,
            output_pixel_width=output_pixel_width,
            source_image_width=source_physical_width,
            sphere_image=sphere_texture,
            source_image=source_texture,
            M_scale=mass_of_black_hole,
            t_anim=t_animation,
            prograde_disk=is_prograde,
            blueprint_filename=blueprint_filename,
            window_width=window_width
        )

    print("\n--- All frames generated successfully. ---")

    # Optional: Display the first frame in the notebook for verification
    first_frame_path = os.path.join(output_folder, "frame_0000.png")
    if os.path.exists(first_frame_path):
        print("Displaying first generated frame:")
        display(IPImage(filename=first_frame_path))
```

25 The Video Encoding Function

[ ]:
```
import subprocess
import os

def encode_video_from_frames(
    image_folder: str,
    output_video_path: str,
    frame_rate: int = 30,
    crf: int = 18
) -> None:
    """
    Encodes a sequence of image frames into a video file using FFmpeg.

    This function requires FFmpeg to be installed and accessible in the system's PATH.

    Args:
        image_folder: The directory containing the sequentially named image frames.
        output_video_path: The full path for the output video file (e.g., 'output/animation.mp4').
        frame_rate: The frame rate for the output video.
        crf: The Constant Rate Factor for the x264 codec (lower is higher quality).
    """
    print(f"--- Starting Video Encoding ---")

    # Ensure the output directory exists
    output_dir = os.path.dirname(output_video_path)
    if output_dir:
        os.makedirs(output_dir, exist_ok=True)

    # Construct the FFmpeg command as a list of arguments
    command = [
        'ffmpeg',
        '-y',  # Overwrite output file if it exists
        '-framerate', str(frame_rate),
        '-i', os.path.join(image_folder, 'frame_%04d.png'),
        '-c:v', 'libx264',
        '-pix_fmt', 'yuv420p',
        '-r', str(frame_rate),
        '-crf', str(crf),
        output_video_path
    ]

    print(f"Running FFmpeg command:\n{' '.join(command)}")

    try:
        # Run the command
        # capture_output=True will store stdout and stderr in the result object
        # text=True will decode them as text
        result = subprocess.run(
            command,
            capture_output=True,
            text=True,
            check=True  # This will raise a CalledProcessError if FFmpeg returns a non-zero exit code
        )
        print("\n--- FFmpeg stdout ---")
        print(result.stdout)
        print("\n--- FFmpeg stderr ---")
        print(result.stderr)
        print(f"\n[] Video encoding successful. File saved to '{output_video_path}'")

    except FileNotFoundError:
        print("\n[!] ERROR: FFmpeg not found.")
        print("Please ensure FFmpeg is installed and accessible in your system's PATH.")
    except subprocess.CalledProcessError as e:
        print(f"\n[!] ERROR: FFmpeg failed with exit code {e.returncode}.")
        print("\n--- FFmpeg stdout ---")
        print(e.stdout)
        print("\n--- FFmpeg stderr ---")
        print(e.stderr)
```

26 Runner and Blueprint Manager

[ ]:
```
import subprocess
import os
import shutil

def run_integrator_and_rename_blueprint(
    project_dir: str,
    executable_name: str,
    args_string: str,
    output_blueprint_name: str,
    # NEW: Added parameter for the output directory for blueprints
    bin_folder: str
) -> None:
    """
    Runs the C geodesic integrator and moves the resulting blueprint file
    to a specified subfolder.
    """
    command_to_run = f"./{executable_name} {args_string}"

    print(f"--- Running command in directory '{project_dir}': {command_to_run} ---")

    try:
        process_result = subprocess.run(
            command_to_run,
            shell=True,
            capture_output=True,
            text=True,
            check=True,
            cwd=project_dir
        )
        print(process_result.stdout)
    except subprocess.CalledProcessError as e:
        print(f"ERROR: The C program exited with an error (exit code {e.returncode}).")
        print("--- Standard Error ---")
        print(e.stderr)
        return

    # The C code always outputs 'blueprint.bin' to its own directory.
    original_blueprint_path = os.path.join(project_dir, "blueprint.bin")

    # --- MODIFICATION: Move blueprint to the specified subfolder ---
    # First, ensure the destination folder exists.
    destination_folder = os.path.join(project_dir, bin_folder)
    os.makedirs(destination_folder, exist_ok=True)

    # Construct the final path for the renamed blueprint.
    new_blueprint_path = os.path.join(destination_folder, output_blueprint_name)

    if os.path.exists(original_blueprint_path):
        print(f"Moving '{original_blueprint_path}' to '{new_blueprint_path}'...")
        shutil.move(original_blueprint_path, new_blueprint_path)
        print("--- Run complete. ---")
    else:
        print("Warning: 'blueprint.bin' was not created by the C program.")

print("Helper function `run_integrator_and_rename_blueprint` defined.")
```

27 Animation Cell

[ ]:
```
import numpy as np
import os
import shutil
import subprocess
from IPython.display import display, Image as IPImage
from typing import Union

def generate_spinning_disk_animation_frames(
    # --- Required Positional Arguments ---
    num_frames: int,
    total_animation_duration: float,
    project_dir: str,
    executable_name: str,
    base_par_filename: str,
    blueprint_folder: str,
    frames_folder: str,
    output_pixel_width: int,
    source_image_width: float,
    sphere_image: Union[str, np.ndarray],
    source_image: Union[str, np.ndarray],
    window_width: float,

    # --- Optional Keyword Arguments ---
    start_time_offset: float = 0.0,
    save_blueprints: bool = False,
    **kwargs # For gamma, intensity_scale, etc.
) -> None:
    """
    Generates a sequence of frames for a spinning disk animation.

    This version includes a `save_blueprints` flag. If False (default), it
    deletes the large temporary blueprint file for each frame after rendering,
    saving significant disk space.
    """
    print(f"--- Generating {num_frames} Frames for Spinning Disk Animation ---")
    if not save_blueprints:
        print("    (Temporary blueprints will be deleted after each frame is rendered)")

    full_blueprint_dir = os.path.join(project_dir, blueprint_folder)
    full_frames_dir = os.path.join(project_dir, frames_folder)
    os.makedirs(full_blueprint_dir, exist_ok=True)
    os.makedirs(full_frames_dir, exist_ok=True)

    base_par_lines = []
    with open(os.path.join(project_dir, base_par_filename), 'r') as f:
        for line in f:
            if not line.strip().startswith("t_start"):
                base_par_lines.append(line)
    base_par_content = "".join(base_par_lines)

    # --- Main Animation Loop ---
    for i in range(num_frames):
        time_progress = (i / (num_frames - 1)) * total_animation_duration if num_frames > 1 else 0
        current_t_start = start_time_offset + time_progress

        print(f"\n--- Processing Frame {i+1}/{num_frames}: t_start = {current_t_start:.2f} M ---")

        # 1. Create temporary parameter file
        temp_par_content = base_par_content + f"\nt_start = {current_t_start:.4f}\n"
        temp_par_filename = "temp_anim_frame.par"
        full_temp_par_path = os.path.join(project_dir, temp_par_filename)
        with open(full_temp_par_path, "w") as f:
            f.write(temp_par_content)

        # 2. Run the C integrator
        command_to_run = f"./{executable_name} {temp_par_filename}"
        try:
            subprocess.run(
                command_to_run, shell=True, capture_output=True, text=True, check=True, cwd=project_dir
            )
        except subprocess.CalledProcessError as e:
            print(f"ERROR: The C program exited with an error for frame {i}.")
            print("--- Standard Error ---")
            print(e.stderr)
            continue

        # 3. Move the output blueprint to its temporary location
        original_blueprint_path = os.path.join(project_dir, "light_blueprint.bin")
        blueprint_frame_name = f"blueprint_frame_{i:04d}.bin"
        temp_blueprint_path = os.path.join(full_blueprint_dir, blueprint_frame_name)

        if os.path.exists(original_blueprint_path):
            shutil.move(original_blueprint_path, temp_blueprint_path)
        else:
            print(f"ERROR: C code ran but did not produce 'light_blueprint.bin'. Skipping rendering.")
            continue

        # 4. Render the image for this frame
        image_frame_name = os.path.join(full_frames_dir, f"frame_{i:04d}.png")

        generate_static_lensed_image(
            output_filename=image_frame_name,
            output_pixel_width=output_pixel_width,
            source_image_width=source_image_width,
            sphere_image=sphere_image,
            source_image=source_image,
            blueprint_filename=temp_blueprint_path,
            window_width=window_width,
            **kwargs
        )

        # --- 5. NEW: Clean up the temporary blueprint file ---
        if not save_blueprints:
            if os.path.exists(temp_blueprint_path):
                os.remove(temp_blueprint_path)
                print(f"  -> Deleted temporary blueprint: {blueprint_frame_name}")

    # --- Final Cleanup ---
    # If we weren't saving blueprints, the blueprint folder should be empty.
    # We can remove it to keep the project directory clean.
    if not save_blueprints:
        try:
            # Check if the directory is empty before removing
            if not os.listdir(full_blueprint_dir):
                os.rmdir(full_blueprint_dir)
        except OSError as e:
            print(f"Could not remove empty blueprint directory: {e}")

    print(f"\n--- All {num_frames} frames generated successfully in '{full_frames_dir}'. ---")

    first_frame_path = os.path.join(full_frames_dir, "frame_0000.png")
    if os.path.exists(first_frame_path):
        print("Displaying first generated frame:")
        display(IPImage(filename=first_frame_path))
```

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

[ ]:
```
import numpy as np
import os

def compare_blueprint_files(
    file1_path: str,
    file2_path: str,
    detailed_report: bool = False,
    tolerance: float = 1e-9
) -> bool:
    """
    Compares two light_blueprint.bin files to check for differences.

    This function performs a byte-for-byte comparison and, if they differ,
    a detailed field-by-field numerical comparison to identify exactly
    what has changed.

    Args:
        file1_path: Path to the first blueprint file.
        file2_path: Path to the second blueprint file.
        detailed_report: If True, prints the first few differing records.
        tolerance: The absolute tolerance for floating-point comparisons.

    Returns:
        True if the files are considered identical, False otherwise.
    """
    print(f"--- Comparing Blueprint Files ---")
    print(f"File 1: {file1_path}")
    print(f"File 2: {file2_path}")

    # --- 1. Basic File Checks ---
    if not os.path.exists(file1_path):
        print(f"ERROR: File not found: {file1_path}")
        return False
    if not os.path.exists(file2_path):
        print(f"ERROR: File not found: {file2_path}")
        return False

    size1 = os.path.getsize(file1_path)
    size2 = os.path.getsize(file2_path)

    if size1 != size2:
        print(f"\n[!] VERDICT: FILES ARE DIFFERENT (Sizes mismatch: {size1} vs {size2} bytes)")
        return False

    # --- 2. Fast Byte-for-Byte Comparison ---
    with open(file1_path, 'rb') as f1, open(file2_path, 'rb') as f2:
        if f1.read() == f2.read():
            print("\n[] VERDICT: FILES ARE IDENTICAL (Byte-for-byte comparison passed).")
            print("This confirms the C code is producing deterministic output.")
            return True

    print("\n[!] NOTICE: Files have the same size but their byte content differs. Performing detailed analysis...")

    # --- 3. Detailed Numerical Comparison (if byte comparison fails) ---
    # This dtype must match the one used for rendering.
    BLUEPRINT_DTYPE = np.dtype([
        ('termination_type', np.int32),
        ('y_w', 'f8'), ('z_w', 'f8'),
        ('stokes_I', 'f8'), ('lambda_observed', 'f8'),
        ('y_s', 'f8'), ('z_s', 'f8'),
        ('final_theta', 'f8'), ('final_phi', 'f8'),
        ('L_w', 'f8'), ('t_w', 'f8'),
        ('L_s', 'f8'), ('t_s', 'f8'),
    ], align=False)

    data1 = np.fromfile(file1_path, dtype=BLUEPRINT_DTYPE)
    data2 = np.fromfile(file2_path, dtype=BLUEPRINT_DTYPE)

    if len(data1) != len(data2):
        print(f"ERROR: Record count mismatch after loading: {len(data1)} vs {len(data2)}")
        return False

    # Compare integer fields exactly
    term_type_mismatch = np.any(data1['termination_type'] != data2['termination_type'])

    # Compare float fields with a tolerance
    float_fields = ['y_w', 'z_w', 'stokes_I', 'lambda_observed', 'y_s', 'z_s',
                    'final_theta', 'final_phi', 'L_w', 't_w', 'L_s', 't_s']

    mismatches = {}
    are_floats_different = False
    for field in float_fields:
        diff = np.abs(data1[field] - data2[field])
        if np.any(diff > tolerance):
            mismatches[field] = np.where(diff > tolerance)[0]
            are_floats_different = True

    if not term_type_mismatch and not are_floats_different:
        print("\n[] VERDICT: FILES ARE NUMERICALLY IDENTICAL (within tolerance).")
        print("The small byte differences are likely due to floating-point representation noise, which is acceptable.")
        return True

    print("\n[!] VERDICT: FILES ARE NUMERICALLY DIFFERENT.")
    print("This indicates the C code output is NOT deterministic and depends on t_start.")

    if term_type_mismatch:
        mismatch_indices = np.where(data1['termination_type'] != data2['termination_type'])[0]
        print(f"\nFound {len(mismatch_indices)} records with mismatched termination types.")
        if detailed_report:
            print("--- First 5 Mismatched Termination Records ---")
            for i in mismatch_indices[:5]:
                print(f"  Record {i}: Type 1 = {data1['termination_type'][i]}, Type 2 = {data2['termination_type'][i]}")
    if are_floats_different:
        print("\nFound differences in the following floating-point fields:")
        for field, indices in mismatches.items():
            print(f"  - Field '{field}': {len(indices)} mismatched records.")
            if detailed_report:
                print("    --- First 5 Mismatched Float Records ---")
                for i in indices[:5]:
                    print(f"      Record {i}: Val 1 = {data1[field][i]:.6e}, Val 2 = {data2[field][i]:.6e}, Diff = {np.
 ↪abs(data1[field][i] - data2[field][i]):.2e}")

    return False

# --- How to Use This Cell ---
# 1. Run your C code with t_start=0 and the camera pointed away from the disk.
# 2. Rename the output: mv project/photon_geodesic_integrator/light_blueprint.bin project/photon_geodesic_integrator/blueprint_t0.bin
# 3. Run your C code again with t_start=1000 (or any other value).
# 4. Rename the output: mv project/photon_geodesic_integrator/light_blueprint.bin project/photon_geodesic_integrator/blueprint_t1000.
 ↪bin
# 5. Run this cell.

# Define the paths to your two blueprint files
blueprint_t0_path = "project/photon_geodesic_integrator/light_blueprint.bin"
blueprint_t1000_path = "project/photon_geodesic_integrator/light_blueprint_0.bin"

# Run the comparison with a detailed report
are_identical = compare_blueprint_files(blueprint_t0_path, blueprint_t1000_path, detailed_report=True)

if are_identical:
    print("\nCONCLUSION: The C code is exonerated. The issue is not in the geodesic integration.")
else:
    print("\nCONCLUSION: The C code is implicated. The integration results depend on t_start, which should not happen for a stationary␣
 ↪metric.")
```

28 Start of visual function call cells

29 Blueprints stats

[ ]:
```
# --- Call the function with your blueprint file and desired bin width ---
blueprint_filename="project/photon_geodesic_integrator/light_blueprint.bin"
#plot_stacked_radial_histogram(blueprint_filename=blueprint_filename, bin_width=0.01)
```

```
analyze_blueprint()

# --- Run the viewer ---
view_binary_blueprint()
```

30 Setting all visualization parameters

[ ]:
```
import os
import numpy as np

print("--- Initializing Master Configuration for Visualization & Animation ---")

# --- Core File & Path Settings ---

# Get the user's home directory (e.g., /home/daltonm)
home_dir = os.path.expanduser('~')

# Path to the directory where the C projects live
# Assumes your notebook is in ~/Documents/
base_project_dir = os.path.join(home_dir, "Documents", "project")

# Path to the specific light integrator project
p_light_integrator_dir = os.path.join(base_project_dir, "photon_geodesic_integrator")

# --- NEW: Define the absolute output directory ---
# This will create /home/daltonm/Documents/Generated_nrpy_images/
p_output_basedir = os.path.join(home_dir, "Documents", "Generated_nrpy_images")



# --- Physics & Scene Parameters ---
p_mass_of_black_hole = 1.0
p_window_width = 1.5

# --- Source & Background Texture Settings ---
p_sphere_texture_file = "starmap_2020.png"

# --- Procedural Disk Generation Parameters ---
p_disk_inner_radius = 6.0
p_disk_outer_radius = 25.0
p_colormap = 'afmhot'
p_disk_temp_power_law = -1.5
p_ring_num = 4
```

```python
p_ring_contrast = 0.7
p_ring_log_spacing = True
p_doppler_factor = 0.3
p_doppler_power = 3
p_hotspot_num = 2
p_hotspot_amplitude = 0.4
p_hotspot_radius_center = 10.0
p_hotspot_radius_width = 4.0
p_shape_num_lobes = 0
p_shape_inner_amplitude = 0.0
p_shape_outer_amplitude = 0.0
p_source_physical_width = 2 * (p_disk_outer_radius + p_shape_outer_amplitude)

# --- General Image Quality & Rendering Settings ---
p_static_image_pixel_width = 400
p_animation_pixel_width = 400
p_intensity_scale = 20.0
p_gamma = 2.2
p_lambda_min_nm = 500  # Fixed min wavelength for color consistency
p_lambda_max_nm = 2250  # Fixed max wavelength


# Path to the specific light integrator project
p_light_integrator_dir = os.path.join(base_project_dir, "photon_geodesic_integrator")
p_blueprint_filename = os.path.join(p_light_integrator_dir, "light_blueprint.bin")

# --- Animation Settings ---
p_anim_name = "detailed_spiral" # A descriptive name for this animation run
# Subfolder for the individual frames
p_anim_frames_folder = os.path.join(p_output_basedir, f"{p_anim_name}_frames")
# Subfolder for the temporary blueprints (can be deleted after)
p_anim_blueprint_folder = os.path.join(p_light_integrator_dir, f"{p_anim_name}_blueprints")
# Final output filename for the video
p_anim_video_file = os.path.join(p_output_basedir, f"{p_anim_name}.mp4")


# --- CORRECTED PATH DEFINITIONS ---

# The folder for temporary blueprints should be in the shared project directory
blueprint_folder = os.path.join(base_project_dir, f"{p_anim_name}_blueprints")

# The folder for the final PNG frames is already correctly defined in the master config
frames_folder = p_anim_frames_folder
# --- END CORRECTION ---
```

```
p_anim_num_frames = 150
p_anim_orbits_at_isco = 2.0
p_anim_is_prograde = True
p_anim_start_time = 70.0 # The starting coordinate time for the animation

print(f"Master configuration loaded.")
print(f"All animation output will be saved in: {p_output_basedir}")
```

31 Disk Generation

[ ]:
```
source_disk_texture = generate_advanced_disk_array(
    pixel_width=1024, # High resolution for the source
    disk_physical_width=p_source_physical_width,
    colormap=p_colormap,
    disk_inner_radius=p_disk_inner_radius,
    disk_outer_radius=p_disk_outer_radius,
    disk_temp_power_law=p_disk_temp_power_law,
    ring_num=p_ring_num,
    ring_contrast=p_ring_contrast,
    ring_log_spacing=p_ring_log_spacing,
    doppler_factor=p_doppler_factor,
    doppler_power=p_doppler_power,
    hotspot_num=p_hotspot_num,
    shape_num_lobes=p_shape_num_lobes,
    shape_inner_amplitude=p_shape_inner_amplitude,
    shape_outer_amplitude=p_shape_outer_amplitude,
    display_image=False # Show the disk we're about to render
)
```

32 Standard Static Image Generation

[ ]:
```
from IPython.display import display, Image as IPImage
import os
import numpy as np

print("--- Generating a Standard Static Lensed Image from Radiative Transfer Data ---")

# --- 1. Define all inputs for the function call ---
# All parameters are now read from the master configuration cell (prefix 'p_').
```

```python
# Define a unique name for the static image file
static_image_name = "carestain_test_0.png"
p_static_image_pixel_width = 400
output_filename = os.path.join(p_output_basedir, static_image_name)

# Choose the source texture. This can be a filename or a pre-generated numpy array.
# For this example, we use the advanced disk generated in a previous cell.
try:
    source_image_texture = source_disk_texture
except NameError:
    print("Warning: `advanced_disk_data` not found. Generating a default disk for this static image.")
    source_image_texture = generate_advanced_disk_array(display_image=False)

# --- 2. The Function Call ---
# This call now uses all the parameters defined in your master configuration cell.
generate_static_lensed_image(
    output_filename=output_filename,
    output_pixel_width=p_static_image_pixel_width,
    source_image_width=p_source_physical_width,
    sphere_image=p_sphere_texture_file,
    source_image=source_image_texture,
    intensity_scale=p_intensity_scale,
    lambda_min_nm=p_lambda_min_nm,
    lambda_max_nm=p_lambda_max_nm,
    gamma=p_gamma,
    blueprint_filename=p_blueprint_filename,
    window_width=p_window_width
)

# --- 3. Display the result ---
if os.path.exists(output_filename):
    print(f"\nDisplaying static image: '{output_filename}'")
    display(IPImage(filename=output_filename))
else:
    print(f"\nERROR: Image file was not created at '{output_filename}'")
```

33 Final Video Encoding

[ ]:
```python
# --- FINAL ANIMATION EXECUTION CELL ---

# 1. Generate the source disk texture to be used for rendering
# This is done once before the main loop.
print("--- Generating Source Disk Texture ---")
```

```
source_disk_texture = generate_advanced_disk_array(
    pixel_width=1024, # High resolution for the source
    disk_physical_width=p_source_physical_width,
    colormap=p_colormap,
    disk_inner_radius=p_disk_inner_radius,
    disk_outer_radius=p_disk_outer_radius,
    disk_temp_power_law=p_disk_temp_power_law,
    ring_num=p_ring_num,
    ring_contrast=p_ring_contrast,
    ring_log_spacing=p_ring_log_spacing,
    doppler_factor=p_doppler_factor,
    doppler_power=p_doppler_power,
    hotspot_num=p_hotspot_num,
    shape_num_lobes=p_shape_num_lobes,
    shape_inner_amplitude=p_shape_inner_amplitude,
    shape_outer_amplitude=p_shape_outer_amplitude,
    display_image=False # Show the disk we're about to render
)

# 2. Calculate animation time window
isco_radius = 6.0 * p_mass_of_black_hole
orbital_period_at_isco = 2 * np.pi * np.sqrt(isco_radius**3 / p_mass_of_black_hole)
animation_duration = orbital_period_at_isco * p_anim_orbits_at_isco
animation_end_time = p_anim_start_time + animation_duration

print(f"\nAnimation will run from t={p_anim_start_time:.2f} M to t={animation_end_time:.2f} M (Duration: {animation_duration:.2f} M).")

# --- Important: Do you want to save the light blueprints?
save_blueprints= False

# 3. Generate all the PNG frames for the animation
generate_spinning_disk_animation_frames(
    num_frames=p_anim_num_frames,
    total_animation_duration=animation_duration,
    start_time_offset=p_anim_start_time,
    project_dir=p_light_integrator_dir, # The C code still runs from its own directory
    executable_name="photon_geodesic_integrator",
    base_par_filename="photon_geodesic_integrator.par",
    blueprint_folder=blueprint_folder, # Pass the corrected, absolute path
    frames_folder=frames_folder,
    output_pixel_width=p_animation_pixel_width,
    source_image_width=p_source_physical_width,
    sphere_image=p_sphere_texture_file,
    source_image=source_disk_texture,
    window_width=p_window_width,
    gamma=p_gamma,
    lambda_min_nm=p_lambda_min_nm,
    lambda_max_nm=p_lambda_max_nm,
    intensity_scale=p_intensity_scale,
    save_blueprints= save_blueprints
)
# 4. Encode the generated frames into an MP4 video
encode_video_from_frames(
    image_folder=p_anim_frames_folder,
    output_video_path=p_anim_video_file,
    frame_rate=15
)
```

```
[ ]: encode_video_from_frames(
         image_folder=p_anim_frames_folder,
         output_video_path=p_anim_video_file,
         frame_rate=8
     )
```