**Material Body Simulator Documentation**

This documentation provides an overview and explanation of the material\_body\_simulator.py code, enabling you to understand and regenerate the code for your own use.

**Overview**

The Material Body Simulator is a Python application that simulates hierarchical material bodies composed of multiple layers and child bodies. It uses **PyQt6** for the GUI framework and **Matplotlib** for rendering the visual representations of the material bodies. The simulation includes features like dynamic updates, rotations, zooming, panning, and density adjustments based on interactions between parent and child bodies.

**Table of Contents**

1. [Key Components](#key-components)
2. [Data Structures and Utility Functions](#data-structures-and-utility-functions)
3. [Visualization Classes](#visualization-classes)
4. [Main Application Class](#main-application-class)
5. [Main Execution](#main-execution)
6. [Usage Instructions](#usage-instructions)
7. [Recreating the Code](#recreating-the-code)
8. [Notes on Missing Modules](#notes-on-missing-modules)

**Key Components**

1. **Data Structures and Utility Functions**: Core algorithms and helper functions for the simulation logic.
2. **Visualization Classes**: Classes responsible for rendering the material bodies within a PyQt6 application.
3. **Main Application Class**: The PyQt6 application window that orchestrates the simulation.
4. **Main Execution**: The entry point of the application where the simulation is initialized and run.

**Data Structures and Utility Functions**

**1. RotationResult Dataclass**

python

Copy code

@dataclass

class RotationResult:

angles: List[float]

min\_diff: float

* **Purpose**: Holds the result of finding the optimal rotation for a body, including the angles that result in the minimum density difference and the minimum difference value.

**2. find\_optimal\_rotation Function**

python

Copy code

def find\_optimal\_rotation(

test\_angles: List[float],

calculate\_density\_diff\_fn,

tolerance: float = 1e-6

) -> RotationResult:

* **Purpose**: Finds the rotation angle that minimizes the density difference between a child body and its parent body.
* **Parameters**:
  + test\_angles: List of angles to test.
  + calculate\_density\_diff\_fn: Function that calculates the density difference for a given angle.
  + tolerance: Acceptable tolerance for comparing density differences.

**3. initialize\_densities Function**

python

Copy code

def initialize\_densities(

body: Dict[str, Any],

num\_patches: int,

compulsory\_increase: float,

parent\_body: Dict[str, Any] = None,

parent\_radius: float = 1.0,

parent\_center: Tuple[float, float] = (0.0, 0.0)

) -> None:

* **Purpose**: Initializes the density profiles for each layer and micro-layer in a body.
* **Parameters**:
  + body: The material body to initialize.
  + num\_patches: Number of angular patches per layer.
  + compulsory\_increase: Minimum required increase in density between layers.
  + parent\_body, parent\_radius, parent\_center: Used for initializing child bodies recursively.

**4. calculate\_density\_mismatch Function**

python

Copy code

def calculate\_density\_mismatch(

body: Dict[str, Any],

external\_density\_info: Tuple[np.ndarray, int, int]

) -> Tuple[float, int, int]:

* **Purpose**: Computes the density mismatch between a body and its external environment (parent body).
* **Returns**: A tuple containing the density difference and indices.

**5. calculate\_all\_patch\_centers Function**

python

Copy code

def calculate\_all\_patch\_centers(

body: Dict[str, Any],

body\_center: Tuple[float, float],

outer\_radius: float,

layer\_thicknesses: List[float],

num\_patches: int,

layer\_indices: List[int] = [0],

num\_micro\_layers: int = 1,

rotation\_angle: float = 0.0

) -> Dict[int, Dict[str, Any]]:

* **Purpose**: Calculates the center positions (x, y) of all patches in specified layers and micro-layers of a body.
* **Parameters**:
  + body\_center: Center coordinates of the body.
  + outer\_radius: Outer radius of the body.
  + layer\_thicknesses: List of layer thicknesses.
  + rotation\_angle: Rotation angle of the body.

**6. find\_nearest\_patches\_vectorized Function**

python

Copy code

def find\_nearest\_patches\_vectorized(

child\_patch\_mappings: Dict[int, List[int]],

parent\_patch\_positions: Dict[str, Any]

) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:

* **Purpose**: Efficiently finds the nearest parent patches and their densities for all child patches.

**7. patch\_mappings Function**

python

Copy code

def patch\_mappings(

child\_patch\_positions: Dict[str, Any],

parent\_patch\_positions: Dict[str, Any],

parent\_center: Tuple[float, float]

) -> Dict[int, List[int]]:

* **Purpose**: Maps child patches to the nearest parent patches using direct distance between patch centers.

**8. calculate\_parent\_patch\_positions Function**

python

Copy code

def calculate\_parent\_patch\_positions(

parent\_body: Dict[str, Any],

parent\_layer\_index: int,

parent\_center: Tuple[float, float],

parent\_radius: float,

rotation\_angle: float = 0.0

) -> Dict[str, Any]:

* **Purpose**: Calculates parent patch positions for a given layer.

**9. update\_child\_body\_density Function**

python

Copy code

def update\_child\_body\_density(

parent\_body: Dict[str, Any],

child\_body: Dict[str, Any],

parent\_center: Tuple[float, float] = (0.0, 0.0),

parent\_radius: float = 1.0

) -> None:

* **Purpose**: Recursively updates the density of a child body and all its descendants based on their parents.

**10. calculate\_child\_patch\_positions Function**

python

Copy code

def calculate\_child\_patch\_positions(

child\_body: Dict[str, Any],

parent\_radius: float,

parent\_center: Tuple[float, float],

rotation\_angle: float = 0.0

) -> Dict[str, Any]:

* **Purpose**: Calculates patch positions for child body's outer layer with respect to parent center.

**11. find\_parent\_body Function**

python

Copy code

def find\_parent\_body(

current\_body: Dict[str, Any],

target\_body: Dict[str, Any],

parent: Dict[str, Any] = None

) -> Dict[str, Any]:

* **Purpose**: Finds the parent body of a given target body in the material object hierarchy.

**12. check\_and\_shed\_higher\_order\_bodies Function**

python

Copy code

def check\_and\_shed\_higher\_order\_bodies(body, external\_density):

* **Purpose**: Checks if a body should shed higher-order bodies (layers or child bodies) based on the external density.

**13. update\_body\_properties\_after\_shedding Function**

python

Copy code

def update\_body\_properties\_after\_shedding(body):

* **Purpose**: Updates body properties after shedding, such as reducing the thickness of the outer layer or removing it entirely.

**14. calculate\_external\_density Function**

python

Copy code

def calculate\_external\_density(body):

* **Purpose**: Calculates the external density experienced by a body based on its parent's properties.

**Visualization Classes**

**MaterialBodyCanvas Class**

* **Inheritance**: Inherits from FigureCanvas of Matplotlib.
* **Purpose**: Renders the material bodies within a PyQt6 widget and handles user interactions like zooming and panning.
* **Key Methods**:
  + \_\_init\_\_: Initializes the canvas, loads color maps, and sets up event handlers.
  + plot\_material\_body: Plots the material bodies recursively.
  + plot\_body: Recursively plots each body and its layers, considering rotation and hierarchical positioning.
  + update\_labels: Updates labels on the plot based on zoom level and visibility settings.
  + Event handlers like on\_scroll, start\_pan, end\_pan, on\_motion handle zooming and panning.

**Example Initialization**:

python

Copy code

class MaterialBodyCanvas(FigureCanvas):

def \_\_init\_\_(self, material\_object, parent=None, width=5, height=4, dpi=100):

# Initialization code...

**Main Application Class**

**MaterialBodySimulator Class**

* **Inheritance**: Inherits from QMainWindow of PyQt6.
* **Purpose**: The main application window that orchestrates the simulation.
* **Key Methods**:
  + \_\_init\_\_: Sets up the UI and initializes the simulation.
  + init\_ui: Sets up the central widget and layout.
  + create\_toolbar: Adds toolbar actions and widgets.
  + Simulation control methods: toggle\_animation, animate\_placement, step\_animation, step\_back, restart\_simulation.
  + Dialog methods: open\_properties\_dialog, open\_edit\_bodies\_dialog, open\_visualization\_settings.
  + File operations: save\_project, load\_project.

**Example Initialization**:

python

Copy code

class MaterialBodySimulator(QMainWindow):

def \_\_init\_\_(self, material\_object):

super().\_\_init\_\_()

self.material\_object = material\_object

# More initialization...

**Main Execution**

At the end of the script, the main execution block initializes the simulation:

python

Copy code

if \_\_name\_\_ == '\_\_main\_\_':

# Define your material object with bodies A, B, C

material\_object = {

"name": "A",

# Body properties...

"child\_bodies": [

{

"name": "B",

# Child body properties...

"child\_bodies": [

{

"name": "C",

# Further nested child bodies...

}

]

}

]

}

app = QApplication(sys.argv)

simulator = MaterialBodySimulator(material\_object)

simulator.show()

sys.exit(app.exec())

**Usage Instructions**

1. **Dependencies**: Ensure you have the following packages installed:
   * Python 3.7 or later
   * PyQt6
   * Matplotlib
   * NumPy
2. **Running the Application**:
   * Save the code in a file named material\_body\_simulator.py.
   * Run the script using the command: python material\_body\_simulator.py.
3. **Interacting with the Application**:
   * Use the toolbar to control the simulation (Play, Pause, Step Forward, Step Back).
   * Use the body selector to choose which body to manipulate.
   * Use the zoom selector to focus on specific bodies.
   * Right-click and drag to pan the view.
   * Scroll the mouse wheel to zoom in and out.
4. **Editing Bodies and Layers**:
   * The application includes dialogs (not fully implemented in the provided code) for editing body properties and visualization settings.
5. **Saving and Loading Projects**:
   * Use the "Save Project" and "Load Project" options in the toolbar to save and load simulation states.

**Recreating the Code**

To regenerate the code, follow these steps:

1. **Set Up the Environment**:
   * Install the required packages using pip install PyQt6 matplotlib numpy.
2. **Create the Main Script**:
   * Start with the import statements.
   * Define the data structures and utility functions as outlined above.
   * Implement the MaterialBodyCanvas and MaterialBodySimulator classes.
   * Define the material\_object with the desired hierarchy of bodies and layers.
   * Add the main execution block to initialize and run the application.
3. **Implement Missing Modules**:
   * The code references dialogs and print\_util modules, which are not provided.
   * You'll need to implement these modules or adjust the code to work without them.
4. **Test the Application**:
   * Run the script and ensure the application window appears.
   * Interact with the simulation to verify functionality.

**Sample Code Structure**:

python

Copy code

# material\_body\_simulator.py

import sys

import pickle

from dataclasses import dataclass

from typing import List, Tuple, Dict, Any

import numpy as np

from PyQt6.QtWidgets import (

QApplication, QMainWindow, QVBoxLayout, QWidget, QToolBar,

QComboBox, QAction, QLabel, QCheckBox, QFileDialog, QSizePolicy

)

from PyQt6.QtGui import QColor

from PyQt6.QtCore import QTimer, Qt

from matplotlib.backends.backend\_qt5agg import FigureCanvasQTAgg as FigureCanvas

import matplotlib.pyplot as plt

from matplotlib.patches import Circle, Wedge

from matplotlib.collections import PatchCollection

# Implement data structures and utility functions...

# Implement the MaterialBodyCanvas class...

# Implement the MaterialBodySimulator class...

if \_\_name\_\_ == '\_\_main\_\_':

# Define the material object with bodies and layers

material\_object = {

"name": "A",

# Body properties...

}

app = QApplication(sys.argv)

simulator = MaterialBodySimulator(material\_object)

simulator.show()

sys.exit(app.exec())

**Notes on Missing Modules**

* **dialogs Module**: The code uses from dialogs import \*, indicating it relies on custom dialog classes for editing properties and settings.
  + **Solution**: You need to implement these dialog classes or remove dependencies by commenting out related code.
* **print\_util Module**: The code uses from print\_util import \*, suggesting it uses custom print utilities for debugging.
  + **Solution**: Implement the required print functions or adjust the code to use standard print statements.
* **Adjustments**: If you cannot provide these modules, you may need to comment out or modify sections of the code that depend on them to ensure the application runs without errors.

**Explanation of Adjustments:**

1. **Incorrect Zoom Calculation in zoom\_to\_body:**
   * **Issue:** The zoom\_to\_body function wasn't correctly calculating the zoom level or center point for child bodies.
   * **Solution:**
     + Modified the zoom\_to\_body function to recursively calculate the bounding box (bounds) of the target body, including all its child bodies.
     + Implemented the calculate\_body\_bounds function to compute the minimum and maximum x and y coordinates considering the body's layers, micro-layers, and child bodies.
     + This ensures that when zooming to a body, the view encompasses the entire structure of the body, including its nested children.
2. **Visibility Checks in is\_body\_visible:**
   * **Issue:** The is\_body\_visible function might have been too restrictive, causing some bodies not to be drawn even when they should be visible.
   * **Solution:**
     + Reviewed and adjusted the is\_body\_visible function to ensure it accurately determines if any part of a body is within the current view limits.
     + Added an is\_point\_visible function to check if specific points (like body centers) are within the view.
     + This ensures that bodies and their labels are drawn whenever they are visible, even partially.
3. **Plotting Logic in plot\_material\_body\_efficient and \_plot\_body\_details:**
   * **Issue:** Errors might have been preventing child bodies from being drawn correctly at different zoom levels.
   * **Solution:**
     + Adjusted the calculation of placement\_angle in plot\_material\_body\_efficient to include both the parent's rotation and the body's own rotation.
     + Ensured that the parent\_rotation is correctly passed and accumulated when plotting child bodies.
     + In \_plot\_body\_details, added checks to only plot labels if the center point is visible using is\_point\_visible.
     + Ensured that the body's rotation is correctly applied when calculating positions, so child bodies are plotted in the correct locations relative to their parents.
4. **Coordinate Systems Consistency:**
   * **Issue:** Inconsistent coordinate systems could cause incorrect positioning of bodies and their layers.
   * **Solution:**
     + Ensured that all position calculations for bodies and layers are relative to the correct parent coordinates.
     + Consistently used the center and radius parameters, and correctly applied rotations when calculating positions.
     + Adjusted the on\_scroll method to correctly calculate relx and rely for zooming based on the cursor's position within the view limits.
5. **Other Adjustments:**
   * **Label Updates:**
     + Modified update\_labels to ensure labels are only drawn for visible bodies.
     + Used is\_body\_partially\_visible to check if a body is within the current view before attempting to draw labels.
   * **Event Handling:**
     + Ensured that calls to self.draw\_idle() are made after redrawing plots to update the display.
     + Adjusted zoom and pan interactions to correctly update limits and redraw efficiently.