Google Summer of Code

GSoC'25 Proposal for TARDIS RT COLLABORATION

# Adding HDF Writing Capabilities to TARDIS Modules

---

## Contents



*Note : I have used LLM at a few places in the document to correct my grammar and ensure that my thoughts are accurately conveyed.*

## Project Details:

- Organization : TARDIS RT Collaboration
- Project Title : Adding HDF Writing Capabilities to TARDIS Modules
- Project Length : 350 hours
- Mentors : Andrew Fullard, Atharva Arya, Abhinav Ohri
- Difficulty : HARD

## Personal Information:


- Name : Karthik Rishinarada
- Email : karthikrk11135@gmail.com
- LinkedIn: Karthik Rishinarada
- Github ID : karthik11135

- Brief Summary : I'm a senior-year college student from NIT Trichy, India. I interned at Capital One, where I worked with a Python team. Over the past few months, I have developed an interest in open source and started exploring open-source codebases. I enjoy listening to music, window shopping, and planning vacations.
- Resume : my resume

## About the Organisation:

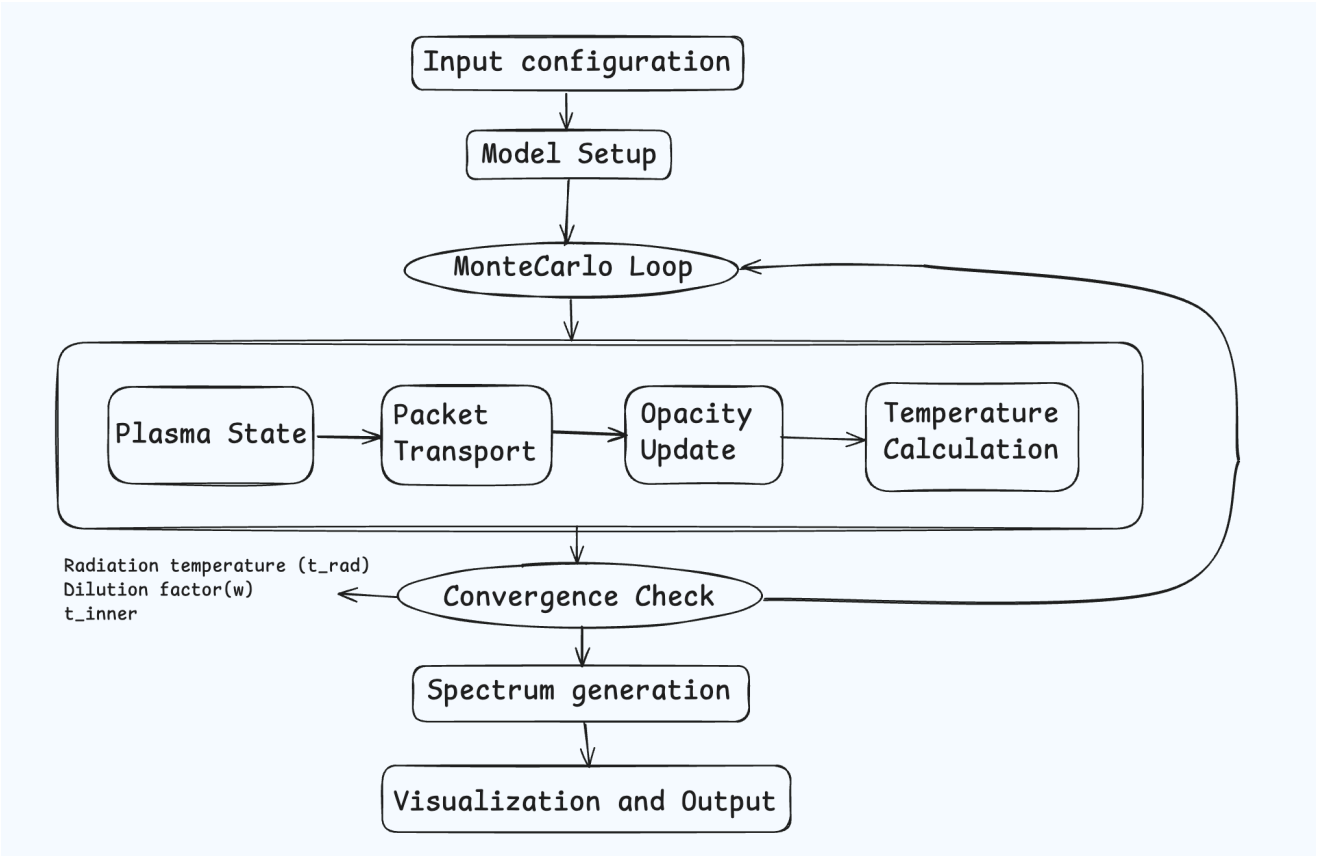
TARDIS (Temperature And Radiative Diffusion In Supernovae) is an open-source radiative transfer code that is developed for simulating supernova spectra. TARDIS helps astrophysicists/researchers understand how exploding stars appear to observers by creating spectra - essentially predicting what these star explosions would look like through a telescope. Additionally TARDIS is quick in evaluations which makes it uniquely powerful for comparing to observations.

Core Components of TARDIS

| Components | Description |
| --- | --- |
| **Configuration & Input** | • Handles model parameters, atomic data, abundance configurations and monte carlo information like number of threads, iterations, convergence strategy etc (YAML or Dict). |

| | |
|---|---|
| **Physical Model Setup (Simulation)** | • Establishes the radial grid structure (ex: Radial 1D Geometry) and initial conditions.<br>• Contains composition (nuclide masses, isotope abundances), packet source (produces packet objects with energies, directions etc), electron densities and time explosions. |
| **Transport state, Plasma and Opacities** | • Implements core Monte Carlo setup for packet propagation through shells.<br>• Handles packet interactions including scattering, absorption, and re-emission.<br>• BasePlasma stores essential plasma properties such as DilutePlanckianRadField, TimeExplosion, JBlues, AtomicData, DilutionFactor, etc.<br>• OpacityState manages line and continuum opacities for each shell, including electron densities and t_electrons.<br>• These data are crucial for the iterations. |
| **Monte Carlo Iteration Handler** | • Controls the main iteration loop for convergence.<br>• The data (dilution factor, radiation temperature and t_inner) is updated in the simulation state and plasma for every iteration.<br>• Monte Carlo and opacities are solved using the updated values.<br>• When the convergence happens, the spectrum solver is initialized and the iteration is run for one last time to get virtual packet energies and the transport state. |

Flow Diagram




## Project Summary:

The current implementation of the HDFWriterMixin class is limited. In terms of storing the HDF file, it can only store data of the following :

1. Scalars are stored in the Series under the path `/scalars`.
2. Basic unit conversions to CGS exist.
3. 1D arrays are stored as `pd.Series`.
4. 2D arrays are stored as dataframes.

The current implementation is used for storing simulation state, regression testing, visualization and it also supports plasma properties and transport state. We want to expand the writing capabilities by automatically storing the data according to a trigger. And then retrieve the data from an HDF file to form an instance.

Reference : HDFWriterMixin

The goal of this project is to enhance the capabilities of the HDF writing. It aims to build simulation state preservation, restoration and incorporate checkpoint integration with regression testing.

Benefits from the project :

1. Simulation state snapshot is stored accurately for future references. In this way, they can be shared with others.
2. Two simulation states can be compared.
3. State reconstruction from an existing HDF file enables users to use the instance directly.
4. Automatic checkpoint integration enables the creation of HDF files whenever the condition occurs.
5. Debugging capabilities are improved.
6. The regression testing techniques are most robust with the checkpoints.

The following table outlines the various types of data that can be stored in HDF files so that they can be retrieved back for analysis.

| **TARDIS Module** | **State Storage** | **Use cases** |
|---|---|---|
| **Model Component** | Geometric configurations, Abundance distributions, Density profiles, Temperature information | Enables simulation state restoration; Allows comparison of different model configurations. |
| **Plasma Module** | Ion and Level populations, Plasma properties (DilutePlanckianRadiationField, HeliumTreatment, JBlues, etc.) | Preserves complex plasma states for analysis; Allows comparison across iterations. |
| **Transport Component** | Interaction of photons, Energies, Opacity states, etc. | Monte Carlo iterations can be restarted; Helps analyze packet propagation patterns. |
| **Spectrum Component** | Line formation regions, spectra at different iterations | Helps analyze line formation processes. |

## First Objective Task:

Task : Add a method to the spectrum class that allows restoring the class from an HDF. Share the notebook in a pull request.

Link to solution : GSOC PR

Explanation : from_hdf method is created on the TARDISSpectrum class to retrieve all the data from the HDF file and store it as an object. The two main options that are required to initialize the spectrum class are the frequencies and luminosities. In the HDF file they are stored in the paths `'/tardis_spectrum/_frequency'` and `'/tardis_spectrum/luminosity'` respectively. The data residing in these paths is retrieved and the class is created.

## My Approach towards the project:

A flexible schema can be created to store all the data in HDF files. A base serialization class can be implemented to serialize complex data structures. Module-specific serialization can be implemented for all modules (e.g., Plasma, Transport, etc.). The schema is later used to retrieve the data and make the simulation.

For checkpoints, a checkpoint manager can be implemented, utilizing various trigger methods (e.g., iteration based, time based, storage based, convergence based, etc.). Whenever a trigger condition occurs, a checkpoint is created. Each checkpoint is stored as an HDF5 file containing the entire simulation state. Iteration specific data can be retrieved from these files. Using restoration logic necessary classes are created from the files. Finally, unit tests can be written, and proper documentation of code changes are to be updated.

Example of how a checkpoint file structure could be :

```
checkpoint.h5
├── metadata/
│   ├── timestamp
│   └── iteration_number
├── state/
│   ├── simulation_state
│   ├── plasma_state
│   ├── transport_state
│   └── atomic_data_references
└── convergence_data/
    ├── t_rad
```

```
    ├── w
    └── electron_densities
```

All the checkpoints can be stored in a folder.

```
/checkpoints/
  simulation_checkpoint_iter_100.h5
  simulation_checkpoint_iter_200.h5
  Simulation_checkpoint_iter_300.h5
```

Here is a code example for a trigger:

```
class ConvergenceTrigger:
    def __init__(self):
        self.last_state = None
    def checkpoint_trigger(self, t_rad, w, el_densities):
        current_state = {
            't_rad': t_rad,
            'w': w,
            'electron_densities': electron_densities
        }
        if self.convergence_changed(current_state):
            return True
        return False

# In this example, the checkpoint trigger returns True when convergence occurs storing the Transport State, Plasma State, and other relevant data.
```

To restore a simulation instance from an HDF5 file, module specific logic can be implemented. The restoration process involves:

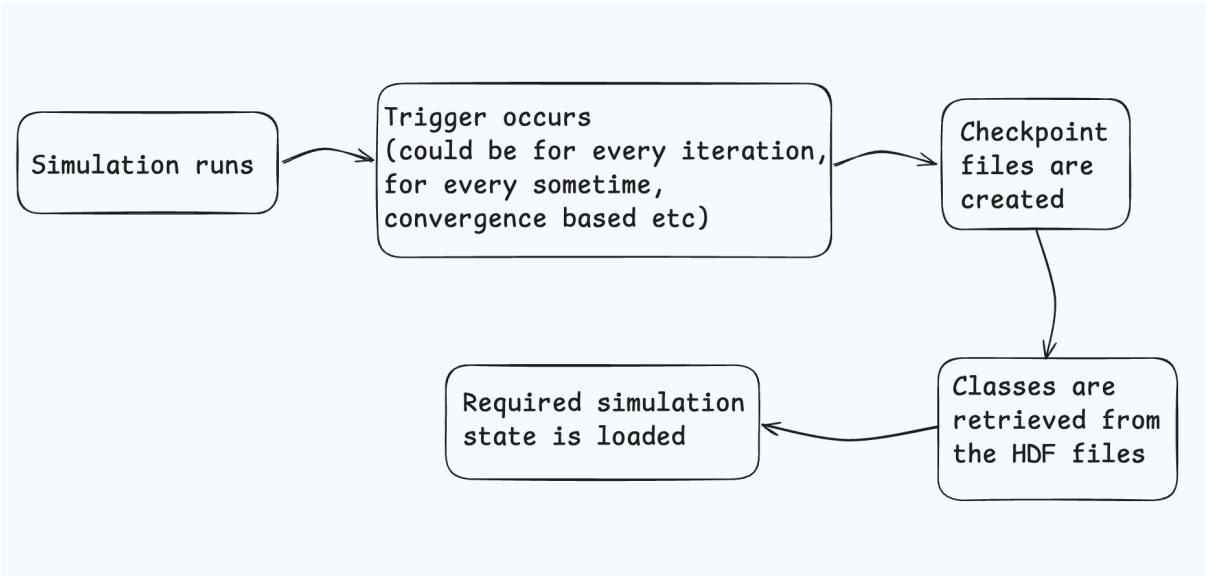
1. Loading the checkpoint (HDF5) file.
2. Retrieving the component states.
3. Reconstructing objects using the stored data.
4. Resuming the simulation from the saved state.

```
@classmethod
def from_checkpoint(cls,hdf_file):
    with h5py.File(hdf_file, r) as f:
        transport_state = f['transport_state']
        return cls.restore_class(transport_state)

@classmethod
def restore_class(cls, state):
    # Retrieve all the transport data from the file
    opacity_state = state['opacity_data']
    packet_data = state['packet_data']
```

```
    return cls(opacity_state, packet_data)
```

Flow Diagram




## Milestones:

Here's a detailed breakdown of milestones, each milestone's description, and deliverables:

| Week | Focus Area | Deliverables |
|---|---|---|
| Week 1 | • Discuss the project in depth with my mentor.<br>• Communicate my ideas and gather feedback.<br>• Begin implementation of the restoration logic. | NA |
| Week 2 | • Continue working on the restoration logic<br>• Code example scripts to demonstrate how to use the restoration functions. | • Restoration logic for BasePlasma and SimulationState<br>• Example scripts to run the functions. |

| | | |
|---|---|---|
| Week 3 | • Finish the restoration logic for SimulationState, Plasma, MacroAtomState and Transport state<br>• Ask for a review of progress and make improvements based on feedback | • Restoration logic for SimulationState, Plasma, MacroAtomState and the transport state from HDF5 files. |
| Week 4 | • Write necessary test cases for the restoration logic<br>• Work on midterm review/presentation | • Test suite for restoration logic |
| Week 5 | • Start working on the convergence trigger logic (storing HDF5 data when convergence occurs)<br>• Implement base class and module-specific logic for SimulationState | • Base trigger logic for SimulationState |
| Week 6 | • Complete the convergence trigger logic<br>• Implement module-specific logic for Plasma, MacroAtomState and transport state. | • Automatic storing of HDF5 files when convergence occurs |
| Week 7 | • Show progress to mentor and get feedback<br>• Write unit tests for the convergence trigger logic.<br>• Use the newly created HDF5 files to test restoration logic | • Test suite for trigger logic |
| Week 8 | • Begin extending the trigger logic to work based on time and iteration count. | NA |

| | | |
|---|---|---|
| | • Get guidance from my mentor and adjust accordingly. | |
| Week 9 | • Finish implementing the different trigger types (convergence, iteration, time) | • Trigger logic that stores data based on multiple trigger types |
| Week 10 | • Write unit tests for all trigger types.<br>• Document all code changes and update user documentation | • Full test suite and updated documentation |
| Week 11 | • Work on final review presentation<br>• Create example scripts for the new features to help users understand usage. | NA |
| Week 12 | • Final review and code polishing<br>• Get mentor feedback and apply final changes | • Final, reviewed codebase ready for submission |

### Why did I choose TARDIS?

Over the past few months, I wanted to explore open-source contributions, so I started looking for Python repositories on GitHub. That's when I came across the TARDIS. I've gone through the entire codebase and gained an understanding of its main components.

While exploring the code, I encountered numerous scientific terms like Monte Carlo Iteration, Plasma State, JBlues, Radiation Field, etc. I really enjoyed the process of researching these terms and understanding them. I often try

to familiarize myself with facts about space, so this aligned well with my interests. One more thing I truly admire is how clearly the documentation is written. The clarity of explanations and the well written code made it much easier for me to understand what's going on.

I'm also extremely fascinated by the fact that I would be working with astrophysicists and researchers. The exposure and learning I would get from this opportunity would be immense, as I cannot think of any other way where I would get the chance to work with scientists directly. Moreover, the idea that my contributions could play even a small role in advancing scientific discovery excites me to fully engage and give my best. These are the reasons why I'm eager to contribute to TARDIS.

## Why am I the best fit?

My first internship offer was from a fintech company — Capital One, where I worked as a Software Developer Intern for two months. During this time, I improved my Python programming skills, which is the primary language used in TARDIS. I improved the runtime of the codebase by 40%. I also became comfortable with managing environments in python, Git workflows, and writing clean code that aligns with company's standards. I learned how to work in a team. During the last three weeks of my internship, I worked in a high-pressure environment putting 14 hours a day. That experience strengthened my discipline and commitment to delivering quality work within tight deadlines. My manager was super happy with my work.

After my internship, I discovered TARDIS and began exploring its code by studying the documentation and working through the quickstart tutorials. I've gone through a few issues in the codebase and tried to solve a few of them.

Main TARDIS repo:

- add test to line info
- from Simulation test for RPacket Plot
- TODO: Tests of Composition
- TODO: Rename tau to tau_event
- Tests for the config validator
- refactor: Remove ConfigWriterMixin class

Regression Data repo:

- Regression data for Composition

Carsus repo:

- fixes NISTWeightsComp initialization error

I strongly believe that I am a good fit for this project because I have a deep understanding of the codebase. I've always responded quickly whenever I'm asked to do some changes in my open PRs.  I am confident that I can hit the ground running from day zero. Apart from that I'm flexible to work according to any timezone.

I view GSOC as a great opportunity to learn from experienced mentors and contribute to a real world project. However my motivation extends beyond GSOC itself. I plan to continue contributing to TARDIS and staying involved even if my application isn't selected, because my primary focus is on learning and making good contributions.

## Conclusion:

In conclusion,  my strong knowledge in Python, experience with open-source contributions, and familiarity with the TARDIS codebase makes me a great fit for this project. While I have my final exams from May 6 to May 13, I will be entirely free afterward and fully dedicated to contributing to this project. As of now I do not have any external obligations during the coding period. I look forward to contributing my best to the TARDIS RT Collaboration

Thank you

---