GSoC'25 Proposal for TARDIS RT COLLABORATION

CARSUS DASHBOARD

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 : CARSUS DASHBOARD
- Project Length : 350 hours
- Mentors : Josh Shields
- 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 National Institute of Technology 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:

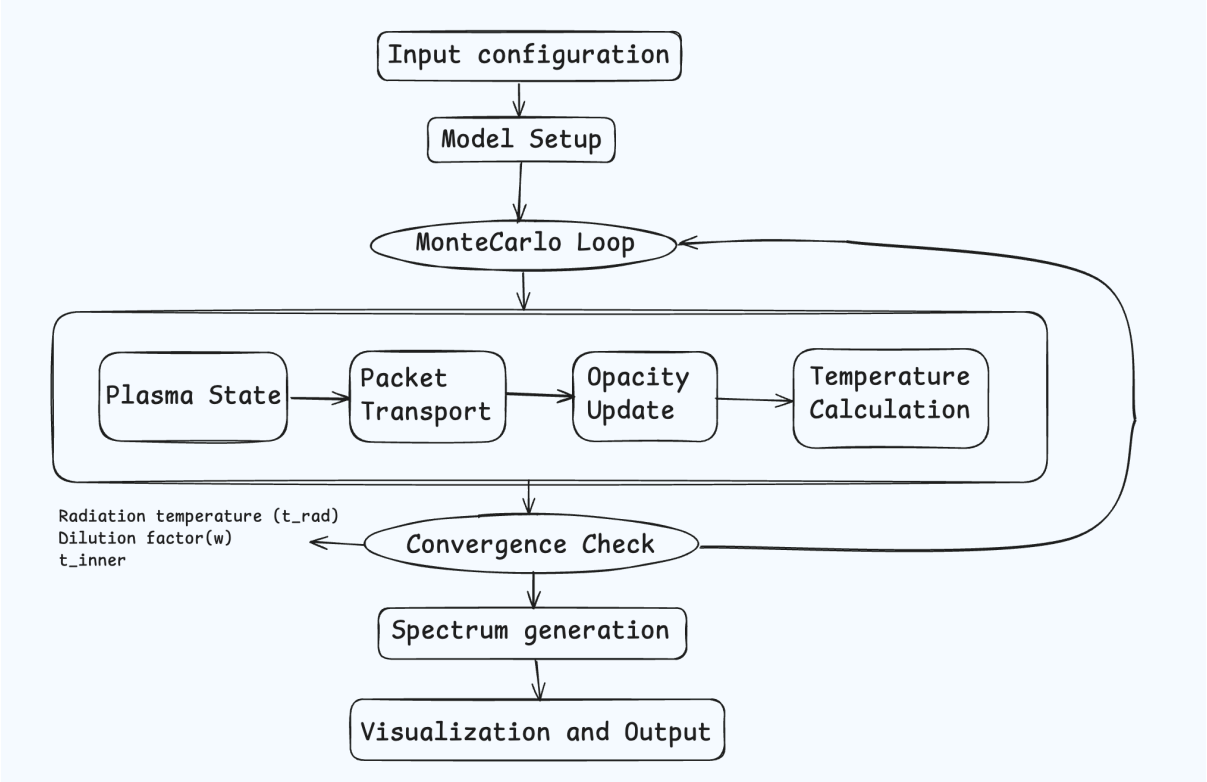
TARDIS (Temperature And Radiative Diffusion In Supernovae) is an open-source radiative transfer code that is developed for simulating supernova spectra. TARDIS helps astrophysicists understand how exploding stars appear to observers by creating spectra - essentially predicting what these star explosions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

List of 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. • 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. • Handles packet interactions including scattering, absorption, and re-emission. • BasePlasma stores essential plasma properties such as DilutePlanckianRadField, TimeExplosion, JBlues, AtomicData, DilutionFactor, etc. • OpacityState manages line and continuum opacities for each shell, including electron densities and t_electrons. • These data are crucial for the iterations.
Monte Carlo Iteration Handler	• Controls the main iteration loop for convergence. • The data (dilution factor, radiation temperature and t_inner) is updated in the simulation state and plasma for every iteration. • Monte Carlo and opacities are solved using the updated values. • 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 traditional way of getting insights from various atomic datasets is to directly look at the modules of TARDIS because CARSUS currently cannot read atomic datasets. Astrophysicists and researchers spend time in creating their custom atomic files, comparing two atomic files using the scripts in the documentation. Everytime a dataset is output (HDF5 file) to TARDIS, it is not possible to get it back to CARSUS.

The Project - CARSUS Dashboard aims to provide a dashboard that facilitates numerous use cases. Researchers can generate various atomic datasets using a variety of available options. A dedicated dashboard for this data would allow users to easily explore and understand the datasets. Instead of retrieving information from TARDIS modules, users can use the dashboard to gain valuable insights. Built using Flask and Jinja2, the application offers multiple API endpoints that allow users to request and visualize insights about atomic data in a nicely formatted way.

Some of the features of the application include:
1. Filtering data according to atomic numbers, ascending/descending order of wavelengths, line/level ids etc.
2. Supports viewing existing atomic datasets.

3. Provides features to compare different atomic datasets, helping researchers validate data across sources.
4. System is designed to support both legacy atomic files and newer formats, ensuring long-term usability.
5. Includes thorough documentation and test coverage to maintain ease of use.

First Objective Task:

Task : Use Jinja2 to generate an HTML Report that investigates an atomic file. Display top 50 rows of levels and lines dataframes from the atomic file for Silicon.

Link to the github repository : Solution

Explanation : Custom Atomic file is created using NIST's atomic weights and ionization energies. GFALL Reader is used to get the data of lines and levels for the Silicon atom. TARDISAtomData class is used to create the atom data. This atom data is loaded on the first server load. When the user hits the endpoint (/get_data) the first 50 rows of both lines and levels data is outputted on the screen.

Screenshots of the data :

Lines

Atomic Energy Lines

ATOMIC NUMBER	ION NUMBER	LEVEL NUMBER LOWER	LEVEL NUMBER UPPER	LINE ID	WAVELENGTH	F_UL	F_LU
14.0	0.0	1.0	484.0	58639.0	1524.02	0.00	0.00
14.0	0.0	1.0	429.0	58701.0	1524.99	0.00	0.00
14.0	0.0	4.0	400.0	60089.0	1991.47	0.00	0.00
14.0	0.0	1.0	396.0	58749.0	1526.14	0.00	0.00
14.0	0.0	4.0	373.0	60097.0	1993.83	0.00	0.00
14.0	0.0	1.0	370.0	58847.0	1527.55	0.00	0.00
14.0	0.0	4.0	348.0	60105.0	1996.65	0.00	0.00
14.0	0.0	1.0	340.0	58941.0	1529.26	0.00	0.00

Levels

Atomic Energy Levels

ATOMIC NUMBER	ION NUMBER	LEVEL NUMBER	ENERGY	G	METASTABLE
14	0	0	0.0000	1.0000	True
14	0	1	0.0096	3.0000	True
14	0	2	0.0277	5.0000	True
14	0	3	0.7810	5.0000	True
14	0	4	1.9087	1.0000	True
14	0	5	4.1319	5.0000	True
14	0	6	4.9201	1.0000	False
14	0	7	4.9296	3.0000	False

My Solution for the project:

Tech Stack : Flask, Jinja2 and TailwindCSS for styling

High level overview of my approach:

Researchers create various atomic datasets. As soon as the dataset is created, it can be stored in the database as an HDF file. The dashboard outputs the data when requests are passed from the frontend. There are two approaches that can be followed to solve this problem.

1. Looping through all the existing HDF files (atomic datasets) and the newly created ones and outputting them to the dashboard.
    a. Pros : The user need not send any requests to the backend for a new atomic file. The application can be static.
    b. Cons : Everybody has access to all the datasets. The dashboard would be overloaded with a lot of datasets. Comparing datasets would be a problem.
2. Every HDF file in the database can be assigned with a proper name and can be given to the user. The user can use that name to fetch details of that particular dataset to the dashboard.
    a. Pros : The user can view the dataset of their specific interest. Users can compare different datasets just by sending requests to the backend with the atomic file names.
    b. Cons : The dashboard must be dynamic. (sending requests to the backend and handling the responses in the frontend state)

I believe that the second approach would be better because it allows for user-specific control. It ensures a cleaner and good dashboard experience. The rest of the proposal is drafted based on that approach. I am completely open to any advice or alternative approaches suggested by my mentor.

My Approach:

The two main features of this approach is to provide a dashboard for existing atomic datasets and provide an option for the user to compare two different atomic datasets. Primarily the dataset is loaded on the server side and part of it (ex: first 100 rows) are rendered on the frontend (like in the first objective task). More data can be retrieved by sending requests to the backend. I've included all the endpoints and a high level overview of how the dashboard would work.

On the server side, we can maintain a list of all available atomic files. When a user requests a specific file, the server loads the corresponding atomic data and returns it. Various endpoints can be implemented to serve the data in specific formats—for example, displaying only the first 100 rows in the dashboard. Additionally, dedicated endpoints can be provided to support comparison between different datasets.

Backend Features

1. Fetching data

Python API endpoints:

Endpoint	HTTP Method	URL Pattern
Retrieve Lines Data	GET	/get_lines_data?atomic_file_name=&?no_of_rows=
Retrieve Levels Data	GET	/get_levels_data?atomic_file_name=&?no_of_rows=
Retrieve Collisions Data	GET	/get_collisions_data?atomic_file_name=&?no_of_rows=
Retrieve Ionization Energies	GET	/ionization_energies?atomic_file_name=&atomic_number=&ion_number=&min_energy=
Retrieve Macro Atom Data	GET	/macro_atom?atomic_file_name=&atomic_number=&ion_number=&min_transition_probability=&?no_of_rows=
Retrieve Collisions Data	GET	/collisions_data?atomic_file_name=&atomic_number=&ion_charge=&min_energy=&max_energy=&?no_of_rows=

Retrieve Cross Sections Data	GET	/cross_sections?atomic_file_name=&atomic_number=&ion_charge=&min_energy=&?no_of_rows=

GET: Retrieves data when a valid atomic_file_name (mandatory) and necessary options (optional) are provided through query parameters.

The query parameters are used to filter the data on the server and send only required data to the frontend. This enables flexible filtering, such as retrieving data with wavelengths greater than 500 nanometers, limiting the output to just the first 10 rows etc.

2. Comparing Atomic Files

Python API endpoints:

Endpoint	HTTP Method	URL Pattern	Body
Fetch key differences	GET	/key_diff?atomic_file1=&atomic_file2=&diff_in=linesorlevels	NA
Fetch ion differences	POST	/ion_diff??atomic_file1=&atomic_file2=	{key_name: string, ion: string or tuple, rtol: int, simplify_output=bool, return_summary=bool, style=bool, style_axis=int}
Fetch the plot for ion differences	POST	/plot_ion_diff?atomic_file_name1=&atomic_file_name2=	{key_name: string, ion: string or tuple, column: string}

Frontend Features

- Users can browse specific parts of the atomic datasets.
- Filtering options for atomic numbers, wavelengths, and metadata.

- Comparison tools to analyze differences between datasets
- Tailwind styling to provide a clear and understandable dashboard.
- Sorting options with respect to a column in the dataset.

Milestones:

Week	Focus Area	Deliverables
Week 1	• Communicate my ideas with the mentor and understand the requirements. • Build a prototype of the dashboard and ask for suggestions to improve my framework.	• NA

Week 2	• Create a backend server and write logic for fetching the atomic datasets:
	  ○ Fetching lines data endpoint
	  ○ Fetching levels data endpoint
	• Ask for feedback on my logic and improve accordingly.
		• A functional backend that sends Jinja2 templates of lines and levels data of atomic datasets to the frontend dashboard.

Week 3	• Finish writing all the endpoints logic for fetching the data of atomic files. • Incorporate filtering logic according to the query parameters.	• A working backend that provides information of various atomic datasets
Week 4	• Write unit tests for all the endpoints. • Document all the code changes made to the repository. • Ask for feedback.	• Full test coverage for the data retrieval part. • Documentation of how to use the endpoints that are required to fetch atomic datasets

Week 5	• Start writing frontend code to output the dashboard in the form of Jinja2 templates. • Ask for design (for the dashboard) suggestions and incorporate best practices. • Refactor the code.	• A dashboard that outputs all the atomic files' information in a nicely formatted fashion.

Week 6	• Incorporate sorting functionality for the datasets on the frontend.
	• Start writing backend code to fetch the comparisons
	  ○ Key differences endpoint
	  ○ Ion difference endpoint
		• Dashboard that facilitates sorting according to any column and filtering according to query parameters.

Week 7	• Finish the backend for comparing atomic datasets. • Communicate with my mentor and ask for feedback.	• A working Comparison API that is useful to compare two different atomic datasets.
Week 8	• Code the frontend for comparing atomic files.	• NA
Week 9	• Documentation for all the code written. • Write unit tests for comparison API.	• Documentation for the code changes. • Unit tests for all the functions.
Week 10	• Refactor the code with best practices. • Write integration tests. • Ask for feedback and improve.	• Deployable dashboard that features atomic data filtering and comparison of different datasets.

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 immensely. 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
