GSoC ‘16 Project Proposal
Extensive Test Suite for TARDIS

BASIC INFORMATION

Name and Contact Information

- **Name:** Karan Desai
- **Email:** karandesai281196@gmail.com
- **IRC Nickname:** d4rth
- **Phone Number:** +91 9924 866 384
- **Github/ Gitter Chat Room:** karandesai-96
- **Skype:** live:karandesai_96
- **Alternate video chat:** http://appear.in/karandesai-96

University and Current Enrollment:

- **University:** Indian Institute of Technology, Roorkee.
- **Field of Study:** Electrical Engineering (Batch of 2018)

Meeting with mentors:

- Reachable anytime easily through Gitter, email or IRC. A well planned session if required:
- Tuesdays, Thursdays or Saturdays between 6:00 pm - 12:00 am IST.

CODING SKILLS

Programming Languages

- Fluent in Python. Sound knowledge of Object Oriented Programming.
- Cython and C/C++ - moderate knowledge (will catch up if needed).
- Fluent in Android App Development using JAVA.

Development Environment

- Ubuntu 14.04 LTS, Python 2.7 installed.
- PyCharm Professional IDE for Python Development.
- Minor changes done in sublime text editor / nano.
- Good familiarity with conda / virtualenv.

Version Control

- Top Preference: Git (very strong concepts)
- Also comfortable with Mercurial and SVN.

PRE-GSOC INVOLVEMENTS

Here is a list of my PRs made so far:

- **#491 (merged):** Included IDE specific files in gitignore.
- **#499:** Chose Atomic Datasets project at first and completed its prelim task.
- **#507:** Contributed towards increasing the code coverage of tardis/util.py.
- **#508:** Shifted to Testing Project**,** completed its prelim task.
- **#510 (merged):** Fixed **#509** and restored failing builds on Travis CI.
- **#526 (merged):** Automated inclusion of conda-requirements in documentation.
- **#529**: Fix a comparison typo in tardis/atomic.py.

PROJECT INTRODUCTION

TARDIS has scientific code. While many people working together on various functionalities - there are possibilities for anything to break and go undetected for a long time. It might misbehave at times, detecting it would get very difficult. Testing a scientific code like TARDIS is crucial - the test suite needs to be up-to-date and should have a good code coverage.

Current Scenario

- TARDIS uses pytest as the primary testing framework. Unit tests are maintained inside the **tests** directories and have tests for individual 'units' (methods).
- To test the full TARDIS code, there is a single class in tardis/test_tardis_full.py with a simple configuration. The current unit tests are fast enough and are executed within around three to four minutes on Travis-CI.
- Current code coverage is approximately **54-55 %** , there are possibilities that some part of code is untested and developers may not be able to comment about its health.
- The current test case for an integration test of full TARDIS code does not exhaustively probe every functionality of TARDIS and hence **does not ensure rigorous testing** of the whole code.

Required Improvements

- TARDIS code must be tested against more complex setups which ensure validation against all physical setups we want to simulate.

- There has to be a framework which ensures seamless addition of a new complex setup for integration testing of TARDIS at any time.
- These complex tests must not be run on Travis corresponding to every pushed commit to Github as they are very slow and resource extensive. They should be run on a dedicated server at regular time intervals.
- Failing integration tests do not describe where it all went wrong - hence even the question of **how the test failed** - is also important.
- If all of this works fine - the results of these tests be made viewable directly to Github.

ACHIEVING PROJECT GOALS

The whole project timeline can be broadly classified into three phases through to completion:

- **Primary Goals :** Deliverable before Mid-Term Evaluation.
  - **1.1 Set up a Framework for efficient handling of Slow Tests.**
  - **1.2 Run slow tests on a server at regular intervals.**
- **Secondary Goals :** To be completed before the programme ends.
  - **2.1 Notify through email once the test is executed.**
  - **2.2 Make appropriate plots according to the results.**
- **Wishlist :** Can be implemented if both Primary and Secondary Goals are completed successfully and the programme is yet to end, also desired to be taken up post GSoC.
  - **3.1 Show the build status on Github.**
  - **3.2 General increase of code coverage by writing Unit Tests.**

Primary Goals

1.1 Set up a Framework for efficient handling of Slow Tests.

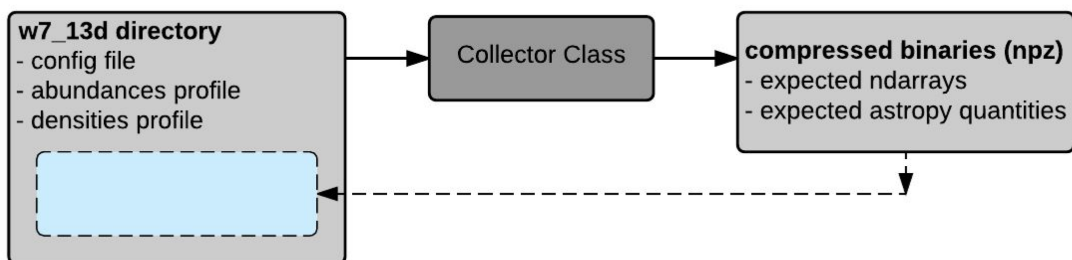
- I have been introduced to two complex setups - Stratified W7 setup and Abn Tom Setup. These extensively test the full code of TARDIS and take considerably long time, hence they can be classified as Slow Tests.
- My proposed directory structure is (only including the additions):

```
tardis
    └─── tests
            ├─── data
            ├─── tests_slow
            │     ├─── w7_13d
            │     │     ├─── tardis_w7.yml
            │     │     ├─── w7_densitites.dat
            │     │     ├─── w7_abundancies.dat
```

```
        │     │     ├──── expected_ndarrays.npz
        │     │     └──── expected_quantities.npz
        │     ├──── abn_tom   # same as w7 within
        │     ├──── base.py
        │     ├──── collector.py
        │     ├──── test_abn_tom.py
        │     └──── test_w7.py
        └──── # existing content in 'tests'
```

- For creating tests, we need benchmark data to perform assertions. This data has to be obtained by doing a run with corresponding setup - using a trustworthy version of TARDIS. Also, on future changes to the codebase, it might become inevitable to update the benchmark data. This cannot be always done manually. For this job, **collector.py** serves the purpose.

- collector.py will have a **Collector** class, explained as (using **w7_13d** dir as example):



- There would be similarities as well as differences between the parameters of Radial 1D Model obtained after the run with each of the slow tests setups. Similarities can be put altogether in base class and assertions for these can be easily done by making **super()** calls from the individual classes, even pytest's parametrization can work. If a parameter in Radial 1D Model is specific to any setup - a separate method can be made within the individual class and assertions can be made accordingly.

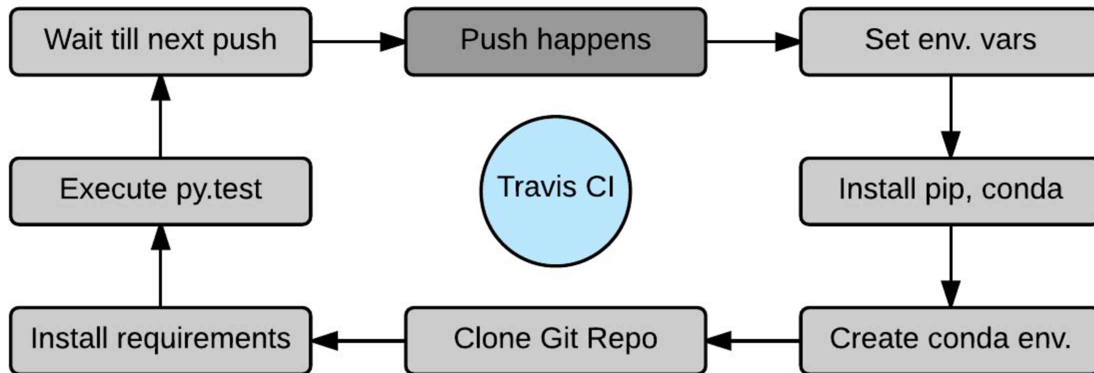
- I have already coded a prototype - one base class and a child class for integration test using Stratified W7 setup, and it is fully functional. There is scope for improvement.

- **REFER COMMENT of PR #508** recent commits also contain the prototype.

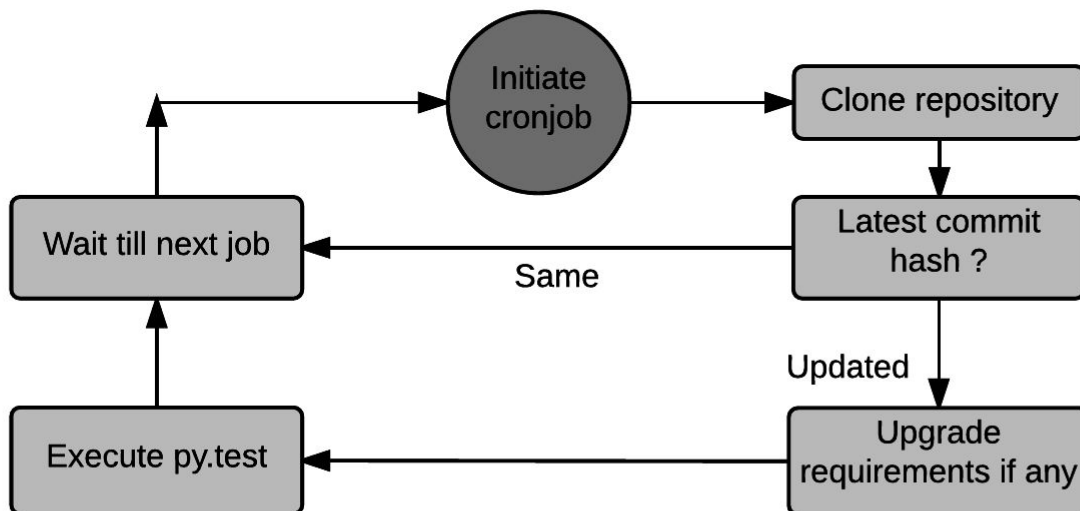
1.2 Run slow tests on a server at regular intervals.

- Slow tests are time consuming and resource extensive and cannot be run on Travis CI, as it would result in sluggish development cycle of the whole TARDIS codebase. As suggested they should be run on opensupernova.org - official group server of TARDIS.

- This can be accomplished using cron jobs. Here is a representation of what mainly happens at our Travis CI (minimal aspects included):



- Slow tests would be run on group server, which is highly customizable as compared to Travis. The flow can be reduced down as described in this representation:



- Upgrading requirements is rarely done hence this **can be skipped and manually done** by the veterans of the organization. This flow is visualized on certain assumptions:
    - As the server is dedicated to TARDIS - the libraries here are the only needed ones for TARDIS and there is no need to set up a conda environment. If there is one - it is already sourced, or this flow will include an additional block to source the existing environment.
    - There is no longer any requirement to always set the environment variables on every run. They can be set and kept as they are.
    - Obtaining **kurucz_cd23_chianti_H_He.h5** is handled by Collector. It can be shifted as a routine of cron job.

- To counter these assumptions, the current flow can easily be extended to include blocks from Travis CI's representation.
- The complete script can be written as a shell script or a python script by making extensive use of **'os' module.** A quick workaround to retrieve commit hash:

```
from subprocess import check_output

def latest_commit_sha1_long():
	return check_output(['git', 'rev-parse', 'HEAD'])

def latest_commit_sha1_short():
	return check_output(['git', 'rev-parse', '--short', 'HEAD'])
```

These two tasks will be the highest priority to be done. As of now - the status of tests can be viewed manually by the owners of servers. Facility of notification through e-mail / github is included further.

Secondary Goals

2.1 Notify through email once the test is executed.

- Once the primary goals are up and functioning, it would be possible by the owners of the server to manually check the build status of the current run. It can be eased up by notification through e-mail. The script for e-mailing can be put in separately and can be executed just after the tests are done. One additional line will be added to the script which is executed by cron job to send email - that is running this script.

- This script will mainly take care of **two jobs** - one is to get the status of just completed run of test, and another to send an email to core team of the organization. Additionally it can retrieve the start time of execution, duration of execution, and the commit hash - and include all this information in the mail.

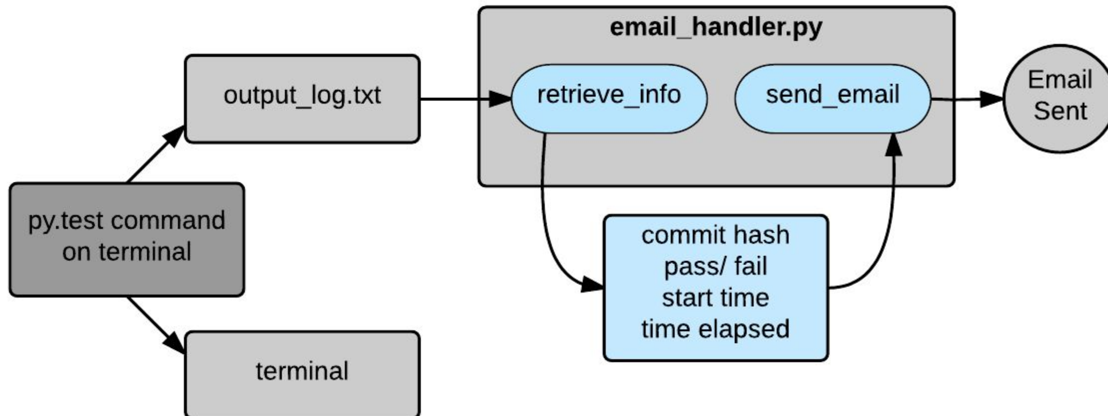
- For retrieving information, the output generated by py.test should be logged into a text file. The log has information of our interest at fixed locations. We can use:

  py.test --run-slow [other params] | tee output_log.txt
- Another approach is to use Python's own **logging** module. Its code would be in the script for tests, and not the script for cron jobs.

- This text file can be ingested into the script and information can be retrieved easily to do well structured format of log generated by py.test.

- Emailing is facilitated by **Simple Mail Transfer Protocol** (SMTP). Python has a very standard and easy to use **smtplib.** Using this library, the facility of notification through email will be up and functional.

- In a nutshell everything would go as (assuming commit hashes are different and a run of tests was triggered):



2.2 Make appropriate plots according to the results.

- Matplotlib is the best plotting library and currently used for plotting purposes in TARDIS. It has functionality to plot the test results with benchmark data - then there is another library named pytest-mpl which is built on top of matplotlib and will easily be linked with pytest. I have looked through the README of its Github repo. Having a look at it seems like it is the best fit for the task and is easy to implement.
- The **`@pytest.mark.mpl_image_compare`** marker can be used for the purpose.
- The plotting facility can be enabled within the base class itself and the plots can be saved in any directory outside TARDIS repository, for example it is in **/tmp/tardis_plots**.

```
tardis_plots
            ├──23a19bd
            │      ├──── w7_13d
            │      │      ├──── j_blue_estimators.png
            │      │      ├──── nubar_estimators.png
            │      │      ├──── montecarlo_luminosity.png
            │      │      ├──── montecarlo_virtual_luminosity.png
            │      │      └──── t_rads.png # more plots
            │      ├──── abn_tom   # same as w7 within
            ├──── 78c1ab3
            └──── # other commits
```

- Further, this directory will be **symlinked with /var/www/html (**using ln -s**)** to make images of plot available on a specific url. The images cannot be viewed directly in browser and just the option to download them will be available - a web page can be designed for the purpose but it kept out of scope of deliverables.

Wishlist

3.1 Show the build status on Github.

- The image can be standard and easily available from shields.io or can be custom made. To display this to Github's README, a proposed method is:

- First of all we must have at least two images - one signifying passing build and another for failing build. If the tests are way too long (days), we can have one more image of pending build too - which signifies that the tests are being run and also shows the status of recent build.

  Here is a one solution I can think of:

- So they are kept in say **/tmp/build_images/all** and the appropriate image to be displayed can be updated as **/tmp/build_images/recent/status_image.png.**

- This path can be **symlinked with /var/www/html** as done above, and hence we can obtain a unique url which can be used in README to display the image from server. This image will be overwritten by anyone image from **/tmp/build_images/all** after every build and this is easily doable, a shell script or python script will be sufficient. This will create a badge on our README just like Travis and coveralls.

3.2 General increase of code coverage by writing Unit Tests.

- Writing some unit tests for untested methods and contributing towards increasing code coverage be an asset to the codebase. I have already done it for **tardis/util.py** and any work done related to writing unit tests, will be carried out exactly in the way followed in PR #507.

PROPOSED TIMELINE

I have semester examinations between 23 April - 1 May. I will inactive between 10 April to 2 May for exam preparations, which is way off the timeline.

I believe I have enough fuel to get started on my goals - as a result to my involvement with the organization for almost two months now. So I will be setting grounds early to avoid stopgaps during the actual GSoC period.

There are a couple of weeks before the actual timeline where I would make sure that I have done all the homework.

All dates mentioned here are weeks - starting from Monday and ending on Sunday.

28 March - 3 April (HomeWork Week)

- **Improvement of Integration Test Classes of Stratified W7 and Abn Tom setups.**
- I have done some work upon the Slow tests using Stratified W7 and Abn Tom Setups. The work and its description so far has been reported in **PR** **#508**. Base class has been made and these two classes for tests have inherited it - much work is done through **super()** calls. I will work on this PR and update it further during this week.

4 April - 10 April (HomeWork Week)

- **Implement pytest-mpl on test_tardis_full.py and test_w7.py**
- I will be practicing code to generate plots on test_tardis_full.py initially - both on passing tests as well as by deliberately failing tests.
- Once this task is done, I will do the same for test_w7.py and think of some way is possible, to have a general architecture tightly bound with main architecture of classes of tests - in case there is too much repeated code for plotting.
- By this I will make sure that I will do this task swiftly during the main timeline.

11 April - 8 May (Hiatus)

- End semester examinations would be near so I will be inactive - though always available on Gitter channel / E-Mail for communication whenever necessary. At times for recreation, I'll pick up an untested method and write simple unit tests to improve code coverage and open a PR.
- Between 2 May - 8 May, I will be packing up from University Campus and moving to home so will be totally inactive these six days.

9 May - 15 May (Community Bonding - I)

- **Set up initial directory structure and collect benchmark data by executing various runs.**
- By this time, I would be involved with TARDIS for more than two months, so I can start coding directly in community bonding period. I further divide this week's task as:
    - Include **collector.py** in the directory structure and make improvements in it after getting feedback from the mentors. **Collector** class functions well enough, complete with documentation and free from PEP8 issues. Clean up and finalization related to this won't take much time.
    - Obtain various config files, abundances and densities profiles for different setups classified under slow tests category. Put them within the directory structure and one by one perform runs using all of them and collecting the benchmark data as proposed - in compressed binary format.

16 May - 22 May (Community Bonding - II)

- **Finalizing the base class and creation of individual classes for Slow Tests.**
- I have created a base class named **SlowTest** in base.py as mentioned in the directory structure. There is scope for improvement. It will be finalized, concluding from a brief discussion in the chat room.
- Making child classes for individual tests inherit the base class has also been implemented to an extent. **super()** calls have been made from TestW7 class to SlowTests class' methods for performing assertions.
- Already done work on this can be viewed in commit 716d7b8 of my PR #508. I will be occupied this week in taking this work further in a fresh PR and leading it to finalization.

23 May - 29 May (Week I)

- **Refining the work done so far on Slow Tests.**
- It is not necessary that everything goes as planned out and there might be unavoidable delays due to a nasty bug hidden from eyesight. My target in this week will be to put a firm pencils down on work done so far - updating documentation and code cleanup.
- **Trivial Wishlist:** My first major contribution to TARDIS is still an open PR - #507 . It misses unit test for one method. I have found a way to write it. I will complete it and shoot the coverage of **tardis/util.py** to 100%.
- I don't think I would be left with more time - but rest of it will go to starting **Week II** work**.**

Milestone Received: Primary Goal Task 1.

30 May - 5 June (Week II)

- **Creating a script from facilitating cron jobs.**
- I have already presented a flow chart in **Project Goals** section. It can all be nicely wrapped in a python script which, in turn will be wrapped by a shell script. It will clone the repo and run the python script - when it is time for the cronjob. Week II dedicated for this task.

6 June - 12 June (Week III)

- **Continuation on script and clean up job for Mid-Term Evaluation.**
- Same as mentioned before - this is a buffer week for preparing the code to be presentable for mid-term evaluation. Finalization of script for cron job and updating its documentation will be the prime target this week.

Milestone Received: Primary Goal Task 2.

These deliverables are minimal viable in my opinion for the mid-term evaluation. If not anything done further before Mid-Term Evaluation - this is what has to be submitted.

13 June - 19 June (Week IV)

- **Update python script for cron jobs to include facility of notifying core team about test status through e-mail.**
- Thanks to the previous experience of using smtplib - this will be completed within a week, and relevant documentation will also be written.

Milestone Received: Secondary Goal Task 1.

19 June - 26 June (Week V)

- **Create plots for the tests and link them with a certain commit hash.**
- By now if everything goes well, I would have already applied **pytest-mpl** for **test_tardis_full.py** and **test_w7.py,** having made sure during HomeWork week that it works perfectly, it will be smooth in implementing it for all the tests - most of it will go in the base class alone it avoid redundant code.

27 June - 3 July (Week VI)

- **Continue with the task of previous week.**
- This week will be dedicated to finalization of the plotting facility. By the end of this week, the tests will be functional with creating plots.

4 July - 10 July (Week VII)

- **Finalize the content of deliverables after Mid-Term Evaluation and update documentation.**
- All the minutiae related to email notification (if in case it remains) as well as plotting in tests will be taken care in this week. This serves as a buffer week before the main deliverables get polished and presentable.

Milestone Received: Secondary Goal Task 2.

My institute re-opens on 18 July, till now I was totally free throughout the day so could easily code for 8 - 10 hours a day. I have kept enough buffers to clear backlogs if any and I am confident that I can work upon the timeline I stated so far. The work done further would be from my **Wishlist** category. I can now work for about 7 - 8 hours a day.

11 July - 17 July (Week VIII)

- **Add tests for C parts of the codebase.**
- Tests will be added for montecarlo C code. Current tests are simply the assertions of expected and obtained results, the idea is to include as many expected failures as possible - for better bug detection.
- **NOTE:** I will be packing up and shifting back to my university for the next of new semester. Due to travelling, I would be inactive on Friday, Saturday and Sunday this week.

18 July - 24 July (Week IX - might be a little inactive)

- **Continue writing tests as the previous week.**
- Conclusion and clean up on writing C tests - continued from previous week.
- **Trivial Wishlist:** Increase as much code coverage as possible for tardis/model.py
- **NOTE:** Registration affairs of new semester, I will be inactive on Monday and Tuesday.

25 July - 31 July (Week X)

- **Show the result of tests on Github.**
- I haven't tried it yet and theoretically I believe it should be no problem - but I kept it in wish list as it might not turn out as expected. For the latter case, I will start work for next week.

1 August - 7 August (Week XI)

- **Increase as much code coverage as possible for tardis/analysis.py**
- Currently it has zero coverage, this week will be dedicated to push up its coverage.

8 August - 14 August (Week XII)

- **Tentative wrap up - cosmetic refinements.**
- Update documentation - write unit tests wherever missed out and do a general code cleanup. Make sure there is nothing left undone and everything is tidy.

15 August - 21 August (Week XIII)

- Prepare content for End Term Evaluation, including the deliverables from wishlist.

22 August - 30 August (Week XIV)

- Buffer week and conclusion of GSoC.

MORE ABOUT ME

I am a 19 year old, second year student currently enrolled in Electrical Engineering (IV Year Course) at IIT Roorkee. I developed a passion for programming and web development in my freshman year. I contribute to open source regularly since about six months now - looking over to repositories of the products I use / come across, trying to give my part of contribution back to the organization whose product has been an asset to me.

I am an active member of Mobile Development Group (SDS) at IIT Roorkee, a bunch of passionate enthusiasts trying to foster mobile development culture in the campus. I have a previous experience to work as per my proposed timeline with a mentor on a summer project.

I made an android game - **MagnetoMania** within the time period of 25 days - almost two weeks ahead of the expected time. The game is kept under alpha testing, the debug version can be downloaded to any android phone from here and can be played. More about me in my resumé.

WHY ME FOR THE PROJECT?

My first contribution to open source was about three months ago. I have made a couple of contributions here and there, whenever I find some code that can be made better, some documentation that can be improved, some method which needs tests. The contribution activity can be viewed in my Github profile, significant highlights are contributions in OpenCog, SageMath (Github mirror) and OWASP-OWTF.

I shelled out around 4000 lines of code within a period of 25 days to make a fully functional android game and I made it possible because of my beforehand planning and necessary homework - for swift coding during the actual timeline. Same would be exercised with this project. I intend to **maintain a dev log** - where I would be updating my work **everyday** - for mentors to know easily what I am up to.

During vacations (5 May - 18 July) I have no other commitments and I seldom get tired of coding - even for recreation I keep coding. Unconditional Python love ! I can easily devote more than 50 hours per week till 18 July. After that my institute reopens and I will have to spare some time for regular academics, hence I could then devote about 35-40 hours a week, but according to my timeline, the primary and secondary goals would be completed by that time, so everything is manageable throughout the timeline. I believe that the allotted work per week is completely doable by me and neither overloaded nor slacked.

I make atomic commits with clean commit messages and well structured PRs - something which I feel is very crucial for this Testing Project. I will be working on this fork and the most recent version of the document can be accessed from here.

WHY DID I CHOOSE TARDIS?

I was searching through previous year's accepted organizations - came to TARDIS Gitter chat just on the day when the ideas page was made public. The ideas page as well as TEPs clearly gave an introduction of what has to be done, and it was understandable by a person who had little knowledge about the codebase as well as astrophysics. The prelim tasks were easily doable and gave a boost.

As the days passed, I got more and more support from the community. Now I know exactly what I have to do during the timeline and pretty confident about the project. I have only applied to one organization for GSoC and that is TARDIS.

I believe that for a scientific code like TARDIS, managing the code base is as important as developing it. While I am having a will to learn the latter, I believe that my contributions targeting the former would ease up others' work, letting them focus more on the scientific aspect of TARDIS. Looking at the TEPs, I am pretty sure that the future versions to be rolled out would be very much better than now and I would love to be a part of the process.

REFERENCES

1. GSoC 2016 Ideas Page
2. TEP001 - Extensive Test Suite for TARDIS
3. TARDIS - ReadTheDocs
4. Pytest Documentation
5. Introduction to Cron
6. Logging Module Documentation
7. SMTP Protocol and smtplib for Python
8. Pytest-mpl Documentation
9. TARDIS Coveralls Reports

* * * * * * * * * *