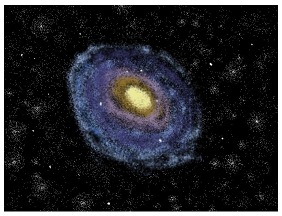

Team Andromeda

Version 2

Software Design Document

February 21, 2020

Clients - Dr. Audrey Thirouin and Dr. Will Grundy
Mentor - Isaac Shaffer
Members - Matthew Amato-Yarbrough, Batai Finley, Bradley Kukuk, John Jacobelli, and Jessica Smith

Table of Contents

1. Introduction

Humans have been studying the universe for thousands of years. Billions of dollars are spent every year on sending probes to other planets and small bodies in an attempt to understand what lies in the cosmos. This continuous research has driven technological development in areas not directly related to space. For example, home insulation, baby formula, and portable computers are a few of the byproducts of space exploration. Other valuable data and theories have been derived from space exploration as well, such as information about early planet formation.

Many observatories like Lowell are gathering information daily to further our comprehension of areas in space such as the Kuiper Belt, which is a region of the solar system beyond the orbit of Neptune containing small bodies such as Pluto. Lowell and other observatories study how these small bodies form and interact with each other. Our clients Dr. Audrey Thirouin and Dr. Will Grundy work for Lowell Observatory. They focus on collecting and analyzing data about small bodies that reside far from Earth. Dr. Thirouin is a research scientist interested in the characteristics of small bodies within the Solar System, while Dr. Grundy is an astronomer that researches Kuiper Belt objects.

The Kuiper Belt is a region containing leftover bodies from the solar system's early history. This makes these celestial bodies valuable for observing conditions similar to that of early planet formation. Currently, our clients are working on modeling binary systems in the Kuiper Belt and need to use techniques that make the most use of the data available to achieve this. Distant objects are often difficult to study because only a pinpoint light source is observable. The fluctuations of the light reflected from the object are recorded and combined to form a light curve. A light curve is a graph that shows brightness values over a given set of time.

Light curves can be used to infer the rotational and physical properties of small bodies. For instance, an asteroid that is non-spherical will reflect more light when a larger portion of its surface is facing an observer. This is because more light from the Sun is being reflected to the observer. Since the object is reflecting more light at certain points in its rotation, the brightness will differ depending on when it is observed in its rotation. The various brightness values can then be graphed, generating the light curve. Information including the period and amplitude of the curve and rough properties such as the rotational period of the object can then be found based on graph data. A broad number of other characteristics can be estimated using light curves by an adept astronomer.

Our clients use light curves to better understand binary systems. Dr. Thirouin and Dr. Grundy can model these systems with spheres and faceted objects using software developed in a previous iteration of the project. Though this software does what is required, our clients wish to expand it to reach its best potential. The current issues they face are:

- Only being able to model spheres and faceted objects, which limits how fast they can model certain shapes
- Having to manually guess and retest parameters, which is time-consuming
- Having enter parameters into a CLI, which is tedious
- Having to manually create a video of the output renders, which is tiresome

Our clients need upgraded software to improve, expand, and streamline the software. They wish to do this through the addition of:

- Triaxial ellipsoid shapes
- A Markov chain Monte Carlo (MCMC) algorithm
- A graphical user interface (GUI)
- A video generator

To implement this solution, the triaxial ellipsoids will need to accurately use ray tracing to simulate the ellipsoid-shaped system, as well as rotate the ellipsoid around a given pole. In addition, the Hamiltonian Monte Carlo (HMC) algorithm will need to calculate a range of values that are likely for a given parameter. Furthermore, the new GUI will increase ease-of-use for the software. Finally, the movie generator will need to compile images into a movie format for users to manipulate as needed.

The solution for this iteration of the project will allow for more accurate modeling of binary systems and help users generate precise characteristics of these systems. Our new implementations will also help simplify and expedite our clients' workflow overall.

2. Implementation Overview

In this iteration of the project, our plan is to add two modules and two submodules to the preexisting six modules implemented last year by Paired Planet Technologies. These modules and submodules will be designed to either add or extend functionalities from the previously implemented modules. By doing so, we can uphold the preexisting modular design and implement additional functionalities within our iteration of the project. The distinct functionalities to be added are triaxial ellipsoid rendering, a Hamiltonian Monte Carlo (HMC) algorithm, a graphical user interface (GUI), and a movie generator. The function that collectively utilizes these features to produce a light curve is called the forward model. To better understand their role in the project, these features are broken down in the following paragraphs.

Triaxial Ellipsoid

The triaxial ellipsoid submodule increases the functionality of the previously implemented shape module. When given the appropriate parameters, this new submodule will be capable of performing calculations for the forward model using triaxial ellipsoid objects. Additionally, it has the ability to display a triaxial ellipsoid object when calling the forward model. Both functionalities are created by using the forward model module in conjunction with the shape and ray tracing modules. The aforementioned parameters will be supplied by the users and will dictate how the triaxial ellipsoid object is shaped and displayed.

Hamiltonian Monte Carlo (HMC)

The HMC module will provide additional functionality to the codebase using the implemented forward model module. This new module will use two submodules to serve the purpose of finding the best set of estimated parameters to match the observed light curve to the predicted light curve. The parameters that are being estimated will be determined by the users. Furthermore, the data for the observed light curve will be provided to the HMC module by the users, while the predicted light curve will be produced using the forward model with the constant parameters provided by the users.

Graphical User Interface (GUI)
The GUI module will allow the forward model module to be user-friendly. This new module will allow our users to enter parameters into the forward model from the GUI rather than entering them at the command line. Additionally, users will be able to see the predicted light curve directly on the screen and have the option to compare the predicted light curve to the observed light curve.

Movie Generator
The movie generator submodule will add new functionality to the forward model in the form of a movie generation script. When the script is run, it will take in a collection of images produced using the forward model and condense them down to a .mp4 file. The script will be accessible from the folder previously mentioned where the collection of images will be stored.

These modules and submodules work together with the previously implemented software solution to provide additional, beneficial functionalities for users. The dependencies, use cases, and design of each module are thoroughly detailed in the following section.

3. Architectural Overview

Our solution will address our clients' problems by using the current API, frameworks, and math libraries to enhance performance and functionality. A framework will be used to provide a GUI that facilitates the entry of parameters for the forward model which will generate a light curve. A separate framework will be added to the GUI, allowing for data plots. Our solution will also include a Shape subclass for the implementation of triaxial ellipsoids. Additionally, an HMC API will be used to handle the implementation of the HMC algorithm. Finally, a video of the images created by the existing API will be created in C++. Below is the dependency graph our team plans to follow for our iteration:

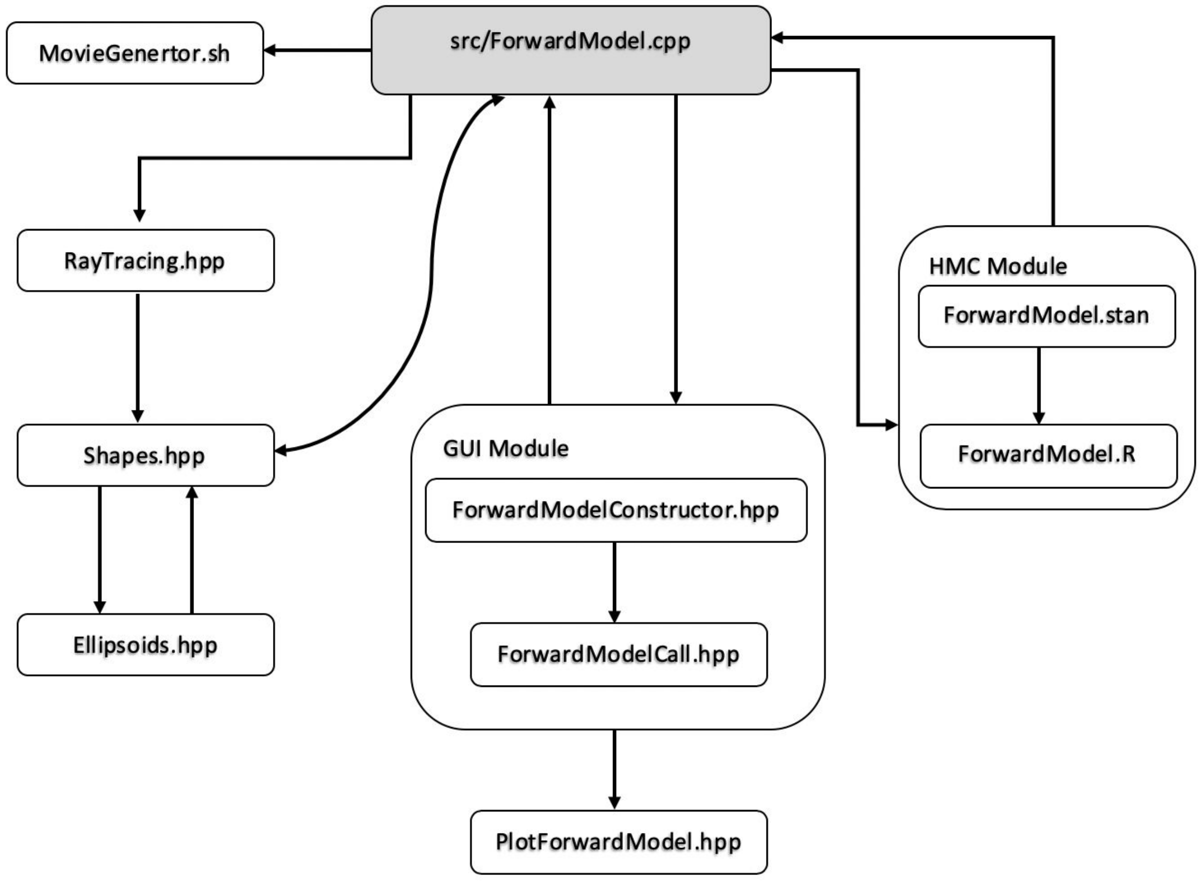

Figure 1: The flow of data within our conceptual modules and submodules

Triaxial Ellipsoid
The addition of an Ellipsoid class to the existing API will enhance our user's research. The Ellipsoid shape will be added as a Shape class submodule. Although Paired Planet Technologies previously instantiated this class, there are some properties that need to be established. This class works in conjunction with Math.hpp, which is a header file that defines mathematical constants and functions.

Hamiltonian Monte Carlo
The Hamiltonian Monte Carlo module will be implemented through R and Stan using the RStan interface. The licht-cpp static library will be used in order to create the predicted light curves used within the Stan model. This will allow the HMC module to explore the parameter space of estimated values using the produced predicted light curves. As such, the user will be able to predict parameters that closely resemble observed light curve data.

Forward Model Graphical User Interface
The Forward Model Graphical User Interface will rely on several different frameworks and APIs to create the best experience for users. The front end will be developed using Tkinter, a Python 3.8.0+ framework that allows for quick and easy development. Ctypes will be used to interact with the shared object in the existing API and call the forward model. Plotly, a Python module, will be used for the plotting functionality of the interface. This allows the user to take in a data set with certain constraints and will generate the light curve for the user. This will allow users to observe, compare, and analyze the data set.

Movie Generator
Our solution for the movie generator will be the implementation of a C++ script that will gather the images generated from the existing API and sow them into a .mp4 file. Users will be able to create these movies for analysis and visual representation. An external library will be used to allow the user to generate movies of the model.

4. Module and Interface Descriptions

In an effort to retain modularity, the overall design involves separate modules and submodules. Each new major module or submodule remains semi-independent and only loosely interfaces with the others. Since there is minimal coupling, the way the modules are linked is loosely defined. Figure 2 shows the comprehensive blueprint of the modules and submodules previously implemented, the current implementation, and how they depend on each other.

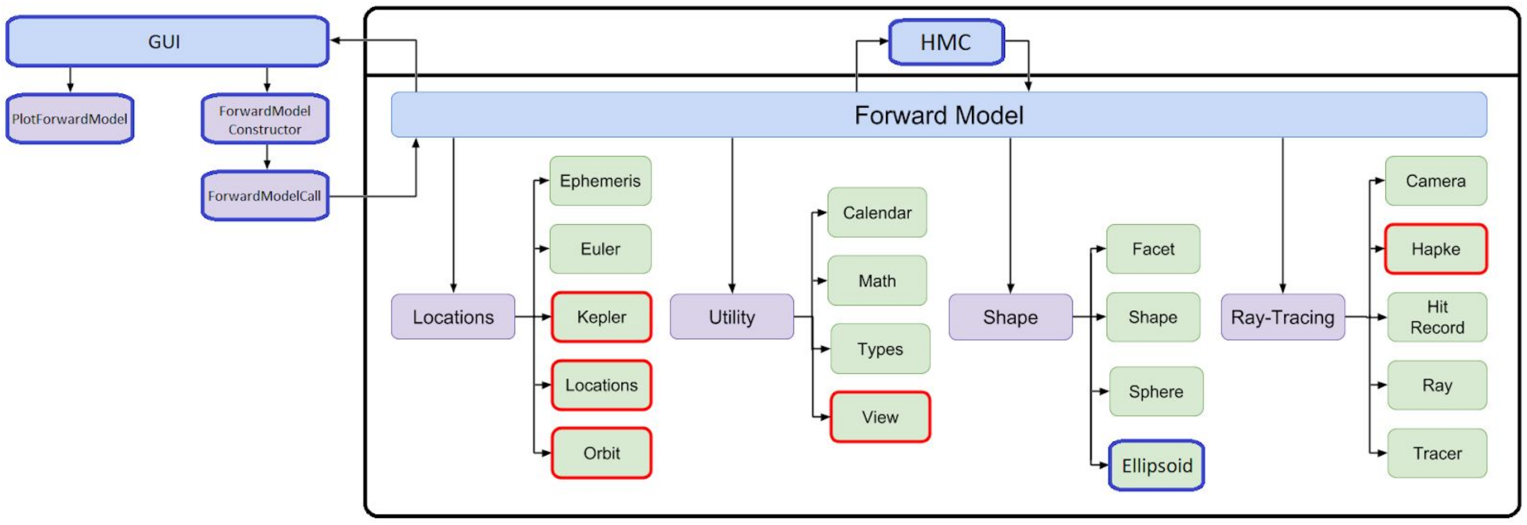

Figure 2: How the code base is split into conceptual modules

4.1 Ellipsoid

The Ellipsoid submodule is the third and final extension of the Shape module. It inherits and defines all functions from the Shape interface. It also uses the Math module for precise calculations. Since many asteroids are ellipsoid-shaped, this class allows for higher accuracy models while retaining a faster run time. While ellipsoids can be generated using facets, this class allows for ellipsoid generation at a much faster rate than the FacetedShape class.

Dependencies

- Shape.hpp
- Math.hpp

Use cases

- After the forward model has been called, the Shape module uses the calculations from the Ellipsoid submodule to generate the shape. The shape is then manipulated by the ray tracer.

Design

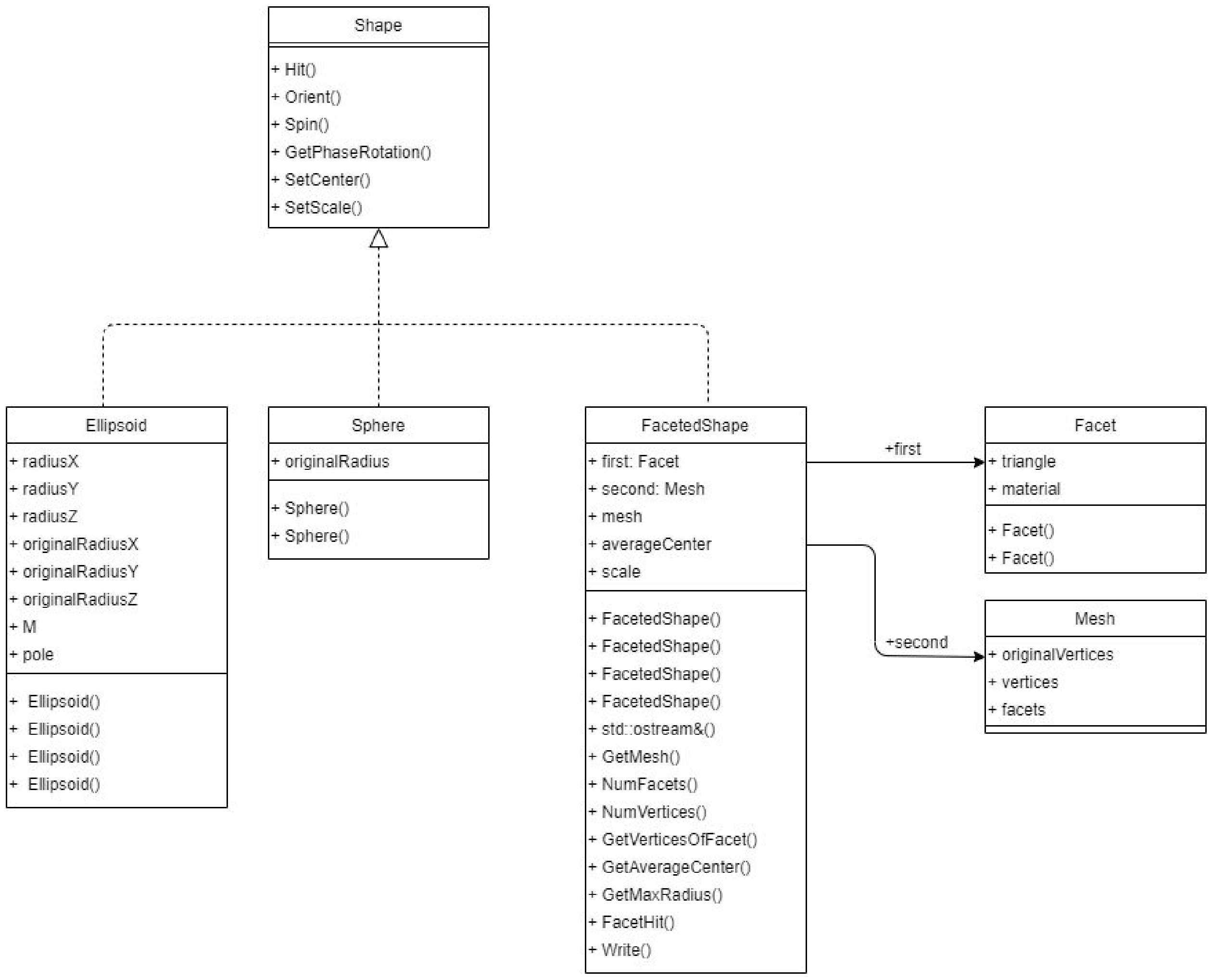

Figure 3: UML diagram describing the relation of all shape classes

The Ellipsoid submodule defines how an ellipsoid is visually displayed and how it can be manipulated. There are four separate constructors and five methods used to help accommodate these manipulations, as seen in Figure 3.

Constructors

- Given radii and Hapke
- Given radii, pole, and Hapke
- Given center, radii, and Hapke
- Given center, radius, and pole, Hapke

Methods

- Orient
    - Realigns the ellipsoids via rotation about a given pole so that it is oriented along the pole.
- Spin
    - Rotates the ellipsoid on the shortest axis to simulate its rotation, using rotation speed, epoch, and time. The y-axis is what is rotated about by default and is up to the user to make this shortest upon ellipsoid creation.
- SetCenter
    - Using a given center, sets the ellipsoid's center which dictates where the object is simulated.
- SetScale
    - Using the original radii given, calculates scale based off given scaling factor and saves into radiusX, radiusY, and radiusZ; also sets the matrix with new radii.
- Hit
    - Calculates the point of origin of the ellipsoid using the ray direction and the triaxial ellipsoid's matrix; sends this information to hitRecord which then calculates whether the object is hittable.

Variables

- radiusX
    - The ellipsoid's scaled x-axis length
- radiusY
    - The ellipsoid's scaled y-axis length
- radiusZ
    - The ellipsoid's scaled z-axis length
- originalRadiusX
    - The ellipsoid's original x-axis length
- originalRadiusY
    - The ellipsoid's original y-axis length

- originalRadiusZ
  - The ellipsoid's original z-axis length
- Eigen::Matrix3d M
  - The matrix that transforms the sphere into an ellipsoid through lens manipulation
- Vector3d pole
  - The pole gives a direction for the ellipsoid to rotate to and rotate on

4.2 Hamiltonian Monte Carlo (HMC)

The HMC module provides the required tools for computing parameter estimates and displaying those estimates to our users. It consists of the following submodules: ForwardModel.stan and ForwardModel.R.

4.2.1 Forward Model Stan Model

The ForwardModel.stan submodule provides the Stan Model necessary for using the RStan package.

Dependencies

- RStan package
- External C++ header file
- Licht-cpp static library file

Use cases

- Estimate user-defined set of parameters given a set of constants.
- Modeling the parameter space of estimated values that produce predicted light curves that potentially match the observed light curve.

Design

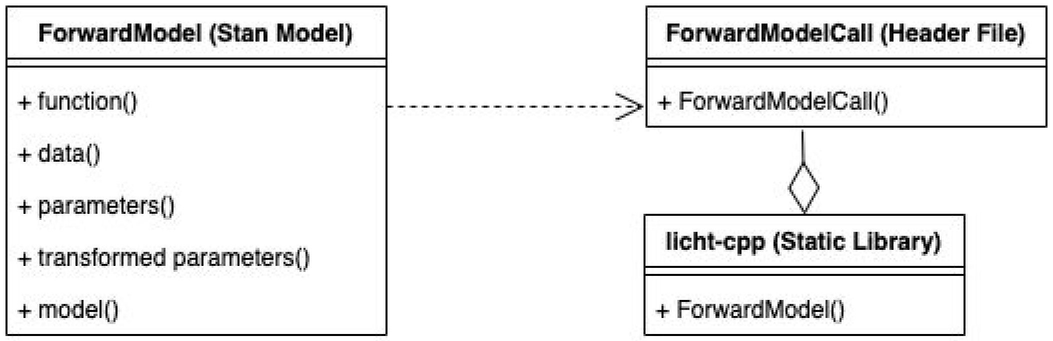
ForwardModel (Stan Model)
+ function()
+ data()
+ parameters()
+ transformed parameters()
+ model()
ForwardModelCall (Header File)
+ ForwardModelCall()
licht-cpp (Static Library)
+ ForwardModel()

Figure 4: UML diagram describing the relation of ForwardModel.stan within the HMC Module

A Stan program is organized into a sequence of named program blocks, the bodies of which consist of statements and variable declarations.

<u>Program Blocks</u>

- Functions block
    - This is where function definitions are included to be used within the user-defined functions.
- Data block
    - The data supplied to the Stan model is stored in the variables declared in this block. This is where the user declares the data types, their dimensions, any restrictions (i.e. upper = or lower =, which act as checks for Stan), and their names. Any names the user gives to their Stan program will also be the names used in other blocks.
- Parameters block
    - This is where the user indicates the parameters they want to model (estimate), their dimensions, restrictions, and name.
- Transformed parameters block
    - This is where the user includes variables to be defined in terms of data and parameters that may be used later and will be saved.

- Model block
    - Statements declared here are used to help define the model. This is where the user includes any sampling statements, including the “likelihood” (model) that they plan to use. The model block is where the user indicates any prior distributions they want to include for their parameters.

<u>Send Data to the External C++ header file</u>
The functions block will send data to an external C++ header file, which in turn calls the licht-cpp static library. This static library contains the compiled C++ files necessary to run the forward model in conjunction with the Forward Model Stan Model.

<u>Parameters</u>
Since the function block within the ForwardModel.stan file will be sending these parameters to an external C++ header file (which in turn calls the licht-cpp static library), these parameters will be required as input within the ForwardModel.stan file.

- ephemerisPrimaryFile
    - Path to Ephemeris .txt file from observer to target
- ephemerisSunFile:
    - Path to Ephemeris .txt file from observer to Sun (note that both Ephemeris files have the same columns, defined in the User Guide)
- inputTimes
    - Array of Julian dates to run the model at
- hapke
    - A vector of HapkeModel instances, one per shape
- windowX
    - Width of the render (pixels)
- windowY
    - Height of the render (pixels)
- numberSamples
    - Number of antialiasing samples
- maxDepth
    - Number of light bounces

- jamma
  - Modifier to render brightness (does not impact lightcurve)
- renderOutputFile
  - Prefix of output render files
- render
  - Whether or not to save renders to images
- antialiasing
  - Whether or not to perform antialiasing
- vFov
  - The vertical field of view (degrees)
- options
  - Which shape options are being used (0=facet, 1=sphere, 2=ellipsoid, 3=vector)
- objPath
  - Array of paths to obj files to read in
- aRadius
  - a of generated Triaxial Ellipsoid / Sphere (km)
- bRadius
  - b of generated Triaxial Ellipsoid (km)
- cRadius
  - c of generated Triaxial Ellipsoid (km)
- facets
  - Custom facets per shape, sets of 3, references vertices with +1 index
- vertices
  - Custom vertices per shape, sets of 3
- spinEpochs
  - Spin epoch per shape (hours)
- spinPoles
  - Cartesian vector per shape
- rotationPeriods
  - Rotation period per shape (hours)
- useSpinStates
  - Whether or not to use spin states
- period
  - Orbital period (days)
- semiMajorAxis
  - Semimajor axis (km)

- eccentricity
    - Eccentricity (dimensionless)
- inclination
    - Inclination relative to J2000 equatorial plane (degrees)
- meanLongitudeAtEpoch
    - Mean longitude at epoch (radians) (epsilon)
- omegaLongitude
    - Longitude of ascending node (radians) (Omega)
- pomegaLongitude
    - Longitude of periapsis, (radians) (w~)
- epoch
    - Reference Julian date (days)
- version
    - Optional descriptive string (perhaps including the date)
- massRatio
    - The percentage of mass that is in the primary (always 1)
- debug
    - Boolean to print out additional information

4.2.2 Forward Model R Script

The ForwardModel.stan submodule provides an interface necessary for fully utilizing the RStan package.

Dependencies

- RStan package
- External C++ header file
- Licht-cpp static library file

Use cases

- Provide an environment for which the Stan model can be executed, and used to display data produced from the Stan model.

Design

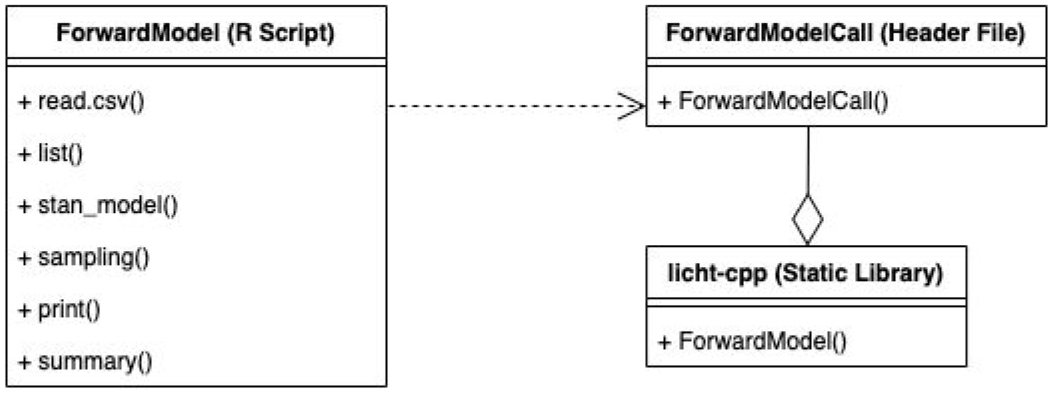

Figure 5: UML diagram describing the relation of ForwardModel.R within the HMC Module

The R script is organized into a series of function calls that allow for the import of the Stan model, its necessary external C++ header file, and the storing and sampling of data provided to the model.

Functions

- read.csv()
  - Reads in data for the Stan model into an R object from a specified .csv file.
- list()
  - Creates a list using the data supplied by the user.
- stan_model()
  - Builds an instance of the specified Stan model. The Stan code used to build the stan model may be contained within the R script, however, this is not necessary and instead may be pointed to in a separate file. An external C++ header is included to provide the implementation of the "ForwardModelCall" declared in the function block.
- sampling()
  - Samples the Stan model and stores the returned data within an R object. The number of iterations, chains and other diagnostic arguments for the Stan model are set here.

- print()
    - Prints a summary of the data contained in the R object determined by sampling(). Included in the summary is model-specific information such as the number of chains, number of total iterations, number of warmup iterations, etc. Date and time samples are drawn, and a brief explanation of n_eff and Rhat is provided.
- summary()
    - This method returns a summary containing quantiles, means, standard deviations (sd), effective sample sizes (n_eff), and split Rhats (the potential scale reduction derived from all chains after splitting each chain in half and treating the halves as chains). For the summary of all chains merged, Monte Carlo standard errors (se_mean) are also reported.

4.3 Forward Model Graphical User Interface

The Forward Model Graphical User Interface provides the user the ability to run the forward model with parameter input from a dedicated interface. Once acceptable parameters are input, the forward model will generate an estimated light curve. Users can utilize this data to compare observed data to the estimated light curve and form characteristics about binary systems.

Dependencies

- Python Version 3.8.0+
    - Needed to use Ctypes and Tkinter
- Ctypes Python Module
    - This module is used to access the shared object created from the licht-cpp API.
- Tkinter Python API
    - This API will be used to create our front end for the graphical user interface
- External C++ Shared Library
    - Shared object that will be used to call the forward model from the licht-cpp API.

- Plotly Python API
  - API that will be used to plot the data generated from the forward model call

Use Cases
- Provides an alternative method for parameter input for the forward model.
- After the forward model has run, it provides the ability to graph light curves for data analysis.

Design

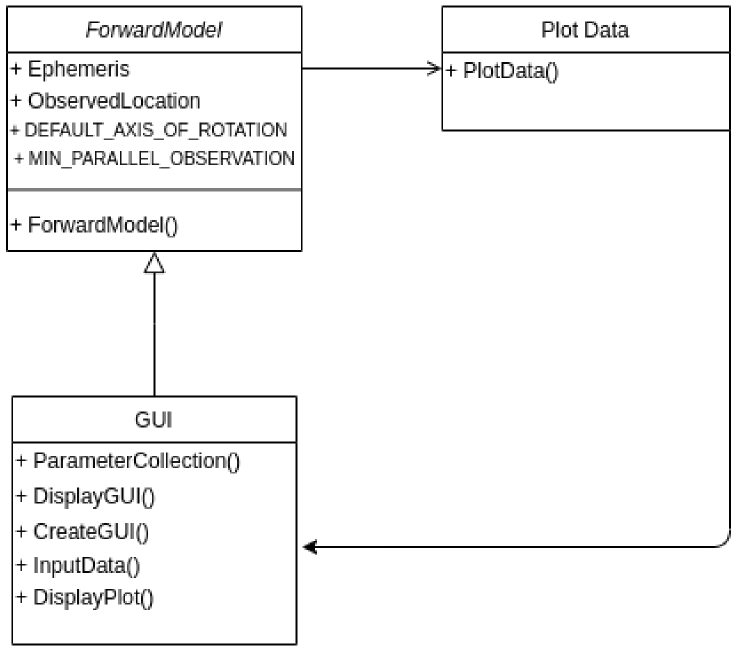

Figure 6: UML diagram of GUI Interface, Forward Model, and the Plotly API

The Forward Model Graphical User Interface module uses the functionality of the forward model. It uses 5 main functions and 1 constructor.

<u>Parameters</u>
The parameters that will be used are the same parameters used for the forward model. They will be taken in by the ParameterCollection function and sent to the ForwardModel function within the C++ Shared Library.

- ephemerisPrimaryFile
  - Path to Ephemeris .txt file from observer to target

- ephemerisSunFile
    - Path to Ephemeris .txt file from observer to Sun (note that both Ephemeris files have the same columns, defined in the User Guide)
- inputTimes
    - Array of Julian dates to run the model at
- hapke
    - A vector of HapkeModel instances, one per shape
- windowX
    - Width of the render (pixels)
- windowY
    - Height of the render (pixels)
- numberSamples
    - Number of antialiasing samples
- maxDepth
    - Number of light bounces
- gamma
    - Modifier to render brightness (does not impact lightcurve)
- renderOutputFile
    - Prefix of output render files
- render
    - Whether or not to save renders to images
- antialiasing
    - Whether or not to perform antialiasing
- vFov
    - The vertical field of view (degrees)
- options
    - Which shape options are being used (0=facet, 1=sphere, 2=ellipsoid, 3=vector)
- objPath
    - Array of paths to obj files to read in
- aRadius
    - a of generated Triaxial Ellipsoid / Sphere (km)
- bRadius
    - b of generated Triaxial Ellipsoid (km)
- cRadius
    - c of generated Triaxial Ellipsoid (km)

- facets
    - Custom facets per shape, sets of 3, references vertices with +1 index
- vertices
    - Custom vertices per shape, sets of 3
- spinEpochs
    - Spin epoch per shape (hours)
- spinPoles
    - Cartesian vector per shape
- rotationPeriods
    - Rotation period per shape (hours)
- useSpinStates
    - Whether or not to use spin states
- period
    - Orbital period (days)
- semiMajorAxis
    - Semimajor axis (km)
- eccentricity
    - Eccentricity (dimensionless)
- inclination
    - Inclination relative to J2000 equatorial plane (degrees)
- meanLongitudeAtEpoch
    - Mean longitude at epoch (radians) (epsilon)
- omegaLongitude
    - Longitude of ascending node (radians) (Omega)
- pomegaLongitude
    - Longitude of periapsis, (radians) (w~)
- epoch
    - Reference Julian date (days)
- version
    - Optional descriptive string (perhaps including the date)
- massRatio
    - The percentage of mass that is in the primary (always 1)
- debug
    - Boolean to print out additional information

Backend Functions

- ForwardModel
  - Calls the forward model from the External C++ Shared Library with the parameters from the Interface object and returns the light curve data that was generated.
- ParameterCollection
  - Takes in parameter input from the Interface and creates a dictionary with the variable name, and the data associated.
- FileInput
  - This function allows the user to input a observed light curve data set from an external file to help compare the observed data to the predicted data.
- PlotData
  - This function plots the data from either the FileInput function or the ParameterCollection function. It takes in the data set and creates a light curve through the Plotly API.

Frontend Functions

- DisplayGUI
  - This function will be called via the command line and will display the graphical user interface.
- DisplayPlot
  - This function will take in the light curve graph that is generated by Plotly.

Constructor

- createGUI
  - This constructor is used to create the graphical user interface object. The reason the interface must be an object is so that data can be passed to the object parameter fields.

4.4 Movie Generator

The movie generator will allow users to stitch together .jpeg files into a .mp4 file through the use of the OpenCV library. The movie generator will allow users to look at how the predicted system behaves and moves within space.

Dependencies

- Existing C++ Shared Library
- OpenCV C++ Library

Use Cases

- Allows the user to create movies of the images produced from the existing API.

Design

The movie generator will be built alongside the existing API. It will be a submodule that gathers the images to be sown together for the movie. It will take no parameters but will be generated by 1 main method and 1 constructor.

<u>Functions</u>

- gatherImages
    - This function will be used to sow together the selected images to generate a movie.

<u>Constructor</u>

- createVideo
    - This constructor will take in the images from gatherImages and use the OpenCV functionality to stitch the images together. This will return a .mp4 in the output files.

5. Implementation Plan

It is imperative that we section our project into reasonable milestones. The project has been strategically split up to break down triaxial ellipsoids, the HMC, the GUI, and the movie generator into realistic, specific goals. These goals will construct a plan for development that encompasses the rest of the semester. We have had active scheduling and task assignments in order to make sure the implementation of our solution is successful.

In the fall, our team broke down the project into four major sections. We considered our interests and experience to choose among working on triaxial ellipsoids, the HMC, the GUI, and the movie generator accordingly. Jes and John are assigned to work together on the triaxial ellipsoid problem because of their experience with mathematics and their interest in ray tracing. Batai and Matt are assigned to work on the HMC problem because of their experience with C and C++, and they strive to challenge themselves by learning new topics. Moreover, they are interested in the application of statistical computation and modeling-based APIs. Brad is assigned to the GUI and movie generator because of his passion for creating GUIs, working with Python, and learning new frameworks.

The Gantt chart in Figure 6 makes use of colored bars to illustrate which tasks have been assigned to which team member. The purple tasks are assigned to John and Jessica, the blue tasks are assigned to Batai and Matt, and orange and green tasks are assigned to Brad.

As seen in the Gantt chart, the major phase titled “Alpha Prototype” is a major milestone upcoming programming task and once all of the major features are implemented, this large task is to be broken down and assigned to team members.

After the Alpha Prototype Demo has been completed, the next step will be to optimize our solution and implement the video generator as a stretch goal. After the core features of our additions have been implemented, our time will be dedicated to enhancing the features of our implementations by optimizing the code. This optimization involves low-level changes, code organization and commenting, and parallelization for certain sections of the API.

6. Conclusion

Space is an exciting, mysterious field. Despite thousands of years of research, humankind has only begun to scratch the surface of understanding the universe. To this day, there are numerous objects floating in the Kuiper Belt that we have yet to explore. Up to 30% of those asteroids are considered to have at least one secondary in their orbit. Studying these binary systems leads way to important answers such as how they are formed, how their orbits work, and how they fit into the solar system.

Our clients Dr. Audrey Thirouin and Dr. Will Grundy work at Lowell Observatory and observe these binary systems in the Kuiper Belt, which they accomplish via software that models binary systems. Since space voyages are time-consuming and costly, telescopic observations from Earth are the most efficient way to test hypotheses of binary systems. Although our clients can prepare observed light curves, it can be difficult to form characteristics of asteroids without a model. Our project is aimed to help Will Grundy and Audrey Thouirin at Lowell Observatory solidify these attributes.

The Hamiltonian Monte Carlo algorithm we are implementing will allow for computed estimations of parameters for observed binary systems. The addition of the triaxial ellipsoid shape will accelerate render speeds for visualization without a high cost of accuracy lost. The GUI will streamline the data input process for users, easing their use of the system. Finally, the video generator will compile a movie for users to use at will.

By formalizing our conceptual design, we hope to instill confidence in our clients. This blueprint is a guide on how we plan to implement our project and meet our requirements. This mitigates the risk of running into issues during development. With this extensive planning, there are no foreseeable roadblocks and the independently testable modules ensure a robust solution. Team Andromeda is excited to implement this design and create an efficient, intuitive solution that exceeds the clients’ needs.

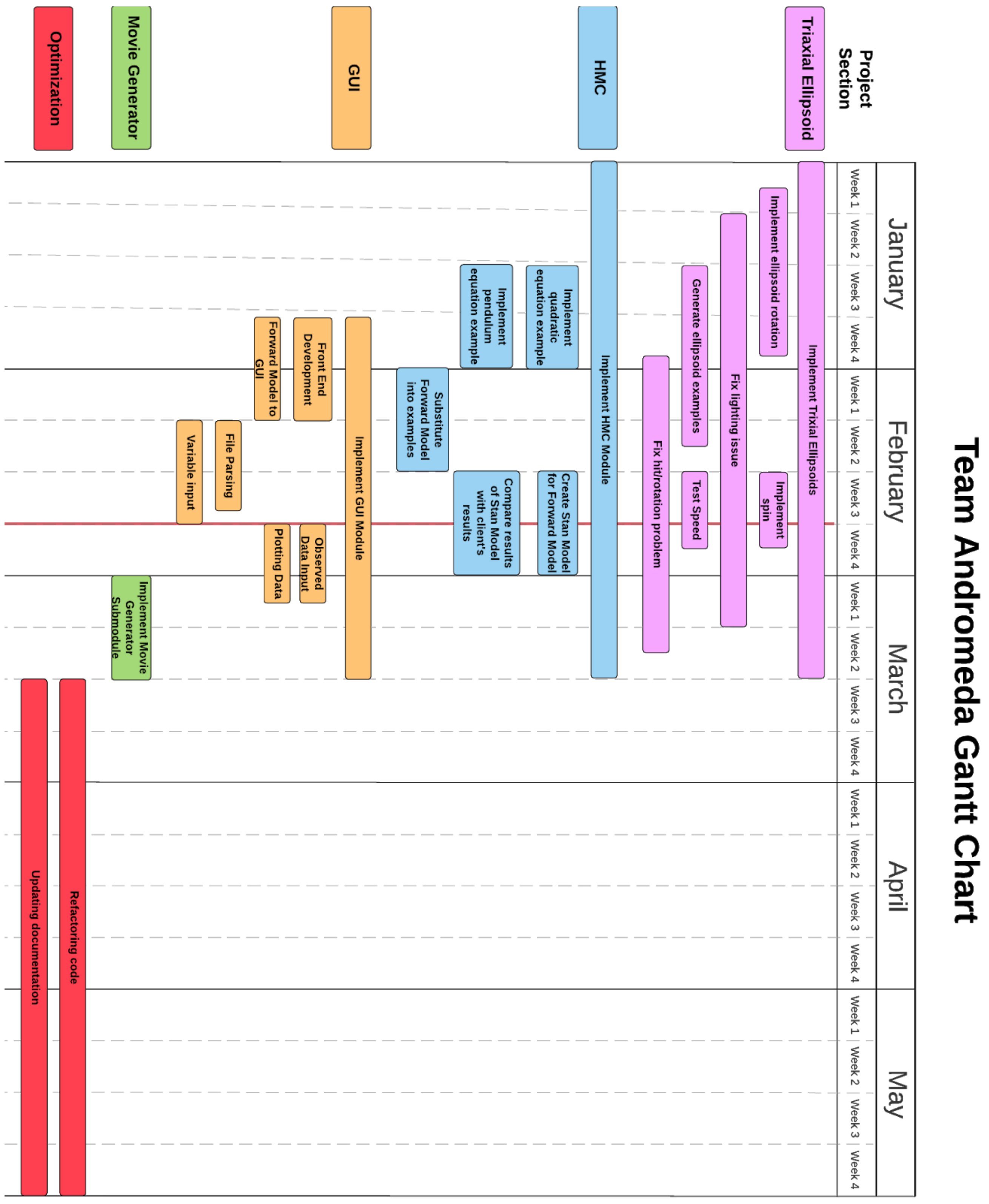

Figure 7: Gantt chart as of 2/21/2020