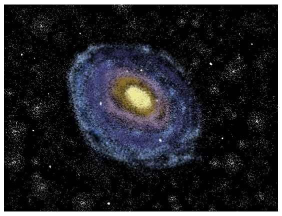

Team Andromeda

Final Report

May 7, 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

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.

The Solar System is poorly understood. Although we as humans have obtained a large amount of knowledge about it, there is always more to learn. To better understand the formation of our Solar System, astronomers look towards its edge. 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. **To better understand our Solar System, we must better understand the Kuiper Belt**. The Kuiper Belt contains 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. Astronomers and scientists 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. To do this, they need to use special techniques which make the most use of the data available.

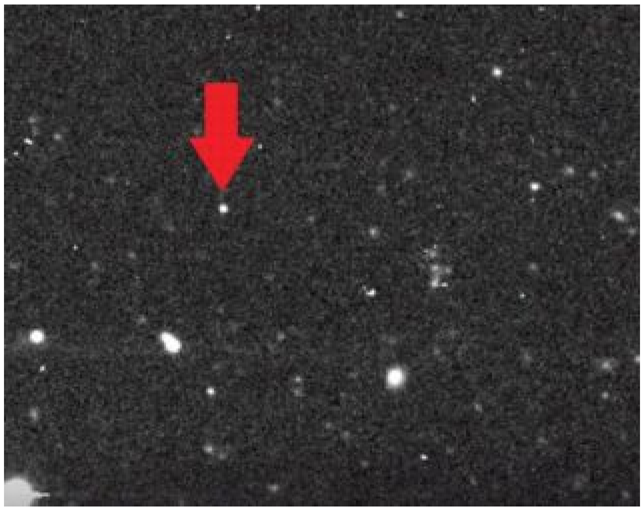

Figure 1: A binary asteroid system just beyond Neptune. Given that each asteroid in this system is just over 100 km in radius, even a 4 meter telescope could not discern any details.

These techniques are needed because objects in the Kuiper Belt are so far away and relatively small that they can only be **observed as a pinpoint of light**, as shown in Figure 1. The main question that led to this project was, "**how can we understand what we cannot see?**". The answer is, of course, software.

Specifically, our clients use a software that takes in a variety of binary system parameters and can **predict how this brightness fluctuates** over time. A user can then solve for unknown parameters. They can do this by comparing the prediction of the software against what was seen in the telescope, then continuously adjust the input parameters until the predicted light curve matches the observed light curve.

Light curves can be used to infer the characteristic information 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.

Though our clients currently have a software, which we call the forward model, capable of generating predicted light curves, their software isn't perfect. At times it can be clunky and tedious to use, which hinders our clients' workflow.

The purpose of this project is to improve and streamline the current forward model. These improvements were achieved through the addition of:

- Triaxial ellipsoid shapes
- A nonlinear minimization algorithm (NLM)
- A graphical user interface (GUI)
- A video generator

These implementations were able to solve a variety of issues that the clients faced. The ellipsoid shape helps provide more a more realistic shape to model with while retaining a fast run time, the NLM algorithm streamlines the continuous adjust of input parameters while trying to match the predicted and observed light curves, the GUI streamlines the process of inputting parameters into the forward model, and the video generator provides a quicker way to combine the series of rendered images output by the forward model. This increase in speed, accuracy, and ease-of-use will advance Lowell's research efforts to analyze binary systems in the Kuiper Belt.

2. Process Overview

The process of developing the code base was challenging due to the high complexity of existing code base and the task at hand. Due to the unique nature of the code base, we followed the instructions given to us by the previous development team, to ensure that the level of quality from the existing code base was used when developing our new functionality for the library.

2.1 Summary

The entire process centered around two main principles: **communication** and **version control**. These two principles were key to our success as a team and due to strict guidelines outlined in our team standards we were able to efficiently navigate through the development process.

Communication

For this project to succeed, our main focus was based on communication due to the size of our team and the complexity of the project. As a team we constantly were communicating with one another via **email** or **Discord**. With this constant communication between team members, we were able to complete our work efficiently. Due to our process being split into three different functionalities, it was paramount that our team be in constant communication so that our different functionalities would integrate with one another.

As for communication between our team and our clients at Lowell Observatory, Batai was our Team Communicator and was in charge of all communication. As our team communicator he was in charge of scheduling all meetings and sending our any questions our team had and sending out our meeting times.

Version Control

For consistent changes and debugging, version control was one of our most important principles. Due to an existing code base we "forked" their existing code base from their **Github** repository and added all additional code to our own fork. As a team we would all commit any changes we made throughout the week to keep the code base as up to date as possible. This allowed for continuous debugging between functionalities. Much of the existing code base was not touched and code was only added throughout the development process. As a team we developed within separate branches, and committed our changes to our respective branches. We had four major branches; **GUITester**, **NLM**, **Ellipsoid**, and **Master**. Once we got to the end of the development process, we merged our branches to master to have all new code added to the existing code base.

2.2 Coding

Every team member was a **coder**, but each member was working on their designated functionality. Jessica and John worked on **Triaxial Ellipsoids** throughout the whole process and developed all of this code. Matthew and Batai worked on the **NLM** functionality throughout the whole process and developed this functionality to work with the existing function of the forward model. Finally, Bradley Donn worked on the **GUI for the forward model** throughout the whole process and developed both the frontend and backend code that calls the existing code base.

As the team lead, John would oversee all code that was being developed and was in charge of assigning tasks for each functionality. Each team member was expected to complete their tasks as they were assigned and by a set time for review or editing purposes.

2.3 Meetings

For the entirety of our development process, our team would meet twice a week for one to two hours depending on upcoming due dates. At these team meetings each member would have to go over what has been completed since the last time that our team met. This was a major component of our success, it allowed for all team members to stay informed with each section of the project. Additionally, during these meetings we would go over task allocation, and upcoming due dates. This created a structure within our team that held each individual teammate accountable for their parts of each portion of this development process.

For meeting with our clients at Lowell Observatory, we would meet once a week. This allowed for constant communication between the team and the clients for any problems that arose. This was also a great opportunity for our team to ask any questions that needed clarification. Commonly, these meetings would last anywhere from one hour to two hours.

3. Requirements

3.1 Functional Requirements

There are three major use cases that needed to be satisfied for this project. These cases included the use of a triaxial ellipsoid object to model and calculate light curves with, the use of a NLM algorithm to produce likely parameters for an observed light curve, and the use of a GUI to pass parameters into the forward model and graph the predicted light curve. This was achieved through the implementation of triaxial ellipsoid shapes, a NLM algorithm, and a GUI respectively.

These features will be used to improve the current forward model by allowing for a greater ease of use for our clients, and are broken down in the following sections.

FR1. Simulate Triaxial Ellipsoids

The software needed to be capable of performing calculations for the forward model using triaxial ellipsoids. It also needed to be able to render triaxial ellipsoid objects when calling the forward model, if renders are desired. This output was generated through the use of the currently implemented ray tracing feature. Additionally, this output needed to be compatible with modifications that the user may request through input parameters. These adjustments to the output included the ability to adjust the resolution or pixel count of the image. The requirements of simulation and calculations of triaxial ellipsoids can be broken down into four sections:

Calculate light curve using triaxial ellipsoids
The API takes in a given amount of parameters to calculate the light curve for a system, including:

- Ephemeris table
- Keplerian orbital elements
- Time value(s)
- Object shape parameters
- Spin states
- Hapke parameters
- Optional accuracy setting

The software needed to be capable of performing these calculations for the forward model using triaxial ellipsoids. All of these parameters work in conjunction to calculate the forward model output and provide a light curve of a system. The triaxial ellipsoid object needed to be compatible with all the above parameters in order to calculate a light curve.

Render triaxial ellipsoids as objects
Rendering images of the system during the calculation of a light curve is also possible. The software can render a triaxial ellipsoid object at the request of the user through the forward model. Since the ellipsoid object is used by the forward model, its render is generated through the use of the ray tracing feature.

The software simulates objects, meaning this was also required of triaxial ellipsoids. Rendering triaxial ellipsoids allows users to visualize how the objects are interacting with each other. Ray tracing is the method currently used to render objects. The forward model ray tracer uses a hit-or-miss calculation to determine whether a ray hits, finds the angle of reflectance of that ray, uses those angles with the Hapke bidirectional reflectance model, and gathers that information to save it as a relative luminosity. This is later corrected for the image. Triaxial ellipsoids needed to be compatible with this process to be renderable.

Allow for manipulation of dimensions
Triaxial ellipsoids describe a large group of shapes. A triaxial ellipsoid can be thought of as spheres that have been shortened or lengthened along their x-axis, y-axis, and/or z-axis in a 3D space. Being able to modify the value of triaxial ellipsoids’ axes was key to implementing these shapes and allows the user to

gain the most from them. These parameters were made clear to users so that they can manipulate the dimensions of the ellipsoid.

Rotate object

The sphere class that was previously implemented did not have to consider rotation, as a sphere is uniform throughout. Triaxial ellipsoids are not uniform throughout and, therefore, needed to consider the possible fluctuations due to rotation. Orientation and rotation functions were implemented for the shape to account for these fluctuations and allowed the model to accurately represent the shape.

The rotation was the most difficult part of the shape to implement due to the math that was needed to calculate it. This was accomplished through matrix multiplication that updates the ellipsoid object depending on what orientation is required.

FR2. Implementation of a NLM

A new key requirement of this project is the implementation of an NLM module. The NLM module serves the purpose of providing sets of estimated parameters for our clients. Additionally, the module displays the results of the minimization to the command prompt. Lastly, this module saves the generated results to an external file. The requirements of the module are explored in further detail in the following sections.

Produce estimated parameters for predicted light curves

The main functionality of the NLM module estimates parameters for predicted light curves. Initially, our clients choose which forward model parameters they want to estimate by supplying their values to the NLM module. This module then provides a multi-dimensional minimization of a chi-square function using these estimated parameters. This produces the minimum chi-square comparison of predicted light curves produced by the forward model against observed light curves. Consequently, this determines the estimated values of our client’s chosen parameters with their corresponding chi-square value.

Display minimization results on the command line

The NLM displays results to the command line. These results show the starting values for estimated parameters, the number of function evaluations that occurred and the final values for each of the parameters the module arrived at. Additionally, data plots for the observed light curve and predicted light curves generate during the execution of the NLM module and display after the completion of the minimization.

Save minimization results to an external file

Our clients would also like for the NLM module to produce a file, such as a .txt file, that would contain final optimized parameters produced by the NLM module.

FR3. Implementation of a GUI

A key requirement that we have is the implementation of a new GUI for the forward model. This GUI will make the forward model more user friendly and allow the user to enter parameters from the interface rather than entering them at the command line. Additionally, it will allow the user to see the predicted light curve directly on the screen where the user will be able to compare the predicted light curve to the observed light curve.

Input to forward model

The GUI must allow the user to enter in variables from the interface. There will be text boxes that will take in a specific type of input, and if the input type is invalid, the GUI will prompt the user to put in an appropriate input. Once all variables are entered, there will be a button that will allow the user to run the forward model with the input parameters.

Due to the large number of parameters to be input for the forward model, there will be a scroll wheel that will allow the user to navigate through the parameter list faster.

Output to forward model

Once the software has acceptable parameters, it will enter those parameters into the forward model and generate a light curve that will be displayed to the screen. Once the light curve is generated to the GUI the user will be able to save the output as a singular image locally on their device.

There will be a tab at the top of the output panel that will allow the user to switch between the generated light curve and the rendered images from the forward model.

Settings

The GUI will start with a standard default for setting configurations. For the settings tab, the user will be able to select settings and, depending on what that setting should do, it will dictate variable input on the GUI for the variables that will not be used, or set those variables to a predetermined constant. There will be multiple settings that will be implemented allowing the user more flexibility in variable input while also constraining those variables that will not be used.

FR4. Generate Video From Forward Model

Our clients want to make the process of generating videos from their images more streamlined. Currently, the program will create images of the model, but does not produce a video. Our clients would like to generate a video of the images that the model would produce. The video generation is broken down into three main requirements below.

Output File

When the video generation software is run, it will take in a collection of images and condense them down to an MP4 file. The video generation software will be accessible from the forward model GUI, and the file location will be put into a text box. From there, the images will be converted to an MP4 file at the press of the “Compile Button”.

Camera Control

We also wish to give the user some control over the pointing of the “camera” in video production. Currently, images are generated while centered on a specified body. This added control would allow the user to choose which body should be the center of the video, and how wide the field of view should be. The user will also be able to control the start, end time, and the time step of the produced video.

Image Collection

As the API renders objects, it will save images of the binary system model into a folder. The images will be a visual representation of the binary system as the individual objects complete their orbit. These images will be rendered from the forward model while it is running, and then later used by the video generator to create a video.

3.2 Performance Requirements

The forward model had room for growth which can help streamline the overall process of its use. Through our additions, our clients have a more efficient workflow when using the forward model. We were conscious of the user, especially with requirements such as usability, and understand how the program is expected to perform. The following sections break down and detail the core requirements needed to provide a high performing software for the clients.

PR1. NLM

The runtime of the NLM module depends on several factors determined by the user. These include the number of parameters they want to estimate, the number of iterations they want the NLM Module to execute, and the number of starting points they supply. Therefore, a minimum runtime cannot be determined due to **runtime depending on user inputs.** The user must be aware of what they are looking for and consider this when utilizing the NLM module.

PR2. GUI

The GUI was implemented to help simplify the use of the forward model. The **design is simplistic** and **does not require much learning from the user**. Input for each parameter was labeled so that the user knows where each parameter belongs. As numerous parameters are needed to use the forward model, users are provided with an explanation of how each parameter influences the forward model within the provided documentation. This is provided in the user guide so that there is an easily accessible reference.

PR3. General Usability and Readability

The functional requirements implemented within our solution are supported with extensive documentation. Explanations for all the functions and their parameters that we develop are provided. Additionally, tutorials of how to properly utilize ellipsoids, the GUI, and the NLM are given to ensure a low learning curve for our clients. The documentation we supply enables any user, as well as future developers, with the information necessary to utilize this application.

3.3 Environmental Requirements

ER1. Linux Systems

The solution needed to be able to be **compiled and run on Linux systems**. It does not need to be able to run on any other type of operating system. This requirement was met by the previous development team, in which they created a cross-platform solution for *Linux*, *macOS*, and *Windows*. Please refer to *Appendix A* for details.

4. Architecture and Implementation

In an effort to retain modularity, the overall design for the new feature implementations involved creating 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. For a further breakdown of the forward model and its modules and submodules, please refer to Paired Planet's final report.

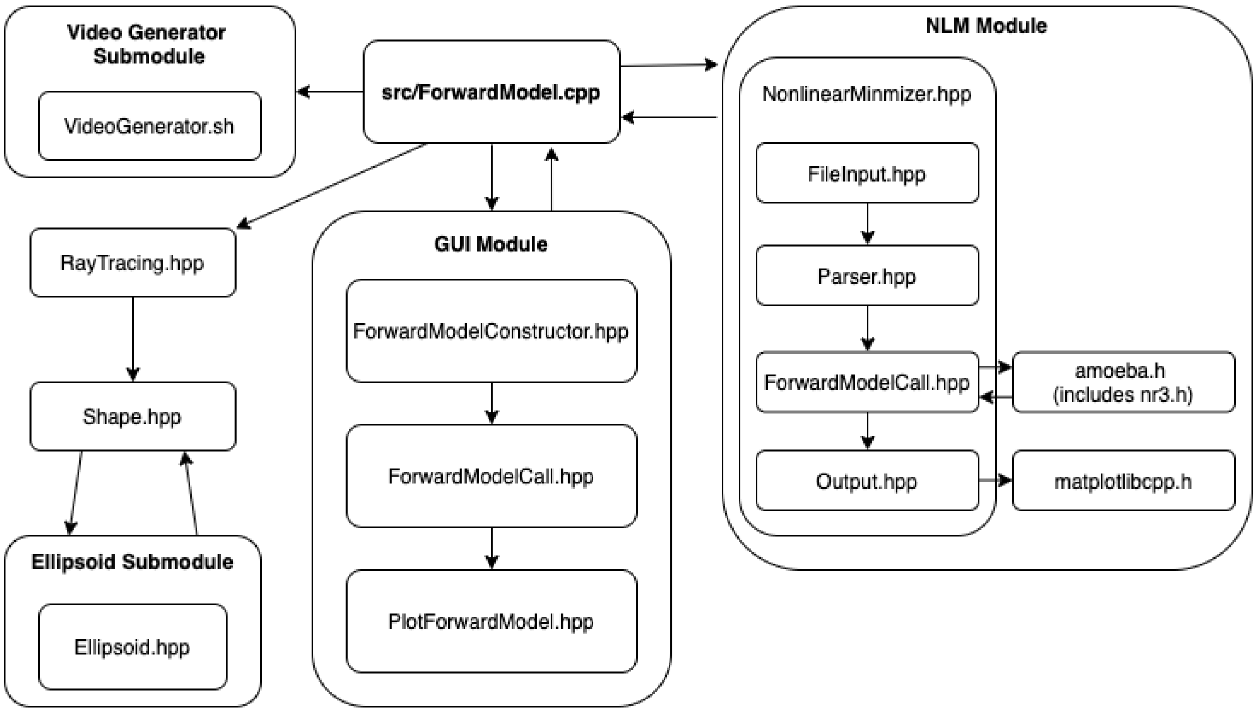

Figure 2: How the forward model and its new features are divided into conceptual modules

4.1 Triaxial Ellipsoids

The Ellipsoid submodule is the third and final extension of the Shape module. Like spheres and faceted objects, 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 faceted objects, this class allows for ellipsoid generation at a much faster rate than a faceted shape.

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 and its light curve are then rendered and calculated via the ray tracer.

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 adjustments.

Constructors

- The ellipsoid is created using the given radii and Hapke
- The ellipsoid is created using the given radii, pole, and Hapke
- The ellipsoid is created using the given center, radii, and Hapke
- The ellipsoid is created using the given center, radius, and pole, Hapke

Methods

- Orient
  - Orient aligns the ellipsoid along a specified pole.
- Spin
  - Spin rotates the ellipsoid on the z-axis to simulate its rotation, using rotation speed, epoch, and time. Since the z-axis is the axis that the ellipsoid rotates about, it is the user's discretion to make this the shortest axis upon ellipsoid creation.
- SetCenter

    - SetCenter uses a given center to set the ellipsoid's center, which dictates where the object is placed.
- SetScale
    - SetScale uses the original radii given, calculates scale based off of the given scaling factor, and saves this information into radiusX, radiusY, and radiusZ. SetScale also sets the matrix with the new radii.
- Hit
    - Hit calculates the point of origin of the ellipsoid using the ray direction and the ellipsoid's matrix. Hit then sends this information to hitRecord which calculates whether the object is hittable.

Variables
- radiusX
    - radiusX represents the ellipsoid's scaled x-axis length.
- radiusY
    - radiusY represents the ellipsoid's scaled y-axis length.
- radiusZ
    - radiusZ represents the ellipsoid's scaled z-axis length.
- originalRadiusX
    - originalRadiusX represents the ellipsoid's original x-axis length.
- originalRadiusY
    - originalRadiusY represents the ellipsoid's original y-axis length.
- originalRadiusZ
    - originalRadiusZ represents the ellipsoid's original z-axis length.
- Eigen::Matrix3d M
    - M is the matrix that transforms the sphere into an ellipsoid through lens manipulation.
- Vector3d pole
    - pole gives a direction for the ellipsoid to orient to and rotate on.

4.2 NLM

The NLM module provides the required tools for **computing parameter estimates and displaying those estimates to the user**. It consists of the following submodules: NonlinearMinimzer.hpp, FileInput.hpp, Parser.hpp, ForwardModelCall.hpp, and Output.hpp. It also utilizes the following libraries to act as the backbone to the NLM algorithm: amoeba.h, and nr3.h. For more information on these libraries, please refer to *Numerical Recipes 3rd Edition: The*

Art of Scientific Computing[1]. A PDF copy of this book can be found in the licht-cpp/nlm/doc/ directory of the licht-cpp API. Moreover, this module also utilizes the matplotlibcpp.h file for plotting. Links to the associated GitHub repository[2] and documentation[3] can be found in the footer of this page.

4.2.1 NonlinearMinimizer.hpp

The NonlinearMinimizer.hpp submodule provides the NLM module with the necessary header files to interface with the forward model and the libraries that provide the functionality necessary to perform minimization.

Dependencies

- forward model Header Files:
    - Ray.hpp
    - Math.hpp
    - View.hpp
    - Error.hpp
    - Euler.hpp
    - Facet.hpp
    - Tracer.hpp
    - Orbit.hpp
    - Shape.hpp
    - Types.hpp
    - Camera.hpp
    - Sphere.hpp
    - Calendar.hpp
    - Ellipsoid.hpp
    - Ephemeris.hpp
    - HitRecord.hpp
    - Locations.hpp
    - HapkeModel.hpp
    - ForwardModel.hpp
    - KeplerTrueAnomaly.hpp
- Plotting:
    - Matplotlibcpp.h
- Minimization Library
    - nr3.h

[1] http://numerical.recipes/
[2] https://github.com/lava/matplotlib-cpp
[3] https://readthedocs.org/projects/matplotlib-cpp/downloads/pdf/latest/

    - Amoeba.h
- NLM Files
    - ForwardModelCall.hpp
    - Parser.hpp
    - FileInput.hpp
    - Output.hpp

Use cases

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

4.2.2 FileInput.hpp

The FileInput.hpp submodule inputs data from the user input file as well as the observed file supplied by the user. While reading in these files, it assigns the data to different vectors that are used in later submodules.

Dependencies

- NonlinearMinimizer.hpp

Use cases

- Pass in the user input file, as well as the associated vectors, and fill the vectors with the information from the user input file.
- Pass in the observed data file from the user, as well as the associated vectors, and fill the vectors with the information from the observed data file.

Functions

- readUserInput(string& filePath, int& counterFit, vector<int>& id, vector<string>& parameter, vector<char>& fit, vector<string>& value, vector<double>& stepSize);
- readObservedData(string& filePath, vector<vector<double>>& julianDate, vector<vector<double>>& magnitude, vector<vector<double>>& errorBar, vector<vector<char>>& filter);

4.2.3 Parser.hpp

The Parser.hpp submodule assigns data taken from the user input file to its corresponding parameter within the forward model argument structure.

Dependencies

- NonlinearMinimizer.hpp

Use cases

- Pass parsed data necessary to run the forward model from the user input file to the forward model argument structure.
- Pass flags from the user input file to specify whether output to the terminal, a data file or a data plot should occur.
- Pass inputs for the operation of the minimizer, such as the tolerance level, to be stored in the forward model argument structure.

Functions

- bool isEqual(const string& a, const string& b)
- FMArgumentParser(FMArgumentStruct& FMArguments, VecDoub_I& estimatedParamsArray, vector<string>& value, vector<char>& fit, vector<int>& id)

4.2.4 ForwardModelCall.hpp

The ForwardModelCall.hpp submodule provides an interface necessary for fully utilizing the forward model.

Dependencies

- NonlinearMinimizer.hpp

Use cases

- Pass in the FMArguments variable and assign the parameters within the FMCall function to the values in the FMArguments struct. Pass those parameters to the forward model.
- Pass in the julianDate vector and assign the "inputTimes" variable to the times in the julianDate vector.

Functions

- vector<Array> FMCall(FMArgumentStruct FMArguments, vector<vector<double> > &julianDate)
- Array inputTimesFiller(vector<vector<double> > &julianDate)

Parameters

Since the function within the ForwardModelCall.hpp file is sending these parameters to the forward model via a licht-cpp.a static library, these parameters are required within the ForwardModelCall.hpp 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

4.2.5 Output.hpp

The Output.hpp submodule provides the functionality necessary to produce NLM output to a terminal, text file and images of graphs displaying predicted and observed light curves.

Dependencies

- NonlinearMinimizer.hpp

Use cases

- Print the values of the chosen estimated parameters at the current step in the minimization process to the users terminal.
- Save produced estimated light curves to a text file.
- Plot data points of predicted and observed light curves to a graph saved as .png files.

Functions

- void terminalOutput(vector<int> positionsOfFitParams, VecDoub_I estimatedParamsArray, vector<string> parameter, int chiSquare)
- void fileOutput(vector<vector<double> > splitPredictedLCs, vector<vector<double> > julianDate, vector<vector<char> > filter)
- void plotOutput(vector<vector<double> > splitPredictedLCs, vector<vector<double> > julianDate, vector<vector<double> > magnitude, vector<vector<double> > errorBar)

4.3 Forward Model GUI

The forward model GUI 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 generates 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

- Qmake 3.0+
    - To compile the forward model Graphical User the user must have Qmake 3.0 or higher.
- Qt5.0+

    - To use the forward model GUI at it’s full capability, the user must have Qt5.0 or greater. All of the latest linux distributions are released with Qt5.0 or greater. If the user is using an older linux distribution, you can download QT5.0+ from the QT Online Installer where the binary files are available.

Use Cases

- Provides an alternative method for parameter input for the ForwardModel.
- Graphical Analysis of predicted light curves generated from the Forward model against Observed Light Curves

The forward model GUI module uses the functionality of the forward model. It uses 5 main functions and 1 constructor.

Parameters

The parameters that are used are the same parameters used for the forward model. They are 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.
- hapke
    - A vector of HapkeModel instances, one per shape.
- windowX
    - Width of the render (pixels) used during forward model call.
- windowY
    - Height of the render (pixels) used during forward model call.
- 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 needed for forward model call.
- render
    - Boolean value that defines whether or not to save renders to images.
- antialiasing
    - Boolean value that defines 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)
- 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.
- SaveParameters
  - Takes in parameter input from the Interface and creates a dictionary with the variable name, and the data associated.
- LoadParameters
  - Loads in all parameters used last within the forward model. This allows for continuity when testing parameter sets.
- 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.

4.4 Video Generator

The video generator bash script allows our users to **stitch together .jpeg files into a .mp4 file** through the use of the FFMpeg library. The videos created allows users to look at how the predicted system behaves and moves within space. The

.jpeg files that are converted can be found in the “renders' folder in the build directory in either the licht-cpp folder or the NLM folder.

Dependencies

- licht-cpp API
- FFMpeg library

Use Cases

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

Design

When the bash script is called by the user, they are given the option to produce a video from the renders in either the licht-cpp or NLM renders folder. They are also given the option to have the video be produced at default speed or slowed down. Lastly, they are given the option to delete the images in the renders folder.

Commands

- Create a video using the FFMpeg command. Here is an example of the command:
    - ffmpeg -i "$userInputPrefix"_Primary_%03d.jpeg -vcodec mjpeg ForwardModelTestPrimary.mp4
- Create a slow down version of the video using FFMpeg command. Here is an example of the command
    - ffmpeg -i "$userInputPrefix"_Primary_%03d.jpeg -vcodec mjpeg -filter:v "setpts=2.0*PTS" ForwardModelTestPrimarySlow.mp4

5. Testing

With all of the calculations and interactions in the code base, we formed a sophisticated testing strategy. Generally, the forward model uses Google’s C/C++ testing framework named GoogleTest. GoogleTest allows us to essentially write functions that call the code and assert that the proper ‘thing’ happens (e.g. throwing a certain exception or returning an extremely precise result). As for specific sections, triaxial ellipsoids, NLM, and the GUI were thoroughly evaluated by unit, integration and usability testing.

5.1 Unit Testing

Unit testing is a practice in which individual functions of software are tested on their performance. Most of the unit tests are simple input and output verification, such as testing if the GUI accepts only valid inputs and verifying that results from the NLM are within an expected range. Other unit tests check if the creation and rotation of triaxial ellipsoids are valid.

To limit the expected output, the team utilized the same testing data used by the previous development team. These data are in the form of ephemeris files, object files, and pre-existing observed data. To standardize unit testing, Google Test was the chosen unit testing library. It is a well-known, cross-platform testing structure, has a respected library, and collaborates with the C++ API codebase.

5.1.1 Triaxial Ellipsoids

To ensure that the addition of triaxial ellipsoids is fully operational, it was paramount to create applicable unit tests. These unit tests were written in the TestShape.cpp and TestRayTracing.cpp files, located in the test folder among other, previously written tests. The goal of these tests was to check if coordinates are set correctly and that the ellipsoid shape could be rendered.

To validate coordinates, there is a test in TestShape.cpp that compares the center and radii of the generated ellipsoid.

Test Shape

1. TestShape Test
    a. **Description:** Create and verify ellipsoid object coordinates
    b. **Main Flow:**
        i. Create an ellipsoid object
        ii. Check if the center of the ellipsoid is still at the origin
        iii. Set values to the center of the object
        iv. Compare center of ellipsoid for expected values
            1. Example Input: shape.SetCenter(Vector3d(1,2,3));
            2. Expected Output: shape.center[0] = 1, shape.center[1] = 2, shape.center [2] = 3
                a. Expected output was predetermined by the team
        v. Check if radii match
        vi. Set new radii values for the object
        vii. Compare radii of ellipsoid for expected values

            1. Example Input: shape.radiusX = 2, shape.radiusY = 3, shape.radiusZ = 4;
            2. Expected Output: shape.radiusX = 2, shape.radiusY = 3, shape.radiusZ = 4
                a. Expected output was predetermined by the team
    c. **Expected Outcome:** Creation of valid ellipsoid object

In addition to testing coordinates, the following test, located in TestRayTracing.cpp, consequently ensures that the ray tracer is able to trace an ellipsoid from different rotations.

Ray Tracing

1. RayTracing Test
    a. **Description:** Creates a singleton render of an ellipsoid by creating an ellipsoid object
    b. **Main Flow:**
        i. Create a shape vector to hold the ellipsoids
        ii. Initialize ellipsoids
            1. Initialize the first ellipsoid with an arbitrary radius, a pole vector with all zero-axes, and the Hapke model
            2. Initialize subsequent ellipsoids with a radius matching the first ellipsoid, a pole vector correlating with unit circle values for rotation, and the Hapke model
        iii. Create vectors for light source and camera location
        iv. Set up the tracer
            1. Expected Output: tracer > 0
    c. **Expected Outcome:** Tracer accurately hits and luminates ellipsoid

5.1.2 NLM

To guarantee that the implementation of the NLM module was performing as expected, a series of unit tests using the Google Test framework was employed. These unit tests are contained in their corresponding tester files stored in the test folder. The unit tests for the NLM module verified that any data passed in and returned via text file are accurate. Additionally, they ensured that the comparison of light curves and estimation of chosen forward model parameters are being performed correctly.

File Input

The NLM module depends on two functions that handle reading input from the user. This input is in the form of an observed data file and a user input file. The observed data file contains observed light curve data. The format used within the file is ordered in the following manner: the first column is the Julian date, the second is the magnitude, the third is the error bar of the magnitude and the fourth is the filter used. Subsequently, the user input file contains the values that the clients are interested in putting into NLM. The format used within the file is ordered in the following manner: the first column is the ID number, the second is the parameter name, the third is the whether the parameter is being fitted, the fourth is the initial starting value, and the fifth is the step size used within the NLM.

1. CorruptedInputFile Test
    a. **Description:** Use a format that is incorrect and verify an exception is thrown
    b. **Main Flow:** User enters a file that does not follow the required format
    c. **Expected Outcome:** Error message reporting the user entered a file with an incorrect format
    d. **Alternative Flow:** User enters a file that does not include the required information
    e. **Expected Outcome:** Error message reporting the user entered a file that is missing required information

2. BadFileNames Test
    a. **Description:** Pass in incorrect file path strings and verify a exception is thrown
    b. **Main Flow:** User enters a file that does not exist
    c. **Expected Outcome:** Error message reporting the user entered a file that does not exist
    d. **Alternative Flow:** User enters a file with invalid extension
    e. **Expected Outcome:** Error message reporting the user entered a file with an incorrect extension

3. ReadFile(observed data) Test
    a. **Description:** Pass in observed data text file and verify that the information was correctly read in
    b. **Main Flow:**

        i. Enter an observed data text file that contains tester data
        ii. Parse file using the read file function for observed data
        iii. Store the data to julianData, magnitude, errorBar and filter vectors
    c. **Expected Outcome:** Data in vectors is correct based on the observed data text file

4. ReadFile(user input) Test
    a. **Description:** Pass in user input text file and verify that the information was correctly read in
    b. **Main Flow:**
        i. Enter a user input text file that contains tester data
        ii. Parse file using the read file function for user input
        iii. Store the data to id, parameterName, fittedParameter, startingValue, and stepSize vectors
    c. **Expected Outcome:** Data in vectors is correct based on user input text file

Parser

The NLM module depends on a parser function in order to appropriately parse the data gathered from the user input file. This information is then stored to a forward model arguments data structure, with each parameter having its own value and data type. Once complete, this structure is used within a function call to the forward model to create the predicted light curve.

1. Parser Test
    a. **Description:** Ensure that the parser function correctly parses already read in tester data to the forward model data structure
    b. **Main Flow:**
        i. Parse and assign data to its appropriate place in the struct
        ii. Compare each of the values within the struct to ensure the expected values
            1. Example Input: double aPrimary = 125, double aSecondary = 138
            2. Expected Output: vector<double> a = {125, 138}
    c. **Expected Outcome:** Valid forward model arguments data structure was created using correctly parsed data

Minimization

The NLM module depends on a NLM function found within the amoeba.h file. This file was created by the authors of *Numerical Recipes 3rd Edition: The Art of Scientific Computing* [4] and was recommended to the team by the clients for use within this section of the project. A PDF copy of this book can be found in the licht-cpp/nlm/doc/ directory of the licht-cpp API. This function is responsible for finding and producing the minimum of a given mathematical equation.

As mentioned at the end of the "Acknowledgements " section (p. xv-xvi) in *Numerical Recipes 3rd Edition,* testing was done on all of the routines in the book. The developers also mentioned that these routines were tested on DEC and Sun workstations running the UNIX operating system and on a 486/33 PC compatible running MS-DOS 5.0 / Windows 3.0. As such, the team considers this NLM function to have been tested thoroughly enough such that additional unit testing for this function is not required.

Plotting

The NLM module depends on a plotting function in order to plot the predicted light curve and the observed light curve. This functionality is provided by the matplotlibcpp.h file. Links to the associated GitHub repository[5] and documentation[6] can be found in the footer of this page. However, since it is impossible to check that the resulting plot is correct without human intervention, ensuring that the information is plotted appropriately was done in integration testing.

5.1.3 Forward Model GUI

Since the functionality of the GUI is dependent on working with the forward model, unit testing was not feasible. Other modules that are used within the GUI such as the ForwardModel.cpp and the Ephemeris.cpp have unit tests within the licht-cpp library. To verify that the forward model GUI works appropriately with all of the necessary modules, a majority of the testing for this module was completed using integration testing.

[4] http://numerical.recipes/
[5] https://github.com/lava/matplotlib-cpp
[6] https://readthedocs.org/projects/matplotlib-cpp/downloads/pdf/latest/

5.1.4 Video Generator
Since the functionality of the Video Generator is dependent on the licht-cpp library, there was no feasible way to have a unit test. To verify that the video generator is working correctly, it was verified through integration testing.

Unit testing was critical to this project in ensuring that all individual modules are functioning properly. It is important that these added modules produce correct results to verify their functionality. Once these individual modules were verified, they were ready to be integrated with other modules and tested as a complete system.

5.2 Integration Testing
Integration testing is used to expose problems with the interaction between modules. While each module of a codebase may function properly, it is important that information that is passed by, or to, other modules is in the proper format and has the values that are needed. Integration testing focuses on the passing of parameters and return values. Therefore, the tests need to check the integration of the modules and submodules that have been implemented.

The Incremental strategy was used to test these modules. In general, Incremental testing joins two or more modules at a time that are logically related. The team tested the triaxial ellipsoid module with NLM and the GUI respectively since there lacks an interface for the GUI and NLM modules to interact.

5.2.1 Triaxial Ellipsoids
Many of the integration tests for ellipsoids came naturally with the addition of the ellipsoid submodule. Since the forward model call will output a light curve and render the ellipsoid image, improper integration from the beginning would make rendering and using ellipsoids impossible. It was critical to test that the light curve created while using an ellipsoid is accurate since this data will either be graphed automatically for the clients via the GUI or given to the user through a text file.

These tests were implemented in TestForwardModel.cpp. The goals of these tests were to render viable asteroid images and generate feasible light curves. As it was of utmost importance to achieve these goals, there were additional ephemeris files added to the data folder within the test folder. Sila Nunam, a binary system previously used for testing by Paired Planet Technologies, was utilized again to test whether the rotation is proper when rendered as ellipsoids. A singular asteroid called Roxane was added to the ellipsoid tests as an added measure for rotation and accuracy. This proved that both binary systems and single asteroids could be rendered and manipulated correctly, confirming that the clients can gather the necessary data. Subsequently, these tests produced a light curve which was verified for accuracy.

Forward Model

1. ForwardModel Test
    d. **Description:** Runs the forward model using Sila Nunam with ellipsoids, as well as the single asteroid Roxane with an ellipsoid
    e. **Main Flow:** Input all the parameters into the forward model for each respective test. Make sure rendering is happening to visually check models
        i. Specify 2 ellipsoids for Sila Nunam, and their correct rotations
        ii. Specify 1 ellipsoid for Roxane with its correct rotation
    f. **Expected Outcome:** The forward model correctly implements the ellipsoid module to produce an accurate model of Sila Nunam and Roxane
    g. **Alternative Flow:** The user does not set the ellipsoids to rotate
    h. **Expected Outcome:** The forward model renders the ellipsoids the same as before, but without any rotations occuring

5.2.2 NLM

NLM's unit tests were completed during the same workflow, resulting in simple integration testing. The user inputs the observed data (which represents the observed light curve) and user input file into the NLM module. NLM calls the forward model to produce the predicted light curve which is compared to observed data. After the comparison, the NLM module conducts its main functionality which results in the execution of the unit tests implemented for the NLM module.

Testing NLM in conjunction with the triaxial ellipsoid submodule only required a change in the parameters passed to the forward model through the NLM parser. Since the forward model GUI has no interface with the NLM module, integration testing between these two modules is not currently possible. Lastly, when testing integration on NLM with the video generator, only a flag needed to be set to produce rendered images within the NLM user input file. These rendered images can be compiled to a video using the video generator and viewed to verify the correct shape is rendered.

To verify integration testing for the NLM module, the addition of the following tests was necessary. These tests ensured that the functionality needed for NLM's interaction with other licht-cpp modules, as well as its own functions, was behaving as expected.

File Input and Parser

1. ReadFileAndParser Test
    a. **Description:** Ensure that the parser function correctly parses tester data to the forward model data structure using tester data that is read in using the read file (user input) function
    b. **Main Flow:**
        i. Read in tester data to act as user input
        ii. Parse and assign data to its appropriate place in the struct
        iii. Compare the values within struct for expected values
            1. Example Input: double aPrimary = 125, double aSecondary = 138
            2. Expected Output: vector<double> a = {125, 138}
    c. **Expected Outcome:** Valid forward model arguments data structure was created using correctly read and parsed data

File Input, Parser, Minimization, and Plotting

1. Nonlinear Minimizer Module Test
    a. **Description:**
        i. Run NLM with read in and correctly parsed data, and verify that the produced output is correct
        ii. Ensure that output is created using the plotting function in conjunction with the other functions found within NLM
    b. **Main Flow:**
        i. Read in tester data to act as user input
        ii. Parse input data into forward model arguments struct
        iii. Forward model takes in argument struct and produces predicted light curve
        iv. Predicted light curve is compared against observed light curve
        v. Continues until chi-square value is within a set tolerance value

        vi. Plot predicted light curve and observed light curve
    c. **Expected Outcome:**
        i. Produced estimates, chi-square value, and light curve data that falls within an expected range
        ii. A .png file is created containing the created plot

File Input, Parser, Minimization, Ellipsoid and Video Generator

1. Nonlinear Minimizer, Ellipsoid and Video Generator Module/Submodule Test
    d. **Description:**
        i. Run NLM with read in and correctly parsed data, and verify that the produced output is correct
        ii. The read in data is tester data attributed to ellipsoid objects, thus testing the NLM module using light curve data derived from ellipsoids
        iii. Images are rendered using the tester data and these rendered images can compiled into a video using the video generator
    e. **Main Flow:**
        i. Read in tester ellipsoid data to act as user input
        ii. Parse input data into forward model arguments struct
        iii. Forward model takes in argument struct and produces predicted light curve
        iv. Predicted light curve is compared against observed light curve
        v. Continues until chi-square value is within a set tolerance value
        vi. Compile rendered images into a video
    f. **Expected Outcome:**
        i. Produced estimates, chi-square value, and light curve data
        ii. Light curve data that falls within an expected range for given ellipsoid data
        iii. Video is generated using the produced rendered images

5.2.3 Forward Model GUI

To verify that the GUI is working appropriately and will not crash for the user, it was necessary to create integration testing for this module. These tests were done manually where the expected outcome is constant. The goal of these tests were to verify that the GUI functionality is working correctly using the existing code base's tests as verification.

Acceptable Parameter Test

To ensure that the GUI is accepting parameters that are valid within the forward model, the test verified that the input for the matching variable is within the constraints of its corresponding type. When testing each variable individually, all other parameters passed to the forward model were a constant from the Sila Nunam test case found within the TestForwardModel.cpp.

1. AcceptableParameters Test
    a. **Description:** Verify that the GUI is accepting parameters that match the type of the variable within the forward model
    b. **Main Flow:**
        i. Compile a list of parameters to be passed into the forward model through the GUI
        ii. Verify which variables are acceptable, and which as the test cases
        iii. Apply constants to all other forward model parameters to keep output constant
        iv. Run forward model through the GUI
        v. Check if the GUI throws an error for the variable.
        vi. Check if the forward model produces a logical light curve
    c. **Expected Outcome**: If the variable was an acceptable value, the forward model will produce a light curve, if it was not an acceptable input the user will be prompted through the GUI to change the value

Plotting

To verify that the light curve plotting functionality is working correctly within the GUI, the test passed constant data sets to the function where the graphed light curve is known and was compared to existing light curve graphs. When testing the plotting functionality, Team Andromeda used the observed light curves provided by the clients to verify that the functionality is working appropriately.

1. Plotting Test
    a. **Description:** Verify that the light curves that are plotted are accurately plotted by the function within the GUI
    b. **Main Flow:**
        i. The constant Sila Nunam Data will be passed into the "Observed Light Curve" file input
        ii. The plotting functionality will be called

        iii. Verify that the light curve created is the same as the constant graph
    c. **Expected Outcome:** The graph created by the plotting functionality will be the same as the constant graph as a comparison

5.2.4 Video Generator

To ensure that the video generator is working correctly, the batch script was run on all renders produced in the GUI and NLM unit testing. This verified that the video is created correctly, and that the speed used when the script is run is correct.

Image Verification

To verify that the .mp4 file created by the video generation script is correct, the test was run every time that the forward model is called with render set to true.

1. ImageVerification Test
    a. **Description:** The test consists of making videos from several series of renders that are output when using the GUI or NLM
    b. **Main Flow:**
        i. The set of images are rendered after the running of either the GUI or NLM
        ii. The batch script is executed to create the video. Input is required from the user to determine speed and whether the current images should be removed. For testing, images are not be removed
            1. The speed that the video is ran at is set to the default speed
            2. The speed that the video is ran at is set to double the default speed
        iii. The .mp4 is created
    c. **Expected outcome:**
        i. The images are in the correct order in the video
        ii. The video is running at the default speed
        iii. The images are not be removed from the render folder

Integration testing is essential to ensure that modules interact properly and that the code connecting each module does not have any defects. Once all integration tests pass, the testing moves out of the automated realm and into the real-world with usability testing.

5.3 Usability Testing

Usability testing is used to assess how well the user is able to navigate the software. This means that real-world testing needed to be done, and that this testing is more social and statistical than unit testing and integration testing. This type of testing is very broad and comes down to if the user is able to use the software effectively. If the users are able to use the software with little to no problems, the usability test is a pass. Alternatively, if the users struggle using the software, the usability test fails.

To ensure that the clients can use the new features as intended, they were provided documentation that walks them through each feature's usage. The documentation focused on each feature individually, such as how ellipsoids are made and used. This helped give the users a broad idea of how the individual features function. Lastly, the documentation focused on a walkthrough of how to use the features holistically.

Specifically, this walkthrough included how to generate ellipsoids, how to effectively utilize NLM, how to navigate the GUI, and how to create a video based on the forward model's output. This allowed for a greater understanding of the software's effectiveness and how it would fit into the user's workflow. Usability tests were a vital part of testing for the software and provided the team with valuable feedback needed to improve the cohesiveness of the software.

5.3.1 Ellipsoid

Ellipsoids were made to be used within the forward model, unlike the GUI and NLM features. GUI and NLM differ because they both utilize the forward model function call to perform their operations. Since ellipsoids are nested within the forward model, their usability was tested during the usability tests for the GUI and NLM features. Thus, ellipsoids did not have their own specific usability test, but rather have had their usability embedded into the forward model itself.

Final Outcome: The GUI and NLM features were able to effectively use the forward model to accurately create and utilize ellipsoids.

5.3.2 NLM

Since there was complexity involved with the implementation of the NLM module, testing with the clients was necessary to ensure it was functioning as they required. To conduct this testing, the team provided documentation on how the NLM module works, and how the module estimates the best fitting parameters.

The clients then used the NLM module by providing observed data of their choice and an input file that specifies which parameters they want to estimate. Additionally, the input file also allowed the clients to decide whether or not they want to use other various functionalities provided by the forward model and NLM such as rendering and plotting. Using this strategy, the clients worked through the use cases for NLM and provided feedback on their experience using the NLM module.

Final Outcome: While the data produced by the NLM module differs based on the input provided, the module produced data that was acceptable for the clients. Additionally, the use of the NLM module met the expectations of the clients in regard to data input/output, parameter estimation and data visualization. These expectations were outlined in the Requirement Document that was signed by the clients.

5.3.3 Forward Model GUI

The GUI connects the users to the software. The best way to test if the users could navigate the GUI was to let them use it independently. To help the users evaluate the GUI, documentation was provided, which explained to the users how to run the GUI, how to correctly input each parameter needed, and how all buttons and drop down lists interact. Creating the usability test in this fashion allowed the users to get hands-on experience with the GUI. This test was vital for evaluating the GUI's usability, and gave the team a greater insight on how the clients will use this feature within their workflow.

Final Outcome: The team allowed the clients a set amount of time to go through the section of documentation for the GUI. Once they went through the section, they had 10 minutes to try using the Sila Nunam data. Finally, they were given 20 minutes to use the interface with their own data and run their own tests.

5.3.4 Video Generator

After allowing the users to test the video generator script, the team gathered their feedback and made any adjustments necessary to have the script meet the clients' standards. The clients' feedback on aspects such as small quality of life adjustments increased the quality of the script.

6. Project Timeline

This project was developed over two semesters. A Gantt chart was utilized to display and identify the status of the tasks that were complete over both semesters. Figure 3 shows the tasks completed during the Fall semester, while Figure 4 shows the tasks completed during the Spring semester.

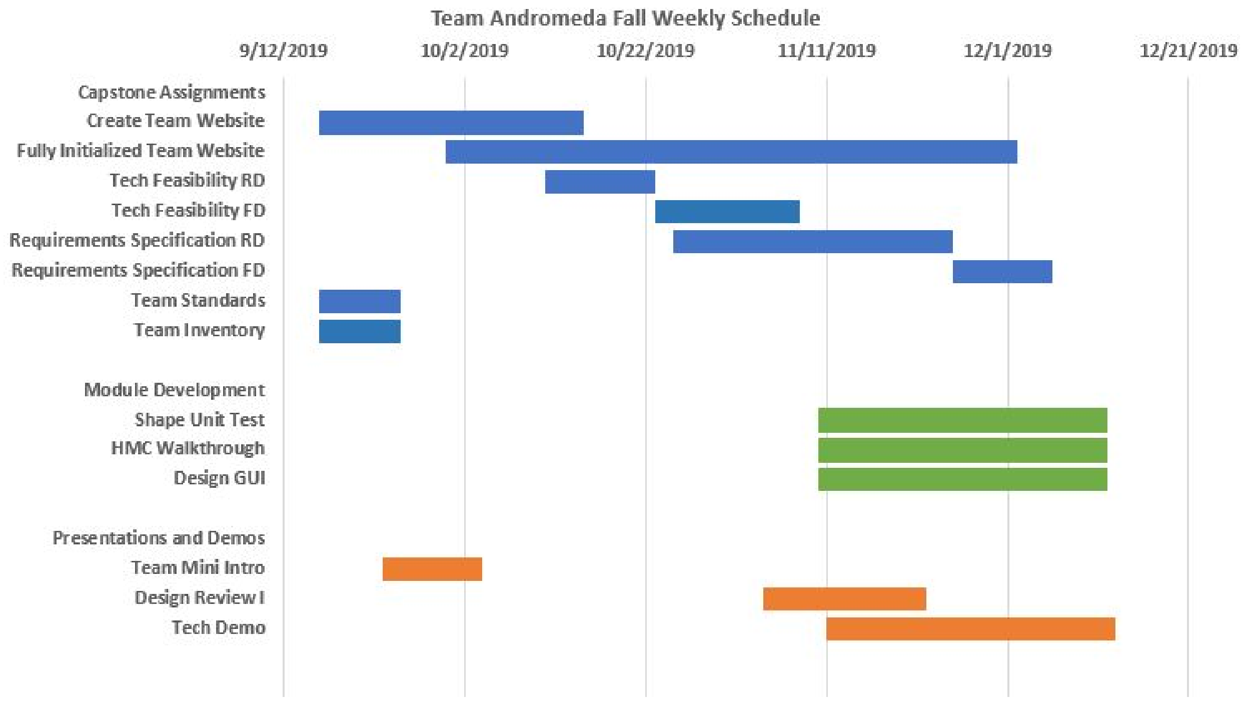

Figure 3: Tasks completed during the Fall 2019 semester

The tasks completed during the Fall semester include the various documents necessary to establish the foundation of our project. These documents include information such as which tools we would be utilizing for development, the requirements laid out by our clients, as well as a demo for our mentor to demonstrate the team's initial development. We also started our website during the Fall semester and have been continuously updating it to reflect changes to our development process.

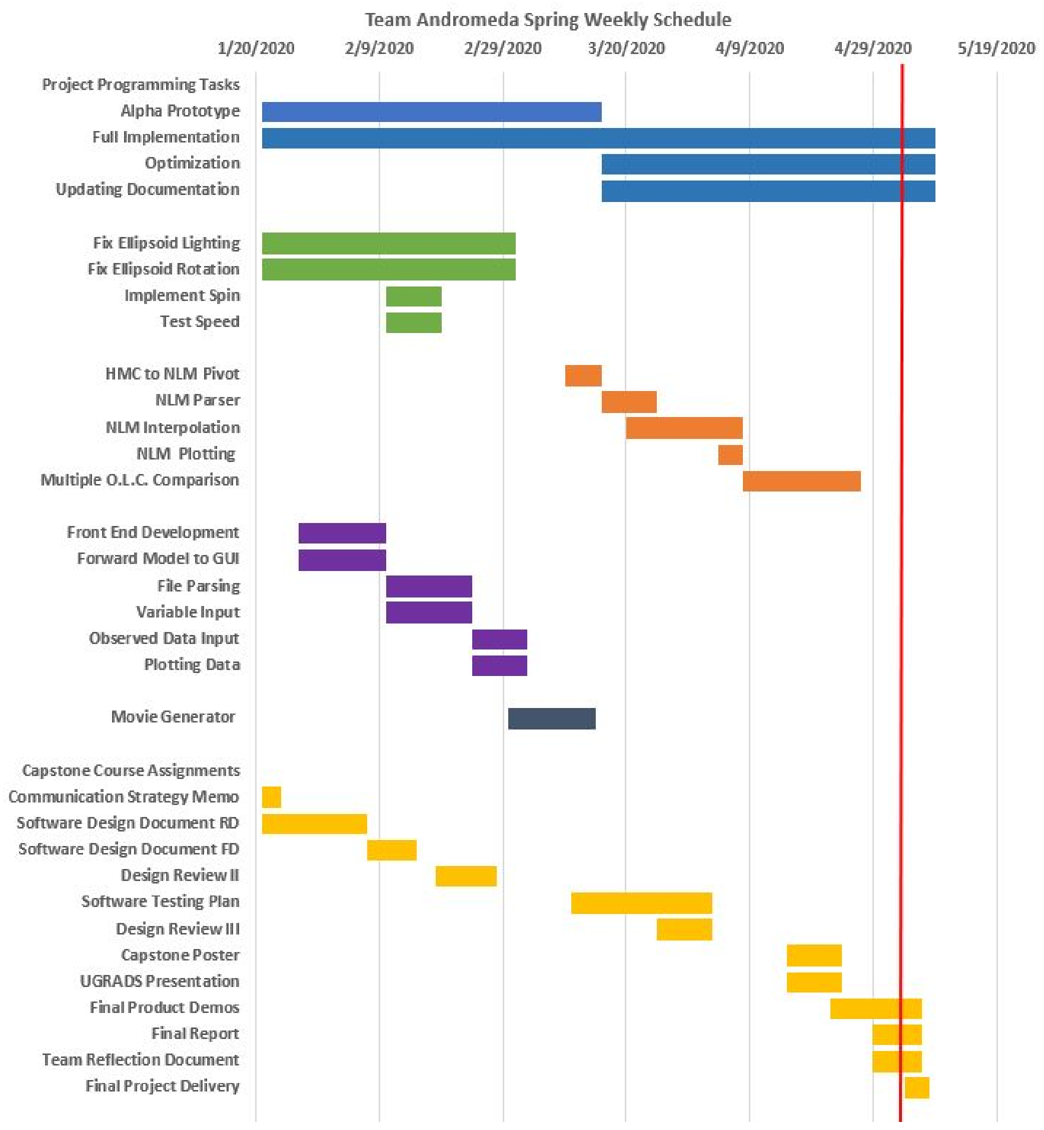

Figure 4: Tasks completed during the Spring 2020 semester

The tasks completed during the Spring semester placed more emphasis on developing the product for our clients. This included developing various aspects for each of the three main components of the project. Fixes to the lighting and rotation of the ellipsoid shape were done during the Spring semester, as well as building out the backend of the GUI and finalizing its front end. Additionally, our team made the switch from HMC to NLM during the Spring semester, which led to completion of tasks such as parsing and plotting for NLM. Lastly, the Spring semester included the final implementation and documentation of our product.

7. Future Work

For future work and features for this project, we have spoken with our clients and they have outlined what the next stages might look like for this project. The following are the next features considering to be implemented:

Implementation of a Hamiltonian Monte Carlo algorithm to explore parameter space

Due to technical difficulties that arose during the implementation of the Hamiltonian Monte Carlo (HMC) algorithm, we were unable to implement the algorithm during this iteration of the project. However, after speaking with our clients, they mentioned including the implementation of the HMC algorithm in a future iteration of the project. This would benefit our clients as it would provide an algorithm that is able to **explore the parameter spaces**, but **requires additional computational time and power**.

Optimization of the GPU

Since the forward model uses ray tracing to render objects, implementing GPU parallelization would be highly beneficial. Optimizing the use of the GPU would greatly **reduce the time spent on rendering**, specifically for faceted shape objects.

Implementing support for n-body systems, where n > 2

The forward model currently supports single objects and binary systems. Since there are systems that can have multiple bodies in their orbit, additional functionality should be added to **support n-body systems**.

Additional GUI preferences

In the future, teams will be adding in more preferences to the GUI, allowing the user to **cut down on parameter input time**, and **select settings** that they would like implemented in the forward model call. These preferences make for a better user experience and create an enhanced workflow.

8. 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 icy objects floating in the Kuiper Belt that are yet to be explored. Up to 30% of those asteroids are considered to have at least one secondary asteroid in their orbit. **Studying these binary systems leads to important discoveries 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 binary asteroids in the Kuiper Belt using 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**. This project aims to help Dr. Grundy and Dr. Thirouin at Lowell Observatory solidify these attributes.

Paired Planet Technologies, the previous development team, was able to initialize this project by crafting an **efficient light curve generator**. To enhance this, specific additions were needed to create **holistic software dedicated to modeling binary systems**. As Team Andromeda worked with the clients and conducted research, the additions were narrowed down to the following specifications:

- Triaxial ellipsoid objects
- A GUI to visualize the behavior of the target system based on the modeling output
- A way to statistically find best fitting parameters based on observed data
- Ability to create a video of the system's motion
- An automatic plot of the light curve

To further enhance the clients' workflow, future development teams will need to implement an **HMC algorithm**, expand the forward model to **support n-body systems**, optimize the software to work with **GPU parallelization**, and work with the clients to add **more GUI preferences.**

As of this instant, the team has refactored and thoroughly tested the software with the clients. Team Andromeda is confident that these additions to the forward model will positively impact the clients' workflow by **quickly generating best fitting parameters**, **cutting the time spent rendering**, and **easing the use of the forward model function**. Getting the opportunity to work as a team to create the GUI, NLM, and the triaxial ellipsoid shape was immensely rewarding. The team is very thankful to have worked with Dr. Thirouin and Dr. Grundy and is looking forward to seeing the future growth of the software.

Appendix A: Development Environment and Toolchain

Hardware

This software is first-and-foremost designed for Linux systems. It was developed on Ubuntu 18.04 as well as Arch Linux (rolling release). It is tested on both Linux Distributions across multiple computers, along with on a Ubuntu 14.04 VM. This VM is provided by Travis-CI and tests every commit. The specs are:

- Software
    - OS: Ubuntu 14.04.5 LTS
    - Runtime kernel version: 4.4.0-101-generic
    - Compiler: clang 5.0.0 / gcc 4.8.4
    - CMake: 3.9.2
- Hardware
    - Memory: 7.5GB
    - Cores: 2
    - Disk space: ~18GB

The clients use multiple versions of Fedora along with CentOS6. The code works on both distributions. Additionally, the code has been manually tested on Windows 10 via a MinGW set-up. Ultimately, our code base is ultra-portable. Complications, if they arise, will often stem from the build system and not the code itself. The build-system, albeit sophisticated, does not do much error checking and hacks some things together. Some specific issues that have been resolved in the past are:

- Windows 10 uses different compiler flags than Linux
- Different compiler versions use different absolute value functions
- IDL uses different sized-integers than GDL
- IDL's interface does not include the required integer types
- There is one method we use that is Linux-specific: drand48
    - This section of code (antialiasing) has been disabled on Windows

Windows 10 introduced some complications when interacting with *IDL* and *GoogleTest*. It's possible that similar complications could be introduced when testing on *macOS*. *MacOS* is untested. Its default compiler, *clang*, has been tested.

The libraries we have decided to use (namely: *GoogleTest*, *Eigen*, *TinyJPEG*) are all cross-platform compatible. Theoretically *macOS* will function as well.

Although the forward model is computationally expensive, any decent machine can be used for development. The main development machine has used an FX-6300 and 12GB of memory. The weakest development machine was an Intel 2-core and 4GB of memory. Apple computers are potentially supported with minimal changes to the build system. An NVIDIA card is recommended, as the most efficient future work is adding CUDA parallelization to the codebase.

Toolchain

We use a lot of technology under-the-hood. Even with this, the dependencies are lightweight and, for the most part, already accounted for. We use the following libraries: GoogleTest, Doxygen, Eigen3, TinyJPEG, CMake.

- GoogleTest has to be compiled and cannot be easily installed. Because of this, we include a 1.7.0 release of GoogleTest in the codebase, for the sake of convenience. No installation has to be done.
- Doxygen is optional -- it is used to generate automatic documentation
- Eigen3 is header only and we include a 3.3.7 release, nothing has to be done
- TinyJPEG is header only and we include the latest release, which was from 2016

Thus, CMake and a compiler (g++ or clang) are required. CMake is installed by default in most Linux distros, though we require minimum version 3.4.

The compilers should support C++17. Although we do not utilize many C++17 features, there is no reason to not require its support, as this version has more optimizations and future potential. As a bare minimum, if there is no method to get C++17 support, C++11 must be supported. Up to Ubuntu 16, the compilers are quite old and need to be manually updated. The build system uses the latest C++ standard supported by the compiler, whether it be the 2011, 2014, or 2017 standard. Note that the build system only uses standards 11, 14, and 17. 20 is not automatically supported.
As far as code development goes, we have used the following set-up:

Operating system:

- Ubuntu 18.04

- Arch Linux (Rolling release)

Code editor: Any text editor or IDE can be used. We used Visual Studio Code. It provides much more functionality than its main competitor, Atom. Our setup was as follows:

- CMake (twxs.cmake)
- CMake Tools (vector-of-bool.cmake-tools)
- CMake Tools Helper (maddouri.cmake-tools-helper)
    - This trio of CMake plugins allow for the Intellisense one might be used to using in normal Visual Studio. In other words, it highlights errors and allows easy navigation of the codebase (e.g. Go-to definition).
- cpplint (mine.cpplint)
    - This an implementation of Google's C++ style-guide that is widely accepted. Almost 100% of the code base follows cpplint. Please continue following it for consistency and professionalism.
- QTCreator
- Qmake

We attempted to use the full Visual Studio IDE in Windows, though ran into annoyances in getting the CMake set-up to cooperate. Visual Studio attempts to take too much control of the build-system. We highly recommend using Visual Studio Code. The plug-ins can be used to actually build the code, though we just used the terminal within VS Code for building and testing the code.

Tools:
- CMake (> 3.4)
    - This stands for cross-platform make. It generates a makefile. You can think of a makefile as assembly code and CMake as C code. It's a language that makes it much easier to set-up sophisticated build options and to manage dependencies. It is the de-facto standard and Zach had experience in it, which is why we use it.
- doxygen
    - This generates automatic documentation based on the comments in the code. Because of this, you must comment in a special format.
- g++ / clang++ (versions that support C++17 are recommended, but the minimum is a version that supports C++11)

    - Compilers are of course required. g++ is often slightly quicker whereas clang++ provides better compilation messages.
- QMake (> 3.0)
    - QMake is the standard compilation tool used when compiling any QT project. It was used due to necessity when developing within QTCreator.
- QT (> 5.0)
    - QT is a framework that allows for powerful, yet beautiful GUI development. It is available in both C++ and Python versions.

Setup
Given that this software is written to be cross-platform and is in a compiled language, the set-up is naturally more complicated than just calling a Python script. Most of the complications have been abstracted from the user by storing dependencies directly in the repository. In the simplest use case -- an up-to-date Linux machine -- the setup is extremely simple. The most complicated case is setting up a development environment in Windows. Set-up instructions have been originally written in the GitHub README and Wiki and have been extracted here.

Licht-cpp Setup
As part of final delivery, licht-cpp should have been installed on a platform of your choice. Over time, however, you may want to move to a new platform or re-install the product.

Licht-cpp is a cross-platform solution that requires a C++ compilation environment using CMake. Ultimately, to install licht-cpp the following commands must be run:

- cd licht-cpp/build
- cmake ..
- make

To accomplish this, follow the set-up for a CMake environment for your system below.

Debian-based systems
- sudo apt install cmake

RedHat-based systems

- dnf install clang clang-devel # If you want to use clang
- dnf install libomp-devel # If openmp is not installed
- dnf install cmake

CentOS 6/7

# Install g++ 8

- sudo yum install centos-release-scl
- sudo yum-config-manager --enable rhel-server-rhscl-7-rpms
- sudo yum install devtoolset-8

# Update terminal

- scl enable devtoolset-8 bash

# Install CMake 3.X

- http://jotmynotes.blogspot.com/2016/10/updating-cmake-from-2811-to-362-or.html

Arch-based systems

- Sudo pacman -S cmake

Windows

Essentially, Windows needs a MinGW setup that supports pthreads, make, and openmp. To do this, first install the MinGW installation manager (https://osdn.net/dl/mingw/mingw-get-setup.exe). Then install the following packages. Many come with the base system of MinGW and MSYS, but they are pasted here for verbosity:

- mingw-developer-toolkit-bin
- mingw32-autoconf-bin
- mingw32-automake-bin
- mingw32-autotools-bin
- mingw32-base-bin
- mingw32-binutils-bin
- mingw32-gcc-bin
- mingw32-gcc-lic
- mingw32-gcc-g++-bin
- mingw32-gdb-bin
- mingw32-gettext-bin
- mingw32-gettext-dev
- mingw32-libatomic-dll

- mingw32-libexpat-dll
- mingw32-libgcc-dll
- mingw32-libgettextpo-dll
- mingw32-libgmp-dll
- mingw32-libgomp-dll
- mingw32-libiconv-bin
- mingw32-libiconl-dev
- mingw32-libiconv-dll
- mingw32-libintl-dll
- mingw32-libisl-dll
- mingw32-libtldl-dev
- mingw32-libltdl-dll
- mingw32-libmpc-dll
- mingw32-libmpfr-dll
- mingw32-libpthreadgc-dev
- mingw32-libpthreadgc-dll
- mingw32-libquadmath-dll
- mingw32-libssp-dll
- mingw32-libstdc++-dll
- mingw32-libtool-bin
- mingw32-libz-dll
- mingw32-make-bin
- mingw32-mingw-get-bin
- mingw32-mingw-get-gui
- mingw32-mingw-get-lic
- mingw32-mingwrt-dev
- mingw32-mingwrt-dll
- mingw32-pthreads-w32-dev
- mingw32-pthreads-w32-lic
- mingw32-w32apidev
- mingw32-wslfeatures-cfg
- msys-autogen-bin
- msys-base-bin
- msys-bash-bin
- msys-bison-bin
- msys-bsdcpio-bin
- msys-bsdtar-bin
- msys-bzip2-bin
- msys-core-bin

- msys-core-doc
- msys-core-ext
- msys-core-lic
- msys-coreutils-ext
- msys-cvs-bin
- msys-diffstat-bin
- msys-diffutils-bin
- msys-dos2unix-bin
- msys-file-bin
- msys-findutils-bin
- msys-flex-bin
- msys-gawk-bin
- msys-grep-bin
- msys-guile-bin
- msys-gzip-bin
- msys-inetutils-bin
- msys-less-bin
- msys-libarchive-dll
- msys-libbz2-dll
- msys-libcrypt-dll
- msys-libexapt-dll
- msys-libgdbm-dll
- msys-libgmp-dll
- msys-libguile-dll
- msys-libguile-rtm
- msys-libiconv-dll
- msys-libintl-dll
- msys-libltdl-dll
- msys-liblzma-dll
- msys-libmagic-dll
- msys-libminires-dll
- msys-libopenssl-dll
- msys-libopts-dll
- msys-libpopt-dll
- msys-libregex-dll
- msys-libtermcap-dll
- msys-libxml2-dll
- msys-lndir-bin
- msys-m4-bin

- msys-make-bin
- msys-mktemp-bin
- msys-openssh-bin
- msys-openssl-bin
- msys-patch-bin
- msys-perl-bin
- msys-rsync-bin
- msys-sed-bin
- msys-tar-bin
- msys-termcap-bin
- msys-texinfo-bin
- msys-vim-bin
- msys-cz-bin
- msys-zlib-dll

Then install CMake: https://cmake.org/download

Compilation is done via the MSYS shell located at 'C:\MinGW\msys\1.0\msys.bat'. Once in its bash shell, the C:\ drive can be accessed with 'cd /c'. Navigate to the licht-cpp build directory and call 'cmake .. -G "MSYS Makefiles"'. This only has to be configured once.

General Configuration
With the right packages now installed, the build system must be configured.

Set CC/CXX compiler
This step is only sometimes needed, as it depends if the system already has a global compiler variable. The codebase is tested on both g++ and clang++. Either may be used. If not already defined, a common way to add a global variable is to modify ~/.bashrc.

Make
CMake is used to build the code. The make step produces a unit test executable (tester) and a shared library (liblicht-cpp.so). CMake can be used as follows:

- cd licht-cpp/build
- cmake ..
- make # add -jX where X is twice the number of your CPU cores

Run
To run the unit tests, call tester. Otherwise, just interface with liblicht-cpp.so. When calling tester, it is important to be within the build directory, as the file paths within the test are relative to it.

Documentation
To generate documentation, run the below command from licht-cpp/:

- doxygen Doxyfile # Generated in doc/

It generates five forms of documentation. The most useful form, in our opinion, is HTML. Open doc/html/index.html to browse the documentation of this code base. As you will see in Appendix F, this documentation can also be outputted as an incredibly useful and extensive PDF. To generate this PDF:

1. Install pdflatex
    a. We have found that you also have to install texlive-latex-extra
2. cd doc/latex
3. make

The output is refman.pdf.

NLM Setup
Similar to licht-cpp, the NLM is also a solution that can be used on any platform. There are only three requirements associated with using the NLM:

- A C++ compilation environment that utilizes CMake
- Python2.7 and Python2.7-dev, Matplotlib, and Numpy installed
- The compilation of the licht-cpp API

In order to start using the NLM, the user will first have to install Python2.7. This includes the standard Python2.7 library, as well as the Python2.7-dev library. Along with these Python installations, the user will have to install Matplotlib and Numpy.

To install Python2.7 and Python2.7-dev, we recommend downloading the packages listed on the official Python website[7] for Python 2.7.18. The Windows

[7] https://www.python.org/downloads/release/python-2718/

and macOS installers found on the supplied page will install both Python 2.7 and Python2.7-dev. However, to install Python2.7-dev for Debian and Redhat systems, the user will need to run an additional command in the terminal:

- Debian systems: sudo apt-get install python2.7-dev
- Redhat systems: please refer to this official packages website[8]

To install Matplotlib and Numpy, we recommend using pip[9] due to it's versatility and reliability. To install these libraries using pip, you will need to run the following commands in a terminal or command prompt window:

- Matplotlib[10]: pip install matplotlib
- Numpy[11]: pip install numpy

Once these libraries have been installed, the licht-cpp static library will need to be built. This is done by following the instructions found in section 2.1 for installing the licht-cpp API.

After installing the licht-cpp API, a static library called "liblicht-cpp.a" will be created in the licht-cpp/build directory. This static library is required to run the NLM as it provides the forward model functionality to the NLM.

Afterwards, the user can install the NLM using similar commands, but in a different directory:

- cd licht-cpp/nlm/build
- cmake ..
- make

Documentation
To generate documentation, run the below command from within licht-cpp/nlm/:

- doxygen Doxyfile # Generated in doc/

[8] https://pkgs.org/download/python2-devel
[9] https://pypi.org/
[10] https://pypi.org/project/matplotlib/
[11] https://pypi.org/project/numpy/

It generates five forms of documentation. The most useful form, in our opinion, is HTML. Open doc/html/index.html to browse the documentation of this code base.

GUI Setup

Similar to the licht-cpp library, the GUI is cross platform and available for use on any platform. To compile the GUI, the user must have qmake3.0 or higher on their system. Due to the difference in compilation, the GUI must be compiled after the licht-cpp library is compiled. The GUI is reliant on the static library (.a file) that is created when licht-cpp is compiled.

To compile the GUI:

- Compile forward model
- cd .. (Back into licht-cpp folder)
- cd GUI/
- qmake
- make

Appendix B: Production Cycle Details

This project is treated as a professional software solution and has some production infrastructure that must be followed. In essence, the following steps are done for each group within the project:

1. Checkout the respective branch.
2. Pull to make sure the branch is up to date.
3. Implement functionality, split up into functional commits.
4. When the feature is finished, push the update and have another group member review the commit.
5. After the code is reviewed and tested for functionality, all others working on that branch should pull to have the most updated code.

To share, maintain, and manage our growing code base, we use Git. The development platform is GitHub, which allows for both public and private repositories for free thanks to student pricing.

Here we also define our workflow for issue tracking and communication, which are closely tied with version control. It is a lot of information up front because there are no assumptions about the team's knowledge of Git and GitHub.

Basics

- The 4 levels of the Git organization
    - Working directory (on local computer): work in VS Code or similar text editor
    - Staging area: include/save changes to the next commit
    - Local repository: commit to project history
    - Remote repository on github.com: share code with collaborators and backup local branches
- We use issues and milestones to communicate code enhancements, bugs, priorities, and current progress.
- We have three types of branches:
    - Master branch:
        - Any commit on the master branch aims to be deployable. Such commits are usually the result of merging/rebasing with a feature or bugfix branch. Once deployable a unique version number is released. Deployable for us means that commits are tested.
    - Bugfix branches:
        - When a bug is discovered on the master branch, a bugfix branch and an issue should be created. The issue should be assigned to the master milestone. A bugfix branch should be named after the respective issue. An example of a bugfix branch would be bugfix_16, which represents issue #16.
    - Feature branches:
        - Everything else should be a feature branch, which is where code development is done before it is merged back to master. Each feature branch needs its own milestone. Any bug fixes needed on a feature branch should be directly committed to the feature branch. Feature branches should have descriptive but concise names in upper camel case, such as feature_BetterErrorMessages.
- Testing involves at least that each line of code was executed and did not throw an error or stopped execution unexpectedly. However, writing

reusable unit test cases (GoogleTest for C/C++) is the preferred way to test our code.

- We use semantic versioning using the format MAJOR.MINOR.PATCH. Every commit to the master branch updates the version number. A backwards-incompatible commit increases MAJOR and resets MINOR and PATCH to 0. A backwards-compatible commit adding new functionality increases MINOR and resets PATCH to 0. A backward-compatible commit fixing bugs (etc) increases PATCH.
- Inspecting a repository
    - State of working directory and staging area: git status
    - History of commits: git log --graph --full-history --oneline --decorate
    - List of branches (* indicates the active branch):
    - Local branches: git branch
    - Remote branches: git branch -r
    - List of remote connections: git remote -v
- Finding stuff in a repository
    - Find all commits which have affected a file: git log -- *<part_of_file_name>*
    - Find the SHA of the last commit that affected a file git rev-list -n 1 HEAD -- <file_path>
    - Find all commits which have deleted files and list the deleted files: git log --diff-filter=D --summary
- Error reporting:
    - All master branch bugs require you to create an issue in GitHub. Ideally, you provide a unit test which demonstrates the failing code. This unit test serves also as a benchmark to identify the solution of the issue. If it is not possible to write a unit test, please provide a minimal reproducible example. Some resources that may help:
        - How to create a Minimal, Complete, and Verifiable example
        - How to Report Bugs Effectively
        - How to make a great R reproducible example?
        - How to write a reproducible example
    - Create a bugfix branch
    - Close the issue with a reference
    - Create a pull request to the master branch, with appropriate reviewers

Creating Issues

Issues describe suggested new features, a symptom of a bug, a proposed change, and so on. If you create an issue, decide to work on one, or you are assigned to one, then:

- Assign it to a team member and/or yourself, if applicable
- Add an in progress label, if you are currently working on it
- Add a priority label, if it has a low priority or a high priority
- Add a category label, such as 'bug' or 'enhancement'

Closing Issues

When the issue is resolved, reference it in the final commit that solves it (which can be done in the title or body).

- Note: Issues will not be closed via reference until the branch is merged to master. If you are resolving an issue on a feature branch, please still use a reference, as it provides beneficial documentation, but you should also manually close the issue afterwards.

Communication (commit messages, comments on issues, etc.)

Why good messages are important: Erlang: Writing good commit messages:

"Good commit messages serve at least three important purposes:

- To speed up the reviewing process.
- To help us write a good release note.
- To help the future maintainers of Erlang/OTP (it could be you!), say five years into the future, to find out why a particular change was made to the code or why a specific feature was added."

How to write good messages: Who-T: On commit messages:
"A good commit message should answer three questions about a patch:

- Why is it necessary? It may fix a bug, it may add a feature, it may improve performance, reliability, stability, or just be a change for the sake of correctness.

- How does it address the issue? For short obvious patches this part can be omitted, but it should be a high level description of what the approach was.
- What effects does the patch have? (In addition to the obvious ones, this may include benchmarks, side effects, etc.)"

Workflow Details (GitKraken GUI)

Our team utilized GitKraken to manage commits. To learn GitKraken, details are added here for convenience, but a full walkthrough should be viewed on the software's website: GitKraken's Start Guide

1. Add a profile to identify your commits
    a. File > Preferences > Profiles > Add A Profile
2. Clone the repo
    a. File > Clone Repo > URL: https://github.com/KukukBrad/licht-cpp.git
3. Create a **new branch** each time you start to develop new functionality or work on improving code.
    a. Double-click `master`
    b. Pull
    c. Select/highlight latest commit
    d. Branch > enter name
4. Work on code
    a. Whenever a significant enhancement or issue arises, or when you think one will arise in the future, document it via an issue, and assign that issue to the milestone.
    b. A good rule of thumb is that other group members should know what you are working on at any given point in time. After initial functionality has been developed, you should aim to document every significant change that will need to be done before the branch can be merged to master. Close the issue when it has been resolved.
    c. As issues get resolved and features get added, *commit* your changes.
5. **Commit** to your development branch regularly and use explanatory commit messages in order to create a transparent work history (e.g., to help with debugging; to find specific changes at a later time). Each commit is a separate logical unit of change and is therefore composed of related changes.

    a. Changes appear on the right pane. When closed, you have to click "View changes" on the top right.
    b. Stage your changes for a specific commit
    c. Write a commit message
    d. Write a summary (should be < 50 characters)
        i. The description should be detailed and can also include keywords to close/fix/resolve issues (e.g. `Closes #45` will close issue #45 in the repository)
    e. *Strongly* follow these guidelines for writing a message
    f. Commit
6. Push commits
    a. Make sure you are on the development branch (double-click it)
    b. In case someone else is working on the same development branch, then import/merge new commits from remote/upstream to local branch (Pull)
    c. In case the master branch has changed considerably or contains important updates, then rebase with master
        i. Right click REMOTE : master > Rebase <branch> onto master
        ii. Remove/resolve conflicts, mark the resolved files by adding/removing them, then continue with the rebase
        iii. Commit changes
        iv. Push/export local project history to remote/upstream (Push)
7. Repeat steps 4-6 until the development branch is ready for deployment. Open a pull request and assign at least one team member as a reviewer. It usually does not hurt if someone other than the developer does a test on a feature, since another person may test differently.
8. If necessary, repeat steps 4-6 to complete review of the pull request, e.g. to deal with issues and fix bugs.
9. After a team member has reviewed the development branch and it has passed all tests, you're almost ready to merge. Pull requests can take some time, and during development it's possible that master will change. If this has happened, use GitKraken to rebase onto master. You may have to solve some conflicts. Afterwards, force push. Avoid using GitHub's "Update branch" button in the pull request. Once the tests pass, you can click merge. After merging, delete the branch and milestone.
10. Increment and add a tag to the master branch’s latest commit (i.e. your merge commit)

Appendix C: Style Guide

Styling for ellipsoids is derived from the previous development team, which follows Google's C++ style guide. Separate styling was used for NLM and GUI. Minor exceptions to Paired Planet's commenting scheme are allowed, such as when it makes more sense to exceed 80 character lines than to make it multi-line.

Doxygen

For generating automatic documentation, *doxygen* uses special comment formats. In this code base, the following format is used:

```
////////////////////////////////////////////////////...
/// Information here
////////////////////////////////////////////////////...
```

One line before and after the comment block is filled with forward slashes up to 80 characters, and comments within the block are triple slashes. Never use traditional block comments (/*).

In-line comments

Start with a space and an uppercase letter. Before the comment is two spaces, unless it needs more to align with other comments.

Spacing

Use 4 spaces instead of tabs. To align comments, use spaces after the code and before the comment.

Other

Without following a large guide, it is impossible to lay out exactly how we program. We just accept that there will be some variations in development style outside of the above rules, but an active effort should be made to match the lead developer and base style of the repository.

Appendix D: Note to Future Developers

As stated in the process overview, this project's history lies in our GitHub repository. It is extremely beneficial to continue using the same repository so that the project's history is retained. Please contact Brad for access to the repository.

Depending on your goals, it may be best to fork the repository, but ideally the project would live on in the same repository and continue to grow, eventually becoming open source. To contribute to this repository, any developer would have to follow a process similar to what we used to remain consistent and professional. We would be happy to aid in the learning process, if needed.

All team members from both Paired Planet and Team Andromeda are available for contact at any time, so please do not hesitate to ask questions about this document or the code. We are committed to ensuring this solution is high for years to come.

Appendix F: Detailed Documentation

Finally, attached is the documentation for our codebase. We write comments in a particular way (doxygen format) that allows a parser to extract it and generate extensive documentation (websites, man pages, LaTeX, etc.). Appendix A explains how this documentation can be generated. We have compiled a LaTeX version and attached it here for your convenience. It is recommended to view it digitally to utilize the hyperlinks. This is a great resource for others to verify publications created using this software, as the source code is included. This also means to be cautious when sharing this appendix while the code base is private, as the entire code base can be reconstructed using it. For this reason, it is included separately from this document.