Software Testing Plan

Version 1

4/3/19

Sponsor
Lowell Observatory

Clients
Audrey Thirouin, Will Grundy

Team
Paired Planet Technologies
Zach Kramer, Brian Donnelly, Matt Rittenback

Mentor
Isaac Shaffer

Table of Contents

1. Introduction

Space exploration has always enthralled humanity. Every year, billions of dollars are spent trying to understand our Solar system. Humanity sends probes to other planets and to small celestial bodies. Humans have not stopped looking for answers in space since the time of Galileo. This search has driven technological development in all areas. Velcro, computers, and GPS are just some of the technological byproducts of space exploration. Every day, observatories like Lowell are gathering information to expand the understanding of the solar system.

The clients, Dr. Audrey Thirouin and Dr. Will Grundy, work at Lowell Observatory to analyze astronomical data. Lowell focuses on analyzing data of small celestial bodies that are challenging to directly observe. Right now, they are working on modeling binary systems in the Kuiper Belt. To do this, they need to use special techniques which make the most use of the data available. For most objects that far away from Earth, only a single point source can be observed. That point source can be used to determine the object brightness at a given point in time. These luminosity recordings can be combined together to form a graph called a light curve, which is a display of luminosity over time.

Light curves can be used to infer properties about the objects that generate them. For example, an asteroid that is non-spherical will reflect more light when a larger amount of surface area is reflecting light from the Sun to the observer. Since the object is reflecting more light at certain points in its rotation, the brightness will be different depending on when it is observed in its rotation. Those brightness values can then be graphed. In most cases this will make the light curve sinusoidal. The rotational speed can then be found based on the period of the curve. The amplitude of the curve can be used to roughly guess at the proportions of the object. A large number of other characteristics can be inferred using light curves.

The clients want to make use of light curves to better understand binary systems, since they are extremely prevalent in the Kuiper Belt. Binary systems are composed of two objects that orbit each other about a point in space called a barycenter. The gravity from both objects affect each other, introducing new challenges in creating a model.

The clients want software that can model these binary systems and calculate light curves. They plan on using this project to generate potential models that fit the observed data of binary asteroid systems. This project solves these problems by creating an integrated API. This API provides a single function that takes in all of the parameters at once and returns a light curve. It will also utilize a custom ray-tracer that is more efficient than the current solution and produces more accurate luminosity values. This increase in speed, accuracy, and ease-of-use will advance Lowell's research efforts to analyze binary systems in the Kuiper Belt.

To ensure a robust product is delivered to the client, the project will be tested extensively with 3 different types of tests. For the first type of testing, unit testing will be used to test and verify that each individual module of the codebase functions properly. This type of testing is important to ensure the calculations are correct and that the shapes are created correctly. After the testing of individual modules, the next type of testing involves integration testing. Integration testing tests the integration of the all of the modules and ensures that different modules are correctly functioning together. In this project, the single API function, the forward model, acts as one large integration test in that it calls all other modules in the codebase. Lastly, the team will conduct usability testing to ensure that the client is satisfied with the functionality of the project. This will also allow them to utilize the codebase for research and for sharing with other astronomers.

This testing plan closely follows the team's development plan, in that all of the individual modules were created first and established that each module functions correctly on its own. Then the team moved onto integration and developed the Forward Model and Ray Tracer, which serve as integration tests as these modules utilize all other modules in the code base. Usability testing has not been possible until after the delivery of the alpha prototype, since the client wants a single function to call that returns a light curve.

The team agreed to implement testing for each of the modules as they were developed to ensure the best product for the client. Since the team implemented the tests alongside development, the testing plan closely matches the development plan and gave the team confidence in the functionality of the modules. The testing for each step of the testing plan is outlined in the following sections.

2. Unit Testing

Unit testing is a practice in which individual functions of software are tested for correct functionality. Most of the unit tests are simple input and output verification, such as testing if the software accepts only valid inputs and verifying correct results from the astronomical calculations. Some unit tests will simply test the creation of certain objects and verify that these objects are being populated with data correctly. Due to the scope of astronomical calculations being used in the project, the client provided the team with equivalent IDL routines to calculate reference values in the testing.

For this project, Google Test is the chosen unit testing library as it is a well-known, cross-platform, and respected library, and functions with the C++ API codebase.

The project was designed to group all of the calculations and components of the solution into individual modules. Each of these individual modules have unit tests written for them to verify their functionality for use inside of the Forward Model. For some of the modules that were translated from IDL, some input is provided to IDL in order to establish a reference value for the C++ module. These values from the different modules are then compared with high precision ranging from 0.01 to 0.000000001 depending on the test. For the rest of the modules that were not translated from IDL, the team had to analyze the output and analyze whether or not the output was correct. After determining the output is correct, that output is saved as a reference value. Most calculations are done to double precision. Due to how machines represent floating point values, it is incorrect to expect values to be equal. Instead, computed values and reference values must be checked to be within a bounding range of each other. For us this often means extremely high precision, though some tests have more pronounced variations and their precision has been adjusted to match that.

2.1 Calendar

The Calendar module defines functions to convert and manipulate dates. It is not currently used by any other module, but exists for future functionality. There are no poorly-defined calculations here, just specific formulas.

1. JulianDay Test
    a. **Description**: Performs conversion of entered Gregorian date into a Julian date and verifies the result
    b. **Main Flow**:
        i. Call GetJulianDay function to convert an entered Gregorian date
        ii. Compare calculated Julian date value to expected value
            1. Input: February 14th, 2013 at 6:00:30
            2. Expected Output: 2456337.750347
                a. Reference value is calculated using the U.S. Navy's Julian Date converter
    a. **Expected Outcome:** Entered Gregorian date is converted into a corresponding Julian date within 0.01 precision

2.2 Ephemeris

The Ephemeris module reads in ephemeris tables. The danger with handling raw file paths is that the user could provide an invalid file path in many forms. After ensuring the path is a valid text file, it is assumed that the user provided a real ephemeris table in the correct format. Thus, only the calculation functions and interpolation functions are left to test.

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. PhaseAngleVariable Test
    a. **Description**: Upload phase angle value for each observation in ephemeris file

b. **Main Flow**:
    i. Create ephemeris object to read in and contain entered ephemeris data from valid ephemeris files
    ii. Compare and validate phase angle values from the ephemeris object
        1. Example Input: ephemeris.phaseAngle[50]
        2. Expected Output: 1.1126
            a. Expected output is retrieved from line 50 of the table inside ephemeris file

c. **Expected Outcome:** Creation of phase angle variable in ephemeris object

4. TimesVariable Test
    a. **Description**: Upload time value for each observation in ephemeris file
    b. **Main Flow**:
        i. Create ephemeris object to read in and contain entered ephemeris data from valid ephemeris files
        ii. Compare and validate time values from the ephemeris object
            1. Example Input: ephemeris.phaseAngle[50]
            2. Expected Output: 2458507.772002259
                a. Expected output is retrieved from line 50 of the table inside ephemeris file
    c. **Expected Outcome:** Creation of time variable in ephemeris object

5. Interpolation Test
    a. **Description**: Perform interpolation with given dates and ephemeris file
    b. **Main Flow**:
        i. Create array to contain entered dates to interpolate with
        ii. Create ephemeris object to contain entered ephemeris data from valid ephemeris files
        iii. Calculate interpolated values through calling the Read function with an updated dateCheck variable that provides the value to interpolate with
        iv. Compare interpolated values to expected values
            1. Expected values were calculated by hand with a preselected date
    c. **Expected Outcome:** Test passes to prove the function is performing interpolation calculations correctly within 0.00001 precision.

2.3 Hapke Model

The Hapke Model module contains functions to compute Hapke's bidirectional reflectance model. This is perhaps the most mathematically complicated module in the project. In the past it has brought to light precision and compiler environment issues.

1. PhaseAngle Test

   a. **Description**: Verify functionality of custom phase angle function by comparing and validating the results with expected results
   b. **Main Flow**:
      i. Create double variable to contain phase angle
      ii. Call custom Henyey Greenstein overload function to compute phase angle
      iii. Compare phase angle value to expected value from IDL routine equivalent
         1. Expected value is retrieved from the IDL equivalent of this module passing the same input
   c. **Expected Outcome:** Test passes to prove the function is performing calculations correctly with a phase angle output that matches the equivalent IDL routine call output within 0.00000001 precision

2. Bidirectional reflectance (doubles) Test
   a. **Description**: Calculate and verify bidirectional reflectance calculations
   b. **Main Flow**:
      i. Create array to contain phase angle function parameters
      ii. Create double variable to contain bidirectional reflectance value
      iii. Calculate the bidirectional reflectance value through the BidirectionalReflectance function
      iv. Compare bidirectional reflectance value to expected result
         1. Expected result is retrieved from the IDL equivalent of this module parsing the same input
   c. **Expected Outcome:** Test passes to prove the function is performing bidirectional reflectance calculations correctly within 0.000000001 precision

3. Bidirectional reflectance (bad cos angle) Test
   a. **Description**: Calculate and verify bidirectional reflectance with 0 degrees for the emission and incidence angles
   b. **Main Flow**:
      i. Create array to contain phase angle function parameter
      ii. Create double variable to contain bidirectional reflectance value
      iii. Calculate the bidirectional reflectance value through the BidirectionalReflectance function
      iv. Compare bidirectional reflectance value to expected result
         1. Expected result is retrieved from the IDL equivalent of this module parsing the same input
   c. **Expected Outcome:** Test passes to prove the function is performing calculations correctly even if cos has a value of 1. This value of cosine is accounted for by performing a special check to reduce the cos to 0.9999999 with the computed value being within 0.000000001 precision

2.4 Kepler True Anomaly

The Kepler True Anomaly module contains a single function to compute Kepler's equation for an elliptical orbit. The calculations are in this module are complex and use Newton's method to compute the anomaly, although not much can actually go wrong in this function. Thus, only one correct output value is tested for in the use case.

1. KeplerTrueAnomaly IDL Test
    a. **Description**: Verify functionality of KeplerTrueAnomaly module by comparing and validating the results with the results from IDL
    b. **Main Flow**:
        i. Create array to contain expected results from IDL
        ii. Create array to establish and contain eccentricity of the orbit
        iii. Create vector of time values with units of fractional period since periapse passage
        iv. For each eccentricity value, solve Kepler's equation through the KeplerTrueAnomaly function
        v. Compare the results from calculating Kepler's equation to the expected IDL results
            1. Expected results are retrieved from the IDL equivalent of this module parsing the same input
    c. **Expected Outcome:** Test passes to prove the module is performing calculations correctly within 0.000002 precision

2.5 Locations

The Locations module calculates the Cartesian locations of the binary system relative to the observer. This is another relatively straightforward module where not much can go wrong.

1. Locations Test
    a. **Description**: Calculate the cartesian locations of the binary system relative to the observer
    b. **Main Flow**:
        i. Create orbit object with no precession
        ii. Create a vector of time values that represent consecutive observations
        iii. Create a vector of delta values that represent the distance to the target in AU
        iv. Create a vector of right ascension values in hours
        v. Create a vector of declination values in degrees
        vi. Create view object containing the delta, right ascension, and declination vectors
        vii. Create location object to find locations of bodies at the given times

        viii. Compare location value to expected value
            1. Expected value is retrieved from the IDL equivalent of this module parsing the same input
    c. **Expected Outcome:** Creation of a location object with values that match the IDL equivalent within 0.0001 precision

2.6 Orbit

The Orbit module handles Keplerian mutual orbits. Orbit is essentially a data structure, but holds some functionality for future cases.

1. Pole Test
    a. **Description**: Report the direction of the orbit pole
    b. **Main Flow**:
        i. Create double variables to contain right ascension and declination values
        ii. Create orbit object with no precession
        iii. Create a pair to contain and compare the ecliptic values to the expected values
            1. Expected values are retrieved through the DirectionOfPole function with right ascension and declination values passed in as parameters
    c. **Expected Outcome:** Creation of a orbit object that correctly returns the direction of the orbit's pole within 0.000001 precision

2.7 Shape

The Shape module defines what a shape is, which basically represents something "hitable" and tracks when a ray of light would hit the shape. This module handles the creation of the shapes, as well as the orientation, spinning and setting of them in space. There are 3 different shapes, a "Facet", a "Sphere", and an "Ellipsoid" shape. Both spheres and ellipsoids are generated shapes that should be used for fitting. A faceted shape is read-in from a provided Wavefront file and should be used for specific systems that have fly-by observations.

1. UploadIrregular Test
    a. **Description**: Create and verify a faceted shape object from a valid input file
    b. **Main Flow**:
        i. Create shape object with provided valid obj file
        ii. Check number of facets and vertices in shape
        iii. Compare position values of vertices to expected values
        iv. Compare values of facets to expected values
    c. **Expected Outcome:** Creation of a valid faceted shape object
    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
f. **Alternative Flow:** User enters a file that does not exist
g. **Expected Outcome**: Error message reporting the user entered a file that does not exist

2. GenerateShape Test
    a. **Description**: Create and verify a sphere object
    b. **Main Flow**:
        i. Create sphere object
        ii. Check center of sphere to see if it is still at the origin
        iii. Set values to the center of the object
        iv. Compare center of sphere for expected values
            1. Example Input: shape.SetCenter(Eigen::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
    c. **Expected Outcome:** Creation of a valid sphere object

Unit testing is critical to this project in ensuring that all individual modules are functioning properly and produce correct results to verify the astronomical mathematics used. Once these individual modules are verified, they are ready to be integrated with other modules and tested as a complete system.

3. Integration Testing

Integration testing is used to expose problems with the interaction between modules. While each module of a codebase may function, 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.

The tests need to check the integration of all of the modules and submodules. Parameters need to be tested for correct units of measurement and correct data types. For example, some parameters should be in arc hours, minutes, and seconds, while other should be in degrees or radians. The code base uses kilometers for all distance units, but at times distances are also measured in ratios.

There are many approaches to integration testing, though most revolve around an idea of calling individual unit tests together. This becomes quite complicated in this code base, because combining all of the unit tests is essentially equivalent to writing the forward model. Because of this, the team utilizes a strategy called Big-Bang testing. Big-Bang testing combines the functionality of all of the unit tests at once, forming a complete system. The reasoning for using this approach is because the purpose of the software is to support a single callable function from within IDL. This function, the forward model, is already the “complete system”. This has the benefit of representing real-world use cases, but the downside of making it hard to isolate any errors found. For this integration testing plan, the forward model, along with another, smaller integrated module, are called within a test environment and have their outputs verified.

3.1 Integration Testing Plan

Throughout the design process, integration tests have been implemented when adding new modules to the workflow. The integration tests are essentially small portions of a use case that focuses on testing a particular module interaction. The testing plan revolves around two key modules of the code the forward model module, which integrates with IDL, orbits, and the tracer modules; and the tracer module, which integrates with Hapke, the forward model, shapes and a large number of submodules. Inside these two main integration testing areas a number of different tests are used to check that each way that the module can be used returns the correct values, and that the modules are receiving the proper parameters at the start.

3.1.1 Tracing module integration testing

The tracing module integration testing focuses on being able to output a single rendered image and return a single intensity value for a light curve. There are a number of testing scenarios used to ensure that submodules are returning expected values.

Table of Trace Module Integration Tests

Test number	Multiple shape renders	Multipl e Objects	Generated shape	Uploaded shape	Modified Hapke	Anti Aliasing	Scale and move	Long distance
1		X	X					
2			X					
3				X				
4		X	X			X		X
5		X		X	X		X	
6	X	X	X					

The six tests for integration all generate at least one rendered image. While the code base is in a stable development state, these tests are run to generate reference values. These reference values are then set in the test. As more development is done, any major deviation from the reference values can indicate that the code is not functioning as intended. These six test cases do not check every possible use case but they do cover a wide range. As the code is continuously developed these test will catch the majority of errors that a change may cause. The final integration testing can be made more robust by trying all possible test cases with the above criteria.

3.1.1 Forward model module integration testing

The forward model integration testing focuses on feeding different parameters and scenarios to the tracer module. The forward model also tests other submodules that modify the system between observations. The forward model has two main integration tests, one for the Sila Nunam system and one for the moon. Both tests can produce a large number of rendered images for debugging.

The Sila Nunam system is an observed system in the kuiper belt. By using a real use case, flaws were exposed that might have otherwise been missed. The Sila Nunam test has two objects at a great distance from the observer. Because of this distance, errors in precision can be easily detected. The two objects experience a mutual event within the observational timeline and create a light curve that shows this. The light curve values are recorded when the code is stable. Any future changes that somehow alter the light curve will cause the test to fail. The objects are generated spheres so no objected rotation can be observed.

The moon test has a single object extremely close to the observer, the opposite of the Sila Nunam test case. The test creates a single faceted shape with the parameters of Earth's moon. We expect the lightcurve to represent the phases of the moon. This is tested by again establishing a reference light curve and then comparing future results after changes. Because the shape is faceted, an amendment to this test is to apply an orientation and rotation to the shape based on the Moon's real data. When establishing a reference value, the spin can be checked against the real Moon's observed spin for the given dates, and this reference value can then be used in further tests.

Integration testing is essential in ensuring that modules interact properly and that the glue code 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

4. Usability Testing

Usability testing extends into the realm of the users. Instead of testing if pieces of *the code* work together, the test is now if *the user* and the code work together. This means that real-world testing has to be done, and that this testing is more social and statistical than unit and integration testing.

The main goals of usability testing are to:

- Optimize the ease-of-use and flow of the solution
- Make sure the user/client is happy

This is a somewhat open-ended concept and ultimately comes down to making the user happy. In order for the user to be happy, the solution should be easy to learn and work with, and its functionality has to meet or exceed their expectations.

4.1 Preface

Usability testing is most commonly known for testing user interfaces, since they can be quite complicated yet are easy to objectify (e.g. five clicks to access a menu item is considered poor design). For this lightcurve modeler, usability testing requires a bit more thought. APIs are not exempt from usability testing, far from it. Developers are users, too, and the design of an API is just as important the “user-interface” workflow.

This API is quite challenging to test, though, because only a single, agreed-upon function is exposed. The parameters are non-negotiable and were provided by the client. The API is not elegant, it just takes in a variety of parameters and outputs a lightcurve. Things that *are* negotiable are additional features, additional documentation, potentially additional parameters or arranging the existing ones in different ways, etc. The client has been quite straightforward in that they anticipate only wanting to tweak or add features and that the “design” of the API is unimportant, since there is no real design. They also do not care much about the underlying design of the code. To provide an example, the client may want a different way to tell the API that they want to use the ephemeris dates and not the custom dates they provided, or they might want additional output values.

In an ideal world, usability testing is done before the code is complete. In many software projects, such as ones that have a GUI component, iterative prototypes can be made without having full functionality and still be tested. For instance, a GUI can be evaluated without its backend. This project requires *everything* to work, due to the requirements. The client is only interested in calling a single function and getting a result. Until they can do this, they are not able to effectively evaluate how they may want to interact with it differently. It is possible to draft up an API before implementing it, though even then the parameters were only partially known and, again, not negotiable. There are many subcomponents that have been internally tested along the way, such as finding the locations of these binary systems, but the client is not interested in testing that. Thus, usability testing has not been possible until now -- after the completion of the alpha prototype.

4.2 Ease-of-use / Flow

Ease-of-use is perhaps the most important aspect to the client. Their current solution is extremely difficult to use and requires many custom steps to predict a lightcurve. The core requirement of this solution is to have a single function to call, which by comparison already satisfies the majority of the “ease-of-use” category.

Another aspect of what makes an API easy-to-use is the parameters. Parameters need to take into account intuitive design but especially the requirements they are imposing on the user. Requiring the user to pass in the locations of the binary system would make the solution much harder to use than calculating this data internally. Discussions have already been had with the client about what the parameters should be, and how much work is expected from the user. Thus, the current requirements have already been agreed upon and will not change, barring minor tweaks.

What has not been tested is how intuitive the parameters are. There are many parameters, over forty at the moment. It is easy to forget what a parameter does or how to use it. The *types* are not negotiable, but documentation, naming, combining parameters, etc., is. To test their ease-of-use, a heavily-documented example call to the forward model from IDL has been provided to the client. The client has been requested to model some of their favorite binary systems and to provide feedback about:

1. How much time it took to create the parameters needed for the call
2. Any issues they ran into, such as the orbital parameters not resetting between calls
3. How intuitive the parameter names were
4. Where more documentation would be useful

This is not a formal survey-like test, this is simply human-to-human feedback. In the end, the client is familiar with the parameters they are passing in, and there is not much optimization to be done in the API-call itself, so usability testing does not play a large role here.

4.3 Client Satisfaction

The client’s satisfaction mostly revolves around the functionality that is provided to them. The clients have provided a priority list of nice-to-have features. The more features that are implemented, the happier they will be. Based off this list, we already have base-line satisfaction, and the more features we implement on it, the happier they will be.

This all of course depends on the solution being both robust and accurate. Thankfully, much of the formula-heavy code has been translated from IDL routines and the results have been verified. The ray-tracing solution is likely more accurate because it traces by pixel and does not require as much precision. Thus, the satisfaction mostly depends on the features available to the client.

To test that the clients are satisfied with the solution, the alpha prototype has been given to them with a list of features. The approach here is near identical to the ease-of-use testing. This time around, the client has been requested to provide feedback on the *features* instead of the ease-of-use. Their satisfaction depends on how well the features meet their use case. There is not a proper test for this, per say, since the clients have already described that the solution exceeds their expectations.

5. Conclusion

Space exploration has always enthralled humanity. Every year, billions of dollars are spent trying to understand what is in the Solar system. Binary asteroid systems have always been challenging to understand due to their small size. The clients, Dr. Audrey Thirouin and Dr. Will Grundy, work at Lowell Observatory on understanding binary systems in the Kuiper Belt. They accomplish this by using software that models binary systems.

One challenge is that these systems are so far away that only a single point source can be observed. The required calculations to account for this are relatively complex and the clients have only had time to develop a partial solution for the modelling. This partial solution is slow and fragmented, meaning the functionality is spread apart and scripts have to be written to utilize the code. The clients would like an API that is significantly faster and integrated, meaning they can receive the model in a single function call in a code base that follows consistent coding and naming practices. The best solution is a modularly designed, high-performance C++ code base with a single exposed function that returns the desired results.

Extensive software testing planning has been done for this document, detailing the unit testing, integration testing, and usability testing strategies for the project. Through the unit testing the team can verify the functionality of the codebase’s individual modules, such as checking the math of an astronomical module or checking user’s string inputs. For integration testing, this project is unique in that the major function call of the project, the Forward Model, is essentially one major integration test. This Forward Model is a form of Big-Bang testing and calls all of the other modules in the project. Due to the project requirements requiring all of the functionality to be implemented at once, for usability testing, we could only focus on the features being provided to the client and how these features are presented and organized.

Robust testing is always necessary in creating software that is easy-to-use but also handles both human or machine errors and provides verification of its calculations. Software testing is used to help developers ensure that a single function performs its single task or a full code base integrates properly. By writing this software testing plan, we are confident that the product is performing the right calculations and individual modules integrate correctly in the Forward Model. The goal of this testing is to provide the client with a robust solution that solves not the original problem, but also handles unseen issues and edge cases. Paired Planet Technologies is confident in its completed alpha prototype and is looking to complete a extensive user guide to finally bring the client a highly-usable software package they can share with other astronomers.