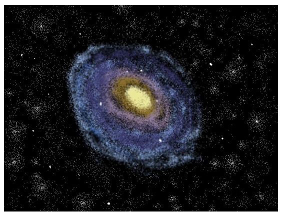

Team Andromeda

Version 1

Software Testing Plan

April 3, 2020

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

Table of Contents

1. Introduction

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

Our clients from Lowell Observatory, Dr. Audrey Thirouin and Dr. Will Grundy, study objects near or in the Kuiper Belt. Dr. Thirouin is a research scientist interested in the characteristics of small bodies within the Solar System, while Dr. Grundy is an astronomer that researches Kuiper Belt objects. The Kuiper Belt is a region containing leftover bodies from the solar system's early history. This makes these celestial bodies valuable for observing conditions similar to that of early planet formation.

Our clients are interested in binary asteroids: two asteroids that orbit each other. However, these systems are so far away that they are only observable by the fluctuation in the brightness of sunlight reflected back at Earth. To observe these fluctuations, the magnitude of light is measured over a period of time, which forms a graph called a light curve.

To extract information about binary systems from their light curves, our clients first use software designed to model binary system physics. This modeling software requires several astronomical parameters and generates a simulated light curve. By comparing simulated light curves with observed light curves, they gain information about these binary systems. Though this software is currently capable of basic modeling, our clients wish to expand it to reach its utmost potential.

To ensure a robust product is delivered to the client, the project will be tested extensively with 3 different types of tests. Firstly, unit tests will be used to verify that each new feature of the codebase functions properly. This type of testing is important to ensure shapes are rendered correctly and that the GUI and amoeba are handling data correctly. After testing individual features, integration tests will be implemented to validate the unification of all new features and that they are correctly functioning with the original modules. The single API function, known as the forward model, can be used as one large integration test because it calls all other modules in the codebase. The forward model will be used to test integration for the new ellipsoid shape submodule. The team will also test the GUI and amoeba integration with regards to how well they utilize the forward model call. Lastly, the team will conduct usability testing to ensure that the client is satisfied with the functionality of the new features. 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, which established that each feature would function correctly on its own. Since that was established, integration was innate to the added features. The ellipsoid module was integrated as it was built, and the GUI and amoeba features integrated the forward model call to function properly. The team agreed to implement testing for each of the features as they were developed to guarantee the best product for the client.

The team implemented multiple tests alongside development and has currently moved on to more unit testing to confirm that each feature is functioning as required. Usability testing has not been achieved yet due to the team gathering continuous feedback from the clients about their wants and needs. In addition, the team is working on unit tests that will ensure the product is working properly prior to client testing. Each test for Team Andromeda's software testing plan is outlined in the following sections.

2. 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 amoeba non-linear minimization are within an expected range. Other unit tests will check if the creation and rotation of triaxial ellipsoids are valid.

To limit the expected output, the team is utilizing the same testing data utilized by the previous development team, Paired Planet Technologies. These data will be in the form of ephemeris files, object files, and pre-existing observed data. To standardize unit testing, Google Test is 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.

2.1 Ellipsoid

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

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

2.1.1 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 consequently ensures that the ray tracer is able to trace an ellipsoid from different rotations.

2.1.2 Ray Tracing

2. 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

2.2 Amoeba

To guarantee that the implementation of the amoeba module is performing as expected, a series of unit tests using the Google Test framework will be employed. These unit tests will be contained in the TestAmoeba.cpp file stored in the test folder. The unit tests for the amoeba module will verify that any data passed in and returned via text file are accurate. Additionally, they will ensure that the comparison of light curves and estimation of chosen forward model parameters are being performed correctly.

2.2.1 File Input

The amoeba module will depend on two functions that handle reading input from the user. This input will be in the form of an observed data file and a user input file. The observed data file will contain 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 will contain the values that the clients are interested in putting into amoeba. 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 amoeba.

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

2.2.2 Parser

The amoeba module will depend 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.

5. 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

2.2.3 Minimization

The amoeba module will depend on a non-linear minimization function found within the amoeba.h file. This file was created by the authors of *Numerical*

Recipes 3rd Edition: The Art of Scientific Computing [1] 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/amoeba/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 non-linear minimization function to have been tested thoroughly enough such that additional unit testing for this function is not required.

2.2.4 Plotting

The amoeba module will depend 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[2] and documentation[3] 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 will be done in integration testing.

2.3 Forward Model Graphical User Interface

Since the functionality of the GUI is dependent on working with the forward model, unit testing is 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 graphical user interface works appropriately with all of the necessary modules, a majority of the testing for this module will be completed using integration testing.

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

2.4 Video Generator

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

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

3.1 Ellipsoid

Much of the integration tests came naturally with the addition of the ellipsoid submodule. The forward model call will output a light curve and render the ellipsoid image. Without proper integration from the beginning, rendering and using ellipsoids would not be possible. It is 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 will be implemented in TestForwardModel.cpp. The goals of these tests are to render viable asteroid images and generate feasible light curves. As it is of utmost importance to achieve these goals, there will be 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, will be utilized again to test whether the rotation is proper when rendered as ellipsoids. A singular asteroid called Roxane will be added to the ellipsoid tests as an added measure for rotation and accuracy. This will prove that binary systems, as well as a singleton asteroid, can be rendered and manipulated correctly, which verifies that the clients can gather the necessary data. Subsequently, these tests will produce a light curve which will be verified for accuracy.

3.1.1 Forward Model

1. ForwardModel Test
    a. **Description:** Runs the forward model using Sila Nunam with ellipsoids, as well as the single asteroid Roxane with an ellipsoid
    b. **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
    c. **Expected Outcome:** The forward model correctly implements the ellipsoid module to produce an accurate model of Sila Nunam and Roxane
    d. **Alternative Flow:** The user does not set the ellipsoids to rotate
    e. **Expected Outcome:** The forward model renders the ellipsoids the same as before, but without any rotations occuring

3.2 Amoeba

Amoeba's unit tests can be completed during the same workflow, resulting in simple integration testing. The user will input the observed data (which represents the observed light curve) and user input file into the amoeba module. Amoeba will call the forward model to produce the predicted light curve which will be compared to observed data. After the comparison, the amoeba module will conduct its main functionality which results in the execution of the unit tests implemented for the amoeba module.

Testing amoeba in conjunction with the triaxial ellipsoid submodule will only require a change in the parameters passed to the forward model through amoeba. Since the forward model GUI has no interface with the amoeba module, integration testing between these two modules is not currently possible. Lastly, when testing integration on amoeba with the video generator, only a flag will need to be set to produce rendered images within the amoeba 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 amoeba module, the addition of the following tests is necessary. These tests will ensure that the functionality needed for amoeba's interaction with other licht-cpp modules, as well as its own functions, is behaving as expected.

3.2.1 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

3.2.2 File Input, Parser, Minimization, and Plotting

2. AmoebaModule Test
    a. **Description:**
        i. Run amoeba 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 amoeba
    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

3.2.3 File Input, Parser, Minimization, Ellipsoid and Video Generator

3. Amoeba, Ellipsoid and Video Generator Module/Submodule Test
  a. **Description:**
    i. Run amoeba with read in and correctly parsed data, and verify that the produced output is correct
    ii. The read in data will be tester data attributed to ellipsoid objects, thus testing the amoeba module using light curve data derived from ellipsoids
    iii. Images will be rendered using the tester data and these rendered images will be compiled into a video using the video generator
  b. **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
  c. **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

3.3 Forward Model Graphical User Interface

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

3.3.1 Acceptable Parameter Test

To ensure that the GUI is accepting parameters that are valid within the Forward Model, the test will verify 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 will be 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

3.3.2 Plotting

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

2. 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

3.4 Video Generator

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

3.4.1 Image Verification

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

1. ImageVerification Test
    a. **Description:** The test will consist of making videos from several series of renders that are output when using the GUI or amoeba
    b. **Main Flow:**
        i. The set of images will be rendered after the running of either the GUI or amoeba
        ii. The batch script will be 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 will not be removed

1. The speed that the video will be run will be set to the default speed
2. The speed that the video will be run will be set to double the default speed

iii. The .mp4 will be created

c. **Expected outcome:**
   i. The images will be in the correct order in the video
   ii. The video will be running at the default speed
   iii. The images will 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.

4. Usability Testing

Usability testing is used to assess how well the user is able to navigate the software. This means that real-world testing will need to be done, and that this testing is more social and statistical than unit 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 will be provided documentation that walks them through each feature’s usage. The documentation will first focus on each feature individually, such as how ellipsoids are made and used. This will help give the users a broad idea of how the individual features function. Lastly, the documentation will focus on a walkthrough of how to use the features holistically.

Specifically, this walkthrough will include how to navigate the GUI, how to generate ellipsoids, how to create a video based on the generated ellipsoids, and how to effectively utilize amoeba. This will allow for a greater understanding of the software’s effectiveness and how it will fit into the users workflow. Usability tests are a vital part of testing for the software and will provide the team with feedback needed to improve the cohesiveness of the software.

4.1 Ellipsoid

Ellipsoids were made to be used within the forward model, unlike the GUI and Amoeba features. GUI and amoeba differ because they both utilize the forward model function call to perform their operations. As ellipsoids are nested within the forward model, usability testing will be performed during the usability tests for the GUI and amoeba features. Thus, ellipsoids do not have their own specific usability test, but rather have had their usability embedded into the forward model itself.

Expected Outcome: The GUI and amoeba features will be able to effectively use the forward model to accurately create and utilize ellipsoids.

4.2 Amoeba

Since there is complexity involved with the implementation of the amoeba module, testing with the clients is necessary to ensure it is functioning as they require. To conduct this testing, the team will provide documentation on how the amoeba module works, and how the module estimates the best fitting parameters.

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

Expected Outcome: While the data produced by the amoeba module will differ based on the input provided, the module will be expected to produce data that is acceptable for the clients. Additionally, the use of the amoeba module will meet 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.

4.3 Forward Model Graphical User Interface

The GUI will connect the users to the software. The best way to test if the users can navigate the GUI is to let them use it independently. To help the users evaluate the GUI, documentation will be provided, which will explain to the user 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 allows the users to get hands-on experience with the GUI. This test is vital for evaluating the GUI's usability, and will give the team a greater insight on how the clients will use this feature within their workflow.

Expected Outcome: The team will allow the clients a set amount of time to go through the section of documentation for the GUI. Once they have gone through the section, they will have 10 minutes to try using the Sila Nunam data. Finally they will be given 20 minutes to use the interface with their own data and run their own tests.

4.4 Video Generator

After allowing the users to test the software, the team will gather their feedback and make any adjustments necessary to have the software meet the clients' standards. While this was covered in the Requirements Specification Document, the clients' feedback on other aspects such as small quality of life adjustments will significantly increase the quality of the software.

5. 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 Will Grundy and Audrey Thouirin at Lowell Observatory solidify these attributes.

To ensure the product can meet the clients' needs, creating specific unit tests, thorough integration tests, and realistic usability tests is a priority. All of these tests are pertinent to evaluating the accuracy of each new module and feature. Ellipsoids must be able to accurately represent real asteroids, the minimization routine of amoeba needs to generate accurate parameter estimations, the GUI must work flawlessly with the existing forward model, and the video generator needs to handle user input and error to create useful videos.

Many of these tests have been created alongside development. However, the team has decided to refactor and add new tests. These new unit tests, integration tests, and usability tests allow for a structured, feasible environment that upholds solid development standards. Formalizing these tests will help deliver a robust solution to our clients in a timely, faultless manner. Team Andromeda is excited to continue working with Will and Audrey and create an efficient, intuitive solution that exceeds their needs.