Contents

1 Introduction

An important part of space exploration is launching satellites into space to orbit planetary bodies, collect large amounts of data, and take images of these bodies. The data and images are sent back down to Earth and used by the planetary science community to create planetary maps. These maps play a crucial role understanding more about these bodies and their surfaces.

Figure 1: Mars Odyssey Map Highlighting Gale Crater

Today, all eyes are set on Mars since it has been chosen as the next stop for manned space exploration. To accomplish this goal, one of the initial missions to Mars was the 2001 Mars Odyssey, commissioned to explore the red planet and transmit its findings back to Earth. It has orbited Mars since October 2001 and is still actively providing essential data today. The Odyssey's primary goals are to provide spatial data of the planet's surface, image surface minerals, and locate potential signs that the planet may have once supported life. See Figure 1 for a map of Mars created using the Odyssey data collected. Without this orbiter and the system of maps it has gifted us, we would have not been able to learn as much about Mars as we have.

The Mars Odyssey mission is just one example of all missions that are carried out. Because there are so many planetary bodies to explore, the planetary science community is very large and consists of scientists, students, and developers from different organizations spanning the globe. Some of these organizations include the United States Geological Survey (USGS), the National Aeronautics and Space Administration (NASA) and 10+ mission teams operating under NASA, other countries' space agencies like the European Space Agency (ESA) and the Japan Aerospace Exploration Agency (JAXA), and 30+ universities. All of these entities play an important role in the research process.

Our client is a small team from the USGS consisting of Trent Hare and Scott Akins. Trent Hare is a cartographer and also works with other companies to try to get them to support the use of planetary data in their programs. Scott Akins is an IT specialist who focuses on web development and databases. They both work for the Astrogeology Science Center (ASC), a subbranch of the USGS, in Flagstaff, AZ.

The ASC consists of developers, students, and scientists all working together to conduct research. Currently, there are approximately 20+ scientists, 20 mission-support/web developers, 20 IT specialists, and 10 students working for the ASC. In the rest of this paper, we will be focusing on the mission-support developers and refer to them as just developers. The ASC is contracted by NASA to support the planetary science community, and they do so in different ways. The developers' main jobs are to create programs that aid in the map-making process and house the data collected during missions so that scientists do not have to store terabytes upon terabytes of data on their own machines. One job of the scientists is to create maps and work alongside planetary missions to conduct important research that would change the future of space exploration forever.

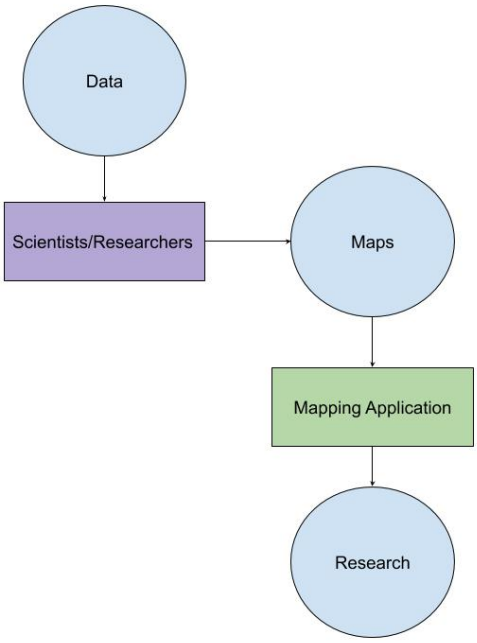

Figure 2: USGS Workflow

1.1 Problem Statement

The main workflow for the USGS scientists when creating maps and doing research is outlined in Figure 2. The scientists and researchers use the data collected during the missions to create maps of planetary bodies. The scientists then use a mapping application to view the maps virtually and are able to finish their research. Because the USGS scientists are mapping many different, complex planetary bodies, there are a few problems with these mapping applications:

1. They only support maps of Earth.
2. They do not allow users to change their latitude and longitude (lat/lon) settings.
3. They do not allow users to change what projection the current map is in.

In order to solve the problems mentioned above, the USGS developers implemented a mapping application using the open-source mapping API OpenLayers (OL). However, the OL implementation has been causing roadblocks for the ASC development team. The problems are:

4. It uses OL version 2 released in 2006.
5. It is not modular.

Because the implementation is 13 years and not modular, it is difficult for the development team to update it. This also makes it difficult to use their code in other mapping applications.

Finally, there is one other problem for both the scientists and the developers. The developers may fix and update their OL implementation, but there is still just one map-viewing application scientists can use to complete their research. The USGS wants to be able to support more scientists by creating more map-viewing applications that will have the functionality needed by the scientists.

1.2 Solution Overview

Our envisioned solution, which aims to solve both the problems of the developers and scientists, is to build a JavaScript (JS) Node package that can be easily installed and used by both groups. This package will

1. Contain a new interactive map built with Leaflet [1], an open-source mapping package, wrapped in a new modern graphical user interface (GUI).
2. Contain a module that wraps the projection and lat/lon back-end code for use in other mapping applications.
3. Allow users to view all bodies supported by the USGS.
4. Contain lat/lon buttons to change the lat/lon settings.
5. Allow users to change the projection of the map.

Please note that we will not be using OL in our solution. Instead, the OL implementation the USGS currently has will eventually use solution /2 when the USGS decides to update their solution.

In addition to the Node package we will be creating, the USGS wants us to create a Python solution to be used in Jupyter Notebooks [2] and a search bar with an auto-complete function to search for all named features on a target.

[1] https://leafletjs.com/
[2] https://jupyter.org/

In order to create a complete solution for the USGS there are requirements that we must follow. To meet coding requirements we must make sure that we are making our project open-source, adhering to Leaflet's coding standards, making sure our Node package is compatible with any web browser, and documenting our code. We also need to follow additional requirements to help the USGS team: follow Web Map Service (WMS) and Web Feature Service (WFS) standards, query USGS's GeoServer, create a user manual, and carry out user testing with the USGS scientists. USGS's GeoServer is where the maps and features are stored. In order to query the server, we will need to follow WMS standards to get maps and WFS to get features. With these requirements in mind, we can examine the implementation of our solution.

2 Implementation Overview

In order to support both the developers and scientists, we are building three different modules: a Node package with a GUI and interactive map, a python interactive map, and an auto-compete search bar. For each of these modules, we will be using a few different technologies.

Suggested by the name, we will be using Node.js [3] to create our Node package. Using Node allows us to create an "environment" that contains all of the dependencies of our package, i.e., Node will install all dependencies for a user. All users will need to do is install our Node package in order to run our interactive map. To compile the code in our Node package, we will be using Babel [4], an open-source JS compiler that compiles JS code into backwards-compatible JS. This way, we can assure that our code can be used in any web browser.

As requested by our clients, we will be using Leaflet to build our interactive maps, instead of using OL. Leaflet supplies classes to create maps and controls, like a zoom control to zoom in/out of a map. Leaflet is written in JS and is easily integrated with Node. In addition to Leaflet, we will be using two packages: PROJ and proj4leaflet. Because Leaflet does not natively support non-Earth projections, we will be using PROJ [5] to define our projections and proj4leaflet [6] to use those projections in Leaflet.

Our JS code will also be ported to Python in order to create interactive maps in a Jupyter Notebook, a web application for developing and using Python. This is so the USGS can access their maps and map information using another medium. To support Leaflet in Jupyter Notebooks, we will have to use the Python libraries ipyleaflet [7] and ipywidgets [8]. ipyleaflet is a Python wrapper for Leaflet that adds Leaflet back-end functionality to Python. ipywidget adds many GUI elements like buttons and slider. We will be using ipywidget to create the Leaflet front-end

[3] https://nodejs.org/en/docs/
[4] https://babeljs.io/docs/en/
[5] http://proj4js.org/
[6] https://kartena.github.io/Proj4Leaflet/
[7] https://ipyleaflet.readthedocs.io/en/latest/
[8] https://ipywidgets.readthedocs.io/en/latest/

functionality in Python.

Instead of using the old user interface the USGS created for Open Layers (OL), we will be creating a new one to enhance user experience. We will be using React [9], a JavaScript library holding functions that render components the user will interact with. With React, we can update the user interface to something more modern and useful to USGS. We will also be using MaterialUI, a package for React, that will give us added user interface (UI) functionality.

Finally, the auto-complete will be using functions from autocomplete.js. Autocomplete.js [10] will be used to parse and display data from the USGS GeoServer.

3 Architectural Overview

Designing our software application requires an in-depth look at how the system will be built from an architectural standpoint. To facilitate software production and provide a general overview of the project, we will be following the high-level architectural structure specified in Figure 3:

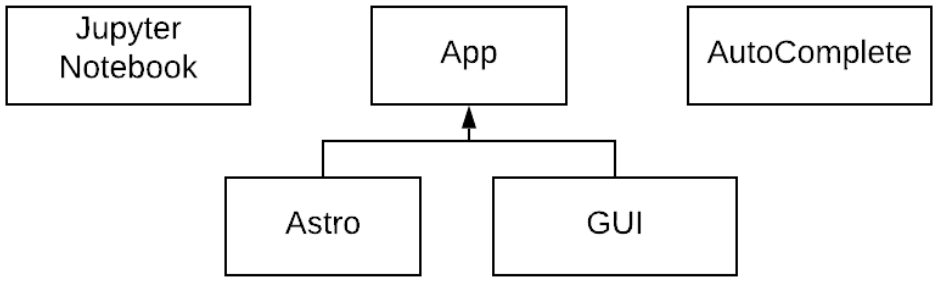

Figure 3: High-level Architecture

This diagram illustrates the architectural and organizational strategy of our application. The root of the structure is the "App", which contains all of the necessary files to build, maintain, and use the planetary map. The Jupyter Notebook and AutoComplete modules will be separate from the main application to provide other utilities. Each module houses all of the necessary components to fulfill the specified functionality or task correlated to its name. A module's child contains specific modular functionality required by the parent module.

3.1 Astro

The Astro module contains the entire back-end functionality of the planetary map. This is where we make calls to the USGS's GeoServer, create the Leaflet map, and add support for changing projections and lat/lon settings. It will contain

[9] https://reactjs.org/

[10] https://tarekraafat.github.io/autoComplete.js//

both a mixture of Leaflet code and standard JS code and follow an object-oriented paradigm. The classes that will be used for the Leaflet solution will be developed following Leaflet's coding standards [11], while the projection and lat/lon back-end code will be written in normal JS so that it can be used in other mapping applications.

3.2 GUI

The GUI module contains all of the components that are required to display the map and console. It consists of the two sub-modules, MapContainer and ConsoleContainer, which hold the root containing elements that display the map and the attached console. The console contains the user interface for the projection and lat/lon switchers. The specific functionality and full contents of this module are described in detail in the Module and Interface Descriptions section of this document (Section 4).

3.3 Jupyter Notebook

The Jupyter Notebook module contains the entire front-end and back-end functionality of the planetary maps that can be used within Jupyter Notebooks. This is where we are mirroring our JavaScript mapping application in Python. It will contain both a mixture of ipyleaflet code, ipywidget's code, and standard Python code.

3.4 Autocomplete

The Autocomplete module will use a package known as autocomplete.js. This package comes with many built-in functions and variables that increase the ease of development. It will allow the user to partially enter in a feature value, see the value in the results list below the search box, select it, and be taken to it. The Autocomplete will encompass all targets, not just the one the user is currently viewing.

4 Module and Interface Descriptions

Each module, described above, together contains the entire logic base of this project. Since this is a complex system with many moving parts, we must ensure our design is fully functioning with intelligent and easy-to-understand choices. To gain an understanding of the specific design decisions we have made, each module is thoroughly described.

[11] https://leafletjs.com/examples/extending/extending-1-classes.html

4.1 Astro

The Astro module contains the classes used to create Leaflet maps, controls, and the code for the switchers. When referencing a Leaflet class, we will reference it by `L.<path_to_class>`. For example, `L.Map` is Leaflet's main map class. This module acts as the back-end for the GUI (see section 4.2).

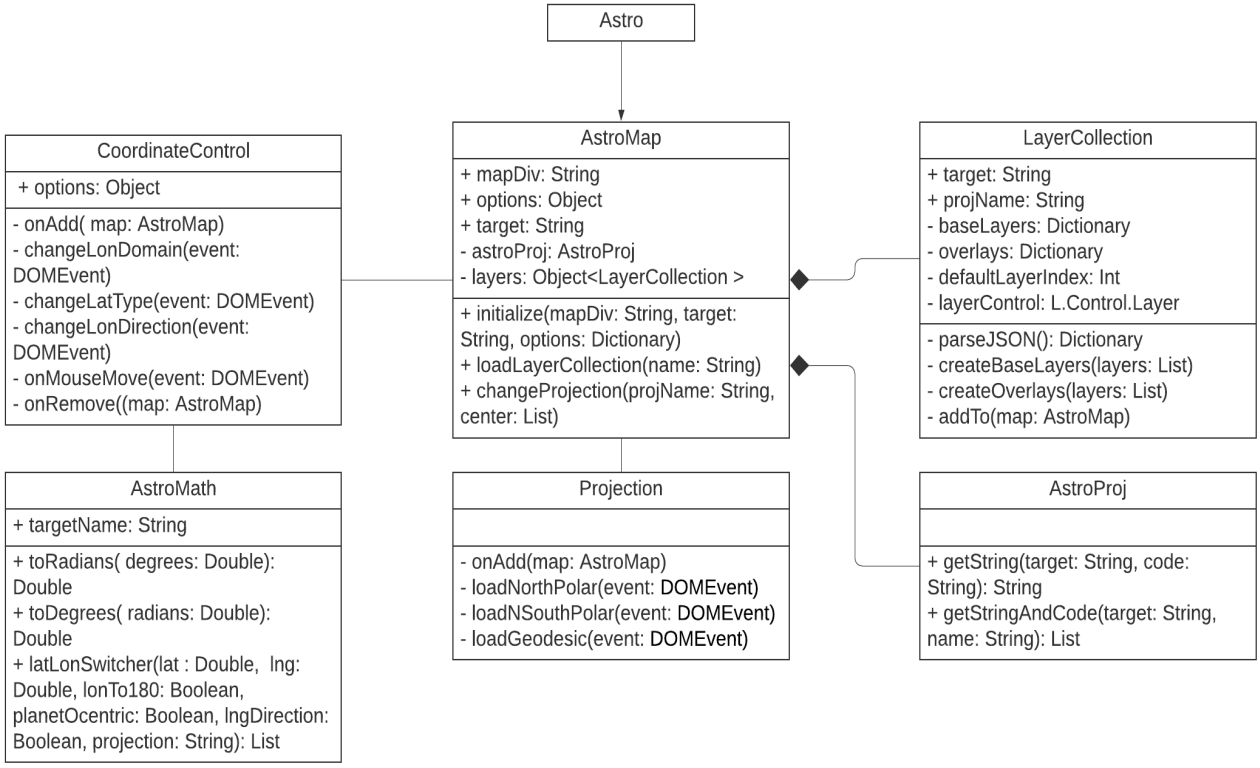

Figure 4: Astro UML diagram

4.1.1 AstroMap

The `AstroMap` class is the main class of the module and utilizes the other classes described below. It will inherit from `L.Map` and be used in the same way as it's parent. That is, `AstroMap` will need layers (`L.Layer`), a coordinate reference system (CRS) or projection, and controls (`L.Control`) to be added to it [12]. In order to support non-Earth projections, we will be utilizing the package proj4leaflet. This package adds the class `L.Proj.CRS` to create non-Earth projections. Because it inherits from the Leaflet base map class, it inherits all of the variables and methods of `L.Map` [13]. This includes methods to render the map in an HTML file, zoom in and out, and pan the map.

[12] https://leafletjs.com/reference-1.6.0.html
[13] https://leafletjs.com/reference-1.6.0.htmlmap-example

In order to add support for the USGS targets and projections, the `AstroMap` class has a target instance variable that represents the planetary body to show the layers for, i.e., Mars, and an instance variable containing the `LayerCollections` (see section 4.1.2), for each projection.

- `initialize()` Constructor method following Leaflet's guidelines. It calls `L.Map.initialize()` with default options set. These options are:
  - `center = [0, 0];` Center of the map
  - `zoom = 1;` Default zoom level
  - `maxZoom = 8;` Max zoom level
  - `crs = L.CRS.EPSG4326;` By default, EPSG:4326, or geodesic, will be set as the map's coordinate reference system (CRS), or projection.

  It will also create `LayerCollections` for the projections supported by the USGS: cylindrical, north-polar stereographic, and south-polar stereographic. By default, the map will load and show the layers for a cylindrical projection. The map will show the layers for one of the projections at a time.
  parameters:
  - `mapDiv (String) - The HTML div to add the AstroMap object to.`
  - `target (String) - The body to show layers for.`
  - `options (Dictionary) - Additional user options. See Leaflet's Map class description for a complete list of options` [14].

- `loadLayerCollection()`
  Adds the `LayerCollection` to itself using `LayerCollection.addTo()` for the requested projection.

  parameters:
  - `name (String) - Name of the projection to load the layers for. Can be either northPolar, southPolar, or cylindrical.`

- `changeProjection()`
  Changes the projection of the map by using the `AstroProj` class (see section 4.1.6) to get a proj-string and proj-code [15], creating a `L.Proj.CRS` object with the string and code, resetting the center of the map, and calling `loadLayerCollection()` to load the layers for the requested projection.

  parameters:
  - `name (String) - Name of the projection to load the layers for.`
  - `center (List) - New center of the map.`

[14] https://leafletjs.com/reference-1.6.0.htmlmap-option
[15] https://proj.org/usage/quickstart.html

4.1.2 LayerCollection

This class stores the base layers (main map layers) and overlays (surface features) for a specific target and projection for quick access and ease of use. The base layers and overlays are both `L.TileLayer.WMS` objects and stored in the instance variables `baseLayers` and `overlays`. The instance variables are dictionaries where the key is the display name of the layer and the value is the `L.TileLayer.WMS` object itself. The variables are created this way following Leaflet's standards and so that the `L.Control.Layer` (layer switcher) recognizes the two different types of layers. As stated previously, layers for a map are stored on the USGS's GeoServer. A JSON was given to us by the USGS that contains the parameters to query the Geoserver for all layers. These parameters include the URL and display name of the layer. To make layer loading fast, this class queries GeoServer and creates the `L.TileLayer.WMS` objects before being added to an AstroMap. Since this class is Leaflet-specific, it follows Leaflet's standards by inheriting `L.Class`.

- `initialize()`
  Constructor that sets the target and projection to load layers for, parses the JSON, queries the GeoServer, and creates the base layers and overlays for future use by the `AstroMap` class.
  parameters:
  - `target (String) - The body to load layers for.`
  - `projName (String) - Name of the projection.`
- `parseJSON()`
  Parses the JSON containing layer information to gather all the information for each layer for the target and projection. Then, stores the layer information for each one in a dictionary. Here is an example of what one of those looks like:
  - `displayname: Viking North (MDIM 2.1)` (Layer name)
  - `format: image/png` (Format of the output)
  - `layer: MDIM21_north` (Layer name in the GeoServer)
  - `map: /maps/mars/mars_npole.map` (Path to the map in the GeoServer)
  - `primary: true` (True if it is a default layer)
  - `projection: north-polar stereographic` (Projection of the layer)
  - `transparent: false` (True if it is an overlay)
  - `type: WMS` (Type of standard to follow for the query)
  - `units: m` (Units the layer is stored in)
  - `url: https://planetarymaps.usgs.gov/cgi-bin/mapserv` (Base URL to retreive the map)

  Note: These dictionaries are not yet `L.TileLayer.WMS` objects.

We will use the displayname, layer, map, and url values to query the GeoSever. Then, we store each dictionary in a list that is contained in the dictionary layers:

```
layers = {
  base: [],
  overlays: []
};
```

The base layer dictionaries being put in the 'base' list and the overlays in the other.

returns:

- `(Dictionary) The layers object containing lists storing the base layers and overlays.`

- `createBaseLayers()`
  Instantiates `L.TileLayer.WMS` objects for each dictionary stored in the base list created from `parseJSON()`. Adds each WMS layer to the `baseLayers` instance variable.

  parameters:

  - `layers (List) - List of base layer dictionaries.`

- `createOverlays()`
  Instantiates `L.TileLayer.WMS` objects for each dictionary stored in the overlays list created from `parseJSON()`. Adds each WMS layer to the overlays instance variable.

  parameters:

  - `layers (List) - List of overlay dictionaries.`

- `onAdd()`
  Tells the map what to do when it wants to add a `LayerCollection` to itself. If the `AstroMap` has layers loaded, it will delete them before adding new ones. Then, adds the base layers and overlays to the AstroMap and selects the map's default layer.

  parameters:

  - `map (AstroMap) - The AstroMap to add the layers to.`

4.1.3 ProjectionControl

Class that inherits from `L.Control`, following Leaflet's standards [16], and handles the back-end when a user clicks on the projection buttons (see section 4.2.4). Since this class inherits `L.Control`, it is added to the `AstroMap` in the same way as other controls, like the zoom control.

- `onAdd()`
  Grabs the projection switcher buttons from the GUI and adds on-click events to the buttons. When a user click the north-polar button, the `loadNorthPolar()` method is called, and so on for the other projections.

  parameters:

  - `map (AstroMap) - The AstroMap to add the control to.`

- `loadNorthPolar()`
  Is called when a user clicks the north-polar button. Calls `AstroMap's changeProjection()` method to change the projection to north-polar stereographic and center of the map.

- `loadSouthPolar()`
  Is called when a user clicks the south-polar button. Calls `AstroMap's changeProjection()` method to change the projection to south-polar stereographic and center of the map.

- `loadCylindrical()`
  Is called when a user clicks the cylindrical button. Calls `AstroMap's changeProjection()` method to change the projection to cylindrical and center of the map.

4.1.4 CoordinateControl

Class that inherits from the class `L.Control` and handles the back-end when a user clicks on the lat/lon buttons (see section 4.2.5). Since this class inherits `L.Control`, it is added to the `AstroMap` in the same way as other controls, like the zoom control.

- initialize() Constructor method following Leaflet's guidelines. It calls `L.Control.initialize(` with default options set. These options are:

  - `separator = '' : \; The character separating latitude and longitude coordinates.`
  - `numDigits = 5; The amount of decimals the latitude and longitude are fixed to.`
  - `prefix = ''"; The String to put in front of the coordinates`
  - `targetPlanet = ''"; The specific target name`

- `onAdd()`
  Grabs the lat/lon buttons from the GUI and adds on-change events to them.

[16] https://leafletjs.com/examples/extending/extending-3-controls.html

It also adds an on mouse-over event to the AstroMap to grab the current mouse position of the user's mouse pointer.
parameters:

  - `map (AstroMap) - The AstroMap to add the control to.`

- `changeLonDomain()`
  Is called when a user changes the longitude domain selector. Changes the longitude domain class variable to false if 0° to 360° is selected and true if −180° to 180° is selected.

- `changeLatType()`
  Is called when a user changes the latitude type selector. Changes the latitude type class variable to false if planetographic is selected and true if planetocentric is selected.

- `changeLonDirection()`
  Is called when a user changes the longitude direction selector. Changes the longitude direction class variable to false if positive west is selected and true if positive east is selected.

- `onMouseMove()`
  Is called when a user moves their mouse over the `AstroMap`. The function uses the class latitude and longitude class variables combined with the `AstroMath` class (see section 4.1.5) to calculate the correct coordinate mouse position of the users mouse pointer.

- `onRemove()`
  Removes the coordinate control from the `AstroMap`.
  parameters:

  - `map (AstroMap) - The AstroMap to remove the control from.`

4.1.5 AstroMath

A helper class that can be used by any mapping application, not just Leaflet, to calculate different longitude and latitude domains and ranges for a specific target. It uses a JSON file in the background to store the targets and their associated radii. The radii and their names are correlated as such:

- `MajorRadius == 3396190.0`
- `MinorRadius == 3376200.0`

Using the target's radii will allow conversion from planetocentric and planetographic latitudes.

- `initialize ()`
  Constructor that grabs the target planet's radii from the JSON based off of target name.
  parameters:

- targetName (String) - The body to get the radii of.

- `toRadins()`
  Takes in a number in degrees and converts it to radians.
  parameters:
  - `degrees (Double) - Decimal number in degrees to be converted to radians.`

  returns:
  - `(Double) The converted value in radians.`

- `toDegrees()`
  Takes in a number in radians and converts it to degrees.
  parameters:
  - `radians (Double) - Decimal number in radians to be converted to degrees.`

  returns:
  - `(Double) The converted value in degrees.`

- `latLonSwitcher()`
  Changes lat/lon coordinates based on certain domains, directions, and ranges. This function will handle longitude ranges from 0° to 360° and −180° to 180°. It will also handle positive east and west longitude directions. For the latitude, it will calculate both planetocentric and planetographic latitudes for every specific target.
  parameters:
  - `lat (Double) - User's current latitude on map.`
  - `lng (Double) - User's current longitude on map.`
  - `lonTo180 (Boolean) - True if longitude range is −180° to 180°, false otherwise.`
  - `planetOcentric (Boolean) - True if latitude is planetocentric, false otherwise.`
  - `positiveEast (Boolean) - True if longitude direction is positive in the east, false, otherwise.`
  - `projection (String) - Current projection of the map.`

  returns:
  - `(List) The converted lat/lon coordinates.`

4.1.6 AstroProj

A helper class that can be used by any mapping application, not just Leaflet, to query for proj-codes and proj-strings for a specific target and projection. Proj-codes are unique identifiers given to a projection. For example, EPSG:4326 is the proj-code for Earth's cylindrical projection, EPSG:32661 is Earth's north-polar projection, and EPSG:32761 is Earth's south-polar proejection. Proj-strings are used to define projections and their centers and units. The `AstroProj` class uses a JSON in the background to store the targets and their associated projections. The projection names and codes are correlated as such:

- `NorthPolar == EPSG:32661`
- `Cylindrical == EPSG:4326`
- `SouthPolar == EPSG:32761`

Because the USGS's GeoServer only accepts Earth proj-codes in queries, we must use Earth proj-codes instead of a target's real proj-code.

- `getString()`
  Returns a proj-string for the target and proj-code given.
  parameters:
  - `target (String) - Name of the target.`
  - `code (String) - The code of the projection.`

  returns:
  - `(String) The requested proj-string.`
- `getStringAndCode()`
  Returns the proj-string for the target and name of the projection given.
  parameters:
  - `target (String) - Name of the target.`
  - `name (String) - The name of the projection. Can be one of the three listed above.`

  returns:
  - `(List) The requested proj-code and proj-string.`

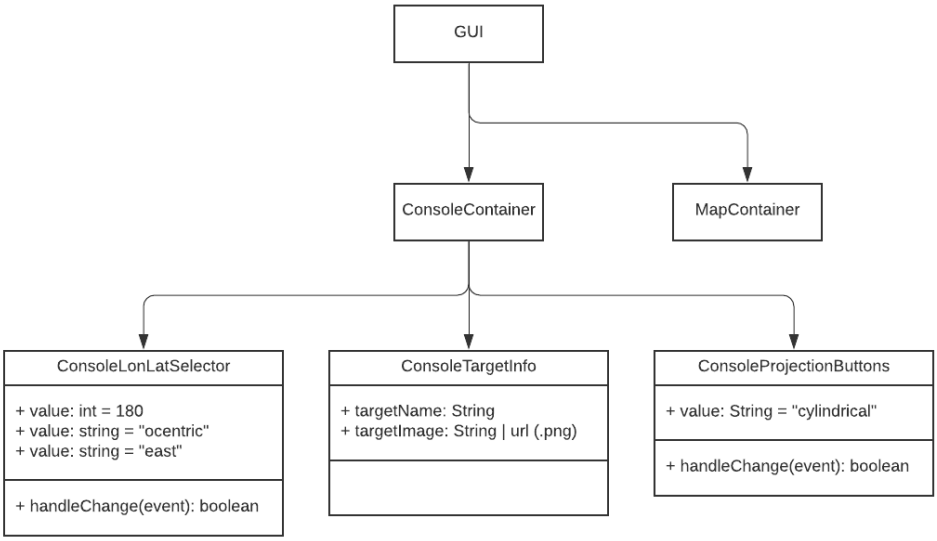

Figure 5: GUI UML diagram

4.2 GUI

The GUI is one of the most important components of a software application; it being the primary means of user interaction and must be designed with ease of use and attractiveness in mind. To achieve this standard, we designed the GUI using Facebook's `React` framework with Google's Material UI components and followed this diagram as a blueprint.

Note: The components in this module do not follow an object-oriented design, rather they are collections of styled HTML elements compiled with the React framework's component system.

4.2.1 MapContainer

The MapContainer component is the base containing element for the embedded Leaflet map. It imports functionality from `Astro` and other modules to display the map inside of the root element, a div tag with `'id="map"'`, inside of the HTML page. The MapContainer also controls the size, styling, and location of the map on the page and is a responsive design for varying screen size.

4.2.2 ConsoleContainer

The ConsoleContainer is the parent container that holds the ConsoleTargetInfo, ConsoleProjectionButtons, and ConsoleLonLatSelectors. The ConsoleContainer organizes its children components' style and locations inside of a root element inside of the HTML page. It creates a 3 column grid that houses each of the child components with responsive design for varying screen size.

4.2.3 ConsoleTargetInfo

The ConsoleTargetinfo is a child component of ConsoleContainer that dis-

plays information about the planet being viewed. It contains a 3 row grid that displays the target planet's name, a small image of the planet, and the coordinates underneath the cursor on the map.

4.2.4 ConsoleProjectionButtons

The ConsoleProjectionButtons is another child component of ConsoleContainer. It displays the projection buttons and controls which projection is showing on the map. In each row of this 3 row grid, there is a button indicating the type of projection currently being viewed on the map. The top button is North Polar, the middle button is Cylindrical, and the bottom button is South Polar.

4.2.5 ConsoleLonLatSelectors

The ConsoleLonLatSelectors is the final child component of ConsoleContainer. It displays three selectors inside of a 3 row grid to choose between latitude and longitude options. These options are Planetographic or Planetocentric, -180° to 180 ° or 0° to 360° latitude ranges, and Positive East or Positive West latitudes. By default the Planetocentric, -180° to 180° range, and Positive East options are selected.

4.3 Jupyter Notebook

The Jupyter Notebook module contains two classes used to create Leaflet maps, controls, and a graphical user interface. In order to achieve this functionality, we designed the Jupyter Notebook module using `ipyleaflet`, a Python Leaflet wrapper, and `ipywidgets`.

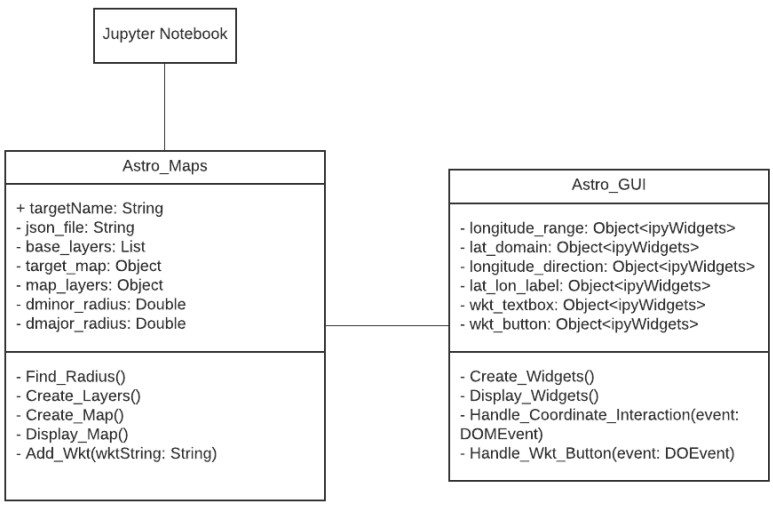

Figure 6: Jupyter Notebook UML diagram

4.3.1 Astro_Map

The `Astro_Map` class is the main class of this module and utilizes the `Astro_GUI` class for the front-end and back-end functionality of the different on-screen buttons and selectors. This class takes a specific target name and uses `ipyleaflet` to create and display a map of the target.

- `initialize ()`
  Constructor method that takes in a specific target name and creates an `ipyleaflet` map instance.
  parameters:
  - `Target (String): The body to show the` layers for.
- `Find_Radius()`
  Finds the minor and major radius of the specific target we are trying to map. Holds these radius values in class variables.
- `Create_Layers()`
  Parses through the local USGS JSON file to find all the layers for the specific target. Holds the layers in a layers object class variable.
- `Create_Map()`
  Creates, loads, and adds controls to an `ipyleaflet` map. Holds the map object in an object class variable.
- `Display_Map()`
  Displays the map object on screen.
- `Add_Wkt()`
  Takes in a well known text (WKT) string and draws the shape onto the planetary map.
  parameters:
  - `wktString (String) - WKT string.`

4.3.2 Astro_GUI

The `Astro_GUI` class creates and monitors the latitude and longitude selectors for the planetary maps in Jupyter Notebook. It does this by utilizing the python module `ipywidgets`.

- GUI Elements
  - ConsoleLonLatSelectors
    The ConsoleLonLatSelectors displays three selectors inside of a 3 row grid to choose between latitude and longitude options. These options

are Planetographic or Planetocentric, -180° to 180 ° or 0° to 360° latitude ranges, and Positive East or Positive West latitudes. By default the Planetocentric, -180° to 180° range, and Positive East options are selected.

  - WKT Text Box
    A text box users can use to enter in well known text strings (WKT strings).
  - WKT Draw Button
    A button users can press to draw the well known text string on the `Astro_Map`.

- `initialize ()`
  Constructor method for GUI class that calls `Create_Widgets()` to initialize the ipywidget objects.

- `Create_Widgets ()`
  Creates and loads the `ipywidget` objects needed for the planetary GUI.

- `Display_Widgets()`
  Displays all the widget GUI objects to screen.

- `Handle_Coordinate_Interaction()`
  On mouse-over of our `Astro_Map`, this function will handle the coordinates of the user's mouse positions by checking the lat/lon selectors. It will put the current mouse position coordinate into the lat_lon_label widget object.

- `handle_Wkt_Button()`
  On the press of the draw button, it will take the string entered into the wkt_textbox widget object and draw it onto the `Astro_map`.

4.4 AutoComplete

The AutoComplete will be responsible for providing the full names of features on a target after they are partially entered inside of a search box on the USGS website. The planetary feature names will be pulled from the USGS GeoServer, and when selected in the AutoComplete, will bring up the map of the target with the requested feature. Users will be able to search for all features from the main USGS website page, and target-specific ones from a target's page. This will be implemented with `autocomplete.js`, a JavaScript library filled with AutoComplete functions.

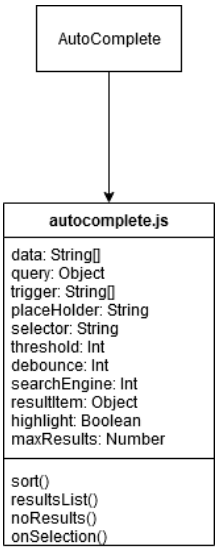

Figure 7: autocomplete.js

- `sort()`
  Used to sort the results list descending, function can also be used to sort ascending.

- `resultsList()`
  List of all the resultItems that start with the current autocomplete query in the data. Displays below the searchbox, can use the threshold variable to establish how many characters must be entered for the list to return an item. maxResults is used to limit the resultsList to a specific number of items. Highlight is used to highlight the characters the are in the search string that are in each returned resultItem. SearchEngine is used to limit the resultsList to starts with (strict) or contains (loose) values. The position attribute can also be edited to choose where the resultsList displays relative to the searchbox. The destination attribute is used to query the the searchbox so the resultsList can display using the position. Finally, the element attribute is how you can chose how the resultsList displays, as a UL, Div, Span, etc..

- `noResults()`
  Designates what actions to take when the current query entered in the searchbox does not return any results. These actions could entail displaying "No Results", running another query, changing the current webpage, etc...

- `onSelection()`
  Designates what actions to take when one of the resultItems on the resultsList is clicked on. The way it is used for the CartoCosmos project is building a URL using the feedback (value clicked on) and send the user to that URL.

5 Implementation Plan

In order to successfully develop the four modules explained above we split our development plan into four different sections: JS Planetary Maps, Jupyter Notebook Planetary Maps, AutoComplete, and testing. Looking at the Gantt chart in the appendix Figure 9 below will show you our implementation plan. From a top level view, our implementation phase can be separated into two sections: development of maps and developments of extras.

The development phase consists of abstracting our demo completed during the fall semester. We separated this abstraction process into five different sections: our AstroMap class (and the other Astro classes besides AstroCoordinate), AstroCoordinate class, AstroGUI class, Jupyter Notebook module, and AutoComplete module. We are currently implementing all these sections at the same time because some of them intersect. In order to develop our JavaScript Leaflet implementation, we need all of the Astro classes and AstroGUI module to connect and work well with each other.

During the development process, we are not only implementing the necessary functions each class needs, but we will also be documenting every class along the way. Documentation is a vital part in our project because eventually we will be delivering our project to USGS. Documenting alongside the implementation of each class will allow us to best represent each class.

The end goal of the development phase is to create a high-quality alpha demo. Our alpha demo will look a lot like our technical demo, but with a new GUI, the ability to map other targets, a Jupyter Notebook solution, and an auto-complete demo. Looking at Figure 8, you can see who is working on each module.

AstroMap	AstroCoordinate	AstroGUI	Jupyter Notebook	AutoComplete
Kaitlyn Lee	Jacob Kaufman	Scott Ames	Jacob Kaufman & Kaitlyn Lee	Chris Moore

Figure 8: Distribution of Labor

After the completion of our development phase, we will begin our testing phase. Since we are creating a product that will be used by researchers, we are going to have two different phases of testing: integrated testing and client testing. The integrated tests will ensure our classes are working and interacting as they are intended to. The client side testing, on the other hand, will ensure our GUI is liked by researchers and easy to use.

6 Conclusion

The planetary science community, which is very large and spans the globe, consists of both developers and scientists that work together to create maps. Without these maps, planetary missions would be impossible. Our clients, Trent Hare

and Scott Akins, who work for USGS, are scientists developers who help in the map-making process. In order to do research with these maps, USGS scientists need a mapping tool that allows them to view non-Earth bodies, change the projection of the map, and chagne the lat/lon settings. Currently, USGS has an OpenLayers (OL) implementation, but as discussed, it is not modular and difficult to update.

In order to solve these problems, we will be creating a new mapping tool with Leaflet and a portable package. The new Leaflet tool will contain extra functionality such as the lat/lon switcher and projection switcher. The package will contain the back-end code for the projection and latitude and longitude switches. Our package will be portable and able to be used in other mapping tools such as OpenLayers. The USGS will be able to use this package in future implementations.

The module architectures detailed in the above sections are the guidelines we have decided to follow in order to make sure our product for the USGS is successful. The architecture is split into four different modules: Astro, GUI, Jupyter Notebook, and AutoComplete. Astro (see section 4.1) outlines how we plan to structure all the back-end code for the JS Leaflet solution. These pertain to creating the planetary maps and the controls used to add different planetary functionality, such as projections, layers, and mouse position coordinates. The GUI (see section 4.2) describes the different components and containers for the front-end of the JS Leaflet solution. Jupyter Notebook (see section 4.3) outlines the Python solution for planetary mapping (it has the same functionality as our JS solution). Finally, the AutoComplete (see section 4.4) outlines the auto-complete functionality to be used on the USGS's website. Like Jupyter Notebook, the AutoComplete module will be a standalone module.

Our project plan details how we are going to break up the project into smaller tasks and when we will have these tasks completed. So far, we have completed the technical feasibility, requirements acquisition, and technical demos. These demos showcased a Leaflet map showing Mars' layers and projections, a Mars Leaflet map being created in a Jupyter notebook, GUI layouts for future implementation, and documentation examples following Leaflet's standards. We are now starting to abstract the Mars demo in order to view other specific targets and add a better GUI. We are also adding functionality to Jupyter Notebooks so users can view and add geometric shapes to our planetary maps. Lastly, we are creating the AutoComplete functionality to be added to the USGS website.

With this project, USGS will be able to easily view and work with all targets in Leaflet, which will speed up and simplify their work in mapping. The Leaflet functionality that we add will be used by USGS and possibly NASA to map out future missions to different moons and planets and will become integral to USGS. Even if USGS stops using Leaflet, our code will be modular so that the USGS can reuse it for other mapping tools. We are very optimistic about completing this project on time while going beyond the expectations for the client. We are excited to be building these tools to accelerate the process of space exploration.

7 Appendix

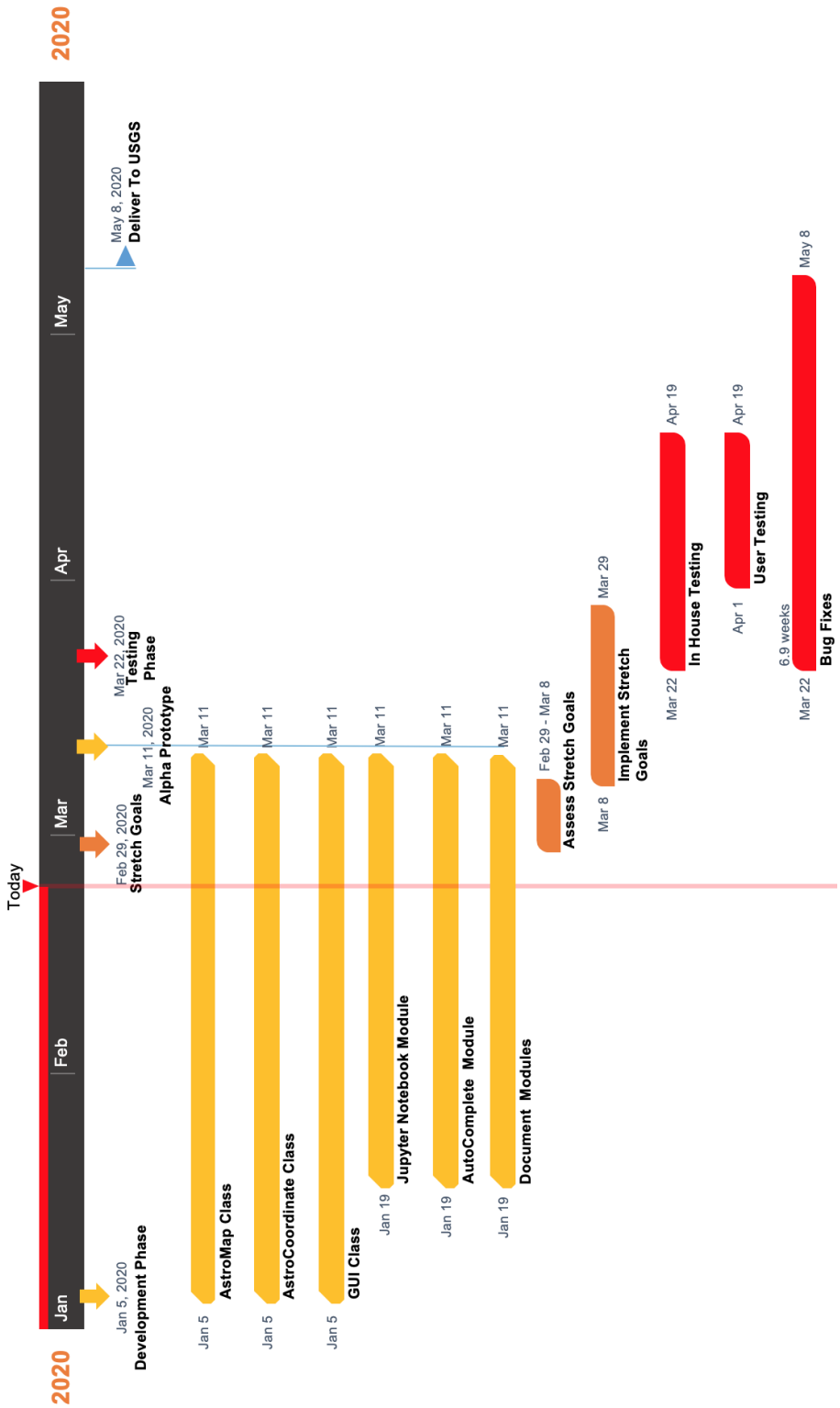

Figure 9: Gantt Chart