Introduction
This document details the continuation of my work on the MASS debugger during fall quarter 2015. As I will no longer be working on this project, this document contains all information regarding both the connection to MASS Java and MASS C++, steps involved in running the debugger with a MASS application, as well as design decisions/algorithms, and the next phase of the debugger's development.

Integration with MASS C++
During the summer quarter the MASSCppConnection class was left incomplete since I was not sure of the best way to make a connection between the debugger, written in Java, and the MASS C++ library. This section will describe the details of the MASSCppConnection class as it is written now.

MASSCppConnection.java
MASSCppConnection.java inherits from the Connection class, along with MASSJavaConnection.java as shown in figure 1. The connection class provides the interface for opening and closing connections between MASS Java and MASS C++ applications.
Though the connection details between the debugger and MASS Java are fairly straightforward and are documented in my MASS Java document, the connection details between the debugger and MASS C++ may need some explaining.

Since the debugger cannot communicate with the MASS C++ library using the transfer of Java Objects as it does in a MASS Java connection, we needed to use an intermediate language in order to communicate these objects. The intermediate language I have chosen is JSON. JSON was chosen since it has abundant 3rd party libraries that handle the parsing of the JSON object on both the C++ and Java front. For the Java side we use the library GSON. It is a google implemented JSON library that is used to convert the strings of JSON data into Java objects. On the C++ side we use the library Rapidjson for converting C++ objects into JSON strings. The only difference now between the connection between the debugger and MASS Java or MASS C++ is that when the debugger makes a data request to the MASS C++ library, the data is returned as a string of JSON data that must now be parsed and converted into a Java object.

**GSON**

I will cover in detail how GSON is used in the debugger in converting the JSON string into Java objects. Depending on the type of request made to the MASS library, you will be returned with different types of JSON strings that need to be parsed accordingly. I will illustrate the process using GSON assuming the data returned is an UpdatePackage. An UpdatePackage contains new state information regarding Places and Agents, location of Agents, number of Agents etc. This JSON string is sent directly to the UpdatePackage via its constructor. Figure 2 shows the construction of the GSON utility classes in the UpdatePackage constructor.

```java
JsonObject jsonObject = new JsonParser().parse(jsonObject.trim()).getAsJsonObject();
JsonObject packet = jsonObject.getAsJsonObject("packet");
JsonArray jsonArr = packet.get("PlaceArray").getAsJsonArray();
Gson googleJson = new Gson();

ArrayList<LinkedTreeMap> list = googleJson.fromJson(jsonArr, ArrayList.class);
PlaceData[] placeDataArr = new PlaceData[list.size()];
```
The constructor start by instantiating some of the GSON utilities used to parse the jsonString. The first line converts the Java String “jsonString” into a jsonObject that can now be parsed. The second line searches the jsonObject for the object called “packet” which contains all Place data as an array. The third line then fetches this place array and saves it as a JSONArray. From here we are ready to iterate through this array to fetch the place data. The following code segment shows this step.

```java
for(int i = 0; i < list.size(); i++)
{
    LinkedTreeMap<String, Object> map = list.get(i);
    PlaceData newPlace = new PlaceData();

    newPlace.setThisPlaceData(((Double)map.get("Number"));
    newPlace.setIndex(((Double)map.get("index")).intValue());
    newPlace.setHasAgents(((Double)map.get("hasAgents"))).intValue() == 1);

    placeDataArr[i] = newPlace;

    ArrayList<LinkedTreeMap> arr = (ArrayList) map.get("AgentArray");
    AgentData[] agents = new AgentData[arr.size()];
```

As you can see we are iterating though the place array, parsing for “Number”, “index”, and “has Agents” data members and constructing PlaceData objects from them. Each Place object also contains an array of Agent objects that reside on that Place. This is accessed from the JSON object via the second to last line that retrieves the agents as an ArrayList. The following shows the parsing of that ArrayList.

```java
for(int j = 0; j < arr.size(); i++)
{
    LinkedTreeMap<String, Object> agentMap = arr.get(j);
    AgentData agent = new AgentData();

    agent.setIsAlive(((Double)agentMap.get("isAlive") == 1.);
    agent.setId(((Double)agentMap.get("id")).intValue());
    agent.setDebugEnabled(((Double)agentMap.get("Number"));
    agent.setIndex(((Double)agentMap.get("index")).intValue());
    agent.setChildren(((Double)agentMap.get("children")).intValue())

    agents[j] = agent;
}
```

We continue by parsing for Agent data members “isAlive”, “id”, “Number”, “index”, and “children”. This data is then saved as an AgentData Object. To complete the constructor, the
UpdatePackage is then instantiated and contains all new Place and Agent data to be passed to the GUI for display.

Rapidjson

The following will demonstrate how Rapidjson is used in the MASS C++ library to construct a JSON String from the C++ objects. The example I will use in the debugInit() method of MASS.cpp. This method collects all Place and Agent data from the simulation, constructs the JSON object, and sends the JSON as a string via socket connection to the debugger. The following snippet shows the construction of the Places in JSON representation.

```cpp
// set place data to JSON
Value &pName = MASS::document["packet"]["placesName"];
pName.SetString(StringRef(placesName.c_str()));
Value &pX = MASS::document["packet"]["placesX"]; pX.SetInt(placesX);
Value &pY = MASS::document["packet"]["placesY"]; pY.SetInt(placesY);
Value &pNumberOfPlaces = MASS::document["packet"]["numberOfPlaces"]; pNumberOfPlaces.SetInt(numberOfPlaces);
```

In the first line, you will notice the Rapidjson object “document”. Document contains the JSON representation that will be constructed. Using bracket notation we can access the JSON fields “packet”, and “placesName”. This returns the field “placesName” from the “packet” object. Nce we have a reference to “placesName” we can set the name of the place in the following line. This is continued with “placesX”, “placesY”, and “numberOfPlaces”. The following shows the same procedure for adding the Agent data to the JSON string using Rapidjson.

```cpp
// set agent data to JSON
Value &aName = MASS::document["packet"]["agentsName"]; aName.SetString(StringRef(agentsName.c_str()));
Value &aNumberOfAgents = MASS::document["packet"]["numberOfAgents"]; aNumberOfAgents.SetInt(numberOfAgents);
```

Once we have the Place and Agent data added to the JSON string via Rapidjson, we must convert the Rapidjson into a JSON string and send it via socket to the debugger. The following outlines this procedure.

```cpp
// write json string to buffer
StringBuffer jbuffer;
Writer< StringBuffer > writer(jbuffer);
MASS::document.Accept(writer);

// send buffer to GUI
const char * json = jbuffer.GetString();
MASS::n = write(MASS::newsockfd, json, strlen(json));
```
Changes to Place and Agent

In order for the debugger to have access to Place and Agent data in a MASS C++ simulation, both user implemented Place and Agent classes must override the method getDebugData(). Place.java and Agent.java both contain these virtual methods as shown below.

```java
virtual double getDebugData();
```

Currently, getDebugData is limited to returning doubles. This is not the case in the MASS Java version. Java has object equivalents for all primitives that allow most numeric data types to be returned as a Number object. C++ does not have this available, so until we can determine a similar method for returning multiple types of numeric data we will use doubles.

A user must override this in their Place/Agent implementation. An example of this can be seen in the MASS C++ simulation “Wave2d” below.

```java
double Wave2D_MASS::getDebugData()
{
    double number = 0;
    if(time == 0)
    {
        number = wave[0];
    }
    else if(time == 1)
    {
        number = wave[1];
    }
    else
    {
        number = wave[2];
    }
    return number;
}
```

In this example, a double, the height of a wave in this simulation, is returned to the MASS C++ library, added to the Number field of the JSON string being constructed, and sent to the debugger GUI to be displayed to the user.

In java there is an analogous method called setDebugData() that allows the user to inject new data into the simulation. For example in the above Wave2d simulation, the user is able to inject a new wave height. This feature is restricted to the MASS Java version currently, but will be added to the MASS C++ version shortly.

Changes to Simulation

In order for the debugger to communicate with your MASS C++ application a couple changes need to be made to the simulation’s main method. These changes are shown below in the Wave2D example application.
The first line of code that needs to be added is a call to the MASS::debugInit() method. This method is designed to initialize the debugging capabilities of the MASS library. First a connection is attempted to be made with the GUI. This is a blocking procedure, meaning the users program will halt until a connection is made to the GUI. This allows the user to execute their program and the GUI in any order, where as previously the user was required to start their program before execution of the GUI. After a connection is made to the GUI, data is gathered into an InitialData object that is then sent to the GUI. This initial data includes the following:

- The class names of the user defined Agent and Place.
- Booleans representing whether or not the user has overridden the get and set debug data methods.
- The data type of the overloaded get and set debug data methods in each, Agent and Place, objects.
- The number of Places and Agents.
- The dimensions of the Places grid.

The initial data that is sent can be expanded by adding data members to the InitialData class. Once the GUI received this InitialData, it displays the user relevant data such as Agent/Place names and sizes. The other data is saved and accessed when needed, such as the user defined data types, which are needed when sending user injected data back to the Agents and Places.

The second line that needs to be added to the users simulation is a call to the MASS::debugUpdate() method. This method is used to transfer data between the users program and the GUI. The type of data sent is dependent upon the user’s action in the GUI. For example if the user presses the “next” button, a packet of data pertaining to current Agents and Places state is sent to the GUI to be displayed in the visualization grid. A complete description of each type of transfer than can be completed by calling the debugUpdate() method is as follows:

1. As stated above, if the user has pressed the “next” button or the “play” button, data is gathered pertaining to the current state of Agents and Places into an UpdatePackage object. This object contains the following member fields which may be extended by adding members to this class:
   
   - An array of PlaceData objects, each of which contain:
     
     a) This places data, as defined by the user overridden getDebugData().
b) The index in which this place exists in the array of Places held by the MASS library.
c) A Boolean representing whether this place contains agents.
d) An array of AgentData objects, each of which contain:
   i. This agent’s data, as defined by the user overridden getDebugData() method.
   ii. This agent’s index corresponding to the index of the place in which it resides.
   iii. This agent’s id number as assigned by the MASS library.

2. If the user injects Place or Agent data using the GUI, an AgentData or PlaceData object is sent back to the MASS library. The contents of each are described above. This data is then set using the Agents/Places setDebugData() method.

3. When the user chooses to end the connection between their program and the GUI or when the user closes the application a Disconnection object is sent from the GUI to MASS in order to do clean up (closing sockets and streams etc.). This is not yet implemented at this time. When it is a complete description of this object and its contents will be shown here.

**Using the debugger with MASS C++**

The following outlines the steps involved in running the debugger with the MASS C++ library. Once the connection settings are in place the procedure is identical to that of the MASS Java library. Regardless, I will document the entire process.

**Connection Settings**

On startup of the debugger GUI, the user is first presented with a user preferences splash screen in order to retrieve user specified preferences/connection settings. This splash screen is shown below.

![Set connection](image)

The user should enter the host and port number and choose what version of the MASS library they are using. For the purposes of this document we will assume the user is using the MASS C++ library.

Once the user enters the host and port and clicks the save button, the main GUI form opens. Host and port information is saved from here on out and the user will not have to enter it again unless they wish to change it.
At this point the GUI informs the user that there is not yet a connection between the GUI and the user’s program. This is shown in figure 9. The connection state will be displayed here. At this point all buttons and field are disabled except for the “connect” button.

Once the connect button has been clicked, a connection will be made to the users program. If successful the displayed connection state will change from figure 9 to figure 10 or an error message will be displayed in this space.

The names and sizes of Agents and Places will be displayed to the user as well. From here the user has a couple options. The user may click the play button or the next button shown in figure 11.
The play button runs through the users entire simulation loop with 1.5 seconds between each iteration. The user may pause the simulation at any time during this loop by clicking the “pause” button. The “next” button on the other hand, will run just one iteration of the simulation and immediately pause it so the user may iterate at their own pace. Figure 12 shows the GUI’s state after the user has pressed either the “play” or the “pause” buttons.

**Work to be completed**

As with the Java version, resizing the place matrix is an issue. Resizing the visualization grid was a known issue in Hongbin Li’s GUI implementation as well. The issue arises when the user has roughly more than 10,000 Places in their simulation, which is very common. After this point the GUI has only a few choices, each with severe drawbacks. One solution is to maximize GridPanel as much as possible and minimize each CellPanel to fit within the GridPanel. The problem with this solution is that one each CellPanel become less than one by one pixels, the grid becomes useless to the user as action listener accuracy fails
at this point. Ideally, the user would be able to zoom in and drag the grid around, similar to the features of Google Maps. The issue with implementing these features are the limitations of the Java Swing library. Although possible, my research indicates this task would be difficult and time consuming. In the future I would like to switch from Swing to JavaFX. JavaFX is java’s newest graphics library that has built in utilities for implementing zoom and drag features. JavaFX would also allow for 3D grids, allowing debugging capabilities for 3D MASS Places, which is currently not supported.

Some features that are to be completed in the following few quarters include agent tracking, simulation speed adjustments, logging. Agent tracking would allow a user to select an agent using the debugger GUI by clicking on it. Once an agent is selected the GUI could highlight the path that the agent takes throughout the runtime of the simulation and record this movement for analysis. This feature would be relatively easy to implement since the debugger already collects agent ID information. With the agent ID we can locate where exactly the agent is moving an easily highlight this movement to the user. Allowing the user to change the pace of the simulation in the GUI would also be another simple feature addition, currently the speed is set at one iteration every 1.5 seconds. The speed is kept as a global constant in the Frame class of the GUI and could adjusted by providing a button or slider to the user. Clearly there are limitations to the speed a user can select since the data is at mercy to network connection speeds. The maximum speed would need to be tested and could perhaps be different for each simulation. Logging is another addition to the debugger GUI that would greatly benefit the user. Not only logging Place and Agent data, but connection status, program flow, etc.