Gazebo Flight Simulator Plugin

Jamison Hubbard - jhubbard@mines.edu
Steven Vennard - ssvennard@mines.edu
Chandler Mitchell - clmitchell@mines.edu
Naail Ali - nali1@mines.edu

June 10, 2020

Contents

1 Introduction

For Field Session 2020, our team was assigned to Sunlight Aerospace to create a custom flight simulator plugin for the Gazebo simulator. Sunlight Aerospace was founded in 2007, and specializes in wireless communications, defense, and unmanned transportation. The problem that Sunlight Aerospace faced was that they previously used `LiftDragPlugin` to calculate forces and moments acting on the aircraft, but that plugin is not accurate. The `LiftDragPlugin` simulates each control surface individually instead of considering the aircraft as a whole. To fix this issue, our team used open-source software and C++ to develop a custom plugin for Gazebo, in order to accurately simulate the behavior of fixed-wing aircraft. The plugin has the capability to take in environmental and model variables, then output aerodynamic forces and moments that will be applied to the model. The provided Gazebo simulation environment will run the plugin and utilize PX4 flight control firmware in order to test Sunlight Aerospace's UAVs.

2 Requirements

2.1 Non-functional Requirements

PX4 code must not be changed PX4 is an open-source flight control firmware that can be found on GitHub. Sunlight Aerospace specified its use in our project, also specified that it could not be changed, so we had to fork the repository in order to include it within the project, but remained vigilant of the fact that it must remain unedited.

Plugin must not modify the environment state or the kinematics of the aircraft The plugin requested for this project has the sole responsibility of calculating the forces and moments acting on the aircraft, and must not make any physical change to the aircraft, which includes the environment state of the aircraft and the kinematics of the aircraft.

Reuse code from references when possible For this project, much of the code required to make the simulation run already existed, so for efficiency it was our responsibility to reuse existing code whenever possible. The Gazebo simulation interfaces with PX4 code, and the entire PX4 code base already existed on GitHub as a repository, so our team forked that repository in order to utilize that code.

Plugin must work, but does not need to be performant Due to lack of time, limited expertise, and lack of access to proprietary equations, our plugin must work, but does not need to realistically calculate the forces and moments acting on the aircraft. Therefore, the task of this project was to make sure the plugin accepts the proper inputs, compiles to Gazebo, and produces outputs, but the outputs did not need to be accurate to real life.

Development and testing occurs in a Linux environment While the technologies used for this project such as PX4 and Gazebo work on Windows and Mac operating systems, these technologies are utilized much better on a Linux operating system, and offer more capabilities and less pitfalls when used in a Linux environment. Two of the team members had native Linux computers, so this was not a problem. The other two team members used a Linux virtual machine to ensure everyone was working in a Linux environment.

2.2 Functional Requirements

Parameters Our plugin must accept specific parameters which describe the aircraft's current environment and position in order to calculate the forces and moments acting on the aircraft. These parameters include the orientation of the aircraft relative to the horizon (attitude), its velocity as it flies through the air, and the direction and speed of any wind in the area. Additionally, every aircraft has surfaces on its body which can be moved or rotated by a pilot or pilot software in order to control the aircraft during flight, and these are called control surfaces. Examples include left and right ailerons, which are flaps on the wings which allow the aircraft to bank. The current positions of these control surfaces must also be passed as parameters to our plugin as their purpose is to affect the aerodynamic behavior of the aircraft.

Compilation Our plugin will output the forces and moments acting on the aircraft and Gazebo will receive those forces and moments in order to simulate the aircraft, so the plugin needs to compile to Gazebo so that Gazebo can properly receive the information.

Outputs Our plugin will produce a total of six outputs, which are the forces and moments acting in the x, y, and z directions of the aircraft. Since our plugin is a skeleton, the values of the forces and moments do not need to be correct, but the plugin must produce some arbitrary value for all of the forces and moments and output those values to Gazebo.

3 System Architecture

The first consideration made concerning our plugin's architecture had to be the software it interfaces with, namely Gazebo flight simulation software and PX4 flight control software. As an overview of this relationship, refer to the diagram in Figure 1. PX4 is a flight control software that can fly many different types of aircraft, including fixed wing aircraft and quadcopters, both in real life and by interfacing with simulators. PX4's role in the project was to control our test aircraft, and was not modified by us in any way. Gazebo is the main simulator, and is responsible for most aspects of the flight, including wind, air pressure, buildings & other obstacles, as well as the aircraft. Gazebo accomplishes this by utilizing many specialized plugins, each of which handles one aspect of the simulation. While our plugin is one of the many that constitute the Gazebo simulation, Gazebo was not modified by us in any way.

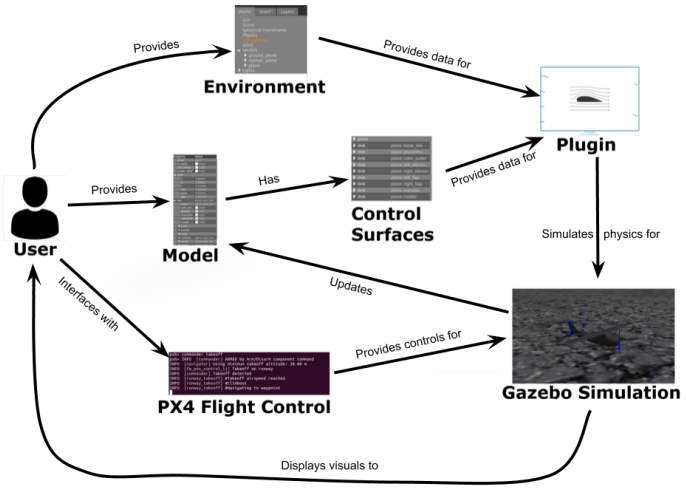

Figure 1: System architecture overview

The user provides Gazebo with an environment model which describes the simulated area where the aircraft will "fly", and a model describing the characteristics of the aircraft. Some of this data is then piped to our plugin, which uses it to calculate the aerodynamic forces and moments acting upon the aircraft at any given time, and outputs these back to Gazebo so that it can update the aircraft's behavior. Additionally, the user interfaces with PX4 flight control software, which uses artificial intelligence to control the aircraft's control surfaces as if it was being piloted.

Our plugin receives parameters from the simulation concerning two things: the environment around the aircraft, and the aircraft's control surfaces as discussed earlier in the Section 2.2. Using this data, our plugin uses a variety of intermediate values such as lift and drag coefficients, and ultimately will find the aerodynamic forces and moments for each control surface. Given there are three axes along which a force or a moment can operate, each control surface will have three forces and three moments which will be calculated by our plugin. The forces and moments of every control surface will then be output to Gazebo so that the aircraft's behavior can be updated based on those forces and moments.

The actual process of calculating those forces and moments from within the plugin has not been implemented by our team as it will require expertise in aerodynamics that we don't have, and instead our client will be adding that functionality later. Ultimately our client will be doing this using control derivatives, which are custom mathematical functions which describe the impact of a control surface on the net forces and moments of the aircraft, given the aerodynamic conditions and control surface position. For example a simple aircraft model might include 4 different control surfaces as shown in Figure 2: left and right ailerons (flaps on the wings that allow an aircraft to bank), an elevator (a horizontal flap on the tail that allows the nose of the aircraft to pitch up and down), and a rudder (a vertical flap on the tail that allows the aircraft to "yaw", that is swivel horizontally). For this aircraft model you would expect there to be a total of 24 control derivative functions: 6 per control surface, where 3 calculate forces and 3 calculate moments. In this case, these 24 control derivatives would then correspond to 24 outputs to the simulation software.

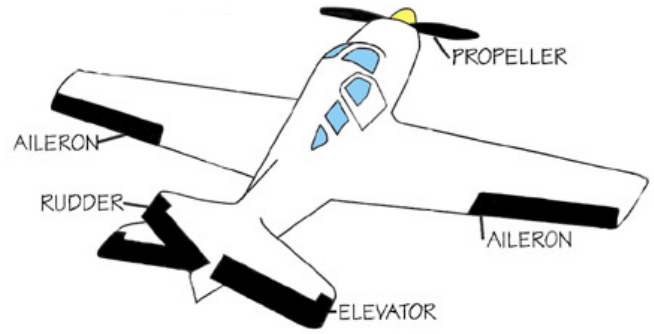

Figure 2: Simple aircraft control surfaces

4 Technical Design

Our plugin has two phases where it can get information from the main Gazebo simulation, as shown in Figure 3. The first phase is implemented by the constructor and `Load` functions, which are called once at the very beginning of the simulation. As such, it is used to load data & identify pointers that will not change during runtime, bind functions so Gazebo knows what to do, and perform other singleton, non-constructor tasks. The second phase is a cycle involving the `OnUpdate` function, which is called whenever the Gazebo simulation updates and so is called many times in a single simulation. The `OnUpdate` function handles loading dynamic data, which is data that changes over the course of the simulation, as well as calculations that use dynamic data. Figure 4 shows a visual depiction of which data is static, and which data is dynamic.

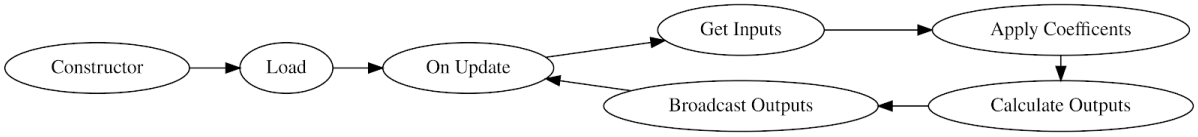

Figure 3: Plugin control flow

During the `Load` function, our plugin identifies and saves the model pointer, the sdf pointer, and the world pointer for later use. These pointers will not change for the duration of the simulation, and so should only be fetched once. The model pointer and the world pointers will be used later during the update cycle to retrieve dynamic information about the aircraft model and its environment. The sdf pointer is used to get data from the Simulation Definition File, or SDF file. The purpose of the SDF file is to contain static data needed for the simulation, and so for our plugin the SDF file contains all 24 control derivatives, the center of pressure, the base link, and other parameters, as they only need to be loaded once. Additionally during this phase we bind the `OnUpdate` function, which is part of the second phase, to the Gazebo event `ConnectWorldUpdateBegin` during the Load function. Binding the `OnUpdate` function to this event tells the main Gazebo simulation which simulator function to call on each update. We hard coded this binding because we only have one `OnUpdate` function, but it is possible to have several update functions, and choose which one to bind based on parameters in the SDF file at runtime. As such, this is done in the `Load` function and not in the constructor.

Once the `Load` function finishes operation, the simulation enters the second phase which is the update cycle. Within the `OnUpdate` function, the current pose (orientation) of the model and each of the control surface positions are obtained using the model pointer. These attributes are retrieved everytime because they constantly change throughout the simulation. Similarly, the current state of the wind and air pressure can be accessed via the world pointer. Once all of the dynamic data has been loaded, the `OnUpdate` function calculates the forces and moments acting on the aircraft at that instant in time. The exact calculations will vary for different classes of aircraft, but the calculations will often utilize the static control derivatives and the dynamic pose data. Once the calculations are complete, the `OnUpdate` function will apply the focus and moments to the center of pressure of the base link, both of which are static data.

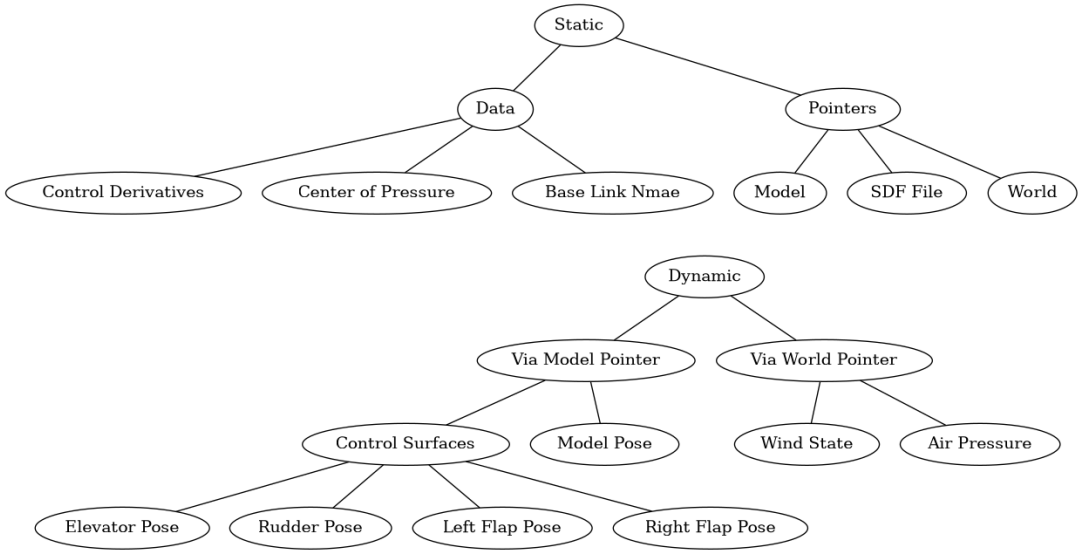

Figure 4: Data model for the plugin

The flow of data in our plugin has been visualized as the data flow diagram in Figure 5. This diagram shows the flow of data, including the responsible function and action being performed. The Load function loads static data from the SDF file and the main Gazebo simulation into long term storage (RAM) to be used many times throughout the duration of the simulation. The `OnUpdate` function loads static and dynamic data into temporary storage (CPU registers) to be used for a single function call only. The `OnUpdate` function then calculates the forces and moments, and sends the results back to the main Gazebo simulation to be applied to the model.

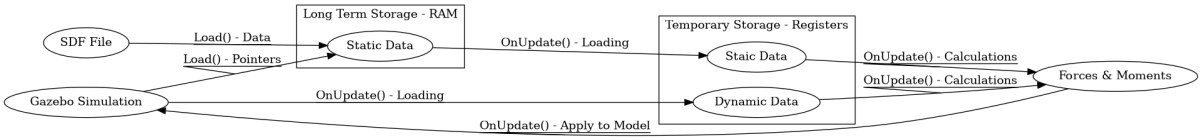

Figure 5: Plugin data flow

5 Quality Assurance

Table 1 details the various elements of our Quality Assurance plan, which we used to guide the general development of the product. The table shows various methods of quality assurance in the left column, and an explanation of how that method of quality assurance improved the system in the right column. Quality assurance methods included technical aspects such as user acceptance testing and integration testing, which encompassed activities such as regular compilation and testing the plugin with a variety of inputs. Holistic methods such as continuous communication were also employed to keep everyone involved on the same page. Both varieties of quality assurance were vital to the process of creating consistent and well-documented code.

Table 1: Quality Assurance Table

Quality Assurance Activity	**Resulting Improvement**
Continuous and accurate communication between the client and team.	Ensured that product development stayed on track, as well as achieved the goals put forth by the client. Avoided any mistakes due to lapses in communication.
Continuous and accurate communication between all members of the team.	Kept development on-track, and also avoided redundant or contradictory work from being produced by different team members.
Realistic goals of what can be accomplished.	Helped make sure that the product produced by the team was usable. Our project remained reasonable in scope in order to avoid spending all 5 weeks on research, as there was too much to cover.
Code should be understandable/readable for further development.	Our product ended up being a skeleton, and not a fully complete plugin. The client has experts in aerodynamics, but we don't. Therefore the final product was made to be relatively easy to adjust for the implementation of more complex equations.
Plugin should be regularly compiled.	Making sure the plugin compiled ensured at a basic level that there were no compilation errors. Making sure the code can run was always high priority.
When testing, making sure our plugin is being added to Gazebo's plugin library, and is being used.	It was highly important that Gazebo was using our plugin during functional testing. Fundamentally, if we weren't testing our own plugin when we ran a simulation, the tests were useless.
Feeding the plugin multiple different inputs, and testing to see that the plugin is using them.	We wanted to ensure that when the plugin is given a set of control derivatives, they were used during the simulation. Therefore making sure input was dynamically being accepted was a key feature to test for.
Creating a model that utilizes our plugin to use with Gazebo.	We aren't experts on what the inputs of the plugin would be, or what the exact output should be, and were told to avoid unit tests. However, by making a model that uses our plugin, we were able to visually check that using our plugin produces varied results with varied inputs.
Code reviews between team members.	Team members viewing and providing feedback on other team members' code ensured the code was as clean and efficient as possible, and also helped eliminate bugs.

6 Results

The ultimate goal of this project was to create a plugin that would take in some values describing the environment and aerodynamic conditions around the aircraft, and could then use those values to update the aircraft's movement and orientation. The plugin skeleton was completed as per specification by the client, and is able to take inputs from a model, and output forces and moments to the simulation environment. We performed dynamic analysis on our plugin by giving it a variety of different inputs for the control derivatives, and watched the aircraft move and tilt in the expected ways in response to those inputs. For example, if the plugin was given a positive input for the moment about the y-axis, we would expect the aircraft's nose to tilt downwards, and when simulated we have watched it do exactly that. Additionally, our compile script successfully built our plugin within Gazebo, so our integration testing was successful.

6.1 Unimplemented Features & Further Extensions

In terms of future work, the plugin is still incomplete in the original project scope functionality. It would need to have a full black box equation that would take the specified inputs, and transform them into the specified outputs using aerodynamic equations. The aerodynamic equations were unclear, and had to remain unimplemented for a functional hand-off to the client. These could be pulled from Simulink auto-generated code, or coded from specific textbook equations.

If we managed to complete the aerodynamic equations, another further extension could've been to add a modular functionality to it. If the model only had some of the aerodynamic coefficients built into it we were looking for, then we could have the plugin adapt to that scenario. Eg. If we only are given a few of the control derivatives, the control surfaces that are unexplained could use the `LiftDragPlugin` as a substitute to estimate the forces and moments.

Our current plugin works within the Gazebo simulation, however it may need some adjustment to work under the PX4 architecture. A model that could show control surface deflections, interface with PX4, and use our plugin was outside the scope of our project. Models under this architecture will be necessary for testing a fully complete plugin.

6.2 Lessons Learned

- As a team we learned how important communication is between all the stakeholders of a given project. Whether it was communicating with our client, or with each other, one of the things we regularly agreed needed improvement at the beginning of our project was our general volume of communication. For example, our client believed that field session lasted the entire summer until our meeting during Sprint 3, where we realized there had been a miscommunication and worked with them to modify our project requirements.
- When working with a team in a virtual setting, it's important to be clear about expectations and dividing work since it can be difficult to work together directly and so people more often complete requirements on their own. As a team, there were times where work was distributed unevenly, or people were unclear about what the team expected of them, and so we learned as part of the Agile process to individually detail everyday what we wished to accomplish.
- We gained an appreciation for how important it is to get a good functional understanding of how a code base or piece of software works when you start developing for it. Much of the roadblocks and difficulties we faced during this project came from our having to write code that would function as a part of existing and large software packages, and having to figure out how they worked so we could integrate our code with their existing functionality.
- Trying to convert auto-generated Simulink code to actual code for our plugin was quite difficult. Variable naming in the auto-generated code was pretty confusing, and the purpose of long, multi-line math equations were not explained, leading to great difficulty understanding the code.
- Sometimes the best way to simulate something accurately is to run a simulation that is as limited

as possible at runtime, and instead to do as much work ahead of time as you can. In our case, the client will be developing complex aerodynamic functions to describe lots of the control surfaces on the aircraft, meaning the only truly simulated control surface is the wing and in this case the overall simulation will be more accurate.

7 Appendix

7.1 Open Source Software

- PX4 Autopilot
  - Website: https://px4.io/
  - Github: https://github.com/PX4/Firmware
- Gazebo Simulation Environment
  - Website: http://gazebosim.org/
  - Github: https://github.com/osrf/gazebo

7.2 Sunlight Aerospace

- Website: https://www.sunlightx.com/
- Assistance/Contacts:
  - Sergey Frolov: https://www.sunlightx.com/?page_id=1339
  - Michael Cyrus: https://www.sunlightx.com/?page_id=1363

7.3 Installation Instructions

1. run `compile.sh`
   - May need to `chmod u+x compile.sh` first
2. Run '`export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:./build`' in the `sunlightPlugin` folder
   - This needs to be run once per reset of environmental variables, such as when using a new shell or after a computer restart

7.4 Image Source

1. Figure 2 obtained from http://learningsource.org/Images/basiccontrol.jpg

7.5 SDF Example

This is an example of what our plugin is expecting within the SDF file.

```
<plugin name=``entire_plane" filename=``libsunlight_simulator.so">
        <!-- Control Derivatives -->
        <!-- (l/r)f -> left/right flap; r -> rudder; e -> elevator -->
        <!-- m -> moment; f -> force -->
        <lf_m_x>1.0</lf_m_x>
        <lf_m_y>2.0</lf_m_y>
        <lf_m_z>3.0</lf_m_z>
        <lf_f_x>4.0</lf_f_x>
        <lf_f_y>5.0</lf_f_y>
        <lf_f_z>6.0</lf_f_z>

        <rf_m_x>1.1</rf_m_x>
        <rf_m_y>2.1</rf_m_y>
        <rf_m_z>3.1</rf_m_z>
        <rf_f_x>4.1</rf_f_x>
        <rf_f_y>5.1</rf_f_y>
        <rf_f_z>6.1</rf_f_z>

        <r_m_x>1.2</r_m_x>
        <r_m_y>2.2</r_m_y>
        <r_m_z>3.2</r_m_z>
        <r_f_x>4.2</r_f_x>
        <r_f_y>5.2</r_f_y>
        <r_f_z>6.2</r_f_z>

        <e_m_x>1.3</e_m_x>
        <e_m_y>2.3</e_m_y>
        <e_m_z>3.3</e_m_z>
        <e_f_x>4.3</e_f_x>
        <e_f_y>5.3</e_f_y>
        <e_f_z>6.3</e_f_z>

        <!-- name of the link to apply forces & moments to -->
        <link_name>base_link</link_name>

        <!-- center of pressure location [where forces are applied] -->
        <cp>-0.5 0 0.05</cp>
</plugin>
```