RAM ROBOTICS AND MECHATRONICS

DEVELOPMENT OF GAZEBO-BASED HIGH-FIDELITY SIMULATION ENVIRONMENT FOR MORPHING-WING UAVS

L.P.M. (Louis) Nelissen

BSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
R.S.M. Sneep, MSc
dr. ir. R.A.M. Rashad Hashem
dr. C.G. Zeinstra

December, 2021

074RaM2021
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE

UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Contents

1 Introduction

1.1 Context

Unmanned aerial vehicles(UAVs) are used for all kinds of applications nowadays. These applications include: agriculture, transportation, navigation and the military field. The most frequently used types of UAVs at the moment are: multi-rotor, fixed-wing, helicopter and fixed-Wing hybrid UAVs. In Figure 1.1 photos of these UAV types can be seen. Because these UAVs have static bodies and generate thrust through propellers, the aerodynamic forces that work on them are relatively simple to describe. The aerodynamic forces that work on these types of UAVs are understood and can be simulated in high-fidelity simulators such as Gazebo, Webots and Coppelia Sim.(1) Although there are flapping and morphing wing drones being developed, the aerodynamic forces that work on them are not fully understood and documented yet. Because these forces are not fully understood, it is not possible to correctly model and simulate flapping and morphing wing drones at the moment. Because there is no accurate way to model and simulate these UAVs, the control possibilities for them are severely limited. The lack of these control possibilities makes it difficult for them to compete with other types of UAVs.

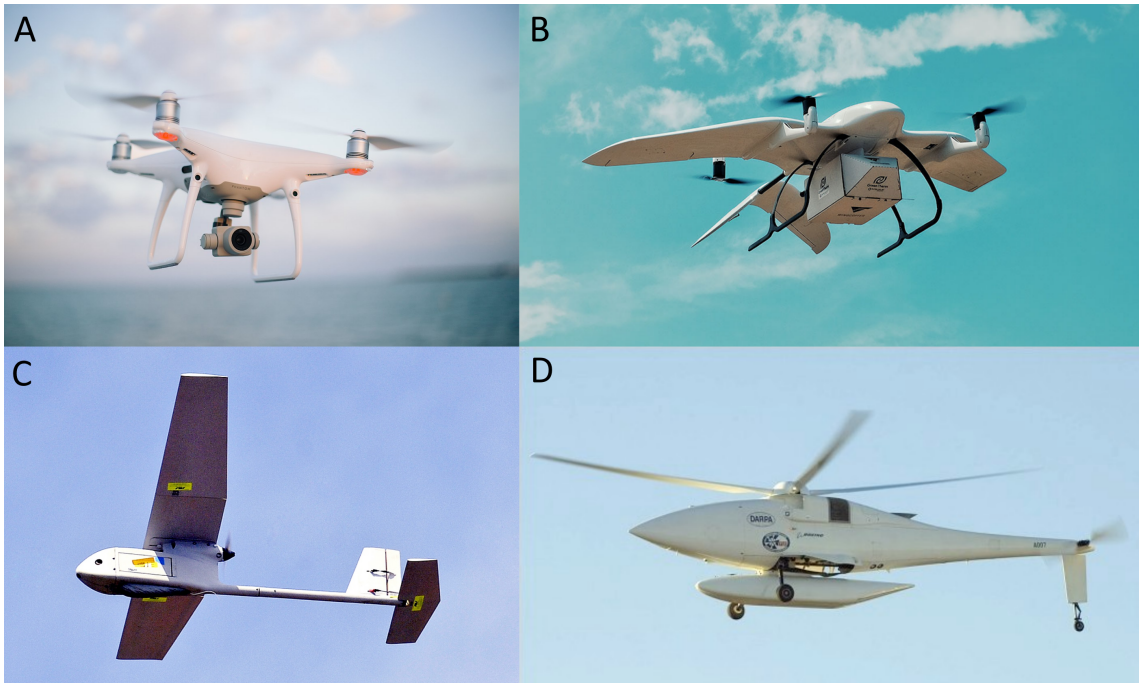

Figure 1.1: Most common UAV types: A. Multi-rotor UAV (Credit: Josh Sorenson), B. Fixed-Wing Hybrid UAV(VTOL) (Credit: Akash 1997 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=94541277), C. Fixed-wing UAV (Credit: U.S. Air Force photo/Dennis Rogers), D. Helicopter UAV(Credit: US Army)

1.1.1 The Portwings project

In Figure 1.2 the Robird that was developed at the University of Twente can be seen. The Robird is an example of a flapping wing UAV which has limited control functionalities. With the making of the Robird, "a system that resembles the steady flapping behaviour of its biological counterpart and can fly untethered, stably up to 80 km/h in up to 5 Beaufort wind speed"(2) was created. Even though this is quite impressive, the Robird has its limitations due to the lack of control capabilities. "The Robird cannot take off on its own, cannot perch, uses symmetric flap-

ping, steers using a number of manifolds placed on the tail and has a minimal autonomy and a restricted operation time due to power consumption"(2). The goal of the Portwings project is to understand and model the generation of vortices by mechanical deformations of surfaces and volumes of the wings and body of birds as well as understanding how these can be controlled. This will be done through the use of port-Hamiltonian(PH) system theory. Port-Hamiltonian system theory allows for the analysis, design, modeling and control of complex dynamical systems. This is possible due to "the emphasis on power flow between subsystems, the separation of the interconnection structure of the system from its components' constitutive relations and the exploitation of this separation in the analysis and control of the system"(3). The Portwings project can use this for the coupling of fluid dynamics theory to dynamically changing surfaces and their actuation. To validate the modelling and scientific understanding of these goals, they aim to build a new robotic bird with unprecedented flight dexterity, asymmetrical flapping capability, flow adaptation and take off and landing similar to a real bird.(2)

Figure 1.2: Robird (Credit: Clear Flight Solutions)

1.1.2 How this project relates

The robotic bird currently used by the Portwings project is controlled as if it is a fixed wing plane. The bird's flapping mechanism is connected to the throttle that would normally control the propeller. The steering of the bird is done through a rudder and elevators similar to those of a fixed-wing plane. In order to realize the improved robotic bird it needs a controller. To make a controller the new bird needs to be modeled and simulated. Currently there is no support for flapping and morphing wing aerodynamics in robot simulators. To create support for these aerodynamics first the current implementation of aerodynamics in these simulators needs to be understood and its applicability for this class of aerial robots needs to be investigated. In this project the aerodynamics support of Gazebo applied on the model of a Cessna c172(fixed-wing plane) will be investigated and documented. This research and the recommendations made from the findings are step one in the creation of a Gazebo-based high-fidelity simulation environment for morphing-wing UAVs.

1.2 Research questions

The goals of this research are to understand how the current aerodynamics implementation of Gazebo is implemented and how it could be extended to simulate flapping and morphing UAVs. To achieve these goals 4 research questions are formulated.

1. How is the current aerodynamics support of Gazebo engineered?
2. How are Gazebo compatible aerodynamic robot models made?
3. What are the functionalities and shortcomings of the software module that is used to simulate aerodynamic forces?
4. How could Gazebo its aerodynamics support be extended to allow for morphing and flapping wing simulations?

1.3 Approach

Before this project a literature review was done. In this literature review, 5 papers where Robot operating system(ROS) and Gazebo are used to simulate and auto pilot fixed wing and multi rotor UAVs are summarized and related to each other. The making of this literature review gave insights into the simulation of UAVs in Gazebo and helped prepare for this project.
Because a literature review was already done in preparation of this project, a 2 step approach is chosen.
First, Gazebo tutorials are to be completed in order to get a thorough understanding of Gazebo and how its models are made.
When the workings of Gazebo simulations are understood the research questions will be investigated. The research questions are investigated using the Cessna simulation, because this is Gazebo's example implementation of its aerodynamics support.
The first research question will be answered through investigation of the files used for the Cessna simulation and the interaction between them.
The second research question will be answered through code analysis of the Cessna model file.
The third research question will be answered through code analysis of the LiftDragplugin and its application on the Cessna.
The last research question will be answered through insights gained during this research.

1.4 Outline

In this report, first some background information on Gazebo and aerodynamic forces is given. Then the research findings are discussed. In the research chapter, first the file structure of the Cessna its aerodynamics implementation is discussed.
Second, the making of a UAV model file is discussed. Third, the LiftDragplugin, which is the software module used to simulate aerodynamic forces in Gazebo, and its application on the Cessna model are discussed.
Last, possible extensions for morphing and flapping wings are discussed.
The conclusion then discusses how the research questions were answered and if the goals were met.
The report is then finished with some recommendations for the implementation of morphing and flapping wings in Gazebo.

2 Background

2.1 What is Gazebo and how does it work

The original creators of Gazebo are Dr. Andrew Howard and his student Nate Koenig. From their need to simulate robots in outdoor environments under various conditions, the concept of a high-fidelity robot simulator was born. "As a complementary simulator to Stage, the name Gazebo was chosen as the closest structure to an outdoor stage. The name has stuck despite the fact that most users of Gazebo simulate indoor environments."(4) The use of Gazebo allows companies to build, simulate and test robots without the excessive time and expensive hardware needed to make a prototype. If these resources are of no concern, Gazebo is still a good tool to test the robotic idea before building it in the real world.
Gazebo allows the user to build a robot and the world the robot will be simulated in. Through the physics engine Gazebo can simulate forces, friction and other interactions between objects. To add functionality to robots or the environment, available plugins can be used or new ones can be made. Through these plugins the user can change the behavior of the world, model, sensor, system, visual and graphical user interface(GUI). Because of these capabilities Gazebo can be used to test algorithms, train artificial intelligence and make robot controllers.(4)
Gazebo was not made for aerial-robotics simulations. Due to the lack of a high-fidelity simulator made specifically for aerial-robots the LiftDragplugin was made to allow basic aerodynamic simulation in Gazebo. In the literature review that was performed it was concluded that using the LiftDragplugin aerial robots composed of fixed wings and propellers could successfully be simulated.(5; 6; 7; 8; 9) The Robird has complicated aerodynamics due to its flapping wings and can therefore not be simulated properly in the current Gazebo configuration. When the aerodynamics support is extended and Gazebo can be used to simulate the Robird, the possibility to make a robot controller could be used for auto-piloting of the Robird. For more information on Gazebo visit their website: http://gazebosim.org/.

2.2 What aerodynamic forces work on a UAV

A UAV in flight has many forces acting on it. These forces can be added up and simplified to a total of 8 forces. The 2 main forces working on the UAV are the gravity force and thrust force. The force gravity imposes on the UAV comes from the earth and therefore is always from the UAVs center of mass down to the earth. The thrust force is generated by a propeller, jet engine or flapping-wing and can be in many different directions depending on the source of thrust. The UAV gains velocity that is generated by the thrust and gravity forces working on it. Once the UAV starts to move it passes through air, this air imposes the other 6 simplified forces upon the UAV. These are the 6 resulting aerodynamic forces that work on a UAV. These composed forces are the lift force, drag force and side force as well as 3 moment forces around the plane axis. These forces are visualized for a fighter jet in Figure 2.1.

The lift force is generated by the difference of air pressure under and above the wing. This difference in pressure is generated by the shape and angle of the wing and is illustrated on a 2d airfoil cross section in Figure 2.2.

The drag force is the resistance caused by the air which the UAV is moving through. The greater the angle the wing has the greater the lift and drag force, this only happens until the angle where stall occurs. Stall occurs when the flow detaches from the top of the wing and creates vortices as can be seen in Figure 2.3. The occurrence of stall results in a decreasing Lift force and increasing drag force.

The side force comes from the angle between the wind and the forward direction(X in Figure 2.1) and the side-winds generated by the vortices that occur at the wingtips. The pitching mo-

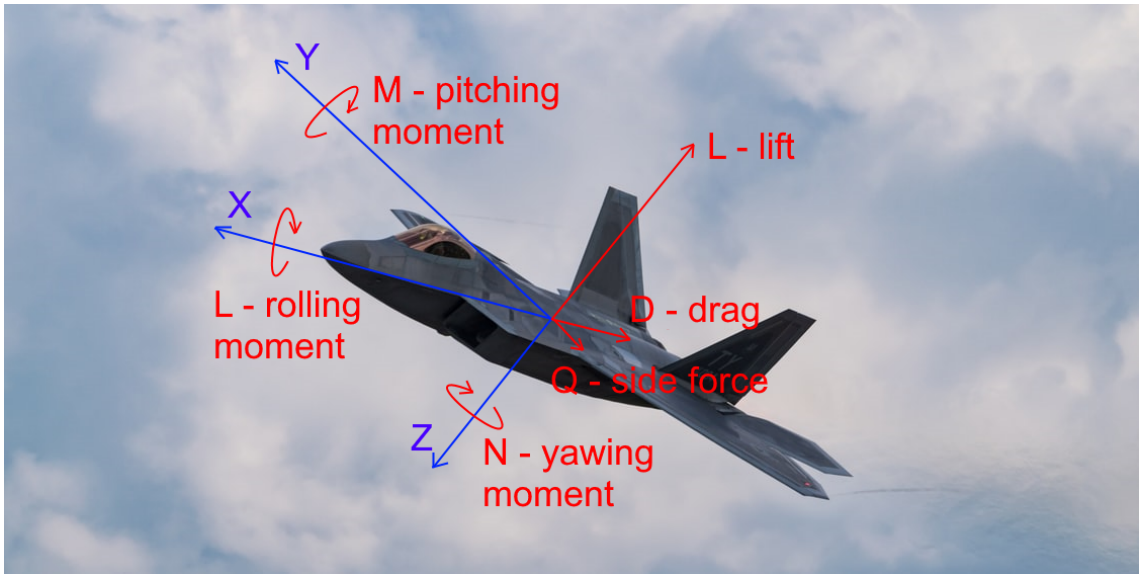

Figure 2.1: The 3 linear and moment forces that work on a UAV (North, East, Down (NED) coordinate system)

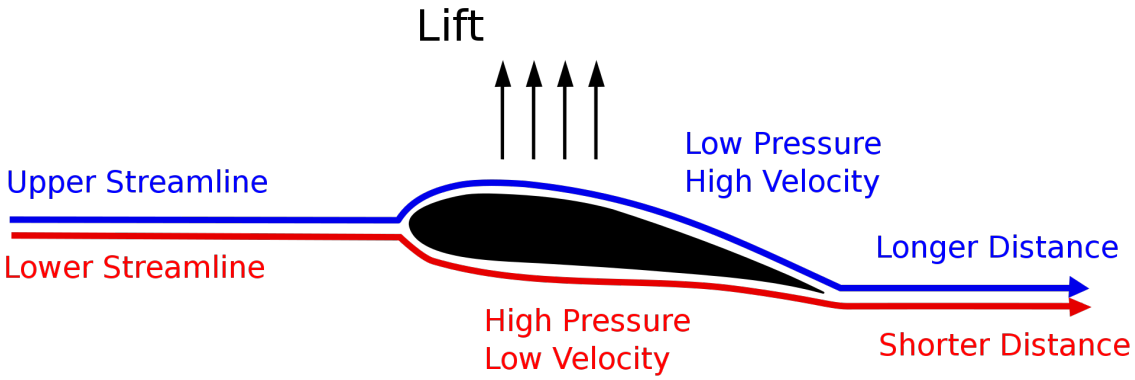

Figure 2.2: Generation of lift force explained (Credit: Tal, Equal transit-time NASA wrong1 es.svg, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=77595905)

ment is caused by the forces in the lift-drag plane. The rolling moment is caused by the forces in the lift-side force plane. The yawing moment is caused by the forces in the drag-side force plane. Using these aerodynamic forces, the thrust force and gravity the translation and rotation of a UAV can be calculated.(10)

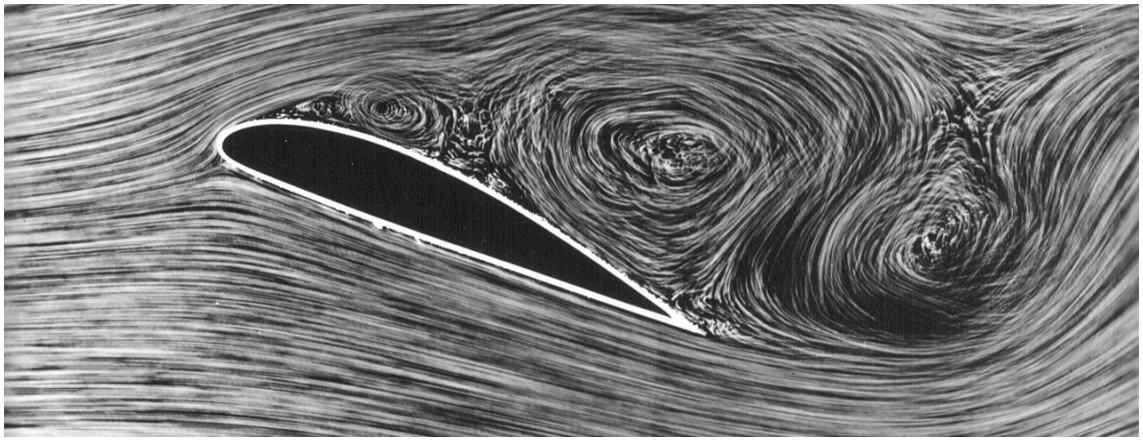

Figure 2.3: The effect of stall visualized (Credit: DLR, CC-BY 3.0, CC BY 3.0 de, https://commons.wikimedia.org/w/index.php?curid=61072555)

3 Research

During the research part of the project the aerodynamics support of Gazebo is investigated through the analysis of the files used in the Cessna simulation. First, the findings on the file types and file structure are discussed. Second, the making of a UAV model is discussed using the Cessna model as an example. Third the LiftDragplugin and its application on the Cessna model are discussed. Last, the possibility of extending the LiftDragplugin for morphing and flapping wing simulation is discussed.

3.1 File Structure

The file structure of the Cessna simulation can be analyzed through the file types used and the interaction between them.

3.1.1 File types

For the simulation of the Cessna in Gazebo, 8 different file types are used. These file types are split up into 4 categories based on the purpose they serve, this separation can be seen in Table 3.1. In the next paragraphs the listed file types will be discussed.

World files	Model files	Plug-in files	Message files
.world	.sdf	.cc	.proto
	.config	.hh	.pb.h
	.dae	.so	

Table 3.1: File types categorized

World files

The .world file type is an XML file type unique to Gazebo. In this file all elements of the simulation are contained. These elements include: robots, lights, sensors, static objects and plugins. The Gazebo server(gzserver)uses this file to create the simulation world and to load all the elements needed for the simulation into that world. In the Cessna simulation this is the cessna_demo.world file, this file contains the declarations for all files listed in table 3.2 except for the message files.(11)

Model files

The .sdf file type is another XML file type unique to Gazebo. Gazebo uses .sdf files instead of the standardized .urdf files to create its robot models and object models. The .sdf file holds all parameters needed for Gazebo to make the model that is described in it. It can also store information on whether it should be rotated and where it is to be placed into the simulation world. In the Cessna simulation there is a model.sdf file for both the Cessna and the sun model. The .config file type is used to store the meta data of the model for example the name of the maker and the version of the model. In the Cessna simulation there is a model.config file for both the Cessna and the sun model. The .dae file type is used to store 3d meshes in XML format. The .dae files can be used in .sdf files to create collision bodies and visuals for parts of the model. In the Cessna simulation multiple links are made using these .dae files this will be further discussed in the making of a UAV model section. As is discussed and as can be seen in table 3.2 both the sun and Cessna have a model.sdf and model.config file. This is because a Gazebo model consist of a map containing its model.sdf, model.config and possibly its .dae files if meshes are used for the models construction. The code of the Cessna model.config file

can be seen in appendix B1 and some code snippets from the model.sdf file can be seen in appendix B2.

Plugin files

The .so file type is a shared library file. This is the file type that is used by Gazebo plugins. There are currently 6 types of plugins for Gazebo: world, model, sensor, system, visual and GUI. These plugins allow the user to change the functionality of Gazebo. A world plugin could for example change the gravity used in the simulation and a GUI plugin could add an overlay displaying coordinates of a model. The plugin files used in the Cessna simulation can be seen in table 3.2. The LiftDragplugin is a model plugin that makes calculations and applies forces on the links it is specified to. The CessnaGUIPlugin maps keys on the keyboard to certain messages. The CessnaPlugin is a model plugin that uses the messages it receives to control the angles of the propeller and control surfaces of the Cessna. Since the .so file type is not in human text, these files are not written by the maker of the plugin. The .so files are generated from the written .cc and .hh files which are the C++ source code file and its header file respectively. The maker of the plugin defines the plugin type and what the plugin is supposed to do in these files. The maker then uses the CMake and the Make function to compile these files into the shared library file. The shared library file that results from this can then be used by the gzserver to perform the specified functionalities. In the Cessna simulation all of the shared library files have their own header and source code file. An edited version of the LiftDragplugin.cc file which calculates and applies aerodynamic forces can be found in appendix A1.(12)

Message files

Gazebo topics communicate using Google protobuf messages. The message types used by the topics are constructed using .proto files. To use the message type constructed in the .proto file, it needs a header file. These header files are generated from the .proto file and is stored as a .pb.h file. Plugins and ROS users can interact with the simulation or each other through the Gazebo topics. In the Cessna simulation the message types constructed in cessna.proto are used to communicate the control messages from the CessnaGUIPlugin to the CessnaPlugin. It is also used by the CessnaPlugin to publish the states of the joints to the state topic. (13)

World files	**Model files**	**Plug-in files**	**Message files**
cessna_demo.world	Cessna/model.sdf	libLiftDragplugin.so	cessna.proto
	Cessna/model.config	libCessnaPlugin.so	cessna.pb.h
	Cessna/meshes(.dae files)	libCessnaGUIPlugin.so	
	Sun/model.sdf		
	Sun/model.config		

Table 3.2: Cessna simulation files categorized

3.1.2 File interaction

The previously discussed file types make use of each other when simulating in Gazebo. For the Cessna simulation the interacting files and how these interact will be discussed. In table 3.2 the files used in the Cessna simulation can be seen. In Figure 3.1 a flow chart of the interaction between the files can be seen. This flowchart is made by analyzing the declarations and use of the other files within the cessna_demo.world file. The files used in the cessna_demo.world file are then analyzed in the same way, from this the ownership and interactions are derived. In this flowchart the filled arrow heads signify the ownership of files and objects. The open arrow-heads signify that the file or object is used by the other file. The line arrowheads signify the file or object applying a force, rotation or translation. What happens in the flowchart is discussed in the next paragraphs.

The cessna_demo.world file declares the world scope. In the world declaration the GUI, ground_plane model, sun model and Cessna model are declared.
Inside of the GUI declaration the CessnaGUIPlugin is declared, this applies the plugin functionalities to the GUI of the simulation. In the CessnaGUIPlugin the cessna.proto file is used for the message types in the Cessna simulation. It logs key presses to certain messages and publishes these messages on the control topic.
The ground_plane model is made by declarations of its collision and visual bodies. To create the visuals of the ground_plane, the grass and runway materials are included from the cloud.
The sun model directory is pulled from the cloud and included in its model declaration.
The Cessna model is made by including the Cessna model directory, within this directory the model.sdf file uses the .dae mesh files to create the visual and collision bodies of the Cessna. Within the Cessna model declaration(in the world file) the other plugins are declared to add functionalities to the model.
The CessnaPlugin is declared under the name cessna_control and takes in the Cessna control and propeller joints as its arguments as well as some PID controller gains. The CessnaPlugin file uses the cessna.proto file for the message types in the Cessna simulation. Through the publishing of messages to the state topic, it communicates the current states and sets the target states of the control and propeller joints.
The LiftDragPlugin is declared under the names of the different surfaces it is calculating the forces for and takes in aerodynamic parameters, link name and control joint name as arguments. The LiftDragPlugin reads the state of the declared control joint that was published by the CessnaPlugin and pulls the velocity of the declared link from Gazebo. It then calculates the aerodynamic forces and applies these on the declared link. In appendix C1, the code of the cessna_demo.world file with its model and plugin declarations can be seen.

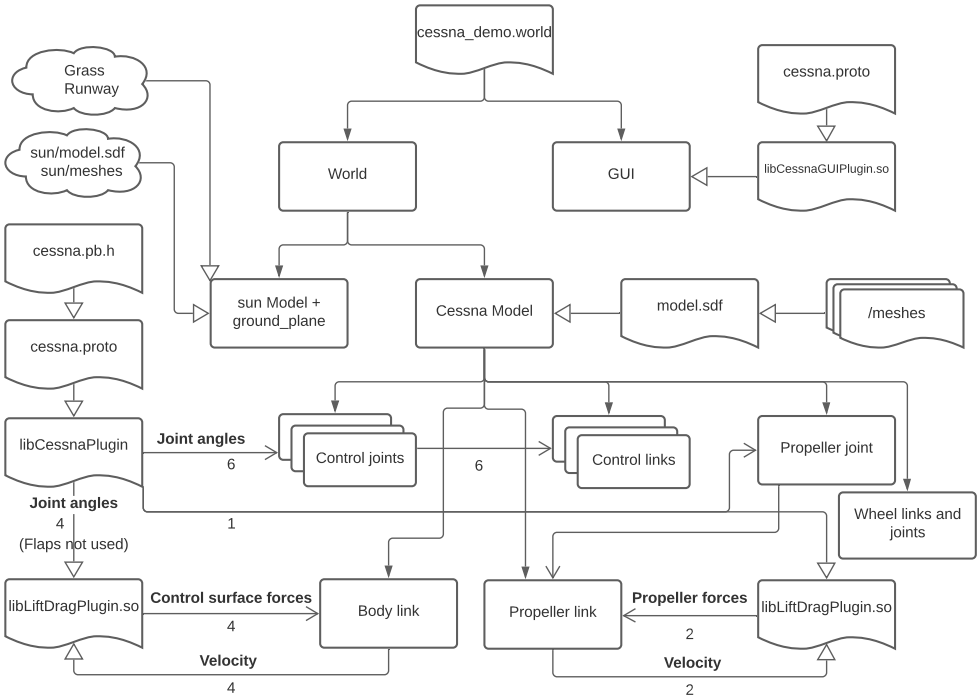

Figure 3.1: File interaction flowchart of cessna_demo.world

3.2 Making a UAV model for Gazebo

The making of a UAV model for Gazebo is best done in the previously discussed .sdf format. The method of making this kind of model will now be discussed using the Cessna model as an example, the config of the Cessna model can be found in appendix B1. First the XML and SDF version need to be listed. The model is then made from 2 types of components: links and joints. The links are the physical components of the model for example a wing. The joints are what connect the links to each other. These can either be a rigid connection or allow for specific movement between the links.

3.2.1 Making links

After declaring the code format the maker should decide what links are needed for the model. For the Cessna model seen in Figure 3.2 these are the body, propeller, flaps, ailerons, rudder, elevator and wheels. When making links it is important to note that the pose(x,y,z,roll,pitch,yaw) of a link is always in relation to the model frame unless otherwise specified. This pose is expressed in the East, North, Up(ENU) coordinate system that is used by Gazebo. Each link has 3 components: its inertial body, its collision body and its visual.
Firstly, the links inertial properties are declared in the inertial function. Here the inertial function uses the links mass, inertial matrix and pose of the center of mass to create the links inertial body. Gazebo can display these inertial bodies in purple. For the Cessna model the inertial bodies can be seen in Figures 3.3 and 3.4.
Secondly, the collision body is made. This is done using the geometry function. The geometry function has sub-functions like box or mesh. Through the sub-function "box", geometry can make a specific box shape. Through the sub-function "mesh" the geometry function can load a 3d file of the type .dae to create the contained shape. The collision function then uses the stated geometry to make a collision body. The collision function has some additional sub-functions that can add physical properties for example friction. These sub-functions can be applied to specific links if needed. In the Cessna model this is used to add roll friction to the wheel surface. Gazebo can display a models collision bodies in orange, this can be seen in Figure 3.5.
Lastly the visual function uses geometry and its sub-functions to create the visible part of the link. The visual and collision bodies should be made with their origin at the links center of mass. This allows for them to share the pose of the inertial and collision bodies. In the Cessna model this is done for the wheels which are made using the cylinder function. For the other plane parts the visual and collision bodies are declared with 0 pose since the meshes used were translated and rotated already. The result of the visual bodies can be seen in Figure 3.2. The code snippets from 2 links of the Cessna model can be seen in appendix B2.

3.2.2 Making joints

After making the links they need to be connected using joints. The Cessna model has 10 joints. In Figure 3.5 the joints can be seen represented by 3 arrows that form a frame. In Figure 3.5, 1 is the propeller joint, 2 is the origin of the model, 3 is the left flap joint, 4 is the left aileron joint, 5 is the elevator joint, 6 is the rudder joint and 7 shows the translation line from model origin to joint position. The other joints in the Cessna model are the flap and aileron joint of the other wing and the wheel joints.
When making a joint there are 4 main attributes that should be considered: the pose, the parent link, the child link and the joint type. Similarly to links, the joints of a model are also positioned in reference to the model frame. The pose of a joint consists of 2 parts: the translation and the rotation. In Figure 3.5 this translation can be seen as the orange lines from the origin of the model frame to the joints. The rotation given to a joint is the same as the rotation given to the child link.(14) In Figure 3.5 it can be seen that the rudder joint(6) is rotated backwards similarly to the rudder itself.

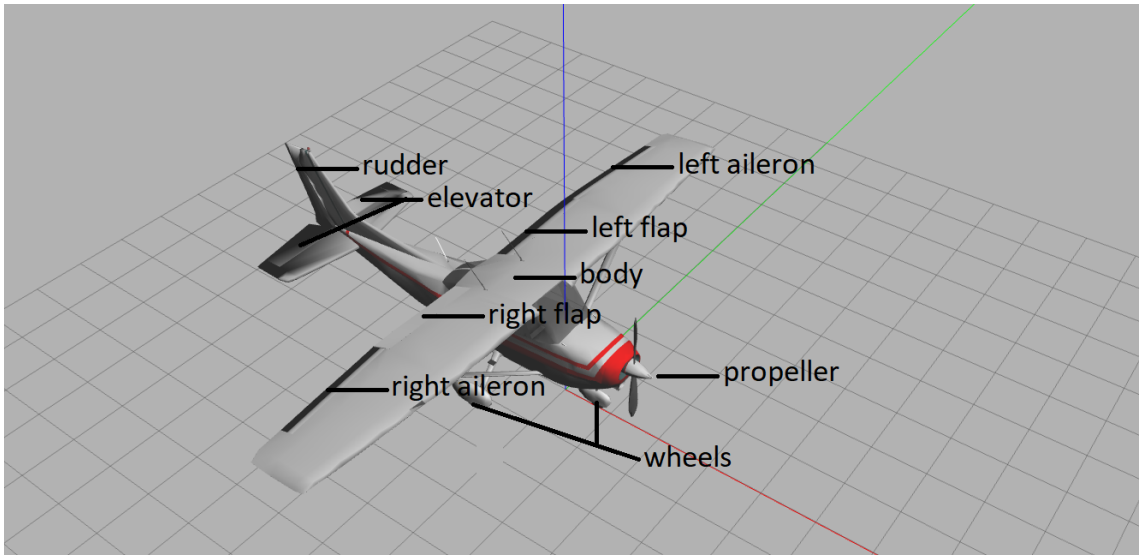

Figure 3.2: Named links of the Cessna model

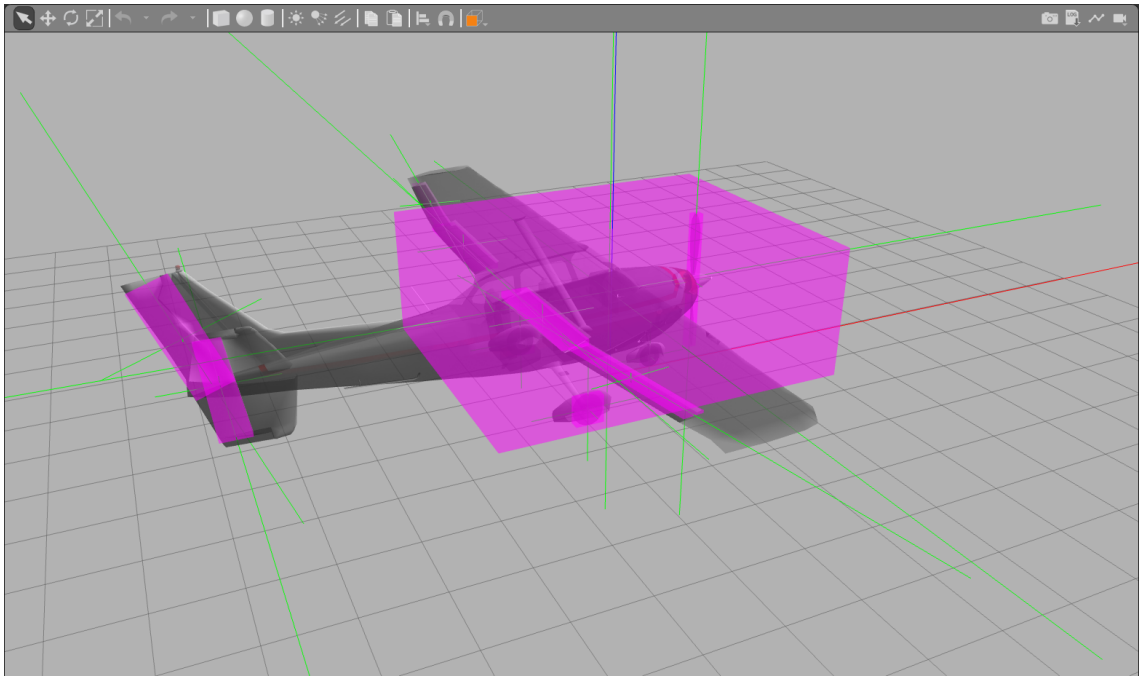

Figure 3.3: Inertial bodies of the Cessna including the main body

The parent link is the main body or the link with the shortest connection to the main body. In the Cessna model, the body link is the parent link for all the joints. The other link connected to the joint is chosen as the child link.
The chosen joint type then determines in which way the movement of the child link is restricted relative to the parent link. In Figure 3.5 it can be seen by the yellow circles that all joints have 1 axis of rotation. This is because all joints of the Cessna are of the type revolute. An overview of the pose of links and joints as well as their interaction can be seen in Figure 3.6. The code snippets from 2 joints of the Cessna model can be seen in appendix B2.

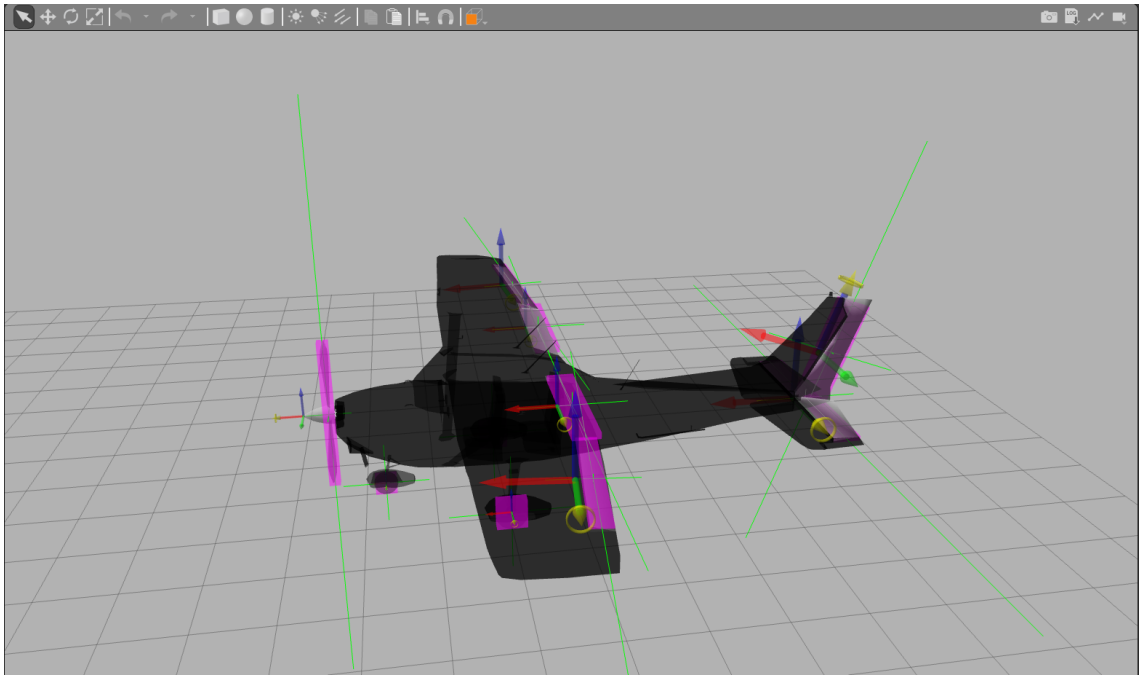

Figure 3.4: Inertial bodies of the Cessna excluding the main the body

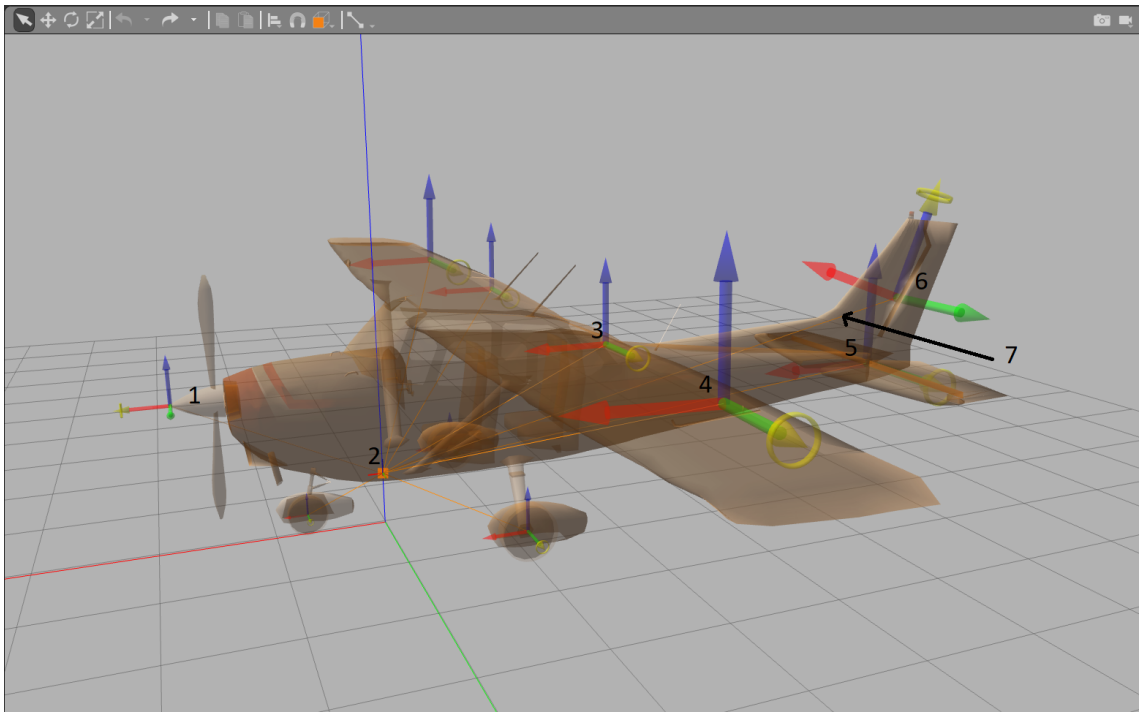

Figure 3.5: Joints and collision bodies of the Cessna labeled

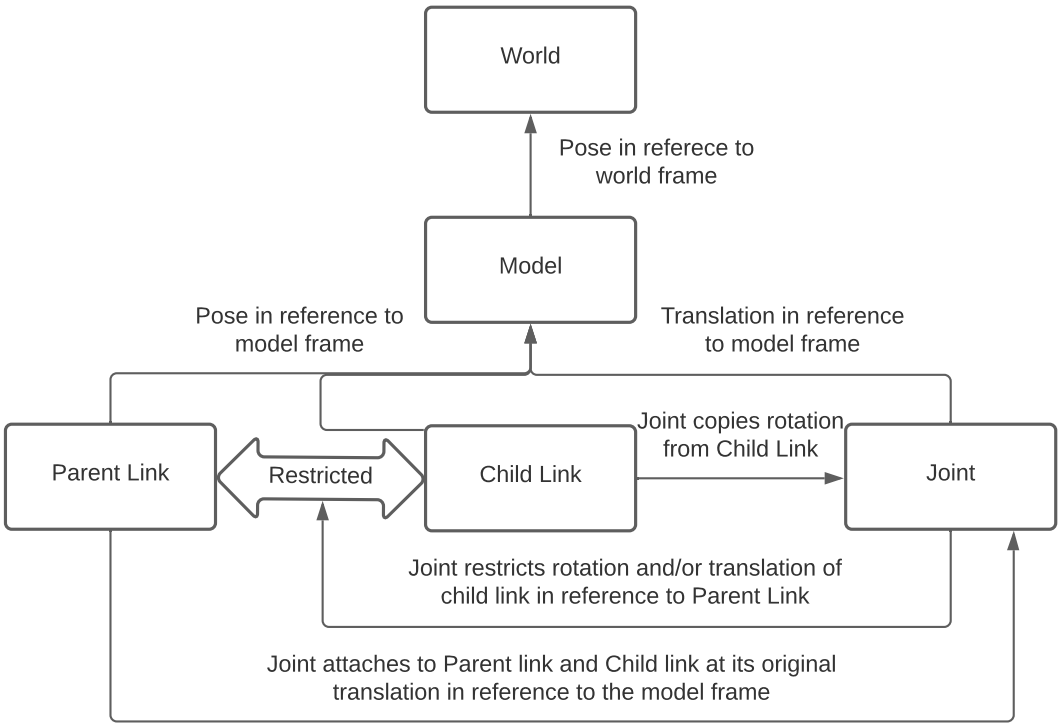

Figure 3.6: Overview of link and joint interaction

3.3 LiftDragplugin

To further investigate how the aerodynamic support of Gazebo is engineered and what the functionalities and shortcomings of the LiftDragplugin are, the LiftDragplugin.cc code is analyzed. When simulating aerodynamics in Gazebo the LiftDragplugin is used to calculate the resulting aerodynamic forces and to apply these on the links of the UAV. In the next paragraphs the assumptions made, the parameters used and the method of calculation used in the LiftDragplugin are discussed. After the code analysis, the implementation of the LiftDragplugin upon the Cessna and the problems with this implementation are discussed for better understanding.

3.3.1 Assumptions

The LiftDragplugin makes 4 assumptions about the simulation of a wing to reduce the complexity of the calculations needed to determine the aerodynamic forces.
The first assumption that is made is that the wing has infinite length. This can be seen through the fact that the finite length 3D wing is evaluated in 2D, which is the equivalent of evaluating a wing of infinite length.
The second assumption that is made is that the airfoil is thin. This can be seen through the fact that tip drag is not taken into account and that the lift coefficient curve is simplified to straight lines. Here the slope from the thin airfoil theory $2\pi/alpha^c$ is used before stall occurs.(10)
The the third assumption that is made is that there are no unsteady effects. This can be seen due to the fact that there is no wind simulated and that the air density is a constant rather than a continuously changing variable which should depend on altitude and position in the world.
The last assumption that is made is that the world is considered to be an open space. This can be seen through the fact that the ground effect is not taken into account when simulating.(15)

3.3.2 Parameters

The LiftDragplugin is initialized for each link it is applied to. When initializing the LiftDragplugin for a link the constants in Table 3.3 are given to it. To apply the aerodynamic forces the LiftDragplugin also needs variables, these variables can be seen in Table 3.4. The LiftDragplugin takes the velocity and pose of the link from Gazebo. The other variables are calculated by the LiftDragplugin using the constants and the information it takes from Gazebo. Some of these parameters are visualized in Figures 3.7 and 3.8.(15)

3.3.3 Calculations

Using the parameters listed in Tables 3.3 and 3.4, the LiftDragplugin can now calculate the aerodynamic forces. In Figure 3.7 these aerodynamic forces can be seen as red arrows at the center of pressure of the wing. Firstly, the wing needs to have a velocity relative to the air in order to generate the lift force L, drag force D and moment M. As mentioned in the assumption section the wind speed is not factored in, therefore the velocity of the wing relative to the inertial frame is used. Secondly, the wing has a shape that helps it generate lift. Depending on this shape the wing will have different lift, drag and moment characteristics. These characteristics can be expressed in lift coefficient c_L, drag coefficient c_D and moment coefficient c_M curves. The LiftDragplugin approaches these curves using the different slope coefficients and the alpha_stall threshold. In Figure 3.9 an example of how the lift and drag coefficient curves and their approximations look can be seen. The red line resembles the lift coefficients and blue line resembles the drag coefficients which were determined by having the wing in an wind-tunnel at different angles and measuring the forces on the wing. The alpha stall line is chosen to be at the angle where stall occurs. The green and yellow line are then each given 2 slopes to resemble the red and blue line respectively 1 slope before and 1 slope after stall occurs. This is implemented this way to avoid the use of a lookup table, which would contain

Constant	Meaning
a0	The angle offset for a link to produce 0 lift
cla	The lift coefficient per alpha when no stall is present
cda	The drag coefficient per alpha when no stall is present
cma	The moment coefficient per alpha when no stall is present
alpha_stall	The alpha value where stall starts to occur
cla_stall	The lift coefficient per alpha when stall occurs
cda_stall	The drag coefficient per alpha when stall occurs
cma_stall	The moment coefficient per alpha when stall occurs
cp	The center of pressure (location in the model frame)
area	The area of the wing from a top view perspective
air_density	The air density used to simulate the links aerodynamic forces
forward	The forward direction of the link at 0 rotation of the model and link
upward	The upward direction of the link at 0 rotation of the model and link
link_name	The name of the link that the LiftDragplugin is applied to
control_joint_rad_to_cl	How much the lift coefficient changes per radian of the control joint value

Table 3.3: The constants of the LiftDragplugin explained

Variable	Meaning
force	The resulting force when adding the lift and drag force
torque	The resulting force based on the calculated moment (not used)
moment	The moment generated by the link(not used)
vel	The linear velocity of the link in relation to the inertial frame
pose	The translation and rotation of the link in relation to the inertial frame
sweep	The angle between the velocity vector and the lift-drag plane
alpha	The angle between the upward and the lift direction(The angle of attack + a0)
lift	The lift force at the center of pressure
drag	The drag force at the center of pressure
q	The dynamic pressure
cl	The lift coefficient
cd	The drag coefficient
cm	The moment coefficient
controlAngle	Angle of the control joint

Table 3.4: The variables of the LiftDragplugin explained

different coefficients at different alphas for both the red and blue line. These values would then have to be interpolated for the exact alpha value of the link. This would increase both the complexity of using the plugin and the simulation time in exchange for improved calculation accuracy.
After determining the lift and drag coefficient, the coefficient of lift is increased or decreased depending on the control joint angle as seen in equation 3.1. The LiftDragplugin is now ready to calculate the lift and drag forces. In equations 3.2 to 3.6 the formulas used to calculated the lift, drag and moment forces can be seen.(16) In Figure 3.10 the flowchart constructed from the code analysis of the LiftDragplugin can be seen. In the flowchart the variables and constants used in all calculations and decisions can be seen. The steps visualized in the flow chart will be elaborated in the next paragraphs.

First, the LiftDragplugin loads all constants listed in table 3.3 from its declaration.
Second, it loads variables from the gzserver: Pose of link, Linear velocity at Center of Pressure and Control joint angle.
Third, the LiftDragplugin starts calculations for the other variables. These variables include the forward and upward direction in the inertial frame, velocity in the lift-drag plane and dynamic pressure.
Fourth, the lift and drag coefficients are determined, here Alpha is compared to alpha stall. If Alpha does not exceed alpha stall, cla and cda are used in the lift and drag coefficient formulas. If Alpha exceeds alpha stall, cla stall and cda stall are used in the lift and drag coefficient formulas.
Fifth, the lift coefficient is adjusted depending on the control joint angle.
Last, the lift force and drag force are calculated using the area of the link, calculated dynamic pressure and their lift and drag coefficients. The lift and drag force are then added and applied at the declared center of pressure.
It is important to note that the moment(torque) calculations are disabled at the time of writing and therefore not explained. In appendix A1 the important parts of the LiftDragplugin code can be seen, in appendix A2 the result of the first round of code analysis can be seen, this is an overview of what happens in which lines of the LiftDragplugin.(15)

$$c_L = c_L + controlJointRadToCL * controlAngle \tag{3.1}$$

$$\vec{V}_{spanwise} = \vec{V} \cdot \hat{s} \tag{3.2}$$

$$V_{effective} = \| \vec{V} - \vec{V}_{spanwise} \| \tag{3.3}$$

$$L = \frac{1}{2} * \rho * V_{effective}^2 * c_L * A \tag{3.4}$$

$$D = \frac{1}{2} * \rho * V_{effective}^2 * c_D * A \tag{3.5}$$

$$M = \frac{1}{2} * \rho * V_{effective}^2 * c_M * \overline{c} * A \tag{3.6}$$

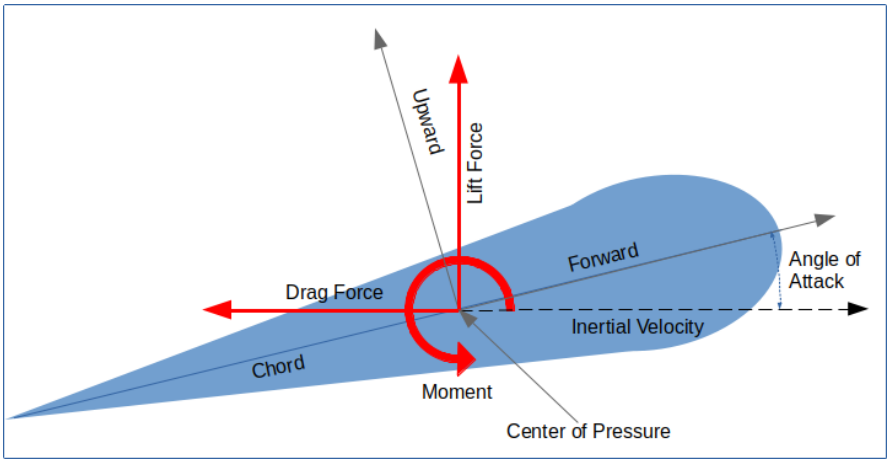

Figure 3.7: Free body diagram of a wing link in the lift-drag plane

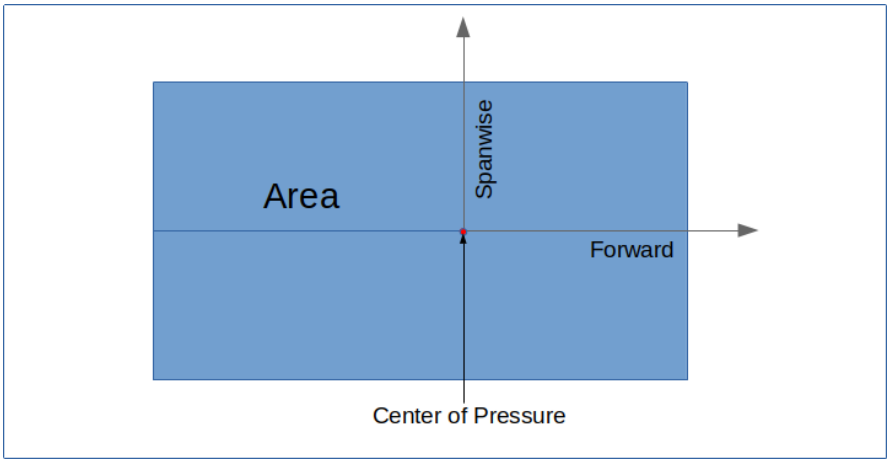

Figure 3.8: Top view of wing link with a chord line

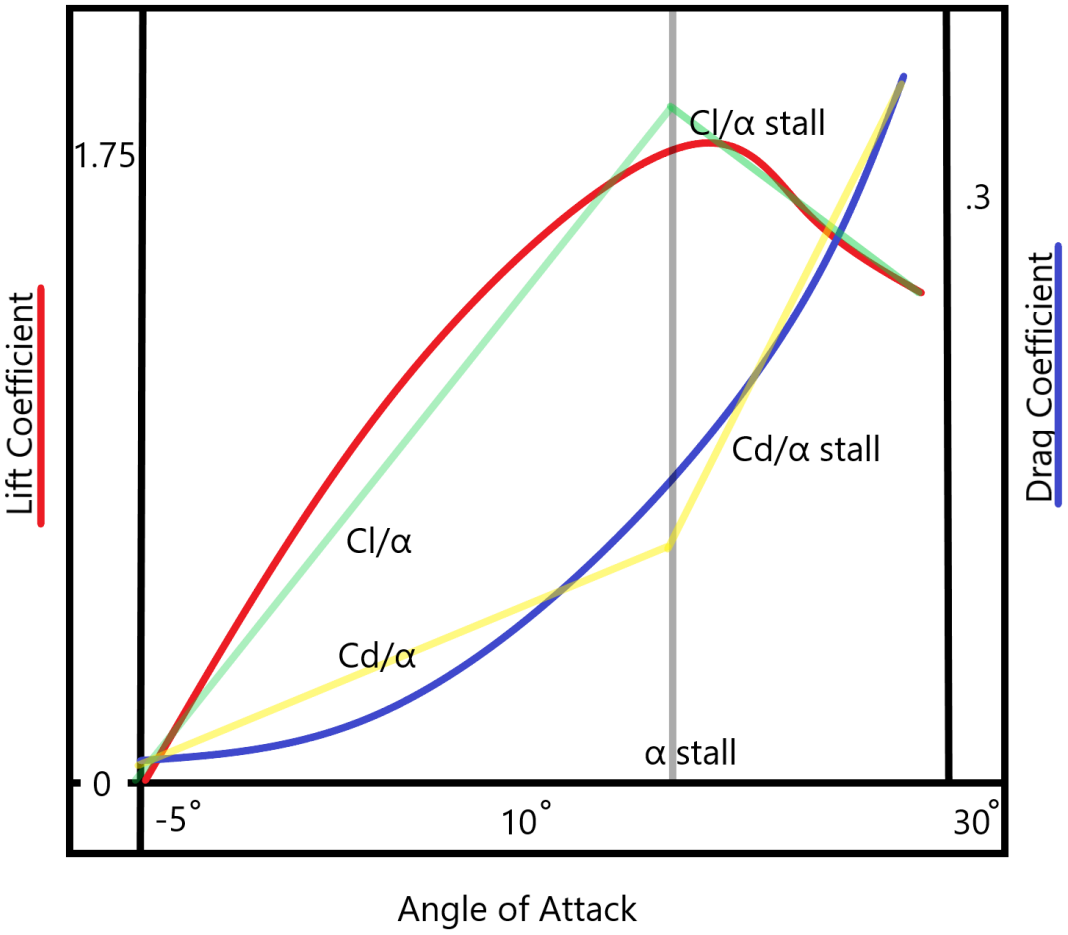

Figure 3.9: Lift and drag coefficient curves approximated by straight lines

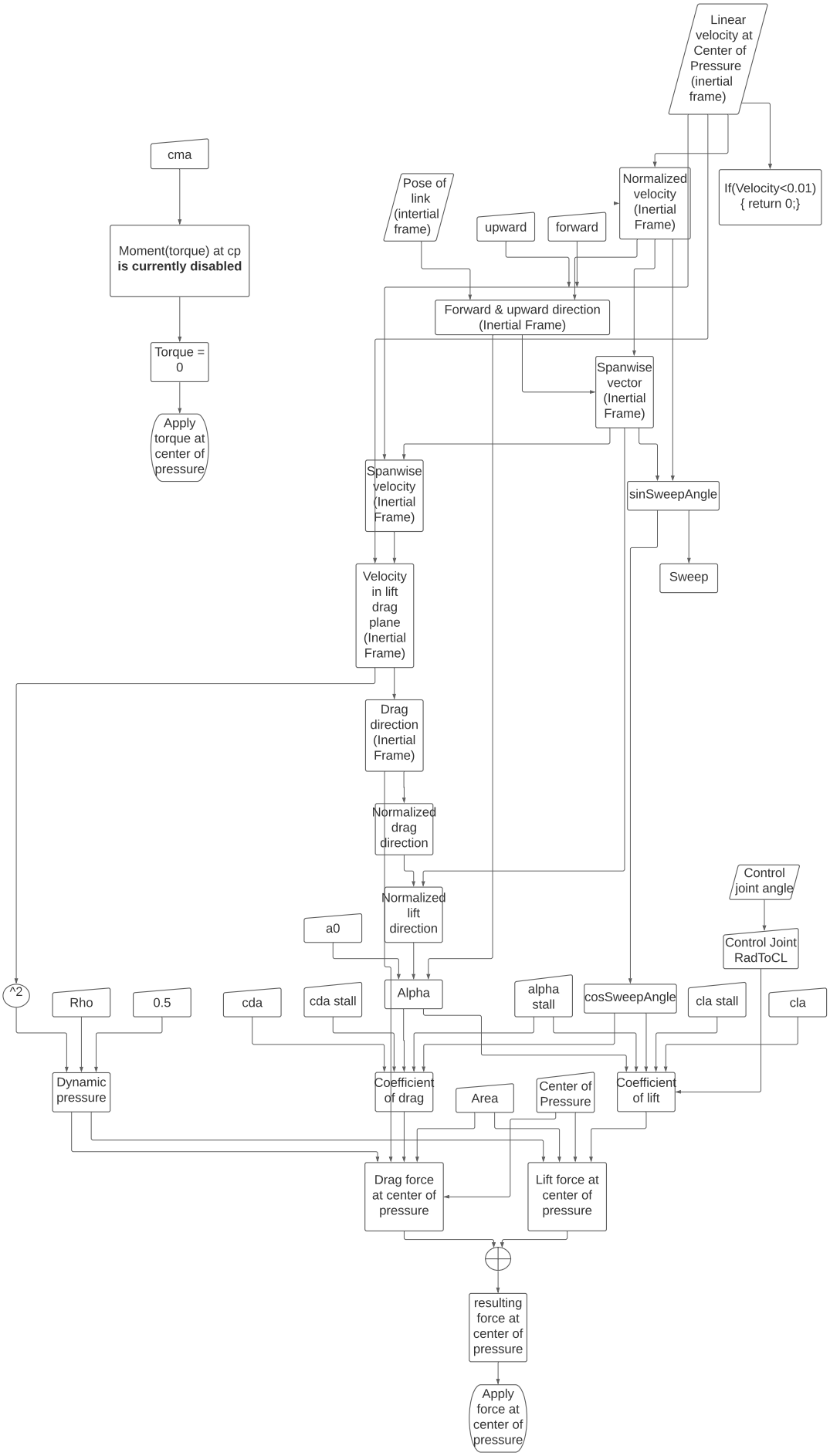

Figure 3.10: Flowchart of the LiftDragplugin

3.3.4 Application on Cessna

The Cessna uses the LiftDragplugin to calculate 6 resulting forces. These forces are: the top propeller blade force, bottom propeller blade force, left wing force, right wing force, elevator force and rudder force. The propeller forces are applied at their centers of pressure upon the propeller link. The other forces are applied at their centers of pressure on the Cessna body. The propeller joint is not initialized as control joint. The rudder and elevator joints are initialized as control joints for the LiftDragplugin and therefore change the lift coefficient according to equation 3.1. The control joints used for the wings are the aileron joints, the flap joints are not used in the LiftDragplugin implementation. Gazebo does the calculations on the results of the forces applied on the inertial and collision bodies of the plane. Gazebo also applies gravity on the bodies, the center of mass of all links can be seen in Figure 3.11.(15)

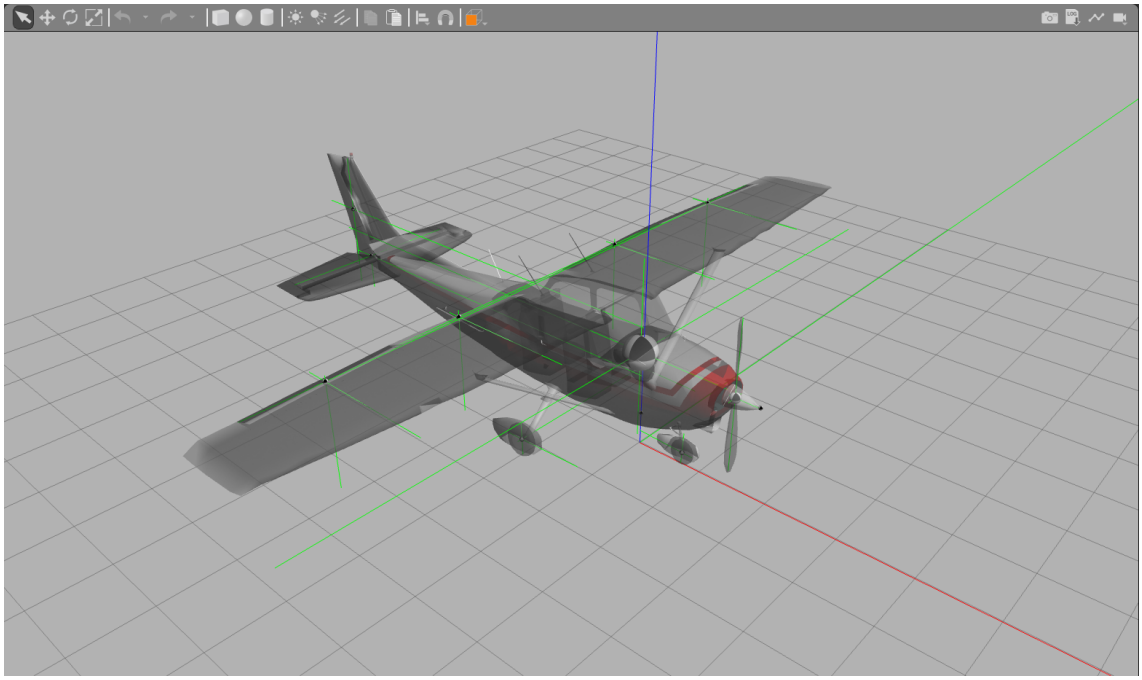

Figure 3.11: The Cessna and its links centers of mass

3.3.5 Problems of the Cessna application

The way the LiftDragplugin is applied on the Cessna model has some problems. Firstly, the LiftDragplugin is applied to the propeller to calculated the thrust force of the Cessna. The use of the thin airfoil theory on rotating blades is completely incorrect and therefore this method should not be used to calculate forces for the propeller. Secondly, the Cessna model has the wings as part of the body link. The wing, rudder and elevator forces are applied to the body at the centers of pressure. The application of these forces of the body is acceptable for the Cessna model since it has fixed wings, but for planes with flapping wings these forces should be applied on the wing links themselves. Thirdly, the centers of pressure are static. When changing the angle of a control surface the "real" center of pressure changes and therefore should be moved. Fourthly, the forces from the LiftDragplugin are only close to accurate when the plane is almost horizontal to the ground plane. Therefore the forces should be modeled differently when the plane is flying at larger angles to the ground plane. Lastly, since the assumptions of thin airfoil theory are used there are only a lift, drag and moment(not used currently) force calculated. As discussed in the background section there should also be a resulting side force and 2 extra moment forces.

3.4 Possible extension for morphing wings

In the next paragraphs the extension possibilities for morphing and flapping wing simulation are discussed with regards to the file structure Gazebo uses, the UAV model file and the Lift-Dragplugin.

3.4.1 Gazebo file structure

The file structure used for the Cessna model simulation, does not impose any limits for the extension to morphing and flapping wing simulations. Through the use of plugins aerodynamic forces can be calculated and applied at a chosen point on the link of interest. The use of C++ files to generate the plugins does however limit the calculation of aerodynamic forces because C++ does not support implicit calculation methods. This will be further discussed in section 3.4.3.

3.4.2 Model file

The method of making a UAV model as discussed in section 3.2 can also work for the creation of flapping and morphing wing models. These models are only limited by the fact that links can not be reshaped during simulation without it being done through the use of joints. An example of what is not possible would be the flexing of a wing as it is flapping. This wing and its flexibility would then have to be approached through the use of multiple shorter wing links connected by joints with specific spring-dampener values.
If a rigid link model for the Robird is made, the Robird would be split up into 6 separate links as can be seen in Figure 3.13. For all links a visual and collision body are made using a .dae mesh file based on the real life Robird. The inertial bodies of the links are determined from the weight and weight distribution of the real life Robird links.
The body(3) is connected to tail(4) using a fixed joint, to the right wing(1) using a revolute joint and to the left wing(2) using a revolute joint. For these joints the body is the parent link and the tail, right wing and left wing are the child links.
The tail is connected to the right flap(5) and left flap(6) using 1 revolute joint each as well. For these joints the tail is the parent link and the flaps are the child links.
An overview of the links and joints used to connect them can be seen in Figure 3.13. This model allows for the flapping motion of the wings and the adjusting of the tail flap angles. The actuation of these joints could be implemented in a model plugin similar to that of the Cessnaplugin.

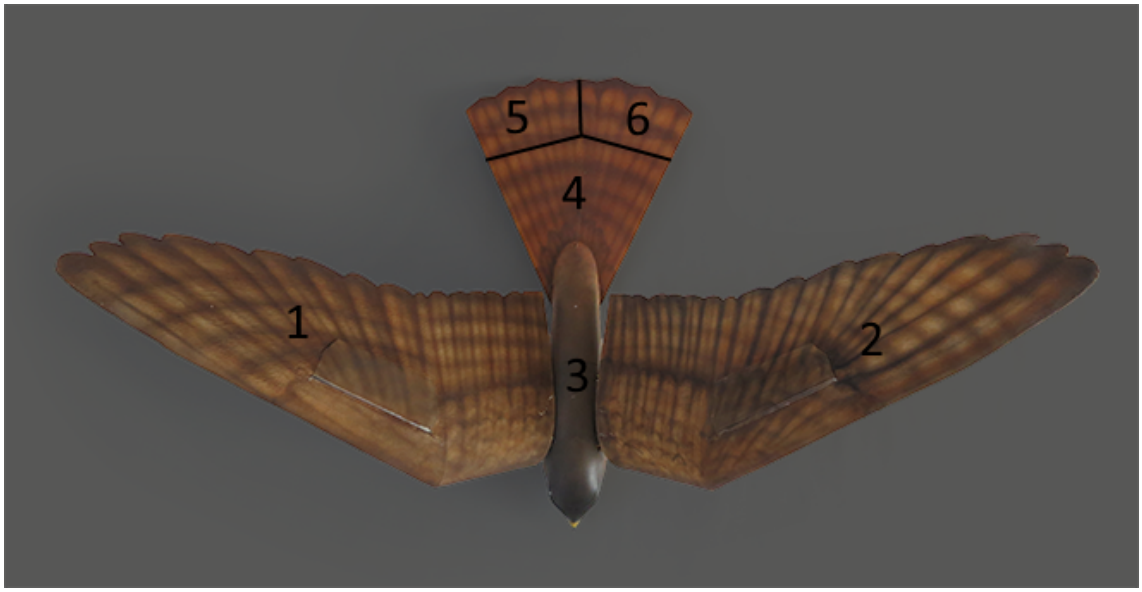

Figure 3.12: Top view of the Robird with its links enumerated.(Credit: Clear Flight Solutions)

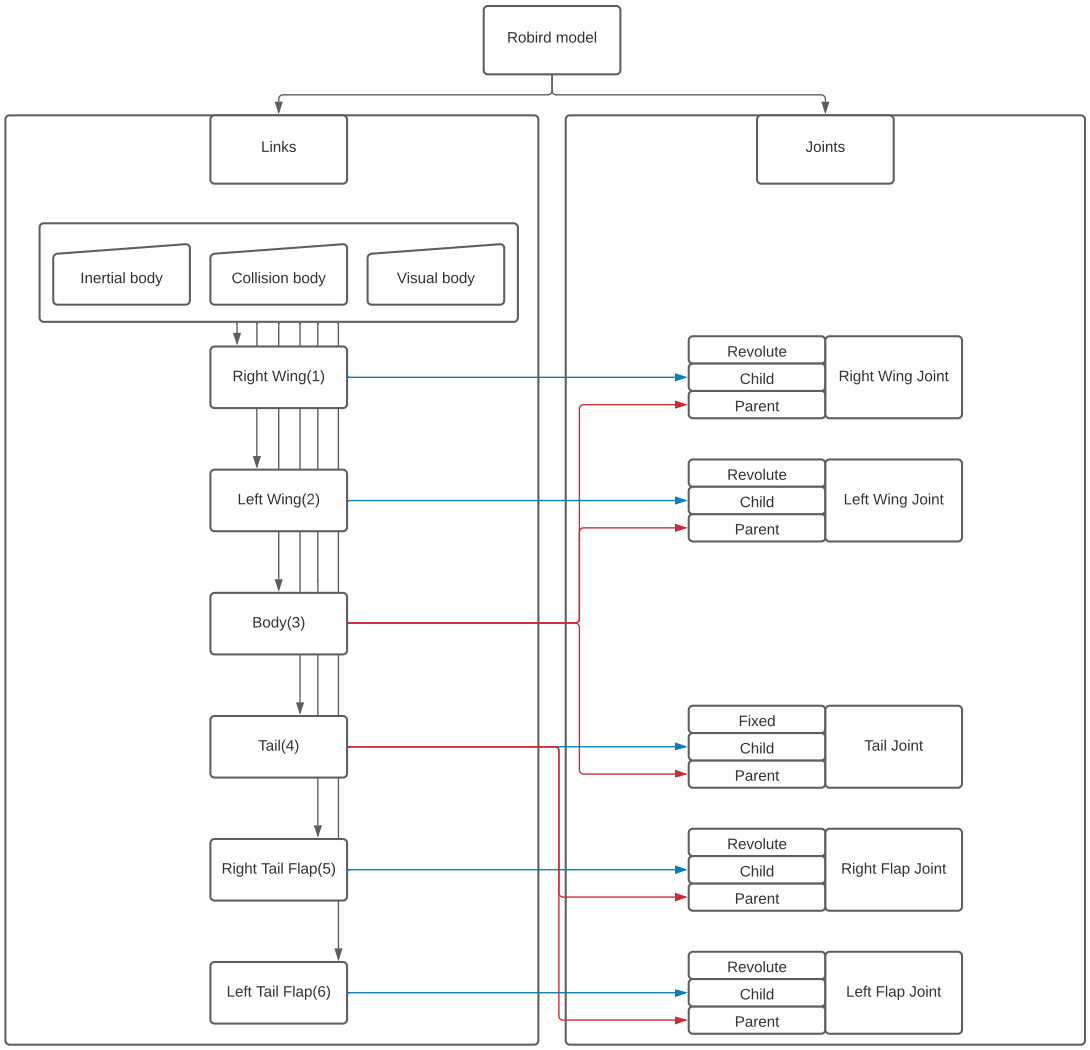

Figure 3.13: Links and Joints needed for the Robird model.

3.4.3 Plugin

To calculate and apply the aerodynamic forces that work on morphing and flapping wings in Gazebo a plugin is needed. The assumptions and the calculations made in the LiftDragplugin are not applicable on morphing and flapping wings. Because of this, these can not be used in the creation of morphing and flapping wing support. Therefore a new plugin that calculates and applies these forces correctly should be made.
Even though the calculations made in the LiftDragplugin can not be used it has 3 properties that could be useful in the development of a new plugin.

1. Application of the plugin on the aerodynamic links individually.
2. Method of loading parameters from the Gazebo simulation.
3. Method of force application on links at a specified point.

This method for loading parameters is the use of a load function similar to lines 67 to 161 og the LiftDragplugin. The method of force application on a link at a specified point can be seen in lines 401 and 402 of appendix A1.
For the force calculation method 2 options were found.
The first option is to calculate the aerodynamic forces in another program. This could be the program where the PH model of the flapping or morphing wing UAV is made. The plugin would function as an application programming interface(API), it would take the forces, joint angles or velocities from the modelling program after every calculation step. The forces, joint angles or velocities that are chosen to be applied can then be forced upon the UAV model in the same way the LiftDragplugin forces are applied. The calculation and simulation steps of Gazebo and the modelling software need to be of the same duration for this setup to work.
The second option is to model the UAV in a program like 20-sim which allows for the model to be exported to C++ format.(17) The resulting C++ files could be changed to load in useful parameters from the Gazebo simulation and to apply the calculated forces to the links of the UAV. These C++ files can then be compiled into the shared library file format Gazebo uses to perform the plugin functionalities. To avoid the repetition of the same calculations, the implementation of this option should be on the whole model not on the individual links. There are 2 downsides to this option. The first downside is that C++ does not support implicit calculation methods which might be necessary for accurate calculations. The second downside is that for every model a new plugin would have to be made from scratch.

4 Conclusion

In the introduction the research questions listed below were introduced.

1. How is the current aerodynamics support of Gazebo engineered?
2. How are Gazebo compatible aerodynamic robot models made?
3. What are the functionalities and shortcomings of the software module that is used to simulate aerodynamic forces?
4. How could Gazebo its aerodynamics support be extended to allow for morphing and flapping wing simulations?

To answer research question 1, the files used in the Cessna simulation were reverse engineered and documented. This question is answered by the whole research chapter except for section 3.4. Because the file types, file structure, model file and LiftDragplugin were all analyzed and should be reproducible from this paper, it can be concluded that the first research question is answered successfully.
To answer research question 2, the steps that should be taken to recreate the Cessna model were documented, to this documentation some methods to improve upon the model are added. To illustrate the types of bodies and joints used by the Cessna, figures were produced. If section 3.2 is used as guide, it should be possible to make a UAV model, it can therefore be concluded that the second research question is answered successfully.
To answer research question 3, first the LiftDragplugin.cc code was investigated. For this code all variables and constants calculated are listed in tables and a flowchart showing the steps made in the code was produced.
Second, the application of the LiftDragplugin upon the Cessna is both illustrated in the file flowchart and explained in the application section.
Third, the problems with the application of the LiftDragplugin upon the Cessna model is discussed for both the Cessna model itself and for a possible morphing wing UAV implementation. From these 3 points it can be concluded that the third research question is answered successfully.
To answer research question 4, in section 3.3.5 the points that are a problem for the extension to morphing and flapping wing support are discussed. In section 3.4 the possibility of extension to morphing and flapping wings is discussed for the file structure, model file and plugin. In section 3.4.1 it is found that the file structure does not impose any limits on the creation of morphing and flapping wing support. In section 3.4.2 it is found that even though flexible links can not be created properly they should not be necessary for the simulation and could be approached by shorter links connected through joints. Additionally a proposed model for the Robird is discussed and illustrated. In section 3.4.3 it is found that the LiftDragplugin can not be extended to allow for morphing and flapping wing support. The parts of the LiftDragplugin that could be used in the creation of a plugin that supports flapping and morphing wing simulation are listed. Additionally 2 methods that could be used to make this new plugin are discussed. It is concluded that the extension possibilities for morphing and flapping wing support were investigated and documented, therefore the fourth research question is answered.
Having answered the 4 research questions, it can be concluded that the current aerodynamics support of Gazebo is understood and documented. It can also be concluded that insight was gained into the development of a Gazebo-based high-fidelity simulation environment for morphing-wing UAVs. Recommendations on how to proceed with this development are made in the recommendations chapter.

5 Recommendations

Researching the current aerodynamics implementation and extension possibilities it was found that it should be possible to create flapping and morphing wing support for Gazebo.
For the Robird model file it is recommended to use a similar model to the one presented in section 3.4.2. The making of flexible wings out of link sections and joints is not recommended, because this complicates the application of the calculated forces and will most likely not have a positive effect on the simulation accuracy. If a rigid link model is made and then found not to be sufficient. This model can always be extended by replacing the insufficient links with multiple partial links connected by joints.
Since the LiftDragplugin can not be extended to simulate flapping and morphing wing UAVs, it is recommended to create a new plugin that can calculate the correct forces. In section 3.4.3, 2 possible ways to make such a plugin are discussed. It is recommended to first implement option one: a plugin that functions as an API. This plugin would take in parameters from the PH model simulation and apply these on the different links of the model. This implementation has the 3 advantages over the 20-sim conversion option.
The first advantage, is that implicit calculation methods can be used.
The second advantage is, that the API plugin can easily be extended or simplified by adding or removing forces or other parameters that should be applied.
The third advantage is, that the plugin does not have to be completely remade for every model.
If it turns out this implementation can not be realized. It is recommended to try the conversion of a 20-sim model to C++ code. The code should then be edited to have all the requirements of a Gazebo plugin and the forces can then be applied as discussed in section 3.4.3. The method of how this plugin is made should be clearly documented in a step by step tutorial so the process can easily be replicated for other models.

A Appendix 1

A.1 LiftDragplugin code

```
/*
 * Copyright (C) 2014 Open Source Robotics Foundation
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
*/

#include <algorithm>
#include <functional>
#include <string>

#include <ignition/common/Profiler.hh>
#include <ignition/math/Pose3.hh>

#include "gazebo/common/Assert.hh"
#include "gazebo/physics/physics.hh"
#include "gazebo/sensors/SensorManager.hh"
#include "gazebo/transport/transport.hh"
#include "plugins/LiftDragPlugin.hh"

using namespace gazebo;

GZ_REGISTER_MODEL_PLUGIN(LiftDragPlugin)

////////////////////////////////////////////////
```

Set some parameters that might not be declared

```
////////////////////////////////////////////////
```

```
void LiftDragPlugin::Load(physics::ModelPtr _model,
                          sdf::ElementPtr _sdf)
{
```

Load declared parameters and variables from Gazebo

```
////////////////////////////////////////////////
void LiftDragPlugin::OnUpdate()
{
  GZ_ASSERT(this->link, "Link was NULL");
  // get linear velocity at cp in inertial frame
  ignition::math::Vector3d vel = this->link->WorldLinearVel(this->cp);
  ignition::math::Vector3d velI = vel;
  velI.Normalize();

  // smoothing
  // double e = 0.8;
  // this->velSmooth = e*vel + (1.0 - e)*velSmooth;
  // vel = this->velSmooth;

  if (vel.Length() <= 0.01)
    return;

  IGN_PROFILE("LiftDragPlugin::OnUpdate");
  IGN_PROFILE_BEGIN(std::string(this->link->GetName()).c_str());

  // pose of body
```

```
184   // pose of body
185   ignition::math::Pose3d pose = this->link->WorldPose();
186
187   // rotate forward and upward vectors into inertial frame
188   ignition::math::Vector3d forwardI = pose.Rot().RotateVector(this->forward);
189
190   ignition::math::Vector3d upwardI;
191   if (this->radialSymmetry)
192   {
193     // use inflow velocity to determine upward direction
194     // which is the component of inflow perpendicular to forward direction.
195     ignition::math::Vector3d tmp = forwardI.Cross(velI);
196     upwardI = forwardI.Cross(tmp).Normalize();
197   }
198   else
199   {
200     upwardI = pose.Rot().RotateVector(this->upward);
201   }
202
203   // spanwiseI: a vector normal to lift-drag-plane described in inertial frame
204   ignition::math::Vector3d spanwiseI = forwardI.Cross(upwardI).Normalize();
205
206   const double minRatio = -1.0;
207   const double maxRatio = 1.0;
208   // check sweep (angle between velI and lift-drag-plane)
209   double sinSweepAngle = ignition::math::clamp(
210       spanwiseI.Dot(velI), minRatio, maxRatio);
211
212   // get cos from trig identity
213   double cosSweepAngle = 1.0 - sinSweepAngle * sinSweepAngle;
214   this->sweep = asin(sinSweepAngle);
215
216   // truncate sweep to within +/-90 deg
217   while (fabs(this->sweep) > 0.5 * M_PI)
218     this->sweep = this->sweep > 0 ? this->sweep - M_PI
219                                   : this->sweep + M_PI;
220
221   // angle of attack is the angle between
222   // velI projected into lift-drag plane
223   //  and
224   // forward vector
225   //
226   // projected = spanwiseI Xcross ( vector Xcross spanwiseI)
227   //
228   // so,
229   // removing spanwise velocity from vel
230   ignition::math::Vector3d velInLDPlane = vel - vel.Dot(spanwiseI)*velI;
231
232   // get direction of drag
233   ignition::math::Vector3d dragDirection = -velInLDPlane;
234   dragDirection.Normalize();

235   // get direction of lift
236   ignition::math::Vector3d liftI = spanwiseI.Cross(velInLDPlane);
237   liftI.Normalize();
238
239   // get direction of moment
240   ignition::math::Vector3d momentDirection = spanwiseI;
241
242   // compute angle between upwardI and liftI
243   // in general, given vectors a and b:
244   //   cos(theta) = a.Dot(b)/(a.Length()*b.Lenghth())
245   // given upwardI and liftI are both unit vectors, we can drop the denominator
246   //   cos(theta) = a.Dot(b)
247   double cosAlpha =
248     ignition::math::clamp(liftI.Dot(upwardI), minRatio, maxRatio);
249
250   // Is alpha positive or negative? Test:
251   // forwardI points toward zero alpha
252   // if forwardI is in the same direction as lift, alpha is positive.
253   // liftI is in the same direction as forwardI?
254   if (liftI.Dot(forwardI) >= 0.0)
255     this->alpha = this->alpha0 + acos(cosAlpha);
256   else
257     this->alpha = this->alpha0 - acos(cosAlpha);
```

```
257     this->alpha = this->alpha0 - acos(cosAlpha);
258
259   // normalize to within +/-90 deg
260   while (fabs(this->alpha) > 0.5 * M_PI)
261     this->alpha = this->alpha > 0 ? this->alpha - M_PI
262                                   : this->alpha + M_PI;
263
264   // compute dynamic pressure
265   double speedInLDPlane = velInLDPlane.Length();
266   double q = 0.5 * this->rho * speedInLDPlane * speedInLDPlane;
267
268   // compute cl at cp, check for stall, correct for sweep
269   double cl;
270   if (this->alpha > this->alphaStall)
271   {
272     cl = (this->cla * this->alphaStall +
273           this->claStall * (this->alpha - this->alphaStall))
274          * cosSweepAngle;
275     // make sure cl is still great than 0
276     cl = std::max(0.0, cl);
277   }
278   else if (this->alpha < -this->alphaStall)
279   {
280     cl = (-this->cla * this->alphaStall +
281           this->claStall * (this->alpha + this->alphaStall))
282          * cosSweepAngle;
283     // make sure cl is still less than 0
284     cl = std::min(0.0, cl);
285   }
286   else
287     cl = this->cla * this->alpha * cosSweepAngle;
288
289   // modify cl per control joint value
290   if (this->controlJoint)
291   {
292     double controlAngle = this->controlJoint->Position(0);
293     cl = cl + this->controlJointRadToCL * controlAngle;
294     /// \TODO: also change cm and cd
295   }
296
297   // compute lift force at cp
298   ignition::math::Vector3d lift = cl * q * this->area * liftI;
299
300   // compute cd at cp, check for stall, correct for sweep
301   double cd;
302   if (this->alpha > this->alphaStall)
303   {
304     cd = (this->cda * this->alphaStall +
305           this->cdaStall * (this->alpha - this->alphaStall))
306          * cosSweepAngle;
307   }
308   else if (this->alpha < -this->alphaStall)
309   {
310     cd = (-this->cda * this->alphaStall +
311           this->cdaStall * (this->alpha + this->alphaStall))
312          * cosSweepAngle;
313   }
314   else
315     cd = (this->cda * this->alpha) * cosSweepAngle;
316
317   // make sure drag is positive
318   cd = fabs(cd);
319
320   // drag at cp
321   ignition::math::Vector3d drag = cd * q * this->area * dragDirection;
322
323   // compute cm at cp, check for stall, correct for sweep
324   double cm;
325   if (this->alpha > this->alphaStall)
326   {
327     cm = (this->cma * this->alphaStall +
328           this->cmaStall * (this->alpha - this->alphaStall))
329          * cosSweepAngle;
330     // make sure cm is still great than 0
    cm = std::max(0.0, cm);
```

```
330       // make sure cm is still great than 0
331       cm = std::max(0.0, cm);
332     }
333     else if (this->alpha < -this->alphaStall)
334     {
335       cm = (-this->cma * this->alphaStall +
336             this->cmaStall * (this->alpha + this->alphaStall))
337           * cosSweepAngle;
338       // make sure cm is still less than 0
339       cm = std::min(0.0, cm);
340     }
341     else
342       cm = this->cma * this->alpha * cosSweepAngle;
343
344     /// \TODO: implement cm
345     /// for now, reset cm to zero, as cm needs testing
346     cm = 0.0;
347
348     // compute moment (torque) at cp
349     ignition::math::Vector3d moment = cm * q * this->area * momentDirection;
350
351     // moment arm from cg to cp in inertial plane
352     ignition::math::Vector3d momentArm = pose.Rot().RotateVector(
353       this->cp - this->link->GetInertial()->CoG());
354     // gzerr << this->cp << " : " << this->link->GetInertial()->GetCoG() << "\n";
355
356     // force and torque about cg in inertial frame
357     ignition::math::Vector3d force = lift + drag;
358     // + moment.Cross(momentArm);
359
    ignition::math::Vector3d torque = moment;
    // - lift.Cross(momentArm) - drag.Cross(momentArm);

    // debug
```

Some lines that print variables to Gazebo for debugging purposes

```
395     // Correct for nan or inf
396     force.Correct();
397     this->cp.Correct();
398     torque.Correct();
399
400     // apply forces at cg (with torques for position shift)
401     this->link->AddForceAtRelativePosition(force, this->cp);
402     this->link->AddTorque(torque);
403     IGN_PROFILE_END();
  }
```

A.2 LiftDragplugin code analysis overview

```
Liftdrag.cc:
First lines are including the necessary files.
38-58   Certain values are declared
67-161  Is used to load in values set in the world and sdf files.
164-404 Is the update function.
        This is where the forces are calculated and applied.

Further elaboration of the update function:
168-170 Get the velocity of the link from gazebo and
        store its normalized velocity.
177     Checks if the velocity is lower than 0.01 if this is the case,
        it returns that no forces are coming from lift drag.
184     Pulls the pose(orientation) of the lift drag body.
187-203 Determines the forward, upward and sweep direction.
        These are the main axis of interest for the lift body.
        The spanwise direction which is the orthonormal vector
        to the forward-upward plane.
205-217 Calculates the sweep angle.
229     Removes the spanwise velocity from the pulled velocity.
232-233 Gets normalized direction of drag.
236-247 Gets normalized direction of lift.
240     Gets direction of moment.
247-262 Compute angle alpha (angle between upward and lift direction):
265-266 Here the dynamic pressure "q" is calculated.
        This is done by taking:
        0.5*rho(fluid density)*speedLDplane^2.
268-295 Computing coefficient of lift at the center of pressure.
298     Computing the lift force at the center of pressure.
300-315 Compute coefficient and check if its positive.
318     Compute drag at cp: cd * q * this->area * dragDirection;
324-340 Check for stall and correct for sweep.
342-346 Set cm to zero since moment calculation is not working properly yet.
348     Compute moment (torque) at cp: cm * q * this->area * momentDirection;
352-353 Calculate moment arm from cg to cp in inertial plane. :
356-360 Here the final force and torque are calculated.:
// force and torque about cg in inertial frame
ignition::math::Vector3d force = lift + drag;
// + moment.Cross(momentArm);
ignition::math::Vector3d torque = moment;
// - lift.Cross(momentArm) - drag.Cross(momentArm);
363-393 Is for debugging
396-398 Is to correct for infinite or NaN forces and center of pressure.
400-402 Apply the calculated force and torque at cg.:
// apply forces at cg (with torques for position shift)
this->link->AddForceAtRelativePosition(force, this->cp);
this->link->AddTorque(torque);
403     The ignite profile is ended.
```

B Appendix 2

B.1 Cessna model.config code

```
<?xml version="1.0"?>

<model>
  <name>Cessna C-172</name>
  <version>1.0</version>
  <sdf version="1.5">model.sdf</sdf>

  <author>
    <name>Carlos Agüero</name>
    <email>caguero@osrfoundation.org</email>
  </author>

  <description>
    Cessna C-172.

    An example world file that uses the cessna and ships with Gazebo is:
    $ gazebo worlds/cessna.world

    The meshes have been imported and adapted from:
    https://3dwarehouse.sketchup.com/model.html?id=5178440145d1d2c2d77055f14f845

    All the internal specifications have been extracted and adapted from:
    https://github.com/tridge/jsbsim/blob/master/aircraft/c172p/c172p.xml
  </description>
</model>
```

B.2 Cessna model.sdf code snippets

XML and SDF declaration + Body Link

```xml
<?xml version="1.0" ?>
<sdf version="1.5">
  <model name="cessna_c172">
    <pose>0 0 0.495 0 0 0</pose>
    <!-- Body of the plane -->
    <link name="body">
      <inertial>
        <mass>680.389</mass>
        <inertia>
          <ixx>1285.315427874</ixx>
          <ixy>0.0</ixy>
          <iyy>1824.930976707</iyy>
          <ixz>0.0</ixz>
          <iyz>0.0</iyz>
          <izz>2666.893931043</izz>
        </inertia>
        <pose>-0.0414 0 0.9271 0 0 0</pose>
      </inertial>

      <collision name="collision">
        <geometry>
          <mesh>
            <uri>model://cessna/meshes/body.dae</uri>
          </mesh>
        </geometry>
      </collision>

      <visual name="visual">
        <geometry>
          <mesh>
            <uri>model://cessna/meshes/body.dae</uri>
          </mesh>
        </geometry>
      </visual>
```

Left Aileron Link

```xml
    <link name="left_aileron">
      <inertial>
        <mass>2</mass>
        <inertia>
          <ixx>0.8434</ixx>
          <ixy>0.0</ixy>
          <iyy>0.0119</iyy>
          <ixz>0.0</ixz>
          <iyz>0.0</iyz>
          <izz>0.855</izz>
        </inertia>
        <pose>-1.65 3.7 1.5 0.05 0 -0.12</pose>
      </inertial>

      <collision name="collision">
        <geometry>
          <mesh>
            <uri>model://cessna/meshes/left_aileron.dae</uri>
          </mesh>
        </geometry>
      </collision>

      <visual name="visual">
        <geometry>
          <mesh>
            <uri>model://cessna/meshes/left_aileron.dae</uri>
          </mesh>
        </geometry>
      </visual>
    </link>
```

Left Aileron Joint

```xml
    <!-- Joint to move the left aileron -->
    <joint name='left_aileron_joint' type='revolute'>
      <parent>body</parent>
      <child>left_aileron</child>
      <pose>-1.45 3.7 1.5 0.05 0 -0.12</pose>
      <axis>
        <xyz>0 1 0</xyz>
        <limit>
          <!-- -30/+30 deg. -->
          <lower>-0.53</lower>
          <upper>0.53</upper>
          <effort>-1</effort>
          <velocity>-1</velocity>
        </limit>
        <dynamics>
          <damping>1.000</damping>
        </dynamics>
      </axis>
      <physics>
        <ode>
          <implicit_spring_damper>1</implicit_spring_damper
        </ode>
      </physics>
    </joint>
```

Right Flap Joint

```xml
    <!-- Joint to move the right flap -->
    <joint name='right_flap_joint' type='revolute'>
      <parent>body</parent>
      <child>right_flap</child>
      <pose>-1.6 -1.55 1.43 -0.02 0 0</pose>
      <axis>
        <xyz>0 1 0</xyz>
        <limit>
          <!-- -30/+30 deg. -->
          <lower>-0.53</lower>
          <upper>0.53</upper>
          <effort>-1</effort>
          <velocity>-1</velocity>
        </limit>
        <dynamics>
          <damping>1.000</damping>
        </dynamics>
      </axis>
      <physics>
        <ode>
          <implicit_spring_damper>1</implicit_spring_damper>
        </ode>
      </physics>
    </joint>
```

C Appendix 3

C.1 Cessna_demo.world Code

update commit
Louis Nelissen authored 1 month ago

f593af73

cessna_demo.world 7 KB

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <gui>
      <!-- A plugin for controlling the Cessna with the keyboard -->
      <plugin name="cessna_keyboard" filename="libCessnaGUIPlugin.so"/>

      <camera name="user_camera">
        <pose>-16 0 4 0 0 0</pose>
      </camera>
    </gui>

    <include>
      <uri>model://sun</uri>
    </include>

    <model name="ground_plane">
      <static>true</static>
      <link name="link">
        <collision name="collision">
          <geometry>
            <plane>
              <normal>0 0 1</normal>
              <size>5000 5000</size>
            </plane>
          </geometry>
          <surface>
            <friction>
              <ode>
                <mu>1</mu>
                <mu2>1</mu2>
              </ode>
            </friction>
          </surface>
        </collision>
        <visual name="runway">
          <pose>700 0 0.1 0 0 0</pose>
          <cast_shadows>false</cast_shadows>
          <geometry>
            <plane>
              <normal>0 0 1</normal>
              <size>1829 45</size>
            </plane>
          </geometry>
          <material>
            <script>
              <uri>file://media/materials/scripts/gazebo.material</uri>
              <name>Gazebo/Runway</name>
            </script>
          </material>
        </visual>

        <visual name="grass">
          <pose>0 0 -0.1 0 0 0</pose>
          <cast_shadows>false</cast_shadows>
          <geometry>
            <plane>
              <normal>0 0 1</normal>
              <size>5000 5000</size>
            </plane>
          </geometry>
          <material>
            <script>
              <uri>file://media/materials/scripts/gazebo.material</uri>
              <name>Gazebo/Grass</name>
            </script>
          </material>
        </visual>

      </link>
    </model>

    <model name="cessna_c172">
```

```
    <include>
      <uri>model://cessna</uri>
    </include>

    <link name="blade_1_visual">
      <pose>1.79 0 1.350 0 0 0</pose>
      <gravity>0</gravity>
    </link>

    <link name="wing_body_debug_visuals">
      <pose>0 0 0.495 0 0 0</pose>
      <gravity>0</gravity>
    </link>

    <!-- Plugins for controlling the thrust and control surfaces -->
    <plugin name="cessna_control" filename="libCessnaPlugin.so">
      <propeller>cessna_c172::propeller_joint</propeller>
      <propeller_max_rpm>2500</propeller_max_rpm>
      <left_aileron>cessna_c172::left_aileron_joint</left_aileron>
      <left_flap>cessna_c172::left_flap_joint</left_flap>
      <right_aileron>cessna_c172::right_aileron_joint</right_aileron>
      <right_flap>cessna_c172::right_flap_joint</right_flap>
      <elevators>cessna_c172::elevators_joint</elevators>
      <rudder>cessna_c172::rudder_joint</rudder>
      <propeller_p_gain>10000</propeller_p_gain>
      <propeller_i_gain>0</propeller_i_gain>
      <propeller_d_gain>0</propeller_d_gain>
      <surfaces_p_gain>2000</surfaces_p_gain>
      <surfaces_i_gain>0</surfaces_i_gain>
      <surfaces_d_gain>0</surfaces_d_gain>
    </plugin>
    <plugin name="propeller_top_blade" filename="libLiftDragPlugin.so">
      <a0>0.4</a0>
      <cla>4.752798721</cla>
      <cda>0.6417112299</cda>
      <cma>0</cma>
      <alpha_stall>1.5</alpha_stall>
      <cla_stall>-3.85</cla_stall>
      <cda_stall>-0.9233984055</cda_stall>
      <cma_stall>0</cma_stall>
      <cp>-0.37 0 0.77</cp>
      <area>0.1</area>
      <air_density>1.2041</air_density>
      <forward>0 -1 0</forward>
      <upward>1 0 0</upward>
      <link_name>cessna_c172::propeller</link_name>
    </plugin>
    <plugin name="propeller_bottom_blade" filename="libLiftDragPlugin.so">
      <a0>0.4</a0>
      <cla>4.752798721</cla>
      <cda>0.6417112299</cda>
      <cma>0</cma>
      <alpha_stall>1.5</alpha_stall>
      <cla_stall>-3.85</cla_stall>
      <cda_stall>-0.9233984055</cda_stall>
      <cma_stall>0</cma_stall>
      <cp>-0.37 0 -0.77</cp>
      <area>0.1</area>
      <air_density>1.2041</air_density>
      <forward>0 1 0</forward>
      <upward>1 0 0</upward>
      <link_name>cessna_c172::propeller</link_name>
    </plugin>

    <plugin name="left_wing" filename="libLiftDragPlugin.so">
      <a0>0.05984281113</a0>
      <cla>4.752798721</cla>
      <cda>0.6417112299</cda>
      <cma>-1.8</cma>
      <alpha_stall>0.3391428111</alpha_stall>
      <cla_stall>-3.85</cla_stall>
      <cda_stall>-0.9233984055</cda_stall>
      <cma_stall>0</cma_stall>
      <cp>-1 2.205 1.5</cp>
      <area>8.08255</area>
      <air_density>1.2041</air_density>
      <forward>1 0 0</forward>
      <upward>0 0 1</upward>
      <link_name>cessna_c172::body</link_name>
      <control_joint_name>cessna_c172::left_aileron_joint</control_joint_name>
      <control_joint_rad_to_cl>-2.0</control_joint_rad_to_cl>
    </plugin>
```

```
      <plugin name="right_wing" filename="libLiftDragPlugin.so">
        <a0>0.05984281113</a0>
        <cla>4.752798721</cla>
        <cda>0.6417112299</cda>
        <cma>-1.8</cma>
        <alpha_stall>0.3391428111</alpha_stall>
        <cla_stall>-3.85</cla_stall>
        <cda_stall>-0.9233984055</cda_stall>
        <cma_stall>0</cma_stall>
        <cp>-1 -2.205 1.5</cp>
        <area>8.08255</area>
        <air_density>1.2041</air_density>
        <forward>1 0 0</forward>
        <upward>0 0 1</upward>
        <link_name>cessna_c172::body</link_name>
        <control_joint_name>
          cessna_c172::right_aileron_joint
        </control_joint_name>
        <control_joint_rad_to_cl>-2.0</control_joint_rad_to_cl>
      </plugin>

      <plugin name="elevator" filename="libLiftDragPlugin.so">
        <a0>-0.2</a0>
        <cla>4.752798721</cla>
        <cda>0.6417112299</cda>
        <cma>-1.8</cma>
        <alpha_stall>0.3391428111</alpha_stall>
        <cla_stall>-3.85</cla_stall>
        <cda_stall>-0.9233984055</cda_stall>
        <cma_stall>0</cma_stall>
        <cp>-5.45 0 0.55</cp>
        <area>2.03458</area>
        <air_density>1.2041</air_density>
        <forward>1 0 0</forward>
        <upward>0 0 1</upward>
        <link_name>cessna_c172::body</link_name>
        <control_joint_name>cessna_c172::elevators_joint</control_joint_name>
        <control_joint_rad_to_cl>-4.0</control_joint_rad_to_cl>
      </plugin>

      <plugin name="rudder" filename="libLiftDragPlugin.so">
        <a0>0</a0>
        <cla>4.752798721</cla>
        <cda>0.6417112299</cda>
        <cma>-1.8</cma>
        <alpha_stall>0.3391428111</alpha_stall>
        <cla_stall>-3.85</cla_stall>
        <cda_stall>-0.9233984055</cda_stall>
        <cma_stall>0</cma_stall>
        <cp>-6 0 1.55</cp>
        <area>1.5329</area>
        <air_density>1.2041</air_density>
        <forward>1 0 0</forward>
        <upward>0 1 0</upward>
        <link_name>cessna_c172::body</link_name>
        <control_joint_name>cessna_c172::rudder_joint</control_joint_name>
        <control_joint_rad_to_cl>4.0</control_joint_rad_to_cl>
      </plugin>
    </model>
  </world>
</sdf>
```

Bibliography

[1] Robotics simulator, Oct 2021. URL: https://en.wikipedia.org/wiki/Robotics_simulator.

[2] The Project Portwings. The portwings project – understanding the secrets of flapping flight. URL: http://www.portwings.eu/.

[3] Ramy Rashad, Federico Califano, Arjan J van der Schaft, and Stefano Stramigioli. Twenty years of distributed port-hamiltonian systems: a literature review. *IMA Journal of Mathematical Control and Information*, 37(4):1400–1422, 2020.

[4] Robotics Open. Gazebo, 2019. URL: http://gazebosim.org/.

[5] Sangwoo Moon, John J Bird, Steve Borenstein, and Eric W Frew. A gazebo/ros-based communication-realistic simulator for networked suas. In *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1819–1827. IEEE, 2020.

[6] Khoa Dang Nguyen and Trong-Thang Nguyen. Vision-based software-in-the-loop-simulation for unmanned aerial vehicles using gazebo and px4 open source. In *2019 International Conference on System Science and Engineering (ICSSE)*, pages 429–432. IEEE, 2019.

[7] Khoa Dang Nguyen and Cheolkeun Ha. Development of hardware-in-the-loop simulation based on gazebo and pixhawk for unmanned aerial vehicles. *International Journal of Aeronautical and Space Sciences*, 19(1):238–249, 2018.

[8] Sarvesh Sonkar, Prashant Kumar, Deepu Philip, and AK Ghosh. Low-cost smart surveillance and reconnaissance using vtol fixed wing uav. In *2020 IEEE Aerospace Conference*, pages 1–7. IEEE, 2020.

[9] Gary Ellingson and Tim McLain. Rosplane: Fixed-wing autopilot for education and research. In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1503–1507. IEEE, 2017.

[10] Aerospaceweb. Aerospaceweb.org | reference for aviation, space, design, and engineering. URL: http://www.aerospaceweb.org/.

[11] Open Robotics. Gazebo components. URL: http://gazebosim.org/tutorials?tut=components.

[12] Open Robotics. Plugins 101. URL: http://gazebosim.org/tutorials?tut=plugins_hello_world&cat=write_plugin.

[13] Open Robotics. Topics subscription. URL: https://gazebosim.org/tutorials?tut=topics_subscribed&cat=transport.

[14] Open Robotics. urdf/xml/joint - ros wiki. URL: http://wiki.ros.org/urdf/XML/joint.

[15] Gazebo. Gazebo : Tutorial : Aerodynamics. URL: https://gazebosim.org/tutorials?tut=aerodynamics&cat=physics.

[16] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. Springer, 2 edition, 2016.

[17] Eric Luper. Component-based modelling and simulation of a kuka lwr+ 4 robotic arm. B.S. thesis, University of Twente, 2017.