# Project Report - Group 7

**TDDE19 Advanced Project Course - AI and Machine Learning**

Hjalmarsson, Adam
`adahj913`

Ericsson, Leon
`leoer843`

Kouznetsov, Daniel
`danko376`

Wahlgren, Alex
`alewa196`

Halvarsson, Erik
`eriha353`

April 9, 2023

## 1 Introduction

Autonomous flight consists of several sub-problems large enough to warrant their own report. This project aims to tackle the problem from scratch, laying a solid, modular foundation to be iteratively improved for time to come. The predetermined task consists of a flight between two airports with possible intercepting air traffic control (ATC) commands. This mission includes actions like taxing, departing, increase/decrease height, landing, etc.

A mission starts with a underlying global plan supplied to the pilot before takeoff. During flight, ATC commands are intercepted arbitrarily and its up to the system to either divert from or adjust the flight plan accordingly. ATC commands follow a strict structure in natural language meaning an interpreter is required to both process natural language and convert it into an instruction for the system. Once a clear task is determined, a motion planner generates a trajectory that adheres to the aircraft's motion constraints. Motion primitives are a popular way to discretize the state space and find a suitable solution. Finally, the trajectory needs to be executed and this execution is to be visualized. An overview of this process can be seen in fig. 1 where the dotted box defines the project scope.

To aid in the processing of ATC commands an existing NLP model was supplied, the task that remains is to interpret the output of the NLP model into logical missions. For the motion planner, motion primitives should be generated offline and preferably there should be a distinction between ground primitives and air primitives. Given generated motion primitives a motion planner will generate a trajectory that is fed to the controller. The controller will combine the trajectory with the current state to determine an error to regulate by resulting in a control signal. The aforementioned segments should be implemented as ROS[9] nodes with visualization of the aircraft using for example Gazebo[5] or RQT[14].

The solution with the small iterative process with a set of subprocesses on the way seems like a good starting point. It both makes it easier to iteratively improve the whole process as well as upgrading the subprocesses, i.e. planning, executing, etc. It also makes the process modular and gives space to try other methods for the subprocesses without the need to rewrite everything.

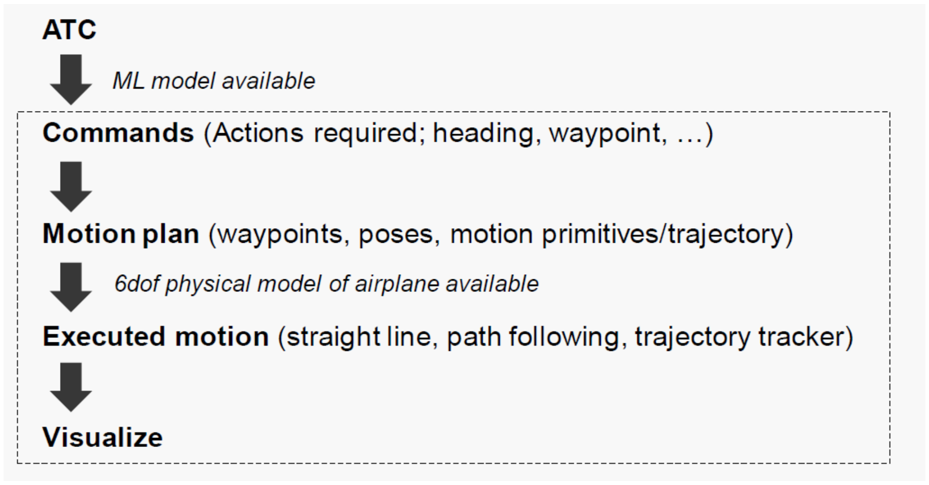

ATC
ML model available
Commands (Actions required; heading, waypoint, ...)
Motion plan (waypoints, poses, motion primitives/trajectory)
6dof physical model of airplane available
Executed motion (straight line, path following, trajectory tracker)
Visualize

Figure 1: An overview of the procedure used to solve the described problem.

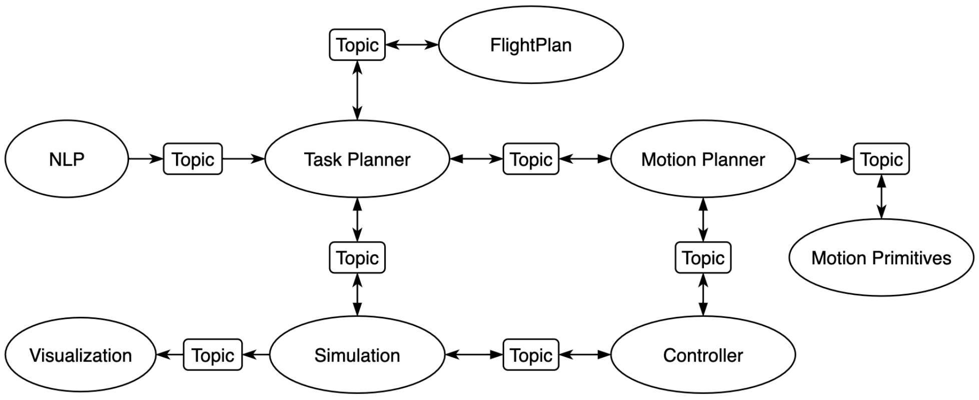


Figure 2: ROS Graph

## 2 Methodology

The project was built as a complete ROS 2 application on top of the latest release, *Humble Hawksbill*[12]. This approach enabled well constructed modularity, ease of distribution and the ability to use state-of-the art open source projects to aid in functionality such as planning, control, simulation and modelling. First we'll introduce the overall project structure defined as a ROS node graph, then we'll continue by describing how each individual subtask was tackled.

### 2.1 ROS 2

Robot Operating System 2 (ROS 2) is a open source robot software library that helps build and distribute robot applications. At the core of ROS 2 lies the *ROS 2 Graph* which is a network of ROS 2 elements processing data together at one time. It encompasses all the project executables and the connections between them if you were to map them all out and visualize them[12].

The principal element in a ROS 2 Graph is a *node* which is responsible for a single, module purpose (e.g. one node for controlling wheel motors). Each node can send and receive data to other nodes via topics, services or actions. Topics are a vital element of the ROS graph, they act as a bus for nodes to exchange messages in a publish-subscribe fashion. Each node may publish and subscribe to any number of topics simultaneously, topics don't have to be point-to-point communication; they can server one-to-many, many-to-one or many-to-many relations as well. Services support a request-response pattern, versus topics' pub-sub model. While topics allow nodes to subscribe to data streams and get continual updates, service servers only provide data when they are specifically called by a service client. Services only support a one-to-many relation with one node acting as the service server. Finally, actions are intended for long running tasks as they consist of three parts: a goal, feedback and a result[11]. Together with nodes, these elements work in concert to comprise a full robotic system, such as an autonomous aircraft.

The ROS 2 graph defining this project can be seen in fig. 2.

### 2.2 Docker

A container is a type of software which lets the user package code together with all dependencies to be able to run the application in different environments independent of what operating system and other configurations the system runs on. A container in Docker is lightweight and contains everything that is needed to run an application ranging from libraries, system tools and settings[4].

Docker containers are built on something called Docker images. This is a template which holds all the specifications for the container such as libraries and dependencies[2]. An image is created using a file called Dockerfile, the structure of this file essentially a list of all these libraries and dependencies and lets the Docker container know what to install when creating the container.

## 2.3 System Overview

The ROS 2 Graph 2 provides a high level overview of the systems function. Before diving into the details of each ROS node, let us first walk through what the system wants to accomplish, in broader strokes.

A flight always begins with a flight plan, of some type, read and initialized from the *Flight Plan* node. Following this, the *Task Planner*, acting as the systems central control unit, initializes the flight by forwarding a few waypoints along the flight plan towards the motion planner. The *Motion Planner*, given waypoints and the aircraft status, generates a feasible motion trajectory. A trajectory consists of intermediary nodes coupled with time-stamps used for speed regulation by the controller. The trajectory is sent to the *Controller*, which reads the aircraft's current state and combines this with the current expected state in the trajectory to generate a control signal. Finally, the control signal is sent to the *Simulation* node, which is a representation of the true aircraft model.

This entire process is sequential and primarily predetermined, however spontaneous ATC commands may occur at any time during the flight. The *NLP* node reads and forwards any ATC commands received during the flight to the *Task Planner* and these commands always take precedent over the flight plan. This infers the current plan in the motion planner and controller may need to be recalculated.

## 2.4 Task Planner

A central part comprising the system is the *Task Planner*. The main purpose of the Task Planner is to keep track of the next desired state of the aircraft. If the task planner only sends one waypoint at a time, a delay will be introduced when this waypoint is reached due to computation time for the next waypoint. To avoid this the *task planner* serves the *motion planner* with multiple goal states up to a horizon at a time so the program does not have to wait every time a goal state/waypoint is reached. At the start the current state and multiple consecutive goal states is then published so that it can be received by the Motion Planner to determine the next course of action. When one goal state is reached the *task planner* is noticed and can publish a new goal state while the *controller* still steers the aircraft towards the next goal. The published states consists of a position and a time, i.e. (x, y, z, t).

Another key aspect of the Task Planner is to keep track on which type of instructions that the system currently follows. There are two different instruction types which originates either from,

- The original flight plan (Flight Plan node)
- ATC commands given in real-time (NLP node)

The flight plan acts as the baseline for the system and consists of waypoints. When the system starts the *flight plan* node sends the basic flight plan to the *task planner.* which means this communication just occur once.

ATC commands however, are received in real-time and have precedence over the flight plan. This entails that whenever a new ATC session is initiated, the *task planner* will start to receive instructions from the *NLP* node and switch over to ATC mode. After receiving these instructions, a goal state is calculated based on the current state and then published to the Motion Planner along with the current state. When instructions are received that the ATC session has ended, the *task planner* will revert back to the original flight plan, and the cycle continues.

## 2.5 Flight Plan

The *Flight Plan* node is responsible for translating a pre-planned flight plan, given as a string of codewords, into individual states that can be understood by the rest of the system. This is done by parsing each codeword in the input string and looking up the corresponding name in a database. The database, which is currently implemented as a Python class, contains the necessary information for each waypoint or route, including the associated states. The node then uses this information to generate the appropriate states and send them to the next node in the system which is the *Task Planner*.

While the current implementation of the database is simple and mostly serves as a demonstration, it can be expanded and improved upon in the future to work better for a real-world system.

## 2.6 NLP & ATC Planner

NLP stands for Natural Language Processing and is a research field where it is possible to assert or create meaning to natural language by applying various computational techniques. It is an attempt to bring structure to otherwise unstructured input by incorporating linguistic knowledge. Techniques in NLP that are based on linguistic knowledge is for example lemmatization, where each word is transformed into it's root form, or part-of-speech tagging, where for example the word 'car' gets tagged with 'NOUN'. After pre-processing and breaking down the linguistic input, other methods such as deep feed-forward networks (DNNs) or Hidden Markov Models (HMMs) can be applied in order to train a model to recognize different features of the input.

The NLP node is responsible for parsing spoken ATC commands into viable text commands that can be interpreted by the ATC Planner and acted upon. This is done through speech recognition, which is a sub-field of NLP. Currently, the ATC Planner receives an ATC message and parses it further by creating viable states with coordinates from the given waypoint. After parsed into one or several states, a path is created and published to the Task Planner which switches to ATC mode, given the mode is not already activated. Depending on the specified ATC action, the published path will look different. For example, a *wait*-action should appropriately send a path with the same state, where as a *direct*-action only needs to send one state representing the waypoint to redirect to. This behaviour is implemented in separate ATCTask classes.

## 2.7 Motion Planner

Motion plans are necessary to be able to move objects with dynamical characteristics from one state to another wanted state in a real world environment. A useful method for this purpose is the lattice-based motion planner described in [7]. The lattice based planner takes the physical constraints of the moving object into consideration with the help of the motion primitives, see section 2.8, and generates vertices in a state lattice. The pre-computed motion primitives will then be the edges between those vertices, i.e. the vertices are determined by the motion primitives. When reasonable motion primitives have been generated offline and the start state and goal state are known, the only thing left is to do a graph search to find a suitable motion between the two states. There are multiple options of search methods that have different guarantees on response time and optimality.

The *Motion Planner* is responsible of planning the tasks coming in from the *Task Planner*. The task consists of a current (start) state and a goal state that includes the following information of the aircraft (x[m], y[m], z[m]), i.e. a position. The goal of the motion planner is to find a good and safe set of motions from the start state to the goal state and forward this plan to the *Controller* node. The motion planner also adds a time for each state so the controller have a reference when regulating the aircraft. The time is set based on the aircraft's defined max speed divided by two. This creates room to both increase and decrease the speed by the controller without a risk getting close to the edges.

## 2.8 Motion Primitives

Motion primitives are a integral part of generating motion plans that are true to the motion dynamics of a plane. From a given state $s$ motion primitives generate a state graph superimposed with kinematic constraints meaning a plane can smoothly transition from $s$ to $s'$. Motion primitives allow us to encode the kinematic constraints of a plane into any environment that the plane is planning in. They also allow for different costs associated with each motion - for example, if we want to avoid hard turns, we might assign a high cost to that particular motion primitive. Motion primitives are pre-computed thus enabling a online lattice based motion planner that uses motion primitives[3].

The complexity of motion primitives can vary heavily but as the initial goal was a simple system, the motion primitives should reflect that. Kinematic motion models for road vehicles are open source and available, therefor a natural approach is extend a motion primitive generator for cars into the third dimension by discretizing the yaw axis with 10-15 angles. In fig. 3 we see an example of how motion primitives set can look in the 3-dimensional case. Generally an aircraft has limited amount of motion changes and there are lesser obstacles constraint to adhere by, therefor a relatively long time step is employed and a sparse primitive space can be generated while still minimizing the risk of not finding a feasible solution. Ordinarily there is a diversity/efficiency trade-off where too sparse primitives restrict the search space while too dense primitives drastically inflate the search time[15].

The initial primitives for this project were minimal, and their complexity was heavily reduced due to time constraints. We built a specific ROS node to generate the primitives for ease of upgrade in the future, but as of right now the primitives generated are seen as a single movement from one node to the next. Looking forward, it would be ideal to discretize the movement between the nodes into more steps instead of just seeing a motion primitive as a complete move between two nodes as this heavily reduces the flexibility of the aircraft.

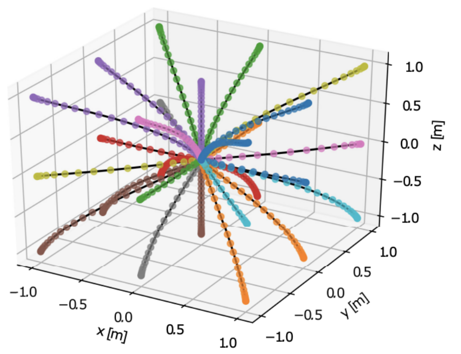


Figure 3: One example of a motion primitive set generated offline that can be used to solve motion planning problems online.

## 2.9 Controller

In general, a controller governs the application of state inputs to drive the system towards a desired state while minimizing a delay, overshoot, or steady-state error and ensuring a level of stability. To achieve this the controller subscribes to the current state $s$, supplied by *Simulation*, and the trajectory $p$, supplied by *Motion Planner*. The aim is to minimize the error $\epsilon = ||\hat{s} - s||$ by deriving $\hat{s}$ from the time-stamp in $s$ resulting in a closed-loop control system[8]. By linearly interpolating between the elements in the trajectory we can estimate $\hat{s}$ at every time point $t$ and given that $s$ contains a time stamp we can estimate a error $\epsilon$ for every given state update $s$.

The resulting control signal is given by a 3D velocity vector which is published to the *Simulation* node. One final verification is made to ensure the magnitude of the velocity vector does not exceed the systems physical limitations (e.g. $v_{max}$). It is of importance that the controller consistently generates control signals that follow a trajectory, to ensure smooth state-transitions. To this end the controller must maintain a non-empty trajectory list. To ensure this, *Controller* signals the *Motion Planner* each time a trajectory element is traversed and in turn the *Motion Planner* decides whether it needs to send subsequent elements.

## 2.10 Simulation

The representation of reality and dynamics of the physical system needs to be independent from the autonomous system that is planning and controlling it. This representation takes the form of the *Simulation* node with its two main objectives: simulate the moving aircraft and publish the aircraft's current state. This entails another responsibility, maintaining the physical model of the aircraft. Combining control signals received from the *Controller*, with the current state of the aircraft and its physical model this node propagates the aircraft through space over time. Ideally this update would occur constantly and therefore be a true representation of reality, in practice however this isn't possible instead the update occurs at a frequency of 1 kHz. The resulting error between the simulation and reality is conveniently named the *simulation error*.

As mentioned the node publishes the aircraft's current state, this occurs at a rate of 100Hz. Notably, each state publication contains a time-stamp enabling subscribers to match and/or disregard relevant information. Finally, *Visualizer* is tightly coupled with this node as its task is to visualize the result from the state update.

## 2.11 Visualization

In order to show the controller what the agent is doing, and to easily be able to tell whether it is performing as expected or not, we need to visualize the data from the simulation node. This is the task of the *Visualization* node, which performs this task by generating a plot of *OpenStreetMap*[10] data using *GeoPandas*[6], and sending the result to a visualizer tool like *RViz2*[13] or *Gazebo*[5]. The node draws each waypoint from the flight plan with lines connecting them, and continually updates a blip representation of the agents location on this map. When waypoints get removed or new waypoints get added through ATC injections, the visualizer updates the path representation on the map.

# 3 Results

This section presents the results of the project. In short the project resulted in having built the structure for a complex system with a good foundation for future development inside of rootless Docker container.

## 3.1 ROS2 Project Structure

The project was built with a vision of it having great modularity. By building the system using ROS2 and leveraging the use of ROS2 nodes the resulting modular architecture allows for individual components of the system to be easily configured, changed, and integrated without disrupting the overall operation of the system. This is important in the development of fully autonomous and complex systems such as this one since the ability to quickly and efficiently modify and update individual components is essential.

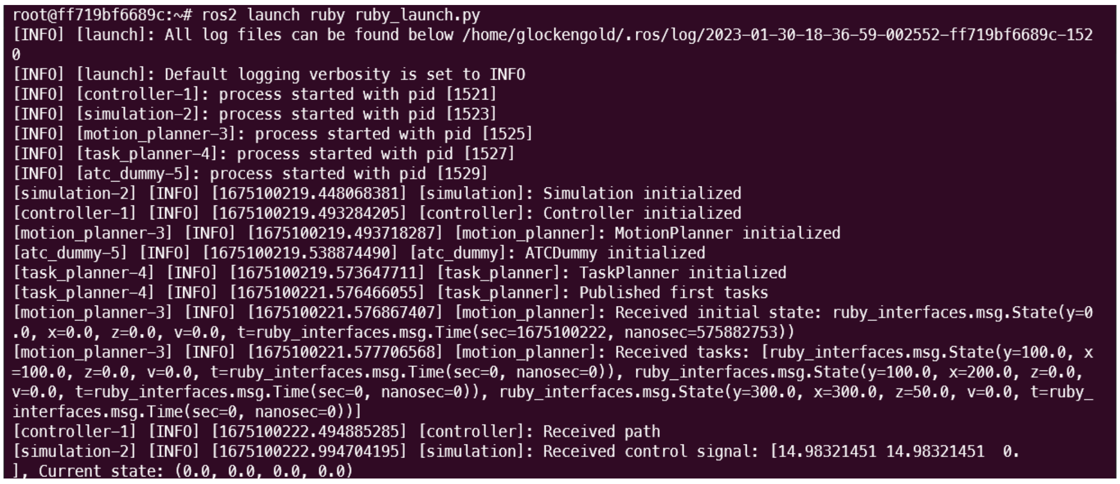
What resulted was a ROS2 project structure which is fully modular and built for future development in mind. All nodes are fully operable by themselves as long as the developer has the communication protocol in mind which is easy to look up. All I/O between nodes are defined using messages which can be find in the interfaces folder.

## 3.2 The Autonomous System

The modular approach to this automation problem resulted in a baseline system that is capable of following a pre-determined trajectory with dynamical characteristics. It is also able to switch from this pre-determined trajectory into following received ATC instructions at run-time. In order to achieve stability in the system and get smooth transitions between to states along the trajectory, the system takes advantage of a closed-loop control system which yields a control signal that is used to update the current aircraft state. The system also takes into account physical limitations, such as max velocity of the aircraft, to ensure that the state-update is accurate. Further, through the use of motion primitives, the system is also capable of adhering to environmental constraints which can be defined further depending on the level of abstraction.

### 3.2.1 A preview of the system

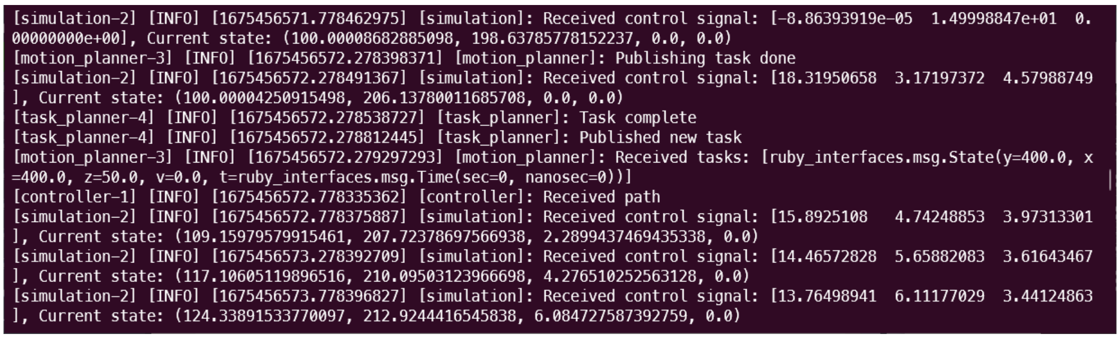
What follows is a short preview of the system as it currently stands. In figure 4 we see the start-up procedure of the system, with each ROS node initializing and sending their first updates along the connections shown in figure 2.



```
root@ff719bf6689c:~# ros2 launch ruby ruby_launch.py
[INFO] [launch]: All log files can be found below /home/glockengold/.ros/log/2023-01-30-18-36-59-002552-ff719bf6689c-152
0
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [controller-1]: process started with pid [1521]
[INFO] [simulation-2]: process started with pid [1523]
[INFO] [motion_planner-3]: process started with pid [1525]
[INFO] [task_planner-4]: process started with pid [1527]
[INFO] [atc_dummy-5]: process started with pid [1529]
[simulation-2] [INFO] [1675100219.448068381] [simulation]: Simulation initialized
[controller-1] [INFO] [1675100219.493284205] [controller]: Controller initialized
[motion_planner-3] [INFO] [1675100219.493718287] [motion_planner]: MotionPlanner initialized
[atc_dummy-5] [INFO] [1675100219.538874490] [atc_dummy]: ATCDummy initialized
[task_planner-4] [INFO] [1675100219.573647711] [task_planner]: TaskPlanner initialized
[task_planner-4] [INFO] [1675100221.576466055] [task_planner]: Published first tasks
[motion_planner-3] [INFO] [1675100221.576867407] [motion_planner]: Received initial state: ruby_interfaces.msg.State(y=0
.0, x=0.0, z=0.0, v=0.0, t=ruby_interfaces.msg.Time(sec=1675100222, nanosec=575882753))
[motion_planner-3] [INFO] [1675100221.577706568] [motion_planner]: Received tasks: [ruby_interfaces.msg.State(y=100.0, x
=100.0, z=0.0, v=0.0, t=ruby_interfaces.msg.Time(sec=0, nanosec=0)), ruby_interfaces.msg.State(y=100.0, x=200.0, z=0.0,
v=0.0, t=ruby_interfaces.msg.Time(sec=0, nanosec=0)), ruby_interfaces.msg.State(y=300.0, x=300.0, z=50.0, v=0.0, t=ruby_
interfaces.msg.Time(sec=0, nanosec=0))]
[controller-1] [INFO] [1675100222.494885285] [controller]: Received path
[simulation-2] [INFO] [1675100222.994704195] [simulation]: Received control signal: [14.98321451 14.98321451  0.
], Current state: (0.0, 0.0, 0.0, 0.0)
```

Figure 4: Start-up sequence of the ROS agent

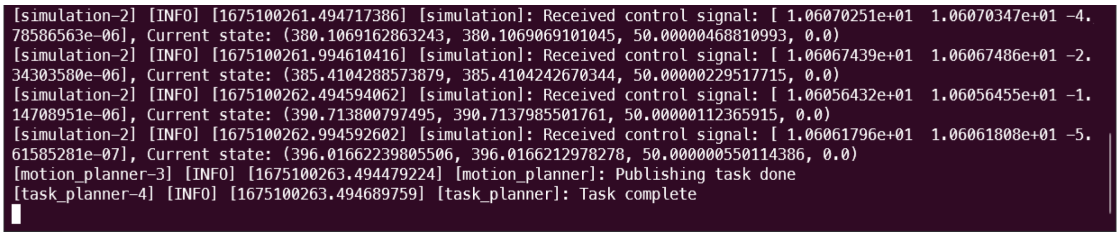
As the agent progresses along the path, each node will continually send status updates. We see the first of these at the end of figure 4, and a snippet from the middle of the flight in figure 5. The most common status message is the one sent by the simulation node, as it has a significantly higher refresh rate than the other nodes. In figure 5, the Motion Planner has received the current state from the Controller and publishes ”task done”, prompting the Task Planner to generate a new set of waypoints. These new waypoints are sent back to the Motion Planner, the route gets updated and the next step sent to the Controller.



```
[simulation-2] [INFO] [1675456571.778462975] [simulation]: Received control signal: [-8.86393919e-05  1.49998847e+01  0.
00000000e+00], Current state: (100.00008682885098, 198.63785778152237, 0.0, 0.0)
[motion_planner-3] [INFO] [1675456572.278398371] [motion_planner]: Publishing task done
[simulation-2] [INFO] [1675456572.278491367] [simulation]: Received control signal: [18.31950658  3.17197372  4.57988749
], Current state: (100.00004250915498, 206.13780011685708, 0.0, 0.0)
[task_planner-4] [INFO] [1675456572.278538727] [task_planner]: Task complete
[task_planner-4] [INFO] [1675456572.278812445] [task_planner]: Published new task
[motion_planner-3] [INFO] [1675456572.279297293] [motion_planner]: Received tasks: [ruby_interfaces.msg.State(y=400.0, x
=400.0, z=50.0, v=0.0, t=ruby_interfaces.msg.Time(sec=0, nanosec=0))]
[controller-1] [INFO] [1675456572.778335362] [controller]: Received path
[simulation-2] [INFO] [1675456572.778375887] [simulation]: Received control signal: [15.8925108   4.74248853  3.97313301
], Current state: (109.15979579915461, 207.72378697566938, 2.2899437469435338, 0.0)
[simulation-2] [INFO] [1675456573.278392709] [simulation]: Received control signal: [14.46572828  5.65882083  3.61643467
], Current state: (117.10605119896516, 210.09503123966698, 4.276510252563128, 0.0)
[simulation-2] [INFO] [1675456573.778396827] [simulation]: Received control signal: [13.76498941  6.11177029  3.44124863
], Current state: (124.33891533770097, 212.9244416545838, 6.084727587392759, 0.0)
```

Figure 5: ROS agent has reached a waypoint, Task Planner sends new route

Furthermore, in figure 6 we see the final stages as the agent reaches its destination, signified by the Motion Planner publishing the ”task done” message, and the Task Planner responding with ”Task complete”.



```
[simulation-2] [INFO] [1675100261.494717386] [simulation]: Received control signal: [ 1.06070251e+01  1.06070347e+01 -4.
78586563e-06], Current state: (380.1069162863243, 380.1069069101045, 50.00000468810993, 0.0)
[simulation-2] [INFO] [1675100261.994610416] [simulation]: Received control signal: [ 1.06067439e+01  1.06067486e+01 -2.
34303580e-06], Current state: (385.4104288573879, 385.4104242670344, 50.00000229517715, 0.0)
[simulation-2] [INFO] [1675100262.494594062] [simulation]: Received control signal: [ 1.06056432e+01  1.06056455e+01 -1.
14708951e-06], Current state: (390.713800797495, 390.7137985501761, 50.00000112365915, 0.0)
[simulation-2] [INFO] [1675100262.994592602] [simulation]: Received control signal: [ 1.06061796e+01  1.06061808e+01 -5.
61585281e-07], Current state: (396.01662239805506, 396.0166212978278, 50.000000550114386, 0.0)
[motion_planner-3] [INFO] [1675100263.494479224] [motion_planner]: Publishing task done
[task_planner-4] [INFO] [1675100263.494689759] [task_planner]: Task complete
```

Figure 6: ROS agent has reached the destination

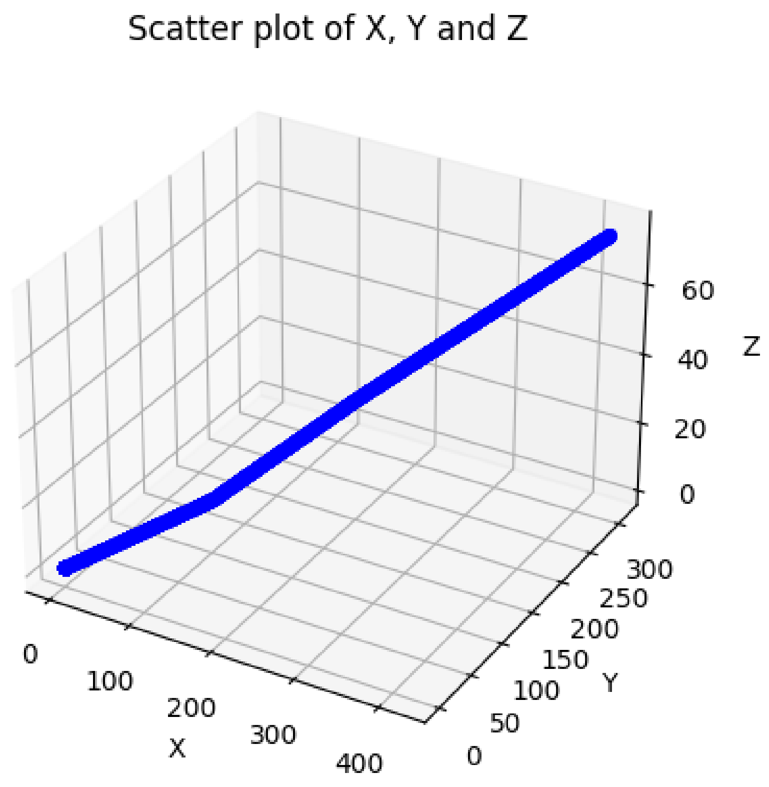


Figure 7: Scatter plot of the coordinates gathered in a test flight of the ROS2 system.

### 3.2.2 ATC interference scenario

In order for the system to be able to respond to ATC commands, it is currently implemented such that the Task Planner listens for new ATC input through a subscription to the ATC Planner. When it retrieves a new ATC plan through this subscription, it will switch to an ATC state where it stops following the global plan and starts following the retrieved ATC plan. In the logs, this is indicated with a message saying `Publishing ATC tasks: ...`, followed by the coordinates moving in the direction of those tasks. When the ATC tasks are completed, the system will switch back to the global plan and follow those tasks. As it stands, there is no implementation of updating the global plan with the new state of the aircraft after following ATC. The procedure for issuing a new ATC command to the system is through publishing a new message to the `/inc_atc` topic. An example of how this can be achieved is shown in Figure 8.

```
ros2 topic pub --once /inc_atc ruby_interfaces/msg/ATCMessage "{action:'direct',target:'CH361'}"
```

Figure 8: Issue ATC Command

## 3.3 Rootless Docker container

All development was made in a rootless Docker container. In order to run the ROS2 project with all the configurations, a set of packages and modules are needed. The rootless Docker container that was set up enables us to run the system on the same conditions every time. The underlying operating system becomes non-important because as soon you opt into the container, you are in the same environment as if you would be in on any other machine. This simplified the development process significantly where the ROS2 setup only needed to be done once, and on one machine, but could be used by all included team members on separate machines. The hand-off of the project also becomes smooth as the setup process on a new machine is well documented on the project Github-page.

# 4 Discussion

## 4.1 Future Work

This section covers insights and discussions around future work for the system components.

#### Motion Planner

As it stands, the motion planner forwards whatever waypoints it received from the task planner to the controller with added time stamps. Instead, it would be more suitable to send the nodes that occur in the next $t = 30$ seconds. This principle would be enabled further if more complex motion primitives are added and a proper discretization of the space between two waypoints is conducted resulting in intermediate nodes between waypoints.

#### Motion Primitives

More complex motion primitives are a natural extension considering the current primitives just consist of a motion between two waypoints. Motion primitives should be distinctly separated between ground and air movement such that they can be stitched together during a takeoff maneuver. A recommended first step would be to look at a vehicle based motion primitive generator such as CommonRoad[1] and consider extending this generator into a z-dimension for the air based primitives.

#### Controller

Extending this node to a simple PID controller takes little effort and would serve as a simple improvement. Long term, however, it's of more interest to consider a model predictive controller. Given a more complicated physical model of the aircraft, perhaps one supplied by Saab, it is possible to generate feasible and more dynamically constrained control signals using an external package such as ACADO[16]. This would allow us to model constraints, both for the aircraft and for the state space, into the resulting control signal.

#### Simulation

Upgrading the physical aircraft model is an important step to making the system more real and its a natural extension of this node. An upgraded model would enable the controller and motion planner to generate more realistic outputs.

#### Visualization

As it stands, the visualization node does not work as intended. A first step for future work would be to finish implementing the intended functionality. In order to visualize the taxying procedure and take-off/landing at airports, a 3D visualization might be preferable.

#### Flight Plan & database

Currently the database only holds what was needed for the development of this project. In the future it is necessary to be able to simulate and run tours from any airports using all waypoints. To be able to do this it would be preferred to make a real database that is not just a Python class. When this has been done the database can be extended to hold all all necessary data available.

### ATC Planner

As with the flight plan, the ATC Planner would in the future need to be linked with a database holding waypoint information so that conversions into coordinates from ATC commands are possible. The switch between following ATC instructions and the original flight plan also needs improvement. One idea in order to make this switch more seamless is to copy the original flight plan, make changes to it according to ATC instructions, and publish it so that it can be followed.

# 5 Conclusion

In conclusion this project has provided both technical and communicative insights.

### Technical

From a technical stand point, the outcomes of this project constitutes an initial ground for autonomous flight between two airports, including the ability to handle commands from air traffic control (ATC). The project is implemented as a complete Robot Operating System (ROS) 2 application, using state-of-the-art open source tools for functionality such as planning, control, simulation, and modeling. The project is divided into several modular components, including an interpreter for natural language ATC commands, a motion planner using motion primitives, and a controller for executing the generated trajectory. The project demonstrates good modularity, ease of distribution, and the ability to iteratively improve and upgrade individual components.

### Communicative

In addition to the technical aspects, the project also included some communicative conclusions regarding working as a team with different responsibilites. Coordinating and scheduling meetings and coding sessions with team members and the project supervisor can be challenging when everyone has different schedules. The starting phase of the project could also be as taking up a large portion when judging by the overall time estimated for the course. But the project as a whole has great oppurtunity for further development and there now exists a solid foundation to work off of.

## References

[1] Prof. Dr.Ing. Matthias Althoff. *Tutorial: Motion Primitive Generator.* URL: `https://commonroad.in.tum.de/tutorials/motion-primitive-generator`. (accessed: 13-12-2022).

[2] Charles Anderson. "Docker". In: *IEEE Software* 32 (2015). DOI: `10.1109/MS.2015.62`. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7093032`.

[3] Kristoffer Bergman, Oskar Ljungqvist, and Daniel Axehill. "Improved Optimization of Motion Primitives for Motion Planning in State Lattices". In: *2019 IEEE Intelligent Vehicles Symposium (IV).* 2019, pp. 2307–2314. DOI: `10.1109/IVS.2019.8813872`.

[4] Docker. *What is a container?* URL: `https://www.docker.com/resources/what-container/`. (accessed: 16-12-2022).

[5] Open Source Robotics Foundation. *Gazebo, robot simulation made easy.* 2014. URL: `https://classic.gazebosim.org/`.

[6] GeoPandas. *Geopandas: Python tools for Geographic Data.* URL: `https://github.com/geopandas/geopandas`. (accessed: 07-01-2023).

[7] David González et al. "A Review of Motion Planning Techniques for Automated Vehicles". In: *IEEE Transactions on Intelligent Transportation Systems* 17.4 (2016), pp. 1135–1145. DOI: `10.1109/TITS.2015.2498841`.

[8] Ivan J. Williams Joseph J. Distefano Allen R. Stubberud. *Theory and Problems of Feedback and Control Systems.* 1967.

[9] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: `10.1126/scirobotics.abm6074`. URL: `https://www.science.org/doi/abs/10.1126/scirobotics.abm6074`.

[10] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org.* `https://www.openstreetmap.org`. 2017.

[11] Open Robotics. *ROS 2 Humble Tutorial.* URL: `https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools.html`. (accessed: 11-10-2022).

[12] Open Robotics. *ROS 2 Release Humble Hawksbill.* URL: `https://docs.ros.org/en/foxy/Releases/Release-Humble-Hawksbill.html`. (accessed: 11-10-2022).

[13] Open Robotics. *Ros2/rviz: Ros 3D Robot visualizer.* URL: `https://github.com/ros2/rviz`. (accessed: 07-01-2023).

[14] Open Robotics. *RQT ROS Wiki.* URL: `http://wiki.ros.org/rqt`. (accessed: 13-12-2022).

[15] Mattias Tiger et al. "Enhancing Lattice-Based Motion Planning With Introspective Learning and Reasoning". In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 4385–4392. DOI: `10.1109/LRA.2021.3068550`.

[16] Milan Vukov. *ACADO Toolkit.* URL: `https://acado.github.io/`. (accessed: 13-12-2022).