UNIVERSITY OF TARTU
Institute of Technology
Computer Engineering

Danel Leppenen

Nav 2 PYIF: Python-based motion planning for ROS 2 Navigation 2

Bachelor's Thesis (12 ECTS)

Supervisor(s):

Robert Valner

Houman Masnavi

Karl Kruusamäe

Tartu 2023

Abstract/Resümee

Nav 2 PYIF: Python-based motion planning for ROS 2 Navigation 2

Abstract:

This thesis demonstrates a controller plugin with a Python Interface (PYIF) which allows Python-based local planners to be used in Nav 2.

Keywords:

C++, ROS 2, Navigation 2, Python, embed, extend

CERCS: P170, Computer science

Nav 2 PYIF: Pythoni põhine liikumise planeerija ROS 2 Navigation 2-le

Lühikokkuvõte:

See töö toob näite Nav 2 *controlleri* pistikprogrammist, millel on Pythoni liides, mis võimaldab Pythonis kirjutatud liikumise planeerijaid kasutada Nav 2-s.

Võtmesõnad:

C++, ROS 2, Navigation 2, Python, manustama, laiendama

CERCS: P170, Arvutiteadus

Table of Contents

Abbreviations

API - Application Programming Interface

CPU - Central Processing Unit

DDS - Data Distrubution Service

IDL - Interface Description Language

Nav 2 - Navigation 2

PYIF - Python Interface

RAM - Random Access Memory

ROS - Robot Operating System

Rviz2 - ROS Visualization2

YAML - YAML Ain't Markup Language

1 Introduction

Autonomous systems have been the cornerstone of technological development since G. Devol and J. Engelberger created the first industrial robot. This system was a stationary robotic arm called the Unimate and it automated the unsavoury task of operating die-casting machines [1, p. 9]. Thereafter autonomous systems have spread from industrial settings to all aspects of life e. g. home appliances like robotic vacuums.

The cornerstone of service robots like autonomous vacuum cleaners is navigation. Navigating means that the autonomous system has to tackle the problem of motion planning which in its simplest form involves using a map of the static environment to find the safest optimal path to the goal. In practice robots usually work in semi-deterministic environments and therefore safely traversing to the goal requires perception and mapping of surroundings. This is achieved by fusing sensor data from multiple sources that motion planners then use for obstacle detection. Additionally motion planners need to accommodate various behaviours from recovering the robot incase of a failure to reach the goal e.g. collision or a dead end to considering how intrusive the plan might be when entering an environment that is not confined to inanimate objects e. g. avoiding routes through groups of bystanders may be preferred [2]. [3]

Consequently solving problems of motion planning is a complex task and it is an active research area. The tool of choice for the bleeding edge research that goes into motion planning is increasingly Python as it facilitates rapid prototyping and supports a variety of GPU accelerated linear algebra libraries [4]. Commonly used framework for applying motion planners to robotic hardware is the Robot Operating System (ROS) which is an open-source framework that provides tools, libraries and conventions to reduce the complexity of creating robotic systems [5]. The most recent generation of the framework ROS 2 contains software packages such as MoveIt! that focuses on motion planning for robotic arms and Navigation 2 (Nav 2) that focuses on the kind of motion planning challenges mentioned previously. Although ROS 2 supports Python, Nav 2 is implemented in C++ thus utilising the advancements in motion planning are often too impractical due to the discrepancy between implementations.

The goal of this thesis is to bridge the gap between Python-based motion planning research and the successor of the widely adopted robotics framework ROS 2 by creating a standard controller plugin that incorporates a Python interface (PYIF) for the Nav 2 package. The

PYIF allows developers to use Python based motion planners in Nav 2 without creating a custom Nav 2 build.

2 Domain overview

The following chapter describes what motion planning is, how it is commonly utilised for research and for practical solutions on autonomous systems.

2.1 Motion planning

The purpose of motion planning is navigating safely and autonomously to a desired destination. Fundamentally this is achieved by giving motion planners information about its surroundings, current location and desired goal. These inputs are then used to calculate velocity commands that will move the robot closer to the goal as shown in Figure 1. [2]

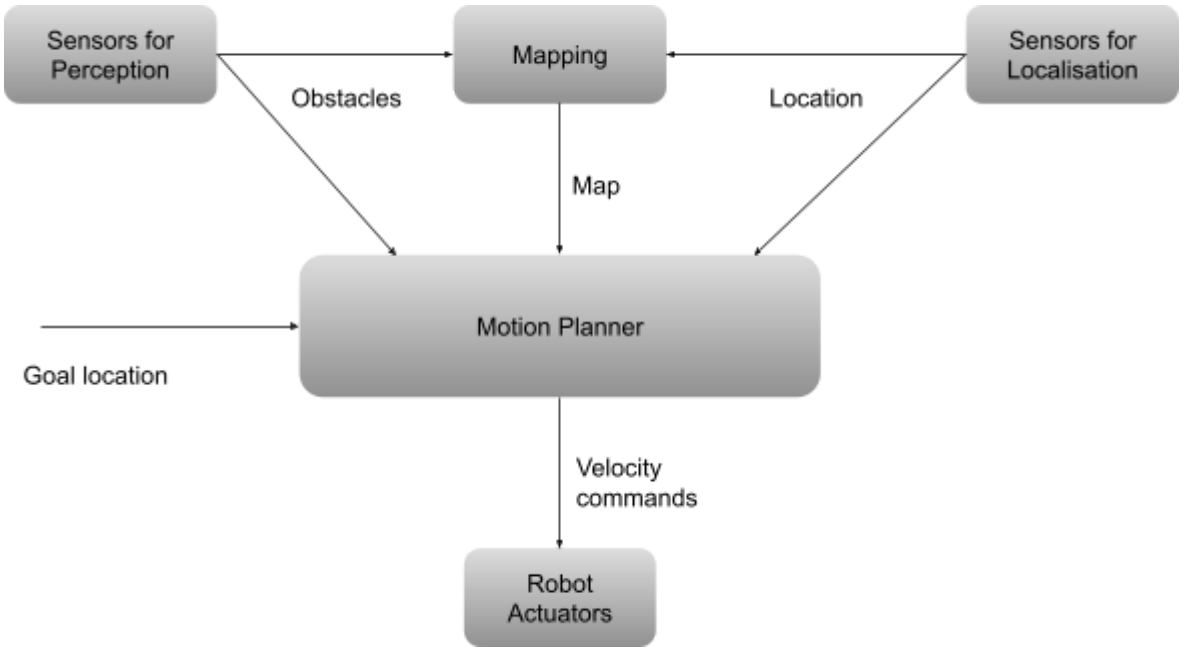

Figure 1. *Function of motion planners in autonomous systems.*

Consequently, navigating in a semi-deterministic environment (i. e. static environment that may be only partially known beforehand) requires perception, localisation and mapping subsystems which fuse sensor data from multiple sources like wheel odometry, IMU, GPS and LIDAR [6]. The data gathered from mapping is commonly modelled as an occupancy grid that discretizes the environment into cells that contain a probability of occupancy also called the cost of the cell [7]. Example of a costmap can be seen in the Figure 2 below.

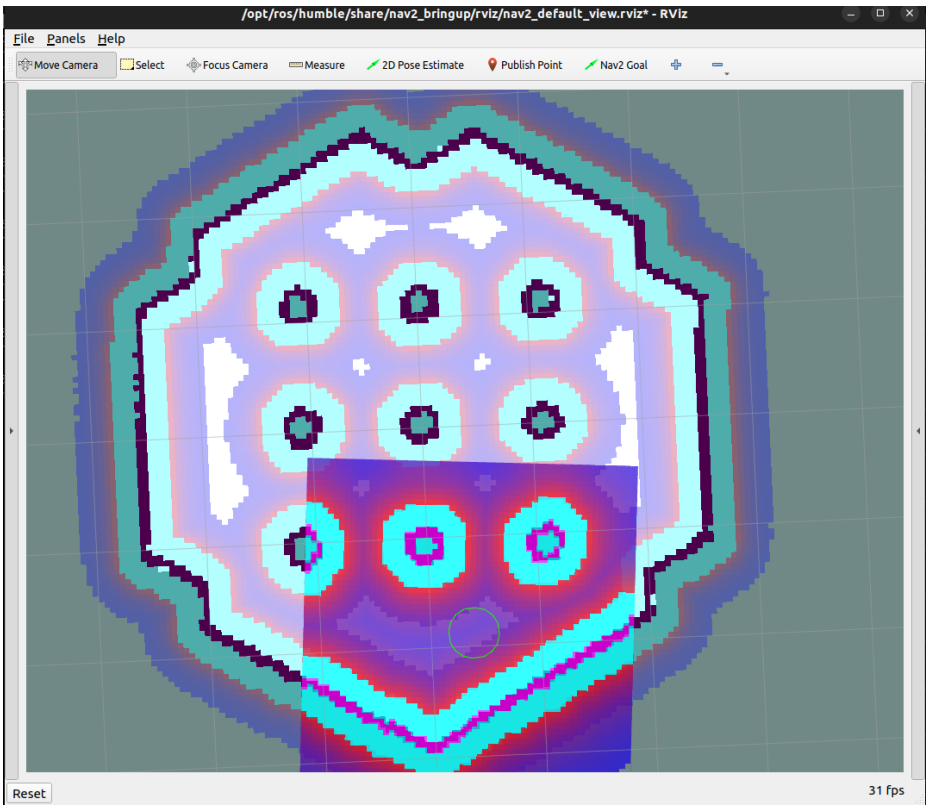

Figure 2. *Example of costmap in ROS 2 visualisation tool Rviz2.*

Taking the map as input techniques such as A* or D* (Figure 3) algorithms are used to find the optimal path on occupancy grids [7] [8]. D* algorithm factors in cost updates for the occupancy grid and is therefore usable for exploratory robots [7]. On the other hand if the environment is dynamic classical strategies like D* perform better alongside reactive strategies like artificial potential fields (Figure 4) [9].

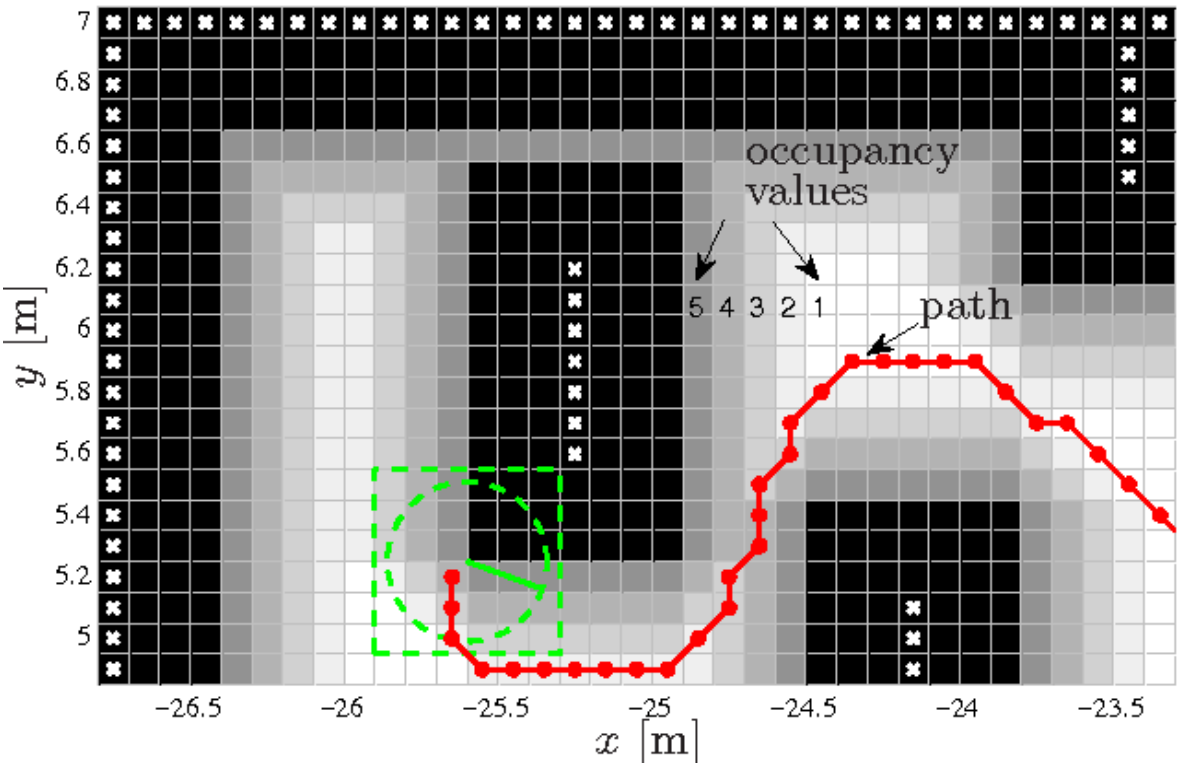

Figure 3. *High level D* plan on occupancy grid showing the most optimal safe path.* [10]

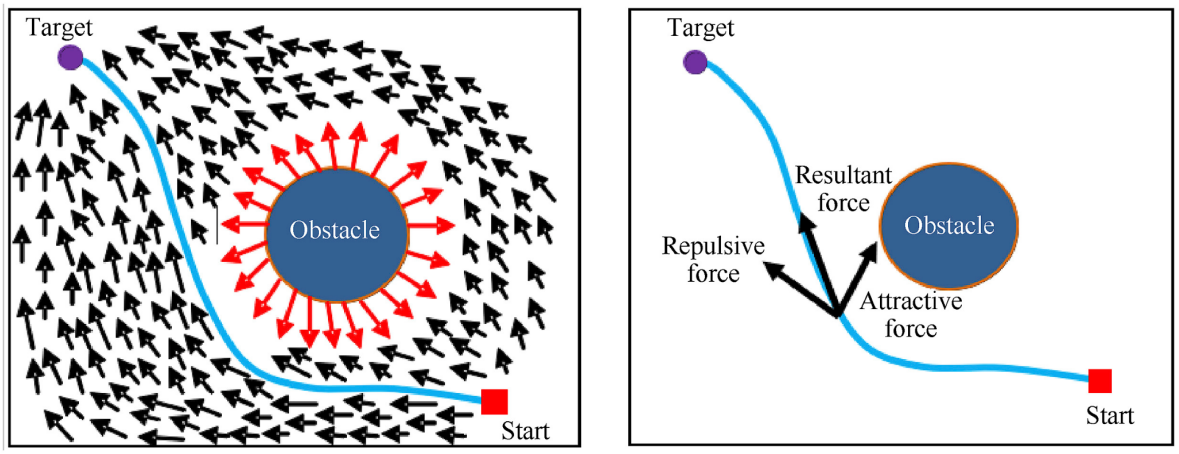

Figure 4. *Artificial potential field showing forces guiding robot towards goal.* [9]

Motion planners therefore combine multiple techniques to cover a wider range of situations efficiently. Common method for implementation includes segregating motion planning into a global and a local planner (Figure 5) as used in [2] and [11]. In these examples classical algorithms are implemented in global planners to create a high-level plan based on the known

environment while reactive strategies are used in local planners to create velocity commands to follow the path as closely as possible while avoiding obstacles.

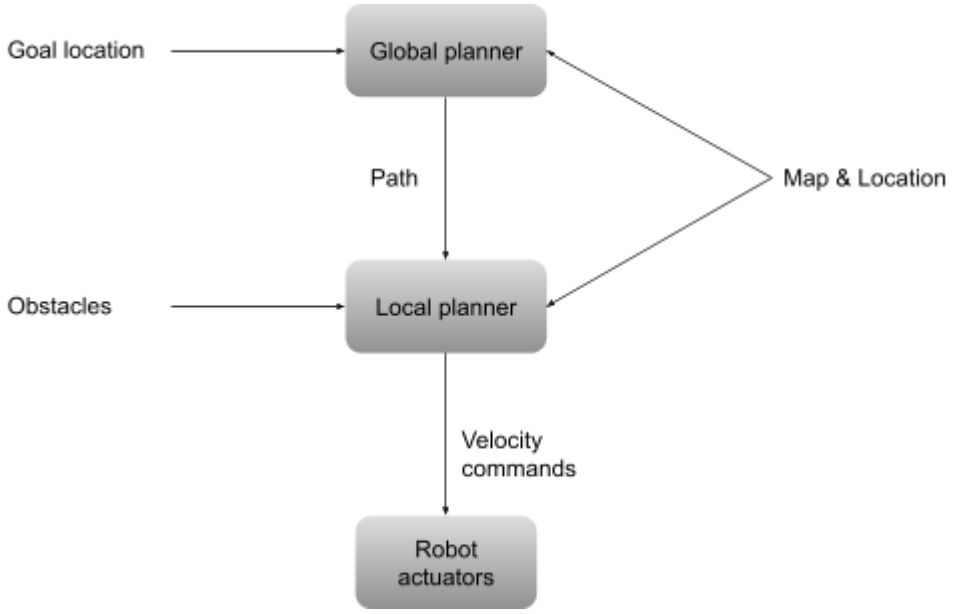

Figure 5. *Common implementation of motion planners in autonomous systems.*

2.2 Research In Motion Planning

Research for developing motion planners is increasingly becoming Python-based. Python is widely adopted in research communities due to its wide array of libraries that facilitate prototyping. Examples of Python-based research for motion planning can be seen in works like "GPU Accelerated Convex Approximations for Fast Multi-Agent Trajectory Optimization" [4] and "Visibility-Aware Navigation With Batch Projection Augmented Cross-Entropy Method Over a Learned Occlusion Cost" [12]. Those works use machine learning libraries like JAX [13] and CUPY [14] in their process.

2.3 ROS 2

Implementing motion planners for practical applications requires adding subsystems for sensors, actuators, mapping and more as shown in Figure 1. The most popular solution in robotics for reducing the complexity of developing such systems is ROS [5].

ROS 2 which is the successor to the original ROS is a modular open-source Linux-based framework [15]. ROS 2 provides middleware for communication, reuse of code by dividing it into separate processes with a specific purpose (e. g. mapping or localisation) and a wide

array of tools from debugging to visualisation (e. g. Rviz2 shown in Figure 2) [15] [5]. The design of ROS makes system components independent and easily swappable [5].

ROS 2 offers multiple communication patterns where the endpoints are known as *nodes*. The most commonly used pattern is the *publisher-subscriber* pattern that allows asynchronous many-to-many communication via *topics* using the Data Distribution Service (DDS) (Figure 6). Another common pattern is called *actions*. Actions are asynchronous communication interfaces that utilise *remote procedure call* along publisher-subscriber pattern for periodic feedback in long running goal-oriented tasks (Figure 6). The middleware interfaces such as topics and actions use ROS *message types* that are defined by the Interface Description Language (IDL). The middleware is also agnostic to the endpoint locations being intra- or inter-process. The nodes can be implemented as *components* where multiple components can be allocated to separate processes or into a single process to reduce latency and resource usage. [15]

The communication middleware is available for Python, Java and C++ allowing processes to communicate between nodes written in different languages using ROS messages [15] .

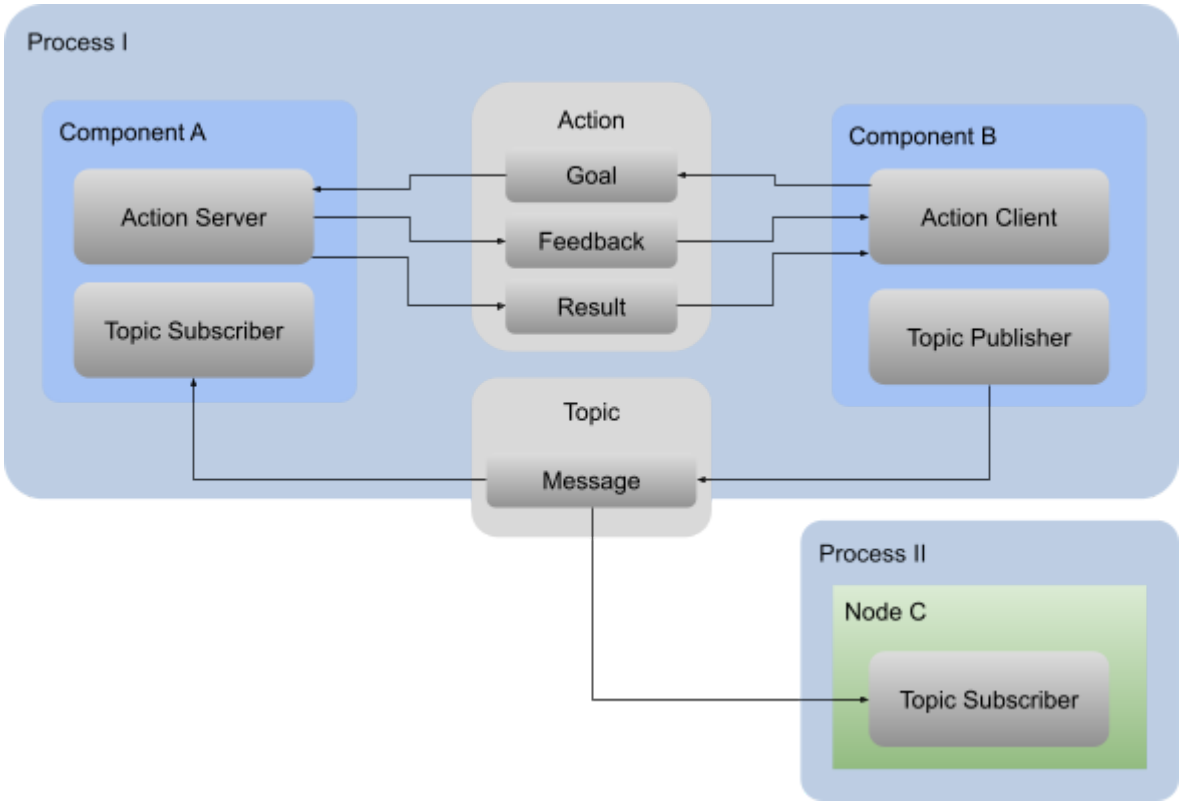

Figure 6. *ROS 2 communication patterns.*

2.4 ROS 2 Navigation 2

Using the tools provided by ROS it is possible to write custom motion planning nodes in Python or C++ and reusing code already created by the community for sensors, actuators, localisation etc.. However ROS 2 already incorporates a standardised, dynamically reconfigurable tool for implementing motion planners called Navigation 2 (Nav 2) [8]. Nav 2 uses two patterns in its design: a behaviour tree and task-specific action servers (Figure 7) [8]. The behaviour tree server tracks the progress of action servers and delegates new tasks to the servers to calculate new paths, create movement commands or mitigate issues that occur while navigating. [8]

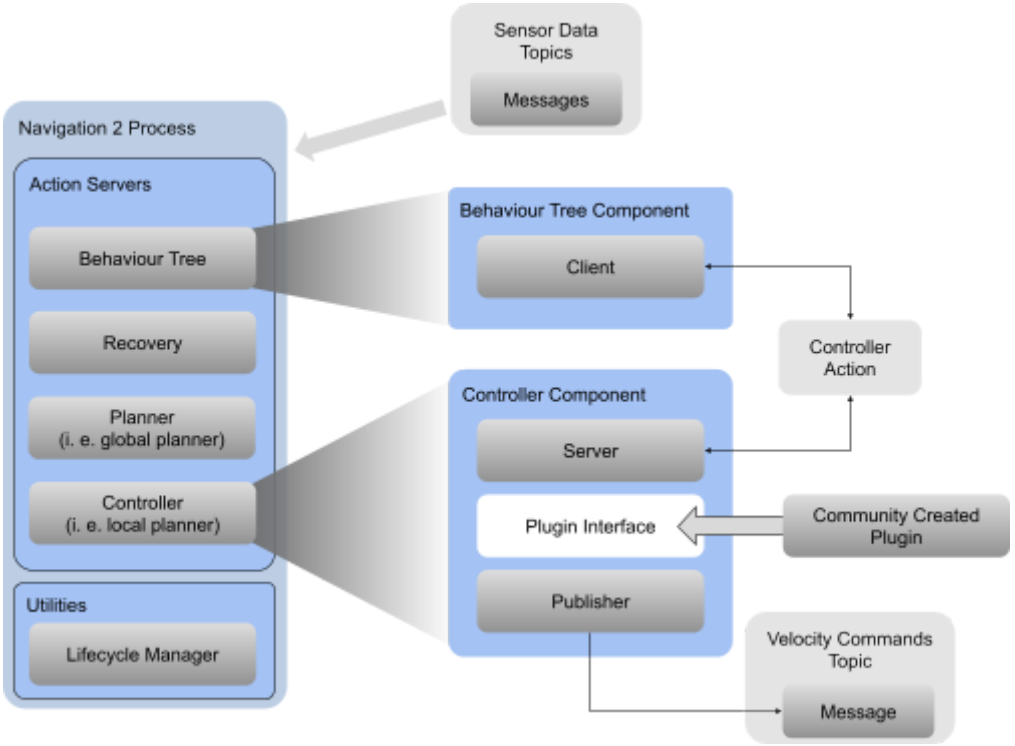

Figure 7. *Architecture of Navigation 2.*

The action servers are ROS 2 components or *composable* nodes that have a plugin interface which allows for run-time configuring of the navigation system (Figure 8) [8]. Usage of components allows to manage the lifecycle of nodes [15] resulting in deterministic program lifecycle and memory allocations [8]. The action servers are managed by the lifecycle manager utility. The action servers include planner (i.e global planner), controller (i.e. local planner) and recovery servers. Plugins for planner and controller servers follow the traditional design where planners implement classical or non-reactive algorithms for optimal paths and controllers implement reactive algorithms to avoid obstacles. The addition of the

recovery server gives the motion planner a set of behaviours that are used to solve navigation failures. [8]

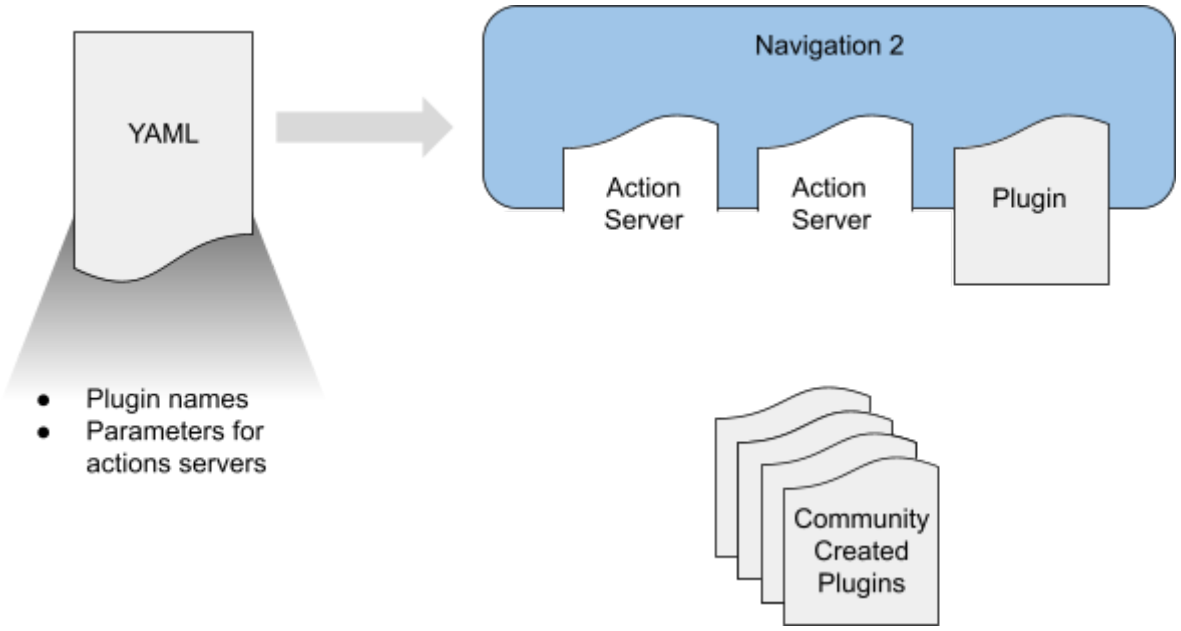

Figure 8. *Nav 2 is configured from a YAML file that holds parameters and plugins for action servers.*

Nav 2 therefore offers an entire ecosystem for implementing and managing new motion planners. The required interface for Nav 2 plugins is seen in Figure 9 where the yellow arrows demonstrate how the virtual functions in the interface must be overridden in the plugin [8]. Consequently, new plugins can only be implemented in C++ and not in Python because Nav 2 is entirely a C++ based application.

```
virtual void configure(
      const rclcpp_lifecycle::LifecycleNode::WeakPtr &,
      std::string name, std::shared_ptr<tf2_ros::Buffer>,
      std::shared_ptr<nav2_costmap_2d::Costmap2DROS>) = 0;
virtual void cleanup() = 0;
virtual void activate() = 0;
virtual void deactivate() = 0;
virtual void setPlan(const nav_msgs::msg::Path & path) = 0;
virtual geometry_msgs::msg::TwistStamped computeVelocityCommands(
      const geometry_msgs::msg::PoseStamped & pose,
      const geometry_msgs::msg::Twist & velocity,
      nav2_core::GoalChecker * goal_checker) = 0;
virtual void setSpeedLimit(const double & speed_limit, const bool
& percentage) = 0;
```

Figure 9. *Example of virtual functions in the plugin interface of the controller server that need to be overridden.*

3 Related Work

This chapter explains how the discrepancy between motion planning research and practical applications has been circumvented so far.

3.1 Workarounds for Python-Based Planners

For now no practical workaround exists for implementing Python-based planners in Nav 2. The only approach is to write a custom Python node using ROS tools as was done in the work of [12] for testing the solution (Figure_10). Although this solution is modular it does not fit in the ecosystem of Nav 2 and can not leverage the tools and off the shelf solutions it provides. This results in a less reliable solution since the node is not managed by Nav 2 and can not benefit from inter-process communication. Circumventing this problem requires rebuilding the Nav 2 software package in a way that incorporates the custom controller but as a consequence creates a custom Nav 2 package which is undesirable for code reusability.

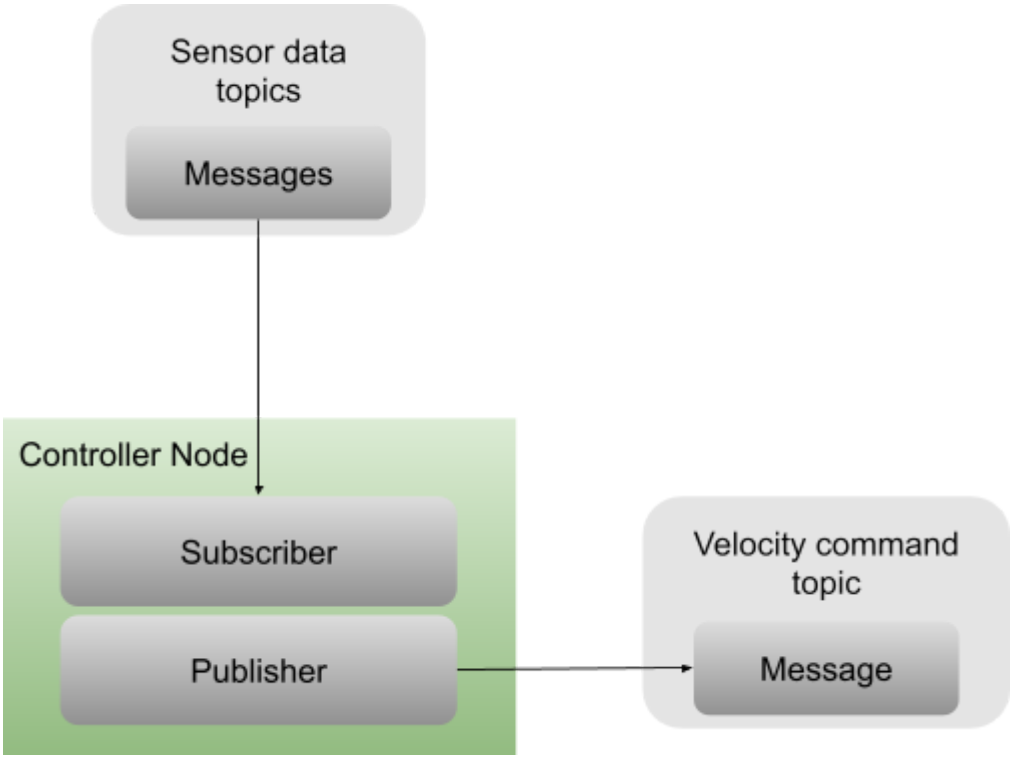

Figure 10. *Simple approach used for testing Python-based planner.*

3.2 Python C-API

Python is an interpreted object-oriented language. The reference implementation of Python is written in C and called CPython [16]. CPython offers tools for converting C-types to Python objects, calling Python objects (e. g. calling Python functions) and more in the Python C-API [17]. CPython therefore has built-in functionality to extend and embed the Python interpreter in C/C++ . Since Python is interpreted, required modules and objects must be imported at runtime and are stored on the heap. For memory management every object holds a reference count. The count is incremented at the creation of the object and decremented when it goes out of scope; at other times the count must be handled explicitly to avoid segmentation faults. [16]

In Figure 11 an example of a pseudocode is given of the embedding process. Embedding requires importing the desired Python module, retrieving the *PyObject* to a callable function, converting C-type parameters into PyObjects and calling the function PyObject with the parameter pack. [16]

```
# function in python_module
def HelloWorld(msgFromCPP):
    print('Hello' + msgFromCPP)


void CPPy :: CallOutPython() {
    PyObject* pyModule = PyImport_Import((char*)"python_module");
    PyObject* pyFunc = PyObject_GetAttrString(pyModule, (char*)"HelloWorld");
    PyObject* pyArgs = PyTuple_Pack(1, PyUnicode_FromString((char*)" World!"));
    PyObject_CallObject(pyFunc, pyArgs);
}
```

Figure 11. *Embedding pseudocode where the upper function in Python is embedded into the lower function in C++ and printing "Hello World!".*

4 Objective

Python is widely adopted in research communities as it facilitates rapid prototyping and has a wide array of available libraries. These include machine learning libraries that are heavily utilised in bleeding edge motion planning research. However the motion planning stack of the most popular robotics framework the Nav 2 is implemented entirely in C++.

Workarounds like porting the code are often unreasonable due to the time it takes or even implausible if the used libraries would need to be ported as well. Workarounds such as writing a standalone navigation package around the Python-based planner do not leverage the tools and off the shelf solutions that Nav 2 offers which often leaves the research unutilised.

The main objective of this thesis is to bridge this discrepancy of implementations between motion planning research and practical applications and use Python-based motion planners in Nav 2 by creating a controller plugin that implements a Python-based motion planner.

4.1 Functional Requirements

The following describes the requirements for the controller plugin solution. The solution must be:

1. Python-based planner agnostic.
2. Non-invasive to Nav 2 for leveraging the tools it provides.
3. Easy to use to be a viable alternative to the workarounds.

4.2 Software Requirements

1. Ubuntu 22.04 running ROS Humble distribution.
2. C++ 17 and Python 3.10 for developing the Nav 2 controller plugin.

5 Design

The solution proposed in this thesis is a controller plugin for Nav 2 that adds a Python interface (PYIF) which allows embedding Python-based local planners from YAML configuration as shown in Figure 12. The Python interface provides tools to convert ROS messages between C++ and Python and fast access to often used PyObjects.

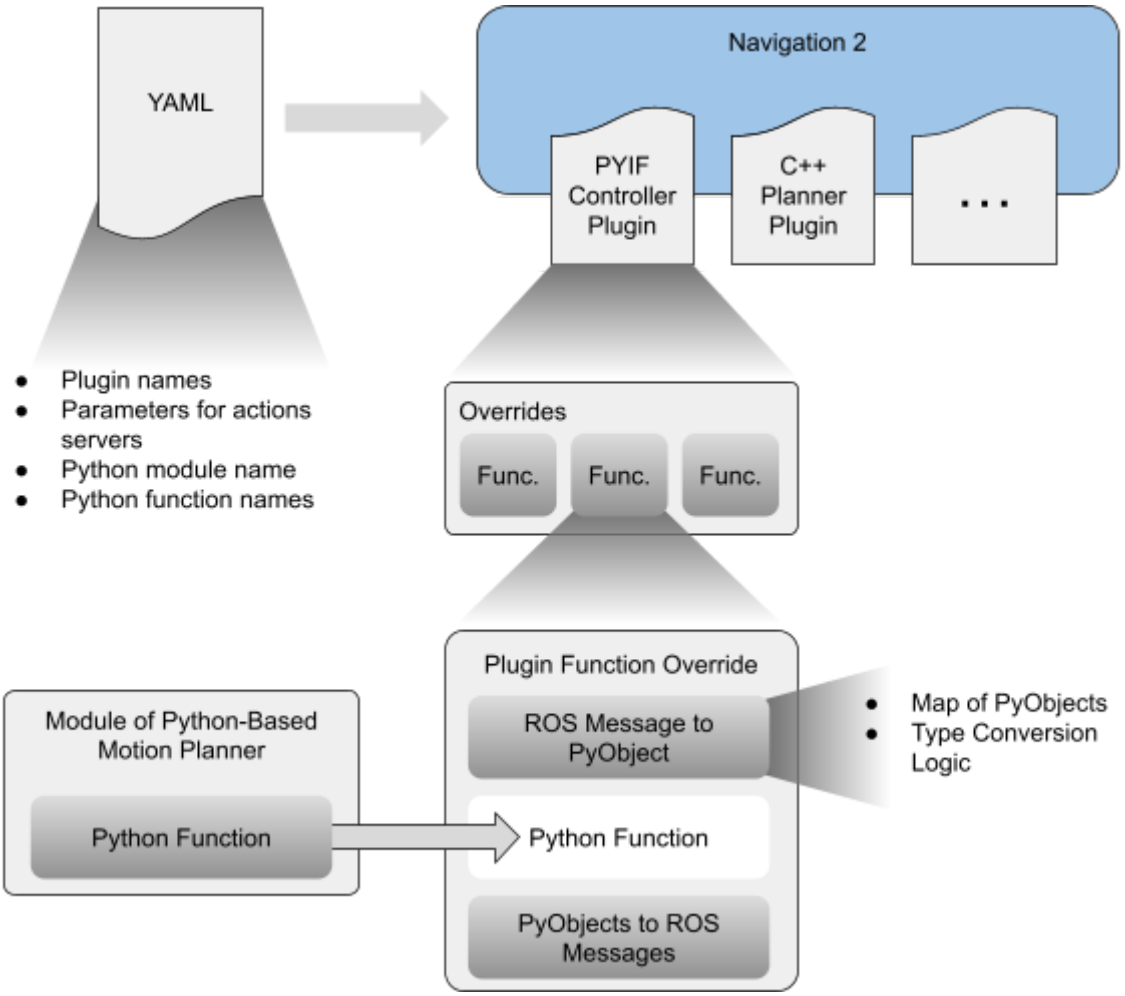

Figure 12. *Nav 2 controller plugin that incorporates Python functions from a motion planning module in the function overrides.*

The following sections describe the components of the Nav 2 PYIF controller plugin and what approaches were used to implement them. Section 5.1 looks into the inner workings of the PIYF and subsection 5.1.1 describes in detail the components that enable non-invasive embedding of Python-based local planner in Nav 2.

5.1 The Nav 2 Python Interface (PYIF) Controller Plugin

The PYIF controller overrides the plugin interface using two patterns. The first pattern is used for managing the Python interpreter and the required modules. This pattern is used for the *configure* function and the *cleanup* function from the plugin interface as in Figure 9.

This pattern is shown in Figure 13 where the action server calls the controller plugin that incorporates the PYIF. The configure function is used for initialising the Python interpreter, importing modules and fetching PyObjects to functions by name and mapping them to avoid redundant importing and function fetching. The cleanup function is used for finalising the Python interpreter and deallocating the mapped PyObjects.

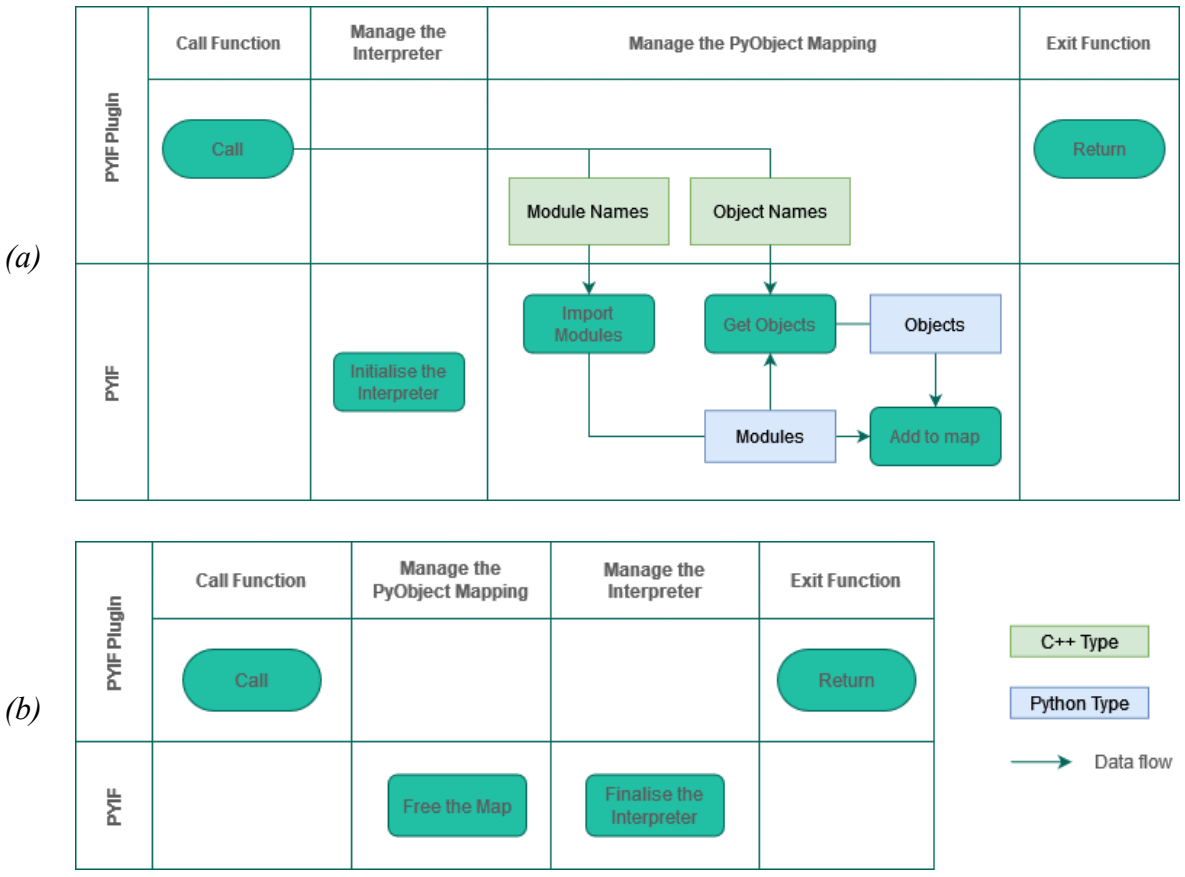

Figure 13. *The first pattern used for managing the interpreter, required modules and often used PyObjects in:*
(a) Configure override
(b) Cleanup override

The PyObject mapping is used to reduce time spent on fetching frequently needed PyObjects by the interpreter and instead holds them in an unordered map. The values stored in the map are the PyObjects and the keys are the names used for importing or fetching them (Figure 14).

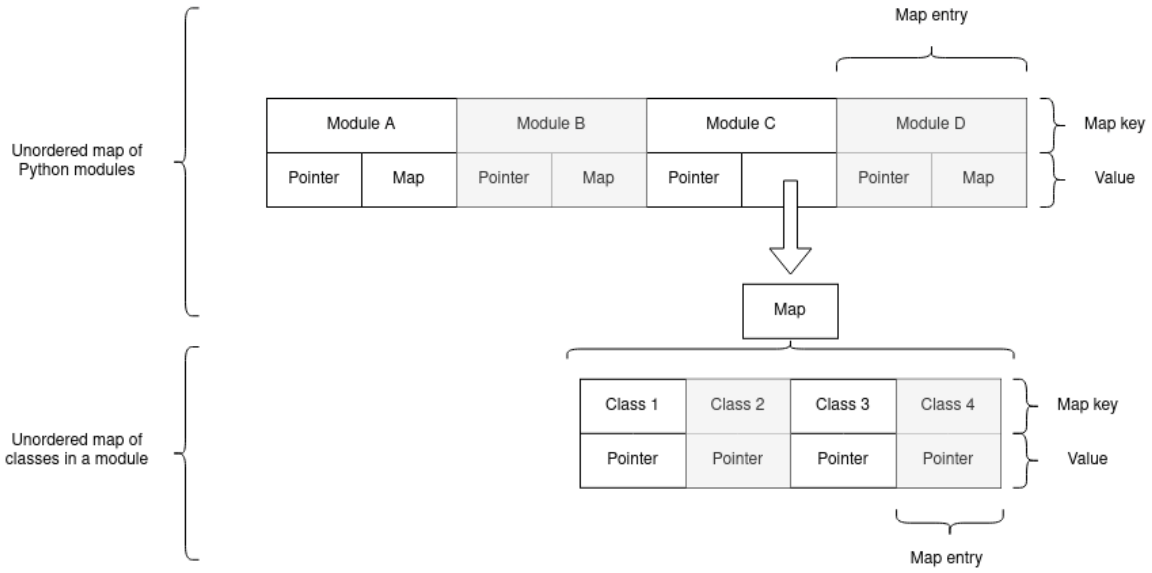

Figure 14. *The PyObject mapping in unordered maps.*

The second pattern is used for embedding Python functions in the plugin. This pattern is used for *setPlan*, *setSpeedLimit* and *computeVelocityCommands* functions of the plugin interface as shown in Figure 9. In order to embed Python in Nav 2 the motion planner is required to add function definitions with signatures analogous to the virtual functions in the controller plugin interface. The following Figure 15 demonstrates the required equivalent definitions based on the setPlan function.

(a)

```
void Plugin::setPlan(const nav_msgs::msg::Path & path) {
  return;
}
```

(b)

```
def setPlan(global_plan):
  return
```

Figure 15. *Analogous functions in:*
(a) C++ code
(b) Python code

The second pattern in PYIF controller that is used for overriding the plugin interface is shown in Figure 16. This pattern is used for passing data to the embedded Python-based motion planner and returning the results. To achieve this the function parameters which are ROS message types are converted between C++ and Python types before and after the Python

function call. This raises the main challenge for embedding Python-based motion planners for Nav 2 which is implementing the missing conversions for ROS message types.

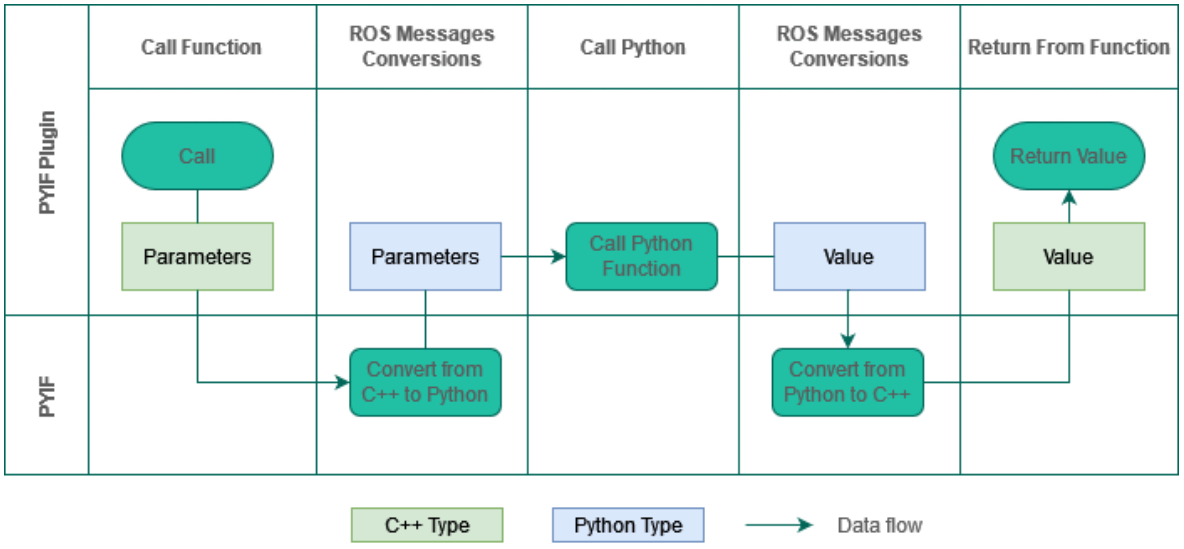

Figure 16. *The second pattern used for embedding Python functions.*

5.1.1 The ROS Message Conversions

ROS message types are hierarchical structures containing nested message types defined by the ROS communication middleware (Figure 17). Converting ROS message types requires conversions for each type that converts between C++ messages and Python messages from the most low level types such as the *point* type that contains *doubles* convertible by the standard Python C-API to the highest level types such as the *pose stamped* type.

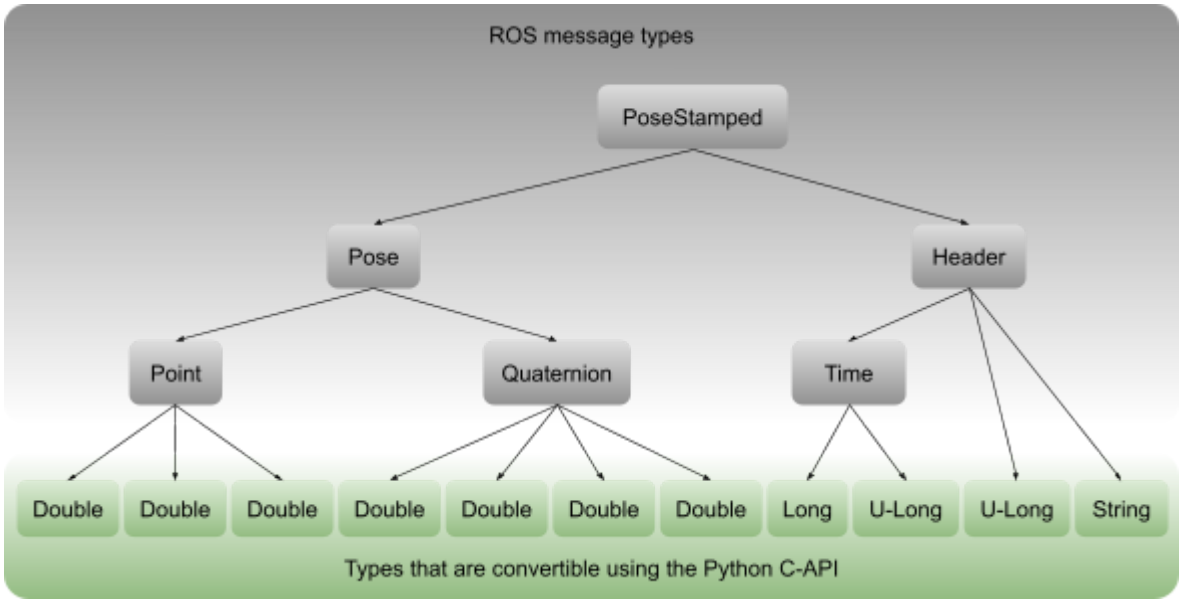

Figure 17. *Hierarchical structure of PoseStamped message type in ROS.*

The conversion for ROS message types uses a generic pattern as shown in Figure 18. Using this pattern conversions for every ROS message type are created. The "Convert Member Fields" section is implemented by Python C-API tools for low-level types like points and by other ROS message conversions for higher-level types.

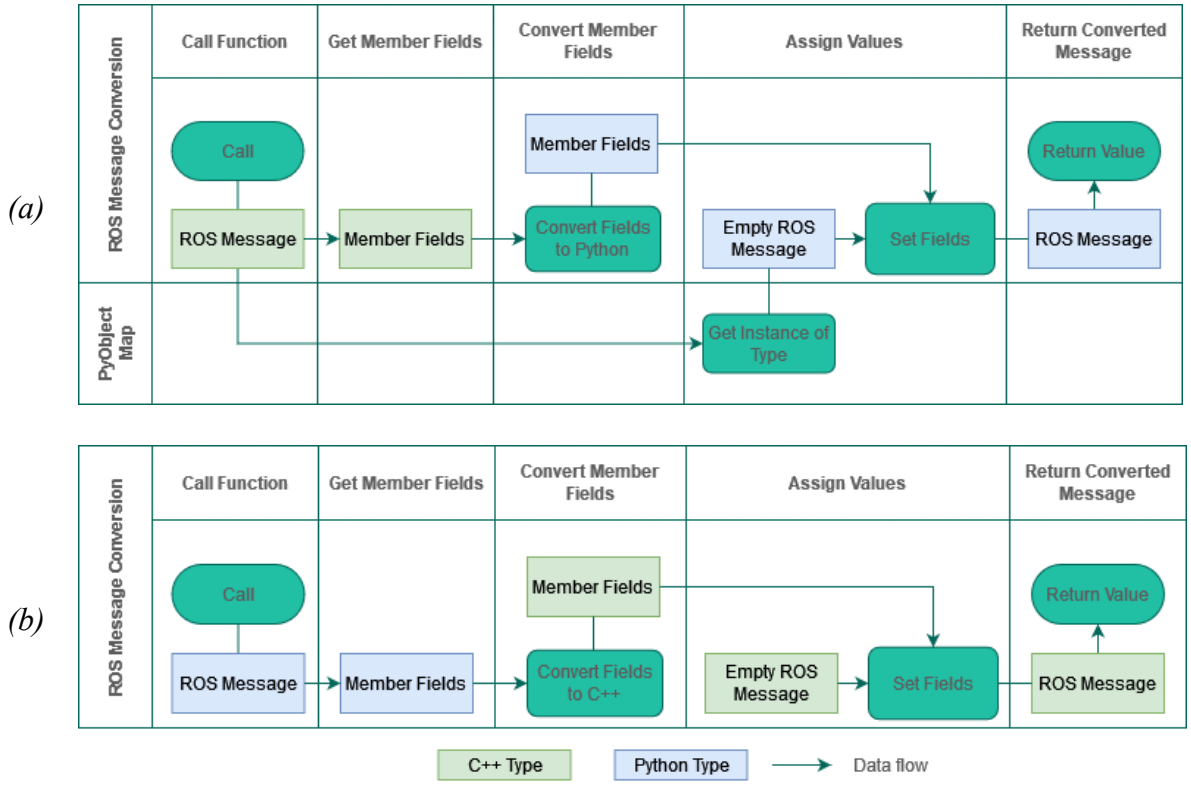

Figure 18. *ROS message conversion pattern for:*
(a) Converting from C++ to Python.
(b) Converting from Python to C++.

6 Results and Demonstration

In this chapter the implementation of the PYIF controller plugin is shown along with demonstration of use. For demonstration a local planner using artificial potential field as the strategy was implemented. The materials to reproduce this including the local planner used in the demonstration are openly available in GitHub[1].

The design of the PYIF controller requires the developer to add the name of the motion planner module along with function names to the YAML configuration file as shown in Figure 19.

```
  FollowPath:
    plugin: "nav2_pyif_controller::PYIFController"
    python_module: "python_controller.apf"
    python_delegates:
      set_plan: "setPath"
      set_speed_limit: "setSpeedLimit"
      compute_velocity_commands: "computeVelocityCommands"
```

Figure 19. *YAML file used configuring the PYIF controller.*

When the Nav 2 process is started the controller action server calls the plugin configure function in Figure 20 initialising the interpreter and importing the module and functions defined in YAML.

```
void PYIFController::configure(...)
{
  ...
  PYIF::Init({
    {python_module_, set_plan_},
    {python_module_, set_speed_limit_},
    {python_module_, compute_velocity_commands_}
  });
  ...
}
```

Figure 20. *Override of the configure function.*

The Python functions are embedded into the plugin overrides as shown in Figure 21 where the passed ROS message type argument is converted to Python type which is packed into a tuple and passed to the Python function.

[1] https://github.com/DanelLepp/nav2_pyif

```
void PYIFController::setPlan(const nav_msgs::msg::Path & path)
{
  ...
  static PyObject* pyGlobalPath;
  ...
  pyGlobalPath = PyPath_FromPath(path); // ROS message type
  ...
  PyTuple_SetItem(arguments, 0, pyGlobalPath);
  ...
  PyObject_CallObject(func_setPath, arguments);
}
```

Figure 21. *Override of the setPlan function.*

The Python C-API object calling protocol then passes the argument to the motion planner shown in Figure 22.

```
def setPath(global_plan):
  global goal_pose
  goal_pose = global_plan.poses[-1]
  global position_all
  position_all = handleGlobalPlan(global_plan)
  return
```

Figure 22. *Python-based motion planner adapter function for the PYIF.*

The latency caused by this implementation is shown in Figure 23 which shows action server plugin calls and the time spent on processing them by the PYIF and the Python-based motion planner.

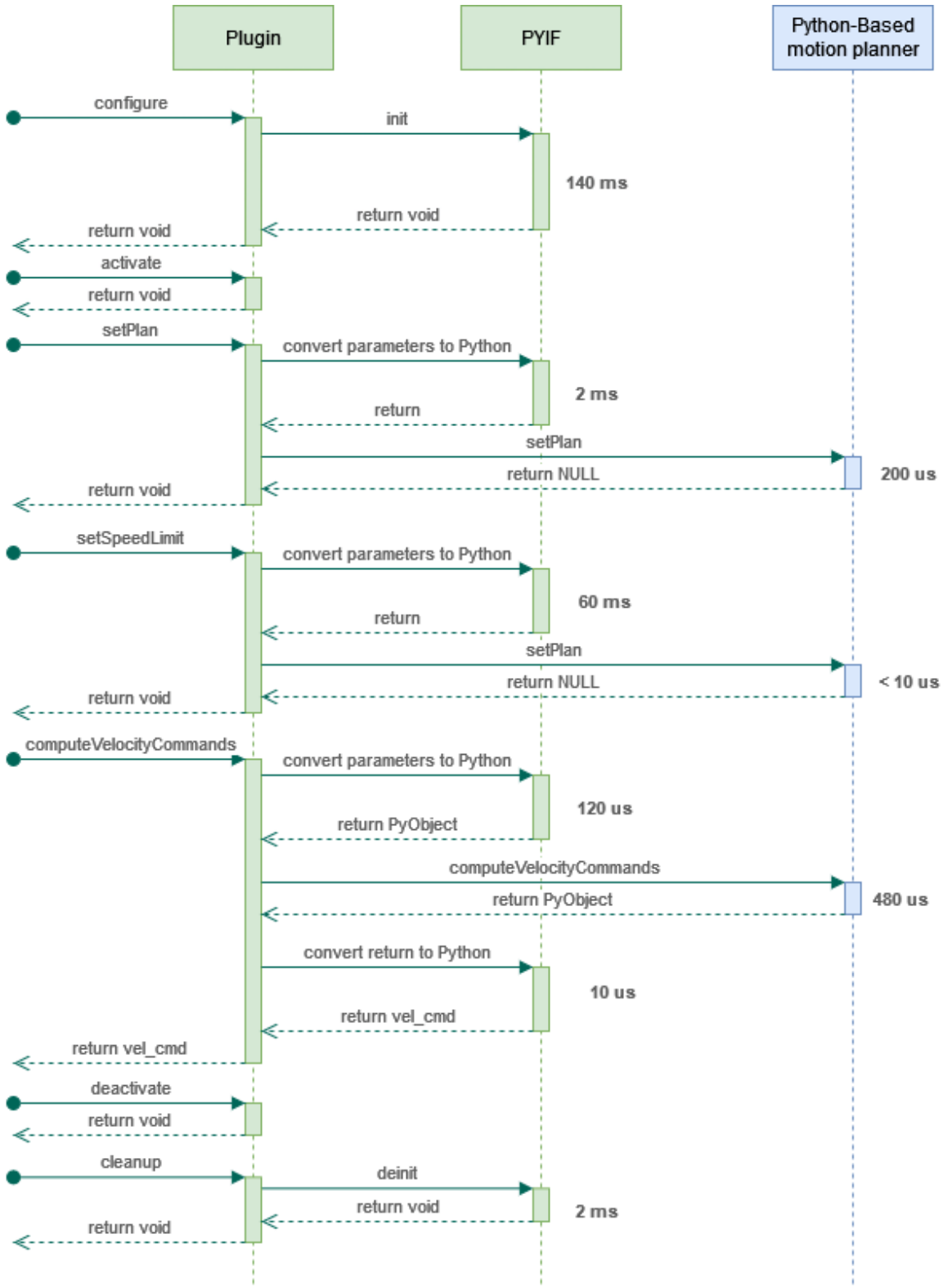

Figure 23. *Time spent on preparing data before and after the Python function calls.*

The most time consuming plugin functional calls like the configure and cleanup are only called once and therefore do not have significant impact on the performance of the plugin.

The latency caused by preparing arguments before and after the Python motion planner calls for frequently used functions such as setPlan, setSpeedLimit and computeVelocityCommands have the greatest performance impact. In this demonstration the computationally light artificial potential field planner took 480 us to calculate velocity commands compared to the total 121 us spent on ROS message type conversions. In Nav 2 local planners are commonly used with update rates around 10-20 Hz making the extra latency caused insignificant.

This demonstration was done on a Dell XPS 9570 machine with Intel i7-8750H CPU and 32 GB of 2400 MHz RAM.

7 Discussion and Future Work

The objective of this thesis was fulfilled and a well functioning PYIF controller plugin was created which allows Python-based local planners to be used in Nav 2. The solution saves valuable time for researchers, allowing them to benchmark their work on practical applications and makes bleeding-edge research simpler to implement for the community.

7.1 Limitations

Currently the plugin can only use local planners as the PYIF is not implemented for the planner server. This leaves room for future plugin implementations in the Nav 2 ecosystem.

The ROS message type conversions were written by hand although ROS uses scripts to automatically generate similar conversions from IDL files in the Python API. These conversions are used to convert between the C-types from the DDS and Python ROS message types.

7.2 Future Work

In order to scale the work of this thesis to other plugins in Nav 2 modified versions of these scripts could be implemented to add conversions between C++ and Python types. Using scripts would also make the PYIF more maintainable in case message definitions are updated.

Acknowledgments

I would like to thank

my supervisors Robert Valner, Houman Masnavi and Karl Kruusamäe for enthusiastic encouragement and useful critiques of this work,

my friends and family that always kept me going, without them this work would not have been possible.

References

[1] J. Wallén, *The History of the Industrial Robot*. Linköping University Electronic Press, 2008. Accessed: Mar. 08, 2023. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-56167

[2] S. F. Chik, C. F. Yeong, E. L. M. Su, T. Y. Lim, Y. Subramaniam, and P. J. H. Chin, 'A Review of Social-Aware Navigation Frameworks for Service Robot in Dynamic Human Environments', *J. Telecommun. Electron. Comput. Eng. JTEC*, vol. 8, no. 11, Art. no. 11, Dec. 2016.

[3] A. Stentz, 'Optimal and efficient path planning for partially-known environments', in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, May 1994, pp. 3310–3317 vol.4. doi: 10.1109/ROBOT.1994.351061.

[4] F. Rastgar, H. Masnavi, J. Shrestha, K. Kruusamäe, A. Aabloo, and A. K. Singh, 'GPU Accelerated Convex Approximations for Fast Multi-Agent Trajectory Optimization', *IEEE Robot. Autom. Lett.*, vol. 6, no. 2, pp. 3303–3310, Apr. 2021, doi: 10.1109/LRA.2021.3061398.

[5] E. Tsardoulias and P. Mitkas, 'Robotic frameworks, architectures and middleware comparison'. arXiv, Nov. 18, 2017. doi: 10.48550/arXiv.1711.06842.

[6] S. Huang and G. Dissanayake, 'Robot Localization: An Introduction', in *Wiley Encyclopedia of Electrical and Electronics Engineering*, John Wiley & Sons, Ltd, 2016, pp. 1–10. doi: 10.1002/047134608X.W8318.

[7] A. Stentz, 'Optimal and Efficient Path Planning for Unknown and Dynamic Environments', *Proc IEEE Int Conf Robot. Autom.*, vol. 10, Feb. 2003.

[8] S. Macenski, F. Martín, R. White, and J. G. Clavero, 'The Marathon 2: A Navigation System', in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2020, pp. 2718–2725. doi: 10.1109/IROS45743.2020.9341207.

[9] B. K. Patle, G. Babu L, A. Pandey, D. R. K. Parhi, and A. Jagadeesh, 'A review: On path planning strategies for navigation of mobile robot', *Def. Technol.*, vol. 15, no. 4, pp. 582–606, Aug. 2019, doi: 10.1016/j.dt.2019.04.011.

[10] M. Dakulović and I. Petrović, 'Two-way D* algorithm for path planning and replanning', *Robot. Auton. Syst.*, vol. 59, no. 5, pp. 329–342, May 2011, doi: 10.1016/j.robot.2011.02.007.

[11] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, 'The Office Marathon: Robust navigation in an indoor office environment', in *2010 IEEE International Conference on Robotics and Automation*, May 2010, pp. 300–307. doi: 10.1109/ROBOT.2010.5509725.

[12] H. Masnavi, J. Shrestha, M. Mishra, P. B. Sujit, K. Kruusamäe, and A. K. Singh, 'Visibility-Aware Navigation With Batch Projection Augmented Cross-Entropy Method Over a Learned Occlusion Cost', *IEEE Robot. Autom. Lett.*, vol. 7, no. 4, pp. 9366–9373, 2022.

[13] J. Bradbury *et al.*, 'JAX: composable transformations of Python+NumPy programs'. 2018. [Online]. Available: http://github.com/google/jax

[14] R. Okuta, Y. Unno, D. Nishino, S. Hido, and Crissman, 'CuPy : A NumPy-Compatible Library for NVIDIA GPU Calculations', 2017. Accessed: May 17, 2023. [Online]. Available: https://www.semanticscholar.org/paper/CuPy-%3A-A-NumPy-Compatible-Library-for-NVIDIA-GPU-Okuta-Unno/a59da4639436f582e483347a4833e7659fd3e598

[15] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, 'Robot Operating System 2: Design, architecture, and uses in the wild', *Sci. Robot.*, vol. 7, no. 66, p. eabm6074, May 2022, doi: 10.1126/scirobotics.abm6074.

[16] G. van Rossum and F. L. Drake, *Extending and embedding the Python interpreter*. Centrum voor Wiskunde en Informatica, 1995.
[17] G. van Rossum and F. L. Drake Jr, 'The Python/C API'. 2010.

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Danel Leppenen**,

1. grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis

Nav2 PYIF: Python-based motion planning for ROS 2 Navigation 2,

supervised by Robert Valner, Houman Masnavi and Karl Kruusamäe,

2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in points 1 and 2.
4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Tartu, **20.05.2023**