Motion Planning with OMPL

Sumeet Gyanchandani
Facoltà di Scienze Informatiche
Università della Svizzera Italiana
sumeet.gyanchandani@usi.ch

Thomas Tiotto
Facoltà di Scienze Informatiche
Università della Svizzera Italiana
thomas.francesco.tiotto@usi.ch

Bart van der Vecht
Faculty of Mathematics and Computer Science
Technische Universiteit Eindhoven
b.v.d.vecht@student.tue.nl

***Abstract*—The goal of our project was to learn to use the Open Motion Planning Library (OMPL) and use it to implement an application capable of solving a motion planning problem for two different robots and graphically display the result.**

1. Introduction

We were tasked with implementing a solver for two different non-holonomic robots using the framework provided by OMPL; the former whose kinematic model was that of a bicycle robot with the steering constrained to $+20° < \gamma < +30°$ and the latter that of a differential drive robot with a left wheel only capable of turning forwards. Our first step was to install and familiarise ourselves with OMPL, using the provided tutorials and demo code. We then proceeded to implement the main application and the two kinematic models using pure C++. Finally, the visualisations were created using OMPL.app, a Python-based GUI provided with OMPL, and MATLAB.

2. OMPL

2.1. Overview

OMPL is a powerful framework that efficiently implements many sampling-based algorithms such as PRM, RRT, EST, SBL, KPIECE, SyCLOP. The library does not explicitly represent the geometry of the workspace or the robot operating in it so we had to implement a computational representation for the robot and provide an explicit state validity checker method.

OMPL also has native benchmarking capabilities allowing the user to compare different planner side-by-side while the paths resulting from the planning can be easily exported in `.txt` format.

In the following subsections we will give an overview of the components we needed to implement to solve our task, with the code details in Section 3.

2.2. SimpleSetup

The OMPL API attempts to abstract the functionality by supplying various wrapper classes, with the most high-level being `SimpleSetup`. By using `SimpleSetup` we were required to supply the state space, state propagator, control space, state validity checker and start and goal states. If we had used geometric planning we would only have had to supply state space, state validity checker and start and goal states. We decided against using geometric planning as a robot is not a one-dimensional point that can move freely in any direction.

2.3. StateSpace

OMPL does not assume any underlying state space but lets the user code his own implementation or provides some commonly used ones: $\mathbb{R}^n$, $SO(2/3)$, $SE(2/3)$ ... A StateSpace represents the space in which the planning is performed, in our case $SE(2)$ to account for (x, y, θ).

2.4. ControlSpace

If one is planning to use a control-based planner, OMPL requires a ControlSpace object to be defined. The constructor takes the state space the controls correspond to and the dimension of the space of controls. In our case the StateSpace is $SE(2)$ and the dimensions to control are 2.

2.5. StatePropagator

Control planning also requires a StatePropagator object that returns the state obtained by applying the control to some supplied initial state. We used the `ODESolver` class, as seems to be standard practice.

2.6. StateValidityChecker

This class is tasked with evaluating a single state to determine if the configuration selected by the sampler collides with an environment obstacle or doesn't respect the constraints of the robot. `SimpleSetup` accepts either an object of type `StateValidityChecker` or a boolean `isValid` function that checks the problem constraints and returns either true or false. We decided to only implement the `isValid` function as it was all that was needed to model the robot's environment.

3. Implementation

We were initially intending to code our application in Python and work with MacOS as host operating system, as this possibility was mentioned in the OMPL Primer document, but we noticed something was wrong already when trying to install OMPL. Namely, the installation process took upwards of 6 hours on MacOS and even more on a virtualised Ubuntu environment. On further investigation we found that the problem was in `Py++`'s speed in generating the Python bindings to the C++ classes and that this was a known, unresolved, issue that dated several years back. On learning that we would have had to rebuild some of the bindings every time we compiled our application, we decided to carry out our work in the native C++ on Ubuntu.

To get a feel of how an OMPL application needed to be structured, we relied on the available demos. We had initially arranged our code over various files but, in the end, realised that the scale of the project didn't warrant the additional complexity and consolidated the (simplified) code into `launch.cpp`, `robotplanner.cpp` and `robotplanner.h`.

As we had installed OMPL, our code could be compiled independently from the core library. The core OMPL library was simply linked into the final binaries using the following command:

```
g++ launch.cpp robotplanner.cpp -lompl
    -std=c++11
```

`launch.cpp` contains various functions, described in greater detail in the following subsections.

3.1. launch.cpp

Out main file is `launch.cpp` where we take care of setting up all the pieces needed for the planning to take place. In particular, we define the environment with its constraints, create the state and control spaces that will be passed to the robot planner instance and set up all the user-defined parameters.

3.1.1. main. In the main function we create an instance of the RobotPlanner class, and call functions on it for setting up the control and state spaces, defining the obstacles, setting the planner and the start and goal states.

3.1.2. bicycleMovements. This function defines the control model for our bicycle robot. The control space consists of 2 dimensions: one for linear speed, and one for the steering angle. Given a control and a state, this function calculates the final state the robot will be in. It does it by integrating the speed over the time defined by `TIMESTEP` to calculate the new instantaneous speeds. It then calculates the new angular velocity θ by taking the tangent of the steering angle multiplied by the inter-axle distance and integrating over `TIMESTEP`.

3.1.3. thymioMovements. This method defines the control model for the thymio robot. In this case, the control space consists of 2 dimensions, one which defines the linear speed for the left wheel, and one dimension for the other. The function calculates the new state for the robot, given some initial state and a control.

3.1.4. boundingBox. We added the "inverted box" constraint that declares any state outside our problem bounds as invalid as it didn't seem to be enough to enforce these with `setBounds()` on the `StateSpace`. The documentation wasn't clear on how to address this point, but the problem we were facing where the robot would wonder off outside the bounds was solved.

3.1.5. outside_circle, outside_rect. These are two helper functions to define obstacles of circular and rectangular shape. They return true if the sampled point is outside the area defined by the obstacles and false otherwise.

3.1.6. manyObstacles and cornerObstacle. These functions have the role of state validator i.e. they must return `false` whenever the sampler selects a state that violates one of the obstacle constraints. manyObstacles uses a series of checks to create the representation of a collection of boxes and circles, studied to give a good overview of the application's workings. corner_cut_off simply places a rectangular obstacle in the top right of the state space.

3.2. robotplanner.cpp

3.2.1. RobotPlanner. The constructor creates the `StateSpace` and `ControlSpace` objects needed for planning with controls and instantiates the `SimpleSetup` object.

3.2.2. postPropagate. Standard implementation of the function.

3.2.3. setMovements. Sets up the state propagator, using the ODE solver that is passed as an argument. We use this function to specify to the planner what control model to use. For example for planning with the Thymio model we pass this function the thymioMovements function as argument.

3.2.4. setObstacles. The function receives a pointer to the obstacle function defined in `launch.cpp` and passes it to the `SimpleSetup` instance. We are declaring the state validity checker (see subsection relative to `obstacles`).

3.2.5. setResolution. The motion is discretised to some resolution (specified as a percentage of the state space) and states are checked for validity at that resolution. If the resolution at which the motions are checked for validity is too large, there may be invalid states along the motion that escape undetected. If the resolution is too small, there can be too many states to be checked along each motion, slowing down the planner significantly. Note that the number

of points generated by the planner is not important for the visualisation as we interpolate between successive states to obtain a smooth movement.

3.2.6. setSpaceBounds. Here we set the (x, y) bounds of the state space, θ is left unbounded (the state propagator enforces that it will always remain between 0 and 2*pi). Our bounds are $x \in [0, 660],\ y \in [0, -440]$. We use these for the planning problem, because these are the default bounds for the environment in OMPL.app that we use as a visualisation of the output path.

3.2.7. setControlSpaceBounds. The bounds on the control space are where the kinematic models for the bicycle and differential drive robots are implemented. For the bicycle model we constrain its speed to be between 0 and 20 and its steering angle to be between $0°$ and $+30°$. For the thymio model we use a linear speed between ± 10 for the right wheel and between -10 and 0 for the left wheel.

3.2.8. setPlanner. Here we simply tell `SimpleSetup` to use one of our control planners of choice: RRT or KPIECE.

3.2.9. setStartAndGoal. We simply tell to construct two valid states and pass them to `SimpleSetup` as Start and Goal.

3.2.10. setOptimizationObjective. If an optimisation objective is defined, we set it using `SimpleSetup`.

3.2.11. setRRTGoalBias. If we are using RRT, we set its bias towards the goal state to see the influence of its value.

3.2.12. plan. We call the `solve` function of the `SimpleSetup` that launches the planning process; we pass the timeout after which the solver stops as a parameter. The remaining code takes care of getting the solution from `ProblemDefinition`, interpolating it to have a smoother path to visualise and outputting it a series of $[x, y, \theta]$ coordinates to `path.txt`, that will be the input file for our visualisation tools.

4. Visualisation

OMPL.app builds upon the motion planning functionality in OMPL by specifying a geometric representation for the robot and its environment and provides a collision checking mechanism for the representation. Meshes representing the environment and the robot can be loaded and OMPL.app will automatically create a collision model.

We are specifically using it for its visualisation capabilities as it can load the `path.txt` file we create with our application and animate the robot in the environment or show the complete path in a single frame. An example of a complete path visualisation is shown in Fig. 1.

We also used MATLAB to implement visualisation of the robot's path as programmatically drawing the environment led to faster testing; also MATLAB gave us more flexibility in our visualisations.

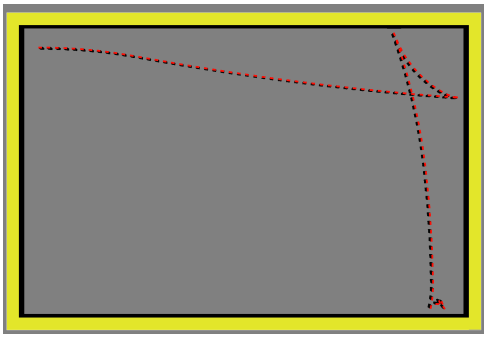

Figure 1. Example visualisation in OMPL.app.

In MATLAB we are displaying the environment with the obstacles and bounding box, the start and goal states and arrows whose orientation represents that of the robot. An example is shown in Fig. 2.

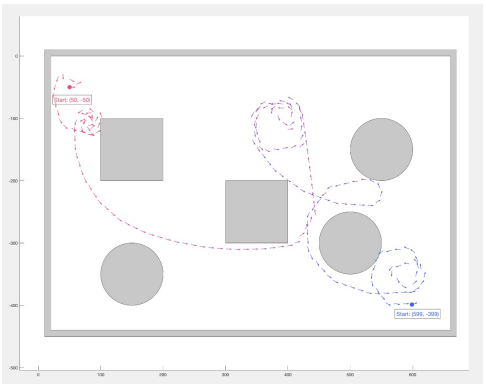

Figure 2. Example visualisation in MATLAB.

5. Testing

We decided to test our application using various environments on which we ran both our bicycle and Thymio-like robots. Each one was tested using both RRT and KPIECE to compare the two algorithms' solution paths.

5.1. Planners

The planners were configured using the parameters shown in Tab. 1 and 2.

TABLE 1. RRT PARAMETERS

timestep	10.0
resolution	0.001
planning time	20.0
goal bias	0.05/0.5
optimisation objective	path length

TABLE 2. KPIECE PARAMETERS

timestep	5.0
resolution	0.001
planning time	20.0
goal bias	0.05

5.2. Environments

The start state was always $(50, -50)$ while the goal state was set to be $(600, -400)$. An rendering of the environment is shown in Fig. 3 with an overlaid grid of $100x100$.

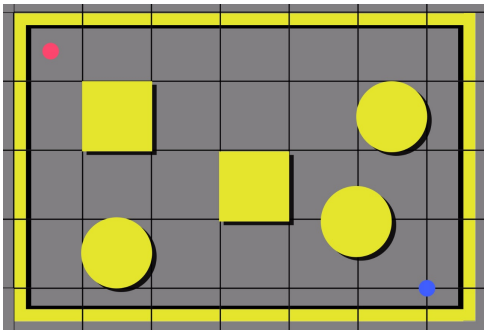

Figure 3. Rendering of the environment with start and goal states.

All our testing was carried out in a rectangular environment where the top left corner is at coordinates $(0, 0)$ while the bottom right is at $(660, -440)$; it is shown in Fig. 4.

The environment with one large obstacle was designed to easily show if our state validity checker was working properly; it consists of a large rectangle whose bottom-left corner is at $(300, -300)$; it is shown in Fig. 5.

The environment with many obstacles consists of two rectangular and three circular obstacles. The left-most rectangle has its top-left corner at $(100, -100)$ while the other at $(300, -200)$; both have a side of 100. The left-most circle is centred at $(150, -350)$, the centre one at $(500, -300)$ and the other at $(550, -150)$; all have a radius of 50. The last environment is shown in Fig. 9.

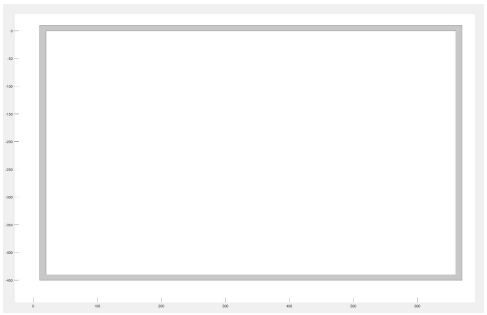

Figure 4. Empty environment.

5.3. Results

5.3.1. Bicycle model. When using the bicycle robot with the steering constrained to $+20° < \gamma < +30°$ we were unable to get any good results, even in the empty environment, as the robot was unable to make progress towards the goal at a reasonable pace. An example of this is shown in Fig. 7. For this reason all our further testing was done with a less constrained robot with the steering set to be $0° < \gamma < +30°$, meaning that the robot can only turn left, or move straight forward.

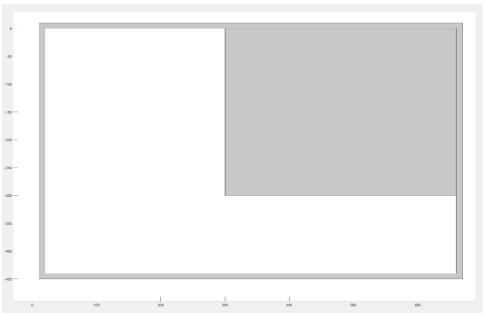

Figure 5. One obstacle environment.

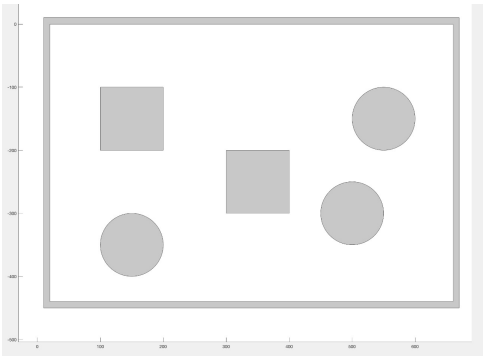

Figure 6. Many obstacle environment.

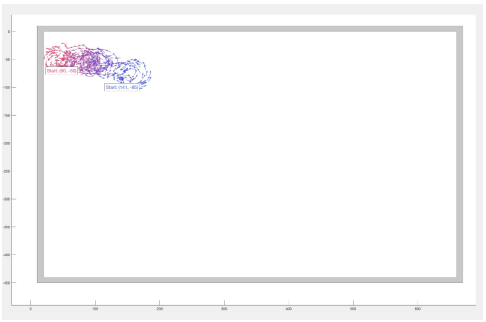

Figure 7. Bicycle model with steering constrained to $+20° < \gamma < +30°$.

We tried running our bicycle robot in the empty environment to get a feel of the quality of the solutions that RRT and KPIECE could provide. It seems that RRT converges in a more direct way towards the goal. However, both planners output multiple unnecessary cycles. This is due to the nature of the planners: e.g. if RRT samples some random state which is in the opposite direction from the goal, the robot will move in that wrong direction for a bit. Then, to fix this

direction, the robot might have to a full cycle, since it can only turn left.

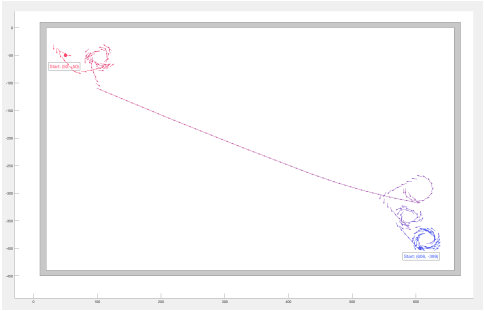

Figure 8. RRT with no obstacles and bicycle model.

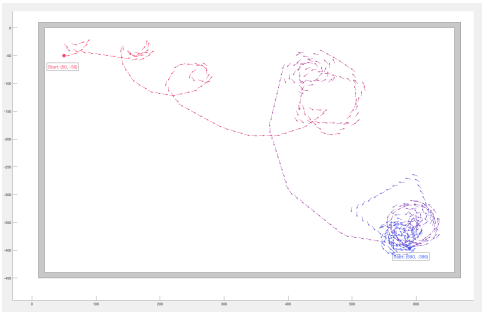

Figure 9. KPIECE with no obstacles and bicycle model.

We can see in Fig. 10 that our state validity check function is correctly working as the planner avoids the obstacle.

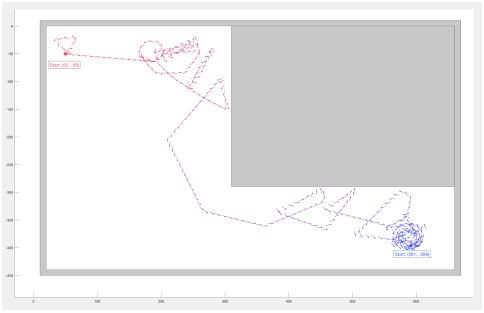

Figure 10. RRT with one obstacle and bicycle model.

In Fig. 11 and 12 we can see the different paths chosen by RRT and KPIECE, but the difference between them isn't convincing enough to declare a winner.

5.3.2. Thymio model. The Thymio, with the left wheel constrained to speeds between 0 and 10 and the right between -10 and 10 obtained better results than the bicycle. As we can see in Fig. 13 and 14, both RRT and KPIECE find much better paths towards the goal state; this is to be expected as the Thymio can turn both left and right, while the bicycle can only turn left.

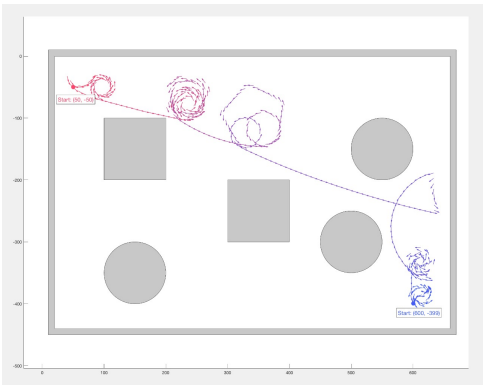

Figure 11. RRT with all obstacles and bicycle model.

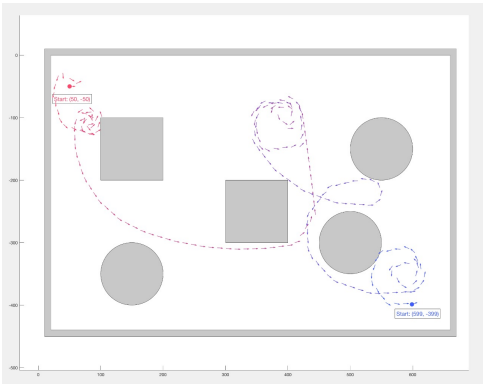

Figure 12. KPIECE with all obstacles and bicycle model.

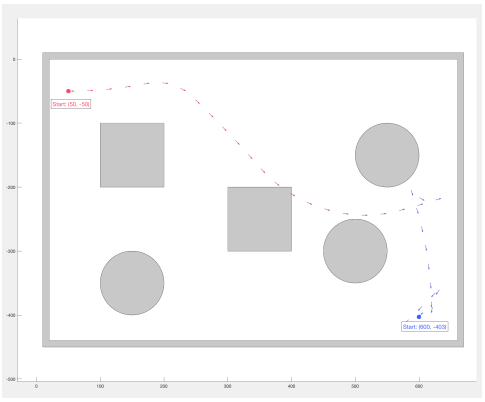

Figure 13. RRT with all obstacles and Thymio model.

5.3.3. Discussion. In all the cases when using the bicycle, the planner arrives close to the goal state but then has trouble reaching it as it tries to also orientate itself at the goal orientation of $0°$. A bicycle obviously struggles to do this, even more because of the constraints on the steering. The Thymio-like robot had no such problem because it could easily reach the goal (x, y) and then turn on itself to reach the goal θ.

We also tried to see what was the effect of chaining various parameters. Increasing the resolution (which is used

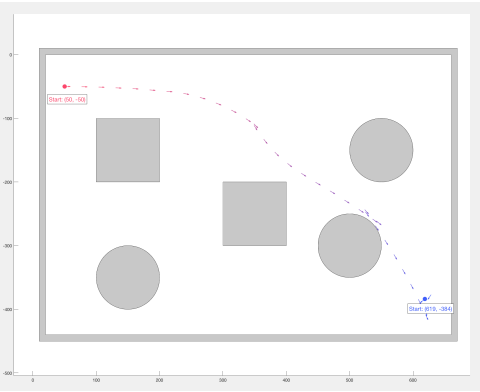

Figure 14. KPIECE with all obstacles and Thymio model.

by the motion validator) results in the robot sometimes teleporting around or moving through obstacles, while making the resolution very small leads to a very slow planner which might not even find the goal state after the given time.

Furthermore, we tried adding an `OptimizationObjective` to the planner, namely one to optimize the path length. However, we did not see any noticeable difference between the path generated with and without this objective.

Finally, we tried various values for the goal bias for the RRT planner. This value indicates how many times (on average) the planner should sample the goal state instead of some random state. Again, we did not see any noticeable difference in the output paths for these different values.

6. Issues

6.1. Incorrect bicycle kinematic model

Not having the Midterm paper at hand, we implemented our bicycle-like robot as an actual bicycle kinematic model, only to later realise that the back wheel should have also turned together with the front one. The substance of the work would not have changed as we would only have to modify the implementation of the `setControlSpaceBounds` function.

6.2. $+20° < \gamma < +30°$ bicycle doesn't reach goal

As mentioned in Sec. 5.3 the bicycle with the steering constrained to $+20° < \gamma < +30°$ was unable to make any good progress towards the goal. With a very long time allowed for the planner, it may have been able to reach it but it didn't seem to be worth it. Therefore we decides to use the bicycle model with a steering angle between 0 and 30 degrees instead.

We realised only towards the end of our work that this issue was a result of our mistake outlined in the previous point, as a robot with both back and front wheel steering would probably have been able to easily reach the goal state.

6.3. Parameter tuning

Some of the biggest issues we faced were in tuning the parameters (resolution, timestep, etc.) of the models to obtain good results. They all influence each other in some way and depended on if we were using the bicycle or the Thymio, so finding the right balance was very time-consuming.

6.3.1. Resolution. The initial value of 0.1 was too large for our problem and the robots were "teleporting" around the environment. In the end, we obtained the best results with a value of 0.001 for both the bicycle and the Thymio.

6.3.2. Timestep. A timestep of 10, instead of the default of 1, was empirically found to work best with our models and values for other parameters.

6.3.3. Planning time. The Thymio was initially having trouble reaching the goal state in time, also because of the too high resolution. To remedy this we increased the timeout to 30 seconds but later realised that, after having changed the other parameters, the original 20 seconds were sufficient.

6.4. Cutting corners

We are not quite sure of the reason, but some instances of our testing showed that the path sometimes cut a corner off an obstacle. We believe that the reason for this is in the parameter tuning, as detailed in the previous point, in particular the step size and resolution. It seems as the state motion validator is failing at some point or that the robot "teleported" to the other side of an obstacle and then its path was interpolated passing through it.

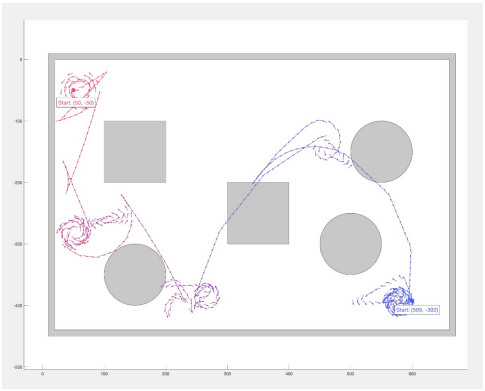

Figure 15. The bicycle model cutting some corners.

7. Conclusion

OMPL is certainly a very powerful framework and this, together with its modularity, is certainly why it has become the de-facto backend in all major robotics frameworks. Unfortunately, its modularity and open-source nature also make

it quite hard to directly work with especially because the documentation of the API and the code examples are somewhat lacking. A quite large amount of work is needed to get good results, especially because of the parameter tuning and the lack of visual references. The bundled OMPL.app is a nice tool that builds on the core OMPl library and makes it easy to plan and visualise in complex environments.