Control Algorithms for Real-Time Motion Planning with Dynamic Obstacles

Tianyi Gu

Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA

Abstract

In this paper, we design two control algorithms: sampling based model–predictive control (SBMPC) and bisection search based model–predictive control (BBMPC). The algorithms are implemented as the controller for a real-time planning system in ROS to enable a Pioneer robot to move quickly in environments with dynamic obstacles. The behaviors of both algorithms are demonstrated through straight and curve line following experiments from simulation and real–world environments. We also discussed several issues of the real-time planning system.

Introduction

Our project is a real-time planning system in ROS to enable a Pioneer robot to move quickly in environments with dynamic obstacles. This requires the development and integration of state-of-the-art technologies in planning, control, and sensing. The focus of this paper is on the control subsystem (or "controller"). In the big picture, given the latest obstacles information and vehicle states, the motion planner computes an action that can achieve a feasible trajectory and a temporal goal point within a certain time bound. The action and the temporal goal are then sent to the controller, which interfaces directly to the vehicle, and is responsible for the execution of the motion plan.

The main challenge for autonomous vehicles moving through dynamic obstacles such as humans and other vehicles is that it often need to do complex online replanning with high frequency. It is important to ensure all modules are efficient enough, as on-board computational resources are typically limited and fast planning rates are often required. In this paper, we present two algorithms for implementing a model-predictive control. Our first algorithm, a sampling based model-predictive control (SBMPC), is able to efficiently find a control that result in a state that is close to the goal. Our second algorithm, bisection search based model-predictive control (BBMPC), performs a local search to systematically find the optimal control that can result in a state closest to the goal. This paper reports on the design and implementation of these two controllers, including how they works in the on-board planning system, for our Pioneer project. The effectiveness of the controller algorithms and the system is discussed, based on the experimental results from both simulation and the Pioneer 3–DX robot.

Related Work

There has been much related work on re-planning algorithms for motion planning with dynamic obstacles. Some of them are focusing on real-world environment, meaning trying to deploy a system on the robot, while others are working on simulation. Since our work is built on both of this two type, we introduce several previous works of each type. We also introduce several related control algorithms here.

Re-planning algorithms

ARA* (Likhachev, Gordon, and Thrun 2003) is an anytime A* algorithm that can reuse as much as possible of the results from previous searches so that it can save considerable computational effort. The work is based on the idea of improving the solution quality by keep decreasing inflation factors. ARA* speeds up by not re-computing the g value that have been correctly computed in the previous iterations. The way to employ ARA* in real-time online planning is to root the search tree at the goal state and expand the tree toward the robot current state so that one can re-use the search tree. To plan on-board, it require accurate prediction of the robot's future state.

To deal with the issue of moving obstacles, two real-time search algorithm RTR* and PLRTA* (Cannon, Rose, and Ruml 2014) incorporate time as part of the state space. RTR* is an extension of R* (Likhachev and Stentz 2008), and PLRTA* is an extension of LSS-LRTA* (Koenig and Sun 2009).They made five major changes to transform R* into RTR*. The modification to LSS-LRTA* is partition g and h values into $static$ and $dynamic$ portions, and then employ different learning strategy on each portion.

Multipartite RRT(MP-RRT) (Zucker, Kuffner, and Branicky 2007) is another real-time search algorithm to deal with dynamic obstacles. It combines the strengths of ERRT, DRRT and RFF. In each planning iteration, they first clean the motion tree by remove the invalid tree nodes and save unconnected subtrees into a stand-by forest. Then a random state is sampling by biasing by the nearest tree in forest or the goal. If the new sampled state is in one of the subtree, then it will try to connect the nearest state on current motion tree to that sample state by solving a boundry value problem, else, it will do conventional RRT extend routine.

D* Lite (Koenig and Likhachev 2005) is a fast re-planning algorithm that builds on LPA* (Koenig, Likhachev, and others 2002), which is an incremental heuristic search method that repeatedly determines shortest paths between two given vertices as the edge costs of a graph change. Both of the first and second version of D* Lite searches from the goal vertex to the start vertex and only update those vertices that are locally inconsistent in the priority queue, while the second version save a lot effort on reordering the priority queue by maintain lower bounds on the keys of vertices. D* Lite achieve the same path as (Focussed) D*, but they are algorithmically different.

A Variable Level-Of-Detail(VLOD) (Zickler and Veloso 2010) algorithm ignores far-future dynamic obstacles to speed up its search. The algorithm is based on RRT (LaValle and Kuffner 2001). A pre-defined *LOD-horizon* is set by user as a time threshold to ignore obstacles. The replanning interval is set by user as well , which depends on the expected domain uncertainty. A "Hallway" domain and another more complex "Maze" domain are used to test the VLOD planner.

On-board Motion Planning System

Multi-resolution lattice state space (Likhachev and Ferguson 2009) is used to solve complex dynamically feasible maneuvers for autonomous vehicles. The configuration space is discretized into a multi-resolution lattice state space. Then the Anytime Dynamic A* algorithm is employed with a combined heuristic function, consisting of freespace heuristic and 2D heuristic, to search the lattice graph backwards from the goal configuration towards the current configuration of the vehicle. They test this approach on a real world vehicle, and the results show that the multi-resolution lattice is faster than a high-resolution lattice as well as guarantee the solution quality. However, the anytime algorithm ADA* does not perform well with dynamic obstacles.

A time–bounded lattice data structure (Kushleyev and Likhachev 2009) is proposed to ensure the real-time performance of online planning. For each obstacle, they first compute a time bound that ensure the error of its position prediction is still within some tolerance. Then the maximum time bound among all obstacles is used to construct a lattice graph, in which all states are six–dimensional (x, y, θ, v, w, t). As soon as a state s has a time stamp later than the time bound, it is projected onto a 2D grid in which all states are two-demensional (x, y) for fast search. A weighted A* is used to search for lattice graph, the heuristic is computed by Dijkstra search in 2D state space. Their experiments are doing both in simulation and real world robots.

HDRC3 (Doherty et al. 2015) is an architecture for Unmanned Aircraft Systems and also can be applied on any other autonomous robotic systems. It has three main layer: control layer, reactive layer, and deliberative layer. The middle ware infrastructure is called DyKnow that is used to create and manage the data flow, which is using ROS. The Temporal Action Logic is used for both specification and on-line monitoring purposes. The collaborative systems is shortly discussed.

Control Algorithms

The closed-loop RRT (CL-RRT) (Kuwata et al. 2009) extends the RRT by making use of a low-level controller and planning over the closed-loop dynamics. The low-level controller takes a lower dimension reference command and runs a forward simulation using a vehicle model and the controller to compute the predicted state. They employ the pure-pursuit controller as the steering control as well as PI control as the speed control. In their application, the controller is used in two different ways. One is in the closed-loop prediction, and the other is the execution of the motion plan in real time.

The model predictive controller (Howard et al. 2014) is a model-based approach to effectively find best control action that take the robot to the goal. This approach is particularly important as the robot move into unstructured domains where the mapping between control inputs and robot motion can be nontrivial. As it is difficult to generate trajectories by inverting the nonlinear, coupled equations of robot motion with limited computational resources, efficient sampling techniques must be developed to achieve the real-time performance.

Another choice is bisection search (Press et al. 1982), which is a simple local search approach that can quickly locate a local optimal solution.

Implementation of the Controller

The big picture is that in each search–execution iteration, the online real–time planner computes an action that can achieve a feasible trajectory and a temporal goal point within a certain time bound, given the motion primitives. The action and the temporal goal are then sent to the controller. The controller then generate a sequence of control command (linear velocity v and angular velocity w) by forward simulation following the model of the robot, so that the robot can achieve the temporary goal as much as possible. In this section, we explain how we design the motion primitives and the detail of the model predictive controller.

Motion Primitives

The motion primitives are defined here to be the actions that connect states in the state space graph and that are feasible motions. It is also the reference information that the planner send to the controller, along with the temporal goal state.

First, we define 3 linear accelerations to apply: FA(fast acceleration), SA(slow acceleration), H(hold current speed), FD(fast deceleration), SD(slow deceleration). Second, we define 5 angular acceleration to apply: HL(Hard Left), SL(Soft Left) and HR(Hard Right), SR(Soft Right) and N(No turn).Therefore, we have totally 25 motion primitives for each state. Here, we give a demonstration of how to get the control parameters ω and v. Let's say the robot is currently in a state with the linear velocity v_c and angular velocity ω_c, and we want to get the two parameters of the motion primitive that consist of linear acceleration A along with

SL	$\pi/8$	rad/s^2
SR	$-\pi/8$	rad/s^2
HL	$\pi/4$	rad/s^2
HR	$-\pi/4$	rad/s^2
N	0	rad/s^2
FA	300	mm/s^2
SA	150	mm/s^2
H	0	mm/s^2
FD	-300	mm/s^2
SD	-150	mm/s^2
δt	250	millisecond

Figure 1: Parameters for motion primitives generation.

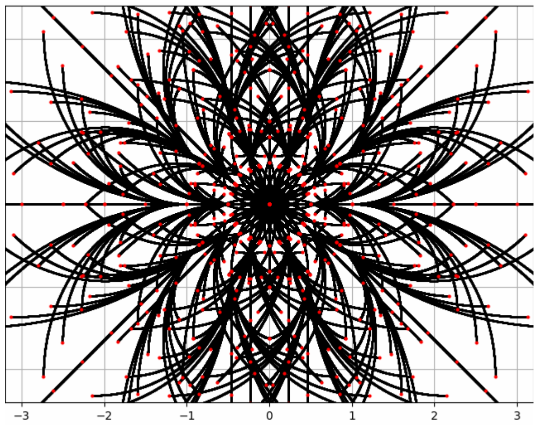


Figure 2: The example of the motion primitives. The red dots are the states; the black lines/curves are trajectories between states.

angular acceleration SL. To get the velocity v, we need one additional parameters v_{max} (which is the maximum velocity of the robot, for pioneer, it is $1.2m/s$) . Then we can get v by the following formula.

$$v = \min(v_c + 300 \times \delta t, v_{max})$$

We can get angular velocity by the following formula.

$$\omega = \omega_c + (\pi/8) \times \delta t$$

Analogously, we then can get the control parameters of all other motion primitives. With the ω and v on hand, we now can simulate the position of the robot in next state by applying the following formula. Figure 2 gives a example of the simulated states(red dots) along with its motion primitives(black curve).

$$\dot{x} = v \cos \Phi$$
$$\dot{y} = v \sin \Phi$$
$$\dot{\Phi} = \omega$$

In our experiments, We find the rotation speed of Pioneer is $100°/s$; this will be the limit of the hard turns. Because of the safety reason, we forbid the hard-turn motion primitive in high speed states.

Model Predictive Controller

To achieve the real–time performance, we employ the robot's motion model to predict the end states given different control inputs. Then we search for the best control input by comparing the distance between all end states and goal state. Finally, we publish this best control to the robot. In the rest of this section, we first describe our design of distance function. Afterward, we explain the details of two approaches that we implement the model predictive controller.

Distance Function

In our application, the robot states are designed as five dimensions (x, y, θ, s, t), where x and y are the position of the robot , θ is the heading angle, s is the speed, and t is the associated time stamp. To measure the distance between two five-dimension states, we need to normalize all the dimensions. Therefore, we define three unit quantities for distance, heading, and speed respectively. Every unit is equally important for the measurement. For example, in our application, we define $1cm$ as one unit distance, $3°$ as one unit heading, $0.15m/s$ as one unit speed. Therefore, $1cm$ offset on position is as important as $3°$ deflection on heading or $0.15m/s$ deviation on speed. Now we can measure the distance between two states–that have the same time stamp–by the following equation. Given two states $(x_1, y_1, \theta_1, s_1, t_1)$ and $(x_2, y_2, \theta_2, s_2, t_2)$ where $t_1 = t_2$, the distance d then can be computed by:

$$d = \frac{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}{unit_{dis}} + \frac{|\theta_1 - \theta_2|}{unit_h} + \frac{|s_1 - s_2|}{unit_s}$$

For instance, in our application, two states with $6cm$ offset on position and $6°$ deflection on heading result in 8 units distance.

Sampling Based Approach

To reduce the search space for the planner, we discretize the control space into several motion primitives. However, to best achieve the goal state, the controller should be able to take any action in the continuous feasible control space (given the associated motion primitive as a reference control). As it is difficult to find the optimal control among the huge continuous control space by applying the nonlinear, coupled equations of robot motion with limited computational resources and time, we investigate two approaches to achieve the real–time performance in our application: sampling based approach and local search based approach. In the first case, we use random sampling techniques to sample a great amount of controls (200 in our application) among all the feasible controls and then choose the best one.

Figure 3 shows the main steps in the sampling based model–predictive control (SBMPC). SBMPC first generate random controls by reference control (line 1) (in the experiment section below, we employ both normal and uniform distribution randomizing technique in this step), initialize best control as none (line 2), and initialize best distance as ∞ (line 3), forward simulate the sample control (line 6), compute distance between the end state and the goal state (line 7). Line 8– 11 ensure the best control can be selected

```
SBMPC($refAction, start, goal$)
 1: $controls$ ←RandomSampling($refAction$)
 2: $c_{best} \leftarrow none$
 3: $d \leftarrow \infty$
 4: $\delta t \leftarrow goal.t - start.t$
 5: for $c$ in $controls$ do
 6:     $end$ ←Propagate($c$, $start$, $\delta t$)
 7:     $d_{cur}$ ←Distance($end$, $goal$)
 8:     if $d_{cur} < d$ then
 9:         $d \leftarrow d_{cur}$
10:         $c_{best} \leftarrow c$
11:     end if
12: end for
```

Figure 3: Pseudocode for sampling based model predictive control.

and afterward published to the robot.

Bisection Search Based Approach

In previous subsection, we develop SBMPC by employing random sampling techniques to achieve a good control with respecting to the efficiency. In this subsection, we pursue another approach: bisection search based model–predictive control (BBMPC), which adapt a local search algorithm, bisection search, to the problem of searching for a good control among all feasible controls.

In our implementation, a control is a two–dimension vector that consist of linear velocity v and angular velocity ω. Given two initial controls, we then use bisection search move along the first dimension to its optimum, then from there along the second dimension to its optimum, cycling through both of the directions as many times as necessary, until the distance function stops decreasing.

Figure 4 shows how we do bisection search on both linear velocity and angular velocity. We first start from two initial controls; one is the reference control that given by the planner; another one is the mean of reference control and the robot's maximum velocities (line 13- 16). Then we do bisection search one dimension a time (line 23- 31), until the distance function stop decreasing (line 34- 39).

Figure 5 and Figure 6 are the two main steps of bisection search algorithm. Given two initial value, the first step is to bracket a local minimum: (1) try guess c_m in the middle (line 41); (2) if c_l is the smallest, move the right to middle (line 46), move middle to left (line 47) and move left to further left by original $r - l$ to double the bracket range; (3) if c_r is the smallest, shift all the three points right (line 67- 69); (4) if c_m is the smallest, local minimum is between left and right point.

The second step is to refine the estimate: (1) try lm between left and middle (line 57); (2) if lm is smaller than middle, shift right to middle and shift middle to lm (line 62-65); (3) otherwise try mr between middle and right (line 58); (4) if mr is smaller than middle, shift left to middle and shift middle to mr; (5) otherwise middle is the smallest, we shift left to lm and right to mr; (6) we keep refine until range between left and right is small enough or the improvement is not significant enough (line 56).

```
BBMPC($refAction, start, goal, v_{max}, \omega_{max},$
      $minRange, minDeltaDis$)
13: $v_l \leftarrow refAction.v$
14: $v_r \leftarrow (refAction.v + v_{max})/2$
15: $\omega_l \leftarrow refAction.\omega$
16: $\omega_r \leftarrow (refAction.\omega + \omega_{max})/2$
17: $v_m \leftarrow none$
18: $\omega_m \leftarrow none$
19: $c_{best} \leftarrow none$
20: $d \leftarrow \infty$
21: $optimize\_v$ ←true
22: while true do
23:     if $optimize\_v$ then
24:         $v_l, v_m, v_r$ ←Bracket($v_l$, $v_r$, $start$, $goal$)
25:         $v_l, v_m, v_r$ ←Refine($v_l$, $v_m$, $v_r$, $start$, $goal$,
                    $minRange$, $minDeltaDis$)
26:         $optimize\_v$ ←false
27:     else
28:         $\omega_l, \omega_m, \omega_r$ ←Bracket($\omega_l$, $\omega_r$, $start$, $goal$)
29:         $\omega_l, \omega_m, \omega_r$ ←Refine($\omega_l$, $\omega_m$, $\omega_r$, $start$, $goal$,
                    $minRange$, $minDeltaDis$)
30:         $optimize\_v$ ←true
31:     end if
32:     $c_{cur} \leftarrow (v_m, \omega_m)$
33:     $d_{cur}$ ←PropagateAndDistance($c_{cur}$, $start$, $goal$)
34:     if $d_{cur} < d$ then
35:         $d \leftarrow d_{cur}$
36:         $c_{best} \leftarrow c_{cur}$
37:     else
38:         break
39:     end if
40: end while
```

Figure 4: Pseudocode for bisection search based model predictive control.

Experiments

This section presents the application results of model–predictive control algorithms on a Pioneer 3-DX robot (Figure 7), both in simulation and real–world environments. We use the differential drive model (described in section 3) to predict the robot's future position. The control inputs to this system are the linear velocity v and angular velocity ω.

Figure 8 shows a snapshot of the hallway environment and the air view in Rviz (a visualization tool). The robot is in the lower middle of the figure. The purple line represent the plan.

The controller module use the Robot Operating System (ROS) to communicate with the robot hardware and other modules. In each plan–execute iteration, the planner send a reference motion to the executive (a central module to coordinate all the modules); the executive then project the robot current state to a target state by simulate the reference motion and send the target state to the controller along with the reference motion. In our application, the time duration T

```
Bracket($c_l, c_r, start, goal$)
41: $c_m \leftarrow (c_l + c_r)/2$
42: $d_l \leftarrow$PropagateAndDistance($c_l$, $start$, $goal$)
43: $d_m \leftarrow$PropagateAndDistance($c_m$, $start$, $goal$)
44: $d_r \leftarrow$PropagateAndDistance($c_r$, $start$, $goal$)
45: if $d_l$ is the smallest then
46:     $c_r \leftarrow c_m$
47:     $c_m \leftarrow c_l$
48:     $c_l \leftarrow 2c_l - c_r$
49: else if $d_r$ is the smallest then
50:     $c_l \leftarrow c_m$
51:     $c_m \leftarrow c_r$
52:     $c_r \leftarrow 2c_r - c_l$
53: end if
54: return $c_l$, $c_m$, $c_r$
```

Figure 5: Pseudocode for bracket a local minimum control in one dimension.

of the plan–execute is 250 millisecond. Figure 9 shows the main loop how controller works. When it receives an target state, the controller start to run at 60Hz and keep searching (line 79) and publishing (line 80) the best control to the robot until the time duration is end (line 77). If it does not receive a new target state when the time duration is end (line 76), it will trigger the estop to emergency stop the robot (line 83). In line 78, the controller listen to the $amcl$ and $rosaria$ topics, which are published by the robot's hardware platform, to update the robot's position, heading and velocity. Figure 10 shows the architecture that the controller communicate with other modules.

Straight Line Results

The following subsections present results from simulation and real–world experiments. In simulation, the system was running on two quad-core 3.5GHz Intel Xeon processors. In real–world experiments, the Pioneer robot is connect to a laptop with a quad-core 3.7GHz Intel processor. In both environments, the controller takes one core and runs at approximately 60Hz.

We first test SBMCP and BBMCP to follow a sequence of straight line goal states. To make a simple test case, we turn off the planner and hard code the straight line plan in the executive. For SBMCP, we test with random sampling both from uniform and Gaussian distribution. The range for uniform random sampling is $[0, 3]$ for v and $[-1.7, 1.7]$ for ω. This setting enable the robot can choose any control command within its limit. The mean for Gaussian random sampling is the reference motion, and we set the standard deviation as 0.1 m/s for v and 0.01 $radius/s$ for $omega$. Figure 11 shows how these three control algorithm behave. The green line is a path of sequence of goal states. The red line is the trajectory of the robot. As we can see, SBMPC with Gaussian random sampling performs best, SBMPC with uniform sampling is the worst, and BBMPC is in between. Figure 12 gives the distance from the end state to the goal state, for every published control from the first to last.

The reason uniform sampling performs bad is because it get wide spread samples, a lot of which are useless controls. While Gaussian sampling technique can focus on controls around the reference motion given by the planner. BBMPC is also able to quickly locate a good control and ignore useless controls. Notice that, in this test, we continually publish the target states without checking the current state of the robot. This means even if the robot is on a wrong direction or far away from the position it should be, it will still receive a target state in the original plan. Considering this, the error will be boosted after the robot take the first wrong control.

We also test straight line following in the real–world environment. Figure 13 shows the behaviors of two sampling based model predictive controller. Here both target path and robot's trajectory are painted in purple. As we can see, the uniform sampling approach can't follow the line can hit the wall but Gaussian sampling works well.

```
Refine($c_l, c_m, c_r, start, goal, minRange, minDeltaDis$)
55: $\delta d \leftarrow \infty$
56: while $|c_l - c_r| > minRange$ and
          $\delta d > minDeltaDis$ do
57:     $c_{lm} \leftarrow (c_l + c_m)/2$
58:     $c_{mr} \leftarrow (c_m + c_r)/2$
59:     $d_{lm} \leftarrow$PropagateAndDistance($c_{lm}$, $start$, $goal$)
60:     $d_{mr} \leftarrow$PropagateAndDistance($c_{mr}$, $start$, $goal$)
61:     $d_m \leftarrow$PropagateAndDistance($c_m$, $start$, $goal$)
62:     if $d_{lm} < d_m$ then
63:         $c_r \leftarrow c_m$
64:         $c_m \leftarrow c_{lm}$
65:         $\delta d \leftarrow d_l - d_{lm}$
66:     else if $d_{mr} < d_m$ then
67:         $c_l \leftarrow c_m$
68:         $c_m \leftarrow c_{mr}$
69:         $\delta d \leftarrow d_m - d_{mr}$
70:     else
71:         $c_l \leftarrow c_{lm}$
72:         $c_r \leftarrow c_{mr}$
73:     end if
74: end while
75: return $c_l$, $c_m$, $c_r$
```

Figure 6: Pseudocode for refine estimate a local minimum control in one dimension.

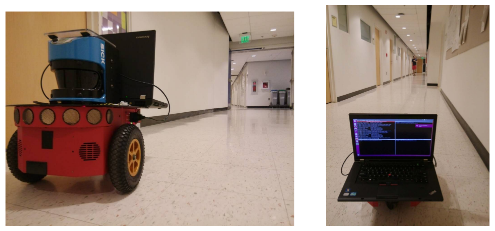


Figure 7: Pioneer 3-DX robot.

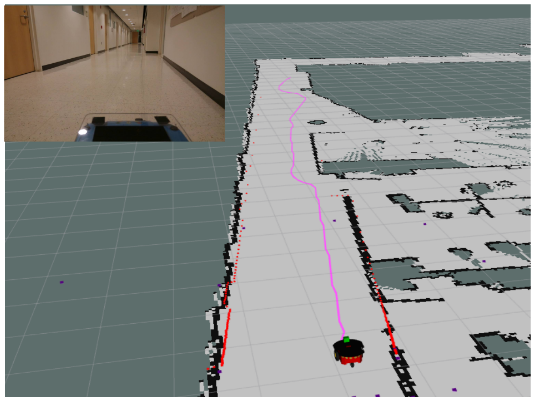


Figure 8: Lane following in the hallway environment.

```
Controller()
76: while received a new target state do
77:     while time duration is not end do
78:         update current state of robot
79:         run SBMCP or BBMCP to find the best control
80:         publish the best control to the robot
81:     end while
82: end while
83: trigger estop
```

Figure 9: Pseudocode for controller main steps.

Curved Path Results

To test curve line following scenarios in the simulation environment, we set the start position at bottom left of the map and goal position at upper right. The dark gray areas in Figure 14 are walls; the robot can only move in light gray areas. The green line is the path published by executive module that the robot need to follow, and the red line is the trajectory of the robot. All three algorithm follow closely with the target curve and take the robot to the goal position. Figure 15 gives the distance measurement from the end state to the goal state, for every published control from the first to last. All of the three algorithms keep their distance function value at a low level (about 10 units).

The reason all three algorithm behave better in curve line test is that the projection step in executive always update the robot's current position to the planner, so it can always re–plan from a correct position and gives a close target to the controller, which makes controller easy to follow it.

We also do curve–line–following test in the real–world environment. Figure 16 shows the behaviors of the sampling based model predictive controller. Here both target path and robot's trajectory are painted in purple. As we can see, the controller is able to follow the curve closely.

Discussion

The ultimate purpose of this project is to to enable a Pioneer robot to move quickly in environments with dynamic obstacles. However, as we do the integral test – together with

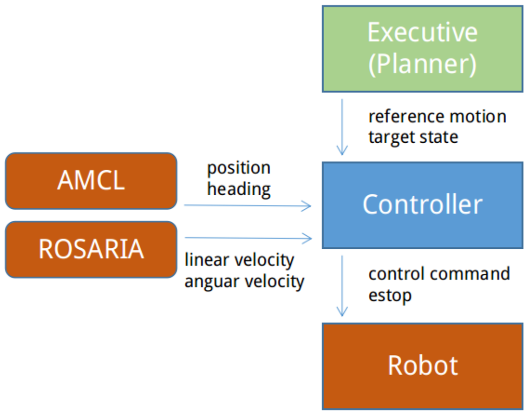


Figure 10: The architecture of the controller module.

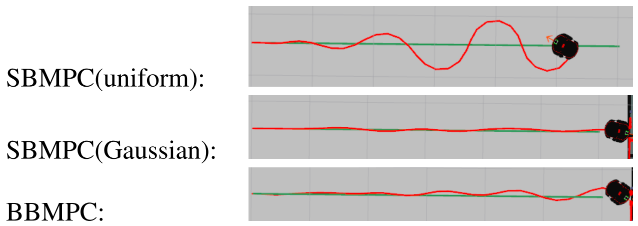


Figure 11: Controller behavior when following a sequence of straight line target.

planning , control, and sensing – several issues emerge and fail the robot to move as what it suppose to do. For example, both mission in Figure 17 are failed; the robot move too close to the wall so that the estop was triggered.

Low AMCL Rate

In the bottom case in Figure 17, we turn off the projection function in the executive, which make the controller to follow a one-shot plan. The mission is failed; the controller is not able to bring the robot back to the plan path when it take a wrong action and end up with run toward the wall. We find out that the robot is heavily rely on the executive to update the robot's position to the planner. However this projection function is designed not to correct the controller's error but for dynamic obstacles.

By looking into the controller log file, we find the controller published some extreme turning and velocity control. It turns out this is cause by incorrect state data that received from AMCL. Although AMCL is run at 60Hz, but the data is actually updated at 10Hz. By further investigate, we find our AMCL rate is limited by the laser rate. Therefore we speed up the laser rate to 60Hz and the system works better in the simulation. However, we are still not able to make the laser faster for the real robot. Figure 18 shows the data collected as the robot is running. We can see the heading data is not continually updated in real–world experiment.

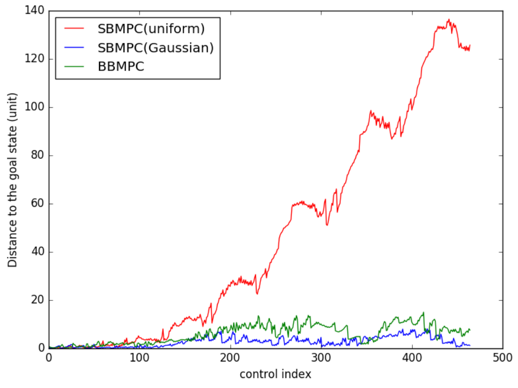


Figure 12: Distance from the end state to the goal state in straight line following test.

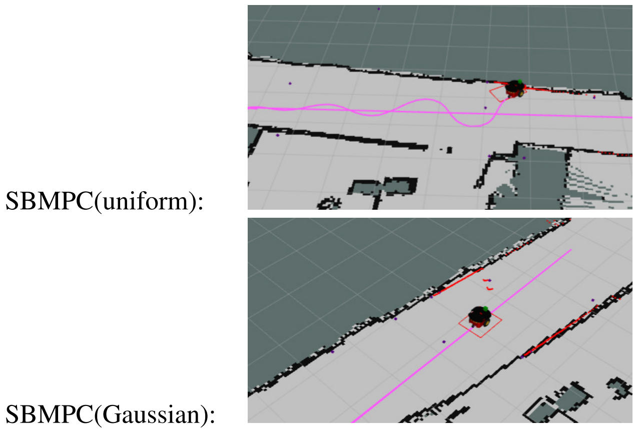


Figure 13: Controller behavior when following a sequence of straight line target in real–world environment.

Under and Over Shooting

The controller also face under and over shooting problems when it try to follow a target state. At the start stage of the project, the robot was always behind the given targets, especially at the beginning that the robot suppose start to move, but it didn't. It turns out a parameter named "latch" in the ROS publisher need to be set as true to take the robot to move with only issue one control command. Otherwise it won't move until it receives a sequence of commands.

The over shooting issue is the scenario that the robot is on or over the target state earlier than ETA. This can cause the robot to take a random control. For example, if the robot is on the target state with some linear velocity, then the best end state is the not to move but to stay in its position. However, the robot has inertia, so it can't stop immediately. Then according to the distance function, a turn–and–slow–down control could result in a same distance as a slow–down control. So the algorithm may result in any "best" control that has the same distance function value.

We now fix the over shooting issue by publish a stop control command when the current state is too close to the target state. It prevent the robot to choose a random "best" control,

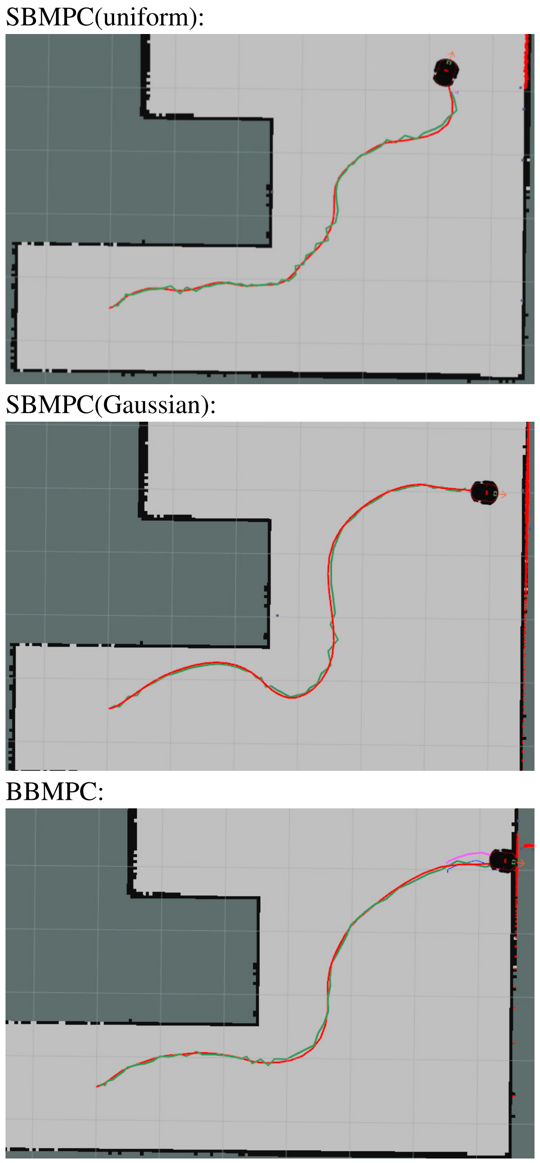


Figure 14: Controller behavior when following a sequence of curve line target.

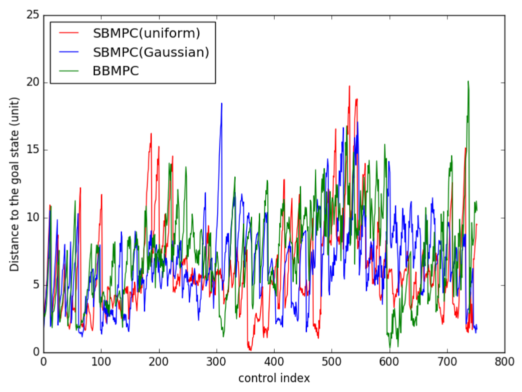


Figure 15: The distance from the end state to the goal state in curve line following test

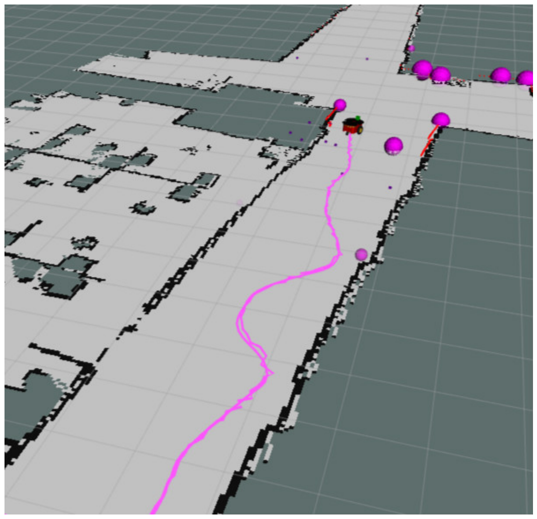


Figure 16: Controller behavior when following a sequence of curve line target in real–world environment

but this still not perfectly solve the problem. We are now considering another solution that make the planner send two target states to the controller so that it has more information to support its reasoning about what to do in over shooting scenario.

Inaccurate Robot Motion Model

As the project going, we find out that the robot is not able to closely follow the original plan path. In Figure 19, the blue line in the left is the original plan for that test, and the red line in the right is the robot's trajectory; we can clearly see they are not the same. Therefore, we design several experiments to test how accurate is the robot motion model. We publish a single control to the robot for a certain time period: 1 second or 5 seconds, and see how the robot achieve that command. Figure 20 shows the results in simulation. As we increase the linear velocity value, the robot lose its ability to turn. This is quite different from our robot model. We also find out the maximum angular acceleration is about 2.4

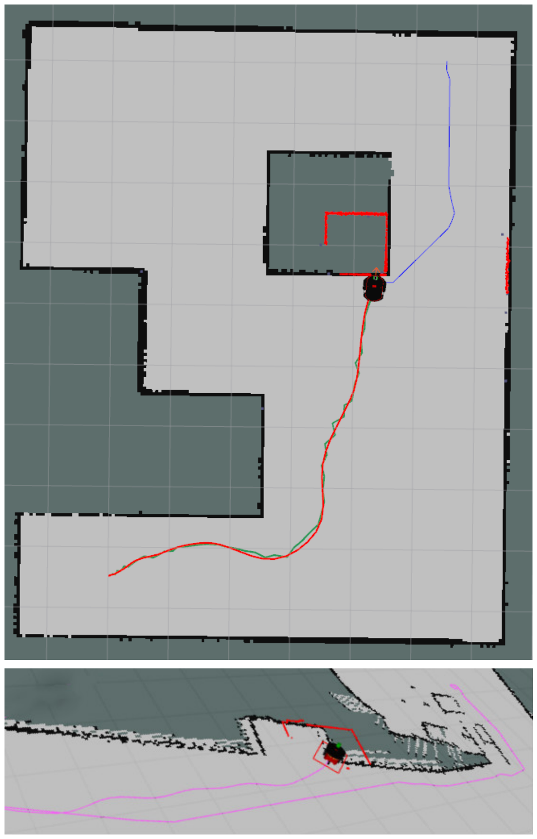
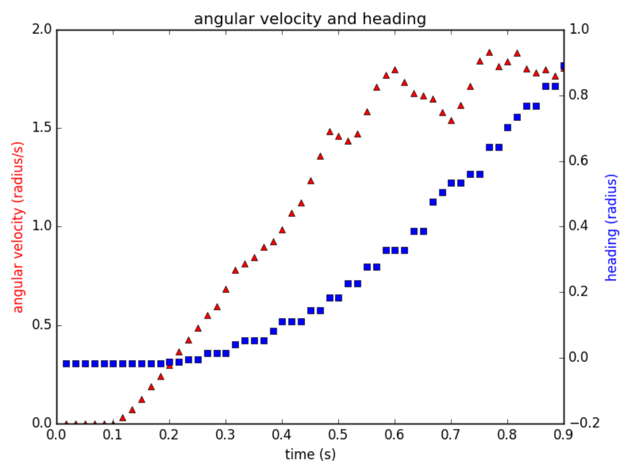


Figure 17: Two fail cases. Top run in simulation. Bottom run in real–world.

$radius/s$, the maximum linear acceleration is 0.65 m/s^2 and the maximum linear velocity is about 3.0 m/s. These parameters are also different from what we get from the manual. We also do the same test for the real–world robot. Figure 21 shows the results. It turns out the configuration of the real robot is quite different from the simulation one. It still get quite good turning performance even it received a linear velocity command. The maximum angular acceleration is about 1.5 $radius/s$, the maximum linear acceleration is about 0.55 m/s^2 and the maximum velocity is 0.75 m/s. This explain why the robot can't closely follow the plan; the reason is that the target state given by the planner is quite above the ability of the robot.

Conclusion

In this paper, we design two control algorithms: sampling based model–predictive control and bisection search based model–predictive control. The algorithms are implemented as the controller for a Pioneer robot. The behaviors of both algorithms are demonstrated through straight line and curve following experiments from simulation and real–world environments. We also discussed several issues of the real–time planning system.



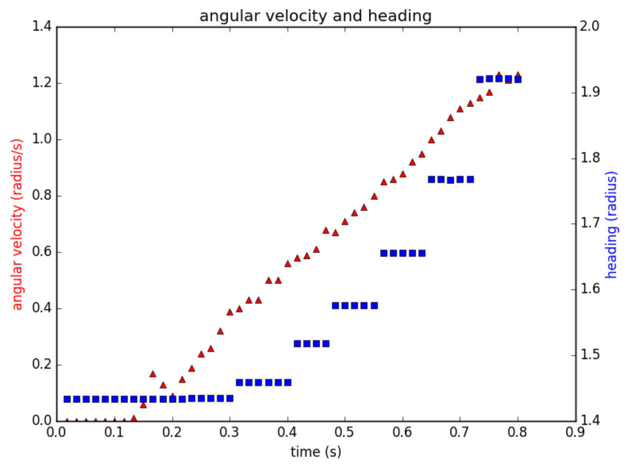


Figure 18: Angular velocity and heading data. Top run in simulation ,and bottom run in real–world.

Acknowledgments

The authors would thank the members of the UNH AI Group and CS980 for their insightful comments.

References

[Cannon, Rose, and Ruml 2014] Cannon, J.; Rose, K.; and Ruml, W. 2014. Real-time heuristic search for motion planning with dynamic obstacles. *AI Communications* 27(4):345–362.

[Doherty et al. 2015] Doherty, P.; Kvarnstrom, J.; Wzorek, M.; Rudol, P.; Heintz, F.; and Conte, G. 2015. Hdrc3: A distributed hybriddeliberative/reactive architecture for unmanned aircraft systems. In *Handbook of Unmanned Aerial Vehicles*. Springer. 849–952.

[Howard et al. 2014] Howard, T. M.; Pivtoraiko, M.; Knepper, R. A.; and Kelly, A. 2014. Model-predictive motion planning: Several key developments for autonomous mobile robots. *IEEE Robotics & Automation Magazine* 21(1):64–73.

[Koenig and Likhachev 2005] Koenig, S., and Likhachev, M. 2005. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics* 21(3):354–363.

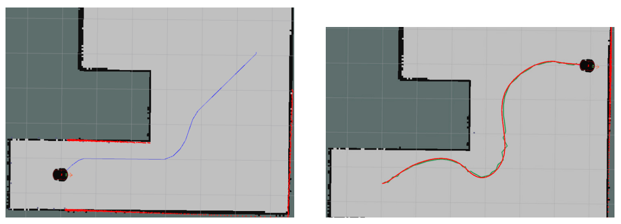


Figure 19: Comparison between the original plan path and the robot's trajectory.

control			end state		
v	ω	t	h	ω	v
0	1.7	1	1.04	1.97	-0.06
0.3	1.7	1	0.74	1.89	0.31
0.6	1.7	1	0.11	1.03	0.49
3	1.7	1	-0.01	-0.01	0.63
0	2.0	1	1.02	2.33	-0.03
0	5.3	1	0.96	2.1	-0.17
0	10	1	0.7	2.21	-0.1
0	20	1	1.1	2.39	-0.05
0.3	0	1	-0.01	-0.02	0.35
0.6	0	1	-0.01	0.01	0.65
1.5	0	1	-0.01	-0.01	0.63
3	0	1	-0.01	0.00	0.65
3	0	5	0.71	0.58	3.04
1.5	0	5	0.21	0.03	1.81

Figure 20: Result of single control test in simulation.

[Koenig and Sun 2009] Koenig, S., and Sun, X. 2009. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems* 18(3):313–341.

[Koenig, Likhachev, and others 2002] Koenig, S.; Likhachev, M.; et al. 2002. Incremental a*. *Advances in neural information processing systems* 2:1539–1546.

[Kushleyev and Likhachev 2009] Kushleyev, A., and Likhachev, M. 2009. Time-bounded lattice for efficient planning in dynamic environments. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, 1662–1668. IEEE.

[Kuwata et al. 2009] Kuwata, Y.; Teo, J.; Fiore, G.; Karaman, S.; Frazzoli, E.; and How, J. P. 2009. Real-time motion planning with applications to autonomous urban driving. *IEEE Transactions on Control Systems Technology* 17(5):1105–1118.

[LaValle and Kuffner 2001] LaValle, S. M., and Kuffner, J. J. 2001. Randomized kinodynamic planning. *The International Journal of Robotics Research* 20(5):378–400.

[Likhachev and Ferguson 2009] Likhachev, M., and Ferguson, D. 2009. Planning long dynamically feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research* 28(8):933–945.

control			end state		
v	ω	t	h	ω	v
0	1.7	1	0.48	1.57	0
0.3	1.7	1	0.43	1.5	0.3
0.6	1.7	1	0.47	1.54	0.5
3	1.7	1	0.11	1.64	0.5
0	2.0	1	0.36	1.46	0
0	5.3	1	0.47	1.45	0
0	10	1	0.51	1.47	0
0	20	1	0.49	1.2	0
0.3	0	1	0	-0.02	0.3
0.6	0	1	0	-0.01	0.53
1.5	0	1	0	0.01	0.54
3	0	1	0	0	0.50
3	0	5	0	0.02	0.75

Figure 21: Result of single control test in real–world.

[Likhachev and Stentz 2008] Likhachev, M., and Stentz, A. 2008. R* search. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 1, 344–350. AAAI.

[Likhachev, Gordon, and Thrun 2003] Likhachev, M.; Gordon, G. J.; and Thrun, S. 2003. Ara*: Anytime a* with provable bounds on sub-optimality. In *NIPS*, 767–774.

[Press et al. 1982] Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; and Flannery, B. P. 1982. *Numerical recipes in C*, volume 2. Cambridge Univ Press.

[Zickler and Veloso 2010] Zickler, S., and Veloso, M. M. 2010. Variable level-of-detail motion planning in environments with poorly predictable bodies. In *ECAI*, 189–194.

[Zucker, Kuffner, and Branicky 2007] Zucker, M.; Kuffner, J.; and Branicky, M. 2007. Multipartite rrts for rapid replanning in dynamic environments. In *Robotics and Automation, 2007 IEEE International Conference on*, 1603–1609. IEEE.