Motion Planning for Robot Soccer

Randal W. Beard
Department of Electrical and Computer Engineering
Brigham Young University

February 18, 2015

Contents

1 Motion Planning Using Waypoints **2**
1.1 System Architecture 3
1.2 Waypoint Planning Using Visibility Graph 5
1.2.1 Paths to Ball 5
1.2.2 Collision Avoidance 6
1.3 Waypoint Planning Using Rapidly Exploring Random Trees . . . 10
1.4 Waypoint Planning Using Dijkstra Search 16
1.5 Ball Intercept 16
1.6 Trajectory Generation 18
1.7 Trajectory Tracking 21
1.8 Smoothing Between Points 23

2 Motion Planning Using Potential Fields **25**
2.1 Potential Fields 25
2.2 Robot Response 29

References **32**

1 Motion Planning Using Waypoints

In the mobile robotics literature, there are roughly two different approaches to motion planning: *deliberative* motion planning where explicit paths and trajectories are computed based on global world knowledge [1, 2, 3], and *reactive* motion planning which uses behavioral methods to react to local sensor information [4, 5]. In general, deliberative motion planning is useful when the environment is known *a priori*, but can become computationally intensive in highly dynamic environments. Reactive motion planning, on the other hand, is well suited for dynamic environments, particularly collision avoidance, where information is incomplete and uncertain, but lacks the ability to specify and direct motion plans.

The objective of these notes is to describe a deliberative approach to motion planning for mobile robots, and to introduce a software architecture for its implementation. In deliberative approaches, robot trajectories are planned explicitly. As a consequence timing can also be specified explicitly. The drawback of deliberative approaches is that they are strongly dependent upon the models used to describe the state of the world and to describe the motion of the ball and the robots. If the models are exact, then the motion planning techniques will be highly effective.

These notes are organized as follows. In Section 1.1 a software architecture is introduced for the implementation of deliberative motion planning. In Section 1.2 we discuss several techniques for waypoint path planning for robot soccer applications. In Section 1.6 we discuss an approach to trajectory generation that is well suited to mobile robots. In Section 1.7 we discuss trajectory tracking techniques for three-wheel mobile robots.

1.1 System Architecture

There are many different approaches to motion planning in robot soccer. There are potentially many different architectures for each possible approach. In this section we will introduce one possible system architecture. This architecture has been used successfully in our research work on unmanned air vehicles (UAVs) [6, 7, 8] and is offered as a possible approach for robot soccer. We hope that introducing one particular architecture will spark new ideas and help you to develop innovative approaches of your own.

The system architecture is shown in Figure 1. At the highest level of the archi-

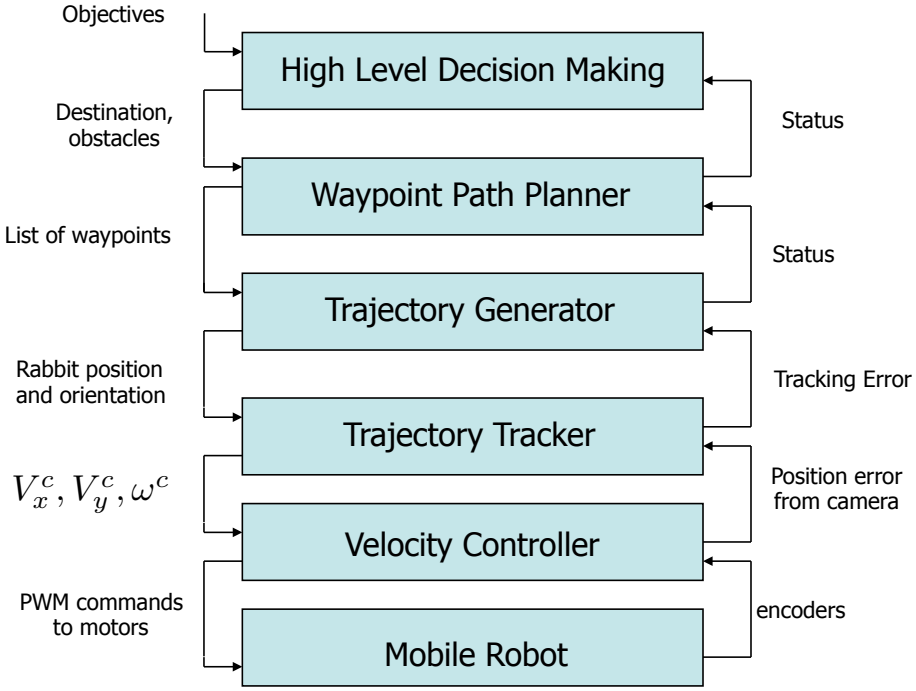

Figure 1: System architecture.

tecture is the High Level Decision Maker. We will not discuss this block in these notes. The High Level Decision Maker uses the current world state and the current objectives, to select destination points for the robots. These are passed to the Waypoint Path Planner which produces straight-line paths, also called waypoint paths, to the target locations. These paths include collision avoidance and possibly ball intercept information. The waypoint paths are simply a set of waypoints

with path velocity information. The role of the Trajectory Generator is to convert the waypoint paths, into a reference trajectory for the robot. In other words, the output of the Trajectory Generator is $(x^r(t), y^r(t), \psi^r(t))^T$ for a three-wheel robot, where (x^r, y^r) are the reference position in world coordinates, and ψ^r is the reference angle of the robot. The Trajectory Tracker converts the reference trajectory and the current configuration of the robot into velocity commands for the robot. For three-wheeled robots the velocity commands are $(V_x^c, V_y^c, \omega^c)^T$, where $(V_x^c, V_y^c)^T$ is the commanded velocity vector, and ω^c is the commanded linear speed. The Velocity Controller converts commanded velocities to pulse width modulation (PWM) commands to be sent to the robot. In these notes, we discuss the Waypoint Path Planner (Section 1.2), the Trajectory Generator (Section 1.6), and the Trajectory Tracker (Section 1.7).

1.2 Waypoint Planning Using Visibility Graph

A waypoint path is sequence of straight-line path segments connecting points. An example of a waypoint path is shown in Figure 2.

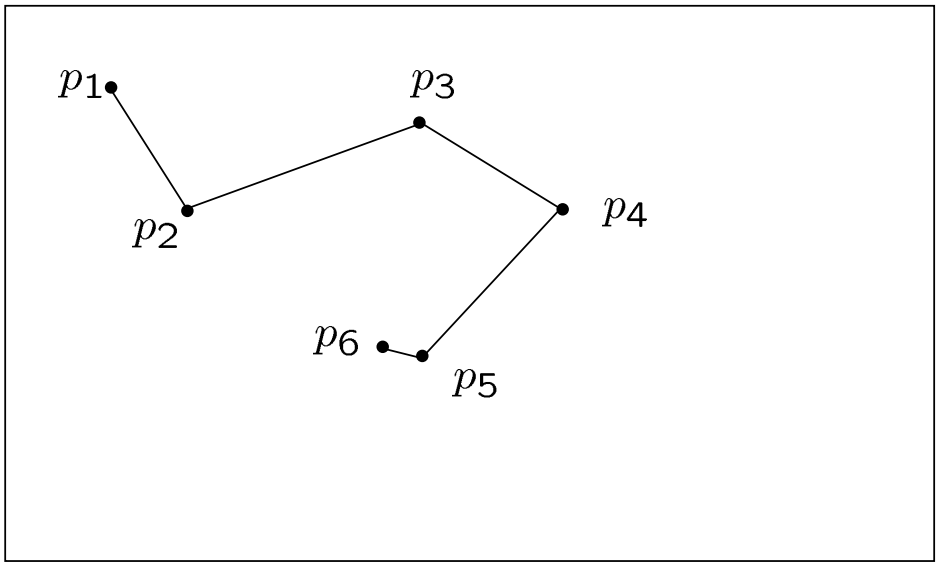

Figure 2: An example of a waypoint path.

1.2.1 Paths to Ball

As will be described in Section 1.7 we will not be able to cause the center of a two wheel the robot to follow waypoint trajectories. As an alternative, we will control the "hand" position of the robot to follow the trajectory. The hand position is defined as the point on the front of the robot that also lies on the line that is perpendicular to the wheel axes, intersecting the center of the robot, as shown in Figure 3. The position of the hand position in the robot frame is $h = (0, L)^\top$.

Since it is the hand position that will be following the trajectory, we must allow a little room for the robot to "straighten-out" after turns. Therefore, if we would like the robot to move from its current location and knock a stationary ball into the goal, we may plan a path like the one shown in Figure 4.

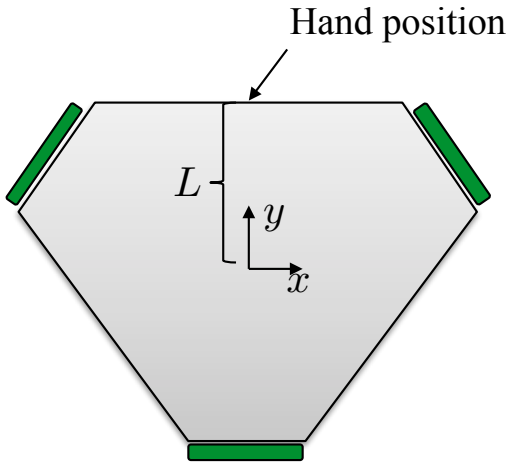

Figure 3: The hand position of the robot.

Suppose that the following variables are available to the path planning routine:

$$\begin{aligned}
\mathbf{r} = (r_x, r_y)^T &\triangleq \text{ center position of robot in world frame,} \\
\psi &\triangleq \text{ heading angle of robot in world frame,} \\
\mathbf{b} = (b_x, b_y)^T &\triangleq \text{ ball position in world frame,} \\
\mathbf{g} = (g_x, g_y)^T &\triangleq \text{ goal position in world frame.}
\end{aligned}$$

Then one possibility for the waypoints $\mathbf{p}_1$–$\mathbf{p}_4$ shown in Figure 4 are

$$\begin{aligned}
\mathbf{p}_1 &= \mathbf{r} + L \begin{pmatrix} \cos\psi \\ \sin\psi \end{pmatrix} \\
\mathbf{p}_2 &= \mathbf{b} - 2L \frac{\mathbf{g} - \mathbf{b}}{\|\mathbf{g} - \mathbf{b}\|} \\
\mathbf{p}_3 &= \mathbf{b} + \frac{1}{2}(\mathbf{g} - \mathbf{b}) \\
\mathbf{p}_4 &= \mathbf{g}.
\end{aligned}$$

1.2.2 Collision Avoidance

Consider the tasking of planning waypoint paths around an opponent as shown in Figure 5.

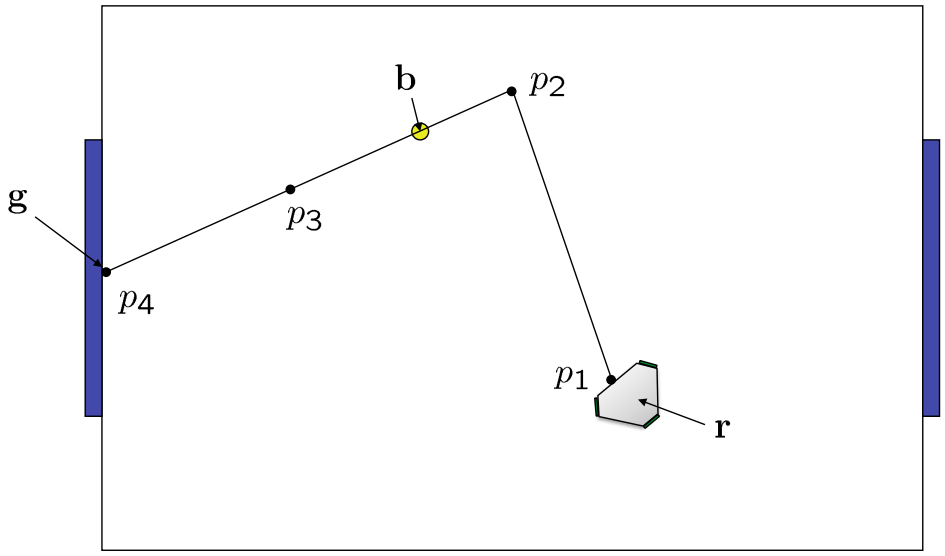

Figure 4: Waypoint path to the ball.

It is desirable to pick $\mathbf{p}_2$ such that the robot will not collide with the opponent as it moves toward the ball. Let $\mathbf{f}$ be the location of the opponent, and suppose that R_f is its radius. Also assume that R_r is the radius of our robot. Then we would like

$$\|\mathbf{p}_2 - \mathbf{f}\| \geq R_r + R_f.$$

The location of $\mathbf{p}_2$ will likely depend on which side of the $\overline{\mathbf{p_1 p_3}}$ line $\mathbf{f}$ is on, and how far $\mathbf{f}$ is from that line.

Consider Figure 6 which defines "RIGHT" and "LEFT" sides of the line, as well as the angle θ and the distance d.

If the angle θ is positive, then we will say that $\mathbf{f}$ is to the LEFT of $\overline{\mathbf{p_1 p_2}}$. On the other hand if θ is negative than $\mathbf{f}$ is said to be to the RIGHT of $\overline{\mathbf{p_1 p_2}}$.

Recall that if $\mathbf{c} = \mathbf{a} \times \mathbf{b}$, then the signed magnitude of $\mathbf{c}$ is $c = \|a\| \, \|b\| \sin(\theta)$. If θ is positive then c will be positive. If θ is negative then c will be negative. In addition $|c|$ will be equal to the d in the figure. Since

$$\begin{pmatrix} p_{2x} - p_{1x} \\ p_{2y} - p_{1y} \\ 0 \end{pmatrix} \times \begin{pmatrix} f_x - p_{1x} \\ f_y - p_{1y} \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ (p_{2x} - p_{1x})(f_y - p_{1y}) - (p_{2y} - p_{1y})(f_x - p_{1x}) \end{pmatrix},$$

if we let

$$c = (p_{2x} - p_{1x})(f_y - p_{1y}) - (p_{2y} - p_{1y})(f_x - p_{1x}),$$

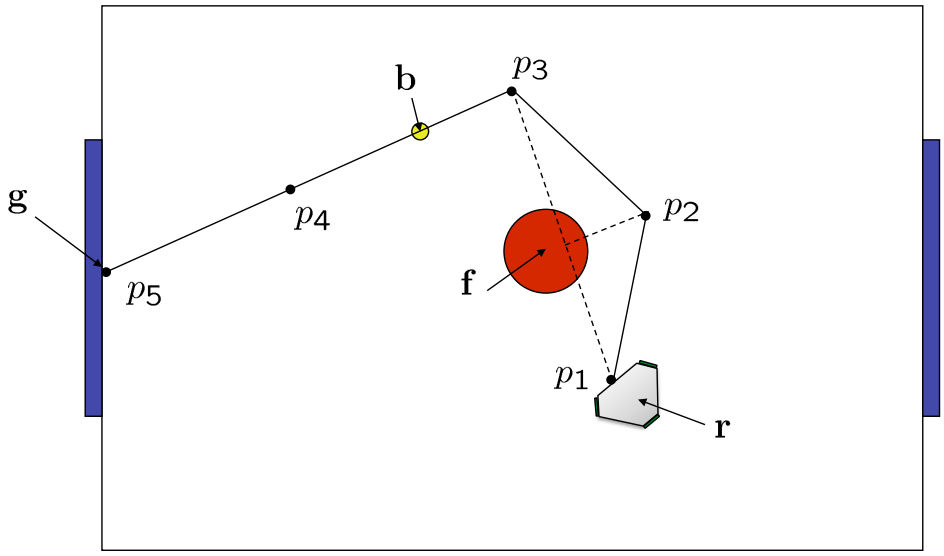

Figure 5: Waypoint path for Collision Avoidance.

then $d = |c| \,/\, \|\mathbf{p}_2 - \mathbf{p}_1\|$ and

$$\text{sign}(c) = \begin{cases} 1 \implies \text{LEFT} \\ -1 \implies \text{RIGHT} \\ 0 \implies \textit{on line}. \end{cases}$$

The point $\mathbf{p}_2$ in Figure 5 will lie on the line that is perpendicular to the vector $\mathbf{p}_3 - \mathbf{p}_1$. Therefore, we need to know how to rotate a vector $\mathbf{q}$ by an angle θ. Consider the geometry show in Figure 7.

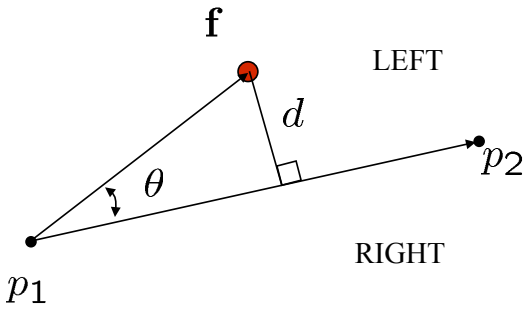

Figure 6: Determining the relative position of object with respect to waypoints $\mathbf{p}_1$ and $\mathbf{p}_2$.

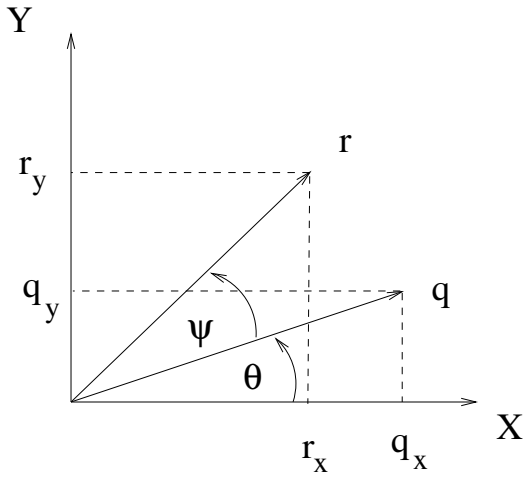

Figure 7: Coordinate frames for derivation or rotation matrix.

Let $\mathbf{r}$ be the rotation of $\mathbf{q}$ by angle θ as shown in Figure 7. Also, let ψ be the original angle of $\mathbf{q}$. Then, letting $\|\mathbf{r}\| = \|\mathbf{q}\| = \alpha$, we get

$$\begin{aligned} q_x &= \alpha \cos\psi \\ q_y &= \alpha \sin\psi \\ r_x &= \alpha \cos(\theta + \psi) \\ r_y &= \alpha \sin(\theta + \psi). \end{aligned}$$

Using the trig identities

$$\begin{aligned} \cos(\theta + \psi) &= \cos\theta\cos\psi - \sin\theta\sin\psi \\ \sin(\theta + \psi) &= \sin\theta\cos\psi + \cos\theta\sin\psi, \end{aligned}$$

we get

$$\begin{pmatrix} r_x \\ r_y \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} q_x \\ q_y \end{pmatrix}.$$

Letting

$$R(\theta) \stackrel{\triangle}{=} \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}, \tag{1}$$

then $\mathbf{r} = R(\theta)\mathbf{q}$, where $R(\theta)$ is called a rotation matrix.

Therefore to place $\mathbf{p}_2$ to the LEFT of $\mathbf{f}$ as shown in Figure 5, we set

$$\mathbf{p}_2 = \mathbf{f} + (R_f + R_r)R(-\frac{\pi}{2})\frac{\mathbf{p}_3 - \mathbf{p}_1}{\|\mathbf{p}_3 - \mathbf{p}_1\|}.$$

1.3 Waypoint Planning Using Rapidly Exploring Random Trees

Another method for planning paths through an obstacle field from a start node to an end node is the Rapidly Exploring Random Tree (RRT) method. The RRT scheme is a random exploration algorithm that uniformly, but randomly, explores the search space. It has the advantage that it can be extended to vehicles with complicated nonlinear dynamics. We assume throughout this section that obstacles are represented in a terrain map that can be queried to detect possible collisions.

The RRT algorithm is implemented using a data structure called a *tree*. A tree is a special case of a directed graph. Figure 8 is a graphical depiction of a tree. Edges in trees are directed from a child node to its parent. In a tree, every node has exactly one parent, except the root, which does not have any parents. In the RRT framework, the nodes represent physical states, or configurations, and the edges represent feasible paths between the states. The cost associated with each edge, c_{ij}, is the cost associated with traversing the feasible path between states represented by the nodes.

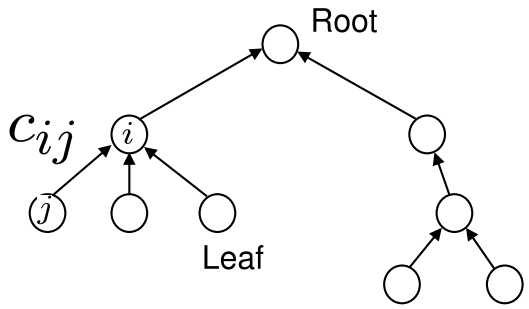

Figure 8: A tree is a special graph where every node, except the root, has exactly one parent.

The basic idea of the RRT algorithm is to build a tree that uniformly explores the search space. The uniformity is achieved by randomly sampling from a uniform probability distribution. To illustrate the basic idea, let the nodes represent north-east locations at a constant altitude, and let the cost c_{ij} of the edges between nodes be the length of the straight-line path between the nodes.

Figure 9 depicts the basic RRT algorithm. As shown in Figure 9(a), the input to the RRT algorithm is a start configuration $\mathbf{p}_s$, an end configuration $\mathbf{p}_e$, and the terrain map. The first step of the algorithm is to randomly select a configuration $\mathbf{p}$ in the workspace. As shown in Figure 9(b), a new configuration $\mathbf{v}_1$ is selected a fixed distance D from $\mathbf{p}_s$ along the line $\overline{\mathbf{p}\mathbf{p}_s}$, and inserted into the tree. At each

subsequent step, a random configuration $\mathbf{p}$ is generated in the workspace, and the tree is searched to find the node that is closest to $\mathbf{p}$. As shown in Figure 9(c), a new configuration is generated that is a distance D from the closest node in the tree, along the line connecting $\mathbf{p}$ to the closest node. Before a path segment is added to the tree, it needs to be checked for collisions with the terrain. If a collision is detected, as shown in Figure 9(d), then the segment is deleted and the process is repeated. When a new node is added, its distance from the end node $\mathbf{p}_e$ is checked. If it is less than D, then a path segment from $\mathbf{p}_e$ is added to the tree, as shown in Figure 9(f), indicating that a complete path through the terrain has been found.

Let $\mathcal{T}$ be the terrain map, and let $\mathbf{p}_s$ and $\mathbf{p}_e$ be the start and end configurations in the map. Algorithm 1 gives the basic RRT algorithm. In Line 1, the RRT graph G is initialized to contain only the start node. The while loop in Lines 2–14 adds nodes to the RRT graph until the end node is included in the graph, indicating that a path from $\mathbf{p}_s$ to $\mathbf{p}_e$ has been found. In Line 3 a random configuration is drawn from the terrain according to a uniform distribution over $\mathcal{T}$. Line 4 finds the closest node $\mathbf{v}^* \in G$ to the randomly selected point $\mathbf{p}$. Since the distance between $\mathbf{p}$ and $\mathbf{v}^*$ may be large, Line 5 plans a path of fixed length D from $\mathbf{v}^*$ in the direction of $\mathbf{p}$. The resulting configuration is denoted as $\mathbf{v}^+$. If the resulting path is feasible, as checked in Line 6, then $\mathbf{v}^+$ is added to G in Line 7 and the cost matrix is updated in Line 8. The *if* statement in Line 10 checks to see if the new node $\mathbf{v}^+$ can be connected directly to the end node $\mathbf{p}_e$. If so, $\mathbf{p}_e$ is added to G in Line 11–12, and the algorithm ends in Line 15 by returning the shortest waypoint path in G.

The result of implementing Algorithm 1 for four different randomly generated obstacle fields and randomly generated start and end nodes is displayed with a dashed line in Figure 10. Note that the paths generated by Algorithm 1 sometimes wander needlessly and that eliminating some nodes may result in a more efficient path. Algorithm 2 gives a simple scheme for smoothing the paths generated by Algorithm 1. The basic idea is to remove intermediate nodes if a feasible path still exists. The result of applying Algorithm 2 is shown with a solid line in Figure 10.

There are numerous extensions to the basic RRT algorithm. A common extension, which is discussed in [**?**], is to extend the tree from both the start and the end nodes and, at the end of each extension, to attempt to connect the two trees. In the next two subsections, we will give two simple extensions that are useful for MAV applications: waypoint planning over 3-D terrain and using Dubins paths to plan kinematically feasible paths in complex 2-D terrain.

Algorithm 1 Plan RRT Path: $\mathcal{W} = \mathsf{planRRT}(\mathcal{T}, \mathbf{p}_s, \mathbf{p}_e)$

Input: Terrain map $\mathcal{T}$, start configuration $\mathbf{p}_s$, end configuration $\mathbf{p}_e$.

Initialize RRT graph $G = (V, E)$ as $V = \{\mathbf{p}_s\}$, $E = \emptyset$.
while The end node $\mathbf{p}_e$ is not connected to G, i.e., $\mathbf{p}_e \notin V$ **do**
  $\mathbf{p} \leftarrow \mathsf{generateRandomConfiguration}(\mathcal{T})$
  $\mathbf{v}^* \leftarrow \mathsf{findClosestConfiguration}(\mathbf{p}, V)$
  $\mathbf{v}^+ \leftarrow \mathsf{planPath}(\mathbf{v}^*, \mathbf{p}, D)$
  if $\mathsf{existFeasiblePath}(\mathcal{T}, \mathbf{v}^*, \mathbf{v}^+)$ **then**
    Update graph $G = (V, E)$ as $V \leftarrow V \bigcup \{\mathbf{v}^+\}$, $E \leftarrow E \bigcup \{(\mathbf{v}^*, \mathbf{v}^+)\}$
    Update edge costs as $C[(\mathbf{v}^*, \mathbf{v}^+)] \leftarrow \mathsf{pathLength}(\mathbf{v}^*, \mathbf{v}^+)$
  end if
  if $\mathsf{existFeasiblePath}(\mathcal{T}, \mathbf{v}^+, \mathbf{p}_e)$ **then**
    Update graph $G = (V, E)$ as $V \leftarrow V \bigcup \{\mathbf{p}_e\}$ $E \leftarrow E \bigcup \{(\mathbf{v}^*, \mathbf{p}_e)\}$
    Update edge costs as $C[(\mathbf{v}^*, \mathbf{p}_e)] \leftarrow \mathsf{pathLength}(\mathbf{v}^*, \mathbf{p}_e)$
  end if
end while
$\mathcal{W} = \mathsf{findShortestPath}(G, C)$.
return $\mathcal{W}$.

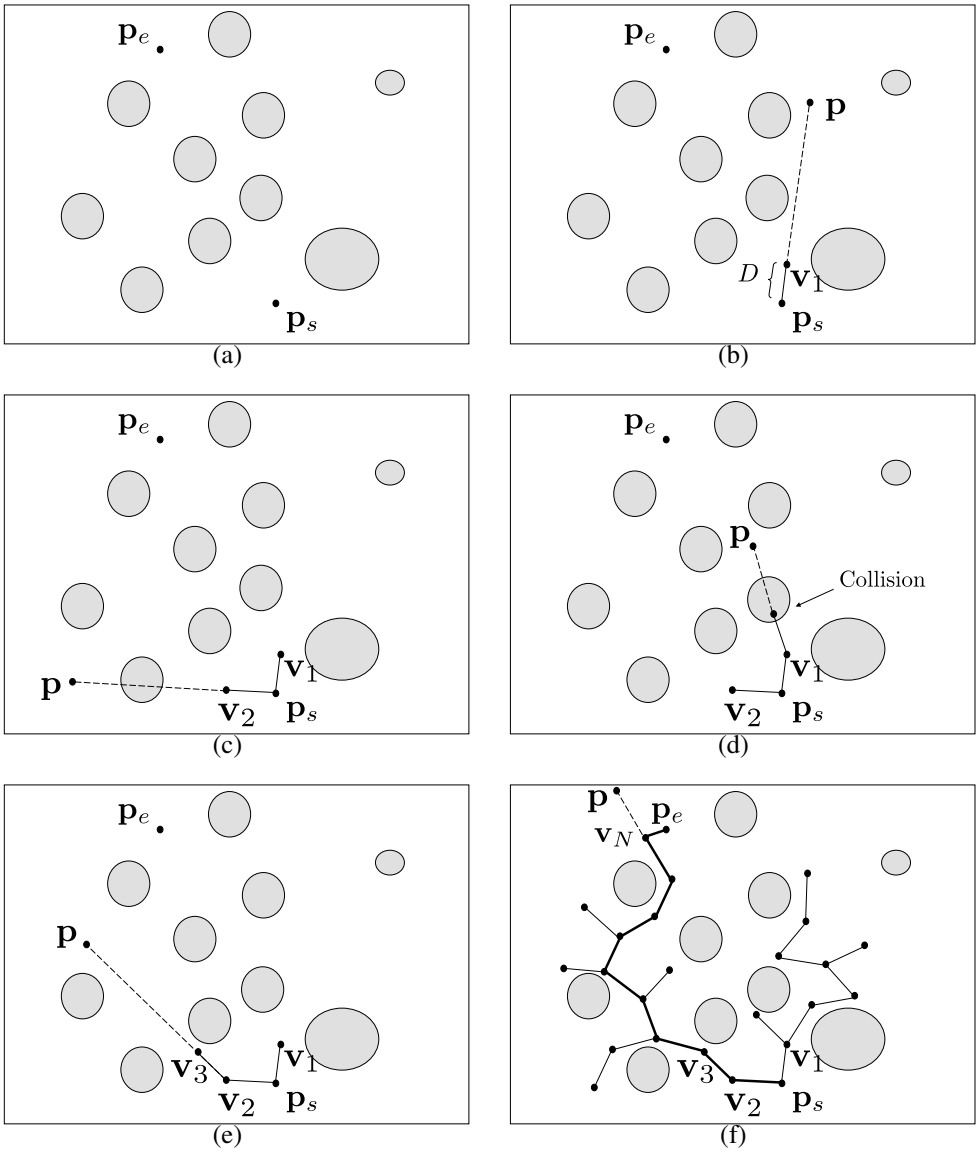

Figure 9: (a) The RRT algorithm is initialized with a terrain map and a start node and an end node. (b) and (c) The RRT graph is extended by randomly generating a point $\mathbf{p}$ in the terrain and planning a path of length D in the direction of $\mathbf{p}$. (d) If the resulting configuration is not feasible, then it is not added to the RRT graph, and the process continues as shown in (e). (f) The RRT algorithm completes when the end node is added to the the RRT graph.

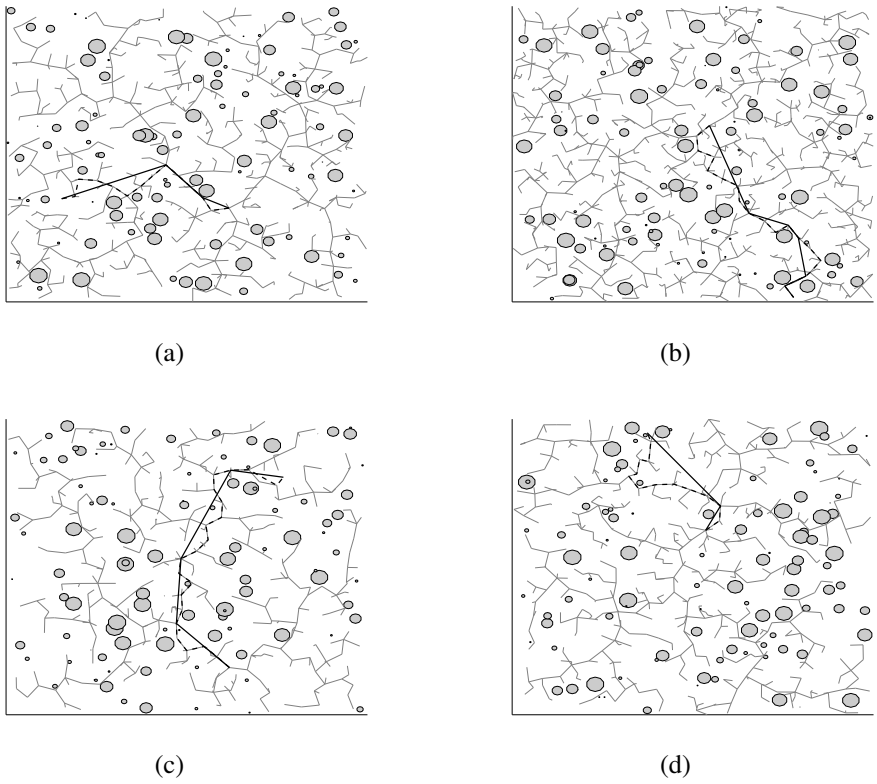

Figure 10: The results of Algorithm 1 for four randomly generated obstacle fields and randomly generated start and end nodes are indicated by dashed lines. The smoothed paths generated by Algorithm 2 are indicated by the solid lines.

Algorithm 2 Smooth RRT Path: $(\mathcal{W}_s, C_s) = \mathsf{smoothRRT}(\mathcal{T}, \mathcal{W}, C)$

Input: Terrain map $\mathcal{T}$, waypoint path $\mathcal{W} = \{\mathbf{w}_1, \ldots, \mathbf{w}_N\}$, cost matrix C.

Initialized smoothed path $\mathcal{W}_s \leftarrow \{\mathbf{w}_1\}$
Initialize pointer to current node in $\mathcal{W}_s$: $i \leftarrow 1$.
Initialize pointer to next node in $\mathcal{W}$: $j \leftarrow 2$
while $j < N$ **do**
  $\mathbf{w}_s \leftarrow \mathsf{getNode}(\mathcal{W}_s, i)$
  $\mathbf{w}^+ \leftarrow \mathsf{getNode}(\mathcal{W}, j+1)$
  if $\mathsf{existFeasiblePath}(\mathcal{T}, \mathbf{w}_s, \mathbf{w}^+) =$ FALSE **then**
    Get last node: $\mathbf{w} \leftarrow \mathsf{getNode}(\mathcal{W}, j)$
    Add deconflicted node to smoothed path: $\mathcal{W}_s \leftarrow \mathcal{W}_s \bigcup \{\mathbf{w}\}$
    Update smoothed cost: $C_s[(\mathbf{w}_s, \mathbf{w})] \leftarrow \mathsf{pathLength}(\mathbf{w}_s, \mathbf{w})$
    $i \leftarrow i + 1$
  end if
  $j \leftarrow j + 1$
end while
Add last node from $\mathcal{W}$: $\mathcal{W}_s \leftarrow \mathcal{W}_s \bigcup \{\mathbf{w}_N\}$
Update smoothed cost: $C_s[(\mathbf{w}_i, \mathbf{w}_N)] \leftarrow \mathsf{pathLength}(\mathbf{w}_i, \mathbf{w}_N)$
return $\mathcal{W}_s$.

1.4 Waypoint Planning Using Dijkstra Search

1.5 Ball Intercept

In this section we will address the problem of planning waypoint paths that intercept the ball along a specified vector $\mathbf{h}$ at a robot speed v as shown in Figure 11. The intercept time will be denotes as T.

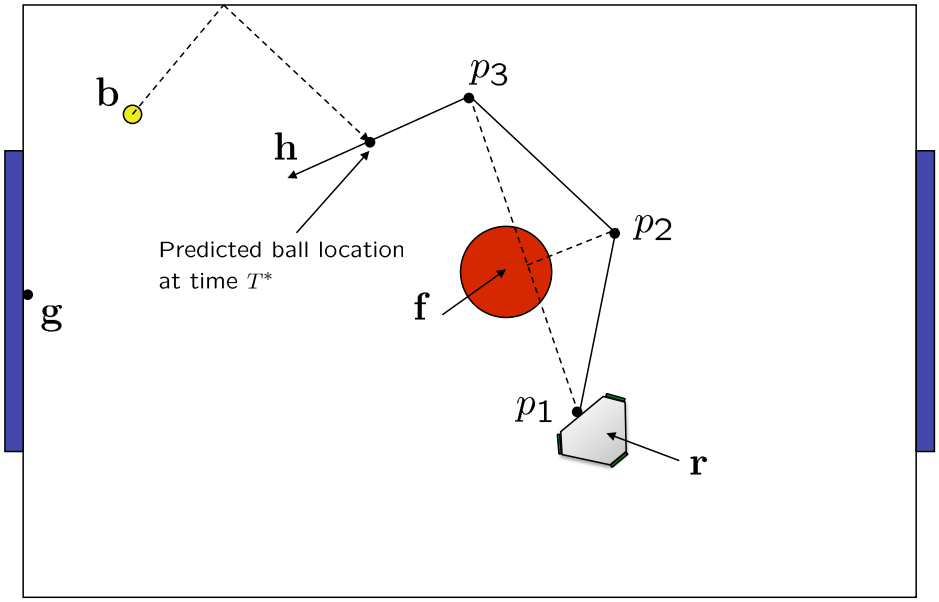

Figure 11: Robot intercepting a ball along vector $\mathbf{h}$.

We will assume that the following functions are available

$\mathbf{b}^+ =$ **ballPredict**(t^+)**:** Returns the predicted ball location at time t^+ into the future.

$\mathcal{P} =$ **planPath**$(\mathbf{p}_1, \mathbf{p}_2, \mathbf{h})$**:** Plans a waypoint path from point $\mathbf{p}_1$, to point $\mathbf{p}_2$, such that the direction of the final leg of the path is identical to $\mathbf{h}$. We assume that collision avoidance is implemented as part of this function.

$L =$ **pathLength**$(\mathcal{P})$**:** Returns the path length of $\mathcal{P}$.

For any $t^+ \geq 0$ we can compute the path length to the predicted ball location as

$$L = \text{pathLength}(\text{planPath}(\mathbf{z}, \text{ballPredict}(t^+), \mathbf{h})),$$

where $\mathbf{z}$ is the current hand location of the robot. The time it takes the robot to traverse this path at speed v is given by

$$T(t^+) = \frac{\text{pathLength}(\text{planPath}(\mathbf{z}, \text{ballPredict}(t^+), \mathbf{h}))}{v}.$$

Define the function

$$g(T) = vT - \text{pathLength}(\text{planPath}(\mathbf{z}, \text{ballPredict}(T), \mathbf{h}))$$

An *intercept time* is defined to be any T^* that satisfies the equation $g(T^*) = 0$. The objective is to find the minimum intercept time, and to follow the resulting path. The minimum intercept time can be found via a binary, or bisection, search algorithm. Note that $g(0) < 0$, and that for some $T > 0$, we are guaranteed that $g(T) > 0$. Therefore the following algorithm will quickly converge to T^*.

Bisection Search

Step 1. Set $T = 0$, $T_1 = 0$.

Step 2. Incrementally increase T until $g(T) > 0$. Set $T_2 = T$.

Step 3. While $\left|g\left(\frac{T_1+T_2}{2}\right)\right| > \epsilon$ if $g\left(\frac{T_1+T_2}{2}\right) > 0$ set $T_2 = \frac{T_1+T_2}{2}$, else set $T_1 = \frac{T_1+T_2}{2}$.

Step 4. Set $T^* = \frac{T_1+T_2}{2}$.

Once T^* is found, plan a path from $\mathbf{z}$ to ballPredict(T^*), to solve the intercept problem

1.6 Trajectory Generation

In this section we will discuss the Trajectory Generator shown in Figure 1. We will assume that a waypoint path has been planned for the robot. Therefore we are given a set of waypoints

$$\mathcal{P} = \{\mathbf{p}_1, \ldots, \mathbf{p}_M\},$$

and a desired velocity v^d along the waypoint path. Our objective is to transform this information into a time-parameterized path for the hand position $\mathbf{z}(t) = (z_x(t), z_y(t))^T$.

Our approach is outlined by the following steps.

1. Parameterize the straightline segment between $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ by a non-dimensional parameter τ which ranges from 0 to 1 over the straightline segment.
2. Derive a differential equation for τ that causes the waypoint path to be traveled at velocity v^d.
3. Modify the differential equation to take into account tracking error.
4. The time-parameterized trajectory is created by integrating τ with respect to time and resetting as the trajectory transitions from one straightline segment to the next.

Given two consecutive waypoints $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$, set $\boldsymbol{\sigma}(\tau; \mathbf{p}_i, \mathbf{p}_{i+1})$ be then τ-parameterized trajectory from $\mathbf{p}_i$ to $\mathbf{p}_{i+1}$, where $\tau \in [0, 1]$. Then

$$\boldsymbol{\sigma}(\tau; \mathbf{p}_i, \mathbf{p}_{i+1}) = (1 - \tau)\mathbf{p}_i + \tau\mathbf{p}_{i+1}.$$

Note that if $\tau = 0$ then $\boldsymbol{\sigma}(0) = \mathbf{p}_i$, and if $\tau = 1$, then $\boldsymbol{\sigma}(1) = \mathbf{p}_{i+1}$.

τ is a parameter that indicates the location of the desired hand position between waypoints $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$. The next step is to allow τ to vary with time. In other words we let $\tau = \tau(t)$. Therefore the time-parameterized reference trajectory is given by

$$\mathbf{z}^r(t) = \boldsymbol{\sigma}(\tau(t)).$$

The velocity of $\mathbf{z}^r$ is given by

$$\begin{aligned} \dot{\mathbf{z}}^r &= \frac{\partial \boldsymbol{\sigma}}{\partial \tau}\dot{\tau} \\ &= (\mathbf{p}_{i+1} - \mathbf{p}_i)\,\dot{\tau}. \end{aligned}$$

Assuming that $\dot{\tau} > 0$, the linear speed is given

$$\|\dot{\mathbf{z}}^r\| = \|\mathbf{p}_{i+1} - \mathbf{p}_i\| \, \dot{\tau}.$$

Therefore, if v^d is the desired linear speed we should set

$$\dot{\tau} = \frac{v^d}{\|\mathbf{p}_{i+1} - \mathbf{p}_i\|}. \tag{2}$$

Note that $\dot{\tau}$ is inversely proportional to the length of the waypoint path. In other words, to maintain a constant velocity when the straightline path segment is short, we need to transition from the beginning of the path to the end of the path much quicker than we will if the straightline path segment is long. Since $\tau \in [0, 1]$ is used to parameterize each straightline segment, τ will need to be reset to zero when the next waypoint $\mathbf{p}_{i+1}$ is reached, i.e., when $\tau = 1$, then τ will need to be reset to $\tau = 0$. An example plot of $\tau(t)$ is shown in Figure 12.

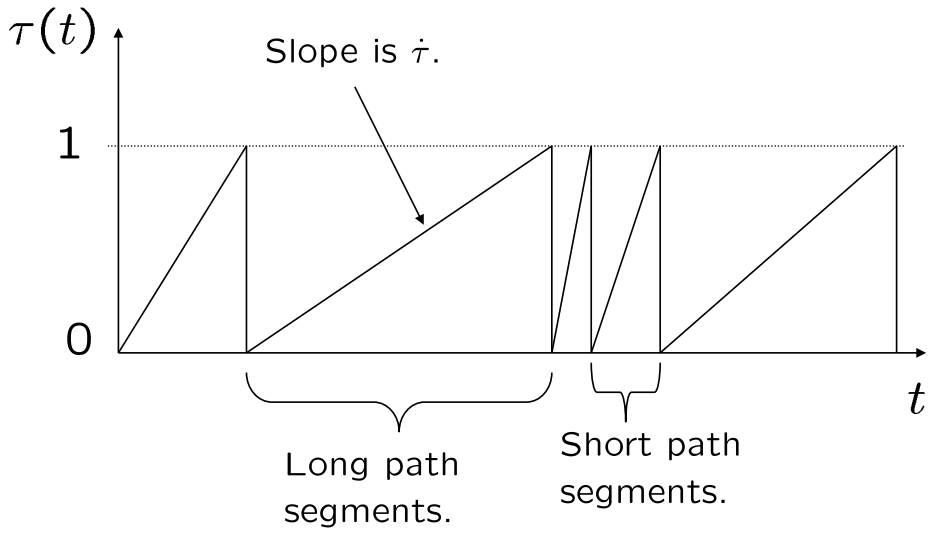

Figure 12: A plot of $\tau(t)$ for an example waypoint path.

One of the potential problems with the trajectory generation scheme that we have outlined above is that the robot may not be able to track the reference trajectory. This may be for a variety of reasons: for example, v^d may be set too high. The trajectory generator outlined above does not have feedback from the tracking error to the reference trajectory. If the robot is lagging behind the reference trajectory we would like to decrease $\dot{\tau}$ so that the reference trajectory slows down to allow the robot to catch up.

Feedback from the tracking error to the reference trajectory can be introduced by modifying the equation for $\dot{\tau}$. Let $\text{error}_{\max}$ be the maximum allowable tracking error that we wish to tolerate. A possible modification of Equation (2) is

$$\dot{\tau} = \frac{v^d}{\|\mathbf{p}_{i+1} - \mathbf{p}_i\|} \left(\frac{\text{error}_{\max} - \|\mathbf{z}(t) - \mathbf{z}^r(t)\|}{\text{error}_{\max}} \right).$$

Note that as the norm of the tracking error $\tilde{\mathbf{z}} = \mathbf{z} - \mathbf{z}^r$ goes to zero, $\dot{\tau}$ approaches Equation (2). However, as $\|\tilde{\mathbf{z}}\| \to \text{error}_{\max}$, $\dot{\tau}$ approaches zero, which implies that the reference trajectory will stop moving.

Example pseudo-code that implements the trajectory generator is given below.

Input. $\mathcal{P} = \{\mathbf{p}_1, \ldots, \mathbf{p}_M\}$, v^d, $\mathbf{z}$.

Initialize.

$i = 1.$ % initialize waypoint counter
$\tau = 0.$ % initialize τ to zero
$\mathbf{z}^r = \mathbf{p}_1.$ % initialize reference trajectory at $\mathbf{p}_1$

At each sample instant (sample rate T_s) do.

if $\tau \geq 1$ **or** $\|\mathbf{z}^r - \mathbf{p}_{i+1}\| < \epsilon$**,**

$i = i + 1.$ % increment waypoint counter
$\tau = 0.$ % reset τ to zero at beginning of new segment
$\mathbf{z}^r = \mathbf{p}_i.$ % reset reference trajectory to next waypoint.

else

$\dot{\tau} = \frac{v^d}{\|\mathbf{p}_{i+1} - \mathbf{p}_i\|} \left(\frac{\text{error}_{\max} - \|\mathbf{z}(t) - \mathbf{z}^r(t)\|}{\text{error}_{\max}} \right)$ % compute $\dot{\tau}$.

$\tau = \tau + T_s \dot{\tau}$ % propagate τ
$\mathbf{z}^r = (1 - \tau)\mathbf{p}_i + \tau \mathbf{p}_{i+1}$ % update reference trajectory.

end

Output. $\mathbf{z}^r$

1.7 Trajectory Tracking

In this section we describe a possible implementation of the Trajectory Tracking block shown in Figure 1. We should note that there are many possible implementations of the trajectory tracker.

The kinematic equations of motion for a three-wheeled mobile robot are given by

$$\begin{aligned} \dot{r}_x &= V_x^c \\ \dot{r}_y &= V_y^c \\ \dot{\psi} &= \omega^c, \end{aligned}$$

where V_x^c, V_y^c, and ω^c are command (control) inputs. As shown in Figure 3, the robot hand position $\mathbf{z}$ is given by

$$\mathbf{z} = \begin{pmatrix} r_x \\ r_y \end{pmatrix} + L \begin{pmatrix} \cos\psi \\ \sin\psi \end{pmatrix}. \tag{3}$$

Differentiating the hand position gives

$$\begin{aligned} \begin{pmatrix} \dot{z}_x \\ \dot{z}_y \end{pmatrix} &= \begin{pmatrix} \dot{r}_x \\ \dot{r}_y \end{pmatrix} + L \begin{pmatrix} -\sin\psi \\ \cos\psi \end{pmatrix} \dot{\psi} \\ &= \begin{pmatrix} V_x^c \\ V_y^c \end{pmatrix} + L \begin{pmatrix} -\sin\psi \\ \cos\psi \end{pmatrix} \omega^c. \end{aligned} \tag{4}$$

If the reference command is given by $z^r(t)$, then the tracking error can be written as

$$\tilde{z} = z - z^r.$$

Differentiating $\tilde{z}$ and using Equation (4) gives

$$\begin{aligned} \dot{\tilde{z}} &= \dot{z} - \dot{z}^r \\ &= \begin{pmatrix} V_x^c \\ V_y^c \end{pmatrix} + L \begin{pmatrix} -\sin\psi \\ \cos\psi \end{pmatrix} \omega^c - \dot{z}^r. \end{aligned}$$

If we assume that the orientation $\psi(t)$ and angular rate command ω^c are selected independently, then we can select the velocity commands to be

$$\begin{pmatrix} V_x^c \\ V_y^c \end{pmatrix} = -L \begin{pmatrix} -\sin\psi \\ \cos\psi \end{pmatrix} \omega^c + \dot{z}^r - \gamma\tilde{z}.$$

Therefore we have that

$$\dot{\tilde{z}} = -\gamma \tilde{z}. \tag{5}$$

From elementary differential equations, the solution of Equation (5) is given by

$$\tilde{z}(t) = e^{-\gamma t} \tilde{z}(0),$$

where $\tilde{z}(0)$ is the initial tracking error. Therefore, the tracking error converges exponentially to zero if $\gamma > 0$.

1.8 Smoothing Between Points

It may be that straight-line waypoint planning will still result in jerky behavior of the robot. An alternative is to plan smooth paths between waypoint configurations. Suppose that the objective is to plan a smooth path between a start position $p_s = (p_{sx}, p_{sy})^\top$ and an end position $p_e = (p_{ex}, p_{ey})^\top$ so that the path also matches the velocity at the start position $v_s = (v_{sx}, v_{sy})^\top$ and the velocity at the end position $v_e = (v_{ex}, v_{ey})^\top$. Similar to previous sections we will use a parameterization of the path denoted by τ. Therefore, the reference trajectory is given by $\mathbf{z}^r(t) = \boldsymbol{\sigma}(\tau(t)) = (\boldsymbol{\sigma}_x(\tau(t)), \boldsymbol{\sigma}_y(\tau(t)))^\top$.

The basic idea will be to write the path using a polynomial basis. A sinusoidal basis could also be used. Therefore, the smooth path can be written as

$$\begin{pmatrix} \boldsymbol{\sigma}_x(\tau) \\ \boldsymbol{\sigma}_y(\tau) \end{pmatrix} = \begin{pmatrix} m_{11} + m_{12}\tau + m_{13}\tau^2 + m_{14}\tau^3 \\ m_{21} + m_{22}\tau + m_{23}\tau^2 + m_{24}\tau^3 \end{pmatrix}, \tag{6}$$

where the coefficients m_{ij} are to be selected to meet the position and velocity constraints at the start and end of the trajectory. It is much more convenient to use matrix notation. Accordingly, define

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \end{pmatrix},$$

and

$$\Phi(\tau) = \begin{pmatrix} 1 \\ \tau \\ \tau^2 \\ \tau^3 \end{pmatrix},$$

then we can write Equation (6) as

$$\boldsymbol{\sigma}(\tau) = M\Phi(\tau).$$

If $\tau = 0$ denotes the beginning of the path and $\tau = 1$ denotes the end of the path, then the position and velocity constraints can be expressed as

$$\begin{aligned} \boldsymbol{\sigma}(0) &= p_s = M\Phi(0) \\ \boldsymbol{\sigma}'(0) &= v_s = M\Phi'(0) \\ \boldsymbol{\sigma}(1) &= p_e = M\Phi(1) \\ \boldsymbol{\sigma}'(1) &= v_e = M\Phi'(1), \end{aligned}$$

where

$$\Phi'(\tau) = \frac{d}{d\tau}\Phi(\tau) = \begin{pmatrix} 0 \\ 1 \\ 2\tau \\ 3\tau^2 \end{pmatrix}.$$

Stacking as a matrix we get

$$\begin{bmatrix} p_s & v_s & p_e & v_e \end{bmatrix} = M \begin{bmatrix} \Phi(0) & \Phi'(0) & \Phi(1) & \Phi'(1) \end{bmatrix}.$$

Therefore, the path coefficients M are given by

$$M = \begin{bmatrix} p_s & v_s & p_e & v_e \end{bmatrix} \begin{bmatrix} \Phi(0) & \Phi'(0) & \Phi(1) & \Phi'(1) \end{bmatrix}^{-1}.$$

2 Motion Planning Using Potential Fields

The objective of this section is to describe a reactive approach to motion planning for mobile robots. In a reactive approach, trajectories are not planned explicitly. Rather, robot interactions are defined explicitly and the robot motion is said to "emerge." The drawback of reactive methods is that it is sometimes difficult to get the robot to do exactly what you want. The reactive approach that will be described in these notes is called the "virtual fields" method and is commonly used in robotics.

The basic idea is to set up a virtual potential or force field which is defined by the location of objects, walls, other robots, the robot of interest, etc.. The robot then responds locally to the field. Accordingly, there are two key elements:

- Definition of the force or potential field.
- Specification of the robot response to the field.

In Section 2.1 we will discuss several common potential fields and how they might be used in robot soccer. In addition we will discuss a possible implementation using a global state variable. In Section 2.2 we discuss possible robot response to potential fields.

2.1 Potential Fields

In this section we will describe several possible potential fields.

A potential field will be described a function $E : I\!R^2 \to I\!R$. For example the function

$$E(\mathbf{x}) = c \tag{7}$$

where c is a constant defines a constant potential field. A constant potential field is not very useful, since there is no gradient to follow. A linear potential field is can be defined by the function

$$E(\mathbf{x}) = \mathbf{a}^T\mathbf{x} + c, \tag{8}$$

where $\mathbf{a}$ is a constant vector, and c is a constant. The gradient of this potential field is given by

$$\frac{\partial E}{\partial \mathbf{x}}(\mathbf{x}) = \mathbf{a}.$$

Therefore the gradient is constant and points in the direction of $\mathbf{a}$. If the robot is programmed to follow the negative gradient of E, then a linear potential field would cause the robot to move in the direction of $\mathbf{a}$.

Another common potential field is a quadratic potential:

$$E(\mathbf{xx}) = \frac{1}{2}(\mathbf{x} - \mathbf{c})^T(\mathbf{x} - \mathbf{c}), \tag{9}$$

where $\mathbf{c}$ is a constant vector. A plot of constant potential lines for (9) is shown in Figure 13.

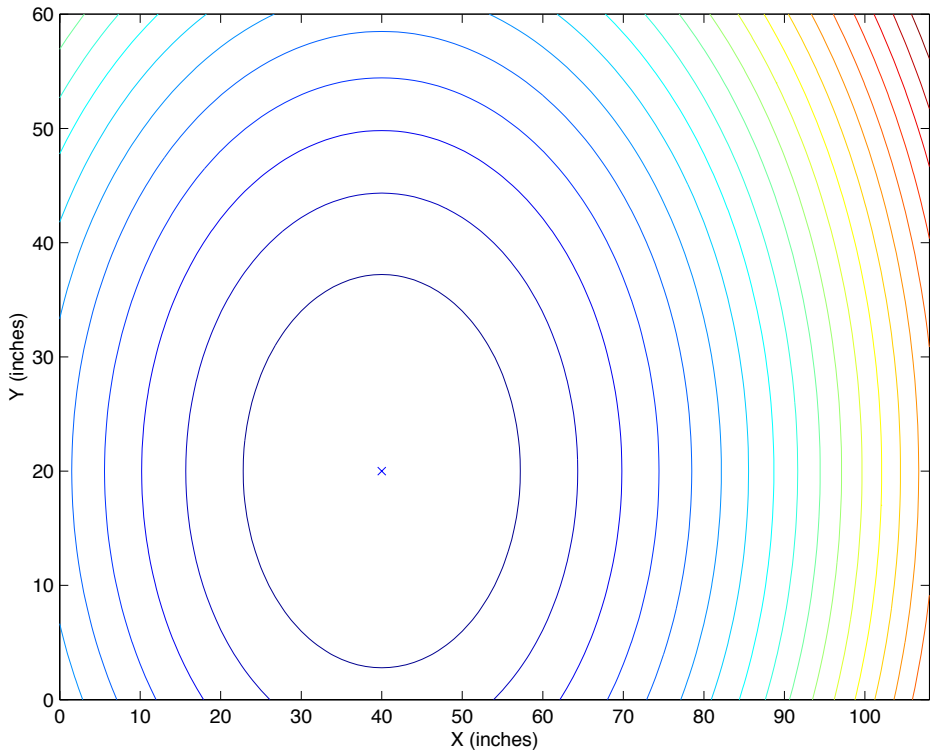

Figure 13: Constant potential lines for quadratic potential field.

The gradient of the potential field defined in (9) is

$$\frac{\partial E}{\partial \mathbf{x}}(\mathbf{x}) = \mathbf{x} - \mathbf{c},$$

which always points away from $\mathbf{c}$. If the robot is programmed to follow the negative gradient of E, the quadratic potential will cause the robot to move in the

direction of $\mathbf{c}$. Equation (9) defines an attractive potential for $\mathbf{c}$. Accordingly, $\mathbf{c}$ is called an attractor. Alternatively, if E is defined as

$$E(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\mathbf{c})^T(\mathbf{x}-\mathbf{c}), \tag{10}$$

then $\mathbf{c}$ is a repulsor, since following the negative gradient of E will cause the robot to move away from $\mathbf{c}$.

As an extension of (9) and (10) the following potential field might be defined:

$$E(\mathbf{x}) = \sum_{\mathbf{a}\in\mathcal{A}} \frac{1}{2}(\mathbf{x}-\mathbf{a})^T(\mathbf{x}-\mathbf{a}) - \sum_{\mathbf{r}\in\mathcal{R}} \frac{1}{2}(\mathbf{x}-\mathbf{r})^T(\mathbf{x}-\mathbf{r})$$

where $\mathcal{A}$ is a set of attractors and $\mathcal{R}$ is a set of repulsors. The gradient of E is given by

$$\begin{aligned}\frac{\partial E}{\partial \mathbf{x}}\mathbf{x}(\mathbf{x}) &= \sum_{\mathbf{a}\in\mathcal{A}}(\mathbf{x}-\mathbf{a}) - \sum_{\mathbf{r}\in\mathcal{R}}(\mathbf{x}-\mathbf{r}) \\ &= (|\mathcal{A}| - |\mathcal{R}|)\mathbf{x} - \left(\sum_{\mathbf{a}\in\mathcal{A}}\mathbf{a} - \sum_{\mathbf{r}\in\mathcal{R}}\mathbf{r}\right),\end{aligned}$$

where $|A|$ is the number of elements in $\mathcal{A}$. In other words, the robot will be attracted (or repulsed from) the center of mass of the attractors and repulsors. A potential field with five attractors and three repulsors is shown in Figure 14.

Suppose that we would like the robot to move toward near-by attractors and move away from near-by repulsors, then we could define the potential field as

$$E(\mathbf{x}) = -\sum_{\mathbf{a}\in\mathcal{A}} \alpha_a e^{-\frac{\gamma_a}{2}\|\mathbf{x}-\mathbf{a}\|^2} + \sum_{\mathbf{r}\in\mathcal{R}} \beta_r e^{-\frac{\gamma_r}{2}\|\mathbf{x}-\mathbf{r}\|^2}, \tag{11}$$

where the constants α_a, γ_a, β_r, and γ_r can be used to specify the strength of the attractor or repulsor. The gradient of (11) is given by

$$\frac{\partial E}{\partial \mathbf{x}}(\mathbf{x}) = +\sum_{\mathbf{a}\in\mathcal{A}} \alpha_a\gamma_a(\mathbf{x}-\mathbf{a})e^{-\frac{\gamma_a}{2}\|\mathbf{x}-\mathbf{a}\|^2} - \sum_{\mathbf{r}\in\mathcal{R}} \beta_r\gamma_r(\mathbf{x}-\mathbf{r})e^{-\frac{\gamma_r}{2}\|\mathbf{x}-\mathbf{r}\|^2}.$$

The potential field with the same attractors and repulsors as Figure 14, but with potential field (11) is shown in Figure 15

It may also be desirable to include a potential field that repulses the robot from the walls. A simple potential field that does the job is

$$E(\mathbf{r}) = \alpha e^{-\frac{\gamma}{2}(r_y-\text{YMAX})^2} + \alpha e^{-\frac{\gamma}{2}r_y^2} + \alpha e^{-\frac{\gamma}{2}(r_x-\text{XMAX})^2} + \alpha e^{-\frac{\gamma}{2}r_x^2}, \tag{12}$$

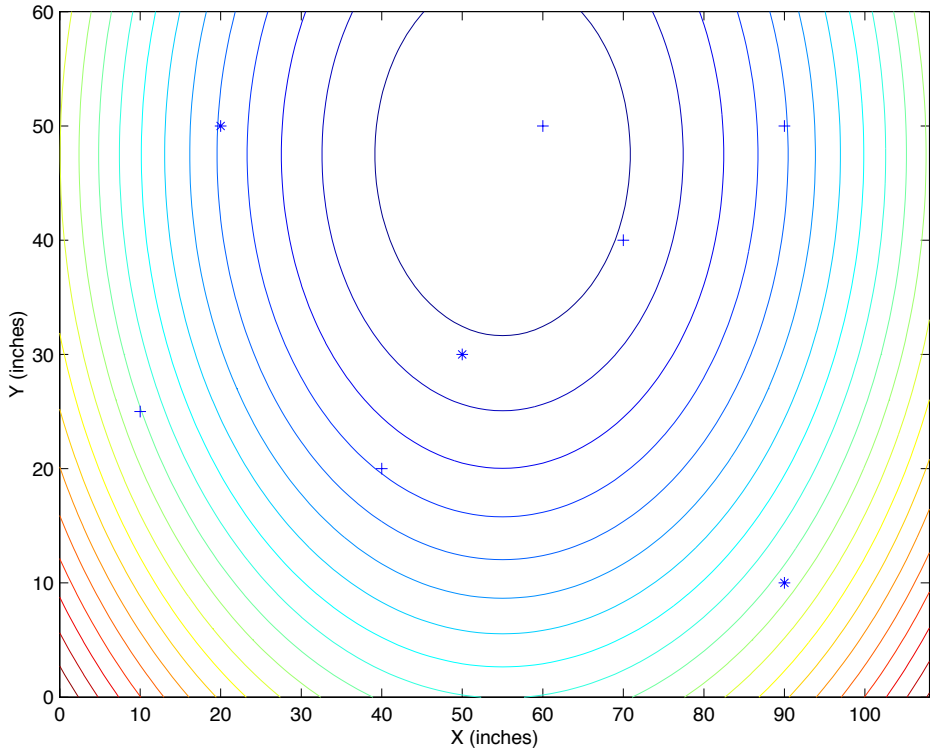

Figure 14: Quadratic potential field with five attractors (+) and three repulsors (*).

with gradient given by

$$\frac{\partial E}{\partial \mathbf{r}}(\mathbf{r}) = -\gamma\alpha(r_y - \text{YMAX})e^{-\frac{\gamma}{2}(r_y-\text{YMAX})^2} - \gamma\alpha r_y e^{-\frac{\gamma}{2}r_y^2} \\ - \gamma\alpha(r_x - \text{XMAX})e^{-\frac{\gamma}{2}(r_x-\text{XMAX})^2} - \gamma\alpha r_x e^{-\frac{\gamma}{2}r_x^2}.$$

The potential field established by (12) is shown in Figure 16.

Combining the potential fields from (11) and (12) results in the potential field shown in Figure 17.

One possible implementation would be to define a list of attractors and repulsors in the global data structure and then to write a skill that computes the gradient of the potential field at the current robot location, given the current positions of the attractors and repulsors. Interesting tactics could be constructed by "guiding" the robot around the field by dynamically changing the positions of the attractors and repulsors.

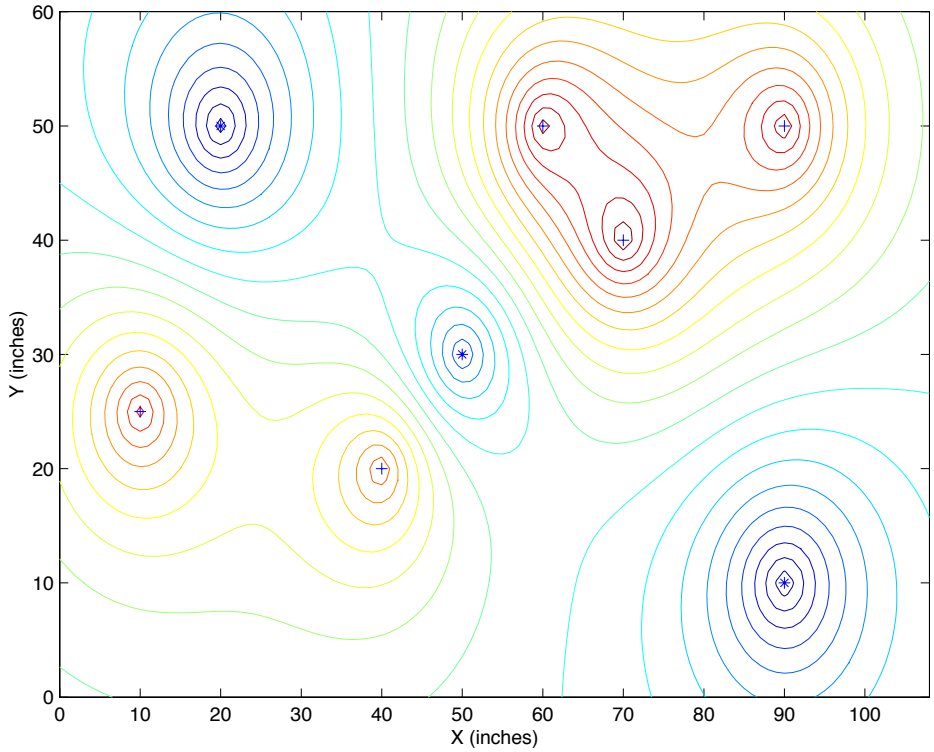

Figure 15: Exponential potential field with five attractors (+) and three repulsors (*).

2.2 Robot Response

The second important aspect of the potential fields method is to define the response of the robot to the potential field. In the previous section we developed the potential fields under the assumption that the robot would be following the negative gradient of the field. Unfortunately, this is not possible to execute exactly due to the nonholonomic nature of the robot.

As an alternative one might attempt to orient the robot in the direction of the negative gradient, while simultaneously moving at velocity which is proportional to the projection of the velocity vector onto the negative gradient. The desired orientation of the robot is given by

$$\psi^d = \text{atan2}(-\frac{\partial E}{\partial r_y}, -\frac{\partial E}{\partial r_x}).$$

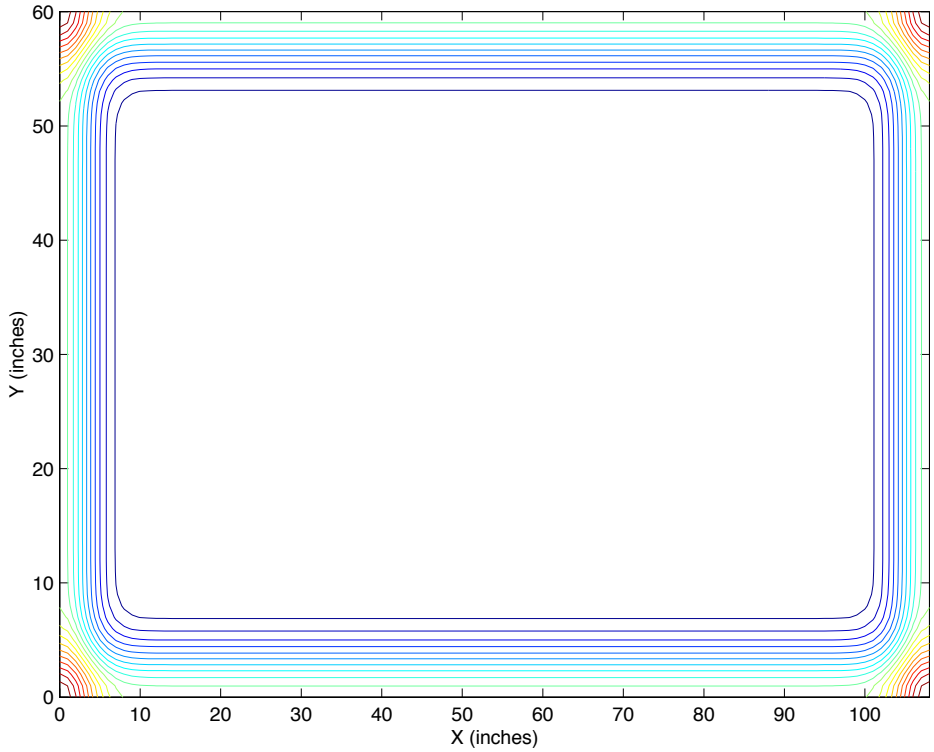

Figure 16: Potential field for avoiding the walls.

Therefore using a proportional control we set

$$\omega^d = -k_p(\psi - \psi^d).$$

The desired velocity is obtained by projecting the direction vector onto the negative gradient:

$$v^d = -\frac{\partial E}{\partial r_x}\cos(\psi) - \frac{\partial E}{\partial r_y}\sin(\psi).$$

If we have implemented a utility that moves the robot at linear speed v^d and angular speed ω^d, then we can invoke this utility to synthesize the desired motion.

References

[1] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," in *Algorithmic and Computational*

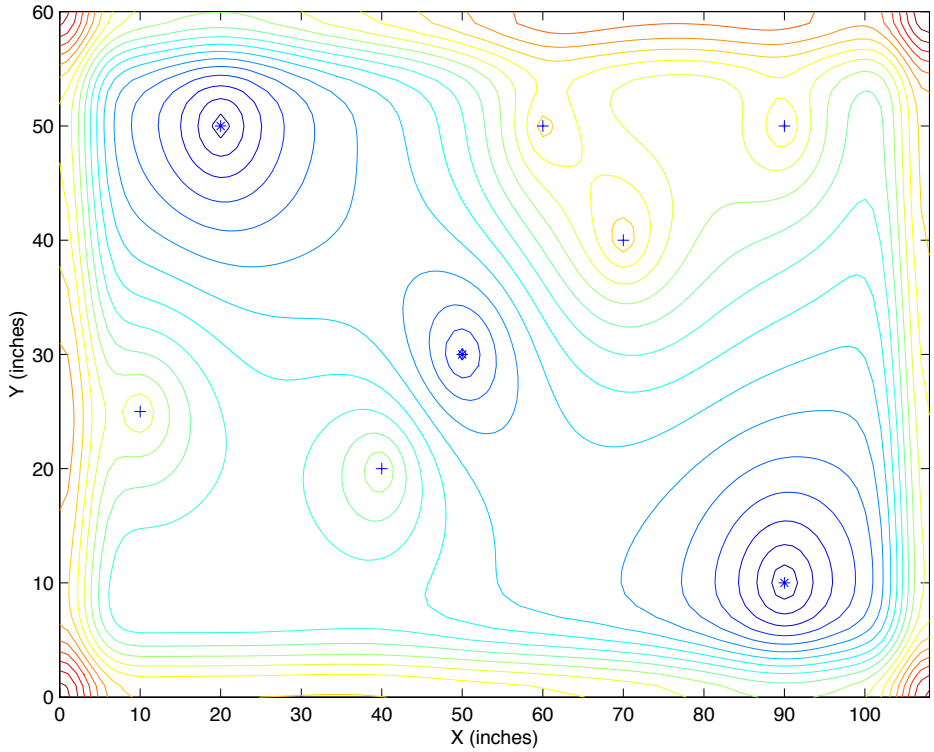

Figure 17: Potential field with attractors and repulsors and wall avoidance.

Robotics: New Directions, pp. 247–264f, A. K. Peters, 2001.

[2] F. Lamiraux, S. Sekhavat, and J.-P. Laumond, "Motion planning and control for Hilare pulling a trailer," *IEEE Transactions on Robotics and Automation*, vol. 15, pp. 640–652, August 1999.

[3] R. M. Murray and S. S. Sastry, "Nonholonomic motion planning: Steering using sinusoids," *IEEE Transactions on Automatic Control*, vol. 38, pp. 700–716, May 1993.

[4] T. Balch and R. C. Arkin, "Behavior-based formation control for multirobot teams," *IEEE Transactions on Robotics and Automation*, vol. 14, pp. 926–939, December 1998.

[5] R. C. Arkin, *Behavior-based Robotics*. MIT Press, 1998.

[6] R. W. Beard, T. W. McLain, M. Goodrich, and E. P. Anderson, "Coordinated target assignment and intercept for unmanned air vehicles," *IEEE Transactions on Robotics and Automation*, vol. 18, pp. 911–922, December 2002.

[7] R. W. Beard and T. W. McLain, "Multiple UAV cooperative search under collision avoidance and limited range communication constraints," in *Proceedings of the IEEE Conference on Decision and Control*, 2003. Submitted.

[8] D. Kingston, R. Beard, T. McLain, M. Larsen, and W. Ren, "Autonomous vehicle technologies for small fixed wing UAVs," in *AIAA 2nd Unmanned Unlimited Systems, Technologies, and Operations–Aerospace, Land, and Sea Conference and Workshop & Exhibit*, (San Diego, CA), September 2003. Paper no. AIAA-2003-6559.