Chapter 5

Safe Navigation

This chapter describes a novel method for safety among a set of cooperating robots. The robots operate under the realistic dynamics constraints of bounded acceleration and bounded velocity while maintaining safety among themselves and with a static environment. Under conditions of noiseless sensing, perfect dynamics, and perfect communication, the system can guarantee no collisions will take place as the robots move about the environment. In more realistic settings, we've found the system to significantly decrease the number and severity of collisions for a team of high performance robots playing RoboCup soccer. The system was used for the past two years on our RoboCup small-size team. We lead the chapter by describing the dynamics model used for the robot and the motion control system employed to reach target points. The following section then describes the safety method, which operates as a post-process to the motion control.

5.0.3 Contributions of this Chapter

- The novel multi-agent *Dynamics Safety Search* (DSS) algorithm is described. It is based on Fox et al.'s single-agent *Dynamic Window* approach [38].
- An outline of a proof of single-agent and multi-agent safety of DSS is contributed. The *Dynamic Window* approach cannot guarantee exact safety even in the single agent case.
- DSS is shown to have polynomial complexity (as opposed to exponential complexity for joint planning), and near linear complexity in simulation testing.

- DSS is demonstrated to aid in collision avoidance on real robots from the RoboCup small size domain.

5.1 Robot Model

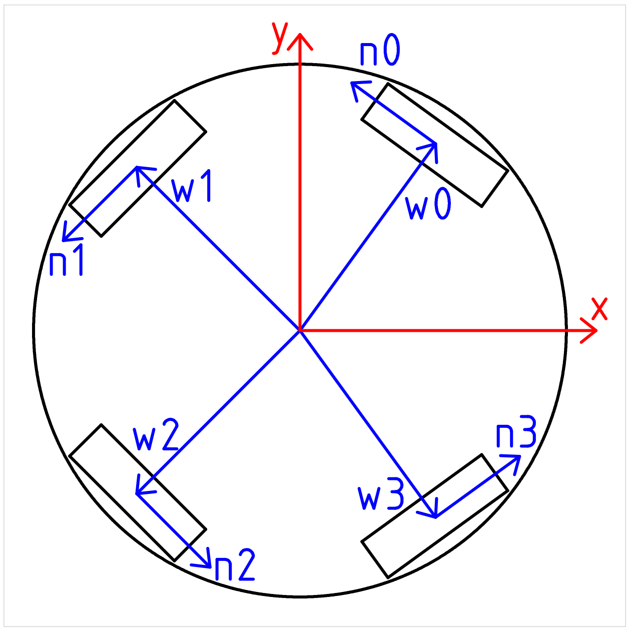
While there are many possible configurations of wheeled robots, each with their associated dynamics, three classes covers many types of wheeled robots:

- Differential drive robots (two or more unsteered wheels or tracks)
- Holonomic robots (three or more omni-directional wheels)
- Car-like steered robots

In this chapter we will assume a holonomic robot model, as those are the robots with which we have the most experience and we have available for testing. While the motion control for each type of robot is significantly different, the safety method from this chapter can be extended for different types of robots, with the core assumptions being that a robot can remain at rest, and can come to a stop while traveling on a straight line. In the derivation of our algorithm, we will use the more restrictive assumptions which attempt to model an electrically driven holonomic robot. The assumptions and limitations of the robot model are summarized as follows:

1. The robot is contained within a safety radius
2. The robot has immediate and direct control of its acceleration
3. The robot acceleration restricted to be within some set
4. The robot has a maximum "emergency stop" deceleration
5. The robot can only change the acceleration at fixed time intervals (control period)
6. The robot has a maximum bounded velocity
7. The robot has no minimum required velocity

Clearly these meet the core assumptions, as an agent can stop on a straight line and come to rest if it directly controls its deceleration up to some bound, with no minimum velocity necessary. In Figure 5.1 we show the layout of the drive system of one the CMDragons RoboCup robots. It has four omni-wheels at fixed angles which. One axis of each wheel is oriented toward the center of the robot and driven, while the other axis rolls freely due to the omni-wheel's perpendicular rollers. We will explore the model more fully in Chapter 5.1, but for the purposes of the safety search it will suffice that the robot and its control system meet the assumptions enumerated above.



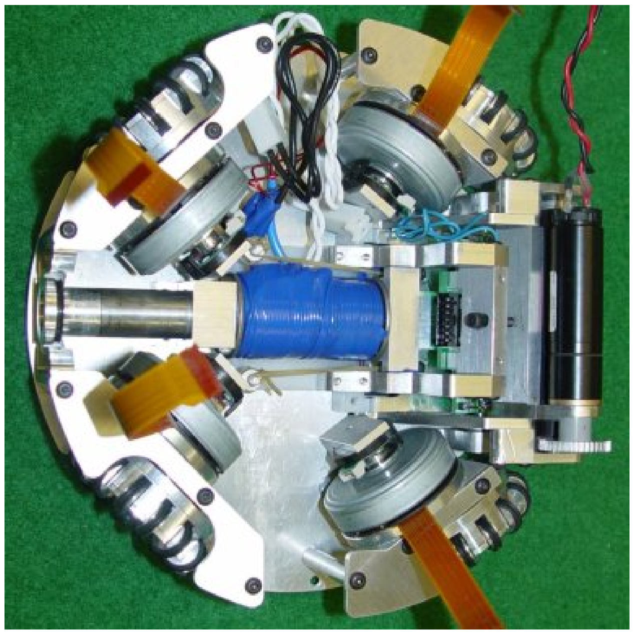


Figure 5.1: Drive layout for a CMDragons holonomic robot with four omni-directional wheels (left), and a picture of a partially assembled robot (right).

The notation used for the model is the following:

- $q(t)$ denotes the robot's position at time t
- $\dot{q}(t)$ denotes the robot's velocity
- $\ddot{q}(t)$ denotes the robot's acceleration
- $A(v)$ denotes the set of possible accelerations at velocity v
- D denotes the maximum "emergency stop" deceleration magnitude
- R denotes the safety radius
- a^k denotes the desired acceleration at decision cycle k (algorithm input)
- u^k denotes the acceleration command at decision cycle k (system control input)

- C denotes the fixed control period

With this notation, we can define the system transfer functions for position (f) and velocity ($\dot{f}$) using Newtonian dynamics:

$$\begin{aligned} f(q, \dot{q}, u, t_\Delta) &= q + \dot{q}t_\Delta + \frac{1}{2}u_i t_\Delta^2 && (5.1) \\ \dot{f}(\dot{q}, u, t_\Delta) &= \dot{q} + u_i t_\Delta && (5.2) \end{aligned}$$

Within the context of the safety algorithm, we will at times need a more complex denotation for q. At each decision cycle k the algorithm will construct a new piecewise function q for each robot i. Thus $q_i^k(t)$ denotes the position function over t at cycle k for robot i. k will be omitted when it is obvious, such as within a single iteration of the algorithm. To identify the individual components of the piecewise function q, we will use $q_i(t, s)$ to denote $q_i(t)$ at component s. At some times, in particular within pseudocode, a shorthand will be used for the "current" state of an agent at time t_0 in iteration k. It uses the following definitions:

$$\begin{aligned} x_{i0} &= q_i^k(t_0) && (5.3) \\ \dot{x}_{i0} &= \dot{q}_i^k(t_0) && (5.4) \\ \ddot{x}_{i0} &= \ddot{q}_i^k(t_0) && (5.5) \end{aligned}$$

Also of note, is that we did not explicitly define the maximum speed bound for an agent, as it can be expressed through the set of valid accelerations, $A(v)$. The set $A(v)$ should be chosen to reflect the agent's physical constraints on acceleration, as well as encoding the maximum achievable or allowable velocity. This can be represented by restricting the set of accelerations to those that would not exceed the maximum velocity after time C, or in other words:

$$\forall_{u \in A(v)} \|v + Cu\| \leq V_{\max} \qquad (5.6)$$

Although in the safety algorithms presented below, no particular form for $A(v)$ is assumed, practical implementations must choose a representation. In modelling a holonomic robot, a "traction circle" has been found to work well in practice [52,77]. This corresponds to an $A(v)$ which is circular (or spherical in higher dimensions) with some radius F, and centered on zero in acceleration space. However, some robots may be able to decelerate faster than their maximum acceleration ($D > F$), and thus it should be included in the model if possible. One way of doing this is a “partial ellipse”, where $A(v)$ is a union of a traction circle of radius F and half of an ellipsoid with a major axis of D and a minor axis of F. The half-ellipse is oriented with its major axis away from the current velocity. A representative acceleration space plot of this shape is shown in Figure 5.2. While both the traction circle and the partial ellipse are only approximations to the true set of possible accelerations (see Sherback et al. [77] for a detailed study), the partial ellipse model has been found to be a good approximation for the CMDragons robots. If $D = F$, the partial ellipse model is equivalent to the traction circle. With either model, $A(v)$ must still be modified to exclude velocities that exceed the maximum bound as described above.

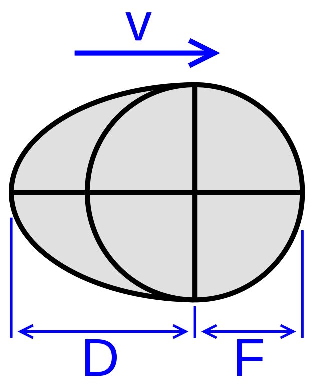


Figure 5.2: The model of $A(v)$ used in applying DSS to small-size soccer robots. F is the maximum acceleration and D is maximum deceleration. It is oriented by the current velocity v.

Finally, we can give a formal definition of safety. This is a refinement of the weaker notion of safety at a particular time expressed in Equation 1.2, as this new version uses a bounded radius model for robots, and even more importantly it describes safety at all times past a particular point, rather than only at a particular moment in time. This “strong” notion of safety is represented as the following function S, which is comprised of two parts. S_e is a boolean function indicating safety between a robot i and the environment. S_r is a safety function indicating safety between n mobile robots. Overall safety in a domain consisting of a static environment and mobile agents is described by the function S, which is a conjunction of S_e and S_r.

$$
\begin{aligned}
S_e(k,t_0) &= \forall_{i\in[1,n]}\,\forall_{t>t_0}\; q_i^k(t)\in C_{\mathit{free}} && (5.7)\\
S_r(k,t_0) &= \forall_{i,j\in[1,n],i\neq j}\,\forall_{t>t_0}\; \|q_i^k(t)-q_j^k(t)\|^2 \geq (R_i+R_j)^2 && (5.8)\\
S(k,t_0) &= S_e(k,t_0)\vee S_r(k,t_0) && (5.9)
\end{aligned}
$$

S_e indicates if an agent i stays in the free configuration space after some time t_0, and thus if true, guarantees the agent has not collided with an obstacle within the configuration space. S_r indicates if the Euclidean distance between any two distinct agents is always at least the sum of their safety radii. Thus if true, it guarantees the agents never pass close enough to collide after t_0. The following section details the Dynamic Window algorithm, which attempts to maintain S_e for a single agent. It is followed by the Dynamic Safety Search algorithm, which guarantees for a set of cooperating robots that S true in one iteration remains true in subsequent iterations, while the robots attempt to execute externally specified goals.

5.2 The Dynamic Window Approach

The goal of the safety system is to act as a post-process to motion control, which normally does is not concerned with avoiding obstacles, instead focusing only on convergence given a goal and the agent's dynamics. Safety could be handled at the motion planning level, subsuming all obstacle avoidance, motion control, and dynamics safety into a single algorithm. However, current solutions treat the resulting problem as a higher dimensional planning problem, resulting in exponential complexity in the number of agents. Thus what is desired is a polynomial complexity algorithm practical for multi-agent teams, and while not complete, can still guarantee safety algorithmically. Our method will extend a single-agent algorithm to reach these goals.

The "Dynamic Window" approach [38] is a search method which elegantly solves the problem of collisions between a robotic agent and the environment. It is a local method, in that only the next velocity command is determined, however it can incorporate non-holonomic constraints, limited accelerations, maximum velocity, and the presence of obstacles into that determination. It can thus provide for safe motion for a robot in a static domain. The search space is the velocities of the robot's actuated degrees of freedom. Fox et al [38] derived the case of a synchro-drive robots with a linear velocity and an angular velocity, while Brock et al [12] developed the case of holonomic robots with two linear velocities. Both methods use the concept of a "velocity space" where actions can be tested for safety. each point in the velocity space corresponds to reaching that velocity within the control period C, and then executing a stop at a deceleration of D until the agent has come to a complete halt. A velocity can considered safe if the robot can travel up to that command during C and then stop without hitting an obstacle. This corresponds to not hitting an obstacle during C and then not hitting an obstacle during the stop (which traces out a line in world coordinates). It would be difficult to search all of velocity space to find the action that minimizes a cost metric, however, due to limited accelerations, velocities are limited to only a small window that can be reached within the acceleration limits of $A(v)$ over the control cycle. An example of a velocity space with an obstacle and an acceleration window is shown in Figure 5.3 [1].

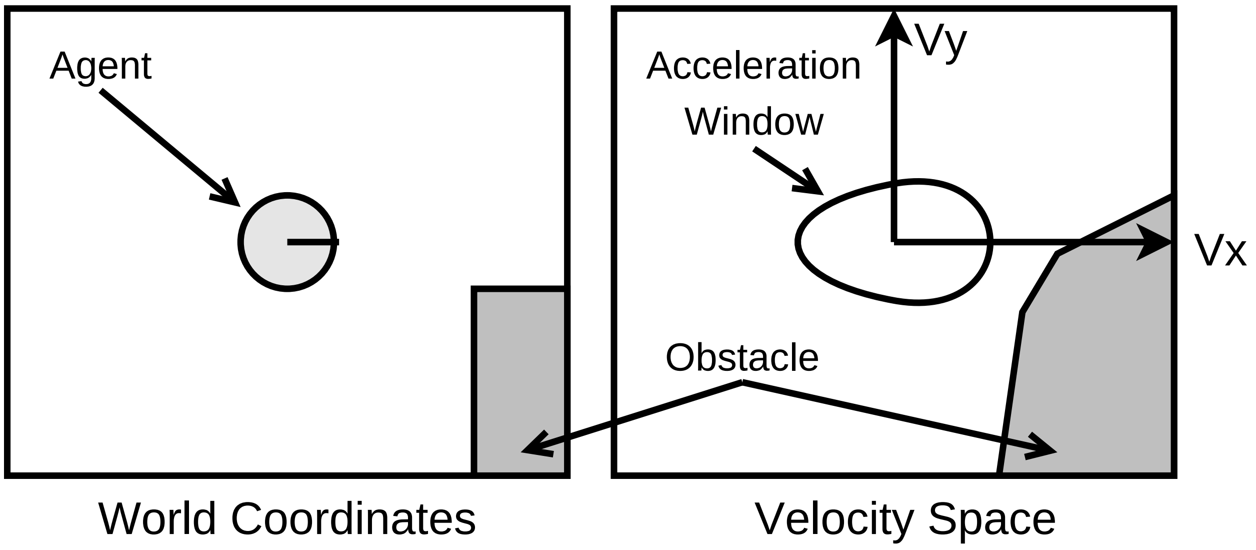


Figure 5.3: Example environment shown in world space and velocity space. Note that this figure is hand-drawn and thus only approximate.

In order to find a safe command, the Dynamic Window method creates a grid over the acceleration window, and evaluates each grid cell with a combination of a safety test and an evaluation metric. The safety test checks if a velocity command would hit an obstacle, resulting in an infinite cost. If the command is safe, then it is evaluated based on heuristics for reaching a desired target (Dynamic Window includes both safety and motion control in a single algorithm). Pseudocode for a variant is given in Table 5.1. The variation is that we have replaced velocity space with a related concept of an acceleration space. The acceleration space is defined using the possible accelerations in the robot model $A(v)$ for one control cycle C, with the results as defined in the transfer functions f and $\dot{f}$. In the code,

[1]Note that the velocity obstacle on the right of the figure was hand drawn rather than calculated, and is thus only approximate

the function *CheckSafetyObs* calculates the position after the control cycle (lines 1-2), and then calculates the stopping deceleration and the time required to stop (lines 3-4). If both of these trajectories are safe, then the command can be executed safely[2]. The main search function, *DynamicWindowSearch* searches a grid for the lowest cost acceleration command that is safe. In Brock et al [12], the Dynamic Window approach was used successfully for a robot moving up to 1m/s in cluttered office environments with dynamically placed obstacles.

```
function CheckSafetyObs($i$:RobotId,$u'$) : Status
1    let $p_1 = f(q_i, \dot{q}_i, u', C)$
2    let $v_1 = \dot{f}(\dot{q}_i, u', C)$
3    let $h = -D\frac{v_1}{\|v_1\|}$
4    let $t_\Delta = \frac{\|v_1\|}{D}$
5    let $p_2 = f(p_1, v_1, h, t_\Delta)$
6    return CheckObsLine($q$,$p_1$,$R+\epsilon$)=Safe $\wedge$ CheckObsLine($p_1$,$p_2$,$R$)=Safe

procedure DynamicWindowSearch($i$:RobotId)
1    let $G$ = SampleUniformGrid($A(\dot{q})$)
2    $e_i \leftarrow \infty$
3    foreach $u' \in G$ do
4        if CheckAccel($i$,$u$)=Safe $\wedge$ Cost($u'$) $< e_i$ then
5            $u_i \leftarrow u'$
6            $e_i \leftarrow$ Cost($u'$)
7        end
```

Table 5.1: The Dynamic Window method for a single agent

5.3 Dynamics Safety Search

Our approach, called *Dynamics Safety Search* or DSS, extends the Dynamic Window approach to multiple velocity and acceleration bounded robots, and replaces the grid-based sampling with a randomized sampling approach which guarantees the preservation of safety if no sensor or action noise is present. It operates for n robots with the model described in Section 5.1. In Table 5.3, Dynamics Safety search is compared against the Dynamic Window

[2]A small positive real number, ϵ, is used to ensure that the parabolic path traced during the control period is included within the straight-line obstacle check. For typical short control cycles $C < 0.1s$, the effect is minimal. Alternatively, a parabolic trajectory check could be implemented by the collision detection library.

approach on which it is based as well as a hypothetical complete motion planner operating on the joint configuration and space defined by the n agents. Compared to Dynamic Window, DSS supports an exact guarantee of safety, and does not require a sufficient resolution in order to find a solution. DSS also supports multiple agents running the algorithm, and supports avoidance of uncontrolled moving obstacles (but without safety guarantees). Finally DSS does not use a fixed grid or resolution, and thus can operate as an anytime algorithm (only constant overhead per agent). Compared to a joint state-space planner, DSS is not complete, and is not guaranteed to find solutions avoiding moving obstacles. However, DSS can operate as an anytime algorithm with $O(n^2)$ complexity, rather than the non-anytime exponential complexity of the planner. If we replace the complete planner with a randomized variant, we can achieve better complexity guarantees but give up both completeness and guaranteed safety. DSS thus occupies a middle ground that is intended to be practical for implementation of multi-robot teams.

	Dynamic Window	Joint Planning	Dynamics Safety Search
Complete	No	Yes	No
Safety guarantee	Resolution	Exact	Exact
Multiple agent support	No	Yes	Yes
Moving obstacle support	No	Yes	Partial
Anytime algorithm	No	No	Yes
Multi-agent complexity	N/A	Exponential	$O(n^2)$

Table 5.2: A comparison of properties of Dynamic Window, explicit planning, and Dynamics Safety Search

The easiest way to understand the DSS algorithm is in a top-down manner. The high-level routines are listed in Table 5.3. The main function *DynamicsSafetySearch*, runs once per decision cycle (not per robot). It starts each iteration by setting each agent's command in u_i as the stopping deceleration if the agent is moving (lines 3-5). This is guaranteed to be a safe action to perform by the previous iteration. An evaluation cost based on the difference between this stop command and the desired acceleration a_i^k is stored in e_i (line 6). This will be used to rank alternatives to find one which most closely matches the desired command. The current approach uses squared Euclidean distance as the metric. Next, the duty cycle for the command, γ_i is set to ensure that the agent comes only to a stop and does not begin to accelerate in the opposite direction (line 7). For small control periods, such as $C < 0.1s$, the effect is negligible, but it is necessary for completeness. In practical implementations, γ_i is almost always equal to C. After this "initialization to stop" phase from (lines 1-7), *DynamicsSafetySearch* then calls a helper function *ImproveAccel* for each agent, which will try to better match the agent's desired action while maintaining safety.

```
function CheckAccel($i$:RobotId, $u'$:Vector) : Status
    if CheckSafetyObs($i$,$u'$)=Unsafe
        then return Unsafe
    foreach $j \in [1, n], j \neq i$ do
        if CheckRobot($i$,$u'$,$j$,$u_j$)=Unsafe
            then return Unsafe
    return Safe

procedure ImproveAccel($i$:RobotId)
    if CheckAccel($i$,$a_i^k$)=Safe then
        $u_i \leftarrow a_i^k$
        $e_i \leftarrow 0$
        $\gamma_i \leftarrow C$
    else
        foreach $j \in [1, m]$
            $u' \leftarrow RandomAccel(\dot{q}_i)$
            if CheckAccel($i$,$u$)=Safe and $\|u' - a_i^k\|^2 < e_i$ then
                $u_i \leftarrow u'$
                $e_i \leftarrow \|u' - a_i^k\|^2$
                $\gamma_i \leftarrow C$
            end
    end

procedure DynamicsSafetySearch()
    foreach $i \in [1, n]$ do
        let $s = \|\dot{q}_i\|$
        if $s > 0$
            then $u_i \leftarrow -D\frac{\dot{q}_i}{s}$
            else $u_i \leftarrow \vec{0}$
        $e_i \leftarrow \|u_i - a_i^k\|^2$
        $\gamma_i \leftarrow \min(\frac{s}{D}, C)$
    foreach $i \in [1, n]$ do
        ImproveAccel($i$)
```

Table 5.3: The high level search routines for velocity-space safety search.

The procedure *ImproveAccel* consists of two major stages. In the first stage, it checks to see if the agent's desired command can be carried out without causing any failure in the safety. It does this by calling the *CheckAccel* function which returns whether or not using a particular acceleration as a command maintains safety for all of the agents. If *ImproveAccel* finds that the desired command can be safely executed (line 1), then it sets that command in u_i (line 2) to be carried out for the duration of the control cycle (line 4). The evaluated cost is zero, since the current command matches the desired command. As a result, further search is not necessary as no better matching command could be found. In practical implementations, it is this short-circuit that results in the efficiency of the algorithm. If the agents are not interfering with each other or close to C-space boundaries, it is normally the case that the agent's desired action will be safe. If however, the desired action cannot be performed, a search is carried out (lines 6-12), which tries to find an acceleration that is safe, but with a lower evaluated cost. In line 7, a random acceleration is sampled from the set of accelerations possible at the current velocity ($A(v)$ in the robot model). This is checked for safety, as well as having a lower cost than the current action (line 8). If both these conditions are met, the action is set and the evaluation cost calculated (lines 9-11).

For the high-level search routines, this leaves the function *CheckAccel.* It returns if a give acceleration is safe, and has a straightforward implementation based on the definition of safety in equations 5.7-5.9. Safety with respect to the environment is handled using the same function as we used in the implementation of the Dynamic Window approach (lines 1-2). Robot agents are handled by checking the new action with each other robot using the function *CheckRobot*, which is described below (lines 3-5). If neither of these checks finds a violation of safety, the action is considered safe (line 6).

In Figure 5.4, and example run of DSS for a single control cycle can be seen. Each agent is a shaded circle, with the desired acceleration shown as an arrow in the first frame. An extruded circle "pill shape" denotes the trajectory of the robot including its action and stopping phases. Because DSS never violates the safety of an existing trajectory, the extruded area can be thought of as a reserved space to stop [3]. The stopping trajectory also indicates the current velocity of the agent, which is along the same direction as the stopping trajectory. To the right of each of the depictions of the environment, a plot of the horizontal velocity with time is given for both agents. In part (a), each agent has its action initialized to stop, and the arrows display the desired acceleration command. The velocity plot shows both agents decelerating to a complete stop. In part (b), agent 1 has chosen a safe action (depicted by the blue arrow) which is as close as possible to the desired action (shown in light gray). The effect on the velocity plot is to accelerate up to time C, and then stop within a new

[3]Although it is a useful approximation to think of these areas as non-overlapping, it is not the case that these areas are necessarily disjoint. It is only the case that at any given point in time, the robots cannot occupy the same circles defined by their safety radius.

reserved stopping area. Agent 1's action was constrained by the rectangular environment obstacle, so it could not match the desired acceleration. In part (c), agent 2 chooses an action depicted by the dark arrow, which matches the desired action as closely as possible while maintaining safety. The effect on the horizontal velocity is shown on the right. Agent 2's action is constrained by the need to avoid hitting agent 1 during the stopping trajectory. In part (d), the agents have advanced by time C and have executed their actions. The remaining trajectory is a valid stopping action. A new iteration of the algorithm can begin. Thus, using DSS attempts to iteratively delay stopping into the future with each iteration, while always maintaining that as an action to fall back on if needed. This is similar to the Dynamic Window approach, but because the stopping action is treated specially, rather than integrated into the grid search as in Dynamic Window, DSS can always guarantee finding a safe action.

We now proceed in describing the lower layers of the algorithm. In Table 5.4, the mid-level functions for DSS are shown. The function *MakeTrajectory* is used to construct a trajectory for an agent starting with a given acceleration u_i. Mathematically it is constructing the piecewise components of the function $q_i^k(t)$. Line 1-3 calculate the time and the resulting position and velocity of executing action u_i, starting from the current state of the robot in x_{i0} and $\dot{x}_{i0}$. The system transfer functions are used to perform the forward prediction of position and velocity (Equations 5.1 and 5.2). In lines 4-6, the results of a stopping action are calculated. h_i is acceleration opposite the agent's current velocity with a magnitude of D, while t_s is the time when the agent will come to a stop. Again the system transfer function is used to compute the position, and the velocity will be zero. Based on the acceleration model of the robot, each of these three phases of motion for the robot (control, stopping, stopped) define a trajectory from t_0 onward, and during each segment the acceleration is constant. Thus the entire trajectory can be modelled as parabolic segments. For this purpose, the parabolic tuple is defined, with the order of members being the position, velocity, and acceleration vectors, respectively, followed by the time interval for which that parabolic motion segment is defined. Lines 7-9 of *MakeTrajectory* construct parabolic tuples for the three phases of motion. These three items are then used the construct a trajectory tuple (Line 10).

Given the function *MakeTrajectory*, and a primitive for checking parabolic segments for collision, the implementation of *CheckRobot* is as follows. Lines 1-2 construct the trajectories for agents i and j. Then each pair of parabolic segments is checked against one another using the primitive checking function *CheckParabolic* (Lines 3-6). The function *CheckParabolic* returns unsafe if two parabolic segments pass within a certain distance of one another (in this case the sum of i and j's safety radii). If no collision is found between any pair of segments, then the actions u_i and u_j are safe in the sense of S_e (Equation 5.7) and the *Safe* status is returned (Line 7).

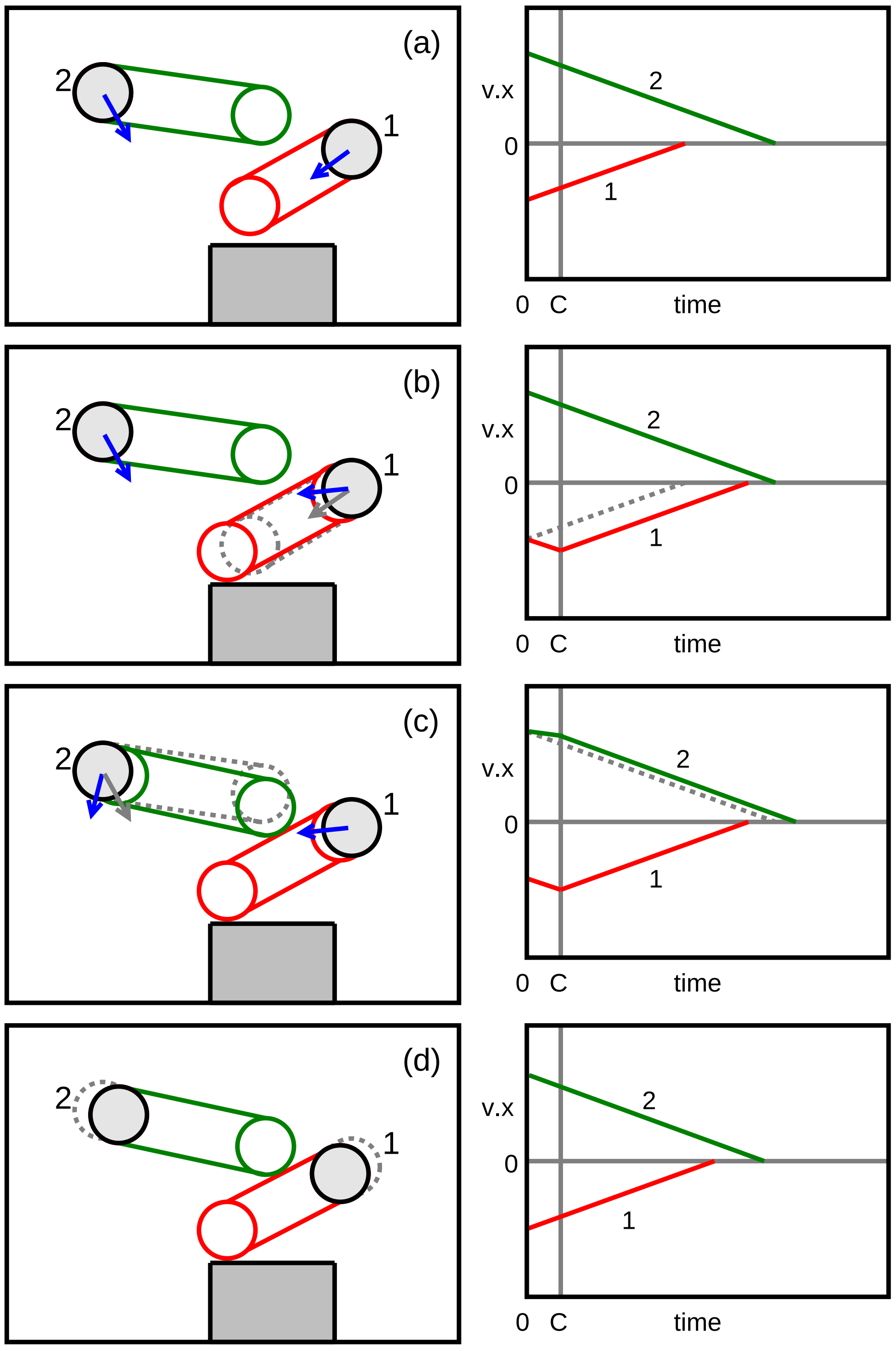


Figure 5.4: An example of an iteration of DSS with two agents. Each agent starts by assuming it will stop (a), and then each agent chooses an action (b)-(c), while making sure the action will allow a safe stop afterward. Finally, the actions are executed (d) and the agents can safely assume that stopping is a valid action.

```
tuple Parabolic = (Vector * Vector * Vector * [Time,Time])
tuple Trajectory = (Parabolic * Parabolic * Parabolic)

function MakeTrajectory(i:RobotId, u_i:Vector) : Trajectory
    let $t_c = t_0 + C\gamma_i$
    let $x_{i1} = f(x_{i0}, \dot{x}_{i0}, u_i, t_c - t_0)$
    let $\dot{x}_{i1} = \dot{f}(\dot{x}_{i0}, u_i, t_c - t_0)$

    let $h_i = -D\frac{\dot{x}_{i1}}{\|\dot{x}_{i1}\|}$
    let $t_s = t_c + \frac{\|\dot{x}_{i1}\|}{D}$
    let $x_{i2} = f(x_{i1}, \dot{x}_{i1}, h_i, t_s - t_c)$

    let $P_{i0} = Parabolic(x_{i0}, \dot{x}_{i0}, u_i, [t_0, t_c])$
    let $P_{i1} = Parabolic(x_{i1}, \dot{x}_{i1}, h_i, [t_c, t_s])$
    let $P_{i2} = Parabolic(x_{i2}, 0, 0, [t_s, \infty])$
    return Trajectory($P_{i0}$,$P_{i1}$,$P_{i2}$)

function CheckRobot(i:RobotId, u_i:Vector, j:RobotId, u_j:Vector) : Status
    let ($P_{i0}$,$P_{i0}$,$P_{i0}$) = MakeTrajectory($i$,$u_i$)
    let ($P_{j0}$,$P_{j1}$,$P_{j2}$) = MakeTrajectory($j$,$u_j$)

    for $a = 0$ to 2 do
        for $b = 0$ to 2 do
            if CheckParabolic($P_{ia}$, $P_{jb}$, $R_i + R_j$) = Unsafe
                then return Unsafe
    return Safe
```

Table 5.4: Robot-robot checking primitive for safety search.

The pseudocode for the primitive function *CheckParabolic* is shown in Table 5.5. After assigning names to the components of the parabolic tuples (lines 1-2), the function calculates the overlapping time interval of the trajectories (line 3). If there is no overlap in the time intervals, there is by definition no time at which a collision could occur, thus the parabolic segments are safe with respect to one another (line 4). In lines 6 and 7, trajectory functions are defined (matching the system transfer function f in the robot model). Next, a fourth-order polynomial "clearance" function $d(t)$ is defined, which is equal to the squared distance between the parabolic function minus the squared total radius. Any value of $d(t)$ less than zero over the interval T indicates a collision, a zero value indicates constant, and the absence of any such values indicates a safe trajectory. Thus the last block of the function (lines 9-15) implements the classical function minimization technique for differentiable functions over an interval. First, lines 9-10 check the boundaries of the interval, and next the real roots of the derivative of $d(t)$ with respect to t are determined. Any root within the time interval T with a non-positive value of $d(t)$ results in the status *Unsafe* being returned (lines 12-14). If none of the extrema are unsafe, the function returns a status of *Safe* (line 15). The approach taken in *CheckParabolic* defining $d(t)$ is similar to the "relative velocity" method of Fiorini et al. [36] for an agent planning among multiple moving obstacles.

```
function CheckParabolic($P_1$:Parabolic,$P_2$:Parabolic,$r$:$\mathbb{R}$) : Status
1    let $(x_1, \dot{x_1}, u_1, [t_{1a}, t_{1b}]) = P_1$
2    let $(x_2, \dot{x_2}, u_2, [t_{2a}, t_{2b}]) = P_2$
-
3    let $T$ = Intersection($[t_{1a}, t_{1b}]$,$[t_{2a}, t_{2b}]$)
4    if $T = \emptyset$ then return Safe
5    let $[t_a, t_b] = T$
-
6    let $p_1(t) = x_1 + \dot{x_1}(t - t_{1a}) + \frac{1}{2}u_1(t - t_{1a})^2$
7    let $p_2(t) = x_2 + \dot{x_2}(t - t_{2a}) + \frac{1}{2}u_2(t - t_{2a})^2$
8    let $d(t) = \|p_1(t) - p_2(t)\|^2 - r^2$
-
9    if $d(t_a) \leq 0$ or $d(t_b) \leq 0$
10       then return Unsafe
11   let $M$ = RealRoots(Deriv($d(t)$,$t$),$t$)
12   foreach $t \in M$ do
13       if $t \in T$ and $d(t) \leq 0$
14           then return Unsafe
15   return Safe
```

Table 5.5: The parabolic trajectory segment check.

Overall, the Dynamics Safety Search algorithm is a fairly involved, but highly modular approach. Each of the subfunctions has a well defined interface and semantics. While this method is not complete in a kinodynamic planning sense, it scales polynomially with the number of robots, and can guarantee safety among multiple moving agents with realisitic motion constraints. The achievement of objectives is difficult to analyze, since DSS uses a reactive and oppurtunistic method for achieving goals within its safety assumptions. However when paired with a path planner which already lacks completeness, the loss may not prove as problematic. In particular, the assumptions of DSS fit well with applications where achievement of task objectives is secondary to safe operation.

5.4 Guarantee of Safety

Ultimately, we want to satisfy the notion of safety described by Equation 1.2 from Chapter 1.3.2. However, that definition of safety is "weak" in that it only considers the position at the current time, and thus it is difficult to show that this definition is met while dynamics constraints are respected. Thus DSS defines the stronger definition of safety embodied in Equation 5.9, where safety is modelled over an entire future trajectory from the current time forward. The DSS safety guarantee applies to agent bounded by a radius, rather than the more general set theoretic definition in Equation 1.2. Within the case of such a circular or spherically bounded agent however, the DSS guarantee of Equation 5.9 is strictly inclusive of the weak definition.

DSS maintains safety by treating Equation 5.9 as an invariant during all operations on the world state. The statement of safety embodied in it defines exactly the kind of safety we would like an algorithm to provide; at all times it maintains a future trajectory that can safely be executed indefinitely. Of course, if a problem instance for DSS starts out as unsafe, there is no guarantee that DSS can return the system to safety. Thus the guarantee we can seek from the safety search method is that it can *maintain* safety if started from any safe situation. In particular, we wish to demonstrate that:

Theorem 5.4.1. *For n robots given the model from Section 5.1, if $S(k, t_0)$ holds, then after time C, and the execution of DynamicsSafetySearch, then $S(k + 1, t_0 + C)$ holds.*

Due to the complexity of DSS, a full proof of the algorithm would be very long, thus the proof is only sketched here in enough detail to justify the claims of safety. Operations on the world state for DSS fall into two main categories, both of which maintain the strong safety invariant:

- The passage of time C maintains the safety invariant if new actions are set to stopping the agent
- Any modification of actions performed by *ImproveAccel* maintains the safety invariant

These two operations cover everything occurring in the top level procedure *DynamicsSafety-Search*. For the first type of system transition, the passage of time is governed by the system transfer functions (Equations 5.1-5.2. Immediately following this, the first part of *DynamicsSafetySearch* sets a stopping action. The other type of transition is a "decision cycle" where the latter part of *DynamicsSafetySearch* calls *ImproveAccel* on each agent. If both of these operations maintain safety individually, the safety invariant overall is maintained. We will first focus on time passage, and then follow with action modification.

In order to show that advancing time does not cause the safety invariant to fail, the critical point is that trajectory $q_i^{k+1}(t)$ follows the path of $q_i^k(t)$ after time C has passed, allowing the safety at iteration k to imply safety at the next iteration. In other words, given a trajectory function q_i^k defined as the following:

$$q_i^k(t) = \begin{cases} q_i^k(t,0) & \text{if } t \in [t_0, t_c] \\ q_i^k(t,1) & \text{if } t \in [t_c, t_s] \\ q_i^k(t,2) & \text{if } t > t_s \end{cases} \tag{5.10}$$

where $t_c = t_0 + C$, and the individual components of q represent the parabolic segments P created in *MakeTrajectory*, we want to show that:

$$q_{i+1}^k(t) = \begin{cases} q_i^k(t,1) & \text{if } t \in [t_c, t_s] \\ q_i^k(t,2) & \text{if } t > t_s \end{cases} \tag{5.11}$$

The can be shown to be the case because the calculations in line 1-3 of *MakeTrajectory* exactly mirror the system transfer functions f and $\dot{f}$. As a result, $q_{i+1}^k(t)$ follows the "tail" of $q_i^k(t)$ for any time after $t_0 + C$. With this fact holding for all robots, and the assumption that all other obstacles are static, any safety guarantees for the original trajectory functions will carry over. Thus, for any trajectory constructed in the previous frame by *MakeTrajectory*, the passage of time will maintain the safety invariant. Of course, if a new action is not chosen, we must show that the stopping action from *DynamicsSafetySearch*. The critical point there is that lines 1-3 construct a function that matches Equation 5.11, with the the action set to the derivative of $q_i^k(t, 1)$. Thus, overall the safety invariant in maintained by advancing time.

Next, we argue that *ImproveAccel* maintains the invariant. This is straightforward, as both cases where an action is set in the function *ImproveAccel* are guarded by conditional statements asserting *CheckAccel* returns a status of *Safe*. The function *CheckAccel* can be shown to mirror the definition of S in Equation 5.9. Both parts of S are expressed, as lines 1-2 matches S_e (Equation 5.7) while lines 3-5 matches S_r (Equation 5.8)

DSS maintains the safety invariant from Equation 5.9 at all times, and thus satisfies the weaker definition of safety at any time instant expressed in Equation 1.2. This guarantee rests on the assumption that all moving agents are participating in the algorithm, and that the obstacles in the configuration space are otherwise static. Also, there can be no sensory error in the positions and velocities, and no action error as expressed in the system transfer functions (Equations 5.1-5.2).

5.5 Improving Efficiency

For a practical implementation, a few approaches can lead to significant gains in the execution speed of the algorithm. The first approach is to cut down on the number of parabolic segments that need to be checked against one another as a nested loop in the function *CheckRobot*. In a typical situation for a pair of robots, the plot of speed versus time would look similar to Figure5.5. In the common case where $\gamma = 1$, the control periods for the two agents are identical, so the first two parabolic motion segments must be checked against one another, but will not overlap with any other segments. After that, both agents will be decelerating, so the "stopping" parabolic segments will need to be checked against one another. Finally, the second agent to come to a stop much check its stopping parabolic segment against the "stationary" segment of the other agent. After this time, both agents are stopped, and the status of their safety will not change. Using this approach, the number of parabolic pairs that need to be checked drops from nine to three. Even in the case where $\gamma < 1$, a similar approach means that only four checks need to carried out.

Another helpful approach is to use broad-phase collision detection techniques as explored in Chapter 3, and in particular the Extent Masks approach described in 3.2.1. A bounding box can be associated with each trajectory tuple, and *CheckRobot* can first check for overlap between the bounding boxes before proceeding with a trajectory check. In cases with many agents, where only a few pairs of agents are within close proximity, this can result in a large speedup. Taking the broad-phase approach further, based on the structure of $A(v)$ one

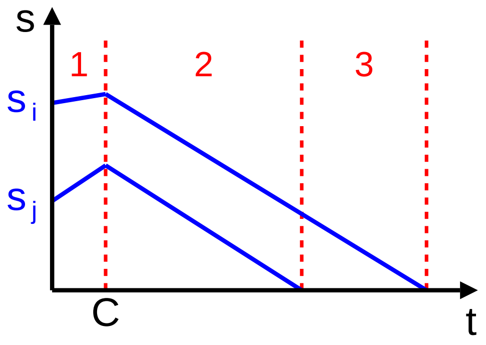


Figure 5.5: Example velocity profile of two agents i and j. Each agent starts at a distinct velocity and executes a control acceleration for time C, and then comes to a stop using deceleration D. This defines three segments of relative motion, each with a constant acceleration.

could construct a bounding box which included *all possible* trajectories, or a box B_i such that for any $u \in A(\dot{x}_{i0})$, the resulting trajectory function $q_i'(t)$ lies within B_i. Any agent or obstacle outside this bounding box need never be checked for safety. The agents and obstacles overlapping the box could be placed in a "potentially colliding set" calculated in the function *ImproveAccel*, placing it outside the search loop for accelerations. This optimization would likely result in a large speedup for domains with many agents.

In the implementation of DSS used in the evaluation and application of the algorithm, both the parabolic pairs optimization and the trajectory bounding box optimization were incorporated. The potential colliding set optimization was not implemented due to the more invasive changes required for the algorithm and supporting libraries, as well as the sufficient performance of the current implementation.

Another question one can ask is what the minimum complexity of the general DSS approach could be. For n agents, m obstacles, and k random samples in the acceleration search, the worst-case complexity is of *DynamicsSafetySearch* $O(kmn^2)$. This results from the complexity of *CheckAccel*, which is $O(mn)$, which is called up to k times in *ImproveAccel*, which is thus $O(kmn)$. The n calls to *ImproveAccel* by *DynamicsSafetySearch* leads to the final complexity. The complexity of the top level procedures derive directly from the approach itself, leading the focus to trying to improve *CheckAccel* is possible. The scaling with obstacles is subject to the problems discussed in Chapter 3, however improving the n factor would be helpful for large numbers of agents by removing the n^2 factor. However, if the algorithm is called with an environment like that shown in Figure 5.6, no spatial data structure based on bounding volumes on trajectories will allow a sub-linear number of checks for a single agent. Thus, the complexity cannot be improved within the bounds of the DSS approach unless addition constraints on parameters or environments are made. In practice however, with more uniform distributions of agents and obstacles, performance has been found to be adequate. This will be explored in detail in the next section.

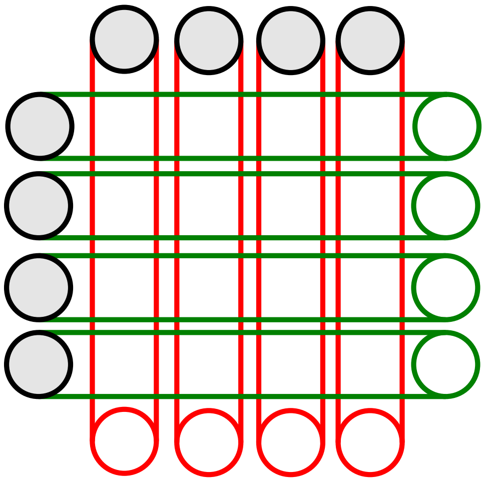


Figure 5.6: An example situation showing n agents with $\Omega(n^2)$ overlapping trajectories.

5.6 Evaluation and Results

The Dynamic Safety Search algorithm was implemented as a C++ object following the approach presented in the pseudocode, but with the optimizations described in Section 5.5. It was first tested in a simple simulator of an environment similar to the RoboCup small size league, but with more complicated environmental obstacles. It was subsequently added to the CMRoboDragons system in 2005 and an updated version in the CMDragons 2006 team.

5.6.1 Simulation Evaluation

The evaluation domain consists of up to ten simulated robots modelled after idealized RoboCup small size robots. The task is to alternately achieve goals on the left and right side of the environment with several obstacles. The domain used for testing is shown in Figure 5.7. The state pictured in the figure is just at the beginning of a test run, with the robots represented as filled circles and their respective goals represented as outlined circles. The straight line indicates the motion control target, and the remaining jagged path is the unoptimized result from an ERRT planner. The stop trajectory is represented as before, as a swept circle, but it is not visible in the figure because the agents are not yet moving. In all the tests, the robots have a diameter of $90mm$ and environment is $5m$ by $4m$. Each robot agent has a command cycle of $C = 1/60$ sec, a maximum velocity of $2m/s$. The maximum acceleration is $F = 3m/s^2$, and the deceleration is $D = 6m/s^2$, and $A(v)$ is modelled as a partial ellipse (see Figure 5.2).

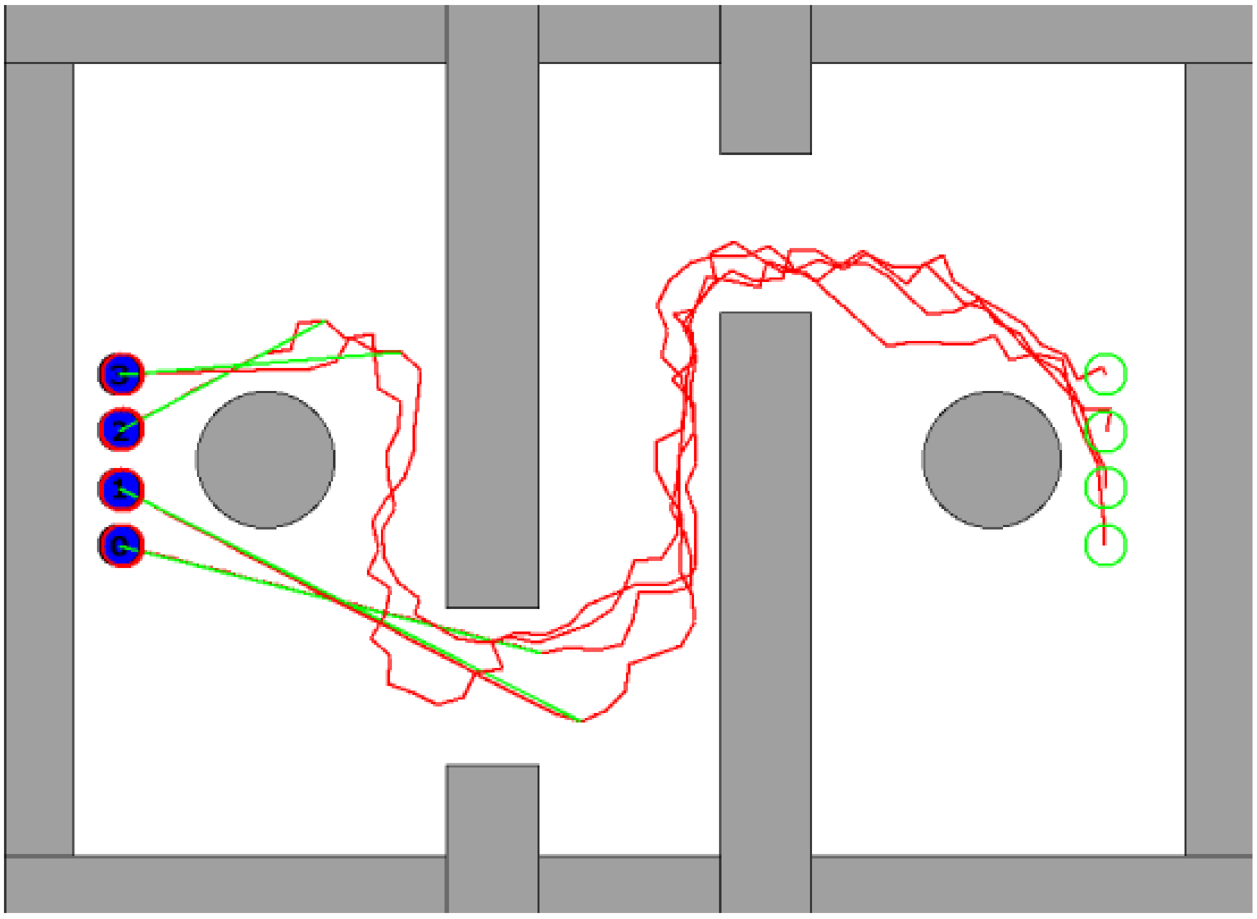


Figure 5.7: The evaluation environment for the DSS algorithm.

For the collision avoidance experiments, four robots were given the task of traveling from the leftmost open area to the rightmost open area, and back again for four iterations. Each robot has separate goal point separated from the others by slightly more than a robot diameter. Because the individual robots have differing path lengths to traverse, after a few traversals robots start interacting while trying to move in opposed directions. Figure 5.8 shows an example situation in the middle of a test run. On average, four full traversals by all of the robots took about 30 seconds of simulated time.

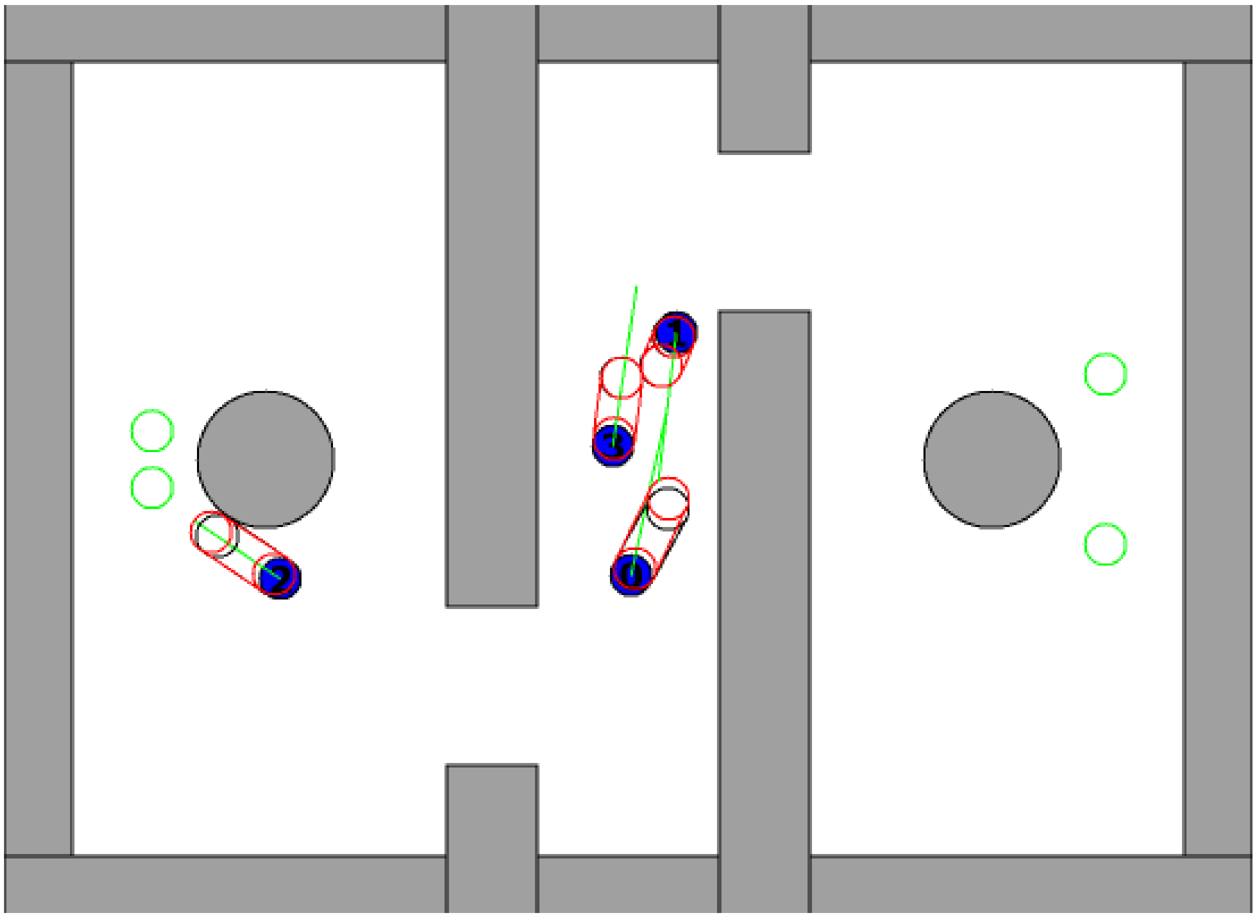


Figure 5.8: Multiple robots navigating traversals in parallel. The outlined circles and lines extending from the robots represent the chosen command followed by a maximum rate stop.

For the evaluation metric, we chose interpenetration depth with obstacles multiplied by the time spent in those unsafe states. This captures both the frequency and severity of collisions in a single metric. To more closely model a real system, varying amounts of position sensor error were added, so that the robot's reported position was a Gaussian deviate of its actual position [4]. This additive random noise represents vision error from overhead tracking systems. It also stresses the system by testing the random search; with Gaussian noise the stopping action is not guaranteed to be safe, and thus a search is required to return the system to safety. Velocity sensing and action error were not modelled in the simulation for simplicity; these errors depend heavily on the specifics of the robot and currently lack an accurate model with wide applicability.

For the first test, we compare two options which both use ERRT and a motion control system, but enable or disable the safety search. Each data point is the average of 40 runs (4 robots, each with 10 runs), representing about 20 minutes of simulated run time. The results are shown in Figure 5.9. It is evident that the safety search significantly decreases the total interpenetration time. Without the safety search, increasing the vision error makes little difference in the length and depth of collisions. Ideally the plotted curve without safety search would be smooth, but due to the random nature of the collisions it displays extremely high variance, and many more runs would be needed to demonstrate a dependence on vision noise. However, even with the noise, it is clear that the curve is significantly worse than when safety search is used. Next, we evaluated only the system with safety search enabled, but using varying extra margins of $1 - 4mm$ around the $90mm$ safety radius of the robots, plotted against increasing vision error (see Figure 5.10). As one would expect, with little or no vision error even small margins suffice for no collisions, but as the error increases there is a benefit to higher margins for the safety search, reflecting the uncertainty in the actual position of the robot. Thus this supports adding an extra margin of safety around the robots based on the expected noise. Such an approach was adopted for the real CMDragons robots.

The other variable of interest is the cost in running time of planning and the safety search. In the tests above, the ERRT planner was limited to 1000 nodes, and the safety search was limited to 500 randomly sampled velocities. The system executed with an average run time of $0.70ms$ per control cycle without the velocity safety search, and $0.76ms$ with it. Thus safety search does not add a noticeable overhead to the navigation. Since this is a real-time system however, we are most interested in times near the worst case. Looking at the entire distribution of running times, the 95th percentiles are $1.96ms$ without safety search and $2.04ms$ with it. In other words, for 95% of the control cycles, runtime was less than $2.04ms$ with safety search, and the execution time was only 4% longer than without safety search. Thus, for systems with an existing execution time budget for path planning, adding the

[4]A description of the CMDragons vision system, including experiments to determine the error model and the appropriate magnitudes appear in Appendix A. The particularly relevant section is A.2

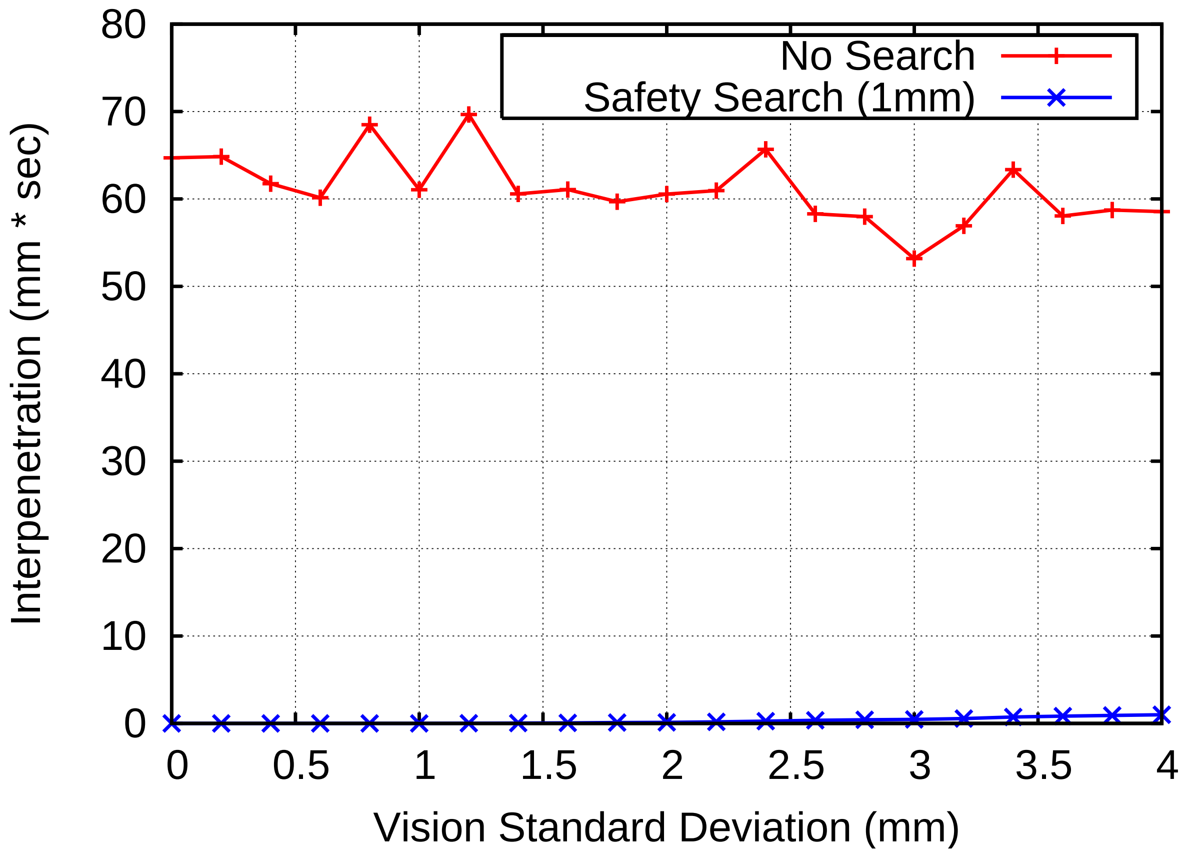


Figure 5.9: Comparison of navigation with and without safety search. Safety search significantly decreases the metric of interpenetration depth multiplied by time of interpenetration.

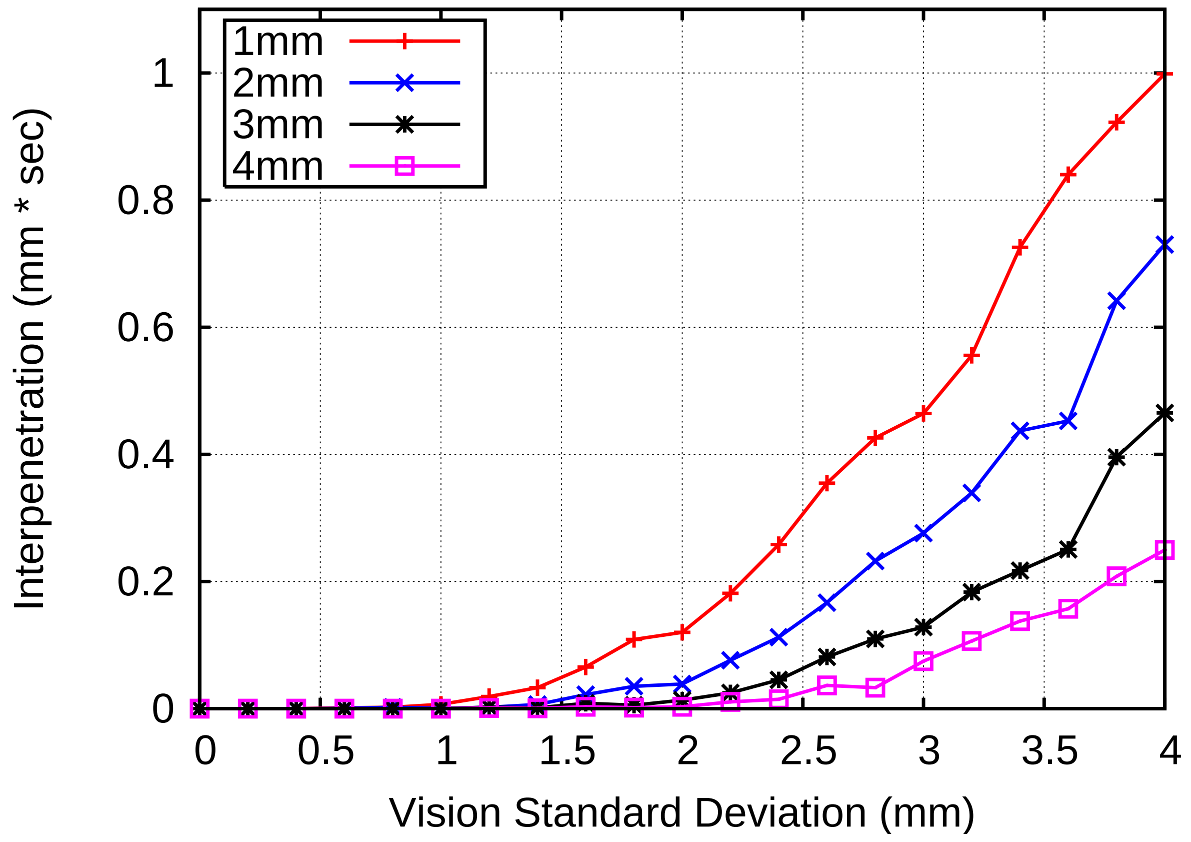


Figure 5.10: Comparison of several margins under increasing vision error. The four different margins used are listed in the key, while increasing vision standard deviation is plotted against the collision metric of interpenetration depth multiplied by time of interpenetration.

safety search is a small additional overhead.

Next, to measure the scalability of the safety search approach, the same traversal task was repeated while varying the number of robots from 1 to 10. With increasing numbers of agents in a fixed-size environment, we can hope to gauge how well the algorithm performs under increasing amounts of clutter due to moving objects. The timing results for safety search are shown in Figure 5.11. The function appears to scale in a roughly linear fashion for more than one agent, though it is too noisy to determine with any certainty. The most important observation is that it does not scale in a particularly super-linear fashion, which would cause difficulties for moderately large teams. As shown earlier, the worst case for DSS is $O(n^2)$, but such cases do not appear to arise in the experiment. The runtime cost of DSS demonstrates that it is applicable to control of agents at high rates of replanning. An equivalent method using joint state-space planning would need to encode at least position and velocity, resulting in a 40 dimensional problem for ten robots. The author is not aware of any current method which could approach 60 Hz replanning in the joint state-space.

5.6.2 Real Robot Evaluation

On the physical robots, objective measurement has proven difficult, although we have qualitatively noted that the frequency of collisions between teammates goes from several times a minute to once every several minutes, for regular soccer play, while it drops very significantly for tasks with highly conflicting navigation goals. In order to demonstrate the effect of DSS, a test domain was created where two pairs of robots would swap positions in a 2.8m traversal across a RoboCup field. An image sequence in Figure 5.12 shows the robots traversing the field without additional obstacles, while Figure 5.13 shows a traversal with five static obstacles in the environment. A video of this test is available on the supplemental materials web page [21]. During 60 seconds of testing, a only a single collision occurred between the moving robots. Minor contact also occurs between a robot and a static obstacle in the second stage of the test. By comparison, this test could not be run with DSS disabled due to the risk of damage to the robots. In limited testing with a decreased speed, the robots could not complete a single traversal without a collision while DSS was disabled.

A second application of DSS was as a post-process to a user tele-operation program. The driving program allows a user to set the target velocity for a robot using a joystick. DSS can be used as a post-process to the user specified command to maintain safety while trying to achieve the target velocity. This allows a safe method for tele-operation, even for novices [5].

[5]Having a novice operator damage a robot during tele-operation at a demo was one of the original motivations for implementing DSS

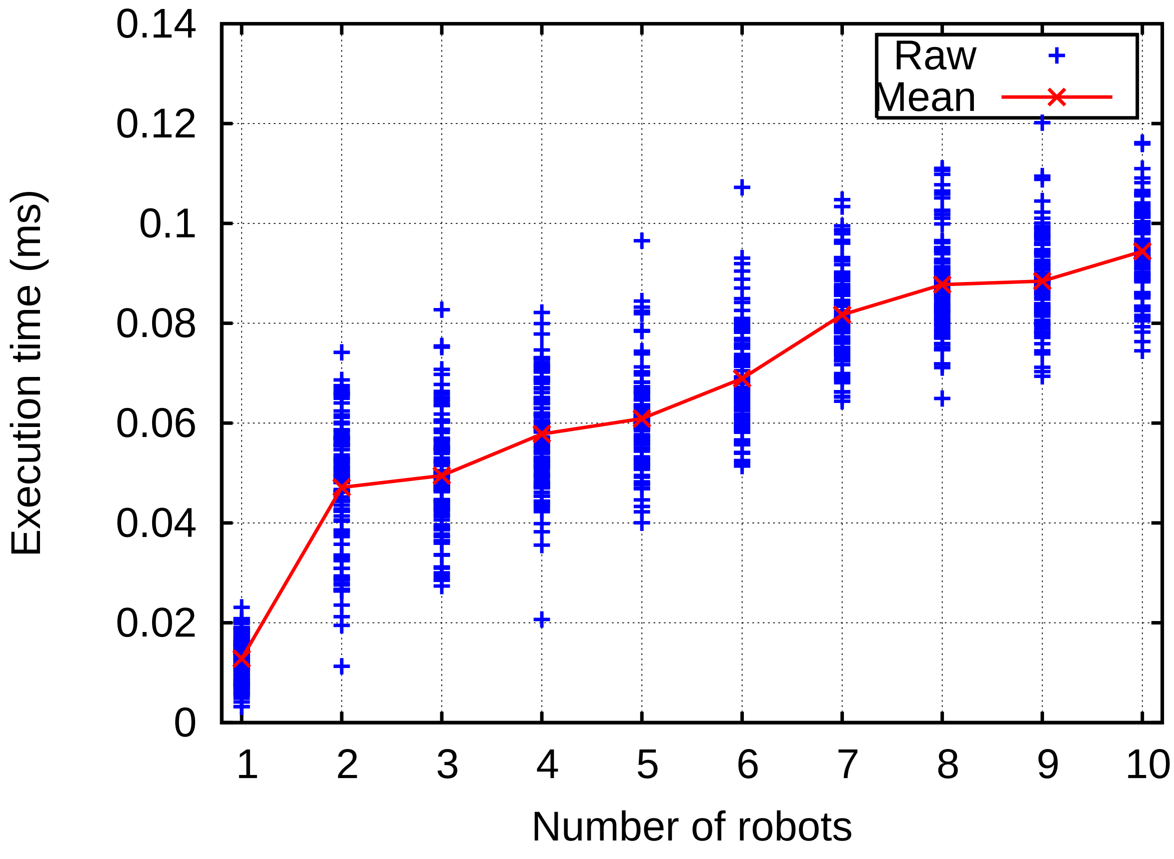


Figure 5.11: Average execution time of safety search for each agent, as the total number of agents increases. For each robot count, 100 trials of the left-right traversal task were run. Both the raw data and means are shown.

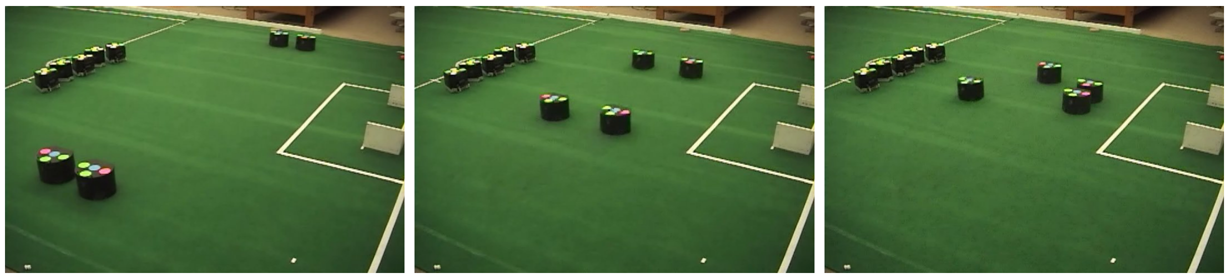


Figure 5.12: A sequence captured while running DSS with four CMDragons robots. All agents completed their 2.8m traversal within 3.1 seconds.

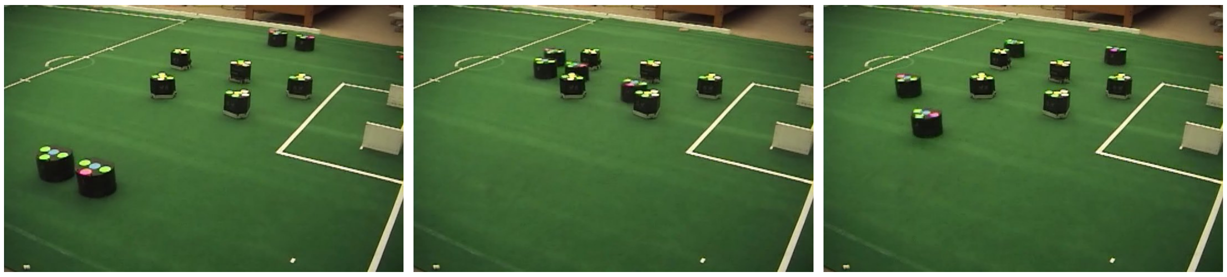


Figure 5.13: A sequence captured while running DSS with four CMDragons robots and five small static obstacles. All agents completed their 2.8m traversal within 2.8 seconds.

A video of this system is available in the supplemental materials [21]. At first, DSS is enabled and the user navigates among a field of static obstacles, with minimal avoidance required on the part of the user. In the second half of the video, DSS is disabled and an expert operator attempts to replicate the motions while avoiding obstacles. With DSS enabled, one collision occurs during 45 seconds of operation, while with DSS disabled, 18 collisions occur during the same time period.

In the RoboCup competitions, there is one impartial source of measurement for the safety of navigation systems. In the small-size league rules, a penalty can be called by a referee against a robot for pushing or hitting an opponent excessively, as well as for a non-goalie robot entering the team's own defense area. The DSS algorithm has been used by our teams for the past two years in the international RoboCup competition; An early version of the DSS algorithm was adopted for the 2005 team entry CMRoboDragons, and the version as described in this chapter was adopted for the 2006 entry CMDragons. In two years of play (13 games), our team received only a single navigational penalty, which occurred while DSS was disabled [6]. During this time, our team finished fourth and first in the tournament. None of the first place teams in 2003-2005 were able to complete the tournament without a pushing penalty.

In terms of execution time, the DSS algorithm has proved efficient when applied on the real robots. When applied during either the traversal task or soccer gameplay, the safety search contributed to less than 1% to overall navigation execution time. This is because the in the more open RoboCup environment, close proximity between agents and with obstacles is rare, allowing DSS to avoid its random search in most cases.

Although the algorithm seeks to guarantee that no collisions occur, on the real robots collisions do still arise. The remaining collisions generally appear to be a result of our imperfect

[6] A behavior for attempting to steal the ball from an opponent mistakenly disabled DSS while driving backwards with the ball, and resulted in a penalty when the robot entered its own defense zone.

model of the robots while operating at high speed, and the resulting errors in tracking. This is particularly true for latency, which while nominally $100ms$, varies up to a whole cycle due to occasional radio packet transmission errors, and more frequently up to $10ms$ due to variations in scheduling and processing time on the controlling host computer. A robot travelling at $1.8m/s$ covers $30mm$ in $10ms$, thus resulting in “glancing” collisions such as occur in the traversal video.

5.7 Conclusion

This chapter described a novel real-time control system for multiple robots acting cooperatively while moving near the limits of their dynamics. The Dynamic Safety Search (DSS) algorithm has the following properties:

- It extends the Dynamic Window Approach to multiple cooperating agents
- It can guarantee safety for robots which operate without any error
- It does not guarantee completeness
- It allows agents to change goals every control cycle
- It scales at $O(n^2)$ with the number of agents, and linearly in practice
- It works well on real robots even with modelling error
- It can be used to create an intuitive safe tele-operation system

It is hoped that DSS offers a practical solution for the safe centralized control of multiple coordinating agents. The primary contribution is to demonstrate theoretical and practical safety for a class of robot systems, to serve as a successful model and example for similar problem domains, and as a starting point for future work which relaxes some of the assumptions. It is hoped that the algorithm can be extended in the future, in particular for distributed control of multiple agents. While the current solution is centralized, relying on perfect communication of world state and actions, the system purposely does not rely on additional communication other than the broadcast of world state and actions. Thus communication is bounded and linear in the number of agents, and in particular, no communication is required for deliberation or other explicit coordination.