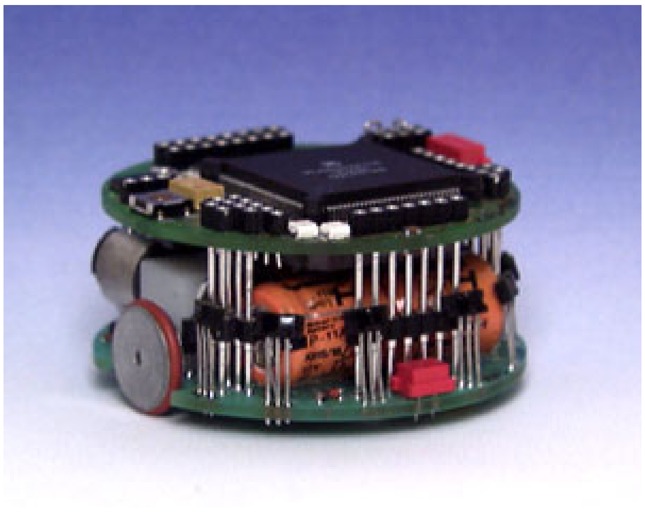
EPFL – Semester project

Distributed hierarchical vehicle control & semi-autonomous collaborative driving



Piyawat Kaewkerd

Assistant: Dr. Philippe Müllhaupt
Professor: MER. Denis Gillet

Semester 7: winter 2005-2006

Lausanne, 14 February 2006

Abstract: different strategies, either collaborative or noncooperative, for collision avoidance between multiple autonomous vehicles are presented in this report. Game theory is the main tool used to find these strategies. Simulations of the trajectories are performed to investigate the limitations of each strategy, and their results are shown. The possibilities of applying these strategies for different domains, such as air, ground and sea transportation, or mobile robotics, are discussed.

Table of contents

1 Introduction

Nowadays, most of the research done in the domain of autonomous mobile robotics is concentrated on the localization of robots, their methods for dealing with cluttered environment or their learning algorithms. One can predict that in near future, with the increasing demand on mobile robots (more than 10% growth the last five years) and the new research trend of distributing tasks to multiple robots (swarm robots), not only the humans will have to learn how to cohabit with their creations, but also the robots themselves will have to find out how to react with one another. This is not only valid for the new generation of domestic appliances such as vacuum cleaners or lawn mowers: cars, for instance, are becoming more intelligent and not solely for research purpose. Indeed, Mercedes-Benz has commercialized in 2005 the first car that makes the break and gas pedals useless on highways, as it automatically keeps a constant distance with the preceding car. In 2006, Honda announced the future release of a new model of cars with automatic lane tracking that also makes the steering wheel archaic on expressways. In a not so distant future, vehicles will be completely autonomous, and, as the mobile robots, will have to learn how to deal with each other.

This semester project proposes some solutions for a small part of these problems, as its aim is to investigate strategies for collision avoidance between autonomous vehicles. To be more accurate, each of the vehicles has to reach its goal as fast as possible while avoiding collision. The principal tool for finding these strategies is game theory, which is a powerful multiple-agent analysis instrument. The proposed strategies may either be cooperative or not, and their advantages and limitations, like the safety or the needed sensor range, are studied. Simulations show the predicted vehicles behavior and allow us to see possible caveats. The possibility and benefits of applying any of these strategies on real systems such as cars, aircrafts, boats or mobile robots are discussed. An objective of this project was to test these strategies on a platoon of small robots, but it has been redefined and, finally, due to the unavailability of those robots and to the lack of time, only simulations on MATLAB are required.

In this report, some general theory and one-vehicle results which are important for the comprehension of the remainder of the work are presented first. Then, the different tested strategies are explained, their simulation results shown, and their advantages, disadvantages and possible applications discussed. After that, the practical implementation of the simulations is discussed, for those who have an interest in doing similar work. The two most important MATLAB functions used for this project are explained, along with the simulations codes: the issues encountered during the programming process are presented and their solutions clarified. Finally, conclusions on the strategies and a discussion on possible future advancements in the domain are given. All the codes written for this project are appended.

2 General theory and preliminary results

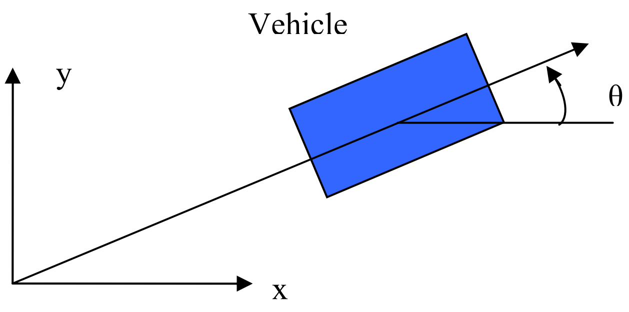
Before the collision avoidance strategies and their results are shown, the important theories and equations used throughout our work are presented and explained. The vehicle dynamics, for instance, are used to compute the vehicles trajectories and their optimal commands. Game theory is our main tool to find and simulate strategies. In the last paragraph of this section, some preliminary one-vehicle simulations are shown in order to best present the two-vehicle case afterwards.

2.1 Vehicle dynamics

We used the following state-space model for the vehicles dynamics:

$$\begin{aligned}\dot{x}_1 &= v_1 \cos(x_3)\\ \dot{x}_2 &= v_1 \sin(x_3)\\ \dot{x}_3 &= v_2\end{aligned} \qquad \text{(Eq. 1)}$$

with the inputs (also called control variables or commands) v_1 and v_2, which represent respectively the speed and the steering. x_1 and x_2 correspond to the x and y coordinates of the position of the vehicle. x_3 is the angle theta between the x-axis and the direction of the vehicle. In our simulations, a vehicle is supposed to be round, and its radius is the only variable needed to define its physical dimension.



2.2 Game theory basics

Game theory, made famous by Nash, is the main research area that tries to provide answers to noncooperative multiple-agent problems. With this tool, it is possible to find the optimal strategies for numerous problems, including pursuit games and battle games. The main applications are found in macroeconomics (this is why Nash received the Nobel Prize in economics) and warfare (rocket pursuing an aircraft, etc.). In our work, we mostly considered the two-player case using Isaacs' theories [1].

In these theories, the players have conflicting goals, and the outcome of their battle is represented by a cost function, also called *payoff*. The optimal strategy for each player is determined by the payoff, which one player strives to minimize while the other one tries to maximize it. For instance, in a pursuit game, the payoff can either be the distance between the two players or the time until capture. In this particular example, the evader is the one that will try to increase the payoff as much as possible and the pursuer is the one that will try to minimize it.

2.2.1 Important game theory equations

The kinematic equations are the differential equations that define the dynamics of the players and relate them to the control variables:

$$\dot{x} = f(x, \phi, \varphi) \qquad \text{(Eq. 2)}$$

x is the state vector, ϕ and φ are the control vector of the first, respectively the second, player. We can see that the kinematic equations describe the dynamics of both players using only one state vector. In our case, the kinematic equations are the concatenated equations of the state-space representation of the two vehicles (Eq. 1), the vector $[v_1, v_2]^T$ of each vehicle being its corresponding control vector ϕ or φ. The states of each vehicle are in the same vector, the three states of the first vehicle being followed by those of the second vehicles. Thus, the problem is in a six-dimension space, but it can be reduced to a three-dimension space, if we only take into account the relative positions of the vehicles, with the referential fixed to one vehicle. However, we did not use a reduced space, because that would make the implementation much harder, with displacing goals whenever the vehicles move, and it would not yield intuitive an easy-to-understand results, as we are more accustomed to seeing results in the world referential.

The standard form of the payoff is:

$$Payoff = \int G(x, \phi, \varphi)dt + H(s) \qquad \text{(Eq. 3)}$$

The lower limit of the time integral refers to the starting point (we can call it t=0) and the upper limit to the termination time (time of capture for example). The function H(s) depends only of the terminal states. If H=0, we have an integral payoff; if G=0, we have a terminal payoff. For instance, in a pursuit game, the time until capture would be an integral payoff (with G=1), whereas the distance between the vehicles after a given time would be a terminal payoff (with H being the distance between the vehicles at termination time).

A *strategy* defines the sets of decisions for a player at each position. Thus, if the strategies of the two players are known, the value of the payoff at the next time (in the case of a discrete game) is uniquely determined. If both players have optimal strategies, the value of the payoff is called the Value and it is the minimax of the payoff:

$$V(x) = \min_{\phi(x)} \max_{\varphi(x)} (Payoff) \qquad \text{(Eq. 4)}$$

Practically, in our simulations, the minimum of the payoff is found either by using a MATLAB function called fmincon, or by computing the payoff at each of the positions reachable in the next time step and choosing the position with the minimal value of the payoff.

Isaacs has proved that the Value satisfies the following first-order differential equation, called the main equation:

$$\min_{\phi} \max_{\varphi} \sum_j [V_j f_j(x, \phi, \varphi) + G(x, \phi, \varphi)] = 0 \qquad \text{(Eq. 5)}$$

We have tried to use this equation in our simulations, but, as the Euler approximation does not yield values that satisfy this equation, a lot more complicated algorithms must be used to solve it, making this task too difficult and time-consuming compared to the insight it might give us. The use of this equation would have given us another method to find the optimal path, but the results would have been nearly the same as the ones given in by our minimum-searching methods described above.

The four equations presented in this section define the principles of game theory and are the most important ones for our application. Other equations can be used, such as the retrogressive path equations, but the difficulties would be the same as those encountered with the main equation, and they would only allow us to compute the path another way. They would not help us find the best collision avoidance strategy, which is our main motivation for using game theory.

For our application, we cannot use game theory without any adaptation. Indeed, the goals of the vehicles are not literally conflicting: the vehicles can cooperate so that their goals are reached faster. Even if they do not cooperate, we cannot simply apply the presented equations. For instance, in one of the proposed strategies (section 3.2), the vehicles use the optimal evasion path, which is the path the evader would use in a pursuit game, but none of them use the optimal pursuer path, as they do not want any collisions. This is the reason why the minimax of the payoff cannot be used in our particular problem. In another strategy (section 3.5), the payoff is not common to our players, but each player has its own cost function, as in Nash game theory. We can say that game theory helps us find strategies and ideas, but its mathematical formalism is not always really useful for implementation.

In most of the proposed strategies, there are two different modes, the goal approach mode and the collision avoidance mode, and the vehicles switch between these two depending on the distance between them. That means that there are two different games in a single strategy: one game only considers the goals and its payoff can be the distances to the goals, whereas the other game is a pursuit game, or some more complicated collaborative game, with a payoff that increases with the risk of collision. In some strategies however, there is only one game, with the distance to the other vehicle always taken into account. More explanations on the payoff and on the approach are given for each strategy.

2.3 Preliminary results with fmincon

Fmincon is a MATLAB function that minimizes a function with constraints and that returns the value of the variables for which the function is minimized (more details in section 4.2). Before simulating any two-vehicle strategies, we first considered the simplest one-vehicle problem, which is a car trying to reach its goal.

Two payoffs have been experimented for this case: the time until the goal is reached (integral payoff) and the distance to the goal after a given time (terminal payoff). When we tried to simulate the vehicle trajectory with the integral payoff, we suspected that the fmincon function could not find the minimum of this kind of non-continuously differentiable function very well. More specifically, we set in MATLAB a maximum amount of time $T_{\max}$ for the vehicle to reach a certain zone around its goal. The payoff is the time needed to reach this zone, and if the vehicle cannot reach the zone, then the payoff is $T_{\max}$. It is clear that between two close trajectories, it is possible to have a step in the cost function, if these trajectories are close to the border of the zone. Besides, most of the trajectories yield the same value of the payoff, as they do not cross the zone. These are severe limitations for the fmincon function to find the minimum of the integral payoff.

Thus, we used the terminal payoff and did some simulations with different degree of maneuverability for the vehicle: we can alter the number of degree of freedom of the vehicle by stating how many times it can change its control variables during the given time T. With three possible command changes, the results are not really good (Figure 1).

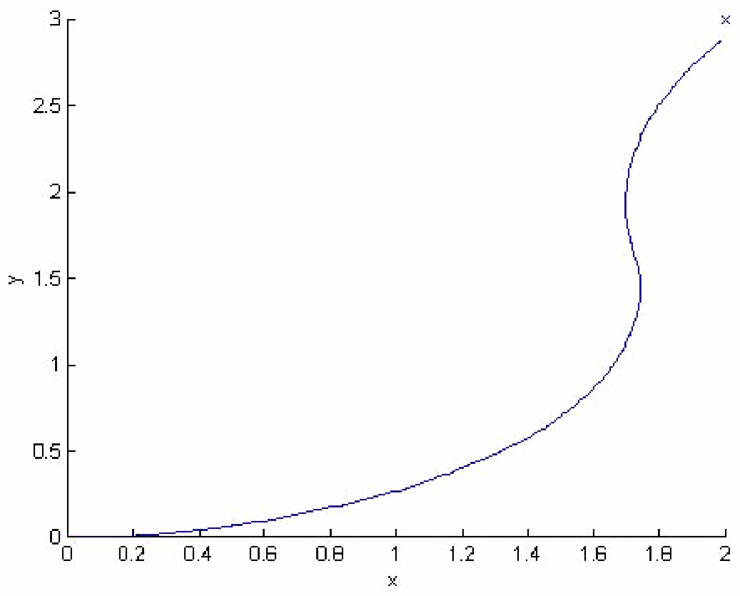


Figure 1

However, with five changes, the results are much better, even though they are not optimal (Figure 2). The more degree of freedom fmincon has, the better results it yields, but the more time it takes for computation.

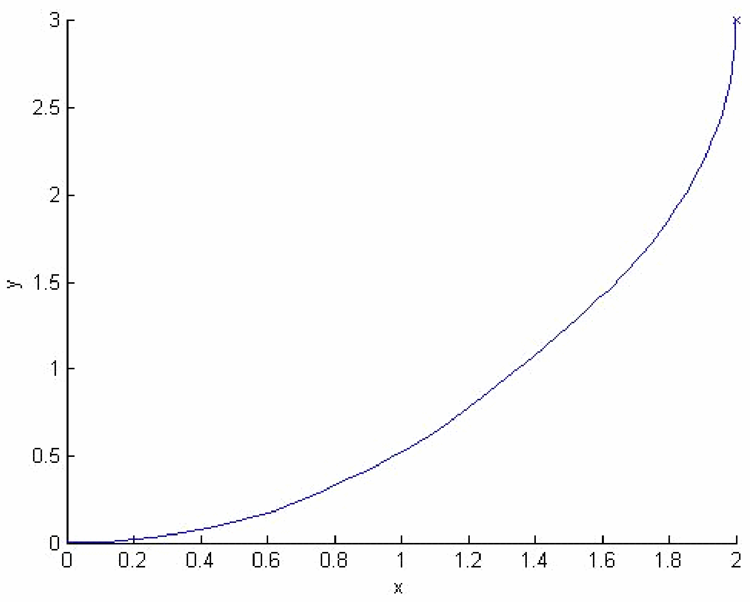


Figure 2

This means that fmincon can be a good way to find the optimal path and find an approximation of the minimum of the payoff, but it has some limitations we have to be aware of.

3 Collision avoidance strategies

In this section, all the strategies that we tested are presented, the first ones being noncooperative. Most of them involve game theory, with a cost function that determines the vehicle behaviors. The possible applications for these strategies are discussed, along with the sensors that would be needed.

3.1 Collision avoidance at the next time

The first strategy that came to our mind was the simplest one: each vehicle chooses the most direct path to its goal which does not cause a collision in the next time step. In other words, they go straight to their goals unless they are to collide at the next time step, in which case they take another direction that prevents them from doing that. Note that the collision should not necessarily (and not desirably) occur between the vehicles' bodies. It can be between their "safe zones", a space around them which should not be entered. The range sensors needed for this strategy need not be of high-performance, as obstacles and vehicles outside of the safe zone are not considered.

In our simulations, the vehicles move with simple motion, which means that, at any time, they can go in any direction they want, as if their dynamics were unaccounted for. This of course implies that it will be impossible for any real vehicles or mobile robots to track the path generated in this section. These simulations are considered as some preliminary work, which can show us some limitations and potential issues we can be confronted to in more advanced simulations.

The cost function they try to minimize is the distance to their goals. The directions they can take are discreet (every 45 degrees in our simulations), and their speed fixed. Thus, there are only a limited number of possible positions they can go to at the next time step. They compute the cost function value at those positions and go to the one where the cost function value is the smallest which satisfies the collision avoidance test.

The results for head-on intersection are good, better than in any other ulterior simulations (Figure 3). In the code, we specified that the vehicles should go to the left if they hesitate between going to the left and to the right when they want to avoid a collision (in the case where the two vehicles are perfectly facing each other). This keeps them from going in the same direction and away from their goals, as it can be seen in the second strategy for instance (3.2).

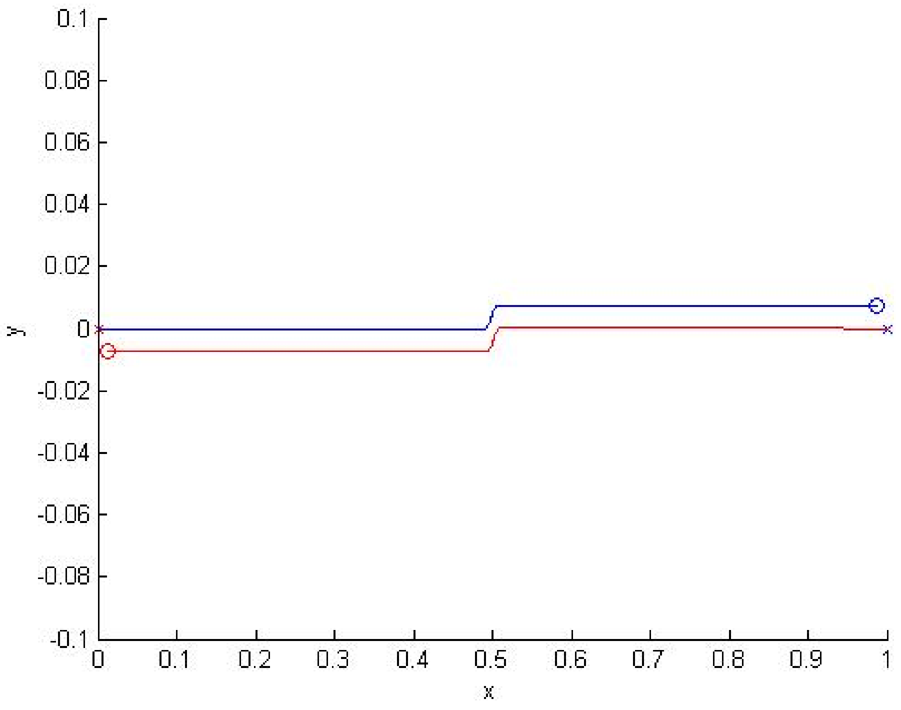


Figure 3

The results for perpendicular intersection are also good (Figure 4). We can see that the behavior of the vehicles near the intersection is not really safe.

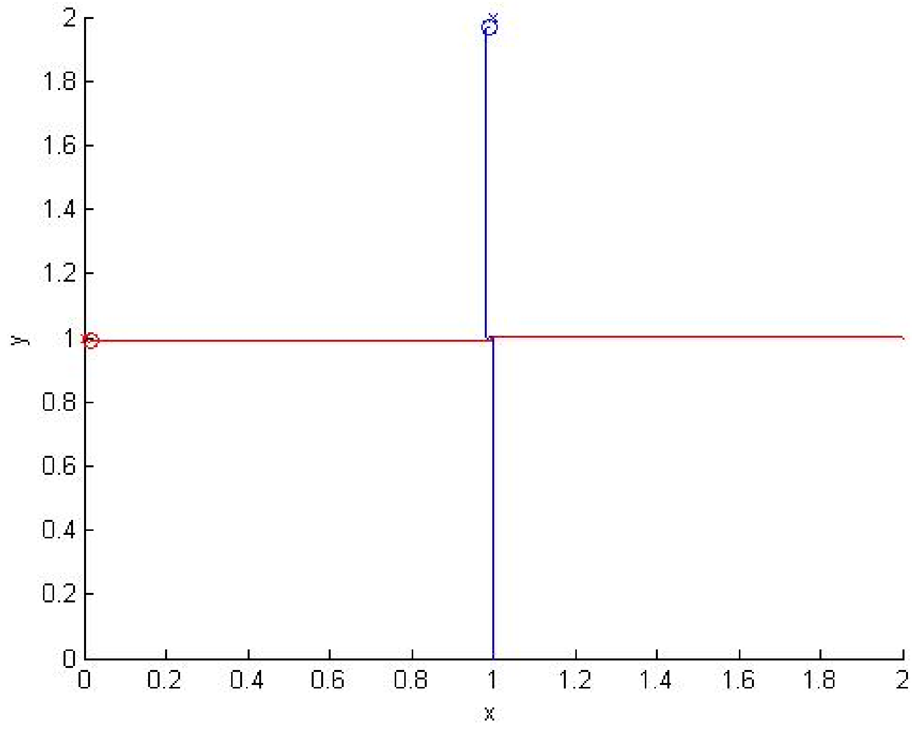


Figure 4

As seen above, this simple strategy yields good results when there are two vehicles with simple motion. Nevertheless, we cannot be satisfied with it. The main reason for that is the impossibility to generalize this strategy to vehicles with more complex dynamics. Indeed, if the vehicles had large radii of curvature and also large alert zones, they would not be able to avoid collision with their limited maneuverability. Furthermore, once, they enter in each other alert zone, they do not have any strategies to limit the damage. Even in the simple motion case, this strategy has its limits: if the alert zones are too large in a head-on intersection situation, the vehicles will have to do a sharp turn. With a better strategy, they would do a smoother trajectory and reach the goal faster using less energy. When the alert zone is really too large, one vehicle can even be blocked, as seen below (Figure 5).

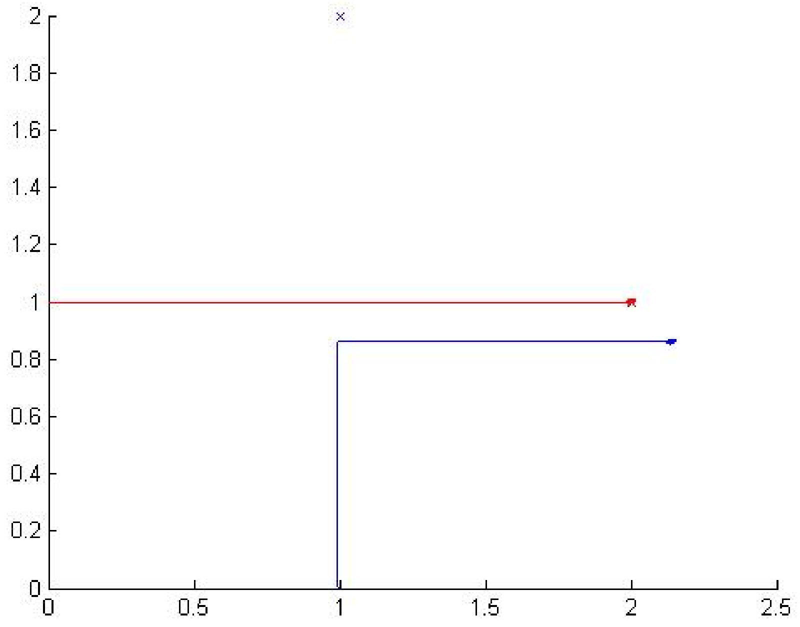


Figure 5

Another caveat with this strategy is that it cannot be applied with more vehicles, even with simple motion. There are indeed going to be situations where they think that they are in a dead-end and won't find any other solutions than going backwards and forwards, and oscillate that way.

Thus, better solutions are searched in the next sections. They may not yield solutions as good as with this strategy for the two specific cases of head-on and perpendicular intersections, but they will generally be more effective and safe.

3.2 Optimal evasion

In previous work from A.M. Bayen et al. [2], the following strategy is proposed when two aircrafts lose temporarily communication between each other and are dangerously close to each other (in each other's alert zone): each airplane has to take into account the worst case scenario, which is the other plane pursuing it. Therefore, when there is a possibility of collision, the situation is similar to a game of pursuit, the only difference coming from the fact that the pursuer is not following its optimal path. Then, at every computing instant, each aircraft has new information about the states of the other aircraft (position, speed) and calculates its new optimal path in order to best evade its supposed pursuer. They keep on doing this until they get out of each other's alert zone.

This strategy is the safest, as, if the alert zone is big enough (relatively to the speed and radius of curvature) and the planes have similar characteristics, they will always be able to avoid collision, even if one of them is aggressive. Each vehicle has to know the current states of the other vehicle within the alert zone, and this requires better sensors than in the previous strategy because the alert zone radius can be huge for very fast and not very maneuverable vehicles (five miles for commercial airplanes). For this type of application, there are usually enough infrastructures for getting the other vehicle states (GPS, ground-based beacons or radar stations, on-board radar, etc.) within this zone. For naval applications, the zone radius can also be bigger than a mile, but satellites and sonar give enough information to apply this strategy. Generally, any outdoor applications can use a satellite-based system for this strategy. For cheaper and smaller indoor robots, which cannot embark a GPS because of technical or financial reasons, a laser or ultrasound sensor has enough range for them to avoid collision. The speed of the other vehicle can be retrieved by computing the difference between two of its positions at different time steps.

Being inspired by this, we did simulations using this strategy: the vehicles with bounded radius of curvature apply this collision avoidance method when their distance is smaller than a given alert radius.

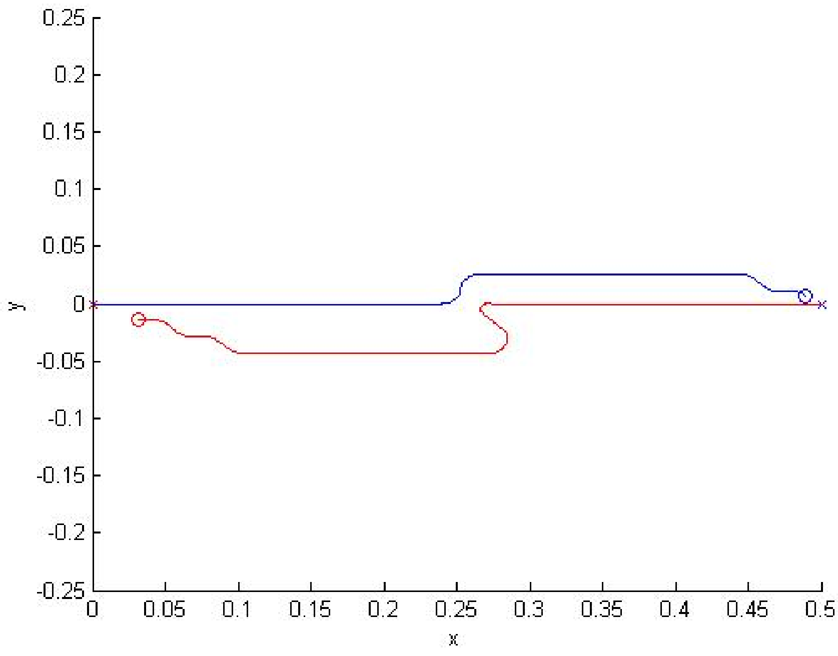


Figure 6

We can see that the results in the case of head-on intersection are far from being optimal (Figure 6). It is however safer than the first strategy, as they wait to be far enough from each other before heading to their respective goal.

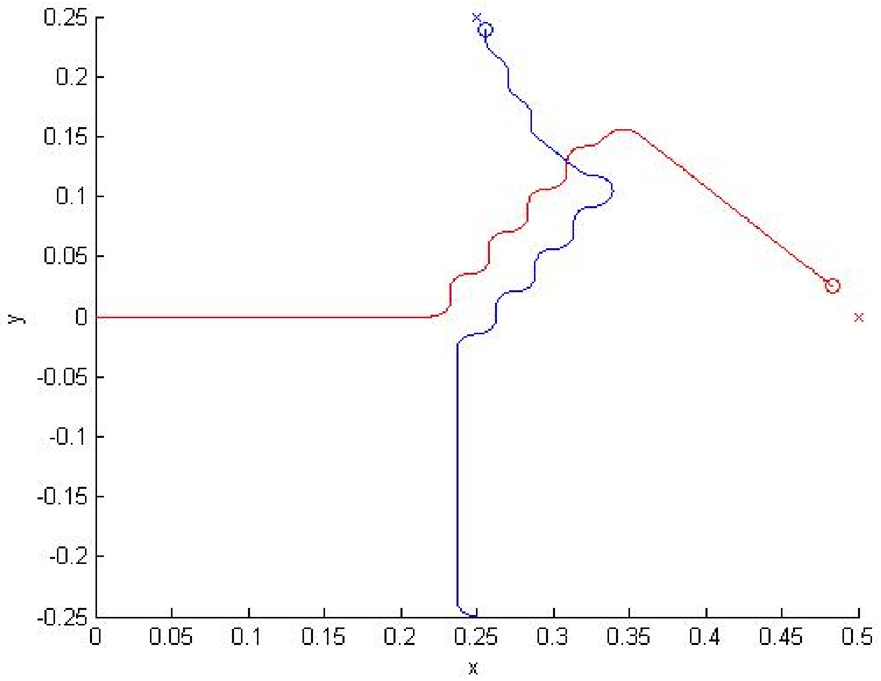


Figure 7

In the case of the vehicles perpendicularly intersecting each other, the results are not those expected: they oscillate by getting closer and farther several times (Figure 7). This is explained by the fact that they have first to avoid each other, and then, their distance being large enough for them to reconsider their goals, they get closer again and so on. The reason why this strategy works better for aircrafts than for cars is that the cars have a smaller radius of curvature and thus can avoid each other by nearly going back the way they came from, whereas the aircrafts do the best they can, but still have to more or less cross each other. If we take this reasoning a step further, two vehicles with simple motion that are perfectly facing each other will always oscillate by doing a backward-forward motion. In our case, we could have better results by reducing the alert radius, but this would not ensure collision avoidance any more, which was the main motivation for us implementing this strategy.

Thus, this strategy does not yield optimal results, but it has the merit of being the safest. Even if we implement a collaborative strategy, we can still use this one in case of the two vehicles losing contact with each other and not being able to cooperate any more. Still, there is another big disadvantage in this technique: the vehicles need a lot of space around them in order to track their optimal path. This can be possible in the air or on sea, but it is very unlikely on the streets or in factories (in the case of industrial autonomous mobile robots).

3.3 A blind vehicle and another one that avoids it

In the previous case, we saw that vehicles can have some unwanted motion when they have the same strategy. Then we could think that if a vehicle is unaware of its surroundings and keep on going towards its goal while the other tries to avoid it, this problem will disappear. This means that we assume that the blind vehicle knows how the other vehicle reacts and therefore knows it can go straight ahead. It is like a semi-collaboration, as they do not communicate their path, but they know their intentions. Although it does not seem to be a realistic hypothesis, it is applied everyday on the roads: at intersections, the cars that come from the right seldom watch their left as they know they have priority and that the cars from the left will avoid collision themselves by stopping. As one must have noticed, it is not the safest strategy, as it is possible that this initial hypothesis is not verified, like when a car denies the priority for example. However, a simple wireless device fixed on the crossroads can make sure that each vehicle knows whether it has the priority or not.

In the following simulation (Figure 8), one vehicle does nothing else than going towards its goal, while the other one uses the optimal evasion strategy developed before.

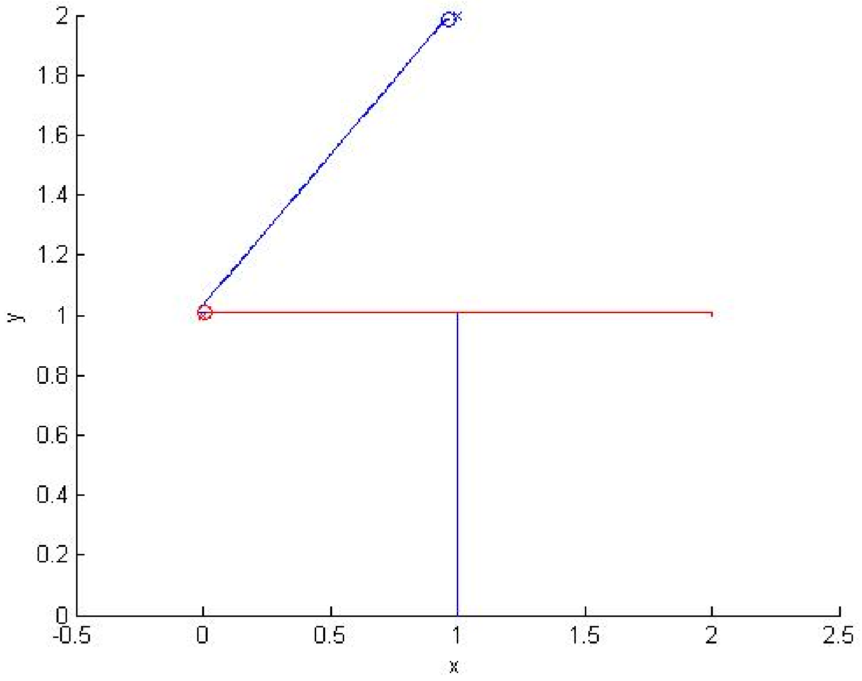


Figure 8

We can see clearly in this simulation that this idea doesn't work, because the evading car thinks that it is being pursued and flees until the other vehicle reaches its goal and stops. One question that comes at this point is: why not making one car stop, while the other one that has priority goes on the intersection before it, like in every day's life? The main reason that it couldn't be done at this point is that, in order to stop and let the other car pass, the evading vehicle has to know the other vehicle's path. Otherwise, the latter could collide with it, as it is unaware of its surroundings, because unlike on the roads, where each vehicle has to stay on its lane, they can go anywhere on the plane in our case. Thus, it is clear that collaborative strategies have to be developed. Some are presented in the following sections.

3.4 Acceleration – deceleration

This strategy, developed a little bit differently in [5], is quite intuitive and is more or less done everyday on the road: if there is a risk of collision between two vehicles in a perpendicular intersection, the one that will arrive first (the one that will be collided on its side) accelerates to further improve its advance, while the other one breaks to safely go behind the first car. When there is a head-on intersection, each vehicle turns right to avoid collision.

There are some simple calculations to do in order to know which car has to accelerate or decelerate. Geometrically (Figure 9), we can see that there are two boundaries on the trajectory of each vehicle between which a potential collision could occur (C11 and C12 for the vehicle 1, C21 and C22 for the vehicle 2).

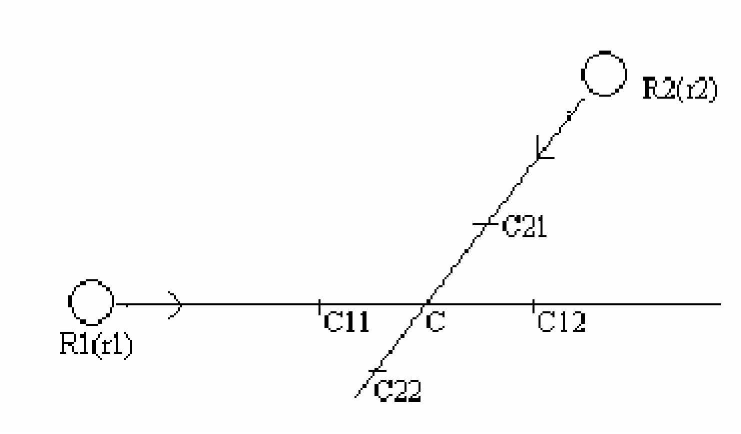


Figure 9

There is a collision if the centers of both vehicles are in these intervals at the same time. To avoid collision, one vehicle has to leave its interval before the other one reaches its first limit. Thus, if t11 and t12 are the times at which the first vehicle reaches respectively C11 and C12, and t21 and t22 the times at which the second vehicle reaches C21 and C22, then we want that:

$$[t11, t12] \cap [t21, t22] = \varnothing \qquad \text{(Eq. 6)}$$

To compute the points Cij, we first imagine that instead of having two vehicles with different radii R1 and R2, we have our first vehicle with a radius of R1+R2, whereas the second one is a point. It is clear now that C21 and C22 are equidistant from the intersection between the two paths C. They are, by definition, the farthest points that the enlarged vehicle can attain on the shrunk vehicle path. These are simply the intersections between that path and the two lines parallel to the first vehicle path and distant from it of R1+R2. Using trigonometry, C21 and C22 are at distance $\frac{R1+R2}{\sin(|\theta_1-\theta_2|)}$ on either side of C. It is easy to prove that the points C11 and C12 are at the same distance on either side of C. The times tij are found by dividing the distance between the current position of the ith vehicle and the point Cij by its current speed.

In order to find the coordinates of the point C, a system of two parametric equations has to be solved. Indeed, by definition, C has to satisfy the equations of both straight lines:

$$k_1\begin{pmatrix}\cos(x_{13})\\ \sin(x_{13})\end{pmatrix}+\begin{pmatrix}x_{11}\\ x_{12}\end{pmatrix}=k_2\begin{pmatrix}\cos(x_{23})\\ \sin(x_{23})\end{pmatrix}+\begin{pmatrix}x_{21}\\ x_{22}\end{pmatrix} \qquad \text{(Eq. 7)}$$

x_{ij} is the jth state of the ith vehicle; k_1 and k_2 are the parameters of the straight lines.

Applying these equations in our collision avoidance function, we tested this strategy. The results for perpendicular intersection show its simplicity and effectiveness (Figure 10). Each circle represents the position of one vehicle at one certain instant. Because of the constant interval between each snapshot, the gap between each circle gives an idea of the speed of the vehicle. Thus, we can see that the blue car, coming from above, slightly increases its speed after four time steps to safely cross the intersection before the red car. The latter noticing that the blue car has already taken enough action to avoid collision does not reduce its speed. Each car then safely arrives at its goal, more quickly and using less actuation energy than with any other strategies so far.

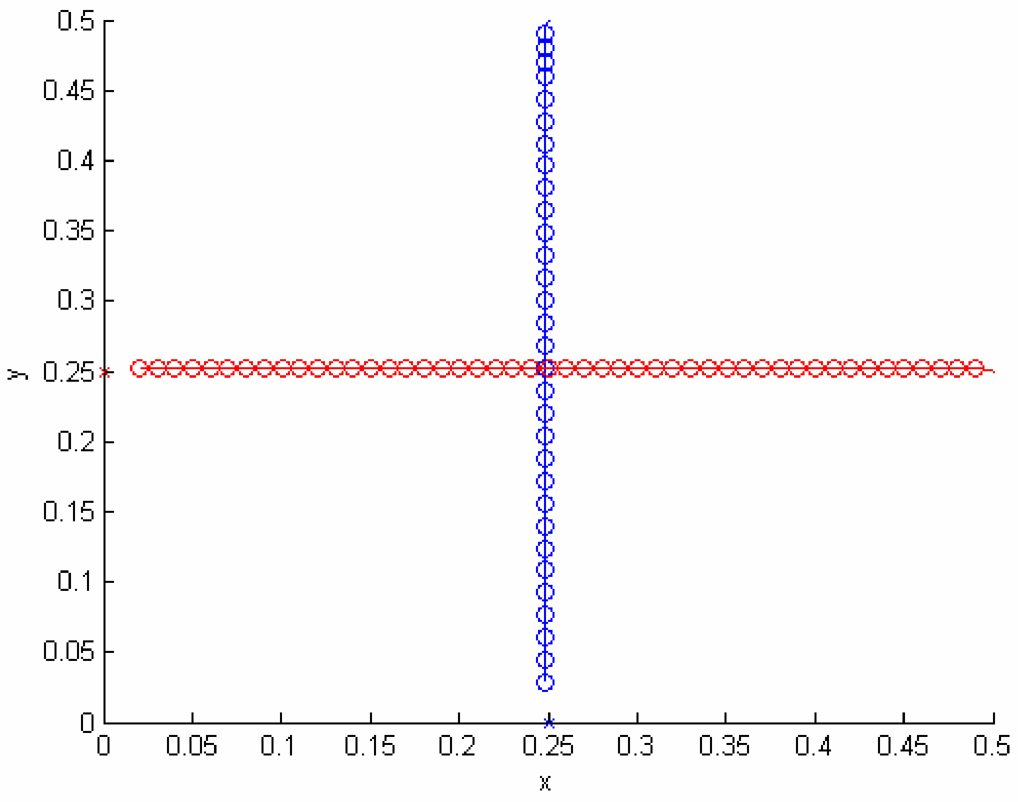


Figure 10

When the vehicles come close to each other at a smaller angle but with the same alert radius as in a perpendicular intersection, they need to take more dramatic actions by accelerating or decelerating more intensively. The simulations show good results for an angle of pi/4 (Figure 11). For a certain angle, there is a limit alert radius for which they will not be able to avoid collision. This happens when, even with one vehicle at its lowest speed and the other one at its maximum speed, the time intervals [t11, t12] and [t21, t22] overlap each other. More precisions are given on how to calculate the minimum alert radius in section 5.5.

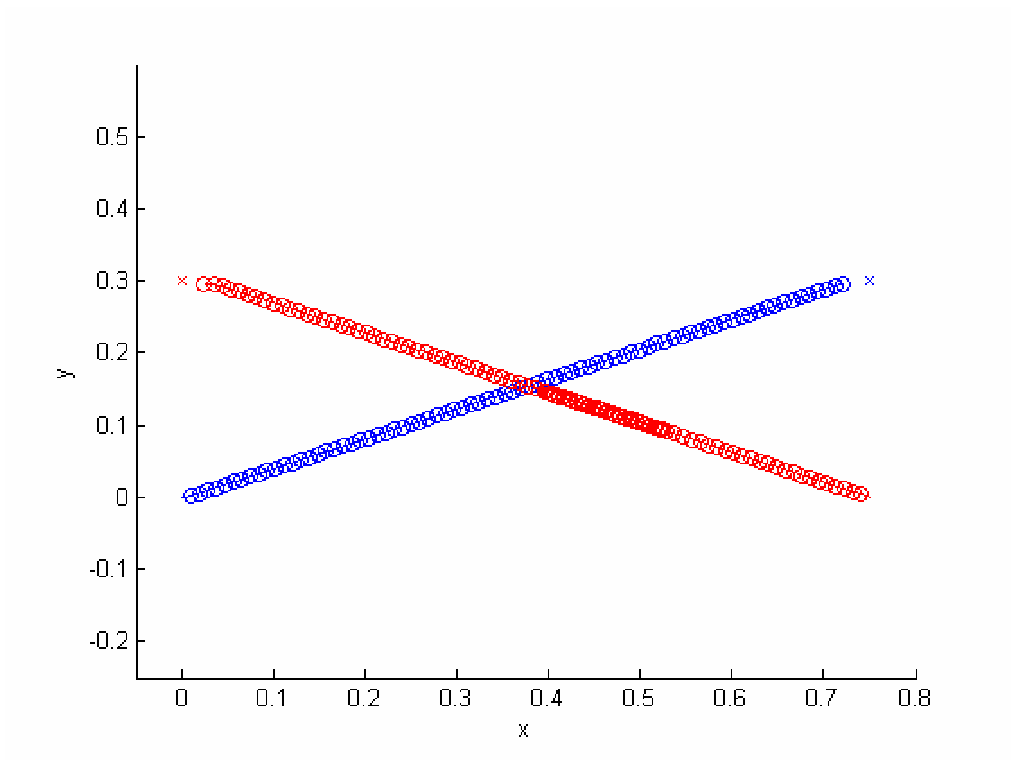


Figure 11

When the intersection angle becomes too small, the minimum alert radius tends to infinity. If two vehicles are facing each other in a head-on intersection, no speed changes can avoid collision (except obviously both vehicles stopping). That is the reason for us to introduce another method to avoid collision when there is a head-on intersection, which is making one or both vehicles turn right. The following simulation shows that only one vehicle needs to turn to avoid collision (Figure 12). The other vehicle, seeing that collision has already been avoided, keeps going.

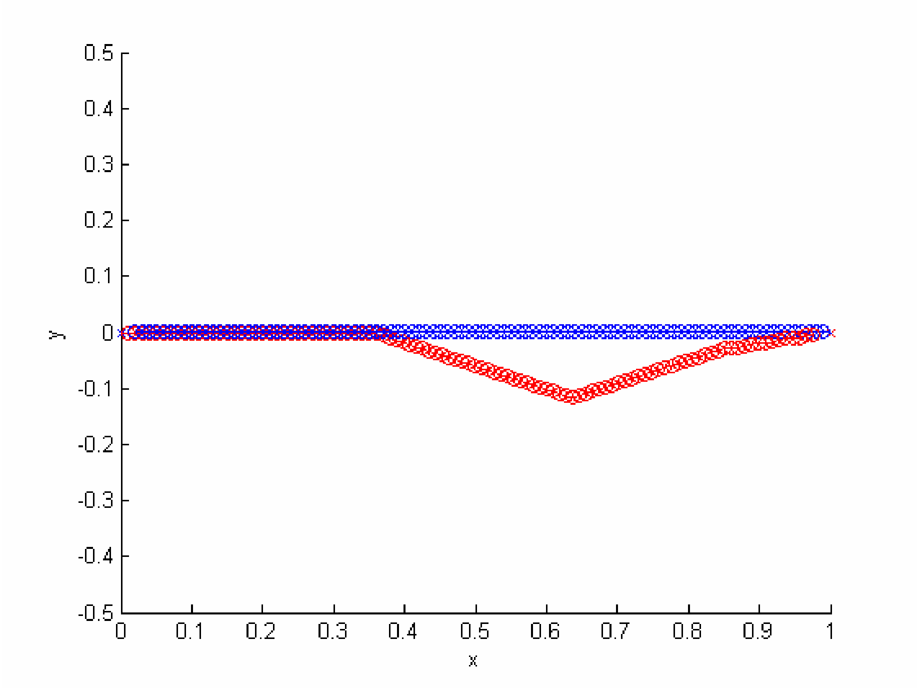


Figure 12

There is a limit to this method: if the vehicles intersect each other on their left sides, one of them will turn right and cross the other vehicle's path, instead of turning left and away from the incoming car (Figure 13). This can be modified quite easily in the code. However, showing how robust this intersecting strategy is, even with unexpected behavior like that, we can notice that after the right turn, each vehicle enters the perpendicular crossing mode and changes its speed to make the intersection safe. Each vehicle keeps its speed and direction configuration as long as they are within alert range of each other, which makes the red car do a detour. This is done to prevent oscillatory movement like in the optimal evasion strategy described in 3.2, where the vehicles change their behavior periodically between collision avoidance mode and goal approach mode. This can nevertheless be changed with some restrictions, like not turning more than a certain angle if still in the alert zone.

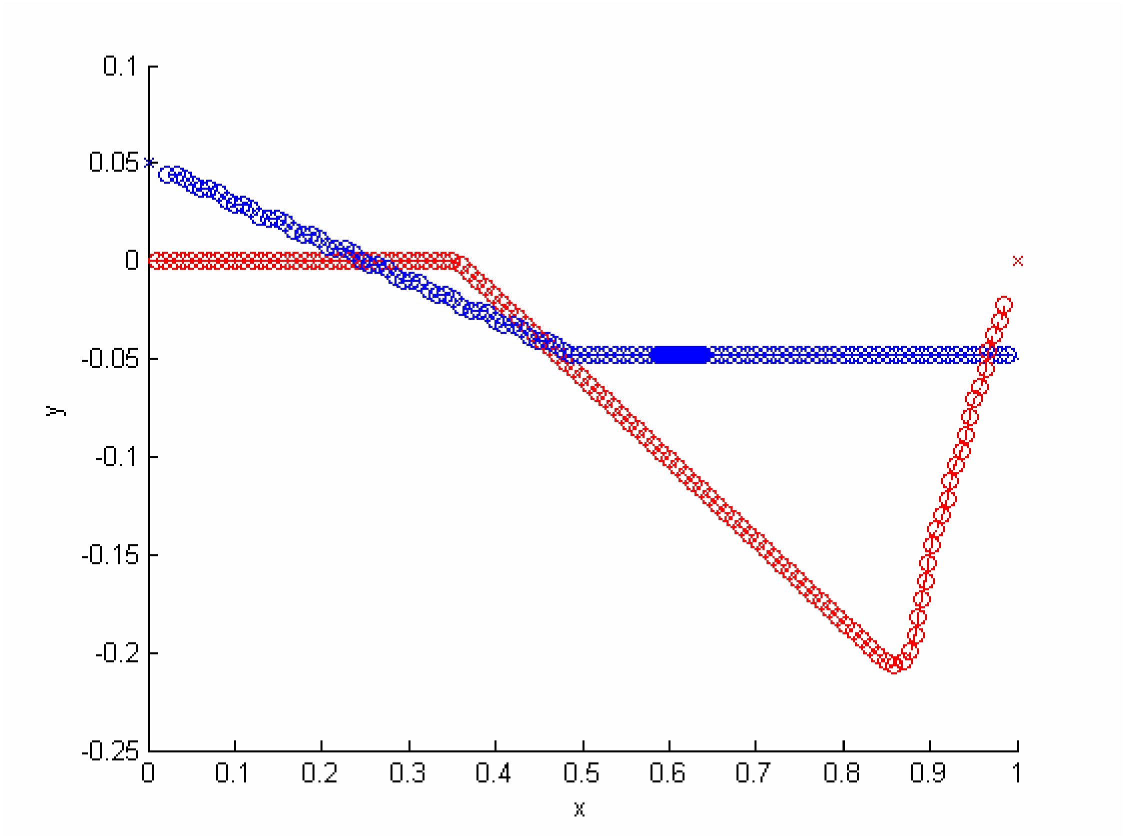


Figure 13

All these simulations prove the effectiveness of this strategy. It can even be applied to many vehicles by propagating the conflict [5]: the vehicles that are closest to the intersection set their speed to avoid collision, which may create a conflict with another vehicle. The latter then sets its speed and so on. If one vehicle cannot change its speed, or is limited in doing it, we can propagate the conflict from this vehicle. If the alert radius is large enough, it is possible to have each vehicle move safely. To prove this, let's first say that the minimum alert radius is the one large enough for each car to stop without any collision. If the alert radius is larger than that, there will always be a safe crossing strategy. An example of conservative strategy can be to make each vehicle stop and then make one go after another.

We can imagine implementing this idea (or to be more exact the perpendicular crossing strategy) in the real world: each car gives in wirelessly their current speed and position to a main computer fixed at the crossroad that sends back the corrected speed for a safe crossing. There is even no need to have a fixed computer at the crossroad, as a distributed calculation done by each incoming car can be done, the algorithm being very simple. Thus, when the vehicles collaborate, instead of sensing other vehicles states, they only have to know theirs and send them to each other. A GPS, a speedometer and an onboard computer with wireless communication– all of them already present in some of nowadays vehicles – compose the equipment needed for this system, and the range of the communication does not have to exceed the kilometer.

Potential gains of having such a computerized system are safety (computers are often more reliable than humans) and efficiency (the traffic can be more fluid with this strategy than with stop signs). The limitations can be the saturation of the number of the cars – when too many cars are present in a certain radius, it may not be possible to apply this strategy – and the range of the communication. It is possible to solve the first issue by having another strategy when there are too many cars (by making them stop and go, like on any traditional red-light crossroads). The second issue only affects the alert radius, and making every car slow down before the crossroad can ensure safety with the reduced alert radius.

This strategy can also be applied in other domains: in the air and on sea, it can be adapted so that only the directions and not the speeds are changed. Indeed, by making the aircrafts or the ships turn, the same effects as accelerating and decelerating can be obtained, as turning on a certain side can make one vehicle gain advance in the intersection or go behind the other vehicle. In mobile robotics, it is also possible to use this strategy, either by making the robots turn or change their speed, depending on the application. Implementation would be made easier if they share the same environment map, so that each other's absolute position can be communicated. Otherwise, some range sensors can be used to find the relative position of the other robots and of their goals (that robots communicate in their coordinates frame), and the strategy can be applied, with more programming effort though, as change of referential must be performed.

We can do a general remark about sensing in cooperative strategies: in noncooperative strategies, the states of the other vehicle must be found out using exteroceptive sensors, as they usually do not communicate, whereas in cooperative strategies, proprioceptive sensors output signals can be shared. This has some advantages, as it is easier to measure the vehicle's own states than to measure the states of a vehicle hundreds of meters away. Besides, exteroceptive sensors are more prone to noise and errors. This is why they often need more signal processing than proprioceptive sensors, which normally give outputs that are directly useable.

3.5 Noncooperative cost function minimization

Before trying to do a more elaborate work using cost functions, we tried to do one more non-collaborative algorithm: inspired from Nash game theory, this technique involves two separate cost functions, one for each vehicle. These are function of the distances between the two cars and between the car and the goal. Their value increases if the vehicles are getting closer to each other or if the considered vehicle is going away from its goal. Using fmincon, the vehicles try to find the best way to go to their goals while keeping a comfortable distance between them by minimizing their respective cost function. There is no switching between two modes, and no alert zones are needed.

The main difference between this strategy and the theories seen in [1] is that, here, there is one cost function for each entity, which it will try to minimize, whereas in Isaacs' theory, there is only one cost function that one player will try to minimize while the other one will try to maximize it, which cannot be applied in our case without some adaptations.

The cost functions are inspired from the potential field, used extensively in mobile robotics:

$$J = dist\_to\_goal^2 + \frac{K}{dist\_to\_obstacle^2}$$

where dist_to_goal is the distance between the current position of the vehicle and its goal, and dist_to_obstacle is the distance between its current position and the obstacle, which is the other vehicle in our case. K is a constant that determines implicitly the distance we want to keep between the vehicles. We could have used a more adapted cost function, with the distance between the vehicles being neglected when superior to a certain threshold, allowing the vehicles to go more optimally towards their goals, but fmincon does not always handle these kinds of discontinuous functions very well. Furthermore, the simulation results would be nearly the same as with the cost function that we used. One can notice that only information on the position of the other vehicle is needed, and some range sensors are sufficient to apply this strategy on a real system.

The problem that can come with this type of cost function is the existence of local minima. In our two-vehicle case, such problem appears only on some single points, for instance when one vehicle is perfectly on the line that joins the other vehicle to its goal. But because these local minima are unstable, a small perturbation (for example one vehicle still moving because of its inertia) solves this issue. If there are more vehicles involved, this local minima problem will be in some cases unsolvable. In some dispositions, the vehicles can follow a path that leads them to a dead-end, and they will have no mean to get out of it, as going backwards will increase their cost functions.

The following two-car simulations show the smooth trajectories provided by this strategy, for head-on (Figure 14) and perpendicular intersection (Figure 15). This can then be a simple solution for basic problems with few vehicles where safety is not the most important parameter. An example of application can be small mobile robots (vacuum cleaners, etc.).

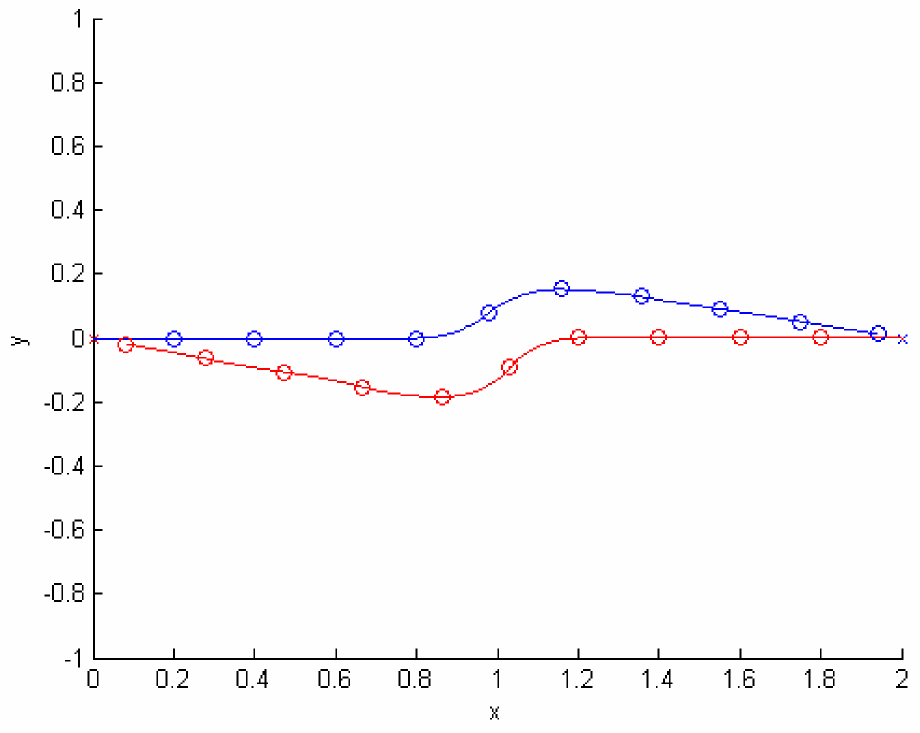


Figure 14

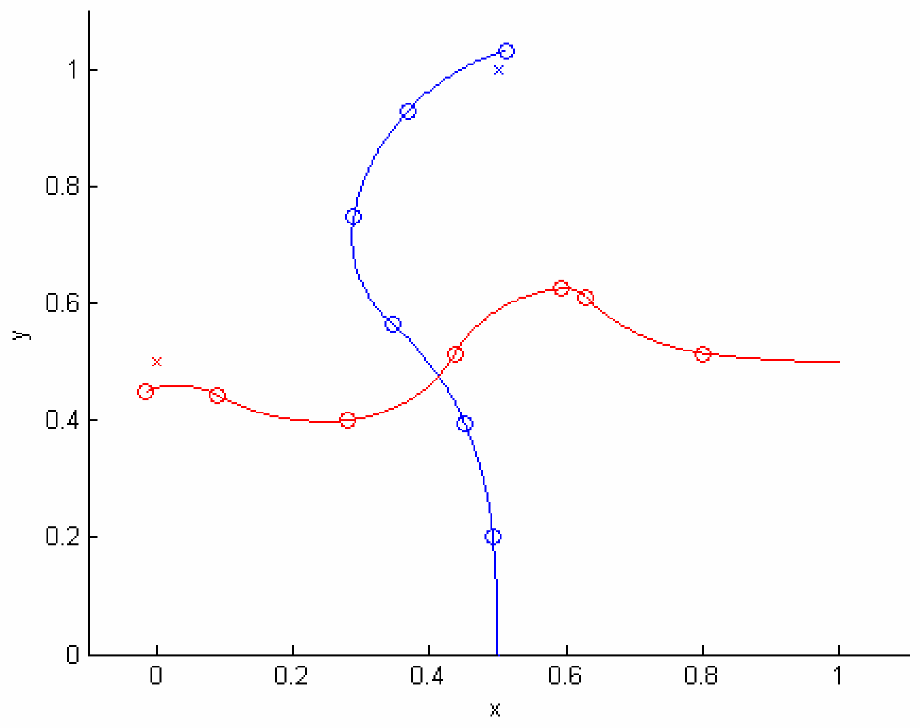


Figure 15

The results show that fmincon can provide good solutions, and encourage us to try to implement the next strategy.

3.6 Cooperative computer-generated path optimization

To fully use the optimization tools offered by MATLAB, we decided to apply our very first code (the one-vehicle fmincon simulation) to our collision avoidance problem. As in the one-vehicle case, we optimize here a cost function by varying a set of commands. One difference comes from the cost function, which also takes into account, besides the distance to the goals, the distance between the two vehicles. The other difference is that we have to optimize the commands of two robots here, which requires more computational resources.

Like in most of our simulations, an alert zone defines the payoff that is to be used: when the vehicles are far away from each other, the payoff is simply the sum of the distances to the goals. We tried two different payoffs for the collision avoidance mode: 1) the distances of the vehicles to their goals added by the inverse of the integral of the distance between them, and 2) the distances to their goals added by a penalty value when there is a collision on the trajectory. Simulations with the first payoff did not yield good results, because collision occurred most of the time. Besides, this payoff is similar to the ones in the previous strategy (3.5).

Thus, we used the second payoff. The way fmincon works is by looking for a minimum value of the cost function in an area near the initial conditions. To do this, it computes different trajectories and their corresponding payoffs, and does a gradient-descent type approach. Our objective is to find a trajectory that minimizes the time to reach the goals, not to have the vehicles go as far away from each other as possible. Indeed, the best trajectory is the one in which the vehicles go very close to each other (if needed) while avoiding collision, as it would minimize any needed detours. The idea is to increase by a constant penalty value the payoff of each trajectory in which a collision occurs, so that it is not chosen. When minimized, this type of payoff should give a trajectory that leads as close to the goals as possible and that makes the vehicles avoid collision by only a small distance.

The results of head-on intersection simulations show the limitations of this technique, at least when using fmincon. The first figure (Figure 16) shows the trajectories of vehicles with radius of 0.05. There is no collision and the form of the trajectories is satisfying.

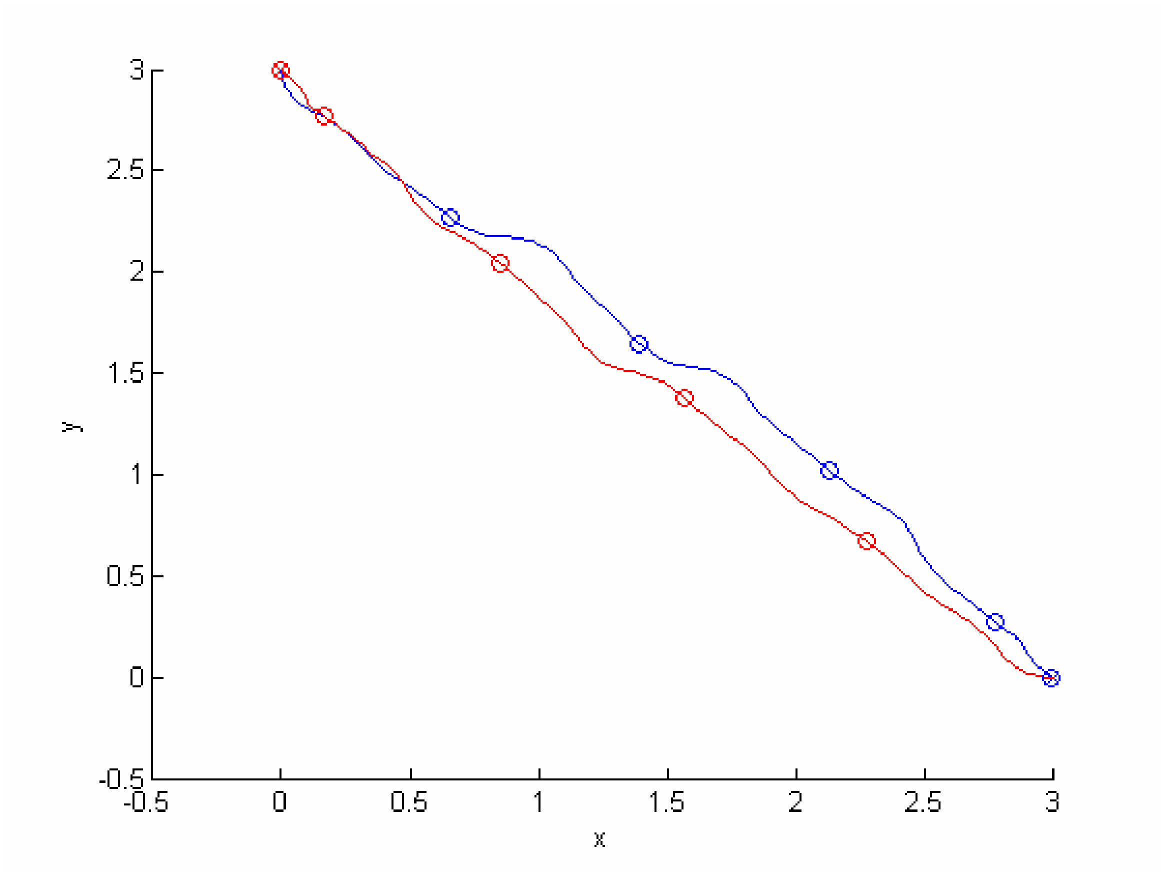


Figure 16

In the next simulation (Figure 17), we double the radii of the vehicles, which can increase the number of collisions and force the algorithm to test more trajectories to get rid off the penalty. Apparently, fmincon does not search in a zone that is big enough, as collisions occur in the simulation.

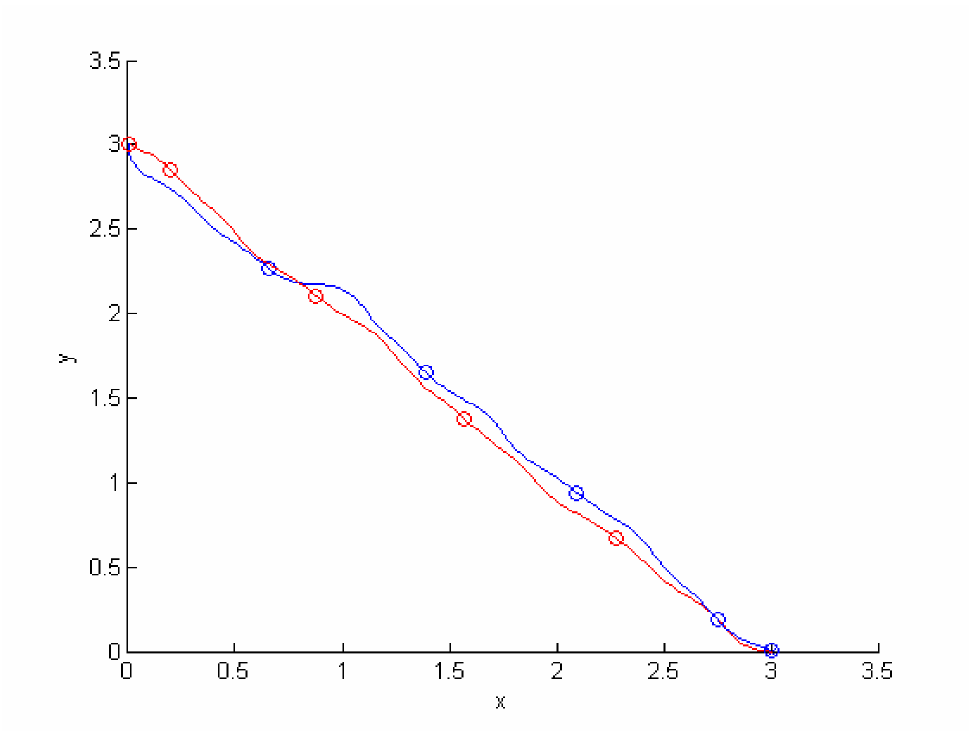


Figure 17

Considering the previous simulation, we can have an idea of the importance of the initial zone of research. If the initial command vectors are not towards the goals, then fmincon might find a better solution. The initial command vectors are not to be confounded with the initial states of the vehicles, as they define for fmincon where to look for the minimum, whereas the initial states describe the system at the beginning. In the two following simulation results, this can be even more apparent. The next figure (Figure 18) shows the trajectories found when the first initial research vectors are pointed away from the goals. There are no collisions, but the trajectories are far from being optimal.

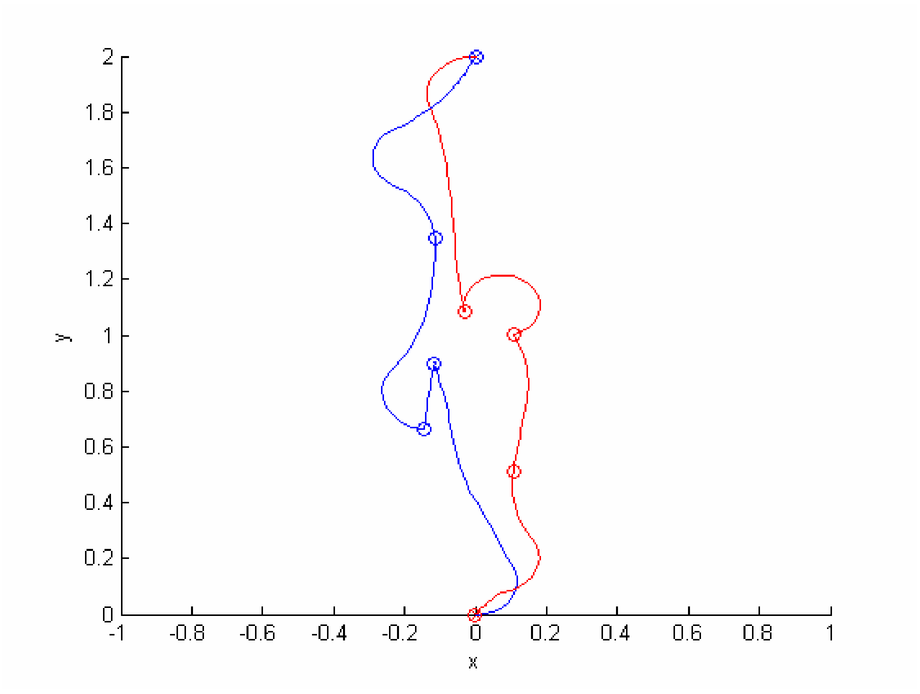


Figure 18

When we change the initial search vector so that it points toward the goals, the results are even worse: one vehicle is stuck (Figure 19). The simulation was stopped after ten iterations. It is unclear why no solutions were found for the blue vehicle to move towards its goal.

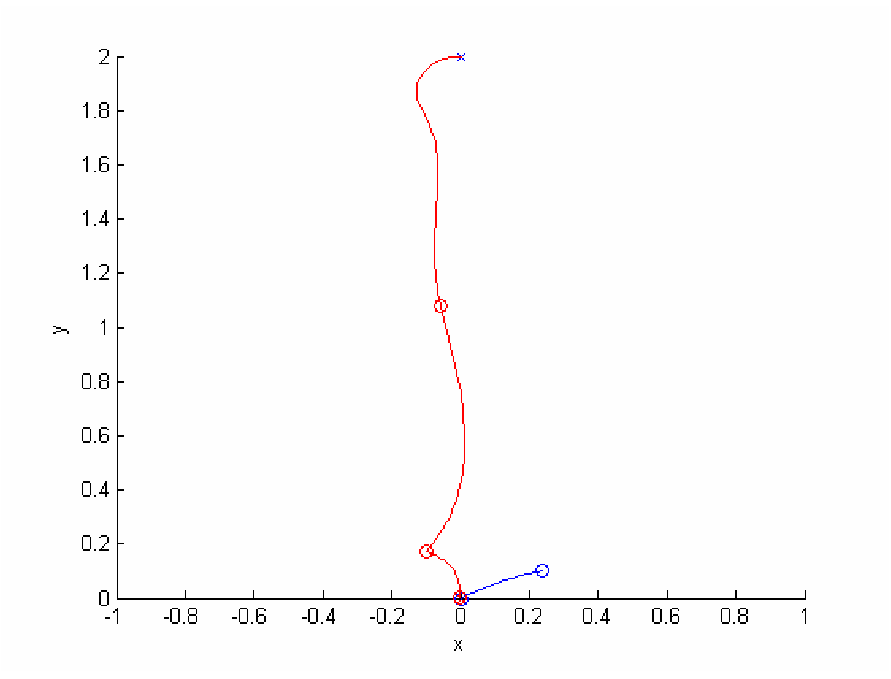


Figure 19

For perpendicular intersections, the results are a little bit better. Even with initial vectors pointing towards the goals, the following collision-free trajectories were found (Figure 20). The vehicles have radii of 0.05.

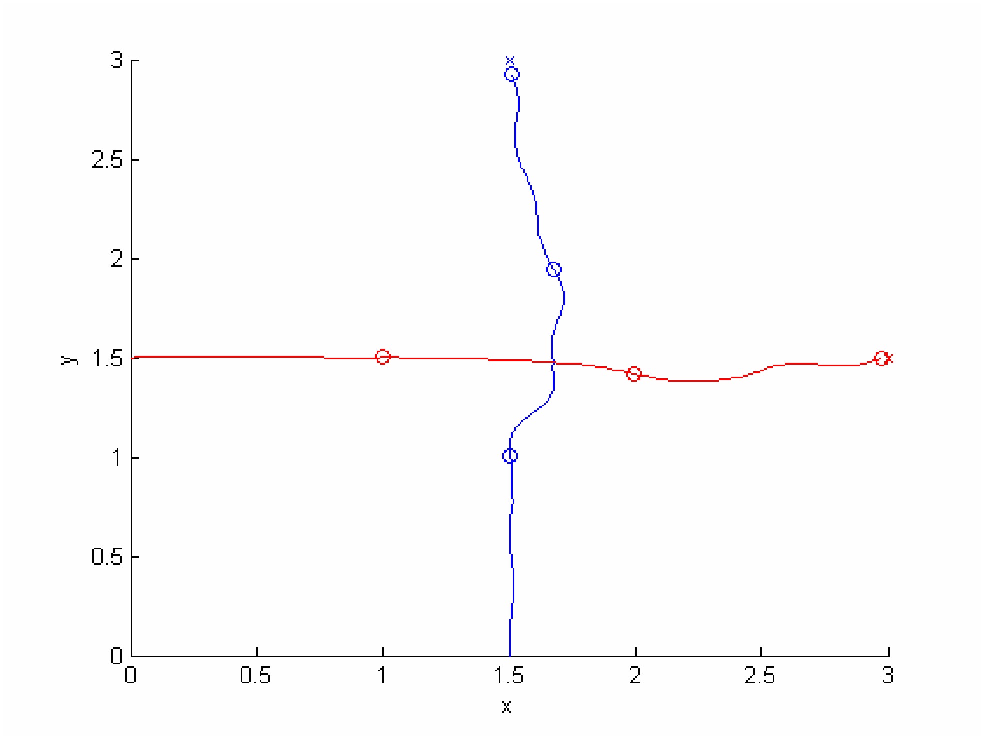


Figure 20

With radii of 0.1, the vehicles still find good trajectories to reach their goals without colliding (Figure 21).

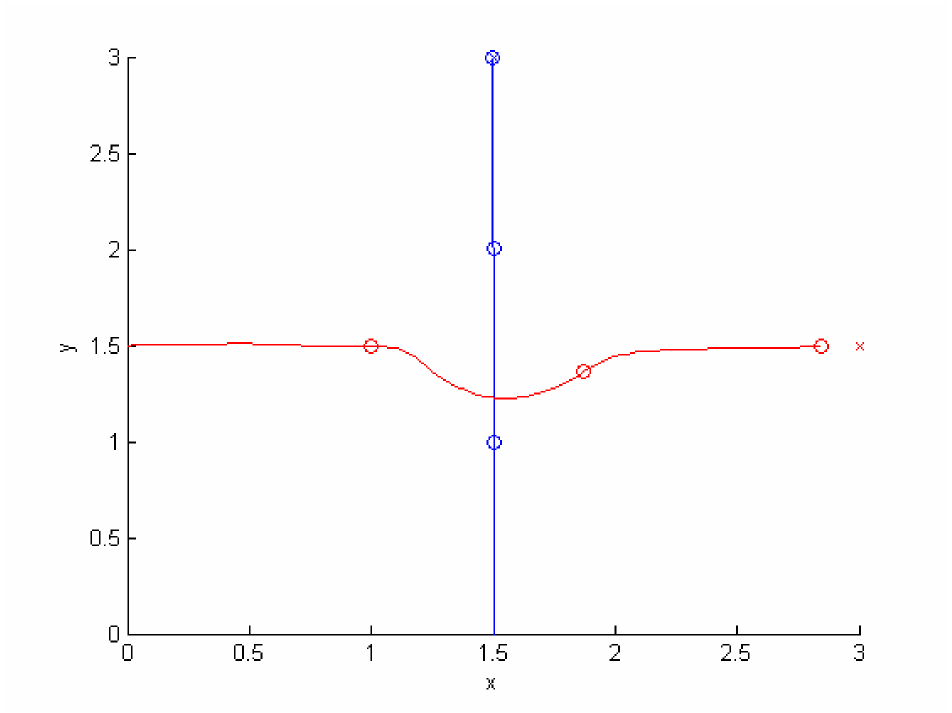


Figure 21

However, when we only give three sets of control variables to optimize (for the previous simulations, four pairs were given in), no collision-free trajectories are found (Figure 22). It seems like there are not enough degrees of freedom for fmincon to avoid the penalty when the initial vectors are towards the goals.

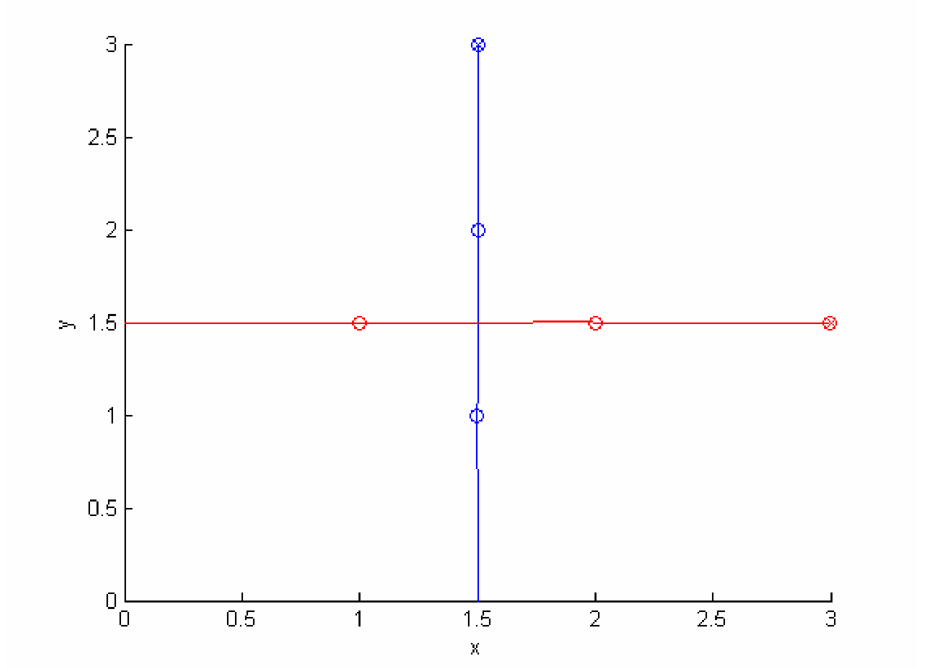


Figure 22

All these simulations show the limitations of such a straightforward approach. The problem is non-convex, and thus, it is difficult for fmincon to find the global minimum. Another issue is the computation time needed to find these trajectories: when there are four or five sets of command to optimize, it can take several minutes for the computer to find a solution. If this strategy is to be applied in any real systems, it is clear that this delay is unacceptable.

Nevertheless, some of the results are quite good, especially when the number of command to optimize is high enough and the initial vector in the good direction (near the optimal direction). We can conclude that with an algorithm more adapted than fmincon, this strategy may be applicable. For instance, the huge computation time can be reduced by decreasing the precision of the answer. Indeed, fmincon takes so much time to do its calculations because of the small perturbations it does near the initial vector to find the best solution. The commands need not be very precise, and looking for a more coarse solution can save some precious time. Furthermore, if we increase the steps between each command, we can search solution in a broader space for the same given time. This allows other better solutions to be found and limits the importance of the initial vector. For instance, in the second simulation (Figure 17), fmincon did not search enough around the initial zone and collision avoidance could not be ensured. With a broader search, this kind of situation will happen less frequently.

The global minimum for this kind of problem is hard to find for any optimization algorithms. Only those that look into every possible case, like genetic algorithms, can find the best solution, but they are often inappropriate for real-time systems. The aim is still to have a large zone in which the solution is searched, and the best way to do that with small computation time is by diminishing the precision. Hence, some minor changes in the fmincon algorithm can enhance dramatically the results given by this strategy, and with some efforts, it can even become applicable. The needed sensors would be approximately the same as for the strategy described in 3.4.

4 MATLAB functions used

Here, we present the principal MATLAB functions used in our simulations. A short explanation of how to use them is also provided.

4.1 Ode45

This function is used to integrate a system of differential equations by Runge-Kutta method. As it only solves first-order differential equations, a state-space representation y' = f(t,y) is required for higher order equations. We only have to give in the initial conditions, the time span and a function that contains the system of equations, and it returns the solutions for some given times. The prototype is:

```
[T,Y] = ODE45(ODEFUN,TSPAN,Y0,OPTIONS,P1,P2...)
```

Odefun is the function that returns the derivatives, tspan = [t0 tfinal] indicates that we want to integrate from t0 to tfinal and y0 is the initial condition vector. The options, p1 and p2 arguments were not used in our simulations, but we could have used p1 and p2 to pass parameters (like the commands) to odefun.

The function returns a vector T of length p which contains the successive time of integration, and a matrix Y of size (p x n) containing the solutions of the system of equations with each line corresponding to a time in T, and each column to a state. We can also write tspan=[t0 t1 … tfinal], to have the solutions at these specific times. We used this to compare the states of our two vehicles in our "cooperative computer-generated path optimization" code (section 5.7), as if we do not specify the times specifically, MATLAB will not always take the same time step and we then cannot compare values at the same time.

To specify the equations, we write a function containing the state equations in another M-file, which we call in ode45 by writing @function_name. The prototype of the function is output=odefun[t,input]. Output is the vector that contains the values of the derivative of the states, input is the vector with the current states values, and t is the time-vector, but it is not the same vector as in tspan, as it is used internally by ode45. If we need to know tfinal for instance, then we have to either pass it as a parameter (with p1 or with a subfunction - see the "cooperative computer-generated path optimization" code, more explanations in section 5.7), or, as we did in the beginning, pass it as an input with its derivative being specified at zero (this is not a desirable solution).

4.2 Fmincon

This function is used to find optimized commands that minimize a certain cost function under constraints. We give in a function that returns the cost function, the initial conditions and the constraints; fmincon tries to minimize the solution by successively computing the cost function with different inputs, and after many iterations, returns the inputs that have yielded the smallest cost function. The prototype is:

[X ,FVAL]=FMINCON(FUN,X0,A,B,Aeq,Beq,LB,UB)

Fun is the function that returns the cost function value given the variables that we want to optimize, x0 is the vector with the initial values of these variables. The fmincon function starts to search for the best solution around these initial conditions. Therefore, if we have an approximate idea of where the solution might be, we should define x0 close to this area, so that the solution is found more rapidly and also to be more certain that the solution returned is not a local minimum.

The other parameters are the constraints on the variables. A is a matrix and B a vector such that A*X<=B (linear inequalities). Aeq and Beq are also respectively a matrix and a vector, but they define a linear equality, such that Aeq*X=Beq. LB and UB are vectors that define the lower and upper bound of the variables, LB<=X<=UB. Most of the time, we only used the constraints LB and UB, which define the speed limits and the minimal radii of curvature of the vehicles. The unused parameters are set to [].

If we have constant parameters to pass in the x0 vector, we can use the equalities constraints to ensure that they are not changed and considered as variables. A better solution would be to pass these parameters using an anonymous function @(x)fun(x,param). This way, the fmincon function only considers x in the minimization process, there is no need to specify in sometimes huge matrices that the parameters are equal to themselves and it uses less computational resources, as the parameters are not considered at all when the solution is being calculated.

The function returns a vector x which contains the optimized values of the variables and the minimized cost function value fval.

Note that there are more parameters that can be defined, like non linear constraints, but we did no use them. Refer to the MATLAB help browser for more information about them.

The problems with this function are the required computational power (it can take several minutes of calculation when there are more than ten variables to optimize) and, more importantly, the fact that it generally finds a local minimum.

5 Implementation of the simulations

In this section, the implementation issues we encountered, and the solutions we found for them, are presented. Furthermore, the codes written in MATLAB in order to simulate the strategies are explained.

5.1 One-vehicle simulation using fmincon

This is the first code written for this project and some details have been improved in subsequent codes (like the way parameters are passed to the simulcar function). It optimizes a set of commands by minimizing a cost function (the distance to the goal in this case). The simulation time is divided in equal intervals for each pair of commands. For instance, if we give four commands, there will be two intervals, with the first two commands (speed and steering) optimized and applied during the first interval. Thus, the more commands we optimize, the more reactive the vehicle will be, as it will be able to change its speed and direction more frequently. However, it increases the computation time required to perform the simulation.

OptimcarMain is the main module that performs the optimization and displays the results. Optimcar is the function that returns the cost function value given the commands of the vehicle, the goal coordinates and the total time of simulation. Simulcar is the function called by ode45 when we want to integrate the differential equations describing the dynamics of the vehicle. It returns the derivative of the state variables of the vehicles, which are x, y and theta.

In optimcarMain, we specify the goal we want to reach, the time given to the vehicle to reach its goal, and the number of commands to optimize. The limits on the vehicle dynamics (maximum speed and minimal radius of curvature) are indicated in the upper and lower bounds of the commands.

In order not to have to modify every function each time we want to change the initial conditions, we call an anonymous function when we use fmincon, which is @(x)optimcar(x,param). This syntax means that we want to optimize the optimcar function with regard to x, and that the other parameters are not to be changed. This allows us to pass the initial conditions to other functions.

The integration of the differential equations is performed after the optimization, with the optimized commands as inputs. We have to put as parameters for the simulcar function, besides the commands, the initial state of the vehicle and the simulation time, as these are information needed to compute the trajectory. We could also have used an anonymous function, as for optimcar, in order to pass these parameters. It is in simulcar that the intervals of time are constructed.

5.2 Collision avoidance at next time

The main function OptimcarMain is straightforward: we first state the goals and the initial position of the two vehicles, along with other parameters as the minimal distance from the goals to reach and the radius of the safe zone around each car. Then, until both vehicles reach their goals, they move one after another closer to their respective goals. To do that, they compute the cost function value around them at each point they can attain at the next time step without bumping into the other car, and then they choose the angle corresponding to the point with the minimum cost function value. The cost function is simply the sum of the distances to the goals. As mentioned in the beginning of this report, game theory has to be adapted here and the minimax of the payoff is not searched. Instead, both vehicles minimize the payoff. This does not change anything, as the theories involving the minimax of the payoff are not applied.

The trajectory with the command is computed using the simulcar function which is a simplified version of the one used above, because there is only one pair of input for the whole time interval, and there is then no need to divide the time frame into smaller intervals.

The angles that the vehicles can choose are fixed in the world referential (there are 8 possible angles in our case). The steering command given to the vehicle is

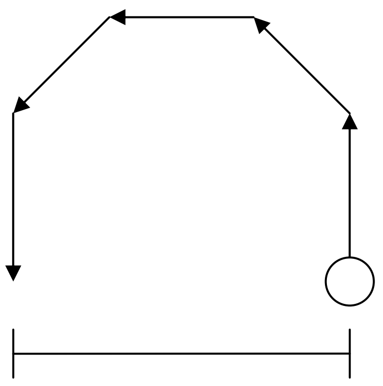
K*(desired_angle - current_angle)

The gain K is there to ensure that the desired point can be reached. Indeed, we do not take into account the dynamics of the vehicle here, and without this proportional gain, the vehicle will turn too slowly to the desired angle and miss the point. The gain must be chosen with regard to the time step and the speed of the vehicle: if the gain is too big, the vehicle will be unstable and turn too much and miss the desired position, and if it is too small, it will not turn quickly enough. If we increase the time step or decrease the speed, we have to decrease the gain K, as the vehicle will have more time and distance to reach its position. With a time step of 0.01 and a speed of 1, a K of around 100 yields good results. We can also find the perfect K theoretically, but as these simulations are done to provide a general idea about a certain algorithm, this kind of approximation can be accepted.

5.3 Optimal evasion

The vehicle behaviors are defined in OptimcarMain: each vehicle moves differently whether there is a risk of collision or not. If the distance between the two vehicles is larger than the alert radius, each of them will try to reach their goals as fast as possible by choosing an optimal angle, in a similar manner as in the previous code. If there is an alert, it will ignore its goal and select an angle so that the distance to the next position of the other vehicle is maximized. Thus, there are two different payoffs, one when there is an alert, and another one when they want to reach their goals, and this approach is used in most of the subsequent strategies. Once the angle is selected (the speed is fixed), the simulcar function is called to find the trajectory of the vehicle. The simulation stops when both vehicles have reached their goals.

In this simulation, the vehicles do not move with simple motion, because we bounded their radii of curvature. To do this, we modified the optimal angle search algorithm: instead of looking all around itself to find the position that minimizes the cost function, the vehicle only consider three positions: straight ahead and the minimum angle step on the left and on the right. The minimum angle is implicitly defined in the angle vector (which contains every possible discreet angle the vehicles can take), as it is the difference between two consecutive angles. For example, in our code, there are eight possible angles, at every 45 degrees. Then, the minimum angle, or the maximum angle the vehicles can steer with, is 45 degrees. To increase the radius of curvature, in case we want to model vehicles with smaller maneuverability, we only have to enlarge the angle vector by putting more possible angles in it, with smaller difference between them. The radius of curvature value can be found geometrically. In our 45-degree case, the radius is (see the figure below) $speed \cdot t_{step} \cdot [\cos(\frac{\pi}{4}) + \frac{1}{2}]$.



2*(radius of curvature)

Hence, the positions reachable with the three angles are compared, and the one that minimizes the distance to the goal is chosen if there is no alert, whereas the one that maximizes the distance to the next position of the other vehicle is chosen if there is an alert. We can see in the code that there are two boundary conditions corresponding to the cases of the current angle being the first or the last element of the angle vector.

Another improvement in this code is the way the angle command (the second element in the input vector, v_2) is defined to avoid some imprecision at the boundaries of the angle vector. It is not anymore a gain multiplied by the difference between the current angle and the desired angle, as it was done previously. We normalized this angle difference between $-\pi$ and π, so that the vehicle does not turn by an angle of (2π – alpha) instead of -alpha when its angle has to change from 0 to –alpha. We use the npi2pi function for this purpose, but we can also use the formula mod(angle + pi, 2 * pi) – pi if the version of MATLAB does not support the npi2pi function.

When there is an alert, the evading vehicle should act as if the other vehicle is a pursuer that moves optimally. It is not exactly the case here, because, to simplify our code, we do not compute the next optimal position of the pursuer, we only state that its next position is the one straight ahead of it. Thus, it is not exactly the worst-case scenario that is taken into account. We can argue, though, that the results of the simulation are not fundamentally changed by this detail of implementation, as the radius of alert is big enough for the vehicle to react in time and far enough from the possible point of collision. The difference would be that the vehicles in our simulation evade less aggressively than in theory, and as we have already seen, oscillations and undesired behavior are already present in our case.

5.4 A blind vehicle and another one that avoids it

The code for this simulation is exactly the same as the previous one, except we took off the collision avoidance part of one of the two vehicles. To be more precise, we erased the alert test of one vehicle, along with everything written in case there is an alert. This vehicle will then do nothing else than going towards its goal, following the "no-alert" algorithm.

5.5 Acceleration – deceleration

The OptimcarMain module is very similar to what have been written above: each vehicle tries to reach its goal unless there is a conflict with another vehicle, in which case the collision avoidance strategy is adopted and the collision function called. The angle selection is done the same way as in the previous code and the speed is fixed to 1 when there is no alert.

The collision avoidance strategy is defined in the collision function. The latter returns the commands that the car should have to avoid collision. If there is an angle difference between the two vehicles superior to pi/16, we consider that it is a perpendicular intersection and compute the times tij. The speed of the car that comes to the intersection first is increased by steps of 0.1 until it reaches the maximal speed of 2 or until it has enough advance to avoid collision, whereas the other car decreases its speed until a minimum of 0.1 or until it is slow enough to avoid collision. This speed change is done in one time step, which implies that the vehicle can attain their top and minimal speed in one time step. To have a more realistic model of the vehicle, with limited acceleration and deceleration, the code can be slightly modified by changing the conditions on the speed change: the vehicles augment or decrease their speeds until they reach their maximum and minimum speed attainable during the next time step, or until the collision is avoided. The maximum speed attainable during the next time step is related to the maximum acceleration this way:

$$\max\_speed\_next = current\_speed + \max\_acceleration * time\_step$$

The minimum speed attainable during the next time step satisfies a similar equation. We kept our less realistic implementation, because bounding the acceleration and deceleration would not give us more insight on this strategy, it would only change the minimum alert radius needed. The minimum alert radius ensuring collision avoidance can be computed easily if the vehicles can stop instantaneously: it is the distance between the two Ci1 points when their intersecting angle is the smallest one taken into account (pi/16 for our case). If the vehicles have limited acceleration and deceleration, the radius needed is increased by the distance for the vehicles to stop multiplied by cos(pi/16). If we were to apply this strategy in the real world, it is clear that the radius alert will be greater than the minimum one, for safety reasons (the maximum deceleration can be influenced by the road condition or the maintenance level of the car) and also for comfort reasons (smoother actions can be taken if there is more time until intersection).

In case of head-on intersection, we make the vehicle turn right. It would be better that it turns in the direction opposite to the incoming car. It still works though, as the vehicle will be capable of avoiding collision by accelerating or decelerating if the other one turns in the wrong direction, as long as the alert zone is big enough.

5.6 Noncooperative cost function minimization

This code is a combination of the previous codes: it uses fmincon nearly the same way as in our first code, but there is a "while" loop so that the simulation stops only when the vehicles have reached their goals. During each loop, the main program makes one vehicle move by optimizing its commands, then tests if there are any collisions, and finally does the same for the other car. Thus, this code can be relatively easily generalized for more vehicles. There is only one payoff for each vehicle, which takes into account both the goals and the other vehicle. There is no changing in behavior that depends on an alert.

In optimcar, the cost function is defined. The repulsion between the vehicles can be implicitly influenced by the constant K in optimcarMain. The simulcar function is the same as in our second code.

So that less computational power is required, this code can be written the same way as the few ones above, which is with discreet angle selection. During each loop, each vehicle computes the cost function value at the few points it can reach at the next time step, and chooses the minimal one. A gain on the angle command ensures that this point can be attained. The main disadvantage of this way of coding is that the trajectories cannot be tracked by a real vehicle. Nevertheless, this is not the reason why we did not choose this solution. The main motivation was our desire to test the capabilities of the fmincon function before writing the next code. Being satisfied with the results provided here, we used fmincon extensively afterwards, and it is only then that we noticed severe limitations in this MATLAB function.

5.7 Cooperative computer-generated path optimization

This last code is a combination of what we have previously written. As always, optimcarMain calls optimcar during each loop to find an optimized set of commands for each vehicle. Simulcar is called whenever we need to know the path of a vehicle for a given set of inputs.

OptimcarMain tests if the vehicles are too close to each other, and optimizes their path in function of that. The value of the variable "alert" is one when the vehicles are within the alert zone of one another, and zero otherwise. This variable is passed to optimcar, which computes a different cost function whether there is an alert or not. Thus, when there is no alert, the cost function is simply the sum of the distance to the vehicles' respective goal, whereas the distance between the vehicles is taken into account when there is a danger of collision.

As in the one-car simulation, we specify the number of pairs of commands we want to optimize during each time step. The results are good with five pairs (a vector of size ten), but it takes several minutes for MATLAB to do the task.

There are many different cost functions we tried for the "alert" case: we tried to put in the inverse of the integral of the distance between the cars, but collision are not always avoided that way, partly because of the way fmincon tries to find the best solution. Another idea was to increase the cost function by a constant (like 100) each time there is a collision on the searched trajectory. To test if there any collisions on the trajectory, we specify the same time step in the ode45 function for both vehicles, then we compute at each of the times given back by this function the distance between the vehicles. If it is smaller than the sum of the radii of the vehicles, then we add the constant in the cost function. The problem is that fmincon cannot handle very well cost functions that have some steps, where the derivative is infinite. It still works when we choose the constant appropriately and when the number of commands to optimize is large enough. This solution was finally chosen.

As pointed out in section 3.6, if the initial command vector given to fmincon (called X0 in its prototype) is toward the other vehicle, it is possible that a collision-free trajectory is not found. The way we coded, however, states that the initial vector is the previous command of the vehicles. To have better results with fmincon, a slight offset in the angle might help us find better solutions. This is also the reason for not initializing the phi and psi vectors towards the goals at the beginning.

The simulcar function has been slightly changed, but it yields exactly the same results as in the one-car simulation (our first code). The way it is done here is more elegant though, as there is a subfunction named eqcar in simulcar that is called via ode45. This subfunction has the same purpose as our previous simulcar modules, as it returns the derivative of the states of the vehicle given the inputs. Thus, the whole differential equation integration is done in simulcar. We did that in order to be able to give parameters in a different vector than the states, as in optimcar. Indeed, in the previous versions of simulcar, we had to concatenate the state vector and the input vector (along sometimes with other parameters such as the total simulation time), and specify that their derivative was zero. This can cause some numerical problems, in case their derivative is not exactly zero but a very small number, and this can also increase the computation time, as the parameters are treated like states that could possibly change. There was another possibility to pass the parameters, only found later: we can use the p1 and p2 arguments in ode45.

6 Conclusion

6.1 Possible project advancement

Some possible improvement that can be done in the continuity of this project: the simulations can be enhanced by taking into account fixed obstacles, such as roads, or moving non-cooperating obstacles, such as humans. A fusion of different strategies can be done to generate a path when all these parameters are considered. Other strategies can be proposed and simulated. The codes already written (section 8) can be helpful for implementing them.

All these proposed strategies can be tested and validated on experimental robots, such as the Khepera. In order to do that with some particular strategies, fmincon has to be adapted to the platform. In a more distant future, the strategies, or more specifically, the acceleration-deceleration strategy, can be implemented on real cars, like the Smarts already used for experiments by EPFL.

6.2 General conclusion

In this semester project, several strategies inspired by game theory were proposed and simulated for the collision avoidance problem. The choice of one of these strategies for a specific application depends on the environment and other parameters, such as the possibility for the vehicles to communicate between each other at a certain distance, their maneuverability and the required level of safety.

For an application on crossroads, the acceleration-deceleration strategy (3.4) is arguably the best one. In cases where the vehicles cannot change their speed easily, such as in the air or on sea, we can adapt this strategy by making the vehicles turn to augment their advance or go behind the other vehicles as much as possible, instead of respectively accelerating or braking. Another solution would be to enhance the cooperative computer-generated path optimization strategy, as explained in the section 3.6, so that it can be applied. Because of the human lives being involved, a high level of safety is needed for the air and sea transportation, and it can be desirable for the vehicles to use the optimal evasion strategy (3.2) in case they temporarily lose any means of communicating between each other.

The strategies that can be used in mobile robotics depend on the environment, the level of intelligence of the robots and the application itself. If there is a lot of obstacles and not much space to navigate in, the optimal evasion strategy is to be excluded. If the robots have very limited computation power, the computer-generated path optimization strategy is not adequate; if they have big inertia and small sensor range, the acceleration-deceleration might not be applicable. Obviously, they have to know their relative positions and speed, for instance with a shared mapping and localization, in order to use cooperative strategies. If they can only rely on their range sensors, it can still be possible to fuse their desired goals and positions in each robot's referential, and then use a collaborative strategy, but one can argue that if they do not even cooperate for the mapping of the environment, there is no reason for them to cooperate when they cross each other. Thus, for these cases, or if there is a need for simplicity, the potential field descent (3.5) can be used, but one has to be aware of the local minima problem.

Nowadays, most of the mobile robots do not have any strategies when it comes to path planning that takes into account their "colleagues": they consider each other in the best case as moving obstacles (like humans), and in the worst case, as fixed obstacles (like walls). Then they apply their collision avoidance techniques, which can be advanced - like path planning using a map or moving obstacle tracking - or more simple, like going around the obstacle (wall following) or stopping. With these robots starting to communicate between each other and even cooperate for some tasks (distributed mapping) to save time and be more effective, it is even more surprising that no more effort is done in this domain.

One can notice that our current technological advance is nearly unused in these domains and that we rely on humans most of the time for our safety: cars are driven by humans and the number of deaths on the roads is seldom decreasing from one year to another; planes have some digital monitoring systems and safety devices, such as proximity alarms, but most of them only urge pilots to avoid collision themselves; air-traffic controllers have huge responsibilities in avoiding mid-air collisions in a saturated sky and are only helped by an alarm that goes off when two planes are too close to each other; sailors are also only assisted by an alarm placed on the front of the boat, by a sonar or a GPS, but the maneuvers are to be done by themselves. One does not have to think long before recalling tragedies occurred because of human inattention or lack of assistance, either on ground, in the air or on sea, and a need for more support and for an automatic dealing of conflicts can be critical in the few years to come.

7 References

[1] R. Isaacs: *Differential Games*, Dover Publications, 1999

[2] A. M. Bayen, S. Santhanam, I. Mitchell and C. Tomlin: A differential game formulation of alert levels in ETMS data for high altitude traffic, in *GNC conference*, 2003

[3] C. Tomlin, G. J. Pappas and S. Sastry: Noncooperative Conflict Resolution, in *Proceedings of the 36th Conference on Decision & Control*, San Diego, CA, 1997
[4] G. Inalhan, D. M. Stipanovic and C. Tomlin: Decentralized Optimization with Application to Multiple Aircraft Coordination
[5] K. M. Krishna and H. Hexmoor: Reactive Collision Avoidance of Multiple Moving Agents by Cooperation and Conflict Propagation
[6] K. M. Krishna and P. K. Kalra: Detection, Tracking and Avoidance of Multiple Dynamic Objects, *Journal of Intelligent and Robotic Systems,* 33: 371-408, Kluwer Academic Publishers, Netherlands, 2002
[7] R. Alami, T. Siméon and K. M. Krishna: On the influence of sensor capacities and environment dynamics onto collision-free motion plans
[8] A. E. Bryson and Y. Ho: *Applied Optimal Control; Optimization, Estimation and Control*, Routledge, 1975

8 Appendix: MATLAB code

8.1 One-vehicle simulation using fmincon

8.1.1 OptimcarMain.m

```
%One-vehicle simulation: we minimize a cost function using fmincon. We give
%a set of the vehicle commands and MATLAB optimizes them.
%This module calls the optimcar function, in which we specify the cost
%function to optimize. The relationship between the cost function and the
%commands are implicit via the simulcar function, in which the dynamics of
%the vehicle are specified.

x1a=2;          %1st coordinate of the goal
x2a=3;          %2nd coordinate of the goal
T=4;            %time given to try to reach the goal
param=[x1a x2a T]; %parameters to give to the other functions
com=[0.2,0.2,0.2,0.9,0.9,0.9,0.2,0.9]; %initial set of commands to be optimized
nbCom = length(com);        %numbers of commands to optimize
ubound=ones(nbCom,1);       %upper bound of the values of the commands
lbound=-ones(nbCom,1);      %lower bound of the values of the commands

%optimization with fmincon
[vopt,fvalOpt] = fmincon(@(x)optimcar(x,param),com,[],[],[],[],lbound,ubound);

disp('Optimized commands')
vopt
disp('Final value of the cost function')
fvalOpt

%simulation of the trajectory using the optimized set of commands
[topt,xopt]=ode45(@simulcar,[0 T],[0 0 0 vopt T]);

figure(1);
clf
hold on                               %plot everything on the same figure
plot(xopt(:,1), xopt(:,2));           %plot the trajectory
plot(x1a,x2a,'x');                    %show the goal
plot(xopt(end,1),xopt(end,2),'o');    %show the final position
hold off;
```

8.1.2 Optimcar.m

```
%returns the cost function value
%@ v: set of commands
%@ param: parameters of the simulation (goal, time)

function f=optimcar(v,param)

%parameters retrieval
```

```
x1a=param(1);
x2a=param(2);
T=param(3);

%integrates the differential equations of the vehicle
[t,x]=ode45(@simulcar,[0 T],[0 0 0 v T]);

%final position of the vehicle
x1T=x(end,1);
x2T=x(end,2);

%cost function: distance to goal
f=(x1a-x1T)^2+(x2a-x2T)^2;
```

8.1.3 Simulcar.m

```
%The function returns the derivative of the states in d
%time t
%state vector x

function d=simulcar(t,x)

%retrieves the final time given in parameter
T=x(end);

%finds which pair of command corresponds to the current integration time t
%n is the number of pair of commands
n = floor((length(x)-3-1)/2);
if (t == 0)
   ind = 1;
else
   ind= ceil(t/(T/n));
end

%pair of commands corresponding to t
v1=x(ind+3);
v2=x(n+ind+3);

%derivative of the states
d=zeros(size(x));
d(1)=v1*cos(x(3));
d(2)=v1*sin(x(3));
d(3)=v2;
```

8.2 Collision avoidance at next time

8.2.1 OptimcarMain.m

```
%Two-car simulation with simple motion
%Simple collision avoidance: each car makes sure that it won't collide with
%the other one in the next time step
```

```
%coordinates of the 1st car's goal
goal1x=1;
goal1y=2;

%coordinates of the 2nd car's goal
goal2x=2;
goal2y=1;

T=0;            %time initialisation
Tbase=0.01;     %step of time
thetaref=[0 pi/4 pi/2 3*pi/4 pi 5*pi/4 3*pi/2 7*pi/4];  %angles the car can choose
x1=[1 0 pi];    %initial conditions of the 1st car
x2=[0 1 0];     %initial conditions of the 2nd car
phi=[1 0];      %control variable of the 1st car
psi=[1 0];      %control variable of the 2nd car
K=100;          %gain to make the cars turn fast enough for the simple motion
angle_opt=1;    %optimum angle to choose to go to the goal
l=0.05;         %distance to goal to attain
r_car=0.01;     %radius of the car (or the safe zone around the car)

figure(1);
clf;
hold on
plot(goal1x,goal1y,'x');
plot(goal2x,goal2y,'rx');

%cost function: sum of the distances to goal
J=(goal1x-x1(1))^2+(goal1y-x1(2))^2+(goal2x-x2(1))^2+(goal2y-x2(2))^2;

%conservative condition so that we are sure that each car is at least at a
%distance l from its goal
while J>l^2
   %car 1
   Jnext2=J;
   %finds the angle that minimizes the cost function with the
   %condition that the next position doesn't create a collision
   for i=1:length(thetaref)
      if (x2(1)-(x1(1)+phi(1)*Tbase*cos(thetaref(i))))^2 + (x2(2)-(x1(2)+phi(1)*Tbase*sin(thetaref(i))))^2 >
(2*r_car)^2
         Jnext1=(goal1x-(x1(1)+phi(1)*Tbase*cos(thetaref(i))))^2+(goal1y-
(x1(2)+phi(1)*Tbase*sin(thetaref(i))))^2+(goal2x-x2(1))^2+(goal2y-x2(2))^2;
         if Jnext1<Jnext2
            Jnext2=Jnext1;
            angle_opt=i;
         end
      end
   end
   thetaref_opt=thetaref(angle_opt);
   %command of the 1st car with a speed of one and the best angle
   phi=[1 K*(thetaref_opt-x1(end,3))];

   %trajectory simulation with the command found
   [t,x1]=ode45(@simulcar,[0 Tbase],[x1 phi]);
   plot(x1(:,1), x1(:,2));
   %current position of the 1st car
   x1=x1(end,1:3);

   %cost function value update
   J=(goal1x-x1(1))^2+(goal1y-x1(2))^2+(goal2x-x2(1))^2+(goal2y-x2(2))^2;
```

```
    %car 2
    Jnext2=J;
    %finds the angle that minimizes the cost function with the
    %condition that the next position doesn't create a collision
    for i=length(thetaref):-1:1
        if (x1(1)-(x2(1)+psi(1)*Tbase*cos(thetaref(i))))^2 + (x1(2)-(x2(2)+psi(1)*Tbase*sin(thetaref(i))))^2 >
r_car^2
            Jnext1=(goal2x-(x2(1)+psi(1)*Tbase*cos(thetaref(i))))^2+(goal2y-
(x2(2)+psi(1)*Tbase*sin(thetaref(i))))^2+(goal1x-x1(1))^2+(goal1y-x1(2))^2;
            if Jnext1<Jnext2
                Jnext2=Jnext1;
                angle_opt=i;
            end
        end
    end
    thetaref_opt=thetaref(angle_opt);
    %command of the 2nd car with a speed of one and the best angle
    psi=[1 K*(thetaref_opt-x2(end,3))];

    %trajectory simulation with the command found
    [t,x2]=ode45(@simulcar,[0 Tbase],[x2 psi]);
    plot(x2(:,1), x2(:,2), 'r');
    %current position of the 2nd car
    x2=x2(end,1:3);

    %cost function value update
    J=(goal1x-x1(1))^2+(goal1y-x1(2))^2+(goal2x-x2(1))^2+(goal2y-x2(2))^2;
    %update of the time to reach goal
    T=T+Tbase;

end

plot(x1(1), x1(2),'o');
plot(x2(1), x2(2),'ro');
xlabel('x');
ylabel('y');

disp('Time elapsed in the simulation')
disp(T)
disp('Final cost function value')
disp(J)
```

8.2.2 Simulcar.m

```
%The function returns the derivative of the states in d
%time t
%state vector x
function d=simulcar(t,x)

% parameters retrieval
v1=x(4);
v2=x(5);

d=zeros(size(x));

%derivatives of the states
d(1)=v1*cos(x(3));
d(2)=v1*sin(x(3));
```

```
d(3)=v2;
```

8.3 Optimal evasion

8.3.1 OptimcarMain.m

```
%Two-car simulation with bounded radius of curvature.
%Each car tries to escape the other one when in alert
%zone by taking the worst case scenario.

%coordinates of the 1st car's goal
goal1x=0.25;
goal1y=0.25;
%coordinates of the 2nd car's goal
goal2x=0.5;
goal2y=0;
T=0;            %time initialization
Tbase=0.01;      %time step
thetaref=[0 pi/4 pi/2 3*pi/4 pi 5*pi/4 3*pi/2 7*pi/4];  %angles the car can choose
thetaref_current=[5 1];         %current angles of the cars
x1=[0.25 -0.25 thetaref(thetaref_current(1))];       %initial conditions of the 1st car
x2=[0 0 thetaref(thetaref_current(2))];     %initial conditions of the 2nd car
phi=[1 0];        %control variable of the 1st car
psi=[1 0];        %control variable of the 2nd car
K=100;           %gain to make the cars turn fast enough and go to the discreet points we want
angle_opt=1;      %optimum angle to choose to go to the goal
l=0.02;          % distance to goal to attain
r_car=0.02;       %radius of the car
alertzone=0.035;   %radius of the alert disc

%value is 1 when goal reached
goal1_achieved=0;
goal2_achieved=0;

%worst next position the other car can be at the next time step
x1_worstnext=[0 0];
x2_worstnext=[0 0];

figure(1);
clf;
hold on
plot(goal1x,goal1y,'x');
plot(goal2x,goal2y,'rx');

%computes the distances to goal and the cost function value
dist_to_goal1=(goal1x-x1(1))^2+(goal1y-x1(2))^2;
dist_to_goal2=(goal2x-x2(1))^2+(goal2y-x2(2))^2;
J=dist_to_goal1+dist_to_goal2;

while goal1_achieved==0 | goal2_achieved==0
  %compute distance between the cars
  dist = (x2(1)-x1(1))^2 + (x2(2)-x1(2))^2;

  %car 1 moves if it hasn't reached its goal
  if (goal1_achieved==0)

    if dist > alertzone^2
      %no alert

      %finds the best angle to minimize the cost function
```

```
        %there are 3 angles to check (straight ahead  and on the sides)
        first_loop=1;
        for i=max([1,thetaref_current(1)-1]): min([length(thetaref),thetaref_current(1)+1])
           Jnext1=(goal1x-(x1(1)+phi(1)*Tbase*cos(thetaref(i))))^2+(goal1y-
(x1(2)+phi(1)*Tbase*sin(thetaref(i))))^2+dist_to_goal2;
           if first_loop==1
              Jnext2=Jnext1;
              angle_opt=i;
              first_loop=0;
           else if Jnext1<Jnext2
                 Jnext2=Jnext1;
                 angle_opt=i;
              end
           end
        end
        %in case the angle is currently zero degree, we must check at
        %the largest angle of theta_ref
        if thetaref_current(1)==1
           Jnext1=(goal1x-(x1(1)+phi(1)*Tbase*cos(thetaref(end))))^2+(goal1y-
(x1(2)+phi(1)*Tbase*sin(thetaref(end))))^2+dist_to_goal2;
           if Jnext1<Jnext2
              Jnext2=Jnext1;
              angle_opt=length(thetaref);
           end
        end
        %in case the angle is currently the largest one of theta_ref, we must
        %check at zero degree
        if thetaref_current(1)==length(thetaref)
           Jnext1=(goal1x-(x1(1)+phi(1)*Tbase*cos(thetaref(1))))^2+(goal1y-
(x1(2)+phi(1)*Tbase*sin(thetaref(1))))^2+dist_to_goal2;
           if Jnext1<Jnext2
              Jnext2=Jnext1;
              angle_opt=1;
           end
        end

     else
        %collision avoidance

        %the car tries to maximize the distance to the worst next
        %position of the other car (which is computed approximately)
        dist_next1=0;
        dist_next2=0;
        for i=max([1,thetaref_current(1)-1]): min([length(thetaref),thetaref_current(1)+1])
           x2_worstnext(1)=x2(1)+psi(1)*Tbase*cos(psi(2));
           x2_worstnext(2)=x2(2)+psi(1)*Tbase*sin(psi(2));
           dist_next1=(x2_worstnext(1)-(x1(1)+phi(1)*Tbase*cos(thetaref(i))))^2 + (x2_worstnext(2)-
(x1(2)+phi(1)*Tbase*sin(thetaref(i))))^2;
           if dist_next1>dist_next2
              dist_next2=dist_next1;
              angle_opt=i;
           end
        end
        %in case the angle is currently zero degree, we must check at
        %the largest angle of theta_ref
        if thetaref_current(1)==1
           dist_next1=(x2_worstnext(1)-(x1(1)+phi(1)*Tbase*cos(thetaref(end))))^2 + (x2_worstnext(2)-
(x1(2)+phi(1)*Tbase*sin(thetaref(end))))^2;
           if dist_next1>dist_next2
              dist_next2=dist_next1;
              angle_opt=length(thetaref);
```

```
            end
        end
        %in case the angle is currently the largest one of theta_ref, we must
        %check at zero degree
        if thetaref_current(1)==length(thetaref)
            dist_next1=(x2_worstnext(1)-(x1(1)+phi(1)*Tbase*cos(thetaref(1))))^2 + (x2_worstnext(2)-
(x1(2)+phi(1)*Tbase*sin(thetaref(1))))^2;
            if dist_next1>dist_next2
                dist_next2=dist_next1;
                angle_opt=1;
            end
        end
    end

    %best angle found
    thetaref_current(1)=angle_opt;
    thetaref_opt=thetaref(angle_opt);
    %optimal command (speed fixed at 1)
    phi=[1 K*npi2pi(thetaref_opt-x1(end,3),'radian')];
    %finds the trajectory given the command that was just found
    [t,x1]=ode45(@simulcar,[0 Tbase],[x1 phi]);
    plot(x1(:,1), x1(:,2));

    %new states of the 1st car
    x1=x1(end,1:3);

    %update its distance to goal
    dist_to_goal1=(goal1x-x1(1))^2+(goal1y-x1(2))^2;
  end

  %update distance between the cars
  dist = (x2(1)-x1(1))^2 + (x2(2)-x1(2))^2;

  %car 2 moves if it hasn't reached its goal
  if goal2_achieved==0
    if dist > alertzone^2
        %no alert

        %finds the best angle to minimize the cost function
        %there are 3 angles to check (straight ahead  and on the sides)
        J=dist_to_goal1 + dist_to_goal2;
        first_loop=1;
        for i= min([length(thetaref),thetaref_current(2)+1]):-1:max([1,thetaref_current(2)-1])
            Jnext1=(goal2x-(x2(1)+psi(1)*Tbase*cos(thetaref(i))))^2+(goal2y-
(x2(2)+psi(1)*Tbase*sin(thetaref(i))))^2+(goal1x-x1(1))^2+(goal1y-x1(2))^2;
            if first_loop==1
                Jnext2=Jnext1;
                angle_opt=i;
                first_loop=0;
            else if Jnext1<Jnext2
                    Jnext2=Jnext1;
                    angle_opt=i;
                end
            end
        end
        %in case the angle is currently zero degree, we must check at
        %the largest angle of theta_ref
        if thetaref_current(2)==1
            Jnext1=(goal2x-(x2(1)+psi(1)*Tbase*cos(thetaref(end))))^2+(goal2y-
(x2(2)+psi(1)*Tbase*sin(thetaref(end))))^2+(goal1x-x1(1))^2+(goal1y-x1(2))^2;
            if Jnext1<Jnext2
            Jnext2=Jnext1;
            angle_opt=length(thetaref);
         end
      end
      %in case the angle is currently the largest one of theta_ref, we must
      %check at zero degree
      if thetaref_current(2)==length(thetaref)
         Jnext1=(goal2x-(x2(1)+psi(1)*Tbase*cos(thetaref(1))))^2+(goal2y-
(x2(2)+psi(1)*Tbase*sin(thetaref(1))))^2+(goal1x-x1(1))^2+(goal1y-x1(2))^2;
         if Jnext1<Jnext2
            Jnext2=Jnext1;
            angle_opt=1;
         end
      end
   else
      %collision avoidance

      %the car tries to maximize the distance to the worst next
      %position of the other car (which is computed approximately)
      dist_next1=0;
      dist_next2=0;
      for i= min([length(thetaref),thetaref_current(2)+1]):-1:max([1,thetaref_current(2)-1])
         x1_worstnext(1)=x1(1)+phi(1)*Tbase*cos(phi(2));
         x1_worstnext(2)=x1(2)+phi(1)*Tbase*sin(phi(2));
         dist_next1=(x1_worstnext(1)-(x2(1)+psi(1)*Tbase*cos(thetaref(i))))^2 + (x1_worstnext(2)-
(x2(2)+psi(1)*Tbase*sin(thetaref(i))))^2;
         if dist_next1>dist_next2
            dist_next2=dist_next1;
            angle_opt=i;
         end
      end
      %in case the angle is currently zero degree, we must check at
      %the largest angle of theta_ref
      if thetaref_current(2)==1
         dist_next1=(x1_worstnext(1)-(x2(1)+psi(1)*Tbase*cos(thetaref(end))))^2 + (x1_worstnext(2)-
(x2(2)+psi(1)*Tbase*sin(thetaref(end))))^2;
         if dist_next1>dist_next2
            dist_next2=dist_next1;
            angle_opt=length(thetaref);
         end
      end
      %in case the angle is currently the largest one of theta_ref, we must
      %check at zero degree
      if thetaref_current(2)==length(thetaref)
         dist_next1=(x2_worstnext(1)-(x1(1)+phi(1)*Tbase*cos(thetaref(1))))^2 + (x2_worstnext(2)-
(x1(2)+phi(1)*Tbase*sin(thetaref(1))))^2;
         if dist_next1>dist_next2
            dist_next2=dist_next1;
            angle_opt=1;
         end
      end
   end

   %best angle found
   thetaref_current(2)=angle_opt;
   thetaref_opt=thetaref(angle_opt);
   %optimal command (speed fixed at 1)
   psi=[1 K*npi2pi(thetaref_opt-x2(end,3),'radian')];
   %finds the trajectory given the command that was just found
   [t,x2]=ode45(@simulcar,[0 Tbase],[x2 psi]);
   plot(x2(:,1), x2(:,2),'r');
```

```
        %new states of the 2nd car
        x2=x2(end,1:3);

        %update its distance to goal
        dist_to_goal2=(goal2x-x2(1))^2+(goal2y-x2(2))^2;
    end

    %The cars stop when they reach their goals
    if dist_to_goal1<l^2
        goal1_achieved=1;
    end
    if dist_to_goal2<l^2
        goal2_achieved=1;
    end

    %update cost function value and time
    J=dist_to_goal1+dist_to_goal2;
    T=T+Tbase;

end

plot(x1(1), x1(2),'o');
plot(x2(1), x2(2),'ro');
xlabel('x');
ylabel('y');

disp('Total time of simulation');
disp(T);
disp('Final cost function value');
disp(J);
```

8.3.2 Simulcar.m

Same as in 8.2.2.

8.4 Acceleration – deceleration

8.4.1 OptimcarMain.m

```
%Main function
%Acceleration - deceleration to avoid perpendicular collision

%coordinates of the 1st car's goal
goal1x=0;
goal1y=0.05;
%coordinates of the 2nd car's goal
goal2x=1;
goal2y=0;

T=0;            %time initialization
Tbase=0.01;     %time step

%implicitly determines the radius of curvature (pi must be dividable by this number)
turning_angle_max=pi/8;

thetaref=[0:turning_angle_max:2*pi-turning_angle_max];  %angles the cars can choose from

thetaref_current=[8 1];                         %current angles of the cars
```

```
x1=[1 -0.05 thetaref(thetaref_current(1))];   %initial conditions of the 1st car
x2=[0 0 thetaref(thetaref_current(2))];        %initial conditions of the 2nd car
phi=[1 0];          %commands of the 1st car
psi=[1 0];          %commands of the 2nd car
K=100;              %gain to make the cars turn fast enough and go to the discreet points we want
angle_opt=1;        %optimum angle to choose to go to the goal
l=0.03;             %distance to goal to attain
r_car1=0.02;        %radius of the car 1
r_car2=0.02;        %radius of the car 2
alertzone=0.3;      %radius of the alert disc

%value is 1 when goal is reached
goal1_achieved=0;
goal2_achieved=0;

figure(1);
clf;
hold on
plot(goal1x,goal1y,'x');
plot(goal2x,goal2y,'rx');

%distances to goals and cost function
dist_to_goal1=(goal1x-x1(1))^2+(goal1y-x1(2))^2;
dist_to_goal2=(goal2x-x2(1))^2+(goal2y-x2(2))^2;
J=dist_to_goal1+dist_to_goal2;

while goal1_achieved==0 | goal2_achieved==0

  %distance between cars
  dist = (x2(1)-x1(1))^2 + (x2(2)-x1(2))^2;

  %tests for collision
  if dist< (r_car1+r_car2)^2
    disp('collision');
  end

  %the 1st car moves if it has not reached its goal
  if (goal1_achieved==0)
    if dist > alertzone^2
      %no alert
      %finds the best angle to minimize the cost function
      %there are 3 angles to check (straight ahead and +/- turning_angle_max)
      first_loop=1;
      for i=max([1,thetaref_current(1)-1]): min([length(thetaref),thetaref_current(1)+1])
        Jnext1=(goal1x-(x1(1)+phi(1)*Tbase*cos(thetaref(i))))^2+(goal1y-
(x1(2)+phi(1)*Tbase*sin(thetaref(i))))^2+dist_to_goal2;
        if first_loop==1
          Jnext2=Jnext1;
          angle_opt=i;
          first_loop=0;
        elseif Jnext1<Jnext2
          Jnext2=Jnext1;
          angle_opt=i;
        end
      end
      %in case the angle is currently zero degree, we must check at
      %2*pi-turning_angle_max
      if thetaref_current(1)==1
        Jnext1=(goal1x-(x1(1)+phi(1)*Tbase*cos(thetaref(end))))^2+(goal1y-
(x1(2)+phi(1)*Tbase*sin(thetaref(end))))^2+dist_to_goal2;
        if Jnext1<Jnext2
```

```
                Jnext2=Jnext1;
                angle_opt=length(thetaref);
            end
        end
        %in case the angle is currently 2*pi-turning_angle_max, we must
        %check at zero degree
        if thetaref_current(1)==length(thetaref)
            Jnext1=(goal1x-(x1(1)+phi(1)*Tbase*cos(thetaref(1))))^2+(goal1y-
(x1(2)+phi(1)*Tbase*sin(thetaref(1))))^2+dist_to_goal2;
            if Jnext1<Jnext2
                Jnext2=Jnext1;
                angle_opt=1;
            end
        end
        %best angle found
        thetaref_current(1)=angle_opt;
        thetaref_opt=thetaref(angle_opt);
        %optimal command (speed fixed at 1)
        phi=[1 K*npi2pi(thetaref_opt-x1(end,3),'radian')];
    else
        %collision avoidance
        %calls the collision function which returns the command to
        %prevent collision
        dec = collision(x1,phi,x2,psi,r_car1,r_car2);


        if dec(2)==-1
            %the car has to turn on the right
            angle_opt=thetaref_current(1)-1;
            if angle_opt<=0;
                angle_opt=length(thetaref);
            end
        elseif dec(2)==1
            %the car has to turn on the left
            angle_opt=thetaref_current(1)+1;
            if angle_opt>length(thetaref);
                angle_opt=1;
            end
        else
            %the car has to go straight ahead
            angle_opt=thetaref_current(1);
        end
        %angle of the car
        thetaref_current(1)=angle_opt;
        thetaref_opt=thetaref(angle_opt);
        %command to avoid collision
        phi=[dec(1) K*npi2pi(thetaref_opt-x1(end,3),'radian')];
    end

    %finds the trajectory given the command that was just found
    [t,x1]=ode45(@simulcar,[0 Tbase],[x1 phi]);
    plot(x1(:,1), x1(:,2));
    %current state of the 1st car
    x1=x1(end,1:3);
    plot(x1(1), x1(2),'o');
    dist_to_goal1=(goal1x-x1(1))^2+(goal1y-x1(2))^2;
end

%distance between the two cars
dist = (x2(1)-x1(1))^2 + (x2(2)-x1(2))^2;
```

```
  %the 2nd car moves if it has not reached its goal
  if goal2_achieved==0
          if dist > alertzone^2
        %no alert
        %finds the best angle to minimize the cost function
        %there are 3 angles to check (straight ahead and +/-
        %turning_angle_max)
                    J=dist_to_goal1 + dist_to_goal2;
        first_loop=1;
        for i= min([length(thetaref),thetaref_current(2)+1]):-1:max([1,thetaref_current(2)-1])
          Jnext1=(goal2x-(x2(1)+psi(1)*Tbase*cos(thetaref(i))))^2+(goal2y-
(x2(2)+psi(1)*Tbase*sin(thetaref(i))))^2+(goal1x-x1(1))^2+(goal1y-x1(2))^2;
          if first_loop==1
            Jnext2=Jnext1;
            angle_opt=i;
            first_loop=0;
          else if Jnext1<Jnext2
            Jnext2=Jnext1;
            angle_opt=i;
            end
          end
        end
        %in case the angle is currently zero degree, we must check at
        %2*pi-turning_angle_max
        if thetaref_current(2)==1
          Jnext1=(goal2x-(x2(1)+psi(1)*Tbase*cos(thetaref(end))))^2+(goal2y-
(x2(2)+psi(1)*Tbase*sin(thetaref(end))))^2+(goal1x-x1(1))^2+(goal1y-x1(2))^2;
          if Jnext1<Jnext2
            Jnext2=Jnext1;
            angle_opt=length(thetaref);
          end
        end
        %in case the angle is currently 2*pi-turning_angle_max, we must
        %check at zero degree
        if thetaref_current(2)==length(thetaref)
          Jnext1=(goal2x-(x2(1)+psi(1)*Tbase*cos(thetaref(1))))^2+(goal2y-
(x2(2)+psi(1)*Tbase*sin(thetaref(1))))^2+(goal1x-x1(1))^2+(goal1y-x1(2))^2;
          if Jnext1<Jnext2
            Jnext2=Jnext1;
            angle_opt=1;
          end
        end
        %best angle found
        thetaref_current(2)=angle_opt;
        thetaref_opt=thetaref(angle_opt);
        %optimal command (speed fixed at 1)
        psi=[1 K*npi2pi(thetaref_opt-x2(end,3),'radian')];
      else
        %collision avoidance
        %calls the collision function which returns the command to
        %prevent collision
        dec = collision(x2,psi,x1,phi,r_car2,r_car1);
        if dec(2)==-1
          %the car has to turn on the right
          angle_opt=thetaref_current(2)-1;
          if angle_opt<=0;
            angle_opt=length(thetaref);
          end
        elseif dec(2)==1
          %the car has to turn on the left
          angle_opt=thetaref_current(2)+1;
```

```
            if angle_opt>length(thetaref);
                angle_opt=1;
            end
        else
            %the car has to go straight ahead
            angle_opt=thetaref_current(2);
        end
        %angle of the car
        thetaref_current(2)=angle_opt;
        thetaref_opt=thetaref(angle_opt);
        %command to avoid collision
        psi=[dec(1) K*npi2pi(thetaref_opt-x2(end,3),'radian')];
    end

    %finds the trajectory given the command that was just found
    [t,x2]=ode45(@simulcar,[0 Tbase],[x2 psi]);
    plot(x2(:,1), x2(:,2),'r');
    %current state of the second car
    x2=x2(end,1:3);
    plot(x2(1), x2(2),'ro');

    dist_to_goal2=(goal2x-x2(1))^2+(goal2y-x2(2))^2;
  end
  %The cars stop when they reach the goal
  if dist_to_goal1<l^2
    goal1_achieved=1;
  end
  if dist_to_goal2<l^2
    goal2_achieved=1;
  end

  %update cost function value and time elapsed
  J=dist_to_goal1+dist_to_goal2;
  T=T+Tbase;

end

plot(x1(1), x1(2),'o');
plot(x2(1), x2(2),'ro');
disp('Total time of simulation');
disp(T);
disp('Final cost function value');
disp(J);
```

8.4.2 Collision.m

```
%returns the angles and the speed the cars should have to avoid collision
%@x1: state vector of the car we want to control
%@phi: current commands
%@x2: state vector of the other car
%@psi: commands of the other car (not used in this version)
%@r_car1: radius of the car we control
%@r_car2: radius of the second car
function decision=collision(x1,phi,x2,psi,r_car1,r_car2)


dec1=[phi];         % decision the car 1 should take
dec2=[psi];                 % decision the car 2 should take
intersection=[0 0];         % coordinates of the intersection

% coordinates of the limit points
c11=[0 0];
```

```
c12=[0 0];
c21=[0 0];
c22=[0 0];

% time at which the cars enter and exit the potential collision zone
t1= [0 0];
t2= [0 0];

%constant close to zero
epsilon=0.01;

dif_ang = npi2pi((x1(3)-x2(3)),'radian');      %difference between the angles of the 2 cars


% nearly parallel cars: head-on intersection
if abs(dif_ang)<pi/16 | abs(dif_ang)>15*pi/16
   dec1(2)=-1; %make the car turn right

% perpendicular intersection
else
   dec1(2)=0;  %no turn

   %finds the coordinates of the intersection between the two cars
   %trajectories
   den=sin(x1(3))*cos(x2(3))-sin(x2(3))*cos(x1(3));
   if den==0
      if abs(npi2pi(x1(3))) < epsilon
         intersection(1)=x2(1);
         intersection(2)=x1(2);
      else
         intersection(1)=x1(1);
         intersection(2)=x2(2);
      end
   else
      k2=((x2(2)-x1(2))*cos(x1(3))+(x1(1)-x2(1))*sin(x1(3)))/(sin(x1(3))*cos(x2(3))-sin(x2(3))*cos(x1(3)));
      intersection(1)=x2(1)+cos(x2(3))*k2;
      intersection(2)=x2(2)+sin(x2(3))*k2;
   end

   %computes the limit points
   dist=(r_car1+r_car2)/sin(abs(dif_ang));
   c11(1)=intersection(1)-dist*cos(x1(3));
   c11(2)=intersection(2)-dist*sin(x1(3));
   c12(1)=intersection(1)+dist*cos(x1(3));
   c12(2)=intersection(2)+dist*sin(x1(3));
   c21(1)=intersection(1)-dist*cos(x2(3));
   c21(2)=intersection(2)-dist*sin(x2(3));
   c22(1)=intersection(1)+dist*cos(x2(3));
   c22(2)=intersection(2)+dist*sin(x2(3));

   %avoids a division by zero to compute the times t1 and t2
   if (abs(npi2pi(x1(3))) < epsilon) | (abs(npi2pi(x1(3)-pi)) < epsilon)
      t1(1)=(c11(1)-x1(1))/(cos(x1(3))*phi(1));
      t1(2)=(c12(1)-x1(1))/(cos(x1(3))*phi(1));
   else
      t1(1)=(c11(2)-x1(2))/(sin(x1(3))*phi(1));
      t1(2)=(c12(2)-x1(2))/(sin(x1(3))*phi(1));
   end
   if (abs(npi2pi(x2(3))) < epsilon) | (abs(npi2pi(x2(3)-pi)) < epsilon)
      t2(1)=(c21(1)-x2(1))/(cos(x2(3))*psi(1));
      t2(2)=(c22(1)-x2(1))/(cos(x2(3))*psi(1));
   else
      t2(1)=(c21(2)-x2(2))/(sin(x2(3))*psi(1));
      t2(2)=(c22(2)-x2(2))/(sin(x2(3))*psi(1));
   end

   %another way to compute the times, but more computationally intensive,
   %and yields bizarrely worse results
   %     t1(1)=sqrt((c11(1)-x1(1))^2+(c11(2)-x1(2))^2)/phi(1);
   %     t1(2)=sqrt((c12(1)-x1(1))^2+(c12(2)-x1(2))^2)/phi(1);
   %     t2(1)=sqrt((c21(1)-x2(1))^2+(c21(2)-x2(2))^2)/psi(1);
   %     t2(2)=sqrt((c22(1)-x2(1))^2+(c22(2)-x2(2))^2)/psi(1);

   %tests if the collision zone is already passed by
   if (t1(1)>0 | t1(2)>0) & (t2(1)>0 | t2(2)>0)

      %the first car comes at the intersection first
      if t1(1)<t2(1) & t1(2)>t2(1)

         %accelerates until safe or until max speed of 2 reached
         while t1(2)>t2(1) & dec1(1)<2
            if abs(npi2pi(x1(3))) < epsilon
               t1(1)=(c11(1)-x1(1))/(cos(x1(3))*dec1(1));
               t1(2)=(c12(1)-x1(1))/(cos(x1(3))*dec1(1));
            else
               t1(1)=(c11(2)-x1(2))/(sin(x1(3))*dec1(1));
               t1(2)=(c12(2)-x1(2))/(sin(x1(3))*dec1(1));
            end
            dec1(1)=dec1(1)+ 0.1;
         end

      %the second car comes at the intersection first
      elseif t1(1)>t2(1) & t1(1)<t2(2)

         %decelerates until safe or until min speed of 0.1 reached
         while (t1(1)<t2(2) & dec1(1)>0.2)
            if abs(npi2pi(x1(3))) < epsilon
               t1(1)=(c11(1)-x1(1))/(cos(x1(3))*dec1(1));
               t1(2)=(c12(1)-x1(1))/(cos(x1(3))*dec1(1));
            else
               t1(1)=(c11(2)-x1(2))/(sin(x1(3))*dec1(1));
               t1(2)=(c12(2)-x1(2))/(sin(x1(3))*dec1(1));
            end
            dec1(1)=dec1(1)- 0.1;
         end
      end
   end
end
%returns the commands
decision=[dec1];
```

8.4.3 Simulcar.m

Same as in 8.2.2.

8.5 Noncooperative cost function minimization

8.5.1 OptimcarMain.m

```
%Two-car noncooperative collision avoidance algorithm. Each car tries to
%minimize its own cost function
```

```
%initialization
Tbase=0.1;              %time step
T=0;                    %initialization of the total simulation time
goal1=[0.5 1];          %coordinates of the 1st car's goal
goal2=[0 0.5];          %coordinates of the 2nd car's goal
x1=[0.5 0 pi/2];        %initial state of the 1st car
x2=[1 0.5 -pi];         %initial state of the 2nd car
ubound=[2;10];          %upper bound of the commands
lbound=[0.1;-10];       %lower bound of the commands
r_car=0.1;              %radius of each car
l=0.05;                 %distance to goal to attain

%Constant that determines the distance between the two cars.
%The bigger K, the more distance they will keep between them.
K=0.01;

%value is 1 when goal reached
goal1_achieved=0;
goal2_achieved=0;

figure(1);
clf;
hold on
plot(goal1(1),goal1(2),'x');
plot(goal2(1),goal2(2),'rx');

warning off optim:fmincon:SwitchingToMediumScale    %turns off warning message

%simulate until the goals are achieved
while goal1_achieved==0 | goal2_achieved==0

  % collision detection
  dist = (x2(1)-x1(1))^2 + (x2(2)-x1(2))^2;
  if dist< (2*r_car)^2
    disp('collision');
  end

  %car1 move
  if (goal1_achieved==0)
    [output] = fmincon(@(input)optimcar(input,[x1,x2,K,Tbase,goal1]),[1,10],[],[],[],[],lbound,ubound);

    %optimized commands
    phi = [output(1) output(2)];

    %simulation with these commands
    [t,x1]=ode45(@simulcar,[0 Tbase],[x1 phi]);
    plot(x1(:,1), x1(:,2));
    x1=x1(end,1:3);
    plot(x1(1), x1(2),'o');
  end

  % collision detection
  dist = (x2(1)-x1(1))^2 + (x2(2)-x1(2))^2;
  if dist< (2*r_car)^2
    disp('collision');
  end

  %car2 move
  if goal2_achieved==0
    [output] = fmincon(@(input)optimcar(input,[x2,x1,K,Tbase,goal2]),[1,10],[],[],[],[],lbound,ubound);
```

```
        %optimized commands
        psi = [output(1) output(2)];

        %simulation with these commands
        [t,x2]=ode45(@simulcar,[0 Tbase],[x2 psi]);
        plot(x2(:,1), x2(:,2),'r');
        x2=x2(end,1:3);
        plot(x2(1), x2(2),'ro');
    end

    %The cars stop when they reach the goal
    dist_to_goal1=(goal1(1)-x1(1))^2+(goal1(2)-x1(2))^2;
    dist_to_goal2=(goal2(1)-x2(1))^2+(goal2(2)-x2(2))^2;
    if dist_to_goal1<l^2
        goal1_achieved=1;
    end
    if dist_to_goal2<l^2
        goal2_achieved=1;
    end
    T=T+Tbase;
end
```

8.5.2 Optimcar.m

```
%Returns the value of the cost function for one car.
%The parameters are separated from the inputs so that only the commands are
%optimized
function f=optimcar(input,param)

%commands retrieval
v=[input(1) input(2)];

%parameters retrieval
x1=[param(1) param(2) param(3)];
x1a=param(end-1);
x2a=param(end);
xcar2=param(4);
ycar2=param(5);
T=param(end-2);
K= param(end-3);

[t,x]=ode45(@simulcar,[0 T],[x1 v]);

x1T=x(end,1);
x2T=x(end,2);

%potential-field-like cost function
f=(x1a-x1T)^2+(x2a-x2T)^2 + K/((xcar2-x1T)^2+(ycar2-x2T)^2);
```

8.5.3 Simulcar.m

Same as simulcar.m in the “collision avoidance at next time” code.

8.6 Cooperative computer-generated path optimization

8.6.1 OptimcarMain.m

```
%Cooperative path generation using fmincon:
%during each loop, a common cost function is minimized to find the future
%commands of each vehicle

T=0;               %time initialization
Tbase=1;           %time step
goal1=[3 0];       %coordinates of the 1st car's goal
goal2=[0 3];       %coordinates of the 2nd car's goal
x1=[0 3 -pi/2];    %initial state of the 1st car
x2=[3 0 pi];       %initial state of the 2nd car
phi=[1 0];         %control variables of the 1st car
psi=[1 0];         %control variables of the 2nd car
r_car=[0.05 0.05]; %radii of the cars
l=0.2;             %distance to goal to attain
lbound=[0;-10];    %lower bound on phi and psi
ubound=[1;10];     %upper bound on phi and psi
nbv=4;             %number of pairs of command to optimize
alertzone=2;       %radius of alert
tcol=[];           %stores the times at which there are collisions
fval=[];           %stores cost function values after each move

%value is 1 when goal reached
goal1_achieved=0;
goal2_achieved=0;

figure(1);
clf;
hold on
plot(goal1(1),goal1(2),'x');
plot(goal2(1),goal2(2),'rx');

%takes off common warning messages
warning off optim:fmincon:SwitchingToMediumScale
warning off MATLAB:nearlySingularMatrix

while goal1_achieved==0 | goal2_achieved==0
   if (x1(1)-x2(1))^2+(x1(2)-x2(2))^2 < alertzone^2
      %alert: alert parameter at 1
      param=[x1,x2,goal1,goal2,Tbase,nbv,r_car,1];
   else
      %no alert: alert parameter at 0
      param=[x1,x2,goal1,goal2,Tbase,nbv,r_car,0];
   end

   %finds the optimal commands
   [vopt,fvalopt] = fmincon(@(v)optimcar(v,param), horzcat(repmat(phi,1,nbv),repmat(psi,1,nbv)), [],
[],[],[],repmat(lbound,2*nbv,1),repmat(ubound,2*nbv,1));
   %optimal commands
   phi = [vopt(1:2*nbv)];
   psi = [vopt(2*nbv+1:4*nbv)];
   %finds the trajectories of the cars
   [t,x1]=simulcar(x1,phi,Tbase);
   [t,x2]=simulcar(x2,psi,Tbase);

   %test if there are any collisions on the given trajectory
   for i=1:length(x1)
      if (x1(i,1)-x2(i,1))^2+(x1(i,2)-x2(i,2))^2 < (r_car(1)+r_car(2))^2
```

```
            tcol=horzcat(tcol,T+t(i));
        end
    end
    plot(x1(:,1), x1(:,2));
    plot(x2(:,1), x2(:,2),'r');
    %new states of the cars
    x1=x1(end,1:3);
    x2=x2(end,1:3);
    plot(x1(1), x1(2),'o');
    plot(x2(1), x2(2),'ro');

    %stores cost function value
    fval=horzcat(fval,fvalopt);

    %the cars stop moving when both have reached their goals
    if (goal1(1)-x1(1))^2+(goal1(2)-x1(2))^2 <l^2
        goal1_achieved=1;
    end
    if (goal2(1)-x2(1))^2+(goal2(2)-x2(2))^2<l^2
        goal2_achieved=1;
    end

    %time update
    T=T+Tbase;

    %limits the number of iterations
    if T>10
        break
    end
end

%displays useful information
disp('Total time');
disp(T);
disp('cost function after each move')
disp(fval)

if tcol
    disp('Collision at time');
    disp(tcol);
else
    disp('No collision')
end
```

8.6.2 Optimcar.m

```
%Returns the cost function value given parameters and inputs. Fmincon
%uses this function to optimize the inputs
function f=optimcar(input,param)

%parameters retrieval
x1=[param(1:3)];
x2=[param(4:6)];
goal1=[param(7:8)];
goal2=[param(9:10)];
T=param(11);
nbv=param(12);
r_car=[param(13:14)];
alert=param(15);

%inputs retrieval
v1=[input(1:2*nbv)];        %set of commands of the 1st car
```

```
v2=[input(2*nbv+1:4*nbv)];  %set of commands of the 2nd car

%trajectory simulation with a set of inputs
[t1,x1]=simulcar(x1,v1,T);
[t2,x2]=simulcar(x2,v2,T);

%distances to goal
f1=(goal1(1)-x1(end,1))^2+(goal1(2)-x1(end,2))^2;
f2=(goal2(1)-x2(end,1))^2+(goal2(2)-x2(end,2))^2;

%the two vectors should have the same size
if size(x1) ~= size(x2)
   keyboard;
end

%cost function value is sum of the distances to goals if no alert
f=f1+f2;

%if alert=1, the cost function is increased if there are collisions between
%the given trajectories
if alert==1
   for i=1:length(x1)
      if (x1(i,1)-x2(i,1))^2+(x1(i,2)-x2(i,2))^2 < (r_car(1)+r_car(2))^2
         f=f1+f2+1000;
         break
      end
   end
end

%{
%another possibility to compute the cost function
K=0;          %constant used only if there is an alert
dist=1;       %initialization

%if risk of collision
if alert==1
   K=10;
   dist=0;
   %computes the integral of the distance between the 2 cars
   for i=1:length(x1)
      dist = dist + (x1(i,1)-x2(i,1))^2+(x1(i,2)-x2(i,2))^2;
   end
end

% if no alert, cost function value is the sum of the distances to goal, else
%there is a term with the integral of the distance between the 2 cars
f=f1+f2+(K/dist);
%}
```

8.6.3 Simulcar.m

```
%Simulcar integrates the differential equation.
%The subfunction eqcar is called by ode45.
function [t,x]=simulcar(x1,v1,T)

%integrates the diff eq
[t,x]=ode45(@eqcar,[0:T/40:T],[x1]);

%This subfunction returns the derivative of the states in d
%@param: time t
```

```
%@param: state vector x
function d=eqcar(t,x)
   %finds the commands for the corresponding interval
   if (t == 0)
      ind = 1;
   else
      ind= ceil(t/(2*T/length(v1)));
   end

   v=[v1(2*ind-1) v1(2*ind)];

   %derivative of the states
   d=zeros(size(x));
   d(1)=v(1)*cos(x(3));
   d(2)=v(1)*sin(x(3));
   d(3)=v(2);

end
end
```