Obstacle Avoidance Algorithm for a mobile robot

G.N.J.P Vermeulen

Final Bachelor Project Mechanical Engineering

Coaches: Dr.ir. L.F.P. Etman, Dr.ir. G. Naus
Summary

The Obstacle Avoiding Algorithm presented in this study is made for mobile robots. We specifically consider the soccer robots used in the robocup league. The specifications these robots have are used to develop the algorithm presented in this report.

In previous studies a rudimentary algorithm was developed. This algorithm used the Newton Method to determine the best possible direction to move in. A step in this Newton direction provides a point where the robot should move towards.

The total target function generates a 3d landscape where the Newton Direction can be used to find the best way along the slope. The Newton Method can be evaluated in all points provided by the total target function. The total target function consist a target function and all penalty functions and barrier functions. The Newton Method determines the first and second order derivatives and uses them to find the best direction.

The basic algorithm can generate a saw-tooth pattern, which is undesirable. Therefore the algorithm is extended with the speed and acceleration constraints the mobile robot has. The constraint can be used to generate a more stable and a more realistic path. These constraints solve the problem of the saw-tooth pattern the old algorithm has.

The experiments demonstrate that the new algorithm works, but is made for speed, and not safety. Collision can still occur. Also, the penalty factors influence the path greatly.

In the recommendation some strategies are provided to improve the Obstacle Avoiding Algorithm.
# Content

## SUMMARY  
2

## 1 - INTRODUCTION  
5

## 2 - THE ALGORITHM  
6
  2.1 - TARGET FUNCTION  
  2.2 - BOUNDARY FUNCTION  
  2.3 - BARRIER FUNCTION  
  2.4 - PENALTY FUNCTION  
  2.5 - NEWTON DIRECTION  
  2.7 - THE BASIC ALGORITHM  
6

## 3 - THE NEW ALGORITHM WITH CONSTRAINTS  
11
  3.1 - ADDING CONSTRAINTS  
  3.2 - OVERSHOOT  
  3.3 - THE NEW ALGORITHM  
11

## 4 - EXPERIMENTS WITH THE NEW ALGORITHM  
16
  4.1 - ONE FIXED OBSTACLE  
  4.2 - TWO FIXED OBSTACLES  
  4.3 - THREE FIXED OBSTACLES  
  4.4 - ONE MOVING OBSTACLE  
  4.5 - TWO MOVING OBSTACLES  
  4.6 - THREE MOVING OBSTACLES  
16

## 5 - OVERALL CONCLUSION AND RECOMMENDATIONS  
27
  5.1 - CONCLUSIONS ON THE OBSTACLE AVOIDING ALGORITHM  
  5.2 - CONCLUSIONS ON THE USE OF THE PENALTY FUNCTION  
  5.3 - RECOMMENDATIONS  
27

## ANNEX A  - MAKING THE HESSIAN POSITIVE DEFINITE  
29

## ANNEX B  - MATLAB CODE  
30
  B.1 - F.M  
  B.2 - DAA.M  
  B.3 - NMVEL.M  
  B.4 - GET_NEWTON_DIRECTION.M  
  B.5 - INVHESST.M  
  B.6 - HESSF.M  
  B.7 - HESSP.M  
  B.8 - GRADT.M  
  B.9 - GRADF.M  
  B.10 - GRADP.M  
  B.11 - B.M  
  B.12 - GET_POINT.M  
  B.13 - GET_POINT_OVERSHOOT.M  
  B.14 - MOVOB.M  
  B.15 - VARR.M  
  B.16 - P.M  
31
1 - Introduction

In the last couple of years mechanical engineers have built a variety of robots. Starting with only robotic arms to do simple, but very accurate work, nowadays more complex robots are being developed. There are now robots under development that have to see, think and act themselves without any human control. One might think of robots that can help people in hospitals to do some simple work. Getting a glass of water for people that can’t get out of bed for example.

There are also robots that play a game of soccer with nine other robots on the playing field. The robots have to see and detect objects as they move around. They must avoid all objects in their path. Not only to avoid damage to themselves and other objects, but also for the safety of humans that walk around the field.

During this Bachelor Final Project an Obstacle Avoidance Algorithm will be developed. This study is a continuation of the Obstacle Avoidance Algorithm studies of J.D.J.M van den Bulk and J.J.P van Heur. The main ideas with obstacle avoidance is the usage of the Newton direction and penalty functions. Two concepts for numerical optimization. The algorithm as developed by J.D.J.M van den Bulk was a concept of proof for non-moving objects and worked in principle. J.J.P van Heur extended the algorithm to be used by moving objects. These objects moved with a constant speed and in a fixed direction. In both studies it was assumed that the robot moved with a constant speed. In this study we will look more at the reality by introducing constraints the robot has.

The problem in this study may be formulated as follows; The mobile robot has a starting point and has to move to a target location. On the way from start location to target location he may encounter various objects which need to be avoided. These objects will be modelled via a boundary function which are extended with a barrier function. By using the Newton direction and the boundary and barrier functions we generate a three dimensional landscape in which the shortest distant to the target position is calculated.

In this report we will look at the excising algorithm. We will explain the use of functions and the existing problems. We will then show some experiments and explain the results and its conclusions. With these conclusions we will explain and present the updated algorithm. Finally the new algorithm will be tested.
2 - The Algorithm

Before we start the explanation of the various functions and methods in the algorithm we need some basic understanding of the used variables and constants. There are certain specifications for the mobile robot which we have to keep in mind. The mobile robot which will be used in our study will get a position update information every 1/32 of a second. In other words, the robot has a thirty-two Hertz information update. The received information contains positions for the robot, positions of all objects on the field and the target position. During this study the target position will be constant.

The robot and all objects have a diameter of 0.5 meter. In the algorithm the mobile robot will be a point in space and all objects will have a diameter of 1 meter. The physical dimension of the mobile robot is accounted for in the physical dimension of the objects. This is done because it is easier to use a point then it is to use a circular robot when determining a path.

The robot has a maximum speed 4 m/s and a maximum acceleration of 5 m/s².

2.1 - Target Function

The mobile robot has a starting point and a target point which he approaches as good as possible. To do this we use a target function. This function reads as follows:

\[ f(\vec{p}, \vec{p}_T) = (x_T - x)^2 + (y_T - y)^2 \]  \hspace{1cm} (2.1)

Here, \( \vec{p} = [x \ y] \), is the current position of the mobile robot and \( \vec{p}_T = [x_T \ y_T] \) is the target position. The minimum of the function is reached when \( \vec{p} = \vec{p}_T \). See also Figure 1.
2.2 - Boundary Function

All areas where the robot is not allowed to go are represented by a boundary. The boundary function gives information about the location and the shape and size of an object. The objects in this study are all circular robots. These objects have a centre $p_b$ and a radius $R = 0.25$ m. The mobile robot which uses the algorithm does not have a shape or size, and is therefore modelled as a point. This makes it easier to use in calculations. It is important to keep in mind that all boundaries need to be 0.25 meters bigger to compensate for the fact that the mobile robot is a point. The boundary function for a circular object reads:

$$g_j (\vec{p}_B, R) = - (x_B - x)^2 - (y_B - y)^2 + R^2$$

The area within the described curve is positive, while the outside will always be negative. $\vec{p}_B = [x_B, y_B]$ is the center of the object $j$.

The target function together with this boundary function $g_j \leq 0$ for all objects will generate an optimization problem where we need to find a feasible path from starting point to target location (i.e. the minimum).

2.3 - Barrier Function

Finding the minimum for the target function is easy. The difficult part is generating the path the robot has to follow without crossing any of the boundaries. To address this problem, we will introduce the barrier function for each object:

$$B_j (\vec{p}, \vec{p}_B, R) = \frac{1}{-g_j (\vec{p}, \vec{p}_B, R)}$$

The barrier function generates a ‘pillar’ with infinite height. See Figure 2. This barrier function is added to the target function which results in a total function that reads:

$$T (\vec{p}, \vec{p}_T, \vec{p}_B, R) = f (\vec{p}, \vec{p}_T) + \sum_j B_j (\vec{p}, \vec{p}_B, R)$$

Figure 2: The total target function when there is one boundary
2.4 - Penalty Function

The penalty function is the function that controls the importance of an obstacle. The current robot position \( \vec{p} \) determines if an object should be of a high priority within the total target function or not. The importance of the penalty fades away in terms of distance towards that object, so that only the important obstacles, those that are close to the robot, are taken into account when calculating the total target function.

The penalty is calculated by determining the distance between the mobile robot and the objects. The distance the robot moves toward or away from the objects is used to increase or decrease the penalty, and thus making the barrier function more or less important to the total target function.

\[
\begin{align*}
 r_{B_j}(\vec{p}, \vec{p}_{B_j}) &= r + e^{-\alpha d} \\
 d &= \sqrt{(x - x_{B_j})^2 + (y - y_{B_j})^2} < 0 \quad (2.5a) \\
 r_{B_j}(\vec{p}, \vec{p}_{B_j}) &= r - e^{-\alpha d} \\
 d &= \sqrt{(x - x_{B_j})^2 + (y - y_{B_j})^2} > 0 \quad (2.5b)
\end{align*}
\]

\( \alpha \) is a factor which is set to 2.0. This value is determined by experimenting with different values and 2.0 being the best. (See also the Bachelor Final Project of J.J.P. van Heur).

\[
d = \sqrt{(x - x_{B_j})^2 + (y - y_{B_j})^2}
\]

is the change in distance between the mobile robot and the obstacles.

The best range for the penalty function is between 0.05 and 1 because the penalty should be positive and not be too big. When it is bigger than one the mobile robot will always stay too far away from the object which is not always preferable. Therefore the penalty will always be set to 0.05 when it becomes smaller than 0.05 and will be set to 1 when it becomes bigger than 1.

The total target function will now read:

\[
T(\vec{p}, \vec{p}_T, \vec{p}_{B_j}, R) = f(\vec{p}, \vec{p}_T) + \sum r_{B_j} \cdot B_j(\vec{p}, \vec{p}_{B_j}, R) \quad (2.6)
\]

2.5 - Newton Direction

In the algorithm the path is determined by looking at the best direction for the robot to move at its current location. The method used is called the Newton Method. We can calculate the so called
Newton Direction, which represents the optimal direction in which a step should be taken. We define the Newton Direction as:

\[ \tilde{s}(\tilde{p}_k) = -H_T^{-1}(\tilde{p}_k) \cdot \nabla T(\tilde{p}_k) \quad (2.7) \]

Here, \( \nabla T \) is gradient of the total target function (2.6) and \( H_T^{-1} \) is the inverse of the Hessian matrix which describes the second order derivative of the total target function evaluated in point \( \tilde{p}_k \). \( \nabla T \) describes the first order derivative, or gradient, of the total target function.

The reason we use the Newton Method is the usage of the second derivative. This means that the slope is not only evaluated for the first derivative but also the second derivative and therefore it is possible to see the slope tilting left or right. That information is important because it can be used to move in the direction in which the slope is turning. An evaluation of the first order derivative would only determine the steepest descent of the slope, and not necessarily the direction of our target point.

Figure 4 shows a height profile and two methods to determine a path. The green line represents the steepest decent where only the first derivative is evaluated. The red line represent Newton’s method where we also take the second derivative into account.

Problems with this method lie in the minimum values of the function. Because the Newton method is a local method the path could get ‘stuck’ in the local minimum and cannot move further. In a real scenario this would probably never happen, because there is noise in the signal which causes the minimum values to jump. Also, the objects on the field will always be moving and this will also cause the minimum values to change. A different problem that can occur is in the so-called saddle points. These points can cause the function to point in the wrong direction, because there are two slopes down. One on either side. To solve this problem, it is important to make sure that the Hessian is positive definite (see also Annex A).

2.7 - The basic algorithm

Using the above functions the basic algorithm as developed by J.D.J.M van den Bulk and J.J.P. van Heur is presented next. In this algorithm a constant speed of 4 m/s is used which can be
represented by a step size of 0.125 m. This step size is obtained when multiplying the speed by the interval in which we receive information. $4 \cdot \frac{1}{32} = 0.125 \text{ m}$. The algorithm is as follows:

1. Set iteration step $k$ equal to $k=1$. Set tolerance factor $\delta$. Set starting values.

2. Inside the loop the following sequence will be used for every $k$:
   
   - Evaluate the Hessian $H_T(\tilde{p}_k)$ and gradient $\nabla T(\tilde{p}_k)$. Check for positive definite of Hessian matrix. If not, the Hessian will be adjusted so that it is positive definite (see also Annex A).
   
   - Determine the Newton Direction: $\tilde{s}(\tilde{p}_k) = -H_T^{-1}(\tilde{p}_k) \cdot \nabla T(\tilde{p}_k)$

   - Normalize the Newton Direction: $|\tilde{s}| = \frac{\tilde{s}}{\sqrt{s_x^2 + s_y^2}}$

   - Determine the step size: $\bar{\Delta q}_k = \tilde{s}(\tilde{p}_k) \cdot 0.125$

   - Determine the new point $\tilde{p}_{k+1} = \tilde{p}_k + \bar{\Delta q}_k$

3. If $\|\nabla T\| < \delta$, if not set $k=k+1$ and repeat the loop (step 2). Otherwise, terminate.
3 - The new algorithm with constraints

The algorithm previously developed as shown in chapter 2 still suffers from a couple of problems. One of them is the saw-tooth pattern that occurs at some points along the path. See also Figure 4. The saw-tooth pattern is due to the fixed step size. At certain steps a smaller step size, i.e. decreasing its speed, is necessary. After some analysis we found that a decrease in the step size could solve this problem. In reality this would mean a decrease in speed. Decreasing the step size to 1/16 of the robot’s maximum speed gave a better path, but when zoomed in very closely it still showed some saw-tooth pattern. See Figure 5.

This effect occurs because the new point is not the best point possible. Taking a fixed step size can cause the new point to be “too far”. This means that the point after that will guide the path back, causing a saw-tooth pattern. The path is zig-zagging along the perfect path.

The reason for this problem was the fact that a constant speed was used for the robot. In reality however, the robot has speed and acceleration constraints. The next step therefore, was a transition to a more real scenario by incorporating these constraints.

3.1 - Adding Constraints

To determine the new point $\vec{p}_{k+1}$ we need to know our speed and acceleration constraints.

Let us assume we have a speed $\vec{v}_k = \begin{bmatrix} v_x^k \\ v_y^k \end{bmatrix}$ and an acceleration $\vec{a}_k = \begin{bmatrix} a_x^k \\ a_y^k \end{bmatrix}$.

The constraints are $|\vec{v}_k| \leq 4 \text{ m/s}$ and $|\vec{a}_k| \leq 5 \text{ m/s}^2$. These constraints are given by the robot’s specifications. $t$ is the time parameter. Now we can determine the speed of the robot at each the next timestep $k+1$:

$$\vec{v}_{k+1} = \vec{v}_k + \vec{a}_k \cdot t$$

(3.1)
Our starting speed will be set to $\vec{v}_k (k = 0) = \frac{0 \text{m}}{s}$. This means it is assumed that the robot starts from a standstill.

We can plot a line in the $(x, y)$–plane which represents all points that lie in the Newton Direction $\underline{s}_k$. This line passes through point $\underline{p}_k$, therefore the line is represented by the formula:

$$y = \frac{s_y}{s_x} \cdot x - \frac{s_y}{s_x} \cdot p_x + p_y \tag{3.2}$$

This line represents all the points that are on the Newton Direction. Our robot needs to move towards one of these points. We can now determine the speed and acceleration that is required to move toward this line. See also Figure 5. In figure 5 we see the point $\underline{p}_X (x, y)$ and two possible Newton Directions (notice that it passes through point $\underline{p}_X (x, y)$). We also see a circle at the end of the speed vector that represents the field of the points we can reach by applying an acceleration on that speed. The radius of this circle can be described by $r = \frac{1}{2} \cdot |\underline{a}_k| \cdot t^2$, were we set $|\underline{a}_k| = \frac{v_{max} - |\underline{v}_k|}{t}$ (This acceleration has the speed constraint within it, to prevent the robot from going over its maximum speed).

With this radius, and the center of the circle at $\underline{p}_k + \underline{v}_k \cdot t$, the circle can be defined as:

$$(x - (p_x + v_x \cdot t))^2 + (y - (p_y + v_y \cdot t))^2 = r^2 \tag{3.3}$$

Consider two possible scenarios for the Newton Direction. One that intersects with the circle, and one that does not intersect with the circle. Let us look at the first scenario, when the line intersects with the circle. Because they intersect the x-value for the new point can be determined by substituting function (3.2) in function (3.3). We get:

$$(x - (p_x + v_x \cdot t))^2 + \left(\frac{s_y}{s_x} \cdot x - \frac{s_y}{s_x} \cdot p_x + p_y - (p_y + v_y \cdot t)\right)^2 = r^2 \tag{3.4}$$

Solving this equation gives us two solutions for $x$. We want the solution which gives us a new point where the acceleration in that new point is biggest. This can be determined by looking at the Newton Directions’ x-value. If the x-value for the Newton Direction is positive, the largest solution
for \( x \) is chosen, otherwise the smallest is chosen. The solution for \( x \) that is found can then be used to find the y-value by solving the line equation (3.2).

There is a special case for \( s_x = 0 \). If \( s_x = 0 \) we cannot use the method described above, because we would have to divide by zero. When \( s_x = 0 \) the Newton direction will point straight up or straight down. Therefore we know that when \( s_x = 0 \) our \( p_x^{k+1} = p_x^k \). We can find the y-value by substituting \( p_x \) into the circle equation (3.3):

\[
(p_x - (p_x + v_x \cdot t))^2 + (y - (p_y + v_y \cdot t))^2 = r^2
\]  

(3.5)

The equation generates two solution for the y-value. The correct solution is chosen by looking at the Newton direction. If it is pointing up, y-value > 0, the largest is chosen, if it points down, y-value < 0, the smallest is chosen.

3.2 - Overshoot

The second possible scenario that can occur is called ‘overshoot’. Overshoot means, that an acceleration big enough to get to a point on the newton direction line cannot be generated. Because a solution cannot be found, we want to come as close as possible to a solution. Therefore the point that is closest to our line will move. This means, that a line that is perpendicular to our newton direction line must be found, and the line should intersect with the center of the circle. See also Figure 6.

An easy way to determine the new point is to add three vectors together, namely \( \vec{p}_k + v_k \cdot t + \vec{q} \) where \( \vec{q} \) is \( \vec{q} = \frac{s_x}{|s_x|} \cdot r \) and \( r \) is a radius as in equation (3.3). Again, there can be two
possible solutions. \( s_\perp \) can point in two directions, namely \( s_\perp = \begin{bmatrix} \hat{s}_y \\ -\hat{s}_x \end{bmatrix} \) or \( s_\perp = \begin{bmatrix} -\hat{s}_y \\ \hat{s}_x \end{bmatrix} \). The correct direction is chosen by evaluating the speed vectors location in regard to the Newton direction. If the speed vector is on the left side in regard to the Newton Direction we need the perpendicular direction \( s_\perp = \begin{bmatrix} \hat{s}_y \\ -\hat{s}_x \end{bmatrix} \). If it is on the right side the direction \( s_\perp = \begin{bmatrix} -\hat{s}_y \\ \hat{s}_x \end{bmatrix} \) is used.

3.3 - The new Algorithm

With the constraints described in chapter 3.1 we get a change in the basic algorithm. In the new algorithm a different speed and accelerations are used which determine the location of the new point \( \tilde{p}_{k+1} \). The algorithm is as follows:

1. Set starting values, startpoint \( \tilde{p}_0 \), target point \( \tilde{p}_T \) and object locations. Set \( k = k_0 \). Set tolerance ratio \( \delta \). It is also important that the starting point \( \tilde{p}_0 \) does not coincide with any object.

2. For every \( k = k+1 \):
   
   - Evaluate the Hessian \( H_T(\tilde{p}_k) \) and gradient \( \nabla T(\tilde{p}_k) \). Check if \( H_T(\tilde{p}_k) \) is positive definite. If not, the Hessian will be adjusted so that it is positive definite (see also Annex A).
   - Determine the Newton Direction: \( \tilde{s}(\tilde{p}_k) = -H_T^{-1}(\tilde{p}_k) \cdot \nabla T(\tilde{p}_k) \).
   - Calculate the new point when moving with maximum acceleration. Two possible scenarios.
     
     i. There are two solutions. The newton direction line intersects with the circle that represents all points that can be reached. We chose the solution best suited by looking at the orientation of the Newton Direction.
     
     ii. There are no solutions. The Newton Direction line and the circle do not intersect. We then have an overshoot of the robot.
   
   - Check if we are still in compliance with our speed constraint \( \tilde{v}_{k+1} \leq 4m/s \). The acceleration \( \tilde{a}_k \) and new speed \( \tilde{v}_{k+1} \) is calculated. If \( \tilde{v}_{k+1} \) is larger than the maximum allowed speed, a new maximum acceleration is determined with
\[ a_{\text{new}} = \frac{(v_{\text{max}} - |\vec{v}_k|)}{t} \] and this is used to re-calculate the new point. Again there are two possible scenarios. We can have solutions or we do not have any solutions. When we do not have any solutions we have overshoot.

- **Overshoot.** Find the vector from the centre of the circle to the edge of the circle closest to the Newton Direction line. Add the distance of the robot it would have travelled and the vector found from the centre to the edge and add both to the current point \( \vec{p}_k \). See also figure 6.

- **Find new acceleration vector and new speed vector.**

3. If \( \|\nabla T\| < \delta \), if not set \( k = k+1 \) and repeat the loop (step 2). Otherwise, terminate.
4  -  Experiments with the new algorithm

With the algorithm we just presented we can do some experiments. In these experiments we will test the algorithm and investigates whether it does what we intended it to do. The first experiments are experiments where the obstacles will be standing still. After that the experiments will have moving obstacles. After every simulated experiment there will be a short conclusion to explain some of the results. The overall conclusion can be found in chapter 5.

4.1  -  One fixed obstacle

We will start the experiments with one obstacle. The penalty factor will be variable and the following starting values will be used:

Starting point; \( \tilde{p}_o = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \)

Target location; \( \tilde{p}_t = \begin{pmatrix} 4 \\ 4 \end{pmatrix} \)

Obstacle position; \( \tilde{p}_{b_1} = \begin{pmatrix} 1.5 \\ 1.45 \end{pmatrix} \)

Radius; \( R = 0.5 \text{ m} \)

Starting penalty factor \( r = 0.1 \)

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Path of robot" /></td>
<td><img src="image2.png" alt="Speed and acceleration" /></td>
<td><img src="image3.png" alt="Penalty factor" /></td>
</tr>
</tbody>
</table>
Starting penalty factor $r = 0.44$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Starting penalty factor $r = 0.5$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4.png" alt="Diagram" /></td>
<td><img src="image5.png" alt="Diagram" /></td>
<td><img src="image6.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Starting penalty factor $r = 1$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image7.png" alt="Diagram" /></td>
<td><img src="image8.png" alt="Diagram" /></td>
<td><img src="image9.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Conclusion: In the simulation where the penalty factor equals 1 we see that our path is smooth and does not collide with the obstacle. We can also see that the path follows the Newton Direction as best as possible. When we look at the speed and acceleration we can see that the constraints are
met at all time. When the robot accelerates and a speed of 4 m/s is achieved the acceleration drops. The small acceleration that is left is used to adjust the course of the robot.

We started the simulations with a small penalty factor however. As can be seen in the simulations, the penalty factor is a big influence on the path. The penalty factor will cause the robot to start an avoiding manoeuvre sooner when it is large, because the object has a higher priority in the total function. Because of this the path is closer to the obstacle in the simulation with \( r = 0.1 \) and further away in the simulation with \( r = 1 \).

Another observation we can make is the “detour” the robot takes around the obstacle. This is best witnessed in the simulation with \( r = 0.1 \). It can be see that the penalty factor has an influence on this effect as well. The reason for this “detour” is the fact that the path is very close to the obstacle. Close to the obstacle the Newton Direction points away from the obstacle almost perpendicular. This causes the robot to have a large overshoot and therefore travels to far along the side of the obstacle. At a certain point, the Newton Direction flips because the robot has travelled too far. Then the robot has to slow down much to be able to turn around. The speed in the simulation with \( r = 0.1 \) drops almost to zero before it accelerates to maximum speed.

4.2 - Two fixed obstacles

Now we will look at the path when there are two static obstacles. The penalty factor will be variable and the following starting values will be used:

Starting point; \( \vec{p}_a = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \)

Target location; \( \vec{p}_t = \begin{pmatrix} 4 \\ 4 \end{pmatrix} \)

Obstacle position; \( \vec{p}_{b_1} = \begin{pmatrix} 1.5 \\ 2.7 \end{pmatrix} \) \( \vec{p}_{b_2} = \begin{pmatrix} 1.5 \\ 1 \end{pmatrix} \)

Radius; \( R = 0.5 \, m \)
Starting penalty factor $r = \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Path of robot" /></td>
<td><img src="image2" alt="Speed and acceleration" /></td>
<td><img src="image3" alt="Penalty factor" /></td>
</tr>
</tbody>
</table>

Starting penalty factor $r = \begin{pmatrix} 1 \\ 0.1 \end{pmatrix}$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4" alt="Path of robot" /></td>
<td><img src="image5" alt="Speed and acceleration" /></td>
<td><img src="image6" alt="Penalty factor" /></td>
</tr>
</tbody>
</table>

Starting penalty factor $r = \begin{pmatrix} 0.1 \\ 1 \end{pmatrix}$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image7" alt="Path of robot" /></td>
<td><img src="image8" alt="Speed and acceleration" /></td>
<td><img src="image9" alt="Penalty factor" /></td>
</tr>
</tbody>
</table>
Starting penalty factor $r = \frac{1}{1}$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Path of robot" /></td>
<td><img src="image2" alt="Speed and acceleration" /></td>
<td><img src="image3" alt="Penalty factor" /></td>
</tr>
</tbody>
</table>

Conclusion: In these simulations we have also used variable penalty factors. In the first simulation we see a nice path. The normal penalty factor is enough to generate a smooth path to the target point. In the second simulation there is still a good path, although it is not as smooth as in the first simulation.

In the third and fourth simulation the lower obstacle has a high penalty factor, making it more important than the top obstacle. This causes the path to start its avoiding manoeuvre very soon and also sends the robot on a collision path with the top obstacle. This case demonstrates that the wrong method to determine the best point in an overshoot situation is used. Because the robot is on a direct collision course it should have braked hard instead of braking slightly and adjust the course. This can be compared to a human running towards a wall. The person would never think of altering its course to avoid the wall but rather brake as much as possible. When running almost parallel to the wall the used method of adjusting course is good however.

Looking at the method of overshoot, we can see that there is still a problem. When the speed of the robot is close to its maximum, the maximum allowable acceleration is determined by looking at this speed. The new acceleration is the acceleration vector that has the same direction as the speed vector. This new vector is used for all direction however, and this is not optimal. In short, this would mean that the maximum acceleration is also used as a maximum deceleration although this should still be $5 \text{ m/s}^2$. Solving this problem could also give the robot more freedom when trying to avoid an obstacle at high speeds.
4.3 Three fixed obstacles

In the final experiment with static obstacles we will plot a path in a field with three obstacles. Because we have three obstacles a so called saddle point can occur. This saddle point can cause an extra minimum to exist and the robot could get ‘stuck’ there. This was also the problem in the previous research that was done.

The penalty factor will be variable and the following starting values will be used:

Starting point; $\tilde{p}_o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$

Target location; $\tilde{p}_t = \begin{pmatrix} 4 \\ 4 \end{pmatrix}$

Obstacle position; $\tilde{p}_{b_1} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$, $\tilde{p}_{b_2} = \begin{pmatrix} 2.5 \\ 1 \end{pmatrix}$, $\tilde{p}_{b_3} = \begin{pmatrix} 2.5 \\ 2.6 \end{pmatrix}$

Radius; $R = 0.5\ m$

Starting penalty factor $r = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix}$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Path of robot" /></td>
<td><img src="image2.png" alt="Speed and acceleration" /></td>
<td><img src="image3.png" alt="Penalty factor" /></td>
</tr>
</tbody>
</table>
Starting penalty factor $r = \begin{pmatrix} 0.7 \\ 0.7 \\ 0.7 \end{pmatrix}$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
</table>

Conclusion: The simulations show a path between the obstacles. The first simulation shows a good path that avoids the saddle point. Because we used constraints for the new point the problem that occurred in a saddle point is solved. At the saddle point, there are two possible direction to take. Because of the constraints only one option remains open for the path to take.

In the second simulation we see the same effect as in other experiments, “detour”.

### 4.4 - One moving obstacle

Now we come to the experiments with moving obstacles. First, we will look at a scenario with one moving obstacle. In all experiments, the black circle in the images will represent the starting points for the obstacles. The penalty factor will be variable and the following starting values will be used:

**Starting point: $\tilde{p}_o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$**

**Target location: $\tilde{p}_t = \begin{pmatrix} 4 \\ 4 \end{pmatrix}$**

**Obstacle position: $\tilde{p}_{b_1} = \begin{pmatrix} 2 \\ 2.5 \end{pmatrix}$**

**Obstacle speed: $\tilde{v}_{b_1} = \begin{pmatrix} -0.01 \\ -0.015 \end{pmatrix}$, obstacle moving toward mobile robot.**

**Radius: $R = 0.5 \text{ m}$**
Starting penalty factor $r = 0.1$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Path of robot" /></td>
<td><img src="image2" alt="Speed and acceleration of robot" /></td>
<td><img src="image3" alt="Penalty factor" /></td>
</tr>
</tbody>
</table>

Starting penalty factor $r = 0.75$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4" alt="Path of robot" /></td>
<td><img src="image5" alt="Speed and acceleration of robot" /></td>
<td><img src="image6" alt="Penalty factor" /></td>
</tr>
</tbody>
</table>

Conclusion: Both simulations look the same. Both show a good path towards the target point. Both simulations have advantages and disadvantages. Let us look at a certain moment during the simulation where the robot passes the obstacle. In figure 8a, taken during the simulation with $r = 0.1$, we can see that the proximity of the robot and the obstacle is very small. They almost collide. In figure 8b we see the same moment for the simulation with $r = 0.75$ and here the distance between the robot and obstacle is larger, and thus more safe. When we compare the speed the robot has during the simulation we see that the top speed of the robot in the simulation with $r = 0.1$ is higher.
than the speed of the simulation with $r = 0.75$. Here, the penalty factor could be used to find a balance between speed and safety.

4.5 - Two moving obstacles

The next experiment consists of two moving obstacles. The penalty factor will be variable and the following starting values will be used:

Starting point; $\vec{p}_o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$

Target location; $\vec{p}_t = \begin{pmatrix} 4 \\ 4 \end{pmatrix}$

Obstacle position; $\vec{p}_{b_1} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$, $\vec{p}_{b_2} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$

Obstacle speed; $\vec{v}_{b_1} = \begin{pmatrix} 0.02 \\ -0.02 \end{pmatrix}$, $\vec{v}_{b_2} = \begin{pmatrix} -0.02 \\ 0.02 \end{pmatrix}$

Radius; $R = 0.5\, m$

All starting penalty’s (from 0.1 till 1.0 with steps of 0.1) generate the same effect. A collision with an object. This can also be seen in figure 9. An explanation will follow in the conclusion of this experiment.

First another scenario with two objects is evaluated. The penalty factor will be variable and the following starting values will be used. Notice that the only change is the location of the second object:

Starting point; $\vec{p}_o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$

Target location; $\vec{p}_t = \begin{pmatrix} 4 \\ 4 \end{pmatrix}$

Obstacle position; $\vec{p}_{b_1} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$, $\vec{p}_{b_2} = \begin{pmatrix} 4 \\ 1 \end{pmatrix}$

Obstacle speed; $\vec{v}_{b_1} = \begin{pmatrix} 0.02 \\ -0.02 \end{pmatrix}$, $\vec{v}_{b_2} = \begin{pmatrix} -0.02 \\ 0.02 \end{pmatrix}$

Radius; $R = 0.5\, m$

Figure 9: Collision with an obstacle
Starting penalty factor \( r = \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix} \)

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Path diagram" /></td>
<td><img src="image" alt="Speed and acceleration graph" /></td>
<td><img src="image" alt="Penalty factor graph" /></td>
</tr>
</tbody>
</table>

Conclusion: The first experiment generated a collision. Our method of path generation was not good. Because the objects get very close together the robot gets into a position where it cannot get out. At an early stage the robot should have decelerated to pass the obstacle on the other side. Unfortunately our algorithm is meant to generate the fastest path possible and not the safest path.

In the second simulation the obstacles are further apart and a smooth and almost perfect path between the objects can be seen. Both obstacles are avoided and the overall speed is very high.

### 4.6 - Three moving obstacles

The final experiment will consist of three moving obstacles. When using three obstacles, saddle points are created which can be cause for an extra minimum in which the robot can get ‘stuck’. The penalty factor will be variable and the following starting values will be used:

Starting point; \( \vec{p}_o = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \)

Target location; \( \vec{p}_t = \begin{pmatrix} 4 \\ 4 \end{pmatrix} \)

Obstacle positions; \( \vec{p}_{b_1} = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \quad \vec{p}_{b_2} = \begin{pmatrix} 3.5 \\ 1 \end{pmatrix} \quad \vec{p}_{b_3} = \begin{pmatrix} 4 \\ 3.1 \end{pmatrix} \)

Obstacle speeds; \( \vec{v}_{b_1} = \begin{pmatrix} 0.02 \\ -0.02 \end{pmatrix} \quad \vec{v}_{b_2} = \begin{pmatrix} -0.02 \\ 0.02 \end{pmatrix} \quad \vec{v}_{b_3} = \begin{pmatrix} -0.005 \\ 0 \end{pmatrix} \)

Radius; \( R = 0.5 \text{ m} \)
Starting penalty factor $r = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix}$

<table>
<thead>
<tr>
<th>Path of robot</th>
<th>Speed and acceleration of robot</th>
<th>Penalty factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Path of robot" /></td>
<td><img src="image2" alt="Speed and acceleration of robot" /></td>
<td><img src="image3" alt="Penalty factor" /></td>
</tr>
</tbody>
</table>

Conclusion: This final simulation shows a nice path toward the target function. It can be seen that there is a small “detour”. This occurs because the path is directed around obstacle two instead of making a more smooth and direct path to the target point. Because of this the third obstacle plays a bigger role than it should. However, there are no collisions and the target point is reached.
5 - Overall Conclusion and Recommendations

In this chapter we will look at the algorithm and its flaws and strong points. We will also give recommendations on how to improve this algorithm.

5.1 - Conclusions on the Obstacle Avoiding Algorithm

After the experiments we have seen some strong and weak points of the algorithm. The path generation is very stable. In saddle points the robot will no longer get stuck, because a direction is chosen and the path is committed to this direction due to constraints. Also, the robot moves in a more smooth way and does not show any saw-tooth pattern.

We designed the path so it would generate the fastest path. This is an advantage, but also a disadvantage. The speed is important because the algorithm is meant for soccer robots, but it also makes the chance of a collision higher. A high speed can cause the robot to be unable to avoid an object, due to its constraints in which it should move. As explained in the experiments, this can be solved by making the robot brake harder when it approaches an obstacle at a certain angle. This will give it a better chance of avoiding an obstacle.

The method we use to determine the best point when the robot gets into an overshoot is not optimal at high speed. The method of determining the maximum allowable acceleration should be altered.

5.2 - Conclusions on the use of the penalty function

The penalty function for the robots adds a value to the target function. The penalty factor influences the path greatly, as can be seen in the experiments. The importance of the obstacles can sometimes become larger at a later stage, when the robot already has a high speed. This causes the path to turn sharply around the obstacles. This is not always in time, and because we are bound to speed and acceleration constraints can cause collisions.

This sharp turning of the path can sometimes occur when the robot is next to, or almost past, an obstacle. Therefore we see the “detour” effect.
5.3  -  Recommendations

The Obstacle Avoiding Algorithm can be improved by providing it with a means to determine the angle in which it approaches an obstacle. This can then be used to adjust the magnitude of braking and therefore make the chance of a collision smaller. Also, this could be an advantage when approaching objects that are moving. A small deceleration could cause the path to take a better route around the obstacle.

The usage of penalty factors should be more active. Different kind of situation call for a different method of penalty factor determination. As can be seen in the experiments, the choice of the penalty factor can cause or prevent collisions.

When looking at moving objects, the speed and direction could also be used to adjust the path. This can prevent unnecessary “detours” around obstacles or moving to close along the obstacle.
Annex A - Making the Hessian positive definite

Sometimes the function can be on a saddle figure. When this happens there are two possible direction. To get the right direction it is important that the Hessian $H_T(\vec{p}_k)$ is positive definite. A popular solution to this problem has been described by Papalambros and Wilde, 2000. To make the Hessian positive defined the determinant is calculated. When it is smaller than zero the Hessian is negative defined and it should be adjusted. The diagonals are changed in such a way that we get an positive definite matrix. We do this by introducing a factor $\mu$.

$$H_T = (H_T + \mu \cdot I) = \begin{pmatrix} a + \mu & b \\ c & d + \mu \end{pmatrix}$$

(A1)

$$\det(H_T + \mu \cdot I) = (a + \mu) \cdot (d + \mu) - b \cdot c = \mu^2 + \mu(a + d) + (a \cdot d - b \cdot c)$$

(A2)

When we equate equation (A2) to zero we will get two possible solutions $\mu_1$ or $\mu_2$. We are looking for the positive definite matrix, therefore we chose the positive solution and multiply this $\mu$ with a number larger than one. This $\mu$ is then added to the diagonals of the Hessian $H_T$. When we do this we know that the Hessian $H_T(\vec{p}_k)$ is positive definite.
Annex B - Matlab code

In this annex the used matlab code will be presented. As an overview the code-tree will be presented.

- oaa.m
  - gradt.m
    - gradf.m
    - gradp.m
  - b.m
- nmvel.m
  - get_newton_direction.m
  - invhesst.m
    - hessf.m
    - hessp.m
      - b.m
  - gradt.m
    - gradf.m
    - gradp.m
      - b.m
  - get_point.m
    - get_point_overshoot.m
      - movob.m
      - varr.m
- f.m
B.1 - f.m

```matlab
function f = f(x,x_t)
    f = (x_t(1) - x(1))^2 + (x_t(2) - x(2))^2; % f gives an parabolic surface with a
                                               % minimum at x_t(1),x_t(2)
```

B.2 - oaa.m

```matlab
% Declaring globals % An array consists of 'k' terms
global X; %Array to store all coordinates
global x_b; %Array to store all object coordinates
global w; %Array to store the speed of all objects
global r; %Array to store all penalties of all objects
%Array to store all acceleration vectors
global acceleration;
global abs_const; %Array to store all absolut speed and accelerations
global new_dir; %Array to store all Newton Direction vectors

% Define playfield (for plotting)
width = 5; % Width of the playfield in the x-direction
height = 5; % Height of the playfield in the y-direction
x1 = 0:0.05:width; % Size of the playfield in the x-direction
x2 = 0:0.05:height; % Size of the playfield in the y-direction

% Initial settings
x = [0 0]; % x- and y-coordinates of starting point
k = 1; % Initialize loop number
```
\( t = 1/32; \) % Time step (delta t)
\( X(k,:) = x; \) % Save iteration points in array
\( \text{target\_error} = 0.3; \) % The tolerance of the stop criterion for algorithm

%%% Target function
\( x_t = [4 4]; \) % x- and y-coordinates of target

%%% Plotting target function
\texttt{surfplot\_f(x1,x2,x_t)} % Toggle plotting on/off

%%% Boundary function
\( x_b = [0 2 3.5 1 4 3.1]; \) % x and y coordinates of the different boundaries
\( R = 0.5; \) % Radius of boundary/boundaries
\( w = 1*[0.02 -0.02 -0.02 0.02 -0.005 0]; \) % Speed obstacles [vx1 vy1 vx2 vy2 ... ... vxn vyn

%%% Plotting boundary function
\texttt{surfplot\_b(x1,x2,R,k)} % Toggle plotting on/off

%%% Penalty function
\( r = [0.1 0.1 0.1]; \) % The starting penaltyfactor for each boundary

%%% Plotting penalty function
\texttt{surfplot\_p(x1,x2,R,k)} % Toggle plotting on/off

%%% starting speed and acceleration vectors
\( \text{speed}(k,:) = [0 0]; \) % Starting Speed of the robot in x-direction and
\( \text{acceleration}(k,:) = [0 0]; \) % Acceleration of the robot in x-direction and y-direction

%%% Calculate new iteration point
\texttt{while norm(abs( gradt(x_t,R,k) )) > target\_error; \% Stop criterium}

\( X(k+1,:) = \text{nvel}(x_t,R,t,k); \) % Compute next iteration point
\texttt{contourplot(x1,x2,x_t,R,k)} % Toggle plotting the contourplot for each time step on/off
\( x_b(k+1,:) = \text{movob}(k+1); \) % Change x_b according to the speed of the obstacles
\( r(k+1,:) = \text{varr}(k+1); \) % Calculate new penaltyfactors for the objects
\( k = k + 1; \) % Increase the loop number
\texttt{end}

\( \text{new\_dir}(k,:) = [0 0]; \) % An extra argument for plotting purposes
\( k = k - 1; \) % undoing the last iteration update for plotting purposes

%%% Plotting total function
\texttt{contourplot(x1,x2,x_t,R,k) \% undoing the last iteration update for plotting purposes}
\texttt{hold on}
\texttt{quiver(X(:,1),X(:,2),new\_dir(:,1),new\_dir(:,2),'r'); \% undoing the last iteration update for plotting purposes}
\texttt{hold off}

%%% Plotting the variable penaltyfactor
\texttt{plot\_varpenaltyfactor(k)} % Toggle on/off plotting

%%% Plotting the speed and acceleration
\texttt{plot\_speed\_acc(k)} % Toggle on/off plotting
function [nmvel] = nmvel(x_t,R,t,k)

% declaring globals
global X;
global speed;
global acceleration;
global abs_const;
global new_dir;

% declaring used constants
a_max = 5;
v_max = 4;
delta_x = 0;
delta_y = 0;

s = get_newton_direction(x_t,R,k);

norm_s = s/norm(s);
new_dir(k,:) = [norm_s(1) norm_s(2)];

sol = get_point(s,a_max,t,k);

if isreal(sol(1)) == 1
    delta_x = sol(1);
    delta_y = sol(1);
end

acceleration(k,1) = (2*(delta_x-X(k,1)) - (speed(k,1)*t)) / (t^2);
acceleration(k,2) = (2*(delta_y-X(k,2)) - (speed(k,2)*t)) / (t^2);

speed(k+1,1) = speed(k,1) + acceleration(k,1)*t;
speed(k+1,2) = speed(k,2) + acceleration(k,2)*t;

if norm(speed(k+1,:)) > v_max
    r_max = (v_max - norm(speed(k,:))) / t;
end

if norm(speed(k+1,:)) > v_max
    sol = get_point(s,r_max,t,k);
else
    delta = get_point_overshoot(s,v_max,t,k);
end

speed(k+1,1) = speed(k,1) + acceleration(k,1)*t;
speed(k+1,2) = speed(k,2) + acceleration(k,2)*t;

if isreal(sol(1)) == 1
    delta_x = sol(1);
    delta_y = sol(1);
else
    delta = get_point_overshoot(s,v_max,t,k);
end

If there was no intersection point, we need to use the overshoot function.
delta_x = delta(1);
delta_y = delta(2); 
end
end

else
    delta = get_point_overshoot(s,v_max,t,k);
    % If there was no intersection point, we need to use the overshoot function
    delta_x = delta(1);
delta_y = delta(2); 
end

acceleration(k,1) = (2*( (delta_x-X(k,1)) - (speed( k,1)*t) )) / (t^2);
acceleration(k,2) = (2*( (delta_y-X(k,2)) - (speed( k,2)*t) )) / (t^2);
speed(k+1,1) = speed(k,1) + acceleration(k,1)*t;
speed(k+1,2) = speed(k,2) + acceleration(k,2)*t;

abs_const(k,:) = [norm(speed(k,:)) norm(acceleration(k,:))];
% Determining the absolute speed and acceleration for plotting purposes in oaa.m

nmvel = [delta_x delta_y]';                             % Return x as an 1 x 2 matrix

B.4 - get_newton_direction.m

get_newton_direction.m

%% Obstacle Avoidance Algorithm %
%% Using the Newton direction %
%% J. van den Bulk %
%% J. van Heur %
%% G.N.J.P Vermeulen %
%% %

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Function to to get the Newton Direction %
%% This function has three arguments: %
%% x_t: x_t is contains the target location %
%% R: R is the radius of the boundaries %
%% k: k is the timestep %
%% %

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function get_newton_direction = get_newton_direction(x_t,R,k)

inv_hess = invhesst(R,k);
% Calculating the inverse total Hessian of target function and boundary function
gradient = gradt(x_t,R,k);
% Calculating the gradient of the total function

get_newton_direction = -inv_hess * gradient';                 % this is the Newton direction
B.5 - invhesst.m

invhesst.m

% Obstacle Avoidance Algorithm
% Using the Newton direction
% J. van den Bulk
% J. van Heur
% G.N.J.P Vermeulen

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Hessian total function
% This function has two arguments:
% R is the radius of the boundary,
% k is the timestep parameter

function [invhesst] = invhesst(R,k)

hesst = hessf + hessp(R,k);
% hesst is an 2 x 2 matrix which represents the Hessian of the total function f + p

a = hesst(1,1);
b = hesst(1,2);
c = hesst(2,1);
d = hesst(2,2);

if det(hesst) < 0
    mu = ( - (a + d) + sqrt( (a+d)^2 - 4 * (a*d-b*c) ) ) /2;
    hesst = [a+1.01*mu b; c d+1.01*mu];
end

a = hesst(1,1);
b = hesst(1,2);
c = hesst(2,1);
d = hesst(2,2);

invhesst = 1 / (a*d - b*c) * [ d -b ; -c a ];       %determine the inverse of the Hessian

B.6 - hessf.m

hessf.m

% Obstacle Avoidance Algorithm
% Using the Newton direction
% J. van den Bulk

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Hessian target function
% Since the target function is a second order function,
% the Hessian results in a constant matrix.

function [hessf] = hessf

% Since the target function is a second order function,
% the Hessian results in a constant matrix.

function [hessf] = hessf
hessf = 2 * [1 0; 0 1]; % hessf is an 2 x 2 matrix which represents the Hessian of the target function f

B.7 - hessp.m

hessp.m

%%%% Obstacle Avoidance Algorithm %
%%%% Using the Newton direction %
%%%% %
%%%% J. van den Bulk %
%%%% J. van Heur %
%%%% G.N.J.P Vermeulen %
%%%% %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%% Hessian penalty function %
%%%% %
%%%% This function has two %
%%%% arguments: %
%%%% R is the radius of the %
%%%% boundary, %
%%%% %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ hessp ] = hessp(R,k)
   global x_b;
global X;
global r;
   [ n m ] = size(x_b); %#ok<ASGLU> %determine the size of the matrix x_b
   hess = [ 0 0 ; 0 0 ];
   for l = 1:(m/2)
      A = -4 * ( 1 / b(x_b(k,2*l-1:2*l),R,k) )^3 * ( x_b(k,2*l-1) - X(k,1) )^2 - ( 1 / b(x_b(k,2*l-1:2*l),R,k) )^2;
      B = -4 * ( 1 / b(x_b(k,2*l-1:2*l),R,k) )^3 * ( x_b(k,2*l-1) - X(k,1) ) * ( x_b(k,2*l) - X(k,2) );
      D = -4 * ( 1 / b(x_b(k,2*l-1:2*l),R,k) )^3 * ( x_b(k,2*l) - X(k,2) )^2 - ( 1 / b(x_b(k,2*l-1:2*l),R,k) )^2;
      hess = hess + r(k,l) * [A B ; B D ]; %determine the Hessian for each boundary
   end
   hessp = 2*hess; % hessp is an 2 x 2 matrix which represents the Hessian of the penalty function p
### B.8 - gradt.m

```matlab
function [ gradt ] = gradt(x_t,R,k)
    gradt = gradf(x_t,k) + gradp(R,k);
```

### B.9 - gradf.m

```matlab
function [ gradf ] = gradf(x_t,k)
    gradf = -2 * [ x_t(1) - X(k,1) x_t(2) - X(k,2) ];
```

### gradp.m

```matlab
function [ gradp ] = gradp(R,k)
    global X;
    global x_b;
    global r;

    [ n m ] = size(x_b);        %determine the size of the matrix x_b
    grad = [ 0 0 ];
    for l = 1:(m/2)
        grad = grad + r(k,l) * 1 /b(x_b(k,2*l-1:2*l),R, k)^2 * [ x_b(k,2*l-1) - X(k,1) x_b(k,2*l) - X(k,2) ]; %calculate de gradient for each boundary
    end
    gradp = 2*grad;  % gradp is an 2 x 1 matrix which represents the gradient of the penalty function p
```

### b.m

```matlab
function b = b(x_b,R,k)
    global X;

    b = - ( x_b(1) - X(k,1) )^2 - ( x_b(2) - X(k,2) )^2 + R^2;
    % b represents an circular boundary with midpoint x_b(1),x(b2) and radius R
```
%% Obstacle Avoidance Algorithm  
%% Using the Newton direction  
%% G.N.J.P. Vermeulen  

function get_point = get_point(s,r,t,k)

% decalaring globals  
global X;  
global speed;  

% declaring used variables  
n_ratio = s(2)/s(1);  
point_ratio = X(k,2) - n_ratio*X(k,1);  
x_point = X(k,1) + speed(k,1)*t;  
y_point = X(k,2) + speed(k,2)*t;  

% solving the equation  
if s(1) == 0  
    sol_x = X(k,1);  
    term1 = 1;  
    term2 = -2*y_point;  
    term3 = (speed(k,1)^2 + y_point^2 - radius^2);  
    if s(2) > 0  
        sol_y = (-term2 + sqrt(term2^2 - 4*term3*term1))/(2*term1);  
    else  
        sol_y = (-term2 - sqrt(term2^2 - 4*term3*term1))/(2*term1);  
    end  
else  
    term1 = n_ratio^2 + 1;  
    term2 = -2*x_point + 2*n_ratio*point_ratio - 2*y_point*n_ratio;  
    term3 = point_ratio^2 - 2*y_point*point_ratio + x_point^2 + y_point^2 - radius^2;  
    if s(1) > 0  
        sol_x = (-term2 + sqrt(term2^2 - 4*term3*term1))/(2*term1);  
        sol_y = s(2)/s(1) * sol_x - s(2)/s(1) * X(k,1) + X(k,2);  
    elseif s(1) < 0  
        sol_x = (-term2 - sqrt(term2^2 - 4*term3*term1))/(2*term1);  
        sol_y = s(2)/s(1) * sol_x - s(2)/s(1) * X(k,1) + X(k,2);  
    end  
end
get_point = [sol_x sol_y]; % The new point

B.13 - get_point_overshoot.m

get_point_overshoot.m

%%% Obstacle Avoidance Algorithm %
%%% Using the Newton direction %
%%% %
%%% G.N.J.P. Vermeulen %
%%% %
%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% function to get the new point %
%%% When we have an overshoot %
%%% situation %
%%% %
%%% This function has four %
%%% arguments: %
%%% s: s is the Newton Direction %
%%% v_max: this is the maximum %
%%% speed allowed %
%%% t: t is a time parameter %
%%% k: k is the timestep %
%%% parameter %
%%% %
%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function get_point_overshoot = get_point_overshoot(s,v_max,t,k)
global X;
global speed;

newton_dir = [s(1) s(2) 0]; % The newton direction as a 3d vector
velo = [speed(k,1) speed(k,2) 0]; % The speed vector as a 3d vector

if (cross(newton_dir,velo)*[1 1 1]') < 0 % Determining the orientation of the newton direction in regard to the speed vector
direction = [-s(2) s(1)];
else
direction = [s(2) -s(1)];
end

a_overshoot_max = (v_max - norm(speed(k,:))) / t; % Determining the radius of the circle in which we are allowed to find our new point
if a_overshoot_max >= 5
    a_overshoot_max = 5;
end

dist = (direction/norm(direction)) * (.5*a_overshoot_max*(t^2)); % The distance we can move away from the circles center
new_point = X(k,:) + (speed(k,:)*t) + dist; % The vector pointing toward our new point.

get_point_overshoot = new_point;
### B.14 - movob.m

```matlab
function [ movob ] = movob(k)
global x_b;
global w;

movob = w + x_b(k-1,:); %determine the new position of the boundary's
```

### B.15 - varr.m

```matlab
function varr = varr(k)
global X;
global x_b;
global r;

[n m ] = size(x_b);
for l = 1:(m/2)
    deltas1(l)= sqrt((X(k,1)-x_b(k,2*l-1))^2+(X(k,2)-x_b(k,2*l))^2);
    deltas2(l)= sqrt((X(k-1,1)-x_b(k-1,2*l-1))^2+(X(k-1,2)-x_b(k-1,2*l))^2);
    if deltas1(l) < deltas2(l)
        varr(l)=r(k-1,l)+exp(-2.0*deltas1(l)); %increase the penaltyfactor
    elseif deltas1(l) > deltas2(l)
        varr(l)=r(k-1,l)-exp(-2.0*deltas1(l)); %decrease the penaltyfactor
    end
end
```
if varr(l)<0.05
    varr(l)=0.05;
else
    varr(l)=r(k-1,l); %keep penaltyfactor constant
end
end

B.16 - p.m

%% Obstacle Avoidance Algorithm
% Using the Newton direction
% J. van den Bulk
% J. van Heur
% G.N.J.P Vermeulen
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Penalty function
%
%% This function has three arguments:
% x is an 1 x 2 matrix with coordinates,
% R is the radius of the boundary,
% k is the timestep parameter
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function p = p(x,R,k)
global x_b;
global r;
[n m] = size(x_b);
p = 0;
for l = 1:m/2
    g = b(x_b(k,2*l-1:2*l),R,k); %define the barrier function
    p = p + r(k,l)*1/(-g); %define the barrier transformation
    if ( x_b(k,2*l-1) - x(1) )^2 + ( x_b(k,2*l) - x(2) )^2 <= R^2
        p = inf; % To make sure that p is plottable, p is also defined in this domain
    end
end