Freie Universität Berlin

Bachelor thesis at the Institute for Computer Science of the Freie Universität Berlin

Dahlem Center for Machine Learning and Robotics

MATRIX-Localization for autonomous model cars

Sven Heinrichsen
student ID: 4780388
s.heinrichsen@fu-berlin.de

advisor: Prof. Dr. Daniel Goehring
first examiner: Prof. Dr. Daniel Goehring
second examiner: Prof. Dr. Dr. habil. Raúl Rojas

Berlin, October 1, 2018

Abstract

In this thesis I implemented MATRIX, a force field pattern matching algorithm for improving a robots' odometry. I first implemented this algorithm in a simulator and later adapted it for autonomous model cars. I experimented with omnidirectional cameras first, but soon adapted the algorithm to be used with a forward-facing camera instead. Experiments showed that MATRIX can be highly effective in the correct circumstances, or rather unhelpful if those conditions are not met.

Statement of originality

I declare that this thesis is the product of my own original work and has not been submitted in similar form to any university institution for assessment purposes. All external sources used, such as books, websites, articles, etc. have been indicated as such and have been cited in the references section.

October 1, 2018

Sven Heinrichsen

Contents

1 Introduction

The idea of autonomous cars, cars that can drive themselves from A to B without causing accidents, has grown very popular in the last couple of years. Volvo even promised that no fatal accidents could happen for the 2020 edition of their cars [21]. While this mentality is of course praiseworthy, it does not change the fact that autonomous vehicles base their decisions on sensor data. Sensor data by its definition can be interpreted in a faulty way. Earlier this year (2018) there was a crash involving a car made by Tesla using its autopilot. Apparently, the vehicle sped up instead of slowing as it was driving towards a wall [9]. This is a classic case of misinterpreted sensor data. The car assumed the road was clear, so it sped up when in fact it was driving towards a wall. For this reason, it is important to create robust algorithms using as many fail-safes as possible to react to erroneously interpreted sensor data.

Autonomous cars highly depend on knowing their precise location, especially in relation to any known obstacles or for finding the optimal path to a specific destination. Finding out the general position can be done by using the Global Positioning System (GPS), which provides a good idea of where the car is. Although GPS works well in most situations, its accuracy can worsen substantially near buildings, bridges and trees due to the so-called multipath-error [15, 23]. For these situations another solution is needed. One concept is to detect the current speed and direction of the car, to find out any translational and orientational differences that happened between the current time and the last time the car received high quality GPS data. This data is then used in a Kalman-Filter to get an accurate idea of where the car is [19]. This concept is still not free of errors, as the sensor data, the measurement of wheel rotation over time, or odometry[5], can not react to wheel slippage [17].

In this thesis I will try to correct the error of the odometry. I will use camera images compare them to an image of the map and calculate the error. Afterwards I will move and rotate the assumed position of the car according to that error. This concept is called MATRIX-localization [11].

Since it is not easy to test the implementation on a real car, I will use model cars, using a street imprinted on a carpet for the image data. These model cars use the Robot Operating System (ROS)[2] to access the different functionalities, like the odometry and camera images.

My implementation of the MATRIX algorithm can be found on my GitHub [3].

2 MATRIX-Algorithm

The following is an explanation of the MATRIX algorithm that was developed by Felix von Hundelshausen, Michael Schreiber, Fabian Wiesel, Achim Liers, and Raúl Rojas [11].

The MATRIX algorithm tries to improve odometry data using an image of the car's surroundings. This is done in three steps.

First, we need to supply a map of the field the robot should use. This map is then used to generate a field of vectors pointing towards the line segments of the map. For the map image I used a scale of 1cm = 1 pixel.

Second, we want to have a general idea where the vehicle is located at the start of the algorithm. For this we need a global localization.

When we have the vector field and a starting position we can combine the camera image with the odometry of the vehicle to get an improved new position when the car moves. For this, first the camera image is converted to a cloud of points, corresponding to the white lines detected in the image. The point cloud is then rotated and translated according to the car's odometry. Every point of the point cloud is matched to the vector field, the vectors are summed up and the resulting vector indicates the direction in which the car should be moved to improve its odometry. A similar approach is used for the rotation of the vehicle. First, a distance vector between the car and a point on a line in the camera image is calculated. Then the cross product of this vector and the corresponding vector of the vector field is calculated to get the torque around the car. The torques for every point on the camera image are added up and the result indicates the rotation direction the car should be rotated in to improve the odometry.

2.1 Generate a vector field

There are two concepts to generate a vector field. The first, easier and much faster way is to calculate the distance vector from a point to the nearest line segment. I detected the line segments using OpenCV's LineSegmentDetector class which is based on a paper by Rafael Grompone von Gioi et al. [22]. I then created a grid of double the size of the map image with a resolution of 10 cm. Since the (newer) map is 5.67m by 3.85m big, the map image is 567 pixels by 385 pixels big. The grid would then span 11.34m by 7.70m with the map in the center. Now we can calculate the distance vector for every cell of the grid to the nearest line segment. The result can be observed in fig. 1.

This method of creating distance vectors can result in imprecise or bad results, as neighbouring vectors can point in completely different directions if they are between two lines. See for example the center of Figure 1. The vectors are both fairly long but pointing in opposite directions.

To smooth these vectors, Rojas et al. proposed calculating a force vector for each cell of the vector field [11, p. 5]. This is accomplished using two key equations. First, the weight is calculated for each cell in the vector field:

$$w_i = \frac{e^{(-l_i/10)}}{\sum_{k=1}^{k=n} e^{(-l_k/10)}}$$

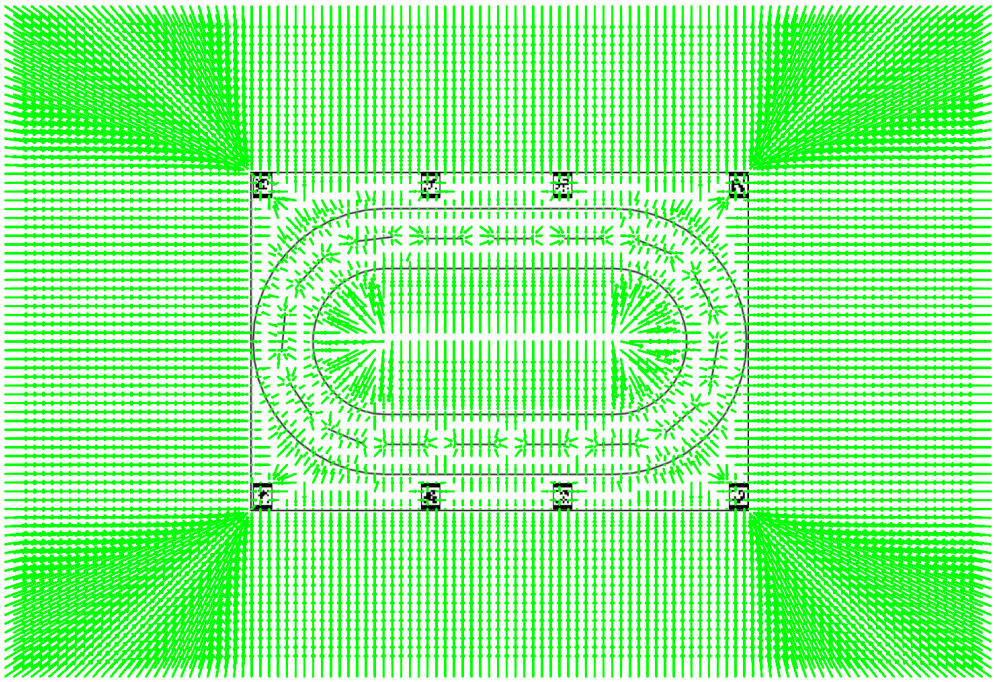


Figure 1: Matrix of distance vectors to the nearest line segment on the field.

Where l_i, $i = 1,\ldots,n$ represents the cells' distances to each of the n line segments. The constant 10 provides good results for distances measured in centimetres [11, p. 5].

The force vector f_j for the j-th cell is then:

$$f_j = \sum_{i=1}^{i=n} w_i d_i$$

Where d_i is the distance vector of the current cell to the closest point on the i-th line segment.

This means that for each cell in the vector field the weights for all line segments have to be calculated. And to calculate the weight for a single line segment the distances to every other line segment also has to be calculated. This results in multiple nested loops on top of the two loops that iterate through the columns and rows of the vector field, but since generating a vector field is only required once per map, this is not a problem.

Computing a quality match by adding all the vector lengths for a particular camera image or computing an odometry correction by adding all the vectors up only takes a couple of micro- to milliseconds, depending on how big the point cloud from the image is.

2.2 Global localization

The global localization works similar to a particle filter, where for each particle position and orientation a quality is calculated, the bad matches are replaced and some noise is added to position and orientation of each particle. This is then repeated until the

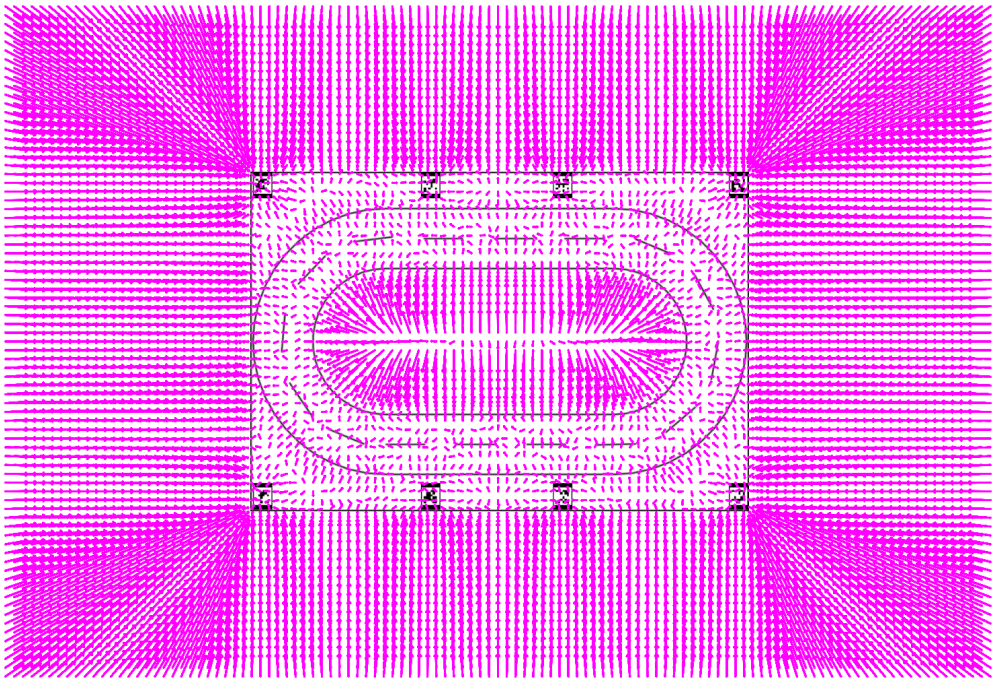


Figure 2: Matrix of force vectors to the nearest line segment on the field.

particles converge on a single point. For the MATRIX algorithm a much simplified variant of that algorithm is used.

First, we limit the global localization to the part of the map that is not symmetrical. Then, we define positions for which to test the matching quality and rotate the camera images a fixed amount of times around each position. For each position and rotation the camera images are matched to the vector field. We add up the lengths of each of the vectors and average the result. The lowest resulting quality value is the best match.

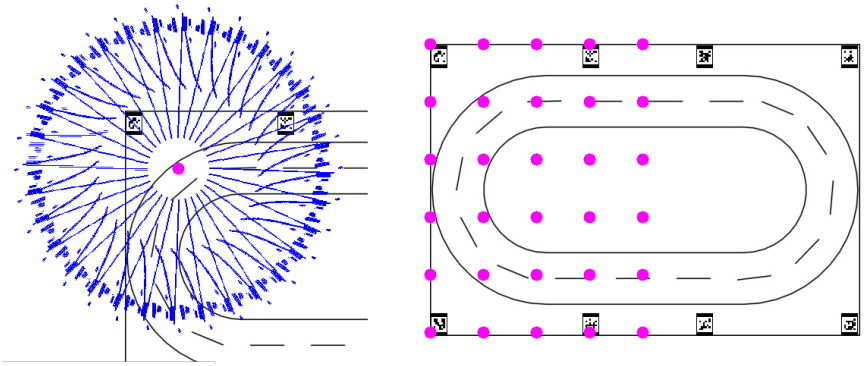


Figure 3: The left image shows 32 rotations of a point cloud from an image. The right image shows the positions these 32 rotations are tested for.

2.3 Odometry correction

We assume that the global position and rotation of the car are known. When the car starts moving, the odometry delivers the movement data in a local coordinate system. We can now take the difference of the odometry data between the last two image frames and calculate the difference of position and rotation of the car. If we add the differences to the current position and rotation we get a new assumed position of the car. Rojas et al. [11, p.7] propose a normal distribution around the new assumed position and rotation to calculate a quality match like in the global localization (2.2). This is costlier the more image points we have, and I did not experience a justifiable benefit from this when experimenting with it, so I decided not to use this method of quality optimization. We can now take the point cloud from the camera image and translate and rotate it around the new assumed position.

When matching the point cloud to the vector field we add the vectors up to get an average direction the vectors point to. For the rotation we calculate the cross product between the distance from the car to a point in the point cloud and the vector it matches to. The result is the direction of the torque around the car. Now we have an idea of the direction and rotation the car should have to better match the map.

Lastly, we scale the calculated force vector and torque by a small percentage of the last movement plus a constant, to add that scaled force vector and torque to the assumed position and rotation of the car. This way we steadily improve the accuracy of the assumed position.

3 Environment and car

To implement the MATRIX algorithm, we need three things: First, an image of the map the car should be driving on. Second, camera images depending on the location of the car, and third the corresponding odometry data containing a steering angle and a position. This data can either be simulated in a simulator or it can be acquired using a real model car and a carpet depicting a road. First, let me explain how the acquisition of this data works.

3.1 ROS

The Robot Operating System, or ROS, is a peer-to-peer middleware package for robots. This basically means that it provides hardware abstraction of a robot on the one hand and it provides a network interface to communicate with ROS in a peer-to-peer environment on the other hand. One could, for example, write a program using ROS on their own computer, connect to a robot over a network and run the program. Data from the robot would automatically be sent to the computer, be processed on that computer and the result sent to whomever would want to use it.

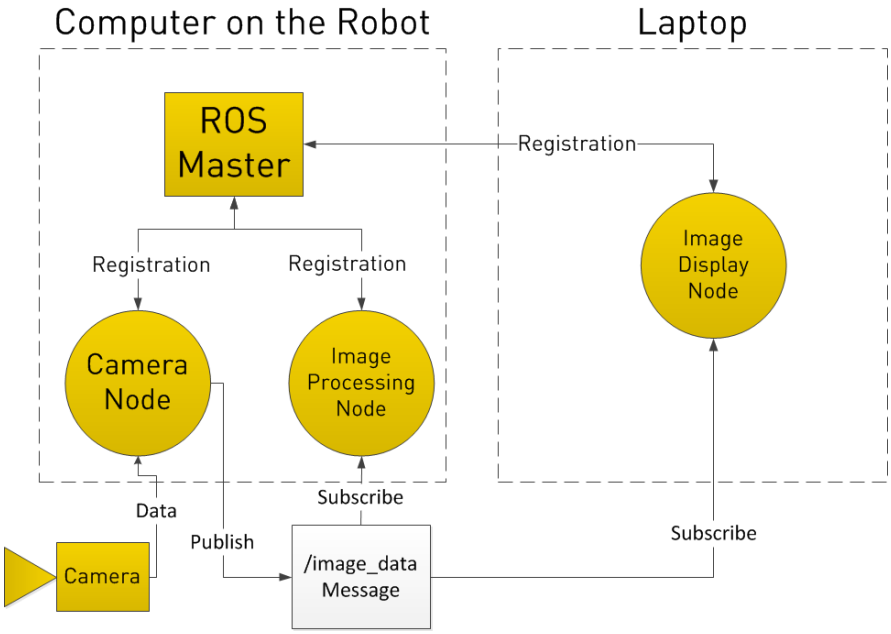


Figure 4: Depiction of how ROS-communication works. First every node tells the master node what topics it subscribes to or publishes on. Next the data is directly transmitted between the nodes, no matter where they are. [5]

Further, ROS is designed in a way as to minimize the overhead of the framework. All of its major functionalities are broken up into their own modules to optionally use and install on a robot. These modules, called nodes, use a messaging system to communicate with each other. A master node coordinates the communication between the nodes. The messages are transmitted over buses called topics. A node can subscribe to a topic, process the data, and then publish the result on a different topic. The master node, a part of the so called 'roscore', tracks publisher and subscriber

nodes and connects them via peer-to-peer directly to each other if they subscribe to and publish on the same topic. This means that, as long as a node knows where the master node is, it can be connected to any other node known to that master node. This allows for a distributed system of nodes across multiple computers.

The roscore is a collection of nodes that are required to run ROS on a system. It contains a ROS Master node, a ROS Parameter Server and a rosout logging node. The Parameter Server is a dictionary for multiple variables that is accessible via network APIs, so that nodes can access or store variables at runtime. This is e. g. useful for storing different exposure settings for different cameras along with their topic names. One node could set the exposure in the Parameter Server so that the camera node reads and applies the exposure settings. As a result, the first node can get usable camera data.

The goal of the design of ROS is to be able to reuse already existing code that might already have been integrated into a different framework or to use ROS-packages in a different framework. To this effect, the framework has multiple implementations in different languages, like Python, C++ and LISP. ROS also does not wrap the `main()`. This results in the ability to use ROS concurrently and complementary to an additional framework.

Of course, it also allows the implementation of algorithms only using the ROS framework as well, which is what I did during this project. Another important part is that ROS offers a big open-source library of modules that can be easily installed and used.

Please refer to the ROS-wiki [6] for more information.

3.2 Model car

The model car is a development of the Dahlem Center for Machine Learning and Robotics of the Freie Universität Berlin. It is called 'AutoNOMOS Mini v3' and was developed for educational purposes. It combines an Odroid board running Linux and ROS with an Arduino Nano Board that controls the motors, Inertial Measurement Unit (IMU)[7] and LED strips. The car uses a Brushless DC-Servomotor 'FAULHABER

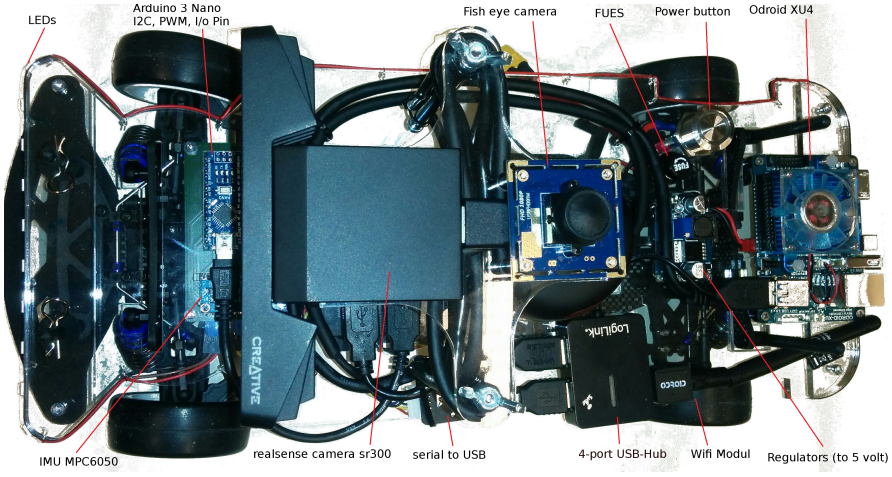


Figure 5: Top-down view of the components of the AutoNOMOS Mini v3 [2]

2232' for forward movement and another servo-motor 'Batan S1213' for steering. The IMU, which is used to complement the odometry and to measure the vehicle's rotation is contained on an MPU-6050-chip. Further on, a rotating laser scanner, the 'RPLIDAR A2 360' is installed to provide measurements of distances around the car. This can be used to detect obstacles. A Kinect-type stereoscopic system, the 'Intel RealSense SR300', also provides distances to objects in front of the car to up to 1.5m using an infrared laser projector system [14, p. 9]. It can also be used as a standard front facing camera, which is what I will do later on. Last but not least, there is a camera pointed at the ceiling, which was used to detect differently colored lightbulbs to provide an imitation of a GPS-system. By now this is deprecated, as we installed ArUco-markers[8] on the cars and the street which are detected by cameras in the ceiling. This way the position of each car can easily be assessed by a master system.

3.3 Simulator

I started off using the seat-car-simulator for the ROS-framework. The simulator was originally created by the Institut de Robòtica i Informàtica Industrial for the SEAT Autonomous Driving Cup [7]. It uses ROS, Gazebo[3] and a virtual version of the previously mentioned model car to simulate a 1:10 scale model of a car. By default, it is equipped with multiple cameras, simulated Lidar-data, odometry, and many more sensors. The simulator was useful when just starting to work on the project. But

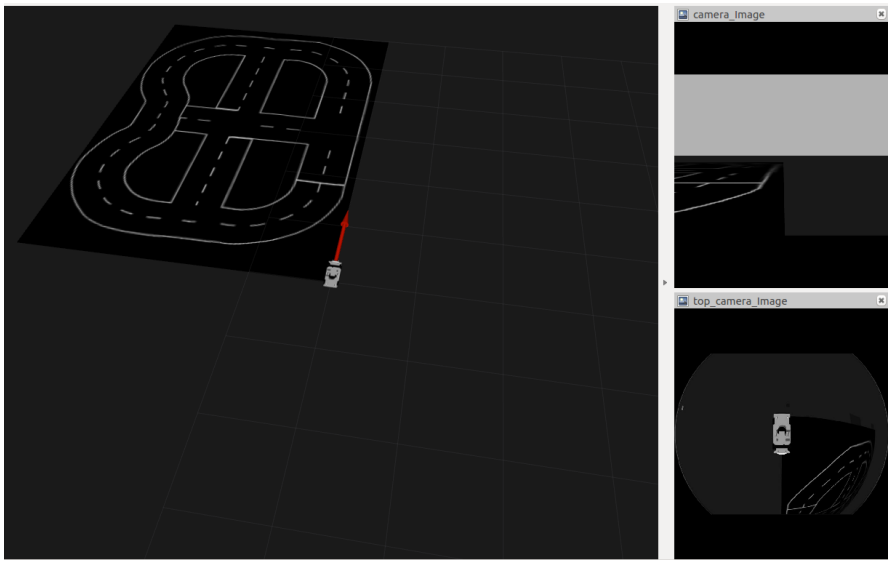


Figure 6: RViz view of the seat-car-simulation. The red arrow depicts the odometry of the car. On the right side are different camera views of the car.

since the odometry is more accurate than not and since the simulator needs a lot of processing time for all of its sensor simulations, it was soon rather unhelpful. At that point I switched to using model cars.

3.4 Map

Over the course of this project I used two different maps printed on carpets. At first, a rather complicated map, the left-hand map of Figure 7. This carpet-map was wearing off and a new course of students was supposed to work with it. So instead of the carpet becoming unusable during the course, it was replaced by the new simpler map on the right-hand side of Figure 7.

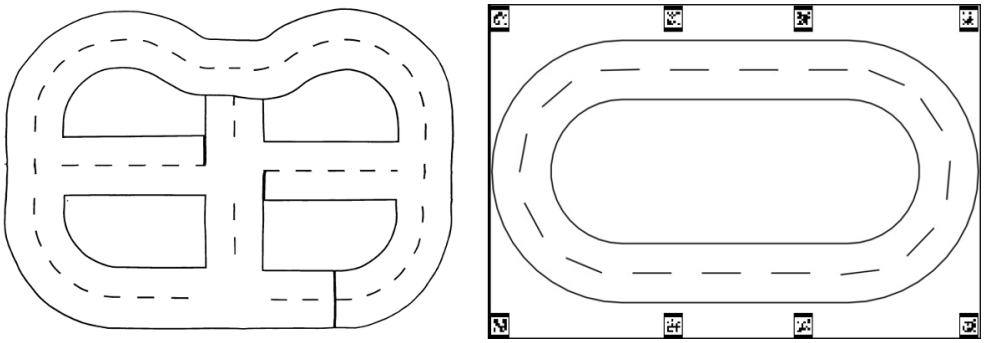


Figure 7: Left: old map, also map used in simulator. Right: new map with ArUco markers.

4 Experimentation

To implement and test the MATRIX-algorithm, I first used the Seat Car Simulator. I proceeded to implement the algorithm at home, using the simulator to test it.

4.1 Simulator

For the simulator to work with MATRIX localization, I set the top camera to float about 40 centimetres above the car, pointing towards the ground. This way I simulated an omnidirectional camera.

4.1.1 Camera calibration

First, I needed to figure out how to get a usable camera image, as the standard camera image is quite distorted and needs a perspective transform (See fig. 6). But since it is a simulator I could not use physical techniques, like a chessboard calibration, to calibrate the camera and a perspective transformation for omnidirectional cameras is difficult. What I used instead was the 'fisheye_camera_matrix' package of the AutoModelCar GitHub. This package let me manually control parameters like focal length and scale to make the image look as close to a bird's-eye view of the map as possible. The configuration data is saved to a file and later used to undistort the camera images. The result is not a real undistorted and perspective corrected image, but it is something very close to it.

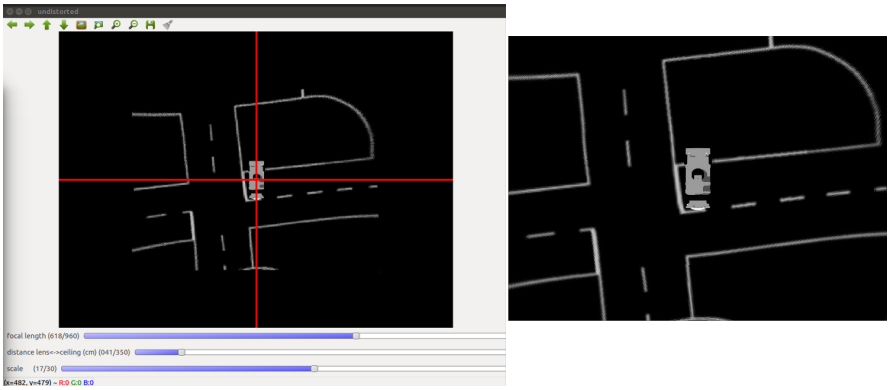


Figure 8: Left: The manual calibration that was used for the simulator. Right: the undistorted image.

Now I only had to rotate and flip the map image until its coordinate system matched the one of the odometry of the simulated car. After creating a force vector field from the map the MATRIX-algorithm can be used.

4.1.2 Simulator results

For a first test of my implementation of the MATRIX-algorithm I let the car drive in circles. This requires the least user input and it lets me concentrate more on how the algorithm does. In the simulator, the odometry-position of the car and the position of the car's top-camera do not match exactly. Thanks to this, some improvement of the odometry is visible.

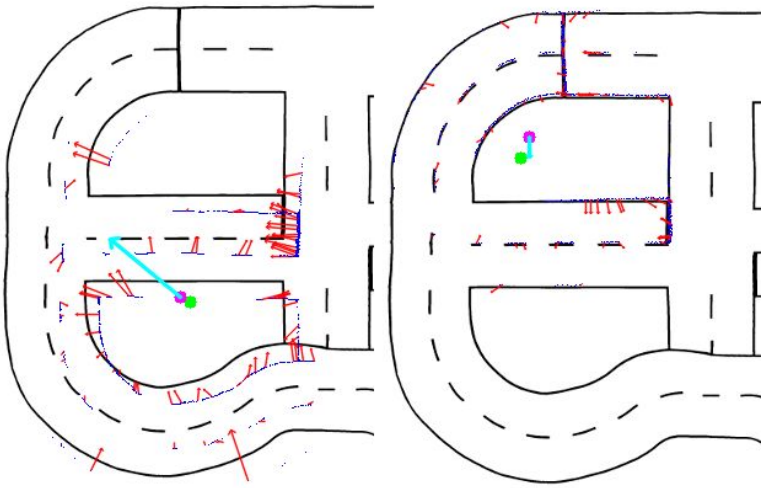


Figure 9: The left picture shows the simulation at the start, and the right picture shows it after half a circle was completed. Light blue Arrow: averaged force vector; red arrows: applicable force vectors; purple dot: MATRIX-improved position of the car; green dot: odometry position; blue dots: point cloud

For this experiment, the allowed MATRIX-improvement is 50% of the movement of the car plus maximum 1 centimetre. As is illustrated in fig. 9, The MATRIX-algorithm is capable of quickly correcting the odometry position with a high accuracy. The difference between the first and the second frame are just about 10 to 15 seconds. And even if the simulation runs longer, it does not over-correct itself.

After this first testing succeeded, I moved on to the model cars to try the MATRIX-localization in practice.

4.2 Real world

For the model car implementation of the algorithm, I first thought of using the same concept as in the simulator. An omnidirectional camera for a surround view of the vehicles surroundings.

4.2.1 Omnidirectional camera

I started off experimenting with a Sony-branded omnidirectional mirror, which uses a special lens along with a mirror to deliver high quality images. Although this method reduces the field of view greatly and introduces a high amount of distortion.

There exist many different approaches to calibrating an omnidirectional camera. They can be differentiated starting with the shape of the mirror (hyperbolic, parabolic), over what information are considered known (mirror and lens parameters) to what projection model and method are used (auto-calibration, grids). In 2004 a generic concept using no parameters was published [20], however it did not yield satisfying results for catadioptric cameras. One automatic approach by S.B. Kang [16] uses multiple images of the environment without the use of calibration patterns, knowledge of camera motion or knowledge of the scene geometry. For this approach, point features are tracked across a sequence of images based on the characteristics of catadioptric

imaging.

Methods using planar grids [24] are also simple to use and provide accurate results. In [8], the authors present a method of calibrating omnidirectional cameras using manual user input and planar grids. The user would have to select each corner of a checker board on multiple images since the initial values of the projection function are difficult to obtain. The projection function is then polynomially approximated. Mei and Rives expanded this idea in [18] by showing that by using an exact model, only four points need to be selected for each calibration grid. Mei and Rives further used the unified model of Geyer and Barreto [10, 13] to be able to calibrate many different types of omnidirectional cameras.

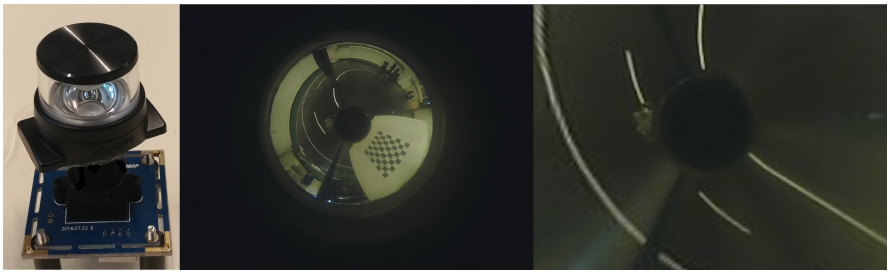


Figure 10: Left: Sony omnidirectional mirror and camera; Center: raw image during calibration; Right: undistorted, perspective transformed image

This last approach has an open source implementation in the OpenCV library. This implementation also provides an automatic perspective transformation, which is exactly what I needed. Fig. 10 shows what the result of this algorithm looks like in practice. The result looks quite good, though the image is a bit distorted on the edges.

However, the Sony mirror is expensive. A cheaper alternative is the custom made mirror shown in fig. 11. Since this conic mirror basically projects its surroundings to a plane without much distortion, the above-mentioned algorithm should still apply to this catadioptric system. As is apparent in fig. 11, the quality of the camera image is very low. As such I was not able to calibrate the camera because no checkerboard was recognized.

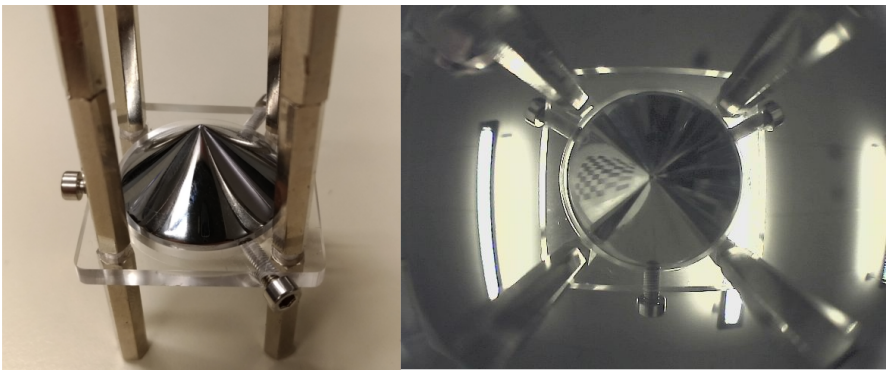


Figure 11: Left: custom made conic mirror; Right: raw image

Of course, it would have been possible to calibrate the camera separately and then apply a perspective transform manually. And this might be a worthwhile pursuit in

the future. But the MATRIX localization is not restricted to using only omnidirectional cameras. As already described in chapter 2, it is, in general, a pattern matching method. And for this reason, only an image of the road with a relation to the car is necessary. Thus, I moved on to using the forward-facing camera of the model car.

4.2.2 Forward facing camera

Using and calibrating a classical forward-facing camera is easy compared to omnidirectional cameras. For the calibration one only needs to use the planar grid-based approach described in [24], which has an implementation in ROS and OpenCV. This still yields a highly distorted view of the road in front of the car (fig. 12, picture 1). After a simple perspective transformation of the picture, it looks very similar to the corresponding sector of the road (fig. 12, picture 2).

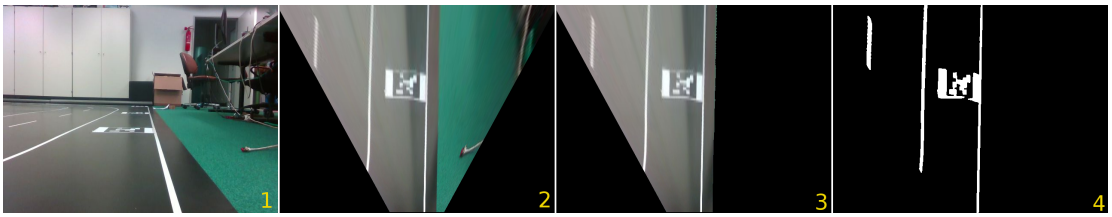


Figure 12: 1: raw image. 2: perspective transformed image. 3: colours blacked out to hide artifacts like the white cable on the right-hand side 4: final thresholded image

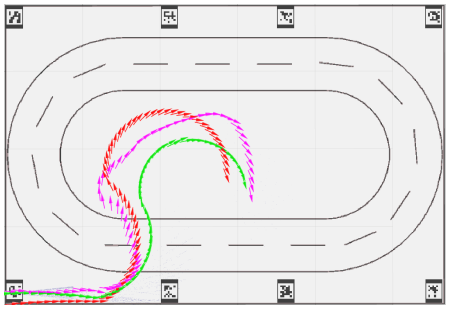
Now we need to filter out unwanted artifacts, like white objects that are anywhere other than the road-map. My approach was to span lines between the bottom center and every pixel on the left, right and top border of the image. Then we iterate from the bottom center to a pixel on a border. If a (green-)colored pixel is found, this pixel and every pixel after that on the respective line is colored black. In this way, everything on the 'wrong' side of the edge between the map-carpet and the green carpet is colored black and thus ignored (see fig. 12, picture 3). Lastly, the image is thresholded, to get a clear idea of where exactly the white markers of the road are (fig. 12, picture 4). This results in a clean black-and-white image of the road.

4.2.3 Real world results

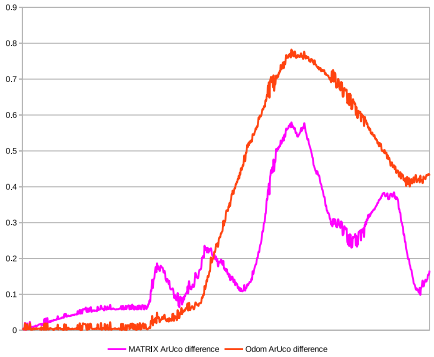
As mentioned briefly in chapter 3.2, we have set up our map with ArUco markers. These enable tracking our ArUco-marker-equipped cars through a camera system. A server receives the camera data and calculates odometry data for every car on the field. This tracking system is very accurate, its error is only about one to two centimetres. I used this ArUco-odometry as reference for the following experiments.

For the first real-world-experiment I let the car drive in three segments: first a straight line, then a left-turn and finally a right-turn. The allowed MATRIX-improvement is 50% of the movement of the car plus maximum 1 centimetre, just like in the simulator. Viewing fig. 13, it becomes apparent that the algorithm did not work as smoothly as it did in the simulator.

This happens for two reasons. First, and more importantly, the image is not entirely free of unnecessary artifacts, that influence the force vector. This can be



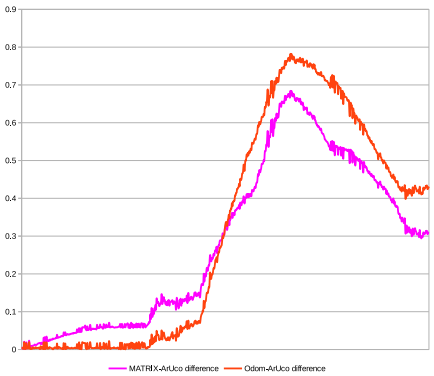
(a) the paths each odometry took.



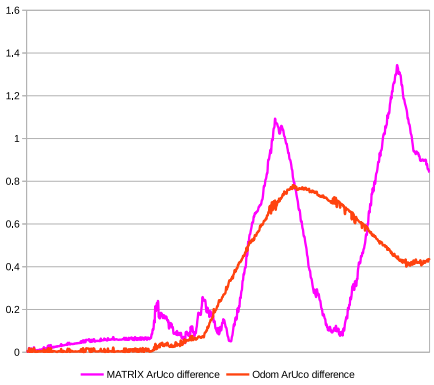
(b) distance between the respective odometry positions and the ArUco-odometry position in meters. Lower is better.

Figure 13: green: ArUco-odometry. red: car-odometry. purple: MATRIX odometry.

e. g. because of reflections on the floor that can be even brighter than the white lines of the street and thus hard to filter out. These reflections could be reduced by turning the camera towards the floor to increase the angle between camera field of view and floor. The other reason has to do with the perspective of the forward-camera. If a line on the camera image is only a little bit misaligned with the map in the beginning, this misalignment increases in the distance (towards the top of the image). A big vector that might not actually represent a movement, but only a wrong rotation of the car is created. But there is no way to differentiate if this vector represents a movement or a rotation. This effect is even increased the less line segments are visible. If a road is not perfectly aligned, one side of the road might create vectors in one direction while the other side of the road would create vectors in the other direction, cancelling each other out. If only one side of the road is visible, there would be nothing that could cancel the vectors out.



(a) MATRIX-translation-correction is 0.1 times the odometry movement. Lower is better.



(b) MATRIX-translation-correction is 1.5 times the odometry movement. Lower is better.

Figure 14: red: car-odometry. purple: MATRIX odometry.

One way to correct this behaviour would be to reduce the maximum possible MATRIX-corrected translation per frame while increasing the maximum possible rotation. But this would partly defeat the purpose of the algorithm, so it is not a viable solution.

Next, I ran the same experiment using different factors for the MATRIX-translation-correction. The results are as expected. For a low factor of 0.1, the artifacts in the image influence the correction barely. The corrected odometry stays near the car's odometry, but moves nearer to the actual position of the car. When I turned the correction up to a factor of 1.5, the correction overcompensated a lot. When there is an unwanted artifact visible, the force vector points to the wrong direction and the car moves a long distance in that direction. After some time, the artifact vanishes because the car does not stop moving and the MATRIX-Odometry is able to correct itself again. Until another artifact is visible. This is well illustrated in fig. 14.

5 Conclusion

Concluding, I implemented the MATRIX force field pattern matching algorithm for autonomous model cars successfully. I verified the correct implementation and effectiveness of the algorithm using a simulator and experimented with omnidirectional cameras for a real-world application. Instead of using an omnidirectional camera, I changed the algorithm to be used with a classical forward-facing camera. A forward-facing camera is much easier to handle on one hand, but introduces unexpected problems to the algorithm on the other hand. Most of these problems could be fixed by using a robust line- or lane-detection. I experimented using different correction factors with the result that a low correction factor is better if the image contains unwanted artifacts. A higher factor is better for perfect images, as that results in faster odometry correction.

I ended up not using the global localization described in chapter 2. For forward facing cameras, it is too error-prone. When I put the car near the (0,0)-coordinate on the map, it would sometimes not detect that location because the angle would not quite match or the position was a bit offset. As a result, the calculated quality would be worse than the quality on different positions and the global localization would output a completely wrong location. Instead I used the ArUco-localization, that was already installed and is very accurate, for the initial pose determination. For a further implementation of the MATRIX algorithm I would recommend using a more robust version of the global localization to determine the starting position than the one proposed in the paper [11].

6 Glossary

Term	Meaning
ROS	The Robot Operating System or ROS is a framework for robot applications. It contains many libraries and tools for e. g. motion planning, object recognition and 3D visualization. More: [6]
Gazebo	Gazebo is a simulator for robots that is able to accurately and efficiently simulate multiple robots in different environments. Gazebo provides high quality physics simulation, a suite of sensors and interfaces for users and programs. More: [1]
OpenCV	OpenCV, or the 'Open Source Computer Vision Libary' is an open source library for computer vision and machine learning algorithms. More: [4]
odometry	The odometry describes the position and orientation of a robot, or car in this case. Motion sensors are used to detect movements and rotations of the car which are then translated to a position and rotation in a local coordinate system.
omnidirectional camera	The term 'omnidirectional camera' describes a camera that can capture a 360-degree view of its environment. This can be achieved by 'stitching together' multiple pictures (polydioptric), by using fisheye lenses (dioptric) or by using a curved mirror in front of a camera (catadioptric).
IMU	The 'Inertial Measurement Unit' is a device that reports the specific force the device experiences. For this purpose it is equipped with accelerometers and a gyroscope that measure acceleration in any direction and orientation in 3D space with its angular velocity respectively.
ArUco-markers	ArUco markers are part of a fiducial marker system used for position estimation for e. g. robot localization or augmented reality [12]. The markers look like QR-Codes but are much simpler.

A Appendix

A.1 Issues while working on the project

In this section I will explain some of the issues I had that delayed the completion of this thesis.

A.1.1 uchar error

While iterating through the camera image to find the white pixels for which to get a force vector, a mysterious mistake happened to me. As a result, the average force vector would be calculated using four different images which would be distorted and translated from each other.

```
for (int x = 0; x < oCamImg.cols; x++)
{
  for (int y = 0; y < oCamImg.rows; y++)
  {
    if (oCamImg.at<uchar>(y, x) > 128)
    {
```

The problem here was that I used `int` instead of `uchar` for a long time, assuming that if the type does not exactly match it would be cast implicitly. As it turns out it is not cast. An `int` is normally 32 bits or 4 bytes, so 4 `uchar` long. What actually happened was that for every index the missing 3 bytes were taken from the image matrix which consists of `uchar`. This results in values greater than 128 for almost every index that contains a white pixel in at least one of its four bytes, depending on if the first bit is a '1'. This way, white pixels are detected in wrong locations and the resulting averaged force vector is wrong.

To my embarrassment this mistake took me almost to the end of the bachelor thesis deadline to detect and to fix. It was really obscure and unexpected to me.

A.1.2 Omnidirectional camera calibration

For the omnidirectional calibration one needs two arrays of arrays of points. In the official example of how to use the omnidirectional calibration library, those arrays are created using matrices:

```
std::vector<cv::Mat> imagePoints, objectPoints;
```

When I tried using these vectors of Mats it never worked. There was always a conflict of data types. Even though it is well documented that `vector<Mat>` should work. I ended up using the older method of declaring the `imagePoints` and `objectPoints` vectors:

```
std::vector<std::vector<cv::Point2f>>   imagePoints;
std::vector<std::vector<cv::Point3f>>   objectPoints;
```

This was very frustrating to debug because I did not know if I made a mistake or if the library was bugged. Turns out the library is bugged.

Bibliography

[1] Gazebo. http://gazebosim.org/.

[2] Hardware (autonomos model v3.1). https://github.com/AutoModelCar/AutoModelCarWiki/wiki/Hardware-(AutoNOMOS-Model-v3.1).

[3] Implementation of this thesis. https://github.com/Draekwon/localization.

[4] Opencv. https://opencv.org/.

[5] Ros 101: Intro to the robot operating system. https://robohub.org/ros-101-intro-to-the-robot-operating-system/ [Online; accessed September 29, 2018].

[6] Ros wiki. http://wiki.ros.org/.

[7] Simple gazebo simulator for the seat autonomous driving cup. https://gitlab.iri.upc.edu/seat_adc/seat_car_simulator.

[8] R. Siegwar A. Scaramuzza D., Martinelli. A toolbox for easy calibrating omnidirectional cameras. *IROS*, 2006.

[9] Tom Barnes. Tesla autopilot caused car to accelerate before fatal crash, investigators find. *Independent*, June 2018. https://www.independent.co.uk/news/world/americas/tesla-car-crash-autopilot-acceleration-california-driver-death-investigation-latest-html [Online; accessed September 22, 2018].

[10] J. P. Barreto and H. Araujo. Issues on the geometry of central catadioptric image formation. *CVPR*, 2001.

[11] R. Rojas et al. Matrix: A force field pattern matching method for mobile robots. http://page.mi.fu-berlin.de/rojas/2003/matrix.pdf, 2003.

[12] S. Garrido-Jurado et al. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, June 2014.

[13] C. Geyer and K. Daniilidis. A unifying theory for central panoramic systems and practical implications. *ECCV*, 2000.

[14] Intel Corporation. *Intel®RealSense™Camera SR300*, June 2016. https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/realsense-sr300-datasheet1-0.pdf [Online; accessed September 23, 2018].

[15] J.M. Juan Zornoza J. Sanz Subirana and M. Hernández-Pajares. Multipath. https://gssc.esa.int/navipedia/index.php/Multipath, 2011. [Online; accessed September 22, 2018].

[16] S. B. Kang. Catadioptric self-calibration. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 201–207, June 2000.

[17] L. Kleeman. Odometry error covariance estimation for two wheel robot vehicles. Technical Report MECSE-95-1, Monash University, 1995.

[18] Christopher Mei and Patrick Rives. Single view point omnidirectional camera calibration from planar grids. *ICRA*, 2007.

[19] G. Ulivi S. Panzieri, F. Pascucci. An outdoor navigation system using gps and inertial platform. http://www.dia.uniroma3.it/~panzieri/Articoli/AIM01.pdf. [Online; accessed September 22, 2018].

[20] P. Sturm and S. Ramalingam. A generic concept for camera calibration. *ECCV*, 2004.

[21] Peter Valdes-Dapena. Volvo promises deathproof cars by 2020. *CNN Money*, January 2016. https://money.cnn.com/2016/01/20/luxury/volvo-no-death-crash-cars-2020 [Online; accessed September 22, 2018].

[22] Rafael Grompone von Gioi et al. Lsd: a line segment detector. *Image Processing On Line*, March 2012. https://www.ipol.im/pub/art/2012/gjmr-lsd/article.pdf [Online; accessed September 22, 2018].

[23] Lawrence R. Weill. Conquering multipath: The gps accuracy battle. http://www2.unb.ca/gge/Resources/gpsworld.april97.pdf. [Online; accessed September 22, 2018].

[24] Z. Zhang. Flexible camera calibration by viewing a plane from unknown orientations. *ICCV*, 1999.