CS-358 MIT: Team Proposal

Arthur Bigot Rokhaya Fall Albert Fares
Hugo Jeannin Thomas Kemper Daniel Polka

April 2nd 2023

1 Description

The main idea of our project is to recreate a real life version of the worldwide famous video game MarioKart. We would have 3 cars and potentially 3 controllers that would be designed by us and then 3d printed. There will be two ways to play the game, the first is challenging autonomous cars controlled by AI and the second playing against real players. Numerous power ups will be available for the players to pick up on the track to make the experience more fun.

1.1 Hardware Description

1.1.1 Cars

For the cars, we're looking for a minimalist and relatively small design, which also packs power and precision. The car design will be specially designed and 3d printed to hold all the electrical components and have an aerodynamic silhouette. It will be a 4 wheel car, rear wheel drive. The aimed dimensions are max 20x10cm. For better precision in turns, we will implement a differential steering. The two front wheels will be steered using a servo. A color sensor mounted to the bottom of the car will track colored tape scattered on the circuit. The color sensor will be used to check that the car makes a full lap, but also to give "power up" to the car. In MarioKart, there are items that can be collected as power ups. In our project, a certain color will correspond to a power up. We can imagine different power ups: the car goes faster, the other car goes slower, it stops the other car,... We have chosen the following components:

Motor: We decided to use the same engine as for the LEGO car. It is powerful enough for the size of our car and already has an integrated gearbox. Moreover to increase the speed of the car, we think to print in 3d gears to change the reduction ratio and to have in our case a less strong reduction.

Servo: We will use the same type of servo we used on our individual projects.

Board: We will use the ESP32 with camera so we can stream what the car sees.

Motor driver: We decided to go with the L298N driver to control our motor. It is well adapted to the motor we have chosen.

Battery: We will use a 1000mAh 7.4V lipo battery to provide a fast and long lasting experience.

Battery protection: We will use a XL6009 Buck-Boost Converter. The input voltage is 5V - 32V which is well suited to our 7.4V battery. Moreover the output voltage is adjustable and allows to have 1.25V - 35V which is perfect for our motor controller which receives 5V maximum.

Battery charger: To charge our the lipo batteries we are using, we will use a lipo battery charger.

Differential: We will use the LEGO differential (number 6573). It corresponds well to the size of our car and very robust enough. This differential works with 3 gears. (number 6589)

Wheel: We will use LEGO wheel. We have chosen the wheel number 56145 and the tyre number 55978. Those rim and tyre are bit larger (22 mm against 14mm) than the ones we used in the first project. We find them a bit more racing and more suitable for MarioKart cars.

1.1.2 Tracks

For the circuit, we're going to keep it as cheap and simple as possible: we want a big enough track to make it interesting, and for this we would like to use a total of 4m^2 in fiber blocks (divided into 16 50cm x 50cm blocks). We would like to store these blocks in the DLL storage room (by stacking each 50cm x 50cm block on top of each other to save space, the blocks won't be permanently stuck together). To assemble the circuit, we would then only have to put these blocks next to each other on the ground.

The circuit borders will be delimited by black tape, and if we have the time at the end of the project we will design and print 3D walls as well. The "power up lines" and "checkpoint lines" will be marked by colourful tape.

Since we don't know how far away the camera can be from the track for the tracking to work properly, we have designed two different tracks. The cars will end up measuring about 9cm in width, so the width of both tracks varies between 20cm and 30cm so that overtakes will be easily doable (track areas with a width of 20cm are not intended as overtaking areas). We have a diagram for each track, where each square represents 10cm in real life. The first circuit is 2m x 2m (see figure 1) and the second is 1.5m x 1.5m (see figure 2). The general circuit outline is in black, the bridge is in blue, the fiber blocks are represented in green and a 5cm border is represented in red (we think that there should be at least 5cm in between the edge of the track and the edge of the fiber block surface). This is a detail, but even if there is a bridge on the circuit (to make it more interesting), it won't cross another part of the track, because the only way we will have to track the cars will be using the camera above the circuit, and without other sensors, a bridge crossing over the track would cause a problematic blind spot in the tracking of the cars.

The phases to build the track are very simple, for this reason the only deadline we are fixing is to get the track done as soon as we receive the fiber blocks and tape. We will start as soon as we get the fiber blocks and aim to finish it straight away since it will be done quickly and will be indispensable for testing:

- Assemble the fiber blocks
- Mark the track borders with tape (and cut the tape at points where it crosses two blocks)
- Mark the track power ups and the track checkpoints

We will use a tripod to hold the iPhone flat at 3.3m above the circuit. We need it to be as stable as possible for optimal precision, and it always has to remain at the same position relative to the track.

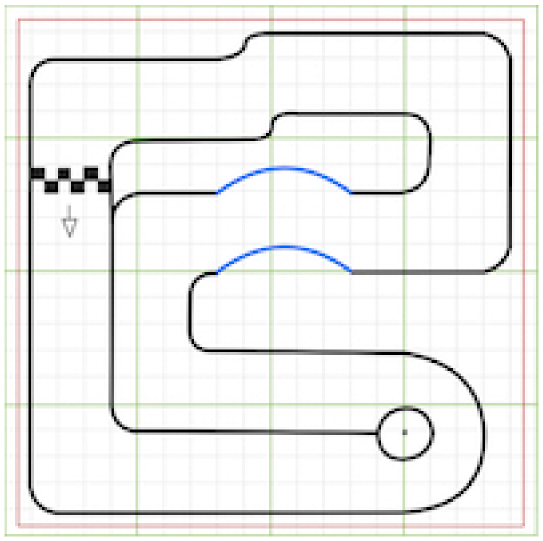


Figure 1: Big circuit

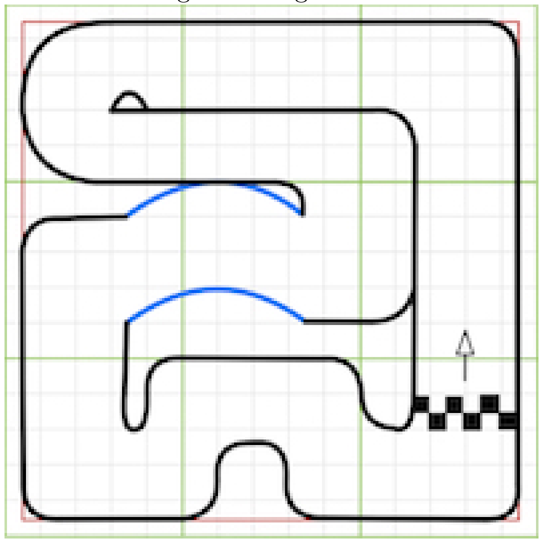


Figure 2: Small circuit

1.2 Software description

In this project, cars will either be remotely controlled or be driven by AI. Remote control will be done via remote controllers. The main hurdle software wise will be to implement the latter. To do so, we would implement a centralized main server that coordinates the actions of the AI driven cars.

The main server would receive two main forms of input: coordinates of the cars and distance between each cars coming from an iOS Application using the device camera, standing above the circuit and color sensors data from on-board sensors of the cars.

1.2.1 iOS Application - Vision Computing and ML

The iOS Application using the device camera would serve to establish a coordinate system and measure distance between the cars. It will use the Vision Framework by Apple combined with the Core ML Framework that will allow us to use Machine Learning in our application to detect the cars using a trained classification model on 3 different images, each clearly different (one image per car, each placed on the roof). The application will be written in Swift. Depending on time limitations, we will either take some already trained models for the three different images or compose the database and train the model by ourselves (taking pictures, labelling them, etc..).

The coordinate system established by the application will be based on the position of the phone with respect to the circuit. Therefore we will always place it at the same position with respect to the circuit using a custom tripod made out of wood.

The coordinates of each car and the distance between each cars will be sent to the server multiple times by second. Processing it on the server side will allow us to determine the real orientation and speed of each car. (The server already having the theoretical orientation and the theoretical speed because it controls the cars and will be able to adjust those values thanks to the real values computed).

For the communication between the iOS and the server controlling the cars, we were thinking of using the UDP for the transmission protocol. Indeed, we don't care about lost packets here and therefore re-transmissions as data will be sent multiple times per seconds. To be sure we don't face packets reordering, we will add a sequence number to each packet we send so that the server know in which order the packets were sent and can process them accordingly.

We already have a first draft of an application detecting QR codes on an iPhone using the frameworks mentioned above (we used a trained database of QR codes) that is very effective, it doesn't compute coordinates or distance between them yet.

Even if we are quite sure that we are going to get the application working considering the code we already have working; if it does not work, we will use a webcam for the overhead camera and use the code from ”Intelligent Traffic Backend” but this will have more latency (and the webcam will need to be taken into account in the budget).

1.2.2 iOS Application

Since the controllers are too expensive to make, we will also make an iOS application (a different one) that enables a player to control a car from their phone. Controlling one of the cars will be the application's only point.

1.2.3 Individual Car Servers

Each car will have a local server receiving inputs from the main server and sending the data from its color sensors to the main server.

In terms of communication between the main server and the cars servers, the idea would be to open a UDP socket between the main server and each of the cars. We would rather use UDP than TCP because dropped packets are not too much of an issue here as the goal is to have the lowest latency possible. We will include a sequence number in each packet header and process the packets accordingly on the main server and car server side to avoid packets re-ordering.
Each car will then be set up with the IP and Port of the UDP server socket.

The overall goal of the server we want to set up is to process the data received as fast as possible and send back inputs to the cars ideally more than 10 times per second. The code on both server and car sides should be very lightweight and optimized to allow fast processing and transmissions.

The color sensors will serve to detect checkpoints and power-up stations (as explained before in section 1.1). If it's a power up, the car will directly process it and activate the effect, we don't need a round trip through the main server that would add latency. The information that the car has crossed a power up or a checkpoint will still be sent to the main server so that the information is displayed on the GUI.

1.2.4 Main Server - Model Predictive Control (MPC)

The cars will be controlled from the main server using Model Predictive Control with the IPopt solver. This will let us calculate the best set of actions that the car will take at each given step and then pick the best controls to send to each car for the next immediate time step.

The MPC will run on the main server, the main server receiving data from the iOS Application and each car's server. The main server and the MPC will both be coded in python, in order for us to access optimisation libraries already existing in python. This main server will receive the coordinates in a buffer that will contain the last X samples with each sample containing the (x, y) positions of the cars from the iOS Application as described in the iOS Application section. X will need to be determined through testing but for now, we are thinking of using the last 10 samples. We will then use those coordinates to determine the orientation of the cars and their velocity. These are the values used later on by the MPC determine the correct course of action.

The MPC solver will also need to possess a reference trajectory for the circuit containing the reference position (x, y). This reference trajectory will be hard-coded and the coordinates are within the grid defined by the iOS app. We need to determine if we want to fit a 3rd degree polynomial curve to determine the cross track error *cte* at each step or if we want to pre-process the waypoints.

The MPC will plan ahead for a certain time-step length N, this will be determined later on because too large of an N will induce a lot of computations that the solver will not be able to provide in real-time (causing latency induced accuraccies) and any model inaccuracies will be compounded for longer periods of times. However if the N used is too small, the model won't be able to find the best control actions and we won't have an optimal control policy. We will need to test out different values of N to find the best one.

The MPC will need to receive an input vector containing the state of each car. This is characterised by its position (x, y) in the track (determined by the camera system as explained in the previous section), its orientation angle ψ and its velocity v. The MPC will then give out as output a vector (the actuation signal) containing the throttle a in range [-1, 1] (full brake and full throttle) and the steering angle δ (its range is yet to be determined because this depends on our car model itself). Here is the definition of our vectors in python:

```python
from collections import namedtuple

State = namedtuple('State', ['x', 'y', 'psi', 'v'])
Actuation_Signal = namedtuple('Actuation_Signal', ['a', 'delta'])
```

Note: *psi* corresponds to the orientation of the car and *delta* to the steering angle.

At each given point in time, we will want to update our state. To do so, we will use the following formulas:

```python
dx = cur.x + cur.v * math.cos(cur.psi) * dt
dy = cur.y + cur.v * math.sin(cur.psi) * dt
dv = cur.v + act.a * dt
dpsi = cur.psi + (cur.v/Lf) * act.delta * dt
```

Here, *dt* is our time interval and will be determined through testing. *cur* is the current state vector and is of type *State* (definition above). The constant Lf corresponds to the radius of the circle formed by the car with constant steering angle and velocity on a flat terrain and is also yet to be determined as we need the cars to do so.

The formulas are basic kinematic equations of motion that will let the solver take into account the cars' dynamics.

The MPC solver will need to use a cost function that the solver will try to minimise. The cost function will almost definitely change when we test out the MPC model on the car. Right now, we are taking inspiration for the cost function from "Model Predictive Control (MPC) for Autonomous Vehicles". But we will need to fine tune it so that it can really respond to our needs. So for now, we have the following representation:

```
Weight = namedtuple('Weight', ['w_cte', 'w_epsi', 'w_roc_a', '
                                  w_roc_delta'])
Cost_Info = namedtuple('Cost_Info', ['cte', 'epsi', 's_dif_velocity
                                  '])

def cost_function(act: Actuation_Signal, cost_info: Cost_Info):
    cost = weights.w_cte * cost_info.cte
    cost += weights.w_epsi * cost_info.epsi
    cost += cost_info.s_dif_velocity
    cost += math.pow(act.a, 2)
    cost += math.pow(act.delta, 2)
    cost += weights.w_roc_a * roc_a
    cost += weights.w_roc_delta * roc_delta
    return cost
```

roc_a and *roc_delta* are constants representing the rate of change of the throttle and steering angle respectively, they are yet to be determined since they also depend on the car model. The $Cost\_Info$ type contains information needed for the cost function:

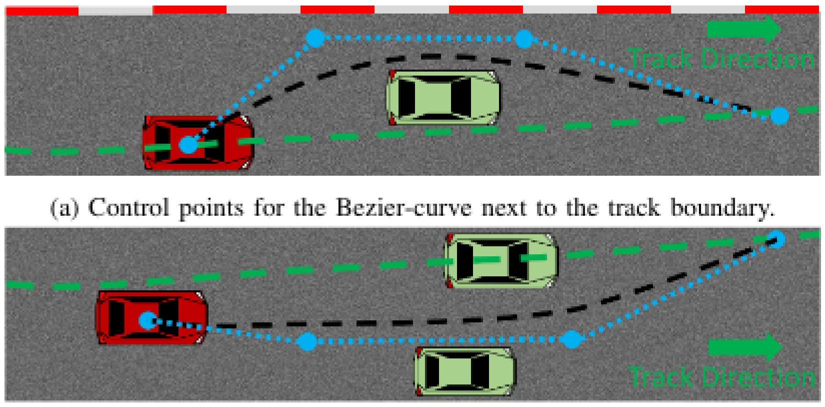
- the *cte* attribute represents the squared difference between the position of the current state and the reference position.
- the *epsi* attribute represents the squared difference between the orientation of the current state and the reference orientation.
- the $s\_dif\_velocity$ attribute represents the squared difference between the velocity of the current state and the reference velocity.

These are then determined in the following way:

```
cte = math.pow(cur.x-ref.x, 2) + math.pow(cur.y-ref.y, 2)
epsi = math.pow(math.abs(cur.psi - ref.psi), 2)
s_dif_velocity = math.pow(cur.v - ref.v, 2)
```

Each of these attributes including *roc_a* and *roc_delta* have weights attached to them. These weights will help with accuracy control (for *cte* and *epsi*) and to guarantee a smooth trajectory (for *roc_a* and *roc_delta*). *weights* is of type $Weight$ and contains all these weights. The weights will be determined and adjusted through testing.

Since we are running a multiplayer game, we cannot simply directly apply the MPC to each car because they will interfere with one another. In order to fix this issue, we want to take inspiration from the "Autonomous Racing with Multiple Vehicles using a Parallelized Optimization with Safety Guarantee using Control Barrier Functions" article and use the MPC when there are no other cars around but when there are surrounding cars blocking the path, we will utilise an optimization-based planner which through the use of Bezier curves will let our car overtake others and get back on the track. Here is a representation of this system from the paper itself:



(a) Control points for the Bezier-curve next to the track boundary.

(b) Control points for the Bezier-curve between two adjacent vehicles.

1.2.5 GUI

As a last bonus, we want to implement a graphical interface with a minimap of the cars around the circuit that also displays the speed and direction of the cars, along with the predicted trajectory of the cars based on the AI prediction.

2 Related projects

- Intelligent Traffic Backend project (MIT project from last year)
- Example Of iOS LiDAR
- Wizards-Chess (MIT project from last year)
- Camera streaming library via RTMP, HLS for iOS
- Car model
- Car model

- Model Predictive Control (MPC) for Autonomous Vehicles
- Autonomous Racing with Multiple Vehicles using a Parallelized Optimization with Safety Guarantee using Control Barrier Functions
- Nonlinear MPC for autonomous racing example

3 Challenges and risks

One of the challenges we will be facing is attaining the wanted performances for the car especially having a fast enough car and a precise steering mechanism.

- The servo can be tough to calibrate to be exactly at 90 degrees so the car might not go straight.
- The motor gear box we want to buy is too slow so we will need to modify the gear ratio to up its speed. This can be a little tricky because the gears are relatively small and the 3D printing machine is not too accurate for such designs.
- The battery we are using is a lipo battery which can be a little dangerous is not used correctly so we will need to be careful with it and we will use a Protection Module.
- Fiber block could tear if one of us isn't careful (or if it gets damaged in storage). At least it wouldn't be a hard fix, but the team would then have to reimburse the damaged fiber block(s) out of pocket. There is also a risk that the cars' wheels slide on the fiber blocks or that the color sensors don't detect the tape.

Another challenge, we will be facing is related to the MPC and the optimisation-based planner. Seeing as how we will need to control three cars with these systems, it could become very computationally heavy. If that is the case, we will need to reduce the time step length of the MPC and/or simplify the overtaking system in order to lessen the computations made and simplify the problem of riding the cars.

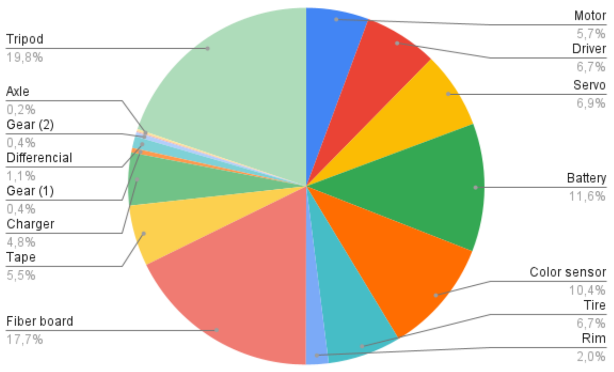
When it comes to the optimisation-based planner, this may be too hard to develop with the information given to us by the camera. If that is the case, we could scale back the project by preparing different routes for each car in advance and then simly run the MPC on each car without the overtaking maneuver.

For the iOS Application, even if we are quite sure that we are going to get the application working considering the code we already have working; if for some reasons we can not get one of the part working (distance algorithm or coordinate system) or the cars are moving too fast to be recognized or any latency issue, we will use a webcam for the overhead camera and use the code from "Intelligent Traffic Backend" as a backup solution but this will have more latency (and the webcam will need to be taken into account in the budget).

4 Components to buy

- 3x ESP32 cam boards (11.95 CHF/piece)
- 3x Motor (5.35 CHF/piece)
- 3x Driver (6.3 CHF/piece)
- 3x Servo (6.55 CHF/piece)
- 3x Battery (10.95 CHF/piece)
- 3x Boosters (4.50 CHF/piece)
- 3x Color Sensors (9.80 CHF/piece)
- 12x Tyres (1.57 CHF/piece (ID:4499234/55978)
- 12x Rims (0.48 CHF/piece (ID:4299389/56145)
- 3x LEGO differential (1 CHF/piece (ID:4299389/56145)
- 9x LEGO gear (0.1 CHF/piece (ID:4299389/56145)
- 3x LEGO gear (0.36 CHF/piece (ID: 6346517/18575)
- 6x LEGO CONNECTOR PEG (0.05 CHF/piece (ID: 4666579/6562)
- 6x LEGO axle (0.09 CHF/piece (ID: 370526/3705)
- 12x LEGO axle (0.09 CHF/piece (ID: 6129995/3705)
- 6x LEGO BUSH FOR AXLE (0.04 CHF/piece (ID: 6271820/42798)
- 16x 0.250 m^2 Fiber boards (3.13 CHF/piece)
- 1x Coloured tape (9.70 + minimum order surcharge of 5.90 = 15.60)
- 1x Tripod (43 CHF/piece + 15 CHF shipping = 56 CHF)

TOTAL = 333.84 CHF (we will pay the extra personally)



5 Structure

In terms of the global milestone, we are setting a goal to have all cars done by week 11 and also have the MPC developed such that if there is only one car on the track, this car will be able to drive and direct itself on the circuit. The car will be able to be detected by the camera using a QR code figuring on top of the car.

We also have more specific deadlines for each aspect of the project separated on a week per week basis per person. These are in the following sections.

5.1 Software deadlines

5.1.1 Main Server - MPC

- Week 7:
  - Rokhaya: Structure the optimization-based planner and define the initial formulas used
  - Hugo and Daniel: Figure out the reference trajectory and how to integrate it with the MPC, prepare interaction between the server and the cars
- Week 8:
  - Rokhaya: Implement the optimization-based planner
  - Hugo and Daniel: Implement the solver into the MPC, prepare integration for the MPC and optimization-based planner
- Week 9:
  - Rokhaya and Hugo and Daniel: Begin testing the system on at first one car, start testing out different settings and value combinations
- Week 10:
  - Rokhaya and Hugo: Fix errors and bugs discovered in week 9, fine tune the settings, further testing, begin testing with multiple cars
- Week 11:
  - Rokhaya and Hugo: Continue testing with multiple cars, adjust the settings, fix errors and bugs discovered
- Week 12:
  - Rokhaya and Hugo: Integrate the MPC and the optimisation-based planner with the server
- Week 13:
  - Rokhaya and Hugo: Fix the bugs and errors discovered in Week 12, some more fine tuning of the settings
- Week 14:
  - Rokhaya and Hugo: Same as Week 13 and finish the documentation of the MPC and the optimisation-based planner

Alongside this, each week we will write documentation for the tasks we are working.

Note: there often figures the names of two individuals for the tasks. This is because the work will be done in parallel between the two of us and there is a lot of testing to be done in order to improve our model. So there will be a lot of iterations needed, where a number of different settings and functions will need to be tested out.

5.1.2 iOS Application - Computer Vision and Machine Learning

- Week 7:
  - Arthur: Set up the application prototype with image recognition for three different type of images (pre-trained model from Robotflow), create the GitHub repository and document the code
- Week 8:
  - Arthur: Implement a coordinate system, test the accuracy of the system, check performance wise how many times a second can compute the coordinates
- Week 9:
  - Arthur: Implement the distance algorithm that computes the distance between each car (recognized images) multiple times by second
- Week 10:
  - Arthur: Implement the custom UDP transmission protocol (with sequence number in the header) in the application
- Week 11:
  - Arthur: Work on the custom UDP protocol with sequence number and catch up on any delay if necessary
- Week 12:
  - Arthur, Albert and Daniel: Depending on time limitations, train a classification model on custom images (one for each car) or choose existing pre-trained model for the images we will place on the roof of each car

Application should be fully done by the end of week 12.

Alongside this, each week the code will be documented and posted on GitHub.

Note: If we are stuck on implementing the distance or the coordinate system for more than one week (one week is allocated for each of those tasks) - we will move back to the implementation of last year project "Intelligent Traffic Backend" as mentioned in 1.2.1).

5.1.3 GUI

- Week 10:
  - Thomas, Albert and Daniel: Designing the MiniMap and starting to work on the GUI
- Week 11:
  - Thomas, Albert and Daniel: Finalizing the GUI and linking it to the server
- Week 12:
  - Albert and Daniel: Designing and coding the controller iOS app
- Week 13:
  - Albert and Arthur: Connecting everything to the server (+ test)

5.2 Hardware deadlines

5.2.1 Cars

- Week 7 and 8:
  - Thomas and Albert: Design the car in 3D
- Week 9:
  - Thomas and Albert: Printing of the car, perfection of the parts and code for the car to run (+ test)

5.2.2 Tracks

- Week 7:
  - Hugo: Code checkpoints and power ups with blanks
  - Daniel: Design + print bridge on Autodesk360
- Week 8 and 9:
  - Hugo: Take code from MPC part to implement checkpoints
  - Daniel: Build circuit + test tape detection
- Week 10:
  - Daniel: Figure out optimal space between checkpoints + design circuit walls
- Week 11:
  - Hugo: Take code from cars part to implement power ups
  - Daniel: Print circuit walls + test bridge and walls
- Week 12:
  - Hugo: Fix potential bugs in the code
- Week 13:
  - Hugo: Try to implement new power ups and/or animations