PROJECT REPORT

Line-following Ackerman car based on PID-controller

Table of contents

Objective

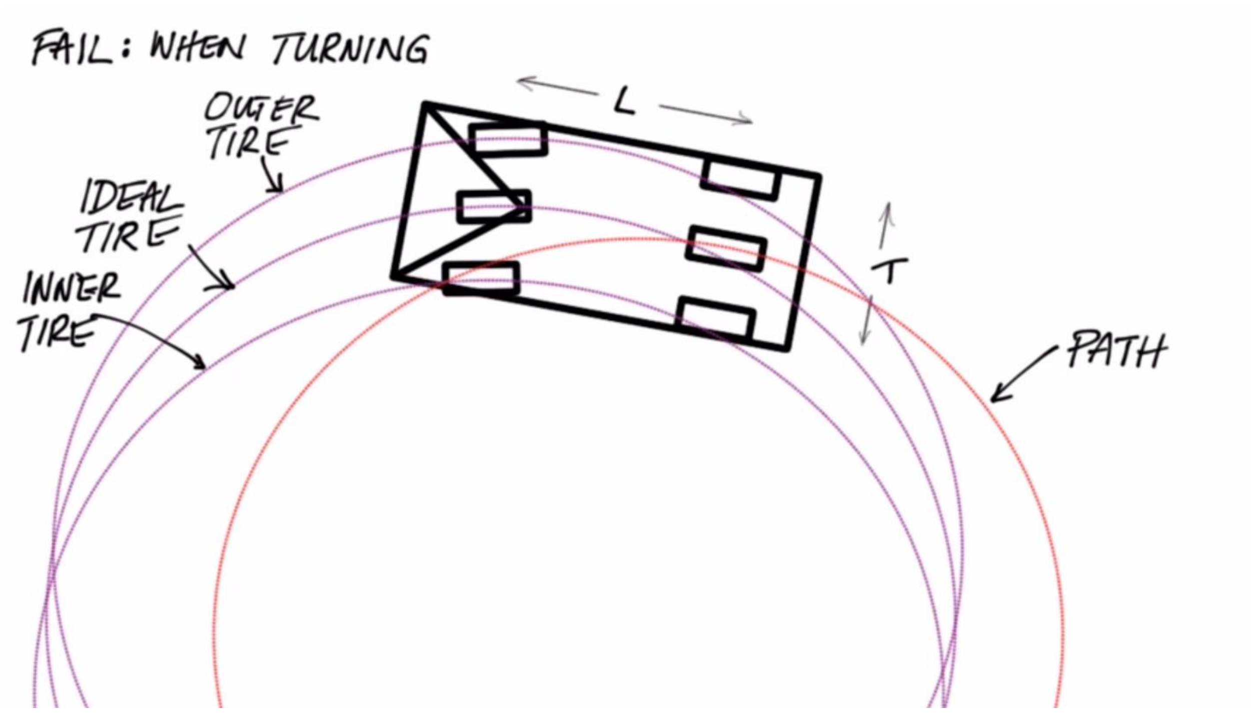
Create a scene in V-REP simulator with Ackermann car model. Then make the car follow the line. Stabilize its movement using PID controller. Choose optimal speed and coefficients corresponding coefficients PID coefficients.

Introduction

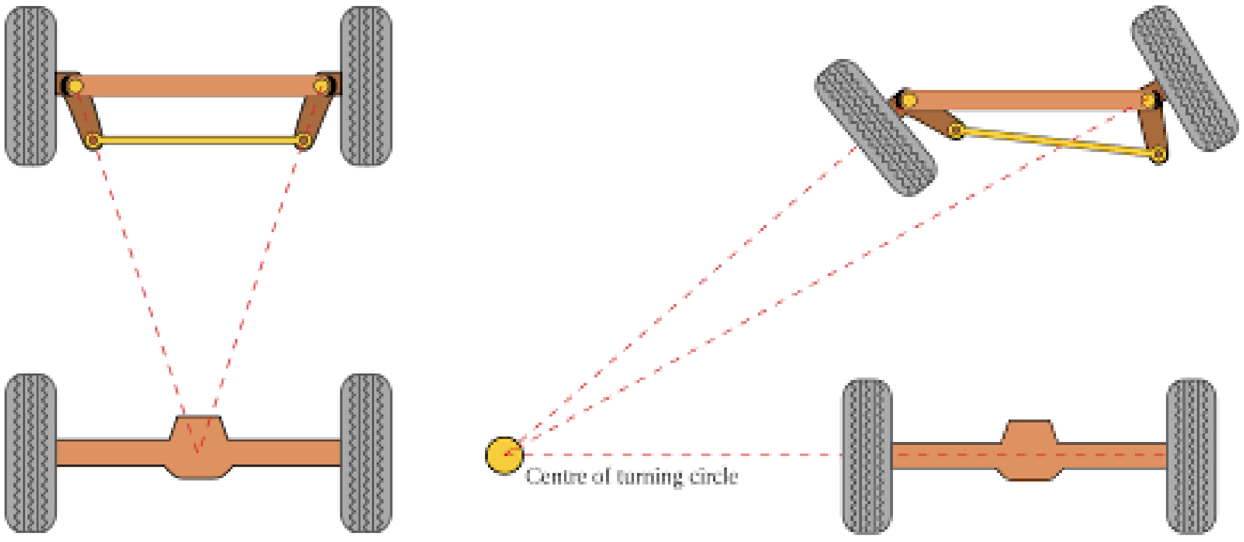
Ackermann steering

Ackermann steering geometry is a geometric arrangement of linkages in the steering of a car or
other vehicle designed to solve the problem of wheels on the inside and outside of a turn needing to trace out circles of different radii.

It allows preventing situation in which left and right wheels of a car move different trajectories (as on the picture below).



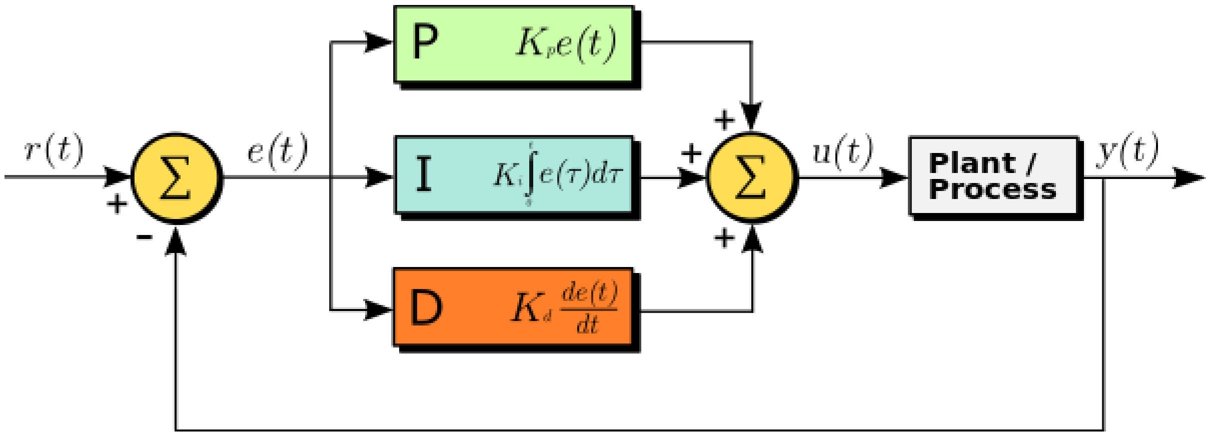
Ackerman steering was invented for solving this problem.



PID-controller

PID-controller is a simple kind of controllers, which is widely used in stabilization of different closed-loop systems. IN continuously calculate the error value and automatically applies correction by combining 3 parts:

- Proportional (P): current error value
- Integral (I): previous errors
- Derivative (D): estimation of future errors by its current rate of change



The overall control function in continuous time domain can be expressed as:

$$u(t) = P + I + D = K_p * e(t) + K_i * \int_0^t e(\tau)d\tau + K_d * \frac{de(t)}{dt}$$

In discrete time domain it has the following form:

$$U(n) = P + I + D = K_p * E(n) + K_i * \sum_{k=0}^{n} E(k) + K_d * (E(n) - E(n-1))$$

So, in this work the formulae of PID-controller in discrete time domain is used.

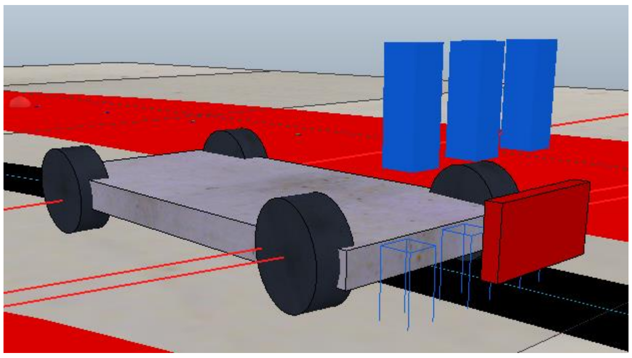
Description of the system

Construction of the Ackermann car

In simulations, the Ackerman car with following parameters was used:

- L = 0.1289 – distance between front and back axis;
- T = 0.0755 - distance between front left and back wheels.

Also it has 3 vision sensors on the front bumper.



Geometry of Ackermann car

On the picture below, you can find description of the car's maneuvers.

For r – desired trajectory radius, desired rotation angle δ can be found as

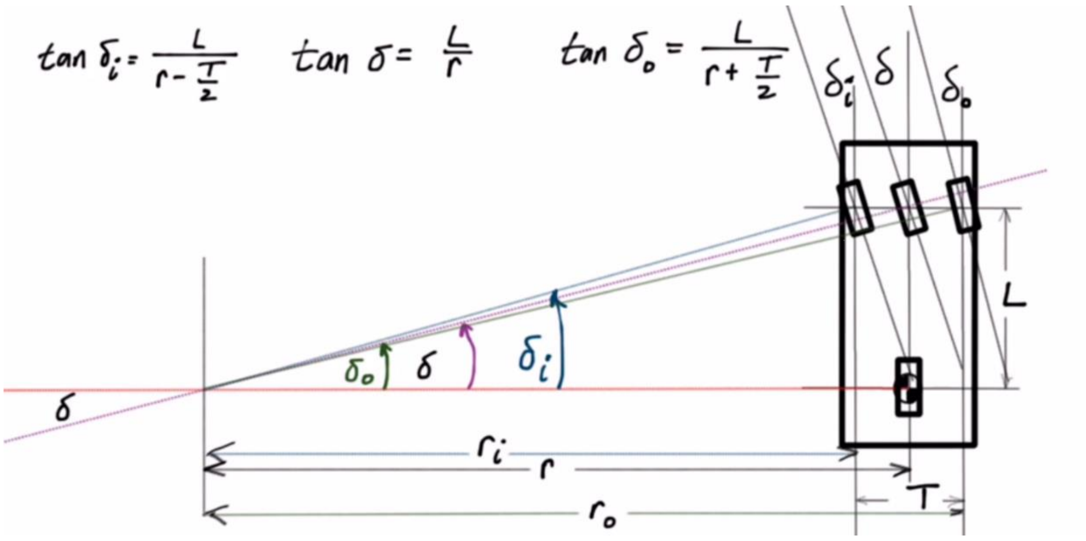
$$\delta = artan\left(\frac{L}{r}\right), \delta_i = artan\left(\frac{L}{r_i}\right), \delta_o = artan\left(\frac{L}{r_o}\right)$$

$$\tan\delta = \frac{L}{r} \Rightarrow r = L / \tan\delta$$

With $r_i = r - T/2 = L * \tan\delta - T/2$, $r_o = r + T/2 = L * \tan\delta + T/2$:

$$\delta_i = artan\left(\frac{L}{L / \tan\delta - \frac{T}{2}}\right), \delta_o = artan\left(\frac{L}{L / \tan\delta + \frac{T}{2}}\right)$$

These equations will be used later.

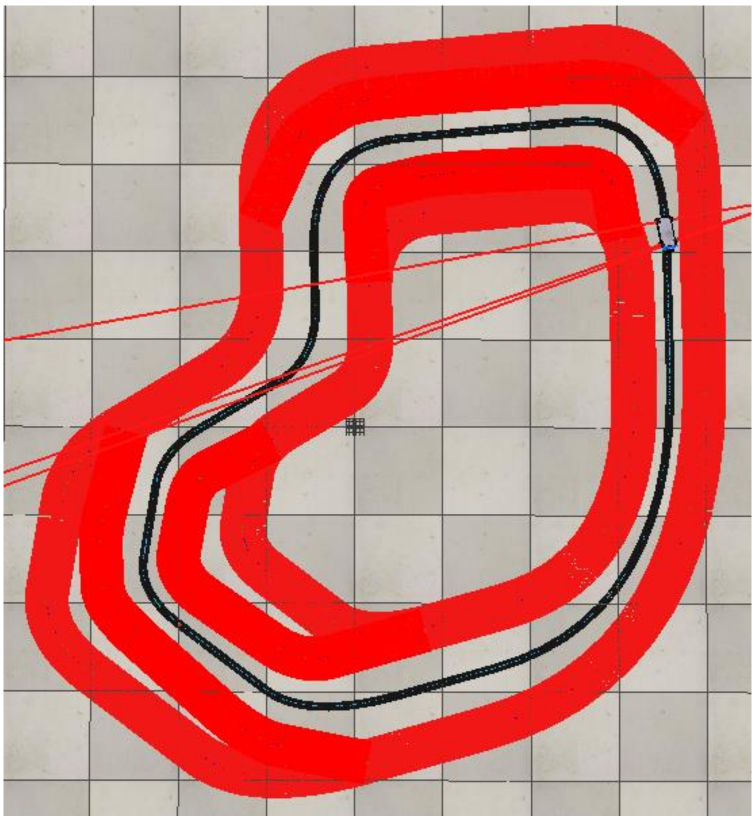


$\tan\delta_i = \frac{L}{r - \frac{T}{2}}$
$\tan\delta = \frac{L}{r}$
$\tan\delta_o = \frac{L}{r + \frac{T}{2}}$
δ_i
δ
δ_o
L
δ_o
δ
δ_i
δ
r_i
r
r_o
T

Environment

In the simulation, the following trajectory was used (black line between red areas).

Red areas don't have influence on car's motion but they are used for detection of its position: if the car is on these areas, then it has left its stability threshold and is not on the line anymore.



Vision Sensors

The Ackermann car model has 3 vision sensors of orthographic projection-type.

Left and right vision sensors are used for detection relative position of the car to the black line and calculating the error. For this case, we take the difference between their average of intensity values. These values vary in [0, 1] (0 for black and 1 for white colour). In a position, when the path lies exactly in the middle of 2 sensors, the tracked values from both sensors would be the same (so, difference is zero, there is no need in correction). The value of difference vary in [-1, 1], so the correction angle would depend on this value.

The third (middle) sensor is used for detecting lap of the car with red area (that means it doesn't following line anymore). For this case the average of red value from the sensor is used. This value is also vary in [0, 1]. The simulation stops when it becomes more than 0.9.

Choice of algorithms and technologies

Choosing environment

At the first part of the project, when the task was small and the whole algorithm fitted in 50 lines of code, it was easier to maintain it in default V-REP' Lua scripts. At that time there was a single scene file containing model and code in it.

Then the tasks became more complex, so that it was more rational to move it to a separate project on Python and control simulation through V-REP remote API. So, now there is a separate file with a model and a separate project with code.

Choosing optimal parameters

At the first stage, choosing the values of speed and PID coefficients was performed heuristically. However, since it is not very efficient way then it was decided to create an automatic test engine.

The first test engine performed test runs of simulation with different combination of parameters from predefined ranges. (The source code for this generator is in file "TestDataGenerator.py").

But it still wasn't so efficient and it was decided to perform the selection of parameters using genetic algorithms. (The source code of the algorithm in file "NEAT_Tester.py").

Checking end of simulation

When the selection of appropriate parameters performed manually, there was no need for automatic detecting the fact of leaving the stable state by the car and moving out of the path.

After creating an automatic test engine there was a need for detecting such situations, stopping simulation and logging its duration. The first idea was to check relative location of the car to the floor, so that if the car ran out of the floor and "flew in space" then the simulation terminated.

However, there were cases when the car ran out of the path, but haven't fell down but hovered on a border of the floor. For this case was decided to check relative location to the floor not of a whole car but only of front wheels, so than whenever their position is lower than position of the floor, then the simulation was stopped.

But then a new problem appeared. In some cases the car after getting off the track could run around for a long time, so that it gave wrong results. The solution for this problem was chosen to additional border lines: when the car crosses them, then simulation terminates. These lines are painted red and detected by middle vision sensor.

Program description

Files overview

The program consists of several files:

- "ackerman_car.ttt" (V-REP scene file)
- Python project for simulation control:
  - "vrep.py", "vrepConst.py", "remoteApi.dll" (V-REP Remote API library files)
  - "main.py" (program starting script)
  - "simulation.py" (contains functions for opening/closing connection to V-REP, run simulation)
  - "LineFollower.py" (Contains algorithm of car's movement, handling objects' states)
  - "TestDataGenerator.py" (creates test cases by generating combinations of different parameters (i.e., speed, K_p, K_i, K_d coefficients) from a given range)
  - "NEAT_Tester.py" (contains algorithm for performing automatic selection of coefficients using genetic algorithms)
  - "config-neat" (settings for neural network used in "NEAT_Tester.py")

Also it contains data gathered by running different test engines (i.e., TestDataGenerator and NEAT_Tester)

Line following algorithm (“LineFollower.py”)

Class *LineFollower* contains the logic of controlling and moving the car. Firstly, it is initialized by receiving IDs of Ackermann car’s components and setting all angles and speeds to zero. Optionally we can also set the limit on duration of simulation.

Then the control loop starts (*LineFollower.run_car()*). It consists of several steps which are executed while the car follows the line (i.e., hasn’t left stable state) or simulation time limit hasn’t exceeded. They are:

- Read data from vision sensors
- Based on this data calculate error, its integral and derivative part
- Using PID coefficients calculate the target correction angle (δ) (car’s rotation angle)
- Using formulas of Ackermann car’s geometry compute rotation angle for both front steerings separately
- Set computed rotation angles
- Check if the car hasn’t left the path
- Check if simulation time hasn’t exceeded
- If 2 previous conditions are held, perform the loop again

Preparing environment (“simulation.py”)

This file provides mechanisms for:

- connecting to V-REP remote API, loading the scene in function *init_connection_scene*
- starting simulation and running the test with certain parameters in function *run_test*
- closing scene and disconnecting from V-REP remote API in function *close_connection_scene*

Tests generation (“TestDataGenerator.py”)

This class provides mechanism for creating combinations of different parameters which are used in running tests.

Class *TestDataGenerator* is initialized with a list of ranges (which themselves are also lists of 3 values: range starting value, end value and a step of advancing parameter value). Then by demand (on *get_test_case* function call) it returns a unique combination of parameters with values from specified ranges.

Testing by genetic algorithms (“NEAT_Tester.py”)

This file provides mechanism for selecting optimal simulation parameters using genetic algorithms based on *neat-python* library.

Settings for the neural network used by this algorithm (i.e., size of population, mutation rate, max fitness value, etc.) are provided in “config-neat” file.

It is initialized by ranges of valid simulation parameters which are used for normalization/denormalization of neural network’ input/output (since it receives/provides data in range [0, 1]).

Then it consequently runs several generations (default value is 100, but can be changed during initialization) by performing tests and evaluating each genome of population by their fitness.

Fitness of genome is calculated as a product of the speed and time of simulation performed with parameters generated by the genome.

Starting point (“main.py”)

This file provides templates of functions for running simulation:

- ones with certain parameters (*run_simple_test*)
- several times with combinations of parameters created by *TestDataGenerator* (*run_test_engine*)
- during execution of genetic algorithm by *NEAT_Tester* (*run_NEAT_tests*)

Results

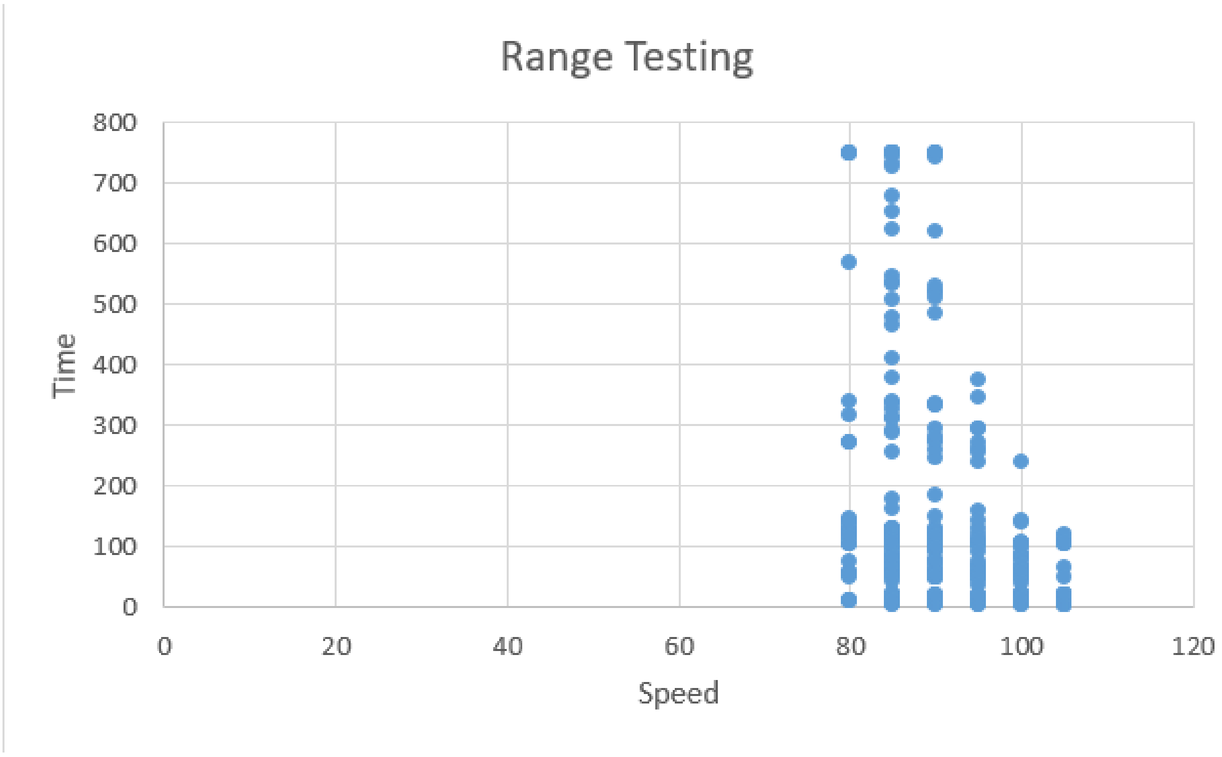
For all tests I’ve set time limit for simulation of 750 sec.

From *TestDataGenerator*

For this testing engine firstly I’ve set next parameters:

- Speed from 80 to 105 with step 5
- Kp from 0 to 10 with step 1
- Ki from 0 to 0.2 with step 0.1
- Kd from 0 to 5 with step 1

Then I’ve got next results (they can be found in “tests_results/results.csv”):



From this diagram we can see, that maximum speed on which the car managed to reach the end of simulation by time limit is 90.

All “successful” results from this testing are in next table:

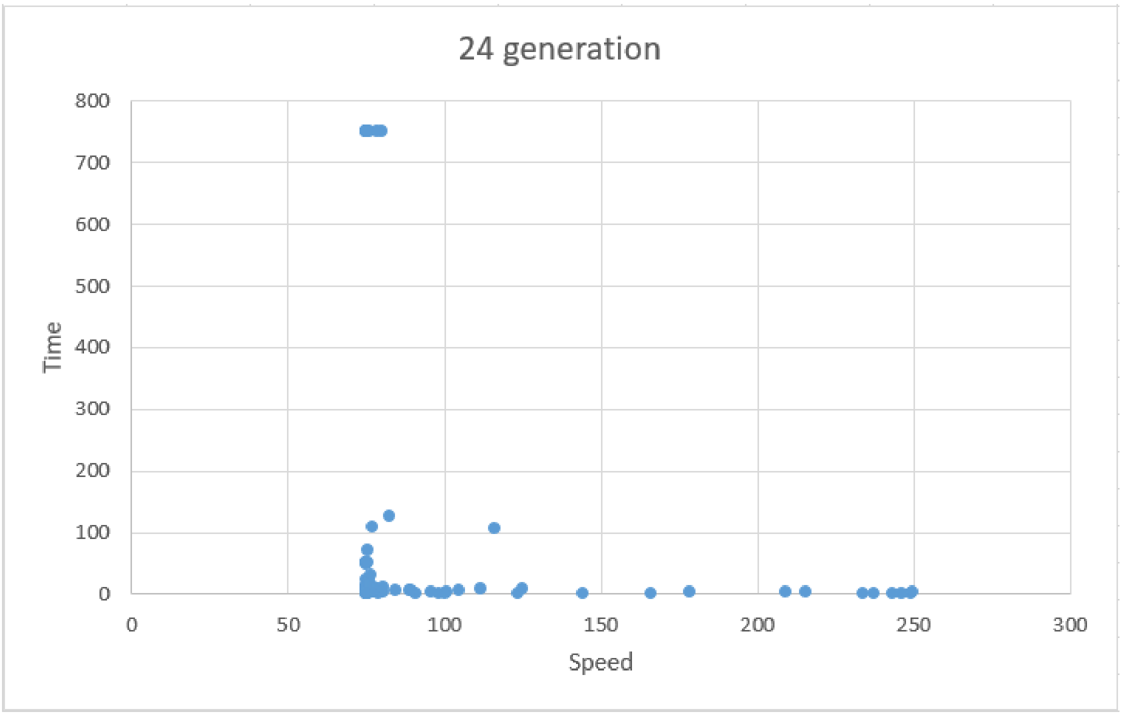
Speed	Kp	Ki	Kd	Time
80	5	0.2	1	750.1915
80	4	0.2	4	750.1836
80	4	0.2	1	750.1825
80	4	0.2	3	750.1804
85	8	0.1	1	750.1659
85	8	0	3	750.1625
90	5	0.2	2	750.1623
85	7	0.2	3	750.1618
85	6	0.2	4	750.1613
85	8	0.1	0	750.1609

90	4	0.2	5	750.1602
90	7	0.2	2	750.1601
85	8	0.2	3	750.1601
85	7	0.2	1	750.1595
85	7	0.1	1	750.1593
85	7	0.2	2	750.1589
85	6	0.2	5	750.1587
85	7	0.1	4	750.1581
85	6	0.1	5	750.1578
85	6	0.2	3	750.1573
85	7	0.2	5	750.1556
85	7	0.2	4	750.1551
85	7	0.1	3	750.155
90	9	0.1	4	750.1543
85	8	0.1	2	750.1486
90	8	0.2	4	750.1436
80	5	0.2	0	750.1384
90	6	0.2	5	750.1127

From *NEAT_Tester*

Statistics by each generation can be found in “tests_results/gen_run2/generations_stat”.

Let look at statistics of 24th generation with population of 100 genomes. From following diagram we can see, that only several genomes reached the end of simulation by time limit and their speed was not more than 81.



Parameters of these “successful” genomes are gathered in a next table:

Speed	Kp	Ki	Kd	Time
75.00	8.23	0.00	5.03	750.19
75.06	8.22	0.00	3.96	750.19
75.00	5.84	0.13	0.00	750.18
78.51	4.53	0.88	3.96	750.17
80.19	5.42	0.48	0.00	750.17
76.24	5.14	0.24	6.89	750.17
75.00	4.21	0.67	2.86	750.13
75.21	4.91	0.47	0.00	750.12
75.00	6.66	0.04	0.00	750.10

Final version software

All resources (code and model) are available here:

https://github.com/DaryaLari/ControlTheoryProject

References

https://en.wikipedia.org/wiki/Ackermann_steering_geometry

https://www.youtube.com/watch?v=i6uBwudwA5o

https://en.wikipedia.org/wiki/PID_controller

http://www.coppeliarobotics.com/

http://neat-python.readthedocs.io/en/latest/