Colegiul Național 'Gheroghe Șincai'
Baia Mare

PYTH FINDER

Îndrumători:
Prof. Anton Carmen
Prof. Contraș Diana
Prof. Pop Grațian

Autor:
Contraș Adrian

2024

PythFinder

Installation 3

Description 4

What makes PythFinder unique..... 5

Future Plans 5

Technologies Used 6

Library Stability 8

Usage 9

Create a Robot 9

Joystick Control 10

Feedback? 12

Presets 12

Trajectory Usage 14

What are Trajectories? 14

How to Create Trajectories? 14

What are Markers? 15

How are Markers processed?..... 15

Example 16

Trajectory Visualisation 17

Velocity Grahp 18

Generate Velocities 19

Interface Settings 20

Library Architecture 21

Trapezoidal Profiles 22

Credits 25

DOCUMENTATION

Creator: Contraș Adrian 🎧

Installation

Before we dive into it, make sure you have:

- a `Python` version greater than 3.10 (last version is recommended);
- `pip` installed on your device (usually pip3 for 3.x versions, but pip works too);
- my team's `font` 'graffitiyouthregular' (used for the interface);

Now the installation it's as easy as writing a command in the command prompt or in Visual Studio's terminal:

```
pip install pythfinder
```

or (on the most devices):

```
pip3 install pythfinder
```

Description

`PythFinder` was developed by team Omega Core in the scope of enhancing motion planning for the First Lego League competition.

Usually teams use blocks for coding their autonomous routines because of the lack of `micropython / python` documentation online. This approach may be faster to compile, but it sacrifices reliability.

With this in mind, I chose micropython as the main language to run on our `EV3` brick. Throughout the 2023-2024 Masterpiece season, I experimented with on-the-go motion calculations and concluded that they were **way too slow** for competitive usage. As a result, my focus shifted more toword pre-calculated motion (also known as feedforward control).

Because `LEGO®` allowed bricks' processors aren't capable of doing fast calculations, creating a script that would do just that seemed to be the way.

So I developed a `trajectory generator tool` that runs locally on your machine and generates a .txt file with all the necessary information for the robot to mimic the desired movements. You just need to copy the generated text into the robot's code folder to be read during the initialization.

On the robot side, there is a simplified version of just the following aspect of the library, along with methods to construct trajectories back from the `.txt` file decomposition.

Because of this operation, after starting the program, for an average max-points scoring code, there will be a time window of 1 - 3 minutes when the robot loads all the data (assuming 7-8 different launches). At this time, the code won't be accessible. Obviously, the time needed for reading data depends on the data amount, which can be manipulated by the user in multiple ways I'll describe later. **I recommend that you start the program at least 4 minutes before the match.**

But '*why would this method be better?*' you might ask. The answer is **consistency**. This library uses techniques found in industrial robotics control systems, enhancing precision through acceleration limitation profiles, multithreading actions for running multiple motor outputs at the same time, and more.

It's a small price to have one of the most reliable autonomous programs in the FLL competition.

To clarify, this library **IS NOT restricted to use exclusively with EV3**, even though it was initially developed for this type of brick. Since the hardware is separate, even robots like **SPIKE PRIME** or **NXT** can benefit from the generated .txt file.

Currently, I've implemented a plug-and-play solution **ONLY** for EV3 bricks, a pythfinder-quick-start. For other types of bricks, a custom implementation will be required to read and utilize the generated data. Additionally, I recommend running the code for all trajectories in a single program to avoid the wait time associated with loading trajectories during the match.

There are numerous libraries for **motion profiling** on GitHub, but I haven't found one specifically tailored for FLL. My mission is to revolutionize programming for this competition, and that vision has become a reality with **PythFinder**!

What makes PythFinder unique?

- it's the first ever FLL motion profiling library in the world \o/;
- unlike roadrunner (the inspiration for this project), this library doesn't install directly on the robot, resulting in faster calculations;
- the simulator, trajectory builder, and generator are all packaged in one single library while maintaining a friendly look similar to roadrunner's;
- the core logic behind the implementation is completely different (which will be covered in the following sections);
- allows the user to manually drive the robot on the simulator with a controller;
- generates all necessary trajectory values into a '.txt' file to be uploaded to the robot (an innovative solution that wasn't used until our library, from what we've seen);
- allows the user to change the interface without changing the code, just from the interface menu;
- allows graph visualization of the velocity and acceleration for better understanding of the robot's behavior;

Future Plans

- This is just the beginning of my library. I've done a bit of everything to showcase a basic concept. Future plans involve **gradual improvements** in every aspect of the simulator (interface, menu, syntax, trajectory generation, and compatibility with other robots);
- Abstraction and leaving room for improvements are essential practices for continuous progress. Expect to see enhanced versions released periodically. Additionally, a **feedback** section is

available on the team's official GitHub to report any issues or requested features, so I can address them in future updates;

- I also plan to develop a `quick-start` guide for each legal brick in FLL. However, currently, I only have access to the EV3 brick for testing and implementation purposes;

  That said, I had the pleasure of connecting with **Arra**, one of the most prestigious teams in the competition. Composed of six enthusiastic young individuals from Pitești, they had the honor of **representing Romania at the World Championship in Houston** for the past two years, achieving outstanding results!

  They were deeply impressed by my project and expressed their desire to help implement the library on other bricks as well.

- Through close collaboration between the Arra and Omega Core teams, I aim to promote this library both nationally and internationally, utilizing our social media networks and various other communication platforms. This will help reach a wider audience and build a larger community that will actively contribute to the ongoing improvement of the library;

Technologies Used

- I chose to use Python for this project because it is the primary language supported by LEGO® bricks. For visualizing the robot, I used **pygame**, the most popular game creation library in Python. It already had all the basic functionalities I needed (image import/export, support for keyboard and controller, and of course, the display window);

  Initially, I intended to load all the code onto the brick, but due to its outdated processor and limited compatibility with Python (the brick requires MicroPython), it was more efficient to separate this program from the actual robot;

- **matplotlib** was added to help users visualize numerical values in a more intuitive way;

- **memory_profiler** was used to create a graph of memory allocation over time (see *Library Stability*);

- I want to emphasize that from the very beginning of the project, my goal was to implement every aspect of the library **from scratch**. This includes the settings menu logic, the physics and mathematics underlying the robot's movement, trajectory generation, and the interactions between the controller and the interface;

Library Stability

- I don’t expect it to be perfect; that’s why I’ve anticipated possible errors and tried to address them before they occur. However, it’s possible that I may have missed some edge cases, given that this is a user-interactive library;
- For known incorrect inputs, I alert the user with custom exceptions;
- I’ve implemented feedback for every part of the code compilation process through informative messages displayed in the output console. Some of these are 'easter eggs,' but most are meant to keep the user informed;
- To ensure the highest safety for users of my library, I’ve uploaded it to the official PyPi platform, which is the dedicated Python site for hosting libraries. This means it has undergone a rigorous validation process, which eliminates any potentially harmful or malicious behavior;

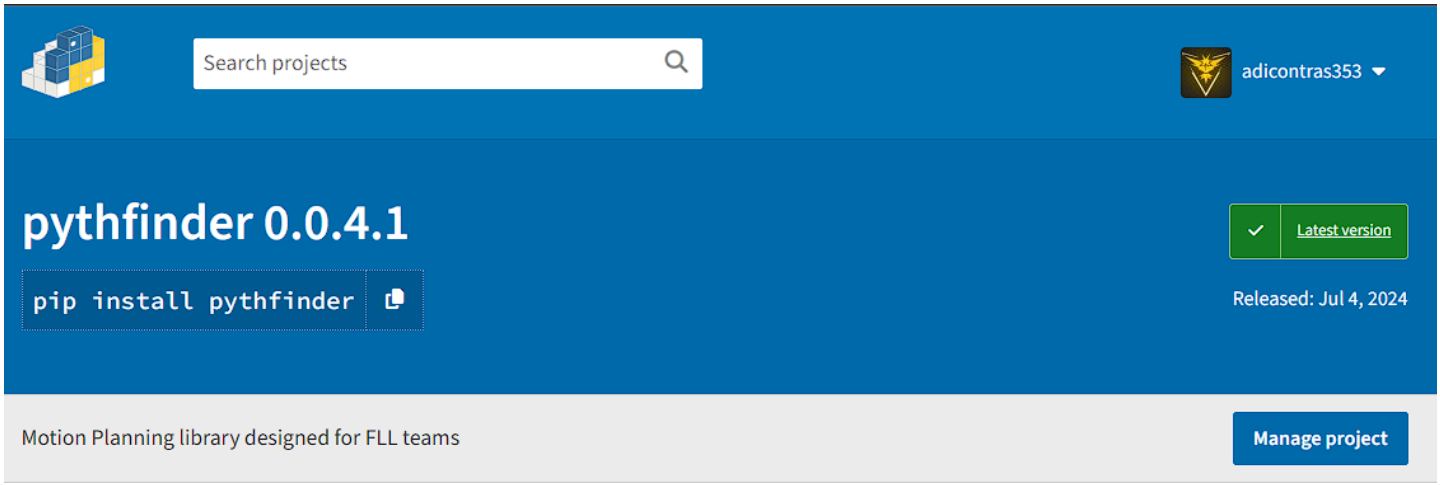


Fig. 1 Security check on PyPi

- User security is paramount in the development of high-quality utility software. Therefore, I have adopted a regular practice of **memory allocation analysis** in my project to ensure there are no risks of memory leaks. By using the specialized **memory_profiler** library, I generated graphs that highlight memory usage over time;

  Clear observations from these graphs, including the one shown in *Fig. 2*, indicate a stabilization of memory consumption after the program starts, thus confirming **normal operation** and the absence of any memory leaks;

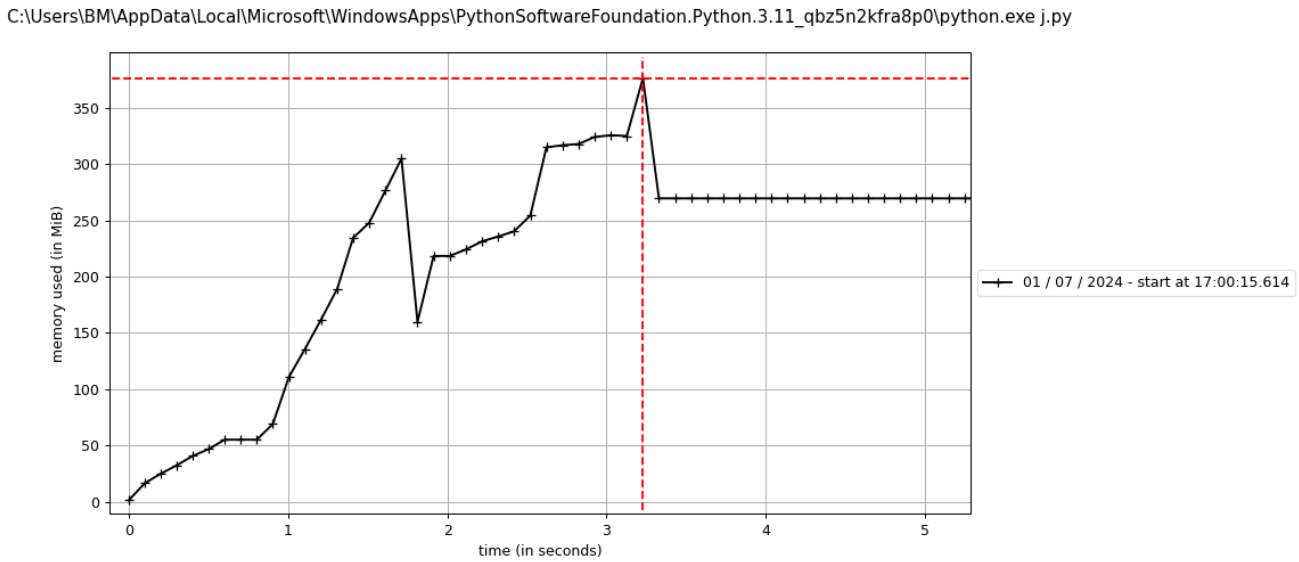


Fig. 2 Memory allocation over time graph

- In addition to its rigorous checks, PyPI provides a robust versioning system, which I have also integrated into my project. I have also implemented a **CHANGELOG** file on GitHub, detailing the changes made in each version to inform users about the improvements. Only **stable** versions are maintained publicly;

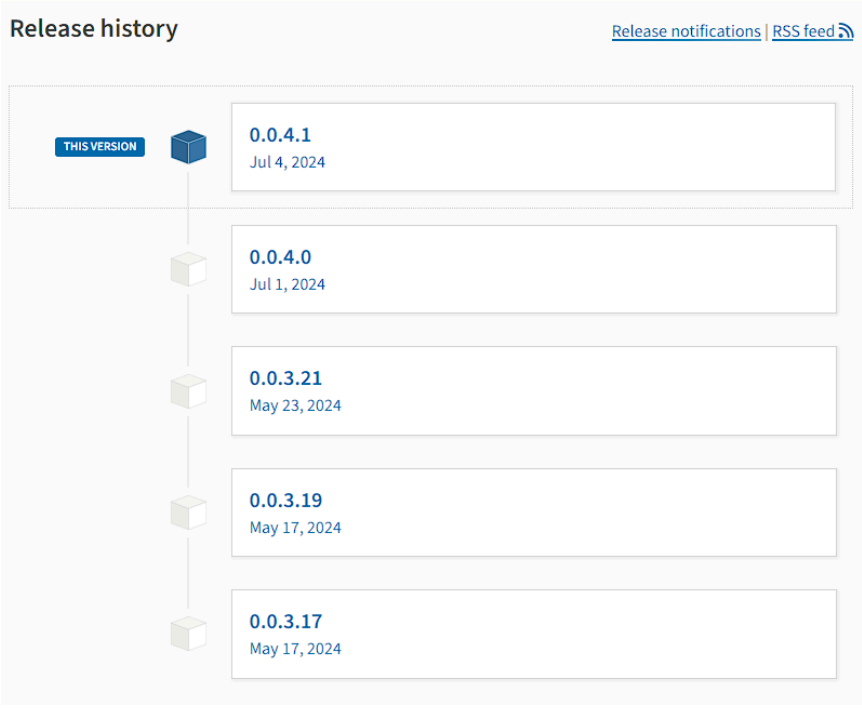


Fig. 3 Versioning system

- To manage my work efficiently and organize its distribution over time, I chose to use the Trello platform. It allows me to track progress and systematically plan future actions;

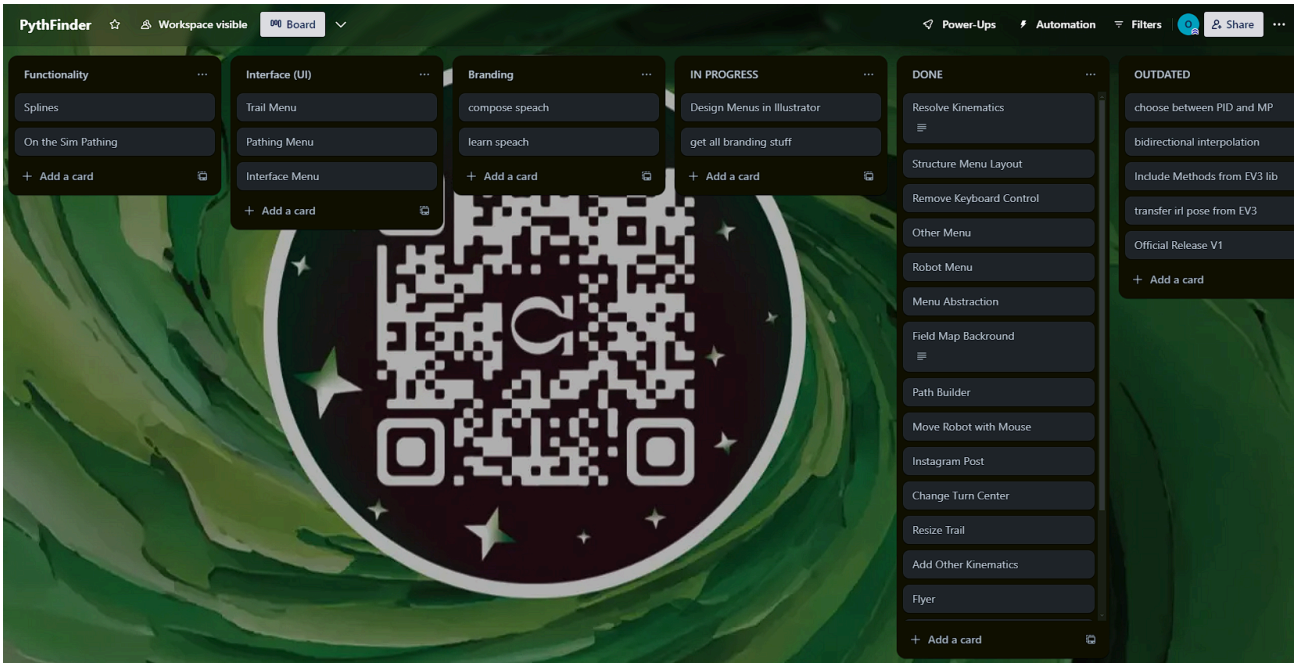


Fig. 4 Organizing using Trello

Usage

To start using this library in your environment, simply create a new python file and import the library:

```
import pythfinder
```

Create a Robot

To enable any robot-visualization elements of the library, you need to create a '*Simulator*' object. This class encapsulates every separate component into one big control center, taking care of the pygame window display, joystick input, and other pygame events.

```
sim = pythfinder.Simulator()
```

This would create a simulator with *default* constants. To override them, simply create a '*Constants*' object with your desired values and pass it to the constructor:

```
# pass your values here
custom_constants = pythfinder.Constants(...)

sim = pythfinder.Simulator(custom_constants)
```

Every time you run the simulator, it'll start with your dataset of constants. You'll learn another way to change constants in the *Interface Settings* section.

Finally, display your simulation:

```
while sim.RUNNING():
    sim.update()
```

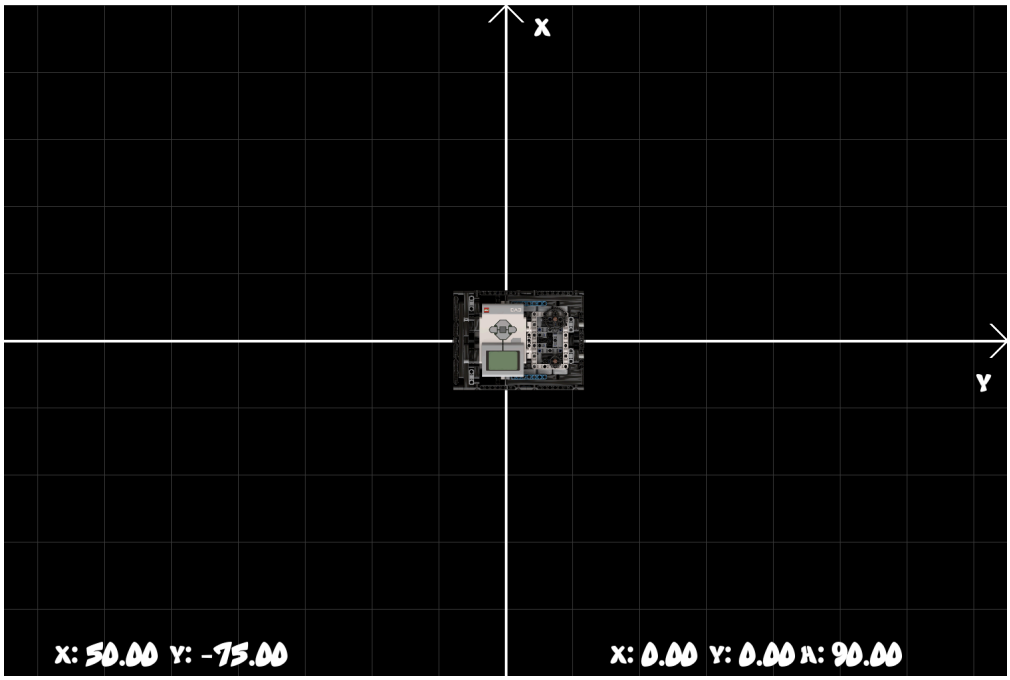


Fig. 5 Initial view of the interface

Please note that the coordinate system used is unconventional compared to the standard one. At the bottom of the simulation window, there are two distinct sets of coordinates displayed. The ones on the left indicate the cursor's position, while those on the right reflect the actual position of the robot relative to the simulated environment.

The code runs until you exit the simulation window. Connecting an *accepted controller* will allow you to move freely across the field.

Joystick Control

PythFinder is built on top of pygame's functionalities, from which it inherits support for XBOX, PS4, and PS5 controllers.

Connecting them is as easy as plugging in the **USB** or connecting it via **Bluetooth**. The simulator will recognize it most of the time; otherwise, it'll raise an error.

The controls used to manipulate the simulator are the following:

*(the order of buttons is: **ps4 / xbox**)*

- `△ / Y` -- go forwards / backwards (when field centric is on);
- `□ / X` -- enter / exit interface setting menu;
- `○ / B` -- reset robot pose to origin / press buttons (when the menu is activated);
- `X / A` -- show / hide trail;
- `left bumper` -- erase trail / set values to default (when the menu is activated);
- `right bumper` -- when held, enters selection mode;
- `D-pad` -- move through the interface menu / select the robot's orientation (when selection mode is on);
- `left joystick` -- control robot's linear velocity + angular velocity (when field centric is on);
- `right joystick` -- control angular velocity (**ONLY** when field centric is off);
- `options / start` -- take a screenshot (found in the 'Screenshots' folder inside the locally installed library location);

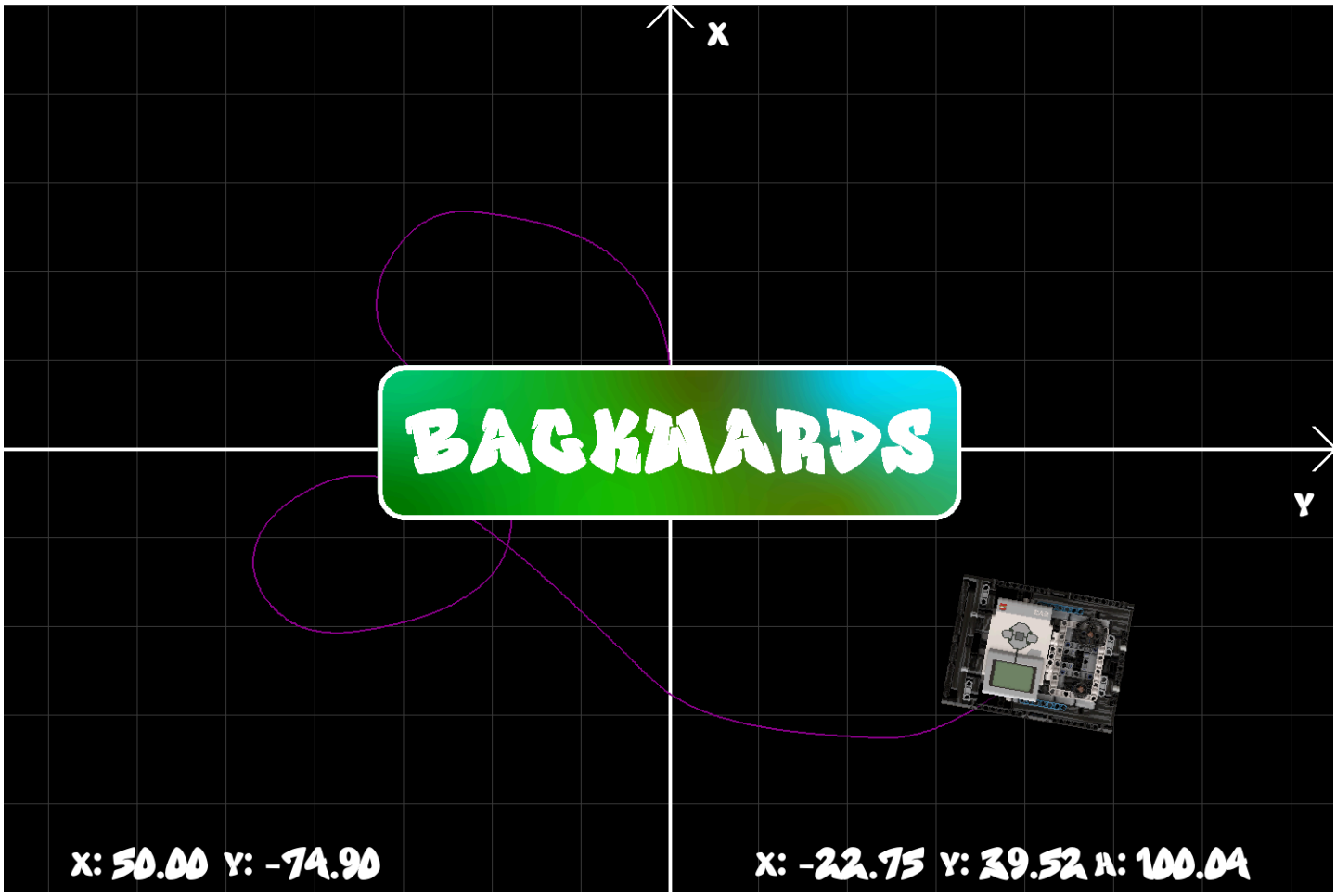


Fig. 6 Movement using a controller

When buttons are pressed, the simulator *alerts the user* to the changes made using both a pop-up window with integrated images, created entirely by me, and text messages displayed in the terminal.

The *trail* left by the robot is managed through separate logic, allowing for the adjustment of both its color and thickness. It consists of lines drawn between all consecutive positions of the robot. Additionally, the *default* settings are configured so that after a short period of inactivity, the trail begins to **fade** out gradually **automatically**, a process that stops when the user moves the robot again.

The trail is not just a collection of points but is divided into **segments**. These segments are defined by the distance between two different positions, a value that can be **adjusted** by the user. This allows each segment to have its own color and thickness, which is extremely useful for drawing a **complex scheme**, often representing strategies on the playing field.

Feedback?

I'll pause the technical documentation to share the feedback I received from people to whom I presented the project:

- Presenting in front of the class during physics, both the **teacher** and **classmates** remarked, '`It's clear that a considerable amount of work has gone into achieving this level of detail and performance.`' A former student of the high school supported this sentiment: '`I took a quick look, it seems super professional.`'
- Speaking with members of other robotics teams from **Satu Mare** (Perpetuum Mobile), **Timișoara** (Cybermoon), and **Alba-Iulia** (Xeo), they were deeply impressed. A member of the Robocorns team from **Baia Mare** even *suggested* that I implement screen resizing, which I did immediately afterward. As mentioned earlier, **Arra** was even enthusiastic about helping out.

Presets

A remarkable **innovation** introduced by this library is the feature called `presets`. These allow users to completely transform the interface's appearance, robot configuration, and chassis type with a single button press. Users have buttons numbered from `1` to `9` on the keyboard, each intended to apply a distinct set of constants that adjust the simulation in various ways. The button with the number `0` resets the interface to the *default* settings.

On top of these predefined options, users have the ability to create their **own** presets, customized according to their individual needs and preferences. This functionality adds an extra level of **flexibility** and **control** over how the `interface`, `robot behavior`, and `simulation parameters` are configured.

By default, button 1 displays the **latest** field from the `FLL` competition:

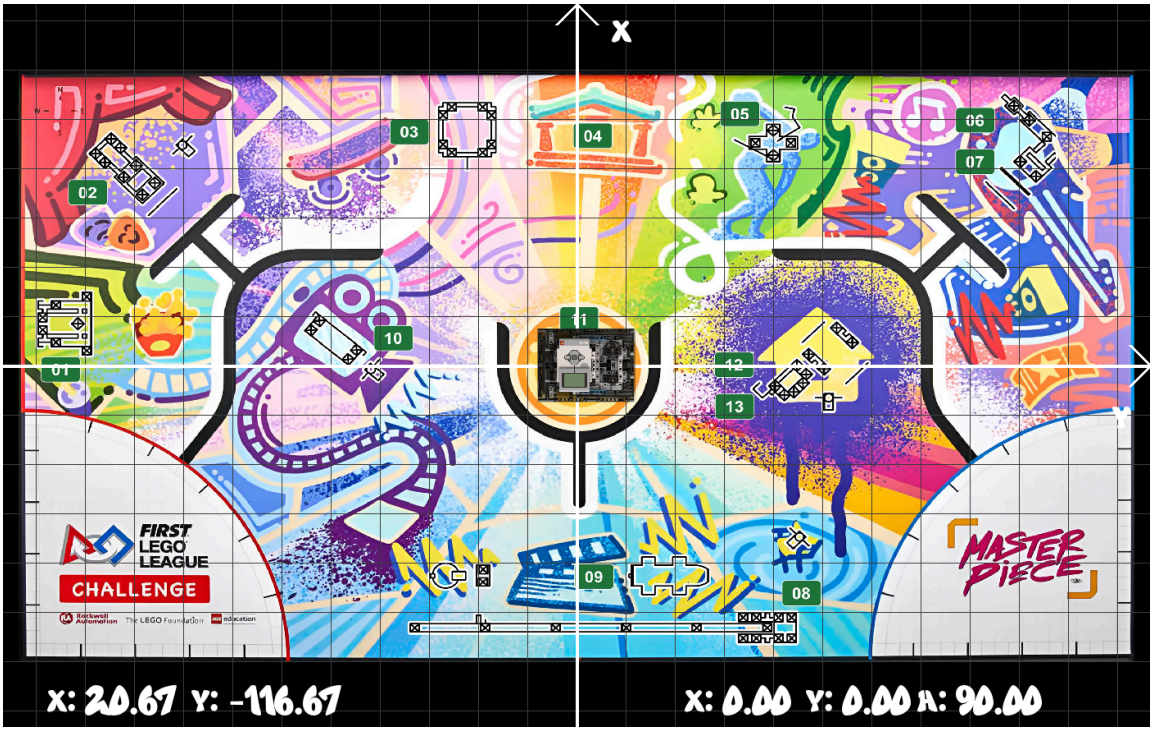


Fig. 7 FLL preset

And button 2 displays the **latest** field from the First Tech Challenge (`FTC`) competition:

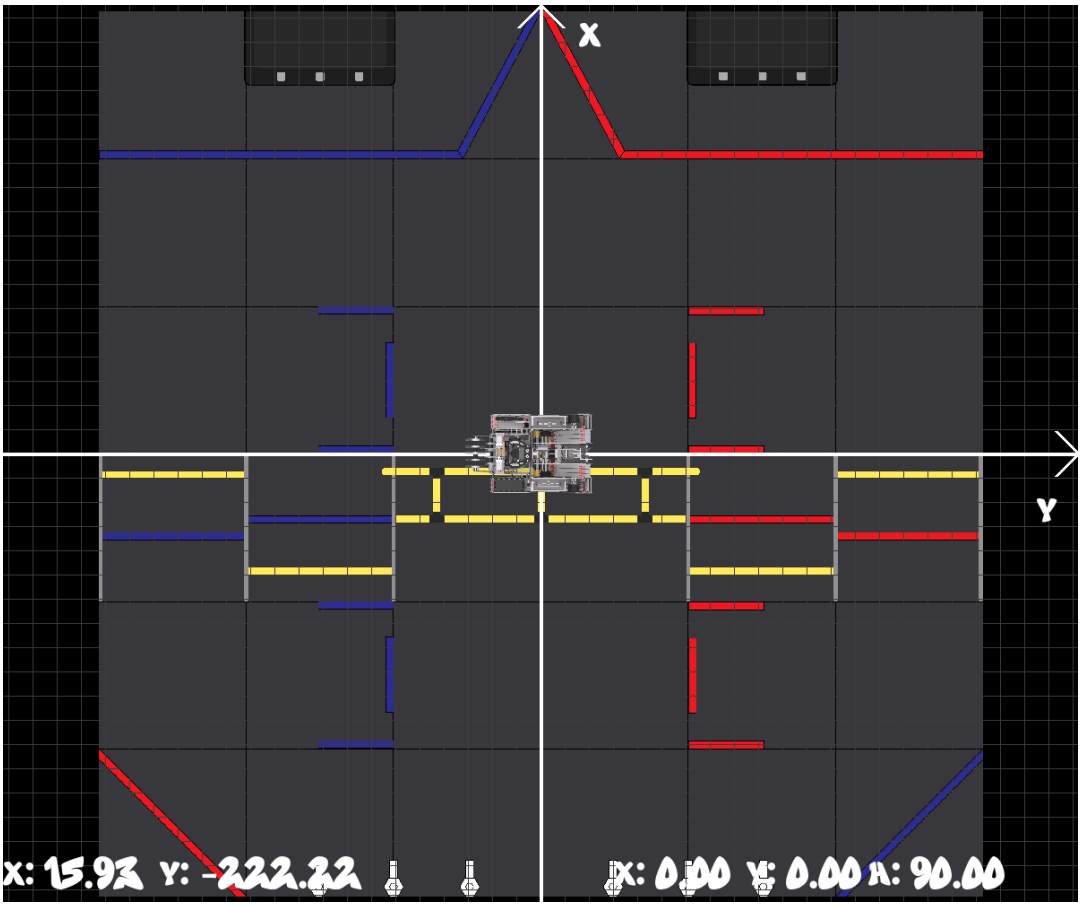


Fig. 8 FTC preset

Trajectory Usage

Ce sunt traiectoriile?

First, we define a specific set of data regarding the robot's position, speed, and distance traveled as a **state of motion**.

Multiple states of motion that exhibit certain similarities are referred to as **motion segments**. These segments are further categorized based on their *complexity*. `Primitives` denote movements with a single degree of freedom (1D), such as pure rotation, pure linear movement, or even stationary states (waiting). These primitives serve as building blocks for `complex` segments, which incorporate two or more primitives and characterize movements with two or three degrees of freedom, primarily intended for *omnidirectional* robots. For For FLL purposes, you'll mostly use primitives, but any motion segment adapts automatically to the chassis used.

Ultimately, all motion segments and auxiliary elements that perform various functions (e.g., other motors usage), known as *markers*, collectively constitute a *`trajectory`*.

How to Create Trajectories?

Trajectories are constructed using the '**TrajectoryBuilder**' class. This class requires a *Simulator* object as a parameter, and optionally, a `starting position` and a `preset` to use. By default, the initial position is set at the origin of the Cartesian coordinate system.

The constructor offers intuitive methods for crafting precise trajectories, incorporating personalised **motion** functions. These functions are engineered to accommodate both omnidirectional and unidirectional robots.

The constructor identifies the type of chassis in use and **adjusts** the provided functions accordingly, as certain chassis types may have physical limitations that render some movements **impossible**. By default, non-holonomic chassis are `tangent` to the trajectory, whereas holonomic chassis are given the option to `interpolate` orientation.

Here is a list of available motion functions:

- *`wait()`*;
- *`inLineCM()`*;
- *`turnToDeg()`*;
- *`toPoint()`* *or* *`toPointTangentHead()`*;
- *`toPose()`* *or* *`toPoseTangentHead()`* *or* *`toPoseLinearHead()`*;

What are Markers?

These functionalities can be integrated with **markers**, facilitating the management of parallel tasks that are independent of the robots' movement by employing `multithreading` techniques. Markers can be configured to activate after a certain **`time`** period or **`distance`**, either **`relative`** to the last motion function or **`absolute`** with respect to the start of the trajectory.

The library also includes special types of markers:

- **`interrupts`**: Disrupt the trajectory's continuity at the specified moment, based on time or distance. Think of interrupts as the sudden braking of a car;
- **`dynamic constraints`**: Allow you to modify portions of the trajectory to operate at different speeds without sacrificing continuity.

Here is a list of all markers:

- *`interruptTemporal()`* *or* *`interruptDisplacement()`*;
- *`addTemporalMarker()`* *or* *`addDisplacementMarker()`*;
- *`addRelativeTemporalMarker()`* *or* *`addRelativeDisplacementMarker()`*;
- *`addRelativeTemporalConstraints()`* *or* *`addRelativeDisplacementConstraints()`*;

`Interrupts` and `Constraints` are **strictly** relative, as we have observed that users find it **difficult** to visualize the trajectory segments to which they apply. They modify the trajectory's course itself, as opposed to markers that call functions and might adversely affect the trajectory's construction. However, if users **request**, I will reintroduce these functionalities, as they were included in the library's initial prototypes.

Markers can also include **negative** values, which are interpreted as relative to the end of the trajectory or motion segment, while **positive** values are interpreted as relative to the beginning of these elements.

How are Markers processed?

Markers are initially **separated** into relative and absolute based on the trajectory segment. Those specified by distance are converted into absolute markers, with *negative* distances corrected to *positive*. The markers are then *sorted* by **priority** (constraints, triggers, functions), **type** (end-time markers), **sign** (negative at the end), and **value**.

The sorted list is divided into distinct groups (constraints, triggers, functions), and each group is processed separately. Function markers are converted into absolute markers, adjusting negative distances to positive and converting distances into time units. This results in a final list consisting solely of absolute time markers and a trajectory modified according to these markers.

Example

After specifying the desired motion, the `.build()` function must be called to compute the trajectory values.

Putting it all together, we obtain:

```
# first launch from our Masterpiece code

START_POSE = Pose(-47, 97, -45)
PRESET = 1

trajectory = (TrajectoryBuilder(sim, START_POSE, PRESET)
              .inLineCM(75)
                    .addRelativeDisplacementMarker(35, lambda: print('womp womp'))
                    .addRelativeDisplacementMarker(-12, lambda: print('motor goes brr'))
                    .addRelativeDisplacementConstraints(cm = 30,
                        constraints2d = Constraints2D(linear = Constraints(
                                                        vel = 10,
                                                        dec = -50)))
                    .addRelativeDisplacementConstraints(cm = 36,
                        constraints2d = Constraints2D(linear = Constraints(
                                                        vel = 27.7,
                                                        acc = 35,
                                                        dec = -30)))
                    .interruptDisplacement(cm = 66)
              .wait(2600)
                    .addRelativeTemporalMarker(-1, lambda: print('motor goes :('))
              .inLineCM(-30)
              .turnToDeg(90)
              .inLineCM(-20)
              .turnToDeg(105)
              .inLineCM(-47)
              .turnToDeg(20)
              .wait(ms = 1200)
                    .addRelativeTemporalMarker(0, lambda: print("spin'n'spin'n'spin.."))
                    .addRelativeTemporalMarker(-1, lambda: print("the party's over :("))
              .turnToDeg(80)
              .inLineCM(-120)
              .build())
```

Trajectory Visualisation

After creating your trajectory, call the '*.follow()*' method and pass the '**Simulator**' object to see your code in action!

This method takes as an optional parameter the following type as a boolean:

- `perfect` = simulator iterates through each motion state and displays the robot in the pre-calculated position. For this mode, you can also change the step size in which the list is iterated. A bigger step size means a faster robot on screen.
- `real` = simulator gives the calculated powers to the robot object, which looks exactly like it would run in real time. This mode is **recommended** for better visualization.

The last optional parameter is '*wait*'. When this boolean is set to True, it waits until the simulator is fully rendered on the user's screen before proceeding with the trajectory. This is useful when perfect following and a big step number are set, it makes you be able to see even the start. Our fifth run looks something like this:

```
# default values
PERFECT_STEPS = 40
PERFECT_FOLLOWING = False
WAIT = True

trajectory.follow(PERFECT_FOLLOWING, WAIT, PERFECT_STEPS)
```

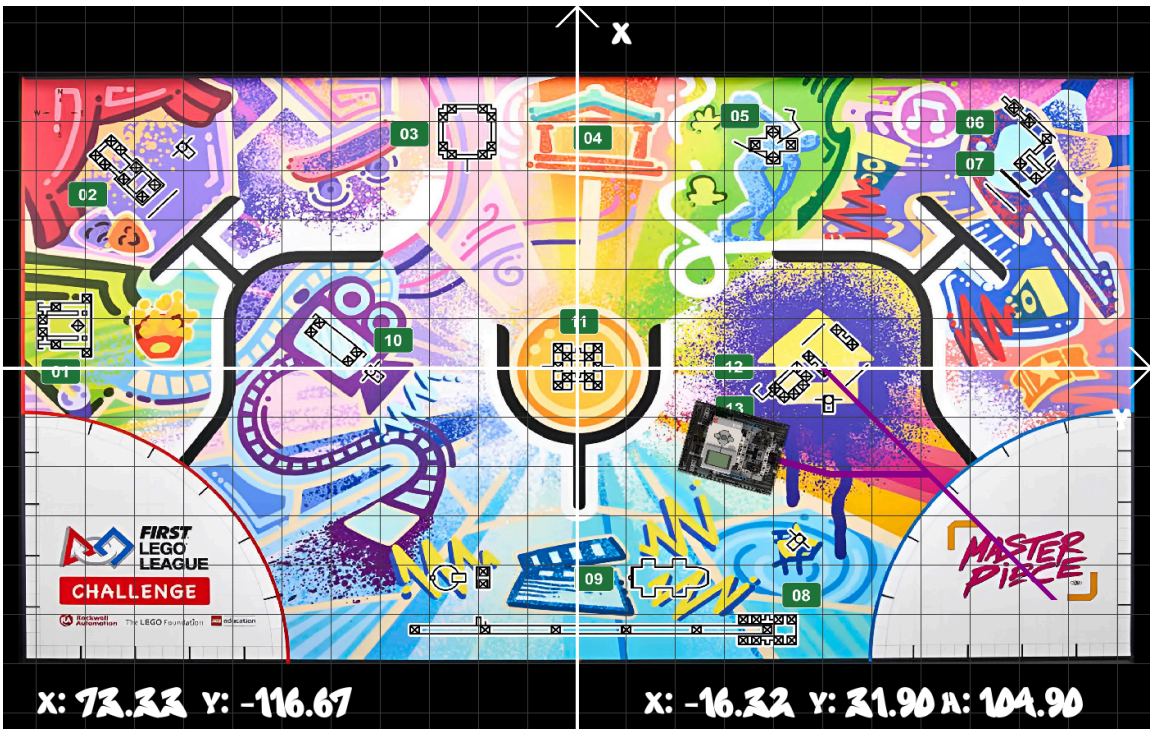


Fig. 9 Viewing the previous code in Autonomous mode

Velocity Graph

To facilitate the understanding of the 'trajectory' concept, I have implemented an easy-to-use graphical visualization method for motion profiles.

I truly believe that this library represents one of the best ways to begin learning the concepts **used in industry**, aiming to assist and inspire future engineers and programmers!

Calling the `.graph()` function will display a Matplotlib graph of the **velocity** and **acceleration** for the left and right wheels. There are also optional parameters to display each value separately. Additionally, users can choose whether they want to view the velocity and acceleration of the wheels or the chassis.

An interesting aspect is the `connect` parameter. By default, it is set to True, causing lines to be drawn between points. Setting it to False reveals discontinuities (in acceleration, as velocity is optimized for continuity).

```
# default values
CONNECT = True
VELOCITY = True
ACCELERATION = True

trajectory.graph(CONNECT, VELOCITY, ACCELERATION)
```

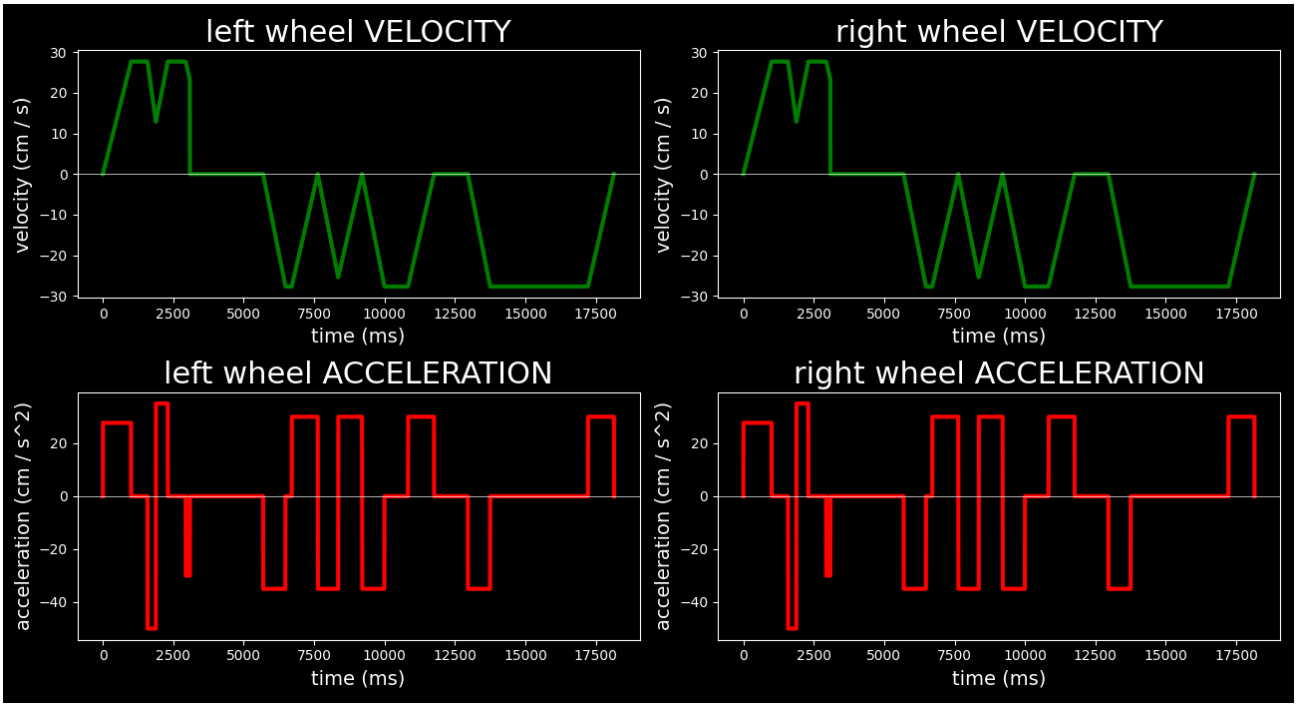


Fig. 10 Graph generated using values from the previous code

Generate Velocities

To actually make the robot move like in the simulator, you'll need to **transfer** the data through a '.txt' file. This is accomplished with the '.generate()' method. Just pass the text file name / path and the step size:

```
STEPS = 6
FILE_NAME = 'test'
WHEEL_SPEEDS = True
SEPARATE_LINES = False

trajectory.generate(FILE_NAME, STEPS, WHEEL_SPEEDS, SEPARATE_LINES)
```

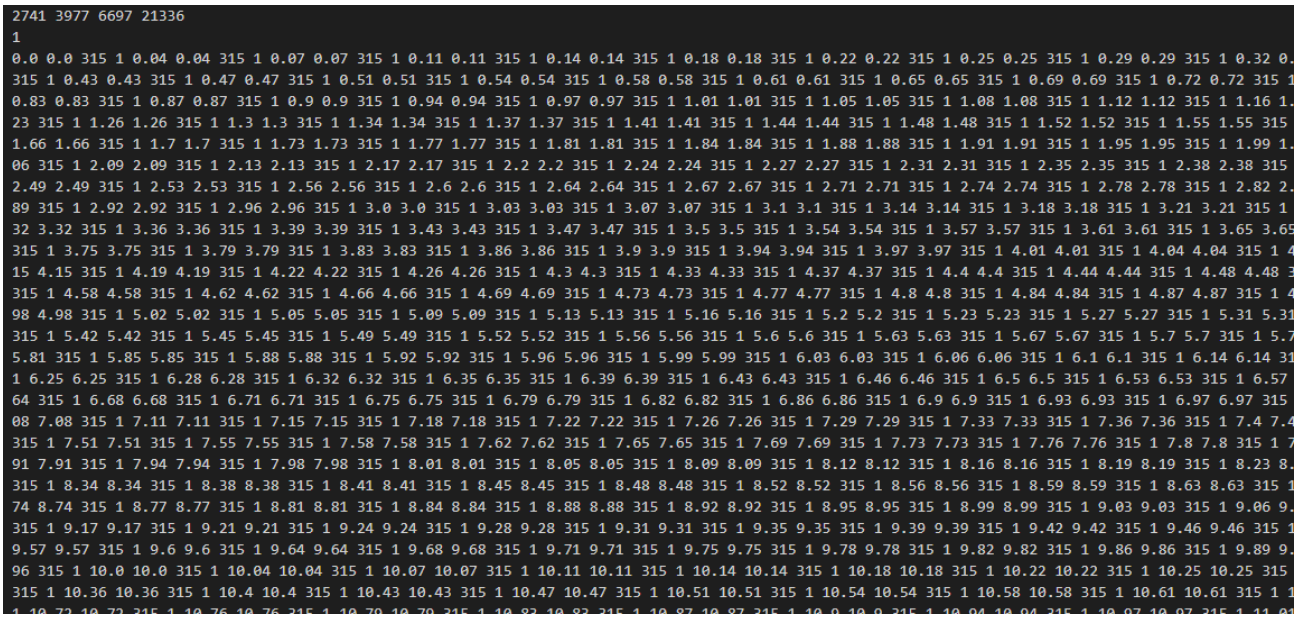


Fig. 11 Example of generated values for the previous code, with step count = 1

The first line generated includes the times at which the markers should act, in chronological order. The second line contains the number of steps to skip when inserting values into the text file, and the last line lists, in order, the following set of data:

- The first **n** values are the speeds for the n wheels of the robot, or the speeds on the X, Y axes, and rotation axis;
- The next value is the angle at which the robot is positioned;
- The final value is the number of consecutive occurrences of the current set.

Each data set represents the state of motion in a **millisecond**, with each sequence being associated with a time after which it will be selected. You can now copy the `.txt` file and load it into `quick-start` to see it in action!

Interface Settings

There are two main ways you can manipulate your simulator environment through constants.

The first way is to simply pass a new instance of '**Constants**' when creating the sim object, changing any of the following values:

```
# constants.py -- simplification

# all modifiable values:
class Constants():
    def __init__(self,
                 pixels_to_dec,
                 fps,
                 robot_img_source,
                 robot_scale,
                 robot_width,
                 robot_height,
                 text_color,
                 text_font,
                 max_trail_len,
                 max_trail_segment_len,
                 draw_trail_threshold,
                 trail_color,
                 trail_loops,
                 trail_width,
                 background_color,
                 axis_color,
                 grid_color,
                 width_percent,
                 backing_distance,
                 arrow_offset,
                 time_until_fade,
                 fade_percent,
                 real_max_velocity,
                 max_power,

                 screen_size,
                 constraints2d,
                 kinematics):
    ...
```

As described in the Create a Robot section, these changes will be automatically applied at the start of the simulation. For an in-depth explanation of the constants, see the documentation.

The second way is through the interface menu (**NOT FULLY IMPLEMENTED YET**) with joystick control. This is a more 'on-the-go' change and will reset every time you restart the simulator.

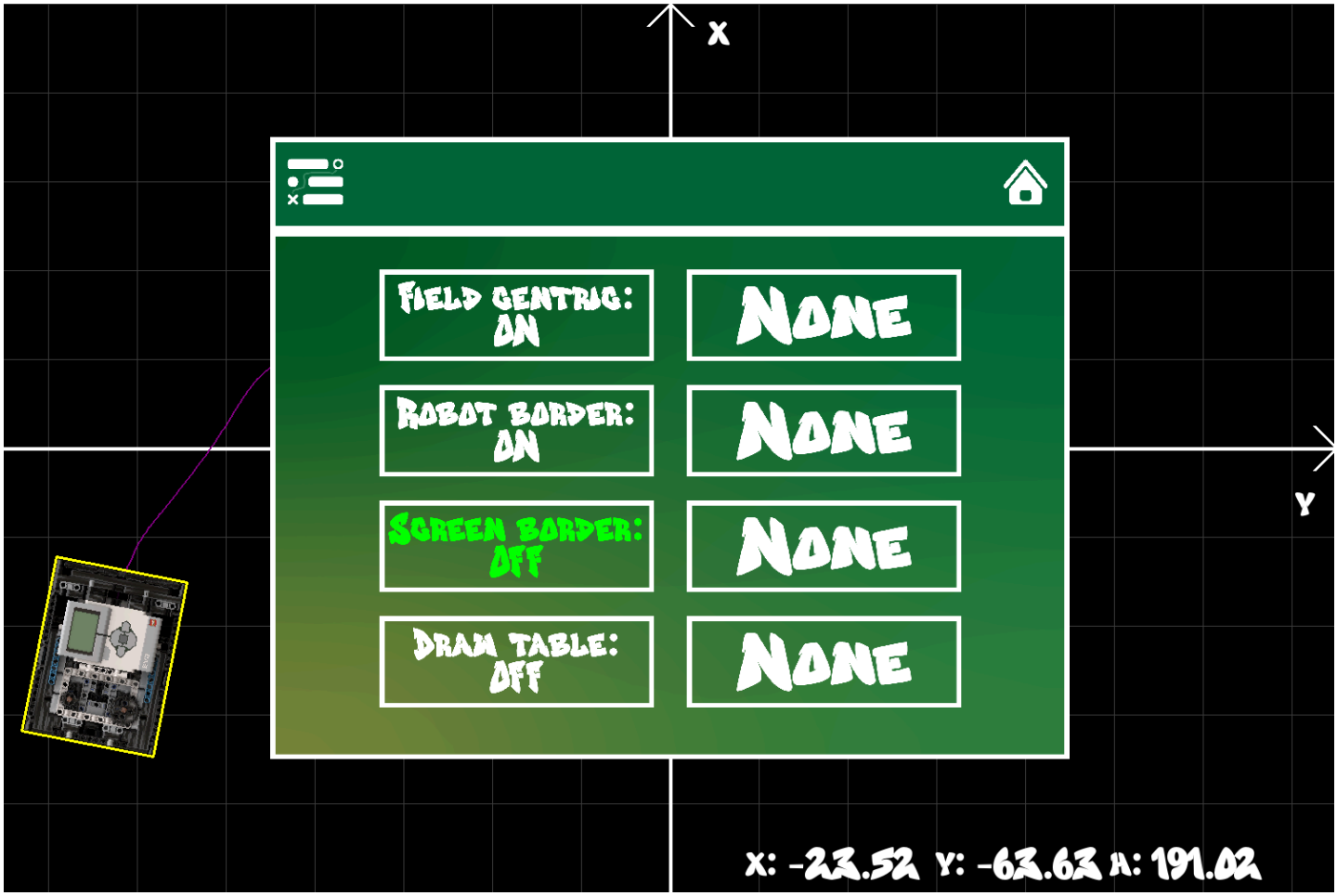


Fig. 12 'Other' section in the interface menu

Library Architecture

PythFinder is designed to be **modular** and incorporates multiple `separate packages`, each with its own functionality isolated from the others.

To ensure code readability, I have used **enums**.

Variable names are descriptive, representing the full names of the objects they refer to.

I have adopted Python programming conventions by prefixing with double underscores (`__`) the functions and variables that **SHOULD NOT** be accessed by users or from outside the class, similar to `private functions`.

I have divided the calculation processes into multiple functions, avoiding long code sequences. This approach facilitates code maintenance and expansion, while also ensuring better organization and structure.

All these practices contribute to creating robust, easy-to-understand, and user-friendly code.

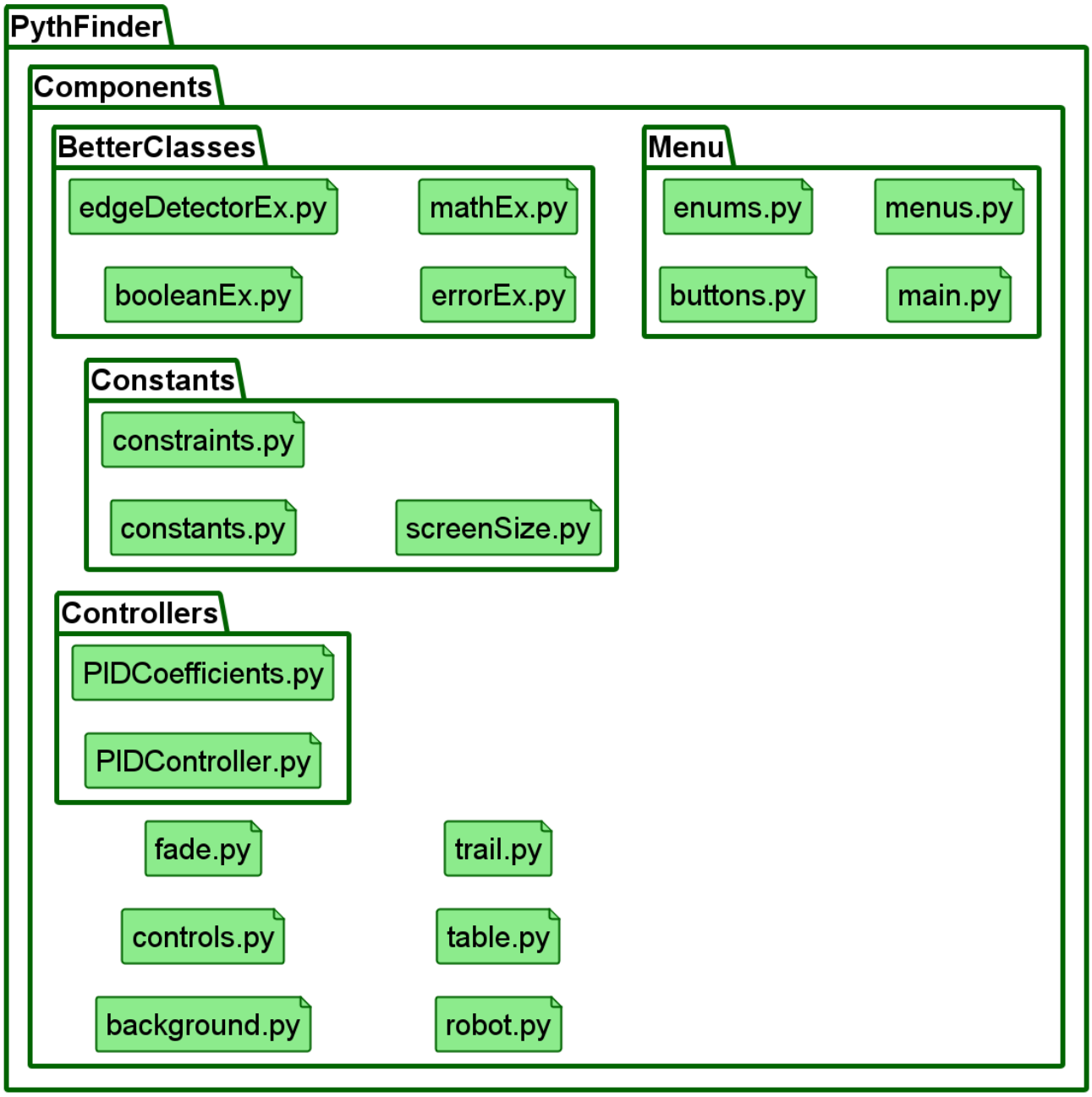


Fig. 13 File organization schema at the core of the library (partial view of files)

Trapezoidal Profiles

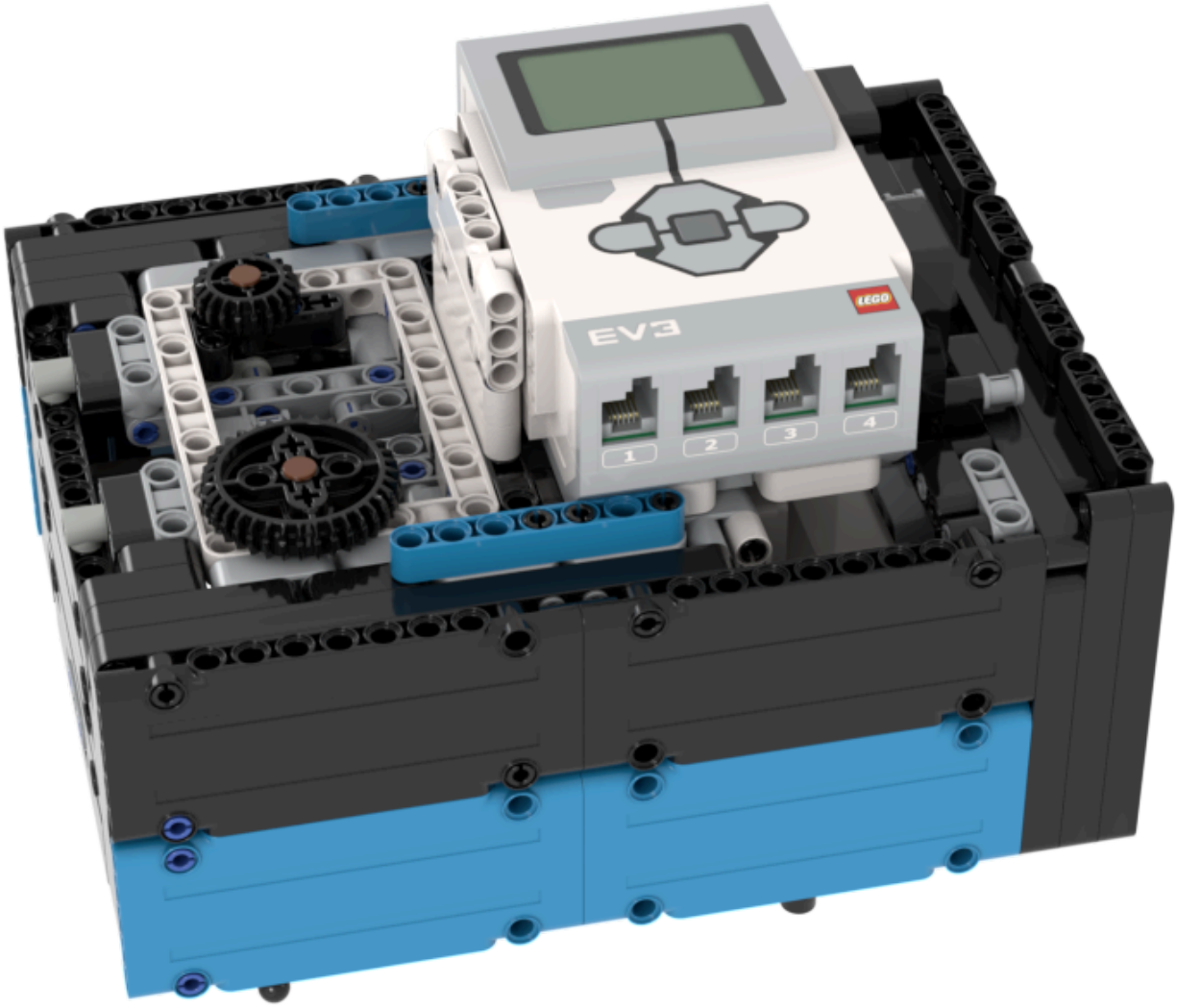
As a bonus for this documentation, I will present my own implementation of `trapezoidal profiles` for acceleration limiting, which form the basis of the library's functionality.

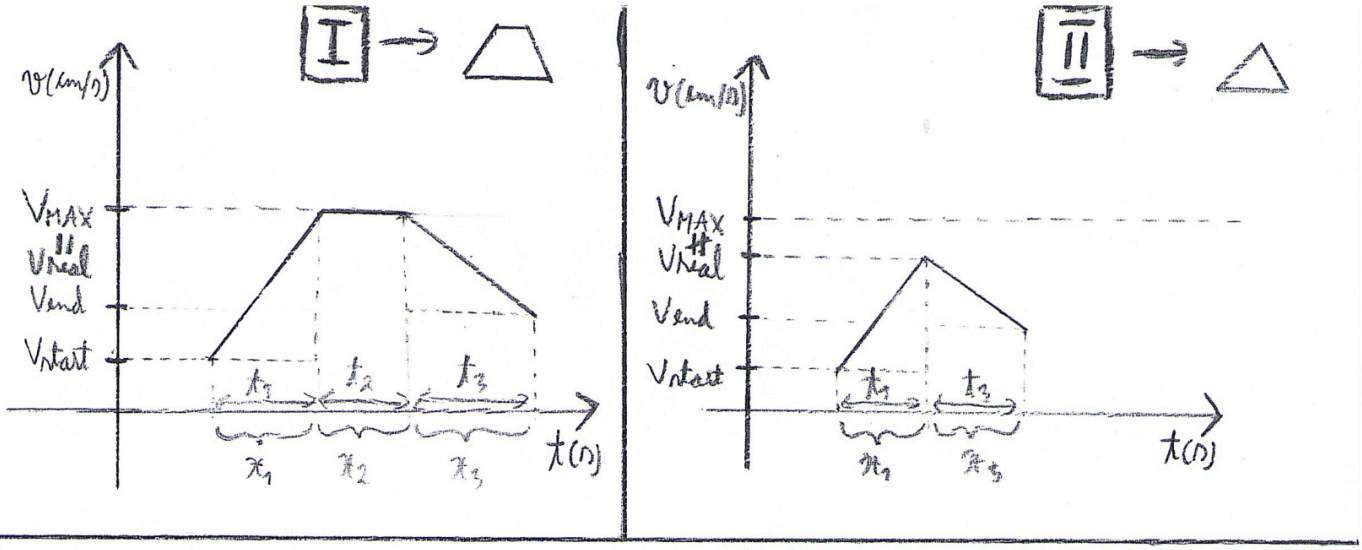
Given the distance the robot needs to travel, the target speed, acceleration, deceleration, initial speed, and final speed, we must analyze each **phase** of the trajectory: the **acceleration phase**, the **constant speed phase**, and the **deceleration phase**. To achieve this, we need to determine the duration of each phase using basic `kinematic` formulas.

Once we know when to transition to the next phase, we can calculate the robot's speed for any millisecond of the trajectory, ensuring smooth and efficient motion. Implementing this algorithm

allows the robot to adhere to all acceleration and deceleration constraints imposed.

There are `2 cases` to consider: when the profile is trapezoidal or triangular. We calculate the time required to reach the desired speed and the time needed to reduce the speed to zero. We then determine the distance covered during these periods to see if we exceed the total allocated distance. If we do not exceed the distance, the profile is trapezoidal, and we only need to calculate the constant speed portion. If we exceed the distance, the profile becomes triangular, and we will need to recalculate the maximum achievable speed and then adjust the remaining times.





STIM: * ACC, DEC; ACC > 0, DEC < 0

- V_{MAX} - viteza maximă a robotelului, din punct de vedere fizic ; $V_{MAX} > 0$
- Vstart - viteza inițială a robotului ; poate fi > sau < decât V_{MAX} (dacă nu intrăm limitările fizice) ; $V_{start} > 0$
- Vend - viteza finală a robotului. Similară cu Vstart
- x - distanța totală. Poate fi < 0, caz în care profilul se oglindește față de OX

TREBUIE SĂ AFLĂM:

- x_1, x_2, x_3, unde $x_1 + x_2 + x_3 = x$, reprezentând distanțele parcurse în fiecare fază.
- t_1, t_2, t_3, unde $t_1 + t_2 + t_3 = t$ (timpul total), reprezentând durata fiecărei faze.

(1) $\begin{cases} x > 0 \Rightarrow x = x, \text{ reverse} = \text{FALSE} \\ x < 0 \Rightarrow x = |x|, \text{ reverse} = \text{TRUE} \end{cases}$

$\begin{cases} V_{start} < V_{MAX} \Rightarrow ACC = ACC \\ V_{start} > V_{MAX} \Rightarrow ACC = -DEC \end{cases}$

$\begin{cases} V_{end} < V_{MAX} \Rightarrow DEC = DEC \\ V_{end} > V_{MAX} \Rightarrow DEC = -ACC \end{cases}$

(2) $t_1 = \frac{|V_{MAX} - V_{start}|}{ACC}$; $t_3 = \frac{|V_{MAX} - V_{end}|}{-DEC}$

$x_1 = \frac{|V_{MAX} - V_{start}|}{2} \cdot t_1 + V_{start} \cdot t_1 = t_1 \left(\frac{|V_{MAX} - V_{start}| + 2V_{start}}{2} \right)$

$x_3 = \frac{|V_{MAX} - V_{end}|}{2} \cdot t_3 + V_{end} \cdot t_3 = t_3 \left(\frac{|V_{MAX} - V_{end}| + 2V_{end}}{2} \right)$

(3) $x_1 + x_3 \leq x$ → trapez

$x_2 = x - (x_1 + x_3)$

$t_2 = \frac{x_2}{V_{MAX}}$

(4) $x_1 + x_3 > x$ → triunghi $t_2 = 0$, evident

$x_1 + x_3 = x$, $x_2 = 0$ (1)

$x_1 = t_1 \left(\frac{V_{MAX} - V_{start} + 2V_{start}}{2} \right) = \frac{V_{MAX} - V_{start}}{2ACC} \cdot \frac{V_{MAX} + V_{start}}{2} = \frac{V_{MAX}^2 - V_{start}^2}{2ACC}$ (2)

$x_3 = \frac{V_{MAX}^2 - V_{end}^2}{-2DEC}$, analog (3)

(1), (2), (3) $\Rightarrow \frac{V_{MAX}^2 - V_{start}^2}{2ACC} - \frac{V_{MAX}^2 - V_{end}^2}{2DEC} = x$

$V_{MAX} = V_{real} = +\sqrt{\frac{2 \cdot x \cdot ACC \cdot DEC + DEC \cdot V_{start}^2 - ACC \cdot V_{end}^2}{DEC - ACC}}$

Credits:

- *libraries used: pygame, matplotlib, pybricks, memory_profiler.*
- *the photo of the robot: studio 2.0*
- *documentation generator: md2pdf*
- *design made with: illustrator*
- *inspiration: roadrunner FTC*
- *font: graffitiyouthregular*
- *images uploaded on: imgbb*
- *fields: reddit*

v. 0.0.4-alpha