Colegiul Național 'Gheorghe Șincai'
Baia Mare

PYTH FINDER

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

Autor:
Contraș Adrian

2024

Cuprins

DOCUMENTAȚIE

Creator: Contraș Adrian 🎧

Instalare

Înainte să începem, asigurați-vă că aveți instalate pe dispozitiv:

- o versiune de `Python` mai mare de 3.10 (ultima versiune e recomandată);
- `pip` (pip3 e pentru versiunile 3.x, dar pip funcționează la fel);
- `font-ul` echipei mele (Omega Core) 'graffitiyouthregular' (folosit în cadrul interfeței);

Acum, pentru a instala librăria, trebuie doar să introduceți următoarea comandă în Command Prompt sau în terminalul de la Visual Studio:

```
pip install pythfinder
```

sau (pe cele mai multe dispozitive):

```
pip3 install pythfinder
```

Descriere

`PythFinder` a fost creat pentru a îmbunătăți procesul de planificare a mișcării autonome a roboților care participă în competiția First Lego League (FLL).

Echipele care participă în această competiție adesea folosesc `block-based programming`, din cauza lipsei de documentație detaliată pentru limbajele `Python / MicroPython` în acest scop. Însă, această metodă poate compromite fiabilitatea și flexibilitatea programelor.

Având în vedere acest aspect, am ales MicroPython ca limbaj de programare pentru cărămida mea de tip `EV3`. În timpul sezonului Masterpiece am experimentat cu mișcări calculate în timp real și am constatat că este o abordare **mult prea lentă** pentru utilizarea competitivă. Prin urmare, am îndreptat atenția către mișcarea precalculată (cunoscută și sub numele de feedforward control).

Din cauza limitărilor de performanță ale procesoarelor `LEGO®` permise în competiție, am hotărât să dezvolt un script care să preia această sarcină. Acest script rulează pe un computer extern, asigurând astfel o viteză de procesare mai mare.

Așadar, am creat o `unealtă de generare a traiectoriilor`, care rulează local pe dispozitivul vostru. Această unealtă generează un fișier .txt care conține toate informațiile necesare pentru ca robotul să imite mișcările dorite. Tot ce trebuie să faceți este să copiați textul generat în folderul de cod al robotului, iar acesta va fi automat citit la inițializare.

În codul robotului, găsiți o versiune simplificată a funcțiilor de 'following' din librărie, alături de funcții pentru reconstrucția traiectoriilor generate în fișierul `.txt`.

Din cauza acestei formalități, robotul necesită aproximativ 2 - 5 minute pentru a încărca toate datele, în cazul unui cod conceput pentru a realiza un punctaj maxim în concurs (asumând 7-8 traiectorii diferite). În tot acest timp, codul nu va putea fi rulat. Evident, timpul necesar pentru citirea datelor depinde de cantitatea acestora, care poate fi manipulată de utilizator în mai multe moduri, detaliate mai târziu. **Recomand să porniți codul cu cel puțin 5 minute înainte de meci.**

V-ați putea întreba, '*de ce ar fi abordarea asta mai bună?*'. Răspunsul constă în **consistență**. Această librărie folosește tehnici găsite în sistemele de control din robotica industrială, oferind îmbunătățiri semnificative în precizie prin utilizarea profilurilor de limitare a accelerației, acțiuni multithreading pentru accesarea mai multor motoare în paralel și multe altele.

Este un preț mic pentru a avea unul dintre cele mai fiabile programe autonome din competiția FLL.

Pentru a clarifica, această librărie **NU este restricționată la utilizarea exclusivă cu EV3**, chiar dacă a fost dezvoltată inițial pentru acest tip de cărămidă. Deoarece hardware-ul este separat de aceasta, chiar și roboții de tip `SPIKE PRIME` sau `NXT` pot beneficia de fișierul .txt generat.

În prezent, am implementat o soluție plug-and-play **DOAR** pentru cărămizile EV3, un pythfinder-quick-start. Pentru alte tipuri de cărămizi, va fi necesară o implementare personalizată a citirii și utilizării datelor generate. De asemenea, recomand să rulați codul pentru toate traiectoriile într-un singur program, pentru a evita timpul de așteptare de la încărcarea traiectoriilor în timpul meciului.

Pe GitHub există numeroase librării pentru **motion profiling**, dar nu am găsit una adaptată în mod special pentru FLL. Misiunea mea este să revoluționez programarea pentru această competiție, așa că am elaborat un plan. Acest plan a devenit realitate cu **PythFinder**!

De ce e PythFinder unic?

- Este prima librărie de motion profiling pentru FLL din lume \o/;
- Spre deosebire de roadrunner (inspirația pentru acest proiect), nu se instalează direct pe robot, rezultând în calcule mai rapide;
- Simulatorul, constructorul de traiectorii și generatorul sunt împachetate într-o singură librărie, menținând un aspect prietenos, similar cu cel al roadrunner-ului;
- Logica de bază din spatele implementării este complet diferită față de ce am găsit online;
- Permite utilizatorului să conducă manual robotul în simulator cu un controller;
- Generează toate valorile traiectoriei necesare într-un fișier '.txt', pentru a fi încărcate în robot (o soluție inovativă care nu a fost utilizată până acum);
- Permite utilizatorului să schimbe interfața fără a schimba codul, doar din meniul interfeței;
- Permite vizualizarea grafică a vitezei și accelerației pentru o mai bună înțelegere a comportamentului robotului;

Planuri pentru viitor

- Acesta este doar începutul librăriei mele. Am făcut 'un pic din toate' pentru a arăta un concept de bază. Planurile viitoare se rezumă la **îmbunătățiri treptate** în fiecare aspect al simulatorului (interfață, meniu, sintaxă, generare de traiectorii și compatibilitatea cu alți roboți);

- Abstractizarea și lăsarea spațiului pentru îmbunătățiri sunt practici esențiale pentru a realiza un progres continuu. Așteptați-vă să lansez versiuni îmbunătățite din când în când. De asemenea, o secțiune de **feedback** este deschisă pe GitHub-ul oficial al echipei pentru a semnala orice probleme sau implementări dorite, astfel încât să știu cum să abordez viitoare update-uri;

- Mai intenționez să dezvolt un `quick-start` pentru fiecare cărămidă legală în FLL. Cu toate acestea, în prezent, nu am acces la alte tipuri de cărămizi în afară de EV3 pentru a putea efectua teste și implementări.

  Cu toate acestea, am avut plăcerea de a intra în contact cu echipa **Arra**, una dintre cele mai prestigioase echipe din cadrul competiției. Formată din șase tineri entuziaști din Pitești, a avut onoarea de **a reprezenta România la campionatul mondial din Houston** în ultimii 2 ani, obținând rezultate remarcabile!

  Aceasta a fost profund impresionată de proiectul meu, exprimându-și dorința de a contribui la implementarea librăriei și pe celelalte cărămizi;

- Prin colaborarea strânsă între echipele Arra și Omega Core, îmi propun să promovez această librărie atât la nivel național, cât și internațional, utilizând rețelele noastre de socializare și diverse alte platforme de comunicare. Astfel, voi reuși să ating o audiență mai extinsă și să creez o comunitate mai vastă, care va contribui activ la îmbunătățirea continuă a librăriei;

Tehnologii utilizate

- Am ales să folosesc Python pentru acest proiect, deoarece este limbajul principal suportat de cărămizile LEGO®. Pentru a vizualiza robotul, am utilizat **pygame**, cel mai popular creator de jocuri în Python. Acesta avea deja toate funcționalitățile de bază de care aveam nevoie (import / export de imagini, suport pentru tastatură și controller și, bineînțeles, fereastra de afișare).

  Inițial, intenționam să încarc tot codul pe cărămidă, dar din cauza procesorului său vechi și a compatibilității limitate cu Python (cărămida are nevoie de MycroPython), a fost mai eficient să separ acest program de robotul propriu-zis;

- **matplotlib** a fost adăugat pentru a ajuta utilizatorul să vizualizeze valorile numerice într-un mod mai intuitiv;

- **memory_profiler** pentru a vizualiza grafic alocarea memoriei în timp (vedeți *Stabilitatea Librăriei*)

- Țin să menționez că, încă de la începutul proiectului, am avut ca obiectiv implementarea fiecărui aspect al librăriei de la **zero**. Aceasta include logica meniului de setări, fizica și matematica ce stau la baza mișcării robotului, generarea traiectoriilor, precum și relațiile dintre controller și interfață;

Stabilitatea librăriei

- Nu mă aștept să fie perfectă; de aceea am luat în considerare posibilele erori și am încercat să le rezolv înainte să apară. Cu toate acestea, este posibil să fi ratat unele cazuri extreme, știind că aceasta este o librărie bazată pe relația cu utilizatorul;
- Pentru input-urile cunoscute ca fiind incorecte, alertez utilizatorul prin excepții personalizate;
- Am implementat feedback pentru fiecare parte a compilării codului prin mesaje sugestive transmise în consola de output. Unele dintre ele sunt 'easter eggs', dar majoritatea informează utilizatorul;
- În scopul de a garanta siguranța maximă a utilizatorilor care utilizează librăria mea, am încărcat-o pe platforma oficială PyPi, cunoscută ca fiind site-ul dedicat de Python pentru gazduirea tuturor librăriilor. Prin urmare, aceasta a fost supusă unui proces riguros de validare, ceea ce a condus la eliminarea oricărui comportament potențial dăunător sau malițios;

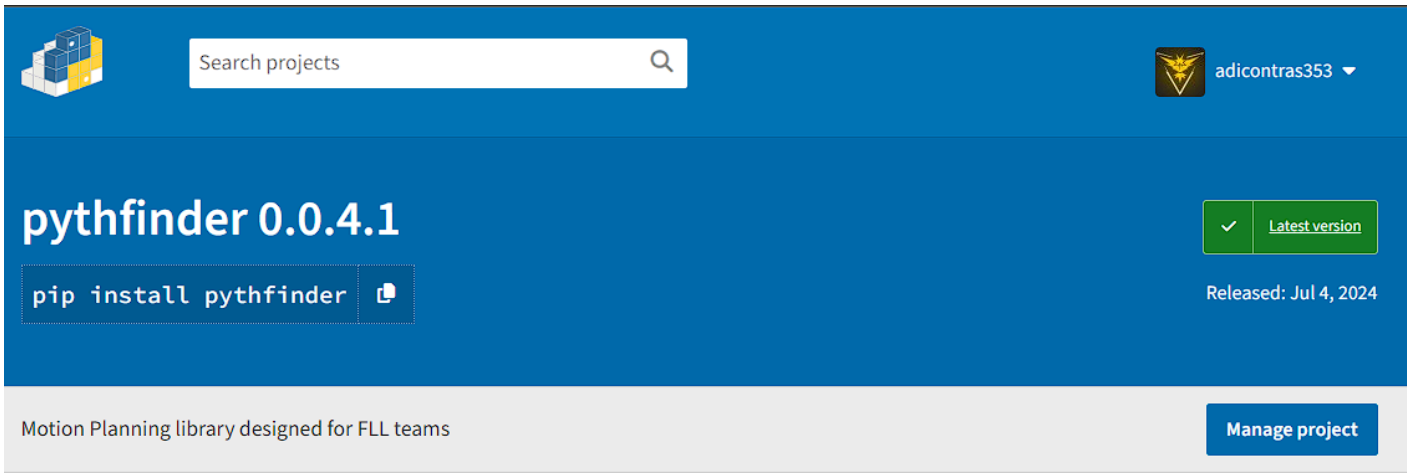


Fig. 1 Verificarea securității pe PyPi

- Securitatea utilizatorului primează în dezvoltarea unui software utilitar de **calitate**. Prin urmare, am adoptat o practică regulată de **analiză a alocării de memorie** în proiectul meu, asigurându-mă că nu există riscul unor scurgeri de memorie. Utilizând librăria specializată **memory_profiler**, am generat grafice care evidențiază evoluția utilizării memoriei în timp;

  Observațiile clare din aceste grafice, inclusiv cel prezentat în *Fig. 2*, indică o stabilizare a consumului de memorie după pornirea programului, confirmând astfel o **funcționare normală** și absența oricărei scurgeri de memorie;

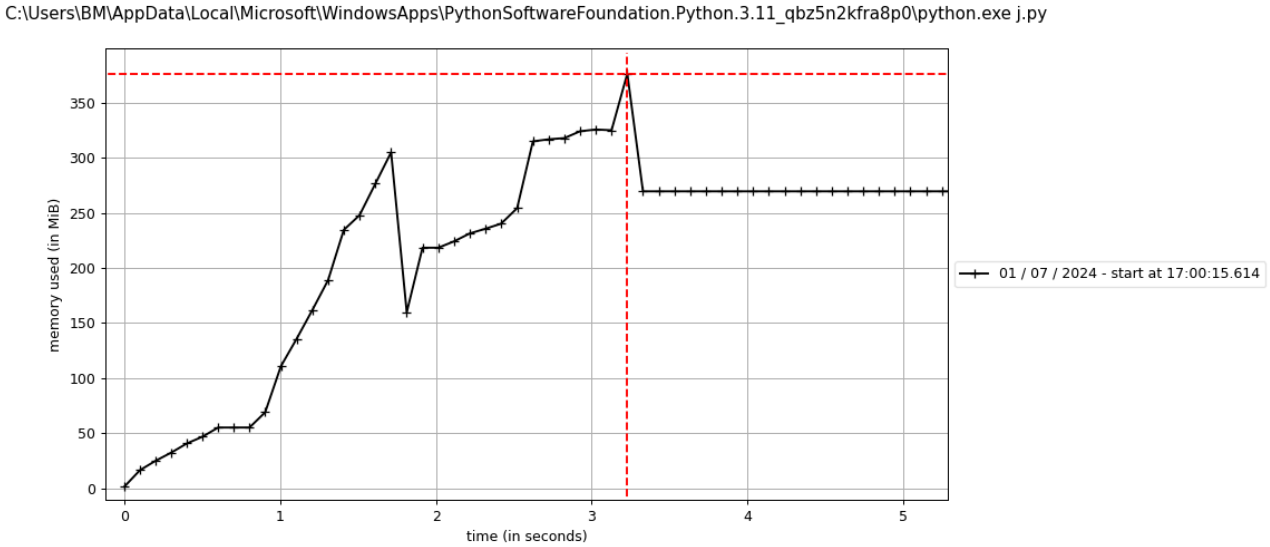


Fig. 2 Grafic al alocării memoriei în timp

- În plus față de riguroasele sale verificări, PyPI furnizează și un sistem de versionare robust, pe care l-am integrat și în proiectul meu. Am implementat și un fișier **CHANGELOG** pe GitHub, în care sunt detaliate modificările fiecărei versiuni, pentru a informa utilizatorii cu privire la îmbunătățirile aduse. Doar versiunile **stabile** sunt menținute publice;

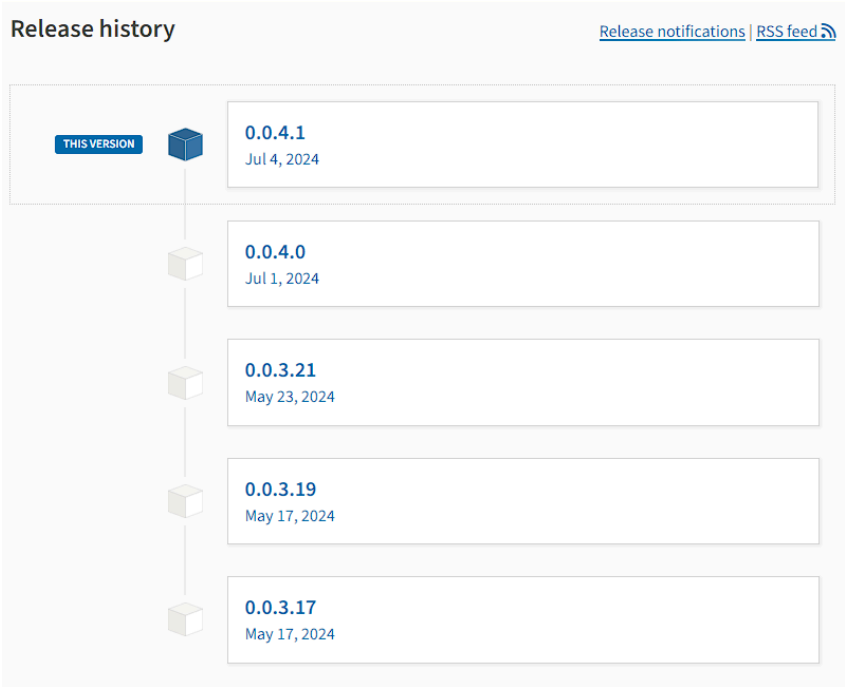


Fig. 3 Sistem de versionare

- În vederea gestionării eficiente și a distribuirii organizate a muncii mele în timp, am ales să utilizez platforma Trello. Aceasta îmi permite să monitorizez progresul realizat și să planific viitoarele acțiuni într-un mod sistematic;

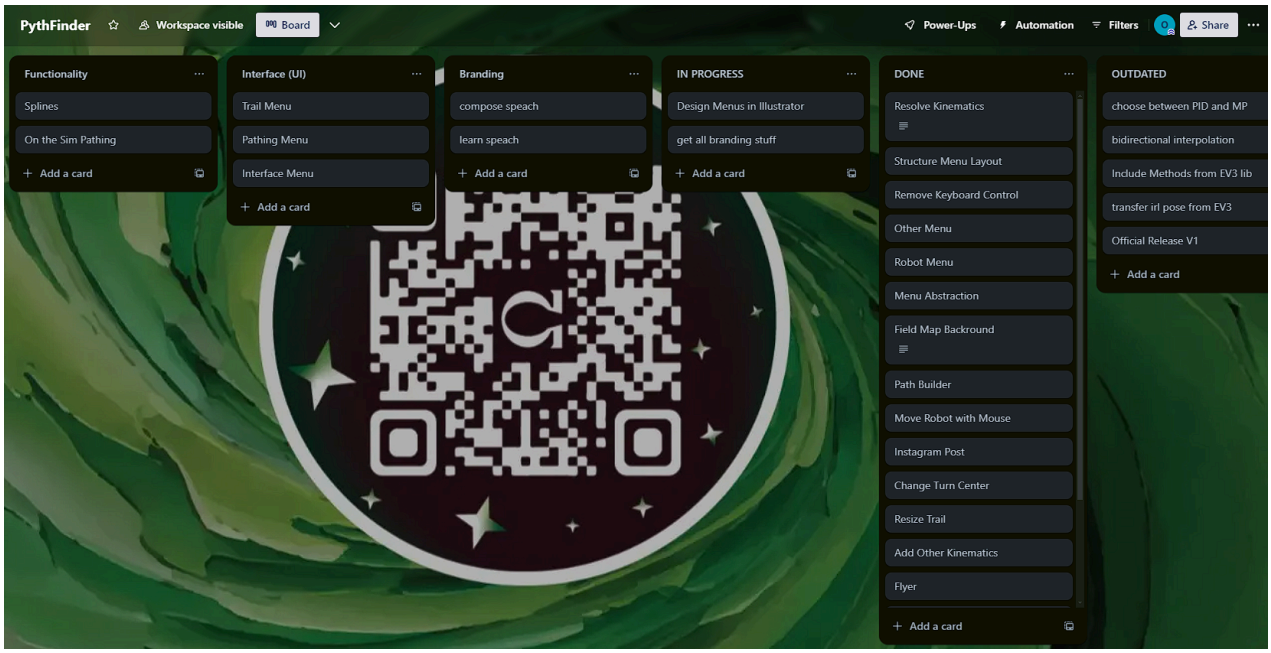


Fig. 4 Organizare folosind Trello

Utilizare

Pentru a începe să utilizați această librărie, pur și simplu creați un nou fișier Python și importați librăria:

```
import pythfinder
```

Crearea unui Robot

Pentru a activa simulatorul și toate funcționalitățile acestuia, trebuie să creați un obiect de tip `Simulator`. Această clasă integrează toate componentele într-un singur centru de control, gestionând afișarea ferestrei, conectarea joystick-ului și alte funcții.

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

Astfel veți crea un simulator cu constante *implicite*. Pentru a le suprascrie, creați un obiect '*Constants*' cu valorile dorite și transmiteți-l constructorului:

```
# transmiteți valorile aici
custom_constants = pythfinder.Constants(...)

sim = pythfinder.Simulator(custom_constants)
```

De fiecare dată când rulați simulatorul, acesta va începe cu setul de constante transmise de voi. Veți învăța o altă modalitate de a modifica constantele în secțiunea *Setările Interfeței*.

În cele din urmă, afișați simularea:

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

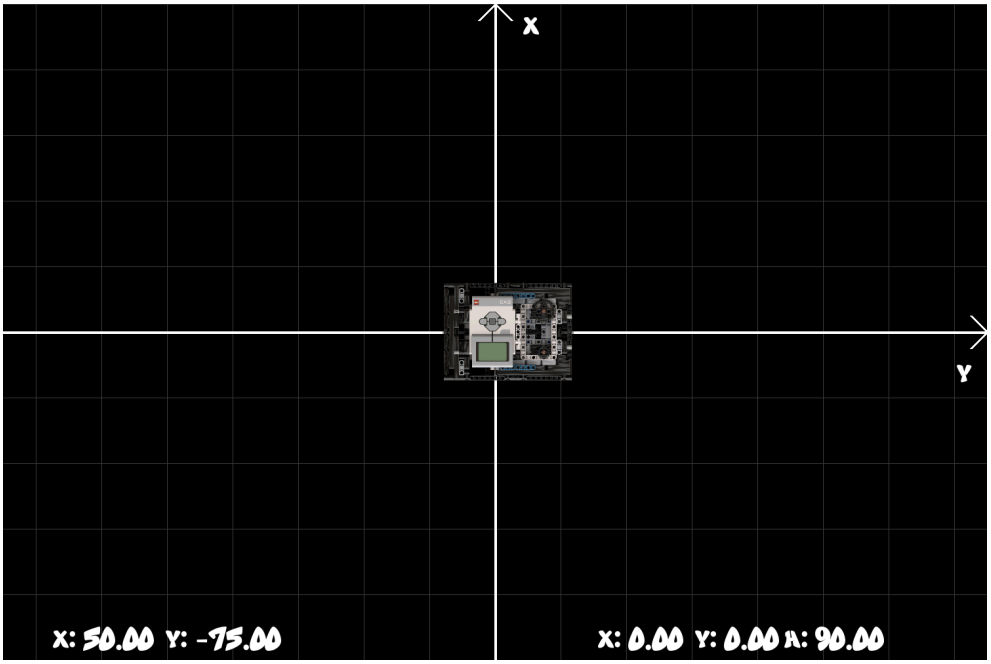


Fig. 5 Vizulizare inițială a inferfeței

Vă atrag atenția că sistemul de coordonate adoptat este neobișnuit față de cel convențional. În partea inferioară a ferestrei de simulare sunt afișate două seturi distincte de coordonate. Cele amplasate în stânga indică poziția cursorului, în timp ce cele din dreapta reflectă poziția efectivă a robotului în raport cu mediul simulat.

Codul rulează până când ieșiți din fereastra simulatorului. Conectarea unui *controller acceptat* vă va permite să vă mișcați liber pe teren.

Control prin joystick

PythFinder se bazează pe funcționalitățile oferite de librăria pygame, de la care împrumută suport pentru controllerele de XBOX, PS4 și PS5.

Controllerul poate fi conectat fie prin **USB**, fie prin **Bluetooth**. Simulatorul îl va recunoaște automat; în caz contrar, va genera o eroare legată de compatibilitate.

Butoanele utilizate pentru manipularea simulatorului sunt următoarele:

*(ordinea butoanelor este: **ps4 / xbox**)*

- `△ / Y` -- merge înainte / înapoi (când 'field centric' este activat);
- `□ / X` -- intră / iese din meniul de setări al interfeței;
- `○ / B` -- resetează poziția robotului la origine / apasă butoanele (când meniul este activat);
- `X / A` -- arată / ascunde urma;
- `left bumper` -- șterge urma / setează valorile implicite (când meniul este activat);
- `right bumper` -- când este apăsat intră în modul de selecție;
- `D-pad` -- mișcă butonul selectat din meniu / selectează orientarea robotului (când modul de selecție este activat);
- `left joystick` -- controlează viteza liniară a robotului + viteza unghiulară (când 'field centric' este activat);
- `right joystick` -- controlează viteza unghiulară (**DOAR** când 'field centric' este dezactivat);
- `options / start` -- face o captură de ecran (găsită în folderul „Screenshots” din locația librăriei instalate local);

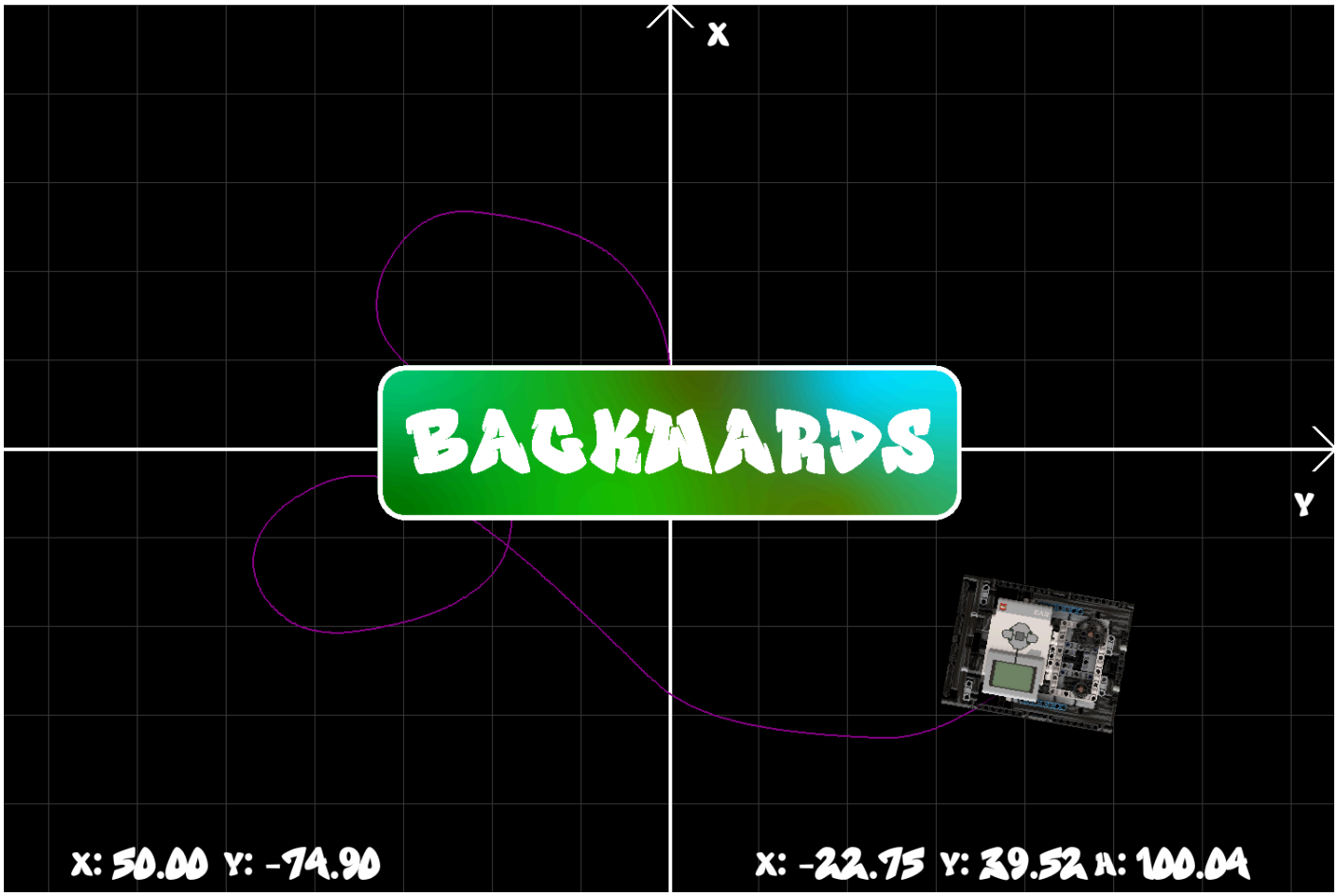


Fig. 6 Mișcare folosind controller-ul

La apăsarea butoanelor, simulatorul *alertează utilizatorul* cu privire la modificările efectuate, utilizând atât o fereastră pop-up cu imagini integrate realizată integral de mine, cât și mesaje text afișate în terminal.

Urma lăsată de robot este gestionată printr-o logică implementată separat, permițând ajustarea culorii și grosimii acesteia. Este formată din liniile trasate între toate pozițiile consecutive ale robotului. În plus, valorile *implicite* sunt configurate pentru ca după o perioadă scurtă de inactivitate, urma să înceapă să se **șteargă** treptat în mod **automat**, proces care încetează atunci când utilizatorul mișcă din nou robotul.

Urma nu este o simplă colecție de puncte, ci este împărțită în **segmente**. Acestea sunt delimitate de distanța dintre două poziții diferite, valoare ce poate fi **ajustată** de utilizator. Astfel, fiecare segment poate avea **propria** sa culoare și grosime, ceea ce este extrem de util în cazul în care se dorește desenarea unei **scheme complexe**, adesea reprezentând strategii pe terenul de joc.

Feedback?

Voi întrerupe lectura documentației tehnice pentru a vă spune feedback-ul pe care l-am primit de la oamenii cărora le-am prezentat proiectul:

- prezentând în fața clasei la ora de fizică, **profesorul** și **colegii** au afirmat că '`Este evident că s-a depus o muncă considerabilă pentru a atinge acest nivel de detaliu și performanță.`'. Un fost elev al liceului a susținut această idee: '`M-am uitat un pic, pare mega profi`';
- vorbind cu membrii altor echipe de robotică din **Satu Mare** (Perpetuum Mobile), **Timișoara** (Cybermoon), **Alba-Iulia** (Xeo), au fost profund impresionați. Un membru din echipa Robocorns din **Baia Mare** chiar mi-a *sugerat* să implementez redimensionarea ecranului, ceea ce am făcut imediat după. Cum spuneam anterior, **Arra** a fost încântată chiar să ajute;

Preseturi

O **inovație** remarcabilă adusă de această librărie este caracteristica numită `preseturi`. Acestea permit utilizatorului să transforme în întregime aspectul interfeței, configurarea robotului și tipul de șasiu printr-o simplă apăsare de buton. Utilizatorul beneficiază butoanele numerotate pe tastatură de la `1` la `9`, fiecare destinat adăugării unui set distinct de constante care ajustează simularea în diverse moduri. Butonul cu cifra `0` are rolul de a reveni interfața la setările *implicite*.

Peste aceste opțiuni predefinite, utilizatorul are posibilitatea să își creeze **propriile** preset-uri, personalizate în funcție de nevoile și preferințele individuale. Această funcționalitate adaugă un nivel suplimentar de **flexibilitate** și **control** asupra modului în care sunt configurate `interfața`, `comportamentul robotului` și `parametrii` simulatorului.

In mod prestabilit, butonul 1 afișează **cel mai recent** teren din competiția `FLL` :

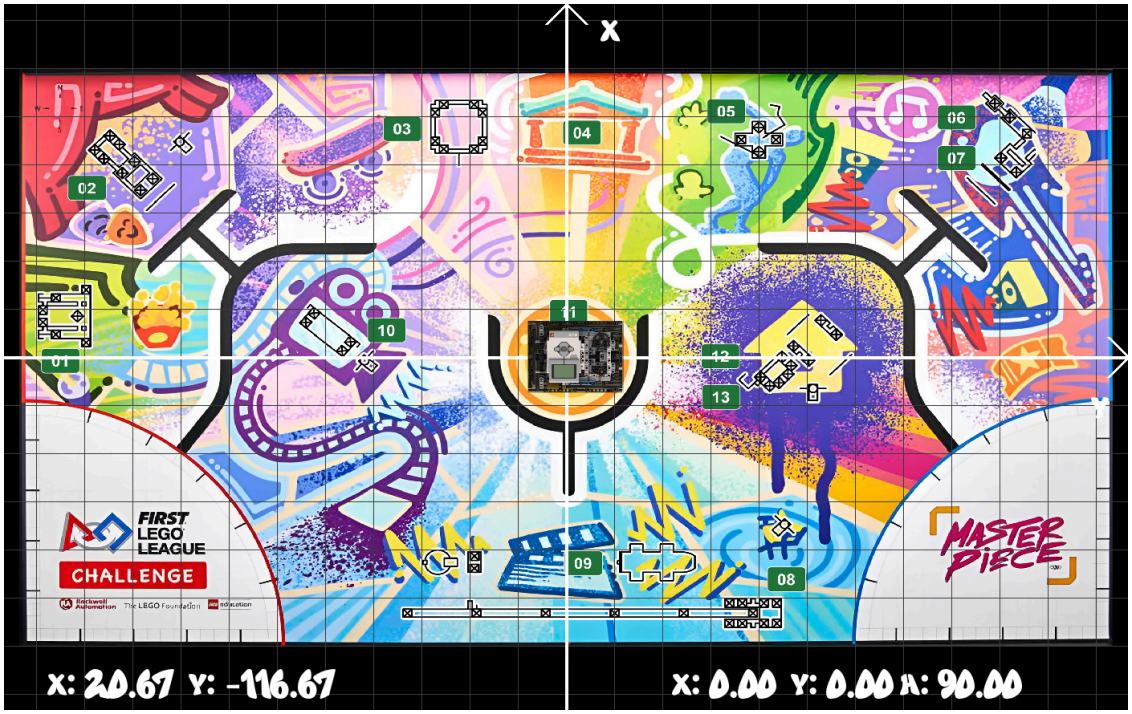


Fig. 7 Preset pentru FLL

Și butonul 2 afișează **cel mai recent** teren din competiția First Tech Challenge (`FTC`):

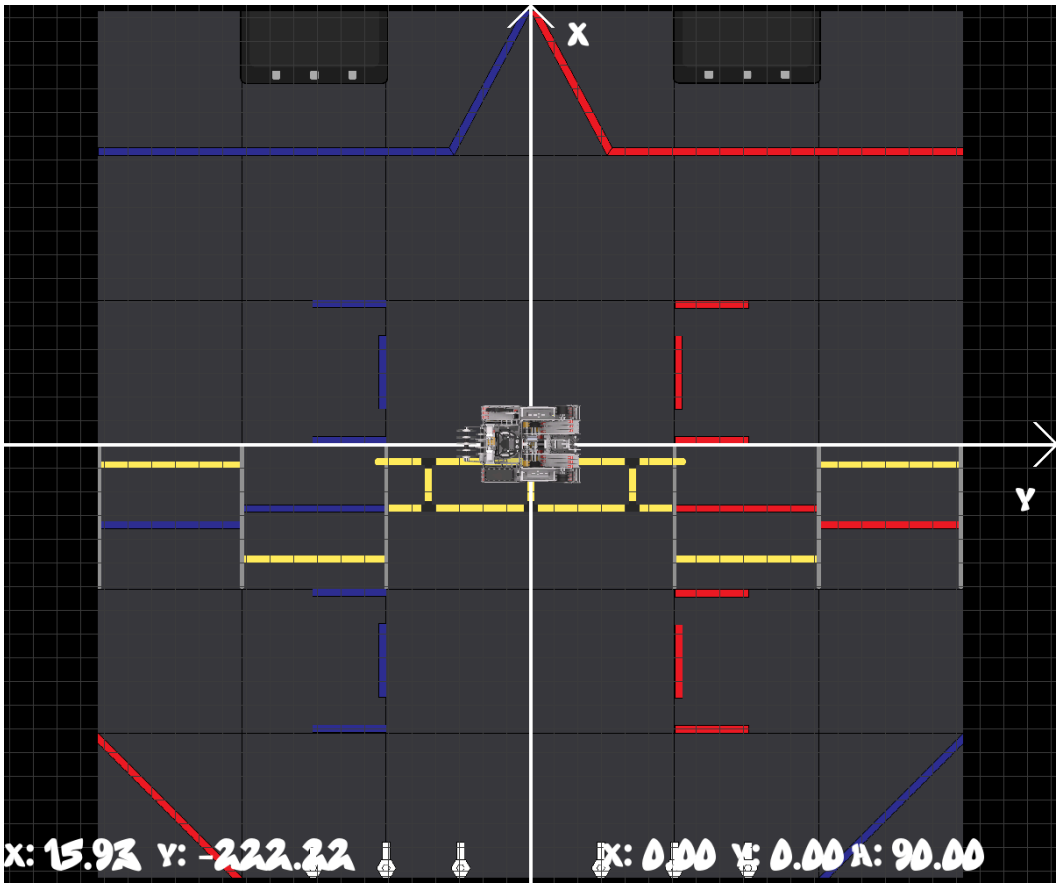


Fig. 8 Preset pentru FTC

Utilizarea traiectoriilor

Ce sunt traiectoriile?

In primul rând, definim un set specific de informații despre robot (poziția, viteza, distanța parcursă) ca o **stare de mișcare**.

Mai multe stări de mișcare prezentând o anumită similaritate poartă denumirea de **segment de mișcare**. Segmentele de mișcare sunt ulterior categorizate după gradul lor de *complexitate*. **Primitivele** reprezintă mișcări cu un singur grad de libertate (1D), cum ar fi rotația pură, mișcarea liniară pură sau chiar staționarea (așteptarea). Aceste primitive sunt folosite pentru a construi segmente **complexe**, care conțin două sau mai multe primitive și caracterizează mișcări cu două sau trei axe de libertate, fiind în principal destinate roboților *omnidirecționali*.

În final, toate segmentele de mișcare și elementele auxiliare care îndeplinesc diverse funcții (de exemplu, activarea unui motor), cunoscute sub numele de marcatori, formează o **traiectorie**.

Cum creez traiectorii?

Traiectoriile sunt construite folosind clasa '**TrajectoryBuilder (constructor de traiectorii)**'. Aceasta ia un obiect de tip *Simulator* ca parametru obligatoriu și, ca parametri opționali, o `poziție de început` și `presetul` pe care să ruleze. În mod implicit, poziția inițială este la originea sistemului cartezian.

Constructorul oferă metode intuitive pentru a crea o traiectorie simplă și dorită, inclusiv funcții de **mișcare** adaptate. Aceste funcții sunt concepute pentru a satisface atât nevoile roboților omnidirecționali, cât și ale celor cu mișcare unidirecțională.

Constructorul recunoaște tipul de șasiu utilizat și **ajustează** funcțiile transmise de utilizator, întrucât unele tipuri de șasiuri pot avea restricții fizice care le fac **imposibile**. În mod implicit, șasiurile non-holonome sunt *tangente* la traiectorie, în timp ce pentru cele holonome se oferă opțiunea de a `interpola` orientarea.

O listă cu toate funcțiile de mișcare:

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

Ce sunt marcatorii?

Aceste funcționalități pot fi integrate alături de **marcatori**, facilitând gestionarea sarcinilor paralele care nu sunt dependente de mișcarea roboților, folosind tehnici de multithreading. Marcatorii pot fi configurați să activeze după o anumită perioadă de **timp** sau **distanță**, fie **relativ** la ultima funcție de mișcare, fie **absolut** în raport cu începutul traiectoriei.

Librăria include și tipuri speciale de marcatori:

- `întrerupătoare`: întrerupe continuitatea traiectoriei la momentul specificat, în funcție de timp / distanță. Imaginați-vă întrerupătoarele ca fiind frânele bruște făcute de o mașină;
- `constrângeri dinamice`: vă permit să modificați porțiuni din traiectorie pentru a rula la viteze diferite, fără a sacrifica continuitatea;

O listă cu toți marcatorii:

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

`Întrerupătoarele` și `Constrângerile` sunt **strict** relative, deoarece am observat că utilizatorilor le este **dificil** să își imagineze la ce segment din traiectorie se aplică. Ele modifică cursul traiectoriei în sine, spre deosebire de marcatori care apelează funcții, putând afecta negativ construcția traiectoriei. Totuși, dacă utilizatorii vor **solicita**, voi reintroduce aceste funcționalități, deoarece au existat în prototipurile inițiale ale librăriei.

Marcatorii pot include și valori **negative**, care sunt interpretate ca fiind relative față de sfârșitul traiectoriei sau al segmentului de mișcare, în timp ce valorile **pozitive** sunt interpretate ca fiind relative față de începutul acestora.

Cum procesează simulatorul marcatorii?

Marcatorii sunt inițial **separați** în relativi și absoluti în funcție de segmentul traiectoriei. Cei specificați după distanță sunt transformați în marcatori absoluti, corectând distanțele *negative* în *pozitive*. Apoi, marcatorii sunt *sortați* după **prioritate** (constrângeri, întrerupătoare, funcții), **tip** (marcatori de timp la sfârșit), **semn** (negative la final) și **valoare**.

Lista sortată este divizată în grupuri distincte (constrângeri, întrerupătoare, funcții) și fiecare grup este procesat separat. Marcatorii de funcții sunt convertiți în marcatori absoluti, ajustându-se distanțele negative în pozitive și transformând distanțele în unități de timp. Astfel, se obține o listă finală exclusiv cu marcatori absoluti de timp și o traiectorie modificată conform acestora.

Un exemplu

După specificarea mișcării dorite, trebuie apelată funcția `.build()` pentru a calcula valorile traiectoriei.
Punând totul cap la cap, obținem:

```
# exemplu din codul echipei mele pentru prima traiectorie, din sezonul Masterpiece

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())
```

Vizualizarea Traiectoriei

După ce ați creat traiectoria, apelați funcția `.follow()` pentru a vedea codul în acțiune!

Această funcție ia ca parametru opțional un boolean, ce reprezintă modul de urmărire:

- `perfect` = simulatorul iterează prin fiecare stare de mișcare și afișează robotul în poziția precalculată. Pentru acest mod, puteți modifica și dimensiunea pasului în care este iterată lista. Dimensiune mai mare a pasului = iluzia unui robot mai rapid pe ecran.
- `real` = simulatorul atribuie vitezele calculate obiectului robot, care arată exact ca și cum ar rula în timp real. Acest mod este **recomandat** pentru a vedea exact cum s-ar comporta în realitate.

Ultimul parametru opțional este `wait`. Când acest boolean este setat la True, așteaptă până când simulatorul s-a încărcat complet pe ecranul utilizatorului înainte de a continua cu traiectoria. Acest lucru este util atunci când sunt setate urmărirea perfectă și un număr mare de pași pentru a nu rata începutul.

```
# valori implicite
PERFECT_STEPS = 10
PERFECT_FOLLOWING = True
WAIT = True

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

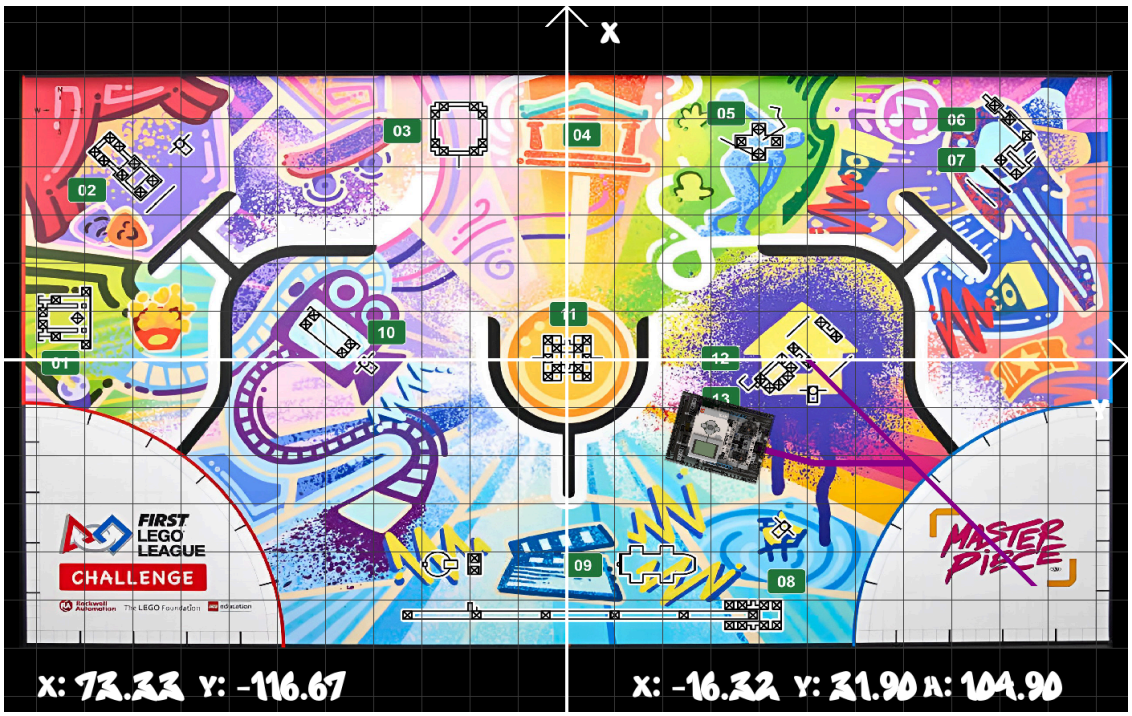


Fig. 9 Vizualizarea codului anterior, în modul autonom

Graficul Vitezei

Pentru a facilita înțelegerea conceptului de 'traiectorie', am implementat o metodă de vizualizare grafică ușor de utilizat a profilelor de mișcare. Aceasta este deosebit de utilă pentru echipele FLL, care includ mulți tineri abia introduși în lumea roboticii.

Cred cu adevărat că această librărie reprezintă una dintre cele mai bune modalități de a începe învățarea conceptelor **utilizate în industrie**, cu scopul de a ajuta și inspira viitorii ingineri și programatori!

Apelul funcției `.graph()` va afișa un grafic Matplotlib al **vitezei** și **accelerației** pentru roata stângă și roata dreaptă. Există și parametri opționali pentru a afișa fiecare valoare separat. În plus, utilizatorul poate alege dacă dorește viteza și accelerația roților sau a șasiului.

Un aspect interesant este parametrul `connect`. Implicit este setat la True, această variabilă desenează linii între puncte. Setarea lui la False arată discontinuitățile (în accelerație, deoarece viteza este optimizată pentru continuitate).

```
# valori implicite
CONNECT = True
VELOCITY = True
ACCELERATION = True

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

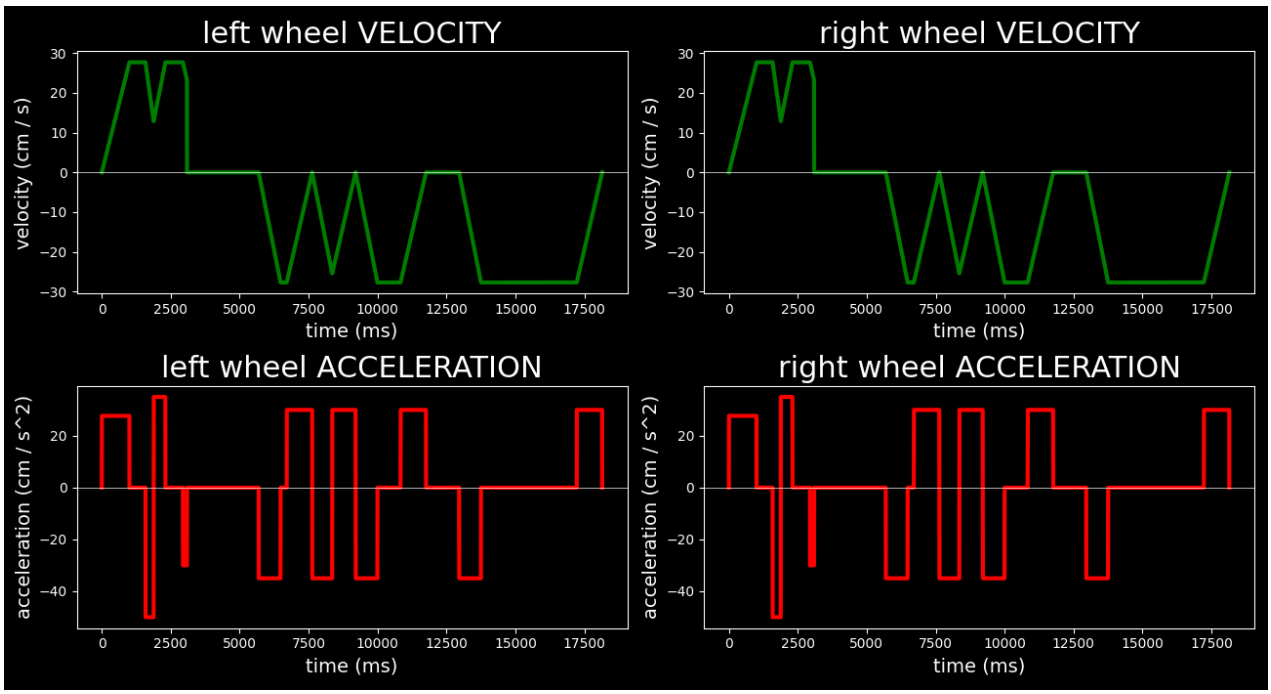


Fig. 10 Graficul generat cu ajutorul valorilor codului anterior

Generarea Vitezelor

Pentru a face robotul să se miște precum în simulator, va trebui să **transferați** datele printr-un fișier '.txt'. Acest lucru se realizează cu funcția '.generate()'. Doar treceți numele / calea fișierului text și dimensiunea pasului:

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

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

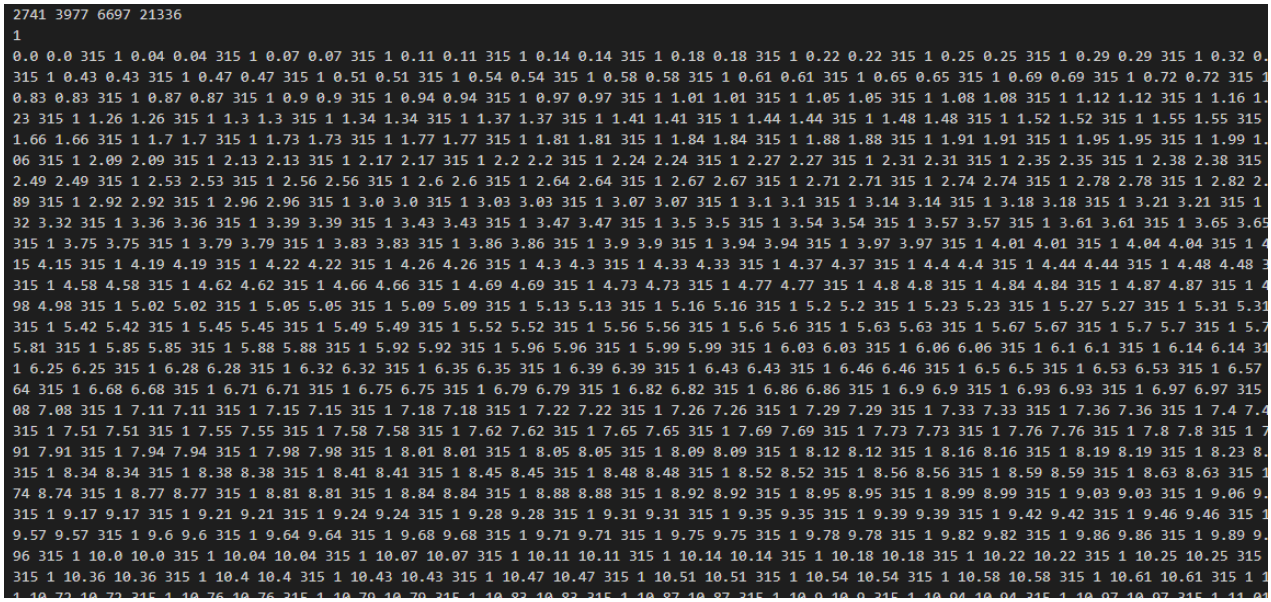


Fig. 11 Exemplu de valori generate pentru codul anterior, cu nr. de pași = 1

Prima linie generată include timpii în care trebuie marcatorii să acționeze, în ordine cronologică. Pe a doua linie se află numărul de pași pe care-i sare în inserarea valorilor în fișierul text, iar pe ultima line se alfă, în ordine, următorul set de date:

- primele **n** valori sunt vitezele pentru cele n roți ale robotului, sau vitezele pe axele X, Y și cea de rotație;
- următoarea valoare e unghiul la care se află robotul ;
- ultima valoare e numărului de apariții consecutive ale setului actual ;

Fiecare set de date reprezintă starea de mișcare într-o **milisecundă**, altfel fiecărei secvențe îi asociem un timp, după care va fi selectată. Acum puteți copia fișierul '`.txt`' și să-l încărcați în `quick-start` pentru a-l vedea rulând!

Setările Interfeței

Există două moduri principale în care vă puteți manipula mediul oferit de simulator prin constante.

Prima modalitate este, alături de preseturi, pur și simplu să treceți o nouă instanță de '**constante (Constants)**' atunci când creați obiectul de simulator, schimbând oricare dintre următoarele valori:

```
# constants.py -- simplificare

# toate valorile modificabile:
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):
    ...
```

După cum este descris în secțiunea Crearea unui Robot, aceste modificări vor fi aplicate automat la începutul simulării.

A doua cale este prin meniul din interfață (**NU ESTE COMPLET IMPLEMENTAT**), prin control cu joystick-ul. Aceasta este o modificare făcută 'pe loc' și se va reseta de fiecare dată când reporniți simulatorul.

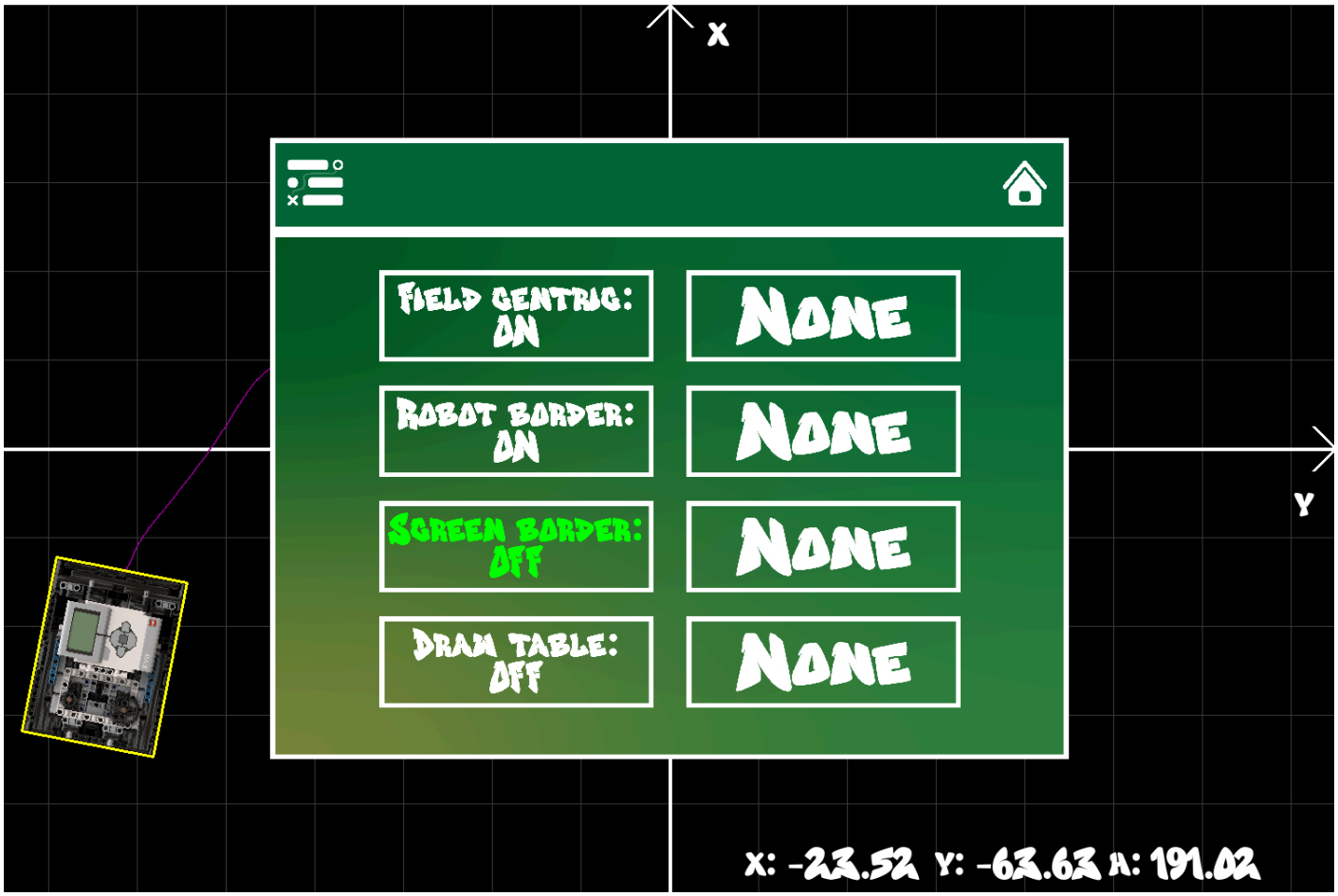


Fig. 12 Secțiunea 'Other' în meniul interfeței

Arhitectura Librăriei

PythFinder este structurat **modular** și încorporează multiple `pachete separate`, fiecare având funcționalitatea sa izolată de celelalte.

Pentru a asigura lizibilitatea codului, am optat pentru utilizarea **enum-urilor**.

Numele variabilelor sunt sugestive, reprezentând denumirile complete ale obiectelor pe care le reprezintă.

Am adoptat convențiile programatorilor Python, prefixând cu două simboluri underscore (`__`) funcțiile și variabilele care **NU** ar trebui accesate de utilizator sau din afara clasei în care se află, similar `funcțiilor private`.

Am împărțit procesele de calcul în multiple funcții, evitând astfel secvențe lungi de cod. Această abordare facilitează mentenanța și extinderea codului, asigurând totodată o mai bună organizare și structurare.

Toate aceste practici contribuie la crearea unui cod robust, ușor de înțeles și de utilizat.

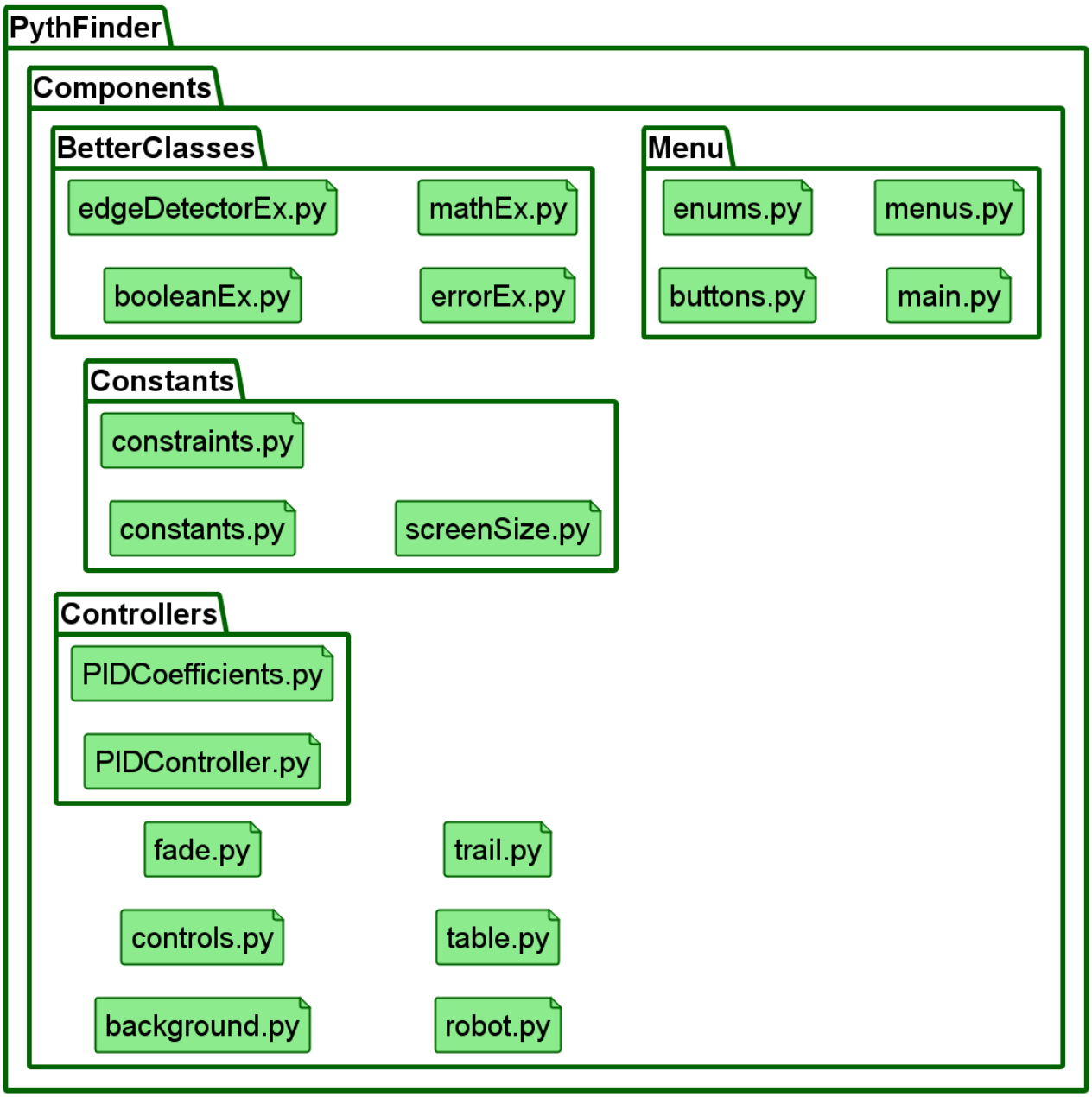


Fig. 13 Schema organizării fișierelor de la baza librăriei (doar o parte din fișiere)

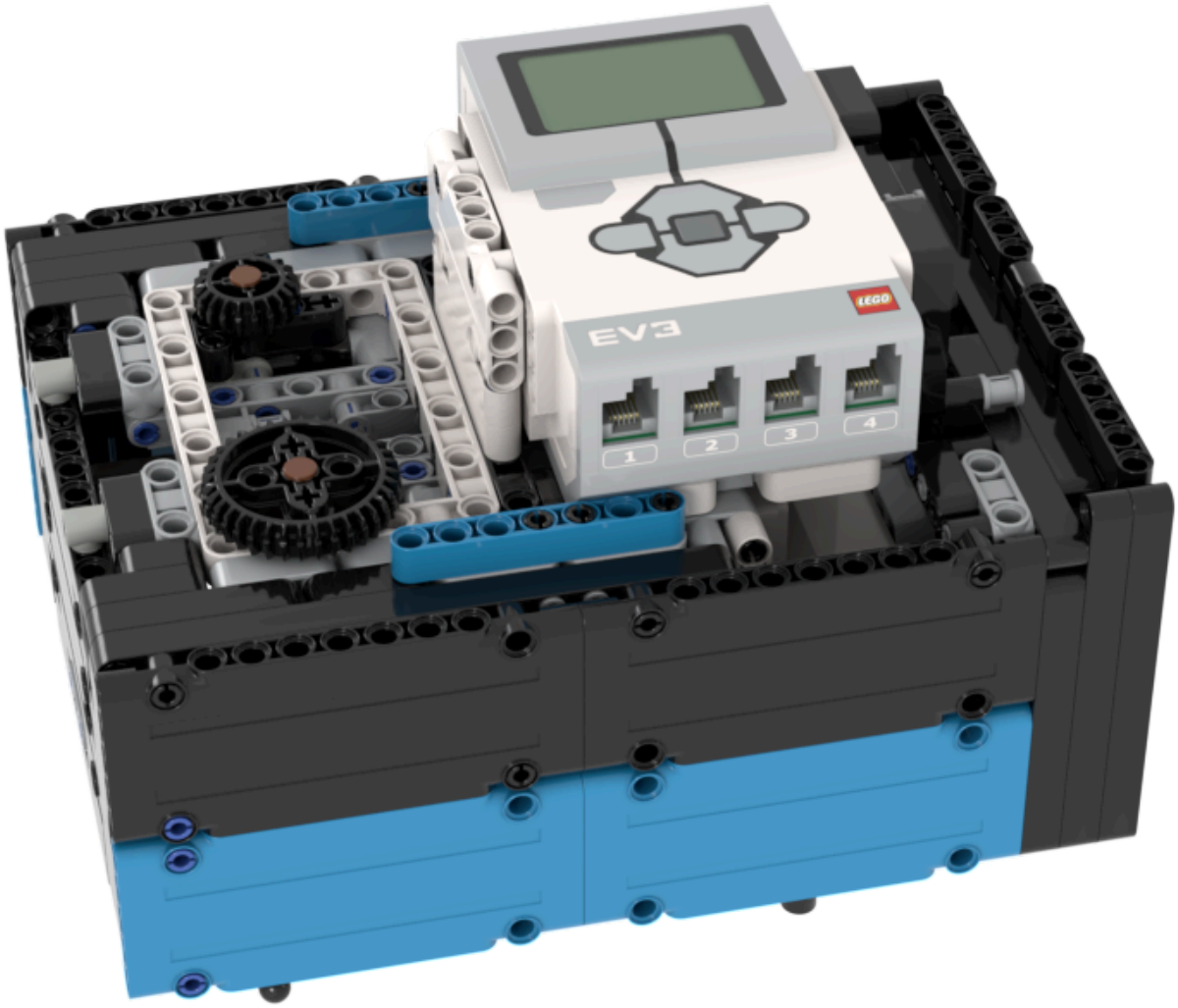
Profiluri Trapezoidale

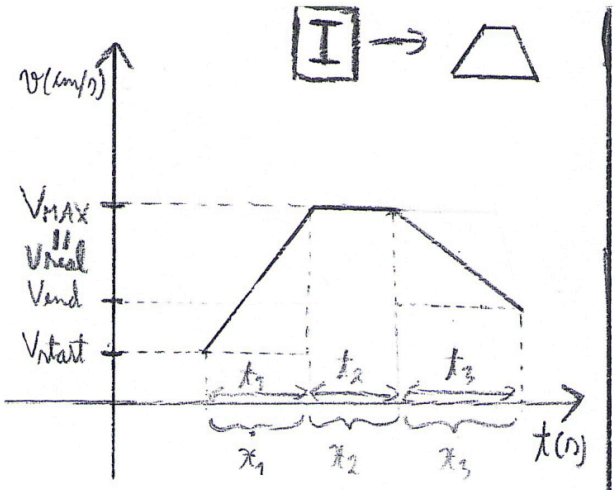
Ca bonus pentru această documentație, îmi voi prezenta propria implementare a `profilurilor trapezoidale` de limitare a accelerației, care stau la baza funcționalității librăriei.

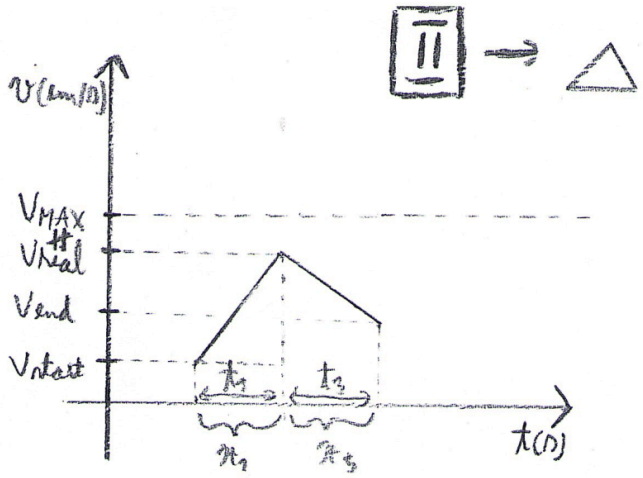
Cunoscând distanța pe care vrem să o parcurgă robotul, viteza la care vrem să ajungă, accelerația, decelerația, viteza inițială și viteza finală, trebuie să analizăm fiecare **secțiune** a traiectoriei: porțiunea de **accelerare**, cea de **viteză constantă** și cea de **decelerare**. Pentru aceasta, este necesar să determinăm durata fiecărei faze, folosind formulele de bază ale `cinematicii`.

Odată ce știm când să trecem la următoarea fază, putem calcula viteza robotului pentru orice milisecundă din traiectorie, asigurând astfel o mișcare fluidă și eficientă. Implementarea acestui algoritm permite robotului să respecte toate limitările de accelerație și decelerație impuse.

Astfel, există `2 cazuri`: când profilul are aspect trapezoidal sau aspect triunghiular. Calculăm timpul necesar pentru a ajunge la viteza dorită și timpul necesar pentru a reduce viteza la zero. Determinăm distanța parcursă în aceste perioade pentru a vedea dacă depășim distanța totală alocată. Dacă nu depășim, profilul este trapezoidal și mai trebuie să calculăm doar porțiunea de viteză constantă. Dacă o depășim, profilul devine triunghiular și va trebui să recalculăm viteza maximă posibilă și apoi restul timpilor.







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

- V_{MAX} - viteza maximă a robotelului, din punct de vedere fizic; $V_{MAX} > 0$
- V_{start} - viteza inițială a robotelului; poate fi > sau < decât V_{MAX} (dacă nu intrăm limitările fizice); $V_{start} > 0$
- V_{end} - viteza finală a robotelului. Similară cu V_{start}
- 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) $t_1 = \frac{|V_{MAX} - V_{start}|}{ACC}$; $t_3 = \frac{|V_{MAX} - V_{end}|}{-DEC}$

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

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

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

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

nu uităm de x_2

Credite:

- *librării folosite: pygame, matplotlib, pybricks, memory_profiler.*
- *poza robotului a fost făcută cu: studio 2.0*
- *imaginile au fost încărcate pe: imgbb*
- *generatorul documentației: md2pdf*
- *design-ul a fost făcut cu: illustrator*
- *imaginile terenurilor: reddit*
- *inspirație: roadrunner FTC*
- *font: graffitiyouthregular*

v. 0.0.4-alpha