VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

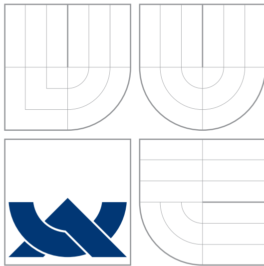
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ALGORITMY PRO ŘÍZENÍ POHYBU DVOUNOHÉHO ROBOTA

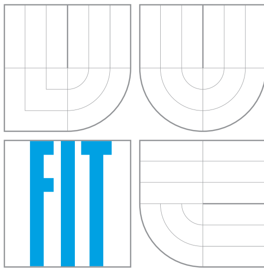
DIPLOMOVÝ PROJEKT
TERM PROJECT

AUTOR PRÁCE JAN POKORNÝ
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ALGORITMY PRO ŘÍZENÍ POHYBU DVOUNOHÉHO ROBOTA

ALGORITHMS FOR MOVEMENT CONTROL OF BIPEDAL ROBOT

DIPLOMOVÝ PROJEKT
TERM PROJECT

AUTOR PRÁCE JAN POKORNÝ
AUTHOR

VEDOUCÍ PRÁCE Ing. DAVID MARTINEK
SUPERVISOR

BRNO 2010

Zadání diplomové práce/10863/2009/xpokor37

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2009/2010

Zadání diplomové práce

Řešitel: **Pokorný Jan, Bc.**
Obor: Inteligentní systémy
Téma: **Algoritmy pro řízení pohybu dvounohého robota**
Algorithms for Movement Control of Bipedal Robot
Kategorie: Umělá inteligence

Pokyny:

1. Prostudujte problematiku simulací fyzikálního prostředí. Seznamte se s používanými algoritmy pro řízení pohybu dvounohých robotů. Seznamte se s principy evolučních algoritmů.
2. Navrhněte simulátor dvounohého robota s jednoduchým grafickým výstupem. Navrhněte algoritmy pro řízení pohybu tohoto robota. Zaměřte se především na algoritmy pro udržení rovnováhy a pohyb vpřed.
3. Implementujte navržený simulátor a demonstrujte v něm navržený algoritmus pro řízení dvounohého robota.
4. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu.

Literatura:

- Yamaguchi J., et al.: Development of a bipedal humanoid robot - control method of whole body cooperative dynamic biped walking, Proceedings of the 1999 IEEE International Conference on Robotics & Automation. Detroit, Michigan 1999. [Online] URL: http://www.cats.rpiscrews.us/~wenj/ECSE641S07/biped_humanoid.pdf
- Další dle pokynů vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Martinek David, Ing.**, UITS FIT VUT
Datum zadání: 21. září 2009
Datum odevzdání: 26. května 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato práce si klade za cíl použít k naučení chůze bipedálního robota softcomputingové metody. Robot je reprezentován virtuálním modelem. Ze začátku je rozebrána motivace a důvody ke zpracování tohoto tématu. Dále je navržen tvar robota, který se použije. Jsou popsány nezbytné knihovny, jakých simulátor simulace využívá. Dále je navržen systém algoritmů, které budou robota učit. Nejdůležitější z nich je SOMA, který je proto blíže popsán. Vzhledem k předpokládané výpočetní náročnosti je část práce věnována optimalizacím a zjednodušením. Jsou rozebrány použité struktury a jejich propojení. Jedna kapitola je věnována měření úspěšnosti aproximace řešení. Na konci je umístěno zhodnocení výsledků práce.

Klíčová slova

softcomputing, robot, bipedal, SOMA, simulace

Abstract

This thesis is focused on using softcomputing method for learning bipedal robot to walk. Robot is represented by virtual model. At the beginning are the motivation and reasons for processing this theme. Next is devised shape of the robot which will be used. Then are selected libraries used by simulation. Further is devised system of robot learning algorithms. The most important of them is SOMA which is therefore described more. Due to assumed computational complexity is part of thesis focused on optimalization and simplification. One chapter focuses on measurement of quality of solution approximation. At the end there is an evaluation of thesis results.

Keywords

softcomputing, robot, bipedal, SOMA, simulation

Citace

Jan Pokorný: Algoritmy pro řízení pohybu dvounohého robota, diplomový projekt, Brno, FIT VUT v Brně, 2010

Algoritmy pro řízení pohybu dvounohého robota

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Davida Martinka. Uvedl jsem všechny literární prameny, ze kterých jsem čerpal.

....
Jan Pokorný
25. května 2010

© Jan Pokorný, 2010.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod 3

2 Motivace 4
2.1 Využití . . . 4
2.2 Výhody pohybu po nohou . . . 4
2.3 Proč bipedálního robota? . . . 5
2.4 Co bylo úkolem této práce . . . 5

3 Návrh 7
3.1 Popis simulátoru . . . 7
3.2 Programovací jazyk . . . 7
3.3 Použité knihovny . . . 7
3.4 Stupně volnosti . . . 8
3.5 Návrh tvaru . . . 8
3.6 Optimalizace a zjednodušení návrhu . . . 9
3.7 Používané postupy konstrukce . . . 10

4 Algoritmus 11
4.1 Princip algoritmu . . . 12
4.2 Optimalizace množství potřebných parametrů . . . 13
4.3 Výběr softcomputingové metody . . . 14

5 Simulátor 15
5.1 Nástroje pro simulaci . . . 15
5.1.1 PyODE . . . 15
5.1.2 Pygame . . . 16
5.2 Vykreslení těles . . . 17

6 SOMA algoritmus 18
6.1 Co je to SOMA . . . 18
6.2 Jak SOMA funguje . . . 18
6.3 SOMA zblízka . . . 18
6.4 Použití . . . 19
6.5 Dynamické funkce . . . 21
6.6 Shrnutí informací . . . 21

7 Propojení SOMA a ODE simulace 22

8 Databáze stavů 23
8.1 Hashovací strom . . . 25
8.1.1 Vkládání do hashovacího stromu . . . 25
8.1.2 Vyhledávání v hashovacím stromě . . . 26
8.2 Databáze s hashovacím stromem . . . 27
8.2.1 Propojení databáze s hashovacím stromem . . . 27

9 Průběh algoritmu 28
9.1 Použití . . . 29

10 Měření úspěšnosti aproximace řešení 30
10.1 Hodnoty potřebné pro výpočet fitness . . . 30
10.2 Rychlost . . . 31
10.3 Výška gondoly . . . 32
10.4 Směr gondoly . . . 32
10.5 Rychlost motorů . . . 33
10.6 Pozice ve směru osy X . . . 33

11 Popis použitých struktur 35
11.1 Soubory . . . 35
11.2 Popis souborů . . . 36
11.2.1 debug.py . . . 36
11.2.2 fitness_func.py . . . 36
11.2.3 somaATE.py . . . 37
11.2.4 database.py . . . 37
11.2.5 machine.py . . . 38
11.2.6 main.py . . . 40

12 Nastavení 41
12.1 Parametry logování . . . 41
12.2 Nastavení fitness funkce . . . 42
12.3 Parametry simulátoru . . . 42
12.4 Nastavení SOMA . . . 43
12.5 Nastavení databáze a hashovacího stromu . . . 44

13 Ukládání a načítání stavu simulace 45
13.1 Modul pickle . . . 45
13.2 Použití modulu pickle v aplikaci . . . 46
13.2.1 Ukládání dat . . . 46
13.2.2 Načítání dat . . . 46

14 Použití a výsledky testování programu 47
14.1 Použití . . . 47
14.2 Testování . . . 48

15 Návrhy možných úprav 50

16 Závěr 51

Kapitola 1

Úvod

Cílem této práce je navrhout algoritmus, který dokáže ovládat stroj se dvěma končetinami. Protože je každý skutečný krok trochu jiný – třeba už kvůli nerovnostem povrchu – nedají se v tomto případě snadno použít standardní výpočetní algoritmy. Proto se zde volí cesta softcomputingových metod, schopných přizpůsobit se nepřesnostem reálného světa, a tak se na něj daleko lépe adaptovat.

Pohyb pomocí končetin je ve skutečnosti velice složitý a vyžaduje velkou dávku koordinace. Přesto se s ním setkáváme každý den. Nejvíce náročný je zřejmě pohyb po dvou nohou – je jasné, že je třeba neustále udržovat rovnováhu narozdíl od vícenohých tvorů. Výhodou je oproti tomu při stejných proporcích větší rozhled a také dosah. Nemluvě o volném páru končetin.

Nohy také umožňují pohyb v nerovném terénu, v čemž výrazně předstihují kola. Proč tedy nenavrhnout stroj, který by k pohybu používal končetiny? Něco takového by jistě našlo spoustu uplatnění.

Kapitola 2

Motivace

2.1 Využití

Roboty chodící po nohou není jistě třeba nikomu představovat. Prakticky každý se s nimi již nějak setkal. Ať už se jednalo o pouhé science fiction, či reálnou prezentaci těchto strojů.

Právě z knih a sci-fi filmů ale plynou poměrně mylné představy o účelnosti těchto strojů – téměř vždy se tam tyto stroje předvádějí jako „kanony na nožičkách" a zcela se ignorují možnosti, které tato forma pohybu nabízí. Jistě, ve filmech to ani jinak nejde a navíc lze předpokládat, že jakmile se zvládne technická stránka věci a roboti s nohami se stanou použitelnými, bude armáda jedním z prvních a zároveň i největších investorů.

Rád bych ale nastínil i další možnosti využití těchto strojů, jakkoliv se zatím mohou zdát fantastické. Mezi prvními mohou být roboti – průzkumníci, jejichž účelem může být průzkum příliš malých, nebezpečných, nebo pro člověka nedostupných míst.

Robot na nohou se může pohybovat v sutinách, prohledávat jeskyně, uplatní se jako záchranář a nohy budou také jistě lépe sloužit svému účelu než kola i při průzkumu jiných planet.

Lidé odkázaní na vozík by mohli využít umělých končetin s podobnými algoritmy, aby se opět zařadili do společnosti. Již dnes se koneckonců vyrábí protézy propojené s nervovým systémem člověka.

Protože je chůze složitý pohyb, existuje také mnoho algoritmů, které se ji snaží napodobit. Cílem této práce je použít alternativní přístup k řešení tohoto problému: Použití softcomputingových metod.

Mezi tyto metody se řadí také například genetické algoritmy, které využívají k hledání optimálního řešení evoluci.

Předností těchto metod je skutečnost, že jejich pomocí lze získat kvalitní řešení, pokud je dostatečně dobře specifikován problém.

Jejich masovějšímu rozšíření však brání jejich občasná nepředvídatelnost. Vzhledem k velkému podílu náhody se totiž dají jen obtížně formálně dokázat.

2.2 Výhody pohybu po nohou

Hlavní výhodou nohou nad koly je jejich schopnost poradit si i s velmi obtížným terénem – například i takovým, jakým se člověk stále obklopuje. Mám na mysli hlavně schody, obrubníky a další každodenní překážky, kterých si právě díky nohám obvykle ani nevšimnete.

Zatímco kola vítězí svou jednoduchostí, nohy přes svou složitost přináší také obrovskou manévrovatelnost i všestranost – automobil převrácený na bok se sám nepostaví, kdežto téměř jakékoliv zvíře s nohami to snadno dokáže. Je snadné otočit se na místě, je možné měnit výšku.

Pokud jsou k dispozici další končetiny, lze jich využít jako dalších nohou ke zvýšení stability.

Také je možné použít nohou jako protizávaží, pokud je třeba trup či horní končetiny vychýlit dál od těžiště stroje.

2.3 Proč bipedálního robota?

Bipedalismus je forma pohybu, kde se organismus pohybuje pomocí dvou končetin, typicky nohou. Z toho pochází i latinské slovo „biped“, značící organismus používající k pohybu dvě nohy.

Ze současných živočišných druhů používá tento způsob pohybu jako primární relativně málo druhů. Větší skupina živočichů je schopna za zvláštních okolností bipedálního pohybu využít – někteří plazi dokáží v případě ohrožení utíkat do bezpečí po dvou nohou.

Bipedální pohyb se vyskytuje ve třech variantách.

- chůze
- běh
- skákání

Jaké jsou výhody a nevýhody právě dvounohého robota? Více nohou zajišťuje vyšší stabilitu a například již šest nohou umožní neustálou stabilní polohu těžiště stroje. To se jistě hodí, pokud je stroj příliš velký a těžký, než aby se rovnováha dala snadno udržet. Pro naše účely je to však zbytečné a dokonce by to valnou měrou komplikovalo výpočet.

Stroj, který se pohybuje po dvou nohou s sebou přináší oproti snížené stabilitě naopak řadu výhod. Hlavní z nich jsou určitě mnohem menší stavební náklady. Každá končetina je složitý aparát a mít jich co nejméně jistě dost ušetří – a to i na hmotnosti celého stroje.

Vyšší labilitu robota je nutno kompenzovat odpovídajícím softwarem. Ten by zároveň mohl – při použití správných algoritmů – docílit vyšší rychlosti pohybu. Není třeba se starat o tolik končetin a jejich postupné přesouvání na nová místa. (Vícenozí roboti mohou sice přesouvat více nohou paralelně, jenže pak ztrácí svou stabilitu, pro kterou byli vybudováni.) Pokud vyjdeme z poznatků z živočišné říše, můžeme říci, že pohyb po více nohou může být rychlejší v případě, že zvíře má dostatečně ohebnou páteř. To u stroje ale nelze předpokládat.

2.4 Co bylo úkolem této práce

Tato práce si klade za cíl použít k naučení chůze bipedálního robota softcomputingové metody. Robot bude reprezentován modelem ve virtuálním prostředí.

- Součástí je tedy popis vytvoření tohoto prostředí a poté i modelu robota pomocí dostupných nástrojů.

- Tvar robota má rozhodující vliv na stanovení dalších cílů, proto byl předem navrhnut. Návrh bylo třeba držet v rozumných mezích tak, aby pokud možno nebyly překročeny hranice časové a výpočetní náročnosti konstrukce modelu.
- Pomocí navrženého tvaru stroje se stanovily jednoduché úkoly, které má robot plnit (např. pohyb kupředu nebo stání v klidu...). Míry úspěšnosti při plnění jednotlivých úkolů se přepočítají na ohodnocovací (fitness) funkci.
- Dále jsou navrženy algoritmy, na které se aplikuje ohodnocovací funkce.
- Parametry algoritmů se přizpůsobí tak, aby robot dosahoval co nejlepších výsledků
- Jakmile je dosaženo dostatečného výkonu - robot chodí dostatečně dobře - vytvoří se obraz tohoto nastavení, který bude možné buď dále rozvíjet, nebo nastavit na další identický stroj.
- Posledním krokem je zhodnocení dosažených výsledků a navržení dalších úprav a vylepšení.

Kapitola 3

Návrh

3.1 Popis simulátoru

Použití simulátoru s sebou přináší mnohé výhody. Jednou z nich je samozřejmě cena. Konstrukce skutečného robota přesahuje možnosti této práce. Na místě je i snadná okamžitá modifikace modelu v případě, že se ukážou jeho dosud skryté nedostatky.

Reaguje-li robot příliš pomalu, než aby byl schopen normálně pracovat, můžeme v simulaci upravit časové měřítko tak, aby byl tento problém odstraněn. V takovém případě je třeba před skutečnou realizací konstrukce třeba uvážit, jak dosáhnout nápravy.

Výhodou prostředí simulátoru je také možnost postupného zesložiťování celé simulace – ať už stroje či okolního prostředí, čímž se zvýší realističnost modelu.

3.2 Programovací jazyk

Pro tvorbu tohoto programu byl zvolen programovací jazyk Python pro jeho přenositelnost a flexibilitu. Jsou pro něj volně k dispozici potřebné knihovny.

Hlavním důvodem použití tohoto jazyka však je skutečnost, že Python je jazyk vhodný k prototypování a rychlému vývoji, což je zcela v zájmu této práce.

Nevýhodou je samozřejmě nižší rychlost běhu. Pokud to bude nutné je možno tento problém kompenzovat již zmíněnou změnou rychlosti běhu simulačního času.

Osvědčí-li se navržený algoritmus, jeho další optimalizace může být přepsání kódu například do C++, popřípadě podobného jazyka vhodného pro realizaci hotových projektů.

3.3 Použité knihovny

Pro vytvoření simulace je použito několika již hotových nástrojů. Samotný algoritmus učení je účelnější vytvořit individuálně, neboť jej je třeba adaptovat na konkrétní aplikaci.

Nezbytné pro simulaci bude vytvoření fyzikálního prostředí, ve kterém se bude robot pohybovat. Zde se nabízí PyODE – Python Open Dynamic Engine.

Dále bude třeba dosáhnout nějaké formy jednoduché vizualizace výstupu. K tomu se zdá vhodná knihovna pygame. Jedná se o knihovnu pro zobrazování grafických primitiv. Pro účely práce je naprosto dostatečná.

Upozorňuji, že instalace těchto knihoven může být pod některými operačními systémy poněkud komplikovaná.

3.4 Stupně volnosti

Jak již bylo řečeno, tvar robota je jedním z určujících faktorů, které přímo ovlivňují složitost výpočtu.

Při návrhu tvaru stroje je třeba brát v potaz omezující podmínky, a to tak, aby byl robot schopen udržet rovnováhu a zároveň se snadno pohybovat. Nejdůležitější z nich je patrně výpočetní náročnost, kvůli které je třeba použít minimální počet stupňů volnosti kloubů stroje.

V oblastech, kde se pracuje s robotickými rameny se často vyskytuje pojem „stupeň volnosti". Tento pojem souvisí s možnosti pohybu tělesa.

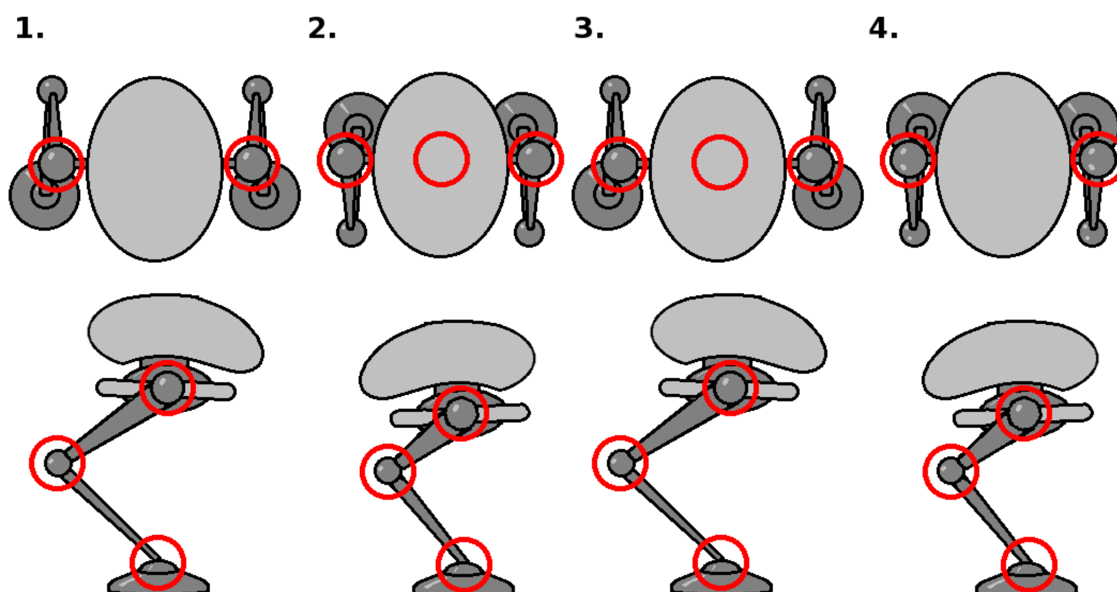
Těleso umístěné volně v prostoru se může pohybovat ve směru os X, Y a Z a podle stejných os se také může otáčet. Volné těleso má tedy 6 stupňů volnosti.

Vzájemná poloha částí končetiny robota je vždy jednoznačně určena určitým počtem údajů. Nejmenší počet těchto údajů udává počet stupňů volnosti oněch částí (kinematická dvojice). Kinematické dvojice, kterých budeme využívat, budou tvořeny členy spojenými rotačními klouby, které mají typicky jeden stupeň volnosti.

Z toho vyplývá, že každý další stupeň volnosti v nějakém kloubu přidává stroji další nastavitelný parametr a tedy rozšiřuje stavový prostor možných řešení o další rozměr. Proto je tedy nutné tyto možnosti dostupným způsobem omezovat.

3.5 Návrh tvaru

K dispozici jsou čtyři návrhy tvaru robota, které připadají v úvahu.



Obrázek 3.1: Návrh možných tvarů robota, červeně vyznačené rotační klouby s jedním stupněm volnosti

Na obrázku 3.1 jsou schematicky znázorněny čtyři možné tvary simulovaného robota.

1. „kolena" se ohýbají opačně než jako u člověka, gondola upevněna neotočně

2. „kolena“ se ohýbají stejným směrem jako u člověka, gondola upevněna otočně

3. „kolena“ se ohýbají opačně než jako u člověka, gondola upevněna otočně

4. „kolena“ se ohýbají stejným směrem jako u člověka, gondola upevněna neotočně

Otočné upevnění gondoly stroje sice přidává do stavového prostoru další rozměr, dá se však předpokládat, že bude-li gondola neustále udržována v přímém směru, může se tak zajistit zvýšená stabilita těžiště celého stroje a zkvalitní se tak plynulost chodu robota.

Ve směru ohybu těchto kolen prozatím nespatřuji žádné výrazné výhody a nevýhody, takže byla vybrána konstrukce čtvrtého typu – s otočnou gondolou a koleny ohýbajícími se opačně než jako u člověka.

Tento směr ohybu také není tak často využíván v jiných studiích, a proto stojí za to jej blíže prozkoumat (obr. 4.2).

Díky těžišti, které se snadno spolu s koleny posouvá nazad, existuje možnost mírně lepší stability stroje, případně i trochu zvýšené schopnosti brždění.

Pokud předpokládáme nenulovou hmotnost končetin, potažmo kloubů, pak čím níže bude gondola, tím více dozadu budou muset být posunuty kolenní klouby a dojde tak k mírnému posunu těžiště.

Pomalý pohyb vpřed bude potřebovat kompenzaci náklonu stroje a tedy přijde i toto malé vylepšení vhod.

Naopak při vyšších rychlostech pohybu a bude-li třeba brzdit, může snížení výšky gondoly hypoteticky snížit setrvačnost robota.

Není jisté, jak moc tato vlastnost ovlivňuje pohybové schopnosti stroje, dá se však předpokládat, že tyto hodnoty jsou velmi malé.

3.6 Optimalizace a zjednodušení návrhu

Jak je již vidět z návrhu stroje 3.1, počítá se s tím, že kolenní klouby nebude možné ohýbat o libovolný úhel. Tato skutečnost má dva důvody:

1. omezením dojde ke zjednodušení reálné konstrukce robota. Lze pak totiž snadno použít výkonnější písty namísto servomotorů, kde vzniká problém s udržením natočení kloubu pod zátěží pokud má stroj udržet ohnuté končetiny.

2. značné zmenšení stavového prostoru řešení (tedy množství pozic, které může robot zaujmout) a tedy urychlení výpočtu, což je v našem případě podstatnější.

Omezení v otočení kloubů bude vhodné aplikovat i na další klouby.

Dalším zjednodušením, které je nasnadě, je celkové vynechání přímého ovládání některých kloubů v „kotnících“ stroje. Plotna na konci končetiny musí být schopna náklonu vpřed, vzad, doprava a doleva.

Pohyb vlevo a vpravo může být v našem případě řízen algoritmem, který plotnu udržuje rovnoběžné s povrchem, po kterém se robot pohybuje. Náklon stroje doleva či doprava je zcela kompenzován klouby umístěnými u „pánve“.

Náklon vpřed již nelze takto eliminovat, protože je třeba mít možnost naklonit stroj kupředu.

Bylo-li by třeba, aby se stroj pohyboval v náročnějším terénu, bude lépe, bude-li natočení ploten řízeno vlastním algoritmem.

3.7 Používané postupy konstrukce

Dvounohý robot není nijak nový nápad a existuje spousta dalších návrhů, které byly podrobně zpracovány [8]. Tato práce zkouší odlišné postupy.

Základem návrhů, se kterými jsem se setkal, je obvykle nějaký oscilátor, který využívá periodicity pohybu po nohou. Pomocí něj pak stroj snadno identifikuje část pohybu, ve které se nachází a postupuje podle předem vytvořeného schématu. Tvorba tohoto schématu se již liší.

Nevýhodou tohoto postupu je skutečnost, že dojde-li k nějaké nepředpokládané události – stačí i nerovnost povrchu, předem připravený postup selže.

Toto částečně řeší návrh [5], [4], který používá tlakový senzor pro detekci kontaktu končetiny se zemí. V okamžiku, kdy se čidlo sepne, dojde k nastavení oscilátoru končetiny do výchozí polohy.

Resetováním oscilátoru lze předejít některým typům nerovností.

Kapitola 4

Algoritmus

Existují dvě cesty, jakými učit robota chodit:

- Vytvoří se jediný virtuální stroj, který bude zdokonalovat svůj pohyb, na základě zabudovaného evolučního algoritmu. Úspěšnost stroje bude hodnocena průběžně.
- Použije se evoluce v plném měřítku. Velké množství strojů s vlastními algoritmy bude soupeřit mezi sebou a vítězní roboti budou zkříženi.

Vzhledem k výpočetní náročnosti druhé možnosti byla zvolena první cesta – stroj se bude učit chodit průběžně. Pokud ovšem dojde k jeho pádu, bude prostředí vyresetováno, aby stroj nemusel vstávat – tento úkol je pro robota náročný a je nad rámec hlavního cíle této práce.

Při řešení úlohy pohybu je třeba vzít v potaz velké mnnožství proměnných a parametrů. Právě proto je třeba jejich počet pokud možno redukovat.

Cílem algoritmu je nalezení extrémů multidimenzionální fitness funkce, jejímiž parametry jsou vstupy určené stavem robota a výstupem této funkce je odpovídající fitness hodnota.

Dimenze funkce jsou dány počtem stupňů volnosti – každý stupeň volnosti umožňuje natočení části stroje podle jedné osy. Rozmezí a velikost rychlosti změny vymezuje právě jeden parametr.

S vybraným modelem máme tak devět dimenzí, mezi jejichž hodnotami se hledají extrémy:

- kolena, horizontální osa – pravá, levá
- kyčle, horizontální osa – pravá, levá
- kyčle, vertikální osa – pravá, levá
- kotníky, horizontální osa – pravá, levá
- gondola, vertikální osa

Je však nutné počítat i s parametry mimo fitness funkci, které jednoznačně určí stav stroje:

Každá součást stroje má tyto parametry:

- pozice, pro osy X, Y a Z

- natočení, rotační matice(3 x 3)
- rychlost, pro osy X, Y a Z
- rychlost rotace, pro osy X, Y a Z

4.1 Princip algoritmu

S nadefinovaným strojem a vstupními parametry můžeme nyní popsat hlavní myšlenku algoritmu.

Základem je samozřejmě simulátor fyzikálního prostředí a v něm umístěný připravený model. Stav modelu je jednoznačně určen pomocí výše popsaných hodnot. Je tedy možné stav stroje okopírovat do druhé identické simulace.

V sekundárním simulačním prostředí je nalezeno optimální nastavení pohybu jednotlivých motorů pohánějících robota tak, aby po krátkém kroku sekundárního simulátoru bylo dosaženo co nejlepší hodnoty fitness. Nalezení této hodnoty je zajištěno evolučním algoritmem – zde toto místo zastává SOMA.

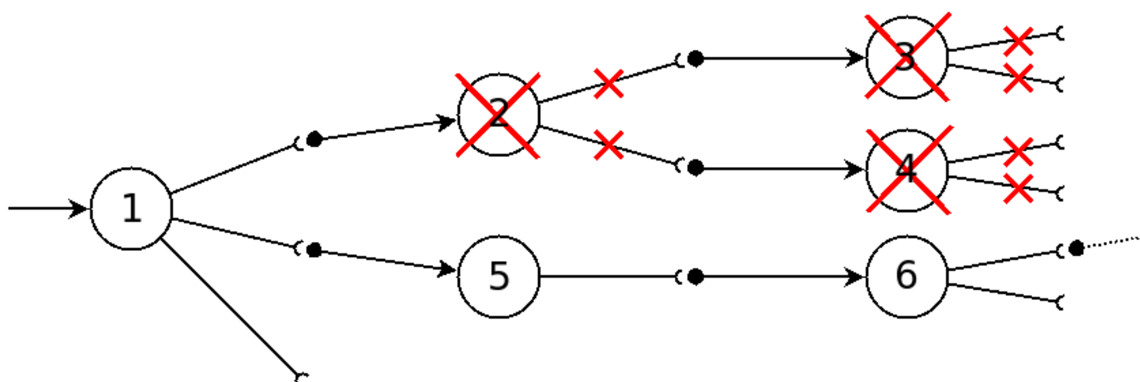
Jejím výsledkem je skupina kandidátních řešení – hodnoty extrémů hledané multidimenzionální funkce. Tyto hodnoty jsou zaznamenány k původnímu stavu.

Nejlepší z hodnot je poté aplikována na primární simulaci a dosáhne se tak nového stavu, a celý cyklus se opakuje.

Každý stav, kterého primární simulace dosáhne, je porovnáván s množinou všech předchozích dosažených stavů. Jakmile je nalezen existující podobný stav, použije se u něj přiložená konfigurace pro přechod do dalšího stavu.

Tento princip samotný si však neporadí s uváznutím v lokálním extrému – situace, kdy je třeba použít horších hodnot fitness, aby bylo možné později úspěšně pokračovat.

Proto je do algoritmu zakomponována možnost „kritického selhání“ – například v případě pádu. Za těchto podmínek je větev předchozího stavu, které bylo použito, označena jako nepoužitelná, simulace se vrátí do předchozího stavu a použije se druhé nejlepší kandidátní řešení, už dříve vygenerované SOMA (obr. 4.1). Jsou-li všechna kandidátní řešení vyčerpána, opět se větev označí jako nepoužitelná a simulace se vrátí o krok zpět.



Obrázek 4.1: Princip algoritmu: Prohledávání prostoru nalezeného SOMA probíhá podle čísel jednotlivých stavů. Jakmile se vyčerpají všechny možnosti, algoritmus se vrací o krok zpět.

4.2 Optimalizace množství potřebných parametrů

Protože stavy stroje bude třeba mezi sebou porovnávat, je dobré množství těchto parametrů zmenšit na minimum.

Každá součást stroje nyní potřebuje pro své jednoznačné určení

$$3(\text{pozice}) + 9(\text{natočení}) + 3(\text{rychlost}) + 3(\text{rychlost rotace}) = 18 \tag{4.1}$$

stavů. Pokud použijeme 12 komponent, dostáváme stav definovaný

$$12 \cdot 18 = 216 \tag{4.2}$$

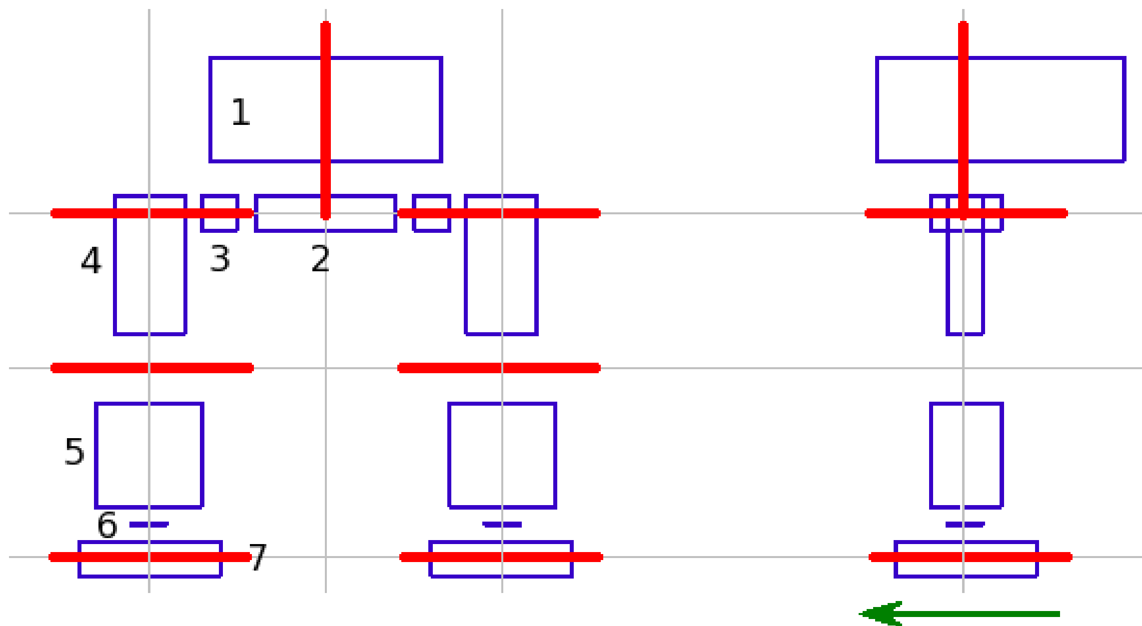
hodnotami.

Tento počet hodnot je velmi vysoký a je nezbytné jej redukovat.

Citelný podíl na počtu rozměrů je tvořen rotačními maticemi robota v různých osách a výrazně tak zvyšují výpočetní náročnost.

Alternativou je v tomto případě použití quaternionu - čtveřice čísel, které tak zredukují počet hodnot popisujících těleso na 13. Tedy stavový prostor nyní bude mít $12 \cdot 13 = 156$ hodnot.

I tuto hodnotu je možné redukovat, lze totiž odstranit parametry udávající pozici a natočení některých těles, pokud si uvědomíme, že jejich pozice je jednoznačně určena pomocí kloubů natočením a umístěním okolních těles.



Obrázek 4.2: Tvar simulovaného robota: Modře tělo robota, červeně jsou vyznačeny osy jednotlivých kloubů, zelená šipka označuje směr pohybu (směr osy Z). 1 – gondola, 2 – pánev, 3 – kyčelní kloub (rotace končetiny vpřed a vzad), 4 – stehno (rotace končetiny vlevo a vpravo, připojeno na kyčelní kloub), 5 – lýtko, 6 – kotník (střed otáčení v chodidle), 7 – chodidlo (spolu s kotníkem umožňují natáčení plotny v osách X a Z)

Tímto způsobem můžeme zcela odstranit tato tělesa:

- pánev
- levé stehno
- pravé stehno
- levý kotník
- pravý kotník

Tím se vyřadí $5 \cdot 7 = 35$ dalších hodnot. Zůstává ještě 121 čísel definujících stav.

Posledním zjednodušením je zanedbání hmotnosti u některých částí stroje, čímž odpadne potřeba počítat jejich setrvačnost a tedy dalších 6 hodnot na takové těleso.

Takto lze vyřadit tato tělesa:

- pánev
- levý kyčelní kloub
- pravý kyčelní kloub
- levé stehno
- pravé stehno
- levý kotník
- pravý kotník

Tím se vyřadí $6 \cdot 7 = 42$ dalších hodnot. Takto nakonec zůstává 79 čísel definujících stav.

Intervaly, ve kterých se jednotlivá čísla pohybují, se liší:

- pozice: $(\langle -2; 2\rangle, \langle -2; 2\rangle, \langle -2; 2\rangle)$
- rotace: $(\langle -1; 1\rangle, \langle -1; 1\rangle, \langle -1; 1\rangle, \langle -1; 1\rangle)$
- rychlost: $(\langle -40; 40\rangle, \langle -40; 40\rangle, \langle -40; 40\rangle)$
- rychlost rotace: $(\langle -40; 40\rangle, \langle -40; 40\rangle, \langle -40; 40\rangle)$

4.3 Výběr softcomputingové metody

Výběr metody může mnohé ovlivnit, ale pro funkci s množstvím parametrů, jsem se rozhodl pro algoritmus SOMA. Využívám toho, že jsem s tímto algoritmem dobře obeznámen a znám jeho možnosti. Vhodnou implementaci knihovny pro SOMA v Pythonu jsem ale nenalezl, a proto jsem se rozhodl napsat a použít vlastní.

Je nutno podotknout, že SOMA poskytuje potřebnou jednoduchost, ale i variabilitu, které bude v tomto případě třeba.

Protože SOMA nepatří mezi známé softcomputingové metody, je vhodné tento algortimus poněkud přiblížit a ukázat jeho jednoduché použití.

Kapitola 5

Simulátor

5.1 Nástroje pro simulaci

Simulátor je nejdůležitější částí navrhovaného systému, protože je to právě simulátor, který produkuje testovací data, a na který jsou aplikovány další algoritmy.

Jeho úkolem je v tomto případě simulovat reálné prostředí, kde platí základní fyzikální zákony. K tomuto účelu byla použita knihovna PyODE – Python Open Dynamic Engine.

Knihovna nemá grafický výstup, proto bylo potřeba použít další nástroj – Pygame.

Jedná se o jednoduchou knihovnu pro zobrazování 2D grafických primitiv. Snadno jí lze využít i v našem případě, aby bylo dosaženo základních grafických výstupů.

Propojení obou knihoven je velmi těsné, takže každé těleso umístěné do simulace může být zároveň zobrazeno.

5.1.1 PyODE

PyODE, jak již bylo zmíněno, je nástroj pro simulaci fyzikálního prostředí s fungujícími Newtonovými zákony. K činnosti využívá knihovnu ODE, která je napsaná v C++ a přejímá její funkce. S trochou představivosti je možné použít manuálové stránky ODE [7] k pochopení příkazů PyODE.

Knihovna postrádá jakýkoliv grafický výstup, hodnoty je tedy nutné dále zpracovávat.

Inicializace vyžaduje manuální nastavení některých parametrů simulovaného světa.

Základem je inicializace prázdného prostředí. Jedná se o virtuální vakuum bez gravitačních sil, či jakéhokoliv tření. Zde je vhodné nastavit míru přesnoti výpočtu, která nepřímou úměrou ovliňuje rychlost běhu simulace.

Do tohoto prostředí je již možné přidávat tělesa základních tvarů. Tento systém si vystačí pouze s kvádry, které jsou definovány:

- hustotou
- rozměry – X, Y, Z
- pozicí – X, Y, Z

PyODE nepracuje s pevně danými fyzikálními jednotkami, hodnoty lze použít jakékoliv, pokud se budou dodržovat odpovídající vztahy mezi nimi. V simulátoru jsou použity hodnoty odpovídající metru, kilogramu a sekundě.

Dále je třeba přidat svisle působící gravitační sílu určenou vektorem $\vec{g} = (0; -9,81; 0)$

Pokud se spustí hlavní smyčka programu, začnou nyní všechna tělesa padat.

PyODE využívá tři základní typy objektů: těleso, kloub a geom.

K vytvoření kompletního stroje je třeba přidat klouby - joints. Kloubem se rozumí jakékoliv spojení dvou těles, které částečně omezuje jejich vzájemnou pozici. PyODE zná více typů kloubů, včetně pístových a kulových, pro naše účely však stačí kloub typu *Hinge*, který narozdíl od většiny ostatních, lze vybavit motorem. Kloub je určen tělesy, které propojuje, svým umístěním v prostoru a také směrem osy, okolo které se může otáčet.

Posledním typem tělesa je geom. Jedná se o virtuální obalové těleso určené pro detekci kolizí. Pokud není použito, tělesa sebou mohou volně procházet.

Ke každému vytvořenému tělesu je tedy přidán jeho geom. Jedním ze speciálních geomů je i podlaha.

Kolize dvou těles PyODE řeší pomocí vytvoření zvláštního druhu kloubu, který je před vstupem do další iterace smyčky opět zrušen. Nad dvěmi klouby takto vytvořenými se volá metoda, kde lze nadefinovat chování v případě srážky. Zde je možno nakonfigurovat pružnost těles a také tření, které je další nutností. Bez něj stroj nebude schopen pohybu.

Hlavní smyčka PyODE simulace nejprve provede výpočet kolizních míst, poté krok o předem danou časovou jednotku a závěrem je třeba opět seznam kolizních míst smazat.

Každé těleso může v kterékoliv fázi vrátit svůj momentální stav a také jej i nastavit.

Stejně tak je možné nastavit i vlastnosti kloubů. Klouby typu pant lze použít jako motory, jsou-li jim nastaveny tyto parametry:

- výkon – hodnota určuje jakou maximální silou motor dokáže působit, aniž by byl přetlačen
- rychlost – hodnota určuje rychlost, jakou se bude motor snažit udržet, znaménko určuje směr rotace

Například nastavíme-li výkon motoru na 200 a rychlost na 0, získáme pevné spojení dvou těles, které vydrží sílu 200 jednotek, než podlehne.

5.1.2 Pygame

Pygame je jednoduchý nástroj vyvinutý za účelem vizualizace jednoduchých počítačových her.

Do programu se začlení pomocí importování modulu `pygame`:

```
import pygame
```

Vytváří jednoduché okno bez ovládacích prvků, ve kterém se vykreslují grafická primitiva.

Proces vykreslování se spouští iterativně s tím, že plynulost zobrazování zajišťuje objekt `pygame.time.Clock`, který podle nastaveného časového kroku `dt` v případě příliš vysoké rychlosti zpomalí běh tak, aby byl grafický výstup plynulý. (V projektu není tento objekt použit, je potřeba maximální rychlost běhu a zpomalení není vhodné.)

Pygame k vykreslování používá metodu `pygame.draw`, která vykreslí daný objekt podle zadaných parametrů do bufferu.

Jakmile jsou tělesa připravena, zavolá se metoda `pygame.display.flip`, která vymění grafické buffery a vykreslí tělesa.

Plátno je třeba po každé iteraci smazat, protože dříve vykreslená tělesa na něm jinak zůstávají.

5.2 Vykreslení těles

Protože simulátor využívá pouze kvádry, k vykreslení se využije metoda:

```
pygame.draw.line(surface, color, startPoint, endPoint, thickness),
```

kde

- `surface` je plátno, kam se má objekt vykreslit,
- `color` je tříprvkový vektor (v Pythonu objekt typu *tuple* nebo *list*) který určuje barvu tělesa (v RGB formátu),
- `startPoint` a `endPoint` jsou dvouprvkové tuple a
- `thickness` je tloušťka čáry.

K zobrazení tělesa v aplikaci je třeba znát jeho umístění, rozměry a natočení. Tyto hodnoty dokáže vrátit PyODE a pomocí nich se dále vypočtou souřadnice rohů kvádru, mezi kterými se vykreslí čáry. Jedna ze souřadnic se při vykreslování nebere v úvahu.

Vykreslení je velmi jednoduché a nepočítá s perspektivou. Jako kompenzace tohoto nedostatku se provádí zobrazení stroje ze dvou směrů.

Vizualizace simulace se dá vypnout v konfiguračním souboru, aby se zvýšila rychlost výpočtu.

Kapitola 6

SOMA algoritmus

6.1 Co je to SOMA

SOMA je jednou z poměrně nových softcomputingových metod, která si však jistě zaslouží pozornost a to hlavně díky své jednoduchosti a účinnosti. Algoritmus je zaměřen na optimalizaci – hledání extrémů n-rozměrných funkcí na základě výpočtu okamžité hodnoty funkce v daném bodě [9].

Zkratka SOMA ve skutečnosti znamená Samo-Organizující se Migrační Algoritmus a princip je velmi podobný optimalizaci pomocí particle swarm (roj částic).

6.2 Jak SOMA funguje

Algoritmus prohledává konečný stavový prostor řešení a snaží se přitom nalézt pokud možno globální extrém. Jestli se jedná o minimum nebo maximum není důležité, obrácení znaménka u hodnoty fitness funkce zajistí obě možnosti.

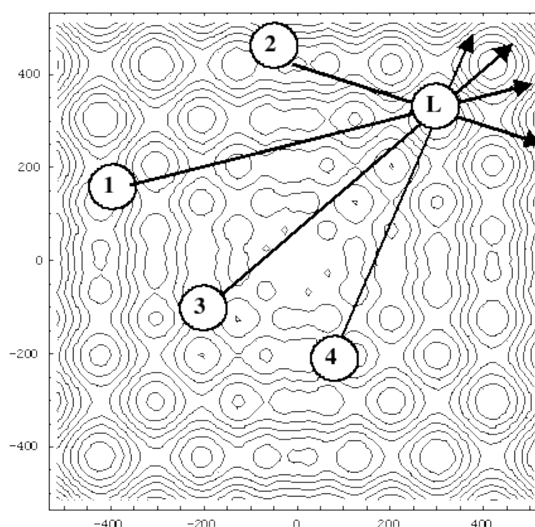
Analytické řešení složitějších optimalizačních problémů je obvykle příliš zdlouhavé. SOMA oproti tomu počítá pouze hodnoty funkce v bodech, kterých poté využívá dále. Je důležité si uvědomit, že SOMA patří mezi softcomputingové algoritmy, a proto je jakýkoliv získaný výsledek jen více či méně přesnou aproximací skutečného řešení.

6.3 SOMA zblízka

Hledání řešení probíhá v ohraničeném prostoru. Čím přesnější je jeho ohraničení, tím bývá algoritmus rychlejší ve zpřesňování aproximace. Na takto vymezenou hyperplochu se nyní zcela náhodně rozmístí body (jedinci), ve kterých se vypočte fitness hodnota. Nyní se mezi jedinci vybere ten s nejlepší fitness – *Leader*. Ten reprezentuje nejlepší dosud nalezené řešení.

Zbytek procesu už probíhá iterativně s tím, že každá další iterace přinese lepší nebo stejnou hodnotu výsledku. S rozmístěnými jedinci a stanoveným Leaderem začíná migrace. Každý jedinec se přesouvá směrem za Leaderem (tím směrem budou patrně dobré hodnoty), přičemž prohledává body na trase, kterou prochází (obr. 6.1). Čím blíže je jedinec k Leaderovi, tím menší vzdálenost je mezi těmito body. Nakonec se přesune na bod, který má nejlepší fitness. Pokud je tato hodnota vyšší, než má Leader, stává se tento jedinec novým Leaderem.

Je jasné, že Leader se pohybu neúčastní, protože nemá daný směr kam jít. Jakmile migrovali všichni jedinci, začíná nová iterace. Popsané strategii se říká *All To One* [1].



Obrázek 6.1: Jedinci se posouvají za Leaderem

Algoritmus obnáší komplikaci, které se říká „problém nahloučení“. Po několika iteracích, kdy se Leader nemění, se totiž obvykle většina jedinců shlukne kolem něj a je-li Leader v lokálním extrému, pak ten globální nebude nalezen. Řešení problému nahloučení je mnoho a v knihovně je implementováno několik z nich:

Prvním je délka trasy migrace jedince, která zde nekončí u Leadera, ale pokračuje až za něj. Jedinec při jedné migraci implicitně urazí 2,5násobek své vzdálenosti od Leadera, což umožní postupnou divergenci od případného lokálního extrému.

Dalším vylepšením je takzvaný „pertubační vektor“. Původní přímá trasa jedince za Leaderem je deformována podle míry pertubace.

Jedinci také „stárnou“. Jakmile překročí určitý počet iterací, jsou přesunuti na náhodně zvolenou pozici [2].

Poslední použitou technikou je autorem navržená strategie *All To Elite*, kdy je Leaderů stanoveno více a jedinec následuje vždy náhodně zvoleného z nich [3]. Tato strategie je pro účel této práce klíčová.

6.4 Použití

Nejprve je třeba mít stanovenou ohodnocovací funkci, která na základě vstupních parametrů vrátí výsledek – fitness. Čím nižší hodnota, tím lépe. Jako parametr musí brát funkce list hodnot – souřadnic na hyperploše. Výstupem pak musí být číslo.

Příklad:

```
# vrátí součet mocnin hodnot libovolně dlouhého vektoru
def f(a):
return sum([x*x for x in a])
```

Pak je třeba vytvořit instanci objektu `cSomaATE`, která jako parametry vyžaduje

- ohodnocovací funkci
- list mezních hodnot pro jednotlivé rozměry – tím je definován i počet rozměrů
- počet jedinců
- maximální stáří jedince – pokud není zadáno, nebo nastaveno na 0, je vypnuto

Příklad:

```
soma = cSomaATO(f,[[-5, 5],[0.0, 12.3],[-1.0, 1.0]],5,5)
```

Nyní je SOMA připravena a stačí jen volat metodu `step()`, která provede jednu iteraci výpočtu a vrátí dvojici:

$$([<\text{list souřadnic nejlepšího jedince}>], <\text{nejlepší nalezená hodnota}>)$$

Opakované volání této metody výsledek zpřesňuje.

Poznámka: Metoda zároveň umožňuje vložit jinou ohodnocovací funkci jako parametr a SOMA ji od toho okamžiku začne používat.

Příklad:

```
# provede tři iterace algoritmu a~pak vypíše konečný výsledek
for i in range(3):
  result = soma.step()
print result
```

Třída cSomaATE má několik vlastností, které jsou předem nastaveny, ale lze je měnit, pokud víte, co děláte. Jsou to:

- pertubation – nastavuje se v mezích $0,0 - 1,0$ (default $0,3$)
- stepMultiplier – relativní vzdálenost mezi prozkoumávanými body, číslo by nemělo beze zbytku dělit $1,00$ (default $0,17$)
- stepDistance – násobek vzdálenosti, kterou jedinec při iteraci projde vzhledem k Leaderovi (default $2,5$)
- eliteSize – počet Leaderů – tedy velikost elity (default 3). Pouze u strategie *All To Elite*

Poznámka:

$$\text{Počet bodů prozkoumávaných každým jedincem} = \frac{\text{stepDistance}}{\text{stepMultiplier}}$$

6.5 Dynamické funkce

Algoritmus je díky své struktuře schopen hledat i extrémy funkcí, které se v čase mění. I v tomto případě dosahuje všeobecně kvalitních výsledků, s ohledem na parametry zkoumané funkce. Vzhledem k povaze této práce se však nebudu těmito parametry zabývat.

6.6 Shrnutí informací

V celém SOMA není použito složitější matematické operace, než je dělení a přesto dosahuje výborných výsledků. Tento stručný výtah neobsahuje zdaleka všechny informace o tomto algoritmu a nezabývá se ostatními strategiemi prohledávání stavového prostoru, jako jsou *All To One*, *All To Random*, *All To All* nebo *All To All Adaptive*.

Případným zájemcům o tuto tematiku bych doporučil internetové stránky docenta Ivana Zelinky věnované SOMA [6].

Kapitola 7

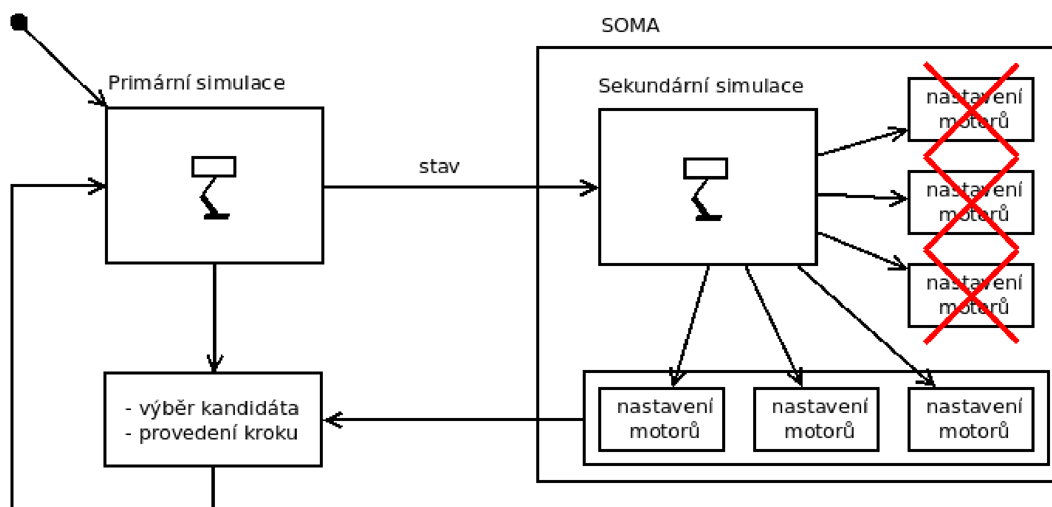
Propojení SOMA a ODE simulace

Nalezení takového postupu, který by zajistil robotu plynulý pohyb, vyžaduje rozsáhlý průzkum stavového prostoru, který má zajišťovat SOMA. Kvalitu výsledků je však třeba nějakým způsobem ověřit, a proto je nutné použít sekundární ODE simulaci.

Hodnota fitness funkce v bodě prozkoumávaném SOMA musí být nějakým způsobem konkrétně definována.

Pro její výpočet se používá aktuální stav robota v primární simulaci a to tak, že se do sekundární simulace vytvoří jeho kopie, nad kterou se provede alespoň jeden krok simulace (obr. 7.1).

Výsledný stav robota se dále zpracuje v závislosti na změnách jeho stavu.



Obrázek 7.1: Propojení SOMA s databází: Z primární databáze se okopíruje stav stroje a převede se do sekundární. SOMA pak vygeneruje kandidátní řešení, ze kterých se jedno vybrere a aplikuje na primární simulaci.

Kapitola 8

Databáze stavů

Protože pohyb po nohou je ve své podstatě periodický, je možné zaznamenávat stavy tak, že po určité době dojde k uzavření cyklu a simulovaný stroj se vrátí do některého z již prozkoumaných stavů.

Aby bylo možné uchovávat hodnoty stavů stroje, kterých již bylo dosaženo, je třeba použít strukturu schopnou stavy efektivně skladovat.

Stav zároveň obsahuje tyto hodnoty:

- ID
- stav stroje
- možné další cesty
- počet potomků
- počet neúspěšných pokusů
- předchozí stav

ID je hodnota jednoznačně určující stav pro jeho vyhledávání. Hodnota ID je číslo, které je s každým nově vytvořeným stavem inkrementováno.

Stav stroje přitom obsahuje všechny informace nutné k vytvoření přesné kopie simulátoru – pokud se tedy použije stejné nastavení a typ stroje.

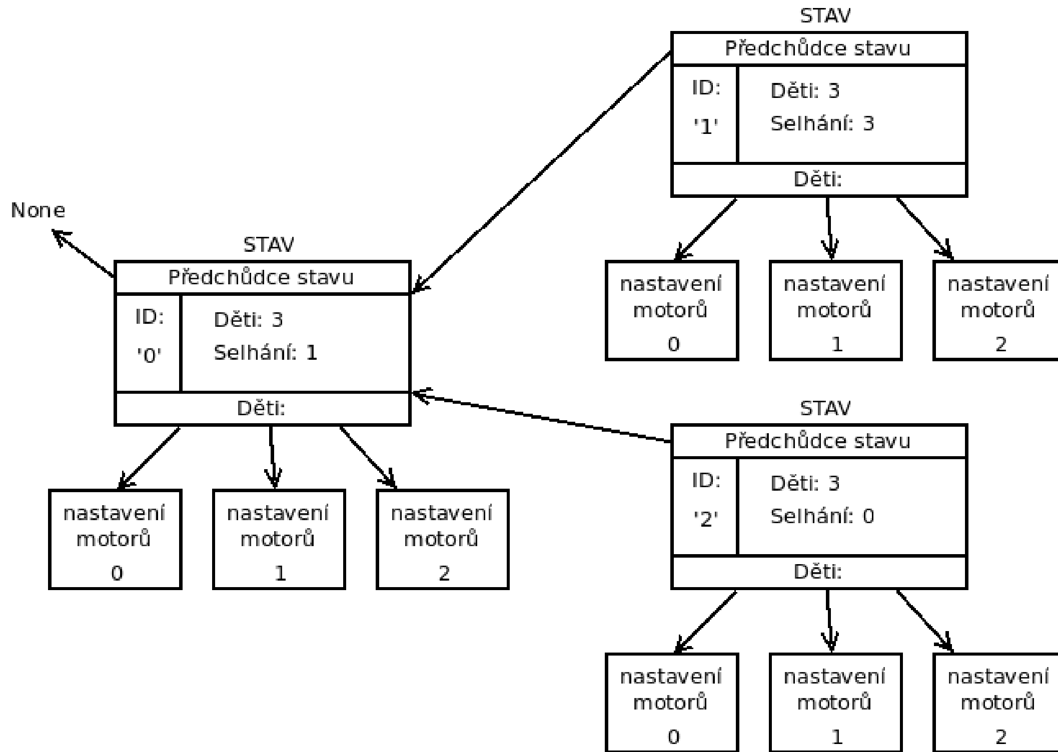
Možné další cesty jsou skupinou výsledků vrácených SOMA a obsahují informace o vhodných konfiguracích pro nastavení rychlostí jednotlivých motorů, tak, aby následující stav dosahoval maximální fitness hodnoty. Součástí každé konfigurace je i informace o dosažené fitness hodnotě, podle kterých je také jejich seznam seřazen.

Počet potomků udává celkové množství těchto konfigurací, které stav obsahuje.

Počtem neúspěšných pokusů je dána informace o množství prozkoumaných konfigurací, které selhaly, a vzhledem k seřazení konfigurací podle úspěšnosti číslo také jednoznačně určuje další větev, která má být prozkoumána. Tak je zajištěno prohledávání nejpravděpodobnějších cest jako prvních.

V případě, že selžou všechy další cesty se provede návrat simulace k předchozímu stavu, přičemž se jeho počet selhání také zvýší, aby zablokoval nyní již kompletně neúspěšnou větev (obr. 8.1).

Protože se v takovém případě musí být k čemu vrátit, je výhodné pamatovat si předchozí stav, ze kterého algoritmus naposledy přešel dále. Tato hodnota se však může měnit, jak bude vidět níže.



Obrázek 8.1: Databáze stavů: Každý stav má definovaného svého předchůdce, vyjímku tvoří počáteční stav. Zároveň si uchovává informace o stavu stroje, ze kterého vychází při hledání potomků.

Vzhledem k množství parametrů určujících stav jej není možné popisovat jednoznačně, ale je třeba použít jistou míru tolerance přesnosti, aby vůbec bylo možné najít jiný podobný stav.

Je-li stav S_0 definován n nezávislými čísly a α je tolerance, pak porovnání s libovolným stavem S_1 bude vyžadovat $5n$ atomických operací:

$$\bigwedge_{i=0}^{n}(S_0 \leq S_1 + \alpha \wedge S_0 \geq S_1 - \alpha) \tag{8.1}$$

Maximální počet oprací potřebných k nalezení podobného stavu v databázi o k prvcích pak bude definován jako:

$$\biguplus_{j=0}^{k}\left(\bigwedge_{i=0}^{n}(S_0 \leq S_k + \alpha \wedge S_0 \geq S_k - \alpha)\right), \tag{8.2}$$

kde $\biguplus$ je operátor, který sčítá jednotlivé atomické operace pro všechny průchody.

V našem případě to znamená pro databázi o více než 20 prvcích $k > 20$ a $n = 79$:

$$\Sigma > 20(79 \cdot 5) \tag{8.3}$$

$$\Sigma > 7900 \tag{8.4}$$

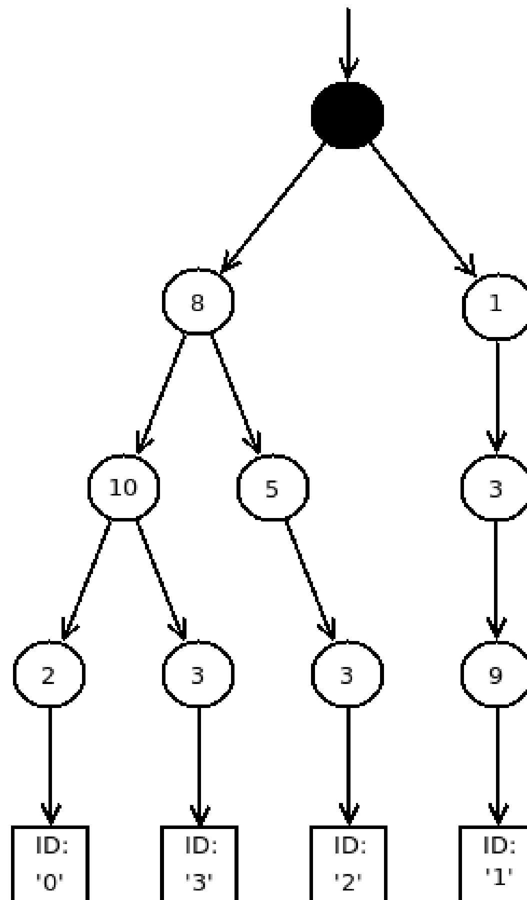
Tedy až 7900 atomických operací pro nalezení podobného stavu v malé databázi.

Tato hodnota je nepřípustná a je třeba ji nějakým způsobem zmenšit. I když se optimalizací základního počtu elementárních operací hodnota výrazně sníží, přesto bude Σ lineárně závislá na velikosti k. Proto je nutné použít jiný přístup.

8.1 Hashovací strom

Takový přístup nabízí hashovací strom. Pokud víme, v jakém oboru hodnot budou ležet prvky vstupní uspořádané n-tice, můžeme stavový prostor každého z prvků rozdělit na několik stejně velkých intervalů vymezených oborem hodnot.

Každý prvek tak může být snadno snadno určen a zařazen. Každý z intervalů rekurzivně opět obsahuje další skupinu intervalů určených pro zařazení dalšího z prvků vstupní n-tice.



Obrázek 8.2: Hashovací strom

8.1.1 Vkládání do hashovacího stromu

Máme-li vstupní vektor $\vec{i}$ o délce $|\vec{i}| = n$, který chceme vložit do hashovacího stromu, je nejprve nutné všechny hodnoty z $\vec{i}$ normalizovat na hodnoty v intervalech $\langle min_j; max_j \rangle$, kde $j \in \langle 0; n \rangle$.

K tomu použijeme následující vzorec, ze kterého získáme normalizovaný vektor $\vec{v}$:

$$\forall j : j \in \langle 0; n \rangle : v_j = \text{round}\left(\frac{100(i_j - min_j)}{max_j - min_j}\right), \tag{8.5}$$

Funkce `round` zaokrouhlí výsledek na celé číslo.

Podle procentuálního nastavení přesnosti a (v programu proměnná `accuracy`) se nyní provede celočíselné dělení vektoru $\vec{v}$ touto hodnotou:

$$\forall j : j \in \langle 0; n \rangle : w_j = v_j \text{ div } a \tag{8.6}$$

Výsledný vektor $\vec{w}$ je uspořádanou n-ticí „klíčů“ k jednotlivým větvím hashovacího stromu.

Každý klíč w_{i+1} tak vytvoří větev stromu vycházející z předchozí větve vytvořené klíčem w_i Do listového uzlu se nakonec umístí ID stavu, ze kterého pochází vstupní vektor $\vec{v}$.

Příklad: Uložení $\vec{v} = (-56; 27; 1)$, pro hodnoty ležící v intervalech $\langle -100; 0 \rangle$, $\langle 0; 50 \rangle$, $\langle 0; 10 \rangle$

Normalizací získáme vektor

$$\vec{v} = (44; 54; 10) \tag{8.7}$$

S přesností $a = 5$ se provede

$$44 \text{ div } 5 = 8 \tag{8.8}$$

$$54 \text{ div } 5 = 10 \tag{8.9}$$

$$10 \text{ div } 5 = 2 \tag{8.10}$$

Vektor klíčů je tedy $\vec{w} = (8; 10; 2)$

Na obrázku 8.2 je vidět strom, ve kterém jsou uloženy stavy:

- ID ’0’ = $(8; 10; 2)$
- ID ’1’ = $(1; 3; 9)$
- ID ’2’ = $(8; 5; 3)$
- ID ’3’ = $(8; 10; 3)$

8.1.2 Vyhledávání v hashovacím stromě

Nalézt podobný vektor v hashovacím stromě je velice rychlé. Stejným způsobem, jako při vkládání do stromu se získá vektor klíčů $\vec{w}$. Začne se od kořenu stromu a vyhledává se prvek se stejným klíčem w_1.

Je-li nalezen, pokračuje se jeho větví dále s následujícím prvkem z $\vec{w}$. V opačném případě hledání okamžitě končí neúspěchem.

Podaří-li se úspěšně nalézt všechny klíče, pak je nalezen podobný stav, jehož ID je umístěno v nalezeném listovém uzlu.

8.2 Databáze s hashovacím stromem

Pomocí hashovacího stromu se maximální počet porovnání po transformaci hodnot na klíče sníží na počet klíčů. Tedy, v našem případě se pro zjištění existence a nalezení podobného stavu provede maximálně 79 porovnání.

Jistou nevýhodou je skutečnost, že přesnost nedosahuje hodnot daných parametrem `accuracy`. Maximální odchylka od správné hodnoty může dosáhnout až hodnoty parametru.

Proto je třeba tento parametr nastavovat opatrně. Bližší informace jsou uvedeny v kapitole věnované nastavení.

8.2.1 Propojení databáze s hashovacím stromem

Aby bylo možné snadno přistupovat k uloženým stavům a využívat již vytvořené cesty, nejsou stavy přímo uloženy v hashovacím stromě. Místo toho jsou v listových uzlech hashovacího stromu uloženy pouze ID hodnoy jednotlivých stavů.

Přidání nového stavu do databáze zároveň automaticky přidává nový stav i do hashovacího stromu.

Kapitola 9

Průběh algoritmu

Kompletní algoritmus využívá všechny popsané části iterativně. Na počátku je však třeba provést inicializaci komponent. Nezbytné parametry se načtou z konfiguračního souboru, popřípadě z argumentů programu.

Inicializuje se databáze, připraví se vizualizace, vytvoří se primární simulace a umístí se do ní stroj. Jeho poloha záleží na tom, zda byla ze souboru načtena existující simulace, či nikoliv.

Následuje spuštění simulační smyčky:

Z primární simulace se pomocí metody `cMachine.getMachineSnapShot` získá aktuální stav a vyhledá se ve stromu stavů.

Neexistuje-li podobný stav, vytvoří se nový, přidá se do stromu a databáze stavů. Zároveň se tento stav stává aktuálním stavem.

Nyní se provede kontrola, zda již má tento stav nějaké potomky. V případě, že nemá, vytvoří se kopie stavu stroje, která se nahraje do sekundární databáze.

Nad sekundární databází se spustí SOMA, která nalezne a vrátí skupinu kandidátních řešení seřazených podle kvality pomocí opakování kroku sekundární simulace s rozdílnými vstupy.

Vrácené hodnoty označují nastavení motorů pro následující krok primární simulace, a to takové, aby výsledný stav dosahoval co nejlepších výsledků.

Pokud nějaká z hodnot vrácených SOMA klesne pod určitou hodnotu (je daná funkcí `cFitnessFunction.check`), pak je výsledek hodnocen jako kritický neúspěch.

Příkladem takového neúspěchu může být například stav robota, kdy se již nedá zabránit pádu.

Každý kritický neúspěch zvyšuje počet selhání aktuálního stavu o 1.

Všechny hodnoty jsou poté zaznamenány do aktuálního stavu jako možnosti k jeho další expanzi.

Nyní se spouští další smyčka, která kontroluje, zda počet potomků aktuálního stavu odpovídá počtu selhání.

Pokud ano, je tento stav celý závadný a provádí se rollback – vrácení k předchozímu stavu, kterému se též zvýší počet neúspěchů o 1. Tento krok se opakuje, dokud se nenalezne stav, který lze expandovat. Přitom se bere zřetel na možnost návratu až k výchozímu stavu.

Pokud nemá stav žádného předchůdce a selhaly mu všechny větve, skončí simulace neúspěchem. Také se provádí kontrola, zda rollbacking nevstoupil do stavu, který již je označen jako neúspěšný. V takovém případě vznikla při rollbackingu smyčka a simulace také končí neúspěšně.

Jinak, pokud je počet selhání nižší než počet potomků, se vybere k expanzi potomek v pořadí odpovídající počtu selhání. Potomci jsou totiž indexováni od 0 a také seřazeni podle kvality. Ti s největší pravděpodobností kvalitních potomků jsou vybíráni dříve.

Nyní se provede expanze vybraného potomka – provede se krok primární simulace s jeho hodnotami a poté se celý cyklus znovu opakuje.

9.1 Použití

Principem řídícího algoritmu je tedy rozšiřování databáze předchozích stavů. Protože chůze je ve své podstatě periodicky se opakující se pohyb, lze vytvořit databázi, kam se budou ukládat předchozí dosažené hodnoty, a kterých se později využije pro účely predikce následujícího stavu.

Další rozšiřování databáze stavů bude učit stroj další pohyby. Zde záleží na nastavení parametru `accuracy`. Příliš malé hodnoty s vysokou požadovanou přesností nevytvoří smyčku, naopak vysoká čísla jsou schopná nalézt podobnost až příliš snadno a dojde k aplikaci naprosto nevhodných hodnot. Hodnota by měla ležet někde v intervalu $\langle 5; 25 \rangle$.

Jakmile je nalezen podobný stav, použijí se jeho doporučené hodnoty. Získaný nový stav však nemusí odpovídat původnímu, do kterého se algoritmus dostal, když vytvářel tento stav. Bude však s velkou pravděpodobností ve stavovém prostoru nedaleko a bude také schopen původní „cestu“ znovu protnout.

Skutečným konečným výstupem algoritmu není tedy jedna smyčka v databázi, ale jejich skupina navzájem se podporující a protínající, schopná použít již vytvořenou cestu i v případě nepravidelností způsobených vnějšími vlivy.

Pomocí algoritmu jsou takto vybírány použitelné části stavového prostoru.

Kapitola 10

Měření úspěšnosti aproximace řešení

Při použití softcomputingových metod jako je SOMA je vždy nutné provádět měření úspěšnosti nalezené aproximace řešení. Protože se řešení hledá v omezeném prostoru, definuje se n-rozměrná funkce, která musí být nad tímto prostorem v každém bodě nějakým způsobem definována.

Každá hodnota z n přitom představuje jeden nastavitelný parametr, který ovlivňuje kvalitu řešení. V našem případě se jedná o nastavení rychlosti (a směru daném znaménkem) motoru. Různé nastavení rychlostí všech motorů tak definuje jeden bod v hyperprostoru. Funkční hodnota v tomto bodě se nazývá *fitness hodnota* a popisuje kvalitu řešení, kterého je dosaženo při použití parametrů definujících bod. V různých aplikacích se fitness používá různým způsobem – vyšší hodnota fitness může znamenat lepší výsledek, ale stejně tak může také znamenat výsledek horší.

V našem případě platí, že čím nižší hodnota fitness, tím blíže je výsledek hledanému optimu.

Správné sestavení a výpočet tvaru funkce je pravděpodobně nejdůležitějším prvkem celé SOMA (i dalších evolučních algoritmů), protože přímo ovlivňuje požadované chování algoritmu. Obvyklým postupem je použít více jednodušších funkcí, jejichž výsledky se nakonec sečtou, a tak vytvoří společný výsledek:

$$\mathrm{f}(\vec{x}) = \Sigma_{i=1}^{j} k_i \cdot \mathrm{f}(x_i), \tag{10.1}$$

kde $j = |\vec{x}|$ je počet jednodušších funkcí a $\vec{k}$ je vektor jejich vah. Důležitost některých částí výpočtu totiž není ekvivalentní a pomocí vah lze poměr důležitosti mezi nimi snadno upravovat.

Výpočet fitness je také časově a výpočetně nejvíce náročnou částí, navíc je tento výpočet prováděn mnohokrát během jediné iterace algoritmu.

10.1 Hodnoty potřebné pro výpočet fitness

Přímo měřitelné parametry se přepočítávají na výstupní fitness funkci a to tak, aby vyšších hodnot bylo dosaženo v případě, že stroj lépe plní své cíle.

Hodnoty aktuálního stavu při plnění cíle nebo cílů bude třeba získat z parametrů, v našem případě se jedná o přesný popis stavu stroje.

Pro každou součást stroje je tedy třeba znát následující hodnoty:

- pozice – pozice vzhledem k nulové souřadnici simulátoru
- rotace – natočení části vzhledem k osám simulátoru
- rychlost – vektor rychlosti posunu středu tělesa vzhledem k simulátoru
- rychlost rotace – rychlost otáčení podle všech os kolem středu tělesa vzhledem k simulátoru

Hodnoty určující klouby lze vynechat, neboť jejich umístění je automaticky upravováno podle pozic těles, na které jsou klouby napojeny.

Dále lze využít výchozího stavu sekundární simulace - pro výpočet relativních změn. Protože je Z souřadnice gondoly z důvodů vyžadovaných databází stavů a také zobrazováním udržována na hodnotě nula, je relativní posunutí kupředu použito pro výpočet fitness hodnoty rychlosti.

Vzhledem k povaze algoritmu je nastavení tvaru jednotlivých jednoduchých funkcí voleno tak, aby za kritický núspěch byl považován součet hodnot, který je větší než 0.

Nyní lze přímo stanovit cíle stroje, které musí plnit během svého pohybu kupředu.

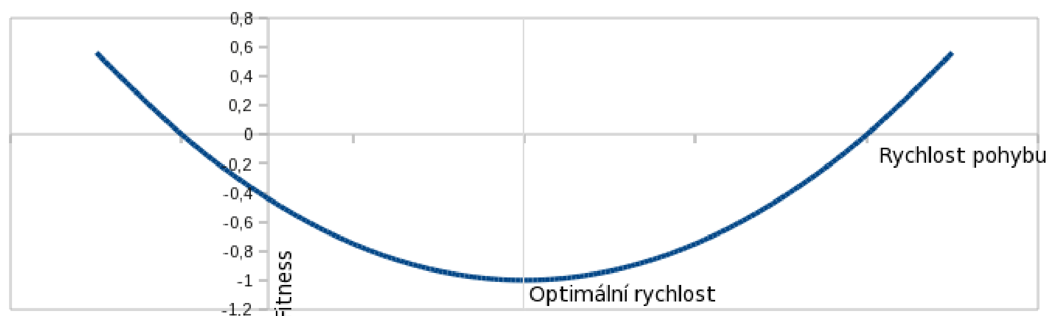
10.2 Rychlost

Úkolem práce je naučit dvounohého robota chodit – tedy je třeba zajistit, aby stroj udržoval stálý pohyb kupředu. Čím vyšší rychlost, tím lepší fitness. Protože čím vyšší rychlost, tím lépe, můžeme prozatím ponechat přímou úměru rychlosti na kvalitě fitness. Bude-li třeba dosáhnout nějaké konkrétní rychlosti, můžeme změnit tvar křivky z lineárního poklesu na např. parabolu s vrcholem u požadované rychlosti (obr. 10.1).

Přílišná rychlost může být na škodu, protože robot nebude schopen dostatečně agilně reagovat na nově vznikající stavy, protože má pouze omezenou maximální rychlost pohybu končetin.

Stejně tak je nutné zohlednit stavy, kdy je možné, že se gondola bude muset pohnout mírně vzad, aby se zabránilo pádu. Nízká hodnota může být kompenzována lepšími výsledky z ostatních jednoduchých funkcí.

Výpočet rychlosti se provádí pomocí rozdílů hodnot Z-ových souřadnic, ze stavu před a po provedení kroku.



Obrázek 10.1: Průběh části fitness funkce dané rychlostí pohybu.

10.3 Výška gondoly

Dále je třeba, aby stroj zachovával vzpřímenou polohu. Algoritmus založený na evolučních principech totiž snadno nalezne lokální extrém, který zde představuje stav s minimální potenciální energií, kdy se gondola dotýká země a stroj se pouze odstrkuje nohami. Proto je druhým atributem určujícím fitness i výška gondoly nad zemí.

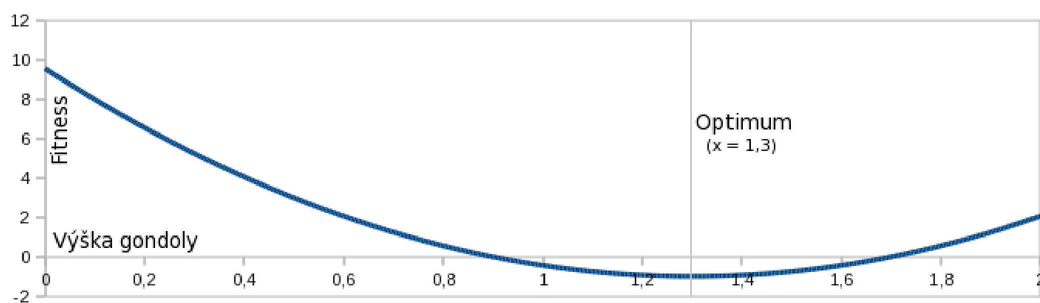
Tvar této funkce bude třeba upravit do vhodného tvaru, aby byla gondola udržována v optimální výšce, ale zároveň byla možná i situace, kdy je snížení těžiště stroje vyžadováno. Snížení však nesmí překročit bezpečnou mez. Taková situace už se bude klasifikovat jako pád na zem a bude třeba provést restart.

Zde by bylo třeba fitness hodně strhnout, ale ukázalo se, že to není vhodné. Pokud se v průběhu fitness funkce objeví rovina, pak jedinci snadněji ztratí směr, kudy se vydat a SOMA vrací velice špatné hodnoty.

Kvůli této skutečnosti je průběh funkce zachován a pád je řešen pomocí funkce `cFitnessFunction.check()`.

Hodnota je vypočtena z aktuální výšky geometrického středu gondoly od země (obr. 10.2).

Této části fitness hodnoty je přiřazena velká váha. Důvodem je skutečnost, že gondola by se měla udržovat v pokud možno klidném stavu a je třeba mít možnost „přehlasovat“ ostatní části fitness, pokud se jedná o předejití pádu.



Obrázek 10.2: Průběh části fitness funkce dané výškou gondoly

10.4 Směr gondoly

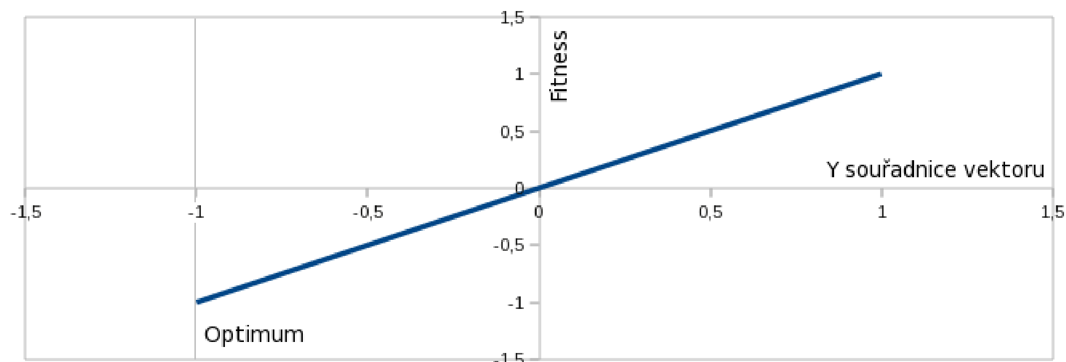
Protože gondola bude obsahovat vitální součásti stroje, je třeba ji udržovat v klidu, aby se zabránilo přílišným otřesům. Dalším důvodem je skutečnost, že hmotnost gondoly tvoří největší podíl hmoty stroje a malé výkyvy ústí k velkým změnám polohy těžiště.

Zajištění gondoly v klidu je provedeno pomocí fitness funkce, která jako vstup bere zápornou hodnotu souřadnice Y normalizovaného vektoru $\vec{R} = (0; 1; 0)$ vztaženého na rotaci gondoly. Vektor vztažený na gondolu vždy míří kolmo vzhůru vzhledem k její horizontální rovině.

Ve vztahu k absolutním souřadnicím a s obráceným znaménkem, má $\vec{R}$ hodnotu odpovídající přiřazené fitness – hodnota dosahuje minima (-1) právě tehdy, když je gondola v horizontální poloze (obr. 10.3).

Tímto způsobem je zajištěno, že gondola má tendenci setrvávat v klidu.

Nežádoucí změny v poloze těžiště jsou podstatným problémem, proto je i váhu této části fitness funkce nastavit jako vysokou.



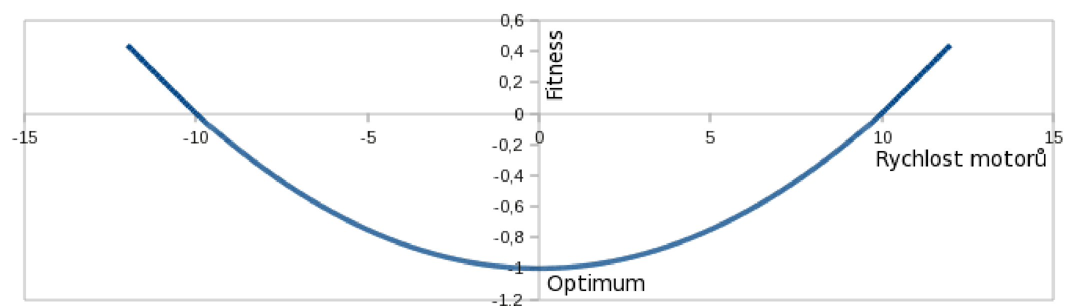
Obrázek 10.3: Směr gondoly: Lineární průběh zajistí nejlepší výsledek tehdy, je-li hodnota -1 – tedy gondola vertikálně.

10.5 Rychlost motorů

První simulace ukázaly, že algoritmus má tendence provádět příliš mnoho zbytečných pohybů, které sice přinášejí určitou míru stability, ale v delším časovém úseku se toto chování nevyplácí.

Toto kompenzuje další přidaná funkce, jejíž úkolem je penalizovat vyšší rychlosti motorů (obr. 10.4). Míra postihů je nastavena jako velmi malá, neboť by jinak bránila stroji i v potřebných pohybech. Tomu odpovídá i nízká váha přiřazovaná výsledku.

Křivka definující tento pohyb je opět parabola, tentokrát s vrcholem, kterým prochází osa Y.

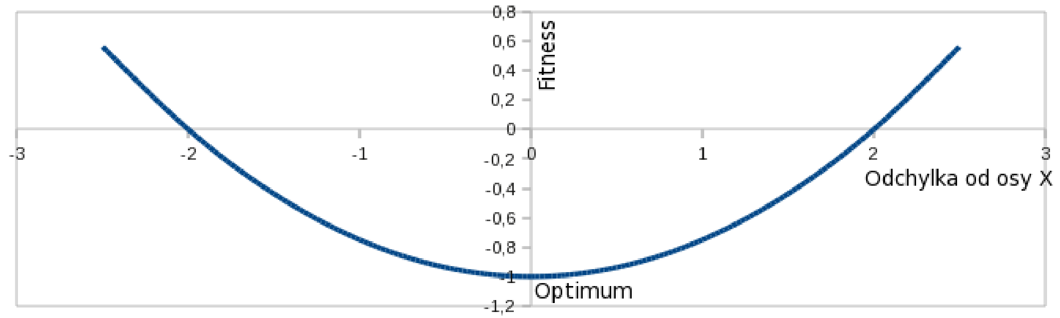


Obrázek 10.4: Rychlost motorů: Pozvolný průběh křivky zajišťuje malou míru postihů, přesto dostatečnou.

10.6 Pozice ve směru osy X

Důležitým prvkem je také snažit se zabránit stroji v přílišném pohybu do stran. Nejedná se o požadovanou vlastnost a navíc omezení pohybu stroje v tomto směru sníží pravěpodobnost zapojování dvojice motorů v pánvi stroje, které pohyb do stran způsobují.

Stejně jako u rychlosti stroje kupředu je i zde pozice vypočtena pomocí rozdílů hodnot X-ových souřadnic ze stavu před a po provedení kroku (obr. 10.5).



Obrázek 10.5: Pozice ve směru X: Vyšší odchylky znamenají vyšší penalizaci.

Výpočetní náročnost kroku sekundární simulace je vysoká a připočteme-li počet jednotlivých částí fitness funkce, je zřejmé, že se jedná o časově velmi nákladnou část výpočtu.

Protože je výpočet prováděn mnohokrát každou iteraci algoritmu, je zde třeba dobře optimalizovat kód.

Kapitola 11

Popis použitých struktur

11.1 Soubory

Zdrojové kódy napsané v jazyce Python jsou umístěny v kořenovém adresáři projektu a jeho podadresářích. Jedná se o adresáře:

- `simulation` – obsahuje soubory týkající se simulace a její vizualizace
- `soma` – adresář se soubory bezprostředně se týkajících SOMA
- `utils` – adresář se soubory obsahujícími podpůrné metody

Jednotlivé soubory jsou pak tyto:

- `main.py` – soubor obsahující hlavní smyčku programu; umístěn v kořenovém adresáři projektu
- `database.py` – zde jsou umístěny struktury týkající se hashovacího stromu a databáze stavů; umístěn v kořenovém adresáři projektu
- `somaATE.py` – knihovna pro SOMA – All To Elite; umístěn v adresáři `soma`
- `config.py` – je konfigurační soubor s hlavními nastaveními; umístěn v kořenovém adresáři projektu
- `machine.py` – obsahuje simulátor a vizualizaci; umístěn v adresáři `simulation`
- `fitness_func.py` – zde je definována fitness funkce používaná SOMA; umístěn v adresáři `soma`
- `debug.py` – soubor s nastaveními pro strukturovaný výpis na obrazovku; umístěn v adresáři `utils`

Součástí programu jsou i soubory `__init__.py`, které jsou obsaženy v každém podadresáři projektu. Tyto souboru neobsahují žádný kód, Python je pouze používá pro informaci, zda se v adresáři nacházejí soubory pro import.

11.2 Popis souborů

11.2.1 debug.py

V adresáři `utils` je umístěn soubor `debug.py`, který obsahuje třídu `cDebug`.

Tato třída je určena k formátování textového výstupu na obrazovku do přehlednější podoby. Protože je k výpisům přidáváno i jméno metody a umístění v souboru, je importována knihovna `inspect`.

Třída obsahuje tyto metody:

- `__init__()` – standardní inicializační metoda objektu, obsahuje nastavení logovacích masek
- `lineNo()` – metoda je nastavena tak, aby při zavolání výpisu vrátila číslo řádku, na kterém je výpis volán
- `currentFile()` – metoda je nastavena tak, aby při zavolání výpisu vrátila jméno souboru, ze kterého je výpis volán
- `putLog(target, cut, message, value = -1)` – metoda, která provádí tisk výpisu na obrazovku. Parametry:
  - `target` – typ výpisu (info, debug, warning, error)
  - `cut` – pokud nastaveno na 1, je výpis zkrácen
  - `message` – text, který má být tisknut
  - `value` – závažnost informace (1 – 4 vzestupně podle důležitosti)
- `log()` – hlavní metoda zajišťující výpis zpráv

Jednotlivé typy výpisů jsou barevně rozlišeny pro snadné rozeznání.

Soubor dále obsahuje vytvoření objektu `dbg` třídy `cDebug`, který se poté používá jako singleton:

```
dbg = cDebug()
```

Příklad použití:

```
dbg.log('Movement distance: %s' %(deltaZ), info=2)
```

11.2.2 fitness_func.py

V adresáři `soma` je umístěn soubor `fitness_func.py`, který obsahuje třídu `cFitnessFunction`.

Třída obsahuje tyto metody:

- `__init__()` – standardní inicializační metoda objektu, obsahuje inicializaci sekundárního simulátoru Také se zde inicializuje aktuální stav stroje pro sekundární simulaci a mezní hodnoty pro jednotlivá tělesa.
- `getCoef(x, b)` – metoda předpočítává parametry z konfiguračního souboru tak, aby se daly použít pro výpočet

- `setSimulation(snapshot, dt)` – metoda nastavuje nový stav stroje pro zpracování a také velikost kroku `dt`.
- `check(result)` – metoda vrací 0, jestliže je hodnota parametru nižší, než je stanovená mez. Jinak vrací 1. Metoda slouží k detekci kritických selhání sekundární simulace.
- `f(a)` – hlavní metoda, která provádí vlastní výpočet fitness hodnoty. Jejím parametrem je n-tice vstupů, zde směr a velikost síly rotace motorů.

Soubor dále obsahuje vytvoření objektu `fitFunc` třídy `cFitnessFunction`, který se poté používá jako singleton:

```
fitFunc = cFitnessFunction()
```

11.2.3 somaATE.py

V adresáři `soma` je umístěn soubor `somaATE.py`, který obsahuje třídu `cSomaATE`.

Třída obsahuje tyto metody:

- `__init__(function, limits, unitCount, maxAge = 0)` – standardní inicializační metoda objektu, obsahuje inicializaci SOMA, nastavení parametrů a provede první výpočet elity; Parametry:
  - `function` – ukazatel na funkci, která počítá fitness
  - `limits` – rozměry stavového prostoru
  - `unitCount` – počet jedinců, kteří budou umístěni na hyperplochu
  - `maxAge` – maximální počet iterací, po kterém bude jedinec respawnován
- `step(function = None)` – metoda provede jednu iteraci algoritmu Volitelný parametr může vložit jinou fitness funkci.
- `setUnits(units)` – jedinci vložení parametrem nahradí některé z již vygenerovaných neelitních jedinců

Při vytvoření objektu třídy `cSomaATE` se provede inicializace počátečních hodnot. Dále už se volá pouze metoda `step`, která vždy provede jednu iteraci a vrátí její nejlepší výsledky. Každé další zavolání zpřesňuje nejlepší dosud nalezené řešení. (Vzdálenost výsledných hodnot od skutečného optima má v případě statické funkce neklesající průběh.)

11.2.4 database.py

V kořenovém adresáři projektu je umístěn soubor `database.py`, který obsahuje třídy `cState` a `cDatabase`.

Třída `cState` obsahuje metodu:

- `__init__()` – standardní inicializační metoda objektu

Třída `cState` kromě inicializace nemá jiné metody a v programu se používá jako struktura pro ukládání dat.

Třída `cDatabase` obsahuje metody:

- `__init__(limits)` – standardní inicializační metoda objektu; Parametry:
  - `limits` – maximální a minimální hranice hodnot, kterých může stav dosáhnout, nutné pro normalizaci
- `normalize(position)` – normalizuje stav zadaný parametrem
- `hashId(id)` – přidá stav již uložený v databázi do hashovacího stromu; Parametry:
  - `id` – jednoznačný identifikátor stavu uloženého v databázi
- `addState(snapshot)` – přidá do databáze nový stav definovaný aktuálním stavem stroje
- `findClose(snapshot)` – nalezne v databázi stav podobný aktuálnímu stavu stroje

11.2.5 machine.py

V adresáři `simulation` je umístěn soubor `machine.py`, který obsahuje třídy `cWorld` a `cMachine`.

Třída `cWorld` obsahuje metody:

- `__init__(visible = 0)` – standardní inicializační metoda, vytvoření simulovaného světa
- `initVisible()` – inicializuje grafický výstup
- `clearScreen()` – smaže obrazovky grafického výstupu
- `addBox(name, mass, x, y, z, pos)` – přidá do simulace kvádr s parametry:
  - `name` – jméno objektu, jednoznačně ho identifikující
  - `mass` – hustota objektu
  - `x` – šířka tělesa
  - `y` – výška tělesa
  - `z` – hloubka tělesa
  - `pos` – pozice tělesa v absolutních souřadnicích
- `addHinge(name, target1, target2, anchor, axis)` – přidá do simulace kloub typu pant s parametry:
  - `name` – jméno kloubu, jednoznačně ho identifikující
  - `target1` – jméno prvního tělesa, ke kterému je kloub napojen
  - `target2` – jméno druhého tělesa, ke kterému je kloub napojen
  - `anchor` – bod, kterým prochází osa, kolem níž se kloub ohýbá
  - `axis` – nastavení směrového vektoru osy
- `near_callback(args, geom1, geom2)` – metoda volaná tehdy, když simulátor detekuje kolizi dvou objektů; Parametry:
  - `args` – definovatelné argumenty

  - `geom1` – první kolidující geom
  - `geom2` – druhý kolidující geom

- `coord1(x, y)` – metoda převede dané koordináty na souřadnice vykreslitelné na výstup (vlevo)
- `coord2(x, y)` – metoda převede dané koordináty na souřadnice vykreslitelné na výstup (vpravo)
- `draw(id)` – metoda, která vykreslí těleso, jednoznačně určené parametrem
- `pygameStep()` – provede krok vizualizace: prohodí buffery a provede posun času
- `step()` – provede krok simulace: vykreslí tělesa, vypočte kolize a posune simulační čas

Třída `cMachine` obsahuje metody:

- `__init__(world)` – standardní inicializační metoda, vytvoření simulovaného světa
- `getBodySnapShot(body)` – metoda vrací informace nutné k vytvoření kopie nastavení tělesa `body`
- `getMachineSnapShot()` – metoda vrací informace nutné k vytvoření kopie nastavení celého stroje
- `setBodySnapShot(body, snapshot)` – metoda nastavuje parametry tělesa (rychlosti a rotace)
  - `body` – hodnota jednoznačne identifikující těleso
  - `snapshot` – výsledek metody `getBodySnapShot`
- `setMachineSnapShot(snapshot)` – metoda nastavuje parametry stroje (rychlosti a rotace); Parametry:
  - `snapshot` – výsledek metody `getMachineSnapShot`
- `engage(joint, velocity)` – metoda spouštění motoru; Parametry:
  - `joint` – jednoznačné určení kloubu (= motoru)
  - `velocity` – rychlost motoru (znaménko ovlivňuje směr)
- `engageMotors(speeds)` – metoda nastavení všech motorů na rychlosti dané parametrem (list hodnot)
- `setVisible(v)` – nastavení viditelnosti těles; parametr může nabývat hodnot 0 a 1.
- `moveToZero(snapshot)` – přesun stroje tak, aby střed gondoly měl X a Z souřadnici 0

11.2.6 main.py

`main.py` je spustitelným souborem projektu a obsahuje metody:

- `usage()` – vypíše na standardní výstup použití programu
- `main()` – hlavní metoda, inicializující ostatní třídy a spouštějící hlavní smyčku programu

Kapitola 12

Nastavení

Součástí programu je i snadno přístupný soubor `config.py` umístěný v kořenovém adresáři projektu.

Soubor obsahuje nastavení parametrů programu.

12.1 Parametry logování

Tyto parametry pouze vypisují informace na standardní výstup a nijak přímo neovlivňují běh programu.

- `dbg.info` – míra podrobnosti logovacích informací informačního charakteru
- `dbg.debug` – míra podrobnosti logovacích informací debugovacího charakteru
- `dbg.error` – míra podrobnosti logovacích informací o chybách
- `dbg.warning` – míra podrobnosti logovacích informací o varováních

Hodnota, která je těmto parametrům přiřazena ovlivňuje, které informace se mají vypsat, přičemž 1 jsou nejméně významné hodnoty a 4 nejvíce. Nastavení parametru na 4 tak vypíše pouze nejzávažnější informace, 1 vypíše všechny.

Pokud se parametry nastaví na 1 (zejména `dbg.info`), bude velké množství výpisů na obrazovku zpomalovat algoritmus.

Standardně jsou nastaveny tyto hodnoty:

```
dbg.info = 2
dbg.debug = 1
error = 1
warning = 1
```

Poznámka: Pro zobrazování výpisů je obvyklý formát:

```
    <typ informace>(<závažnost>),
    file:„<cesta k˜souboru, kde byla informace získána>\
    at <řádek>:
„<text zprávy>\
```

Typ informace je v linuxové konsoli zobrazován barevně pro snadnější vyhledávání.

12.2 Nastavení fitness funkce

SOMA algoritmus pro výpočty potřebuje vypočtat hodnotu fitness, kterou získá provedením kroku sekundární simulace. Hodnocení kvality výsledků nastavují tyto parametry:

- `optimalHeight` – optimální výška gondoly
- `heightX0` – bod, ve kterém křivka výšky gondoly protne osu X
- `optimalSpeed` – optimální rychlost kupředu
- `speedX0` – bod, ve kterém křivka rychlosti kupředu protne osu X
- `motorsX0` – bod, ve kterém křivka rychlostí motorů protne osu X
- `sideX0` – bod, ve kterém křivka odchylky pohybu do strany protne osu X

Standardně jsou nastaveny tyto hodnoty:

```
optimalHeight = 1.3
heightX0 = 1.25
optimalSpeed = 0.1
speedX0 = -0.01
motorsX0 = 30.0
sideX0 = 0.2
```

Výška gondoly je počítána od jejího geometrického středu. Příliš vysoká nebo příliš nízká výška získává horší ohodnocení, optimum je dáno parametrem `optimalHeight`. `heightX0` definuje bod, ve kterém výsledek začíná negativně ovlivňovat výslednou fitness hodnotu. Analogicky tak platí i u ostatních parametrů. `motorsX0` a `sideX0` mají optima předem nastavené na $X = 0$.

12.3 Parametry simulátoru

Simulátor umožňuje nastavit některé hodnoty tak, aby se lépe přizpůsobil potřebám uživatele.

- `maxIterations` – počet iterací, po kterém se algoritmus ukončí
- `visualization` – zda je zapnuté okno s vizualizací primární simulace
- `dt` – délka kroku simulace v sekundách

Maximální množství iterací odpovídá počtu různých stavů, kterými algoritmus projde. Pokud se databáze načte ze souboru, algoritmus začíná z počátečního stavu a rychle projde až k zatím nalezenému stavu. Cesta, kterou mezitím projde se také počítá do celkového počtu iterací.

Hodnotu `maxIterations` lze přetížit parametrem z příkazové řádky.

Parametr `visualization` akceptuje hodnotu 0 nebo 1, podle toho, zda uživatel chce zobrazovat grafický výstup. Vypne-li se nastavením hodnoty na 0, pak výpočet poběží o něco rychleji. Toto má však význam, pouze použije-li SOMA malé množství jedinců, jinak doba zobrazování v době trvání jedné iterace nehraje podstatnou roli.

Parametr `dt` nastavuje dobu, po jakých skocích simulátor pracuje a na jakou dobu kupředu je vypočtena hodnota nového stavu stroje v sekundární simulaci.

Nastavení na delší intervaly než cca 0,1s způsobuje poruchy v simulátoru, který pak nepřesně spočítá místa a časy kolizí mezi tělesy, a proto jej nedoporučuji.

Standardní nastavení parametrů simulátoru:

```
maxIterations = 10
visualization = 1
dt = 1.0/25.0
```

12.4 Nastavení SOMA

SOMA jako klíčový prvek systému umožňuje nastavit řadu parametrů. Většinu z nich však není třeba měnit, proto k nim není v konfiguračním souboru přístup.

Parametry, které nejvíce ovlivňují výpočet jsou:

- `units` – množství použitých jedinců
- `elite` – počet vrácených kandidátů
- `maxAge` – počet kol SOMA, po kterém se jedinci respawnují
- `somaSteps` – počet iterací SOMA na jednu iteraci celého algoritmu

Množství použitých jedinců nejvíce ovlivňuje rychlost algoritmu. Každý jedinec standardně provede 14 výpočtů fitness v každé iteraci SOMA, několik dalších výpočtů je provedeno při kontrole hodnot elity.

Každý výpočet přitom představuje provedení kroku sekundární simulace.

Počet iterací SOMA na iteraci algoritmu je parametr, který výpočet ovlivňuje stejně jako počet jedinců. Rozdílem je skutečnost, že další iterace SOMA vychází z té předchozí, a tedy hlavně zpřesňuje již nalezené řešení.

Celkový počet provedených kroků sekundární simulace S můžeme vypočíst jako:

$$S = 14 \cdot u \cdot i, \tag{12.1}$$

kde u je počet jedinců a i počet iterací SOMA.

Dvě iterace SOMA algoritmu pro 20 jedinců tedy provedou cca 560 kroků sekundární simulace.

Parametr `maxAge` umožňuje předcházet nahloučení jedinců kolem lokálního extrému tím, že jedinci jsou po uplynulém počtu iterací náhodně přesunuti na jiné souřadnice. Pokud je hodnota 0, pak k respawnu nedochází. Nemá smysl používat nenulovou hodnotu pro malý počet iterací.

Počet vrácených kandidátů rychlost simulace prakticky neomezuje. Jakmile jeho hodnota přesáhne polovinu z celkového počtu jedinců, algoritmus nemusí pracovat správně. Optimální rozmezí počtu elitních jedinců je $\langle 2; \frac{\text{units}}{4} \rangle$.

Standardní nastavení parametrů SOMA:

```
units = 50
elite = 3
maxAge = 0
somaSteps = 2
```

12.5 Nastavení databáze a hashovacího stromu

Důležitým parametrem, který ovlivňuje databázi a hashovací strom je tolerance, s jakou je kandidátní stav vyhodnocen jako podobný nějakému jinému z databáze.

- `accuracy` – míra tolerance při určování podobnosti

Zatímco příliš nízká hodnota tolerance způsobí, že téměř není možné nalézt podobný stav, příliš vysoké hodnoty způsobí nedostatečnou přesnost vypočtených hodnot a zvýší tak pravděpodobnost selhání.

Standardní hodnota tolerance:

```
accuracy = 10
```

Množství parametrů zvyšuje variabilitu algoritmu, jejich správné nastavení pro optimální funkčnost je však věcí delšího experimentování.

Kapitola 13

Ukládání a načítání stavu simulace

Simulátor své zatím dosažené výsledky uchovává ve vlastní databázi, která ale není perzistentní. Algoritmus učení přitom probíhá velmi dlouho, a tak je nutné mít možnost nějakým způsobem uchovávat zatím dosažené výsledky a poté mít možnost je znovu načíst.

Proto je součástí implementace i část, která pravidelně ukládá zatím dosažené výsledky do souboru.

13.1 Modul pickle

Modul `pickle` obsahuje základní algoritmy potřebné k reprezentaci objektů jazyka Python takovým způsobem, aby je bylo možné uložit do textového souboru a opět je z něj načíst.

Pro tyto postupy se používají pojmenování „*serializace*“ a „*deserializace*“ ačkoliv v dokumentaci k modulu se objevují termíny „*pickling*“ a „*unpickling*“.

Modul je schopen serializovat i hierarchicky náročné objekty, což je vhodné. Práce s modulem je navíc velmi jednoduchá a přehledná.

Pro použití modulu se importuje modul `pickle`:

```
import pickle
```

Objekt se serializuje pomocí metody `pickle.dump`, která jako druhý parametr požaduje file descriptor.

Příklad: Ukládání objektu `db` do souboru `db_dump.obj`.

```
file = open('db_dump.obj', 'w')
pickle.dump(db, file)
file.close()
```

K opětovné deserializaci se používá příkaz `pickle.dump` s parametrem typu file descriptor. Metoda jako výsledek vrací odkaz na objekt.

Příklad: Načítání objektu `db` ze souboru `db_dump.obj`.

```
file = open('db_dump.obj', 'r')
db = pickle.load(file)
file.close()
```

13.2 Použití modulu pickle v aplikaci

Veškeré informace o průběhu algoritmu jsou primárně uloženy v objektu typu `cDatabase`, který zastřešuje i objekt hashovacího stromu a tím ve své hierarchii obnáší kompletní relevantní informace o dosavadním průběhu algoritmu.

13.2.1 Ukládání dat

Cílový soubor, kam se má průběh ukládat, je určen parametrem z příkazového řádku.

Vzhledem k povaze aplikace, která běží iterativně, je ukládání prováděno na konci každé z nich. Tím je také zabráněno ztrátě dat.

Ukládání dat je implicitně deaktivované. Používá se jen tehdy, je-li parametrem programu specifikován cílový soubor.

13.2.2 Načítání dat

Je-li programu zadán jako vstupní parametr soubor, ze kterého se má načítat, pak se ze souboru pomocí modulu importují data. Před začátkem první iterace se provede automaticky nastavení stroje do výchozího stavu.

Algoritmus poté prochází již vygenerovanými stavy – a tedy již vykonává naučenou cestu – dokud se stroj nedostane do nového stavu, který opět zakomponuje do databáze.

Kapitola 14

Použití a výsledky testování programu

14.1 Použití

Program je napsán pod operačním systémem Linux a spouští se jako konsolová aplikace pomocí příkazu:

```
./main.py
```

Takto spuštěný program provede výpočet podle nastavení v konfiguračním souboru `config.py`.

Další nastavení lze provést pomocí argumentů programu. Jejich stručný popis lze získat spuštěním programu s parametrem `-h` nebo `--help`:

```
./main.py --help
```

V takovém případě program vypíše použití a skončí.

Pomocí argumentu programu je také možné přetížit počet provedených iterací:

```
./main.py -r 1000
```

nastaví počet iterací algoritmu na 1000.

Dalšími parametry mohou být vstupní a výstupní soubory:

```
./main.py -i in.obj
```

nebo

```
./main.py --input==in.obj
```

Oba příkazy použijí jako vstup soubor `in.obj`

Výstupní soubor se pak zadává jako:

```
./main.py -o in.obj
```

nebo

```
./main.py --output==in.obj
```

Parametry je možné kombinovat a používat v libovolném pořadí. Je-li jako vstup i výstup zadán stejný soubor, pak je vstupní soubor přepsán novým výstupem.

14.2 Testování

Vzhledem k povaze práce bylo provedeno testování, které vyhledává výsledky závislé na parametrech nejvíce ovlivňující výsledky.

Protože není dobře možné porovnávat výsledky po více iteracích, kdy se stavy robotů velice liší, jsou získané výsledky průměry z hodnot získaných z průběhu padesáti prvních iterací.

Parametry, které přímo neovlivňují algoritmus, byly nastaveny na empiricky získané hodnoty.

```
optimalHeight = 1.3
heightX0 = 1.25
optimalSpeed = 0.1
speedX0 = -0.01
motorsX0 = 30.0
sideX0 = 0.2
dt = 1.0/25.0
accuracy = 10
```

Zároveň bylo upraveno i nastavení vah jednotlivých fitness funkcí tak, aby přílišná váha některé z nich nebránila v dostatečném prosazení ostatních. Váhy byly nastaveny takto:

- Výška gondoly: $w = 2,5$
- Pohyb v ose X: $w = 3,0$
- Natočení gondoly: $w = 5,0$
- Rychlost motorů: $w = 0,5$
- Pohyb kupředu: $w = 0,7$

V tomto případě se jedná o poměrně labilní stav, kdy například zvýšení váhy udržující střed gondoly v nastavené výšce přesáhlo míru postihu vzniklého naklopením gondoly, a tedy stroj získal tendenci okamžitě po začátku simulace sklápět gondolu.

Dalším případem je i udržování stálé X-ové pozice, kde nedostatečná váha umožní stroji nežádoucí pohyby.

S uvedeným nastavením byly naměřeny tyto hodnoty:

Pro `somaSteps = 1`, tedy jednu iteraci SOMA:

	Počet jedinců			
Průměrná pozice v ose	20	50	100	200
X	$-0,095$	$0,112$	$0,007$	$-0,009$
Y	$1,297$	$1,300$	$1,300$	$1,318$
Z	$0,157$	$-0,129$	$-0,033$	$-0,038$

Tabulka 14.1: Průměrná pozice gondoly po 50 iteracích algoritmu s 1 iterací SOMA

Z tabulky 14.1 je vidět, že průměrná odchylka od osy X s rostoucím množstvím použitých jedinců klesá a stroj tedy přesněji sleduje trasu.

Průměrná výška gondoly se vychyluje od optimální hodnoty minimálně.

Pohyb kupředu je realizován poklesem souřadnice Z a je vidět, že vyšší počet jedinců obecně zajistí kvalitnější řešení, ale nemusí tomu tak být vždy. Příliš vysoká hodnota může již znamenat i pád.

Pro `somaSteps = 2`:

	Počet jedinců			
Průměrná pozice v ose	20	50	100	200
X	$-0,001$	$-0,022$	$-0,031$	$-0,057$
Y	$1,304$	$1,313$	$1,305$	$1,320$
Z	$-0,175$	$-0,145$	$-0,031$	$0,154$

Tabulka 14.2: Průměrná pozice gondoly po 50 iteracích algoritmu se 2 iteracemi SOMA

Po dvou iteracích by SOMA měla vracet přesnější výsledky.

Je vidět (tabulka 14.2), že souřadnice již nemají takový rozptyl hodnot, jako když byla provedena pouze 1 iterace SOMA.

Provede-li SOMA 3 iterace (`somaSteps = 3`), pak je výsledkem tabulka 14.3.

	Počet jedinců			
Průměrná pozice v ose	20	50	100	200
X	$-0,029$	$-0,008$	$0,002$	$-0,013$
Y	$1,293$	$1,308$	$1,306$	$1,300$
Z	$-0,055$	$-0,051$	$-0,137$	$-0,117$

Tabulka 14.3: Průměrná pozice gondoly po 50 iteracích algoritmu se 3 iteracemi SOMA

Zejména na výsledcích pozice v ose X je vidět, že každá další iterace přináší vyšší zpřesnění výsledku. Otázkou však zůstává, zda je toto upřesnění dostatečně výhodné.

Každá iterace algoritmu totiž trvá konstantní dobu v závislosti na počtu použitých jedinců.

Nalezení rovnováhy mezi počtem jedinců a iterací je dalším úkolem.

Nutno upozornit, že větší část vhodného nastavení dalších parametrů lze daleko snadněji získat z pozorování chování robota během simulace a podle toho ovlivňovat parametry i jejich váhy při výpočtu celkové fitness.

Kapitola 15

Návrhy možných úprav

Algoritmus využívá databáze k tomu, aby nalezl vhodné následující řešení. V případě, že takové nenalezne, pak se provádí časově náročný výpočet.

Tento výpočet však není nutné počítat sekvenčně, jeho paralelizace je dobře možná a výrazně by mohla urychlit celý proces.

Dále by bylo dobré upravit hodnoty vracené SOMA tak, aby se navrhovaná řešení dostatečně lišila. Zatím je tento problém řešen pomocí vracení více jedinců. Alternativou by bylo vložit algoritmu jako vstup předem připravenou smyčku, ze které by mohl vycházet a zdokonalovat ji.

Dalšího vylepšení by se dalo dosáhnout, pokud by stav uložený v hashovacím stromě mohl upravovat své již jednou získané hodnoty tak, aby použitý potomek dosáhl vyšší univerzality.

Velkým přínosem algoritmu by jistě bylo sestavení rozhodovacích stromů, pomocí kterých by se mohly individuálně nastavit tolerance k jednotlivým proměnným popisujícím stav a některé by třeba bylo možné zcela vypusit. Výrazně by se tak mohl urychlit učící proces.

Komplikací však stále zůstává skutečnost, že práce se softcomputingovými algoritmy se neobejde bez jisté dávky stochastičnosti a tudíž není vůbec, nebo jen velmi těžko dokazatelná. Úspěšnost se posuzuje pouze empiricky, takže se nedá vyloučit ani možnost kompletního selhání navrženého algoritmu.

Kapitola 16

Závěr

V této práci byl navržen algoritmus, který využívá algoritmus SOMA pro hledání vhodného postupu, jak naučit chodit dvounohého robota. Konečné výsledky jsou uspokojivé, přestože se zatím nepodřilo dosáhnout plynulého pohybu kupředu. Navržený algoritmus se ukázal být možnou cestou pro další rozvoj v tomto směru – minimálně jako jeden z pomocných postupů. Nicméně se jedná o prototyp a tedy je třeba provést některé úpravy.

Vzhledem k velkému množství nastavitelných parametrů programu lze jen obtížně odhadnout jejich ideální nastavení a jejich hodnoty byly získány odhadem na základě empiricky získaných výsledků. Bylo by jistě vhodné další studii zaměřit na jejich optimalizaci a další úpravy algoritmu.

Literatura

[1] B. V. Babu Godfrey C. Onwubolu. *New optimization techniques in engineering.* Springer, Berlin, 2004. ISBN 3-540-20167-X.

[2] Michal Hlavinka. *Diferenční evoluce pro dynamické problémy – Bakalářská práce.* Vysoké učení technické v Brně - Fakulta informačních technologií, 2006.

[3] Jan Pokorný. *Variace evolučního SOMA algoritmu pro dynamické úlohy – Bakalářská práce.* Vysoké učení technické v Brně - Fakulta informačních technologií, 2007.

[4] WWW stránky. A bipedal walk using central pattern generator(cpg) [cit. 2010-01-03]. `http://www.brain.kyutech.ac.jp/~ishii/Paper/2004/2004_BrainIT_Ishii.pdf`.

[5] WWW stránky. Oscillator-controlled bipedal walk with pneumatic actuators [cit. 2010-05-22]. `www.oit.ac.jp/bme/~tsujita/works/movic06.pdf`.

[6] WWW stránky. Ivan Zelinka: Osobní stránky věnované SOMA [cit. 2010-05-23]. `http://www.ft.utb.cz/people/zelinka/soma/`.

[7] WWW stránky. Russel Smith: Open Dynamics Engine [cit. 2010-05-23]. `http://ode.org/`.

[8] et al. Yamaguchi J. *Development of a bipedal humanoid robot – control method of whole body cooperative dynamic biped walking.* Proceedings of the 1999 IEEE International Conference on Robotics and Automation. Detroit, Michigan, 1999. [Online cit 2010-05-23] `http://www.cats.rpiscrews.us/~wenj/ECSE641S07/biped_humanoid.pdf`.

[9] Ivan Zelinka. *Umělá inteligence v problémech globální optimalizace.* BEN – technická literatura, 2002. ISBN 80-73000-69-5.