Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

Dokumentácia k inžinierskemu dielu

Tím BAREKO

Bc. Michal Baňas

Bc. Šimon Harvan

Bc. Ernest Loureiro

Bc. Daniel Lukáč

Bc. Marko Moravčík

Bc. Lukáš Rešutík

Bc. Dávid Roba

Vedúci projektu: Ing. Ivan Kapustík

Predmet: Tímový projekt I

Ročník: 2017/2018

Obsah

Úvod

Bc. Daniel Lukáč

RoboCup je medzinárodná súťaž v robotickom futbale, na ktorom sa zúčastňujú univerzity z rôznych krajín sveta. Existuje viacero líg pre RoboCup a jednou z nich je aj liga pre simulovaný robotický futbal. Fakulta informatiky a informačných technológií Slovenskej technickej univerzity sa venuje vývoju simulovaného robotického futbalu už od roku 2000.

Tento projekt je vyvíjaný každým rokom v rámci predmetov Diplomový projekt, Bakalársky projekt a Tímový projekt. Naša fakulta organizuje taktiež turnaje v simulovanom robotickom futbale s názvom RoboCup at FIIT. Počiatky tohto projektu boli vyvíjané v 2D prostredí a od roku 2006 sa naša fakulta zamerala na vývoj v 3D.

Hráčov v simulovanom robotickom futbale tak nahradili modely robotov, ktoré boli vytvorené na základe reálnej podoby robotov z nesimulovaného robotického futbalu. Každý hráč je zložený z viacerých častí, ktoré sú prepojené ohybnými kĺbmi a disponuje rôznymi funkciami pre pohyb, kopnutie do lopty, zorientovanie sa v priestore, postavenie a ďalšie, ktorým sa budeme v našom projekte venovať.

Hlavným cieľom projektu je pokračovanie na vývoji hráča simulovaného robotického futbalu, za účelom zdokonalenia už existujúcich riešení a vytvorenia ďalších. Súčasťou projektu je aj testovací nástroj pre prácu s agentom, ktorý budeme v našom projekte využívať. Testovací nástroj bol vyvinutý pre podporu budúceho vývoja projektu a obsahuje radu funkcií pre prácu s agentom.

Táto dokumentácia bude obsahovať technickú časť k práci, ktorá bude na projekte vykonaná a bude poskytovať na projekt ucelený pohľad, pre pochopenie jeho častí. V dokumentácii zahrnieme naše vylepšenia a vykonanú prácu a budeme sa venovať zisteniam skutočností, ktoré zanalyzujeme.

2 Celkový pohľad na Robocup projekt

Bc. Lukáš Rešutík

Výsledkom niekoľko ročného projektového snaženia v rámci Robocup-u na FIIT STU je vývoj 3D hráča robotického futbalu pod názvom JIM. V súčasnosti je postavený na programovacom jazyku Java s využitím XML a projekt je rozdelený do troch častí:

- Jim
- RoboCupLibrary
- TestFramweork

Program aktuálne beží na serveri rcsserver3d-0.6.7. Server má informácie o aktuálnom stave hry, ponúka grafické zobrazenie pomocou rcsmonitor3d.

Dôležitou súčasťou projektu je Wiki stránka projektu. Na jej doplnení, aktualizovaní jej štruktúry sa zamerali staršie tímy, avšak náš tím považuje za nutné Wiki neustále aktualizovať a dopĺňať, ktoré budú nápomocné pre nasledujúce tímové, bakalárske a diplomové projekty. Táto kapitola opisuje celkový pohľad na aktuálny stav systému a všetkých jeho súčastí.

2.1 Jim

Bc. Lukáš Rešutík

Predstavuje Java aplikáciu samotného agenta, ktorý sa pripojí k serveru a hrá futbal. Agent by mal dodržiavať nakonfigurovanú a naprogramovanú taktiku. Pred spustením agenta je nutné spustiť RCSS server. Aplikácia agenta obsahuje jednoduché ovládanie prostredníctvom GUI umožňujúce opätovné načítanie pohybov a preplánovanie. Inicializačným bodom agenta je trieda *sk.fiit.jim.init.Main*.

Agent dokáže kopať do lopty, otočiť sa o 60 stupňov, postaviť sa za maximálne za 3s, stabilizovať sa pri vstávaní, behať, chodiť. Taktiež má agent naimplementované základné vnímacie taktiky (či tím útočí alebo bráni).

Spustenie agenta je pomerne zdĺhavý proces. Na počítači so slabším výkonom čas spustenia sa pohybuje cca. 10 sekúnd. Na začiatku spúšťania Jima sa vždy vykonajú nasledovné operácie:

- Načítavanie pohybov z XML súborov v adresári moves
- Načítanie anotácií
- Spracovanie parametrov príkazového riadku
- Spustenie TFTP servera použitého pre komunikáciu s TestFrameworkom
- Prípadné vytvorenie GUI
- Vstup do hlavného cyklu

2.1.1 Pohyby

Správanie agenta je riadené kódom, a možno ho rozdeliť do niekoľkých vrstiev. Prvou vrchnou vrstvou je plánovač riadený triedou *HighSkillPlanner* nachádzajúci sa v balíku *sk.fiit.jim.agent.highskill.runner*. Plánovač vytvára inštancie tzv. high skillov (vyšších pohybov). High skilly počas svojho behu vyberajú nižšie pohyby (nižšia vrstva) tzv. low skill, ktorý sa ma vykonať. Low skill predstavuje len skupinu metadát, zapísaných pomocou XML, pričom určujú jeho názov a ďalšie informácie.

SkillsFromXmlLoader he trieda, ktorá slúži na načítanie pohybov z XML súborov. Triedy *LowSkills* a *Phases* spravujú objekty fáz a pohybov a sú umiestnené v balíku *sk.fiit.jim.agent.moves*. Keďže načítanie pohybov je značne pomalé, ukladá sa ich parsovaná podoba do súboru *./movecache*, ktorý sa pri budúcom načítaní použije, pokiaľ od jeho vytvorenia nebol žiadny pohyb zmenený. V takom prípade sa súbor vytvorí nanovo.

2.2 RoboCupLibrary

Bc. Daniel Lukáč

Knižnica RoboCupLibrary obsahuje triedy, ktoré sú potrebné pre rôzne matematické výpočty, anotácie alebo reprezentáciu geometrických objektov. Táto knižnica sa vužíva len minimálne a v súčasnosti neexistuje dokumentácia alebo bližší popis pre získanie informácii o jej využití a to ani na Wiki. Avšak aj napriek jej minimálnemu využitiu je pre projekt potrebná.

Náš tím v súčastnosti obohatil túto knižnicu efektívnejšími výpočtami sin a cos funkcií, ktoré sú implementované v celom projekte. Triedy z knižnice sa využívajú ako v Test frameworku tak aj v agentovi. Cieľom je čo najviac odstrániť tieto závislosti medzi Jim-om a TestFrameworkom, čo však v súčasnosti nie je dodržané. V akademickom roku 2014/2015 prebehol kompletný refactoring tejto knižnice, tímom Infinity. Refactoring zahŕňal dodržanie konvencie kódu, zmazanie nepotrebných tried a funkcií a tak isto oddelenie testov od logiky knižnice.

2.3 TestFramework

Bc. Dávid Roba

Využíva sa na získanie spätnej väzby od hráča. Hlavným cieľom je vytvoriť robotického futbalového trénera, ktorý by schopný učiť hráčov novým taktikám a pohybom automaticky.

Na interakciu medzi agentom a serverom sa využíva posielanie správ, ktoré obsahujú tzv. S-výrazy, pričom hráč len odosiela správy a TestFramework ich len príjma. Server (TestFramework) je implementovaný

pomocou triedy AgentMonitor v balíku sk.fiit.testframework.monitor, ktorá slúži na spracovanie nových spojení od agentov. Každé z týchto spojení je reprezentované triedou AgentMonitorThread, ktorá sa nachádza tiež v balíku sk.fiit.testframework.monitor a slúži na spracovanie prichádzajúcich správ od agenta prostredníctvom triedy AgentMonitorMessage, ktorá je umiestnená v rovnakom balíku ako trieda AgentMonitorThread. V agentovi sa na komunikáciu s TestFrameworkom využíva trieda TestFrameworkCommunication, ktorá sa nachádza v balíku sk.fiit.jim.agent.communication.testframework.

2.4 Identifikované nedostatky

Bc. Dávid Roba

Na základe testovania projektu a tiež podľa dokumentácií z predchádzajúcich rokov boli identifikované nasledujúce nedostatky:

- Použité trigonometrické funkcie (*sin* a *cos*) patria medzi najpomalšie matematické operácie
- Pri vnímaní čiar sa do žiadnej miery nezohľadňuje náklon hráča.
- Komunikácia Jima s TestFrameworkom cez pomalý TFTP protokol
- Chôdza pomocou ZMP (Zero Moment Point) nie je implementovaná v žiadnej z taktík.
- Meranie disciplín turnaja v TestFramework-u neobsahuje všetky disciplíny turnaja.

3 Ciele

Bc. Daniel Lukáč

Na úvodnom stretnutí sme sa oboznámili s problematikou RoboCupu a stretli sme sa s tímom z predošlého akademického roka. Zástupcovia tímu nám odprezentovali časť och práce a oboznámili nás s niektorými nedostatkami. Na základe týchto skutočností sme stanovili ciele našej práce na projekte.

3.1 Ciele pre zimný semester

1. **Optimalizácia pomalých funkcií**

Projekt obsahuje mnoho funkcií, ktoré je možné optimalizovať. V prvotnej fáze sme sa zamerali na funkcie počítajúce sin a cos, ktoré sme úspešne implementovali. K optimalizovaným funkciám je potrebné vytvoriť správne testovanie a sledovať ich správanie.

2. **Zlepšenie orientácie Jima**

Cieľom je pokračovať na rozpracovaní vnímania čiar, ktoré je rozpracované minuloročným tímom a pokúsiť sa implementovať funkciu pre vnímanie len samotných čiar. V prvotnej fáze je pre túto implementáciu

potrebné venovať čas analýze už doposiaľ vytvoreným funkciám pre vnímanie čiar.

3. **Analýza a prípadná optimalizácia stabilizácie**

Je potrebné analyzovať zisťovanie náklonu z pevných bodov ihriska, získaných zo servera a zisťovanie náklonu z gyroskopu Jima. Na základe analýz je potrebné pokúsiť sa zlepšiť stabilizáciu Jima. Cieľ súvisí s cieľom 2. Zlepšenie orientácie Jima.

4. **Aktualizácia Wiki**

Súčasťou projektu je Wiki webová stránka, z ktorej čerpajú informácie študenti pri riešení bakalárskych a diplomových projektov, ako aj pre tímy pokračujúce v rámci tímového projektu. Preto je potrebné túto stránku udržiavať aktualizovanú.

4 Používané technológie

Bc. Lukáš Rešutík

4.1 Server

V rámci projektu prebieha komunikácia medzi agentom a serverom oboma smermi. Agent posiela informácie serveru o zmene nastavenia jednotlivých kĺbov a server posiela agentovi informácie o aktuálnom stave hry v S-správach.

Predchádzajúce tímy experimentovali s rôznymi verziami rcsserver3d na rôznych operačných systémoch (napr. Windows, Unix). Táto podkapitola sa venuje rôznym verziám, ich rozdielom a možnosti použitia na projekte.

Niekoľko hodín práce sme sa pokúšali spojazdniť novšie verzie servera (0.6.10, 0.6.9) avšak stále sa ich nepodarilo úspešne spustiť. Preto používame pôvodnú verziu 0.6.7. Nasledujúcim tímom preto doporučujeme používať starší server.

4.1.1 rcsserver3d-0.6.8

- Podporuje rozoznávanie tímov (identifikácia správ, od ktorého tímu prišla)
- Pridanie tabule so skóre
- Maximálne 11 druhov robotov

4.1.2 rcsserver3d-0.6.9

- Pridanie nového pravidla – lopta sa musí dotýkať protihráča alebo spoluhráča ak sa lopta nenachádza v stredovom kruhu
- 11 metrové kopy (penalty) sú priame

4.1.3 rcsserver3d-0.6.10

- Nové modeli pre vizuálne odlíšenie iných typov robotov

4.2 Pluginy

Bc. Lukáš Rešutík

V rámci vývojového procesu je veľmi výhodné použiť vybraté pluginy do nástroja Eclipse, ktoré zlepšujú kvalitu kódu a zároveň uľahčujú prácu v tíme BAREKO počas práce na projekte. Nižšie uvádzame len použiteľné pluginy. Tieto pluginy taktiež využívali, niektoré staršie tímy a doporučujú ich používať.

- FindBugs
- CheckStyle
- Eclipse PMD

4.2.1 FindBugs

Findbugs je plugin, ktorý detekuje chyby v jazyku Java, pričom využíva statickú analýzu založenú približne na 200 vzoroch. Jeho výstupom je poukázanie na zlý kód (nekonečné rekurzívy, nesprávne použtie Java knižníc, uviaznutia v kóde). Doporučuje sa použiť na rozsiahle projekty, kde je predpoklad zhruba 1 poruchy na 1000 – 2000 riadkov kódu.

4.2.2 Checkstyle

CheckStyle je plugin, ktorého úlohou je dohliadať na dodržiavanie štandardov v kóde. Výsledkom tohto pluginu je zlepšená čitateľnosť kódu. V tímovom prostredí, kde každý člen má svoj štýl písania kódu (napr. na základe programovacieho jazyka), je nutné mať jeden štandard písania kódu, ktorý je zhrnutý v metodike code conventions.

4.2.3 Eclipse PMD

Eclipse PMD je plugin, ktorý integruje analyzátor zdrojového kódu do prostredia Eclipse. Účelom PMD je vyhľadanie zlých vzorov kóde. Tento plugin, po každom uložení zdrojového kódu, preskenuje kód a nájde

potencionálne problémy (bugy, neoptimálne, duplicitné, kognitívne náročné časti kódu). Tento plugin ponuká možnosť automatickej opravy, ak je to možné.

5 UT Austin

Bc. Ernest Loureiro

Tím UT Austin Villa je tímom Texaskej Univerzity v Austine (US), ktorý pôsobí v tejto oblasti od roku 2004. Patria medzi svetovú špičku v simulovanom robotickom futbale a od roku 2011 sú majstrami sveta, s výnimkou roku 2013, kedy skončili na druhom mieste. Svoj úspech pripisujú tzv. „vše-smerovej" chôdzi, pri ktorej robot iba prekračuje na mieste a smer chôdze je udávaný pomocou náklonu, tj. preneseniu váhy. Vďaka takejto implementácii sa môže robot hýbať všetkými smermi, čo z neho robí veľmi rýchleho protivníka. Túto implementáciu uverejnili na portály GitHub v januári 2016. Na rozdiel od robota na našej fakulte, robot tímu Austin Villa je naprogramovaný pomocou C++, kvôli čomu je komplikované niektoré funkcie prevziať a aplikovať ich na nášho robota. Zároveň je tento kód vyvíjaný pod OS Ubuntu, ktorým náš tím nedisponuje.

6 Kalmanov Filter

Bc. Dávid Roba, Bc. Marko Moravčík

6.1 Analýza starších tímov FIIT ohľadom kalmana

Ako prvý, tím v roku 2009 implementoval kalmanov filter do projektu 3D futbal. Od vtedy ho používali všetky ostatné tímy pričom nedošlo k žiadnym zmenám v jeho implementácií.

Okrem využitia Apache commons Kalman filter knižnice, ktorá vykonáva výpočty Kalmana, tak došlo k implementácií nasledovných tried:

- KalmanForVector – Kalmanov filter pre vektor
- KalmanForVariable – popisuje triedu určujúcu výpočet linearno-kvadratického Gaussovho regulátora pre premennú
- KalmanTest – trieda, slúžiaca na testovanie Kalmanovho filtra
- KalmanAdjuster – trieda slúžiaca na výpočet pozície objektov na ihrisku(lopta, zástavky) použitím kalmanovho filtra, čím sa redukuje šum a tak sa znižuje chybovosť vnášaná serverom -> žiadna zmena od roku 2009

- KalmanAdjusterTest - tri testy pre Kalmanove filtre. Ich úlohou je preveriť funkčnosť metód zodpovedných za redukciu šumu pri výpočte pozície lopty a fixného bodu (ľavá rohová zastávka domáceho tímu)

6.2.2 Analýza implementácie kalmana

6.2.1 Definícia

Algoritmus, ktorý používa sériu meraní pozorovaných v priebehu času, obsahujúcich štatistický šum a iné nepresnosti na odhad premenných v širokom spektre procesov. Využíva k tomu nielen naposledy nameraných dát a model systému, ale aj vektor údajov o predchádzajúcom stave systému. Je široko používaný pre spracovanie signálov, navigáciu, robotiku a ďalšie úlohy.

6.2.2 Výhody

- Dobré výsledky v praxi z dôvodu optimality a štruktúry
- Pohodlná forma pre online spracovanie v reálnom čase
- Jednoduché formulovanie a implementácia s ohľadom na základné pochopenie
- Meracie rovnice nemusia byť obrátené

6.2.3 Kroky

Lineáre systémy

V uvedených rovniciach *A*, *B* a *H* sú matice, *k* je časový index, *x* je tzv. stav systému, *u* je známy vstup do systému a *w* a *z* sú šumy. Premenná *w* reprezentuje tzv. procesný šum a *z* tzv. merací šum. Každá z týchto veličín je (obvykle) vektor a preto obsahuje viac ako jednu zložku. Vektor *x* obsahuje všetky dostupné informácie o súčasnom stave systému, ale nie je možné ho merať priamo. Namiesto toho meriame *y* , ktoré je funkciou x (teda *y=f*(*x*)), ale je "zašumená" šumom *z* . Je možné použiť *y* ako pomoc na získanie odhadu x, ale nemôžeme nutne vziať informáciu z *y* ako reprezentujúcu hodnotu, pretože obsahuje aj šum.

- stavová rovnica prechodu $x_{k+1} = Ax_k + Bu_k + w_k$
- rovnica výstupu $y_k = Hx_k + z_k$

Keďže je pohyb hráča priamočiary, môžeme povedať, že stav pozostáva z jeho polohy p a jeho rýchlosti v. Vstupná veličina u je nariadené zrýchlenie a výstupná veličina y je meraná poloha. Povedzme, že sme schopní meniť akceleráciu a merať polohu každých Ts sekúnd. V takomto prípade bude rýchlosť v definovaná rovnicou:

$$v_{k+1} = v_k + T_s\, u_k$$

To znamená, že rýchlosť vzdialená jednu vzorku od teraz (teda *Ts* sekúnd) bude rovná súčasnej rýchlosti

prenásobenej nariadeným zrýchlením u a intervalom Ts. Predošlá rovnica však nedáva presnú hodnotu pre T_{k+1}. Namiesto toho bude rýchlosť ovplyvnená šumom. Šum rýchlosti je náhodnou premennou, ktorá sa mení s časom. Preto realistickejšie zapísaná rovnica vyzerá nasledovne:

$$v_{k+1} = v_k + T_s\, u_k + \sim v_k$$

kde $\sim v_k$ je šum rýchlosti. Podobnú rovnicu je možne odvodiť aj pre polohu p:

$$p_{k+1} = p_k + T_s\, v_k + \frac{1}{2} T_s^2 u_k + \sim p_k$$

kde $\sim p_k$ je šum rýchlosti . Na základe čoho môžeme definovať stavový vektor, ktorý pozostáva z polohy a rýchlosti:

$$x_k = \begin{pmatrix} pk \\ vk \end{pmatrix}$$

Keďže je meraná výstupná veličina úmerná polohe, je možné do rovníc za premenné A,B a H dosadiť nasledovné matice:

$$X_{k+1} = \begin{pmatrix} 1 & Ts \\ 0 & 1 \end{pmatrix} x_k + \begin{pmatrix} \frac{Ts^2}{2} \\ Ts \end{pmatrix} u_k + w_k$$

$$Y_k = (1 \;\; 0)\, x_k + z_k$$

Kde matica A sa nazýva predikčná matica a matica B sa nazýva riadiaca matica. Keďže predpoveď nemusí byť sto percentne presná, vstupuje v tomto bode do procesu Kalmanov filter. Riešenie Kalmanovým filtrom nie je možné aplikovať kým nie sú splnené určité predpoklady o šume. Je potrebné si uvedomiť, že v rovniciach (modelu systému) v úvode vystupujú dve premenné - w je šum v procese a z je šum v meraní. Taktiež je ďalej možné skonštatovať, že neexistuje korelácia medzi w a z - teda v akomkoľvek čase k, w_k a z_k sú nezávislé náhodné premenné. Potom je možné zadefinovať kovariančné matice šumu Q a R ako:

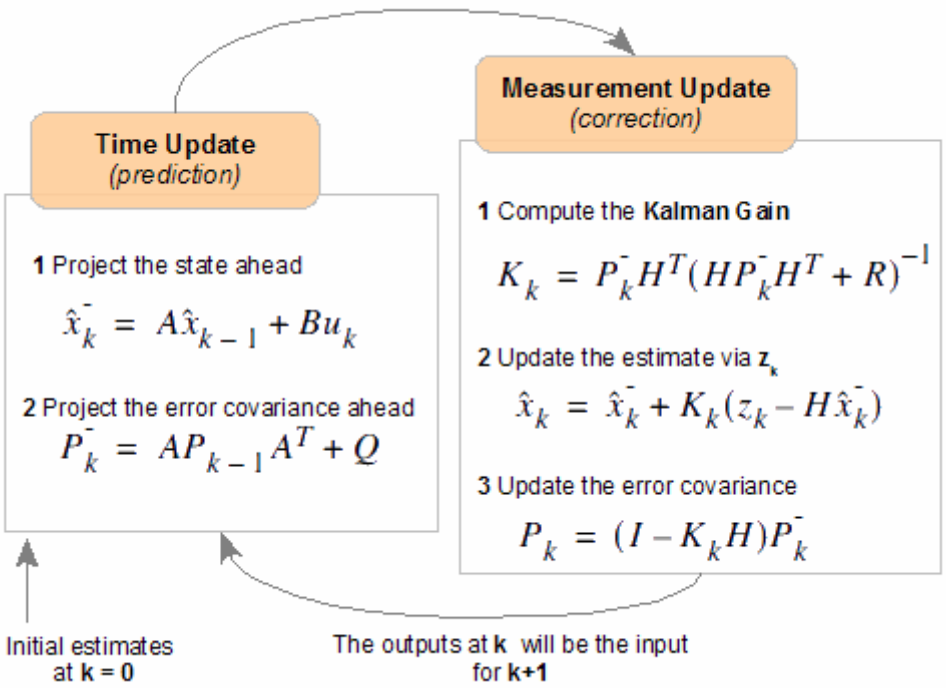
- Kovariancia šumu v procese: $Q = (w_k w_k^T)$
- Kovariancia šumu v meraní: $R = (z_k z_k^T)$

kde w^T a z^T označujú transpozíciu náhodných vektorov šumov. Teraz je možné použiť Kalamnové rovnice.

6.2.4 Kalmanove rovnice

Kalmanov filter sa delí na dve fázy. Predikčnú (časovú), v ktorej sa určuje nový najlepší odhad $\hat{x}_k^-$ ako predikcia z predchádzajúceho najlepšieho odhadu plus korekcia známych vonkajších vplyvov. A chyba kovariancie P_k^- je predpovedaná na základe predchádzajúcej chyby kovariancie spolu s určitou dodatočnou chybou z prostredia.

Tieto hodnoty sú následne použité v druhej fáze Kalmanovho filtra, ktorá sa nazýva Korekčná (filtračná), ktorá pozostáva z rovníc na nájdenie $\hat{x}_k$, ktorý predstavuje odhad *x* v čase *k* (teda, to čo chceme nájsť). Druhý výraz v rovnici je tzv. Korekcia P_k a reprezentuje veľkosť korekcie, akou treba korigovať odhad stavu na základe meraní. Rovnica K_k sa nazýva rovnica Kalmanovho zisku (zosilnenia). Rozborom rovnice pre K zistíme, že ak je šum z merania veľký, *R* bude veľké, takže *K* bude malé a nebudeme dávať veľkú váhu meraniu *y* , keď budeme počítať ďalšie. Na druhej strane však ak je šum v meraní malý, *R* bude malé a *K* veľké, čo znamená, že meraniu priradíme veľkú váhu pri výpočte ďalšieho.



Obr.1: Priebeh Kalmanovho filtra

7 Wiki

Bc. Ernest Loureiro

Hlavnou prácou tímu A55 Kickers (akad.rok 2012/2013) bola wikipédia k projektu RoboCup na našej fakulte. Táto wikipédia zoskupuje všetky poznatky o tomto projekte naprieč ľuďmi a tímami, ktoré na tomto projekte pracovali. Je rozčlenená do siedmich častí nasledovne:

1. Analýza tímov – obsahuje analýzy všetkých predchádzajúcich tímov, ktoré na projekte pracovali, ako aj zahraničných tímov súťažiacich v RoboCupe

2. Jim – obsahuje informácie o agentovi, ako s ním pracovať, ako implementovať nové techniky a skilly

3. TestFramework – popisuje fungovanie a implementáciu aplikácie TestFramewok

4. Wiki – Obsahuje informácie pre prácu so samotnou wikipédiou

5. Záverečné práce – záverečné práce diplomantov a bakalárov

6. Ruby old – staré dokumenty z čias keď niektoré funkcie boli implementované pomocou jazyka ruby

7.Nezaradené – obsahuje všetky ostatné dokumenty

Pomocou tejto wikipédie je možné prehlbovať svoje znalosti o projekte na základe zdedených dokumentov a následne na týchto znalostiach stavať pokrok v tomto projekte.

Náš tím prispel k fungovaniu tejto wikipédie aktualizáciou návodov, konkrétne návodu na „rozbehanie“ prostredia pod OS X, a následným formátovaním celej stránky s návodmi a inštaláciami, nakoľko bola táto stránka spísaná neprehľadne a častokrát nezrozumiteľne. Do budúcna by sme chceli opraviť nefunkčné odkazy a tejto stránke, ktoré nám pri inštalácii spôsobovali problémy, a aktualizovať návody na jednotlivé operačné systémy.

8 Implementácia vlastnej funkcie sin/cos

Bc. Marko Moravčík

Po analýze časti kódu, kde dochádza k výpočtom sin a cos, prostredníctvom knižnice Math (Math.sin(RADIAN_ANGLE) , Math.cos(RADIAN_ANGLE)), sme sa zhodli, že dokážeme zvýšiť výkonnosť systému implementovaním vlastných funkcií sin a cos, ktoré fungujú nasledovne:

Funkcia *calculateCosAndSin* slúži na výpočet sin a cos , a následne uloženie výsledných hodnôt do tabuľky.

```
public void calculateCosAndSin() {

    double actAngle = 0;

    while(actAngle < 90) {
        sinArray[(int)(actAngle*100+0.05d)] = Math.sin(Math.toRadians(actAngle));
        cosArray[(int)(actAngle*100+0.05d)] = Math.cos(Math.toRadians(actAngle));

        actAngle += 0.01;
        actAngle = (double) Math.round(actAngle * 100) / 100;
    }
}
```

Obr.2: Funkcia *calculateCosAndSin*

Funkcia *getCos*, vyberie z vyrátanej tabuľky hodnotu cos podľa zadaného uhla.

```
public double getCos(double angle) {

    angle = Math.toDegrees(angle);
    if (angle >= 0 && angle < 90) {
        return cosArray[(int)(angle*100+0.05d)];
    }
    else if (angle >= 90 && angle < 180) {
        return -cosArray[(int)((180-angle)*100+0.05d)];
    }
    else if (angle >= 180 && angle < 270) {
        return -cosArray[(int)((angle-180)*100+0.05d)];
    }
    else if (angle >= 270 && angle < 360) {
        return cosArray[(int)((360-angle)*100+0.05d)];
    }
    else return 0;
}
```

Obr.3: Funkcia *getCos*

Funkcia *getSin*, vyberie z vyrátanej tabuľky hodnotu sin podľa zadaného uhla.

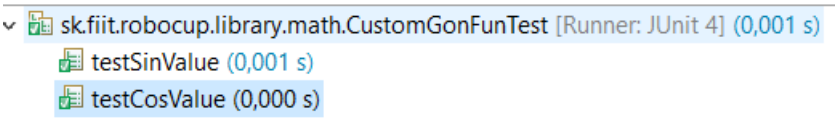
```
public double getSin(double angle) {

    angle = Math.toDegrees(angle);
    if (angle >= 0 && angle < 90) {
        return sinArray[(int)(angle*100+0.05d)];
    }
    else if (angle >= 90 && angle < 180) {
        return sinArray[(int)((180-angle)*100+0.05d)];
    }
    else if (angle >= 180 && angle < 270) {
        return -sinArray[(int)((angle-180)*100+0.05d)];
    }
    else if (angle >= 270 && angle < 360) {
        return -sinArray[(int)((360-angle)*100+0.05d)];
    }
    else return 0;
}
```

Obr.4: Funkcia *getSin*

Funckie sú implementované v triede *CustomGonFun* v balíku *sk.fiit.robocup.library.math* v projekte *RoboCupLibrary*.

Naša implementácia bola následne otestovaná s využitím JUnit testov. Ako prvá sa otestovala správnosť výsledkov, ktoré vypočítali naše funkcie.

Výsledok testu:



Obr.5: Ukážka testov

Testom sme dokázali, že výsledky sú správne. Testovala sa aj rýchlosť vykonania funkcii v porovnaní s pôvodnými funkciami z knižnice Math. Pri 10 000 000 opakovaniach. Výsledky s využitím našej implementovanej funkcie boli niekoľkokrát rýchlejšie ako pri pôvodnej (približne o 2 sekundy).

9 Analýza vnímania prostredia

Táto kapitola sa zaoberá analýzou vnímania prostredia hráčom. Spôsobom ako reprezentuje priestor, akým spôsobom vníma čiary. Aký vplyv na vnímanie prostredia majú náklony hlavy, chôdza.

9.1 Použitie Kalmanovho filtra na pozíciu hráča

Bc. Dávid Roba

Pozícia hráča sa určuje v triede *AgentModel.java* volaním funkcie *updatePosition(data)* z triedy *AgentPositionCalculator.java*, ktorej parametrom sú prijaté dáta zo servera. Pozícia sa určuje 3 rôznymi spôsobmi na základe toho, koľko fixných bodov a čiar vidí hráč v danej chvíli.

9.1.1 oneLineSeen(ParsedData)

V prípade, že hráč nevidí žiadny fixný bod a vidí iba jednu čiaru, určí sa jeho pozícia podľa nasledujúcich krokov.

1. Z posledných 10-tich pozícií hráča sa vyhodnotí, v ktorej časti ihriska sa hráč nachádza. Pri vyhodnotení sa ignoruje stredový kruh (t.j. uvažujú sa iba časti 1-4). Vyhodnotenie časti prebieha hlasovaním – každý záznam má vplyv na určenie podľa typu, akým bola daná poloha vyhodnotená. Celkové skóre pre danú časť ihriska je súčet typov vyhodnotení pozície hráča pre danú časť ihriska.
2. Identifikujú sa koncové body čiary – bod, ktorý je naľavo od hráča má väčšie *Phi* a bod, ktorý je napravo má naopak menšie *Phi*.
3. Oba koncové body čiary sa otočia v závislosti od aktuálneho natočenia hráča.
    a) Ak je rozdiel medzi y-ovými súradnicami bodov po natočení väčší ako rozdiel medzi x-ovými – hráč vidí horizontálnu čiaru (pohľad zhora na ihrisko, pričom bránky sú na ľavej a pravej strane)
    b) V opačnom prípade vidí hráč vertikálnu čiaru
4. Na základe predchádzajúcich pravidiel, súradníc bodov a časti ihriska, kde sa pravdepodobne nachádza, sa vytvorí čiara, kde by sa mohol hráč nachádzať.
5. Od poslednej známej pozície (posledný záznam v histórii) sa vytvorí kružnica s polomerom rovným *accDistance* z poslednej položky v histórii.
6. Výpočet priesečníka kružnice s úsečkou.
    a) Ak je jeden priesečník – nastaví sa daný bod ako miesto, kde sa hráč nachádza

b) Inak sa vytvorí nová čiara z posledných dvoch záznamov v histórii. Vypočíta sa priesečník tejto čiary a čiary, kde by sa hráč mohol nachádzať.
   i. Ak neexistoval priesečník s kružnicou – priesečník dvoch čiar sa nastaví ako poloha, kde by sa mal hráč nachádzať
   ii. Inak sa nastaví priesečník kružnice, ktorý je bližšie k priesečníku dvoch čiar

9.1.2 oneFixedObjectSeen(ParsedData, int)

V prípade, že hráč vidí jeden kontrolný bod a čiary, funkcia vráti 2 namapované body z čiar, podľa nasledujúcich krokov.

1. Ak hráč vidí vlajku, pravdepodobne vidí aj ďalšie (minimálne) dve čiary. Najskôr sa zistí, či čiara, ktorú hráč vidí, má spoločný bod s vlajkou (vzor čiar „čiara a bod“). Informácia o tom, že daná čiara má spoločný bod, nepostačuje na určenie, o ktorú čiaru sa jedná (môžu byť dve). Ak agent vidí *F1R* – „horizontálnu čiaru“, vidí druhý bod čiary pod väčším uhlom theta (*theta1*) ako „vertikálnu čiaru“ (*theta2*). Uhly *theta1* a *theta2* sú theta uhly ku koncovým bodom čiary.
   a) Ak hráč vidí *F1R* a *theta1* > *theta2*, potom bod, ktorý hráč vidí pod uhlom *theta1*, je bodom z „horizontálnej čiary“ a *theta2* je bodom z vertikálnej čiary.
   b) Ak hráč vidí *F2R* a *theta1* > *theta2*, potom bod, ktorý hráč vidí pod uhlom *theta1*, je bodom z „vertikálnej čiary“ a *theta2* je bodom z „horizontálnej čiary“.
   c) Ak hráč vidí *F1L* a *theta1* > *theta2*, potom bod, ktorý hráč vidí pod uhlom *theta1*, je bodom z „vertikálnej čiary“ a *theta2* je bodom z „horizontálnej čiary“.
   d) Ak hráč vidí *F2L* a *theta1* > *theta2*, potom bod, ktorý hráč vidí pod uhlom *theta1*, je bodom z „horizontálnej čiary“ a *theta2* je bodom z „vertikálnej čiary“.

   Následne sa vypočíta dĺžka videnej časti čiary a odpočíta/pripočíta sa dĺžka čiary k súradnici fixného bodu v závislosti od čiary a vlajky (fixný bod).
2. Ak hráč vidí bránku, najskôr sa zistí, či čiara, ktorú hráč vidí, je koncová „vertikálna čiara“ ihriska. Následne sa vypočíta vzdialenosť od priemetu bodu ku koncom čiary, ktoré vidí agent. Ďalej nasleduje výpočet absolútnych súradníc bodov – pripočítanie/odpočítanie vzdialenosti od y-ovej súradnice bránkového bodu v závislosti od uhla, pod akým hráč vidí daný bod.

9.1.3 justLinesSeen(ParsedData, int)

V prípade, že hráč vidí niekoľko čiar, vráti funkcia n–namapovaných bodov z čiar. Táto metóda na určenie pozície hráča ešte doposiaľ nebola implementovaná.

9.2 Kalmanov filter

Výsledkom tejto analýzy je zistenie, že Kalmanov filter sa v projekte nepoužíva pri určovaní pozície hráča. V projekte je implementovaný ako observer *KalmanAdjuster*, ktorý je naviazaný na prijatú správu zo servera. Implementované je to spôsobom, že sa automaticky optimalizuje/odhaduje

poloha lopty a pevné rohy ihriska – vlajky.

9.2 Analýza vnímania čiar pri náklone hlavy

Bc. Daniel Lukáč

9.2.1 Reprezentácia čiar

Čiary sú v projekte reprezentované triedou *public class Line.* Každá čiara je v triede definovaná ako absolútna hodnota prvého bodu, typu Vector3D a druhého bodu typu Vector3D. K týmto premenným sa pristupuje pomocou metód get() a set(), ktoré sú súčasťou triedy *public class Line.*

```
/** absolute position of first point of line **/
private Vector3D position1 = Vector3D.ZERO_VECTOR;

/** absolute position of second point of line **/
private Vector3D position2 = Vector3D.ZERO_VECTOR;
```

Obr.6: Implementácia čiar

9.2.2 Hlava

Otáčanie hlavy funguje na princípe xml súborov, ktoré sa nachádzajú v priečinku move v Jimovi. Xml súbory obsahujú radu parametrov, pre otáčanie jednotlivých kĺbov Jima. Pohyby hlavy sú reprezentované súbormi:

- head_left_120.xml
- head_right_120.xml
- head_left_down.xml
- head_right_down.xml

Súbory head_left_120.xml a head_right_120.xml zabezpečujú otáčanie hlavy do strán pod maximálnym uhlom 120° a súbory head_left_down.xml a head_right_down.xml zabezpečujú náklony hlavy nadol a nahor.

Analýzou sme zistili, že náklony hlavy nahor a nadol boli z implementácie vylúčené tímom Infinity v akademickom roku 2014/2015, v dôsledku nulovej pridanej efektivity pre vnímanie prostredia a lopty. Tým sme nemohli analyzovať vnímanie čiar pre úklony. Jim tak vníma čiary len vo vodorovnej polohe, čím nezohľadňuje žiaden náklon hlavy, resp. vidí len vodorovne. Ďalej sme analýzou prišli na to, že Jimova hlava pri chôdzi z časti mení svoju rotáciu vzhľadom na celé telo, no táto rotácia v priestore sa pri vnímaní čiar nijako nezohľadňuje.

```
} /*else if (!leftDownLook) {
    leftDownLook = true;
    return LowSkills.get("head_left_down");
}
else if (!rightDownLook) {
    rightDownLook = true;
    return LowSkills.get("head_right_down");
} */
else if (!leftLook) {
    leftLook = true;
    return LowSkills.get("head_left_120");
}
else if (!rightLook) {
    rightLook = true;
    return LowSkills.get("head_right_120");
}
```

Obr.7: Skilly pre pohyb hlavy

```
    } /*else {
        if (moveForward(relativizedBall)) {
            logger.log(LogType.HIGH_SKILL, "Move forward");
            return LowSkills.get(walkSkill);
            /*if(playerDistance(relativizedBall)){
                planner.addHighskillAsFirst(new HeadDownSkill());
            } else{
                return LowSkills.get(walkSkill);
            }*/

       /* } else if (moveBack(relativizedBall)) {
            logger.log(LogType.HIGH_SKILL, "Move back");
            return LowSkills.get("walk_back");
        } else {
            return kick();
        }
    }*/
}else if (!isInVerticalRange(relativizedBall)) {
    logger.log(LogType.HIGH_SKILL, "Move vertical");

    if (moveForward(relativizedBall)) {
        /*this can be used for executing some action in the moment when player come to ball
         * if(playerDistance(relativizedBall)){
            System.out.println("we are before ball");
            logger.log(LogType.HIGH_SKILL, "Move forward");
             planner.addHighskillAsFirst(new HeadDownSkill());
        }*/
```

Obr.8: Zakomentované pohyby hlavy