Факултет инжењерских наука Универзитета у Крагујевцу

Назив студијског програма: Машинско инжењерство

Ниво студија: Мастер академске студије

Модул: Примењена механика и аутоматско управљање

Предмет: Пројектовање система аутоматског управљања

Број индекса: 373/2015

Тијана Д. Цвијовић

Програмирање кретања LEGO Segway мобилног робота у програмском језику Python

мастер рад

Комисија за преглед и одбрану:

1. **Др Милан Матијевић, ред. проф.**
2. _____
3. _____

Датум одбране:_____

Оцена:_____

У оквиру овог мастер рада кандидат треба да опише методологију програмирања и праћења кретања LEGO Segway мобилног робота (самобалансирајућег LEGO робота на два точка) у програмском језику *Python*. Коришћењем објектно орјентисаног програмског језика *Python* треба реализовати генерисање и праћење задате трајекторије робота, уз истовремено одржавање стања лабилне равнотеже самобалансирајућег робота и синхроно кретање точкова мобилног робота. Реализује се управљање и надзор. Потребно је урадити два илустративна примера: кретање по кружници, и кретање по протоколу: одржавање равнотеже робота у тачки А, праволинијско кретање до тачке Б, одржавање равнотеже робота у тачки Б. Демонстрира се кретање LEGO Segway NXT робота: самобалансирање, кретање по трајекторији (кружној и праволинијској, са задавањем и праћењем профила пута, брзине и убрзања). Рачунар снима и приказује остварене дијаграме пута, брзине и убрзања, и када се робот креће по праволинијској путањи, и када се креће по кружници.

Препоручена литература:

[1] Жељко Крстић 338/2013, Имплементација контролера LEGO Segway мобилног робота у програмском језику Python, мастер рад, Факултет инжењерских наука Универзитета у Крагујевцу, 2017

[2] Тијана Цвијовић, и група студената, Трка LEGO Segway мобилних робота - Имплементација контролера LEGO Segway мобилног робота у програмском језику Python, пројектни задатак из ПСАУ, Факултет инжењерских наука Универзитета у Крагујевцу, 2017

[3] Yamamoto, NXTway-GS Model Based Desing – Control of self balancing two – wheeled robot built with LEGO Mindstorms NXT, http://www.mathworks.com/matlabcentral/fileexchange/13399-embedded-coder-robot-nxt-demo

Крагујевац, јул, 2017.

ментор:

др Милан Матијевић , ред. проф.

_____

САДРЖАЈ

РЕЗИМЕ

У овом раду је приказано управљање LEGO Segway NXT мобилног робота помоћу програмског језика Python. У раду је приказано оригинално Yamamotovo решење у Matlabu, на основу кога се и базира Python код. Приказане су карактеристике робота које се односе на програмабилну коцку истог, моторе, сензоре и остале LEGO делове. Такође је представљен и математички модел робота и изведене су потребне једначине. Коришћењем повратне спреге по стању израчуната су појачања серво контролера задуженог за балансирање и кретање робота. Оригинално решење је приказано помоћу Simulink блокова, а затим приказано решење у Python програмском језику. На крају су дати потребни дијаграми.

Кључне речи**:** LEGO NXT robot, Python, Yamamoto, Simulink

ABSTRACT

LEGO Segway NXT mobile robot control program made using Python programming language is presented in this paper. This project is done using original Yamamoto's solution in Matlab, which is also the base for this Python code. Robot's characteristics such as programmable brick, motors, sensors and other LEGO parts are shown in this paper. Also, robot's mathematical model is presented and required equations are performed. Using state feedback servo controller gains needed for LEGO NXT self balancing and moving are calculated. The original solution is shown using Simulink blocks, and also solution in Python programming language is presented. Required diagrams are given in the end.

Key words: LEGO NXT robot, Python, Yamamoto, Simulink

1. УВОД

Велика заступљеност аутоматизовних процеса у различитим гранама индустрије условљава примену програмабилних индустријских робота који успешно замењују људску снагу у различитим пословима. Роботи различитих димензија остварују боље резултате од човека у пословима где је потребна висока прецизност и поновљивост, што омогућава њихову све већу заступљеност у индустрији. Због својих карактеристика, индустријски роботи доводе до повећања продуктивности у производњи. У почетку је развој робота био усмерен на потребе индустрије док су данас заступљени хуманоидни роботи који служе за негу старијих, хируршке захвате и сл.

Поред поменутих, развијени су и роботи попут LEGO робота који своју примену налазе у едукацији. LEGO делови који подсећају на познате LEGO коцкице служе за повезивање микро рачунара робота са сензорима, актуаторима и осталим деловима у циљу добијања програмабилног LEGO робота (слика 1.1). Различитим комбинацијама LEGO роботике могуће је добити жељени изглед робота у зависности од задатка за који је исти намењен.

Слика 1.1. *Пример комбинације LEGO роботике – LEGO NXT програмабили робот*

Тема овог рада се односи на програмирање кретања програмабилног LEGO NXT робота коришћењем програмског језика Python, па је у том смислу, пре све потребно пројектовати контролер који омогућава балансирање и кретање робота. Пројектовање контролера у програмском језику Python је омогућено коришћењем оригиналног решења приказаног у Simulinku. Програмски језик Python налази све већу примену код програмирања LEGO робота, а у раду је представљен значај истог.

2. ПРИМЕНА LEGO MINDSTORMS РОБОТА

Напредак науке условљава развој технике и технологије што се пре свега огледа у индустријским аутоматизованим процесима у којима роботи имају водећу улогу приликом реализовања истих.

LEGO роботи се значајно разликују од индустријских робота**,** међутим ови програмабилни роботи имају велику улогу у едукацији, а уз одговарајуће модификације налазе примену у различитим гранама науке. Најважнија компонента LEGO сета је микрорачунар тј. LEGO коцка која омогућава извршавање дефинисаног задатка. LEGO коцку је могуће директно програмирати а могуће је и написати програм којим се задају инструкције роботу и исти пребацити на коцку помоћу USB или Bluetooth конекције. У наредном поглављу је представљен историјски развој LEGO робота, и приказане предности новијих генерација робота у односу на претходне, што се пре свега огледа у њиховој примени.

Напоменуто је да се LEGO роботи могу користити у едукацији, где се кроз програмирање истих превазилазе традиционалне методе учења, а студенти се кроз практичну наставу оспособљвају за решавање реалних проблема везаних за кинематику робота и компоненте попут сензора и актуатора. Коришћењем различитих сензора и стандардних LEGO коцкица као саставних елемената овог сета, LEGO роботи се могу прилагодити потребама најмлађих, па се деца кроз игру упознавању са основним појмовима у роботици.

Поред примене у образовању, LEGO роботи налазе примену у различитим гранама инжењерства и индустрије, а честа је њихова примена и у медицини, где успешно изводе сложене задатке где се захтева велика прецизност. Познато је да се користе у истраживањима на Универзитету у Кембрицу, где налазе своју улогу у задацима везаним за добијање вештачких костију (слика 2.1).

Слика 2.1. *Примена LEGO робота у медицинским истраживањима*

3. ИСТОРИЈСКИ РАЗВОЈ LEGO MINDSTORMS РОБОТА

Почеци LEGO програмабилних робота се везују за осамдесете године прошлог века. Професори са MIT универзитета у Америци су пре свега желели да омогуће деци употребу њиховог програмског језика LOGO у циљу контроле најједноставнијих робота. LEGO је тада већ креирао своју колекцију која укључује конструкторске сетове са моторима, а челници компаније су желели да LEGO бренд прошире сетовима који имају едукативан значај. Заједнички циљ је успоставио сарадњу између LEGO компаније и LOGO програмског језика истраживача са MIT-а.

Први у низу пројеката био је LEGO LOGO, који је омогућавао деци да дизајнирају сопствене машине помоћу LEGO коцкица различитих димензија а користећи LOGO програмски језик за управљање. Корисницима је било омогућено да кроз најједноставније експерименте науче о механичком дизајну и софтверу LEGO сета. Даља сарадња довела је до имплементирања напреднијих сетова који су поред омогућеног напред – назад управљања подразумевали и контролу сензора и мотора. Мана поменутих сетова се огледала у томе што су LEGO конструкције биле жичано повезане са рачунарима. Челници су желели да направе LEGO коцку који би имала функцију рачунара, тј. желели су да иста буде потпуно програмабилна, чиме би се потреба за рачунаром смањила. Сама конструкција LEGO програмабилне коцке није била скроз једноставна. Цена LEGO коцке је морала да буде прилагођена корисницима, а њене димензије и маса LEGO сету, па је дизајнирање исте представљало прави изазов.

У јануару 1998, LEGO Mindstorms сет је представљен на краљевској академији уметности у Лондону, док је до децембра исте године и продат.

До сада су представљене три генерације LEGO Mindstorms сета: The Robotics Invention System (1998), Mindstorms NXT (2006) и Mindstorms EV3 (2013) [1].

3.1. LEGO RCX програмабилна коцка

Програмабилна коцка првог сета Robotics Invention System била је RCX Robotic Command eXplorers). Програмирање је било омогућено коришћењем RCX кода или ROBOLAB - а који се базирао на LabView софтверу. RCX програмабилна коцка, приказана на наредној слици, имала је процесор од 16 MHz, и 32 K RAM-а. Програмира се једноставним пребацивањем програма са рачунара у коцку. Такође је омогућено самостално програмирање робота преко RCX коцке, што је и био циљ LEGO компаније.

Поред коцке, сет је укључивао и два мотора, два сензора додира и један светлосни сензор. У скалду са тим, коцка садржи три улазна прикључка за сензоре, као и три излазна за моторе. LCD екран служи за приказивање основних информација везаних за ниво батерије, статус улазних и излазних уређаја и сл [1].

Slika 3.1.1. *LEGO RCX програмабилна коцка*

NXT програмабилна коцка је коришћена у овом раду, па ће бити касније детаљније представљена.

3.2. LEGO EV3 програмабилна коцка

LEGO EV3 програмабилна коцка представља последњи, најновији модел треће генерације LEGO Mindstorms сета. Званично је пуштена у продају 1. Септембра 2013. године. Садржи процесор од 300 MHz, као и процесор од 64 MB RAM меморије и 16 MB Flash меморије. Исту је могуће помоћу USB – а повезати са рачунаром, а такође је обезбеђена и WiFi и Bluetooth конекција. Програмирање је могуће помоћу рачунара и директно преко плоче.

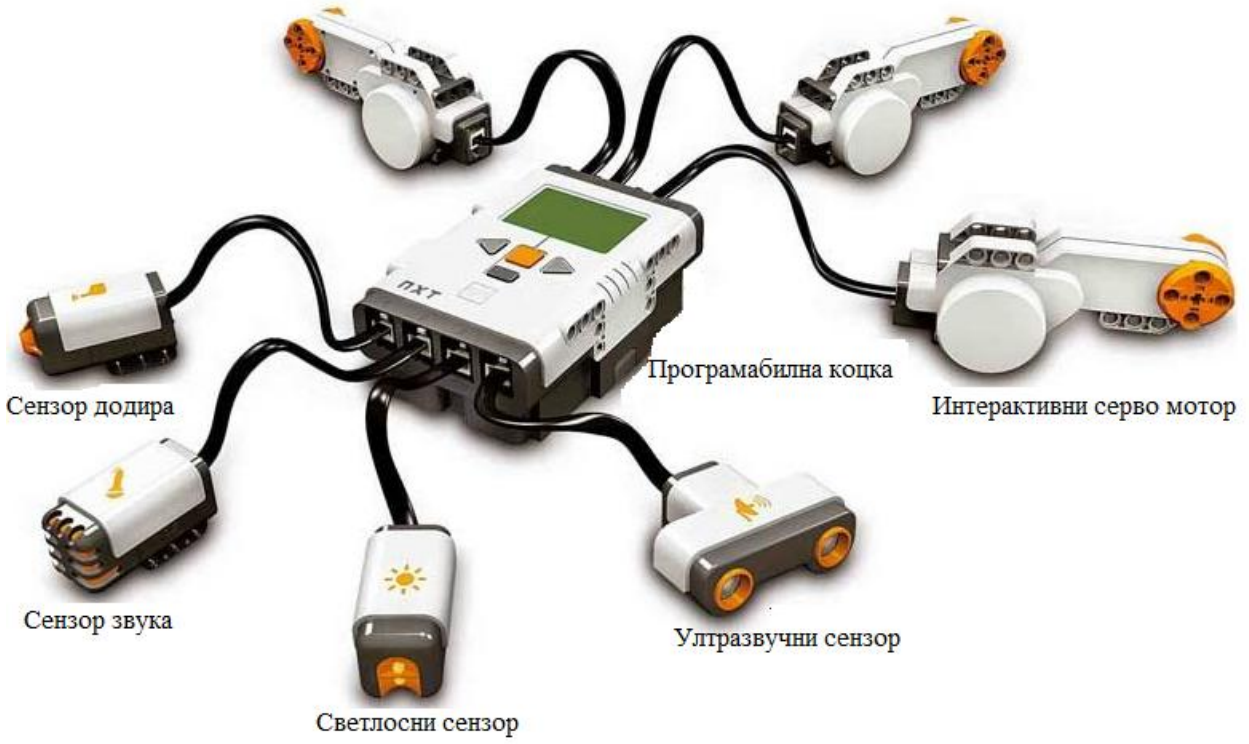
Едукациони сет користи LabView и садржи два велика и један средњи мотор, два сензора додира, један сензор боје, жироскопски и ултразвучни сензор. LEGO EV3 програмабилна коцка подржава и Linux оперативни систем. Приказана је на слици 4 [1,2].

Слика 3.2.1. *LEGO EV3 програмабилна коцка*

4. LEGO NXT ПРОГРАМАБИЛНИ РОБОТ

LEGO Mindstorms NXT пакет садржи различите компоненте намење за моделирање прилагодљивих, програмабилних робота. Поменути пакет садржи микро рачунар, такозвану NXT коцку, која управља системом, као и сет сензора намењених за мерење одговарајућих физичких величина. Поред сензора, LEGO пакет садржи и актуаторе тј. извршне органе попут мотора једносмерне струје, као и остале LEGO делове у виду коцкица различитих облика и димензија потребних за креирање жељених механичких система.

На наредној слици је приказана NXT коцка са прикљученим моторима са горње и сензорима са доње стране. У раду ће детаљно бити објашњење поменуте компоненте.



Слика 4.1. *LEGO NXT програмабилна коцка са деловима*

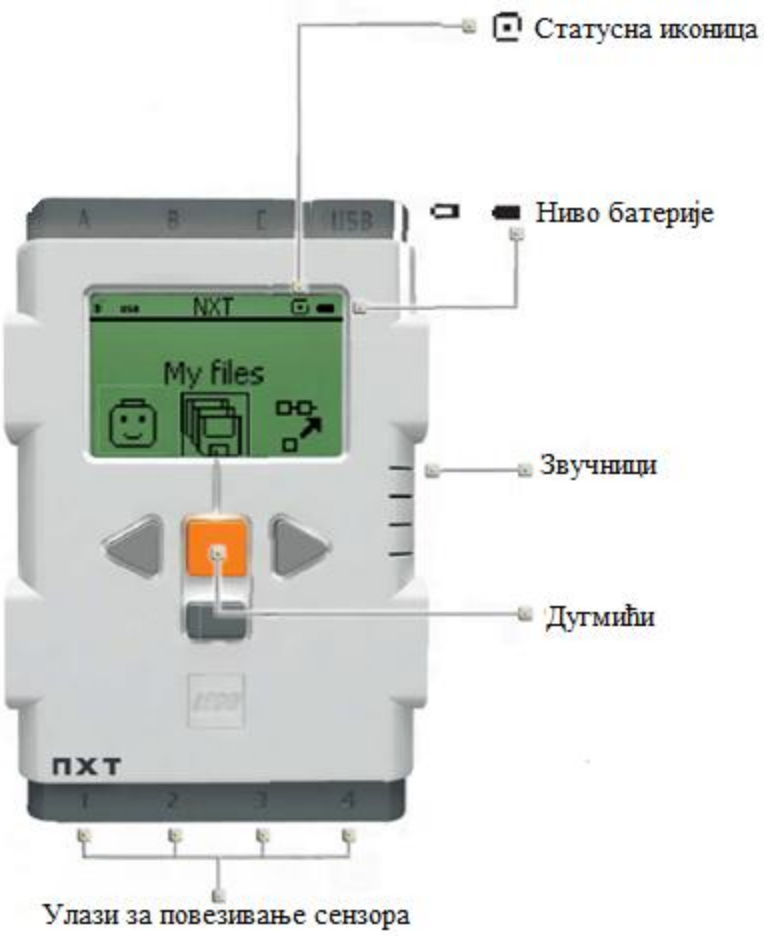
4.1. LEGO NXT програмабилна коцка

Основна компонента LEGO робота је поменути микро рачунар, NXT коцка. Повезивање LEGO NXT коцке са рачунаром је крајње једноставан процес. Потребно је исту укључити и помоћу USB кабла остварити конекцију са рачунаром. Након што рачунар идентификује нови уређај, аутоматски се врши инсталирање LEGO Mindstorms Education NXT софтвера и омогућава се коришћење коцке. NXT коцка обезбеђује програмирање робота у циљу добијања жељеног понашања.

Коцка је конструисана тако да обезбеђује управљање и до три мотора а поседује и четири улаза за одговарајуће сензоре. Иста садржи 32 – битни главни процесор (256 kB flash меморије и 64 kB RAM меморије). Поред поменутог, NXT коцка поседује 8 – битни микроконтролер (4 kB flash меморије и 512 Bytes RAM). На предњој страни овог микро рачунара налази се екран резолуције 60 × 100 пиксела. Екран омогућава приказивање менија и подешавања, као и приказивање осталих информација које се односе на потрошњу батерије као и коришћење Bluetooth уређаја. NXT коцка као напајање користи пуњиву литијумску батерију, која обезбеђује снагу потребну за покретање робота. Могуће је користити и шест AA батерија као и алкалне батерије. Батерије се постављају на полеђини коцке, а коришћење различитих врста батерија није препоручљиво. Уколико је ниво батерије испод 10 % на екрану се приказује светлеће обавестење да је иста празна.

Зарад комплетнијег приказивања компонената коцке, иста ће бити приказана на две наредне слике. На првој слици је приказана NXT коцка са поменутим екраном, где се уочавају иконица која се односи на ниво батерије, као и статусна исконица која се ротира када је коцка укључена. Са бочне стране коцке налази се звучник који ради уколико су звукови укључени у програм. На средини коцке се налази наранџасто дугме које омогућава укључивање робота и одабир жељеног менија. Поред наранџастог, два сива дугмета у облику стрелица са одговарајућих страна, омогућавају претрагу менија. Испод главног, наранџастог дугмета, налази се сиво дугме које служи за повратак на претходни мени као и за брисање. Такође могуће је променити име коцке која се у овом случају зове NXT, што је приказано на горњој линији екрана у средини. Име може садржати максимално осам карактера, а промену истог омогућава NXT прозор.

Са доње стране коцке налазе се четири улаза која су намењена за повезивање сензора. Улази су означени бројевима 1, 2, 3 и 4 [2].



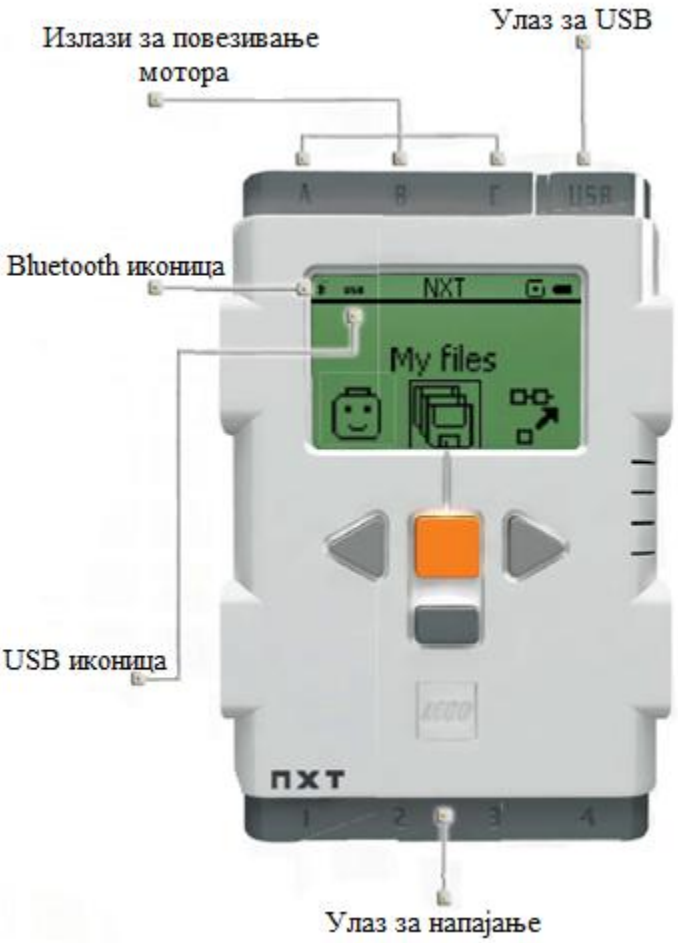
Слика 4.1.1. *LEGO NXT програмабилна коцка са означеним деловима (улази за сензоре, звучници, дугмићи, статусна иконица и ниво батерије)*

На наредној слици је такође приказана NXT коцка где се са горње стране исте уочава прикључак за USB. Прикључивањем USB кабла могуће је пребацити програм са рачунара на коцку. Такође омогућено је пребацивање података са коцке на рачунар. Уколико је плоча преко USB кабла повезана са рачунаром, поред имена плоче, у горњем делу екрана, појављује се иконица која означава да је поменута конекција остварена. Уколико се ова иконица не налази на екрани значу да је веза између плоче и рачунара прекинута.

Поред овог прикључка, такође са горње стране коцке, налазе се и три излаза (A, B, C) која служе за повезивање са моторима. Поред USB –а, комуникација се може остварити и помоћу Bluetooth-а На екрану, у горњем левом углу налази се Bluetooth иконица која одређује тренутни статус ове конекције. Уколико веза није успостављена, поменута иконица се не појављује на екрану. Уколико је успостављена, иконица се појављајује на екрану у горњем левом углу.

Такође, са доње стране се налази прикључак за напајање, који је користан уколико је неопходно да се батерије напуне. У том случају се коцка снабдева потребном снагом

преко одговарајућег напајања, тако што се адаптер конектује са истом преко утичнице са доње стране [2].



Слика 4.1.2. *LEGO NXT програмабилна коцка са означеним деловима (улаз за напајање, USB иконица, Bluetooth иконица, излази за повезивање мотора, прикључак за USB)*

4.1.1. NXT главни мени

У овом делу ће бити објашњен сам NXT мени, са доступним пакетима. Пре свега, на екрану се уочава главна My files датотека, где се складиште програми извршени преко NXT коцке и пребачени са рачунара. У овој датотеци се налазе три датотеке: Software files, NXT files и Sound Files. У првој датотеци се налазе програми пребачени са рачунара, у другој програми који су програмирани на коцки, док трећа датотека садржи звукове који су део пребачених програма. Приликом пребацивања фајлова, исти се аутоматски складиште у одговарајући фолдер [2].

Слика 4.1.1. *Главни мени са датотекама (Software files, NXT files, Sound Files)*

Такође, битно је нагласити да програмирање робота поред рачунара омогућава и сама NXT коцка, односно да рачунар није увек неопходан за програмирање. Уколико се програмирање робота остварује помоћу саме коцке потребно је користити мени NXT Program и успоставити жељено кретање робота. Такође је неопходно да одговарајући сензори и актуатори буду прикључени.

Још један доступан мени је View мени који омогућава тестирање жељених сензора или актуатора ради добијања тренутног стања о њиховим подацима. Потребно у овом менију изабрати иконицу која се односи на сензор или мотор који се тестира, а затим изабрати улаз на који је исти прикључен. Након тога се на екрану добијају подаци о датотекама сензора или мотора [2].

Слика 4.1.2. *Тестирање сензора/мотора помоћу View менија*

Settings мени служи за различите врсте подешавања NXT програмабилне коцке. Могуће је подешавати звучник, брисати претходно сачуване програме као и подешавати такозвани "Sleep mode". Подешавање звучника се врши у опсегу од 0 када је исти искључен до 4 када је звучник максимално појачан. "Sleep mode" омогућава искључивање коцке након одговарајућег времена тачније након 2, 5, 10, 30 или 60 минута. Такође бирањем опције "Never" ова опција се искључује а плоча остаје укључена све док је корисник не искључи. Поред поменутих опција, могуће је брисање свих података из одговарајућих датотека [2].

Слика 4.1.3. *Мени који се односи на подешавања NXT програмабилне коцке – Settings мени*

Поред наведених, доступан је и Bluetooth мени, клоји служи за подешавање wireless конекције између коцке и других Bluetooth уређаја, попут мобилних телефона или рачунара [2].

4.2. Сензори LEGO Mindstorms NXT пакета

Сензори су саставни део LEGO Mindstorms NXT сета. Исти представљају битне компоненте које служе за мерење одговарајућих величина. Основна функција сензора је омогућавање интеракције робота са околином где се помоћу повратне спреге добијају информације о окружењу робота и на основу истих доносе одлуке везане за понашање робота.

LEGO Mindstorms NXT пакет садржи сензор додира, сензор светлости, сензор звука, сензор боје, жироскопски сензор као и ултразвучни сензор.

4.2.1. Сензор додира

Сензор додира приказан на наредној слици је веома једноставан за употребу. Сензор додира представља прекидач који када је притиснут ради, а када је отпуштен не ради. Са предње стране се уочава наранџасти отвор на средини који се помера уназад када је притиснут тастер и употпуњује електрично коло, што доводи до протока струје, при чему се сензор покреће. Када се дугме отпусти, враћа се у првобитан положај а проток електричне енергије се зауставља.

У складу са наведеним, постоје само два стања у зависности од тога да ли је дугме притиснуто или није. Како сензор додира може заузети само два стања, тако је могуће произвести само две различите вредности. Уколико је дугме притиснуто сензор има вредност логичке 1, а уколико није сензор има вредност логичке 0. Помоћу овог сензора робот може да обавља различите задатке. Могуће је обезбедити кретање робота у жељеном смеру притиском тастера [3].

Слика 4.2.1. *Сензор додира*

4.2.2. Светлосни сензор

Овај сензор обезбеђује информације о јачини светлости. Пројектован је тако да одређује јачину светлости од најсветлијих до најтамнијих површина у свом окружењу. Такође пружа информације о јачини светлости на обојеним површинама. Очитавање се врши процентуално. Највиши проценат од 100 % одговара површинама које су највише осветљене, док се најмањи проценат 0 % постиже уколико сензор поставимо на места где има веома мало светлости.

Светлосни сензор се може користи за кретање робота, креирањем линије коју роботи прате помоћу сензора светлости [3].

Слика 4.2.2. *Светлосни сензор*

4.2.3. Сензор звука

LEGO NXT сензор звука обезбеђује информације о јачини звука у dB. Вредност се такође приказује у процентима, где 100 % одговара највишим вредностима, а 0 % најнижим вредностима. Максимална вредност коју овај сензор може да детектује је 90 dB.

Сензор звука обезбеђује извршавање многих задатака. На пример, могуће је направити програм где робот када детектује звук започиње жељену операцију и исту прекида када чује неки други звук [2].

Слика 4.2.3. *Сензор звука*

4.2.4. Ултразвучни сензор

Овај сензор налази велику примену и саставни је део LEGO сета, а најчешће се користи код задатака где је потребно да LEGO робот избегне неку препреку. Ултразвучни сензор одређује растојање између сензора и објеката у непосредној близини истог, слањем кратких звучних таласа до жељене препреке и мерењем времена потребног да талас стигне до препреке и одбије се назад до извора. Знајући брзину звука може се израчунати удаљеност од објекта. Најбоља очитавања су забележена када су објекти већих димензија, док је мерење удаљености од танких, малих или кривих објеката отежано. Такође се не препоручује истовремено коришћење два ултразвучна сензора јер је могуће да утичу један на другог.

Ултразвучни сензор је дигиталан сензор који мери удаљеност од препреке у центиметрима и инчима. Могуће је мерити удаљеност од 0 до 255 cm са прецизношћу од +/- 3 cm [3].

Слика 4.2.4. *Ултразвучни сензор*

4.2.5. Сензор за боју

На наредној слици је приказан сензор за боју и правилно постављање истог. Овај сензор може да разликује до шест боја (белу, зелену, плаву, црвену и црну). Добра карактеристика овог сензора је што може да ради као лампа која емитује светло различитих боја (плаво, зелено и црвено светло). Уколико сензор емитује плаво светло значи да робот мирује и да је комуникација са рачунаром успостављена тј. да робот може да прихвати задатак. Уколико је робот у покрету емитује зелено светло, док црвено светло емитује уколико удари у препреку.

Сензор за боју је веома користан јер се робот може програмирати тако да различито реагује на сваку боју и сл [3].

Слика 4.2.5. *Позиционирање сензора за боју (лево) и сензор за боју (десно)*

4.2.6. Жироскопски сензор

Жироскоп је конструисан тако да да ради беспрекорно са LEGO NXT плочом. Повезује се се истом помоћу стандардне жице и користи аналогни интерфејс. Жироскоп садржи једноосни сензор који детектује ротацију и враћа вредност која представља број степени по секунди ротације.

Да би се добило прецизно очитавање некада је неопходно читати вредности из једне позиције током неког времена (на пример 2 секунде) и из тих очитавања наћи средњу вредност. Та средња вредност офсета се мозе касније користити у даљим калкулацијама ради повецања прецизности.

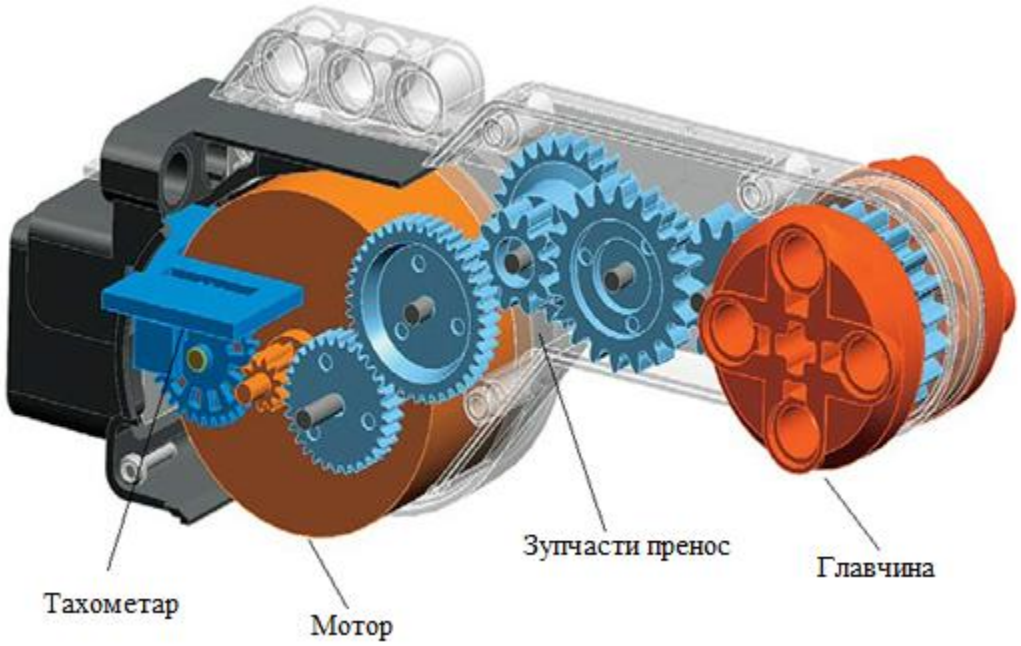
Оса мерења је у вертикалној равни са зсензором постављеним тако да је његов црни део окренут нагоре, као сто је представљено на слици 4.2.6. Сензор може да чита 300 пута у секунди, што се слаже са брзином узорка од 300 Hz [4].

Слика 4.2.6. *Жироскопски сензор (лево) и опсег сензора (десно)*

4.3. LEGO NXT серво мотор

Интерактивни LEGO NXT серво мотор представља скуп више делова као што су зупчаници, ротациони сензор или тахометар, мотор и главчина. Серво мотори представљају актуаторе LEGO роботу где претварањем електричне енергије добијене из извора напајања у механичку, омогућавају различите начине кретања у зависности од намене.

LEGO серво мотори су интерактивни, што значи да поседују уграђени ротациони сензор тј. тахометар. Тахометар мери брзину обртања на основу чега се добија обртање осовине у степенима или пуним обртајима, са грешком од $\pm 1°$. Овакав начин мерења омогућава прецизније управљање роботиским покретима. На наредној слици је приказан пресек серво мотора ради приказивања делова истог [2,4]



Слика 4.3.1. *Интерактивни LEGO NXT серво мотор са означеним деловима*

Номиналне карактеристике мотора:

- Напајање 9 V
- Обртни момент 16.7 Ncm
- Угаона брзина 177 min^{-1}
- Снага 2.03 W
- Ефикасност 41 %

LEGO NXT програмабилни робот садржи три серво мотора, од којих су два погонска мотора директно везана на точкове робота, док трећи служи за ротацију ултразвучног сензора. Ротација сензора омогућава мерење удаљености робота од препрека у различитим смеровима. Омогућено је задавање смера окретања мотора, а регулација се изводи преко ПИД регулатора. Управљање моторима се врши помоћу PWM сигнала, што ће бити објашњено у даљем раду.

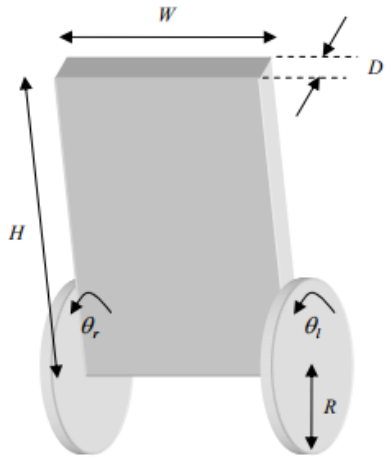
LEGO робот користи инкрементални енкодер. Овај енкодер поседује два сигнала фазно померана, за добијање информација о смеру и обртању мотора. Енкодери обезбеђују информације о окретању осовине мотора са великом тачношћу.

Поред NXT програмабилне плоче, сензора и актуатора, саставни делови LEGO пакета су и точкови који обезбеђују кретање LEGO робота. У зависности од купљеног пакета, точкови могу бити различитих димензија и маса. На наредној слици су приказане три врсте точкова који се монтирају на LEGO робота [5].

Слика 4.3.1. *Точкови различитих димензија LEGO NXT робота*

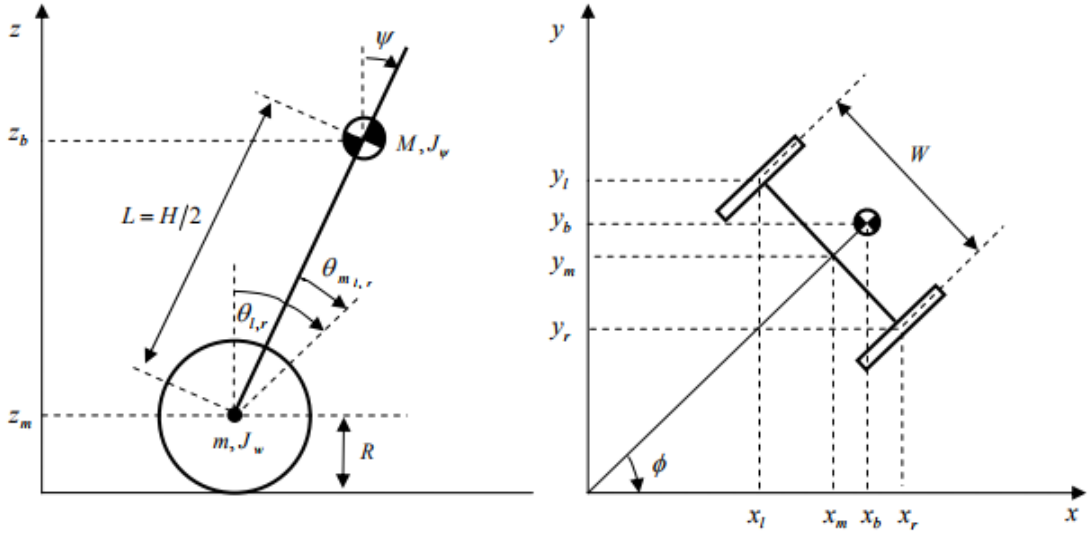
5. МОДЕЛ СИСТЕМА У ПРОСТОРУ СТАЊА

Математички модел објекта управљања знатно олакшава управљање, па га је зато неопходно извести. На основу математичког модела добијају се једначине кретања робота на основу којих се даље дизајнира контолер. Мобилни робот спада у групу малих робота, па се LEGO самобилансирајући робот може посматрати као модел инверзног клатна, што је приказано на слици 5.1.



Слика 5.1. *Модел инверзног клатна на два точка*

На наредној слици је приказан поглед са стране и поглед одозго на модел инверзног клатна са точковима. Такође су означене и генералисане координате система: вертикални нагиб клатна ψ, угао левог и десног точка $\theta_{l,d}$, угао левог и десног мотора $\theta_{ml,d}$ и угао заокретања клатна у хоризонталној равни φ [6].



Слика 5.2. *Поглед са стране модела инверзног клатна (лево) и поглед одозго (десно)*

Физички параметри LEGO NXT самобалансирајућег робота су приказани у наредној табели

Табела 1. Физички параметри LEGO NXT самобалансирајућег робота

Вредност	Ознака	Величина
$\boldsymbol{g = 9.81}$	$[m/s^2]$	Убрзање земљине теже
$\boldsymbol{m = 0.03}$	$[kg]$	Маса точка
$\boldsymbol{R = 0.04}$	$[m]$	Полупречник точка
$\boldsymbol{J_w = mR^2}$	$[kgm^2]$	Момент инерције точка
$\boldsymbol{M = 0.6}$	$[kg]$	Маса робота
$\boldsymbol{W = 0.14}$	$[m]$	Ширина робота
$\boldsymbol{D = 0.04}$	$[m]$	Дебљина робота
$\boldsymbol{H = 0.144}$	$[m]$	Висина робота
$\boldsymbol{L = H/2}$	$[m]$	Растојање центра масе од осе точка
$\boldsymbol{J_\psi = ML^2/3}$	$[kgm^2]$	Момент инерције за вертикално нагињање
$\boldsymbol{J_\varphi = M(W^2 + D^2)/12}$	$[kgm^2]$	Момент инерције за хоризонтално нагињање
$\boldsymbol{J_m = 1 \times 10^{-5}}$	$[kgm^2]$	Момент инерције ротора мотора
$\boldsymbol{R_m = 6.69}$	$[\Omega]$	Статички електрични отпор мотора
$\boldsymbol{K_b = 0.468}$	$[Vs/rad]$	Коефицијент повратне електромоторне силе
$\boldsymbol{K_t = 0.317}$	$[Nm/A]$	Коефицијент момента мотора
$\boldsymbol{n = 1}$		Преносни однос зупчаника
$\boldsymbol{f_m = 0.0022}$		Коефицијент трења између мотора и тела робота
$\boldsymbol{f_w = 0}$		Коефицијент трења између точка и подлоге

Једначине кретања инверзног клатна се добијају коришћењем Лагранжеових једначина на основу координатног система приказаног на претходнох слици. Координате везане за тачку M која је центар точка се добијају на следећи начин:

$$(x_m,\ y_m, z_m) = (R\theta cos\varphi, R\theta sin\varphi, R) \tag{1}$$

Затим се могу изразити угао заокретања клатна у хоризонаталној равни φ и средња вредност заокретнутости точка θ у функцији левог и десног угла точка $\theta_{l,d}$ на следећи начин:

$$(\theta, \varphi) = \left(\frac{1}{2}(\theta_l + \theta_r), \frac{R}{W}(\theta_r - \theta_l)\right) \tag{2}$$

Координате левог и десног точка се могу изразити у функцији од координата центра, тачније од тачке М.

$$(x_{l,}\, y_{l,}\, z_l) = \left(x_m - \frac{W}{2} sin\varphi, y_m + \frac{W}{2} cos\varphi,\ z_m\right) \tag{3}$$

$$(x_{r,}\, y_{r,}\, z_r) = \left(x_m + \frac{W}{2} sin\varphi, y_m - \frac{W}{2} cos\varphi,\ z_m\right) \tag{4}$$

Такође се могу извести и координате центра масе система, тачке Б.

$$(x_{b,}\, y_{b,}\, z_b) = (x_m + Lsin\psi cos\varphi, y_m + Lsin\psi cos\psi,\ z_m + Lcos\psi) \tag{5}$$

Како бисмо извели Лагранжеове једначине потребно је израчунати кинетичку енергију транслације T_t, кинетичку енергију ротације T_r и потенцијалну енергију *U*.

Кинетичка енергија транслације T_t је дата следећом релацијом:

$$T_t = \frac{1}{2} m(\dot{x}_l^2 + \dot{y}_l^2 + \dot{z}_l^2) + \frac{1}{2} m(\dot{x}_r^2 + \dot{y}_r^2 + \dot{z}_r^2) + \frac{1}{2} M(\dot{x}_b^2 + \dot{y}_b^2 + \dot{z}_b^2) \tag{6}$$

Ротациона кинетичка енергија се рачуна на следећи начин:

$$T_r = \frac{1}{2} J_w \dot{\theta}_l^2 + \frac{1}{2} J_w \dot{\theta}_r^2 + \frac{1}{2} J_\psi \dot{\psi}_l^2 + \frac{1}{2} J_\varphi \dot{\varphi}_l^2 + \frac{1}{2} n^2 J_m (\dot{\theta}_l - \dot{\psi})^2 + \frac{1}{2} n^2 J_m (\dot{\theta}_r - \dot{\psi})^2 \tag{7}$$

Потенцијална енергија је дата следећом релацијом:

$$U = mgz_l + mgz_r + Mgz_b \tag{8}$$

Лагранжијан L се рачуна као разлика кинетичке и потенцијалне енергије.

$$L = T_t + T_r - U \tag{9}$$

Како систем поседује три генералисане координате θ, и φ, имаће и три Лагранжеове једначине.

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = F_\theta \tag{10}$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\psi}}\right) - \frac{\partial L}{\partial \psi} = F_\psi \tag{11}$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\varphi}}\right) - \frac{\partial L}{\partial \varphi} = F_{\varphi} \tag{12}$$

Заменом познатих величина и сређивањем претходних једначина добија се:

$$[(2m + M)R^2 + 2J_w + 2n^2J_m]\ddot{\theta} + (MLRcos\psi - 2n^2J_m)\psi - ML\ddot{R}sin\psi = F_{\theta} \tag{13}$$

$$\begin{gathered}(MLRcos\psi - 2n^2J_m)\ddot{\theta} + \left(ML^2 + J_{\psi} + 2n^2J_m\right)\ddot{\psi} - MgLsin\psi \\ - ML^2\varphi^2 sin\psi cos\psi = F_{\psi}\end{gathered} \tag{14}$$

$$\left[\frac{1}{2}mW^2 + J_{\varphi} + \frac{W^2}{2R^2}(J_w + n^2J_m) + ML^2sin^2\psi\right]\ddot{\varphi} + 2ML^2\dot{\psi}\dot{\varphi}sin\psi cos\psi = F_{\varphi} \tag{15}$$

Уколико се узму у обзир обртни момент мотора и вискозно трење, генералисане силе се добијају на следећи начин:

$$\left(F_{\theta}, F_{\psi}, F_{\varphi}\right) = \left(F_l + F_r, F_{\psi}, \frac{R}{W}(F_r - F_l)\right) \tag{16}$$

$$F_l = nK_t i_l + f_m\left(\dot{\psi} - \dot{\theta}_l\right) - f_w\dot{\theta}_l \tag{17}$$

$$F_r = nK_t i_r + f_m\left(\dot{\psi} - \dot{\theta}_r\right) - f_w\dot{\theta}_r \tag{18}$$

$$F_{\psi} = -nK_t i_l - nK_t i_r - f_m\left(\dot{\psi} - \dot{\theta}_l\right) - f_m\left(\dot{\psi} - \dot{\theta}_r\right) \tag{19}$$

У претходним релацијама, $i_{l,r}$ је струја електромотора.

Пошто се мотором управља напоноским PWM сигналом, не може се користити струја директно, па је потребно одредити однос струје $i_{l,r}$ и напона $v_{l,r}$ коришћењем једначина мотора једносмерне струје.

Једначина мотора једносмерне струје, када се занемари трење, дата је следећом релацијом:

$$L_m i_{l,r} = v_{l,r} + K_b\left(\dot{\psi} - \dot{\theta}_l\right) - R_m i_{l,r} \tag{20}$$

Уз претпоставку да је индуктивност мотора занемарљива, струја мотора се рачуна на следећи начин:

$$i_{l,r} = \frac{v_{l,r} + K_b\left(\dot{\psi} - \dot{\theta}_l\right)}{R_m} \tag{21}$$

Преко напона на мотору могуће је изразити генералисане силе, што је приказано наредним изразима:

$$F_\theta = \frac{\alpha}{2}(v_i + v_r) - (\beta + f_w)\dot{\theta} + \beta\dot{\psi} \tag{22}$$

$$F_\psi = -\alpha(v_i + v_r) + 2\beta\dot{\theta} - 2\beta\dot{\psi} \tag{23}$$

$$F_\varphi = \frac{R}{W}\alpha(v_r - v_l) - \left(\beta + \frac{W}{R}f_w\right)\dot{\varphi} \tag{24}$$

Где су:

$$\alpha = \frac{nK_t}{R_m} \tag{25}$$

$$\beta = \frac{nK_tK_b}{R_m} + f_m \tag{26}$$

5.1. Једначине стања инверзног клатна на два точка

Применом теорије аутоматског управљања једначине стања се могу извести линеаризацијом једнаћина кретања око равнотежне тачке NXT система. То значи да се усваја ограничење $\psi \to 0$ $(sin\psi \to \psi,\ cos\psi \to 1)$ а да се елементи другог реда попут $\dot{\psi}^2$ занемарују [6].

$$[(2m + M)R^2 + 2J_w + 2n^2J_m]\ddot{\theta} + (MLR - 2n^2J_m)\ddot{\psi} = F_\theta \tag{27}$$

$$(MLR - 2n^2J_m)\ddot{\theta} + \left(ML^2 + J_\psi + 2n^2J_m\right)\ddot{\psi} - MgL\psi = F_\psi \tag{28}$$

$$\left[\frac{1}{2}mW^2 + J_\varphi + \frac{W^2}{2R^2}(J_w + n^2J_m)\right]\ddot{\varphi} = F_\varphi \tag{29}$$

Уочава се да претходне једначине садрже само генералисане координате. Једначине садрже генералисану координату θ, док једначина садржи само генералисану координату φ, па се могу матрично изразити:

$$E\begin{bmatrix}\ddot{\theta}\\ \ddot{\psi}\end{bmatrix}+F\begin{bmatrix}\dot{\theta}\\ \dot{\psi}\end{bmatrix}+G\begin{bmatrix}\theta\\ \psi\end{bmatrix}=H\begin{bmatrix}v_l\\ v_r\end{bmatrix} \tag{30}$$

$$E=\begin{bmatrix}(2m+M)R^2+2J_w+2n^2J_m & MLR-2n^2J_m\\ MLR-2n^2J_m & ML^2+J_\psi+2n^2J_m\end{bmatrix} \tag{31}$$

$$F=\begin{bmatrix}\beta+f_w & -\beta\\ -2\beta & 2\beta\end{bmatrix} \tag{32}$$

$$G=\begin{bmatrix}0 & 0\\ 0 & -MgL\end{bmatrix} \tag{33}$$

$$H=\begin{bmatrix}\alpha & \alpha\\ -\alpha & -\alpha\end{bmatrix} \tag{34}$$

$$K\ddot{\varphi}+I\dot{\varphi}=J(v_r-v_l) \tag{35}$$

$$I=\beta+\frac{W}{R}f_w \tag{36}$$

$$J=\frac{R}{W}\alpha \tag{37}$$

$$K=\frac{1}{2}mW^2+J_\varphi+\frac{W^2}{2R^2}(J_w+n^2J_m) \tag{38}$$

Даље се промењиве $\mathbf{x_1}$ и $\mathbf{x_2}$ сматрају векторима стања док је $\mathbf{u}$ вектор улаза.

$$\mathbf{x_1}=[\theta,\psi,(\theta,)\dot{}\psi\dot{}\,]^T,\quad \mathbf{x_2}=[\varphi,\dot{\varphi}]^T,\quad \mathbf{u}=[v_r,v_l]^T \tag{39}$$

Сада се могу извести једначине стање инверзног клатна на два точка у матричном облику:

$$\dot{\mathbf{x_1}}=A_1\mathbf{x_1}+B_1\mathbf{u} \tag{40}$$

$$\dot{\mathbf{x_2}}=A_2\mathbf{x_2}+B_2\mathbf{u} \tag{41}$$

$$A_1=\begin{bmatrix}0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\\ 0 & A_1(3,2) & A_1(3,3) & A_1(3,4)\\ 0 & A_1(4,2) & A_1(4,3) & A_1(4,4)\end{bmatrix} \tag{42}$$

$$B_1 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ B_1(3) & B_1(3) \\ B_1(4) & B_1(4) \end{bmatrix} \tag{43}$$

$$A_2 = \begin{bmatrix} 0 & 1 \\ 0 & -I/K \end{bmatrix}, \qquad B_2 = \begin{bmatrix} 0 & 0 \\ -J/K & J/K \end{bmatrix} \tag{44}$$

Даље се рачунају елементи матрица:

$$A_1(3,2) = -MgLE(1,2)/\det(E) \tag{45}$$

$$A_1(4,2) = -MgLE(1,1)/\det(E) \tag{46}$$

$$A_1(3,3) = -[(\beta + f_w)E(2,2) + 2\beta E(1,2)]/\det(E) \tag{47}$$

$$A_1(4,3) = [(\beta + f_w)E(1,2) + 2\beta E(1,1)]/\det(E) \tag{48}$$

$$A_1(3,4) = -\beta(E(2,2) + 2E(1,2))/\det(E) \tag{49}$$

$$A_1(4,4) = -\beta(E(1,2) + 2E(1,1))/\det(E) \tag{50}$$

$$B_1(3) = \alpha(E(2,2)/2 + E(1,2))/\det(E) \tag{51}$$

$$B_1(4) = -\alpha(E(1,2)/2 + E(1,1))/\det(E) \tag{52}$$

$$\det(E) = E(1,1)E(2,2) - E(1,2)^2 \tag{53}$$

6. ФОРМИРАЊЕ ПОВРАТНЕ СПРЕГЕ ПО СТАЊУ

За формирање повратне спреге по стању потребно је пре свега дефинисати основне појмове, као што је сам концепт стања, променљиве стања, вектор стања и сл.

Стање динамичког система представља скуп најмањег броја променљивих које се називају променљиве стања. Познавање ових променљивих у тренутку $t = t_0$, као и у тренутку $t \geq t_0$ комплетно одређује понашање система у сваком тренутку за које је $t \geq t_0$.

Променљиве стања динамичког система су променљиве које описају стање истог. Уколико са n променљивих описујемо понашање динамичког система, као и уколико су познати почетни услови када је $t = t_0$ и улаз у систем када је $t \geq t_0$, онда је стање динамичког система у потпуности дефинисано. Треба напоменути да променљиве стања није увек могуће мерити. То могу бити променљиве које нису мерљиве нити опсервабилне, што не искључује могућност да исте буду променљиве стања. Наведена особина је велика предност описаних система.

Поред концепта стања и променљивх стања, вектор стања се такође дефинише. Уколико је n променљивих стања потребно да се опише понашање система, онда се n променљивих стања посматрају као компоненте вектора **x**. Овај вектор се назива вектор стања. Вектор стања одређује стање *x(m)* система у сваком временском тренутку $t \geq t_0$, уколико је познато почетно стање у тренутку $t = t_0$, као и улаз система *y(m)* у тренутку $t \geq t_0$.

Променљиве стања се приказају у простору стања. Простор стања представља n димензионалан простор састављен од x оса ($x_1, x_2, x_3 \ldots x_n$).

Оваква врста анализе поред укључује три врсте променљивих: улазне, излазне као и променљиве стања [7].

Једначина стања и једначина излаза линеарног континуланог система су приказане следећим изразима:

$$\dot{\mathbf{x}}(\mathrm{t}) = \mathbf{A}\mathbf{x}(\mathrm{t}) + \mathbf{B}\mathbf{u}(\mathrm{t}) \tag{54}$$

$$\mathbf{y}(\mathrm{t}) = \mathbf{C}\mathbf{x}(\mathrm{t}) + \mathbf{D}\mathbf{u}(\mathrm{t}) \tag{55}$$

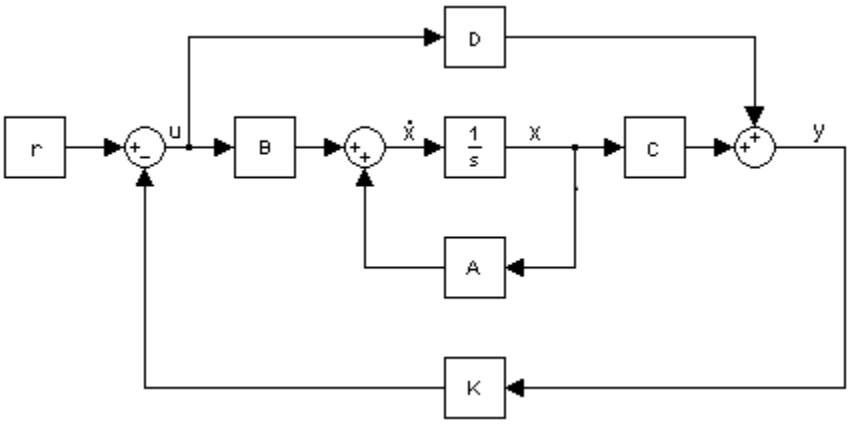
$$\mathbf{u}(t) = \mathbf{r}(t) - \mathrm{К}\mathbf{x}(t) \tag{56}$$

У првој релацији $\dot{\mathbf{x}}(\mathrm{t})$ представља вектор промене стања, **A** је матрица стања, док је **B** матрица улаза.

У другој једначини $\mathbf{y}(\mathrm{t})$ представља вектор излаза, **C** је матрица излаза а **D** матрица спрегнутог улаза.

К у трећој релацији представља појачање повратне спреге по стању, док је **u** вектор улаза.

Структура повратне спреге по стању је приказана је на слици 6.1.



Слика 6.1. *Повратна спрега по стању*

Након замене последње релације у прву и сређивања, добија се:

$$\dot{\mathbf{x}} = (A - BK)\mathbf{x} + B\mathbf{r} \rightarrow \dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{r} \tag{57}$$

$$\mathbf{y} = (C - DK)\mathbf{x} + D\mathbf{r} \rightarrow \mathbf{c} = C\mathbf{x} + D\mathbf{r} \tag{58}$$

Како стабилност линераних система аутоматског управљања не зависи од величине улазних сигнала, она је одређена матрицом $(A - BK)$.

Уколико једначина $det[sI - A - BK] = 0$ има све леве корене, тада је посматрани систем стабилан. Повратна спрега по стању утиче на стабилност. Уколико је систем контролабилан и опсервабилан помоћу повратне спреге по стању систем може постати стабилан.

Повратна спрега по стању се пројектује ради добијања жељеног спектра полова или у циљу остваривања жељеног критеријума квалитета.

У овом раду је иста пројектована ради оптимизације критеријума и одређивања појачања K преко линеарно квадратне контроле - *LQ control*, што ће бити објашњено у даљем раду [7].

6.1. Серво контрола

Основни разлози за коришћење серво контролера се односе на утицај истих на смањивање грешке стационарног стања, смањивање осетљивости параметара као и на побољшање времена одзива система. Бржи одзив система такође условљава и брже успотављање стационарног стања, док смањивање грешке стационарног стања условљава

већу стабилност система аутоматског управљања. Мања осетљивост улазних и излазних параметара код серво система значи да исти толеришу одступања од одређене вредности параметара, што је још једна позитивна карактеристика ових система.

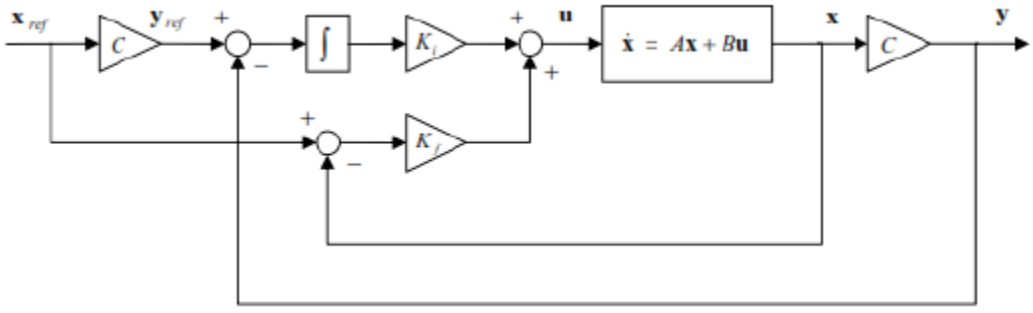
Серво контролу можемо поделити у две класе, које се односе на различите проблем. Прва класа серво контроле се бави праћењем задатог управљања, где је основни задатак да стварно понашање система буде идентично жељеном. Оваква контрола се назива "Feedforward**".** Иста обезбеђује управљање изводи без грешке, под претпоставком да је математички модел система тачан а оптерећења позната.

Друга класа серво контроле се односи на уклањање поремећаја у систему. Поремећаји могу бити унутрашњи и спољашњи. Унутрашњи се налазе унутар система, док се спољашњим поремећајима сматра утицај окружења. За уклањање поремећаја користе се контролери, од којих је најпознатији ПИД контролер. За разлику од Feedforward контроле која реализује управљање у циљу обезбеђивања нулте грешке стационарног стања, серво контролери регулишу непознате поремећаје и грешке у моделирању.

Серво контрола комбинује претходно описана два типа серво контроле у циљу обезбеђивања најбољих перформанси [8].

6.1.1. Серво контролер

Претходно је био описан појам серво контроле, која укључује серво контролере. Серво системи укључују интеграторе коју уклањају грешку стационарног стања на одскочне улазе. На наредној слици је приказан блок дијаграм ПИД серво контролера у симулинку.



Слика 6.1.1. *Серво контролер у Симулинку*

Једначине стања и излаза су приказане наредним формулацијама:

$$\dot{\mathbf{x}}(\mathrm{t}) = \mathrm{A}\mathbf{x}(\mathrm{t}) + \mathrm{Б}\mathbf{u}(\mathrm{t}) \tag{59}$$

$$\mathbf{y}(\mathrm{t}) = \mathrm{C}\mathbf{x}(\mathrm{t}) \tag{60}$$

Како је грешка управљања улазна променљива контролера, потребно је исту израчунати као разлику естимиарног стања **x** и жељеног стања $\mathbf{x_{ref}}$**:**

$$\boldsymbol{e}(t) = C(\mathbf{x}(t) - \mathbf{x_{ref}}(t)) \tag{61}$$

Једначина стања за интеграл грешке **z**(t) се рачуна на следећи начин:

$$\dot{\boldsymbol{z}}(t) = C(C(\mathbf{x}(t) - \mathbf{x_{ref}}(t)) \tag{62}$$

Вектор стања проширеног система:

$$\bar{\mathbf{x}} = [\mathbf{x}(t), \mathbf{z}(t)]^T \tag{63}$$

Редови вектора **x** и **u** су су **n**, **m**.

Једначине стања система са интегратором:

$$\begin{bmatrix}\dot{\mathbf{x}}(t)\\ \dot{\mathbf{z}}(t)\end{bmatrix} = \begin{bmatrix}A & 0_{n\times m}\\ C & 0_{m\times m}\end{bmatrix}\begin{bmatrix}\mathbf{x}(t)\\ \mathbf{z}(t)\end{bmatrix} + \begin{bmatrix}B\\ 0_{m\times m}\end{bmatrix}\mathbf{u}(t) - \begin{bmatrix}0_{n\times m}\\ I_{m\times m}\end{bmatrix} C\mathbf{x}_{\text{реф}} \tag{64}$$

Уз претпоставку да је систем стабилан претходна једначина конвергира у:

$$\begin{bmatrix}\dot{x}(\infty)\\ z(t)\end{bmatrix} = \begin{bmatrix}A & 0_{n\times m}\\ C & 0_{m\times m}\end{bmatrix}\begin{bmatrix}x(\infty)\\ z(\infty)\end{bmatrix} + \begin{bmatrix}B\\ 0_{m\times m}\end{bmatrix}u(\infty) - \begin{bmatrix}0_{n\times m}\\ I_{m\times m}\end{bmatrix} C\mathbf{x}_{\text{реф}} \tag{65}$$

Уколико од једначине (65) одутзмемо (64) добијамо једначину стања:

$$\begin{bmatrix}\dot{\mathbf{x}}_{\mathrm{e}}(t)\\ \dot{\mathbf{z}}_{\mathrm{e}}(t)\end{bmatrix} = \begin{bmatrix}A & 0_{n\times m}\\ C & 0_{m\times m}\end{bmatrix}\begin{bmatrix}\mathbf{x_e}(t)\\ \mathbf{z_e}(t)\end{bmatrix} + \begin{bmatrix}B\\ 0_{m\times m}\end{bmatrix}\mathbf{u_e}(t) \rightarrow \frac{d}{dt}\bar{\mathbf{x}}_{\mathbf{e}}(t) = \bar{A}\bar{\mathbf{x}}_{\mathbf{e}}(t) + \bar{B}\bar{\mathbf{u}}_{\mathbf{e}}(t) \tag{66}$$

Где су:

$$\mathbf{x_e} = \mathbf{x}(t) - \mathbf{x}(\infty), \qquad \mathbf{z_e} = \mathbf{z}(t) - \mathbf{z}(\infty), \qquad \mathbf{u_e} = \mathbf{u}(t) - \mathbf{u}(\infty) \tag{67}$$

Да би систем био стабилан уводи се повратна спрега по стању:

$$\mathbf{u_e}(t) = -K\bar{\mathbf{x}}_\mathbf{e}(t) = -K_f\mathbf{x_e}(t) - K_i\mathbf{z_e}(t) \tag{68}$$

Уколико су испуњени услови x(∞)→x_{ref}, z(∞)→0, y(∞)→0, тада је улаз u(t):

$$\mathbf{u}(t) = -K_f(\mathbf{x}(t) - \mathbf{x_{ref}}(t)) - K_i C\int(\mathbf{x}(t) - \mathbf{x_{ref}}(t))\,dt \tag{69}$$

Постоје два начина за израчунавање појачања повратне спреге по стању K_f и K_i.:

- Подешавањем полова
- LQ контролом

У овом раду појачања су израчуната Линеарном квадратном контролом [6].

6.2. Линеарна квадратна контрола

Код пројектовања дигиталних система претпоставља се дас у захтеви у погледу квалитета динамичког понашања задата жељеним спектром полова или преко квадратног индекса перформанце који треба минимизирати дуж трајкторије система у простору стања. Индекс перформансе је приказан наредном формулом:

$$J_N = \sum_{k=0}^{N} L[\mathbf{c}(k),\ \mathbf{r}(k),\ \mathbf{u}(k)] \tag{70}$$

У претходном изразу вектори **c, r, u** су вектори излаза, улаза и управљачких променљивих.

Потребно је успоставити оптимално управљање које за одабрану функцију минимизира неки критеријум. LQ контрола подразумева оптимално управљање које значи минимизацију неког критеријума.

Индекс перформансе је дат следећом релацијом:

$$J_N = \sum_{k=0}^{N}(\mathbf{x^T}(k)Q\mathbf{x}(k) + \mathbf{u^T}(k)R\mathbf{u}(k)) \tag{71}$$

Где су Q и R реалне константне матрице.

Претпоставља се да је модел објекта управљања дат у виду диференцијалне једначине стања и алгебарске једначине израза:

$$\dot{\mathbf{x}}(\mathrm{t}) = \mathrm{A}\mathbf{x}(\mathrm{t}) + \mathrm{B}\mathbf{u}(\mathrm{t}) \tag{72}$$

$$\mathbf{y}(\mathrm{t}) = \mathrm{C}\mathbf{x}(\mathrm{t}) \tag{73}$$

Дискрететизацијом ових једначина добија се једначина објекта управљања:

$$x(k+1) = E\mathbf{x}(k) + F\mathbf{u}(k) \tag{74}$$

Где су:

$$E(T) = e^{\boldsymbol{AT}} \tag{75}$$

$$F(T) = (\int_0^T e^{\boldsymbol{A}\tau}\, d\tau)B \tag{76}$$

У претходним изразима $e^{\boldsymbol{AT}}$ представља фундаменталну матрицу система, док је T периода одабирања.

Прво се бирају тежинске матрице Q и R у циљу добијања жељеног понашања система. При томе матрица Q се мора бирати као позитивно дефинитна или семидефинитна. Уколико је матрица Q позитивна тада је $\sum_{k=0}^{N} \mathbf{x^T}(k)\mathbf{Q}\mathbf{x}(k)$ позитивна у свим тачкама простора стања. Правилним избором Q матрице систем се оптимизује у погледу брзине реаговања и претека стабилности.

Поред Q матрице, подешава се и R матрица за коју је битно да буде позитивно дефинитна, реална и симетрична. Минимизацијом $\sum_{k=0}^{N} \mathbf{u^T}(k)\mathbf{R}\mathbf{u}(k)$ смањују се модули амплитуде управљачких променљивих на улазима објекта управљања тј. утиче се да управљачке променљиве буду у опсезима које могу да обрађују извршни органи.

Закључује се да се усвајањем позитивно дефинитне или позитивно семидефинитне матрице Q и усвајањем позитивно дефинитне симетричне матрице R добија аналитичко решење за оптимално управљање:

$$\mathbf{u}(k) = -K\mathbf{x}(k) \tag{77}$$

У претходној релацији К представља матрицу појачања. Потребно је одредити коефицијенте ове матрице [9].

6.2.1. Одређивање појачања К

Динамичким програмирањем се опримална стратегија одређује уназад. Рачуна се оптимално управљање и индекс перформансе од тренутка одабирања N. У тренутку N-j вредност индекса перформансе је:

$$S_{j+1} = S_j^0 + \mathbf{x}^{\mathbf{T}}(N-j)Q\mathbf{x}(N-j) + \mathbf{u}^{\mathrm{T}}(N-j)R\mathbf{u}(N-j) \tag{78}$$

Оптимално управљање:

$$\boldsymbol{m}^0(N-j) = -[\boldsymbol{F}^T\boldsymbol{P}(N-j+1)\boldsymbol{F} + \boldsymbol{R}]^{-1}\boldsymbol{F}^T\boldsymbol{P}(N-j+1)\boldsymbol{E}x(N-j) \tag{79}$$

Минимизиран индекс перформансе:

$$S_{j+1}^0 = x^T(N-j)P(N-j)x(N-j) \tag{80}$$

Где је P реална и симетрична матрица. Добија се као решење Рикатијеве једначине за дискретни систем [9].

$$\boldsymbol{P}(N-j) = \boldsymbol{E}^T\boldsymbol{P}(N-j+1)[\boldsymbol{E} - \boldsymbol{F}\boldsymbol{K}(N-j)] + \boldsymbol{Q} \tag{81}$$

Резултат извршења алгоритма даје оптимално управљање:

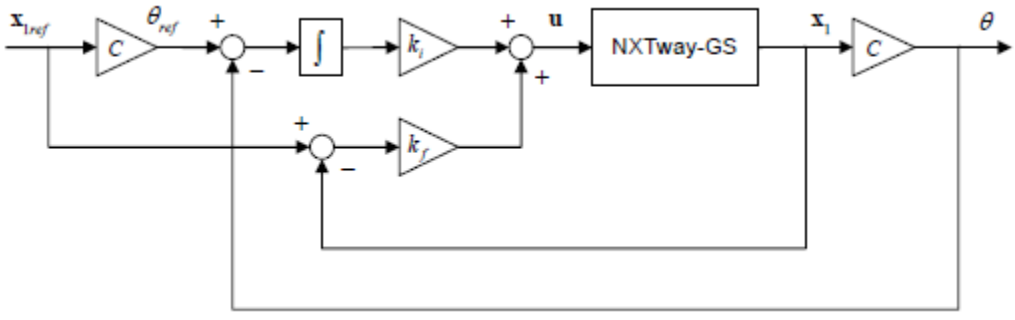
$$\boldsymbol{m}^0(N-j) = -\boldsymbol{K}(N-j)\boldsymbol{x}(N-j) \tag{82}$$

Са матрицама појачања стања **K**, у тренуцима k = 0,1,…N-1:

$$\mathbf{K}(0), \mathbf{K}(1), \dots, \mathbf{K}(N-1) \tag{83}$$

6.3. Пројектовање контролера самобалансирајућег робота

У овом случају се користи серво контролер чија је структура приказана на наредној слици. Као управљана величина користи се средња угаона позиција точкова θ. Коришћење неке друге промењиве величине није могуће јер систем не би био контолабилан [6].



Слика 6.3.1. *Серво контролер NXT робота*

Са слике се види да је k_f коефицијент појачања за повратну спрегу, док је k_i коефицијент интегралног појачања. Вредности ових појачања се добијају коришћењем линеарне квадратне контроле **(**Linear Quadratic Control – LQ**),** која је претходно описана.

Одређују се тежинске матрице Q и P.

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 6 \times 10^5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 4 \times 10^2 \end{bmatrix} \tag{84}$$

$$R = \begin{bmatrix} 1 \times 10^3 & 0 \\ 0 & 1 \times 10^3 \end{bmatrix} \tag{85}$$

Елемент Q матрице Q (2,2) је тежински фактор за угао нагиба, док је Q (5,5) тежински фактор интегралног појачања за грешку угла.

Коефицијенти појачања се могу добити помоћу Матлаб команде *lqr* па се добијају следеће вредности:

$$k_f = [-0.8 \quad -66.3799 \quad -1.2293 \quad -7.0540] \tag{86}$$

$$k_i = -0.4472 \tag{87}$$

У циљу регулисања осцилација робота потребно је правилно подесити коефицијент појачања k_f.

6.4. ПИ контролер

Даље се уводе додатне команде:

- Потребно је задавањем брзине левом и десном точку мотора обезбедити ротацију самобалансирајућем роботу;

- Користити П контролу за синхронизацију точкова када се робот креће праволинијски. Ово се уводи јер се мотори не крећу истим брзина чак и када се погоне истим PWM сигналом;

- Треба напоменути, да само П дејство није задовољавајуће јер робот не успева да се креће равномерно праволинијски као у оригиналном решењу, већ долази до појаве грешке и до његовог неправилног кретања. У том смислу је уведен и интегратор, који служи за елеминисање грешке, па се за контролу робота користи ПИ контролер [6].

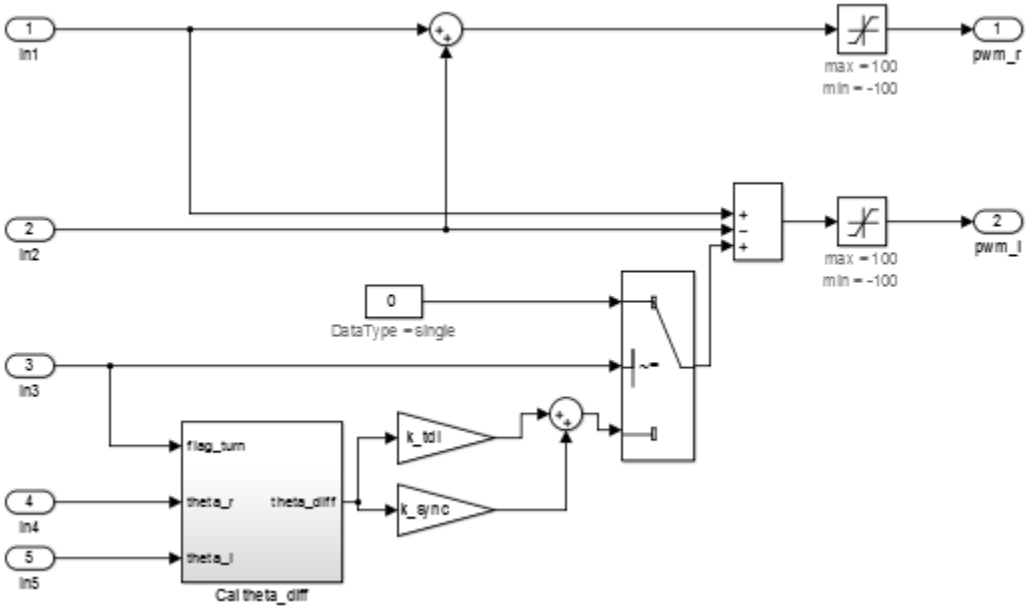
ПИ контролер одређује излаз из контролера у свакој периоди одабирања T и изражава се следећом релацијом:

$$u(t) = u + K(t) + \frac{K}{T_i}\int_0^t e(t)dt \tag{88}$$

У претходној релацији *u(t)* представља улаз у систем, *K* пропорционално појачање, *e* је грешка управљања, док је T_i интегрална константа.

Овај контролер садржи пропорционално и интегрално дејство, а главна предност у односу на П контролер је могућност уклањања грешке.

На наредној слици је приказана блок структура серво контролера NXT самобилансирајућег робота са означеним појачањима.



Слика 6.4.1 *Блок дијаграм контролера NXT самобилансирајућег робота*

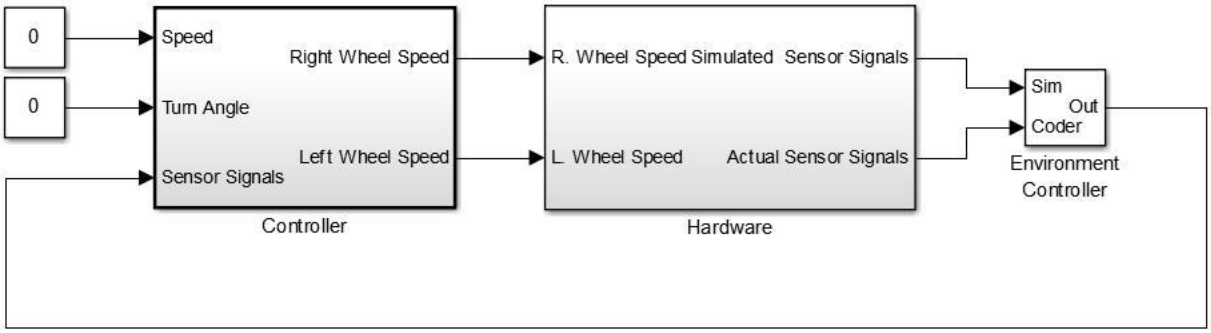
7. УПРАВЉАЊЕ LEGO NXT РОБОТА У MATLABU

Основни циљ овог рада је да се покаже подударност резултата имплементације контролера у Matlabu и Pythonu, тј. показати да је управљање LEGO NXT робота у ова два програма исто.

Даље ће бити приказана структура модела у Simulinku. Детаљно ће бити описани блокови LEGO робота који се односе на пројектовање контролера, симулацију кретања као и на стабилност, брзину и сл.

7.1. Симулинк модел

На наредној слици се налази структура модела за симулацију и контролу LEGO NXT самобалансирајућег робота у Simulinku у Matlabu. Приказани модел се састоји од основних компонената које се односе на контролер, хардвер и на окружење. Блокови који се односе на контролер и на хардвер састоје се од више подблокова који ће детаљно бити објашњени [10,11]



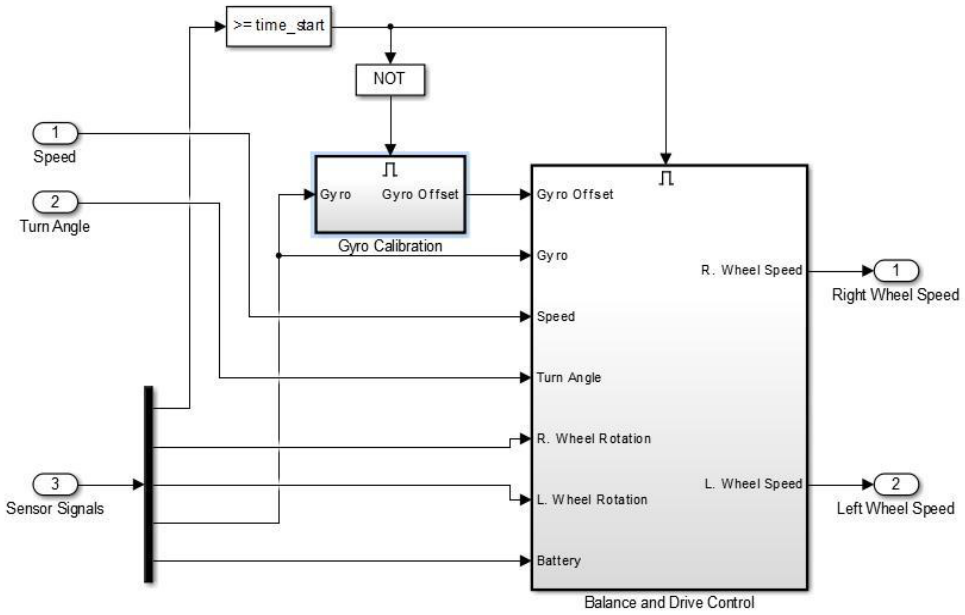
Слика 7.1.1.*Управљачка структура LEGO NXT робота у Симулинку*

На претходној слици се уочава да је први у низу блок који се односи на контролер. Исти као улазе сигнале прима брзину, угао заокретања и сигнале са сензора, док су излазни сигнали брзине левог и десног точка. Излази овог блока представљају улазе у наредни блок, који се односи на хардвер, који успоставља везу са роботом. Излази из овог блока су симулирани и стварни сигнали са сензора. Последњи у низу је блок који се односи на окружење, тј овај блок дефинише активне и неактивне блокове модела, у зависности од тога да ли је у питању симулација модела или се врши превођење у код и учитава на робота.

Даље ће бити детаљно објашњени представљени блокови и подблокови истих.

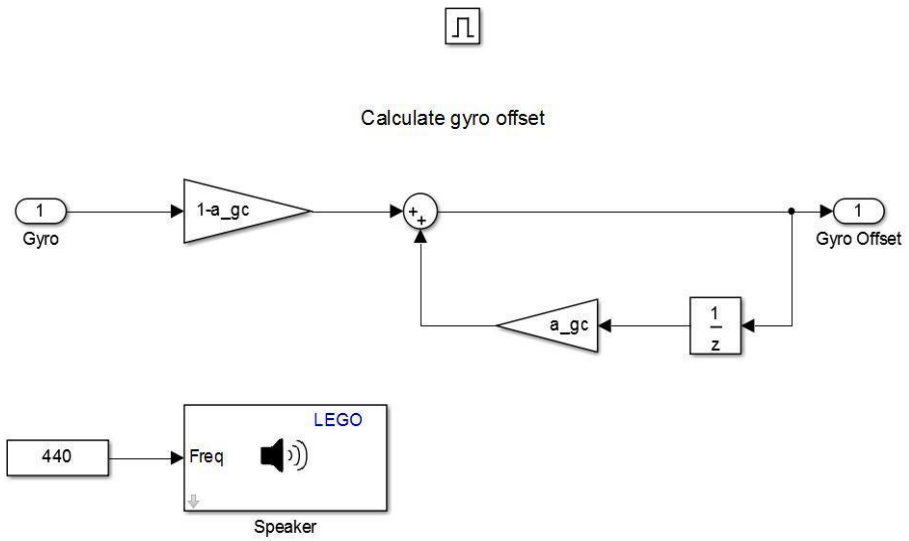
7.1.1. Контролер

На наредној слици је приказана унутрашњост *Controller* блока. Види се да су улази брзина точкова, угао заокретања и вредности које се узимају са сензора, као што је поменуто. Вредности са сензора се односе на вредност напона на батеријама, угао ротације левог и десног точка, као и на вредност која се учитава са жироскопа. Такође се уочава да се саставни подблокови овог блока односе на калибрацију жироскопа, балансирање и кретање робота сходно Yamamotovom решењу. Први подсистем који се односи на калибрацију жироскопа дефинише офсет истог, док се у другом подсистему налази алгоритам управљања и пропорционални конторлер који служи за регулисање окретања мотора. Већ поменути излази из блока *Controller* су угаоне брзине десног и левог точка (слика 7.1.1.1.) [10,11].



Слика 7.1.1.1. *Блок Controller са подсистемима за калибрацију жироскопа (Gyro Calibration) и балансирање и кретање робота (Balance and Drive Control)*

Даље ће бити објашњен први подисистем контролера робота, блок који се односи на калибрацију жироскопа (*Gyro Calibration*). Овај блок је значајан јер се врши уклањање шума са сигнала који се учитава са жироскопа. То омогућава дискретни филтер првог реда а поступак је шематски приказан на наредној слици.. Поред тога постоји и блок који служи за репродукцију звучног сигнала. Он служи да обавести корисника да је у току калибрација жироскопа. Калибрација жироскопа траје 4 секунде, при чему се одређује офсет.



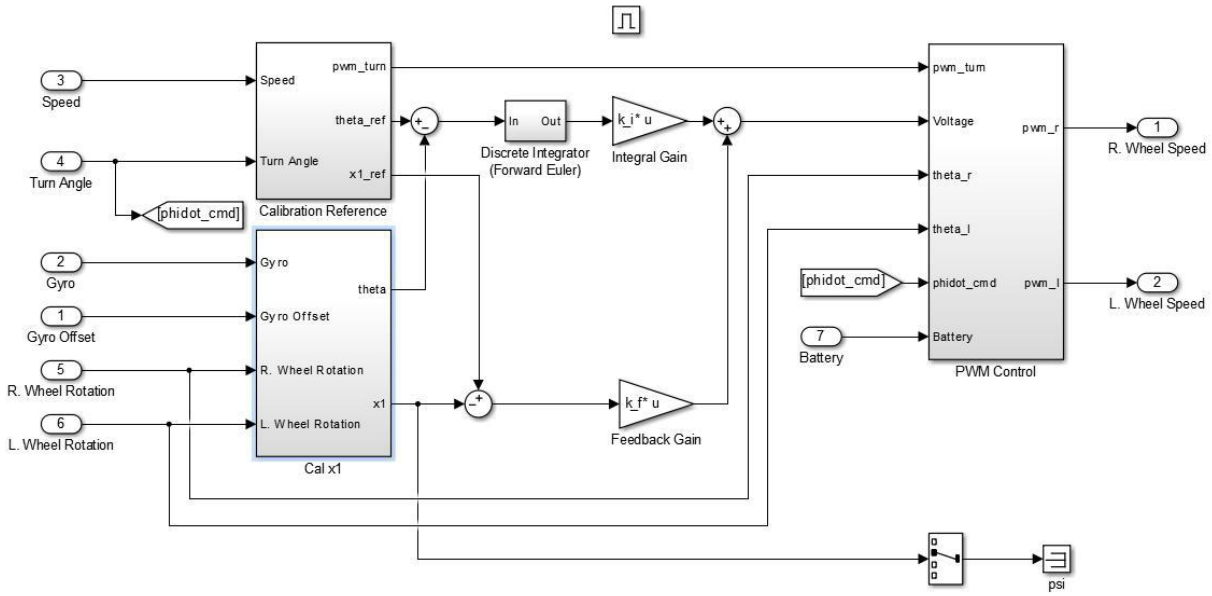
Слика 7.1.1.2. *Блок за калибрацију жироскопског сензора (Gyro Calibration)*

Након калибрације жироскопа врши се подешавање подсистема који се односи на балансирање робота и кретање истог (*Balance and Drive Control*). Овај подсистем као улазе прима вредност са жироскопа, офсет жироскопа, угао заокретања, и сигнале са сензора: ниво напона на батеријама, ротацију десног и левог точка. Излази из овог подсистема контролера су брзине левог и десног точка.

Блок који врши балансирање робота и кретање се састоји од више подсистема који се налазе на наредној слиоци. То су пре свега подсистеми *Calibration Reference, Cal x1* и *PWM Control.*

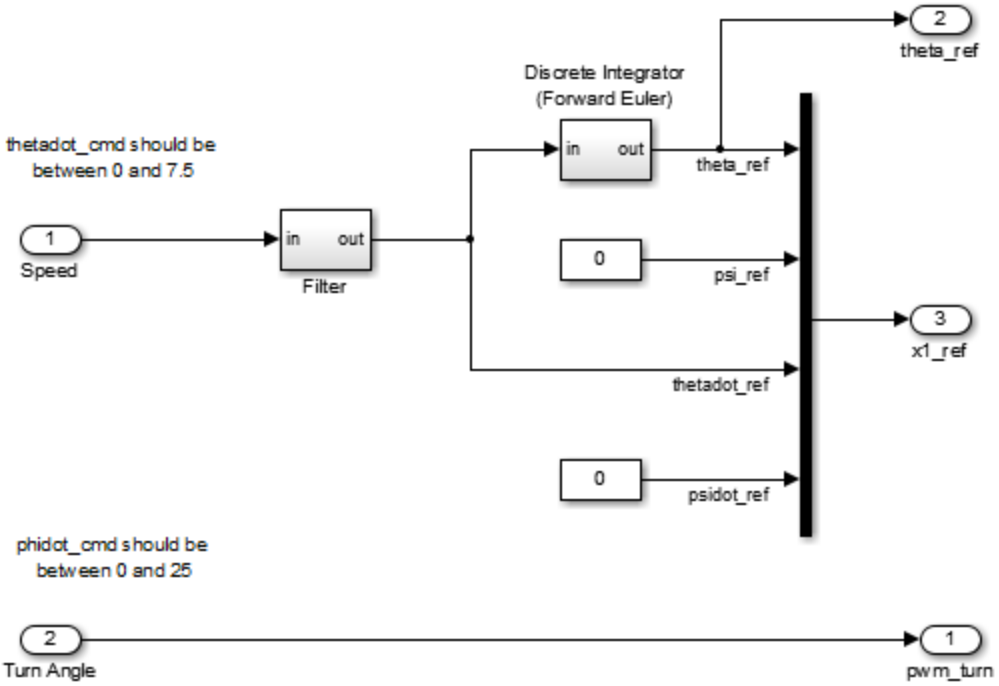
Први подсистем служи за одређивање референтних стања, дносно овај блок служи за контролу нагиба и заокретања. Подсистем *Cal x1* служи за пропрачунавање тренутних стања, док се у последњем блоку рачуна управљачки сигнал тј. шаљу се одговарајући PWM сигнали на улазе мотора. Ови подсистеми ће детаљније бити објашњени. Поред њих, налазе се и други блокови који су намењени за добијање жељеног управљања.

У складу са наведеним, закључује се да се у подсистему *Balance and Drive Control* одређују тренутна стања, референти угао, врши се дискретна интеграција грешке углова у точковима и на крају се дати PWM сигнал шаље мотору.



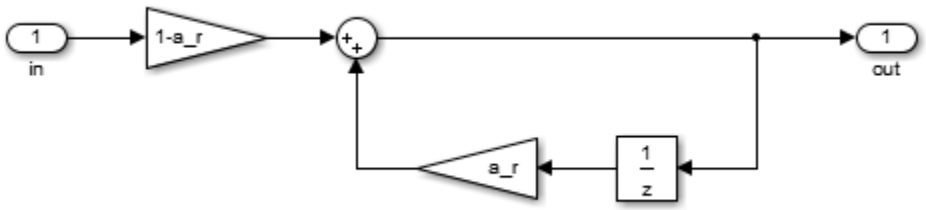
Слика 7.1.1.3. *Подсистем контролера Balance and Drive Control и подсистеми истог (Calibration Reference, Cal x1 i PWM Control)*

Затим ће бити описан подсистем *Calibration Reference* који се користи за калибрацију референтне брзине и угла заокретања. На наредној слици се уочава да се овај блок састоји од подблока дискретни интегратор, док се пре њега налази нископропусни филтар. Брзина и угао заокретања представљају улазе у систем, док су излази референтни угао заокретања по периоди, угао заокретања укупног робота као и низ од четири величине: угаона брзина точкова, угао окретања точкова, угаона брзина нагиба и угао нагиба робота.



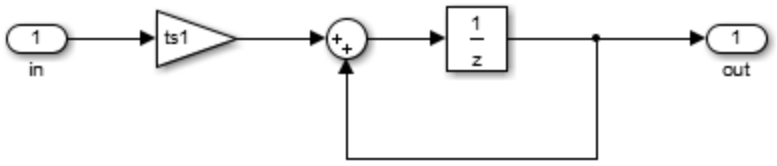
Слика 7.1.1.4. *Блок за калибрацију референте брзине и угла заокретања (Calibration Reference)*

Подсистем овог блока је већ поменути нископропусни филтар (слика 7.1.1.5.). Исти служи за елиминацију шума са контролера, који настаје уколико се брзина робота задаје екстерно преко контролера.



Слика 7.1.1.5. *Нископропусни филтар као подсистем блока за калибрацију*

Поред нископропусног филтра, један од подсистема блока за калибрацију је и дискретни интегратор, помоћу кога се интеграли брзина како би се добио угао окретања. Дискретни интегратор је приказан на слици 7.1.1.6.

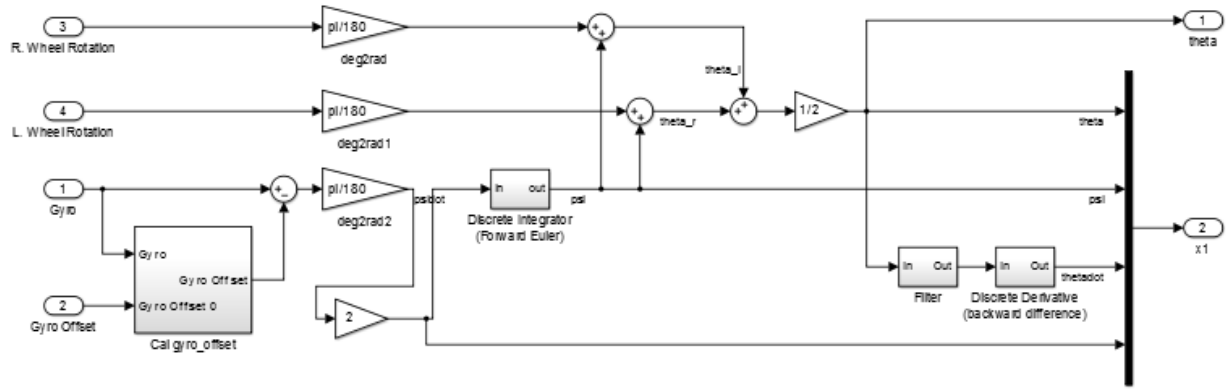


Слика 7.1.1.6. *Подсистем блока за калибрацију – дискретни интегратор*

Следећи подсистем блока *Balance and Drive Control* је подсистем задужен за одређивање тренутних вредности углова окретања и нагиба и брзина окретања и нагиба, који представљају излазе подсистема *Cal x1***.** Структура подсистема је приказана на слици 7.1.1.7.

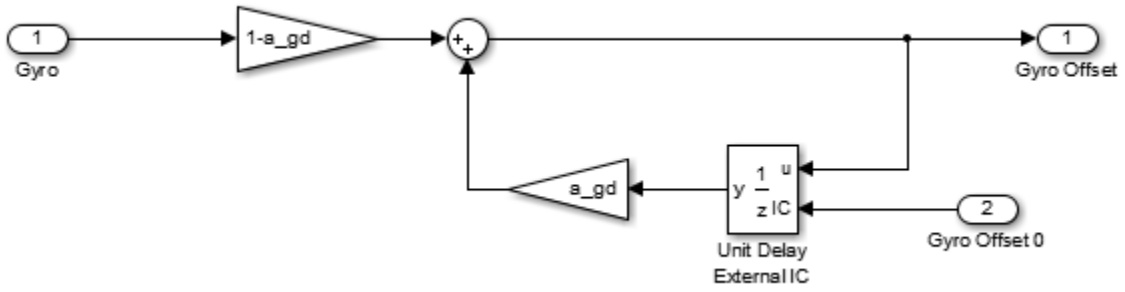
Улази овог подсистема су ротације левог и десног точка, вредности са жироскопа и офсет жироскопа. Овај систем такође садржи подсистеме који служе за превођење углова окретања точкова из степена у радијане. Такође је присутан и подсистем где се врши подешавања офсета.

Вредност са жироскопа је брзина нагињања робота *psidot*, па је ту величину потребно интегралити како би се добио угао нагињања. Добијена вредност се сабира са вредностима оба угла окретања и то представља стварни угао окретања. Ради добијања средње вредности ова вредност се дели са 2. Угао окретања се филтира а затим диференцира како би се добила брзина окретања на точковима. Излаз из блока је угао окретања, као и низ од четири величине као у претходном систему.



Слика 7.1.1.7. *Подсистем за одређивање тренутних вредности углова и брзина нагиба и окретања точкова Cal x1*

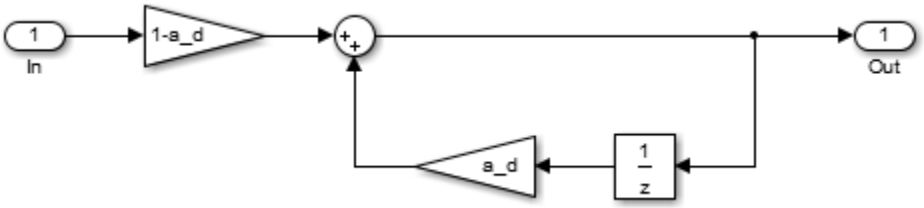
Наредни блок *Cal gyro_offset* представља подсистем претходног и служи да одржи оффсет жироскопа на почетно прорачунатом нивоу и да га у следећем периоду поново прорачуна. Овај блок садржи блок *Unit Delay* који обавља претходно описану функцију. Овај блок прво одржава офсет на првом пропрачунатом нивоу, а сваког наредног пута га прерачунава и задржава на претходном нивоу.



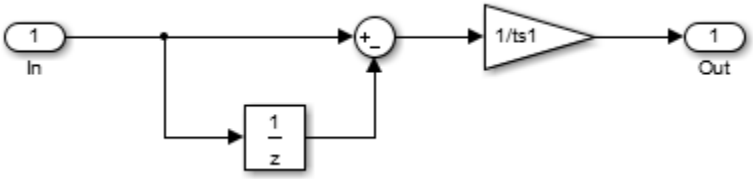
Слика 7.1.1.8. *Cal gyro_offset блок за одржавање офсета жироскопа на почетном нивоу*

Помоћу дискретног интегратора добија се угао нагиба, а структура интегратора је приказана на слици 7.1.1.5.

Да би се добила угаона брзина са енкодера који дају тренутне вредности угла, потребно је пре свега филтирати тај сигнал, а затим исти диференцирати. Ово омогућавају блокови за филтрирање и диференцирање који су приказани на наредним сликама.



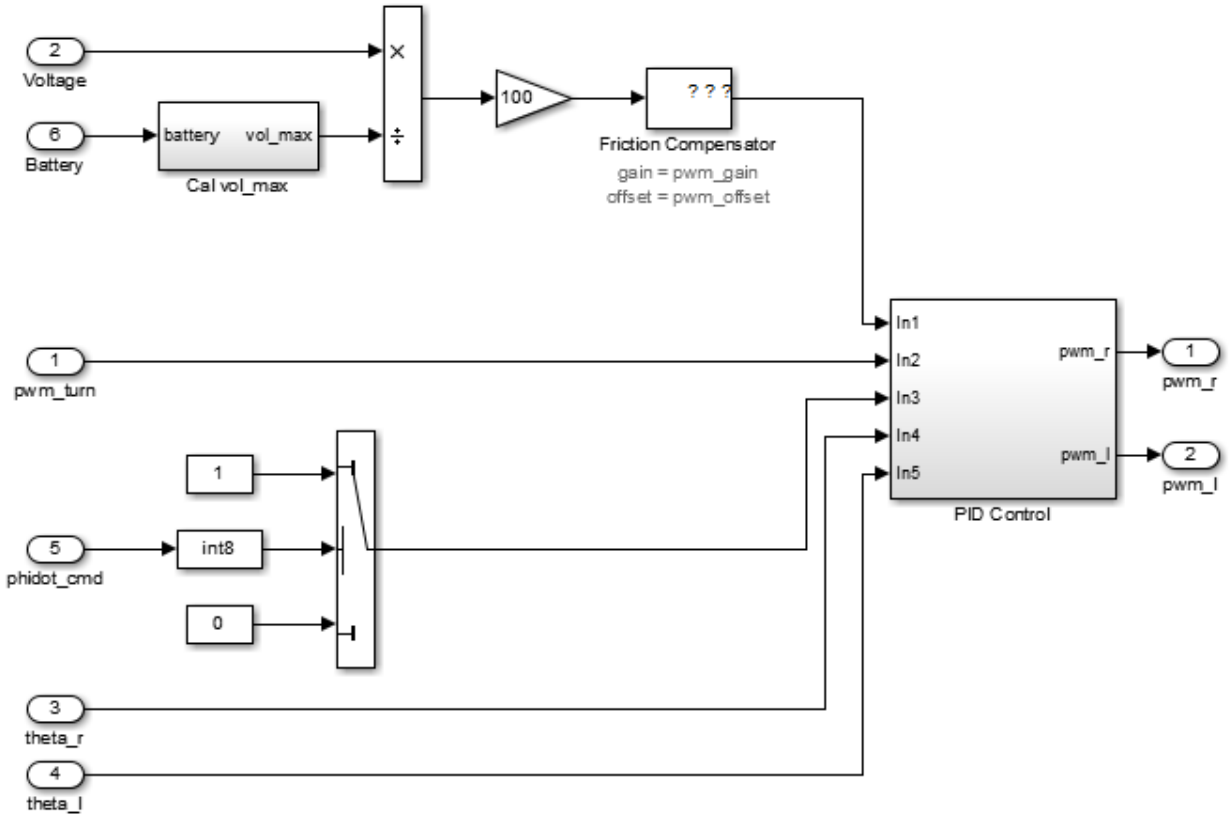
Слика 7.1.1.9. *Блок за филтрирање сигнала са енкодера (Filter)*



Слика 7.1.1.10. *Блок за диференцирање угла окретања (Discrete Derivative)*

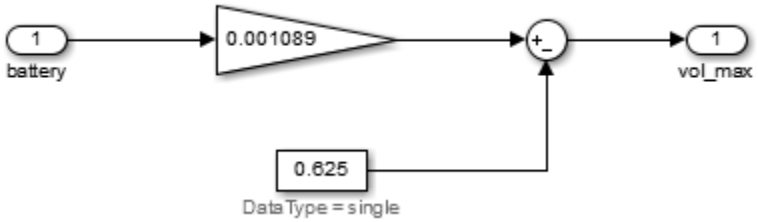
Наредни подсистем система *Balance and Drive Control* је *PWM Control*, где се врши генерисање PWM сигнала који се шаљу на моторе робота. Структура је прикана на слици 7.1.1.11. Саставни део овог подсистема је блок који се односи на ПИД контролу и блок који рачуна напон PWM сигнала на основу стања батерије *Cal vol_max*.

У првом делу блока види се да се рачуна PWM сигнал на основу стања батерије, а затим се исти шаље у наредни блок где се одлучује да ли се сигнал шаље на оба мотора. Пре тога се сигнал смешта у коло за компензовање трења електромотора.



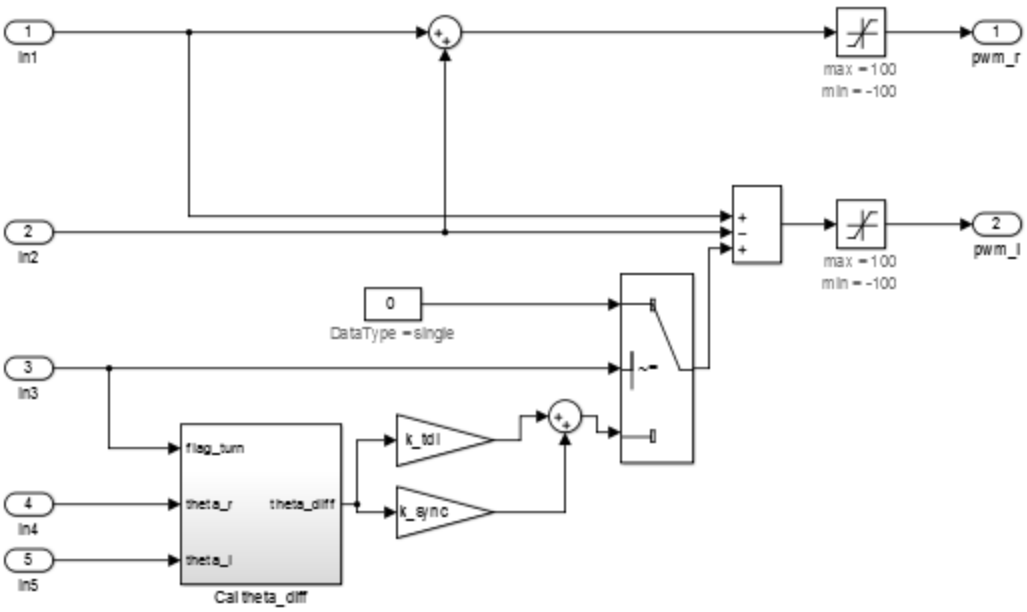
Слика 7.1.1.11. *Подсистем за израчунавање PWM сигнала (PWM Control)*

Структура подсистема *Cal vol_max*.је приказана на наредној слици.



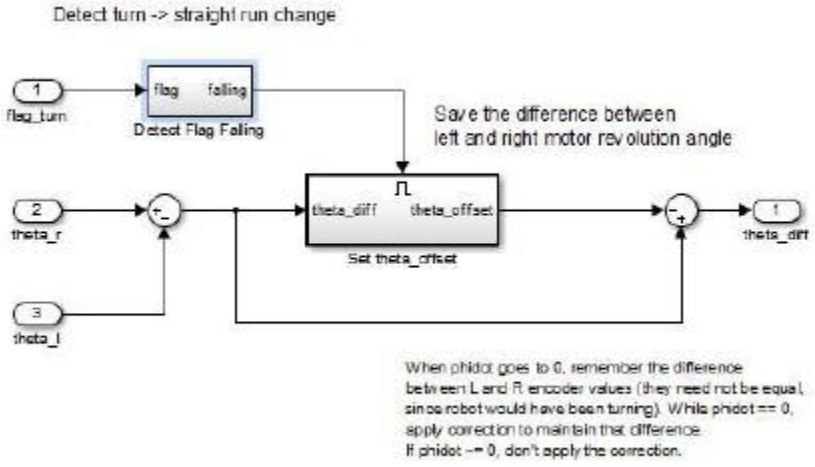
Слика 7.1.1.12. *Структура подсистема Cal vol_max.*

У блоку *PID Control* се врши ограничавање ПWM сигнала, као и подешавање појачања k_sync који служи за синхронизацију точкова**.** Овај подсистем је приказан на слици 7.1.1.13. и садржи блок који служи за регулисање заокретања *Cal theta_diff.*



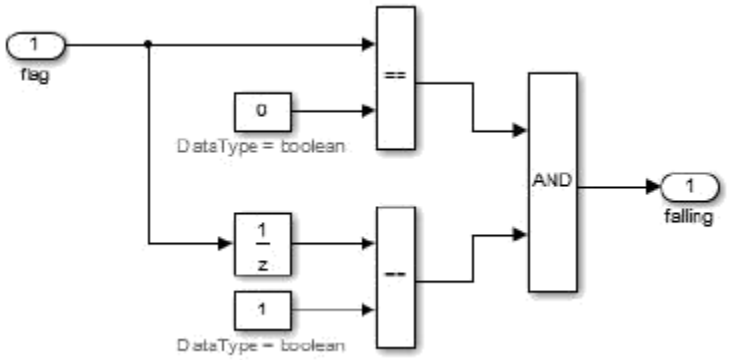
Слика 7.1.1.13. *Подсистем PID Control*

У наредном блоку се одређује да ли је жељено заокретање извршено. Уколико јесте, поставља се офсет угла који је потребно одржавати. Ако је жељено заокретање нула, на леви мотор се додаје сигнал управљања пропорционалан разлици заокретања мотора, након чега се генеришу сигнали управљања који се шаљу на моторе. У овом блоку се рачуна разлика између углова заокретања точкова. Уколико је угао једнак задатом углу токретанај тада се та вредност угла између точкова задржава.



Слика 7.1.1.14. *Блок за израчунавање разлике углова заокретања (Cal theta diff)*

На наредној слици је приказано коло које одређује да ли је потребно наставити окретање робота да би се достигао жељени угао или је потребно зауставити окретање у том тренутку како би се робот кретао право.



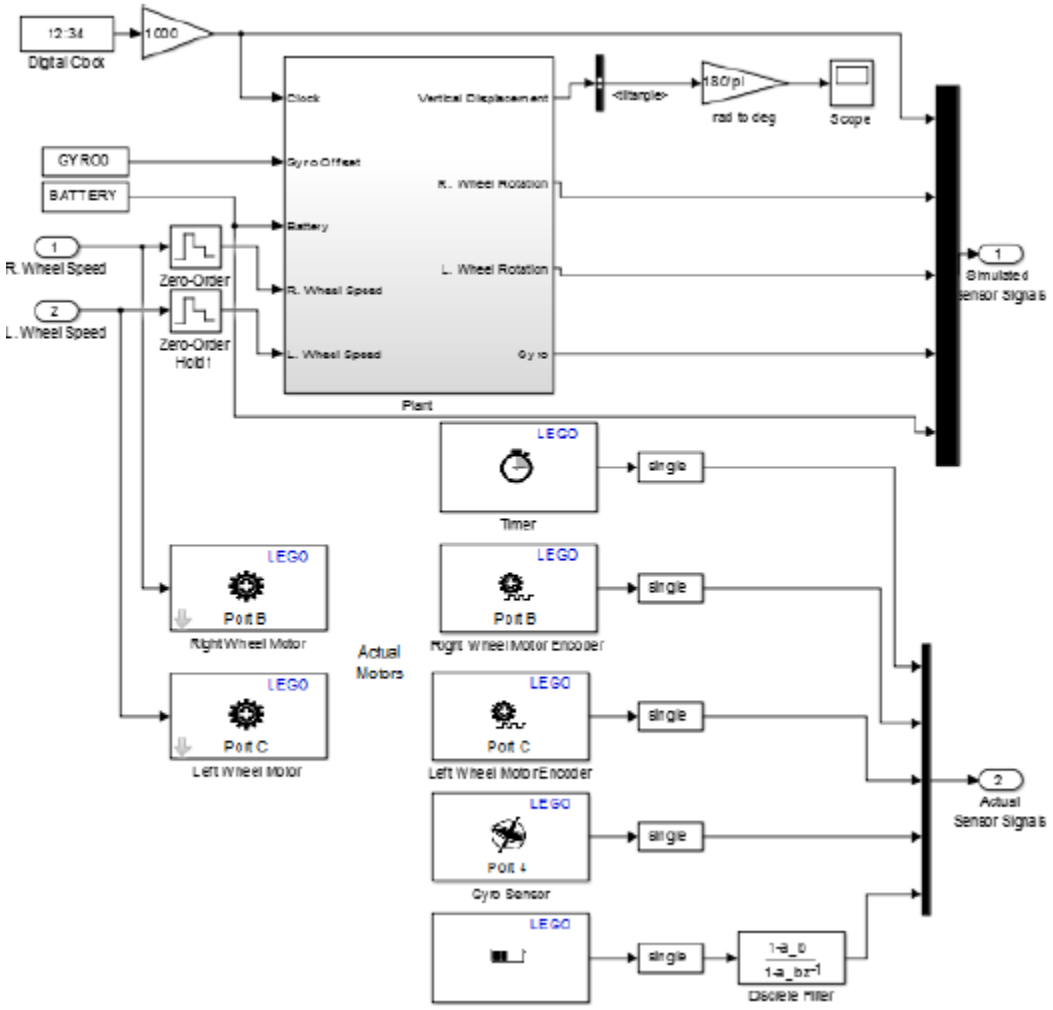
Слика 7.1.1.15. *Блок за одређивање услова за заокретање мотора (Detect Flag Faling)*

На слици 7.1.1.16. приказан је блок који служи за одређивање офсета за угао theta.

Слика 7.1.1.16. *Блок за одређивање офсета угла theta*

7.1.2. Хардвер

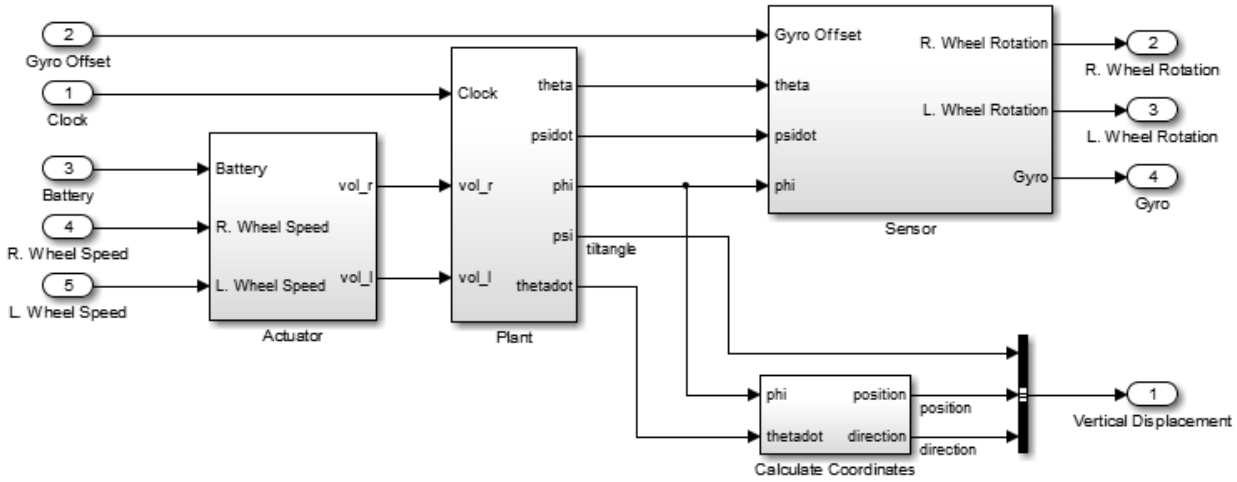
Поред контролера, као саставног блока симулинка, описан је и хардвер. Овај подсистем који је приказан на слици 7.1.2.1. састоји се од блока *Plant* који садржи пројектован модел робота у простору стања. Такође садржи и улазе и излазе NXT пакета који обезбеђују везу са сензорима и актуаторима робота, а који се налазе у доњем делу [10,11].



Слика 7.1.2.1. *Блок Hardver*

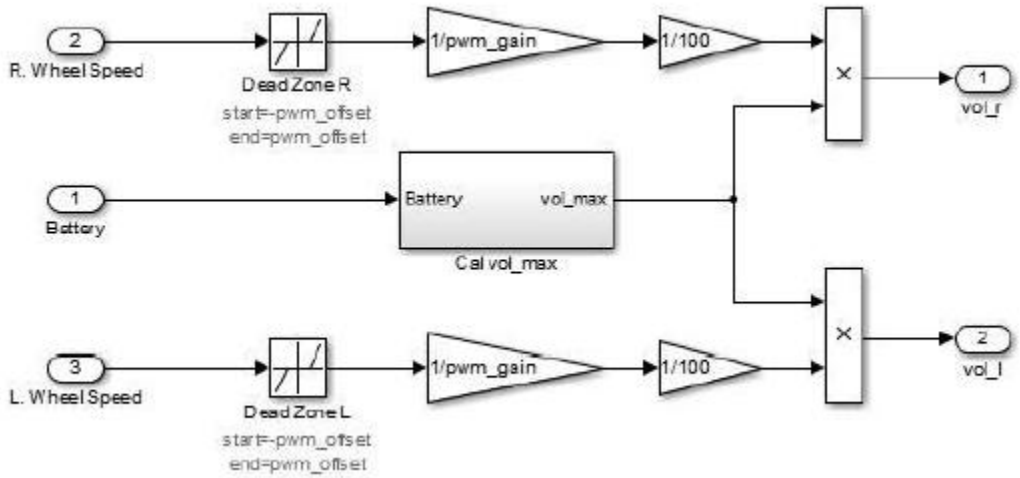
Споменути подсистем блока *Hardver*, блок *Plant* је приказан на наредној слици и исти се састоји од неколико блокова. Овај подсистем представља математички модел Лего NXT робота. Улази у блок су офсет жироскопа, сат, батерија и брзине левог и десног точка. Излази из блока су углови окретања левог и десног точкова, вредност са жироскопаа и вертикално померање.

У склопу овог блока налазе се подсистеми који се односе на актуаторе Лего робота *(Actuator)*, *сензоре (Sensor)*, израчунавање координата *(Calcute Coordinates)* и блок *Plant*. Ови подсистеми ће бити детаљно објашњени.



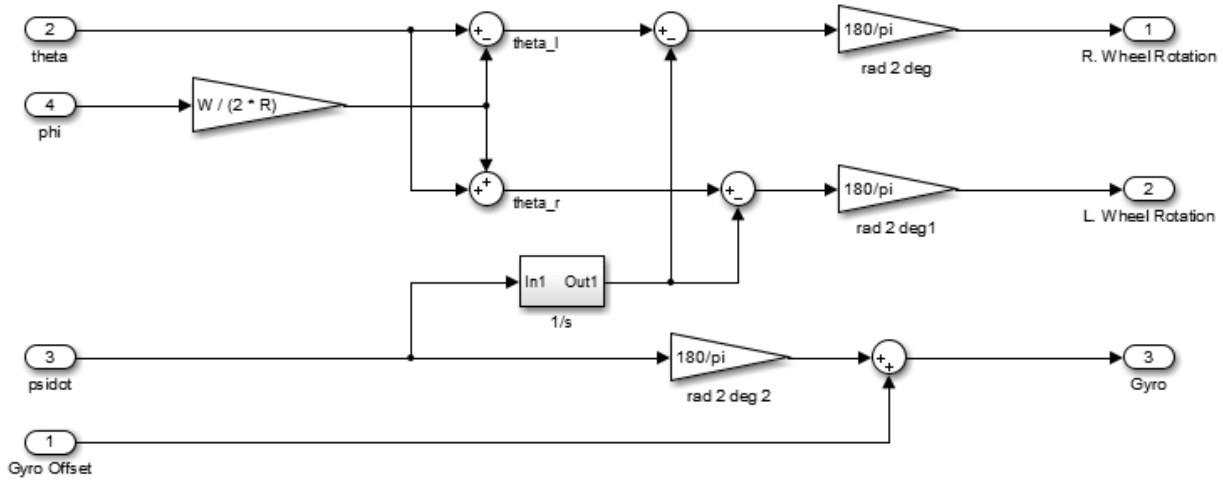
Слика 7.1.2.2. *Блок Plant*

На наредној слици је приказан блок који се односи на актуаторе LEGO робота. У овом блоку се рачуна напон на основу PWM сигнала са контролера. Узимајући у обзир Кулоново и вискозно трење, потребно је моделирати мртву зону пре пропрачуна напона. Улази у систем су ниво батерије и брзине левог и десног точка, док су излази напони на точковима. У склопу овог система налази се и подсистем *Cal vol_max* који одређује зависност максималног напона мотора и напона на батеријама, који је приказан на слици 7.1.1.12.



Слика 7.1.2.3. *Блок за симулацију мотора (Actuator)*

Поред блока који се односи на извршне органе LEGO робота, подсистем који служи за подешавање сензора приказан је на слици. 7.1.2.4. Улази у овај блок су угао окретања, угао хоризонталног заокретања, брзина нагиба и офсет са жироскопа. Излази из подсистема су ротације десног и левог точка и вредност са жироскопа. Овај подсистем служи за одређивање сигнала који долазе са жироскопа и енкодера. Подсистем садржи блокове који преводе вредности углова из радијана у степене.

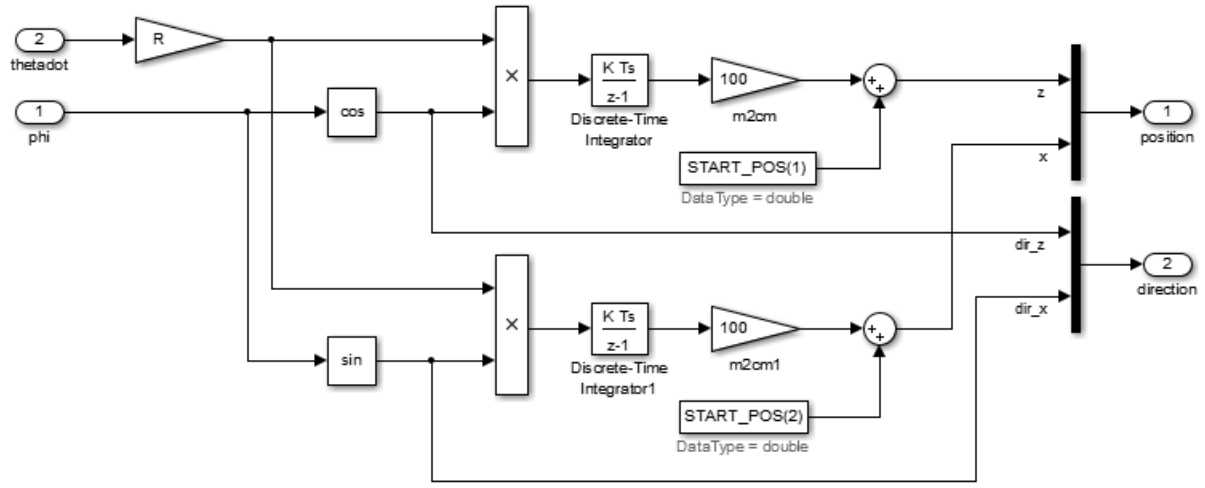


Слика 7.1.2.4. *Блок за подешавање сензора (Sensor)*

У оквиру претходно описаног система, налази се и блок који представља дисретни интегратор (слика 7.1.2.5.), који служи да на основу брзине нагиба *psidot* израчуна угао нагиба *psi*.

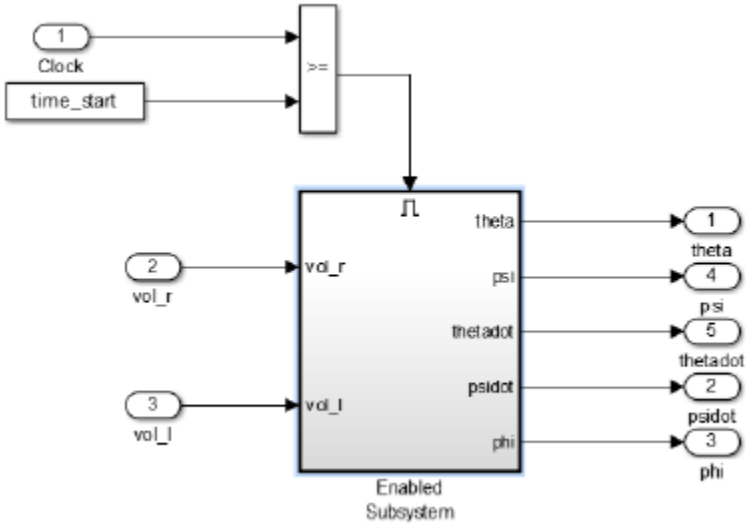
Слика 7.1.2.5. *Дискретни интегратор за израчунавање угла нагиба*

Блок *Calculate Coordinates* рачуна позицију и правац робота на основу унетих углова. Овај блок се користи за визуелно исцртавање кретања робота и није предмет овог рада.



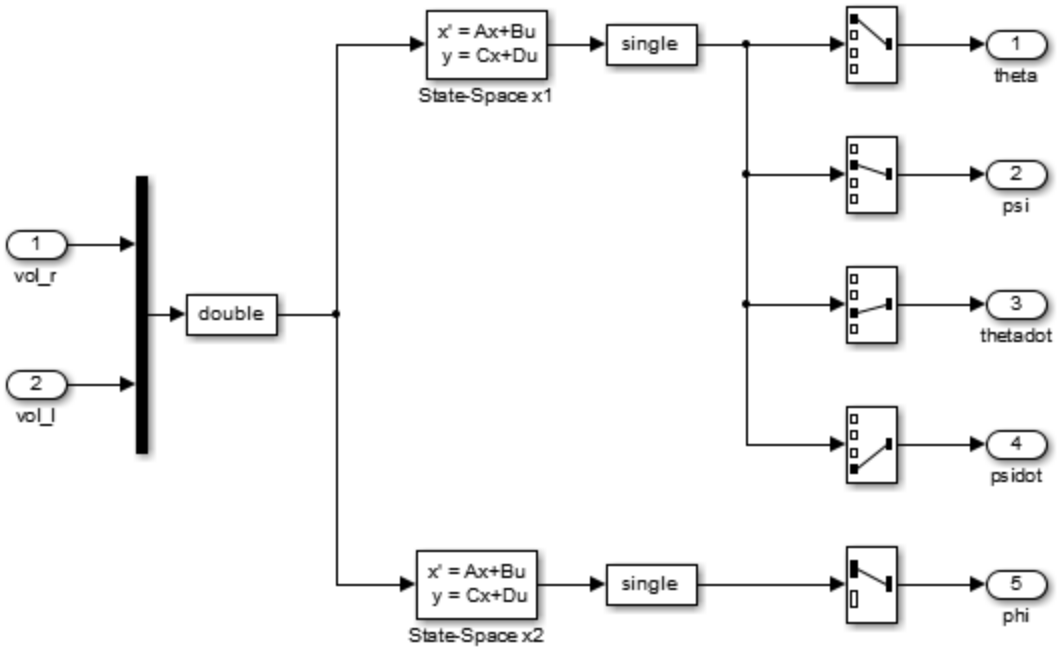
Слика 7.1.2.6. *Блок за израчунавање позиције и правца (Calculate Coordinates)*

Последњи у низу подсистема је подсистем *Plant* налази са блоком *Enabled Subsystem* чије је извршавање условљено временом калибрације жироскопа. Уколико је тренутно време мање од времена подешавања жироскопа наредни блок се неће извршавати.



Слика 7.1.2.7. *Подсистем Plant*

У наредном блоку се врши израчунавање тренутних вредности заокретања и брзине заокретања рачунају помоћу једначина стања система. Пошто имамо два система једначина, помоћу једног се рачунају нагиб и окретање точкова, док се други користи за рачунање хоризонталног заокретања.



Слика 7.1.2.8. *Подсистем Enabled Subsystem*

8. КРЕИРАЊЕ PYTHON КОДА

Креирање Python кода је урађено на основу претходно описаним блокова у Simulinku. Python код прати оригинално Yamamotovo решење у циљу приказивања идентичних резулатата.

Програмирање LEGO робота помоћу програмског језика Python је могуће извршити на два начина:

- Могуће је директно на рачунару извршити код написан у Python-у слањем команди роботу помоћу USB кабла или Bluetooth-a. Међутим наведени приступ може бити ограничен дужином USB кабла или дометом Bluetooth-a.

- Поред описаног начина, могуће је Python код превести у nxc код и исти пребацити на коцку, чиме робот постаје независтан од рачунара.

У овом раду написан је Python код који је помоћу BricxCC конвертора преведен у nxc код, који је затим пребачен на коцку. Nxc значи Not eXactly C, што значи да робот није у могућности да чита C код директно. Помоћу nxc програма могуће је директно програмирати робота без даљег конвертовања. Потребно је компајлирати код и прабацити на коцку, што ће бити објашњено у даљем раду.

Циљ рада се односи на праволинијско и кружно кретање LEGO NXT робота. Python код за оба кретања је веома сличан, и исти ће бити приказан са истакнутим разликама које се односе на кретања.

У овом поглављу ће детаљно бити описана Python скрипта у циљу приказивања идентичности исте са Simulink блоковима. Такође ће бити описан поступак компајлирања nxc фајла и пуштања робота у рад.

8.1. Праволинијско кретање LEGO NXT робота

У раду је приказано праволинијско кретање робота. Дефинисане су променљиве и функције које су за исто карактеристичне.

8.1.1. Дефинисање променљивих

Почетне линије Python скрипте се односе на променљиве. Потребно је у складу са Yamamotovim решењем поставити вредности променљивих, као и вредности константи.

На наредним сликама приказане су коришћене променљиве. Вредности које нису описане, биће објашњење у даљем раду јер углавном представљају збир више променљивих па је њихово разумевање прегледније у даљем коду.

```
#definisanje promenljivih
gyroOffset = 0                  #ofset ziroskopa
gyroOffset1=0
a_gc = 0.8                      #konstanta filtera
a_r = 0.9960                    #konstanta filtera
a_gd = 0.999                    #faktor zaboravljanja
a_d = 0.8                       #konstanta filtera
calRefFilOut = 0
ts1 = 0.004                     #perioda odabiranja
theta_ref = 0                   #referentni ugao okretanja
thetadot_ref = 0                #referentna ugaona brzina
thetadot_cmd = 5                #ugaona brzina koja se zadaje
phidot_cmd =0                   #ugaona brzina horizontalnog okretanja
psi = 0                         #ugao nagiba
psi_ref = 0                     #referentan ugao nagiba
psidot_ref = 0                  #referentna ugaona brzina nagiba
pwm_turn = 0                    #pwm signal okretanja
gyro = 0                        #vrednost sa ziroskopa
psidot = 0                      #ugaona brzina nagiba
rWheelRotation =0               #ugao okretanja desnog tocka
lWheelRotation = 0              #ugao okretanja levog tocka
 theta_l = 0                     #ugao okretanja levog tocka
 theta_r = 0                     #ugao okretanja desnog tocka
 theta = 0                       #srednja vrednost zaokretanja tocka
 thFilt = 0                      #filt. srednja vrednost zaokretanja tocka
 thetadot = 0                    #ugaoona brzina okretanja
 theta_error = 0                 #greska ugla
 theta_errorIntegral = 0         #integraljena vrednost greske ugla
 theta_offset = 0;               #vrednost ofseta
 theta_diff = 0                  #dif. vredonost ugla okretanja
 thetaIntegral = 0               #integralna vredonost ugla okretanja
```

Слика 8.1.1.1. *Промењиве програма*

```
k_i = -0.4472                       #integralno pojacanje povratne sprege
k_fTheta = -0.8                     #pojacanje ugla okretanja
k_fPsi = -66.3799                   #pojacanje ugla nagiba
k_fThetadot = -1.2293               #pojacanje brzine okretanja
k_fPsidot = -7.0540                 #pojacanje brzine nagiba
thFiltOld = 0                       #filtrirana vrednost ugla okretanja
voltage = 0                         #definisanje napona
vol_max = 0                         #definisanje maksimalnognapona
voltageBatteryCompensated = 0       #napon na bateriji
voltageFrictionCompensated = 0      #viskozno trenje
pwm_gain = 1                        #pojacanje pwm signala
pwm_offset = 0                      #ofset pwm signala
phiExists = 0
phiExistsDelay = 0
psidotTerm = 0
thetaTerm = 0
thetadotTerm = 0
pwm_r = 0                           #pwm signal za desni tocak
pwm_l = 0                           #pwm signal za levi tocak
psiTerm = 0
falling = False
k_sync = 0.5                        #pojacanje za sinhroniciju tockova
pastTime = 0
startTime = 0
thetaSync = 0                       #ukupni ugao zaokretanja
thetadifInt =0
k_tdI = 1.5                         #integralno pojacanje
```

Слика 8.1.1.2. *Променљиве програма*

Након дефинисања променљивих следи извршавање програма по функцијама, а сходно оригиналном решењу. Називи функција одговарају у потпуности Simulinku. Истеће бити приказани редоседом који одговара Simulinku.

8.1.2. Дефинисање функција

Први задатак који се извршава у Simulinku је калибрација сензора представљена на слици 7.1.1.2. Калибрацији жироскопа одговара Python функција приказана на слици 8.1.2.1. Друга линија кода значи да се емитује тон фреквенције 440 Hz, у трајању од 10 милисекунди.

```
def GyroCalibration():
        gyroOffset = SensorHTGyro(GYRO_PORT) * (1 - a_gc) + gyroOffset * a_gc
        PlayTone(440, 10)
```

Слика 8.1.2.1. *Функција која се односи на калибрацију жироскопског сензора*

Наредни блок у Simulinku је блок за прорачунавање брзине на точковима *Balance and Drive Control*. Формулација истог у Pythonu је дата на следећој слици. Види се да

систем за баланисирање и кретање зависи од рада наведених таскова, који одговарају слици 7.1.1.3. Такође је приказано како се добија променљива theta_error, која представља разлику референтног угла окретања и укупног угла окретања.

```
def BalanceAndDriveControl():
        CalibrationReference()
        CalX1()
        theta_error = theta_ref - theta
        CalculateControlSignal()
        PWMControl()
```

Слика 8.1.2.2. *Функција која се односи на калибрацију жироскопског сензора*

Даље се дефинише функција *CalculateControlSignal*. Ова функција се не дефинише као посебна у Simulinku, а овде је одређена ради боље прегледности. Функција се налази у склопу Balance and Drive Control блока и служи за одређивање дејства променљивих, где се на основу Simulink блока углови и угаоне брзине множе са одговарајућим појачањима.

```
def CalculateControlSignal():
        DiscreteIntegrator(theta_errorIntegral, theta_error, ts1)
        thetaIntegral   = theta_errorIntegral * k_i
        psiTerm = (psi_ref - psi) * k_fPsi
        psidotTerm  = (psidot_ref - psidot) * k_fPsidot
        thetaTerm   = (theta_ref - theta) * k_fTheta
        thetadotTerm = (thetadot_ref - thetadot) * k_fThetadot
        voltage = thetaIntegral + psiTerm + psidotTerm + thetaTerm + thetadotTerm
        voltage = voltage
```

Слика 8.1.2.3. *Функција која се односи на одређивање дејства променљивих*

Наредни блок у Simulinku је *Calibration Reference* и приказан је на слици 7.1.1.4. Python код који прати тај блок је дат на наредној слици. Последња линија кода се односи на доњи део Simulink блока, где се види да PWM сигнал заокретања узима вредност брзине хоризонталног заокретања, који је у Pythonu ознаћен са *phidot_cmd.* У склопу овог блока налазе се блокови који се односе на филтар и на дискретни интегратор. Дискретни интегратор приказан на слици 7.1.1.6. је представљен у трећој линији кода.

```
def CalibrationReference():
        CalRefFilter()
        thetadot_ref = calRefFilOut
        DiscreteIntegrator(theta_ref,thetadot_ref,ts1)
        pwm_turn = phidot_cmd
```

Слика 8.1.2.4. *Функција Calibration Reference*

У Python програмском језику филтру са слике 7.1.1.5., одговарају линије кода приказане на следећој слици. Улаз у филтар је угаона брзина која је у Pythonu означена са *thatadot_cmd.* На овај начин је и дефинисана и променљива *calRefFilOut.*

```
def CalRefFilter():
  calRefFilOut = thetadot_cmd * (1 - a_r) + calRefFilOut * a_r
```

Слика 8.1.2.5. *Функција која се односи на програмирање филтра CalRefFilter*

После блока *Calibration Reference* на слици 7.1.1.7 приказан је други подсистем блока *Balance and Drive Control, Cal x1*, који служи за одређивање тренутних вредности углова и брзина нагиба и окретања. Python код за овај подсистем је приказан на слици 8.1.2.6. У склопу ове функције налазе се и функције који се односе на *CalGyro_offset, CX1Filter i DiscreteIntegrator*, што одговара Simulinku. У другој линији је израчунат угаона брзина нагиба *psidot*. У четвртој и петој линији кода дат је поступак рачунања левог и десног угла окретања, док се седма линија односи на стварни угао окретања.

```
def CalX1():
  CalGyro_offset()
  psidot = DEG2RAD * (gyro - gyroOffset1)*2
  DiscreteIntegrator(psi, psidot, ts1)
  theta_l = psi + lWheelRotation * DEG2RAD
  theta_r = psi + rWheelRotation * DEG2RAD
  theta   = (theta_l + theta_r) / 2
  thFiltOld = thFilt;
  CX1Filter()
  thetadot = (thFilt - thFiltOld) / ts1
```

Слика 8.1.2.6. *Функција која се односи на podsistem Cal x1*

Функција *CalGyro_offset* која је Simulink блоком приказана на слици 7.1.1.8 дата је на наредној слици. Добија се вредност промењиве *gyroOffset1*.

```
def CalGyro_offset():
    gyroOffset1 = gyro * (1 - a_gd) + gyroOffset1 * a_gd
```

Слика 8.1.2.7. *Функција CalGyro_offset*

Блок за филтрирање угла окретања дат на слици 7.1.1.9, а у Pythonu се дефинише на следећи начин. Добија се вредност угла окретања *theta* након филтрирања – *thFilt*.

```
def CX1Filter():
        thFilt = theta * (1 - a_d) + thFilt * a_d
```

Слика 8.1.2.7. *Функција CX1Filter*

Угаона брзина окретања *thetadot* је дата у последњој линији кода са слике 8.1.2.6, и објашњава блок који се налази на слици 7.1.1.10. Промењљива *thFiltOld* представља угао окретања у претходном временском тренутку.

Наредни подсистем *PWM Control*, служи за израчунавање PWM сигнала и приказан је на слици 7.1.1.11. У Python програмском језику приказан је на слици 8.1.2.8. Прва линија кода се односи на израчунавање напона на батеријима, што је у Simulinku приказано на слици 7.1.1.12. Друга линија кода се односи на горњи део блока, где се добија вредност промењиве *voltageBatteryCompensated.* Трећа линија кода дефинише улаз ln1 у блок који се односи на ПИД контролер.

Након тога се налази if петља која се односи на доњи део *блока PWM Control*, тачније на дефинисање брзине хоризоналног окретања *phidot_cmd*. Уколико је вредност брзине *phidot_cmd* различита од нуле, онда вредност промењиве постаје 1, док је супротном случају 0.

```
def PWMControl():
        vol_max = (BatteryLevel()) * 0.001089 - 0.625
        voltageBatteryCompensated = 100 * voltage / vol_max
        voltageFrictionCompensated = sign(voltageBatteryCompensated) * (pwm_gain * abs(voltageBatteryCompensated) + pwm_offset)
        phiExistsDelay = phiExists
        if (phidot_cmd != 0):
                phiExists = 1
        else:
                phiExists = 0

        PIDControl()
```

Слика 8.1.2.8. *Функција PWMControl*

Функција *PIDControl* приказана на слици 7.1.1.13. се дефинише на начин који је приказан на слици 8.1.2.9. У првој линији кода се дефинише промењива *pwm_r*, која представља PWM сигнал десног точка. Иста представља суму промењивих означених са улазима ln1 и ln2 у Simulinku (*voltageFrictionCompensated* и *pwm_turn*). Затим се помоћу if петље ограничана PWM сигнал десног точка тј. промењива *pwm_r*, на вредности од -100 до 100.

Такође се дефинише и ограничава и PWM сигнал за леви точак тј. промењива *pwm_l*, што је приказано у последњим линијама кода. Дефинише се дискретни интегратор и if петља. Уколико је вредност промењиве *phiExists* различита од нуле следи да је тхетаСунц 0, док је супротном случају вредност промењиве *thetaSyn*c дата линијом кода након команде else.

```
def PIDControl():
        pwm_r = voltageFrictionCompensated + pwm_turn
        if (pwm_r > 100):
             pwm_r = 100
        if (pwm_r < -100):
             pwm_r = -100
        CalThetaDiff()
        DiscreteIntegrator(thetadifInt, theta_diff, ts1)
        if (phiExists != 0):
            thetaSync = 0
        else:
            thetaSync = theta_diff * k_sync + thetadifInt * k_tdI
        pwm_l = voltageFrictionCompensated - pwm_turn + thetaSync
        if (pwm_r > 100):
             pwm_r = 100

        if (pwm_r < -100):
             pwm_r = -100
```

Слика 8.1.2.9. *Функција PIDControl*

У склопу ове функције дата је функција *CalThetaDiff* која је приказана на слици 7.1.1.14. У Pythonu је ова функција приказана на слици 8.1.2.10. Дефинише се променљива *theta_diff* у другој линији кода.

Поред тога направљена је if петља са променљивом *falling*. То је boolean променљива и може имати вредност тачно (true) или нетачно (false). У овом случају уколико је вредност променљиве true, добијају се вредности променљивих *theta_offset* и *theta_diff*.

```
def CalThetaDiff():
        DetectFlagFalling()
        theta_diff = theta_r - theta_l
        if (falling):
           theta_offset = theta_diff

           theta_diff = theta_diff - theta_offset
```

Слика 8.1.2.10. *Функција CalThetaDiff*

Функција *DetectFlagFalling*, приказана на слици 7.1.1.15., у Pythonu је дата наредбама приказаним на следећој слици. Уколико је вредност променљиве *phiExists* једнака 0 и *phiExistsDelay* једнака 1, онда је вредност променљиве *falling* тачна.

```
def DetectFlagFalling():
        falling = (phiExists == 0) & (phiExistsDelay == 1)
```

Слика 8.1.2.11. *Функција DetectFlagFalling*

Добијени сигнали се прослеђују моторима на излазима A и C и формира се функција *Controller*. Командом OnFwd мотори се крећу напред.

```
def Controller():

    gyro = SensorHTGyro(GYRO_PORT)
    BalanceAndDriveControl()
    rWheelRotation = MotorRotationCount(OUT_A)
    lWheelRotation = MotorRotationCount(OUT_C)
    OnFwd(OUT_A, pwm_r)
    OnFwd(OUT_C, pwm_l)
```

Слика 8.1.2.12. *Функција Controller*

8.1.3. Task main

Task main је функција која се прва покреће и која позива друге функције. Дефинише се улаз жироскопа. Затим следи бесконачна for петља, где се 250 пута ради калибрација жироскопа, а између сваке калибрације се чека 20 милисекунди.

Након тога променљива *gyroOffset1* добија вредност *gyroOffset*, који садржи информацију о углу нагиба робота. Променљива *startTime* бележи време када је програм почео да ради. Променљива *pastTime* добија вредност променљиве *startTime*.

Последња у низу петља while је бесконачна и се односи на то да ли је након извршавања задатака контролера прошло 4 милисекунде. Уколико није провера се све док не прође 4 милисекунде, где се тренутно време ресетује и све креће од почетка.

While True петља у средини служи за заустављање робота након одређеног времена. Уводи се променљива *repeated* која има вредност 0. Формира се нова if петља. Сваки пут када се изврши претходно дефинисана whilee петља, где време извршавања траје 4 милисекунде, променљива *repeated* се увећа за 1. Када променљива *repeated* достигне 5000, што је у ствари 20 секунди, што је време за које робот иде право, укључује се опција else која за последицу има смањење брзине до самог заустављања.

```
def task_main():
        SetSensorHTGyro(GYRO_PORT)

for i in range(0, 250):
    GyroCalibration()
    Wait(20)


    gyroOffset1 = gyroOffset
    startTime = CurrentTick()
    pastTime = startTime
    repeated = 0
    while True:
     if (repeated < 5000):
         repeated = repeted+1
     else:
      thetadot_cmd = thetadot_cmd * 0.9
          Controller()
     while (CurrentTick() - pastTime < 4):
           continue

     pastTime = CurrentTick()
```

Слика 8.1.3.1. *Task main*

8.2. Кретање LEGO NXT робота по кружници

Поред праволинијског кретања LEGO NXT робота, помоћу програмског језика Python омогућено је и кретање истог по кружници. Исте промењиве величине се користе и код кретања робота по кружници, уз мале измене приказане на слици 8.2.1.

На слици се уочава да је вредност брзине хоризонталног заокретања *phidot_cmd* 12, односно да је различита од нуле колико је била приликом праволинијског кретања робота. Повећање брзине хоризонталног заокретања доводи до смањења пречника круга.

Остали параметри су идентични као код праволинијског кретања (слике 8.1.1.1. и 8.1.1.2).

```
gyroOffset = 0                  #ofset ziroskopa
gyroOffset1=0
a_gc = 0.8                      #konstanta filtera
a_r = 0.9960                    #konstanta filtera
a_gd = 0.999                    #faktor zaboravljanja
a_d = 0.8                       #konstanta filtera
calRefFilOut = 0
ts1 = 0.004                     #perioda odabiranja
theta_ref = 0                   #referentni ugao okretanja
thetadot_ref = 0                #referentna ugaona brzina
thetadot_cmd = 5                #ugaona brzina koja se zadaje
phidot_cmd =12                  #brzina horizontalnog okretanja
```

Слика 8.2.1. *Промењиве код кружног кретања*

Функције написане у Pythonu су идентичне и код кретања робота по кружници. Налазе се на сликама од 8.1.2.1 до 8.1.2.12. Разликује се task main, јер се код кретања по кружници поред угаоне брзине *thetadot_cmd* смањује и брзина хоризонталног окретања *phidot_cmd.*

```
def task_main():
            SetSensorHTGyro(GYRO_PORT)

for i in range(0, 250):
    GyroCalibration()
    Wait(20)


    gyroOffset1 = gyroOffset
    startTime = CurrentTick()
    pastTime = startTime
    repeated = 0
    while True:
     if (repeated < 5000):
         repeated = repeted+1
     else:
      thetadot_cmd = thetadot_cmd * 0.9
      phidot_cmd = phidot_cmd * 0.9
           Controller()
     while (CurrentTick() - pastTime < 4):
            continue

     pastTime = CurrentTick()
```

Слика 8.2.2. *Task main*

9. КРЕИРАЊЕ PYNXC КОДА

Помоћу pynxc конвертора код се конвертује у pynxc. Након пребацивања у pynxc, BricxCC аутоматски додељује свакој промењивој тип int па је потребно свуда променити у тип float, што је приказано на слици 9.1.

```
float gyroOffset = 0;
float gyroOffset1 = 0;
float a_gc = 0.8;
float a_r = 0.9960;
float a_gd = 0.999;//0.9990;
float a_d = 0.8;
float calRefFilOut = 0;
float ts1 = 0.004; //4 ms timestep
float theta_ref = 0;
float thetadot_ref = 0;
float thetadot_cmd = 5;
float phidot_cmd =10;
float psi = 0;
float psi_ref = 0;
float psidot_ref = 0;
float pwm_turn = 0;
float gyro = 0;
float psidot = 0;
long  rWheelRotation = 0;
long  lWheelRotation = 0;
```

Слика 9.1. *Додељивање типа промењивој*

Поред поменутих разлика, карактеристика nxc кода је да уколико се одговорајуће функције односе на одређену, оне морају бити изнад ње у програму, што значи да је битан распоред истих, у односу на Python, где није битан, већ се програм правилно извршава независно од распореда.

Правилан распоред у омогућава фунцкија inline void (слика 9.2).

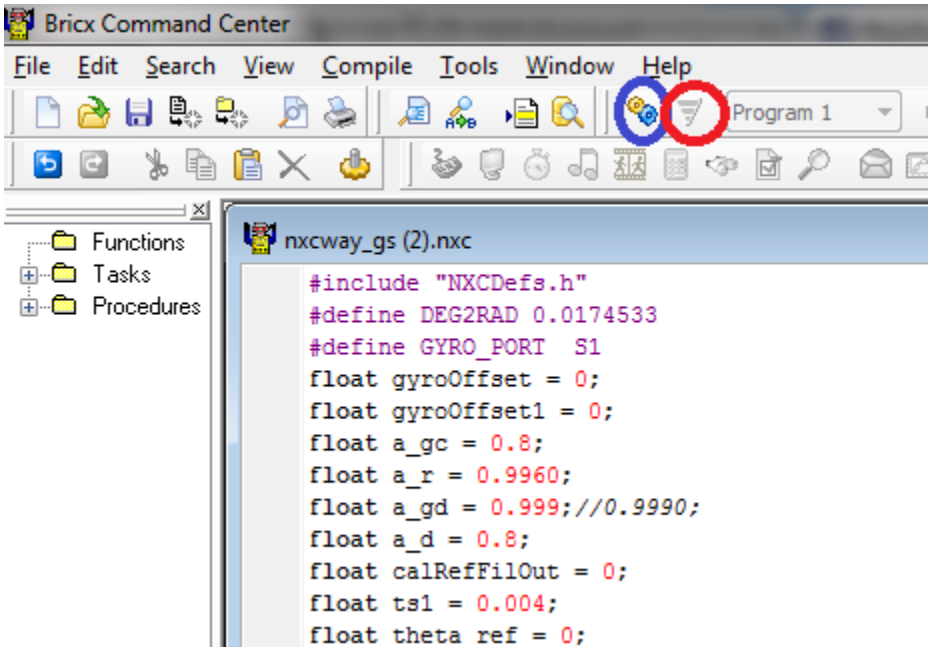
```
}
inline void DetectFlagFalling(){
       falling = (phiExists == 0) && (phiExistsDelay == 1);
}
inline void CalThetaDiff(){
       DetectFlagFalling();
       theta_diff = theta_r - theta_l;
       if (falling){
          theta_offset = theta_diff;
       }
      theta_diff = theta_diff - theta_offset;
```

Слика 9.2. *Inline void функција*

Других већих разлика у односу на Python нема.

9.1. Пребацивање програма на коцку

Након конвертовања програма и извршења претходно описаних исправки, потребно је програм компајлирати и исти пребацити на коцку. На наредној слици је приказано Bricx CC окружење и а плавим и црвеним круговима су означене иконице за компајлирање и пребацивање, респективно.



Слика 9.1.1. *Иконице за компајлирање и пребацивање програма*

Залеђено дугме за пребацивање означава да веза са роботом није успостављена. Када се програм пребаци на робота, чује се кратак звук.

Робот се са рачунаром повезује помоћу USB кабла. Како би пуштање робота у рад било успешно изведено потребно је проверити да ли су сви улази повезани на начин како је дефинисано у програму.

Пребачен програм се налази у фолдеру My files/Software files. Бирањем опцијеrun притиском наранџастог дугмета робот почиње са радом, док притискањем дугмета сивог испод престаје са извршавањем програма.

10. ДИЈАГРАМИ

Један од дефинисаних задатака мастер рада се односи на прикупљање података и исцртавање дијаграма у циљу приказивања понашања робота. У програму је потребно додати део који се односи на исцртавање графика (слика 10.1).

Потребно је креирати фајл где се чувају вредности жељених променљивих. Креирани фајл, који представља Excel табелу са вредностима, се након пуштања налази у коцки робота, и потребно је исти пребацити на рачунар ради даље обраде. Поновним покретањем робота, претходни резултати се аутоматски бришу и прави се поново нов фолдер за смештање резултата.

У Matlabu су направљени дијаграми који се односе на позицију и брзину праволинијског кретања и кретања по кружници.

```
    string asd = StrCat(FormatNum("%d", kT++),", ",FormatNum("%3.3f", theta), ", ", FormatNum("%3.3f", thetadot));
    WriteLnString(fH, asd,bv);
    brojac = 0;}
}

task main(){
DeleteFile("grafik.csv");
    result = CreateFile("grafik.csv", 32768, fH);
```

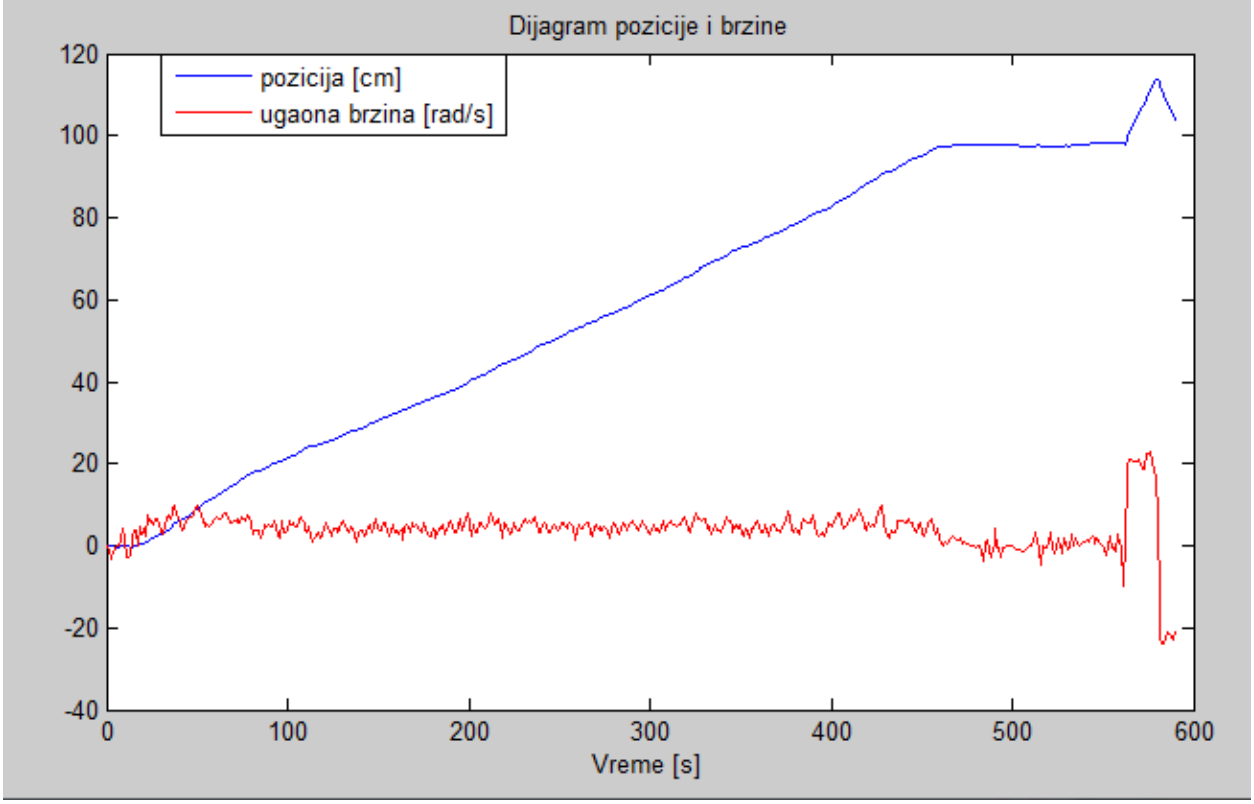
Слика 10.1. *Део кода који се односи на снимање резултата*

10.1. Дијаграм праволинијског кретања LEGO NXT робота

На наредном дијаграму је плавом бојом приказана позиција робота (угао *theta*), док је црвеном бојом приказана брзина истог *thetadot_cmd.*

Што се тиче позиције види се да се робот креће праволинијски скоро целом дужином, осим у последњем делу дијаграма где он полако успорава, што је условљено смањењем брзине. Уочава се мали прескок који означава кретање, до потпуног смиривања робота.

Када се посматара брзина исто се примећује да је константна осим на самом крају где се постепено смањује. Добијени графици одговарају кретању робота.

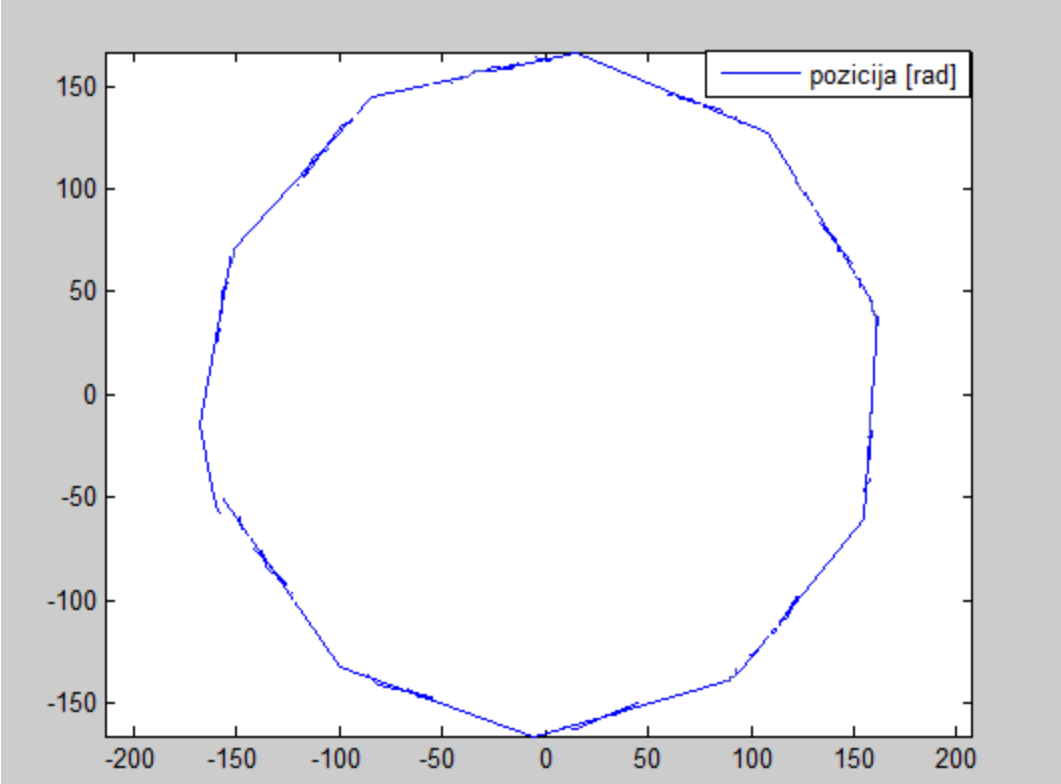


Слика 10.1.1. *Дијаграм позиције (theta) и брзине праволинијског кретања (thetadot_cmd) робота*

10.2. Дијаграми кретања LEGO NXT робота по кружници

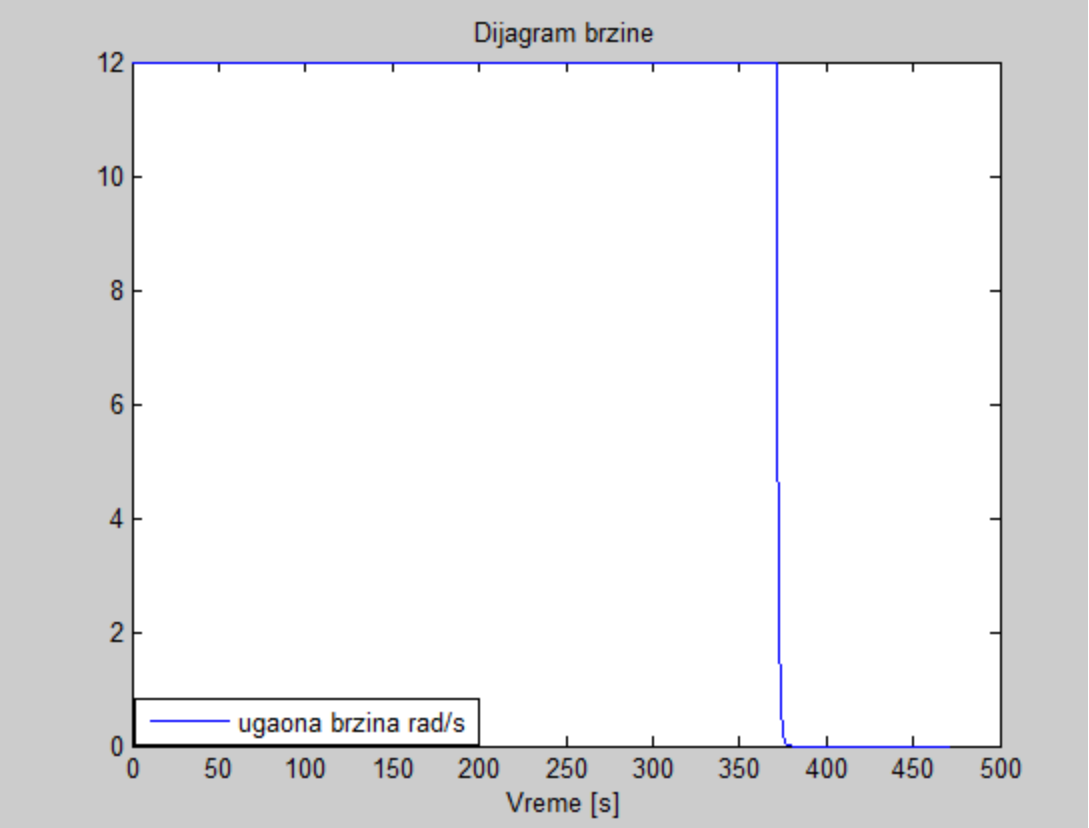
Приликом кретања робота по кружници направљени су дијаграми који се односе на позицију (угао *phi*) и брзину кретања *phidot_cmd*. Дијаграми су приказани одвојено на наредним сликама.

На слици 10.2.1. се налази позиција робота током кретања по кружници. Угао хоризонталног заокретања у оригиналном решењу није дат, већ је изведен интеграњем угаоне брзине хоризонталног заокретања. Само кретање робота по кружници није у потпуности идеално, што се и примећује на оштријим ивицама на слици.



Слика 10.2.1. *Кретање робота по кружници (phi)*

На слици 10.2.2. је приказана угаона брзина хоризонталног заокретања. Види се да иста има вредност 12, што је задато у програму,а затим пада до 0, што означава крај једног круга где се робот и зауставља.



Слика 10.2.2. *Угаона брзина хоризонталног окретања phidot_cmd*

11. ЗАКЉУЧАК

Циљ мастер рада подразумевао је пројектовање контролера мобилног робота у програмског језику Python. Овај задатак представља прави изазов за студенте на предмету Пројектовање система аутоматског управљања, где се годинама уназад унапређују већ постојећа решења везана за кретање робота.

Коришћењем LEGO роботике студенти замењују традиционалне методе учења програмирањем LEGO NXT робота у одговарајућим програмским језицима, чими се стиче увид у реалне проблеме који се јављају приликом програмирања.

Као што је поменуто, задатак се односио на имплементирање контролера на основу решења приказаног у Simulinku, што је урађено и у потпуности приказано у раду. Програмирано је праволинијско и кружно кретање робота, а такође су и направљени видео записи истих. Поред видео записа приказани су и потребни дијаграми, ради бољег разумевања кретања робота.

Даљи рад би подразумевао програмирање сложенијег кретања у циљу праћења понашања робота.

Поред наведеног, овај рад служи да наредним генерацијама пружи што више информација о самом LEGO Mindstorms NXT пакету и омогући лакше програмирање истог.

12. ЛИТЕРАТУРА

[1] hackeducation.com, Интернет адреса: http://hackeducation.com/2015/04/10/mindstorms, Приступљено:05.09.2017.

[2] generationrobots, Интернет адреса: http://www.generationrobots.com/media/Lego-Mindstorms-NXT-Education-Kit.pdf, Приступљено: 15.08.2017.

[3] uwindsor.ca, Интернет адреса:http://nxt.cs.uwindsor.ca//499football/features_limitations.pdf, Приступљено: 20.08.2017.

[4] J.Rieper, B. Bisballe Nyeng, K. Sohn, "Marvin – The Balancing Robot", University of Aarhus, January 2009.

[5] М.Матијевић, В.Цвјетковић, В.Филиповић, Н.Јовић, "Основни концепти аутоматике и мехатронике са LEGO програмабилнимм роботом", Техника, вол. 4/2014, 2014.

[6] Yamamoto, NXTway-GS Model Based Design – Control of self-balancing two-wheeled robot built with LEGO Mindstorms NXT, Интернет адреса: http://www.mathworks.com/matlabcentral/fileexchange/13399-embedded-coder-robot-nxt-demo, Приступљено: 20.08.2017.

[7] K. Astrom, R. Murray, "Feedback Systems", Princeton University Press, New Jersey, February 2009.

[8] compumotor.com, Интернет адреса: http://www.compumotor.com/whitepages/ServoFundamentals.pdf , Приступљено: 12.09.2017.

[9]М.Стојић, "Дигитални системи управљања", Крагујевац, 2012.

[10] Жељко Крстић 338/2013, Имплементација контролера LEGO Segway мобилног робота у програмском језику Python, мастер рад, Факултет инжењерских наука Универзитета у Крагујевцу, 2017.

[11] Тијана Цвијовић, и група студената, Трка LEGO Segway мобилних робота - Имплементација контролера LEGO Segway мобилног робота у програмском језику Python, пројектни задатак из ПСАУ, Факултет инжењерских наука Универзитета у Крагујевцу, 2017.

[12] bricxcc, Интернет адреса: http://bricxcc.sourceforge.net/nbc/nxcdoc/NXC_Guide.pdf, Приступљено:10.09.2017.

ПРИЛОГ

pravo.py

```
#definisanje promenljivih
gyroOffset = 0                #ofset ziroskopa
gyroOffset1=0
a_gc = 0.8                   #konstanta filtera
a_r = 0.9960                  #konstanta filtera
a_gd = 0.999                  #faktor zaboravljanja
a_d = 0.8                    #konstanta filtera
calRefFilOut = 0
ts1 = 0.004                   #perioda odabiranja
theta_ref = 0                 #referentni ugao okretanja
thetadot_ref = 0              #referentna ugaona brzina
thetadot_cmd = 5              #ugaona brzina koja se zadaje
phidot_cmd =0                 #ugaona brzina horizontalnog okretanja
psi = 0                      #ugao nagiba
psi_ref = 0                   #referentan ugao nagiba
psidot_ref = 0                #referentna ugaona brzina nagiba
pwm_turn = 0                  #pwm signal okretanja
gyro = 0                     #vrednost sa ziroskopa
psidot = 0                    #ugaona brzina nagiba
rWheelRotation =0             #ugao okretanja desnog tocka
lWheelRotation = 0            #ugao okretanja levog tocka
theta_l = 0                   #ugao okretanja levog tocka
theta_r = 0                   #ugao okretanja desnog tocka
theta = 0                    #srednja vrednost zaokretanja tocka
thFilt = 0                   #filt. srednja vrednost zaokretanja tocka
thetadot = 0                  #ugaoona brzina okretanja
theta_error = 0               #greska ugla
theta_errorIntegral = 0       #integraljena vrednost greske ugla
theta_offset = 0;             #vrednost ofseta
theta_diff = 0                #dif. vredonost ugla okretanja
thetaIntegral = 0             #integralna vredonost ugla okretanja
k_i = -0.4472                 #integralno pojacanje povratne sprege
k_fTheta = -0.8               #pojacanje ugla okretanja
k_fPsi = -66.3799             #pojacanje ugla nagiba
k_fThetadot = -1.2293         #pojacanje brzine okretanja
k_fPsidot = -7.0540           #pojacanje brzine nagiba
```

```
thFiltOld = 0                #filtrirana vrednost ugla okretanja
voltage = 0                  #definisanje napona
vol_max = 0                        #definisanje maksimalnognapona
voltageBatteryCompensated = 0        #napon na bateriji
voltageFrictionCompensated = 0       #viskozno trenje
pwm_gain = 1                       #pojacanje pwm signala
pwm_offset = 0                     #ofset pwm signala
phiExists = 0
phiExistsDelay = 0
psidotTerm = 0
thetaTerm = 0
thetadotTerm = 0
pwm_r = 0                    #pwm signal za desni tocak
pwm_l = 0                    #pwm signal za levi tocak
psiTerm = 0
falling = False
k_sync = 0.5                 #pojacanje za sinhroniciju tockova
pastTime = 0
startTime = 0
thetaSync = 0                #ukupni ugao zaokretanja
thetadifInt =0
k_tdI = 1.5                  #integralno pojacanje

def DiscreteDerivative( derivative,variable,  dt):
   derivative=(-variable)/dt

def DiscreteIntegrator(integral, integrand,  dt):
        integral += integrand * dt

def GyroCalibration():
        gyroOffset = SensorHTGyro(GYRO_PORT) * (1 - a_gc) + gyroOffset * a_gc
        PlayTone(440, 10)

def BalanceAndDriveControl():
        CalibrationReference()
        CalX1()
        CalculateControlSignal()
        PWMControl()

def CalculateControlSignal():
```

```
        DiscreteIntegrator(theta_errorIntegral, theta_error, ts1)
        thetaIntegral    = theta_errorIntegral * k_i
        psiTerm = (psi_ref - psi) * k_fPsi
        psidotTerm  = (psidot_ref - psidot) * k_fPsidot
        thetaTerm   = (theta_ref - theta) * k_fTheta
        thetadotTerm = (thetadot_ref - thetadot) * k_fThetadot
        voltage = thetaIntegral + psiTerm + psidotTerm + thetaTerm + thetadotTerm
        voltage = voltage

def CalibrationReference():
        CalRefFilter()
        thetadot_ref = calRefFilOut
        DiscreteIntegrator(theta_ref,thetadot_ref,ts1)
        pwm_turn = phidot_cmd

def CalRefFilter():
  calRefFilOut = thetadot_cmd * (1 - a_r) + calRefFilOut * a_r

def CalX1():
  CalGyro_offset()
  psidot = DEG2RAD * (gyro - gyroOffset1)*2
  DiscreteIntegrator(psi, psidot, ts1)
  theta_l = psi + lWheelRotation * DEG2RAD
  theta_r = psi + rWheelRotation * DEG2RAD
  theta   = (theta_l + theta_r) / 2
  thFiltOld = thFilt;
  CX1Filter()
  thetadot = (thFilt - thFiltOld) / ts1

def CalGyro_offset():
    gyroOffset1 = gyro * (1 - a_gd) + gyroOffset1 * a_gd

def CX1Filter():
        thFilt = theta * (1 - a_d) + thFilt * a_d


def PWMControl():
        vol_max = (BatteryLevel()) * 0.001089 - 0.625
        voltageBatteryCompensated = 100 * voltage / vol_max
```

```
        voltageFrictionCompensated = sign(voltageBatteryCompensated) * (pwm_gain * abs(voltageBatteryCompensated) + pwm_offset)
        phiExistsDelay = phiExists
        if (phidot_cmd != 0):
                phiExists = 1
        else:
                phiExists = 0

        PIDControl()

def PIDControl():
    pwm_r = voltageFrictionCompensated + pwm_turn
    if (pwm_r > 100):
        pwm_r = 100
    if (pwm_r < -100):
        pwm_r = -100
    CalThetaDiff()
    DiscreteIntegrator(thetadifInt, theta_diff, ts1)
    if (phiExists != 0):
       thetaSync = 0
    else:
       thetaSync = theta_diff * k_sync + thetadifInt * k_tdI
    pwm_l = voltageFrictionCompensated - pwm_turn + thetaSync
    if (pwm_r > 100):
        pwm_r = 100

    if (pwm_r < -100):
        pwm_r = -100


def CalThetaDiff():
    DetectFlagFalling()
    theta_diff = theta_r - theta_l
    if (falling):
      theta_offset = theta_diff
      theta_diff = theta_diff - theta_offset

def DetectFlagFalling():
    falling = (phiExists == 0) & (phiExistsDelay == 1)
```

```
def Controller():

    gyro = SensorHTGyro(GYRO_PORT)
    BalanceAndDriveControl()
    rWheelRotation = MotorRotationCount(OUT_A)
    lWheelRotation = MotorRotationCount(OUT_C)
    OnFwd(OUT_A, pwm_r)
    OnFwd(OUT_C, pwm_l)

def task_main():
            SetSensorHTGyro(GYRO_PORT)

for i in range(0, 250):
    GyroCalibration()
    Wait(20)


    gyroOffset1 = gyroOffset
    startTime = CurrentTick()
    pastTime = startTime
    repeated = 0
    while True:
     if (repeated < 5000):
         repeated = repeted+1
     else:
      thetadot_cmd = thetadot_cmd * 0.9
      Controller()
     while (CurrentTick() - pastTime < 4):
           continue

     pastTime = CurrentTick()
```

pravo.nxc

```
long  rWheelRotation = 0;
long  lWheelRotation = 0;
float theta_l = 0;
float theta_r = 0;
float theta = 0;
float thFilt = 0;
```

```
float thetadot = 0;
float theta_error = 0;
float theta_errorIntegral = 0;
float theta_offset = 0;
float theta_diff = 0;
float thetaIntegral = 0;
float k_i = -0.4472;
/*float k_fTheta = -0.8351; //-0.8
float k_fPsi = -34.1896; //-66.3799
float k_fThetadot = -1.0995; //-1.2293
float k_fPsidot = -2.8141; //-7.0540   */
float k_fTheta = -0.8   ;
float k_fPsi = -66.3799          ;
float k_fThetadot = -1.2293    ;
float k_fPsidot = -7.0540     ;
float thFiltOld = 0;
float voltage = 0;
float vol_max = 0;
float voltageBatteryCompensated = 0;
float voltageFrictionCompensated = 0;
float pwm_gain = 1;
float pwm_offset = 0;
int phiExists = 0;
int phiExistsDelay = 0;
float psidotTerm = 0;
float thetaTerm = 0;
float thetadotTerm = 0;
int pwm_r = 0;
int pwm_l = 0;
float psiTerm = 0;
bool falling = false;
float k_sync = 0.5;
unsigned int pastTime = 0;
unsigned int startTime = 0;
float thetaSync = 0;
float thetadifInt = 0;
float k_tdI = 1.5;

float thetaAc=0;
float thetadot1=0;
```

```
inline void DiscreteDerivative(float &derivative, float variable, float dt){
	derivative = (-variable) / dt;
}

inline void DiscreteIntegrator(float &integral, float integrand, float dt){
	integral += integrand * dt;
}

inline void CalGyro_offset(){
	gyroOffset1 = gyro * (1 - a_gd) + gyroOffset1 * a_gd;
}

inline void CalRefFilter(){
  calRefFilOut = thetadot_cmd * (1 - a_r) + calRefFilOut * a_r;
}

inline void CalibrationReference(){
	CalRefFilter();
	thetadot_ref = calRefFilOut;
	DiscreteIntegrator(theta_ref,thetadot_ref,ts1);
	pwm_turn = phidot_cmd;
}

inline void  GyroCalibration(){
	gyroOffset = SensorHTGyro(GYRO_PORT) * (1 - a_gc) + gyroOffset * a_gc;
	PlayTone(440, 10);
}

inline void CX1Filter(){
	thFilt = theta * (1 - a_d) + thFilt * a_d;
}
inline void CalX1(){
  CalGyro_offset();
	psidot = DEG2RAD * (gyro - gyroOffset1) * 2;
	DiscreteIntegrator(psi, psidot, ts1);
	theta_l = psi + lWheelRotation * DEG2RAD;
	theta_r = psi + rWheelRotation * DEG2RAD;
	theta   = (theta_l + theta_r) / 2;
```

```
        thFiltOld = thFilt;
        CX1Filter();
        //DiscreteDerivative(thetadot, thFilt, ts1);
        thetadot = (thFilt - thFiltOld) / ts1;
        thetadot1=thetadot;

        thetaAc=(thetadot-thetadot1)/ts1;
}
inline void CalculateControlSignal(){
        DiscreteIntegrator(theta_errorIntegral, theta_error, ts1);
        thetaIntegral    = theta_errorIntegral * k_i;
        psiTerm          = (psi_ref - psi) * k_fPsi;
        psidotTerm       = (psidot_ref - psidot) * k_fPsidot;
        thetaTerm        = (theta_ref - theta) * k_fTheta;
        thetadotTerm     = (thetadot_ref - thetadot) * k_fThetadot;
        voltage          = thetaIntegral + psiTerm + psidotTerm + thetaTerm +
thetadotTerm;
        voltage = voltage;
  ClearScreen();
  NumOut(2, LCD_LINE7, rWheelRotation);
}
inline void DetectFlagFalling(){
     falling = (phiExists == 0) && (phiExistsDelay == 1);
}
inline void CalThetaDiff(){
     DetectFlagFalling();
     theta_diff = theta_r - theta_l;
     if (falling){
       theta_offset = theta_diff;
     }
    theta_diff = theta_diff - theta_offset;
}
inline void  PIDControl(){
     pwm_r = voltageFrictionCompensated + pwm_turn;


          //NumOut(2, LCD_LINE5, gyro);
     pwm_r = (pwm_r > 100) ? 100 : (pwm_r < -100) ? -100 : pwm_r;
     CalThetaDiff();
     DiscreteIntegrator(thetadifInt, theta_diff, ts1);
```

```
      thetaSync = (phiExists != 0) ? 0 : (theta_diff * k_sync + thetadifInt * k_tdI);
      pwm_l = voltageFrictionCompensated - pwm_turn + thetaSync;

      pwm_l = (pwm_l > 100) ? 100 : (pwm_l < -100) ? -100 : pwm_l;
         //NumOut(2, LCD_LINE3, pwm_r);
                           //NumOut(2, LCD_LINE4, pwm_l);

}
inline void PWMControl(){
          vol_max = (BatteryLevel()) * 0.001089 - 0.625;
          voltageBatteryCompensated = 100 * voltage / vol_max;
          voltageFrictionCompensated = sign(voltageBatteryCompensated) * (pwm_gain *
abs(voltageBatteryCompensated) + pwm_offset);
  phiExistsDelay = phiExists;
          if (phidot_cmd != 0){
    phiExists = 1;
          } else {
    phiExists = 0;
          }
          PIDControl();
}

inline void  BalanceAndDriveControl(){

          CalibrationReference();
          CalX1();
          theta_error = theta_ref - theta;
          CalculateControlSignal();
          PWMControl();
}

inline void Controller(){
 /* if (CurrentTick() - startTime < 4000){
     StartTask(GyroCalibration);
  } else {  */
    gyro = SensorHTGyro(GYRO_PORT);
    BalanceAndDriveControl();
    rWheelRotation = MotorRotationCount(OUT_A);
    lWheelRotation = MotorRotationCount(OUT_C);
    OnFwd(OUT_A, pwm_r);
```

```
   OnFwd(OUT_C, pwm_l);
                /*          SetOutput(OUT_A,          Power,          pwm_r,
OutputMode,OUT_MODE_REGULATED,
          RegMode, OUT_REGMODE_SPEED,
               RunState,OUT_RUNSTATE_RUNNING);
                                        SetOutput(OUT_C,          Power,          pwm_l,
OutputMode,OUT_MODE_REGULATED,
          RegMode, OUT_REGMODE_SPEED, RunState,OUT_RUNSTATE_RUNNING
);              */
     //NumOut(2, LCD_LINE6, psidot);
 // }
 if (brojac++ == 10){
   string asd = StrCat(FormatNum("%d", kT++),", ",FormatNum("%3.3f", theta), ", ",
FormatNum("%3.3f", thetadot));
   WriteLnString(fH, asd,bv);
   brojac = 0;}
}

task main(){
DeleteFile("grafik.csv");
   result = CreateFile("grafik.csv", 32768, fH);
          SetSensorHTGyro(GYRO_PORT);

    for (int i = 0; i < 250; i ++){
     GyroCalibration();
     Wait(20);
    }
    gyroOffset1 = gyroOffset;
       //NumOut(2, LCD_LINE1, gyroOffset);
    startTime = CurrentTick();
    pastTime = startTime;
         int repeated = 0;
    while(true){
    if (repeated < 5000)
                {
                        repeated++;
                }
                else
                {
```

```
    thetadot_cmd = thetadot_cmd * 0.9;


                }
        Controller();
        while (CurrentTick() - pastTime < 4) continue;

        pastTime = CurrentTick();

    }
}
```

krug.py

```
#definisanje promenljivih
gyroOffset = 0              #ofset ziroskopa
gyroOffset1=0
a_gc = 0.8                  #konstanta filtera
a_r = 0.9960                #konstanta filtera
a_gd = 0.999                #faktor zaboravljanja
a_d = 0.8                   #konstanta filtera
calRefFilOut = 0
ts1 = 0.004                 #perioda odabiranja
theta_ref = 0               #referentni ugao okretanja
thetadot_ref = 0            #referentna ugaona brzina
thetadot_cmd = 5            #ugaona brzina koja se zadaje
phidot_cmd =12              #ugaona brzina horizontalnog okretanja
psi = 0                     #ugao nagiba
psi_ref = 0                 #referentan ugao nagiba
psidot_ref = 0              #referentna ugaona brzina nagiba
pwm_turn = 0                #pwm signal okretanja
gyro = 0                    #vrednost sa ziroskopa
psidot = 0                  #ugaona brzina nagiba
rWheelRotation =0           #ugao okretanja desnog tocka
lWheelRotation = 0          #ugao okretanja levog tocka
theta_l = 0                 #ugao okretanja levog tocka
theta_r = 0                 #ugao okretanja desnog tocka
theta = 0                   #srednja vrednost zaokretanja tocka
thFilt = 0                  #filt. srednja vrednost zaokretanja tocka
thetadot = 0                #ugaoona brzina okretanja
```

```
theta_error = 0              #greska ugla
theta_errorIntegral = 0        #integraljena vrednost greske ugla
theta_offset = 0;             #vrednost ofseta
theta_diff = 0               #dif. vredonost ugla okretanja
thetaIntegral = 0             #integralna vredonost ugla okretanja
k_i = -0.4472                #integralno pojacanje povratne sprege
k_fTheta = -0.8               #pojacanje ugla okretanja
k_fPsi = -66.3799              #pojacanje ugla nagiba
k_fThetadot = -1.2293          #pojacanje brzine okretanja
k_fPsidot = -7.0540           #pojacanje brzine nagiba
thFiltOld = 0                #filtrirana vrednost ugla okretanja
voltage = 0                  #definisanje napona
vol_max = 0                   #definisanje maksimalnognapona
voltageBatteryCompensated = 0   #napon na bateriji
voltageFrictionCompensated = 0  #viskozno trenje
pwm_gain = 1                  #pojacanje pwm signala
pwm_offset = 0                #ofset pwm signala
phiExists = 0
phiExistsDelay = 0
psidotTerm = 0
thetaTerm = 0
thetadotTerm = 0
pwm_r = 0                    #pwm signal za desni tocak
pwm_l = 0                    #pwm signal za levi tocak
psiTerm = 0
falling = False
k_sync = 0.5                 #pojacanje za sinhroniciju tockova
pastTime = 0
startTime = 0
thetaSync = 0                 #ukupni ugao zaokretanja
thetadifInt =0
k_tdI = 1.5                  #integralno pojacanje


def DiscreteDerivative( derivative,variable,  dt):
   derivative=(-variable)/dt

def DiscreteIntegrator(integral, integrand,  dt):
        integral += integrand * dt
```

```
def GyroCalibration():
        gyroOffset = SensorHTGyro(GYRO_PORT) * (1 - a_gc) + gyroOffset * a_gc
        PlayTone(440, 10)

def BalanceAndDriveControl():
        CalibrationReference()
        CalX1()
        CalculateControlSignal()
        PWMControl()

def CalculateControlSignal():
        DiscreteIntegrator(theta_errorIntegral, theta_error, ts1)
        thetaIntegral    = theta_errorIntegral * k_i
        psiTerm = (psi_ref - psi) * k_fPsi
        psidotTerm  = (psidot_ref - psidot) * k_fPsidot
        thetaTerm   = (theta_ref - theta) * k_fTheta
        thetadotTerm = (thetadot_ref - thetadot) * k_fThetadot
        voltage = thetaIntegral + psiTerm + psidotTerm + thetaTerm + thetadotTerm
        voltage = voltage

def CalibrationReference():
        CalRefFilter()
        thetadot_ref = calRefFilOut
        DiscreteIntegrator(theta_ref,thetadot_ref,ts1)
        pwm_turn = phidot_cmd

def CalRefFilter():
  calRefFilOut = thetadot_cmd * (1 - a_r) + calRefFilOut * a_r

def CalX1():
  CalGyro_offset()
  psidot = DEG2RAD * (gyro - gyroOffset1)*2
  DiscreteIntegrator(psi, psidot, ts1)
  theta_l = psi + lWheelRotation * DEG2RAD
  theta_r = psi + rWheelRotation * DEG2RAD
  theta   = (theta_l + theta_r) / 2
  thFiltOld = thFilt;
  CX1Filter()
  thetadot = (thFilt - thFiltOld) / ts1
```

```
def CalGyro_offset():
   gyroOffset1 = gyro * (1 - a_gd) + gyroOffset1 * a_gd

def CX1Filter():
        thFilt = theta * (1 - a_d) + thFilt * a_d


def PWMControl():
        vol_max = (BatteryLevel()) * 0.001089 - 0.625
        voltageBatteryCompensated = 100 * voltage / vol_max
        voltageFrictionCompensated = sign(voltageBatteryCompensated) * (pwm_gain *
abs(voltageBatteryCompensated) + pwm_offset)
        phiExistsDelay = phiExists
        if (phidot_cmd != 0):
                phiExists = 1
        else:
                phiExists = 0

        PIDControl()

def PIDControl():
    pwm_r = voltageFrictionCompensated + pwm_turn
    if (pwm_r > 100):
        pwm_r = 100
    if (pwm_r < -100):
        pwm_r = -100
    CalThetaDiff()
    DiscreteIntegrator(thetadifInt, theta_diff, ts1)
    if (phiExists != 0):
       thetaSync = 0
    else:
       thetaSync = theta_diff * k_sync + thetadifInt * k_tdI
    pwm_l = voltageFrictionCompensated - pwm_turn + thetaSync
    if (pwm_r > 100):
        pwm_r = 100

    if (pwm_r < -100):
        pwm_r = -100
```

```
def CalThetaDiff():
      DetectFlagFalling()
      theta_diff = theta_r - theta_l
      if (falling):
        theta_offset = theta_diff
        theta_diff = theta_diff - theta_offset

def DetectFlagFalling():
      falling = (phiExists == 0) & (phiExistsDelay == 1)

def Controller():

   gyro = SensorHTGyro(GYRO_PORT)
   BalanceAndDriveControl()
   rWheelRotation = MotorRotationCount(OUT_A)
   lWheelRotation = MotorRotationCount(OUT_C)
   OnFwd(OUT_A, pwm_r)
   OnFwd(OUT_C, pwm_l)

def task_main():
            SetSensorHTGyro(GYRO_PORT)

for i in range(0, 250):
   GyroCalibration()
   Wait(20)


   gyroOffset1 = gyroOffset
   startTime = CurrentTick()
   pastTime = startTime
   repeated = 0
   while True:
    if (repeated < 5000):
       repeated = repeted+1
    else:
     thetadot_cmd = thetadot_cmd * 0.9
     phidot_cmd = phidot_cmd * 0.9
     Controller()
    while (CurrentTick() - pastTime < 4):
```

```
        continue

    pastTime = CurrentTick()
```

krug.nxc

```
#include "NXCDefs.h"
#define DEG2RAD 0.0174533
#define GYRO_PORT  S1
int brojac = 0;
short bv;
unsigned int result;
byte fH;
long kT = 0;
float gyroOffset = 0;
float gyroOffset1 = 0; // should be initialized to initial condition (gyroOffset1 = gyroOffset)
float a_gc = 0.8;
float a_r = 0.9960;
float a_gd = 0.999;//0.9990;
float a_d = 0.8;
float calRefFilOut = 0;
float ts1 = 0.004; //4 ms timestep
float theta_ref = 0;
float thetadot_ref = 0;
float thetadot_cmd = 5;
float phidot_cmd =12;
float psi = 0;
float psi_ref = 0;
float psidot_ref = 0;
float pwm_turn = 0;
float gyro = 0;
float psidot = 0;
long  rWheelRotation = 0;
long  lWheelRotation = 0;
float theta_l = 0;
float theta_r = 0;
float theta = 0;
float thFilt = 0;
float thetadot = 0;
float theta_error = 0;
```

```
float theta_errorIntegral = 0;
float theta_offset = 0;
float theta_diff = 0;
float thetaIntegral = 0;
float k_i = -0.4472;
/*float k_fTheta = -0.8351; //-0.8
float k_fPsi = -34.1896; //-66.3799
float k_fThetadot = -1.0995; //-1.2293
float k_fPsidot = -2.8141; //-7.0540   */
float k_fTheta = -0.8   ;
float k_fPsi = -66.3799         ;
float k_fThetadot = -1.2293    ;
float k_fPsidot = -7.0540     ;
float thFiltOld = 0;
float voltage = 0;
float vol_max = 0;
float voltageBatteryCompensated = 0;
float voltageFrictionCompensated = 0;
float pwm_gain = 1;
float pwm_offset = 0;
int phiExists = 0;
int phiExistsDelay = 0;
float psidotTerm = 0;
float thetaTerm = 0;
float thetadotTerm = 0;
int pwm_r = 0;
int pwm_l = 0;
float psiTerm = 0;
bool falling = false;
float k_sync = 0.5;
unsigned int pastTime = 0;
unsigned int startTime = 0;
float thetaSync = 0;
float thetadifInt = 0;
float k_tdI = 1.5;

float phi = 0;
inline void DiscreteDerivative(float &derivative, float variable, float dt){
        derivative = (-variable) / dt;
}
```

```
inline void DiscreteIntegrator(float &integral, float integrand, float dt){
	integral += integrand * dt;
}

inline void CalGyro_offset(){
	gyroOffset1 = gyro * (1 - a_gd) + gyroOffset1 * a_gd;
}

inline void CalRefFilter(){
  calRefFilOut = thetadot_cmd * (1 - a_r) + calRefFilOut * a_r;
}

inline void CalibrationReference(){
	CalRefFilter();
	thetadot_ref = calRefFilOut;
	DiscreteIntegrator(theta_ref,thetadot_ref,ts1);
	pwm_turn = phidot_cmd;
	DiscreteIntegrator(phi,pwm_turn,ts1);
}

inline void  GyroCalibration(){
	gyroOffset = SensorHTGyro(GYRO_PORT) * (1 - a_gc) + gyroOffset * a_gc;
	PlayTone(440, 10);
}

inline void CX1Filter(){
	thFilt = theta * (1 - a_d) + thFilt * a_d;
}
inline void CalX1(){
  CalGyro_offset();
	psidot = DEG2RAD * (gyro - gyroOffset1) * 2;
	DiscreteIntegrator(psi, psidot, ts1);
	theta_l = psi + lWheelRotation * DEG2RAD;
	theta_r = psi + rWheelRotation * DEG2RAD;
	theta   = (theta_l + theta_r) / 2;
	thFiltOld = thFilt;
	CX1Filter();
	//DiscreteDerivative(thetadot, thFilt, ts1);
	thetadot = (thFilt - thFiltOld) / ts1;
```

```
}
inline void CalculateControlSignal(){
        DiscreteIntegrator(theta_errorIntegral, theta_error, ts1);
        thetaIntegral    = theta_errorIntegral * k_i;
        psiTerm          = (psi_ref - psi) * k_fPsi;
        psidotTerm       = (psidot_ref - psidot) * k_fPsidot;
        thetaTerm        = (theta_ref - theta) * k_fTheta;
        thetadotTerm     = (thetadot_ref - thetadot) * k_fThetadot;
        voltage          = thetaIntegral + psiTerm + psidotTerm + thetaTerm +
thetadotTerm;
        voltage = voltage;
  ClearScreen();
  NumOut(2, LCD_LINE7, rWheelRotation);
}
inline void DetectFlagFalling(){
     falling = (phiExists == 0) && (phiExistsDelay == 1);
}
inline void CalThetaDiff(){
     DetectFlagFalling();
     theta_diff = theta_r - theta_l;
     if (falling){
       theta_offset = theta_diff;
     }
    theta_diff = theta_diff - theta_offset;
}
inline void  PIDControl(){
     pwm_r = voltageFrictionCompensated + pwm_turn;


          //NumOut(2, LCD_LINE5, gyro);
     pwm_r = (pwm_r > 100) ? 100 : (pwm_r < -100) ? -100 : pwm_r;
     CalThetaDiff();
     DiscreteIntegrator(thetadifInt, theta_diff, ts1);
     thetaSync = (phiExists != 0) ? 0 : (theta_diff * k_sync + thetadifInt * k_tdI);
     pwm_l = voltageFrictionCompensated - pwm_turn + thetaSync;

     pwm_l = (pwm_l > 100) ? 100 : (pwm_l < -100) ? -100 : pwm_l;
       //NumOut(2, LCD_LINE3, pwm_r);
                    //NumOut(2, LCD_LINE4, pwm_l);
```

```
}
inline void PWMControl(){
	vol_max = (BatteryLevel()) * 0.001089 - 0.625;
	voltageBatteryCompensated = 100 * voltage / vol_max;
	voltageFrictionCompensated = sign(voltageBatteryCompensated) * (pwm_gain *
abs(voltageBatteryCompensated) + pwm_offset);
  phiExistsDelay = phiExists;
	if (phidot_cmd != 0){
   phiExists = 1;
	} else {
   phiExists = 0;
	}
	PIDControl();
}

inline void  BalanceAndDriveControl(){

	CalibrationReference();
	CalX1();
	theta_error = theta_ref - theta;
	CalculateControlSignal();
	PWMControl();
}

inline void Controller(){
 /* if (CurrentTick() - startTime < 4000){
    StartTask(GyroCalibration);
  } else {  */
   gyro = SensorHTGyro(GYRO_PORT);
   BalanceAndDriveControl();
   rWheelRotation = MotorRotationCount(OUT_A);
   lWheelRotation = MotorRotationCount(OUT_C);
   OnFwd(OUT_A, pwm_r);
   OnFwd(OUT_C, pwm_l);
		/*	SetOutput(OUT_A, Power, pwm_r,
OutputMode,OUT_MODE_REGULATED,
	RegMode, OUT_REGMODE_SPEED,
		RunState,OUT_RUNSTATE_RUNNING);
					SetOutput(OUT_C, Power, pwm_l,
OutputMode,OUT_MODE_REGULATED,
```

```
            RegMode, OUT_REGMODE_SPEED, RunState,OUT_RUNSTATE_RUNNING
);                  */
     //NumOut(2, LCD_LINE6, psidot);
 // }
 if (brojac++ == 10){
   string asd = StrCat(FormatNum("%d", kT++), ", ", FormatNum("%3.3f", pwm_turn), ",   ",
",FormatNum("%3.3f", phi));
   WriteLnString(fH, asd,bv);
   brojac = 0;}
}

task main(){
DeleteFile("krug.csv');
   result = CreateFile("krug.csv', 32768, fH);
            SetSensorHTGyro(GYRO_PORT);

     for (int i = 0; i < 250; i ++){
      GyroCalibration();
      Wait(20);
     }
     gyroOffset1 = gyroOffset;
        //NumOut(2, LCD_LINE1, gyroOffset);
     startTime = CurrentTick();
     pastTime = startTime;
          int repeated = 0;
     while(true){
     if (repeated < 6000)
                    {
                              repeated++;
                    }
                    else
                    {

      thetadot_cmd = thetadot_cmd * 0.9;
      phidot_cmd=phidot_cmd * 0.9;

                    }
        Controller();
        while (CurrentTick() - pastTime < 4) continue;
```

```
        pastTime = CurrentTick();

    }
}
```