ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

STRATEGIC PLANNING FOR MODULAR ROBOTIC STRUCTURES

M.Sc. THESIS

Mehmet Cem ŞANLI

Department of Mechanical Engineering

Mechatronics Engineering Programme

AUGUST 2015

<u>**ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY**</u>

STRATEGIC PLANNING FOR MODULAR ROBOTIC STRUCTURES

M.Sc. THESIS

Mehmet Cem ŞANLI
(518091048)

Department of Mechanical Engineering

Mechatronics Engineering Programme

Thesis Advisor: Assoc. Prof. Dr. Pınar BOYRAZ

AUGUST 2015

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

MODULER ROBOTİK YAPILARDA STRATEJİK PLANLAMA

YÜKSEK LİSANS TEZİ

Mehmet Cem ŞANLI
(518091048)

Makine Mühendisliği Anabilim Dalı

Mekatronik Mühendisliği Programı

Tez Danışmanı: Doç. Dr. Pınar BOYRAZ

AĞUSTOS 2015

Mehmet Cem Şanlı, a M.Sc. student of ITU Graduate School of Science and Technology student ID 518091048, successfully defended the thesis/dissertation entitled "**STRATEGIC PLANNING FOR MODULAR ROBOTIC STRUCTURES**", which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Assoc. Prof. Dr. Pınar BOYRAZ** İstanbul Technical University	....
Jury Members :	**Prof. Dr. Şeniz ERTUĞRUL** Istanbul Technical University	....
	Asst. Prof. Dr. Akın DELİBAŞI Yıldız Technical University	....

Date of Submission : 4 May 2015
Date of Defense : 19 August 2015

To my family and friends,

FOREWORD

Firstly, I would like to thank all people who have helped and inspired me during my M.Sc. study in Istanbul Technical University. I would especially like to thank to my supervisor Assoc. Prof. Pınar BOYRAZ for her endless patience and support during my study. Without her guidance this work would not be possible.
I would like to dedicate this study to my family and friends and I am truly thankful for their lifelong support.

May 2015

Mehmet Cem ŞANLI
Systems Engineer

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

STRATEGIC PLANNING FOR MODULAR ROBOTIC STRUCTURES

SUMMARY

The advancements in robotics increases the expectations from robots in terms of capability and capacity. Today people expect robots to be more autonomous, more functional, more versatile and more affordable. A robot that can play the violin, but only playing the violin is not fascinating anymore. Today we expect from the robot that plays the violin to come and ask our wish, present what we wished, avoid the obstacles while doing this, adapt to a dynamic environment it has not been before and solve its own problems if it has any while operating. Besides, these robots should be affordable.

The purpose of this study is to design a functional and versatile modular robotic structure and develop a strategic planning algorithm that can answer these demanding expectations. The modular robotic system is expected to be able to create configurations that can implement quadruped and wheeled locomotion methods and to have configuration specific abilities such as passing over or under obstacles. Another expectation is to decide the appropriate configuration to implement proper locomotion method and configuration specific ability with the help of the developed strategic planning algorithm.

The modular robotic structure designed in this study is a chain type modular robotic system which are known for their suitability in implementing advanced locomotion methods easily. To overcome the general self reconfiguration problem of this class, the modules are designed to be self mobile. To control the modular robotic structure a strategic planning algorithm is developed. The architecture of the algorithm can be classified as hybrid deliberative/reactive. The hybrid architecture is chosen for its two layered architecture which benefits both from the advantages of hierarchical paradigm and reactive paradigm. After inspecting different simulation programs such as Webots, Gazebo and V-Rep. The last one is found to be the best choice for the needs of this study based on its good documentation and ease of use.

Modules of the robotic structure are designed to have a visual sensor, a force sensor, three joints and four connection points. The locomotion method of the modules are similar to an inchworm and it is developed by analysing the kinematic chain of a single module. A position control algorithm is also developed which is mainly used for assembly of configurations. The assembly procedure of configurations is determined and a role distribution algorithm is developed.

Two configurations to implement quadruped and wheeled locomotion are designed. The configuration specific abilities for both configurations are designed and the design proces of the robotic structure is finalized.

After the robotic structure is designed, the development of the strategic planning algorithm started. A two layered hybrid deliberative/reactive architecture is developed to control the robotic structure. The deliberative layer is used to generate a plan consisting of sub goals to drive the robotic structure from its initial state to the desired goal state. The reactive layer of the algorithm is more like a feedback controller. This layer is used to execute the plan generated by the deliberative layer.

The designed robotic structure and strategic planning algorithm is tested in V-Rep by creating a complex test area. In this complex test area, there are obstacles that can be passed over or under with the configuration specific abilities of the robotic structure positioned between the initial state of the robotic structure and its desired goal state. The behavior and performance of the whole structure is tested based on its success to reach the desired goal state.
The test results of this study is presented in detail and interpreted. According the test results, it is proven that with good design and strategic planning algorithm, modular robotic structures are more functional and versatile over their monolithic counterparts.

MODÜLER ROBOTİK YAPILARDA STRATEJİK PLANLAMA

ÖZET

Robotik alanındaki araştırmalar arttıkça ve robotiğin dayalı olduğu mekanik, elektronik ve yazılım gibi alanlarda hızına yetişilemeyen gelişmeler oldukça, insanların robotlardan beklentileri de değişmeye ve gün geçtikçe daha da talepkar olmaya başladı. İnsanlar artık robotlardan daha otonom, daha fonksiyonel, daha çok yönlü ve daha hesaplı olmalarını bekliyorlar. İnsanoğlu artık keman çalabilen ancak sadece keman çalabilen bir robotu eskisi kadar göz kamaştırıcı bulmuyor. Artık zamanında hayranlıkla izlenen keman çalabilen robotun, gelip isteklerimizi sorması, isteklerimizi sunarken karşısına çıkan engele takılmaması, değişik bir ortama girdiğinde hemen adapte olması, hatta bir problem yaşadığında problemini kendi kendine çözmesini bekliyoruz. Aynı zamanda bunları yapabilen bir robotun daha ucuza mal edilmesini ve bize daha hesaplı bir fiyatla sunulmasını istiyoruz.
Robotik araştırmacıları, insanların bu talepkar beklentilerini karşılayabilmek için değişik alanlarda çalışmalarını sürdürmeye devam ediyorlar. Bir kısmı daha otonom ve daha fonksiyonel, operatör kontrolünden daha bağımsız robotlar geliştirmek için yapay zeka araştırmalarına yönelirken, bir kısmı elektronik ve mekanik optimizasyonlarla hem hesaplı hem çok yönlü robotlar yaratmaya çalışıyorlar. Bu tarz istekleri karşılayabilecek monolitik robotların olduğu ve daha bir çoğunun da geliştirilebileceği inkar edilmese de daha fonksiyonel, daha çok yönlü ve daha hesaplı robotik sistemlerin geliştirilmesi genellikle çoklu robot sistemleri ile sağlanmaktadır. Çoklu robot sistemleri denildiğinde ilk akla gelen iki tip robotik sistem vardır; (1) sürü robot sistemleri ve (2) modüler robotik sistemler. İki sistem de birbirine çok benzer görünse de aslında temel prensipleri açısından oldukça büyük farkları bulunmaktadır.
Bu iki sistemin birbirinden ayrıldığı en temel nokta, modüler robotiğin temelinde birleşerek daha fonksiyonel robotlar oluşturmak varken, sürü robotikte birleşmek gibi bir olgu yoktur. Modüler robotik sistemlerde çoğu zaman modüller tek başlarına görev yapmazken, sürü robot sistemlerinde her robot tek başına çalışır ve sürü kendisinden beklenen görevleri bu şekilde yerine getirir. Her iki sistem çok sayıda robottan oluşsa da fonksiyonellik açısından, sürü robotik sistemlerdeki robotlar modüler robotik sistemlerdekilere oranla oldukça basittirler. Genel kontrol sistemleri açısından da iki robotik sınıf arasında farklar vardır. Sürü robotik sistemlerde daha dağıtılmış kontrol sistemleri kullanılırken, modüler robotik sistemlerde dağınık kontrol algoritmaları kullanılsa da sistemin temelinde birleşmek bulunduğundan, kontrol sistemlerinde merkezi ögeler de çoğunlukla bulunur.
Bu çalışmanın amacı insanların robotlardan yeni beklentileri doğrultusunda daha fonksiyonel ve çok yönlü bir modüler robotik sistem tasarlamak ve bu sisteme bir stratejik planlama algoritması uygulayarak sistemin fonksiyonelliğini ve çok yönlülüğünü ön plana çıkarmaktır. Tasarlanacak modüler robotik sistemden beklentiler dört ayaklı ve tekerlekli gibi değişik ilerleme metodlarını uygulayabilecek bir yapıda olması, konfigürasyonlara özel engel üzerinden atlama ya da engel

altından geçme gibi yeteneklere sahip olması ve geliştirilen stratejik planlama algoritması sayesinde doğru ilerleme metodunu ve gerekli yeteneğini kullanabileceği doğru konfigürasyonu seçerek hedefine kolayca ulaşabilmesidir.
Bu çalışmada modüler robotik bir sistemin tasarlanmasının ana nedeni modüler robotik sistemlerin genel prensiplerinin basit modüllerin birleşerek ya da zaten birleşik şekilde çalışan modüllerin bağlantı şekillerini değiştirerek daha karmaşık ve daha fonksiyonel robotik konfigürasyonlar oluşturabilmeleridir. Yeniden insanların robotlardan beklentilerine atıfta bulunarak, tekerlekli bir konfigürasyonda hızlı bir şekilde hareket eden bir robotun karşısına çıkan bir engeli fark edince üstünden geçemeyerek çalışmasını durdurmak yerine, dört ayaklı yürüyebilen bir konfigürasyona geçerek engelin üstünden geçebilmesi ve yoluna tekrar hızlı bir şekilde tekerlekli konfigürasyonuna dönerek devam edebilmesi çok yönlülük ve fonksiyonellik açısından modüler robotik sistemlerin neler sunabileceğinin sadece küçük bir örneğini temsil etmektedir.
Modüler robotik sistemler geometrik yapılandırılmalarına göre üç farklı sınıfa ayrılabilirler. Zincir tipi (chain type) modüler robotik sistemler, ilk olarak ortaya çıkan sistemlerdir ve genellikle gelişmiş fonksiyonellikleriyle ön plana çıkarlar. Yeniden yapılanma konusunda zorluklar çekseler de gelişmiş ilerleme metodlarını sorunsuzca uygulayabildiklerinden önemli bir sınıftır. Daha sonra ortaya çıkan örgü tipi (lattice type) modüler robotik sistemlerse gelişmiş yeniden yapılanma yetenekleri ile zincir tipi sistemlerin bu yöndeki eksiklerini kapatsa da fonksiyonellik açısından yetersizdirler. Bu sınıf robotlar genelde basit ilerleme metodlarını uygulamadan öteye geçememişlerdir. Son olarak ortaya çıkan ve iki modüler robot tipinin önemli özelliklerini biraraya getiren hibrid tip ise oldukça ilgi çekicidir. Hibrid tip modüler robotlar gerektiğinde örgü dizilimine sahip olarak yeniden yapılanma problemini kolaylaştırırken, daha ileri seviye ilerleme metodlarını uygulamak için zincir dizilimine dönüşebilirler.
Bu çalışmada geliştirilmesi planlanan robotik yapının değişik ilerleme metodlarını kolayca uygulayabilmesi daha önemli olduğundan, modüler robotik sistemin zincir tipi olması gerektiğine karar verilmiştir. Zincir tipi modüler robotik sistemlerdeki genel yeniden yapılandırılma problemini aşmak için modüller toprak kurdu benzeri bir ilerleme metodunu uygulayarak pozisyon değiştirebilme yeteneğine sahip olacak şekilde tasarlanmışlardır.
Geliştirilecek stratejik planlama algoritmasının yapısına karar vermek için günümüze kadar geliştirilmiş robotik paradigmalara bir göz atmak gereklidir. Robotik alanında ortaya çıkan ilk paradigma, hiyerarşik paradigmadır. Bu kontrol mimarisinde robot sensörleri yardımıyla çevresinden gerekli bilgileri toplar, topladığı bilgilerle çevresinin dijital bir modelini oluşturur, bu modele göre yapması gereken eyleme karar verir ve uygular. Oldukça kolay uygulanabilir ve güvenilir bir sistem olmasına rağmen bu sistem robot tarafından tanınmayan ya da hızlı değişen dinamik ortamlarda oldukça düşük performans göstermektedir. Hiyerarşik paradigmanın dinamik ortamlarda yetersiz kalması üzerine geliştirilen reaktif paradigmada ise planlama gibi bir temel özellik bulunmaz. Reaktif paradigma ile çalışan bir robot sensörleri ile çevreden gerekli bilgileri alır ve bu bilgiler direk olarak eyleme çevrilir. Planlama ve modelleme gibi yüksek işlem gücü gerektiren özellikler sistemde bulunmadığından bu robotların dinamik ortamlarda tepki süreleri oldukça düşüktür. Bu sebeple engelden kaçma ve hedefe ulaşma gibi basit davranışları kolaylıkla yerine getirebilirler. Reaktif paradigmada planlama gibi temel bir özellik bulunmasa da sensörlerin eyleyicileri direk olarak kontrol ederek robotun sürekli tek bir eylem gerçekleşmesinin önüne davranışlar programlayarak geçilir. Davranışlar arasında

hiyerarşik bir sıralama yapıldığında robotun istenilen şekilde hareket etmesi sağlanabilir. Yine de reaktif paradigma uzun mesafeli amaçlara ulaşmak için programlanması gereken davranışların karmaşıklığı ve robot davranışlarını optimize etmenin zorluğu nedeniyle limitlerine ulaşmıştır. Bu nedenle hiyerarşik ve reaktif paradigmaların önemli özelliklerini barındıran hibrit bilinçli/reaktif paradigma ortaya çıkmıştır. Bu hibrit paradigma iki katmandan oluşur. Bilinçli katman hiyerarşik paradigmaya benzer bir yapıdayken, reaktif katman bir sıralayıcı eklenmesinin dışında reaktif paradigmanın aynısıdır. Bu hibrit paradigmaya göre robot önce sensörleri yardımıyla çevresel bilgileri toplar ve bilinçli katman tarafından amaca ulaşmak için alt amaçlardan oluşan uygulanabilir bir plan üretilir. Planın uygulanması kısmında ise reaktif katman devreye girer ve robotun hızlıca hedefe ulaşması sağlanır. Robot reaktif bir şekilde hedefine doğru ilerlerken, çevresel sensör verileri bilinçli katmanla paylaşılır ve gerekirse plan revize edilir. Robotun durumu ve plana uygunluğu reaktif katmana eklenen sıralayıcı tarafından kontrol edilir ve planın alt amaçları tamamlandıkça yeni hedefler sıralanır.
Hibrit bilinçli/reaktif paradigma bu çalışmanın amaçları doğrultusunda yaratılmak istenen stratejik planlama uygulaması için en uygun paradigmadır. Bu nedenle algoritmanın mimarisi hibrit bilinçli/reaktif paradigmaya uygun olarak geliştirilmiştir.
Modüler robotik sınıflar ve robotik paradigmalar dışında çalışmanın tamamlanması için önemli kararlardan biri de kullanılacak simülasyon ortamının belirlenmesidir. Modüler robotik bir sistemi simüle edebilmek için simülasyon programlarında aranacak en önemli özellikler programın simülasyon sırasında modüllerin mekanik olarak birleşip ayrılmalarını desteklemesi ve simülasyon sırasında birden fazla modülün farklı şekilde programlanmasını sağlayabilecek programlama esnekliği sunmasıdır. Bu amaçla bu iki önemli özelliğe de sahip olan Gazebo, Webots ve V-Rep isimli üç farklı simülasyon ortamı incelenmiş ve çalışma için en uygun simülasyon ortamının V-Rep olduğuna karar verilmiştir. Simülasyon yazılımı olarak V-Rep'in seçilmesindeki sebepler Webots gibi ticari bir yazılım olsa da eğitim amacıyla tam sürümünün kullanılmasının mümkün olması ve diğer iki programa göre kolay kullanıma ve oldukça düzenli ve detaylı bir dökümantasyona sahip olmasıdır. Özellikle Gazebo açık kaynaklı bir program olmasına rağmen dökümantasyon ve kullanım kolaylığı açısından yetersiz bir yazılım olarak değerlendirilmiştir.
Simülasyon ortamı da belirlendikten sonra, V-Rep içerisinde modüler robotik sistemin tek modülünün mekanik yapısı, eklemleri, sensörleri ve bağlantı noktaları kararlaştırılarak yapısal tasarımı tamamlanmıştır. Yapının tasarlanmasının ardından modülün kinematik analizi yapılarak uygun bir ilerleme metodu geliştirilmiştir. Geliştirilen bu ilerleme metoduna uygun olarak, modüllerin özellikle birleşme amacıyla kullanacakları bir pozisyon kontrol algoritması geliştirilmiş ve tek modül tasarımı tamamlanmıştır.
Modül tasarımlarının ardından, modüler robotik yapının değişik ilerleme metodları uygulamak ve özel yeteneklerini sergilemek için oluşturması gereken konfigürasyonların tasarımlarına geçilmiştir. Bu amaçla dört ayaklı ilerleme metodunu uygulayabilen ve yerdeki engellerin üzerinden geçebilen bir dört ayaklı konfigürasyon ve tekerlekli ilerleme metoduna sahip ve boyunu kısaltarak tünel benzeri engellerin altından geçebilen bir tekerlekli konfigürasyon tasarlanmıştır. İki konfigürasyon da altışar modülün birleşmesiyle oluşacak şekilde tasarlanmıştır. Konfigürasyonları oluşturmak için modüllerin birleşme şekillerini belirleyen bir rol dağıtım algoritması oluşturulmuş ve konfigürasyonları oluşturmak için modül bağlantılarının nasıl yapılacağı belirlenmiştir. Tek modül tasarımında olduğu gibi iki

konfigürasyon için de kinematik analiz metodu ile uygun ilerleme metodları ve özel yeteneklerini uygulama biçimleri geliştirilmiştir.

Modüler robotik sistem yapısal olarak tasarlanıp simülasyon ortamında oluşturulduktan sonra, stratejik planlama algoritmasının geliştirilmesine geçilmiştir. Daha önce bahsedildiği gibi hibrit bilinçli/reaktif paradigmaya uygun bir algoritma geliştirilmiştir. Algoritmanın hiyerarşik katmanı simülasyon başında bütün modüllerin görsel ve kuvvet sensörlerini kullanarak çevrede bulunan cisimleri algılaması ve zeminin sürtünme katsayısını belirlemesi ile robotik yapıyı başlangıç durumundan istenen amaç durumuna getirmek için alt amaçlardan oluşan bir plan üretir. Üretilen plandaki alt hedefler, bir pozisyona ulaşmak olabileceği gibi, bir konfigürasyondan başka bir konfigürasyona dönüşme ya da konfigürasyona özel yetenekleri uygulama olabilir.

Plan üretildikten sonra reaktif katman yine sensör verilerinden faydalanan sıralayıcı kısmının belirlediği geçici hedeflerle eylem kısmına geçer ve robotik yapının sırayla alt hedeflere ulaşmasını sağlar. Reaktif katmanın aktif olduğu ve robotik yapının eylem halinde olduğu süre içerisinde de sensör verileri hiyerarşik katman ile paylaşılır. Robotun bulunduğu dinamik ortamda herhangi bir değişiklik olması durumunda plan revize edilebilir. Revize dahi edilse eylem kısmına geçilmeden önce reaktif katmanın sıralayıcısı arada tampon görevi gördüğünden robot normal çalışmasına devam edebilir.

Tasarlanan modüler robotik yapının ve stratejik planlama algoritmasının test edilmesi amacıyla V-Rep içerisinde karmaşık bir test düzeneği oluşturulmuştur. Bu test düzeneğinde robotik yapının başlangıç durumu ile hedef durumu arasında engeller konulmuş ve robotun davranışı test edilmiştir.

Bu çalışmada yapılan testlerin sonuçları detaylı olarak paylaşılmış ve yorumlanmıştır. Test sonuçlarına göre modüler robotik yapıların iyi bir modül tasarımı ve stratejik planlama algoritması ile monolitik benzerlerine oranla çok daha fonksiyonel ve çok yönlü olduğu kanıtlanmıştır.

1. INTRODUCTION

As a relatively new research field, modular robotic structures have gained high interest among researchers due to its versatility and its suitability for mass production. Modular robotic structures are versatile because they can change morphology. With a good design, a modular robotic structure can practice quadruped, wheeled or limbless locomotion methods. They are also suitable for mass production because generally every module in a modular robotic structure is identical and they are interchangeable. Even if the structure does not have identical modules, the number of different type of modules does not harm the process of mass producing because the modules are often interchangeable.

In this study, a chain type modular robotic structure that consists of self-mobile modules which are able to create different configurations that can operate in various locomotion modes such as wheeled and quadruped is created. To overcome the general self-reconfiguration problem for the chain type modular robots, the modules are designed to have self-mobility. A hybrid deliberative/reactive strategic planning algorithm is developed to ensure the system takes the suitable configuration to pass over configuration specific obstacles and reach its goal. The system is designed and tested in a highly versatile simulation environment called V-Rep (Virtual Robot Experimentation Platform).

In the second chapter, a background for this study is given comprehensively. In the first section of the second chapter, the background of modular robotics and notable modular robotic systems are explained. In the second section of the second chapter, general robotic paradigms are explained to give an insight to the reader about the strategic planning algorithm developed in this study. In the third and the last section of the second chapter, a brief explanation about simulation environments and some well-known simulation programs such as Gazebo, Webots and V-Rep are presented. The reasons for using V-Rep in this study are also explained in this part.

The third section is about the developed modular robotic structure in this work. In the first section of the third chapter, the single module design is explained in detail. The design considerations, development of the locomotion method for a single module and control methods of the single module is explained in this section. In the second section the configurations that can be created by the robotic structure is presented. Their assembly, locomotion methods, control algorithms and configuration specific abilities to pass obstacles are explained. In the last section of the third chapter, the strategic planning algorithm developed for the robotic structure is explained. The strategic planning algorithm is developed to ensure that the structure reaches its target.

In the fourth chapter, simulation and test results regarding the performance of the modular robotic system in a complex test area which is created in the simulation software. The robotic system is tested based on the performance of the developed strategic planning algorithm. The test area consists of configuration specific obstacles between the initial state and the target state is created in the simulation software.

The fifth chapter is about the conclusions of this study. In this chapter the test results are discussed and the findings are reported. Some recommended additions to the modular robotic system design and strategic planning algorithm is also shared.

1.1 Purpose of Thesis

The purpose of this thesis is to develop a strategic planning algorithm and implement it to a modular robotic structure which can assemble various configurations to operate in different locomotion modes such as quadruped and wheeled. The strategic planning algorithm is expected to make the robotic structure reach a given goal state by assembling proper configurations to pass configuration specific obstacles.

2. BACKGROUND

In this section a background for the related work of this study is given. In Section 2.1, a brief history of the modular robotics research field and most notable robots designed in the sense of classification and progression were presented. In section 2.2, the robotic paradigms are presented and their operating principles are explained briefly. In section 2.3, some well-known simulation programs such as Gazebo, Webots and V-Rep are presented.

2.1 Modular Robotics

The idea of distributed robotic systems emerged in the 1980s. It supposed that instead of building monolithic and inflexible robots, developing a cellular design inspired by nature is more efficient in reaching versatile robot structures. The robots would be able to change their shapes by splitting their cellular modules and rearranging them in a different configuration. One example given by Toshio Fukuda who is also known to be the creator of the philosophical foundation for the field of modular robotics was a robot that could move into environments that are difficult to reach and once inside, it can change its shape to accomplish a task.

The first implementation of the presented idea was completed by Toshio Fukuda. CEBOT [1] was built for that purpose in 1988. CEBOT was consisting of three different types of modules which were actuation modules, structural modules and tool modules. Since the definition of self-reconfigurable was not clear when CEBOT was developed it was specified as a multi-robot system consisting of mobile robots.

The first modular robot aiming at self-reconfiguration problem was created by Mark Yim in 1993. PolyPod [2] was able to implement different gaits with the connection of different types of modules. Polypod was dynamically reconfigurable, but it was not able to demonstrate self-reconfiguration. PolyPod is known to be the predecessor of chain type modular robots.

In 1993 and 1994, the first examples of lattice type modular robots were introduced. In 1993 Metamorphic [3] was built by Gregory Chirikjian and in 1994 Satoshi Murata built Fracta [4]. Both robots had the ability to change their shape in two dimensions. In these robots the configuration of modules was forming a lattice which eases the problem of self-reconfiguration. That caused emergence of another class which is called lattice type modular robots.

In 1998, two new chain type modular robots arrived in the scene. CONRO [5] was built by Andres Castanõ and a new version of PolyPod which is called PolyBot [6] was developed by Mark Yim. Both robots were able to implement various locomotion methods, but self-reconfiguration was still an important issue for the chain type modular robots.

While chain type modular robots were still struggling to demonstrate self-reconfiguration, in 1998 two new lattice type modular robots achieved self-reconfiguration in three dimensions. 3D Fracta [7] which is the improved version of the Fracta robot was developed by Satoshi Murata and Molecule [8] was built by Keith Kotay and Daniella Rus.

Up to that point, two distinctive clasess of modular robots were present which were superior to their complementary classes in different ways. While lattice type robots were able to demonstrate self-reconfiguration in three dimensions, chain type robots could not achieve self-reconfiguration. Chain type robots were superior to their lattice type complements by their increased ability to implement advanced locomotion gaits although they were not self-reconfigurable.

After the distinction between two classes was clearly defined, another class of modular robots has emerged. M-TRAN [9] which was developed by Satoshi Murata in 1999 had the properties of both a chain type and a lattice type modular robot. This new class is called hybrid type modular robot as it merges the properties of both chain type and lattice type modular robots. The hybrid nature of M-TRAN came from its ability to exist in both lattice structure to achieve self-reconfiguration and chain structure to make locomotion problem easier.

ATRON, which is the second hybrid type of robot, was built in 2003 by Jorgensen at the University of Southern Denmark, Odense. The novel idea behind ATRON [10,11] was fascinating. ATRON modules had only one actuator and they showed that 3D self-reconfiguration can be achieved even with one actuator. This was made possible by arranging the rotational axis of each module perpendicular to each other.

Another hybrid type of modular robot was introduced in 2006 by Wei-Min Shen. SuperBot [12] had an extra degree of freedom compared to M-TRAN which had two actuators parallel to each other. In SuperBot, an extra actuator is added to control the orientation between these actuators. Similar to M-TRAN, SuperBot also had the ability to exist in both lattice and chain structures.

2.2 Robotic Paradigms

Robotic paradigms can be defined as the control architectures that characterize the behavioral cycle of robots. The paradigms can be described in two ways: (1) by the relationship between the three primitives of robotics which are sense, plan, act; (2) by the way sensory data is processed and distributed through the system. Robotic paradigms can be listed as hierarchical, reactive and hybrid deliberative/reactive.

2.2.1 Hierarchical paradigm

The hierarchical approach focuses mainly on the planning aspect of operation of a robot. The robot senses its environment, plans its next action based on the acquired data and then executes the appropriate action using its actuators. Before taking any action, the robot plans its next action from the knowledge it has gathered about its surroundings up to that point. Figure 2.1 shows the relationship between the robotic primitives in hierarchical paradigm.

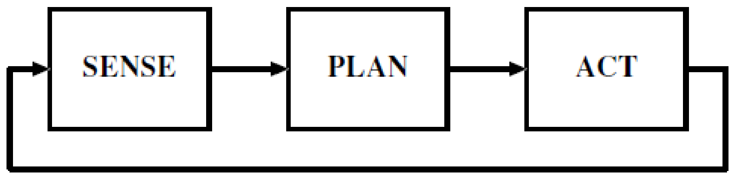


Figure 2.1 : Relationship of the robotic primitives in hierarchical paradigm.

The first robot operating under the hierarchical paradigm is "Shakey the robot" [17] which was developed at Stanford Research Institute in 1966. Control architecture of Shakey was composed of three basic parts which were sensing, planning and executing. The sensing system was translating camera image into an internal world model, the planner was using this world model to generate a plan to achieve the goal and the executor was applying control inputs according to the plan generated.

The components of the robot in this case are said to be horizontally organized. Information from the world in the form of sensor data has to filter through several intermediate stages of interpretation before finally becoming available for a response. The emphasis in these early systems was in constructing a detailed world model and then carefully planning out what steps to take next. The problem was that while the robot was constructing its model and planning what to do next, the world was likely to change. Therefore the robots exhibited the odd behavior that they would perceive, process and plan and then they would lurch into action for a couple of steps before beginning the cycle all over again. This is called look and lurch behavior. This behavior was a proof of the inability of these systems to cope with dynamic environments.

2.2.2 Reactive paradigm

The issues with the hierarchical paradigm caused the emergence of reactive or behavioral paradigm. In 1986 Rodney A. Brooks published an article which described a type of reactive architecture called the *subsumption* architecture [13]. This architecture became the dominant approach within the reactive robot architectures. Reactive paradigm was heavily used in robotics between 1988 and 1992. As shown in Figure 2.2, reactive paradigm removes the planning primitive from the architecture.

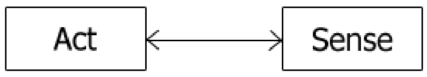


Figure 2.2 : Relationship of the robotic primitives in reactive paradigm.

In the reactive paradigm, the actions taken by the robot are direct results of sensor data acquired. Although this implies that the robot takes only one type of action, this is not the case. To avoid the robot taking only one action, layers of interacting finite state machines which connect sensor data to actuators are added. These finite state machines are called behaviors. Depending on the sensor data received, one or more behaviors can be activated simultaneously. To avoid the confliction between these activated behaviors, different handling mechanisms are developed. In the *subsumption* architecture there is a hardware implemented overriding mechanism that enables selection of higher level behaviors over low level behaviors. Figure 2.3 shows the levels of behaviors and their relationship with the sensing and acting primitives.

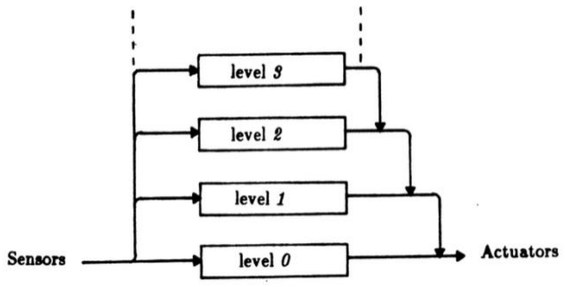


Figure 2.3 : Levels of behaviors in reactive paradigm.

2.2.3 Hybrid deliberative/reactive paradigm

In spite of the simple nature of the architecture and its adaptability to dynamic environments, reactive paradigm reached its limits due to the difficulties of composing behaviors for long range goals and optimizing robot behavior. These problems caused the return of the planning primitive in hybrid deliberative/reactive paradigm. The hybrid paradigm emerged in the 1990s and it is still the active area of research. Figure 2.4 shows the relationship of the robotic primitives in the hybrid paradigm.

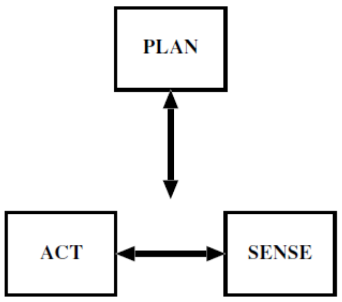


Figure 2.4 : Relationship of the robotic primitives in hybrid paradigm.

A robot working under the hybrid paradigm firstly plans how to accomplish a mission or a task using a global world model. For that purpose, the planner decomposes the task into subtasks and then activates the suitable behaviors to complete each subtask. The behaviors are executed same as the reactive paradigm and when the mission is completed, the planner generates another plan. The sensing organization in the hybrid paradigm is more complex. Sensor data can both be used by the behaviors and the planner. For example, an obstacle detected by a sensor which does not activate "avoid obstacles" behavior in the reactive paradigm can be used in the hybrid paradigm to create a map of the environment and can use this information when a new plan is generated. There can be also planner specific sensors which are not used by behaviors.

The hybrid architectures can be characterized by a layering of capabilities where low level layers provide reactive capabilities and high level layers provide the more computationally intensive capabilities. Three layered architectures are the most popular variant of these hybrid architectures. The layers on these architectures are; (1) controller/reactive layer, (2) sequencer/executive layer, (3) planner/deliberative layer.

The controller layer provides low level control and it is characterized by a tight sensor-action loop. Controller elements should have low computational complexity to allow them to react quickly to stimuli and execute basic behaviors fast. The sequencer layer is between the low level controller and the higher level planner layers. It accepts directives from the planner and sequences them for the reactive layer. The sequencer layer is also responsible for integrating sensor information into an internal state representation. The planner or deliberative layer contains the heaviest computational components and generates complex solutions tasks.

2.3 Robotic Simulation Environments

In this section, some well-known robotic simulation software programs are presented. The first two programs Gazebo and Webots are briefly explained. The simulation software used in this study, V-Rep, is explained in detail. Reasons for choosing V-Rep as the simulation software over other alternatives for this study is also explained in this section.

2.3.1 Gazebo

Gazebo [18] is an open source outdoor dynamics simulator. Development of Gazebo started as a part of the Player project [19] at the University of Southern California in

2002. The purpose was to develop a complementary dynamics simulator to the 2D simulator Stage. In 2012 Gazebo became an independent project under the Open Source Robotics Foundation. Features of Gazebo are listed below.

- Support for multiple physics engines such as ODE (Open Dynamics Engine), Bullet, Simbody and DART (Dynamic Animation and Robotics Toolkit).
- Advanced 3D graphics with OGRE (Object Oriented Graphics Rendering Engine).
- Plugin support for robot, sensor and environmental control.
- Wide variety of supported robot models such as PR2, Pioneer2 DX, iRobot Create and TurtleBot.

Gazebo is still under development and the developers announced that Windows support is work in progress.

2.3.2 Webots

Webots [20] is a commercial robot simulator which uses ODE (Open Dynamics Engine) library for dynamic simulations. Its development is started in 1996 at the Swiss Federal Institute of Technology.

Features of Webots

- ODE support for physics simulation.
- C, C++, Java, Python and MATLAB support for programming robots.
- Complete library of customizable sensors and actuators.
- Robot controllers can be transferred to real robots. Supported robots are Aibo, Lego Mindstorms, Khepera, Koala and Hemission.
- Support for controllable connector devices to simulate modular robotic structures.
- Able to record simulations in AVI or MPEG format.

2.3.3 V-Rep

V-Rep [21] is a general purpose robot simulator with integrated development environment providing the ability to model and simulate sensors, mechanisms, robots and whole systems. V-Rep is developed by Coppelia Robotics and its first official release was in 2010. By the developers of the platform, V-Rep is defined as "the Swiss army knife among robot simulators" due to its versatility and modular structure to cope with simulating complex robotic systems. V-Rep is used in a wide variety of application areas such as fast prototyping and verification, fast algorithm development, remote monitoring, hardware control, etc. V-Rep supports three different physics engines which are ODE, Bullet and Vortex.

A simulation scene in V-Rep consists of 3 central elements. These are (1) scene objects, (2) calculation modules and (3) control mechanisms. Scene objects are the main entities used to build the scene. Calculation modules are the functions that handle calculations in the simulation. Control mechanisms are simply the code provided by the user to control the simulated entities.

Scene objects in V-Rep and their brief explanations taken from V-Rep Manual [22] are given below.

- *Shape* is a rigid mesh that is composed of triangular faces.
- *Joint* is a joint or an actuator. Four types are supported: (1) revolute joints, (2) prismatic joints, (3) screws and (4) spherical joints.
- *Graph* is used to record and visualize simulation data.
- *Dummy* is a point with orientation. Dummies are multipurpose objects that can have many different applications.
- *Proximity sensor* detects objects in a geometrically exact fashion within its detection volume. V-Rep supports pyramid, cylinder, disk, cone and ray type proximity sensors.
- *Vision sensor* is a camera type sensor , reacting to light, colors and images.
- *Force sensor* is an object able to measure forces and torques that are applied to it. It also has the ability to break if a given threshold is overshot.
- *Mill* is a convex volume that can be used to perform cutting operations on shape objects.
- *Camera* is an object that allows seeing the simulation scene from various view points.

- *Light* is an object that allows illuminating the simulation scene.
- *Path* is an object that defines a path or trajectory in space. It can be used for various purposes, also as a customized joint or actuator.
- *Mirror* can reflect images/light, but can also operate as an auxiliary clipping pane.

Figure 2.5 shows the visual representations of scene objects in the scene view and hierarchy tree.

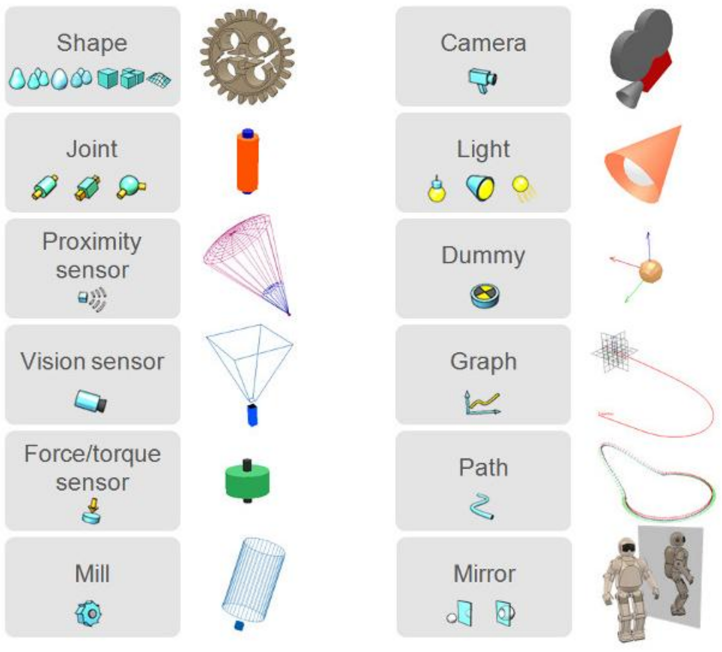


Figure 2.5 : Visual representations of scene objects in the scene view and scene hierarchy in V-Rep.

Some of the scene objects can have special properties to allow other objects or calculation modules to interact with them. These properties are given below.

- *Collidable* objects can be tested for collision against other collidable objects.
- *Measurable* objects can have the minimum distance between them and other measurable objects calculated.
- *Detectable* objects can be detected by proximity sensors.
- *Cuttable* objects can be cut by mills.
- *Renderable* objects can be seen or detected by vision sensors.

- *Viewable* objects can be looked through, looked at or their image content can be visualized in views.

Besides these properties, each object has a position and orientation within the scene.

Calculation modules are the functions that handle calculations in the simulation. These modules are used by the simulation software to update the simulation world, but they can also be used by the user. The following are the calculation modules and their brief explanations.

- Collision detection module allows tracking, recording and visualizing collisions that might occur between any collidable entities.
- Minimum distance calculation module allows tracking, recording and visualizing minimum distances between any measurable entities.
- Inverse kinematics calculation module allows solving any type of inverse or forward kinematic problem in a very efficient way.
- Geometric constraint solver module allows solving inverse or forward kinematic problems while offering a great extent of interaction possibilities to the user.
- Dynamics module allows dynamically simulating objects or models to achieve object interactions.
- Path planning module allows performing path planning calculations for objects in 2-6 dimensions. Additionally, non-holonomic path planning for car-type vehicles is also supported.
- Motion planning module allows performing motion planning calculations for manipulators.

Each calculation module (except the dynamics module) allows registering calculation objects that are user defined. Calculation objects are different from scene objects, but are indirectly linked to them by operating on them. This means that calculation objects cannot exist by themselves.

- Collision detection objects (or collision objects) rely on collidable objects.
- Minimum distance calculation objects (or distance objects) rely on measurable objects.
- Inverse kinematics calculation objects (or IK groups) rely mainly on dummies and kinematic chains, where joint objects play a central role.
- Geometric constraint solver objects (or mechanisms) rely mainly on dummies and kinematic chains, where joint objects play a central role.
- Path planning objects (or path planning tasks) rely mainly on dummies, a path object, and collidable or measurable entities.
- Motion planning objects (or motion planning tasks) rely mainly on IK groups, and collidable or measurable entities.

3. ROBOTIC STRUCTURE AND STRATEGIC PLANNING

3.1 Single Module Design

3.1.1 Design considerations

The main goal of the design process related to a single module of the reconfigurable robots is to achieve self-mobility of a single module in order to realize autonomous assembly of the robot. In addition to self-mobility of the individual modules, they have to be versatile enough to achieve locomotion when arranged in several configurations such as quadruped and wheeled.

3.1.2 Structure

A single module of the reconfigurable modular robot consists of three main parts. These parts can simply be named as wheel, foot and body. The design is very similar to an articulated (elbow) manipulator. While the articulated manipulator has its base fixed, in the module design, the base is not fixed and it is used as the wheel. The structure of a single module in the simulation environment is shown in Figures 3.1 and 3.2. Figure 3.3 illustrates the structure and terminology associated with the articulated manipulator.

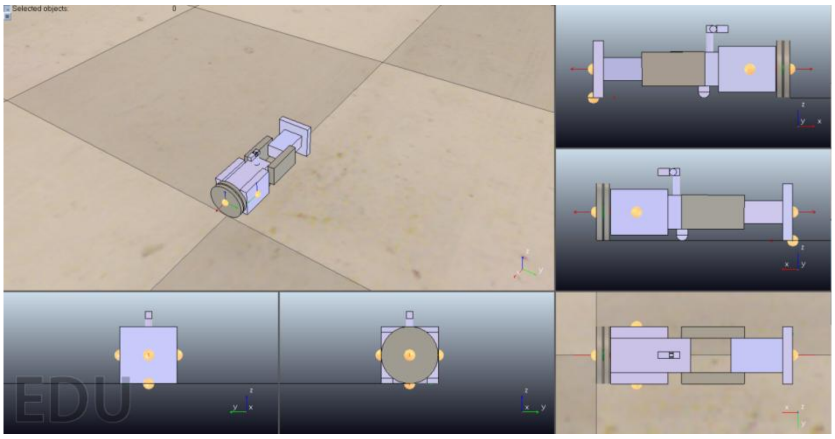


Figure 3.1: Single module in simulation software.

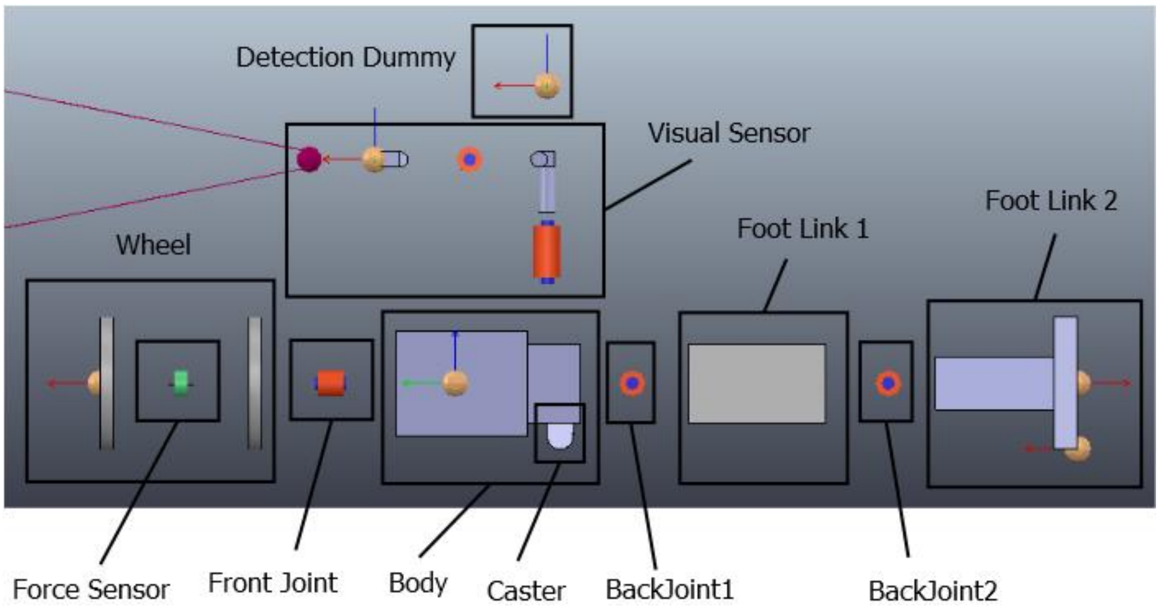


Figure 3.2: Single module parts shown in exploded view.

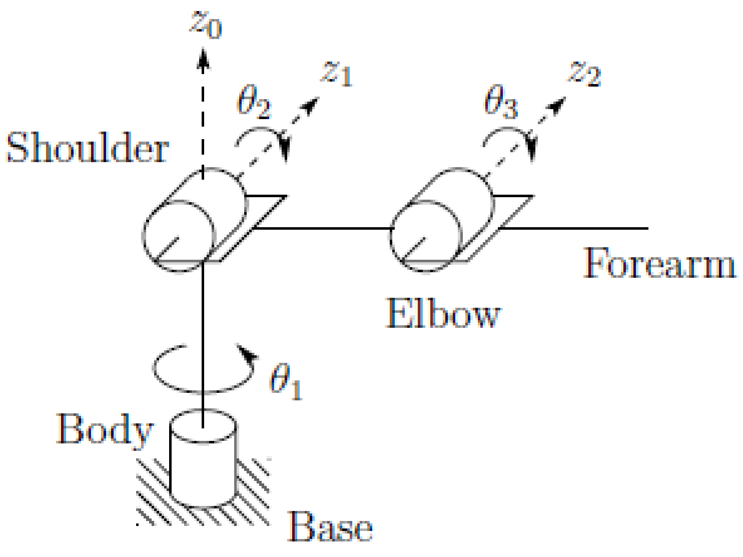


Figure 3.3: Articulated manipulator.

The wheel part is used for orientation control when the module does not have a role in a configuration and when it is not part of a larger kinematic chain. The wheel base consists of two cylinders which have a force sensor attached between them and a revolute joint which connects the cylinders to the main body of the module. The force sensor is used to estimate friction coefficient of the terrain.

The wheel has varying tasks in different configurations. For example, in the quadruped walking configuration, in which a single module takes the role of a leg and acts as an articulated manipulator, the wheel forms the base of the articulated manipulator. In the wheeled configurations, this part acts as the wheel and plays the main role in locomotion. In limbless locomotion mode, the periodic change in the joint position of the wheel makes it possible to achieve forward movement. In traveling wave locomotion mode, the joint positions of the wheel of some modules are used to control the orientation of the whole structure. The wheel has a connection point in its center.

The foot is the part that makes it possible for the module to move in longitudinal direction while operating alone. It consists of two rigid links which are connected with a revolute joint. The first link is connected to the body with a revolute joint and the second link has an orthogonal plate attached to it. This plate has an important role when it comes to the locomotion of the single module. It is essential that this part has a higher static friction coefficient than that of the cylinder of the wheel. The local reference frame is placed at the center of the lower tip of this plate because this position is the center of rotation of the module.

The foot has important functions in different locomotion modes. While the module is operating as a leg in the quadruped walking mode, the first and the second joints of the foot form the shoulder and elbow and the second link forms the forearm of the articulated manipulator. In the wheeled configurations, the joint positions of the foot can change the axle length of the structure. In limbless locomotion modes, forward movement is achieved by the periodic movement of the foot links.

The body is the uniting part of the module. The wheel and the foot are connected to the body with their revolute joints. A caster is attached to the lower side of the body. The caster acts as a pivot point in case the tip of the foot loses contact with the terrain. The pole-like structure on which the visual sensor is positioned is attached to the upper side of the body with a revolute joint. Two connection points are located on the left and right sides of the module.

3.1.3 Sensor implementations

3.1.3.1 Force sensor

Force sensors are used to estimate the friction coefficient between the robot external surface and the terrain. Since the decision algorithm for intelligent locomotion and reconfiguration rely highly on this estimation, the role of force sensor is important for cooperative behavior of the modules.

In the simulation software, the force sensors are needed to be positioned between two rigid shapes. To satisfy this requirement, the wheel part of the module is created by using two identical cylindrical shapes and the force sensor is positioned between these two cylinders. Figure 3.4 shows the positioning of the force sensor. The front cylinder of the wheel is exposed to friction force while the module is moving and it transfers this force to a certain location so that the force sensor can operate. Next, the readings of the force sensor can be used in the decision algorithm for intelligent locomotion.

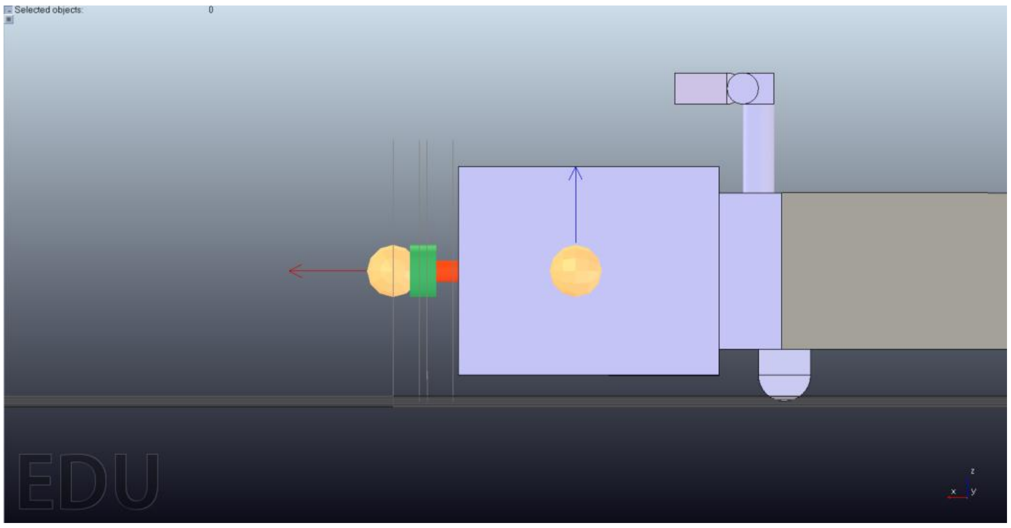


Figure 3.4: Force sensor.

3.1.3.2 Visual sensor

Visual sensors are the main unit of the modules for sensing the environment around them. They have uses both in the assembly phase in which the modules are operating alone and in the cooperative phase in which the modules have different roles in a configuration. The visual sensors are created using the proxy sensors in the simulation software and attached to the body of modules with a pole-like structure consisting of two links and two revolute joints.

In the assembly phase, one of the most important uses of the visual sensors is localization of other modules in the neighborhood. After the decision for the locomotion mode is made, the modules start scanning the area around them with the help of their visual sensors. After the scan, a basic coordinate system is created using the module or obstacle data acquired. During the assembly process, the modules continue scanning the environment to update their position on the coordinate system if they are not able to get locked to their target modules.

The creation of visual sensors in the simulation software can be considered as simulating a camera by using proxy sensors. The original proxy sensors in the simulation software are capable of identifying objects (shapes or dummies) that are part of a running simulation and returning axial distances based on its own coordinate frame. These capabilities of the proxy sensor make it possible to use it like a camera in the simulation environment.

Another important issue in the implementation process of the visual sensors is positioning. Since all mechanical parts of the module are facing orientation and position changes during locomotion, having the visual sensor positioned at a stationary point would not represent the real situation. Giving the visual sensor a relative independency regarding orientation and position control to adjust the visual angle makes the localization problem easier. Therefore the visual sensor is positioned at the end of a pole-like structure that has two links and two revolute joints and is attached to the body of the module. The first joint that connects the pole-like structure to the body controls gamma orientation and the second joint controls the beta orientation of the visual sensor. This control is very important in assembly phase in which the visual sensor of a moving module is required to lock a target module for connection.

3.1.4 Connection mechanism

The connection mechanism is the most important design element in the modular reconfigurable mobile robots because it enables the assembly of several configurations. In the simulation, connection of two modules is assumed to be completed by creation of a link between the connection points of modules. In each module there are four connection points. The connection points are positioned at the front side of the wheel, left and right sides of the body and at the back of the foot. Although the connection points on any module can create links between other connection points on another module, body-to-body and wheel-to-wheel connections are not used.

The connection points are created by using dummies in the simulation software. Dummies are not physical entities. They are points with orientation and can be seen as reference frames. Dummies have lots of uses in the simulation software, but their capability of creating links between other dummies makes them suitable to be used as connection points. There are several types of links between dummies which can be used for different purposes, but "Dynamics, overlap constraint" type of link is used for making the dummies behave as connection points. When created between two dummies, this type of link makes the dummies try to overlap their respective position and orientation. Therefore the parent shapes of the two dummies (e.g. the wheel part of a module and the foot part of another module), which are physical entities; act as if they are physically connected to each other.

Since "Dynamics, overlap constraint" type of link between two dummies make them overlap their positions and orientations, their position and orientation should be set accordingly to allow wheel-to-foot, wheel-to-body, foot-to-body and foot-to-foot connections. Table 3.1 shows the position (*x*, *y*, *z*) and orientation (*α*, *β*, *γ*) values for each dummy relative to reference frame of the module. Recall that the reference frame is positioned at the center of the lower tip of the foot. Figure 3.5, Figure 3.6, Figure 3.7 and Figure 3.8 show wheel-to-foot, wheel-to-body, foot-to-body and foot-to-foot connections between two modules and corresponding dummy orientations. Note that in the figures, the modules are not connected, but are about to connect and some parts of the module are not visible or shown as wireframe for better understanding. In all figures the first module is stationary at position *(0, 0, 0)* and orientation *(0°, 0°, 0°)* and the second module is the connecting module.

Table 3.1 : Dummy positions and orientations.

	Position (x, y, z)	Orientation (α, β, γ)
DummyFx	(0.172, 0, 0.025)	(0, 0, 0)
DummyLx	(0.137, 0.025, 0.025)	(0, 0, -90)
DummyRx	(0.137, -0.025, 0.025)	(0, 0, 90)
DummyBFx	(0, 0, 0.025)	(0, 0, 0)
DummyBSx	(0, 0, 0.025)	(0, 0, 180)

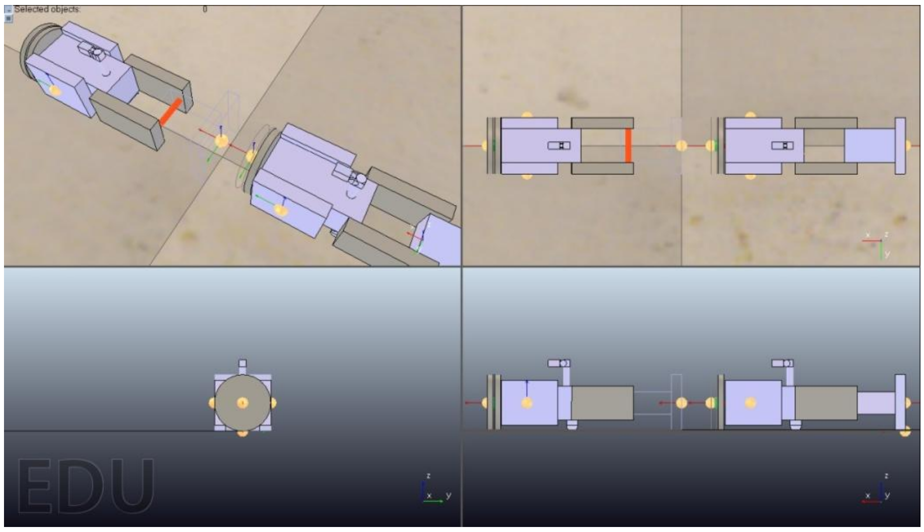


Figure 3.5: Wheel-to-foot connection.

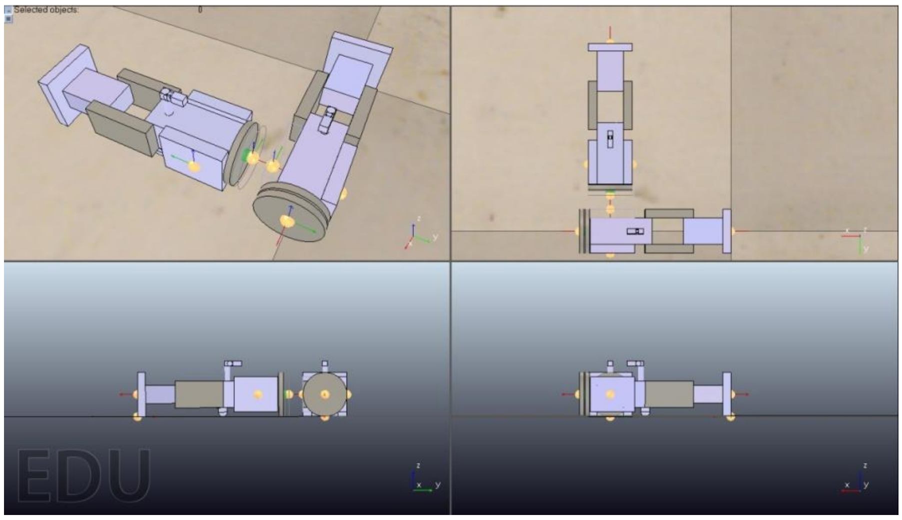


Figure 3.6: Wheel-to-body connection.

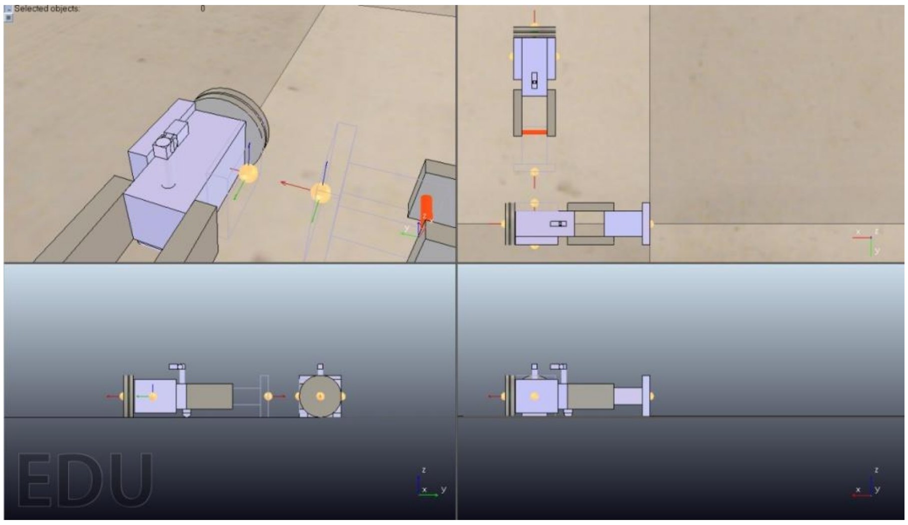


Figure 3.7: Foot-to-body connection.

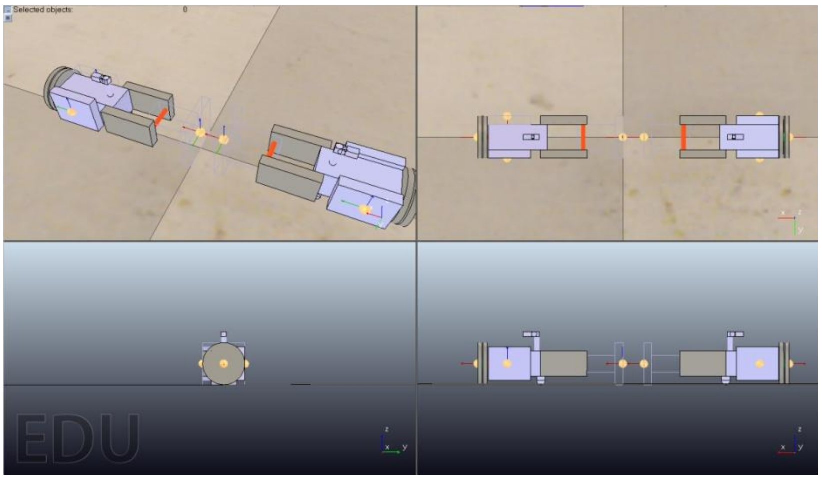


Figure 3.8: Foot-to-foot connection.

The reason for having two different dummies on the foot is the orientation differences between connecting dummies in different connection types. DummyBS is the first dummy created for the module to accomplish foot-to-body connections. When the module needed to make foot-to-foot connections, the existing dummies (DummyBS) at the foot of both modules have different orientations. While the orientation of DummyBS of the first module (DummyBSx1) is (0°, 0°, 180°), the orientation of the DummyBS of the second module (DummyBSx2) will be (0°, 0°, 0°). If it is left that way and the dummies are linked to each other, they will try to overlap their positions and orientations causing the modules to also overlap. To overcome this problem another dummy is added to the back connection point which is DummyBF. This dummy has the same position with DummyBS, but has -180^0 difference in gamma orientation. This difference makes the orientation of this dummy equal to the orientation of the DummyBS of the stationary module in foot-to-foot connections. Addition of DummyBF also solves the same orientation problem in wheel-to-foot connections. The first dummy is named as DummyBS because it was the standard for foot-to-body connections before there was not any need for DummyBF. The name Dummy BF is given to the second dummy because it is added to make foot-to-foot connections possible.

3.1.5 Locomotion

3.1.5.1 Kinematics of a single module

As expressed before the design of a single module is very similar to an articulated manipulator. Therefore the kinematic model of a single module is also similar to the kinematic model of an articulated manipulator. Figure 3.9 shows the coordinate frames assigned for using Denavit - Hartenberg convention and D-H parameters for the kinematic chain are shown in Table 3.2. Note that the base of the kinematic chain is the connection point of the wheel and the end-effector is the lower tip of the foot. d_1 is the distance between the connection point of the wheel and the first joint of the foot and it is 77.5 millimeters. d_2 is the distance between the two joints of the foot and it is 47.5 millimeters long. d_3 is the distance between the second joint of the foot and the outer center of the orthogonal plate and it is 47 millimeters. a_3 is the distance between the lower tip of the foot and the outer center of the orthogonal plate attached to the foot and its value is 25 millimeters.

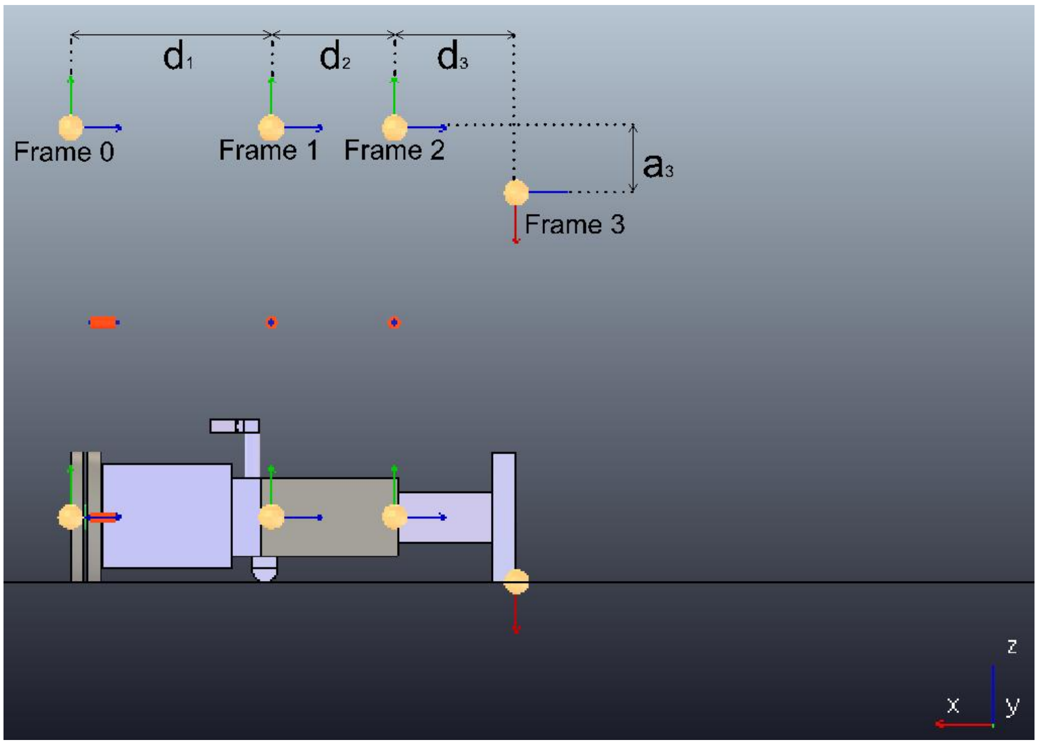


Figure 3.9: Coordinate frames for the first kinematic chain.

Table 3.2 : D-H parameters of the first kinematic chain.

	$Rot_z(\theta_i)$	$Trans_z(d_i)$	$Trans_x(a_i)$	$Rot_x(\alpha_i)$
1	θ_1	d_1	0	α_1
2	0	d_2	0	α_2
3	$-\pi/2$	d_3	a_3	0

Since the homogeneous transformation matrices are represented as a product of four basic transformations, any homogeneous transformation matrix associated to a link *i* can be expressed generally as shown in equation **(3.1)**.

$$\boldsymbol{A_i} = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

Using **(3.1)** and placing the corresponding D-H parameters associated to each link, the homogeneous transformation matrices can be derived. The derivation of the homogeneous transformation matrices associated to each link is shown in equations **(3.2)** to **(3.4)**.

$$\boldsymbol{A_1} = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1)\cos(\alpha_1) & \sin(\theta_1)\sin(\alpha_1) & 0 \\ \sin(\theta_1) & \cos(\theta_1)\cos(\alpha_1) & -\cos(\theta_1)\sin(\alpha_1) & 0 \\ 0 & \sin(\alpha_1) & \cos(\alpha_1) & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

$$\boldsymbol{A_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha_2) & -\sin(\alpha_2) & 0 \\ 0 & \sin(\alpha_2) & \cos(\alpha_2) & d_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

$$\boldsymbol{A_3} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & -a_3 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

The homogeneous transformation matrix that transforms the coordinates of the end effector (tip of the foot) to the base (connection point of the wheel) can be derived as shown in **(3.5)**.

$$\boldsymbol{T_3^0 = A_1 A_2 A_3} \tag{3.5}$$

Since the calculations are too complex, an m-file is created to calculate the position of the end-effector for different values of α_1 and α_2 in MATLAB. The m-file also transforms the coordinates of the end-effector from the coordinate frame of the base of the model to the absolute frame of V-Rep. The transformation is shown in equations **(3.6)** to **(3.8)**. The transformation assumes that the lower tip of the wheel is positioned at the origin of the absolute frame of V-Rep. Therefore the base of the kinematic chain (connection point of the wheel) is positioned at *(0, 0, 0.025)*.

$$x_{local} = -z_{model} \tag{3.6}$$

$$y_{local} = -x_{model} \tag{3.7}$$

$$z_{local} = y + 0.025 \tag{3.8}$$

The first kinematic model is useful in situations in which the wheel and the body of the module are assumed to be stationary. An example to this situation is the action taken by the module to achieve forward linear motion where the tip of the foot is needed to be positioned in a closer position to the body while the body and the wheel have to be stationary. That means that the first kinematic model is pretty helpful and informative for creating a forward locomotion method for the single module, but this does not mean that it is also useful for creating a backward locomotion method in which the wheel and the body of the module will not be stationary. Therefore to create a backward locomotion method, another kinematic model in which the tip of the foot has to be stationary while the wheel and the body of the module have to be moving has to be created.

Second kinematic model is mainly used for creating a gait for backward locomotion in which the tip of the foot is stationary and it pulls the wheel and body of the module. Although there is no joint in the original design of the module, one virtual joint is added to the tip of the foot in the kinematic chain. This joint helps adjusting the height of the end-effector which is generally contacting the terrain because of dynamic constraints while moving backwards. Basically the joint is not a part of the gait to be created, but it is helpful for design purposes. Therefore, the base of the kinematic chain is assumed to be the joint on the tip of the foot and the end effector is assumed to be the tip of the wheel. Also the position of the caster of the body has to be known for designing a backward locomotion gait, so it can be seen as an end-effector, too. Figures 3.10 and 3.11 show the coordinate frames assigned to each link and Table 3.3 shows the D-H parameters for the kinematic chain.

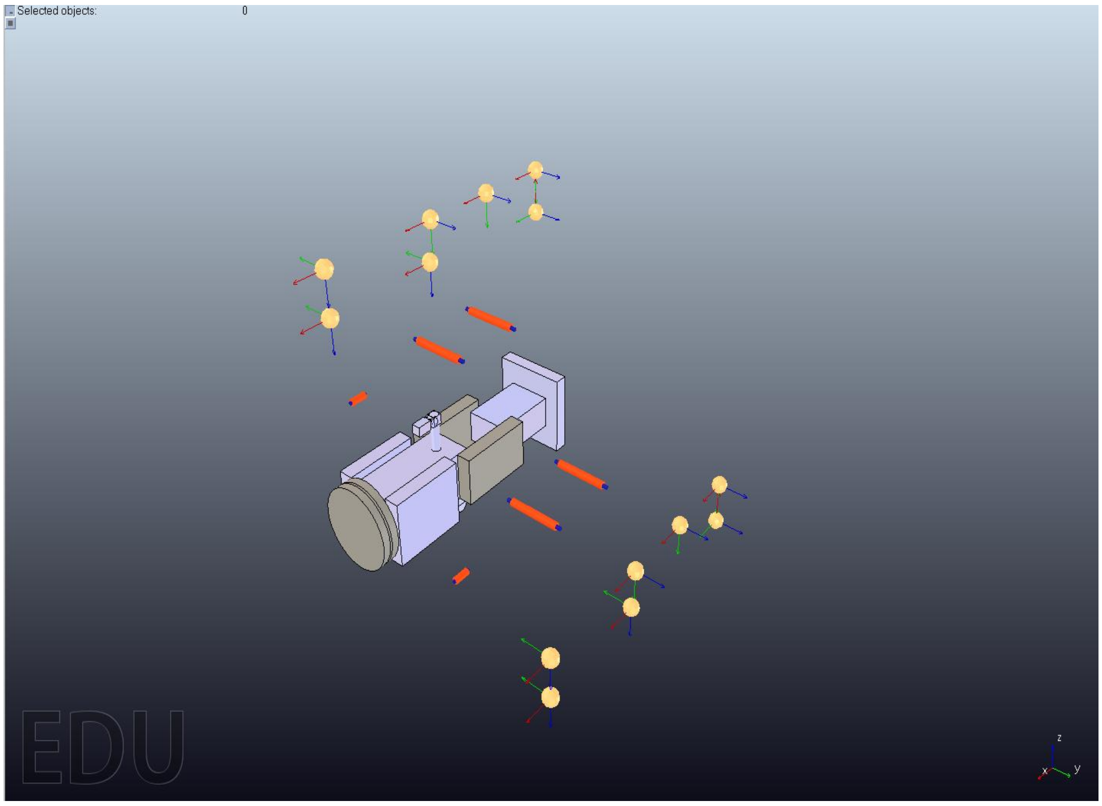


Figure 3.10: Coordinate frames for the second kinematic chain.

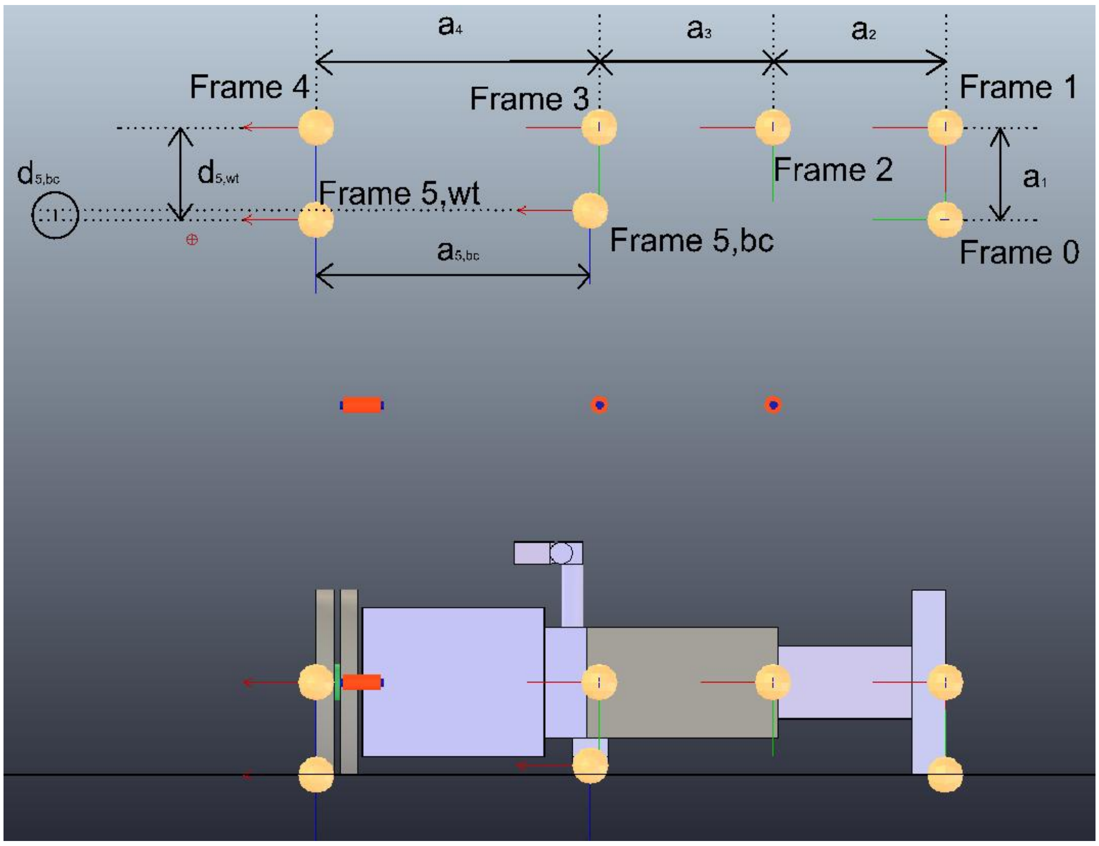


Figure 3.11: Coordinate frames for the second kinematic chain.

Table 3.3 : D-H parameters for the second kinematic chain.

	$Rot_z(\theta_i)$	$Trans_z(d_i)$	$Trans_x(a_i)$	$Rot_x(\alpha_i)$
1	θ_1	0	a_1	0
2	$\pi/2$	0	a_2	0
3	θ_3	0	a_3	0
4	θ_4	0	a_4	$-\pi/2$
5wt	0	d_{5wt}	0	0
5bc	0	d_{5bc}	a_{5bc}	0

Note that a_1 is the distance between the lower tip of the foot and the outer center of the orthogonal plate attached to the foot and its value is 25 millimeters. a_2 is the distance between the outer center of the orthogonal plate and the second joint of the foot and it is 47 millimeters. a_3 is the distance between the two joints of the foot and it is 47.5 millimeters long. a_4 is the distance between the outer center of the wheel and the first joint of the foot and it is 77.5 millimeters. d_{5wt} is the distance between the lower tip of the wheel and the outer center of the wheel and it is also 25 millimeters. d_{5bc} is the height of the center of the body caster and a_{5bc} is the longitudinal distance between the center of the body caster and the lower tip of the wheel. They are 5 millimeters and 75 millimeters long, respectively. Placing the D-H parameters to the general matrix form shown in **(3.1)** yields homogeneous transformation matrices associated to each link. Equations **(3.9)** to **(3.14)** show the derivation of these matrices.

$$\boldsymbol{A_1} = \begin{bmatrix} cos(\theta_1) & -sin(\theta_1) & 0 & a_1cos(\theta_1) \\ sin(\theta_1) & cos(\theta_1) & 0 & a_1sin(\theta_1) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

$$\boldsymbol{A_2} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & a_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

$$\boldsymbol{A_3 = \begin{bmatrix} cos(\theta_3) & -sin(\theta_3) & 0 & a_3cos(\theta_3) \\ sin(\theta_3) & cos(\theta_3) & 0 & a_3sin(\theta_3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \tag{3.11}$$

$$\boldsymbol{A_4 = \begin{bmatrix} cos(\theta_4) & 0 & -sin(\theta_4) & a_4cos(\theta_4) \\ sin(\theta_4) & 0 & cos(\theta_4) & a_4sin(\theta_4) \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \tag{3.12}$$

$$\boldsymbol{A_{5wt} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_{5wt} \\ 0 & 0 & 0 & 1 \end{bmatrix}} \tag{3.13}$$

$$\boldsymbol{A_{5bc} = \begin{bmatrix} 1 & 0 & 0 & a_{5bc} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_{5bc} \\ 0 & 0 & 0 & 1 \end{bmatrix}} \tag{3.14}$$

The homogeneous transformation matrices that transform the coordinates of the end effectors (tip of the wheel and caster of the body) to the base (the joint on the tip of the foot) can be derived as shown in equations **(3.15)** and **(3.16)**.

$$\boldsymbol{T_{5wt}^{0} = A_1A_2A_3A_4A_{5wt}} \tag{3.15}$$

$$\boldsymbol{T_{5bc}^{0} = A_1A_2A_3A_4A_{5wt}A_{5bc}} \tag{3.16}$$

As in the case with the first kinematic model, the calculations are too complex to handle without computer support. Another m-file is created to calculate the positions of the end-effectors for varying values of θ_2 and θ_3. It also calculates the suitable value for θ_1 using the trigonometric relation given in **(3.17)**. After the suitable θ_1 value is calculated, the program recalculates the positions of the end-effectors and then assuming the base of the kinematic chain is the origin, transforms it to the absolute frame of V-Rep using equations **(3.18)** to **(3.20)**.

$$\boldsymbol{\theta_1 = tan^{-1}(\frac{x_{model}}{y_{model}})} \tag{3.17}$$

$$\boldsymbol{x_{local} = y_{model}} \tag{3.18}$$

$$\boldsymbol{y_{local} = z_{model}} \tag{3.19}$$

$$\boldsymbol{z_{local} = x_{model}} \tag{3.20}$$

3.1.5.2 Locomotion methods and gait design

The module is expected to be moving forward and backward in the longitudinal direction. Using the kinematic models of the module, two different locomotion methods can be created. The locomotion methods are created by designing poses and switching the poses periodically, which in turn causes the foot to push or pull the rest of the module providing forward or backward locomotion.

The easiest way to achieve forward linear motion is to create propulsion by the use of the foot. This can be done by moving the tip of the foot to a forward position, then (with the help of friction) letting the foot push the rest of the module forward. Creating a periodic gait like this provides forward linear motion to a single module.

The periodic gait for forward movement can be created by repeated implementation of two different poses. The purpose of the first pose is to bring the tip of the foot to a forward position. The important point to take into consideration while having the module take this pose is to make sure the tip of the foot is not contacting the terrain until the final position is reached. The second pose is the same pose of a module at rest, but the importance of this pose is the action taken by the module to form this pose. The tip of the foot should be contacting the terrain while this pose is being taken so that the static friction force ensures that the tip of the foot is not moving and causing the module to move forward, i.e. to push forward. Figure 3.12 shows the motion sequence of described locomotion move.

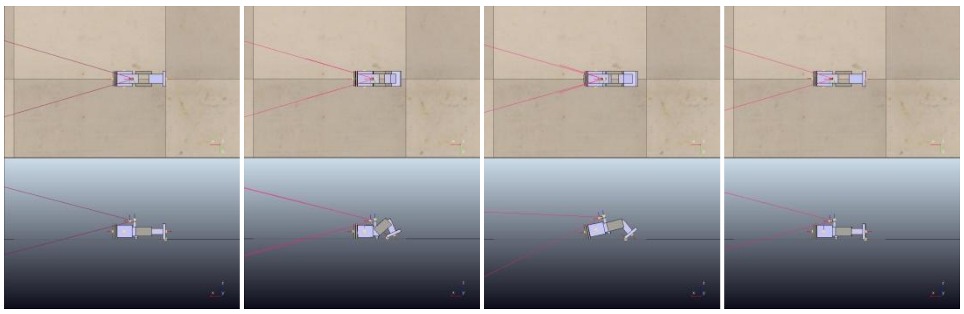


Figure 3.12: Forward locomotion of a single module.

From position control perspective, the module is expected to be moving as fast as possible while the distance from goal position is high. This velocity will be referred as "high velocity". Since accuracy is another issue for the module positioning, the gait is also expected to provide smaller linear displacements when the distance from goal position is lower than the displacement which the high velocity provides. Similarly this velocity will be referred as "low velocity". The high velocity and the low velocity can be adjusted by finding the suitable joint angles for the first pose of the gait.

To design the gait to provide the high velocity, the first pose of the gait should be designed to provide maximum displacement while satisfying the height condition. To determine the suitable joint positions for the first pose of the gait, a modified version of the m-file created for the first kinematic model is used. The modification makes the m-file a searching program which calculates the end-effector position for different α_1 and α_2 values and returns a table satisfying the height condition. As expressed before the end-effector should not contact the terrain while the module is taking the first pose, so the table consists of α_1 and α_2 values that make the end-effector have its z value on the coordinate system between 5 millimeters and 7.5 millimeters. The pseudo-code of the search program created is given in Table3.4. The code of the m-file can be found in Appendix-A.

Table 3.4 : Pseudo-code of the search program.

```
01 FOR($\alpha_1$ values from -$90^0$ to $90^0$)
02 ---FOR($\alpha_2$ values from -$120^0$ to $120^0$)
03 ------Create $A_1$, $A_2$, $A_3$ matrices
04 ------Calculate $T_3^0 = A_1*A_2*A_3$
05 ------//Pull coordinates from $T_3^0$ for kinematic frame then transform it to V-Rep absolute frame
06 ------$x_{model} = T_3^0(1,4)$
07 ------$y_{model} = T_3^0(2,4)$
```

```
      $z_{model} = T_3^0(3,4)$
      $x_{vrep} = -z_{model}$
      $y_{vrep} = -x_{model}$
      $z_{vrep} = y_{model} + 0.025$
      Calculate distance covered $x_d = 0.172 - x_{vrep}$
      IF($x_d > 0.072$ AND $0.005 < z_{vrep} < 0.0075$) THEN
         count += 1
         Table(count,1) = count
         Table(count,2) = $\alpha_1$
         Table(count,3) = $\alpha_2$
         Table(count,4) = $x_{model}$
         Table(count,5) = $z_{model}$
         Table(count,6) = $x_d$
      ENDIF
   ENDFOR
ENDFOR
Print Table
```

After running the m-file, the resulting table listing α_1 and α_2 values satisfying the height condition are given in Table 3.5.

Table 3.5 : α_1 and α_2 values for the first pose of the high velocity gait for forward locomotion.

	α_1	α_2	x (m)	z (m)	x_d (m)
1	-47	120	-0.0997	0.0075	0.0723
2	-46	120	-0.0994	0.0071	0.0726
3	-45	120	-0.0991	0.0067	0.0729
4	-44	119	-0.0997	0.0061	0.0723
5	-44	120	-0.0988	0.0063	0.0732
6	-43	119	-0.0994	0.0057	0.0726
7	-43	120	-0.0985	0.0060	0.0735
8	-42	118	-0.0999	0.0051	0.0721
9	-42	119	-0.0990	0.0054	0.0730
10	-42	120	-0.0981	0.0056	0.0739
11	-41	120	-0.0978	0.0053	0.0742

As it can be seen from the table, the maximum displacement is achieved when α_1 is (-41^0) and α_2 is (120^0). Therefore the first pose of the gait for high velocity is determined as shown in Figure 3.13.

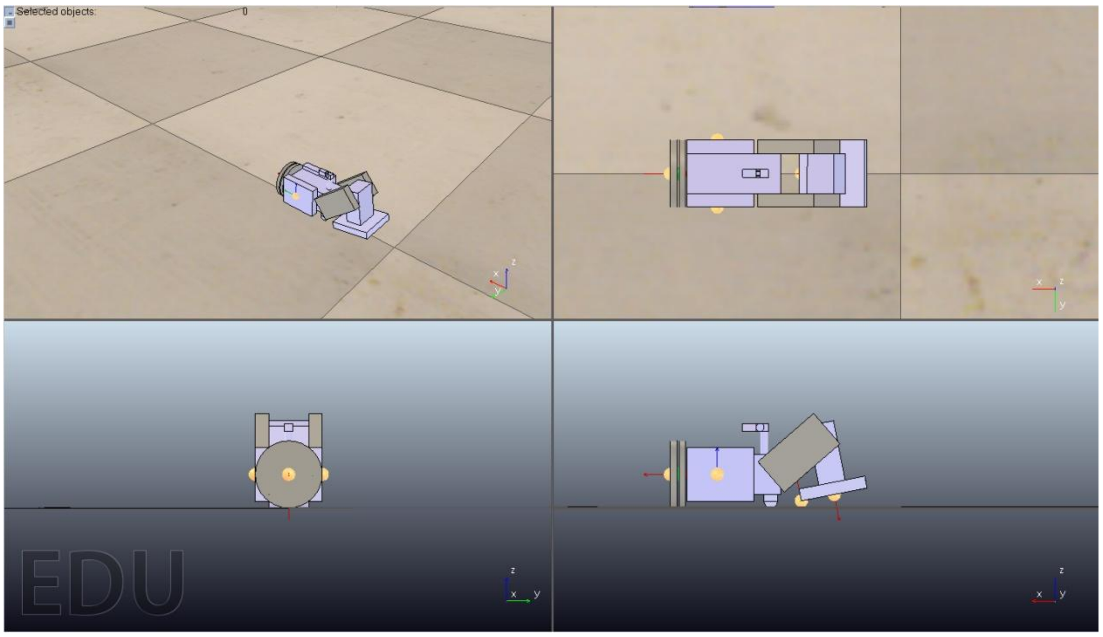


Figure 3.13: First pose of the high velocity gait of forward locomotion.

To get smaller displacements to determine the low velocity, the first pose of the gait can be redesigned and the distance traveled in a single step can be reduced to have less positioning errors. To redesign the first pose, the modified m-file used before can be run again by changing the distance traveled condition to be around 25 millimeters. As before, the m-file returns a table listing the α_1 and α_2 values for desired traveling distances. The α_1 and α_2 values from the resulting table are listed in Table 3.6.

Table 3.6 : α_1 and α_2 values for first pose of the low velocity gait for forward locomotion.

	α_1	α_2	x (m)	z (m)	x_d (m)
1	-36	62	-0.1472	0.0098	0.0248
2	-35	60	-0.1484	0.0097	0.0236
3	-35	61	-0.1477	0.0092	0.0243
4	-34	59	-0.1489	0.0090	0.0231
5	-34	60	-0.1482	0.0085	0.0238
6	-34	61	-0.1474	0.0079	0.0246
7	-33	59	-0.1486	0.0078	0.0234
8	-33	60	-0.1479	0.0073	0.0241
9	-33	61	-0.1471	0.0067	0.0249
10	-32	59	-0.1483	0.0066	0.0237
11	-32	60	-0.1475	0.0060	0.0245
12	-31	58	-0.1487	0.0059	0.0233
13	-31	59	-0.1480	0.0053	0.0240

Using the table, the suitable values for α_1 and α_2 are found to be (-33^0) and (61^0). Figure 3.14 shows the first pose of the low velocity gait for forward locomotion.

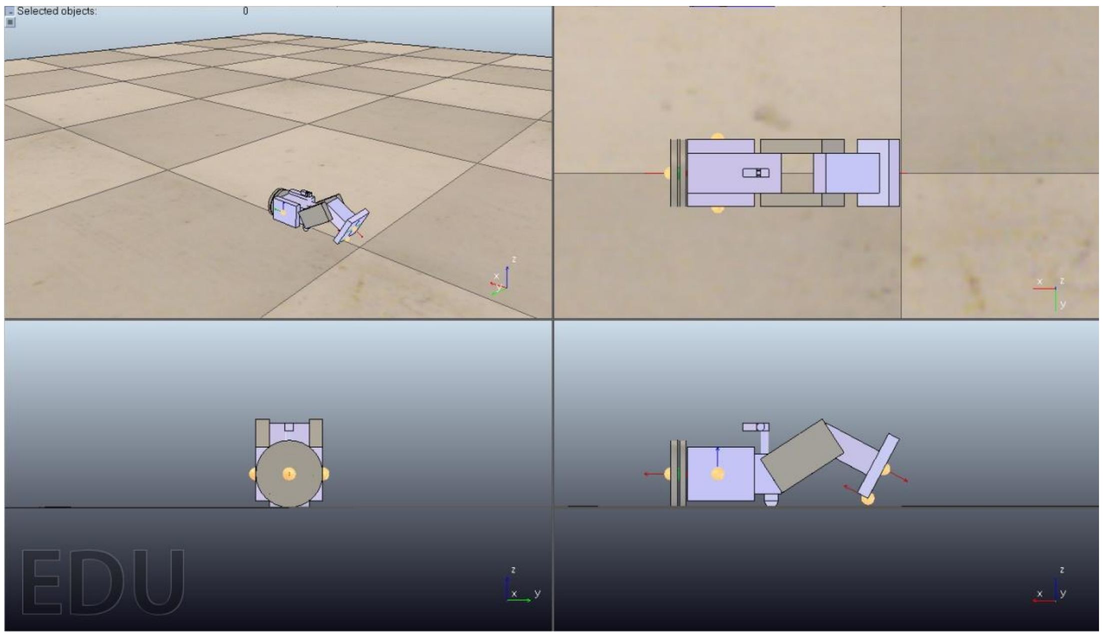


Figure 3.14: First pose of the low velocity gait of forward locomotion.

A single step of the high velocity and low velocity gaits for forward locomotion is completed after the foot returns to the reset position where α_1 and α_2 are (0^0).

The backward locomotion method design is similar to the design of the forward locomotion method. While the module is moving backwards, the foot of the module is expected to pull the wheel and the body parts. The difference between the two methods is that the forward locomotion can be achieved by two different poses because the last pose of the module is also the starting pose of the gait. In the backward locomotion method, this seems inapplicable. After the foot pulls the rest of the parts to a backward position, the tip of the foot has to lose contact with the terrain causing the caster on the body to be the pivot point. Only after losing contact with the terrain, the module can return to its starting pose. Therefore, in the backward locomotion method, the module needs to have two extra poses, making the total poses of the gait four in one step. The first extra pose, which is the second pose of the gait, is for making the caster contact the terrain and the second is a transition pose for the foot to ensure it is not contacting the terrain while returning to the reset position.

The design process of the poses of the backward locomotion gait is similar to that of the poses in the forward locomotion gait. The m-file created for the second kinematic model is modified to be a search program which calculates the end-effector positions for varying values of θ_3 and θ_4 and returning the values satisfying several conditions by creating a look-up table. Since the caster becomes the pivot point in the second pose, the displacement in one step of the gait is determined in this pose. Therefore, the gait design is based on the second pose of the gait. Similar to the forward locomotion gait design, two different value pairs will be determined for the high velocity and the low velocity for the design of the second pose.

To determine the θ_3 and θ_4 values for the second pose of the gait to achieve high velocity, several conditions have been set to narrow down the search. These conditions are;

- The distance covered by the wheel of the module should be at least 45 millimeters,
- The caster should be contacting the terrain. This means that z-value of the center of the cylinder that is used as the caster should be 5 millimeters high from the terrain at most.

Table 3.7 shows the pseudo-code of the search program created for the second kinematic model. After the conditions are added, the search program returns the table as shown in Table 3.8. The code of the m-file can be found in Appendix-B.

Table 3.7 : Pseudo-code of the search program created for the second kinematic model.

```
01 FOR($\alpha_1$ values from $-90^0$ to $90^0$)
02    FOR($\alpha_2$ values from $-120^0$ to $120^0$)
03       Create $A_1$, $A_2$, $A_3$, $A_4$, $A_{5wc}$, $A_{5bc}$ matrices
04       Calculate $T_{5wt}^0 = A_1*A_2*A_3*A_4*A_{5wt}$
05       Calculate $T_{5bc}^0 = A_1*A_2*A_3*A_4*A_{5wt}*A_{5bc}$
06       //Pull coordinates from $T_{5wt}^0$ for kinematic frame
07       $x_{model} = T_{5wt}^0(1,4)$
08       $y_{model} = T_{5wt}^0(2,4)$
09       $z_{model} = T_{5wt}^0(3,4)$
10       Calculate required $\theta_1 = atan(x_{model}/y_{model})$
11       Recalculate $A_1$
12       Recalculate $T_{5wt}^0 = A_1*A_2*A_3*A_4*A_{5wt}$
13       Recalculate $T_{5wt}^0 = A_1*A_2*A_3*A_4*A_{5wt}*A_{5bc}$
14       //Pull coordinates from $T_{5wt}^0$ and $T_{5bc}^0$ for kinematic frame then transform it
         to V-Rep absolute frame
15       $x_{model} = T_{5wt}^0(1,4)$
16       $y_{model} = T_{5wt}^0(2,4)$
17       $z_{model} = T_{5wt}^0(3,4)$
18       $x_{wt} = y_{model}$
19       $y_{wt} = z_{model}$
20       $z_{wt} = x_{model}$
```

```
21 ------$x_{model} = T_{5bc}^{0}$ (1,4)
22 ------$y_{model} = T_{5bc}^{0}$ (2,4)
23 ------$z_{model} = T_{5bc}^{0}$ (3,4)
24 ------$x_{bc} = y_{model}$
25 ------$y_{bc} = z_{model}$
26 ------$z_{bc} = x_{model}$
27 ------Calculate distance covered $x_d = 0.172 - x_{wt}$
28 ------IF($x_d > 0.045$ AND $z_{bc} < 0.005$)
29 ---------count += 1
30 ---------Table(count,1) = count
31 ---------Table(count,2) = $\theta_1$
32 ---------Table(count,3) = $\theta_3$
33 ---------Table(count,4) = $\theta_4$
34 ---------Table(count,5) = $x_d$
35 ---------Table(count,6) = $z_{bc}$
36 ------ENDIF
37 ---ENDFOR
38 ENDFOR
39 Print Table
```

Table 3.8 : θ_3 and θ_4 values for second pose of the high velocity gait for backward locomotion.

	$\theta_1(^0)$	$\theta_3(^0)$	$\theta_4(^0)$	x_{dif}(m)	z_{caster}(m)
1	-50.38	86	-36	0.0450	0.0045
2	-50.07	87	-38	0.0451	0.0036
3	-50.69	87	-37	0.0455	0.0041
4	-51.30	88	-36	0.0459	0.0046
5	-49.75	88	-40	0.0452	0.0027
6	-50.37	88	-39	0.0456	0.0032
7	-50.99	88	-38	0.0460	0.0037
8	-51.61	88	-37	0.0464	0.0042
9	-52.23	88	-36	0.0468	0.0047
10	-50.67	89	-40	0.0461	0.0028
11	-51.29	89	-39	0.0465	0.0033
12	-51.91	89	-38	0.0469	0.0038
13	-52.53	89	-37	0.0473	0.0043
14	-53.16	89	-36	0.0478	0.0048
15	-51.59	90	-40	0.0470	0.0029
16	-52.21	90	-39	0.0474	0.0034
17	-52.84	90	-38	0.0478	0.0039
18	-53.47	90	-37	0.0482	0.0044
19	-54.09	90	-36	0.0487	0.0049

While satisfying the conditions, the maximum displacement is achieved when θ_3 is (90^0) and θ_4 is (-36^0). Therefore, the second pose of the gait to achieve high velocity is determined as shown in Figure 3.16. The same pose can be redesigned to have low velocity using the same way by changing the first condition. When the desired displacement in the first condition is changed to be around 20 millimeters, the program returns the table as shown in Table 3.9.

Table 3.9 : θ_3 and θ_4 values for second pose of the low velocity gait for backward locomotion.

	$\theta_1(^0)$	$\theta_3(^0)$	$\theta_4(^0)$	x_{dif}(m)	z_{caster}(m)
1	-26.47	50	-24	0.0198	0.0044
2	-26.24	51	-26	0.0197	0.0034
3	-26.76	51	-25	0.0202	0.0040
4	-26.01	52	-28	0.0197	0.0024
5	-26.53	52	-27	0.0201	0.0030
6	-27.04	52	-26	0.0205	0.0036
7	-25.77	53	-30	0.0197	0.0014
8	-26.29	53	-29	0.0201	0.0020
9	-26.81	53	-28	0.0205	0.0026
10	-25.53	54	-32	0.0197	0.0004
11	-26.05	54	-31	0.0201	0.0010
12	-26.57	54	-30	0.0204	0.0016
13	-25.81	55	-33	0.0201	0.0000
14	-26.33	55	-32	0.0205	0.0006

It can be seen from the table that when θ_3 is (52^0) and θ_4 is (-27^0) the desired displacement of 20 millimeters is achieved.

After the second pose is determined for both high velocity and low velocity, the first and the third poses can be determined for these velocities. The strategy for designing these poses is simpler. The requirements for the first pose of the high velocity gait are;

- The distance covered by the end-effector should not exceed 45 millimeters,
- The caster should not be contacting the terrain,
- θ_3 and θ_4 values should be close to (90^0) and (-36^0), respectively.

Running the program with these conditions returns the table as shown in Table 3.10.

Table 3.10 : θ_3 and θ_4 values for first pose of the high velocity gait for backward locomotion.

	$\theta_1(^0)$	$\theta_3(^0)$	$\theta_4(^0)$	x_{dif}(m)	z_{caster}(m)	angle_dif(^{0})
1	-49.87	70	-12	0.0430	0.0156	44
2	-50.22	71	-13	0.0433	0.0151	42
3	-50.79	71	-12	0.0439	0.0157	43
4	-51.36	71	-11	0.0446	0.0162	44
5	-51.14	72	-13	0.0442	0.0152	41
6	-51.71	72	-12	0.0449	0.0158	42

As it can be seen from the table, the most suitable values are (72^0) for θ_3 and (-13^0) for θ_4. Figure 3.15 shows the first pose of the high velocity gait for backward locomotion. For the low velocity version of the first pose, the conditions are changed as;

- The distance covered by the end-effector should not exceed 20 millimeters,
- The caster should not be contacting the terrain,
- θ_3 and θ_4 values should be close to (52^0) and (-27^0), respectively.

The resulting table is shown at Table 3.11.

Table 3.11 : θ_3 and θ_4 values for first pose of the low velocity gait for backward locomotion.

	$\theta_1(^0)$	$\theta_3(^0)$	$\theta_4(^0)$	x_{dif}(m)	z_{caster}(m)	angle_dif(^{0})
1	-23.29	31	0	0.0182	0.0150	48
2	-23.77	31	-1	0.0188	0.0157	49
3	-24.07	32	0	0.0189	0.0153	47
4	-24.55	32	-1	0.0196	0.0160	48
5	-24.86	33	0	0.0197	0.0156	46
6	-25.17	34	-1	0.0198	0.0152	44

Using the table, the best setting for the first pose of the low velocity gait is found to be ($\theta_3=34^0$) and ($\theta_4=-1^0$).

Designing the third pose is simpler than designing the other poses. The third pose is used for placing the foot to a higher position from ground so that while returning to the reset position, the tip of the foot does not contact the terrain. For that purpose the third pose for the high velocity gait is set to be (θ_3=45^0) and (θ_4=-45^0). For the low velocity gait, the setting is (θ_3=15^0) and (θ_4=-15^0). Figures 3.15 to 3.17 show the poses for the high velocity gait while Figures 3.18 to 3.20 show the poses for the low velocity gait.

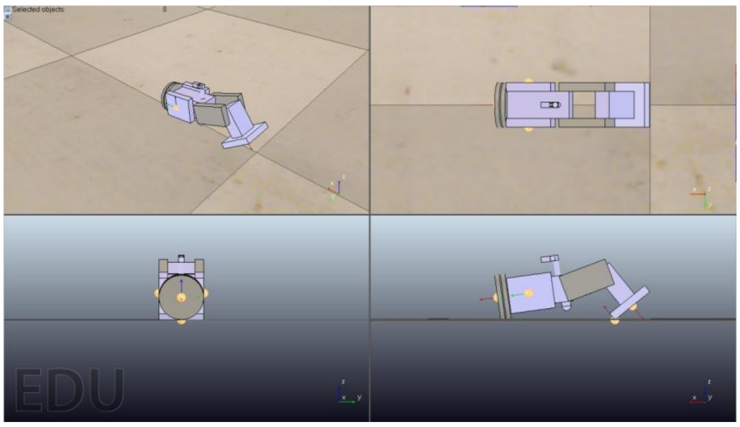


Figure 3.15: The first pose of the high velocity gait for backward locomotion.

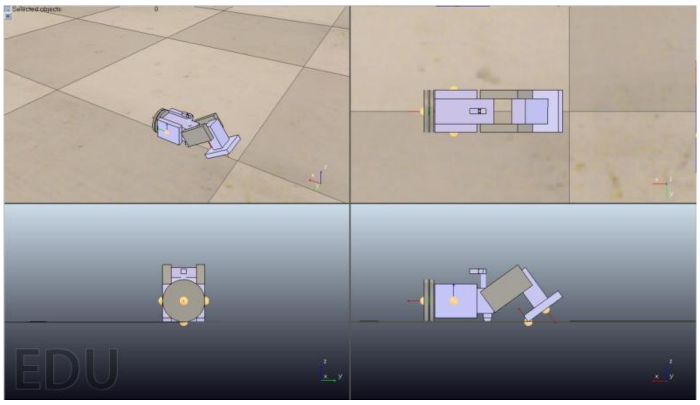


Figure 3.16: The second pose of the high velocity gait for backward locomotion.

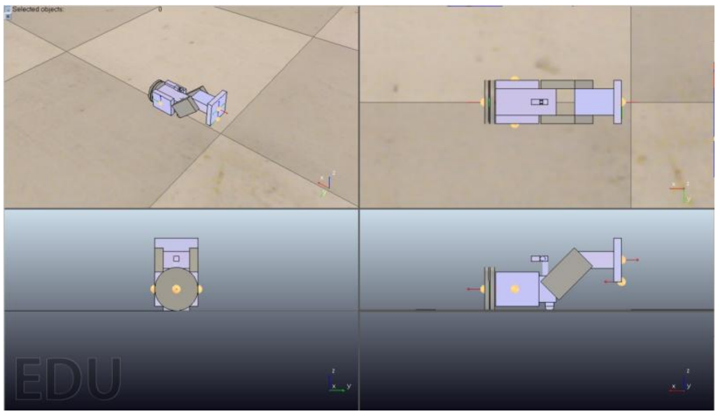


Figure 3.17: The third pose of the high velocity gait for backward locomotion.

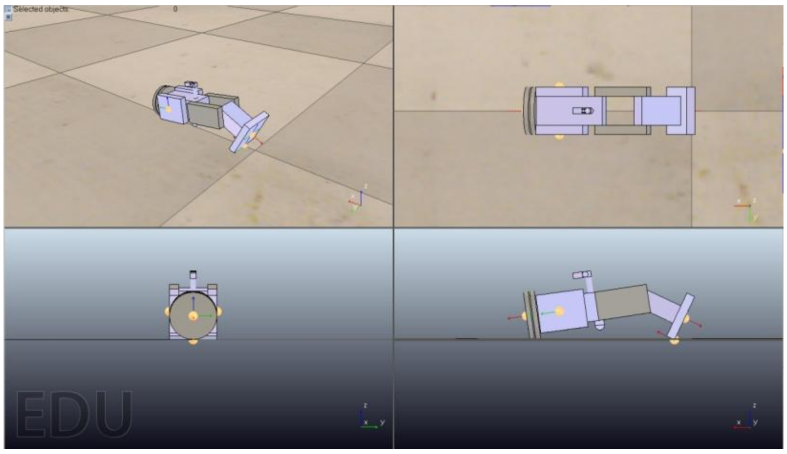


Figure 3.18: The first pose of the low velocity gait for backward locomotion.

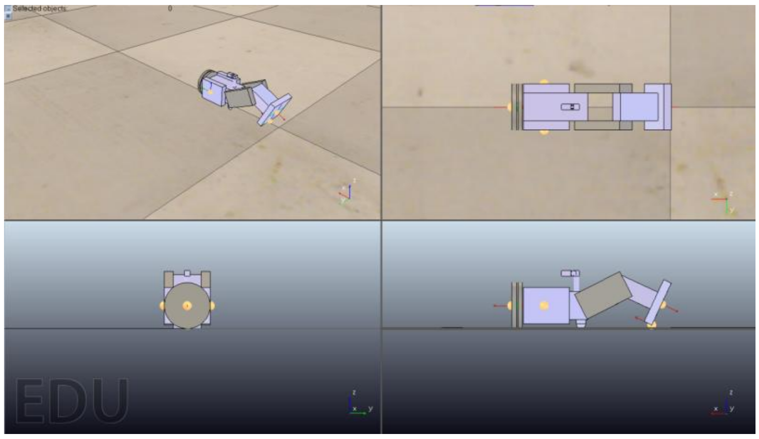


Figure 3.19: The second pose of the low velocity gait for backward locomotion.

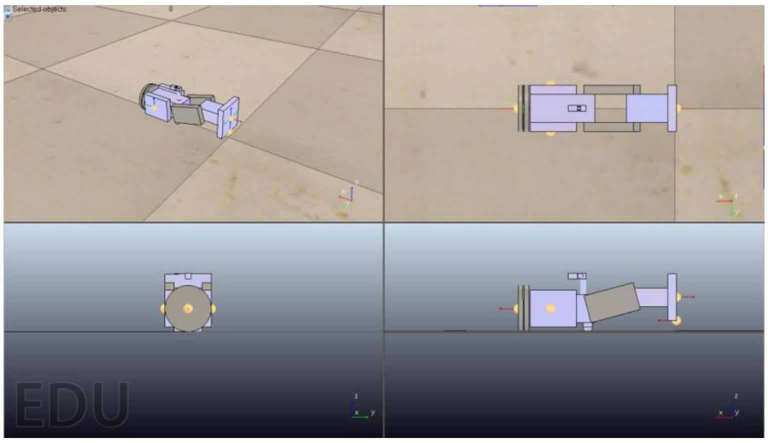


Figure 3.20: The third pose of the low velocity gait for backward locomotion.

Similar to the forward locomotion gaits, a single step of the high velocity and the low velocity gaits for backward locomotion is completed after the foot returns to the reset position where θ_3 and θ_4 values are (0^0).

3.1.5.3 Motions kinematics and position control

The motion kinematic model of the single module consists of the heading angle modifications controlled by the front wheel speed of the module and the forward or backward motion controlled by the push or pull effect of the foot. General motion kinematic model of the single module shown in Figure 3.21 can be desribed as given in equations (3.21) to (3.23). The reason for having difference equations to describe motion on longitudinal direction is that the motion is discrete. Note that γ_{mod} is the heading angle of the module and ω is the angular velocity. d is the longitudinal displacement caused by the selected gait.

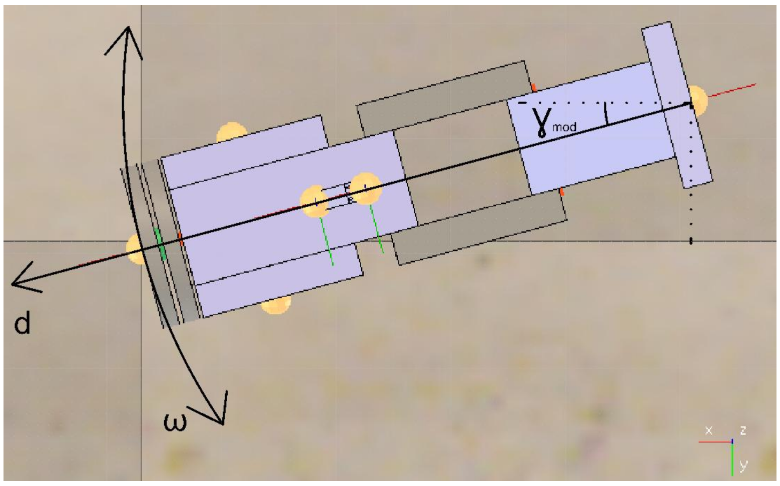


Figure 3.21: Longitudinal displacement and angular velocity of a module.

$$x_{mod}[k] = x_{mod}[k-1] + d\sin(\gamma_{mod}[k-1]) \qquad (3.21)$$

$$y_{mod}[k] = y_{mod}[k-1] + d\cos(\gamma_{mod}[k-1]) \qquad (3.22)$$

$$\gamma_{mod}[k] = \gamma_{mod}[k-1] + \omega \qquad (3.23)$$

The longitudinal displacement d depends on both the selected gait and the friction coefficient of the terrain. It is determined experimentally in the simulation environment for different gaits and terrains with varying friction coefficients.

Determining longitudinal displacement is simple. After each cycle of the gait is completed, x and y coordinates of the module is recorded and longitudinal displacement is calculated using (3.24). Since the calculated value varies in each cycle, the average of the recorded longitudinal displacements is calculated whenever a new sample is added for 60 seconds of simulation time. This procedure is applied on terrains which have friction coefficients varying from 0.2 to 1.0 for all four different gait and velocity couples the module can practice.

$$d = \sqrt{(x_{mod}[k] - x_{mod}[k-1])^2 + (y_{mod}[k] - y_{mod}[k-1])^2} \quad \textbf{(3.24)}$$

Table 3.12 shows average longitudinal displacement for forward locomotion gait on terrains with varying friction coefficients and the time it takes to complete one full cycle of the gait. Note that the subscripts "fhv" and "flv" stand for forward high velocity and forward low velocity.

Table 3.12 : Longitudinal displacement and cycle time of forward locomotion gait.

Terrain Friction Coefficient	High Velocity		Low Velocity	
	d_{fhv}(m/cycle)	t_{fhv}(s/cycle)	d_{flv}(m/cycle)	t_{flv}(s/cycle)
1.0	0.0635	0.95	0.0162	0.65
0.8	0.0652	0.95	0.0165	0.65
0.6	0.0662	0.95	0.0153	0.65
0.4	0.0663	0.95	0.0124	0.65
0.2	0.0502	0.95	0.0106	0.65

The average longitudinal displacement and the cycle time values for backward locomotion gait are given in Table 3.13. The subscripts "bhv" and "blv" stand for backward high velocity and backward low velocity similar to the forward locomotion case.

Table 3.13 : Longitudinal displacement and cycle time of backward locomotion gait.

Terrain Friction Coefficient	High Velocity		Low Velocity	
	d_{bhv}(m/cycle)	t_{bhv}(s/cycle)	d_{blv} (m/cycle)	t_{blv} (s/cycle)
1.0	0.0233	1.05	0.0123	0.80
0.8	0.0210	1.05	0.0113	0.80
0.6	0.0186	1.05	0.0104	0.80
0.4	0.0161	1.05	0.0091	0.80
0.2	0.0117	1.05	0.0075	0.80

Similar to longitudinal displacement, angular velocity ω is also dependent on terrain friction coefficient, but it is not needed to determine the actual ω value. To decide which gait to use, the module has to calculate the time it takes for it to adjust its heading angle. Therefore, determining t_{180} which is the time it takes for the module to change its orientation by ±180° on different terrains will be satisfactory for deciding the appropriate gait.

Since the module can rotate around its reference position, the orientation modification provided by the wheel is not used as a steering mechanic. Another reason for that is applying a high angular velocity to the wheel while the module is moving can cause it to tumble. For these reasons, the angular velocity of the wheel is kept at a low value for correcting small orientation errors while the module is moving. The low velocity is determined as 90°/sec and it is only applied when the orientation difference is less than 5°. To modify the heading angle while the module is not moving, a high angular velocity value is also determined. Since the maximum angular velocity that the wheel joint can provide is 360°/sec, the high angular velocity value is set as 300°/sec and it is used for adjusting the orientation of the module when the orientation error is greater than 5°.

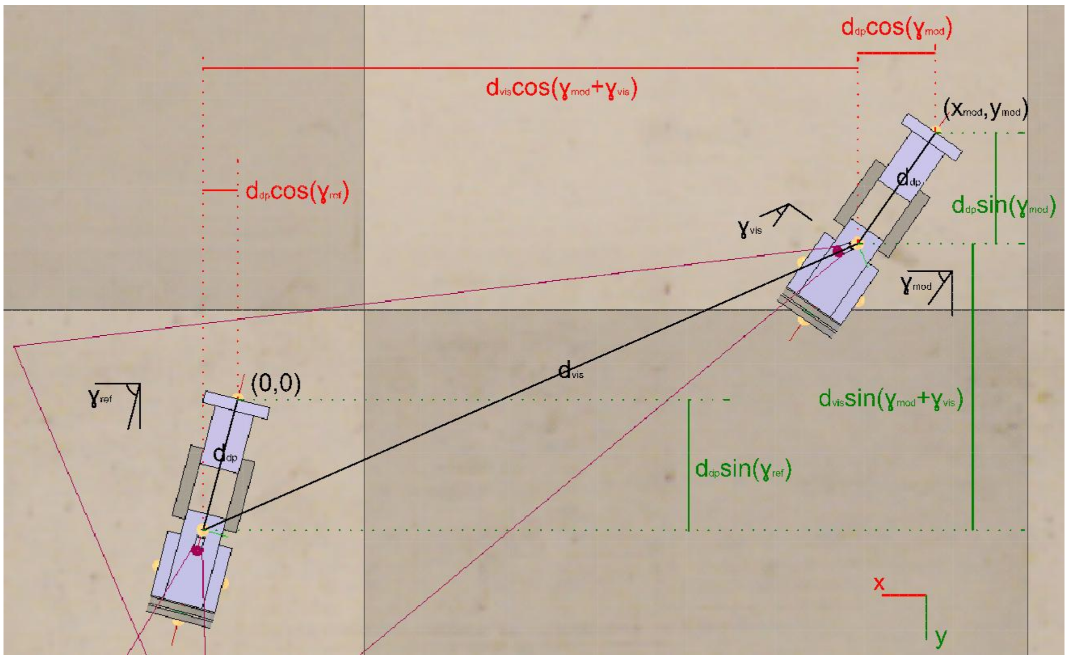
Since the low velocity is relatively small and its effect on the gait decision is neglectable, only the high angular velocity is used to determine t_{180}. Table 3.14 shows t_{180} in terrains with varying friction coefficients.

Table 3.14 : Time required for a 180^0 rotation in varying terrains

Terrain Friction Coefficient	t_{180}(s)
1.0	4.20
0.8	4.25
0.6	4.35
0.4	4.55
0.2	4.40

The modules move to assemble a configuration and as will be explained later, each module has a different role in a configuration and these roles also differ in the assembly phase. In the assembly phase, the modules are expected to be in a predetermined position having a predetermined orientation depending on their roles. For that purpose, a position control algorithm is developed for modules to position themselves according to their roles as fast as possible.

The module firstly scans the area to find the module which has the first role. This module is used as reference by all modules in the configuration and they calculate their target positions relative to this module. The module which has the first role acts as a leader or anchor point for the whole system. After the reference module is found, the module calculates its own position by using the data received from the visual sensor. The module also calculates its target position depending on the role it has. Equations (3.25) and (3.26) show the calculations of the initial module position using the reference module. Figure 3.22 is also given for better understanding.



$d_{dp}\cos(\gamma_{mod})$
$d_{vis}\cos(\gamma_{mod}+\gamma_{vis})$
(x_{mod}, y_{mod})
$d_{dp}\cos(\gamma_{ref})$
d_{dp}
$d_{dp}\sin(\gamma_{mod})$
γ_{vis}
γ_{mod}
γ_{ref}
(0,0)
d_{vis}
$d_{vis}\sin(\gamma_{mod}+\gamma_{vis})$
d_{dp}
$d_{dp}\sin(\gamma_{ref})$
x
y

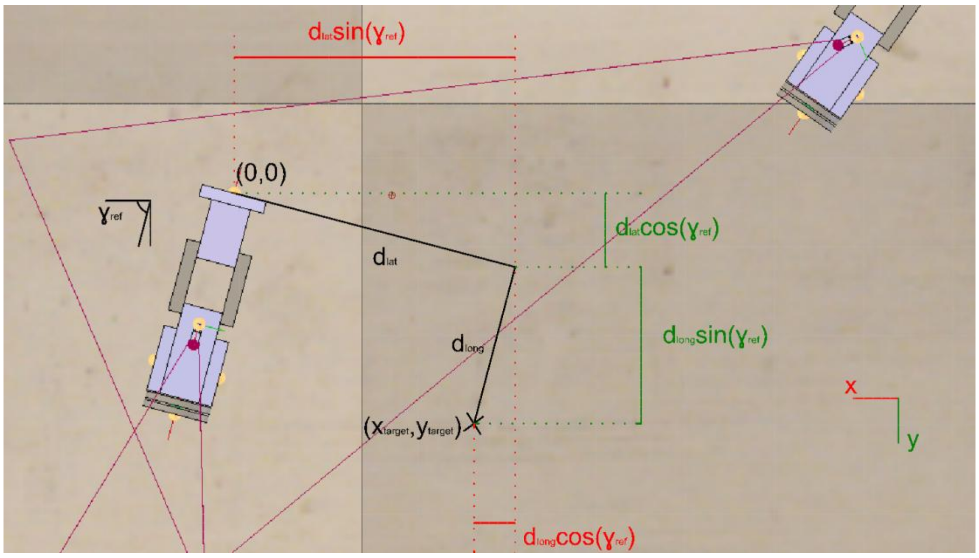
Figure 3.22: Calculating module position using the reference module.

$$x_{mod} = d_{dp}\cos(\gamma_{ref}) - d_{vis}\cos(\gamma_{mod} + \gamma_{vis}) - d_{dp}\cos(\gamma_{mod}) \qquad \textbf{(3.25)}$$

$$y_{mod} = d_{dp}\sin(\gamma_{ref}) - d_{vis}\sin(\gamma_{mod} + \gamma_{vis}) - d_{dp}\sin(\gamma_{mod}) \qquad \textbf{(3.26)}$$

In Figure 3.22 and equations (3.25) and (3.26) ddp stands for the distance between the detection dummy and the reference position of each module. ddp value is the same on each module and it is 102 millimeters. dvis is the distance data provided by the visual sensor. γmod, γvis and γref are the gamma orientations of the moving module, its visual sensor and the reference module, respectively.

The calculation of target position using the reference module is shown in Figure 3.23 and equations (3.27) and (3.28). Note that dlat stands for the required lateral distance of the target position relative to the reference module position. Similarly, dlong is the required longitudinal distance for the target position.



$d_{lat}sin(\gamma_{ref})$
(0,0)
γ_{ref}
d_{lat}
$d_{lat}cos(\gamma_{ref})$
d_{long}
$d_{long}sin(\gamma_{ref})$
x
y
(x_{target}, y_{target})
$d_{long}cos(\gamma_{ref})$

Figure 3.23: Calculating target position using the reference module.

$$x_{target} = -d_{lat}\,sin(\gamma_{ref}) + d_{long}\,cos(\gamma_{ref}) \qquad \textbf{(3.27)}$$

$$y_{target} = d_{lat}\,cos(\gamma_{ref}) + d_{long}\,sin(\gamma_{ref}) \qquad \textbf{(3.28)}$$

After the target position and the position of the module are calculated, the required orientation to reach the target and the difference between the heading angle of the module is calculated using the trigonometric relations given in equations (3.29) and (3.30).

$$\gamma_{req} = tan^{-1}\left(\frac{y_{target} - y_{mod}}{x_{target} - x_{mod}}\right) \qquad \textbf{(3.29)}$$

$$\gamma_{dif} = \gamma_{req} - \gamma_{mod} \qquad \textbf{(3.30)}$$

The distance between the module and the target is calculated using (3.31).

$$d_{target} = \sqrt{(x_{target} - x_{mod})^2 + (y_{target} - y_{mod})^2} \qquad \textbf{(3.31)}$$

After the orientation difference and the distance from target is calculated, the controller decides the direction of locomotion and the velocity. This is done by calculating the required time to reach the target in each option. In both options, the calculated time intervals can be separated as t_1, t_2 and t_3. Since the friction coefficient of the terrain is known, it is possible to calculate t_1, t_2 and t_3 values approximately because d and ω values are known.

t1 is the time it takes to have the required heading angle to reach the target. For forward locomotion option, $t_{1forward}$ is the time it takes to make the orientation difference 0° and for backward locomotion option, t1backward is the time needed to make the orientation difference ±180°. Equations (3.32) and (3.33) show the calculation of t_1.

$$t_{1forward} = \frac{\gamma_{dif}}{180°} t_{180} \tag{3.32}$$

$$t_{1backward} = \frac{\gamma_{dif} \pm 180°}{180°} t_{180} \tag{3.33}$$

t_2 is the time needed to reach the target position by implementing the appropriate gait. t_2 value is calculated by using the distance between the module and the target position and the suitable d values of high and low velocity of the forward or backward gaits. Equations (3.34) to (3.35) show the calculation of t_2 value for both gaits.

$$t_{2forward} = \left\lfloor \frac{d_{target}}{d_{fhv}} \right\rfloor t_{fhv} + \left\lfloor \left(d_{target} - \left\lfloor \frac{d_{target}}{d_{fhv}} \right\rfloor d_{fhv} \right) \frac{1}{d_{flv}} \right\rfloor t_{flv} \tag{3.34}$$

$$t_{2backward} = \left\lfloor \frac{d_{target}}{d_{bhv}} \right\rfloor t_{bhv} + \left\lfloor \left(\frac{d_{target}}{d_{bhv}} - \left\lfloor \frac{d_{target}}{d_{bhv}} \right\rfloor \right) * \frac{d_{bhv}}{d_{blv}} \right\rfloor t_{blv} \tag{3.35}$$

Lastly, t_3 is the time required for the module to take the desired heading angle after reaching the target position. It changes depending on the heading angle the module arrives at the target position. Assuming there will not be too much disturbance while moving to the target position, the arriving orientation will be equal to the calculated required heading angle in the initial position. Calculating the time to change the heading angle from the required orientation in the initial position and the desired heading angle in the target position can yield the approximate solution for t_3. The calculations are shown in (3.36) and (3.37).

$$t_{3forward} = \frac{\gamma_{des} - \gamma_{req}}{180°} t_{180} \quad \textbf{(3.36)}$$

$$t_{3backward} = \frac{\gamma_{des} - \gamma_{req} \pm 180°}{180°} t_{180} \quad \textbf{(3.37)}$$

When all time intervals are calculated for both locomotion methods, the siumlation option which takes less time is selected. After the appropriate gait is selected, the execution phase starts. Using the wheel joint, the module adjusts its heading angle equal to the required heading angle. When the orientation difference is 0°, the velocity to be applied is determined. Velocity is determined based on the distance from target position. If the distance is greater than d_{high} of the selected gait, then the velocity is set as high. If the distance is between d_{high} and d_{low}, then the distance is set as low and if the distance is lower than d_{low}, the module decides that it has reached the position and changes its heading angle to the desired heading angle. The pseudo-code of the position control algorithm of the single module is given in Table 3.15.

Table 3.15 : Pseudo-code of the single module position control algorithm.

```
01 WHILE (not reached to the target)
02 ---Calculate target position, module position
03 ---Calculate distance, required gamma orientation
04 ---WHILE(gait not decided)
05 ------Calculate $t_{1forward}$, $t_{2forward}$, $t_{3forward}$
06 ------Calculate $t_{1backwar}d$, $t_{2backward}$, $t_{3backward}$
07 ------Calculate $t_{forward}$, $t_{backward}$
08 ------IF(min($t_{forward}$,$t_{backward}$) == $t_{forward}$) THEN
09 ---------gait = forward
10 ------ENDIF
11 ------IF(min($t_{forward}$,$t_{backward}$) == $t_{backward}$) THEN
12 ---------gait = backward
13 ------ENDIF
14 ------gait decided
15 ---ENDWHILE
16 ---Calculate gamma difference $\gamma_{dif} = \gamma_{req} - \gamma_{mod}$
17 ---IF(gait == backward) THEN
18 ------Recalculate gamma difference $\gamma_{dif} = \gamma_{dif} \pm 180$
19 ---ENDIF
20 ---WHILE($|\gamma_{dif}|>5^0$)
21 ------set wheel speed = high
22 ------set gait velocity = 0
23 ---ENDWHILE
24 ---WHILE($|\gamma_{dif}|<5^0$)
25 ------IF($|\gamma_{dif}|>0.5^0$) THEN
```

```
            set wheel speed = low
        ENDIF
        IF($|\gamma_{dif}|<0.5^0$) THEN
            set wheel speed = 0
        ENDIF
        IF(gait==forward) THEN
            IF($d_{target}<d_{high}$) THEN
                set gait velocity = high
            ENDIF
            IF($d_{low}<d_{target}<d_{high}$) THEN
                set gait velocity = low
            ENDIF
            IF($d_{target}<d_{low}$) THEN
                target reached
                set gait velocity = 0
            ENDIF
        ENDIF
        IF(gait==backward) THEN
            IF($d_{targe}t<d_{high}$) THEN
                set gait velocity = high
            ENDIF
            IF($d_{low}<d_{target}<d_{high}$) THEN
                set gait velocity = low
            ENDIF
            IF($d_{target}<d_{low}$) THEN
                target reached
                set gait velocity = 0
            ENDIF
        ENDIF
    ENDWHILE
ENDWHILE
WHILE(not have desired orientation)
    Calculate gamma orientation difference
    IF($|\gamma_{dif}|>5^0$) THEN
        set wheel speed = high
    ENDIF
    IF($0.5<|\gamma_{dif}|<5^0$) THEN
        set wheel speed = low
    ENDIF
    IF($|\gamma_{dif}|<0.5^0$) THEN
        set wheel speed = 0
        have desired orientation
    ENDIF
ENDWHILE
```

3.2 Cooperative Locomotion Modes and Configurations

To complete more advanced tasks, the robotic structure can assemble different configurations such as quadruped or wheeled.

In this section the assembly, locomotion methods and specific abilities of quadruped and wheeled configurations are explained in detail.

3.2.1 Roles and communication in cooperative modes

The modules of the robotic structure can assemble different configurations. In configuration mode each module has a role that determines the way it will function before or after assembly. Role distribution is done after the system decides to assemble a configuration. The roles are distributed to modules based on their positioning. In assembly phase, a module determines its target assembly position based on its role in the configuration. After the assembly is done and the configuration is created, the module which has the first role (Role#1) becomes the master of the configuration and sends commands to other modules based on the requirements of the system.

Another important system in cooperative modes is the communication. The modules need to communicate with each other mainly in the scanning phase to localize themselves and initiate role distribution algorithm, in the assembly phase to notify other modules about their connection status and in the configuration phase for Role#1 to issue commands to other modules.

Communications in V-Rep is handled by variables called "Script Simulation Parameter". These variables are parts of scripts which can be read and written by other scripts that are running in the same simulation. Since every module has a script in V-Rep, they also share common script simulation parameters and use them to share data and notify other modules. The script simulation parameters can be read using "*simGetScriptSimulationParameter(Script Handle, "SSP Name")*" function. To write on parameters, the function "*simSetScriptSimulationParameter(Script Handle, "SSP Name", value)*" is used in V-Rep scripts.

3.2.2 Quadruped locomotion

The robotic structure can assemble a quadruped walker configuration by the connection of six modules.

3.2.2.1 Structure and assembly

In this configuration the body is formed by two modules which are connected to each other by their back connection points. The rest of the modules form the legs of the walker robot. The modules operating as legs connect their front connection points to the right or left connection points of the modules forming the body of the walker robot. Figure 3.24 shows the walker robot standing still.

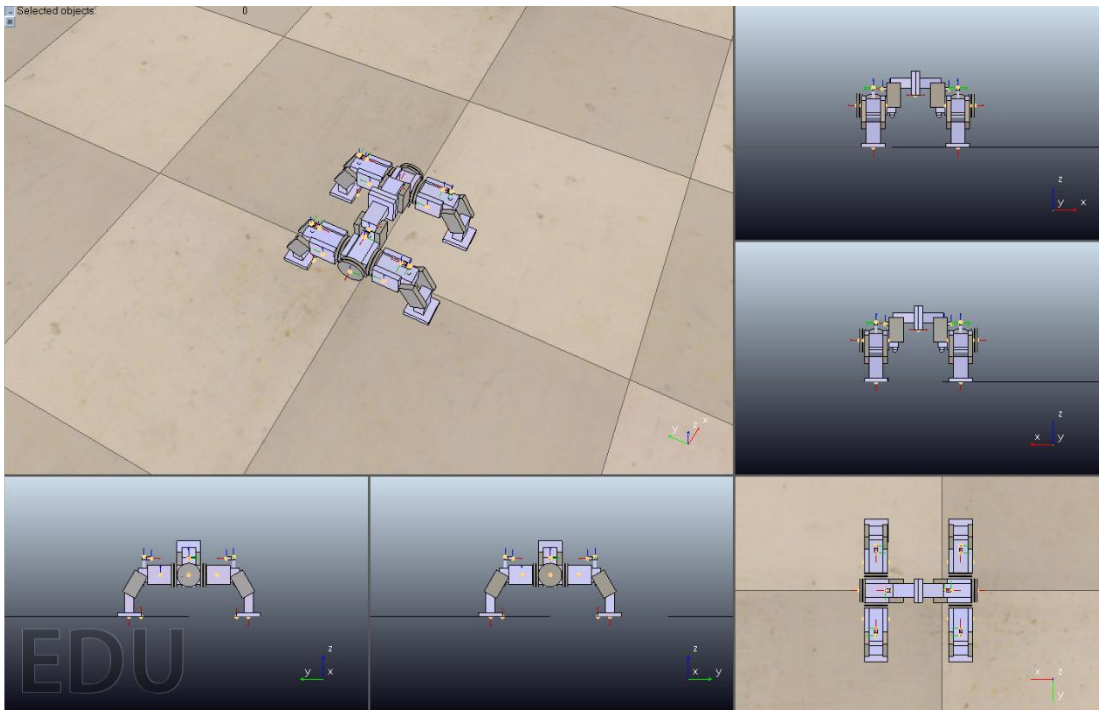


Figure 3.24: The walker robot standing still.

The modules forming the body of the configuration are Role#1 and Role#2. It is assumed that Role#1 points forward direction for the configuration. Role#2 points to the backward direction. The modules forming the front legs of the configuration are Role#3 and Role#4 and the modules forming the hind legs of the configuration are Role#5 and Role#6.

Role#3 is assumed to be the right front leg of the walker robot and this module connects to the right connection point of Role#1 with its wheel. Role#4 is assumed to be the left front leg of the configuration. The wheel of this module connects to the left connection point of Role#1.

The modules forming the right and left hind legs of the walker robot are Role#5 and Role#6, respectively. Since Role#2 points backward, the right hind leg connects to the left connection point and left hind leg connects to the right connection point of Role#2.

Figure 3.25 and Table 3.16 shows predetermined module positions for the assembly of the quadruped configuration.

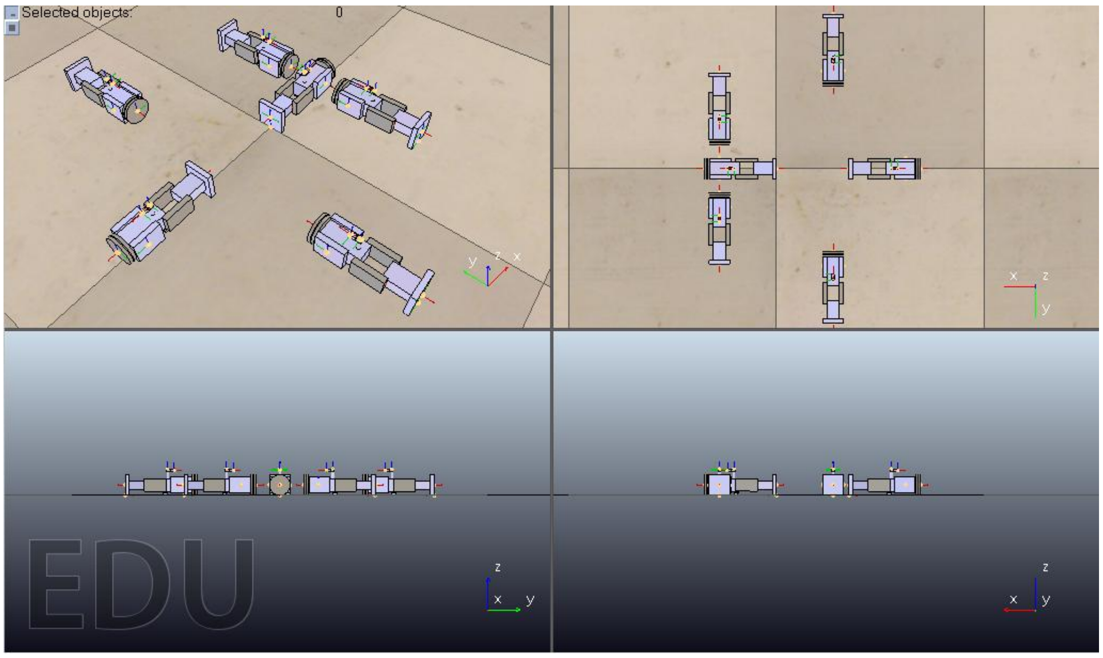


Figure 3.25: Predetermined module positions for quadruped configuration assembly.

Table 3.16 : Predetermined module positions for quadruped configuration assembly.

Role	Part	Position(x, y, z)	Orientation(α, β, γ)
1	Body Front	(0, 0, 0)	(0, 0, 0)
2	Body Back	(-0.175, 0, 0)	(0, 0, 180)
3	Front Right Leg	(0.137, -0.230, 0)	(0, 0, 90)
4	Front Left Leg	(0.137, 0.23, 0)	(0, 0, -90)
5	Back Right Leg	(-0.137, -0.372, 0)	(0, 0, 90)
6	Back Left Leg	(-0.137, 0.372, 0)	(0, 0, -90)

3.2.2.2 Leg kinematics and gait design

The walker robot is expected to move on both longitudinal and lateral directions. Since the movement characteristics of legs differ in these different locomotion styles, different gaits should be applied for both locomotion styles. These gaits are called trotting and sidling. Trotting is used for moving in longitudinal direction and sidling is used for moving in lateral direction. Although the locomotion methods in longitudinal and lateral directions are said to have different gaits, in truth their timetables are same. The difference is the variation of the leg positions defined for each of them.

Timetable used in both locomotion methods belongs to trot gait. The trot gait is generally used for low-speed walking. It is commonly seen to be used by quadruped animals like horses or dogs in nature. The diagonal legs act together in this gait. The timetable of the gait is shown in Table 3.17.

Table 3.17 : Timetable of trotting and sidling gaits.

Step	Front Right	Front Left	Back Right	Back Left
0	0	0	0	0
1	0	x	x	0
2	-1	1	1	-1
3	x	0	0	x
4	1	-1	-1	1

In the timetable, "x" means that the leg is not contacting the terrain, "1" and "-1" are the forward and backward positions. "0" is the reset position of the leg. After the fourth pose, the legs return to pose one and the gait becomes periodic. Changing the positions of states "1" and "-1" in the timetable makes the robot walk backwards.

Unlike trotting, sidling is not a common walking gait. Crabs, which are not quadrupeds, generally use this gait. It is also named crabbing for this reason. The diagonal legs act together like trotting. The timetable implemented to the walker robot is the same timetable used in trotting. The only difference is "1" and "-1" represents right or left depending on the desired side of movement. If the robot is to sidle to its right, "1" means right and "-1" means left and vice-versa for sidling to its left.

Since the desired leg positions are defined vaguely by the timetable, by modifying the first kinematic model created for the single module, the joint positions can be determined for realizing the walking-gaits and clearly defining the leg positions for each gait. The modifications of the kinematic model are for changing the end-effectors. For trotting, the end-effectors need to be the right and left tips of the legs.

Similarly for sidling, the end effectors need to be the upper and lower tips of the leg. The base of the model is still the connection point of the wheel. The need for having two different end-effectors for each gait is that the foot of the module which acts as a leg contacts the terrain on different tips while swinging forward or backward while the robot is moving in longitudinal or lateral directions. The left tip of the foot contacts the terrain while swinging forward and the right tip contacts the terrain while swinging backwards on a module which acts as a right sided leg of the walking robot and vice-versa for a module acting as a left sided leg of the robot while the robot is trotting. When the robot is sidling towards a lateral direction, the lower tip of the foot contacts the terrain while swinging forward and the upper tip contacts the terrain while swinging backwards for the legs on that side of the robot.

The kinematic model used for designing the foot positions is the modified version of the first kinematic model created for the single module. The modification is done for extending the end-effectors from only lower tip to all four tips of the foot. Figure 3.26 and Figure 3.27 show the coordinate frames associated to a right sided leg of the quadruped walker and Table 3.18 shows the corresponding D-H parameters for each joint.

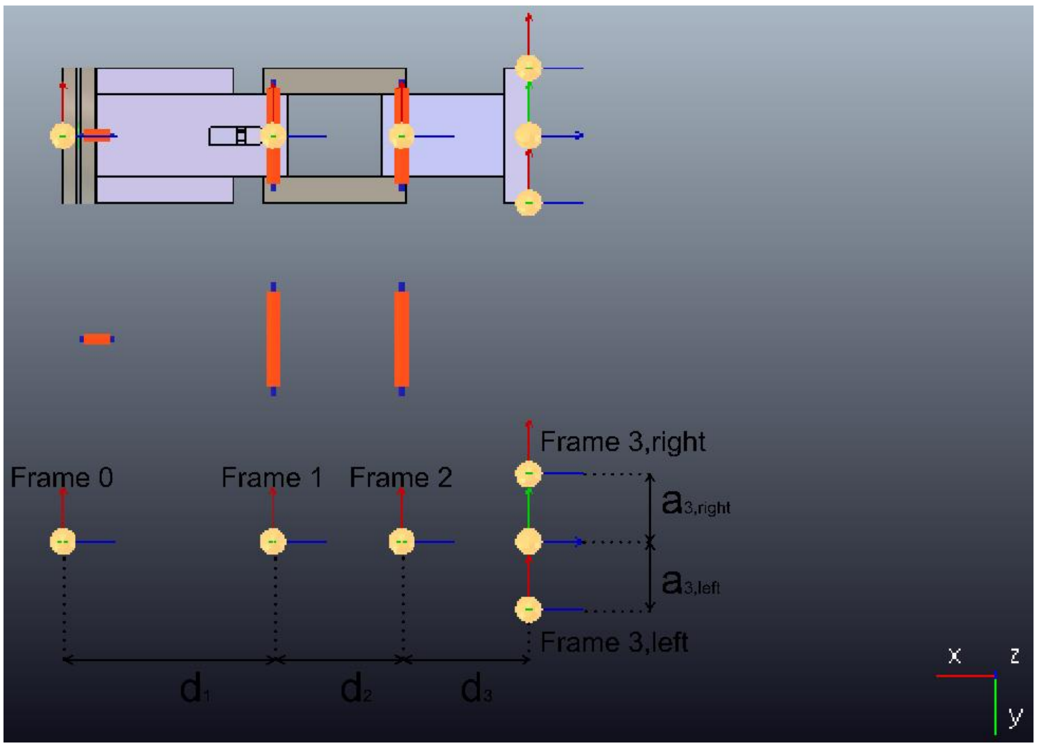


Figure 3.26: Coordinate frames used in the kinematic model.

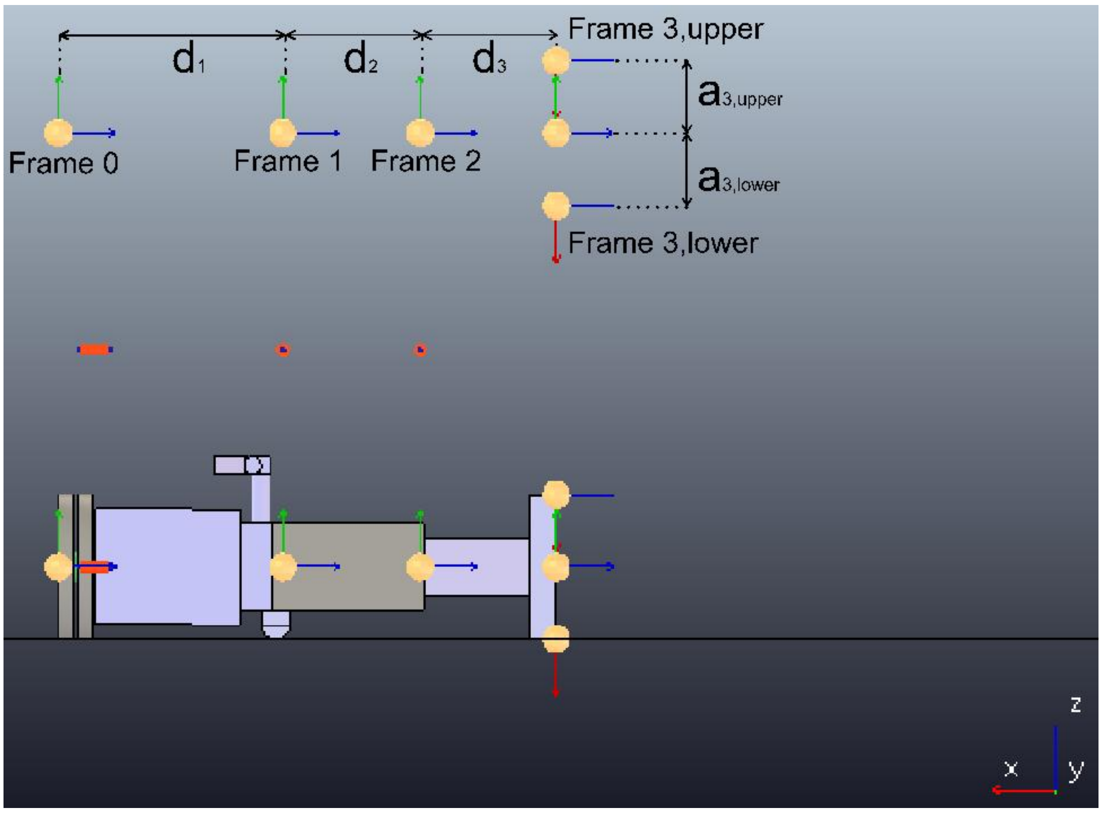


Figure 3.27: Coordinate frames used in the kinematic model.

Table 3.18 : D-H parameters of the kinematic chain

	$Rot_z(\theta_i)$	$Trans_z(d_i)$	$Trans_x(a_i)$	$Rot_x(\alpha_i)$
1	θ_1	d_1	0	α_1
2	0	d_2	0	α_2
3_{lower}	$-\pi/2$	d_3	a_{3lower}	0
3_{upper}	$-\pi/2$	d_3	a_{3upprt}	0
3_{right}	0	d_3	a_{3right}	0
3_{left}	0	d_3	a_{3left}	0

Note that θ_1 stands for the wheel position, α_1 and α_2 stand for the first and the second joints of the foot, respectively. d_1 is the distance between the base of the kinematic chain (connection point of the wheel) and the first joint of the foot which is 77.5 millimeters. d_2 is the distance between the two joints of the foot and its value is 47.5 millimeters. d_3 is the distance between the second joint of the foot and the outer center of the orthogonal plate attached to the foot and it is 47 millimeters. a_{3lower}, a_{3upper}, a_{3left} and a_{3right} values are all distances between the outer center of the orthogonal plate and the center of the tips of the foot. Their values are all 25 millimeters. Due to frame positioning, a_{3lower} and a_{3right} are assigned to 0.025 and a_{3upper} and a_{3left} are assigned to -0.025 in the calculations.

The general matrix form of the homogeneous transformation matrices associated to each link was given in **(3.1)**. Placing D-H parameters to this general form for each link *i* yields the homogeneous transformation matrices. Equations **(3.38)** to **(3.43)** show these matrices.

$$A_1 = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1)\cos(\alpha_1) & \sin(\theta_1)\sin(\alpha_1) & 0 \\ \sin(\theta_1) & \cos(\theta_1)\cos(\alpha_1) & -\cos(\theta_1)\sin(\alpha_1) & 0 \\ 0 & \sin(\alpha_1) & \cos(\alpha_1) & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{(3.38)}$$

$$A_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha_2) & -\sin(\alpha_2) & 0 \\ 0 & \sin(\alpha_2) & \cos(\alpha_2) & d_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{(3.39)}$$

$$A_{3lower} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & -a_{3lower} \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{(3.40)}$$

$$A_{3upper} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & -a_{3upper} \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{(3.41)}$$

$$A_{3right} = \begin{bmatrix} 1 & 0 & 0 & a_{3right} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{(3.42)}$$

$$A_{3left} = \begin{bmatrix} 1 & 0 & 0 & a_{3left} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{(3.43)}$$

The transformation matrices that transform the coordinates of the end effectors (tips of the foot) to the base (connection point of the wheel) can be derived as shown in equations **(3.44)** to **(3.47)**.

$$T_{3lower}^{0} = A_1 A_2 A_{3lower} \tag{3.44}$$

$$T_{3upper}^{0} = A_1 A_2 A_{3upper} \tag{3.45}$$

$$T_{3right}^{0} = A_1 A_2 A_{3right} \tag{3.46}$$

$$T_{3left}^{0} = A_1 A_2 A_{3left} \tag{3.47}$$

Since the calculations are complex, a modified version of the m-file created for the single module kinematic model is used to analyze the end-effector positions. The m-file also transforms the coordinates of the end-effector from the coordinate frame of the base of the model to the absolute frame of V-Rep. The transformation is shown in equations **(3.48)** to **(3.50)**. The transformation assumes that the base of the kinematic chain (connection point of the wheel) is positioned at the origin of the absolute frame of V-Rep.

$$x_{local} = -z_{model} \tag{3.48}$$

$$y_{local} = -x_{model} \tag{3.49}$$

$$z_{local} = y_{model} \tag{3.50}$$

After the coordinates of the end-effectors are clearly determined in the absolute frame of V-Rep, a suitable value for θ_1 for the forward and backward positions of the leg should be determined for the trot gait. For trotting, only the right and left tips of the foot positions are needed and the only variable is the wheel position. The first and the second joint positions are fixed at 60^0 and 30^0, respectively. For varying θ_1 values, the x, y, z coordinates of the left and right tips of the foot are shown in Figure 3.28 and Figure 3.29.

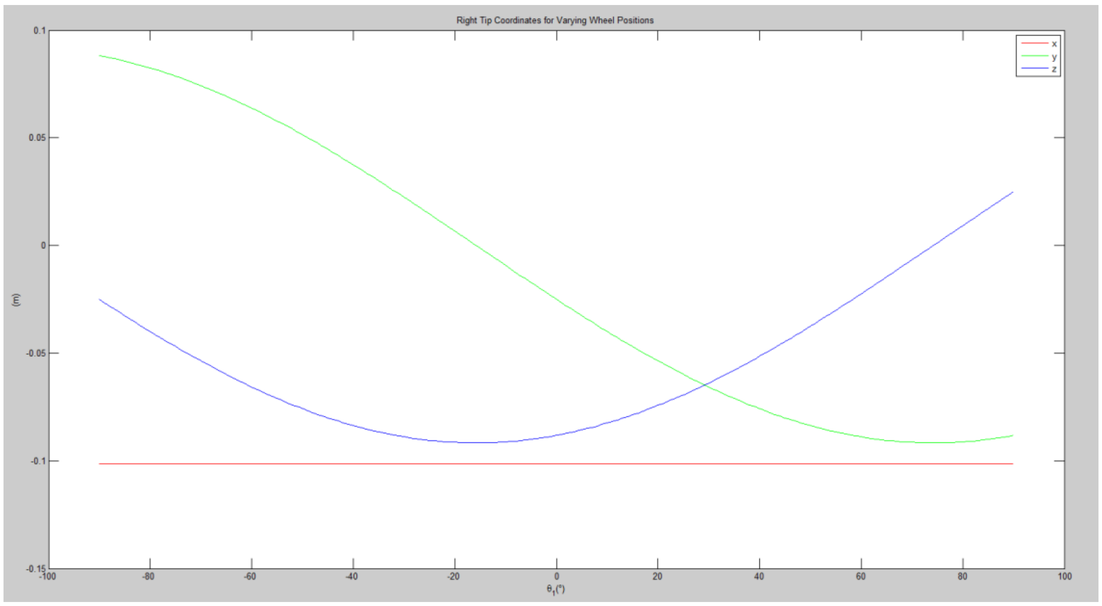


Figure 3.28: x-y-z coordinates of right tip of the foot for varying θ_1 values.

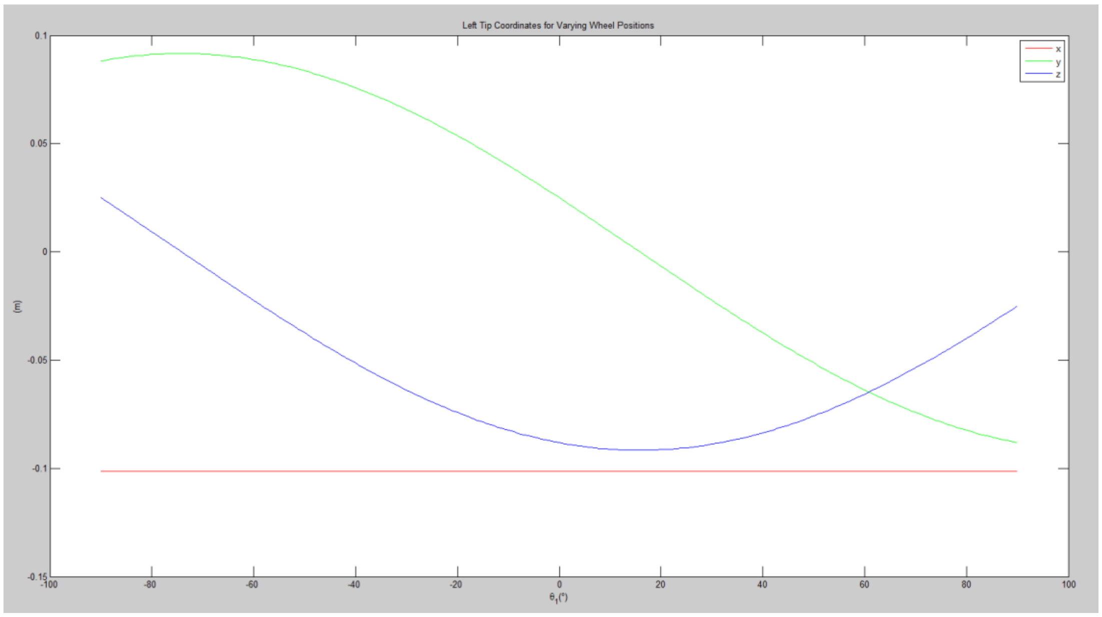


Figure 3.29: x-y-z coordinates of left tip of the foot for varying θ_1 values.

There are two constraints for the selection of θ_1 while swinging forward and backward. The first constraint is the height constraint. Since the legs have to lose contact with the ground after reaching the backward position to move to the forward position, the height of the tips of the legs that are contacting the terrain should be lower than 47 millimeters. For a right sided leg, this means that while θ_1 is positive the z_l value should be lower than -0.047 because while θ_1 is positive the left tip is contacting the terrain and while θ_1 is negative z_r value should be lower than -0.047 to avoid contact between the lifted legs and the terrain. Figure 3.30 shows the z_r and z_l values for varying θ_1 values and the -0.047 limit.

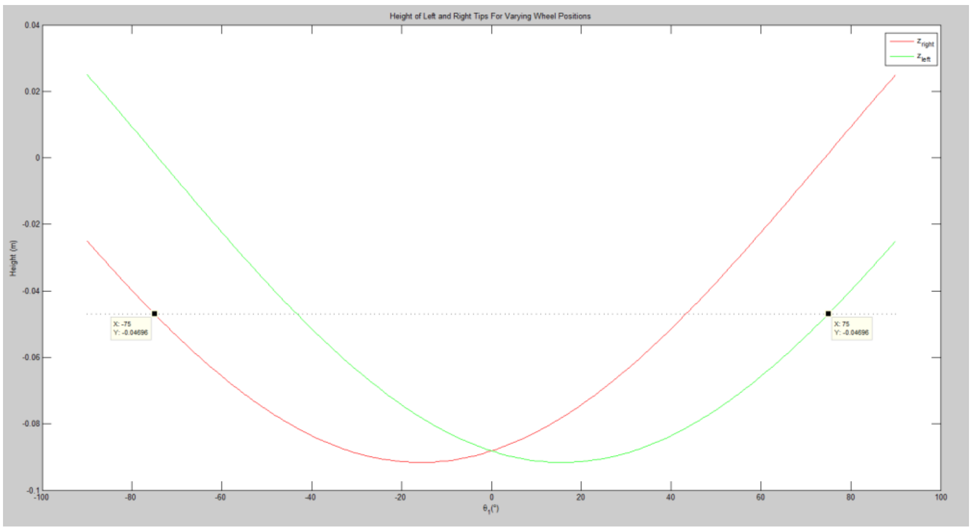


Figure 3.30: z_r and z_l values for varying θ_1 values.

The second constraint is the non-collision constraint. The longitudinal displacement of tips of the legs that are not contacting the terrain should not exceed 89.5 millimeters because there is a period when the hind leg is in the forward position and the front leg is in the backward position at the same side of the robot. The length of 89.5 millimeters is half of the distance between connection points of the front legs and the hind legs. Therefore y_l of a right sided front leg cannot be higher than 0.0895 and y_r of a right sided hind leg cannot be lower than -0.0895. Figure 3.31 shows the y_r and y_l values for varying θ_1 values and the limits.

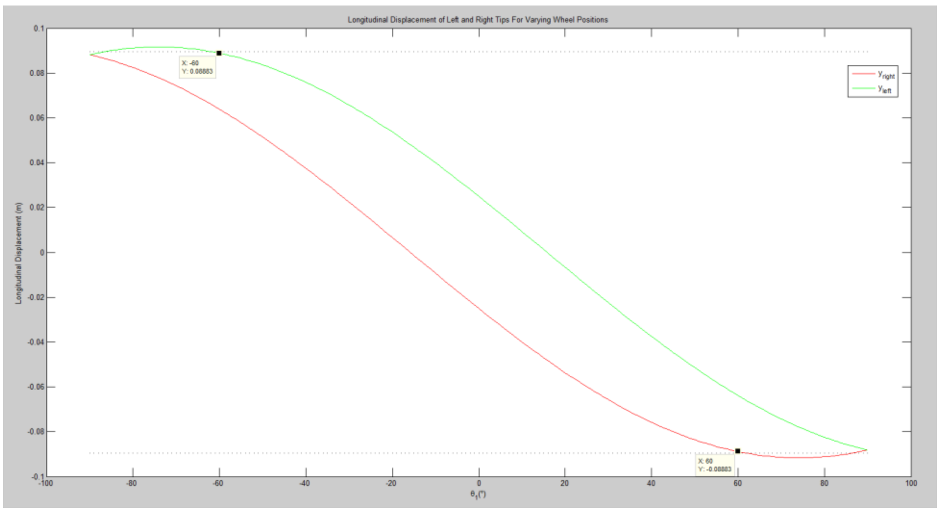


Figure 3.31: y_r and y_l values for varying θ_1 values

From the figures, it can be seen that the height constraint is not as limiting as the non-collision constraint. Due to the non-collision constraint, the maximum swing range of a leg should not exceed 120^0 (between -60^0 and 60^0). For more safety, the implemented maximum swing range is set to be 90^0 (between -45^0 and 45^0). For a standard trotting to move forward, the swing range is set to be 60^0 (between -30^0 and 30^0) to provide space for the orientation controller to increase or decrease the swing range. Table 3.19 shows the joint positions for each leg while trotting forward and Figure 3.32 to Figure 3.35 show the walker robot state in each step of the gait.

Table 3.19 : Joint positions of legs while trotting forward.

Step	Front Right (3)			Front Left (4)			Back Right (5)			Back Left (6)		
	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$
0	0	60	30	0	60	30	0	60	30	0	60	30
1	0	60	30	0	0	90	0	0	90	0	60	30
2	-30	60	30	-30	60	30	30	60	30	30	60	30
3	0	0	90	0	60	30	0	60	30	0	0	90
4	30	60	30	30	60	30	-30	60	30	-30	60	30

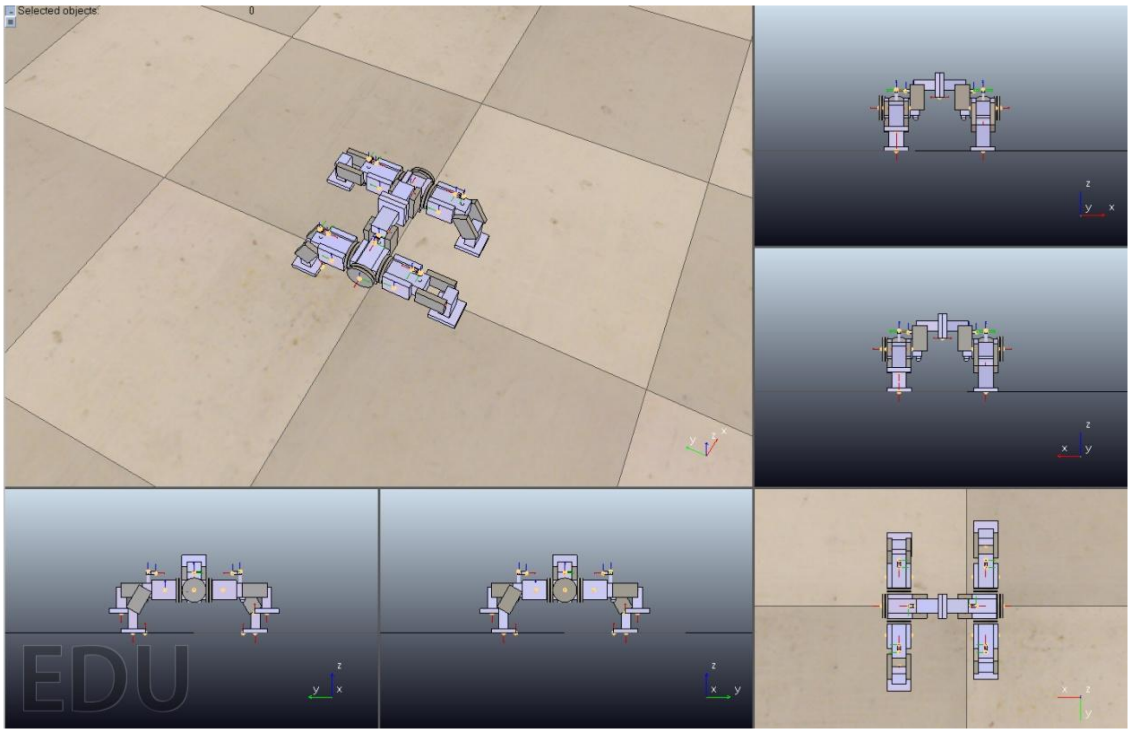


Figure 3.32: First step of the trot gait.

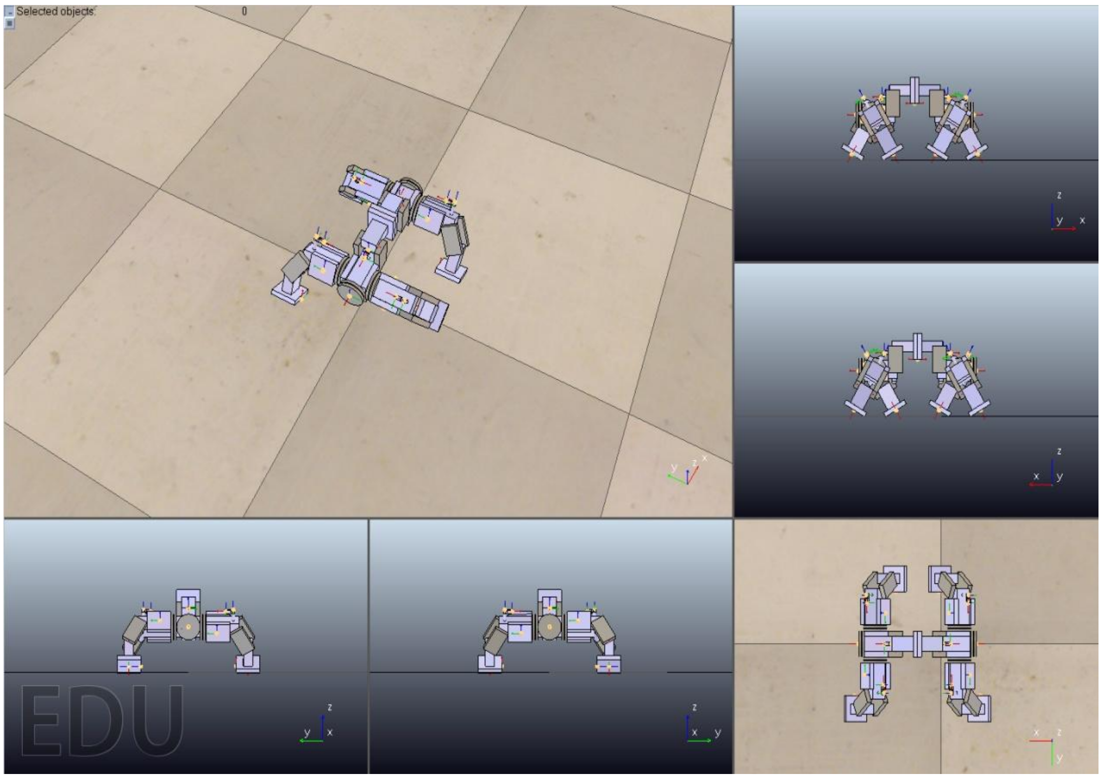


Figure 3.33: Second step of the trot gait.

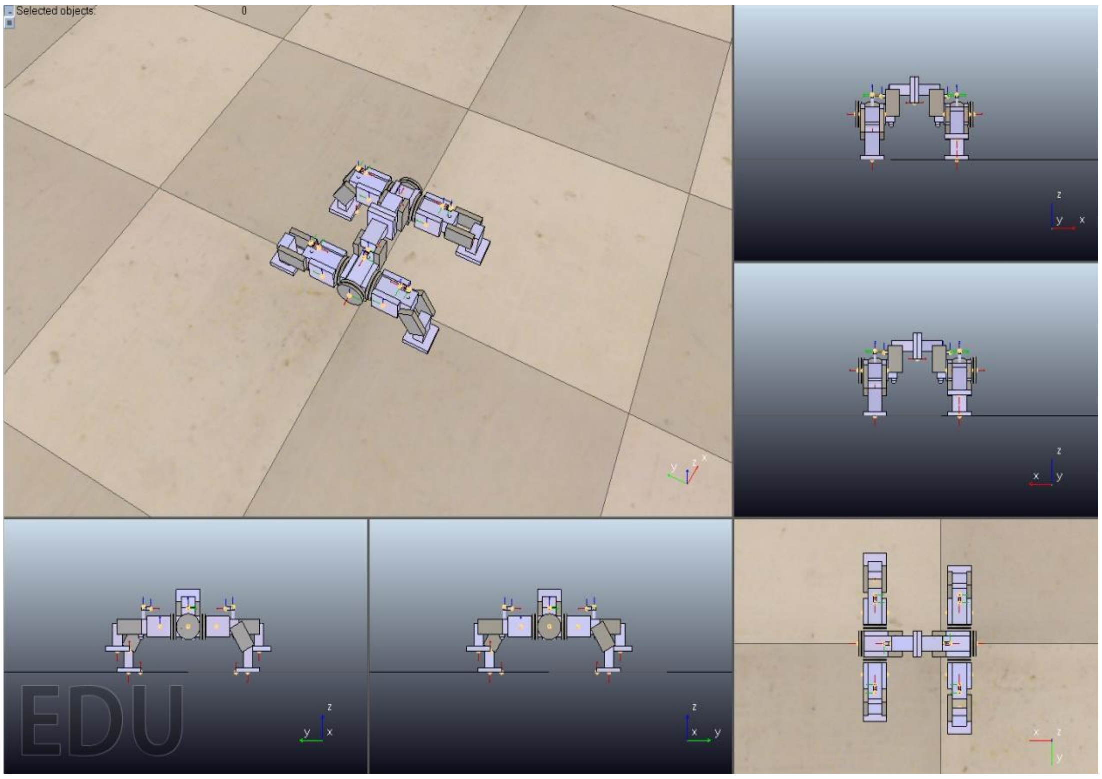


Figure 3.34: Third step of the trot gait.

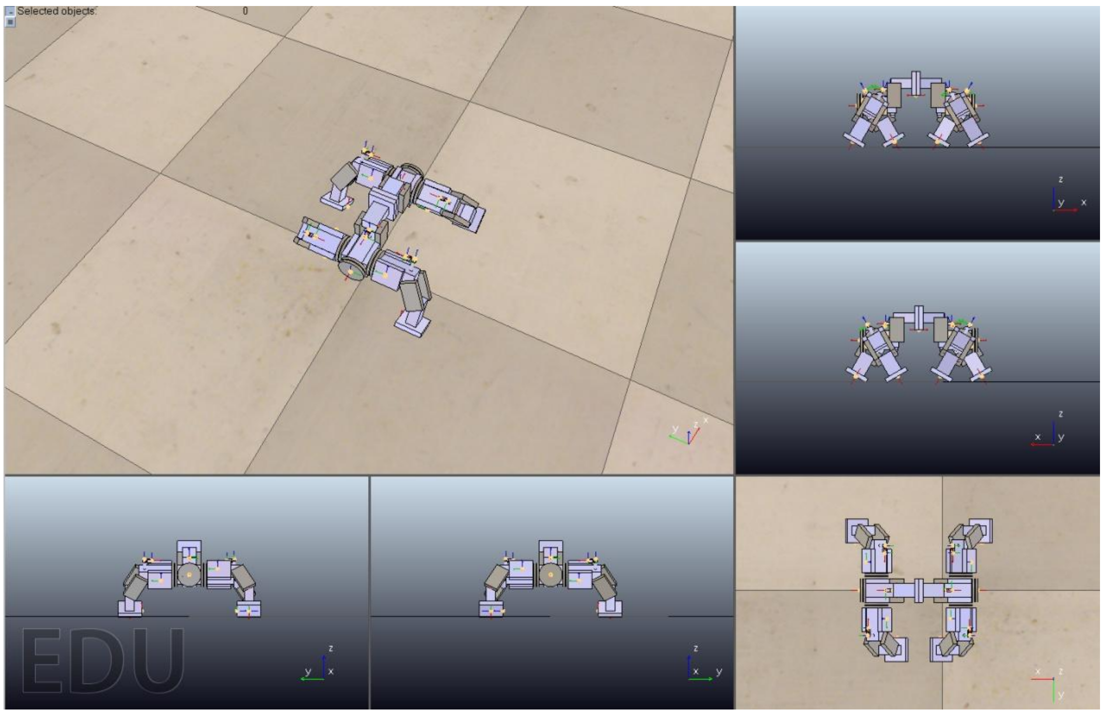


Figure 3.35: Fourth step of the trot gait.

The second part of the kinematic calculations is for sidling. For sidling, positions of the upper and lower tips of the foot should be analyzed. The only variable is the position of the second joint of the foot, while the wheel and the first joint of the foot positions are fixed at 0^0 and 60^0, respectively. Figure 3.36 and Figure3.37 show the end-effector positions for varying α_2 values.

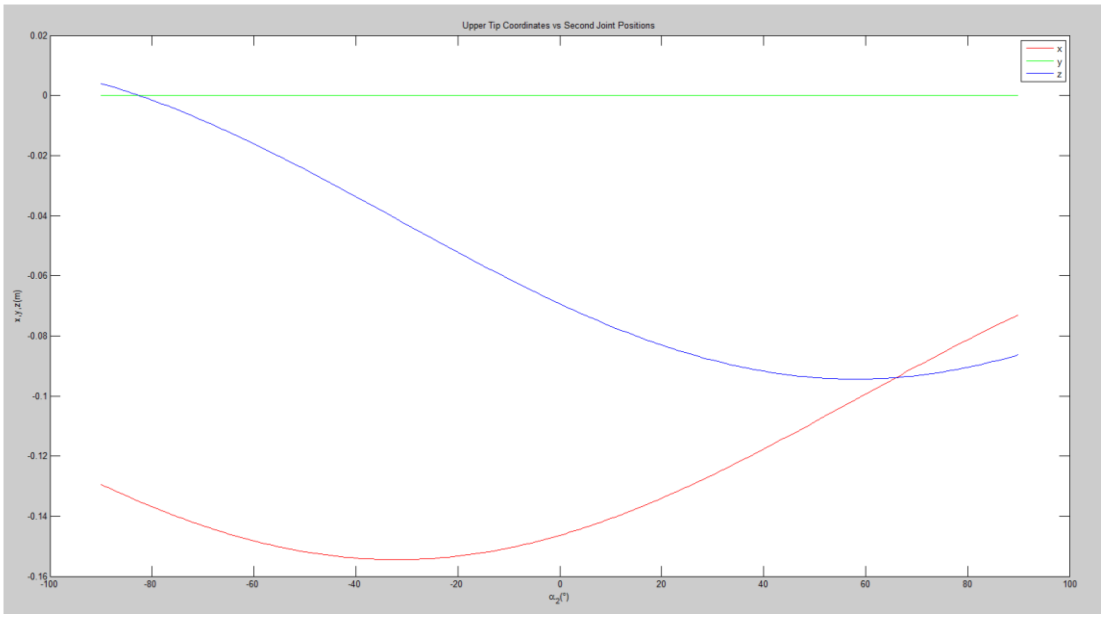


Figure 3.36: x-y-z coordinates of upper tip of the foot for varying α_2 values.

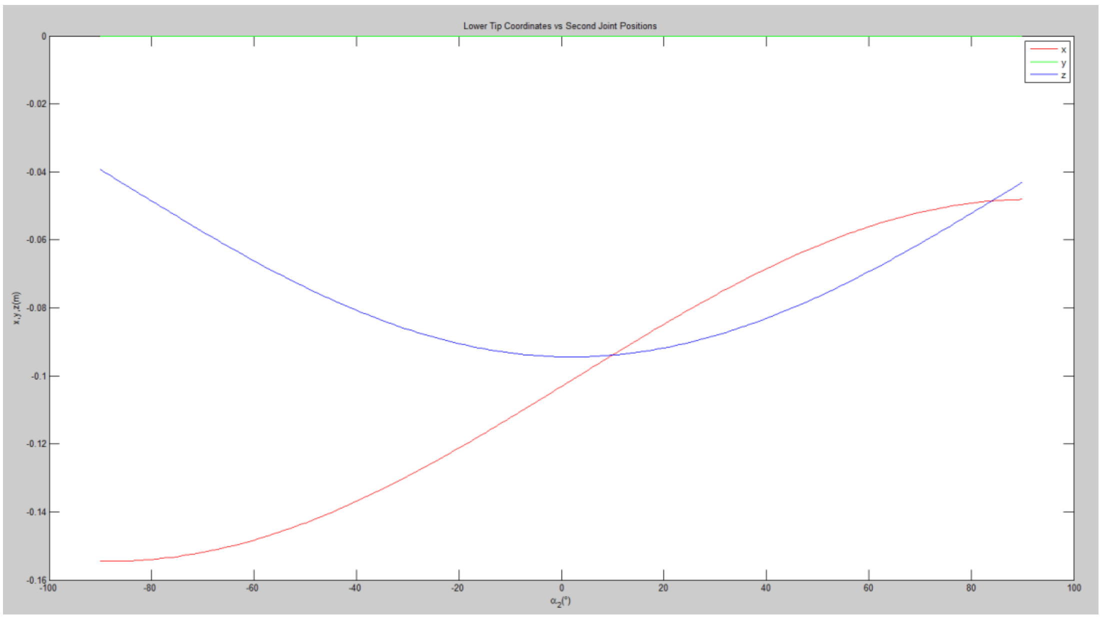


Figure 3.37: x-y-z coordinates of lower tip of the foot for varying α_2 values.

While sidling, the distance between the operating parts of legs are not as close as in the case while trotting. Therefore there is no need for a non-collision constraint. The only constraint regarding sidling is the same height constraint set for the trot gait. When the second joint position is higher than 30^0, z_{upper} should be less than -0.047 and when the second joint position is lower than 30^0, z_{lower} should be less than -0.047. Figure 3.38 shows the z_{upper} and z_{lower} values for varying α_2 values and -0.047 limit.

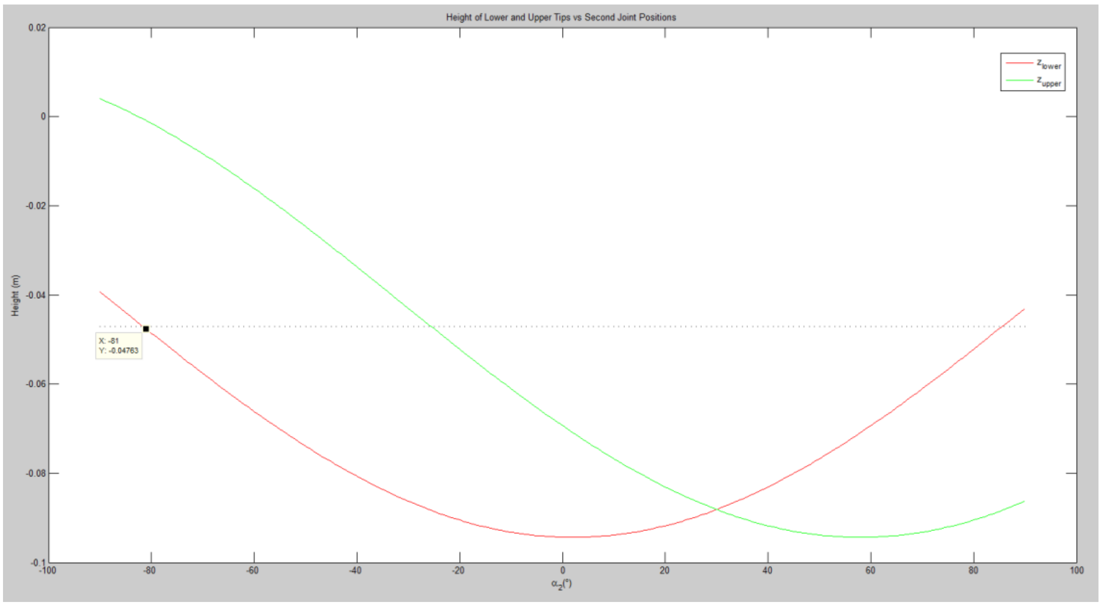


Figure 3.38: z_{upper} and z_{lower} values for varying α_2 values.

Due to the height constraint, α_2 cannot be lower than -81°. Since the height constraint is not very limiting, the swing range is set as 120^0 (between -30^0 and 90^0) to provide room for the orientation controller to increase it up to 150^0 (between -45^0 and 105^0) or decrease it down to 90^0 (between -15^0 and 75^0). Table 3.20 shows the joint positions and Figures 3.39 to Figure 3.42 show the robot state while the robot is sidling to its left.

Table 3.20 : Joint positions of legs while sidling to left.

Step	Front Right (3)			Front Left (4)			Back Right (5)			Back Left (6)		
	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$
0	0	60	30	0	60	30	0	60	30	0	60	30
1	0	60	30	0	0	90	0	0	90	0	60	30
2	0	60	-30	0	60	-30	0	60	90	0	60	90
3	0	0	90	0	60	30	0	60	30	0	0	90
4	0	60	90	0	60	90	0	60	-30	0	60	-30

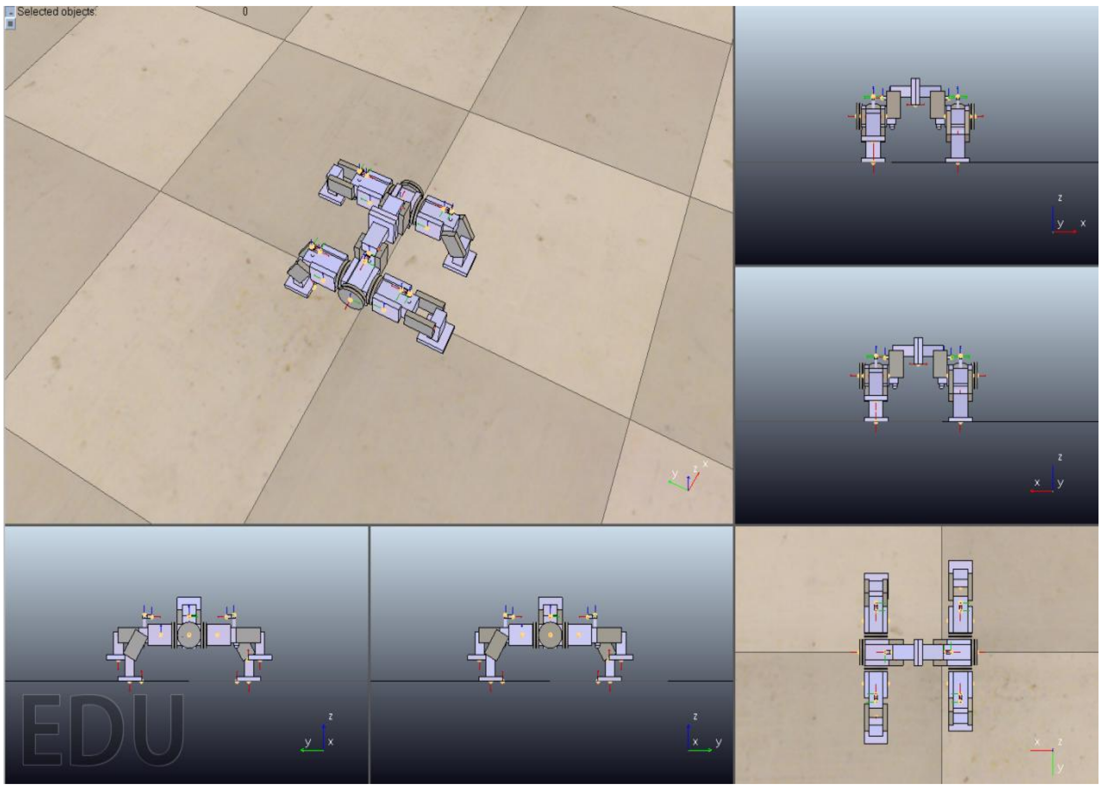


Figure 3.39: First step of the sidling gait.

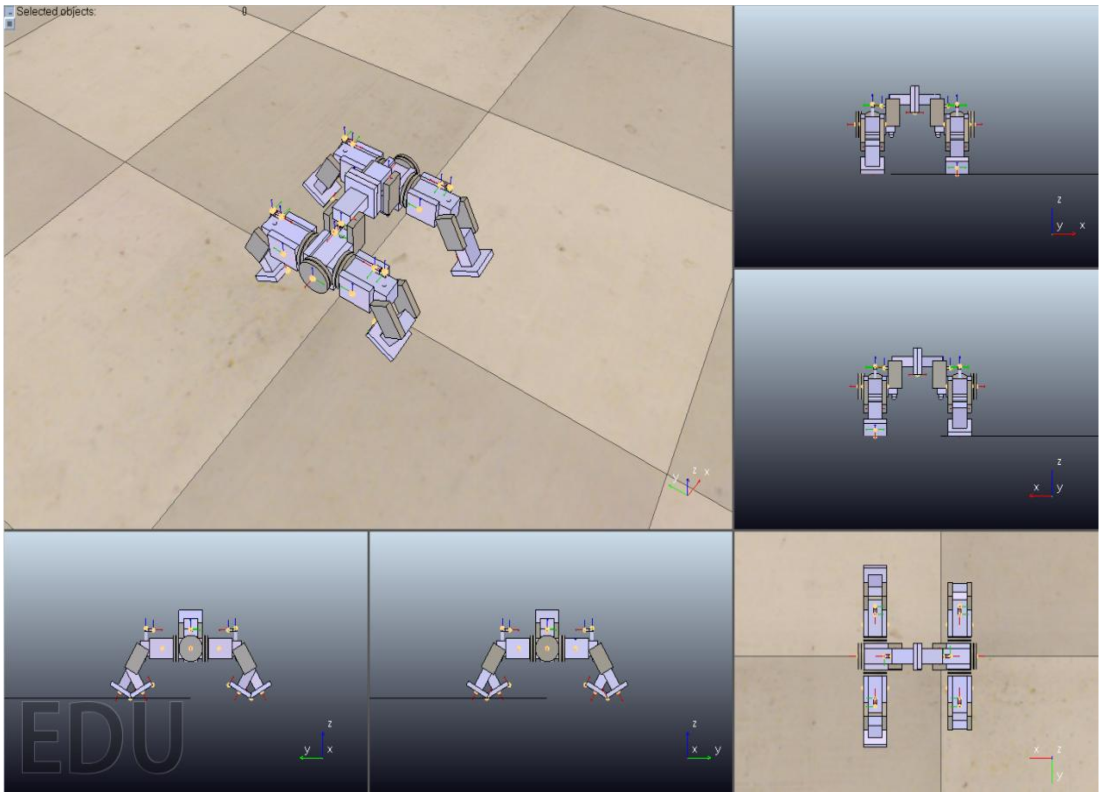


Figure 3.40: Second step of the sidling gait.

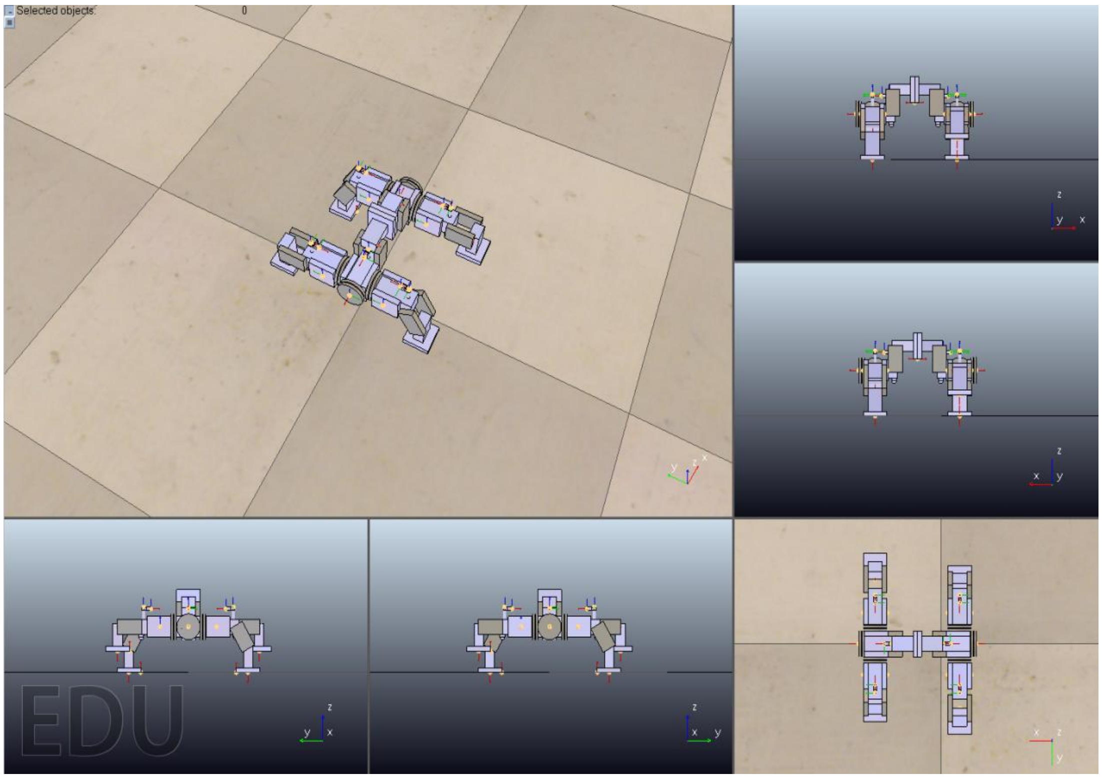


Figure 3.41: Second step of the sidling gait.

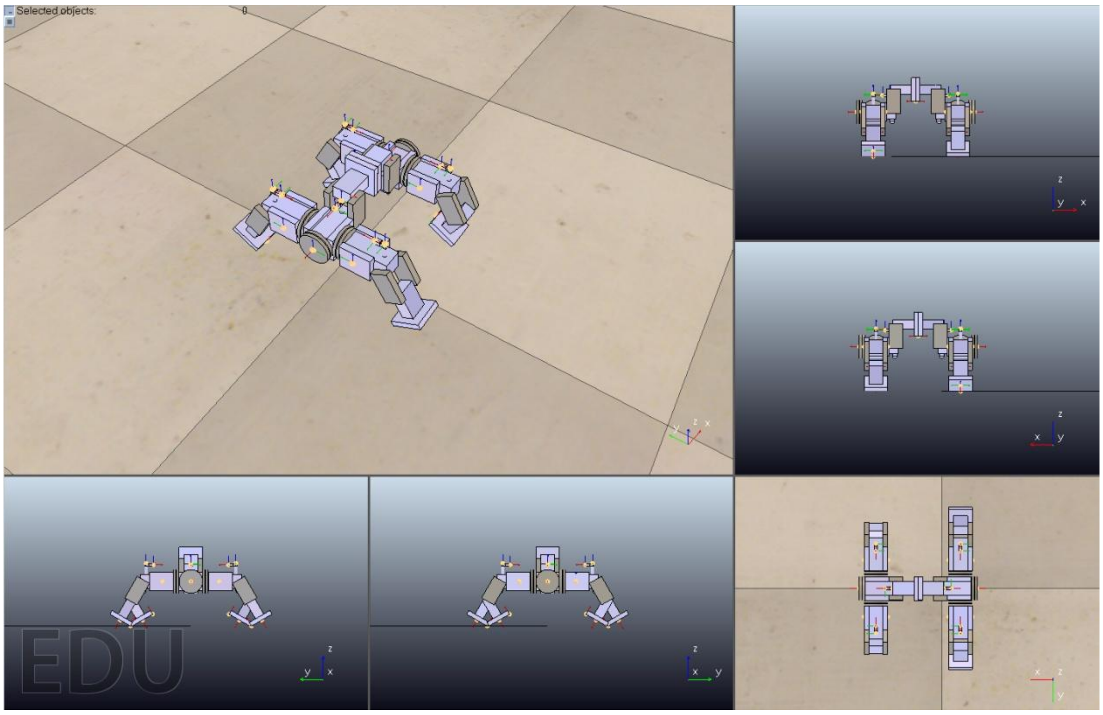


Figure 3.42: Fourth step of the sidling gait.

3.2.2.3 Motion kinematics and position control

The quadruped walker robot is able to move back and forth in longitudinal and lateral directions. The robot can also move on a curved path to control its orientation. The orientation control of the vehicle is similar to differential drive vehicles. Although the robot cannot rotate around its center, the difference between distance covered by the opposite sided legs causes an arc-like movement which gives the opportunity to control the orientation of the whole structure.

A switching controller is added to control the orientation of the robot for each gait. Both controllers have four states which gradually change the orientation of the structure more aggressively. To steer on a desired side of the robot, the controller decreases the swing range of the legs on the desired side and increases the swing range of the legs on the opposite side. The change in the swing range is 30^0 in the most aggressive state In the middle states, this value is 20^0 and 10^0. In the most passive state the swing range of the legs on the desired side is decreased by 10^0 and the swing range of opposite sided legs is unchanged. Table 3.21 shows the general state of leg joint positions for trot gait. Note that "l" and "r" stand for the effect of the controller on the swing range of the legs. "l" is the applied control input to the left sided legs and "r" is the applied control input to the right sided legs.

Table 3.21 : General joint positions of legs while trotting.

Step	Front Right			Front Left			Back Right			Back Left		
	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$
0	0	60	30	0	60	30	0	60	30	0	60	30
1	0	60	30	0	0	90	0	0	90	0	60	30
2	-30-r	60	30	-30-l	60	30	30+r	60	30	30+l	60	30
3	0	0	90	0	60	30	0	60	30	0	0	90
4	30+r	60	30	30+l	60	30	-30-r	60	30	-30-l	60	30

The controller states for sidling is the same, but it applies to the second joint of the foot instead of the wheel joint. The general state of leg joint positions is shown in Table 3.22. Similar to trotting case "f" and "b" are the effect of the controller on the swing range of the legs. "f" is the control input applied to the front legs and "b" is the control input applied to the legs at the back.

Table 3.22 : General joint positions while sidling.

Step	Front Right			Front Left			Back Right			Back Left		
	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$
0	0	60	30	0	60	30	0	60	30	0	60	30
1	0	60	30	0	0	90	0	0	90	0	60	30
2	0	60	-30-f	0	60	-30-f	0	60	90+b	0	60	90+b
3	0	0	90	0	60	30	0	60	30	0	0	90
4	0	60	90+f	0	60	90+f	0	60	-30-b	0	60	-30-b

The general motion kinematic model of the quadruped walker robot shown in Figure 3.43 can be expressed as shown in equations **(3.51)** to **(3.53)**.

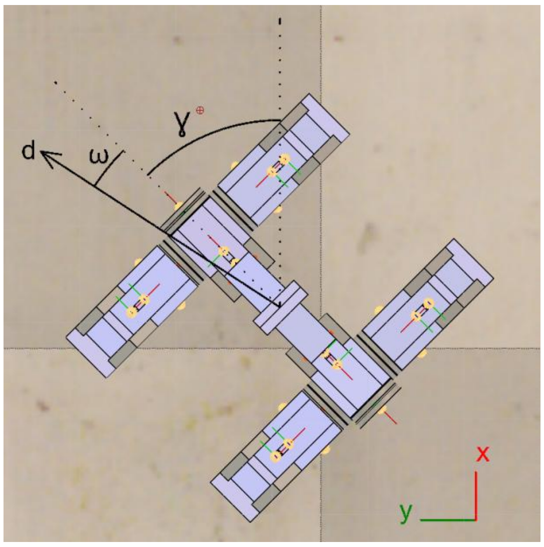


Figure 3.43: Longitudinal displacement and angular velocity of the quadruped walker.

$$\boldsymbol{x[k] = x[k-1] + d sin(\gamma[k-1] + \omega)} \tag{3.51}$$

$$\boldsymbol{y[k] = y[k-1] + d cos(\gamma[k-1] + \omega)} \tag{3.52}$$

$$\boldsymbol{\gamma[k] = \gamma[k-1] + \omega} \tag{3.53}$$

Similar to the single module case, the longitudinal displacement d and angular velocity ω are dependent on friction coefficient of the terrain. These values are determined experimentally in the simulation software by using terrains with varying friction coefficients. The experimental setup is the same as the single module case. Only difference is ω value is determined for the quadruped walker. The longitudinal displacement is determined by using equation **(3.24)** and taking its average on a 1 minute period same as the single module case. ω is determined by using **(3.54)** and its average is also calculated.

$$\boldsymbol{\omega = \gamma[k] - \gamma[k-1]} \tag{3.54}$$

Table 3.23 and Table 3.24 show longitudinal displacement and angular velocity while trotting forward on a curved path.

Table 3.23 : Longitudinal displacement and angular velocity while trotting in more aggressive states of the controller.

Terrain Friction Coefficient	r=15, l=-15			r=10, l=-10		
	ω(^{0}/cycle)	d(m/cycle)	t(s/cycle)	ω(^{0}/cycle)	d(m/cycle)	t(s/cycle)
1.0	19.95	0.101	1.25	14.72	0.106	1.25
0.8	20.86	0.106	1.25	15.04	0.112	1.25
0.6	19.08	0.106	1.20	13.35	0.112	1.20
0.4	16.61	0.095	1.20	8.59	0.099	1.20
0.2	7.14	0.058	1.20	3.76	0.06	1.20

Table 3.24 : Longitudinal displacement and angular velocity while trotting in more passive states of the controller.

Terrain Friction Coefficient	r=5, l=-5			r=0, l=-5		
	$\omega(^0$/cycle)	d(m/cycle)	t(s/cycle)	$\omega(^0$/cycle)	d(m/cycle)	t(s/cycle)
1.0	7.81	0.107	1.25	4.42	0.094	1.25
0.8	8.09	0.114	1.25	4.71	0.099	1.25
0.6	6.53	0.116	1.20	3.56	0.105	1.20
0.4	3.91	0.101	1.20	2.11	0.095	1.20
0.2	1.47	0.058	1.20	0.36	0.057	1.20

Equations (3.51) to (3.53) also hold for sidling. For better understanding Figure 3.42 is modified as shown in Figure 3.44. Table 3.25 and 3.26 show longitudinal displacement and angular velocity while sidling to left on a curved path.

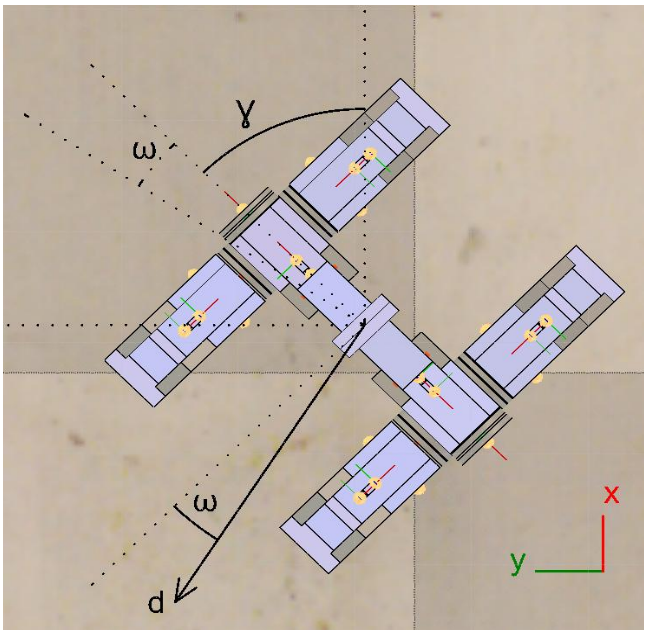


Figure 3.44: Longitudinal displacement and angular velocity of the quadruped walker.

Table 3.25 : Longitudinal displacement and angular velocity while sidling in more aggressive states of the controller.

Terrain Friction Coefficient	f=15, b=-15			f=10, b=-10		
	$\omega(^0$/cycle)	d(m/cycle)	t(s/cycle)	$\omega(^0$/cycle)	d(m/cycle)	t(s/cycle)
1.0	3.88	0.107	1.90	2.49	0.106	1.80
0.8	4.96	0.111	1.90	3.54	0.111	1.80
0.6	6.94	0.116	1.85	4.84	0.120	1.80
0.4	7.87	0.109	1.80	5.79	0.113	1.80
0.2	7.58	0.081	1.80	4.22	0.81	1.70

Table 3.26 : Longitudinal displacement and angular velocity while sidling in more passive states of the controller.

Terrain Friction Coefficient	f=5, b=-5			f=0, b=-5		
	$\omega(^0$/cycle)	d(m/cycle)	t(s/cycle)	$\omega(^0$/cycle)	d(m/cycle)	t(s/cycle)
1.0	1.74	0.107	1.85	1.02	0.103	1.80
0.8	1.51	0.111	1.80	0.85	0.105	1.80
0.6	2.35	0.121	1.80	1.22	0.115	1.80
0.4	1.89	0.115	1.80	1.08	0.110	1.80
0.2	1.94	0.080	1.80	0.82	0.075	1.80

Table 3.27 shows the longitudinal displacement while the robot is trotting forward and sidling left without the effects of the orientation controller. The angular velocity value is not recorded in these cases.

Table 3.27 : Longitudinal displacement while trotting forward and sidling left without controller effect.

Terrain Friction Coefficient	r=0, l=0		f=0, b=0	
	d(m/cycle)	t(s/cycle)	d(m/cycle)	t(s/cycle)
1.0	0.106	1.25	0.110	1.80
0.8	0.112	1.25	0.113	1.80
0.6	0.115	1.20	0.124	1.80
0.4	0.104	1.20	0.118	1.80
0.2	0.058	1.20	0.080	1.80

The position control algorithm of the quadruped configuration is different from the control algorithms of single module and wheeled configuration because the quadruped configuration cannot rotate around its center. Orientation control is a harder problem for this configuration than the single module or wheeled configuration. Therefore target state orientation should be controlled before arriving the target position in this case.

To control both the position and the orientation of the quadruped robot a switching controller is designed which controls the swing range of the right and left sided legs while trotting and front and hind legs while sidling. The swing range control is based on the desired angle of the target state and lateral and longitudinal distance from the target position. Figure 3.45 shows 12 possible actions which can be implemented by the quadruped configuration.

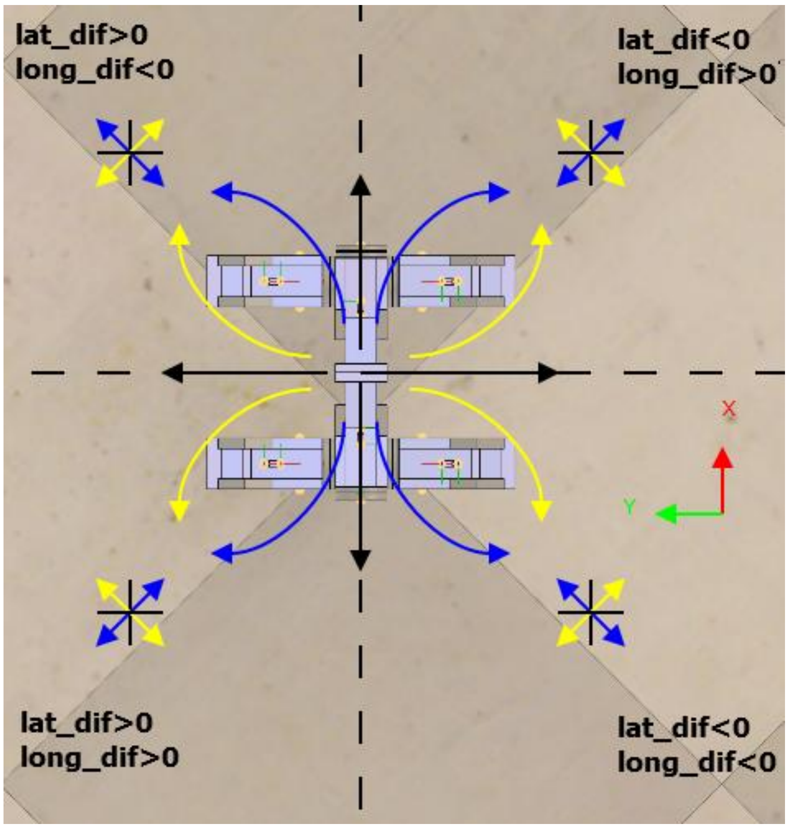


Figure 3.45: 12 possible actions which can be implemented by the quadruped configuration.

In the figure, blue arcs represent the motion when trotting and yellow lines represent the motion when sidling. The small double sided arrows show the new orientation range. Blue arrows represent the orientation range when trotting and the yellow arrows represent orientation range when sidling. As it can be seen from the figure, based on the target and configuration position, the coordinate system can be broken down into four quadrants. The other important property of the configuration is that the quadruped walker can move towards any of the quadrants while changing its orientation clockwise or counter clockwise depending on its decision to trot or sidle. The position control algorithm is designed using these properties. The lateral and longitudinal distance data acquired from the visual sensor is used to determine on which quadrant the target is positioned and using the orientation difference between the robot and the target state, the decision to trot or sidle is made. The pseudo code of the position control algorithm of the quadruped configuration can be found in Table 3.28. Note that p-code assumes the visual sensor is locked to the reference target and the lateral and longitudinal differencee between the target and the gamma orientation desired is correctly calculated.

Table 3.28 : Pseudo-code for the position control algorithm of the quadruped configuration.

```
IF(lat_dif>0 AND long_dif>0) THEN
    IF(-90<ornt_dif<0 AND 90<ornt_dif<180) THEN
        trotting=0
        sidling=1
    ENDIF
    IF(0<ornt_dif<90 AND -180<ornt_dif<-90) THEN
        trotting=1
        sidling=0
    ENDIF
ENDIF
IF(lat_dif<0 AND long_dif>0) THEN
    IF(-90<ornt_dif<0 AND 90<ornt_dif<180) THEN
        trotting=1
        sidling=0
    ENDIF
    IF(0<ornt_dif<90 AND -180<ornt_dif<-90) THEN
        trotting=0
        sidling=1
    ENDIF
ENDIF
IF(lat_dif<0 AND long_dif<0) THEN
    IF(-90<ornt_dif<0 AND 90<ornt_dif<180) THEN
        trotting=0
        sidling=1
    ENDIF
    IF(0<ornt_dif<90 AND -180<ornt_dif<-90) THEN
        trotting=1
        sidling=0
    ENDIF
ENDIF
IF(lat_dif>0 AND long_dif>0) THEN
    IF(-90<ornt_dif<0 AND 90<ornt_dif<180) THEN
        trotting=1
        sidling=0
    ENDIF
    IF90<ornt_dif<90 AND -180<ornt_dif<-90) THEN
        trotting=0
        sidling=1
    ENDIF
ENDIF
```

The pseudo code is very simple compared to the actual position control algorithm applied. In the actual algorithm trotting and sidling is done more complex with different swing ranges on the right and left sided legs while trotting and different swing ranges on the front and hind legs while sidling. The orientation difference limits also change depending on the friction coefficient of the terrain. The actual position control algorithm of the robotic system can be found in the move_quad() function of the module script given in Appendix-C.

3.2.2.4 Passing over obstacles

The ability to pass over obstacles is the most important property of the quadruped configuration and the main reason for the robotic structure to decide assembling a quadruped walker. Passing over obstacle ability is designed as a simple sequence of three poses given in Table 3.29

Table 3.29 : Joint positions of the poses for passing over obstacles sequence.

Step	Front Right			Front Left			Back Right			Back Left		
	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$	$\theta_1(^0)$	$\theta_2(^0)$	$\theta_3(^0)$
0	0	60	30	0	60	30	0	60	30	0	60	30
1	-135	60	30	135	60	30	-135	60	30	135	60	30
2	135	60	30	-135	60	30	135	60	30	-135	60	30

The poses given in Table 3.29 can be seen visually in figures 3.46 to 3.48.

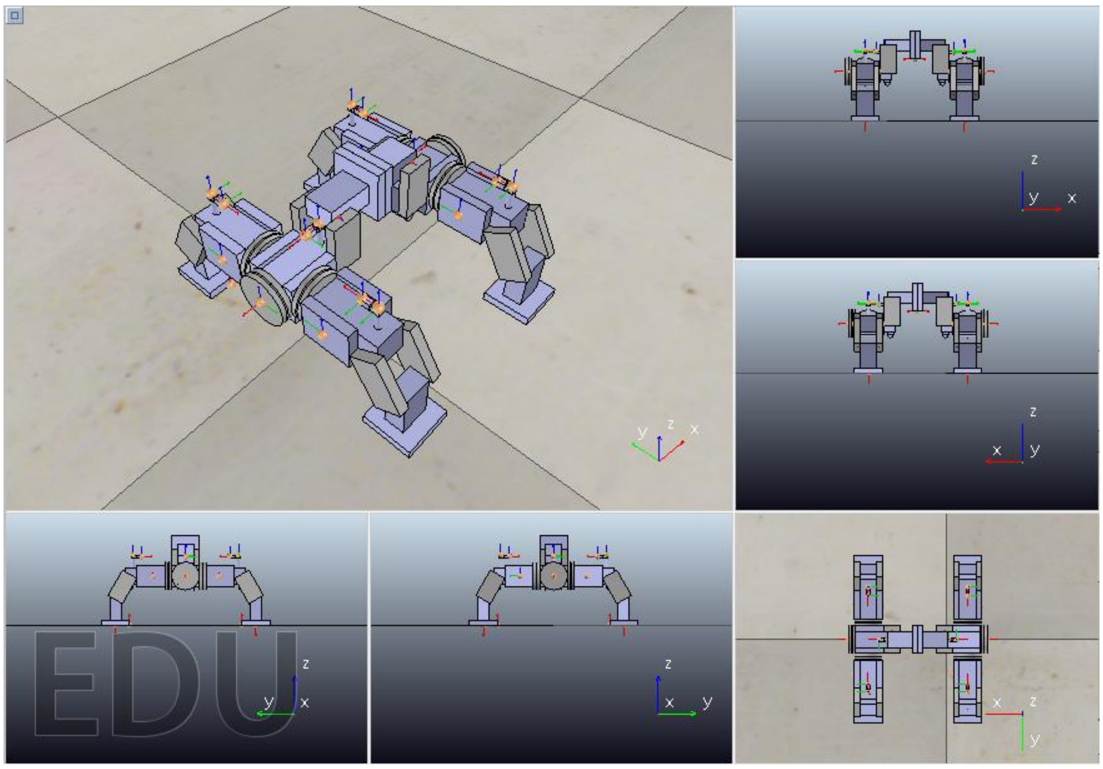


Figure 3.46: First pose of passing over obstacles sequence.

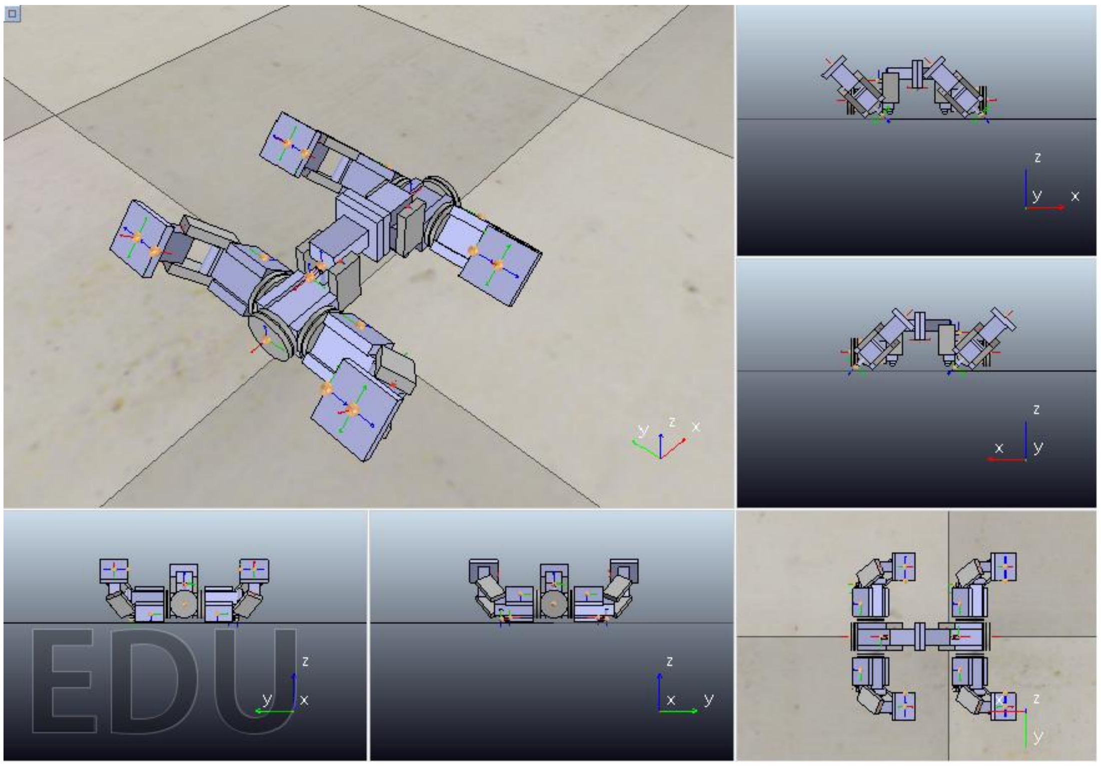


Figure 3.47: Second pose of passing over obstacles sequence.

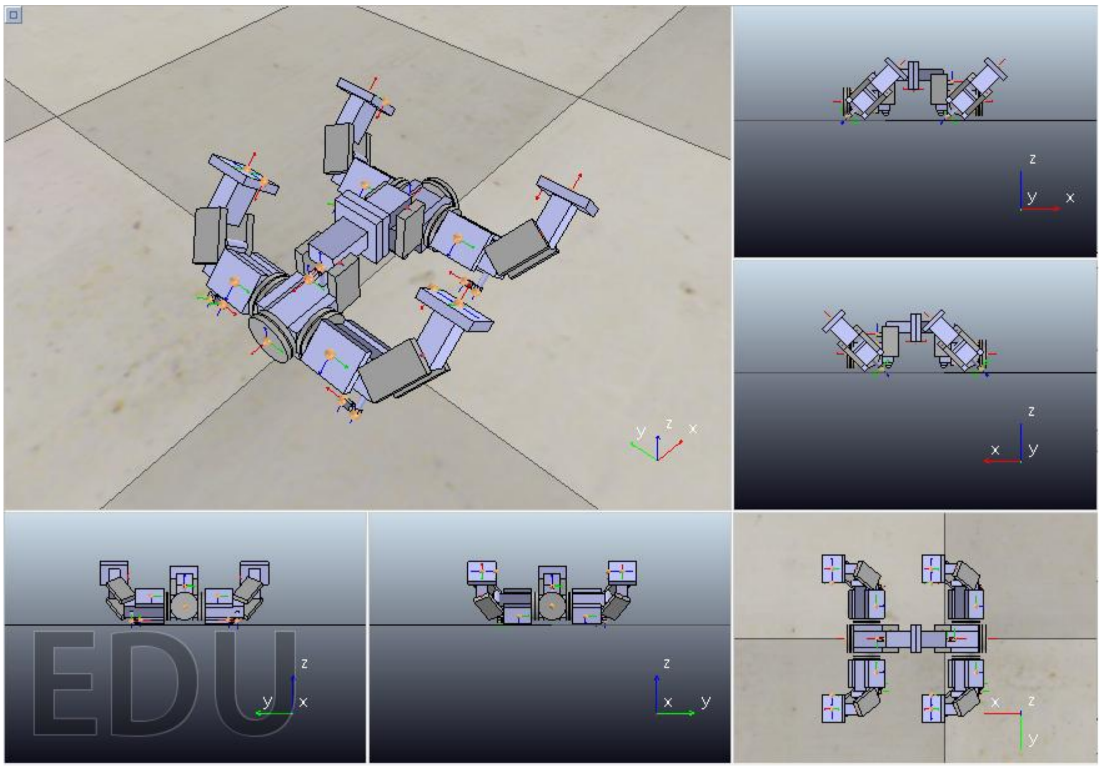


Figure 3.48: Third pose of passing over obstacles sequence.

Passing over obstacles sequence is repeated until the quadruped walker passes over the obstacle.

3.2.3 Wheeled locomotion

3.2.3.1 Structure and role distribution

The body of the configuration is formed by two modules making a foot-to-foot connection. The rest of the modules form the wheels of the robot and they make a foot-to-body connection with the modules forming the body. Four-wheeled robot is shown in Figure 3.49.

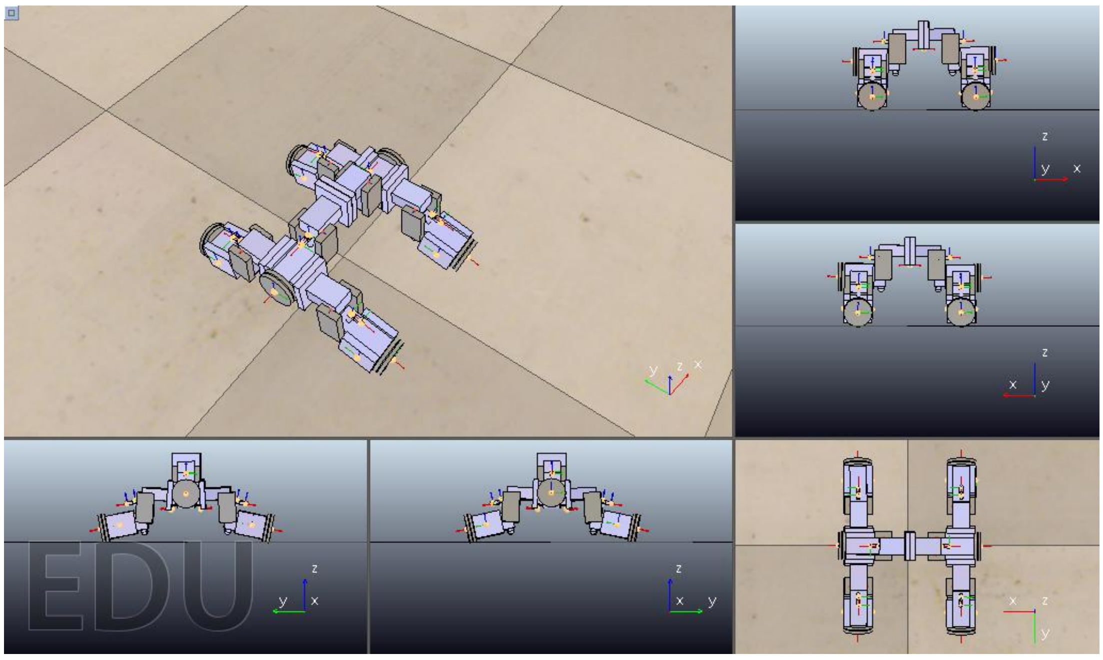


Figure 3.49: Wheeled configuration.

The joint positions of modules in normal pose of the wheeled configuration is given in Table 3.30

Table 3.30 : Joint positions of the normal pose of the wheeled configuraion.

Role	Part	Front Joint	BackJoint1	BackJoint2
1	Body Front	0	-90	90
2	Body Back	0	-90	90
3	Front Right Leg	0	-75	90
4	Front Left Leg	0	-75	90
5	Back Right Leg	0	-75	90
6	Back Left Leg	0	-75	90

The role distribution of this configuration is very similar to the configuration to achieve quadruped locomotion. The two modules forming the body are Role#1 and Role#2 and the wheel of Role#1 is assumed to be pointing the forward direction.

Role#3 makes a foot-to-body connection with Role#1 and forms the front right wheel. Role#4 forms the front left wheel and its foot connects to the left side of the body of Role#1. The wheels at the back are created by Role#5 and Role#6. Role#5 creates the right rear wheel and its foot connects to the left connection point of Role#2. The module forming the left rear wheel is Role#6 and it makes a foot-to-body connection with Role#2 on the left side.

The predetermined module positions and orientations for each role to assemble the wheeled configuration is given in Table 3.31.

Table 3.31 : Predetermined module positions for wheeled configuration.

Role	Part	Position(x, y, z)	Orientation(α, β, γ)
1	Body Front	(0, 0, 0)	(0, 0, 0)
2	Body Back	(-0.175, 0, 0)	(0, 0, 180)
3	Front Right Wheel	(0.137, -0.230, 0)	(0, 0, -90)
4	Front Left Wheel	(0.137, 0.230, 0)	(0, 0, 90)
5	Back Right Wheel	(-0.137, -0.230, 0)	(0, 0, -90)
6	Back Left Wheel	(-0.137, 0.230, 0)	(0, 0, 90)

Similar to the assembly of the quadruped walker configuration, in the table the module which has the first role is assumed to be positioned in the origin and its gamma orientation is γ_1. Figure 3.50 shows the module positions before they start to connect.

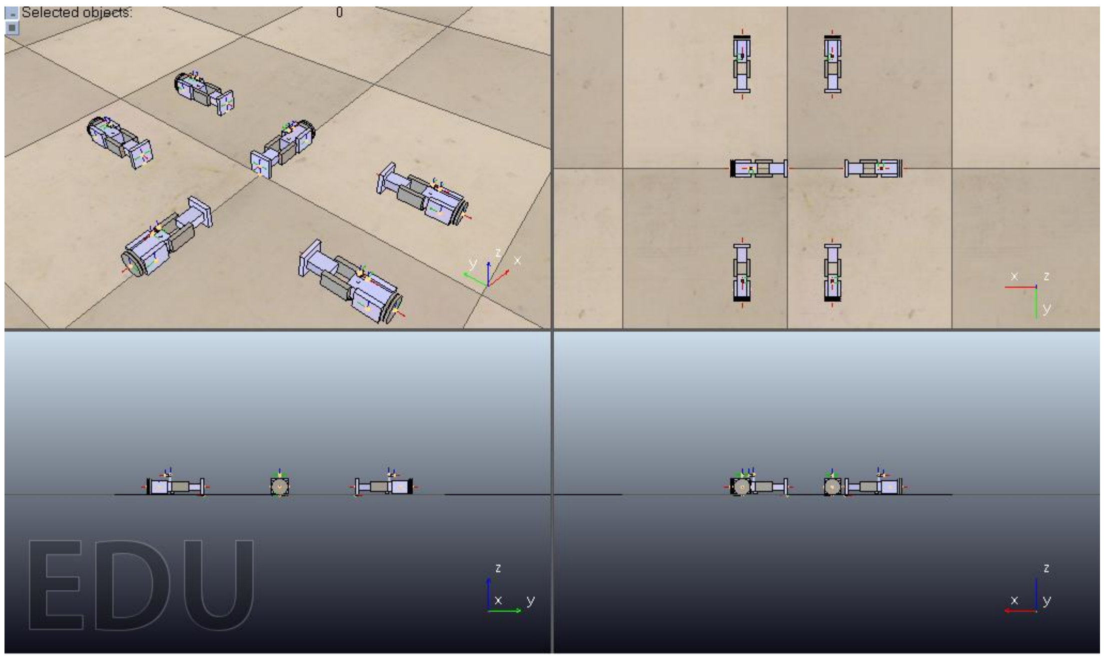


Figure 3.50: Predetermined module positions for wheeled configuration.

The reason for having the predetermined positions of modules having the third, the fourth, the fifth and the sixth roles is their connection types. Since they make foot-to-body connection with the first and second modules, they have to be positioned away enough from these modules in case they decide to reach their predetermined positions by forward locomotion gait. If their predetermined positions were not away at least one module length, they could collide with the modules they have to connect.

In actual assembly phase the modules do not wait to connect to their target connection points. The modules which have the second, the third and the fourth roles move to complete connection after they arrive their predetermined positions and adjust their heading angles as desired. The modules which have the fifth and the sixth roles wait for the second module to complete its connection before connecting even if they have the required position and orientation.

3.2.3.2 Motion kinematics and position control

An experimental motion kinematic model of a skid-steered vehicle that is developed in [14] is used in this configuration. The development of the model is also explained in this part.

The control inputs for the wheeled configuration are V_l and V_r which are the linear velocities of wheels on the left and right with respect to the robot frame. The relationship between the control inputs and the motion kinematics of the robot can be stated as shown in (**3.55**) where v_x and v_y is the vehicle's translational velocity with respect to local frame of the vehicle and ω_z is the angular velocity.

$$\begin{pmatrix} \boldsymbol{v_x} \\ \boldsymbol{v_y} \\ \boldsymbol{\omega_z} \end{pmatrix} = \boldsymbol{A} \begin{pmatrix} \boldsymbol{V_l} \\ \boldsymbol{V_r} \end{pmatrix} \tag{3.55}$$

To develop a motion kinematic model for the wheeled configuration, the effects of control inputs on motion kinematics of the structure should be determined. The most important variable that determines the motion kinematic model is the *ICR* (Instantaneous Center of Rotation) of the vehicle. The formal definition of *ICR* is given in [23] as;

"The instant centre of rotation, also called instantaneous centre or instant centre, is the point in a body undergoing planar movement that has zero velocity at a

particular instant of time. At this instant, the velocity vectors of the trajectories of other points in the body generate a circular field around this point which is identical to what is generated by a pure rotation."

For a skid steered vehicle such as the wheeled configuration of this modular structure, besides the *ICR* of the vehicle, also the *ICR*s of left sided and right sided wheels can be expressed as given in equations (**3.56**) to (**3.58**).Note that in the equations ICR_l, ICR_r and ICR_v stands for *ICR* of the left sided wheels, *ICR* of the right sided wheels and *ICR* of the wheeled configuration, respectively.

$$\boldsymbol{ICR_v = (x_{ICR_v}, y_{ICR_v})} \tag{3.56}$$

$$\boldsymbol{ICR_r = (x_{ICR_r}, y_{ICR_r})} \tag{3.57}$$

$$\boldsymbol{ICR_l = (x_{ICR_l}, y_{ICR_l})} \tag{3.58}$$

In [15], it is already stated that there is a line parallel to the *x* axis of the vehicle frame on which ICR_l, ICR_r and ICR_v lies on. Figure 3.51 shows the visual representations of the *ICR* values.

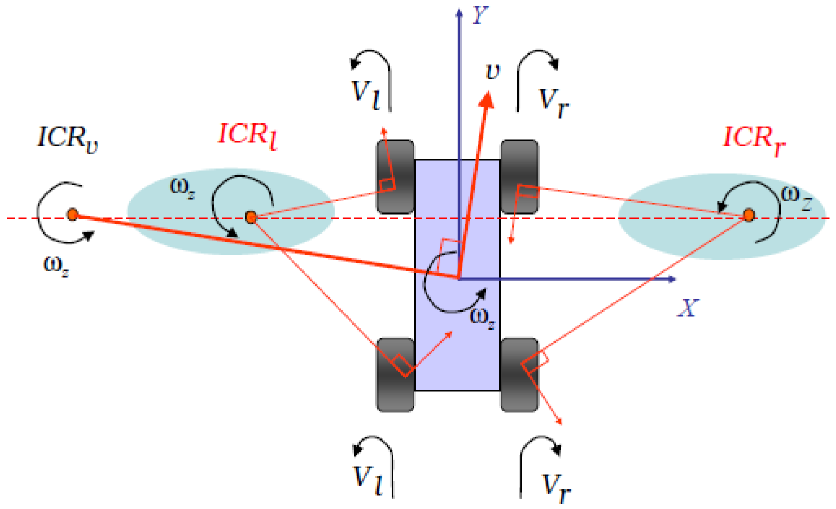


Figure 3.51: Visual representation of ICR values.

From Figure 3.47 and equations(**3.56**) to (**3.58**), *ICR* values can be used to relate control inputs V_l and V_r to motion kinematic variables of the wheeled configuration v_x, v_y and w_z. Equations (**3.59**) to (**3.62**) show these relations.

$$\boldsymbol{x_{ICRv} = \frac{-v_y}{\omega_z}} \tag{3.59}$$

$$\boldsymbol{x_{ICRl} = \frac{\alpha_l V_l - v_y}{\omega_z}} \tag{3.60}$$

$$\boldsymbol{x_{ICRr} = \frac{\alpha_r V_r - v_y}{\omega_z}} \tag{3.61}$$

$$\boldsymbol{y_{ICRv} = y_{ICRl} = y_{ICRr} = \frac{v_x}{\omega_z}} \tag{3.62}$$

In equations (**3.60**) and (**3.61**), α_l and α_r stands for correction factors. They are values between 0 and 1.00 which are used for determining mechanical factors which are not taken into consideration in the motion kinematic model.

The elements of matrix *A* given in (**3.55**) depend on *ICR* coordinates of the wheels on left and right side of the configuration and correction factors. Therefore using equations (**3.59**) to (**3.62**), the *A* matrix can be formed as shown in (**3.63**).

$$\boldsymbol{A = \frac{1}{x_{ICRr} - x_{ICRl}} \begin{bmatrix} -y_{ICRv}\alpha_l & y_{ICRv}\alpha_r \\ x_{ICRr}\alpha_l & -x_{ICRl}\alpha_r \\ \alpha_l & \alpha_r \end{bmatrix}} \tag{3.63}$$

As stated before, the wheeled robot assembled in this configuration is symmetric around the local *x* and *y* axes. This means that *ICR*s lie symmetrically on the local *x* axis and y_{ICRv} is equal to zero. Applying these to (**3.63**), matrix *A* takes the form;

$$\boldsymbol{A = \frac{\alpha}{2x_{ICR}} \begin{bmatrix} 0 & 0 \\ x_{ICR} & x_{ICR} \\ -1 & 1 \end{bmatrix}} \tag{3.64}$$

where $\alpha = \alpha_r = \alpha_l$ and $x_{ICR} = -x_{ICRl} = x_{ICRr}$.

Since the kinematic relationship is dependent to x_{ICR} and α value, the determination of these values is the next step. Two experimental methods are proposed to find these parameters in [16]. Note that these methods are applicable for symmetric models. They do not contemplate the asymmetric effects of the center of mass or mechanical misalignments.

The first method to determine x_{ICR} value can be applied by using equal opposite control inputs of V_r and V_l in terrains with varying friction coefficients. The vehicle is expected to rotate about its z axis. By measuring the distance traveled by the wheels and the actual rotated angle, the x_{ICR} value can be determined like shown in (**3.65**).

$$\boldsymbol{x_{ICR} \cong \frac{\int V_r dt - \int V_l dt}{2\varphi}} \tag{3.65}$$

To find the correction factor α, equal control inputs of V_r and V_l is applied. The vehicle is expected to move on a straight line. The correction factor can be determined by measuring the distance traveled by the wheels and the actual distance traveled in straight motion by the vehicle. The equation given in (**3.66)** shows the method to determine the correction factor.

$$\boldsymbol{\alpha \cong \frac{2d}{\int V_r dt + \int V_l dt}} \tag{3.66}$$

After the x_{ICR} and α values are determined for the vehicle and the motion kinematic model is created, the position control algorithm can be developed. Since the wheeled configuration can rotate around its center like the single module, the control algorithms are very similar. The difference is that the wheeled structure can use the wheel speed differences on different sides of the configuration for orientation correction to avoid the obligation to stop. For that purpose, an orientation controller is added to modify the heading angle of the configuration. The orientation controller modifies the heading angle of the robot by simply decreasing the speed of the wheels on the required turning side of the robot proportional to the orientation error. To prevent the controller from stopping the wheels on any side of the robot, the minimum speed limit is set as 25%.

3.2.3.3 Passing under obstacles

Wheeled configuration can adjust its axle positions to lower its height and pass under tunnel-like obstacles. To pass under obstacles, wheeled configuration only changes its pose and uses its general position control algorithm. The module joint positions for the low height pose of the wheeled configuration is given in Table 3.32 and Figure 3.52 shows the visual representation of low height pose of the configuration.

Table 3.32 : Joint positions of the pass under obstacles pose of the wheeled configuraion.

Role	Part	Front Joint	BackJoint1	BackJoint2
1	Body Front	0	0	0
2	Body Back	0	0	0
3	Front Right Leg	0	0	30
4	Front Left Leg	0	0	30
5	Back Right Leg	0	0	30
6	Back Left Leg	0	0	30

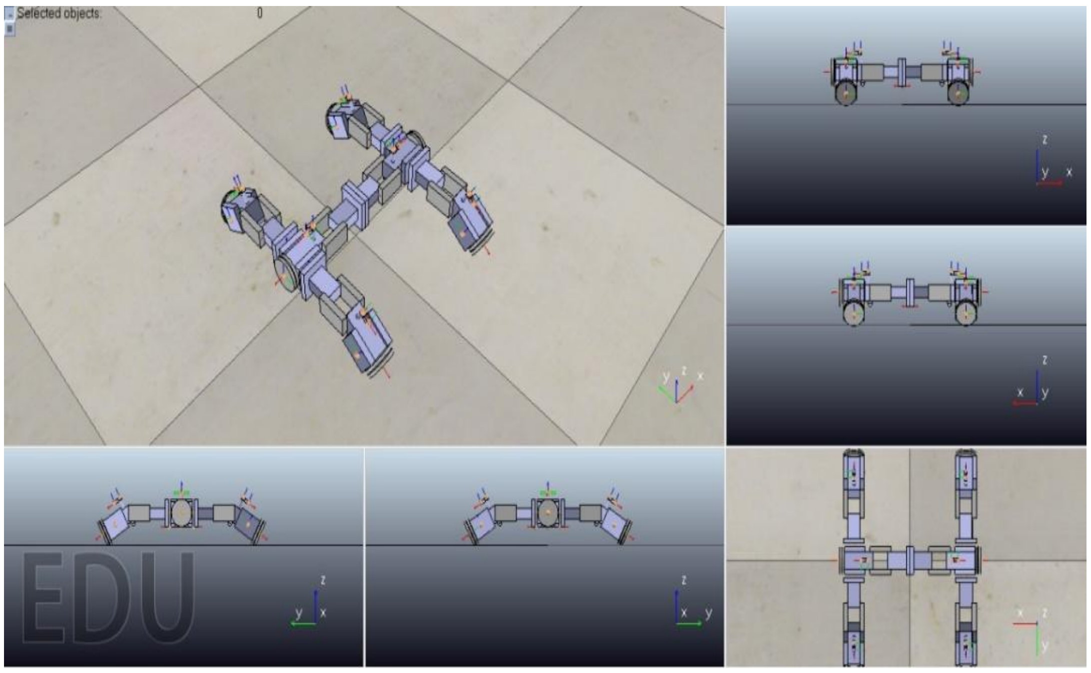


Figure 3.52: Low height pose of the wheeled configuration.

3.3 Strategic Planning

The strategic planning algorithm is the main control unit of the whole robotic structure. It determines the actions to be taken by the structure such as moving to a position, assembling or disassembling a configuration and executing configuration specific actions depending on the data received from the environment. As expressed before the strategic planning algorithm developed in this study is classified as a hybrid deliberative/reactive type robotic paradigm. The deliberative side of the algorithm is based on generating a plan that will drive the robotic structure from the initial state to the desired goal state while reactive part of the algorithm is more like a controller which is based on executing the plan generated in the deliberative layer. The general strategic planning algorithm is shown in Figure 3.53

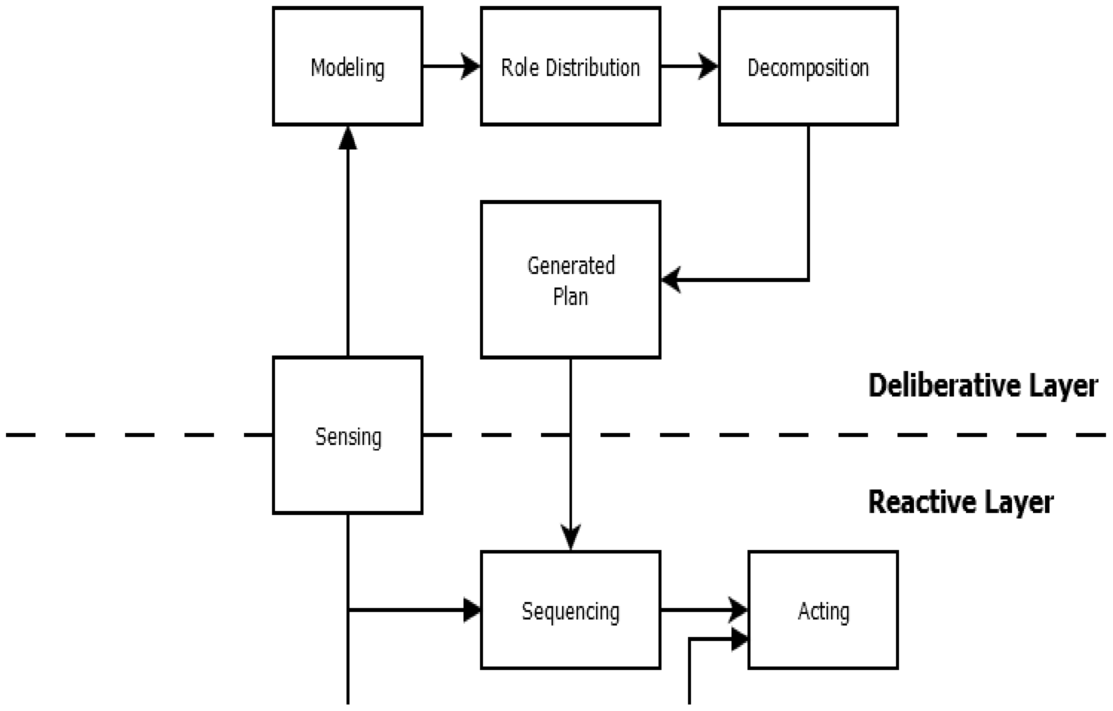


Figure 3.53: General strategic planning algorithm.

3.3.1 Deliberative layer

The deliberative layer of the strategic planning algorithm is for generating a feasible plan to drive the robotic structure from the initial state to the target state. Generation of a plan is completed in five steps. These steps are sensing, modeling, role distribution and decomposition. Figure 3.54 shows the general workflow of the deliberative layer.

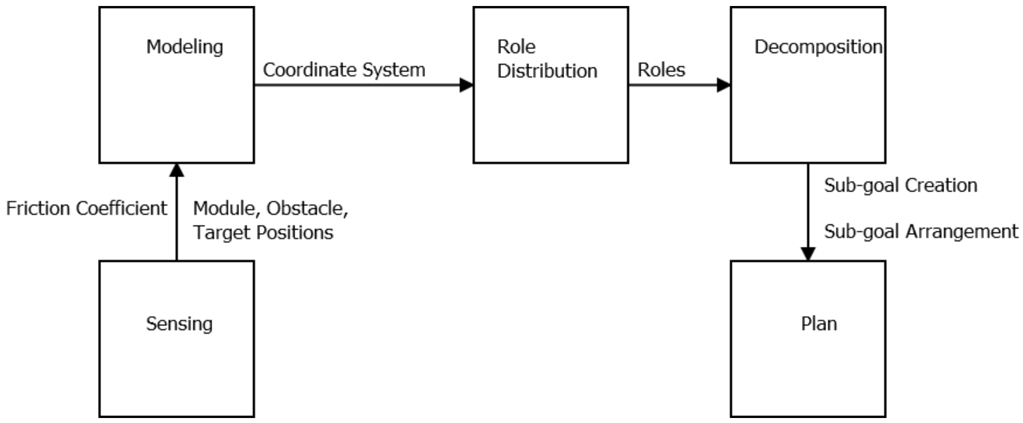
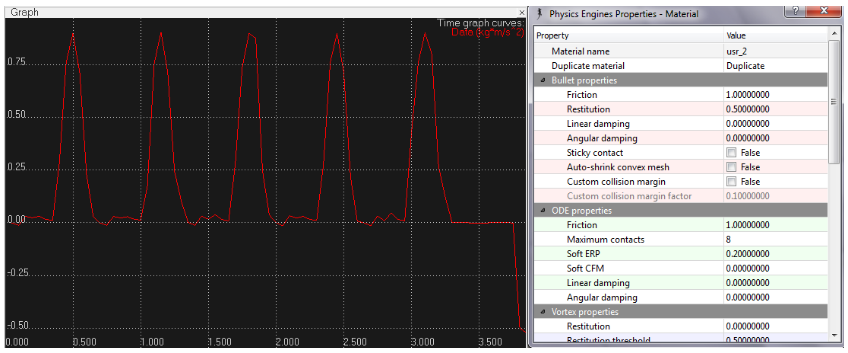


Figure 3.54: General workflow of deliberative layer.

3.3.1.1 Deliberative sensing phase

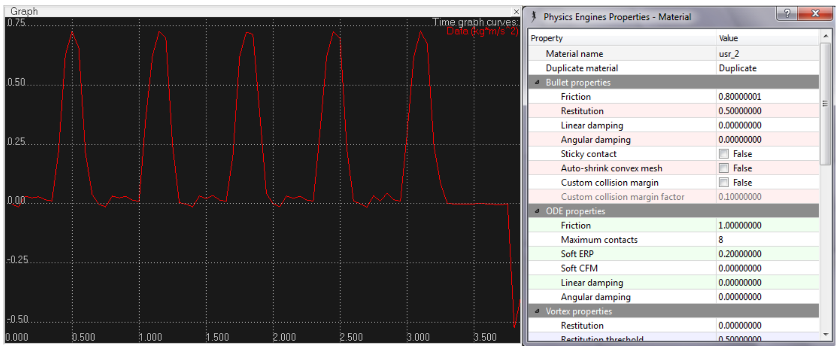
The sensing phase is about determining environmental conditions. This phase is vital for both the deliberative and the reactive layer of the strategic planning algorithm. It is essential for deliberative layer because generation of a plan to reach the desired goal state starts with getting the environmental data such as obstacle positions and friction coefficient of the terrain and the plan is generated according to the data received. Sensing phase is vital for the reactive part because in the reactive layer of the algorithm, actions taken by the robotic structure is actually reactions to the sensory data. Even if a plan is generated to reach the desired target position, sensing phase continues in the reactive layer of the algorithm as a part of the feedback mechanism.

The sensing phase of the deliberative layer starts with the friction coefficient estimation of the terrain. While estimating friction coefficient, each module takes five forward low velocity steps and reads the force sensors positioned between their wheel part when the module is in the reset pose. Figures 3.55 to 3.59 show the force sensor readings in terrains with different friction coefficients.



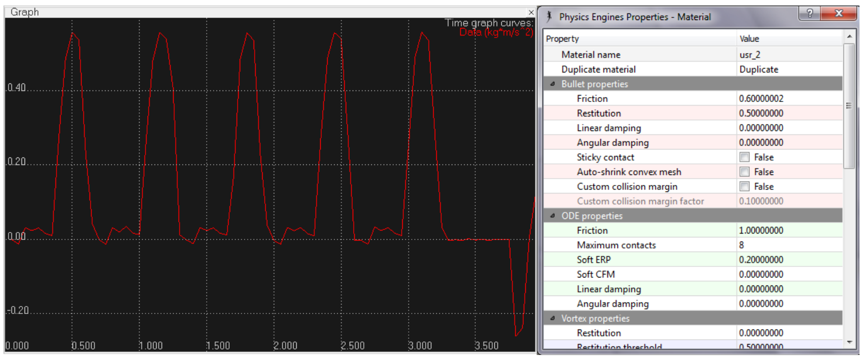
Property	Value
Material name	usr_2
Duplicate material	Duplicate
Bullet properties	
Friction	1.00000000
Restitution	0.50000000
Linear damping	0.00000000
Angular damping	0.00000000
Sticky contact	☐ False
Auto-shrink convex mesh	☐ False
Custom collision margin	☐ False
Custom collision margin factor	0.10000000
ODE properties	
Friction	1.00000000
Maximum contacts	8
Soft ERP	0.20000000
Soft CFM	0.00000000
Linear damping	0.00000000
Angular damping	0.00000000
Vortex properties	
Restitution	0.00000000
Restitution threshold	0.50000000

Figure 3.55: Force sensor readings when friction coefficient is 1.0.



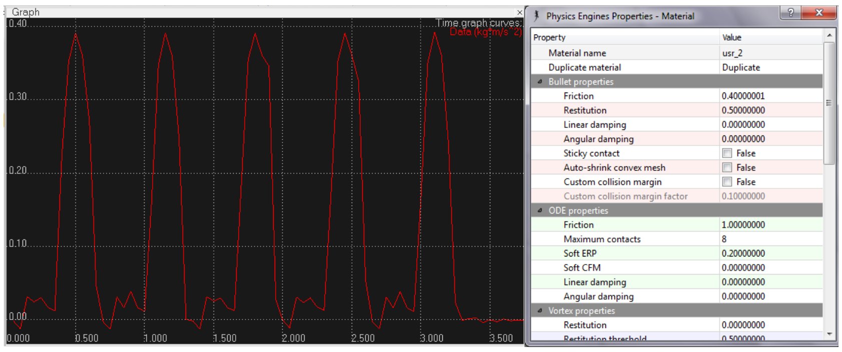
Property	Value
Material name	usr_2
Duplicate material	Duplicate
Bullet properties	
Friction	0.80000001
Restitution	0.50000000
Linear damping	0.00000000
Angular damping	0.00000000
Sticky contact	False
Auto-shrink convex mesh	False
Custom collision margin	False
Custom collision margin factor	0.10000000
ODE properties	
Friction	1.00000000
Maximum contacts	8
Soft ERP	0.20000000
Soft CFM	0.00000000
Linear damping	0.00000000
Angular damping	0.00000000
Vortex properties	
Restitution	0.00000000
Restitution threshold	0.50000000

Figure 3.56: Force sensor readings when friction coefficient is 0.8



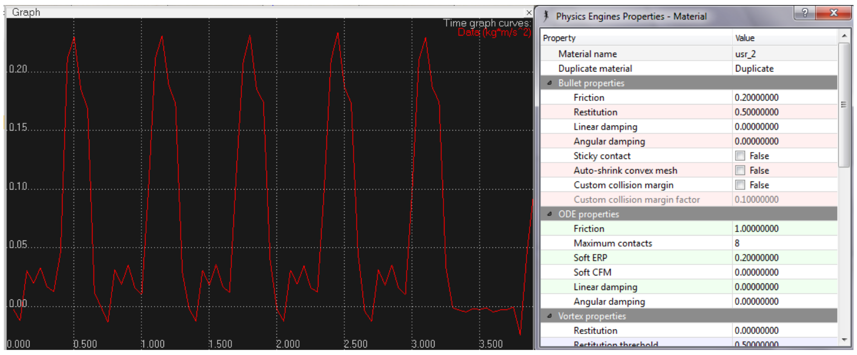
Property	Value
Material name	usr_2
Duplicate material	Duplicate
Bullet properties	
Friction	0.60000002
Restitution	0.50000000
Linear damping	0.00000000
Angular damping	0.00000000
Sticky contact	False
Auto-shrink convex mesh	False
Custom collision margin	False
Custom collision margin factor	0.10000000
ODE properties	
Friction	1.00000000
Maximum contacts	8
Soft ERP	0.20000000
Soft CFM	0.00000000
Linear damping	0.00000000
Angular damping	0.00000000
Vortex properties	
Restitution	0.00000000
Restitution threshold	0.50000000

Figure 3.57: Force sensor readings when friction coefficient is 0.6.



Property	Value
Material name	usr_2
Duplicate material	Duplicate
Bullet properties	
Friction	0.40000001
Restitution	0.50000000
Linear damping	0.00000000
Angular damping	0.00000000
Sticky contact	False
Auto-shrink convex mesh	False
Custom collision margin	False
Custom collision margin factor	0.10000000
ODE properties	
Friction	1.00000000
Maximum contacts	8
Soft ERP	0.20000000
Soft CFM	0.00000000
Linear damping	0.00000000
Angular damping	0.00000000
Vortex properties	
Restitution	0.00000000
Restitution threshold	0.50000000

Figure 3.58: Force sensor readings when friction coefficient is 0.4.



Property	Value
Material name	usr_2
Duplicate material	Duplicate
Bullet properties	
Friction	0.20000000
Restitution	0.50000000
Linear damping	0.00000000
Angular damping	0.00000000
Sticky contact	False
Auto-shrink convex mesh	False
Custom collision margin	False
Custom collision margin factor	0.10000000
ODE properties	
Friction	1.00000000
Maximum contacts	8
Soft ERP	0.20000000
Soft CFM	0.00000000
Linear damping	0.00000000
Angular damping	0.00000000
Vortex properties	
Restitution	0.00000000
Restitution threshold	0.50000000

Figure 3.59: Force sensor readings when friction coefficient is 0.2.

The maximum value read while the module has the reset pose is taken and recorded. This process is done for each step the module takes and at the end the average of these five readings is taken. The average of the reading is transformed to a friction coefficient estimation based on Table 3.33.

Table 3.33 : Friction estimation based on force sensor readings.

Friction coefficient estimated	Force sensor reading lower limit	Force sensor reading upper limit
1.0	0.75	-
0.8	0.65	0.75
0.6	0.45	0.65
0.4	0.30	0.45
0.2	-	0.30

After the friction coefficient of the terrain is estimated, the modules start the initial scan to localize other modules, obstacles and targets. In the initial scan each module moves its camera 360°, calculates positional difference of every module, obstacle or target it identified and records it. The data collected by each module is used in the modeling phase to create a coordinate system on which "Module#1" is centered. Note that "Module#1" is not the module which has the first role. Role distribution algorithm is initiated after the coordinate system is created in the modeling phase.

3.3.1.2 Modeling phase

The sensing phase ends with the localization scan in which each module scans its environment and collects positional data of other modules, obstacles or targets. In modeling phase the acquired data is used to create a coordinate system for the plan to be generated. The origin of the coordinate system is assumed to be the reference point of Module#1.

The creation process of the coordinate system starts from Module#1. This module uses the positional data it collected in the sensing phase and calculates the position of other objects it identified while scanning. After it calculates the position of the modules, it sends the positional data to each module it identified, marks them as scanned and notifies the scanned modules.

The scanned modules do what Module#1 does after they are notified and marked as scanned with simple differences. The scanned modules does not recalculate the positions of modules that are already marked as scanned and they calculate the positions of the unmarked modules based on their own received position.

While the scanned modules are notified and marked, for obstacles and targets this method does not hold. The calculated obstacle and target positions are shared with other modules by the module which identifies and calculates them. This is done because after role distribution, the strategic planning algorithm is initiated by only the module which has the first role. By sharing the obstacle and target position data, it is ensured that Role#1 knows the positions of these entities.

The modeling phase ends after every module is marked as scanned and all positional data regarding obstacles and targets are shared among the modules.

3.3.1.3 Role distribution phase

Role distribution phase is separated from the plan generation process because either the plan generated requires assembling wheeled or quadruped configuration, the role distribution will be the same. To avoid unnecessary communication protocols, role distribution is done before the execution of the strategic planning algorithm to ensure that Role#1 executes the strategic planning algorithm and knows the generated plan.

Role distribution algorithm is initiated by Module#1 and the rest of the modules are notified after the roles are determined. Role distribution is performed based on the positional data of the coordinate system created in the modeling phase. The first step of role distribution is to determine the module which will have the first role. To determine the first module the average of all module positions are taken and the closest module to this average position is chosen as Role#1.

After Role#1 is chosen the assembly positions that are given in Table() for quadruped configuration and Table() for wheeled configuration are calculated based on the orientation of Role#1. The pseudo code of the role distribution algorithm is given in Table 3.34

Table 3.34 : Pseudo code of the role distribution algorithm.

```
FOR(i=1,6,1)
---pos_x_sum += pos_x(Module#i)
---pos_y_sum += pos_y(Module#i)
ENDFOR
pos_x_avg =  pos_x_sum/6
pos_y_avg =  pos_y_sum/6
FOR(i=1,6,1)
---x_dif(i)= pos_x_avg -  pos_x(Module#i)
---y_dif(i)= pos_y_avg -  pos_y(Module#i)
```

```
   dist(i) = sqrt(x_dif(i)^2 + y_dif(i)^2)
ENDFOR
dist_min = min(dist(1), dist(2), dist(3), dist(4), dist(5), dist(6))
//Selection of Role#1
FOR(i=1,6,1)
   IF(dist_min==dist(i)) THEN
      Role#1 = Module#i
      gpos_x(1) = pos_x(Role#1)
      gpos_y(1) = pos_y(Role#1
   ENDIF
ENDFOR
FOR(i=2,6,1)
   gpos_x(i) = gpos_x_const(i) + gpos_x(1)
   gpos_y(i) = gpos_y_const(i) + gpos_y(1)
ENDFOR
//Selection of Role#2
FOR(i=1,6,1)
   x_dif(i)= gpos_x(2) -  pos_x(Module#i)
   y_dif(i)= gpos_y(2) -  pos_y(Module#i)
   dist(i) = sqrt(x_dif(i)^2 + y_dif(i)^2)
   IF(Module#i==Role#1) THEN
      dist(i)=HUGE
   ENDIF
ENDFOR
dist_min = min(dist(1), dist(2), dist(3), dist(4), dist(5), dist(6))
FOR(i=1,6,1)
   IF(dist_min==dist(i)) THEN
      Role#2 = Module#i
   ENDIF
ENDFOR
//Selection of Role#3
FOR(i=1,6,1)
   x_dif(i)= gpos_x(3) -  pos_x(Module#i)
   y_dif(i)= gpos_y(3) -  pos_y(Module#i)
   dist(i) = sqrt(x_dif(i)^2 + y_dif(i)^2)
   IF(Module#i==Role#1 OR Role#2) THEN
      dist(i)=HUGE
   ENDIF
ENDFOR
dist_min = min(dist(1), dist(2), dist(3), dist(4), dist(5), dist(6))
FOR(i=1,6,1)
   IF(dist_min==dist(i)) THEN
      Role#3 = Module#i
   ENDIF
ENDFOR
//Selection of Role#4
FOR(i=1,6,1)
   x_dif(i)= gpos_x(4) -  pos_x(Module#i)
   y_dif(i)= gpos_y(4) -  pos_y(Module#i)
   dist(i) = sqrt(x_dif(i)^2 + y_dif(i)^2)
```

```
    IF(Module#i==Role#1 OR Role#2 OR Role#3) THEN
        dist(i)=HUGE
    ENDIF
ENDFOR
dist_min = min(dist(1), dist(2), dist(3), dist(4), dist(5), dist(6))
FOR(i=1,6,1)
    IF(dist_min==dist(i)) THEN
        Role#4 = Module#i
    ENDIF
ENDFOR
//Selection of Role#5
FOR(i=1,6,1)
    x_dif(i)= gpos_x(5) -  pos_x(Module#i)
    y_dif(i)= gpos_y(5) -  pos_y(Module#i)
    dist(i) = sqrt(x_dif(i)^2 + y_dif(i)^2)
    IF(Module#i==Role#1 OR Role#2 OR Role#3 OR Role#4) THEN
        dist(i)=HUGE
    ENDIF
ENDFOR
dist_min = min(dist(1), dist(2), dist(3), dist(4), dist(5), dist(6))
FOR(i=1,6,1)
    IF(dist_min==dist(i)) THEN
        Role#5 = Module#i
    ENDIF
ENDFOR
//Selection of Role#6
FOR(i=1,6,1)
    x_dif(i)= gpos_x(6) -  pos_x(Module#i)
    y_dif(i)= gpos_y(6) -  pos_y(Module#i)
    dist(i) = sqrt(x_dif(i)^2 + y_dif(i)^2)
    IF(Module#i==Role#1 OR Role#2 OR Role#3 OR Role#4 OR Role#5) THEN
        dist(i)=HUGE
    ENDIF
ENDFOR
dist_min = min(dist(1), dist(2), dist(3), dist(4), dist(5), dist(6))
FOR(i=1,6,1)
    IF(dist_min==dist(i)) THEN
        Role#6 = Module#i
    ENDIF
ENDFOR
```

3.3.1.4 Decomposition phase

When role distribution is done, decomposition phase is initiated by Role#1. In decomposition phase, the target state of the system is decomposed, creating sub-goals based on the model of the environment. Decomposition is a forward process, meaning that it starts from the initial state of the robotic structure and continues creating sub-goals until the target state is reached.

The subgoals are represented as arrays of 6 values in the script written in V-Rep. The values in the array are desired configuration, V-Rep handle of the reference object, desired lateral difference, desired longitudinal distance, desired gamma difference and transportation mode.

Desired configuration can be 1 or 2 depending on the configuration required by the plan. For desired configuration, "1" means quadruped configuration and "2" means wheeled configuration. V-Rep handle of reference object is the identity of the object that the configuration is using as a reference. In V-Rep every entity has a handle that identifies it in the scripts. V-Rep handle of modules, obstacles and targets are collected while scanning in the sensing phase and they are used for searching and locking to the reference objects by script commands.

Desired lateral and longitudinal difference values are straightforward. The configuration determines the target state position using these values. Similar to lateral and longitudinal differences, gamma difference is the desired gamma difference between the target object and the configuration. The target orientation is determined by using this value.

Transportation mode determines whether the configuration will execute its special ability while trying to reach the target position, or not. This value can be 0 or 1. 0 means the configurations transport normally, executing their position control algorithms. When transport mode value is 1, quadruped walker just executes the pass over function to pass over the ground obstacle and wheeled configuration changes its pose to pass under the obstacle and executes its normal position control algorithm.

The sub-goals are created in a group based on the obstacle types. If there is a ground obstacle in the surroundings, the sub-goal group created for this obstacle consists of three sub-goals. The fisrt sub-goal is created to position the configuration facing some distance away from the ground obstacle. The second sub-goal is used to execute the pass over obstacle ability of the configuration. This sub-goal’s transport_mode value is 1, so the configuration executes the pass over obstacle function and gets the next sub-goal. The third sub-goal is created for that reason. Similar to the first sub-goal, the third sub-goal is used to position the configuration some distance away from the obstacle after passing over it. Position and orientation of the configuration while passing over obstacles cannot be controlled and after the function is executed there should be another sub-goal to set the configuration on track. This is the reason for having three sub-goals for a ground obstacle.

If there is a lath obstacle in the simulation scene, the group created for this obstacle consists of two sub-goals because there is no need for an extra sub-goal to correct position and orientation errors while passing under the obstacle. Passing under obstacle ability is just an adjustment of the pose of the wheeled configuration code and the configuration is driven by the position control algorithm. Therefore there is no position or orientation errors while passing under the obstacle. Similar to the ground obstacle case the first sub-goal lets the configuration to position itself some distance away from the lath obstacle facing it. The second sub-goal can be seen as the combination of the second and the third sub-goals of the ground obstacle case. This sub-goal requests the configuration to be position some distance away from the obstacle facing away from it and its transport_mode value is 1.

The ordered arrays which represent sub-goal states form a matrix which is the representation of a plan in V-Rep scripts. Ordering the sub-goal states is done based on the obstacle distance between the goal state and the obstacles. The first rows of the plan matrix are consisted of the sub-goal states that are related to the farthest obstacle and the last row of the plan matrix is the goal state of the system. Representing the plan as a matrix is useful because in the sequencing phase of the reactive layer, the target states that are fed to the acting unit are the rows of this matrix and when a target state is reached the only reaction of the sequencer is feeding the next row of the plan matrix to the acting phase.

3.3.2 Reactive layer

Reactive layer of the strategic planning algorithm acts as a feedback controller to execute the generated plan to reach the goal state. Like a feedback controller the reactive control consists of a continuous sensing and acting loop. There is also a scheduling part in addition to traditional sense-act architecture of reactive paradigm. The scheduler takes the data collected in the sensing phase and interprets them based on the state of the robotic structure and the target state. Based on this interpretation, the joints are driven according to requirements in the acting phase. Target states are determined by the scheduler due to the sub-goals of the generated plan in the deliberative planning phase. The scheduler also determines the target state based on the sub-goal that the robotic structure is trying to achieve. Figure 3.60 shows general workflow of reactive layer.

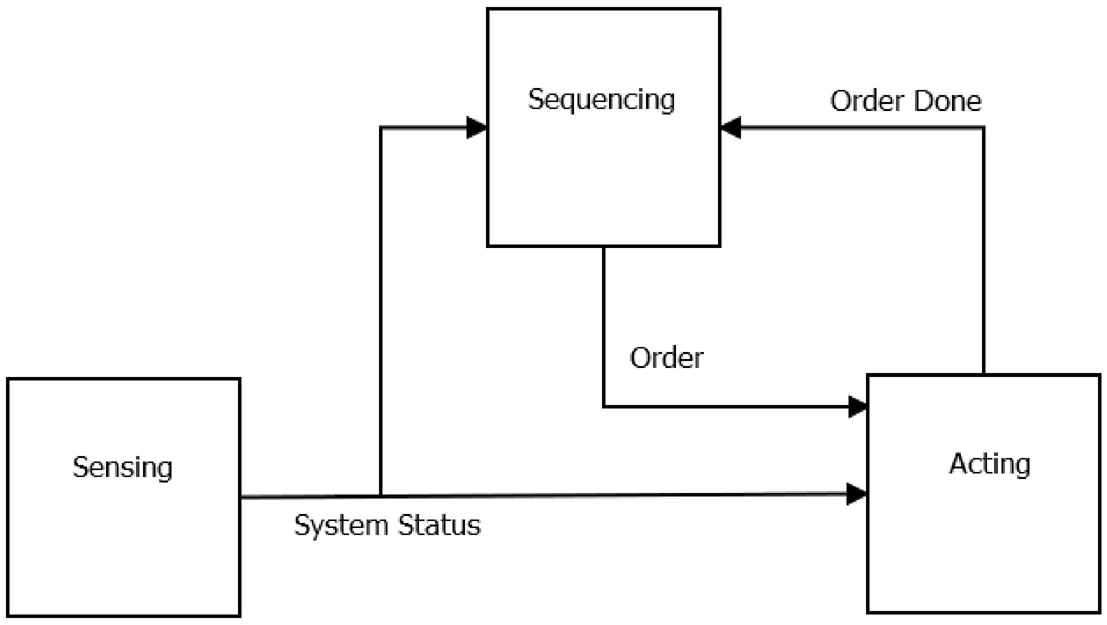


Figure 3.60: Reactive layer workflow.

3.3.2.1 Reactive sensing phase

The sensing phase of the reactive layer is used like the sensor of the feedback control mechanism. In this phase the visual sensor of Role#1 is always active. It is locked to the target position until it is reached. In this phase the orientation and distance values are measured and sent to two different units. The first unit is the sequencing unit of the reactive layer to check if the system has reached to the desired sub-goal state. The second unit is the acting unit of the reactive layer to transfer sensor data into motor signals depending on the positional error.

Unlike the deliberative sensing phase, sensing in reactive layer is continuous. The sensing phase is always active, locking to a reference object or if not locked searching for it and continually feeding data to the sequencing and acting unit.

3.3.2.2 Sequencing phase

The sequencing unit of the reactive layer is used as a low level action based decision mechanism. The main function of the sequencing unit is to keep the acting part of the reactive layer on track. The plan generated in the deliberative layer is shared with the sequencer and after getting the plan, the sequencer organizes the relationship between the sensing unit and the acting unit of the reactive layer.

The organization is done by comparing the sensory data with the sub-goal state requirements. If the configuration requirement of the sub-goal does not match with the status of the system, the sequencer issues a reassembly command to the acting unit. If the transportation mode requires the use of the special ability of the configuration the sequencer calls the corresponding functions to control the acting unit. If the configurations match and there is no request for a special ability, the sequencer issues no commands to the acting unit and lets it to position itself based on the sensory data.

When the sub-goal state is reached, the sequencer gets the next row of the plan matrix and continues comparing until the goal state is reached. Figure 3.61 shows the flowchart of the sequencing unit.

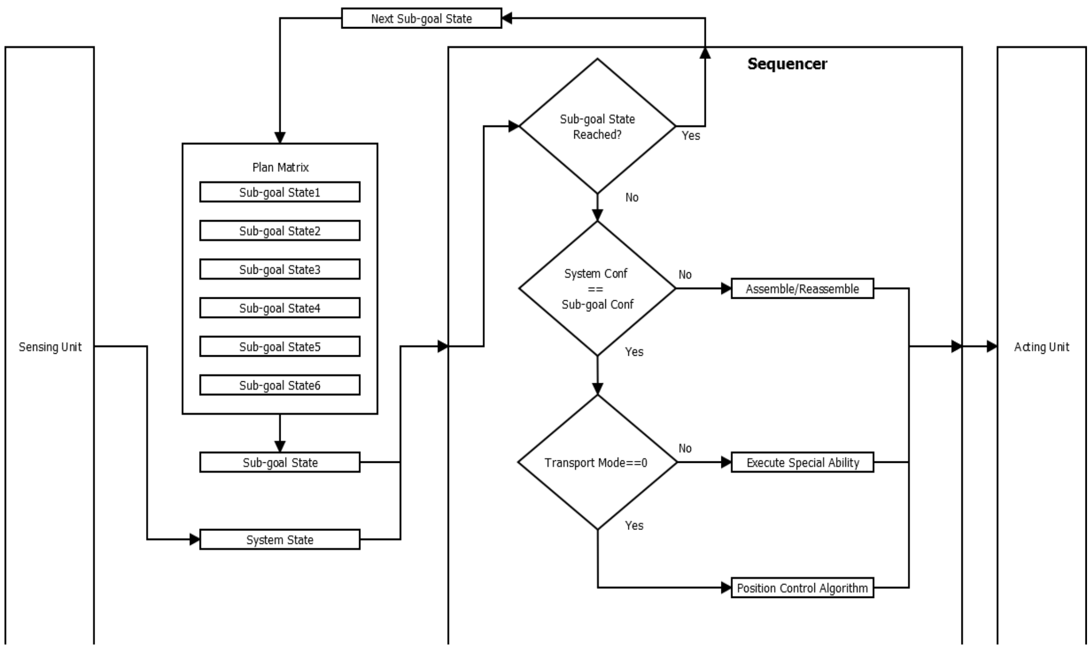


Figure 3.61: Flowchart of the sequencing unit.

If the requirements are not satisfied the sequencer does nothing, but if they are satisfied, the sequencer changes the reference value of the acting unit if the subgoal has a positioning target. If the subgoal is assembly or ability targeting, the sequencing unit sends the appropriate order for the acting unit to execute the appropriate protocol.

3.3.2.3 Acting phase

Acting phase of the reactive control layer is simply the execution phase of the appropriate actions, depending on the sensory data and the sub goal target. The actions to be executed depends on the sub goal of the plan the robot is following. If the subgoal has a positioning target, the executed action will be adjusting foot positions based on the gait for the quadruped configuration or setting wheel speeds for the wheeled configuration.

The acting unit controls the states of three joints and four connection points of all the modules in the system. Therefore any action requiring an adjustment on the statesof these joints and connection points needs to be executed on the acting unit.

Acting phase is not just an actuation phase. Acting unit not only controls joint positions, but also executes protocols and processes ordered by the sequencing unit. The orders executed by the acting unit are processes like assembly/reassembly, configuration position control and ability execution. The acting unit also checks if the issued order is satisfied and notifies the sequencer to issue another order.

4. SIMULATION AND TEST RESULTS

4.1 Test Scene

To test the modular robotic system and the strategic planning algorithm, a test scene is created in V-Rep which consists of two obstacles and a target dummy. One of the obstacles is a ground obstacle which can be passed over in a quadruped configuration and the other obstacle is a lath obstacle which can be passed under in a wheeled configuration. The test scene is shown in figures 4.1 and 4.2.

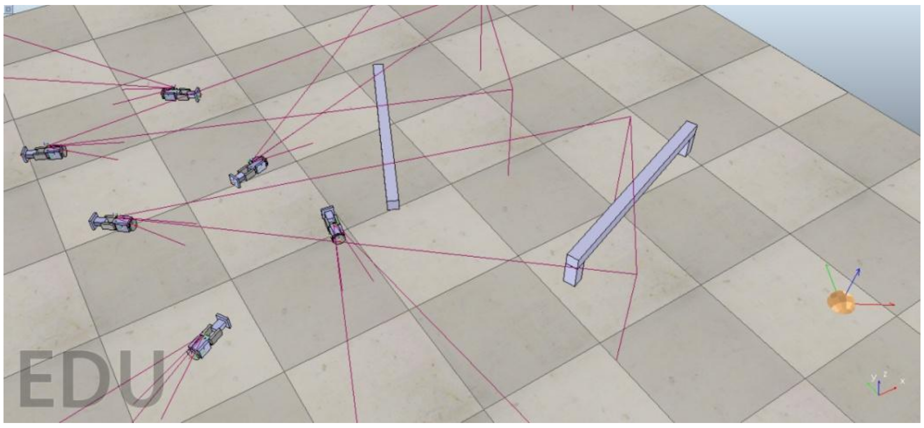


Figure 4.1 : Test scene created in V-Rep.

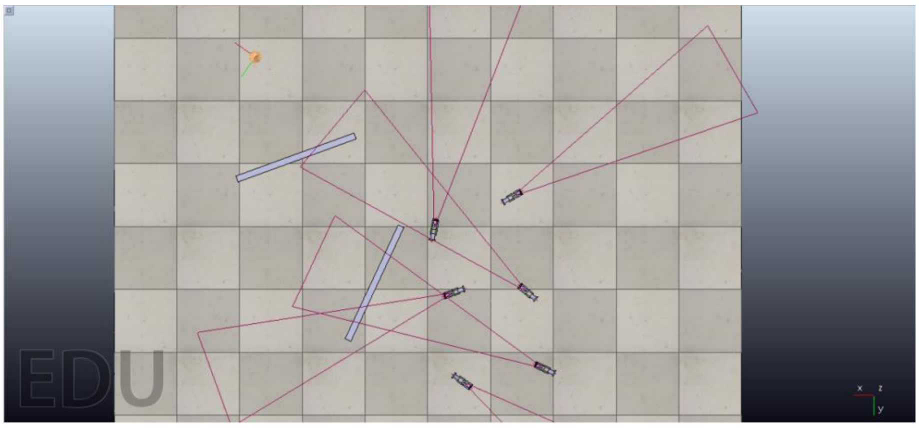


Figure 4.2 : Top view of the test scene.

The test scene is created to prove the strategic planning algorithm works as intended and lets the system reach the target state. For that purpose the robotic system should;

- Scan its surroundings and identify the modules, the target and the obstacles,

- Model the environment in a coordinate system,
- Execute the role distribution and strategic planning algorithm to generate a plan using the model,
- Execute the generated plan by assembling/reassembling, controlling its position-orientation in quadruped or wheeled configurations and executing the pass over and pass under obstacle abilities when needed,
- Have the state that is defined by the target dummy at the end of the simulation.

To check the system status during simulation, some additions are made to the original code to get some logging information. The modules are positioned randomly, but in a way that lets Module#1 to be chosen as Role#1. This is done to unite the logging data given by Module#1 and Role#1 in a single console window. In addition to script based log data, the internal graph system of V-Rep is used to keep track of the reference position of Module#1.

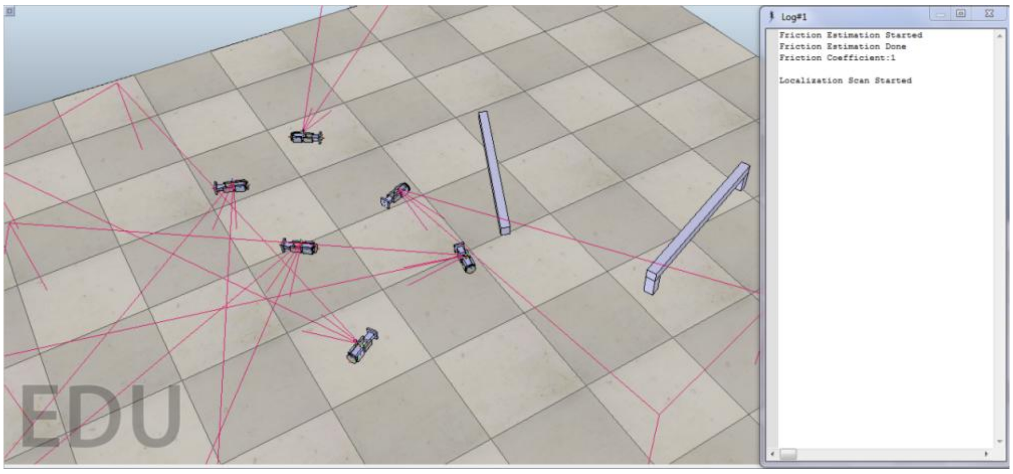
All of the six modules have the same code written in their scripts which only differs by module specific variable definitions. Besides these variables, each variable, function or method is identical in each script. The script is given in Appendix-A with comments and explanations, but the important functions and methods used in the scripts will also be explained briefly in this section.

4.2 The Simulation

The simulation starts with the friction coefficient estimation which is the first part of the deliberative sensing phase. The modules take five forward low velocity steps and read their force sensors to estimate the friction coefficient of the terrain.

In the programming perspective, this phase mainly depends on the function estimate_friction(). This function is continually called in the friction estimation phase. Since it is called continually it takes its internal parameters as input and returns them as output to be used again.

The script logger gives only the estimated friction coefficient in this phase of the simulation. Figure 4.3 shows Module#1 log.

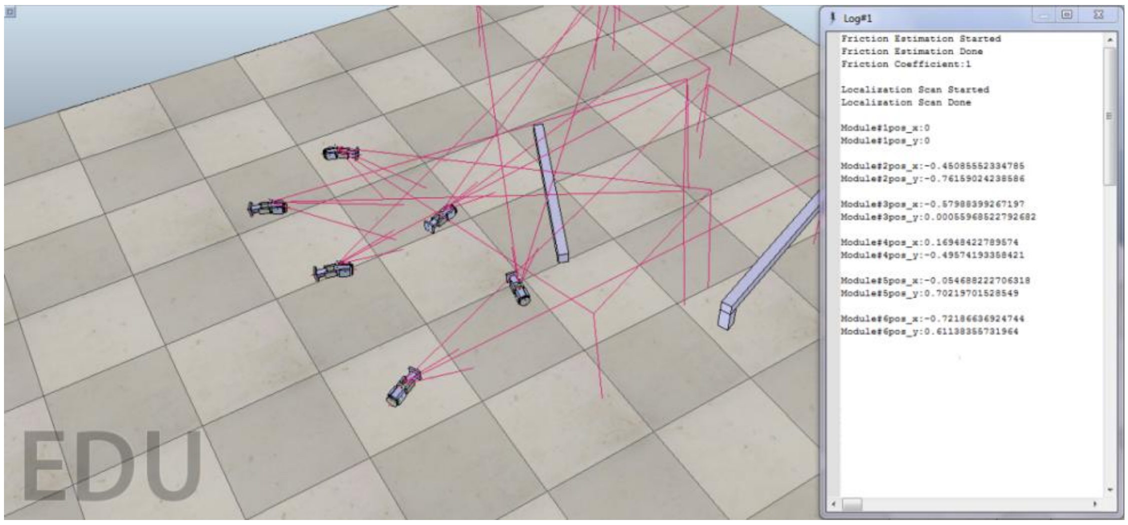


```
Friction Estimation Started
Friction Estimation Done
Friction Coefficient:1

Localization Scan Started
```

Figure 4.3 : Module#1 log on friction coefficient estimation

After friction estimation is completed, the modules start the localization scan to identify other modules, obstacles and targets in their surroundings. In this part of the sensing phase every module is marked as scanned and their positions are shared throughout the system.

In localization scan part, initial_scan() function called continually to control the visual sensor orientation of the modules, read the result and record the position data. The log output of Module#1 in this part is module, obstacle and target positions. The console window showing log output is given in Figure 4.4



```
Friction Estimation Started
Friction Estimation Done
Friction Coefficient:1

Localization Scan Started
Localization Scan Done

Module#1pos_x:0
Module#1pos_y:0

Module#2pos_x:-0.45085552334785
Module#2pos_y:-0.76159024238586

Module#3pos_x:-0.57988399267197
Module#3pos_y:0.00055968522792682

Module#4pos_x:0.16948422789574
Module#4pos_y:-0.49574193358421

Module#5pos_x:-0.054688222706318
Module#5pos_y:0.70219701528549

Module#6pos_x:-0.72186636924744
Module#6pos_y:0.61138355731964
```

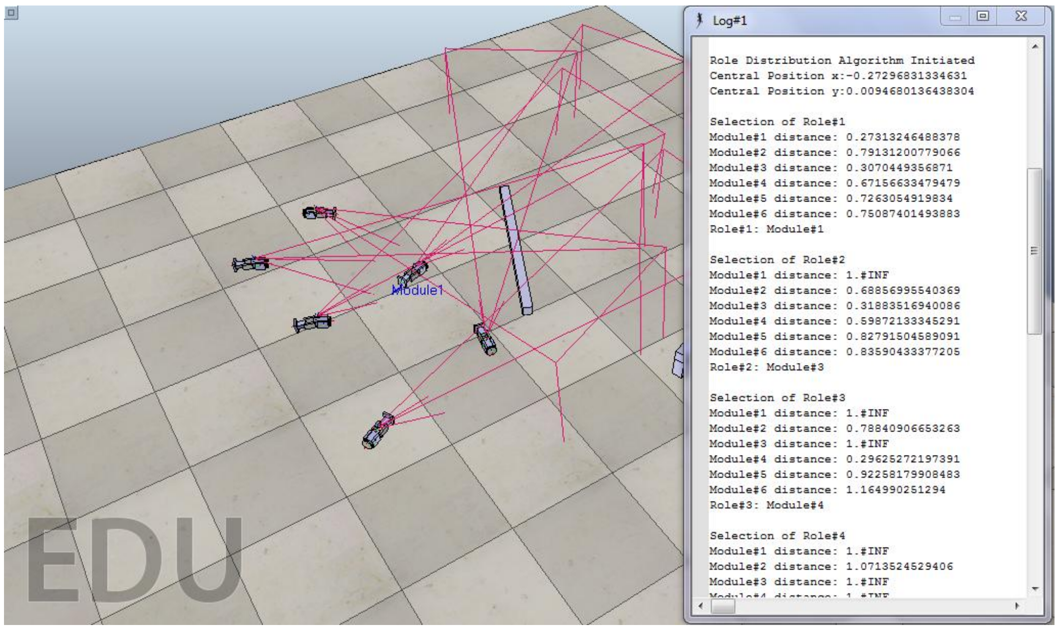
Figure 4.4 : Module#1 log on localization scan.

When the initial scan is completed, the sensing phase of the deliberative layer ends and strategic planning starts. The robotic structure starts distributing roles because either the plan generated requires assembling wheeled or quadruped configuration, the role distribution will be the same. To avoid unnecessary communication protocols, role distribution is done before executing decomposition part of the strategic planning algorithm to ensure that Role#1 executes decomposition algorithm and knows the generated plan.

As stated before, role distribution is done based on the module positions in the simulation scene. The role distribution algorithm is initiated by Module#1 and the function role_distribution() is used. This function is executed in a single call and it does the necessary calculations like central position of module group, assembly positions and distance of modules to assembly positions.

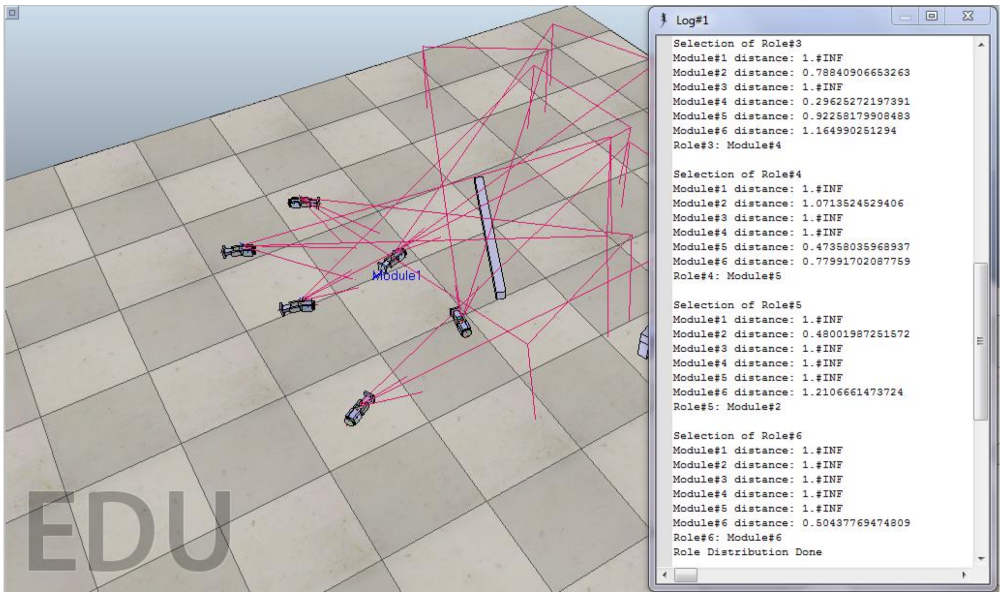
The function first calculates the central position of the module group. Then it starts calculating the distance between this central position and each module. The closest module to this central position is selected as Role#1. After the first role is given, the assembly positions are calculated based on the position of Role#1. Selection of other roles is carried on similar to selection of Role#1. The distances between module positions and role assembly positions are calculated and each role is given to the module which has the lowest distance starting from Role#2 to Role#6.

When the calculations and role distribution is done, Module#1 shares the role distribution result using script simulation parameters and notifies other modules. The calculations and role distributions are also given as log output by Module#1. The console window showing log output of role distribution is given in Figure 4.5 and Figure 4.6.



```
Role Distribution Algorithm Initiated
Central Position x:-0.27296831334631
Central Position y:0.0094680136438304

Selection of Role#1
Module#1 distance: 0.27313246488378
Module#2 distance: 0.79131200779066
Module#3 distance: 0.3070449356871
Module#4 distance: 0.67156633479479
Module#5 distance: 0.7263054919834
Module#6 distance: 0.75087401493883
Role#1: Module#1

Selection of Role#2
Module#1 distance: 1.#INF
Module#2 distance: 0.68856995540369
Module#3 distance: 0.31883516940086
Module#4 distance: 0.59872133345291
Module#5 distance: 0.82791504589091
Module#6 distance: 0.83590433377205
Role#2: Module#3

Selection of Role#3
Module#1 distance: 1.#INF
Module#2 distance: 0.78840906653263
Module#3 distance: 1.#INF
Module#4 distance: 0.29625272197391
Module#5 distance: 0.92258179908483
Module#6 distance: 1.164990251294
Role#3: Module#4

Selection of Role#4
Module#1 distance: 1.#INF
Module#2 distance: 1.0713524529406
Module#3 distance: 1.#INF
Module#4 distance: 1.#INF
```

Figure 4.5 : Module#1 log on role distribution part 1.



```
Selection of Role#3
Module#1 distance: 1.#INF
Module#2 distance: 0.78840906653263
Module#3 distance: 1.#INF
Module#4 distance: 0.29625272197391
Module#5 distance: 0.92258179908483
Module#6 distance: 1.164990251294
Role#3: Module#4

Selection of Role#4
Module#1 distance: 1.#INF
Module#2 distance: 1.0713524529406
Module#3 distance: 1.#INF
Module#4 distance: 1.#INF
Module#5 distance: 0.47358035968937
Module#6 distance: 0.77991702087759
Role#4: Module#5

Selection of Role#5
Module#1 distance: 1.#INF
Module#2 distance: 0.48001987251572
Module#3 distance: 1.#INF
Module#4 distance: 1.#INF
Module#5 distance: 1.#INF
Module#6 distance: 1.2106661473724
Role#5: Module#2

Selection of Role#6
Module#1 distance: 1.#INF
Module#2 distance: 1.#INF
Module#3 distance: 1.#INF
Module#4 distance: 1.#INF
Module#5 distance: 1.#INF
Module#6 distance: 0.50437769474809
Role#6: Module#6
Role Distribution Done
```

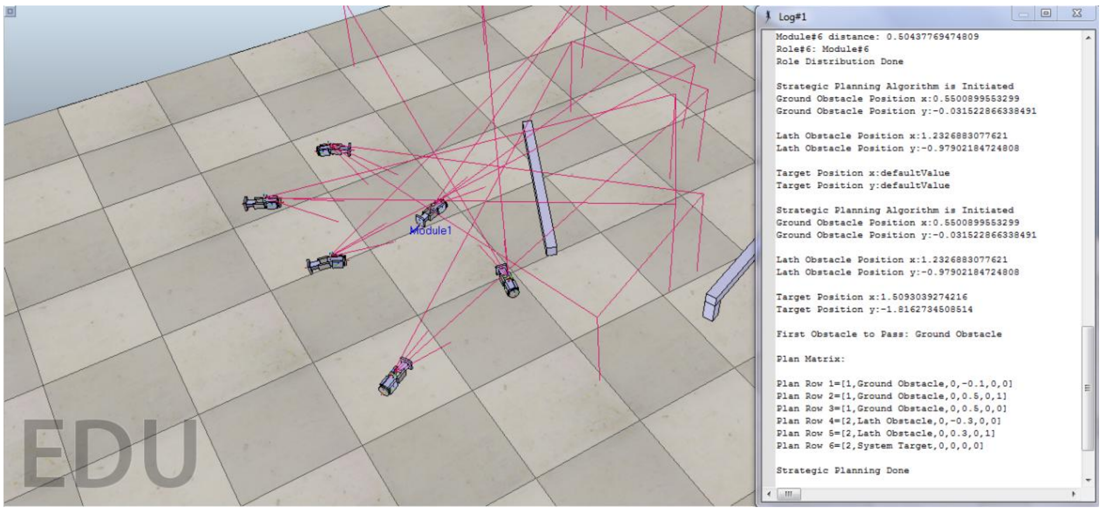
Figure 4.6 : Module#1 log on role distribution part 2.

As it can be seen from the figure, each role is given to the module which is the closest to the assembly position of the role. After role distribution is completed and Role#1 is determined. The modules are notified by Module#1 to get their roles. The modules use get_role() function of the script to learn their roles. This function is pretty simple. It only checks the script simulation parameters of Module#1 and finds the role of the module calling the function and returns this value.

When each module is notified and got its role in the system, Role#1 starts initiating the decomposition part of the strategic planning algorithm. This part is the main part of the strategic planning algorithm that generates the plan according to target and obstacle positions. In this part strategic_planning() function is used. Similar to role_distribution() function, this function is executed in a single call. As stated before decomposition process creates sub-goal arrays to reach the target state and depending on the obstacle and target positions, arranges them in an order and creates a plan matrix.

In the test scene the ground obstacle is closer to the robotic system, so it is the first obstacle to pass. Therefore passing over obstacle sub-goal group arrays are placed in the first three rows of the plan matrix. The ground obstacle is followed by the lath obstacle, so the fourth and the fifth rows of the plan matrix are the sub-goal group of the lath obstacle. The sixth and the last row of the plan matrix is the target state of the system.

After decomposition is done, the plan matrix is created by uniting the sub-goal arrays based on the obstacle and target positions. Role#1 also gives log output of the generated plan. The log output is shown in Figure 4.7



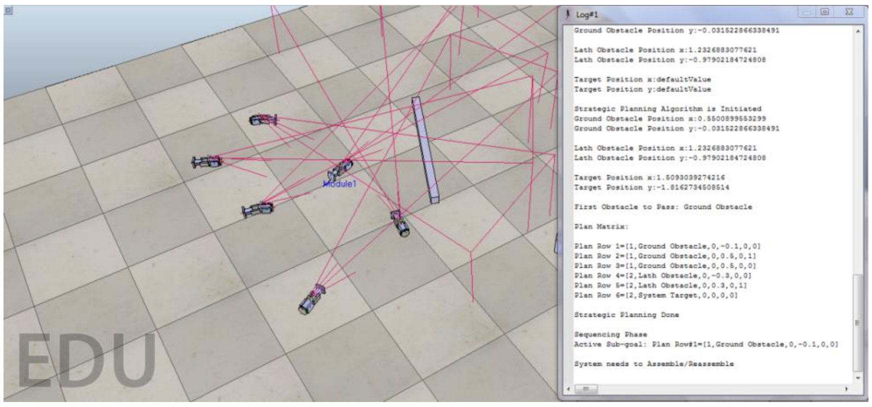
```
Module#6 distance: 0.50437769474809
Role#6: Module#6
Role Distribution Done

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done
```

Figure 4.7 : Decomposition phase and creation of the plan matrix.

The deliberative layer function ends when the decomposition part is passed and reactive layer is activated after the plan is generated. The processing work in the reactive layer is carried on by the sequencing unit which gets the sub-goal state of the plan and compares it with the system status and issues proper commands to the acting unit.

When the reactive layer is activated, sequence() function is called by Role#1 to get the active goal-state. This function compares the configuration required by the sub-goal state and the actual configuration and if needed executes the assembly/reassembly process. If the required and actual configurations are same, then the sequencer compares the transportation mode requested by the goal-state. If the transportation mode needs a special ability (transport_mode variable is equal to 1) then the sequencer executes the required special ability of the configuration whether it is passing over or under obstacles. If there is no request for a special ability, then the sequencer executes normal position control algorithm of the configuration using lateral difference, longitudinal difference and gamma difference parameters of the sub-goal state. After the sequence() function is executed by Role#1, other modules are notified and they also call sequence() function in their scripts to get the issued commands and execute them. sequence() function also shares the active state via the log output.

In the test scene, sequence() function is called by Role#1 to get the active state after the plan is generated. The active state shown by the log output is given in Figure 4.8



```
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
```

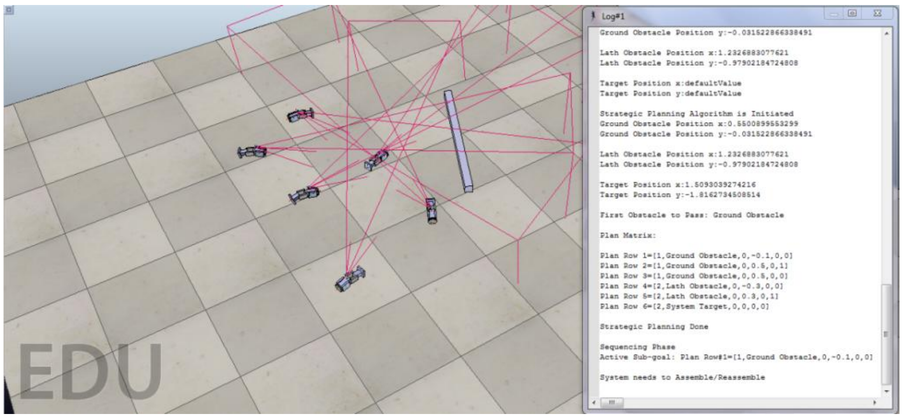
Figure 4.8 : Active state fetched by the sequence() function, Plan Row 1.

The active sub-goal state is [1, Ground Obstacle, 0, -0.1, 0, 0]. Since the required configuration and the actual configuration of the robotic system do not match, the sequencer issues an assembly/reassembly command. The assembly process is mainly based on single module movement and position control. For this part there are ten functions created in the script. These functions are used for getting the target state of a module based on its role, searching the reference target of the role, locking to the target and calculating difference, direction and velocity decisions, position and orientation control and connection.

Function get_target_state() is used for getting the target of a single module that has a role other than #1. This function takes configuration and role as input and returns target handle, lateral, longitudinal and gamma difference. After getting the target handle, the module calls search_target() and lock_target() functions to find and lock the its target. search_target() function is called continually until the module finds its target. Until the target is found, search_target() function rotates the visual sensor. When the module finds its target, lock_target() function is called continually to fix the visual sensor pointing the target. Functions search_target() and lock_target() are complements of each other. When the target is not found, the module calls search_target() and searches the target continually and when it finds its target, it stops calling search_target() and calls lock_target() until the sub-goal state is reached. When the visual sensor of the module locks to its target, the module calculates the difference with its target using calculate_difference() function. This function takes target handle as input and using the output of the visual sensor, calculates the difference between the target position and module reference position dummy.

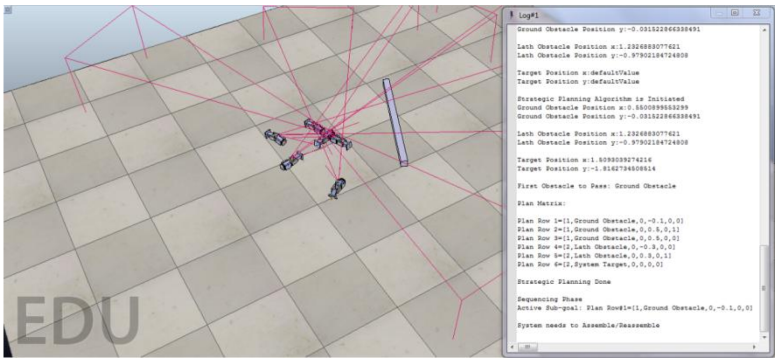
After the difference between the target and the module is calculated, the module decides the gait and velocity to implement. The functions decide_direction() and decide_velocity() are used to determine the appropriate gait. When the gait is determined, the module calls correct_ornt() function to adjust its orientation to the calculated orientation required to reach the module target. correct_ornt() function takes orientation difference as input and controls the wheel velocity depending on the difference. The modules call move_single() function to move towards the decided direction with the decided velocity after the orientation difference with the required orientation is 0°. move_single() and correct_ornt() functions are also called continually until the module target is reached. Therefore they also take their internal parameters as inputs and returns these values in the same time.

When the modules arrive to their first assembly positions, they check if they are allowed to move to their next assembly target by calling assembly_step_up() function. This function is always true for Role#2, Role#3, Role#4, meaning they can immediately move to their next assembly targets. Role#5 and Role#6 has to wait for Role#2 to connect to its target. Modules write 1 to their script simulation parameters called "connected" when they complete connection with their target, so when called by Role#5 or Role#6, assembly_step_up() function checks "connected" status of Role#2 and becomes true if it is 1 and false if it is not. After the modules reach to their last assembly positions, they call function connect() and their corresponding dummies create "dynamics, overlap constraint" type of link. Figures 4.9 to 4.12 show the visual representation of the assembly phase in the test scene.



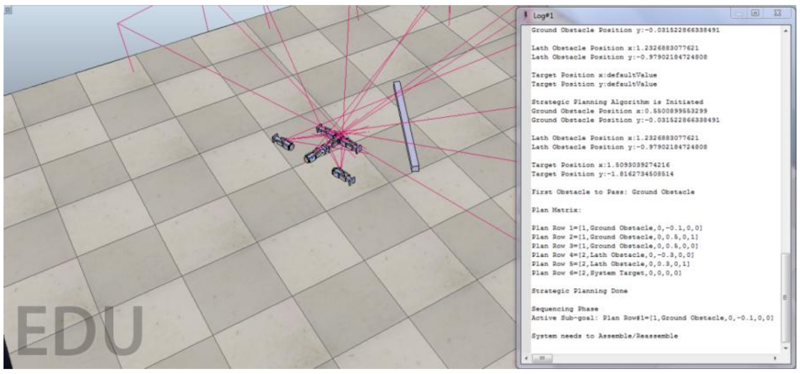
```
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
```

Figure 4.9 : Assembly phase in the test scene part 1.



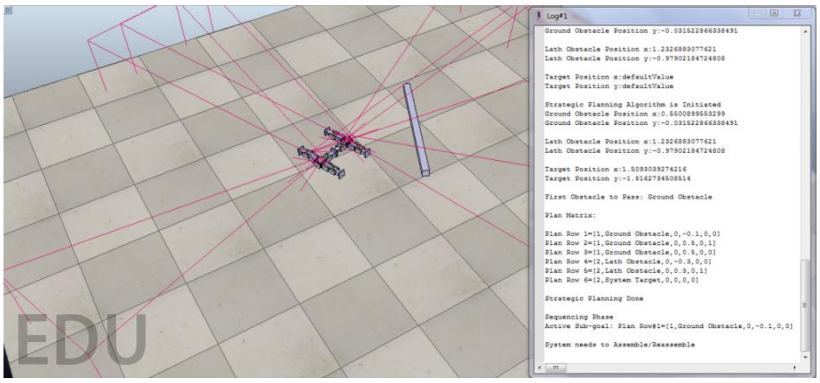
```
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
```

Figure 4.10 : Assembly phase in the test scene part 2.



```
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
```

Figure 4.11 : Assembly phase in the test scene part 3.

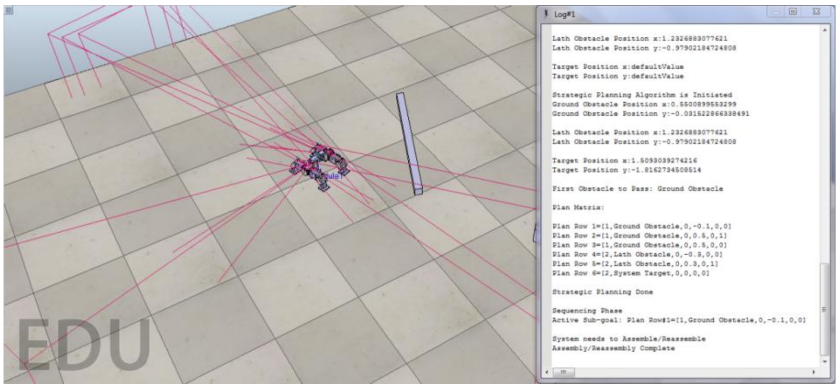


```
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
```

Figure 4.12 : Assembly phase in the test scene part 4.

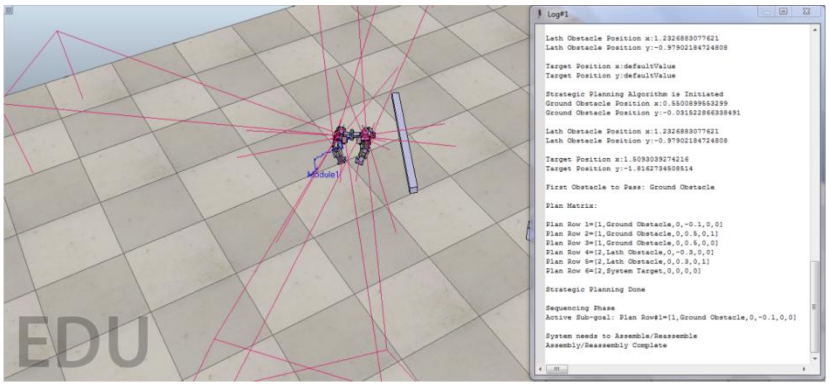
When the assembly of quadruped configuration is done, the sequencer compares the sub-goal state and the system status. Since the configurations match and transportation_mode is 0, the sequencer executes normal position control algorithm. Based on the position control algorithm of the quadruped configuration, the system moves towards the ground obstacle. In this part search_target(), lock_target(), calculate_difference(), qd_walk() and send_order() functions are called continually by Role#1.

The function qd_walk() takes lateral, longitudinal and gamma differences as inputs and returns a gait decision. send_order() function takes script handle and required joint positions for three joints and using the script simulation parameters, sends these values to other modules to adjust their joint positions. When an order is sent by Role#1 to any module in the configuration, the module adjusts its joint positions and writes "1" to its script simulation parameter "order_done". When all modules have completed the order, qd_walk() function calculates new joint positions until the sub-goal is reached. Figures 4.13 to 4.15 show this movement in the test scene.



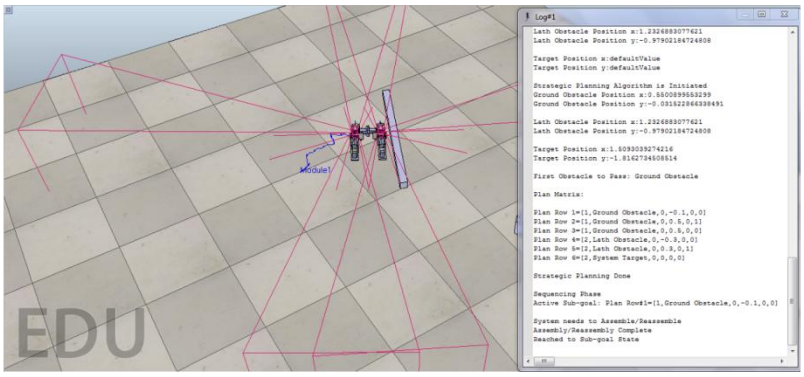
```
Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
```

Figure 4.13 : Quadruped configuration movement part 1.



```
Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
```

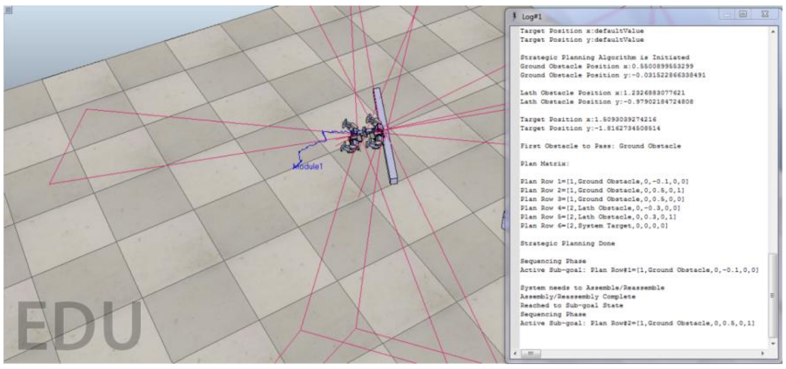
Figure 4.14 : Quadruped configuration movement part 2.



```
Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
```

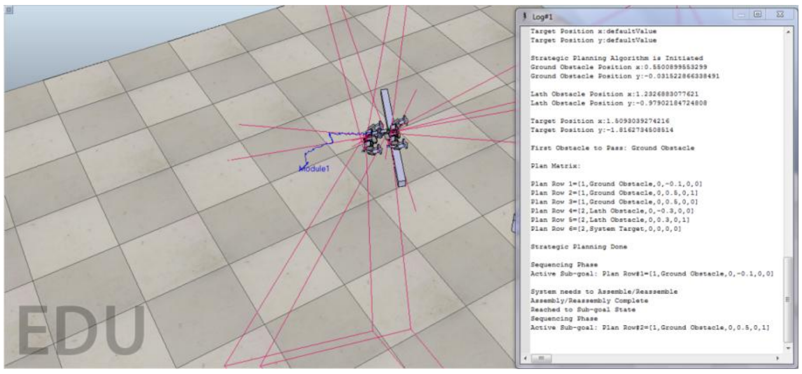
Figure 4.15 : Quadruped configuration movement part 3.

When the quadruped walker reaches to the ground obstacle, the sequencer gets the next sub-goal array. The new sub-goal state becomes [1, Ground Obstacle, 0, 0.5, 0, 1]. Notice that the transportation_mode value is 1. Therefore the sequencer issues the pass over command and the configuration passes over the obstacle. To pass over the obstacles pass_over() function is called continually. This function sends periodic joint positions to the modules until the obstacle is passed. Figures 4.16 to 4.20 show the passing over the ground obstacle process.



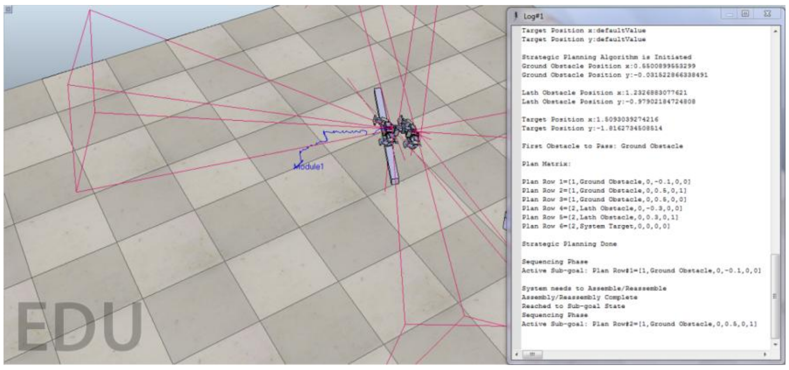
```
Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]
```

Figure 4.16 : Quadruped walker passing over ground obstacle part 1.



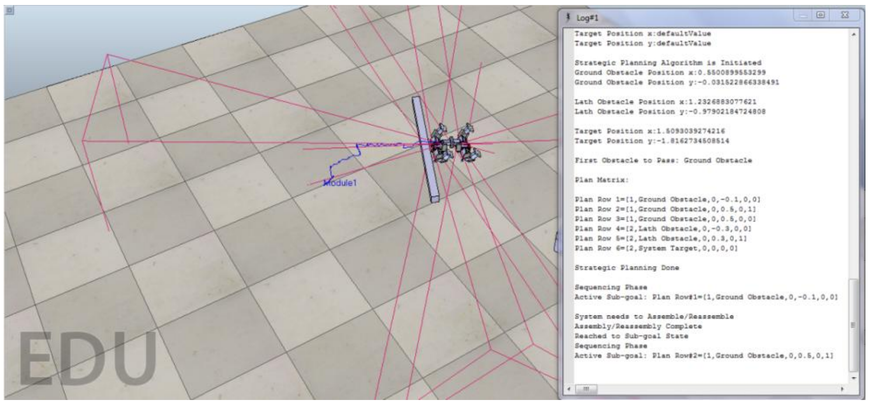
```
Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]
```

Figure 4.17 : Quadruped walker passing over ground obstacle part 2.



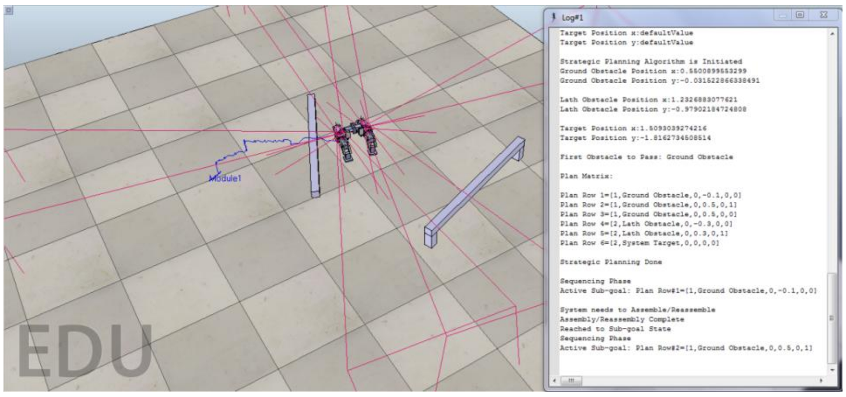
```
Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]
```

Figure 4.18 : Quadruped walker passing over ground obstacle part 3.



```
Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]
```

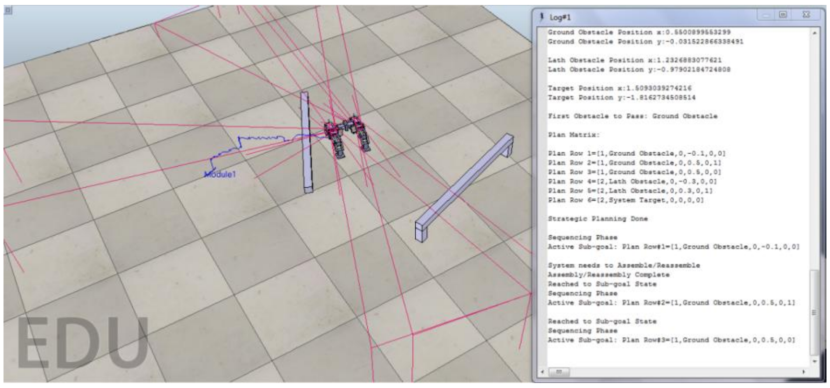
Figure 4.19 : Quadruped walker passing over ground obstacle part 4.



```
Target Position x:defaultValue
Target Position y:defaultValue

Strategic Planning Algorithm is Initiated
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]
```

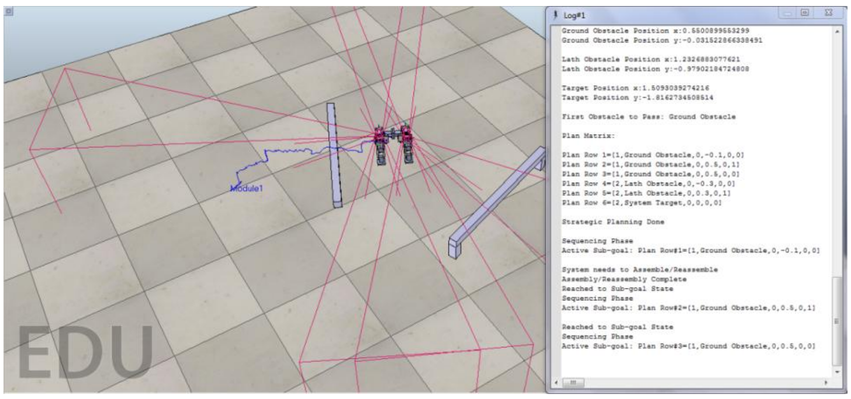
Figure 4.20 : Quadruped walker passing over ground obstacle part 5.

After the pass over command is initiated, the sequencer gets active again and fetches the next goal-state. The new sub-goal array becomes [1, Ground Obstacle, 0, 0.5, 0, 0]. Since the configurations match and the transportation mode does not need a special ability, the normal position control algorithm is executed again. Figure 4.21 and 4.22 show the repositioning of the quadruped configuration.



```
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]
```

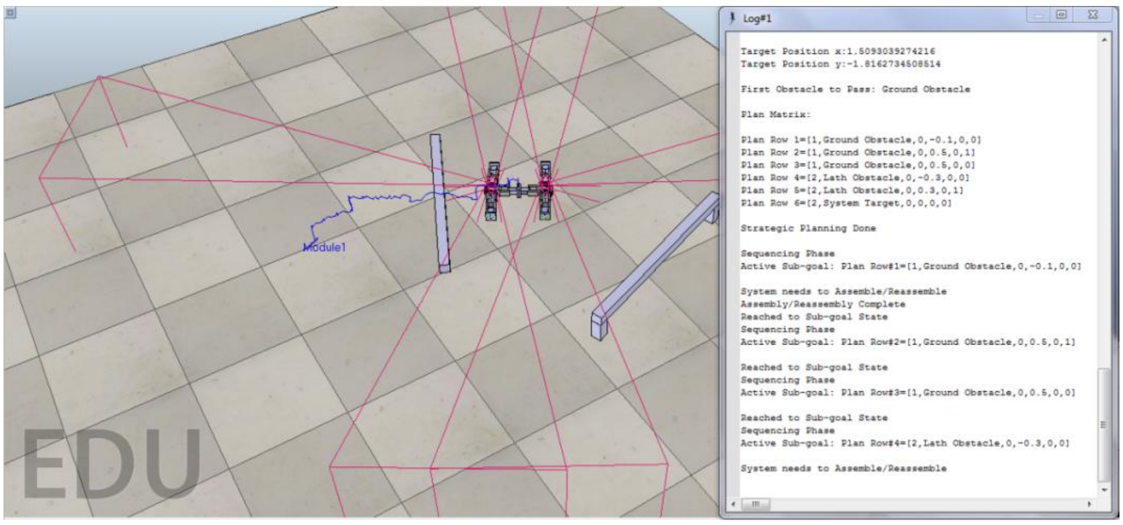
Figure 4.21 : Quadruped configuration repositioning part 1.



```
Ground Obstacle Position x:0.5500899553299
Ground Obstacle Position y:-0.031522866338491

Lath Obstacle Position x:1.2326883077621
Lath Obstacle Position y:-0.97902184724808

Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]
```

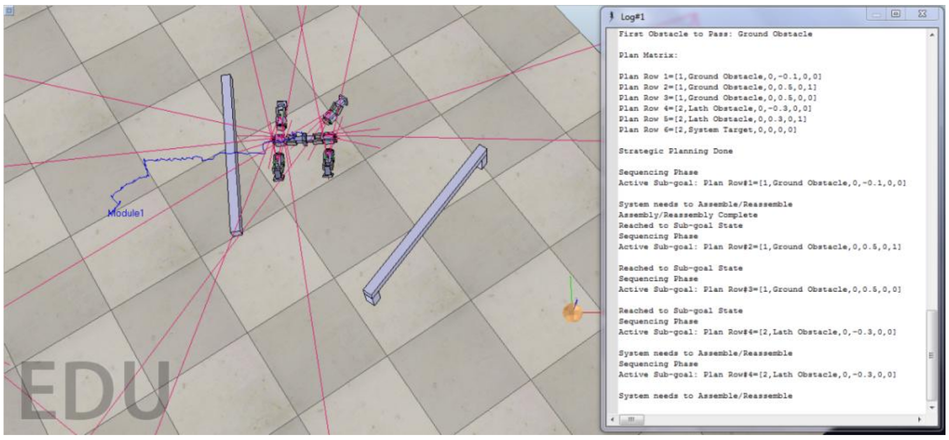
Figure 4.22 : Quadruped configuration repositioning part 2.

When the quadruped walker reaches to the sub-goal state, the sequencer fetches the next sub-goal array. The new sub-goal state is [2, Lath Obstacle, 0, -0.3, 0, 0]. Since the configurations does not match, the sequencer issues an assemble/reassemble command. Assembling and reassembling are very similar in the robotic structure. They are triggered by the same command, but if the system is in a configuration, Role#1 calls reassemble() function and the modules execute a protocol before implementing the assembly phase. If the system is in a quadruped configuration before the assemble/reassemble command is given, Role#1 and Role#2 does not break their connections and the rest of the modules break their connections and take three steps backwards with high velocity. If the system is in the wheeled configuration, Role#1 and Role#2 does not break connections and the rest of the modules break their connections before taking 3 steps forwards with high velocity. After this protocol is completed, the modules act as if it is an assemble process. Figures 4.23 to 4.26 show the reassembly process in the test scene visually.



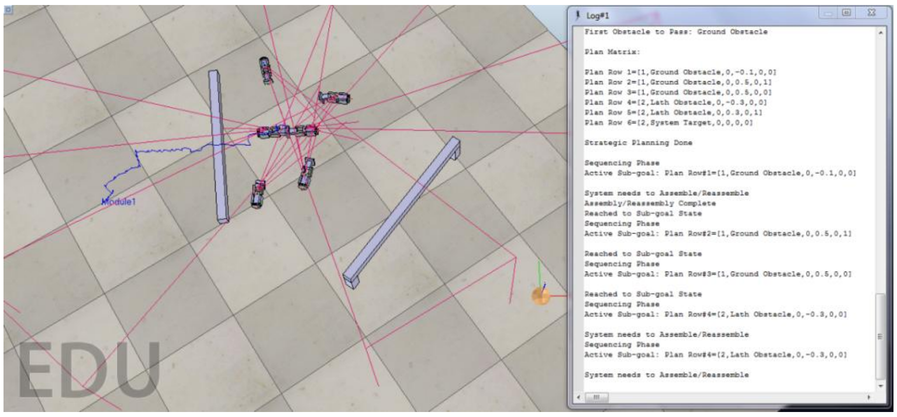
```
Target Position x:1.5093039274216
Target Position y:-1.8162734508514

First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
```

Figure 4.23 : Quadruped configuration disassembling.



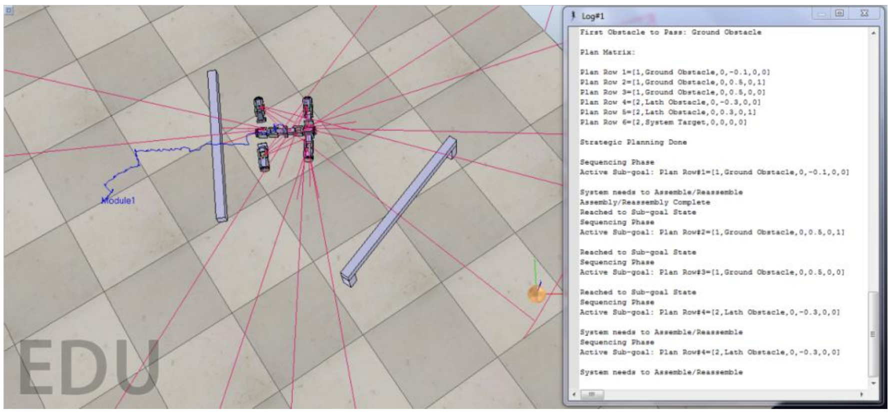
```
First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
```

Figure 4.24 : System reassembling wheeled configuration part 1.



```
First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
```

Figure 4.25 : System reassembling wheeled configuration part 2.

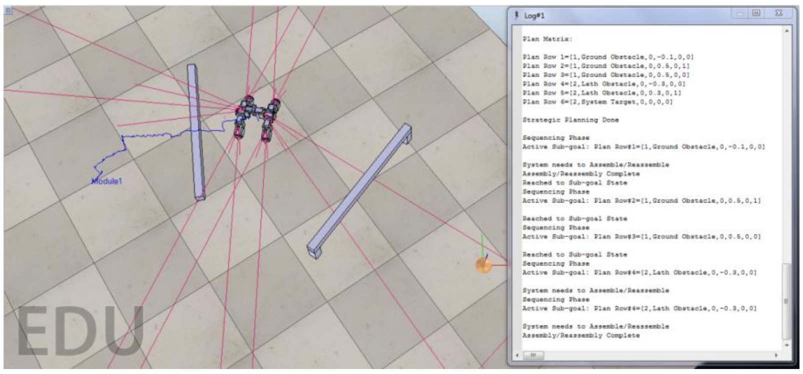


```
First Obstacle to Pass: Ground Obstacle

Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
```

Figure 4.26 : System reassembling wheeled configuration part 3.

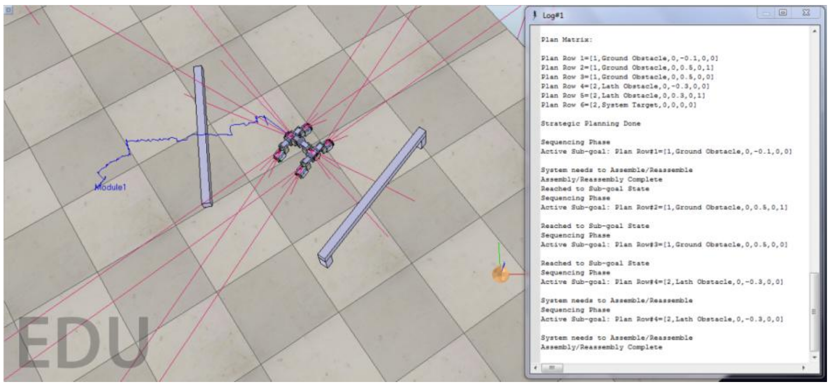
After the assembly of the wheeled configuration is completed, the configuration comparison matches and since there is not a special ability request, the sequencer executes the position control algorithm for the wheeled configuration. Similar to quadruped configuration, position control of the wheeled algorithm uses functions search_target() and lock_target(). After target is found and the visual sensor locks to it, calculate_difference() function calculates lateral and longitudinal distance between the target and the configuration. Depending on the result of the calculate_difference() function Role#1 calls correct_ornt_whld() to adjust the orientation of the configuration to the required gamma orientation or calls move_whld() to move towards the target. Role#1 uses send_order() function to other modules to control their joint positions. The modules forming the wheels interpret the send_order() command differently in wheeled configuration. They use the front joint position value as speed value and set the rotational speeds of their front joints to the sent value. In wheeled configuration, there is not a special function to control the pass under lath obstacle process. Instead of this, correct_ornt_whld() and move_whld() function take transport_mode value as input and adjust their axle positions depending on the value of this parameter.

The wheeled configuration reaches the lath obstacle and Role#1 calls sequence() function again to get the next sub-goal state. Figures 4.27 and 4.28 show the movement of wheeled configuration until it reaches to the lath obstacle.



```
Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
```

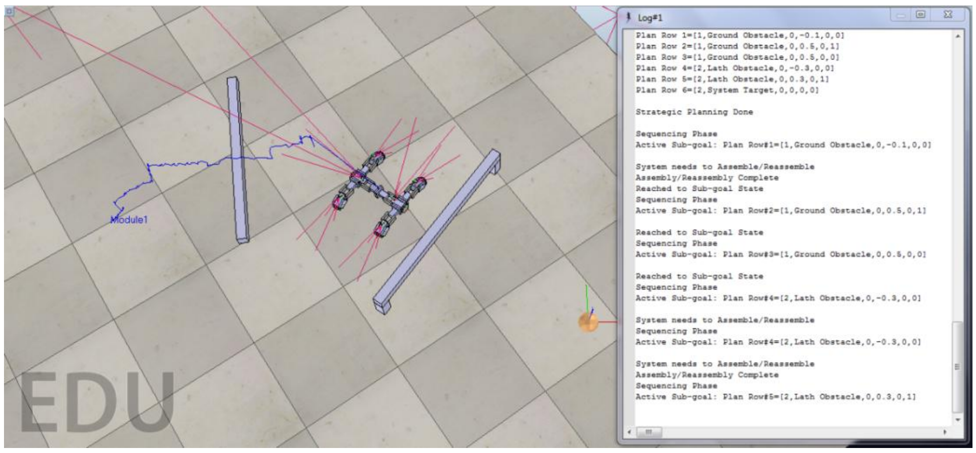
Figure 4.27 : Wheeled configuration moving to lath obstacle part 1.



```
Plan Matrix:

Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
```

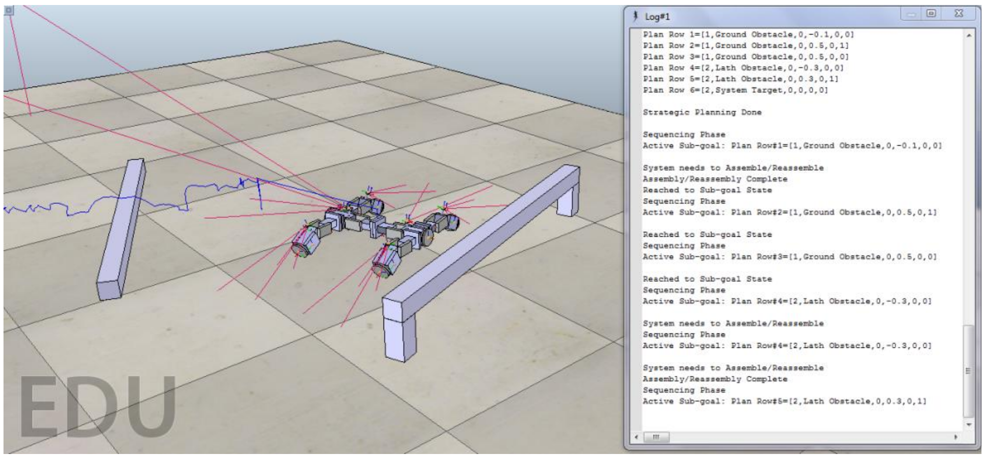
Figure 4.28 : Wheeled configuration moving to lath obstacle part 2.

The sequencer gets the new sub-goal state array as [2, Lath Obstacle, 0, 0.3, 0, 1]. The wheeled configuration changes its pose because the transportation_mode value is 1. Figures 4.29 and 4.30 show the new pose of the configuration in the test scene.



```
Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Sequencing Phase
Active Sub-goal: Plan Row#5=[2,Lath Obstacle,0,0.3,0,1]
```

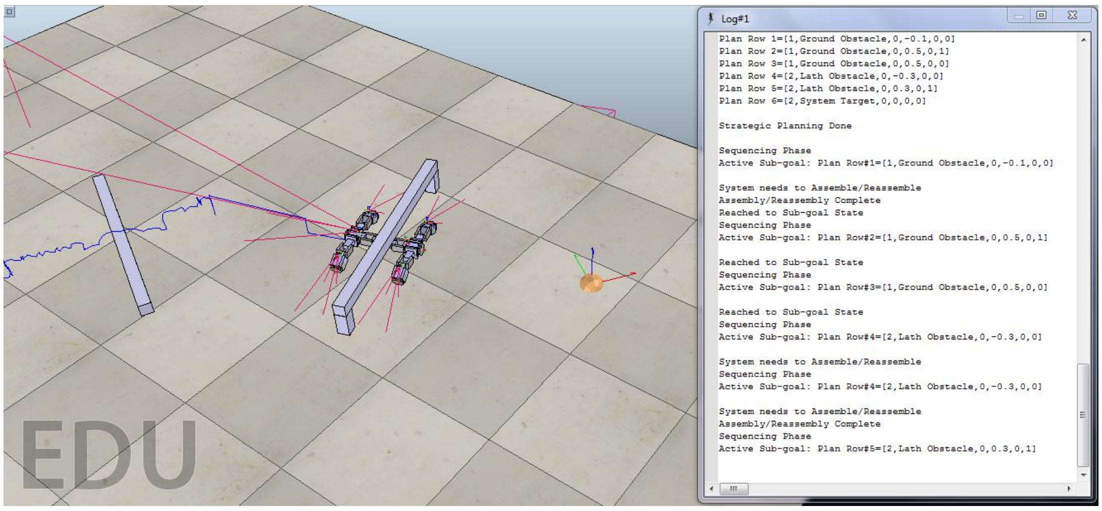
Figure 4.29 : Wheeled configuration adjusting pose to pass under lath obstacle.



```
Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Sequencing Phase
Active Sub-goal: Plan Row#5=[2,Lath Obstacle,0,0.3,0,1]
```

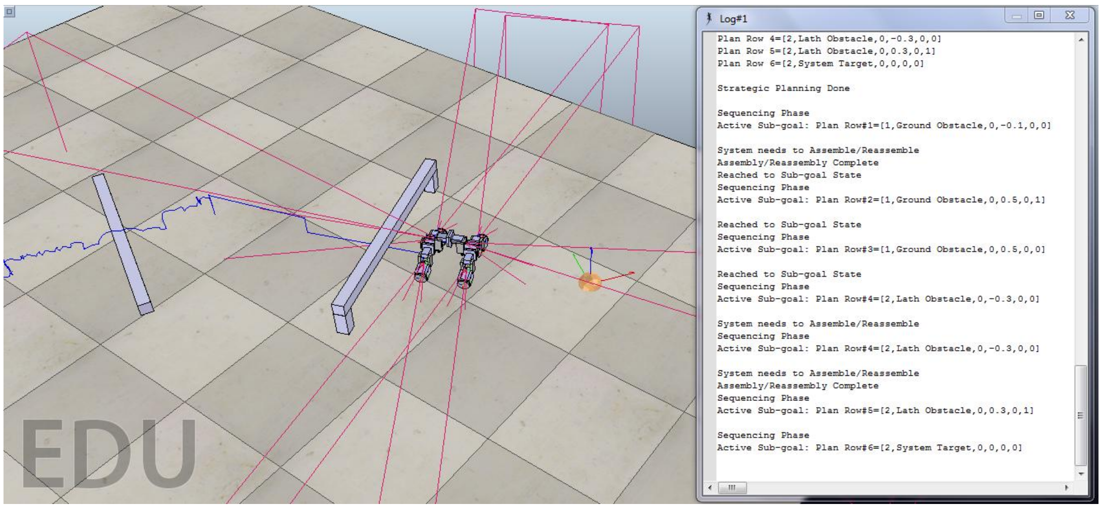
Figure 4.30 : Wheeled configuration adjusting pose to pass under lath obstacle.

After the wheeled configuration adjusts its height to pass under the lath obstacle, position control algorithm of the wheeled configuration is executed normally. When the configuration passes under the obstacle and reaches to the goal-state, the sequencer gets the next sub-goal state which is [2, System Target, 0, 0, 0, 0]. This sub-goal state is also the target state of the system. Since the configurations match and transport_mode is 0, the wheeled configuration returns back to its normal pose and moves to the target position. Figures 4.31 to 4.34 show this movement.



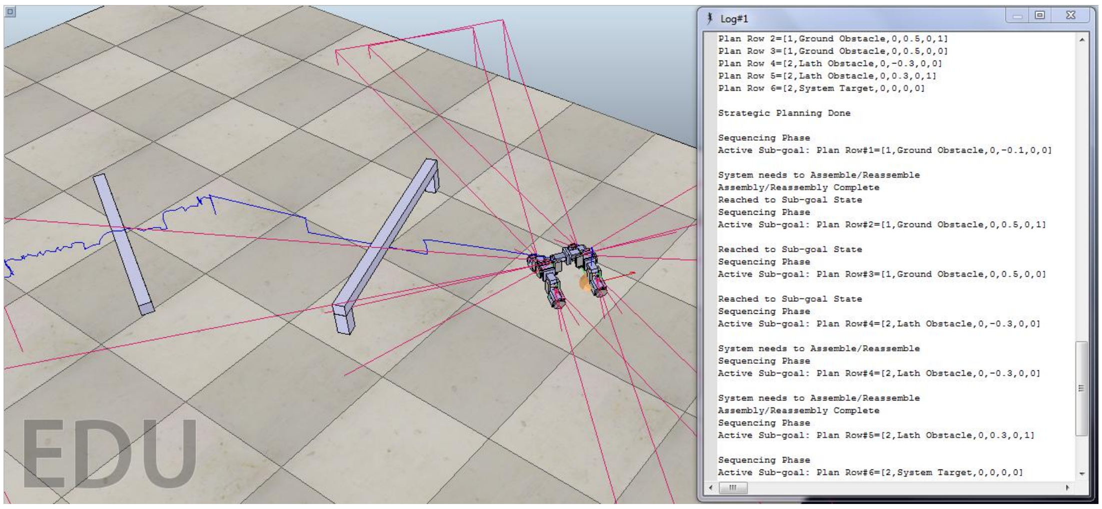
```
Plan Row 1=[1,Ground Obstacle,0,-0.1,0,0]
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Sequencing Phase
Active Sub-goal: Plan Row#5=[2,Lath Obstacle,0,0.3,0,1]
```

Figure 4.31 : Wheeled configuration passing under lath obstacle.



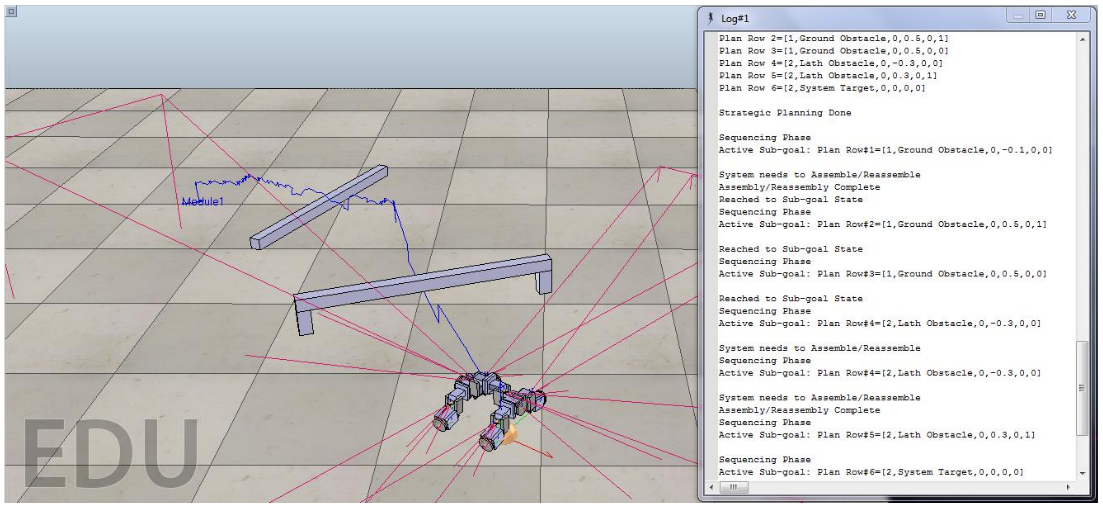
```
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Sequencing Phase
Active Sub-goal: Plan Row#5=[2,Lath Obstacle,0,0.3,0,1]

Sequencing Phase
Active Sub-goal: Plan Row#6=[2,System Target,0,0,0,0]
```

Figure 4.32 : Wheeled configuration returning to original pose.



```
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Sequencing Phase
Active Sub-goal: Plan Row#5=[2,Lath Obstacle,0,0.3,0,1]

Sequencing Phase
Active Sub-goal: Plan Row#6=[2,System Target,0,0,0,0]
```

Figure 4.33 : Wheeled obstacle moving to target.



```
Plan Row 2=[1,Ground Obstacle,0,0.5,0,1]
Plan Row 3=[1,Ground Obstacle,0,0.5,0,0]
Plan Row 4=[2,Lath Obstacle,0,-0.3,0,0]
Plan Row 5=[2,Lath Obstacle,0,0.3,0,1]
Plan Row 6=[2,System Target,0,0,0,0]

Strategic Planning Done

Sequencing Phase
Active Sub-goal: Plan Row#1=[1,Ground Obstacle,0,-0.1,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#2=[1,Ground Obstacle,0,0.5,0,1]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#3=[1,Ground Obstacle,0,0.5,0,0]

Reached to Sub-goal State
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Sequencing Phase
Active Sub-goal: Plan Row#4=[2,Lath Obstacle,0,-0.3,0,0]

System needs to Assemble/Reassemble
Assembly/Reassembly Complete
Sequencing Phase
Active Sub-goal: Plan Row#5=[2,Lath Obstacle,0,0.3,0,1]

Sequencing Phase
Active Sub-goal: Plan Row#6=[2,System Target,0,0,0,0]
```

Figure 4.34 : Wheeled configuration in target state.

The simulation ends when the robotic system reaches to the target state of the system defined by row six of the plan matrix. The blue trace shown on Figure 4.34 is created by using V-Rep's graph tools. It is a 3D curve showing the position of the reference position of Module#1 throughout the simulation.

5. CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusion

The purpose of this study is to prove that with a good module design and strategic planning algorithm, modular robotic structures can show great functionality and versatility over their monolithic counterparts while being affordable due to their suitable nature for mass production. For that purpose, a chain type modular robotic structure is designed and a hybrid deliberative/reactive type of strategic planning algorithm is developed and tested in a simulation environment called V-Rep.

To overcome the general self-reconfiguration problem of chain type systems, the modules of the system are designed to achieve self mobility. The self-mobility of a single module is achieved by adding a wheel for orientation control and a foot to practice an inchworm like locomotion method for propulsion in longitudinal direction. The module has three revolute joints; one for control of the wheel and two for manipulating the foot part of the module. Besides these joints, the module has a visual sensor attached to a pole like structure with two degrees of freedom for controlling the visual orientation and a force sensor embedded between two cylindrical discs of the wheel for friction estimation. The single module is designed to have four connection points for making connections with other modules to create more functional configurations.

After single module structure is determined, the locomotion gait for positioning of the modules is designed. The gait is designed to provide long and short steps in forward and backward directions by kinematics analysis of the foot chain. The gait designed is used in the position control algorithm of the single module which is mainly used for assembly of configurations. The position control algorithm is used to decide the step size and direction to be applied to reach the target position of the single module in optimum time.

Assembly of configurations is simply multi module position control. A role distribution algorithm based on initial states of modules is developed to position the modules before connecting to build a configuration. The positions for each role are predetermined and each module positions itself individually and connects to its target in this phase.

After the assembly process is determined, two configurations consisting of six modules to implement quadruped and wheeled locomotion are designed. The locomotion methods are designed using kinematic models of each configuration. For quadruped walker configuration a trotting gait to move in longitudinal direction and a sidling gait to move in lateral direction are developed. Since the wheeled locomotion is more straightforward there is no need to develop a locomotion method other then controlling the wheels in pairs to steer or rotate in place. The configurations are also designed to have configuration specific abilities. The quadruped walker configuration has the ability to pass over obstacles in the ground and the wheeled configuration has the ability to change its height to pass under obstacles. Similar to the single module position control algorithm, using motion kinematic models, position control algorithms for each configuration are developed.

The strategic planning algorithm developed in this study can be classified as hybrid deliberative/reactive control architecture. The algorithm consists of two layers; (1) deliberative layer to generate a feasible plan consists of sub goals to drive the robotic structure from its initial state to the desired goal state and (2) reactive layer to execute the plan similar to a feedback control mechanism.

After both the robotic structure design and strategic planning algorithm development are completed, the whole structure is tested in the simulation environment. In the test area there are obstacles between the desired goal state and the initial state of the robotic structure. The obstacles are passable by the implementation of configuration specific abilities. This test area is designed to test the overall performance of the whole robotic structure with its control algorithm as a whole.

The test showed that six simple robots having no specific ability can pass over and under obstacles to reach a desired goal position by cooperating and building more functional configurations. This proves that modular robotic structures can be more functional and more versatile over their monolithic counterparts.

5.2 Recommendations

In this study, a modular robotic structure which can change its shape to implement different locomotion methods is designed and created in simulation environment. To control this modular robotic structure, a hybrid deliberative/reactive strategic planning algorithm is also developed. In this section, some recommendations for future alterations on the robotic structure and strategic planning algorithm are shared.

The designing, creating and testing processes of this study is done in simulation environment due to time and resource constraints. To realize the study, some additions should be done to the presented structure. Firstly, the visual sensors of the modules are simulated cameras using proxy sensors in the simulation software and they do not have real life counterparts. In a real life implementation of this study, cameras or advanced infrared or ultrasonic distance sensors can be used. Camera usage will bring extra coding work for image processing and that may mean extra electronics load. Distance sensors may not answer the needs of the structure. Therefore the feasibility of any sensor solution should be analyzed thoroughly.

Secondly, this study does not present an applicable connection mechanism for modules. The connection mechanism presented in this study is just a representation of a connection with the use of dummies specific to the simulation software V-Rep. To realize the robotic structure, a proper connection mechanism should be designed and implemented.

Lastly, the communication method applied in the robotic system is not applicable in real life. The communication method used in the system uses shared variables called script simulation parameters. Therefore, if the system is to be realized, another communication method should be set.

Besides realization of the system, there can be other development options for the system. This study is mainly concerned on creating and implementing a strategic planning algorithm to a modular robotic structure. The position control algorithms of modules or configurations are generally kept at a basic level. These algorithms can be developed further and more optimal results can be achieved. Similarly there is a lot of room for development in the designed strategic planning algorithm. The hybrid architecture is open for further development and numerous functionalities can be added in the strategic planning part such as changing the generated plan by the reactive layer sensory data, adding an obstacle avoidance behavior and adding a protocol for broken modules.

REFERENCES

[1] **T. Fukuda and S. Nakagawa (**1988). Approach to the dynamically reconfigurable robotic system, *Intelligent and Robotic Systems,* 1(1):55-72.

[2] **M. Yim** (1994). Locomotion with a unit-modular reconfigurable robot, PhD thesis, Department of Mechanical Engineering, Stanford University, Stanford, CA.

[3] **G. S. Chirikjian** (1994). Kinematics of a metamorphic robotic system, *Proc., IEEE Int. Conf. on Robotics and Automation*, volume 1, pages 449-455, San Diego, CA.

[4] **S. Murata, H. Kurokawa and S. Kokaji** (1994). Self assembling machine, *Proc., IEEE Int. Conf. on Robotics and Automation*, pages 441-448, San Diego, CA.

[5] **A. Castano, W.-M. Shen and P. Will** (2000). Towards deployable robots with inter-robot metamorphic capabilities, *Autonomous Robots*, 8(3):309-324.

[6] **M. Yim, D. G. Duff and K. D. Roufas** (2000). PolyBot: A modular reconfigurable robot, *Proc., IEEE Int. Conf. on Robotics and Automation*, volume 1, pages 441-448, San Diego, CA.

[7] **S. Murata, H. Kurokawa, E. Yoshida, K. Tomita and S. Kokaji** (1998). A 3-D self-reconfigurable structure, *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 432-439, Leuven, Belgium.

[8] **D. Rus and K. Kotay** (1997). Versatility for unknown worlds: Mobile sensors and self-reconfiguration, *Proc., Field and Service Robotics*, Berlin, Germany.

[9] **S. Murata, K. Tomita, E. Yoshida, H. Kurokawa and S. Kokaji** (2000). Self reconfigurable robot-module design and simulation, *Proc. 6th Int. Conf. on Intelligent Autonomous Systems*, pages 911-917, Venice, Italy.

[10] **M. W. Jorgensen, E. H. Ostergaard and H. H. Lund** (2004). Modular ATRON: Modules for a self-reconfigurable robot, *Proc. IEEE/RSJ Int. Conf. on Robots and Systems*, pages 2068-2073, Sendai, Japan.

[11] **E. H. Ostergaard, K. Kassow, R. Beck and H.H. Lund** (2006). Design of the ATRON lattice-based self-reconfigurable robot, *Autonomous Robots*, 21(2):165-183.

[12] **W.-M. Shen, M. Krikovon, M. Rubinstein, C.h. Chiu, J. Everst and J. B. Venkatesh** (2006). Multimode locomotion via self-reconfigurable robots, *Autonomous Robots*, 20(2):165-177.

[13] **Brooks, R**. (1986). A robust layered control system for a mobile robot. Robotics and Automation, IEEE Journal of [legacy, pre-1988] 2 (1): 14–23.

[14] **Anthony Mandow, Jorge L. Martinez** (2007). Experimental kinematics for wheeled skid-steer mobile robots, *Intelligent Robots and Systems*, pages 1222-1227.

[15] **J. L. Martinez, A. Mandow, J. Morales, S.Pedraza and A. Garcia Perezo** (2005). Approximating kinematics for tracked mobile robots, *International Journal of Robotics Research*, vol. 24, no. 10, pp. 867-878.

[16] S. **Pedraza, R. Fernandez, V. Munoz and A. Garcia Cerezo** (2000). A motion control approach for a tracked mobile robot, *Proc.of the 4th IFAC International Symposium on Intelligent Components and Instruments for Control Applications*, pp. 147-152, Buenos Aires, Argentina.

[17] **Url - 1** <*https://en.wikipedia.org/wiki/Shakey_the_robot*>, date retrieved 18.08.2015.

[18] **Url - 2** <*http://www.gazebosim.org/* >, date retrieved 25.04.2015.

[19] **Url - 3** <*http://playerstage.sourceforge.net/*>, date retrieved 14.08.2015.

[20] **Url - 4** <*http://www.cyberbotics.com/*>, date retrieved 29.04.2015.

[21] **Url - 5** <*http://www.coppeliarobotics.com/*>, date retrieved 29.04.2015.

[22] **Url - 6** <*http://www.coppeliarobotics.com/helpFiles/index.html*> date retrieved 29.04.2015

[23] **Url - 7** <*http://en.wikipedia.org/wiki/Instant_centre_of_rotation*> date retrieved 08.12.2014.

APPENDICES

APPENDIX-A: Search Program of the First Kinematic Model

```
clear
clc

count = 0;

d1 = 0.0775;
d2 = 0.0475;
d3 = 0.0470;
a3 = 0.025;
th1 = 0;

for angle1 = -90:0
  for angle2 = -120:120
      al1 = angle1*pi/180;
      al2 = angle2*pi/180;
      th3 = 0;

      A1 = [cos(th1) -sin(th1)*cos(al1) sin(th1)*sin(al1) 0; sin(th1) cos(th1)*cos(al1) -
cos(th1)*sin(al1) 0; 0 sin(al1) cos(al1) d1; 0 0 0 1];
      A2 = [1 0 0 0; 0 cos(al2) -sin(al2) 0; 0 sin(al2) cos(al2) d2; 0 0 0 1];
      A3 = [0 1 0 0; -1 0 0 -a3; 0 0 1 d3; 0 0 0 1];

      A = A1*A2*A3;

      vect = [A(1,4) A(2,4) A(3,4)];
      vect_vrep = [-vect(3) -vect(1) vect(2)+0.025];

      if(vect_vrep(3)>0.005)
         if(vect_vrep(3)<0.01)
            if(vect_vrep(1)>-0.1)
               count = count + 1;
               res(count,1) = count;
               res(count,2) = angle1;
               res(count,3) = angle2;
               res(count,4) = vect_vrep(1);
               res(count,5) = vect_vrep(3);
               res(count,6) = vect_vrep(1)+0.1720;
               px(count) = count;
               py1(count) = vect_vrep(1);
               py1_dif(count) = vect_vrep(1)+0.1720;
               py2(count) = vect_vrep(3);
            end
         end
      end

   end
end

res
```

```
plot(px,py1)
```

APPENDIX-B: Search Program of the Second Kinematic Model

```
clear
clc

th1 = 0;
th3 = 15*pi/180; %backjoint2
th4 = 60*pi/180; %backjoint1

a1 = 0.025;
a2 = 0.0470;
a3 = 0.0475;
a4 = 0.0775;
d5wt = 0.025;
a5bc = -0.075;
d5bc = -0.005;

count = 0;

for i1 = -90:1:90;
   th3 = i1*pi/180;
   for i2 = -90:1:90;
       th1 = 0;
       th4 = i2*pi/180;

       A1 = [cos(th1) -sin(th1) 0 a1*cos(th1); sin(th1) cos(th1) 0 a1*sin(th1); 0 0 1 0; 0 0 0 1];
       A2 = [0 -1 0 0; 1 0 0 a2; 0 0 1 0; 0 0 0 1];
       A3 = [cos(th3) -sin(th3) 0 a3*cos(th3); sin(th3) cos(th3) 0 a3*sin(th3); 0 0 1 0; 0 0 0 1];
       A4 = [cos(th4) 0 -sin(th4) a4*cos(th4); sin(th4) 0 cos(th4) a4*sin(th4); 0 -1 0 0; 0 0 0 1];
       A5wt = [1 0 0 0; 0 1 0 0; 0 0 1 d5wt; 0 0 0 1];
       A5bc = [1 0 0 a5bc; 0 1 0 0; 0 0 1 d5bc; 0 0 0 1];

       Awt = A1*A2*A3*A4*A5wt;

       th1 = atan2(Awt(1,4), Awt(2,4));
       A1 = [cos(th1) -sin(th1) 0 a1*cos(th1); sin(th1) cos(th1) 0 a1*sin(th1); 0 0 1 0; 0 0 0 1];

       C1 = A1*A2;
       C2 = C1*A3;
       Awt = C2*A4*A5wt;
       Abc = Awt*A5bc;

       cond1 = C1(1,4);
       cond2 = C2(1,4);
       pos_km(1) = Awt(1,4);
       pos_km(2) = Awt(2,4);
       pos_km(3) = Awt(3,4);

       pos_lf(1) = pos_km(2);
       pos_lf(2) = pos_km(3);
       pos_lf(3) = pos_km(1);

       f1h = abs(90-i1);
       f2h = abs(-36-i2);

       f1l = abs(52-i1);
```

```
        f2l = abs(-27-i2);

%       if( (pos_lf(1)<0.127)&& (pos_lf(1)>0.122) && (pos_lf(1)<0.172)  && (Abc(1,4) <= 0.005)
&& (Abc(1,4) > 0.0025) )% && (th1*180/pi > -30) )     %caster pivot high-vel search
%       if( (pos_lf(1)<0.1525)&& (pos_lf(1)>0.1515) && (pos_lf(1)<0.172)  && (Abc(1,4) <= 0.005)
&& (Abc(1,4) > 0.000) )% && (th1*180/pi > -30) )     %caster pivot low-vel search
%       if( (pos_lf(1)<0.172)&& (pos_lf(1)>0.127) && (pos_lf(1)<0.172) && (Abc(1,4) >= 0.015)
&& (f1h+f2h<45) )     %first pose high-vel search
        if( (pos_lf(1)<0.154)&& (pos_lf(1)>0.152) && (pos_lf(1)<0.172) && (Abc(1,4) >= 0.015) &&
(f1l+f2l<50) )    %first pose low-vel search
%       if( (pos_lf(1)<0.10)&& (pos_lf(1)>0.08) && (pos_lf(1)<0.172) && (th1*180/pi > -30) &&
(th1*180/pi < 30) && (Abc(1,4) >= 0.02) )     %max displacement
            count = count + 1;
            pos_lf_array(count,1) = count;
            pos_lf_array(count,2) = th1*180/pi;
            pos_lf_array(count,3) = i1;                              %backjoint2
            pos_lf_array(count,4) = i2;                              %backjoint1
            pos_lf_array(count,5) = 0.172 - pos_lf(1);
            pos_lf_array(count,6) = Abc(1,4);
            pos_lf_array(count,7) = f1h+f2h;
            pos_lf_array(count,8) = f1l+f2l;

        end
    end
end
pos_lf_array

for v = 1:1:count;
    px(v) = pos_lf_array(v,1);
    py1(v) = pos_lf_array(v,5);

end
plot(px,py1)
```

APPENDIX-C: Lua Code of the Module Scripts in V-Rep

```
if (sim_call_type==sim_childscriptcall_initialization) then

-- Put some initialization code here
--//Script Specific Handles and Variables------------------------------------------------------------
    console = simAuxiliaryConsoleOpen("Log#1",1000,10100)
    log_flag_dummy = 0
    Script = simGetScriptHandle("Cylinder1x1")
    Cylinder = simGetObjectHandle("Cylinder1x1")

    det_point = simGetObjectHandle("DetPointx1")

    conn_points = {simGetObjectHandle("DummyFx1"),simGetObjectHandle("DummyBFx1"),
                simGetObjectHandle("DummyRx1"),simGetObjectHandle("DummyLx1"),
                simGetObjectHandle("DummyBSx1")}

    Fix_FS = simGetObjectHandle("Fix_FSx1")

    cam_dummy = simGetObjectHandle("CamDummyx1")
    pos_dummy = simGetObjectHandle("Positionx1")

    FrontJoint = simGetObjectHandle("FrontJointx1")
    BackJoint1 = simGetObjectHandle("BackJoint1x1")
    BackJoint2 = simGetObjectHandle("BackJoint2x1")
    FS = simGetObjectHandle("FS_Cylinderx1")

    Cam = simGetObjectHandle("Camx1")
    CamJoint1 = simGetObjectHandle("CamJoint1x1")
    CamJoint2 = simGetObjectHandle("CamJoint2x1")
--Script Specific Handles and Variables//---------------------------------------------------------------
    Scripts = {simGetScriptHandle("Cylinder1x1"),simGetScriptHandle("Cylinder1x2"),
              simGetScriptHandle("Cylinder1x3"),simGetScriptHandle("Cylinder1x4"),
              simGetScriptHandle("Cylinder1x5"),simGetScriptHandle("Cylinder1x6"),
              simGetScriptHandle("target_dummy_1"),simGetScriptHandle("target_dummy_2"),
              simGetScriptHandle("target_dummy_3")}

    det_points = {simGetObjectHandle("DetPointx1"),simGetObjectHandle("DetPointx2"),
                 simGetObjectHandle("DetPointx3"),simGetObjectHandle("DetPointx4"),
                 simGetObjectHandle("DetPointx5"),simGetObjectHandle("DetPointx6"),
                 simGetObjectHandle("target_dummy_1"),simGetObjectHandle("target_dummy_2"),
                 simGetObjectHandle("target_dummy_3")}

    Roles = {0,0,0,0,0,0}

    conf = 0
    conf_req = -1
--conf_req == -1 means there is no request for configuration

    plan_row = {0,0,0,0,0,0}

    plan_row1 = {0,0,0,0,0,0}
    plan_row2 = {0,0,0,0,0,0}
    plan_row3 = {0,0,0,0,0,0}
    plan_row4 = {0,0,0,0,0,0}
    plan_row5 = {0,0,0,0,0,0}
    plan_row6 = {0,0,0,0,0,0}
    plan_row7 = {0,0,0,0,0,0}
    plan_row8 = {0,0,0,0,0,0}
    plan_row9 = {0,0,0,0,0,0}
    plan_row10 = {0,0,0,0,0,0}
--plan_rowx format: {conf_req,reference_handle,lat_const,long_const,gamma_const,transport_mode}
--transport_mode=0 normal, transport_mode=1 ability

--These variables will be a part of strategic planning!!!!!//-----

    simSetScriptSimulationParameter(Script,"Notification1",0)
    simSetScriptSimulationParameter(Script,"Notification2",0)
    simSetScriptSimulationParameter(Script,"Notification3",0)
    simSetScriptSimulationParameter(Script,"scanned",0)

    ms_result = 0
    forw_step = 0
    back_step = 0
    forw_step_f = 0
    back_step_f = 0

    direction = 0
    forced_direction = 0
    velocity = 0

    J1pos = 0
    J2pos = 0
    Cylpos = 0
    a = 0
    CamJ1pos = 0
    CamJ2pos = 0
    pj1 = 0
    pj2 = 0
    search_counter = 0

--//Variables for Friction Test------------------------------------------------
    friction_estimation_done = 0
    friction_step_counter = 0
    fs_read_counter = 0
    fs_force_reading_1 = {0,0,0,0,0,0,0,0,0,0}
    fs_force_reading_2 = {0,0,0,0,0,0,0,0,0,0}
    fs_force_reading_3 = {0,0,0,0,0,0,0,0,0,0}
    fs_force_reading_4 = {0,0,0,0,0,0,0,0,0,0}
    fs_force_reading_max = {0,0,0,0}
    friction_coeff = 0

--siralama - 0.2,0.4,0.6,0.8,1.0
    dfhv_list = {0.0502,0.0663,0.0662,0.0662,0.0635}
    tfhv_list = {0.95,0.95,0.95,0.95,0.95}
    dflv_list = {0.0106,0.0124,0.0153,0.0165,0.0162}
    tflv_list = {0.65,0.65,0.65,0.65,0.65}

    dbhv_list = {0.0117,0.0161,0.0186,0.0210,0.0233}
    tbhv_list = {1.05,1.05,1.05,1.05,1.05}
    dblv_list = {0.0075,0.0091,0.0104,0.0113,0.0123}
    tblv_list = {0.80,0.80,0.80,0.80,0.80}

    t180_list = {4.40,4.55,4.35,4.25,4.20}

    w_qtr_ag_list = {19.95,20.86,19.08,16.61,7.14}
    d_qtr_ag_list = {0.101,0.106,0.106,0.095,0.058}
    t_qtr_ag_list = {1.25,1.25,1.20,1.20,1.20}

    w_qtr_md1_list = {14.72,15.04,13.35,8.59,3.76}
    d_qtr_md1_list = {0.106,0.112,0.112,0.099,0.06}
    t_qtr_md1_list = {1.25,1.25,1.20,1.20,1.20}

    w_qtr_md2_list = {7.81,8.09,6.53,3.91,1.47}
    d_qtr_md2_list = {0.107,0.114,0.116,0.101,0.058}
    t_qtr_md2_list = {1.25,1.25,1.20,1.20,1.20}

    w_qtr_ps_list = {4.42,4.71,3.56,2.11,0.36}
    d_qtr_ps_list = {0.094,0.099,0.105,0.095,0.057}
    t_qtr_ps_list = {1.25,1.25,1.20,1.20,1.20}

    w_qsd_ag_list = {3.88,4.96,6.94,7.87,7.58}
    d_qsd_ag_list = {0.107,0.111,0.116,0.109,0.081}
    t_qsd_ag_list = {1.90,1.90,1.85,1.80,1.80}

    w_qsd_md1_list = {2.49,3.54,4.84,5.79,4.22}
    d_qsd_md1_list = {0.106,0.111,0.120,0.113,0.081}
    t_qsd_md1_list = {1.80,1.80,1.80,1.80,1.70}

    w_qsd_md2_list = {1.74,1.51,2.35,1.89,1.94}
    d_qsd_md2_list = {0.107,0.111,0.121,0.115,0.080}
    t_qsd_md2_list = {1.85,1.80,1.80,1.80,1.80}

    w_qsd_ps_list = {1.02,0.85,1.22,1.08,0.82}
    d_qsd_ps_list = {0.103,0.105,0.115,0.110,0.075}
    t_qsd_ps_list = {1.80,1.80,1.80,1.80,1.80}

--Variables for Friction Test//----------------------------------------------------
--//Variables for initial scan----------------------------------------------------

    initial_scan_done = 0
    scanned_mods = 0
    RdetModbin = {0,0,0,0,0,0,0,0,0}
    ModPos_x = {0,0,0,0,0,0,0,0,0}
    ModPos_y = {0,0,0,0,0,0,0,0,0}

--Variables for initial scan//----------------------------------------------------
--//Variables for Phases---------------------------------------------
    sensing_done = 0
    sp_done = 0
    rd_done = 0

    pos_x_sum = 0
    pos_y_sum = 0
    x_dif = {0,0,0,0,0,0}
    y_dif = {0,0,0,0,0,0}
    dist = {0,0,0,0,0,0}
    gpos_x = {0,0,0,0,0,0}
    gpos_y = {0,0,0,0,0,0}

--Variables for Phases//---------------------------------------------

    assembly_counter = 1
    target_state_reached = 0
    target_pos_reached = 0
    target_ornt_reached = 0
    target_found = 0

    lat_const = 0.5
    long_const = 0.5
    gamma_const = 0

    ornt_req = 0
    ornt_des = 0
    ornt_dif = 0

--//Four Legged Parameters----------------------------------------------------
    c4wa = 0
    c4wp = 0
    c4wa_tr = 0
    c4wa_sd = 0
    order_done_counter = 0
    s = -30
    r = 0
    l = 0
    tr= 0
    sd = 0
    qd_po_counter = 0

    assembly_phase_done = 0
    reassemble_step=0
    reassemble_move_counter = 0
--Four Legged Parameters//----------------------------------------------------
--//Four Wheeled Parameters----------------------------------------------------
    c4wh = 0
```

```
--Four Legged Parameters//----------------------------------------------------

  function estimate_friction(friction_estimation_done)
    if(log_flag_dummy==0) then
      simAuxiliaryConsolePrint(console,"Friction Estimation Started\n")
      log_flag_dummy=1
    end
    if(friction_estimation_done~=1) then
      if(friction_step_counter<5) then
        ms_result,J1pos,J2pos,forw_step,back_step = move_single(1,1,forw_step,back_step,0)

        if(forw_step==0)then
          friction_estimation_done = 0
          fsresult,force,torque=simReadForceSensor(FS)
          fs_read_counter = fs_read_counter + 1
          if(friction_step_counter==1)   then
            fs_force_reading_1[fs_read_counter] = force[3]
          end
          if(friction_step_counter==2)   then
            fs_force_reading_2[fs_read_counter] = force[3]
          end
          if(friction_step_counter==3)   then
            fs_force_reading_3[fs_read_counter] = force[3]
          end
          if(friction_step_counter==4)   then
            fs_force_reading_4[fs_read_counter] = force[3]
          end
          if(ms_result==1)   then
            friction_step_counter = friction_step_counter + 1
            fs_read_counter = 0
          end
        end
      end
      if(friction_step_counter==5)   then
        fs_force_reading_max[1] =
math.max(fs_force_reading_1[1],fs_force_reading_1[2],fs_force_reading_1[3]
                        ,fs_force_reading_1[4],fs_force_reading_1[5],fs_force_reading_1[6]
                        ,fs_force_reading_1[7],fs_force_reading_1[8],fs_force_reading_1[9]
                        ,fs_force_reading_1[10])
        fs_force_reading_max[2] =
math.max(fs_force_reading_2[1],fs_force_reading_2[2],fs_force_reading_2[3]
                        ,fs_force_reading_2[4],fs_force_reading_2[5],fs_force_reading_2[6]
                        ,fs_force_reading_2[7],fs_force_reading_2[8],fs_force_reading_2[9]
                        ,fs_force_reading_2[10])
        fs_force_reading_max[3] =
math.max(fs_force_reading_3[1],fs_force_reading_3[2],fs_force_reading_3[3]
                        ,fs_force_reading_3[4],fs_force_reading_3[5],fs_force_reading_3[6]
                        ,fs_force_reading_3[7],fs_force_reading_3[8],fs_force_reading_3[9]
                        ,fs_force_reading_3[10])
        fs_force_reading_max[4] =
math.max(fs_force_reading_4[1],fs_force_reading_4[2],fs_force_reading_4[3]
                        ,fs_force_reading_4[4],fs_force_reading_4[5],fs_force_reading_4[6]
                        ,fs_force_reading_4[7],fs_force_reading_4[8],fs_force_reading_4[9]
                        ,fs_force_reading_4[10])

        fs_force_reading_avg = (fs_force_reading_max[1]+fs_force_reading_max[2]+
                      fs_force_reading_max[3]+fs_force_reading_max[4])/4

        if(fs_force_reading_avg>0.15 and fs_force_reading_avg<0.25) then
          friction_coeff = 0.2
    dfhv = dfhv_list[1]
    tfhv = tfhv_list[1]
    dflv = dflv_list[1]
    tflv = tflv_list[1]

    dbhv = dbhv_list[1]
    tbhv = tbhv_list[1]
    dblv = dblv_list[1]
    tblv = tblv_list[1]

    t180 = t180_list[1]

end
if(fs_force_reading_avg>0.25 and fs_force_reading_avg<0.45) then
    friction_coeff = 0.4

    dfhv = dfhv_list[2]
    tfhv = tfhv_list[2]
    dflv = dflv_list[2]
    tflv = tflv_list[2]

    dbhv = dbhv_list[2]
    tbhv = tbhv_list[2]
    dblv = dblv_list[2]
    tblv = tblv_list[2]

    t180 = t180_list[2]
end
if(fs_force_reading_avg>0.45 and fs_force_reading_avg<0.65) then
    friction_coeff = 0.6

    dfhv = dfhv_list[3]
    tfhv = tfhv_list[3]
    dflv = dflv_list[3]
    tflv = tflv_list[3]

    dbhv = dbhv_list[3]
    tbhv = tbhv_list[3]
    dblv = dblv_list[3]
    tblv = tblv_list[3]

    t180 = t180_list[3]
end
if(fs_force_reading_avg>0.65 and fs_force_reading_avg<0.75) then
    friction_coeff = 0.8

    dfhv = dfhv_list[4]
    tfhv = tfhv_list[4]
    dflv = dflv_list[4]
    tflv = tflv_list[4]

    dbhv = dbhv_list[4]
    tbhv = tbhv_list[4]
    dblv = dblv_list[4]
    tblv = tblv_list[4]

    t180 = t180_list[4]
end
if(fs_force_reading_avg>0.75)   then
```

```
                friction_coeff = 1

                dfhv = dfhv_list[5]
                tfhv = tfhv_list[5]
                dflv = dflv_list[5]
                tflv = tflv_list[5]

                dbhv = dbhv_list[5]
                tbhv = tbhv_list[5]
                dblv = dblv_list[5]
                tblv = tblv_list[5]

                t180 = t180_list[5]
            end
            friction_estimation_done = 1
            simAuxiliaryConsolePrint(console,"Friction Estimation Done\n")
            simAuxiliaryConsolePrint(console,"Friction Coefficient:")
            simAuxiliaryConsolePrint(console,friction_coeff)
            simAuxiliaryConsolePrint(console,"\n\n")
            log_flag_dummy=0
        end
    end
    return friction_coeff,friction_estimation_done
end

function initial_scan(CamJ1pos)
    if(log_flag_dummy==0) then
        simAuxiliaryConsolePrint(console,"Localization Scan Started\n")
        log_flag_dummy=1
    end

    Cam_act_pos = math.deg(simGetJointPosition(CamJoint1))

    for i=1,9,1 do
        result,distance,detP = simCheckProximitySensor(Cam,det_points[i])

        if(result==1) then
            fn_pj1 = simGetJointPosition(CamJoint1)
            fn_pj2 = simGetJointPosition(CamJoint2)
            ornt = simGetObjectOrientation(pos_dummy,-1)

            matrix = simGetObjectMatrix(cam_dummy,-1)
            matrix[4] = 0
            matrix[8] = 0
            matrix[12] = 0
            det = {detP[3],detP[1],detP[2]}
            target = simMultiplyVector(matrix,det)
            dum_pos = {0,0,0}
            dum_pos[1] = target[1] + 0.102*math.cos(ornt[3]) +
(0.0045+0.011*math.cos(fn_pj2))*math.cos(ornt[3]+fn_pj1)
            dum_pos[2] = target[2] + 0.102*math.sin(ornt[3]) +
(0.0045+0.011*math.cos(fn_pj2))*math.sin(ornt[3]+fn_pj1)
            dum_pos[3] = target[3] + 0.175 + 0.06*math.sin(fn_pj2)

            RdetModbin[i] = 1
            Scornt = simGetScriptSimulationParameter(Scripts[i],"ornt")
            ModPos_x[i] = dum_pos[1]-0.102*math.cos(Scornt)
            ModPos_y[i] = dum_pos[2]-0.102*math.sin(Scornt)
            result = 0
        end
        end
        if(Cam_act_pos<0)then
           Cam_act_pos = Cam_act_pos + 360
        end
        if(math.abs(CamJ1pos-Cam_act_pos)<10) then
           CamJ1pos = CamJ1pos + 30
           simSetJointTargetPosition(CamJoint1,math.rad(CamJ1pos))
        end
        if(CamJ1pos>=360) then
           if(Script==Scripts[1]) then
              simSetScriptSimulationParameter(Script,"pos_x",0)
              simSetScriptSimulationParameter(Script,"pos_y",0)
              for i=1,9,1 do
                 if(RdetModbin[i]==1 and simGetScriptSimulationParameter(Scripts[i],"scanned")==0)
then
                    simSetScriptSimulationParameter(Scripts[i],"pos_x",ModPos_x[i])
                    simSetScriptSimulationParameter(Scripts[i],"pos_y",ModPos_y[i])
                    simSetScriptSimulationParameter(Scripts[i],"scanned",1)
                 end
              end
              for i=2,6,1 do
                 if(simGetScriptSimulationParameter(Scripts[i],"scanned")==1) then
                    scanned_mods = scanned_mods + 1
                 end
              end
              if(scanned_mods==5) then
                 initial_scan_done = 1
                 CamJ1pos = 0
                 simSetScriptSimulationParameter(Script,"scanned",1)

                 simAuxiliaryConsolePrint(console,"Localization Scan Done\n\n")

                 for i=1,6,1 do
                    simAuxiliaryConsolePrint(console,"Module#")
                    simAuxiliaryConsolePrint(console,i)
                    simAuxiliaryConsolePrint(console,"pos_x:")

simAuxiliaryConsolePrint(console,simGetScriptSimulationParameter(Scripts[i],"pos_x"))
                    simAuxiliaryConsolePrint(console,"\n")
                    simAuxiliaryConsolePrint(console,"Module#")
                    simAuxiliaryConsolePrint(console,i)
                    simAuxiliaryConsolePrint(console,"pos_y:")

simAuxiliaryConsolePrint(console,simGetScriptSimulationParameter(Scripts[i],"pos_y"))
                    simAuxiliaryConsolePrint(console,"\n\n")
                 end
                 log_flag_dummy=1

              end
              scanned_mods = 0
           end
           if(Script~=Scripts[1] and simGetScriptSimulationParameter(Script,"scanned")==1) then
              for i=2,9,1 do
                 if(RdetModbin[i]==1 and simGetScriptSimulationParameter(Scripts[i],"scanned")==0)
then
                    pos_x_scr = simGetScriptSimulationParameter(Script,"pos_x")
                    pos_y_scr = simGetScriptSimulationParameter(Script,"pos_y")
                    simSetScriptSimulationParameter(Scripts[i],"pos_x",ModPos_x[i]+pos_x_scr)
                    simSetScriptSimulationParameter(Scripts[i],"pos_y",ModPos_y[i]+pos_y_scr)
                    simSetScriptSimulationParameter(Scripts[i],"scanned",1)
```

```
            end
          end
          if(simGetScriptSimulationParameter(Scripts[1],"scanned")==1) then
            initial_scan_done = 1
            CamJ1pos = 0

          end
        end
      end
      return initial_scan_done, CamJ1pos
    end

    function role_dist()
--//role distribution code here-------------------------
      simAuxiliaryConsolePrint(console,"Role Distribution Algorithm Initiated\n")
      for i=1,6,1 do
        pos_x_sum = pos_x_sum + simGetScriptSimulationParameter(Scripts[i],"pos_x")
        pos_y_sum = pos_y_sum + simGetScriptSimulationParameter(Scripts[i],"pos_y")
      end
      pos_x_avg = pos_x_sum/6
      pos_y_avg = pos_y_sum/6

      simAuxiliaryConsolePrint(console,"Central Position x:")
      simAuxiliaryConsolePrint(console,pos_x_avg)
      simAuxiliaryConsolePrint(console,"\n")
      simAuxiliaryConsolePrint(console,"Central Position y:")
      simAuxiliaryConsolePrint(console,pos_y_avg)
      simAuxiliaryConsolePrint(console,"\n\n")
      simAuxiliaryConsolePrint(console,"Selection of Role#1\n")

      for i=1,6,1 do
        x_dif[i] = pos_x_avg - simGetScriptSimulationParameter(Scripts[i],"pos_x")
        y_dif[i] = pos_y_avg - simGetScriptSimulationParameter(Scripts[i],"pos_y")
        dist[i] = math.sqrt(x_dif[i]^2 + y_dif[i]^2)
        simAuxiliaryConsolePrint(console,"Module#")
        simAuxiliaryConsolePrint(console,i)
        simAuxiliaryConsolePrint(console," distance: ")
        simAuxiliaryConsolePrint(console,dist[i])
        simAuxiliaryConsolePrint(console,"\n")
      end
      dist_min = math.min(dist[1],dist[2],dist[3],dist[4],dist[5],dist[6])
--//Selecting Role#1
      for i=1,6,1 do
        if(dist_min==dist[i]) then
          simAuxiliaryConsolePrint(console,"Role#1: Module#")
          simAuxiliaryConsolePrint(console,i)
          simAuxiliaryConsolePrint(console,"\n")
          simSetScriptSimulationParameter(Script,"Role1",Scripts[i])
          gpos_x[1] = simGetScriptSimulationParameter(Scripts[i],"pos_x")
          gpos_y[1] = simGetScriptSimulationParameter(Scripts[i],"pos_y")
          orntr1 = simGetScriptSimulationParameter(Scripts[i],"ornt")
        end
      end
      gpos_x[2] = gpos_x[1] - 0*math.sin(orntr1) + (-0.292)*math.cos(orntr1)
      gpos_y[2] = gpos_y[1] + 0*math.cos(orntr1) + (-0.292)*math.sin(orntr1)

      gpos_x[3] = gpos_x[1] - (-0.23)*math.sin(orntr1) + (0.035)*math.cos(orntr1)
      gpos_y[3] = gpos_y[1] + (-0.23)*math.cos(orntr1) + (0.035)*math.sin(orntr1)

      gpos_x[4] = gpos_x[1] - (0.23)*math.sin(orntr1) + (0.035)*math.cos(orntr1)
```

```
        gpos_y[4] = gpos_y[1] + (0.23)*math.cos(orntr1) + (0.035)*math.sin(orntr1)

        gpos_x[5] = gpos_x[1] - (-0.372)*math.sin(orntr1) + (-0.239)*math.cos(orntr1)
        gpos_y[5] = gpos_y[1] + (-0.372)*math.cos(orntr1) + (-0.239)*math.sin(orntr1)

        gpos_x[6] = gpos_x[1] - (0.372)*math.sin(orntr1) + (-0.239)*math.cos(orntr1)
        gpos_y[6] = gpos_y[1] + (0.372)*math.cos(orntr1) + (-0.239)*math.sin(orntr1)
--Selecting Role#2
        simAuxiliaryConsolePrint(console,"\nSelection of Role#2\n")
        for i=1,6,1 do
            x_dif[i] = gpos_x[2] - simGetScriptSimulationParameter(Scripts[i],"pos_x")
            y_dif[i] = gpos_y[2] - simGetScriptSimulationParameter(Scripts[i],"pos_y")
            dist[i] = math.sqrt(x_dif[i]^2 + y_dif[i]^2)
            if(Scripts[i]==simGetScriptSimulationParameter(Script,"Role1")) then
                dist[i] = math.huge
            end
            simAuxiliaryConsolePrint(console,"Module#")
            simAuxiliaryConsolePrint(console,i)
            simAuxiliaryConsolePrint(console," distance: ")
            simAuxiliaryConsolePrint(console,dist[i])
            simAuxiliaryConsolePrint(console,"\n")
        end
        dist_min = math.min(dist[1],dist[2],dist[3],dist[4],dist[5],dist[6])
        for i=1,6,1 do
            if(dist_min==dist[i]) then
                simAuxiliaryConsolePrint(console,"Role#2: Module#")
                simAuxiliaryConsolePrint(console,i)
                simAuxiliaryConsolePrint(console,"\n")
                simSetScriptSimulationParameter(Script,"Role2",Scripts[i])
            end
        end
--Selecting Role#3
        simAuxiliaryConsolePrint(console,"\nSelection of Role#3\n")
        for i=1,6,1 do
            x_dif[i] = gpos_x[3] - simGetScriptSimulationParameter(Scripts[i],"pos_x")
            y_dif[i] = gpos_y[3] - simGetScriptSimulationParameter(Scripts[i],"pos_y")
            dist[i] = math.sqrt(x_dif[i]^2 + y_dif[i]^2)
            if(Scripts[i]==simGetScriptSimulationParameter(Script,"Role1") or
                Scripts[i]==simGetScriptSimulationParameter(Script,"Role2")) then
                dist[i] = math.huge
            end
            simAuxiliaryConsolePrint(console,"Module#")
            simAuxiliaryConsolePrint(console,i)
            simAuxiliaryConsolePrint(console," distance: ")
            simAuxiliaryConsolePrint(console,dist[i])
            simAuxiliaryConsolePrint(console,"\n")
        end
        dist_min = math.min(dist[1],dist[2],dist[3],dist[4],dist[5],dist[6])
        for i=1,6,1 do
            if(dist_min==dist[i]) then
                simAuxiliaryConsolePrint(console,"Role#3: Module#")
                simAuxiliaryConsolePrint(console,i)
                simAuxiliaryConsolePrint(console,"\n")
                simSetScriptSimulationParameter(Script,"Role3",Scripts[i])
            end
        end
--Selecting Role#4
        simAuxiliaryConsolePrint(console,"\nSelection of Role#4\n")
        for i=1,6,1 do
            x_dif[i] = gpos_x[4] - simGetScriptSimulationParameter(Scripts[i],"pos_x")
            y_dif[i] = gpos_y[4] - simGetScriptSimulationParameter(Scripts[i],"pos_y")
            dist[i] = math.sqrt(x_dif[i]^2 + y_dif[i]^2)
            if(Scripts[i]==simGetScriptSimulationParameter(Script,"Role1") or
               Scripts[i]==simGetScriptSimulationParameter(Script,"Role2") or
               Scripts[i]==simGetScriptSimulationParameter(Script,"Role3")) then
               dist[i] = math.huge
            end
            simAuxiliaryConsolePrint(console,"Module#")
            simAuxiliaryConsolePrint(console,i)
            simAuxiliaryConsolePrint(console," distance: ")
            simAuxiliaryConsolePrint(console,dist[i])
            simAuxiliaryConsolePrint(console,"\n")
         end
         dist_min = math.min(dist[1],dist[2],dist[3],dist[4],dist[5],dist[6])
         for i=1,6,1 do
            if(dist_min==dist[i]) then
               simAuxiliaryConsolePrint(console,"Role#4: Module#")
               simAuxiliaryConsolePrint(console,i)
               simAuxiliaryConsolePrint(console,"\n")
               simSetScriptSimulationParameter(Script,"Role4",Scripts[i])
            end
         end
      --Selecting Role#5
         simAuxiliaryConsolePrint(console,"\nSelection of Role#5\n")
         for i=1,6,1 do
            x_dif[i] = gpos_x[5] - simGetScriptSimulationParameter(Scripts[i],"pos_x")
            y_dif[i] = gpos_y[5] - simGetScriptSimulationParameter(Scripts[i],"pos_y")
            dist[i] = math.sqrt(x_dif[i]^2 + y_dif[i]^2)
            if(Scripts[i]==simGetScriptSimulationParameter(Script,"Role1") or
               Scripts[i]==simGetScriptSimulationParameter(Script,"Role2") or
               Scripts[i]==simGetScriptSimulationParameter(Script,"Role3") or
               Scripts[i]==simGetScriptSimulationParameter(Script,"Role4")) then
               dist[i] = math.huge
            end
            simAuxiliaryConsolePrint(console,"Module#")
            simAuxiliaryConsolePrint(console,i)
            simAuxiliaryConsolePrint(console," distance: ")
            simAuxiliaryConsolePrint(console,dist[i])
            simAuxiliaryConsolePrint(console,"\n")
         end
         dist_min = math.min(dist[1],dist[2],dist[3],dist[4],dist[5],dist[6])
         for i=1,6,1 do
            if(dist_min==dist[i]) then
               simAuxiliaryConsolePrint(console,"Role#5: Module#")
               simAuxiliaryConsolePrint(console,i)
               simAuxiliaryConsolePrint(console,"\n")
               simSetScriptSimulationParameter(Script,"Role5",Scripts[i])
            end
         end
      --Selecting Role#6
         simAuxiliaryConsolePrint(console,"\nSelection of Role#6\n")
         for i=1,6,1 do
            x_dif[i] = gpos_x[6] - simGetScriptSimulationParameter(Scripts[i],"pos_x")
            y_dif[i] = gpos_y[6] - simGetScriptSimulationParameter(Scripts[i],"pos_y")
            dist[i] = math.sqrt(x_dif[i]^2 + y_dif[i]^2)
            if(Scripts[i]==simGetScriptSimulationParameter(Script,"Role1") or
               Scripts[i]==simGetScriptSimulationParameter(Script,"Role2") or
               Scripts[i]==simGetScriptSimulationParameter(Script,"Role3") or
               Scripts[i]==simGetScriptSimulationParameter(Script,"Role4") or
               Scripts[i]==simGetScriptSimulationParameter(Script,"Role5")) then
            dist[i] = math.huge
        end
        simAuxiliaryConsolePrint(console,"Module#")
        simAuxiliaryConsolePrint(console,i)
        simAuxiliaryConsolePrint(console," distance: ")
        simAuxiliaryConsolePrint(console,dist[i])
        simAuxiliaryConsolePrint(console,"\n")
    end
    dist_min = math.min(dist[1],dist[2],dist[3],dist[4],dist[5],dist[6])
    for i=1,6,1 do
        if(dist_min==dist[i]) then
            simAuxiliaryConsolePrint(console,"Role#6: Module#")
            simAuxiliaryConsolePrint(console,i)
            simAuxiliaryConsolePrint(console,"\n")
            simSetScriptSimulationParameter(Script,"Role6",Scripts[i])
        end
    end
    for i=1,6,1 do
        simSetScriptSimulationParameter(Scripts[i],"Notification1",1)
    end
    simAuxiliaryConsolePrint(console,"Role Distribution Done\n\n")
    rd_done=1
    return 1
end

function get_role()
    rd_done = 0
    if(simGetScriptSimulationParameter(Script,"Notification1")==1) then

        Roles =
{simGetScriptSimulationParameter(Scripts[1],"Role1"),simGetScriptSimulationParameter(Scripts[1],"
Role2"),

simGetScriptSimulationParameter(Scripts[1],"Role3"),simGetScriptSimulationParameter(Scripts[1],"
Role4"),

simGetScriptSimulationParameter(Scripts[1],"Role5"),simGetScriptSimulationParameter(Scripts[1],"
Role6")}

        if(simGetScriptSimulationParameter(Scripts[1],"Role1")==Script) then
            role = 1
        end
        if(simGetScriptSimulationParameter(Scripts[1],"Role2")==Script) then
            role = 2
        end
        if(simGetScriptSimulationParameter(Scripts[1],"Role3")==Script) then
            role = 3
        end
        if(simGetScriptSimulationParameter(Scripts[1],"Role4")==Script) then
            role = 4
        end
        if(simGetScriptSimulationParameter(Scripts[1],"Role5")==Script) then
            role = 5
        end
        if(simGetScriptSimulationParameter(Scripts[1],"Role6")==Script) then
            role = 6
        end
        simSetScriptSimulationParameter(Script,"Notification1",0)
        simSetScriptSimulationParameter(Script,"role",role)
        rd_done = 1
        end
        return rd_done,role
    end
    function strategic_planning()
        simAuxiliaryConsolePrint(console,"Strategic Planning Algorithm is Initiated\n")
        for i=7,9,1 do
            if(simGetScriptSimulationParameter(Scripts[i],"Type")==3) then
                sys_target_handle = det_points[i]
                sys_target_pos_x = simGetScriptSimulationParameter(Scripts[i],"pos_x")
                sys_target_pos_y = simGetScriptSimulationParameter(Scripts[i],"pos_y")
                simAuxiliaryConsolePrint(console,"Target Position x:")
                simAuxiliaryConsolePrint(console,sys_target_pos_x)
                simAuxiliaryConsolePrint(console,"\n")
                simAuxiliaryConsolePrint(console,"Target Position y:")
                simAuxiliaryConsolePrint(console,sys_target_pos_y)
                simAuxiliaryConsolePrint(console,"\n\n")
            end
            if(simGetScriptSimulationParameter(Scripts[i],"Type")==2) then
                sys_wh_obs_handle = det_points[i]
                sys_wh_obs_pos_x = simGetScriptSimulationParameter(Scripts[i],"pos_x")
                sys_wh_obs_pos_y = simGetScriptSimulationParameter(Scripts[i],"pos_y")
                simAuxiliaryConsolePrint(console,"Lath Obstacle Position x:")
                simAuxiliaryConsolePrint(console,sys_wh_obs_pos_x)
                simAuxiliaryConsolePrint(console,"\n")
                simAuxiliaryConsolePrint(console,"Lath Obstacle Position y:")
                simAuxiliaryConsolePrint(console,sys_wh_obs_pos_y)
                simAuxiliaryConsolePrint(console,"\n\n")
            end
            if(simGetScriptSimulationParameter(Scripts[i],"Type")==1) then
                sys_qd_obs_handle = det_points[i]
                sys_qd_obs_pos_x = simGetScriptSimulationParameter(Scripts[i],"pos_x")
                sys_qd_obs_pos_y = simGetScriptSimulationParameter(Scripts[i],"pos_y")
                simAuxiliaryConsolePrint(console,"Ground Obstacle Position x:")
                simAuxiliaryConsolePrint(console,sys_qd_obs_pos_x)
                simAuxiliaryConsolePrint(console,"\n")
                simAuxiliaryConsolePrint(console,"Ground Obstacle Position y:")
                simAuxiliaryConsolePrint(console,sys_qd_obs_pos_y)
                simAuxiliaryConsolePrint(console,"\n\n")
            end
        end
        sys_wh_obs_dist = math.sqrt((sys_target_pos_x-sys_wh_obs_pos_x)^2 + (sys_target_pos_y-
sys_wh_obs_pos_y)^2)
        sys_qd_obs_dist = math.sqrt((sys_target_pos_x-sys_qd_obs_pos_x)^2 + (sys_target_pos_y-
sys_qd_obs_pos_y)^2)

--plan_rowx format: {conf_req,reference_handle,lat_const,long_const,gamma_const,transport_mode}
--transport_mode=0 normal, transport_mode=1 ability
        if(sys_wh_obs_dist>sys_qd_obs_dist) then
            simAuxiliaryConsolePrint(console,"First Obstacle to Pass: Lath Obstacle\n")
            plan_row1={2,sys_wh_obs_handle,0,-0.3,0,0}
            plan_row2={2,sys_wh_obs_handle,0,0.3,0,1}
            plan_row3={1,sys_qd_obs_handle,0,0,-0.3,0}
            plan_row4={1,sys_qd_obs_handle,0,0,0.5,1}
            plan_row5={1,sys_qd_obs_handle,0,0.5,0,0}
            plan_row6={1,sys_target_handle,0,0,0,0}
        end
        if(sys_wh_obs_dist<sys_qd_obs_dist) then
            simAuxiliaryConsolePrint(console,"First Obstacle to Pass: Ground Obstacle\n\n")
            plan_row1={1,sys_qd_obs_handle,0,-0.1,0,0}
            plan_row2={1,sys_qd_obs_handle,0,0.5,0,1}
        plan_row3={1,sys_qd_obs_handle,0,0.5,0,0}
        plan_row4={2,sys_wh_obs_handle,0,-0.3,0,0}
        plan_row5={2,sys_wh_obs_handle,0,0.3,0,1}
        plan_row6={2,sys_target_handle,0,0,0,0}
    end

    simAuxiliaryConsolePrint(console,"Plan Matrix:\n\n")
------------------------------
    simAuxiliaryConsolePrint(console,"Plan Row 1=[")
    simAuxiliaryConsolePrint(console,plan_row1[1])
    simAuxiliaryConsolePrint(console,",")
    if(plan_row1[2]==sys_target_handle) then
        simAuxiliaryConsolePrint(console,"System Target")
    end
    if(plan_row1[2]==sys_qd_obs_handle) then
        simAuxiliaryConsolePrint(console,"Ground Obstacle")
    end
    if(plan_row1[2]==sys_wh_obs_handle) then
        simAuxiliaryConsolePrint(console,"Lath Obstacle")
    end
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row1[3])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row1[4])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row1[5])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row1[6])
    simAuxiliaryConsolePrint(console,"]\n")
-------------------------------------

    simAuxiliaryConsolePrint(console,"Plan Row 2=[")
    simAuxiliaryConsolePrint(console,plan_row2[1])
    simAuxiliaryConsolePrint(console,",")
    if(plan_row2[2]==sys_target_handle) then
        simAuxiliaryConsolePrint(console,"System Target")
    end
    if(plan_row2[2]==sys_qd_obs_handle) then
        simAuxiliaryConsolePrint(console,"Ground Obstacle")
    end
    if(plan_row2[2]==sys_wh_obs_handle) then
        simAuxiliaryConsolePrint(console,"Lath Obstacle")
    end
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row2[3])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row2[4])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row2[5])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row2[6])
    simAuxiliaryConsolePrint(console,"]\n")
-----------------------------

    simAuxiliaryConsolePrint(console,"Plan Row 3=[")
    simAuxiliaryConsolePrint(console,plan_row3[1])
    simAuxiliaryConsolePrint(console,",")
    if(plan_row3[2]==sys_target_handle) then
        simAuxiliaryConsolePrint(console,"System Target")
    end
    if(plan_row3[2]==sys_qd_obs_handle) then
       simAuxiliaryConsolePrint(console,"Ground Obstacle")
    end
    if(plan_row3[2]==sys_wh_obs_handle) then
       simAuxiliaryConsolePrint(console,"Lath Obstacle")
    end
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row3[3])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row3[4])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row3[5])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row3[6])
    simAuxiliaryConsolePrint(console,"]\n")
-----------------------------------

    simAuxiliaryConsolePrint(console,"Plan Row 4=[")
    simAuxiliaryConsolePrint(console,plan_row4[1])
    simAuxiliaryConsolePrint(console,",")
    if(plan_row4[2]==sys_target_handle) then
       simAuxiliaryConsolePrint(console,"System Target")
    end
    if(plan_row4[2]==sys_qd_obs_handle) then
       simAuxiliaryConsolePrint(console,"Ground Obstacle")
    end
    if(plan_row4[2]==sys_wh_obs_handle) then
       simAuxiliaryConsolePrint(console,"Lath Obstacle")
    end
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row4[3])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row4[4])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row4[5])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row4[6])
    simAuxiliaryConsolePrint(console,"]\n")
-------------------------------------
    simAuxiliaryConsolePrint(console,"Plan Row 5=[")
    simAuxiliaryConsolePrint(console,plan_row5[1])
    simAuxiliaryConsolePrint(console,",")
    if(plan_row5[2]==sys_target_handle) then
       simAuxiliaryConsolePrint(console,"System Target")
    end
    if(plan_row5[2]==sys_qd_obs_handle) then
       simAuxiliaryConsolePrint(console,"Ground Obstacle")
    end
    if(plan_row5[2]==sys_wh_obs_handle) then
       simAuxiliaryConsolePrint(console,"Lath Obstacle")
    end
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row5[3])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row5[4])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row5[5])
    simAuxiliaryConsolePrint(console,",")
    simAuxiliaryConsolePrint(console,plan_row5[6])
    simAuxiliaryConsolePrint(console,"]\n")
```

```
----------------------------------

        simAuxiliaryConsolePrint(console,"Plan Row 6=[")
        simAuxiliaryConsolePrint(console,plan_row6[1])
        simAuxiliaryConsolePrint(console,",")
        if(plan_row6[2]==sys_target_handle) then
            simAuxiliaryConsolePrint(console,"System Target")
        end
        if(plan_row6[2]==sys_gd_obs_handle) then
            simAuxiliaryConsolePrint(console,"Ground Obstacle")
        end
        if(plan_row6[2]==sys_wh_obs_handle) then
            simAuxiliaryConsolePrint(console,"Lath Obstacle")
        end
        simAuxiliaryConsolePrint(console,",")
        simAuxiliaryConsolePrint(console,plan_row6[3])
        simAuxiliaryConsolePrint(console,",")
        simAuxiliaryConsolePrint(console,plan_row6[4])
        simAuxiliaryConsolePrint(console,",")
        simAuxiliaryConsolePrint(console,plan_row6[5])
        simAuxiliaryConsolePrint(console,",")
        simAuxiliaryConsolePrint(console,plan_row6[6])
        simAuxiliaryConsolePrint(console,"]\n\n")
----------------------------------
        plan_counter = 1
        current_target = 0
        sp_done = 1
        simAuxiliaryConsolePrint(console,"Strategic Planning Done\n\n")

        return sp_done
    end
    function sequencing(plan_counter)
        simAuxiliaryConsolePrint(console,"Sequencing Phase\n")
        if(plan_counter==1) then
            plan_row = plan_row1
        end
        if(plan_counter==2) then
            plan_row = plan_row2
        end
        if(plan_counter==3) then
            plan_row = plan_row3
        end
        if(plan_counter==4) then
            plan_row = plan_row4
        end
        if(plan_counter==5) then
            plan_row = plan_row5
        end
        if(plan_counter==6) then
            plan_row = plan_row6
        end

        simAuxiliaryConsolePrint(console,"Active Sub-goal: Plan Row#")
        simAuxiliaryConsolePrint(console, plan_counter)
        simAuxiliaryConsolePrint(console,"=[")
        simAuxiliaryConsolePrint(console,plan_row[1])
        simAuxiliaryConsolePrint(console,",")
        if(plan_row[2]==sys_target_handle) then
            simAuxiliaryConsolePrint(console,"System Target")
        end
```

```
  if(plan_row[2]==sys_qd_obs_handle) then
    simAuxiliaryConsolePrint(console,"Ground Obstacle")
  end
  if(plan_row[2]==sys_wh_obs_handle) then
    simAuxiliaryConsolePrint(console,"Lath Obstacle")
  end
  simAuxiliaryConsolePrint(console,",")
  simAuxiliaryConsolePrint(console,plan_row[3])
  simAuxiliaryConsolePrint(console,",")
  simAuxiliaryConsolePrint(console,plan_row[4])
  simAuxiliaryConsolePrint(console,",")
  simAuxiliaryConsolePrint(console,plan_row[5])
  simAuxiliaryConsolePrint(console,",")
  simAuxiliaryConsolePrint(console,plan_row[6])
  simAuxiliaryConsolePrint(console,"]\n\n")
  conf_req = plan_row[1]
  current_target = plan_row[2]
  lat_const = plan_row[3]
  long_const = plan_row[4]
  gamma_const = plan_row[5]
  transport_mode = plan_row[6]
  simSetScriptSimulationParameter(Script,"conf_req",conf_req)
  for i=1,6,1 do
    simSetScriptSimulationParameter(Scripts[i],"Notification2",1)
  end
  sq_done = 1
  return sq_done,conf_req
end

function get_plan(role)
  if(simGetScriptSimulationParameter(Script,"Notification2")==1) then
    conf_req = simGetScriptSimulationParameter(Roles[1],"conf_req")
    if(conf_req==1) then
      if(role==1) then
        simSetScriptSimulationParameter(Script,"target_handle", det_point)
        simSetScriptSimulationParameter(Roles[2],"connection_target",conn_points[2])
        simSetScriptSimulationParameter(Roles[3],"connection_target",conn_points[3])
        simSetScriptSimulationParameter(Roles[4],"connection_target",conn_points[4])
      end
      if(role==2) then
        simSetScriptSimulationParameter(Roles[5],"connection_target",conn_points[4])
        simSetScriptSimulationParameter(Roles[6],"connection_target",conn_points[3])
      end
      if(role==3) then

      end
      if(role==4) then

      end
      if(role==5) then

      end
      if(role==6) then

      end
    end
    if(conf_req==2) then
      if(role==1) then
        simSetScriptSimulationParameter(Script,"target_handle", det_point)
        simSetScriptSimulationParameter(Roles[2],"connection_target",conn_points[2])
```

```
                simSetScriptSimulationParameter(Roles[3],"connection_target",conn_points[3])
                simSetScriptSimulationParameter(Roles[4],"connection_target",conn_points[4])
            end
            if(role==2) then
                simSetScriptSimulationParameter(Roles[5],"connection_target",conn_points[4])
                simSetScriptSimulationParameter(Roles[6],"connection_target",conn_points[3])
            end
            if(role==3) then

            end
            if(role==4) then

            end
            if(role==5) then

            end
            if(role==6) then

            end
        end
        simSetScriptSimulationParameter(Script,"Notification2",0)
        simSetScriptSimulationParameter(Script,"conf_req",conf_req)
        simSetScriptSimulationParameter(Script,"role",role)
        sq_done = 1
    end
    return sq_done,conf_req
end

function get_target_state(conf_req,role,assembly_counter)
    if(conf_req==1) then
--Teze yazilanlar yanlis tezi duzelt
        if(role==2) then
            target_handle = simGetScriptSimulationParameter(Roles[1],"target_handle")
            lat_cons={0,0}
            long_cons={-0.292,-0.112}
            gamma_cons={180,180}
        end
        if(role==3) then
            target_handle = simGetScriptSimulationParameter(Roles[1],"target_handle")
            lat_cons={-0.23,-0.19}
            long_cons={0.035,0.035}
            gamma_cons={90,90}
        end
        if(role==4) then
            target_handle = simGetScriptSimulationParameter(Roles[1],"target_handle")
            lat_cons={0.23,0.19}
            long_cons={0.035,0.035}
            gamma_cons={-90,-90}
        end
        if(role==5) then
            target_handle = simGetScriptSimulationParameter(Roles[1],"target_handle")
            lat_cons={-0.372,-0.19}
            long_cons={-0.239,-0.239}
            gamma_cons={90,90}
        end
        if(role==6) then
            target_handle = simGetScriptSimulationParameter(Roles[1],"target_handle")
            lat_cons={0.372,0.19}
            long_cons={-0.239,-0.239}
            gamma_cons={-90,-90}
            end
        end

        if(conf_req==2) then
            if(role==2) then
                target_handle = simGetScriptSimulationParameter(Roles[1],"target_handle")
                lat_cons={0,0}
                long_cons={-0.292,-0.112}
                gamma_cons={180,180}
            end
            if(role==3) then
                target_handle = simGetScriptSimulationParameter(Roles[1],"target_handle")
                lat_cons={-0.23,-0.04}
                long_cons={0.035,0.035}
                gamma_cons={-90,-90}
            end
            if(role==4) then
                target_handle = simGetScriptSimulationParameter(Roles[1],"target_handle")
                lat_cons={0.23,0.04}
                long_cons={0.035,0.035}
                gamma_cons={90,90}
            end
            if(role==5) then
                target_handle = simGetScriptSimulationParameter(Roles[1],"target_handle")
                lat_cons={-0.372,-0.04}
                long_cons={-0.239,-0.239}
                gamma_cons={-90,-90}
            end
            if(role==6) then
                target_handle = simGetScriptSimulationParameter(Roles[1],"target_handle")
                lat_cons={0.372,0.04}
                long_cons={-0.239,-0.239}
                gamma_cons={90,90}
            end
        end

        return target_handle, lat_cons[assembly_counter],
long_cons[assembly_counter],gamma_cons[assembly_counter]
    end

    function ornt_dif_corr(ornt_dif)
        ornt_dif = math.deg(ornt_dif)
        if(math.abs(ornt_dif)>=180) then
            if(ornt_dif<0) then
                ornt_dif_dummy = ornt_dif+360
            end
            if(ornt_dif>0) then
                ornt_dif_dummy = ornt_dif-360
            end
            ornt_dif = ornt_dif_dummy
        end
        return math.rad(ornt_dif)
    end
--//Search Function---------------------------------------------------------
--Search function of the module takes target handle as input returns
--target_found status
    function search_target(target_handle, search_counter, CamJ1pos)
        simSetJointTargetPosition(CamJoint2,math.rad(-10*search_counter))
        result,distance,detP = simCheckProximitySensor(Cam,target_handle)
        if(result==1) then
```

```
            target_found=1
            search_counter = 0
        end
        if(result~=1) then
            CamJ1pos = CamJ1pos + 10
            if(CamJ1pos > 360) then
                CamJ1pos = 0
                search_counter = search_counter + 1
                if(search_counter==15) then
                    search_counter = 0
                end
            end
            simSetJointTargetPosition(CamJoint1,math.rad(CamJ1pos))
        end


        return target_found, search_counter, CamJ1pos
    end --Function End
--Search Function//----------------------------------------------------------

--//Locking Function--------------------------------------------------------
--Makes the camera lock to its target. This function is called continually
--to keep the module tracking its target. Takes target handle as input,
--returns nothing
    function lock_target(target_handle)
        result,distance,detP = simCheckProximitySensor(Cam,target_handle)

        if(result==1) then
            CamJ1pos = simGetJointPosition(CamJoint1)
            CamJ2pos = simGetJointPosition(CamJoint2)

            CamJ1pos_rot = math.atan2(detP[1],detP[3])
            CamJ2pos_rot = math.atan2(detP[2],detP[3])

            CamJ1pos = CamJ1pos+CamJ1pos_rot
            CamJ2pos = CamJ2pos+CamJ2pos_rot

            simSetJointTargetPosition(CamJoint1,CamJ1pos)
            simSetJointTargetPosition(CamJoint2,CamJ2pos)
            target_found = 1
        end
        if(result==0) then
            target_found = 0
        end
        return target_found,CamJ1pos
    end --Function End
--Locking function//--------------------------------------------------------

--//Distance Calculation Function------------------------------------------
--Calculates lateral and longitudinal distance between target and module
--reference position. Takes target handle, returns lateral distance, longitudinal
--distance and gamma orientation difference.
--mode=0: single module calculation
--mode=1: quadruped or wheeled calculation
    function calculate_difference(target_handle,lat_const,long_const,mode)
        target_pos_reached = 0
        result,distance,detP = simCheckProximitySensor(Cam,target_handle)
        if(result==1) then
            target_found = 1
            CamJ1pos = simGetJointPosition(CamJoint1)
            CamJ2pos = simGetJointPosition(CamJoint2)

            ornt = simGetObjectOrientation(pos_dummy,-1)
            target_ornt3 = simGetObjectOrientation(target_handle,-1)
            target_ornt = target_ornt3[3]

            matrix = simGetObjectMatrix(cam_dummy,-1)
            matrix[4] = 0
            matrix[8] = 0
            matrix[12] = 0
            det = {detP[3],detP[1],detP[2]}
            target = simMultiplyVector(matrix,det)
            if(mode==0) then
               long_dif = target[1] + 0.102*math.cos(ornt[3]) +
(0.0045+0.011*math.cos(CamJ2pos))*math.cos(ornt[3]+CamJ1pos)
               lat_dif = target[2] + 0.102*math.sin(ornt[3]) +
(0.0045+0.011*math.cos(CamJ2pos))*math.sin(ornt[3]+CamJ1pos)

               long_dif = long_dif - lat_const*math.sin(target_ornt)+long_const*math.cos(target_ornt)
               lat_dif = lat_dif + lat_const*math.cos(target_ornt)+long_const*math.sin(target_ornt)
            end
            if(mode==1) then
               long_dif = target[1] + 0.0525*math.cos(ornt[3]) +
(0.0045+0.011*math.cos(CamJ2pos))*math.cos(ornt[3]+CamJ1pos)
               lat_dif = target[2] + 0.0525*math.sin(ornt[3]) +
(0.0045+0.011*math.cos(CamJ2pos))*math.sin(ornt[3]+CamJ1pos)

               long_dif = long_dif - lat_const*math.sin(target_ornt)+long_const*math.cos(target_ornt)
               lat_dif = lat_dif + lat_const*math.cos(target_ornt)+long_const*math.sin(target_ornt)

               dist = math.sqrt(long_dif^2 + lat_dif^2)
               ornt_dif = math.atan2(lat_dif,long_dif)-ornt[3]
               long_dif = dist*math.cos(ornt_dif)
               lat_dif = dist*math.sin(ornt_dif)
            end
            if(mode==2) then
               long_dif = target[1] + 0.0525*math.cos(ornt[3]) +
(0.0045+0.011*math.cos(CamJ2pos))*math.cos(ornt[3]+CamJ1pos)
               lat_dif = target[2] + 0.0525*math.sin(ornt[3]) +
(0.0045+0.011*math.cos(CamJ2pos))*math.sin(ornt[3]+CamJ1pos)

               long_dif = long_dif - lat_const*math.sin(target_ornt)+long_const*math.cos(target_ornt)
               lat_dif = lat_dif + lat_const*math.cos(target_ornt)+long_const*math.sin(target_ornt)
            end
         end
         if(result~=1) then
            target_found = 0
         end

         return target_found,lat_dif,long_dif

   end --Function End
--Distance Calculation Function//---------------------------------------------

--//Direction Decision Function----------------------------------------------
--Decides the direction of the moving gait, forward or backward. Takes lateral
--and longitudinal difference between the target and the module and returns
--direction decision to be used in move_single() function. "direction=1" means forward and
--"direction=-1" means backward.
   function decide_direction(lat_dif,long_dif,gamma_const,forced_direction)
```

```
        ornt_req = math.atan2(lat_dif,long_dif)
        ornt_des = target_ornt + math.rad(gamma_const)

        Modornt3 = simGetObjectOrientation(pos_dummy,-1)
        Modornt = Modornt3[3]
        ornt_dif_forw = ornt_dif_corr(ornt_req-Modornt)
        ornt_dif_back = ornt_dif_corr(ornt_req-Modornt+math.rad(180))

        t1_forw = math.abs(math.deg(ornt_dif_forw))*t180/180
        t1_back = math.abs(math.deg(ornt_dif_back))*t180/180

        distance = math.sqrt((lat_dif^2)+(long_dif^2))
        t2_forw = math.floor(distance/dfhv)*tfhv + (math.floor(distance-
math.floor(distance/dfhv)*dfhv)/dflv)*tflv
        t2_back = math.floor(distance/dbhv)*tbhv + (math.floor(distance-
math.floor(distance/dbhv)*dbhv)/dblv)*tblv

        t3_forw = math.abs(math.deg(ornt_dif_corr(ornt_des-ornt_req))*t180/180)
        t3_back = math.abs(math.deg(ornt_dif_corr(ornt_des-ornt_req+math.rad(180)))*t180/180)

        if(forced_direction==0) then
            if(t1_forw+t2_forw+t3_forw>t1_back+t2_back+t3_back) then
                direction = -1
            end
            if(t1_forw+t2_forw+t3_forw<t1_back+t2_back+t3_back) then
                direction = 1
            end
        end
        if(forced_direction~=0) then
            direction = forced_direction
        end
        if(direction==-1) then
            ornt_req = ornt_req + math.rad(180)
        end
        return direction,ornt_req
    end --Function End
--Direction Decision Function//-----------------------------------------------

--//Velocity Decision Function-----------------------------------------------
--Decides the velocity of the moving gait, high or low. Takes lateral
--and longitudinal difference between the target and the module and returns
--velocity decision to be used in move_single() function. "velocity=2" means
--high velocity and "velocity=1" means low velocity.
    function decide_velocity(direction,lat_dif,long_dif)
        distance = math.sqrt((lat_dif^2)+(long_dif^2))
        if(direction==1) then
            if(distance>=dfhv) then
                velocity = 2
                target_pos_reached = 0
            end
            if(distance<dfhv and distance>=dflv) then
                velocity = 1
                target_pos_reached = 0
            end
            if(distance<dflv) then
                target_pos_reached = 1
--              velocity = 0
            end
        end
    if(direction==-1) then
       if(distance>=dbhv) then
          velocity = 2
          target_pos_reached = 0
       end
       if(distance<dbhv and distance>=dblv) then
          velocity = 1
          target_pos_reached = 0
       end
       if(distance<dblv) then
          target_pos_reached = 1
--           velocity = 0
       end
    end

    return target_pos_reached,velocity
 end --Function End
--Velocity Decision Function//------------------------------------------------

--//Orientation Correction Function--------------------------------------------
 function correct_ornt(J1pos,J2pos,ornt_req)
    Modornt3 = simGetObjectOrientation(pos_dummy,-1)
    Modornt = Modornt3[3]
    ornt_dif = ornt_dif_corr(ornt_req-Modornt)
    FJpos = math.deg(simGetJointPosition(FrontJoint))

    if(math.abs(math.deg(ornt_dif))<1)  then
       wheel_spd = 0
    end
    if(math.abs(math.deg(ornt_dif))<10) then
       if(math.deg(ornt_dif)<0) then
          wheel_spd = -1.8*math.deg(ornt_dif)
       end
       if(math.deg(ornt_dif)>0) then
          wheel_spd = -1.8*math.deg(ornt_dif)
       end
    end
    if(math.abs(math.deg(ornt_dif))>10) then
       J1pos = 15
       J2pos = 0

       if(math.deg(ornt_dif)<0) then
          wheel_spd = 18
       end
       if(math.deg(ornt_dif)>0) then
          wheel_spd = -18
       end
    end
    FJpos = FJpos + wheel_spd
    simSetJointTargetPosition(FrontJoint,math.rad(FJpos))
    simSetJointTargetPosition(BackJoint1,math.rad(J1pos))
    simSetJointTargetPosition(BackJoint2,math.rad(J2pos))

    return J1pos,J2pos,ornt_dif
 end --Function End


--Orientation Correction Function//--------------------------------------------
--//Single Locomotion Function------------------------------------------------
--Increases the step of the locomotion gait by one whenever its called. Joint
```

```
--position controls are done in the main loop because of V-Rep limitations. Takes
--direction, velocity, forward step and backward step value. Returns updated forward
--step and backward step value
  function move_single(direction,velocity,forw_step,back_step,ornt_dif)
    ms_result=0
    if(math.abs(math.deg(ornt_dif))<10) then
      if(math.abs(J1pos-math.deg(simGetJointPosition(BackJoint1)))<1 and
        math.abs(J2pos-math.deg(simGetJointPosition(BackJoint2)))<1) then
        ms_result=1
        if(direction==1)then
          if(velocity==1) then
            if(forw_step==0) then
              forw_step_f = 1
            end
            if(forw_step==1) then
              J1pos = -33
              J2pos = 61
              forw_step_f = 2
            end
            if(forw_step==2) then
              J1pos = 0
              J2pos = 0
              forw_step_f = 0
            end
          end
          if(velocity==2) then
            if(forw_step == 0) then
              forw_step_f = 1
            end
            if(forw_step==1) then
              J1pos = -41
              J2pos = 120
              forw_step_f = 2
            end
            if(forw_step==2) then
              J1pos = 0
              J2pos = 0
              forw_step_f = 0
            end
          end
        end
        if(direction==-1)then
          if(velocity==1) then
            if(back_step==0) then
              back_step_f = 1
            end
            if(back_step==1) then
              J1pos = -1
              J2pos = 34
              back_step_f = 2
            end
            if(back_step==2) then
              J1pos = -27
              J2pos = 52
              back_step_f = 3
            end
            if(back_step==3) then
              J1pos = -15
              J2pos = 15
              back_step_f = 4
```

```
                  end
                  if(back_step==4) then
                     J1pos = 0
                     J2pos = 0
                     back_step_f = 0
                  end
               end
               if(velocity==2) then
                  if(back_step == 0) then
                     back_step_f = 1
                  end
                  if(back_step==1) then
                     J1pos = -13
                     J2pos = 72
                     back_step_f = 2
                  end
                  if(back_step==2) then
                     J1pos = -36
                     J2pos = 90
                     back_step_f = 3
                  end
                  if(back_step==3) then
                     J1pos = -30
                     J2pos = 30
                     back_step_f = 4
                  end
                  if(back_step==4) then
                     J1pos = 0
                     J2pos = 0
                     back_step_f = 0
                  end
               end
            end
            simSetJointTargetPosition(BackJoint1,math.rad(J1pos))
            simSetJointTargetPosition(BackJoint2,math.rad(J2pos))
         end
      end

      return ms_result,J1pos,J2pos,forw_step_f,back_step_f

   end --Function End
--Single Locomotion Function//-----------------------------------------------
   function assembly_step_up(conf_req,role,assembly_counter)
      forced_direction = 0
      step_up = false
      if(conf_req==1) then
         if(role==1) then

         end
         if(role==2) then
            step_up = true
            if(assembly_counter>=1) then
               forced_direction=-1
            end
         end
         if(role==3) then
            step_up = true
         end
         if(role==4) then
            step_up = true
```

```
        end
        if(role==5) then
--           if(simGetScriptSimulationParameter(Roles[2],"assembly_counter")>assembly_counter or
            if(simGetScriptSimulationParameter(Roles[2],"connected")==1) then
                step_up = true
            end
        end
        if(role==6) then
--           if(simGetScriptSimulationParameter(Roles[2],"assembly_counter")>assembly_counter or
            if(simGetScriptSimulationParameter(Roles[2],"connected")==1) then
                step_up = true
            end
        end
    end
    if(conf_req==2) then
        if(role==1) then

        end
        if(role==2) then
            step_up = true
            if(assembly_counter>=1) then
                forced_direction=-1
            end
        end
        if(role==3) then
            step_up = true
            if(assembly_counter>=1) then
                forced_direction=-1
            end
        end
        if(role==4) then
            step_up = true
            if(assembly_counter>=1) then
                forced_direction=-1
            end
        end
        if(role==5) then
            if(simGetScriptSimulationParameter(Roles[2],"assembly_counter")>assembly_counter or
                simGetScriptSimulationParameter(Roles[2],"connected")==1) then
                step_up = true
            end
            if(assembly_counter>=1) then
                forced_direction=-1
            end
        end
        if(role==6) then
            if(simGetScriptSimulationParameter(Roles[2],"assembly_counter")>assembly_counter or
                simGetScriptSimulationParameter(Roles[2],"connected")==1) then
                step_up = true
            end
            if(assembly_counter>=1) then
                forced_direction=-1
            end
        end
    end

    return step_up,forced_direction
end
function connect()
    result = false
```

```
        if(conf_req==1) then
            if(role==1) then

            end
            if(role==2) then
simSetLinkDummy(conn_points[5],simGetScriptSimulationParameter(Script,"connection_target"))
                result = true
            end
            if(role==3) then
simSetLinkDummy(conn_points[1],simGetScriptSimulationParameter(Script,"connection_target"))
                result = true
            end
            if(role==4) then
simSetLinkDummy(conn_points[1],simGetScriptSimulationParameter(Script,"connection_target"))
                result = true
            end
            if(role==5) then
                if(simGetScriptSimulationParameter(Roles[2],"connected")==1) then
simSetLinkDummy(conn_points[1],simGetScriptSimulationParameter(Script,"connection_target"))
                    result = true
                end
            end
            if(role==6) then
                if(simGetScriptSimulationParameter(Roles[2],"connected")==1) then
simSetLinkDummy(conn_points[1],simGetScriptSimulationParameter(Script,"connection_target"))
                    result = true
                end
            end
        end
        if(conf_req==2) then
            if(role==1) then

            end
            if(role==2) then
simSetLinkDummy(conn_points[5],simGetScriptSimulationParameter(Script,"connection_target"))
                result = true
            end
            if(role==3) then
simSetLinkDummy(conn_points[2],simGetScriptSimulationParameter(Script,"connection_target"))
                result = true
            end
            if(role==4) then
simSetLinkDummy(conn_points[2],simGetScriptSimulationParameter(Script,"connection_target"))
                result = true
            end
            if(role==5) then
                if(simGetScriptSimulationParameter(Roles[2],"connected")==1) then
simSetLinkDummy(conn_points[2],simGetScriptSimulationParameter(Script,"connection_target"))
                    result = true
                end
        end
        if(role==6) then
            if(simGetScriptSimulationParameter(Roles[2],"connected")==1) then

simSetLinkDummy(conn_points[2],simGetScriptSimulationParameter(Script,"connection_target"))
                result = true
            end
        end
    end
    return result
end
function send_order(ScrHandle, BJ1pos_conf, BJ2pos_conf, Cylpos_conf)
    simSetScriptSimulationParameter(ScrHandle,"BJ1pos_conf",BJ1pos_conf)
    simSetScriptSimulationParameter(ScrHandle,"BJ2pos_conf",BJ2pos_conf)
    simSetScriptSimulationParameter(ScrHandle,"Cylpos_conf",Cylpos_conf)
    simSetScriptSimulationParameter(ScrHandle,"Order_sent",1)
    simSetScriptSimulationParameter(ScrHandle,"Order_done",0)
    return 0
end
function angular_diff_rad(angle1,angle2)
    angle1 = math.atan2(math.sin(angle1),math.cos(angle1))
    angle2 = math.atan2(math.sin(angle2),math.cos(angle2))

    return (angle1-angle2)
end
function conf_init(conf,transport_mode)
    if(conf==1) then
        simSetJointTargetPosition(BackJoint1,math.rad(-90))
        simSetJointTargetPosition(BackJoint2,math.rad(90))
        send_order(Roles[2],-90,90,0)
        send_order(Roles[3],60,30,0)
        send_order(Roles[4],60,30,0)
        send_order(Roles[5],60,30,0)
        send_order(Roles[6],60,30,0)
    end
    if(conf==2) then
        if(transport_mode==0) then
            simSetJointTargetPosition(BackJoint1,math.rad(-90))
            simSetJointTargetPosition(BackJoint2,math.rad(90))
            send_order(Roles[2],-90,90,0)
            send_order(Roles[3],-75,90,0)
            send_order(Roles[4],-75,90,0)
            send_order(Roles[5],-75,90,0)
            send_order(Roles[6],-75,90,0)
        end
        if(transport_mode==1) then
            simSetJointTargetPosition(BackJoint1,math.rad(0))
            simSetJointTargetPosition(BackJoint2,math.rad(0))
            send_order(Roles[2],0,0,0)
            send_order(Roles[3],30,0,0)
            send_order(Roles[4],30,0,0)
            send_order(Roles[5],30,0,0)
            send_order(Roles[6],30,0,0)
        end
    end
    conf_init_done = 1
    return conf_init_done
end
```

```
 function move_quad(lat_dif,long_dif)
    order_done_counter = 0
    for i = 2,6,1 do
       if(simGetScriptSimulationParameter(Roles[i],"Order_done")==1) then
          order_done_counter = order_done_counter + 1
       end
    end
    if(order_done_counter==5) then
       c4wa = c4wa + 1
       if(c4wa==5) then
          c4wa = 0
--//Decision of movement style-----------------------------------------
          Modornt3 = simGetObjectOrientation(pos_dummy,-1)
          Modornt = Modornt3[3]
          ornt_req3 = simGetObjectOrientation(current_target,-1)
          ornt_req = ornt_req3[3]
          ornt_dif = ornt_dif_corr(ornt_req-Modornt)
          if(lat_dif>0 and long_dif>0) then
             if(math.deg(ornt_dif)<-5) then
                sd = 1
                tr = 0
                s = 30
                r = -15
                l = 15
             end
             if(math.deg(ornt_dif)>5) then
                sd = 0
                tr = 1
                s = 30
                r = 0
                l = -5
                if(ornt_dif>10) then
                   r = 5
                   l = -5
                end
                if(ornt_dif>15) then
                   r = 10
                   l = -10
                end
                if(ornt_dif>20) then
                   r = 15
                   l = -15
                end
             end
          end
          if(lat_dif>0 and long_dif<0) then
             if(math.deg(ornt_dif)<-5) then
                sd = 0
                tr = 1
                s = -30
                r = -5
                l = 0
                if(math.deg(ornt_dif)<-10) then
                   r = -5
                   l = 5
                end
                if(math.deg(ornt_dif)<-15) then
                   r = -10
                   l = 10
                end
```

```
        if(math.deg(ornt_dif)<-20) then
            r = -15
            l = 15
        end
    end
    if(math.deg(ornt_dif)>5) then
        sd = 1
        tr = 0
        s = 30
        r = 15
        l = -15
    end
end
if(lat_dif<0 and long_dif>0) then
    if(math.deg(ornt_dif)<-5) then
        sd = 0
        tr = 1
        s = 30
        r = -5
        l = 0
        if(math.deg(ornt_dif)<-10) then
            r = -5
            l = 5
        end
        if(math.deg(ornt_dif)<-15) then
            r = -10
            l = 10
        end
        if(math.deg(ornt_dif)<-20) then
            r = -15
            l = 15
        end
    end
    if(math.deg(ornt_dif)>5) then
        sd = 1
        tr = 0
        s = -30
        r = 15
        l = -15
    end
end
if(lat_dif<0 and long_dif<0) then
    if(math.deg(ornt_dif)<-5) then
        sd = 1
        tr = 0
        s = -30
        r = -15
        l = 15
    end
    if(math.deg(ornt_dif)>5) then
        sd = 0
        tr = 1
        s = -30
        r = 0
        l = -5
        if(math.deg(ornt_dif)>10) then
            r = 5
            l = -5
        end
        if(math.deg(ornt_dif)>15) then
```

```
                r = 10
                l = -10
            end
            if(math.deg(ornt_dif)>20) then
                r = 15
                l = -15
            end
        end
    end
--Decision of movement style//-----------------------------------------
    if(math.deg(ornt_dif)>-5 and math.deg(ornt_dif)<5) then
        if(math.abs(lat_dif)>math.abs(long_dif)) then
            sd = 1
            tr = 0
            if(lat_dif<0) then
                s = -30
            end
            if(lat_dif>0) then
                s = 30
            end
        end
        if(math.abs(long_dif)>=math.abs(lat_dif)) then
            sd = 0
            tr = 1
            if(long_dif<0) then
                s = -30
            end
            if(long_dif>0) then
                s = 30
            end
        end
    end --if(math.deg(ornt_dif)>-5 and math.deg(ornt_dif)<5)
end --if(c4wa==5)
if(tr==1) then
    c4wa_tr = c4wa
end
if(sd==1) then
    c4wa_sd = c4wa
end
if(c4wa_tr==0 and tr==1) then
    simSetJointTargetPosition(BackJoint1,math.rad(-90))
    simSetJointTargetPosition(BackJoint2,math.rad(90))
    send_order(Roles[2],-90,90,0)
    send_order(Roles[3],60,30,0)
    send_order(Roles[4],60,30,0)
    send_order(Roles[5],60,30,0)
    send_order(Roles[6],60,30,0)
end
if(c4wa_tr==1 and tr==1) then
    send_order(Roles[4],0,90,0)
    send_order(Roles[5],0,90,0)

    send_order(Roles[3],60,30,0)
    send_order(Roles[6],60,30,0)
end
if(c4wa_tr==2 and tr==1) then
    send_order(Roles[4],60,30,-s-l)
    send_order(Roles[5],60,30,s+r)

    send_order(Roles[3],60,30,-s-r)
```

```
            send_order(Roles[6],60,30,s+l)
        end
        if(c4wa_tr==3 and tr==1) then
            send_order(Roles[4],60,30,0)
            send_order(Roles[5],60,30,0)

            send_order(Roles[3],0,90,0)
            send_order(Roles[6],0,90,0)
        end
        if(c4wa_tr==4 and tr==1) then
            send_order(Roles[4],60,30,s+l)
            send_order(Roles[5],60,30,-s-r)

            send_order(Roles[3],60,30,s+r)
            send_order(Roles[6],60,30,-s-l)
        end

        if(c4wa_sd==0 and sd==1) then
            simSetJointTargetPosition(BackJoint1,math.rad(-90))
            simSetJointTargetPosition(BackJoint2,math.rad(90))
            send_order(Roles[2],-90,90,0)
            send_order(Roles[3],60,30,0)
            send_order(Roles[4],60,30,0)
            send_order(Roles[5],60,30,0)
            send_order(Roles[6],60,30,0)
        end
        if(c4wa_sd==1 and sd==1) then
            send_order(Roles[4],0,90,0)
            send_order(Roles[5],0,90,0)

            send_order(Roles[3],60,30,0)
            send_order(Roles[6],60,30,0)

        end
        if(c4wa_sd==2 and sd==1) then
            send_order(Roles[4],60,30-s-r,0)
            send_order(Roles[5],60,30+s+l,0)

            send_order(Roles[3],60,30-s-r,0)
            send_order(Roles[6],60,30+s+l,0)
        end
        if(c4wa_sd==3 and sd==1) then
            send_order(Roles[4],60,30,0)
            send_order(Roles[5],60,30,0)

            send_order(Roles[3],0,90,0)
            send_order(Roles[6],0,90,0)
        end
        if(c4wa_sd==4 and sd==1) then
            send_order(Roles[4],60,30+s+r,0)
            send_order(Roles[5],60,30-s-l,0)

            send_order(Roles[3],60,30+s+r,0)
            send_order(Roles[6],60,30-s-l,0)
        end
    end --if(order_done_counter==5)

    return 1
end
function qd_pass_over(qd_po_counter)
```

```
    order_done_counter = 0
    for i = 2,6,1 do
       if(simGetScriptSimulationParameter(Roles[i],"Order_done")==1) then
          order_done_counter = order_done_counter + 1
       end
    end
    if(order_done_counter==5) then
       c4wp = c4wp + 1
    end
    if(c4wp==4) then
       c4wp = 0
       qd_po_counter = qd_po_counter+1
    end
    if(c4wp==0) then
       send_order(Roles[3],60,30,0)
       send_order(Roles[4],60,30,0)
       send_order(Roles[5],60,30,0)
       send_order(Roles[6],60,30,0)
    end
    if(c4wp==1) then
       send_order(Roles[3],60,30,-135)
       send_order(Roles[4],60,30,135)
       send_order(Roles[5],60,30,-135)
       send_order(Roles[6],60,30,135)
    end
    if(c4wp==2) then
       send_order(Roles[3],60,30,135)
       send_order(Roles[4],60,30,-135)
       send_order(Roles[5],60,30,135)
       send_order(Roles[6],60,30,-135)
    end
    if(c4wp==3) then
       send_order(Roles[3],60,30,0)
       send_order(Roles[4],60,30,0)
       send_order(Roles[5],60,30,0)
       send_order(Roles[6],60,30,0)
    end
    return qd_po_counter
end
function move_whld(lat_dif,long_dif,transport_mode)

    ornt_req = math.atan2(lat_dif,long_dif)
    Modornt3 = simGetObjectOrientation(pos_dummy,-1)
    Modornt = Modornt3[3]
    ornt_dif = ornt_dif_corr(ornt_req-Modornt)
    if(transport_mode==0) then
       if(math.deg(ornt_dif)<-1) then
          simSetJointTargetPosition(BackJoint1,math.rad(-90))
          simSetJointTargetPosition(BackJoint2,math.rad(90))
          send_order(Roles[2],-90,90,0)
          send_order(Roles[3],-75,90,30)
          send_order(Roles[4],-75,90,30)
          send_order(Roles[5],-75,90,30)
          send_order(Roles[6],-75,90,30)
       end
       if(math.deg(ornt_dif)>1) then
          simSetJointTargetPosition(BackJoint1,math.rad(-90))
          simSetJointTargetPosition(BackJoint2,math.rad(90))
          send_order(Roles[2],-90,90,0)
          send_order(Roles[3],-75,90,-30)
```

```
            send_order(Roles[4],-75,90,-30)
            send_order(Roles[5],-75,90,-30)
            send_order(Roles[6],-75,90,-30)
        end
        if(math.deg(ornt_dif)<1 and math.deg(ornt_dif)>-1) then
            simSetJointTargetPosition(BackJoint1,math.rad(-90))
            simSetJointTargetPosition(BackJoint2,math.rad(90))
            send_order(Roles[2],-90,90,0)
            send_order(Roles[3],-75,90,-30)
            send_order(Roles[4],-75,90,30)
            send_order(Roles[5],-75,90,-30)
            send_order(Roles[6],-75,90,30)
        end
    end --if(transport_mode==0)
    if(transport_mode==1) then
        if(math.deg(ornt_dif)<-1) then
            simSetJointTargetPosition(BackJoint1,math.rad(0))
            simSetJointTargetPosition(BackJoint2,math.rad(0))
            send_order(Roles[2],0,0,0)
            send_order(Roles[3],30,0,30)
            send_order(Roles[4],30,0,30)
            send_order(Roles[5],30,0,30)
            send_order(Roles[6],30,0,30)
        end
        if(math.deg(ornt_dif)>1) then
            simSetJointTargetPosition(BackJoint1,math.rad(0))
            simSetJointTargetPosition(BackJoint2,math.rad(0))
            send_order(Roles[2],0,0,0)
            send_order(Roles[3],30,0,-30)
            send_order(Roles[4],30,0,-30)
            send_order(Roles[5],30,0,-30)
            send_order(Roles[6],30,0,-30)
        end
        if(math.deg(ornt_dif)<1 and math.deg(ornt_dif)>-1) then
            simSetJointTargetPosition(BackJoint1,math.rad(0))
            simSetJointTargetPosition(BackJoint2,math.rad(0))
            send_order(Roles[2],0,0,0)
            send_order(Roles[3],30,0,-30)
            send_order(Roles[4],30,0,30)
            send_order(Roles[5],30,0,-30)
            send_order(Roles[6],30,0,30)
        end
    end --if(transport_mode==1)

    return 1
end
function correct_ornt_whld(target,transport_mode)
    ornt_des3 = simGetObjectOrientation(target,-1)
    ornt_des = ornt_des3[3]
    Modornt3 = simGetObjectOrientation(pos_dummy,-1)
    Modornt = Modornt3[3]
    ornt_dif = ornt_dif_corr(ornt_des-Modornt)
    if(transport_mode==0) then
        if(math.deg(ornt_dif)<-1) then
            simSetJointTargetPosition(BackJoint1,math.rad(-90))
            simSetJointTargetPosition(BackJoint2,math.rad(90))
            send_order(Roles[2],-90,90,0)
            send_order(Roles[3],-75,90,30)
            send_order(Roles[4],-75,90,30)
            send_order(Roles[5],-75,90,30)
            send_order(Roles[6],-75,90,30)
            target_ornt_reached = 0
        end
        if(math.deg(ornt_dif)>1) then
            simSetJointTargetPosition(BackJoint1,math.rad(-90))
            simSetJointTargetPosition(BackJoint2,math.rad(90))
            send_order(Roles[2],-90,90,0)
            send_order(Roles[3],-75,90,-30)
            send_order(Roles[4],-75,90,-30)
            send_order(Roles[5],-75,90,-30)
            send_order(Roles[6],-75,90,-30)
            target_ornt_reached = 0
        end
        if(math.deg(ornt_dif)<1 and math.deg(ornt_dif)>-1) then
            simSetJointTargetPosition(BackJoint1,math.rad(-90))
            simSetJointTargetPosition(BackJoint2,math.rad(90))
            send_order(Roles[2],-90,90,0)
            send_order(Roles[3],-75,90,0)
            send_order(Roles[4],-75,90,0)
            send_order(Roles[5],-75,90,0)
            send_order(Roles[6],-75,90,0)
            target_ornt_reached = 1
        end
    end --if(transport_mode==0)
    if(transport_mode==1) then
        if(math.deg(ornt_dif)<-1) then
            simSetJointTargetPosition(BackJoint1,math.rad(0))
            simSetJointTargetPosition(BackJoint2,math.rad(0))
            send_order(Roles[2],0,0,0)
            send_order(Roles[3],30,0,30)
            send_order(Roles[4],30,0,30)
            send_order(Roles[5],30,0,30)
            send_order(Roles[6],30,0,30)
            target_ornt_reached = 0
        end
        if(math.deg(ornt_dif)>1) then
            simSetJointTargetPosition(BackJoint1,math.rad(0))
            simSetJointTargetPosition(BackJoint2,math.rad(0))
            send_order(Roles[2],0,0,0)
            send_order(Roles[3],30,0,-30)
            send_order(Roles[4],30,0,-30)
            send_order(Roles[5],30,0,-30)
            send_order(Roles[6],30,0,-30)
            target_ornt_reached = 0
        end
        if(math.deg(ornt_dif)<1 and math.deg(ornt_dif)>-1) then
            simSetJointTargetPosition(BackJoint1,math.rad(0))
            simSetJointTargetPosition(BackJoint2,math.rad(0))
            send_order(Roles[2],0,0,0)
            send_order(Roles[3],30,0,0)
            send_order(Roles[4],30,0,0)
            send_order(Roles[5],30,0,0)
            send_order(Roles[6],30,0,0)
            target_ornt_reached = 1
        end
    end --if(transport_mode==1)
    return target_ornt_reached
end
function stop(conf,transport_mode)
    if(conf==1) then
```

```
        simSetJointTargetPosition(BackJoint1,math.rad(-90))
        simSetJointTargetPosition(BackJoint2,math.rad(90))
        send_order(Roles[2],-90,90,0)
        send_order(Roles[3],60,30,0)
        send_order(Roles[4],60,30,0)
        send_order(Roles[5],60,30,0)
        send_order(Roles[6],60,30,0)
    end
    if(conf==2) then
        if(transport_mode==0) then
            simSetJointTargetPosition(BackJoint1,math.rad(-90))
            simSetJointTargetPosition(BackJoint2,math.rad(90))
            send_order(Roles[2],-90,90,0)
            send_order(Roles[3],-75,90,0)
            send_order(Roles[4],-75,90,0)
            send_order(Roles[5],-75,90,0)
            send_order(Roles[6],-75,90,0)
        end
        if(transport_mode==1) then
            simSetJointTargetPosition(BackJoint1,math.rad(0))
            simSetJointTargetPosition(BackJoint2,math.rad(0))
            send_order(Roles[2],0,0,0)
            send_order(Roles[3],30,0,0)
            send_order(Roles[4],30,0,0)
            send_order(Roles[5],30,0,0)
            send_order(Roles[6],30,0,0)
        end
    end
    return 1
end
function reassemble(conf,role,reassemble_step)
    if(reassemble_step==0) then
        simSetScriptSimulationParameter(Script,"connected",0)
        simSetScriptSimulationParameter(Script,"order_sent",0)
        simSetScriptSimulationParameter(Script,"order_done",0)
        simSetJointTargetPosition(BackJoint1,math.rad(0))
        simSetJointTargetPosition(BackJoint2,math.rad(0))
        if(math.abs(math.deg(simGetJointPosition(BackJoint1)))<0.1 and
            math.abs(math.deg(simGetJointPosition(BackJoint2)))<0.1) then
            forw_step=0
            back_step=0
            J1pos=0
            J2pos=0
            reassemble_step = 1
        end
    end
    if(reassemble_step==1) then
        if(role~=1 and role~=2) then
            for i=1,5,1 do
                simSetLinkDummy(conn_points[i],-1)
                simSetScriptSimulationParameter(Script,"connected",0)
            end
        end
        reassemble_step = 2
    end
    if(reassemble_step==2) then
        if(role~=1 and role~=2) then
            if(reassemble_move_counter<5)   then
                if(conf==1) then
```

```
                        ms_result,J1pos,J2pos,forw_step,back_step = move_single(-
1,2,forw_step,back_step,0)
                        if(back_step==0 and ms_result==1) then
                            reassemble_move_counter = reassemble_move_counter + 1
                        end
                    end
                    if(conf==2) then
                        ms_result,J1pos,J2pos,forw_step,back_step = move_single(1,2,forw_step,back_step,0)
                        if(forw_step==0 and ms_result==1) then
                            reassemble_move_counter = reassemble_move_counter + 1
                        end
                    end
                end
                if(reassemble_move_counter==3)  then
                    conf=0
                    forw_step=0
                    back_step=0
                    J1pos=0
                    J2pos=0
                end
            end
            if(role==1) then
                conf=0
                conf_init_done = 0
                sq_done = 0
            end
            if(role==2) then
                simSetScriptSimulationParameter(Script,"connected",1)
                simSetJointTargetPosition(FrontJoint,0)
                simSetJointTargetPosition(CamJoint1,0)
                simSetJointTargetPosition(CamJoint2,0)
                target_ornt_reached = 0
                target_pos_reached = 0
                target_state_reached = 0
                assembly_phase_done = 1
                forced_direction=0
                conf=conf_req
                assembly_req=0
            end
        end

        return conf,reassemble_step
    end
end


if (sim_call_type==sim_childscriptcall_actuation) then

    -- Put your main ACTUATION code here
    --Main Loop Starts Here--------------------------------------------------------
    ornt3 = simGetObjectOrientation(pos_dummy,-1)
    ornt = ornt3[3]
    simSetScriptSimulationParameter(Script,"ornt",ornt)
    simSetScriptSimulationParameter(Script,"conf",conf)
    simSetScriptSimulationParameter(Script,"assembly_counter",assembly_counter)
        --//Sensing Phase--------------------------------------------------------------------------------
    if(sensing_done~=1) then
        if(friction_estimation_done~=1) then
            friction_coeff,friction_estimation_done = estimate_friction(friction_estimation_done)
        end
        if(friction_estimation_done==1) then
            if(initial_scan_done~=1) then
                initial_scan_done, CamJ1pos = initial_scan(CamJ1pos)
            end
            if(initial_scan_done==1) then
                sensing_done=1
            end
        end
    end
    --Sensing Phase//-----------------------------------------------------------------------------
    --//Role Distribution Phase-----------------------------------------------------------------------
    if(sensing_done==1 and rd_done~=1) then
        if(Script==Scripts[1]) then
            role_dist()
        end
        rd_done,role = get_role()
    end
    --Role Distribution Phase//-----------------------------------------------------------------------

    if(rd_done==1 and sp_done==0 and role==1) then
        sp_done = strategic_planning()
    end
    --//Strategic Planning Phase--------------------------------------------------------------------
    if(rd_done==1 and sq_done~=1) then
        if(role==1) then
            sq_done,conf_req = sequencing(plan_counter)
        end
        sq_done,conf_req = get_plan(role)
        if(conf_req==conf) then
            assembly_req = 0
        end
        if(conf_req~=conf) then
            assembly_req=1
            assembly_phase_done = 0
            simAuxiliaryConsolePrint(console,"System needs to Assemble/Reassemble\n")
        end
    end
    if(assembly_req==1 and assembly_phase_done~=1) then
        if(conf==1 or conf==2) then
            conf,reassemble_step = reassemble(conf,role,reassemble_step)
        end
        if(conf==0) then
            if(role~=1) then

                target_handle,lat_const,long_const,gamma_const =
get_target_state(conf_req,role,assembly_counter)

                if(target_state_reached~=1) then

                    if(target_pos_reached~=1) then

                        target_found,CamJ1pos = lock_target(target_handle)

                        if(target_found~=1) then
                            target_found,search_counter,CamJ1pos =
search_target(current_target,search_counter,CamJ1pos)
                        end --if(target_found~=1)

                        if(target_found==1) then
                        if(math.abs(math.deg(simGetJointPosition(BackJoint1)))<1 and
math.abs(math.deg(simGetJointPosition(BackJoint2)))<1) then
                            target_found,lat_dif,long_dif =
calculate_difference(target_handle,lat_const,long_const,0)
                            direction,ornt_req =
decide_direction(lat_dif,long_dif,gamma_const,forced_direction)
                            target_pos_reached,velocity = decide_velocity(direction,lat_dif,long_dif)
                        end
            --correct orientation is called with gamma_const=0!
                        J1pos,J2pos,ornt_dif = correct_ornt(J1pos,J2pos,ornt_req)
                        ms_result,J1pos,J2pos,forw_step,back_step =
move_single(direction,velocity,forw_step,back_step,ornt_dif)

                    end --if(target_found==1)

                end --if(target_pos_reached~=1)
                if(target_pos_reached==1) then
                    if(target_ornt_reached~=1) then
                        target_found,CamJ1pos = lock_target(target_handle)
                        J1pos,J2pos,ornt_dif = correct_ornt(J1pos,J2pos,ornt_des)
                        if(math.abs(math.deg(ornt_dif))<1)  then
                            target_ornt_reached = 1
                        end
                    end
                    if(target_ornt_reached==1)  then
                        simSetJointTargetPosition(BackJoint1,0)
                        simSetJointTargetPosition(BackJoint2,0)
                        if(math.abs(math.deg(simGetJointPosition(BackJoint1)))<1 and
                            math.abs(math.deg(simGetJointPosition(BackJoint1)))<1) then

                            step_up,forced_direction=assembly_step_up(conf_req,role,assembly_counter)
                            if(step_up) then
                                assembly_counter = assembly_counter + 1
                                step_up,forced_direction=assembly_step_up(conf_req,role,assembly_counter)

                                target_ornt_reached = 0
                                target_pos_reached = 0
                            end
                            if(assembly_counter==3) then
                                if(connect()) then
                                    simSetScriptSimulationParameter(Script,"connected",1)
                                    simSetJointTargetPosition(FrontJoint,0)
                                    simSetJointTargetPosition(CamJoint1,0)
                                    simSetJointTargetPosition(CamJoint2,0)
                                    target_ornt_reached = 0
                                    target_pos_reached = 0
                                    target_state_reached = 0
                                    assembly_phase_done = 1
                                    forced_direction=0
                                end
                            end
                        end
                    end
                end
            end --if(target_state_reached~=1)
        end --if(role~=1)
        if(role==1) then
            simSetObjectParent(Cylinder,Fix_FS,true)
            connected_counter = 0
            for i=2,6,1 do
```

```
                if(simGetScriptSimulationParameter(Roles[i],"connected")==1) then
                    connected_counter = connected_counter + 1
                end
            end
            if(connected_counter==5) then
                simSetScriptSimulationParameter(Script,"connected",1)
                simAuxiliaryConsolePrint(console,"Assembly/Reassembly Complete\n")
                assembly_phase_done = 1
                conf = conf_req
                simSetScriptSimulationParameter(Script,"conf",conf)
            end
        end --(role==1)
    end --if(conf==0)
end --if(assembly_req==1 and assembly_phase_done~=1)
    --//Configuration Phase--------------------------------------------------
if(assembly_phase_done==1) then
    if(simGetScriptSimulationParameter(Roles[1],"connected")==1) then
        if(role==1) then
            if(simGetObjectParent(Cylinder)~=-1) then
                simSetObjectParent(Cylinder,-1,true)
            end
            if(conf_init_done~=1) then
                conf_init_done = conf_init(conf,transport_mode)
            end
            target_found,CamJ1pos = lock_target(current_target)
            if(conf==1) then
                if(transport_mode==1) then
                    qd_po_counter = qd_pass_over(qd_po_counter)
                    if(qd_po_counter==2) then
                        stop(conf,transport_mode)
                        target_pos_reached = 1
                        CamJ1pos = 0
                        CamJ2pos = 0
                        simSetJointTargetPosition(CamJoint1,math.rad(CamJ1pos))
                        simSetJointTargetPosition(CamJoint2,math.rad(CamJ2pos))
                        if(math.abs(math.deg(simGetJointPosition(CamJoint1)))<=2 and
                            math.abs(math.deg(simGetJointPosition(CamJoint2)))<=2) then
                            target_state_reached = 1
                            qd_po_counter = 0
                        end
                    end
                end
                if(transport_mode==0) then
                    if(target_state_reached~=1) then
                        if(target_pos_reached~=1) then
                            if(target_found==0) then
                                stop(conf,transport_mode)
                                target_found,search_counter,CamJ1pos =
search_target(current_target,search_counter,CamJ1pos)
                            end
                            if(target_found==1) then
                                target_found,CamJ1pos = lock_target(current_target)
                                target_found,lat_dif,long_dif =
calculate_difference(current_target,lat_const,long_const,1)
                                distance = math.sqrt(lat_dif^2 + long_dif^2)
                                if(distance>0.1) then
                                    move_quad(lat_dif,long_dif)
                                end
                                if(distance<=0.1) then
                                    stop(conf,transport_mode)
```

```
                              target_pos_reached=1
                           end
                        end --if(target_found==1)
                     end --if(target_pos_reached~=1)
                     if(target_pos_reached==1) then
                        CamJ1pos = 0
                        CamJ2pos = 0
                        simSetJointTargetPosition(CamJoint1,math.rad(CamJ1pos))
                        simSetJointTargetPosition(CamJoint2,math.rad(CamJ2pos))
                        if(math.abs(math.deg(simGetJointPosition(CamJoint1)))<=2 and
                           math.abs(math.deg(simGetJointPosition(CamJoint2)))<=2) then
                           target_state_reached = 1
                        end
                     end --if(target_pos_reached==1)
                  end --if(target_state_reached~=1)
               end --if(transport_mode==0)
               if(target_state_reached==1) then
                  simAuxiliaryConsolePrint(console,"Reached to Sub-goal State\n")
                  plan_counter = plan_counter + 1
                  sq_done=0
                  for i=1,6,1 do
                     simSetScriptSimulationParameter(Roles[i],"Notification3",1)
                  end
                  if(plan_counter<7) then
                     target_state_reached = 0
                     target_pos_reached = 0
                     target_ornt_reached = 0
                  end
                  if(plan_counter==7) then
                     target_state_reached = 1
                     target_pos_reached = 1
                     target_ornt_reached = 1
                     sq_done=1
                  end
               end --if(target_state_reached==1)
            end --if(conf==1)
            if(conf==2) then
               if(target_state_reached~=1) then
                  if(target_pos_reached~=1) then
                     if(target_found==0) then
                        stop(conf,transport_mode)
                        target_found,search_counter,CamJ1pos =
search_target(current_target,search_counter,CamJ1pos)
                     end
                     if(target_found==1) then
                        target_found,CamJ1pos = lock_target(current_target)
                        target_found,lat_dif,long_dif =
calculate_difference(current_target,lat_const,long_const,2)
                        distance = math.sqrt(lat_dif^2 + long_dif^2)
                        if(distance>0.1) then
                           move_whld(lat_dif,long_dif,transport_mode)
                        end
                        if(distance<=0.1) then
                           stop(conf,transport_mode)
                           target_pos_reached=1
                        end
                     end --if(target_found==1)
                  end --if(target_pos_reached~=1)
                  if(target_pos_reached==1) then
                     if(target_ornt_reached~=1) then
```

```
                    target_ornt_reached = correct_ornt_whld(current_target,transport_mode)
                end
                if(target_ornt_reached==1) then
                    stop(conf,transport_mode)
                    CamJ1pos = 0
                    CamJ2pos = 0
                    simSetJointTargetPosition(CamJoint1,math.rad(CamJ1pos))
                    simSetJointTargetPosition(CamJoint2,math.rad(CamJ2pos))
                    if(math.abs(math.deg(simGetJointPosition(CamJoint1)))<=2 and
                        math.abs(math.deg(simGetJointPosition(CamJoint2)))<=2) then
                        target_state_reached = 1
                    end
                end
            end --if(target_pos_reached==1)
        end --if(target_state_reached~=1)
        if(target_state_reached==1) then
            plan_counter = plan_counter + 1
            sq_done=0
            for i=1,6,1 do
                simSetScriptSimulationParameter(Roles[i],"Notification3",1)
            end
            if(plan_counter<7) then
                target_state_reached = 0
                target_pos_reached = 0
                target_ornt_reached = 0
            end
            if(plan_counter==7) then
                target_state_reached = 1
                target_pos_reached = 1
                target_ornt_reached = 1
                sq_done=1
                stop()
            end
        end --if(target_state_reached==1)
    end --if(conf==2)
end--if(role==1)

if(role~=1) then
    assembly_counter = 1
    conf = simGetScriptSimulationParameter(Roles[1],"conf")
    simSetScriptSimulationParameter(Script,"conf",conf)
    if(simGetScriptSimulationParameter(Script,"Notification3")==1) then
        sq_done=0
        simSetScriptSimulationParameter(Script,"Notification3",0)
    end
    if(conf==1) then
        if(simGetScriptSimulationParameter(Script,"Order_done")==0 and
simGetScriptSimulationParameter(Script,"Order_sent")==1) then

simSetJointTargetPosition(BackJoint1,math.rad(simGetScriptSimulationParameter(Script,"BJ1pos_co
nf")))

simSetJointTargetPosition(BackJoint2,math.rad(simGetScriptSimulationParameter(Script,"BJ2pos_co
nf")))

simSetJointTargetPosition(FrontJoint,math.rad(simGetScriptSimulationParameter(Script,"Cylpos_con
f")))

if(math.abs(math.deg(ornt_dif_corr(math.rad(simGetScriptSimulationParameter(Script,"BJ1pos_conf
"))-simGetJointPosition(BackJoint1))))<5 and
```

```
math.abs(math.deg(ornt_dif_corr(math.rad(simGetScriptSimulationParameter(Script,"BJ2pos_conf"))
-simGetJointPosition(BackJoint2))))<5 and

math.abs(math.deg(ornt_dif_corr(math.rad(simGetScriptSimulationParameter(Script,"Cylpos_conf"))-
simGetJointPosition(FrontJoint))))<5) then

                              simSetScriptSimulationParameter(Script,"Order_done",1)
                              simSetScriptSimulationParameter(Script,"Order_sent",0)
                          end
                      end
                  end
                  if(conf==2) then
                      if(simGetScriptSimulationParameter(Script,"Order_done")==0 and
simGetScriptSimulationParameter(Script,"Order_sent")==1) then


simSetJointTargetPosition(BackJoint1,math.rad(simGetScriptSimulationParameter(Script,"BJ1pos_co
nf")))

simSetJointTargetPosition(BackJoint2,math.rad(simGetScriptSimulationParameter(Script,"BJ2pos_co
nf")))

simSetJointTargetPosition(FrontJoint,math.rad(simGetScriptSimulationParameter(Script,"Cylpos_con
f")))

                          if(math.abs(simGetScriptSimulationParameter(Script,"BJ1pos_conf")-
math.deg(simGetJointPosition(BackJoint1)))<3 and
                              math.abs(simGetScriptSimulationParameter(Script,"BJ2pos_conf")-
math.deg(simGetJointPosition(BackJoint2)))<3) then

                              simSetScriptSimulationParameter(Script,"Order_done",1)
                              simSetScriptSimulationParameter(Script,"Order_sent",0)
                          end
                      end
                      wheel_spd = simGetScriptSimulationParameter(Script,"Cylpos_conf")

simSetJointTargetPosition(FrontJoint,simGetJointPosition(FrontJoint)+math.rad(wheel_spd))
                  end--(conf==2)
              end--if(role~=1)
          end--if(simGetScriptSimulationParameter(Roles[1],"connected")==1)
    end--(if(assembly_phase_done==1)
    --Configuration Phase//--------------------------------------------------

end


if (sim_call_type==sim_childscriptcall_sensing) then

    -- Put your main SENSING code here

end


if (sim_call_type==sim_childscriptcall_cleanup) then

    -- Put some restoration code here
    for i=1,5,1 do
        simSetLinkDummy(conn_points[i],-1)
    end
```

```
end
```

CURRICULUM VITAE

Name Surname:	Mehmet Cem ŞANLI
Place and Date of Birth:	Mersin,1986
E-Mail:	cem_sanli@hotmail.com
B.Sc.:	Systems Engineer (Automation and Control Engineer)