Performance Comparison between Inverse Kinematic Algorithms on 6R Robotic Manipulators

by

Iván Ruiz Carrión

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
MECHANICAL ENGINEERING

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS
2013

Approved by:

_____	_____
Dr. David Dooner, PhD Member, Graduate Committee	Date
_____	_____
Dr. Ricky Valentín, PhD Member, Graduate Committee	Date
_____	_____
Dr. Lourdes Rosario, PhD President, Graduate Committee	Date
_____	_____
Dr. Raul Macchiavelli, PhD Representative of Graduate Studies	Date
_____	_____
Dr. Ricky Valentín, PhD Chairperson of the Department	Date

Abstract

One of the most important aspects of controlling a robotic manipulator is the solution of its Inverse Kinematics problem. Numerical methods, although more computationally challenging than algebraic or geometric methods, are more flexible and are easier to implement once the geometry of the manipulator becomes too complex. Algorithms that exploit these numerical methods are often programmed into a robot to solve the inverse kinematics problems in real-time. In this study, the performances of two different algorithms that solve the inverse kinematic problem for a 6R manipulator are compared; the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms were programmed using MatLab7.5. The time needed to solve the problem and the precision of the result were measured. Both algorithms were executed on an *AMD Turion 64 X2/ 2.2GHz* and an *Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz* processor environment in order to compare the effect of processor architecture on the algorithm's performance. It was determined that the MDFP was faster to reach a result than the NR, and that the NR provided greater result precision than the MDFP. This behavior was observed in both processor architectures.

Resumen

Uno de los aspectos más importantes del control de un manipulador robótico es la solución del problema de la cinemática inversa. Métodos numéricos, aunque son computacionalmente más complicados que los métodos algebraicos, son más flexibles y fáciles de implementar una vez la geometría del robot se torna complicada. Algoritmos que aprovechan las capacidades de estos métodos numéricos se implementan en robots para que así puedan resolver la cinemática inversa del mismo en tiempo real. En este estudio se compara el desempeño de dos algoritmos de cinemática inversa para manipuladores de seis grados de libertad (6R); el algoritmo Modified Davidon Fletcher Powell (MDFP) y el Newton Raphson (NR) fueron programados usando MatLab7.5. El tiempo que les tomó a los algoritmos en resolver el problema y la precisión de este resultado fueron medidos. Ambos algoritmos fueron ejecutados en un procesador AMD Turion 64 X2/ 2.2GHz y en un Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz para así determinar si la arquitectura del procesador afectaba el desempeño de el algoritmo. Se determinó que el algoritmo MDFP es más rápido en encontrar el resultado, sin embargo el NR proveía una precisión de resultado más alta. Este comportamiento se observó en ambas arquitecturas de procesador.

To my Wife Yarilyn, your wise words gave meaning to my writing

To my family, thanks for giving me the motivation to continue moving forward.

To Professor Musa Jouaneh, even though I was not your student, you received me with open doors and sent me in the right direction.

To my graduate advisor; Professor Lourdes Rosario, thanks for sharing your wisdom with me and leading me through this difficult passage.

Contents

List of Tables

List of Figures

1 Introduction

1.0.0 Motivation

The study of inverse kinematic algorithms for robots is of great importance in the fields of engineering and the industry. Robotic manipulators are often used for manufacturing processes and operations that have high precision requirements, or are extremely dangerous or strenuous for humans. Examples of these applications are bomb disposal robots (e.g., Howard[1]) or robots used for unassisted surgery (e.g, Gerhardus [2]). Although for humans it is relatively simple to move and orient the hands in different ways; computationally, this process presents a challenge using computers once the time between input and response is required to be minimal. This is because in order to move an end effector into a desired position there can be more than one way for the robot to locate its links. To maximize the efficiency of these manipulators, algorithms have been developed with the capacity to determine the smallest or most effective link movement required for accurate end effector positioning. As technology and applications become more complex, faster yet computationally light algorithms are required. A response delay of less than a second can mean the difference between the life and death of a patient, the dangers of a bomb not being diffused in time, or the economic losses due to delays in manufacturing lines.

When researching for this project in the databases, no publications regarding a performance comparison between inverse kinematic algorithms were found. As a result, the need to compare the performance of different inverse kinematic algorithms is evident.

This project intends to accomplish two key goals. The first goal is to program and execute two different inverse kinematic algorithms that employ distinct approaches for solving the inverse kinematic problem of a robot and measure the time used by the algorithm to solve the problem and the precision of the result. The computer architecture in which the algorithm is implemented must play a role in its performance, now that the architecture of a processor determines how the information is stored in memory and how it is processed. As a result, the second goal of this project is to make this comparison using different computer processors in order to address the influence of the computer architecture on the performance of the algorithm. The information generated by accomplishing these two goals was then used to make an in depth experimental comparison of the two algorithms.

The algorithms that were evaluated in this study were selected based on the method being employed to solve the inverse kinematic equation. The algorithm presented in **A Control Algorithm of General 6R Mechanic Arm Based on Inverse Kinematics** (Qu et al. [3]), utilizes the Newton-Rahpson method, which is one of the most widely used methods for solving the inverse kinematic problem. The algorithm presented in **Inverse Kinematic Solution for Robot Manipulator Based on Electromagnetism-like and Modified DFP Algorithms** (Feng et al. [4]), uses the **Modified David Fletcher Powell (MDFP)** method to find the solution for the inverse kinematics of the robot. The **MDFP**, which is a modified version of the **DFP** Quasi-Newton method, employs a different approach in reaching the final solution of the problem, which makes it an ideal candidate for a comparison.

Due to safety of proprietary information, the real-time performance of robots is not revealed by the manufacturers (e.g. Lewis et al. [5]). This makes the process of comparing the performance of newly developed inverse kinematic algorithms difficult since there is no easy way to contrast them. Therefore, this project aims to inspire future work in this field by encouraging additional algorithms to be programmed and contrasted in a similar way, this will increase the amount of information gathered in a database. This database will contain information about inverse kinematic algorithm techniques and the time and iterations it takes for these algorithms to solve the problem in a given architecture. Having such record will be of great value to the scientific community since it will grant access to performance information comparison between algorithms. Using this tool, a programmer should be able to easily determine if a new algorithm excels above others; therefore, faster and more efficient algorithms could be developed. Future ramifications of this study may incorporate the knowledge from individuals in the field of electrical and computer engineering to increase the detail in this database. The final result would be a valuable algorithm testing tool for the field of robotics.

1.1.0 Summary of the Following Chapters

Chapter 2 delivers the necessary background theory necessary for this investigation. Chapter 3 talks about the Methodology used in the investigation. Chapter 4 provides the results acquired during the investigation and Chapter 5 is the discussion of these results. Finally, Chapter 6 discusses the conclusions that were reached after the analysis was completed.

2 Background Theory

2.0.0 Introduction to Robots

Robots are an increasing influence in our daily lives. These machines have evolved in such a way that are now being applied in key tasks, such as military support, space exploration, emergency rescue operations, and many others (e.g Hodge, Guzzio, Powers [6]–[8]). It is essential to understand the basic characteristics of robots in order to model their behavior. Niku describes in some detail various important aspects of robots such as their classifications, components, degrees of freedom, types of joints and coordinate systems, programming modes, reference frames, characteristics, and languages used to program them (e.g., Niku et al. [9]).

2.1.0 Classifications

These are based depending on the function and configuration of the joints. There are different types of automated machines working in the industry today, many of which have or lack certain characteristics that make it difficult classify them as robots or not. Many associations, such as the Japanese Industrial Robot Association (JIRA), have developed a ranking system that classifies the different characteristics that a machine should have in order to be considered a robot.

Table 1
Classifications of Robots According to the Japanese Industrial Robot Association (JIRA) (e.g., Niku et al. [9]**)**

Classification	Description
Class 1	Manual Handling Device: a device with multiple degrees of freedom, actuated by an operator.
Class 2	Fixed Sequence Robot: a device that performs the successive stages of a task according to a predetermined, unchanging method, which is hard to modify.
Class 3	Variable Sequence Robot: same as class 2 but easy to Modify
Class 4	Playback Robot: a human operator performs the task manually by leading the robot, which records the motions for later playback: the robot repeats the same motions according to the recorded information.
Class 5	Numerical Control Robot: the operator supplies the robot with a movement program rather than teaching it the task manually.
Class 6	Intelligent Robot: a robot with the means to understand its environment and the ability to successfully complete a task despite changes in the surrounding conditions under which it is to be performed.

In addition to the JIRA's classification there are other entities that have taken the task of analyzing where is the line that distinguishes a robot from an automatic machine. Other classifications of robots have been developed as well. The Robotics Institute of America (RIA), for instance, only considers classes three through six of the JIRA classification as robots and the association Francaise de Robotique (AFR) has a different classification as listed in Table 2.

Table 2 Classifications of robots according to The Association Francaise de Robotique (AFR) (e.g., Niku et al. [9]**)**

Classification	Description
Type A	Handling devices with manual control to telerobotics.
Type B	Automatic handling devices with predetermined cycles.
Type C	Programmable, servo controlled robots with continuous or point-to-point trajectories.
Type D	Same as C but with capability to acquire information from its environment.

2.2.0 Robot Components

The robot component or systems are: the manipulator or rover, the end effector, actuators, sensors, controller, processor, and software.

The manipulator or rover is often considered the main body of the robot; this component consists of the links, joints, and other parts of the robot's structure elements. However, taking into consideration the manipulator itself is not enough to consider the system to be a robot. The end effector is the part of the robot that is connected to the last joint in the manipulator. Most of the time it is employed to handle objects, paint or weld while other times it can be used to connect to other machines. Usually, robots are shipped from the manufacturer with a simple gripper; it is up to the costumer to design an appropriate end effector for the task at hand. It is the responsibility of company engineers or external consultants to design the end effector best suited for their needs. Some examples of different end effectors are: a welding torch that can be employed if the robot is needed to weld metal parts together, a paint spray gun that can be used to precisely apply an even coating of paint to the finished product, or glue laying devices and part handlers. In many cases the action of the end effector is controlled either by its controller or an external device such as a programmable logic controller (PLC). Actuators can be considered as the driving force or "the muscles" of the manipulators. Controllers command the actuators by electric signals causing them to move the robot joints and links. There are many types of actuators ranging from pneumatic to electrical motors. Some common types of actuators are: servomotors, stepper motors, pneumatic actuators, and hydraulic actuators. Other more specialized actuators can be employed in a robot depending on the required task; however they are always controlled by a controller.

Sensors work similarly to the different senses. They help the robot acquire information about the status of its internal structure and the workspace surrounding them. The location of each link at any given time is necessary for the controller to know the robot's configuration, this is why sensors are used to acquire and send information about different joints to the controller. Robots employ sensor information to determine their internal configuration similar to our central nervous system. They also employ external sensors such as specialized cameras, touch and tactile sensors to receive information about the outside world. Visual, force and stress feedback are some examples of the many types of information sensors can provide. External information is often used by robots to avoid obstacles or collisions with objects or other machines in the workplace as well as to regulate forces used to handle delicate components or parts. In addition, speech synthesizers and other devices are included in some robots to help them communicate and interact better with the outside world.

The controller uses the processor's information and current coordinates to send the appropriate commands to the actuator to reach the specified robot coordinates. For example, if the robot is required to pick up an object but that needs to be handled at a certain angle, the controller will verify whether or not the manipulator is located at the right orientation using sensory feedback, it will issue appropriate commands command to move the actuators of the joints until the sensor feedback reaches a desired value. In more sophisticated systems the velocity and the force exerted by the robot is also controlled.

The processor is often referred as "the brain" of the robot. The movement of a robot joint, its speed, and its direction are calculated by this component. It also oversees the synchronized

actions between the controller and the sensors. This component is generally a computer that is specialized and dedicated to this purpose. Just like a regular computer, it requires an operating system, programs and exterior equipment. Depending on the system, the controller and the processor can be both combined into one unit or be used as two separate units. Usually the controller is provided by the manufacturer but the processor is not, it is up to the customer to integrate a processor into the robot. The software includes three strains of programs. One is the operating system that runs the processor, next there is the robotic software which calculates the motions of the joints based on the kinematics equations of the robot. These results are then sent to the controller. It is important to note that this software may exist at many different levels, ranging from machine language to sophisticated robot languages. Finally, the third kind is the group of application-oriented routines and programs created to control the robot's components for the specific task they should be performing.

2.3.0 Robot Degrees of Freedom

To fully describe the position of a point in 3D space three coordinates are necessary. These can be based on the Cartesian coordinate system or another suitable one as long as three coordinates are provided, two coordinates will provide too little information while four coordinates will create an over-constrained system.

By analogy, a robotic arm requires three degrees of freedom to position an object in 3D space, but it also needs three additional degrees of freedom in order to orient it; this means that a robot requires a total of six degrees of freedom in order to position and orient an object in 3D space (Spong et al. [10]). A degree of freedom in a robot can be defined as its ability to move or

rotate along a particular axis. If a robotic arm is able to position an object anywhere along the x axis then it has one degree of freedom, if the robot is also able to locate the object anywhere along the y axis then it will have two degrees of freedom and if the z axis can be reached as well, then the robot has the three degrees of freedom, allowing it to position an object anywhere in a 3D space (within its workplace). As an example, consider a gantry crane (using x,y,z coordinates) that can place an object at any location specified by an operator as long as it is within its 3D workspace. It is imperative to understand that although these three degrees of freedom are sufficient for the robot to position an object in a 3D space, they are not enough to fully describe the object in space since there are infinite ways it can orient the body once it reaches a desired location. Because of this, a rotation about each axis is required to be instructed to the robotic manipulator in order for the robot to fully position and orient an object in a 3D space. As a result, six degrees of freedom are required for a robotic manipulator to achieve such a task.

A robot having less than six degrees of freedom will not be able to position an object in any arbitrary location and orientation, it will only be able to do so within the range its degrees of freedom allow. For instance, consider a three degrees of freedom robot, which can only move in the x-y-z directions, the robot will be able to pick an object and place it somewhere along its axis, however, it will not be able to orient it in any direction, the original orientation will be maintained in any location is placed. A robot with five degrees of freedom can also be considered, it can move in the x-y direction and can rotate an object about the three axis; this robot will be able to rotate an object to any desired orientation however it will only be able to

place it somewhere within the x-y plane since there is no z axis degree of freedom. This situation also happens in any other robot configuration.

It is also important to consider the implications of a robot that has seven degrees of freedom instead of six or less. A system with seven degrees of freedom will not have a unique solution to the kinematic equation to move the object. This means that the robot will have infinite ways to reach a desired position and orientation for an object in space. This implies that the controller will have to decide which path is going to be used to reach the final location. One possibility is to use an optimization routine that finds the shortest or fastest path to reach this location. Then, a computer considers all solutions to find the shortest response to perform it. This additional requirement makes seven degrees of freedom robots not suitable for the industry. There are certain exceptions to this, for instance, a robot with seven degrees of freedom can still be used without the need to solve lengthy mathematical processes if one of its degrees of freedom is known. Consider a robot with six degrees of freedom mounted in a conveyor belt or a moving platform. Since the robot position along the workspace changes with its moving base, this degree of freedom is added, thus creating a seven degree of freedom system. However, since the location of the base at any given time is known, then the solution for this conditional degree of freedom is trivial.

There are cases where a joint may have the ability to move, but its movement is not fully controlled. For example, consider a linear joint actuated by a pneumatic cylinder, the arm can only be fully extended or fully retracted, but no controlled position can be achieved between the two extremes. In this case, the convention is to assign only a half degree of freedom to the

joint. This means that the joint can only be at specified locations within its limits of movement. Another possibility for a half degree of freedom is to assign only particular values to the joint. For example, suppose a joint is made to be only at 0, 30, 60 and 90 degrees. Then, as before, the joint is limited to only a few possibilities, and therefore, has a partial degree of freedom.

Many industrial robots possess fewer than six degrees of freedom. Robots with three and a half, four and five degrees of freedom are in fact very common. So long as there is no need for the additional degrees of freedom, these robots perform very well. For example, suppose you insert electronic components into a circuit board, the circuit board is always laid flat on a known work surface, and consequently, its height (z value) relative to the base does not need to change. Therefore, there is only a need for two degrees of freedom along the x- and y- axes to specify any location on the board for insertion. Additionally, suppose that the components are to be inserted in any direction on the board, but the board is always flat. In that case there is a need for one degree of freedom to rotate about the vertical axis (z) in order to orientate the component above the surface. Since there is also need for a half degree of freedom to fully extend the end effector to insert the part or to fully retract it to lift the robot before moving, only three and a half degrees of freedom are needed: two to move over the board, one to rotate the component, and half to insert or retract. Insertion robots are very common and are extensively used in the electronic industry. Their advantage is that they are simple to program, less expensive, smaller, and faster. Their disadvantage is that although they may be programmed to insert components on any size board in any direction, they cannot perform other jobs. They are limited to what three and a half degrees of freedom can achieve, but they can perform a variety of functions within this design limit (Spong et al. [10]).

2.4.0 Robot Joints

It does not matter how many degrees of freedom a robot has, it always requires joints that are capable of achieving such intrinsic movements. These joints can be classified as linear, rotary, sliding, or spherical. Spherical joints are widely used; however, they possess multiple degrees of freedom, which make them difficult to control. As a result they are mostly used in research robots. The most common robot joints are the linear (prismatic) joint or the rotary (revolute) joint. Prismatic joints provide linear movements. They are usually composed of hydraulic or pneumatic cylinders or linear actuators. These joints are used in gantry cylindrical, or spherical robot types. Revolute joints rotate, many of these joints are hydraulic or pneumatic but most revolute joints are electrically driven with stepper motors or servo motors. The configurations of the robots usually follow the coordinate frames that define them. For instance prismatic joints are defined by **P**, while revolute joints are denoted by **R**, spherical joints are defined by **S**. If the robot is composed by different types of joints, its configuration is denoted by a succession of **P**, **R** and **S**. For example, a robot with three prismatic joints and three spherical joints can be defined as a **3P3S** robot. Table 3 specifies various common configurations for positioning the hand of the robot and Figure 1 shows various robot coordinate frames.

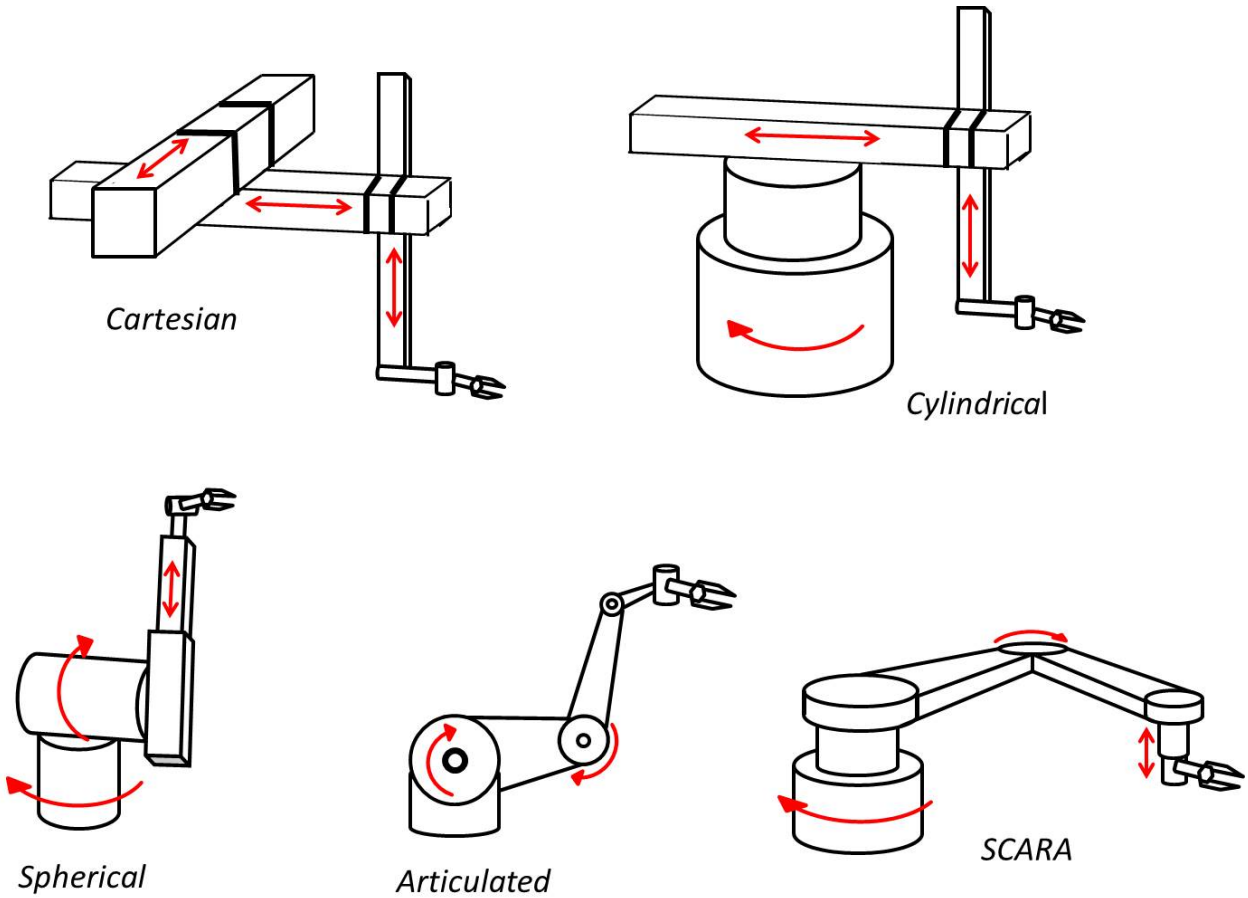


Figure 1. Several robot coordinate frames.

Table 3 Common Configurations for robot hand positioning(e.g., Niku et al. [9]**)**

Configuration	Description
Cartesian/Rectangular/Gantry (3P)	These robots are made of three linear joints that position the end effector, which are usually followed by additional revolute joint that orientate the end effector.
Cylindrical (PRP)	Cylindrical coordinate robots have two prismatic joints and one revolute joint for positioning the part, plus revolute joints for orientating the part.
Spherical (P2R)	Spherical coordinate robots follow a spherical coordinate system which has one prismatic and two revolute joints for positioning the part, plus additional revolute joints for orientation.
Articulated/anthropomorphic (3R)	An articulated robot's joints are all revolute, similar to a human's arm. They are the most common configuration for industrial robots.
Selective Compliance Assembly Robot Arm (SCARA)	SCARA robots have two (or three) revolute joints that are parallel and allow the robot to move in a horizontal plane, plus an additional prismatic joint that moves vertically. SCARA robots are very common in assembly operations. Their specific characteristic is that they are more compliant in the x-y plane but are very stiff along the z-axis, therefore providing selective compliance.

2.5.0 Robot Reference Frames

Robots move relative to various 3D coordinate frames. Robot movements are unique for each one and they are usually addressed as world, joint and tool reference frames. The world reference frame is a universal coordinate frame, defined with the x-, y- and z- axes. In this case, the joints of the robot move at the same time in a coordinated way causing a movement across the 3D space. This type of reference frame is normally used to define the motion of the robot relative to other objects parts or machines with which the robot communicates. A joint reference frame is used to define the movements of each of the robot's joints. Due to this, each joint is controlled individually causing them to move one at a time. The motion of the hand will change depending on the type of joint used in the robot at the time, for instance if a revolute joint is used then the hand will move in the circle defined by the axis. The tool reference frame specifies movements of the robot's hand in accordance to a frame attached to the hand, as a result, its motions are relative to a local *x, y, z*- frame attached to the robot's hand. In contrast to a universal world frame, the local tool frame moves along with the robot. This requires all of the robot's joints to move simultaneously in order to create coordinated motions about the tool frame. This frame approach is normally used if the robot hand is required to move away or get closer to an object or if it is required to insert objects (e.g., Niku et al. [9]).

2.6.0 Programming Modes

Robots may be programmed in a number of different modes, depending on the robot and how sophisticated it is. Examples of commonly used programming modes in the industry are the joint mode, the lead to teach mode, the continuous walk-through and the software mode.

The joint mode consists on setting up switches and hard stops that control the path of the robot. This programming mode is often used along with programmable logic controllers (PLC). The lead through or teach mode, is employed by moving the robot joints with a teach pendant. Once the location of the robot is reached, it is saved “taught” into the controller. Once the robot is fully programmed, it will move to those locations and orientations. Usually, this is executed from point-to-point which means that the motion between the points is not programmed and only the final destination of the end effector can be guaranteed.

The continuous walk-through mode is similar to the lead through mode. All the robot joints are moved at the same time while the robot is continuously sampling and recording the motion. As a result the complete motion between two points is accurately played back by the robot. These motions can be taught by physically moving the end effector, or by “wearing” the robot arm and moving it across the workplace. Painting or welding robots can be programmed by expert painters or welders using this method.

The software mode requires the operator to program the robot using a written online or offline set of commands that instruct the controller on what motions must be executed. This is the most sophisticated approach to program a robot, and as a result is the most versatile. It can include sensory information, conditional statements (such as while or if) and branching. It is required for the operator to possess a working knowledge of the command syntax in order to program the robot using this mode.

2.7.0 Robot Characteristics

It is necessary to understand common features that define robots. The most common characteristics of robots are; payload, which is the weight the robot can carry and maintain its integrity to its other specifications. For instance, robots usually have the capability of lifting objects much heavier than specified, however by doing so their accuracy in placement and motion is compromised. The robot payload is usually very small when compared to the robot's actual weight. The robot's reach can be defined as the maximum distance the robot can reach within its workplace. A dexterous point within the robot's reach is one that the robot can reach with any desired orientation, and a non-dexterous point is that on which the robot cannot accomplish this, these points are usually located within the limits of the robot reach. Reach depends on the robot's joints, lengths and configurations. This is a very important specification of the robot that must be considered before installing a robot in the workplace. A robot's precision can be defined as how accurately the robot can reach a desired point. This depends greatly on the on the resolution of the actuators, on sensorial feedback on how many positions and orientations were used to test the robot with that load and at what speed. These are important concepts that have to be considered when the robot precision is crucial to the task. Usually, industrial robots have a precision in the range of 0.0254 millimeters or better. Repeatability can be regarded as the capability of the robot to reach a desired point in multiple attempts. Consider a robot that is required to move to a certain point 100 times, it may not reach the same point every time but it will remain within a certain radius from the point. The radius of the circle created by the repeated motions of the robot is called repeatability. Repeatability is usually specified for a certain number of runs, which is why it is important for a

manufacturer to specify the number of runs, the payload and the orientation used to acquire certain repeatability.

2.8.0 Robot Languages

In most cases the robot manufacturer designs a robot language along with the robot itself. Therefore in order to be able to work with a particular robot, its programming language must be learned. Other robots base their programming language on some common language such as COBOL, Basic, C, and FORTRAN. Depending on the design and application, the robot language's sophistication varies, ranging from machine level to artificial intelligence. Depending on the level of the languages they are interpreter based or compiler based.

Interpret-based languages execute one line of the program at a time, converting the command line into machine language so that it can be understood by the controller. Each line is executed in sequence until the last line of code has been performed or an error is detected. This type of language proves useful for debugging since the commands continue to be performed until the error is detected. On the other hand, since every line has to be interpreted every time, execution is slower and not very efficient.

Compiler-based programs use a compiler to convert the program into machine language before it is executed. Since this process is done only once, this type of programming is faster and more efficient. However since it cannot be tested until it has been compiled, debugging is more difficult. Depending on the level of complexity of the task presented to the robot, the level of programming should vary. Different levels of robotic languages used today include; the Micro-computer or Machine language level consists on writing the program directly into machine

language. This level of programming is quite basic and is very efficient but is difficult to understand and to follow. Eventually, all programs have to be compiled to this machine level in order to be understood by the controller, however, more complex programs are usually written in a higher level format that is easier to understand and to follow. The point-to-point level writes the coordinates of the motion sequentially and the robot follows the points. This is a very primitive and simple type of program that is easy to use but not very powerful since it lacks sensorial feedback, branching and conditional statements. The primitive motion level can develop more sophisticated programs including sensory information branching and conditional statements. Finally, the structured programming level which is harder to learn but is very powerful and allows for sophisticated programming.

2.9.0 Programming Languages

As embedded processors and applications become more and more complex a need to replace classic assembly-level software programming with high level language compilers is on the rise. Rainer [11] describes several programming languages that have gained importance in the design of embedded systems:

The C language is more than 20 years old and yet is still used for software development, this is because this language has been tested, used and debugged over the years, it is easy to learn and is available for many computer platforms and the existence of “legacy” source code. One of the disadvantages of using C is that it allows for a very low level, assembly-like, and machine dependent method of programming. However, for embedded systems this feature is very convenient, features including direct access to physical memory via pointers, post increment of

variables and “dirty” type casts are widely used for writing system software in a high level language. Finally, C allows easy inclusion of machine specific functions due to its separately compilable modules features. C++ on the other hand is an object-oriented variant of the C language developed to combine advantages of object oriented software with the flexibility offered by C. Since C is being included as a subset of C++, this language is replacing the original language in a systematic way. C++ encourages high level programming, based on class libraries for primitive data structures. On the other hand C++ enables the reuse of existing code. C++ on embedded systems also possesses disadvantages, features such as throw-catch mechanism for exception handling and templates for typeless function parameters which can be convenient for a programmer might compile a code with a large overhead in size and performance. As a result, the Embedded C++ Technical Committee has agreed on a simple version of the C++ called EC++ which is ideal for embedded development. Although tools for development that support EC++ are available, the language itself is not as widespread yet.

Another object oriented language that exists today is Java. This language was developed for platform independent applications that could run via Internet. As a result, this language is fully machine-independent and it does not support hardware oriented elements like pointers. Since it is usually compiled into a byte code representation Java allows very compact codes. For embedded systems, Java is a strong candidate since it provides features like garbage collection and multithreading. However, these features in some cases require special runtime systems and also byte code interpreters must be present on-chip so that the code can be executed. In order for Java to be executed in embedded systems more efficiently, customized versions of the language such as Embedded Java are being specified.

The data flow language DFL was developed for the specific needs of DSP systems. It resembles C in syntax but there are language constructs that are useful when describing DSP algorithms. Among these there are specifications of saturation and rounding behavior for arithmetic operations, concept of delayed systems (requirement for digital filters), and ability to capture signal flow graphs via text which is a common representation of DSP algorithms. Even though this language allows for a convenient programming mode in the DSP area, it has not seen a widespread success on the market since it is difficult to establish a new language standard for narrow application domains.

Another powerful programming language is MatLab, described by Franklyn et al. [12] as a technical computing environment created by Math-Works, Inc. It is a programming language which basic data element is array, which provides a powerful platform for matrix operation, graphic representation of functions and data, as well as algorithm implementation, numerical analysis and many other functions. This language has become a useful tool that allows solving complex computational problems without the need of additional code to perform routine processes such as matrix operations.

2.10.0 Robot Kinematics

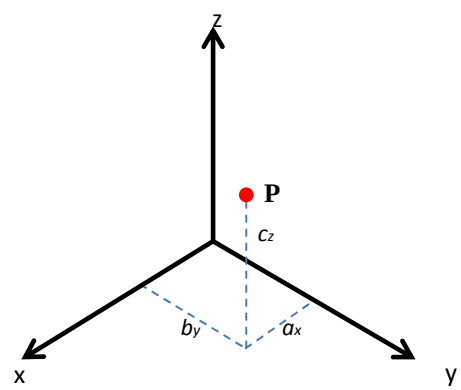
Consider the simple task of grabbing a glass of water. The exact position of the glass is known, in order to grab it the hand must be located within its vicinity, this requires the joints in the arm to rotate and translate in so that the hand can grab the glass. The human brain can accomplish this task with seamlessly little effort. However, robots may have a more difficult situation now that in order to accomplish such a task, the kinematics of the robot arm must be known and

solved. This process requires intensive calculations and can be divided into two types. The forward kinematic problem consists of knowing variables of the joints in order to determine the final position of the end effector. The inverse kinematic problem involves finding the variables of the robot's joints necessary to locate the end effector in a desired location. This latter requires a rather more difficult mathematical approach and will be discussed within this section. Niku et al. [9] provides an introduction to some of the most important aspects of robot kinematics such as the vector and matrix formulation used to control a robot and also the algebra used to solve the matrices.

2.10.1 Representation of points and vectors in space

As shown in the figure below, it is possible to represent a point by its three coordinates along a reference frame with the following equation.

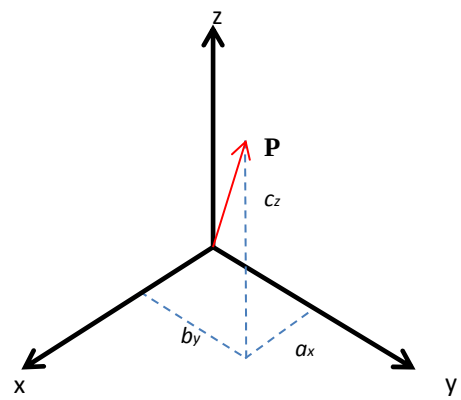
$$P = a_x \boldsymbol{i} + b_y \boldsymbol{j} + c_z \boldsymbol{k} \tag{1}$$



Where a_x, b_y and c_z are the coordinates of the point relative to the reference frame, it is possible to use other coordinate systems to represent a point.

A vector can be represented using the coordinates of its head and tail, if the vector starts at point Q and ends at point R then it can be represented by the following equation.

$$P = (Q_x - \mathrm{R_x})\boldsymbol{i} + (Q_y - R_y)\boldsymbol{j} + (Q_z - R_z)\boldsymbol{k} \tag{2}$$



If the vector starts at the origin then it can be represented by:

$$P = a_x\boldsymbol{i} + b_y\boldsymbol{j} + c_z\boldsymbol{k} \tag{3}$$

or

$$P = \begin{bmatrix} a_x \\ b_y \\ c_z \end{bmatrix} \tag{4}$$

Where a_x, b_y and c_z are the three components of the vector in the reference frame. This representation can include a scale factor.

$$P = \begin{bmatrix} P_x \\ P_y \\ P_z \\ w \end{bmatrix} \tag{5}$$

Here P_x, P_y and P_z are numbers that when divided by *w* yield a_x, b_y, and c_z (e.g.; $P_x = a_x$ w). This scaling factor is extremely important since it can help change the magnitude of the vector without having to change direction of the vector. By *w* being larger than 1 the vector enlarges

and by being smaller than 1 the vector contracts, but the actual values of P_x, P_y and P_z need not be changed. There are two special cases to this. When *w*=1 the vector remains unchanged but when *w*=0 , although the length of the vector grows to infinity, its direction still remains unchanged thus they can be used as direction vectors.

2.10.2 Representation of frames in space

A frame is usually represented by three orthogonal axes such as x , y and z, normally robots work using more than one reference frame at any given time. The universal reference frame $F_{x,y,x}$ and a set of axes {n, o, a} that represent the frame that moves relative to the universal frame can be seen in Figure 4.

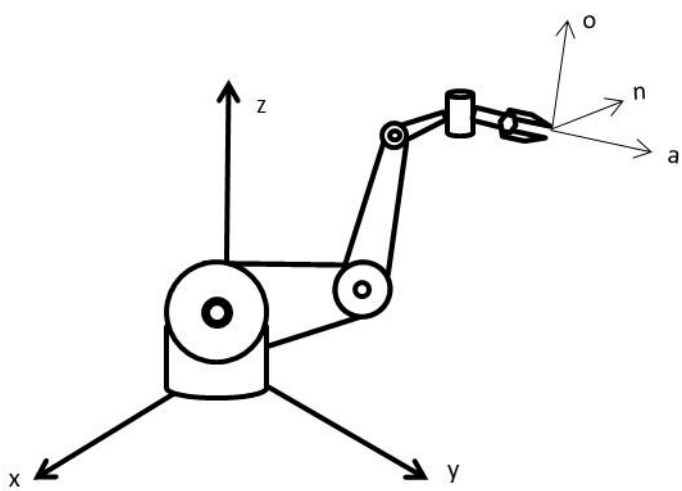


Figure 4. Robot frame relative to a universal reference frame.

Each axis direction in frame $F_{n,o,a}$ located at the origin of the universal reference frame $F_{x,y,z}$ can be written as the three directional cosines relative to the reference frame. As a result, this can be represented in matrix form as;

$$F = \begin{bmatrix} n_x & o_x & a_x \\ n_y & o_y & a_y \\ n_z & o_z & a_z \end{bmatrix} \tag{6}$$

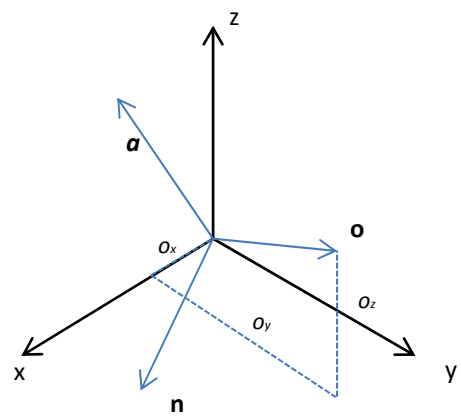


Figure 5. Representation of a moving frame at the origin of the universal reference frame.

To fully describe a frame relative to the universal reference frame both the location of its origin and the directions of its axis must be specified. If the frame is not located at the origin then its location can be described by a vector between the moving frame and the universal reference frame as shown in Figure 6. Similarly to the previous case, the frame can be represented in matrix form by (7):

Rotation Matrix

$$F = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7}$$

Translation Matrix

As it can be seen in equation (7) the first three vectors are used as directional vectors thus *w*=0, however, for the vector representing the origin of the frame relative to the universal reference frame *w*=1 since the length of this vector is important. Adding this fourth row of scaling factors

makes the matrix a (4x4) although it can be represented by a (3x4), using this extra row makes it a homogeneous matrix which allows finding its inverse and in addition the multiplication of two homogeneous matrices becomes more practical.

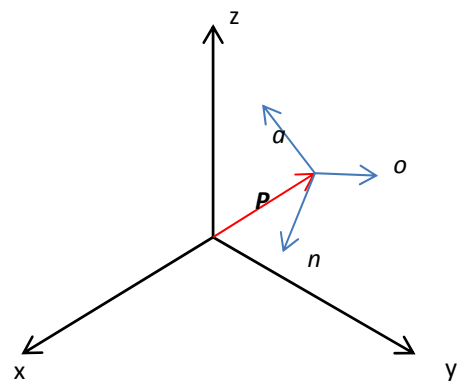


Figure 6. Frame represented in a frame.

2.10.3 Representing a rigid body in space

If it is desired to represent an object in space, it can be done by simply attaching a frame to the object and describing the frame instead. Since the frame is permanently attached to the object, the location of the object relative to that frame is always known, see Figure 7. This frame can be represented as before, by a matrix where the origin of the frame and the three vectors representing its orientation relative to the universal reference frame are shown. See equation (8).

$$F_{object} = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8}$$

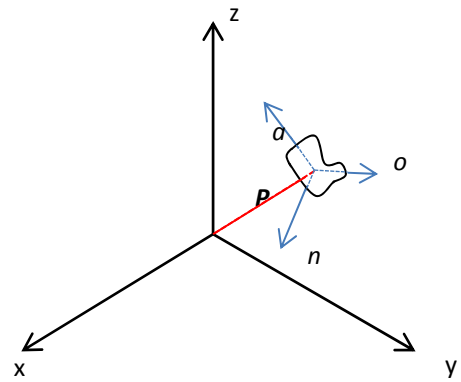


Figure 7. Representation of an object in space.

As mentioned earlier, an object in space can move along three axes and rotate among them, thus it has six degrees of freedom. As a result, all that is needed to fully define the object in space are six variables, but equation (8) provides twelve variables, nine describing orientation and three describing the position of the frame, the scaling factors are not considered since they must be known. It is clear that in order to reduce this representation to six variables several constrains must be applied. As long as the system is not indeterminate, a total of six constrains are required in order to achieve this.

These variables can be determined by knowing the fact that the unit vectors n, o, a are perpendicular and that each unit vector's length that is represented by its directional cosines must be equal to 1. This can be written as follows:

$$\begin{aligned} \boldsymbol{n} * \boldsymbol{o} &= 0 \\ \boldsymbol{n} * \boldsymbol{a} &= 0 \\ \boldsymbol{a} * \boldsymbol{o} &= 0 \end{aligned}$$

$$\begin{aligned} |\boldsymbol{n}| &= 1 \\ |\boldsymbol{o}| &= 1 \\ |\boldsymbol{a}| &= 1 \end{aligned} \tag{9}$$

Therefore, in order for the frame to be correct, the values of the variables representing the frame in a matrix must satisfy the set of equations in equation (9).

2.10.4 Matrix transformation

A transformation can be defined as a movement in space. Since a transformation is a change in the state of the frame, the movement of a frame relative to another frame can be represented similarly to a frame representation. These transformations can be either pure translation, pure rotation about an axis, or a combination of translations and rotations.

In order to represent pure translation it is important to understand that the frame is moving in space without any change in its orientation, thus the term pure translation. This means that the directional vectors remain in the same direction and do not change. The only thing that changes is the location of the origin of the reference frame relative to the universal reference frame. This new location can be represented by adding the vector of the original position relative to the universal reference frame with the vector representing the translation.

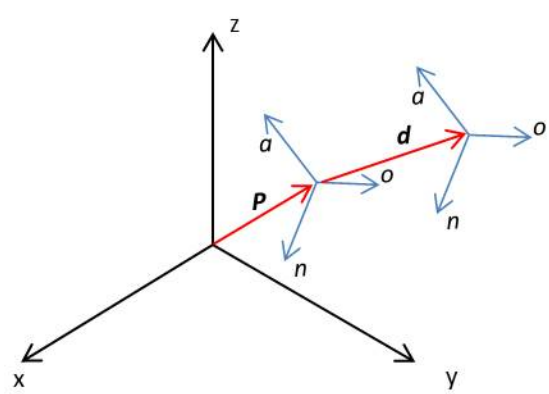


Figure 8. Pure translation in space.

Mathematically, this is achieved by pre multiplying the frame with a matrix that represents the translation transformation. Since the directional vectors don't change, this matrix has the form shown in equation (10).

$$T = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{10}$$

Here d_x, d_y, and d_z are the three components of the pure translation vector relative to the x, y, z axes of the universal reference frame. The first three columns represent the non-rotational state and the last one represents the translation. Thus by pre multiplying the matrix T to the matrix representing the original position, equation (11) is obtained.

$$F_{new=} \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} n_x & o_x & a_x & (p_x + d_x) \\ n_y & o_y & a_y & (p_y + d_y) \\ n_z & o_z & a_z & (p_z + d_z) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{11}$$

In order to understand pure rotation of a frame, it would be best to consider a simplified case in which the frame origin is located at the origin of the universal reference frame first.

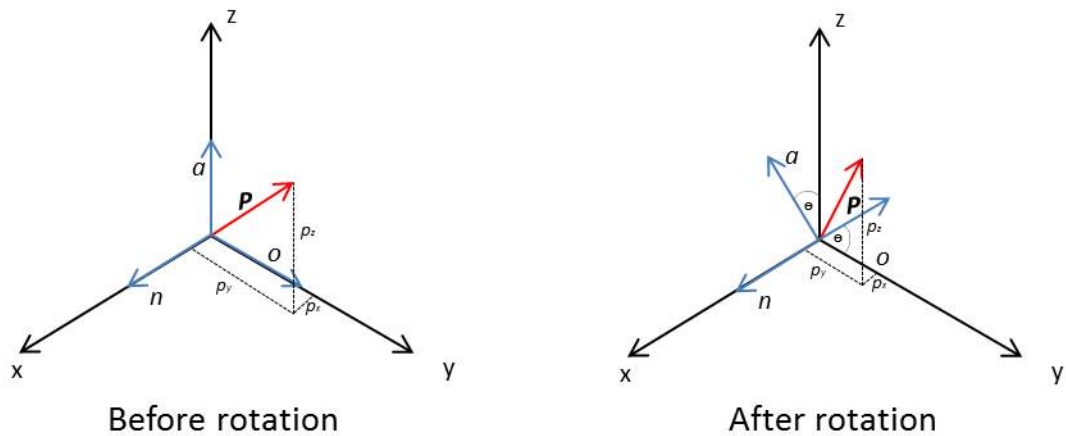


Figure 9. Coordinates in a rotating frame before and after rotation.

In Figure 9 , the frame $F_{n,o,a}$ is located at the origin of the reference frame $F_{x,y,z}$ and it rotates about the x axis by an angle of Θ. A point **P** is also attached to the frame $F_{n,o,a}$, and as the frame rotates, the point rotates with it. The point location is represented with the coordinates p_x, p_y and p_z relative to the reference frame and p_n, p_o and p_a relative to the moving frame. Note that after rotation, the coordinates p_n, p_o and p_a remained the same relative to the moving frame $F_{n,o,a}$. However, the coordinates p_x, p_y and p_z relative to the reference frame have changed. In order to determine these new values the following equations are used.

$$p_x = p_n \tag{12}$$

$$p_y = p_o \cos\theta - p_a \sin\theta \tag{13}$$

$$p_z = p_o \sin\theta + p_a \cos\theta \tag{14}$$

Equation (12) states that there is no change in p_x since it is rotating about the x axis, the change in p_y and p_z is denoted by the other equations. Equations (12), (13), and (14) can also be combined and rewritten in matrix form by;

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} p_n \\ p_o \\ p_a \end{bmatrix} \tag{15}$$

For the cases in which the frame is rotating about a different axis, equation (15) can be rewritten as equations (516) and (17) for rotation about the y and z axis respectively.

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} p_n \\ p_o \\ p_a \end{bmatrix} \tag{16}$$

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_n \\ p_o \\ p_a \end{bmatrix} \tag{17}$$

If the movement of a point attached to the origin of a frame $F_{n,o,a}$ which rotates relative to a universal reference frame consists of both translations and rotations, the procedure to determine its final coordinates can be simplified into a set of translations and rotations executed in a particular order. This order is very important and cannot be changed since geometric transformations performed in different order result in completely different orientations.

2.10.5 Forward and inverse kinematics of robots

Forward kinematics consists of determining the end effector coordinates given the joint variables while inverse kinematics is the reverse. Consider a robot with several links moving,

the robot stops, and the angle and position of its joints at that particular position are known. Using this information the location of the end effector can be determined using the equations of the forward kinematic analysis. Therefore, the location of the end effector at any instant can be determined using this approach. On the other hand, if the final location of the end effector is desired, then the angle and position of each joint that makes the end effector to reach that location must be determined. This approach is known as the inverse kinematic analysis, and these equations are much more important since the robot uses them to determine the joint variables at which the links must be in order for the end effector to reach the desired location.

For forward kinematics, a set of equations that describe the current configuration of the robot shall be developed, such that once values are added to the joint angles and positions, the position and orientation of the end effector can be determined. These equations can then be used to derive the inverse kinematic equations of the robot. Similarly to what was mentioned earlier, a total of six degrees of freedom are needed to describe a frame attached to the end effector relative to a reference frame as can be seen in Figure 10 the undefined connection between both frames represent the configuration of the robot which can vary depending on the robot to be modeled.

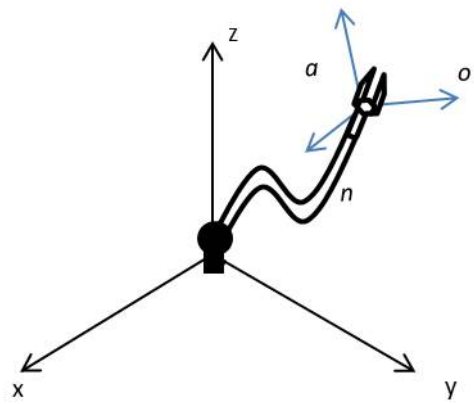


Figure 10. Robot hand relative to reference frame.

It is simpler to understand the modeling for forward or inverse kinematic equations if both position and orientation are explained separately; first the equations describing the end effector's position shall be developed, then the equations describing its orientation. Finally both sets of equations are combined into what is called the transformation matrix.

As mentioned earlier the position of the origin of a frame attached to the end effector has three degrees of freedom, thus, it can be fully defined by three pieces of information. As a result, this position can be defined by any customary system of coordinates (Cartesian, spherical, or cylindrical).

A Cartesian coordinate system is shown in Figure 1. A robot can be described in terms of the Cartesian coordinates, as shown in Figure 11, with three linear displacements along the x, y, and z axes.

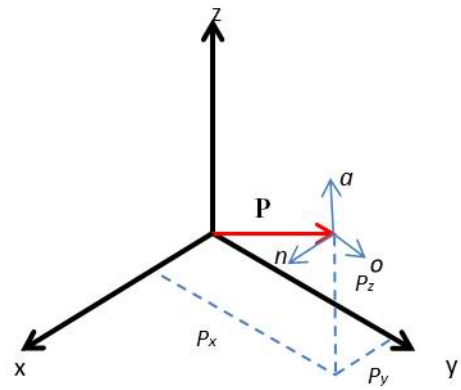


Figure 11. Cartesian coordinates.

The vector **P** is;

$$P = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \tag{18}$$

In this type of configuration all link joints are linear and the positioning is achieved by moving those joints along the axes. There is no rotation in this description and the transformation matrix representing this coordinate system is a simple translation matrix.

$$T_P^R = T_{cart}(P_x, P_y, P_z) = \begin{bmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{19}$$

Equation (19) represents the transformation between the reference frame *R* and the frame attached to the end effector *P*. $T_{cart}(P_x, P_y, P_z)$ represents the Cartesian transformation matrix. Regarding the inverse kinematic solution, the desired solution must be set equal to P.

For cylindrical joints there are two linear translations and one rotation. There is a translation r along the x axis, a rotation α with respect to the z axis and a translation l along the z axis as shown in Figure 12.

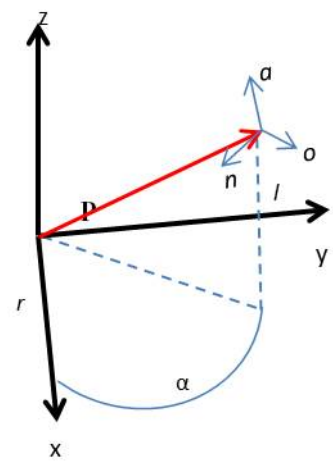


Figure 12. Cylindrical coordinates.

Since all of these transformations are with respect to the universal reference frame, the global transformation matrix can be found by pre multiplying each matrix.

$$T_P^R = T_{cyl}(r, \alpha, \mathrm{l}) = T_{rans}(0,0,l)Rot(z,\alpha)T_{rans}(r,0,0) \tag{20}$$

$$T_P^R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C\alpha & -\mathrm{S}\alpha & 0 & 0 \\ S\alpha & \mathrm{C}\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & r \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{21}$$

$$T_P^R = T_{cyl}(r, \alpha, \mathrm{l}) = \begin{bmatrix} C\alpha & -\mathrm{S}\alpha & 0 & rC\alpha \\ S\alpha & \mathrm{C}\alpha & 0 & rS\alpha \\ 0 & 0 & 1 & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{22}$$

In equation (22), the first three columns represent the orientation and the S and C represent sine and cosine respectively. The last column describes translation of one frame with respect to another. Hence, when describing a joint using cylindrical coordinate system, due to the rotation

α about the z axis, the orientation of the frame attached to the end effector will change due to translation. Thus, by post-multiplying the cylindrical coordinate matrix by a rotation matrix of $Rot(a, -\alpha)$ the frame can be restored to the same location but parallel to the universal frame.

Finally, the spherical coordinate system consists of one linear motion and two rotations. Its sequence is a translation along the z axis, a rotation β about the y axis and a rotation γ about the z axis, as shown in Figure 13. Similarly to the cylindrical system, since these transformations are done relative to the universal reference frame, by pre-multiplying each matrix, the three transformations can be found.

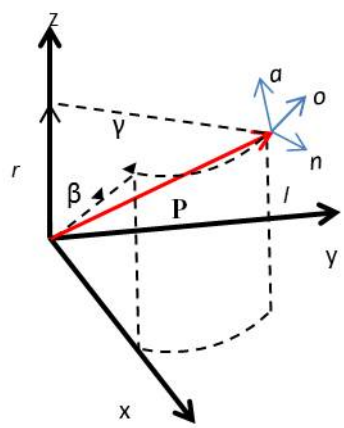


Figure 13. Spherical coordinates.

$$T_P^R = T_{sph}(r, \beta, \gamma) = Rot(z, \gamma)Rot(\gamma, \beta)T_{rans}(0,0,r) =$$

$$\begin{bmatrix} C\gamma & -S\gamma & 0 & 0 \\ S\gamma & C\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C\beta & 0 & S\beta & 0 \\ 0 & 1 & 0 & 0 \\ -S\beta & 0 & C\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{23}$$

$$T_P^R = T_{sph}(r, \beta, \gamma) = \begin{bmatrix} C\beta C\gamma & -S\gamma & S\beta C\gamma & rS\beta C\gamma \\ C\beta S\gamma & C\gamma & S\beta S\gamma & rS\beta S\gamma \\ -S\beta & 0 & C\beta & rC\beta \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{24}$$

The first three columns of equation (24) represent the orientation of the frame after these transformations and the last column represents its position. Here too, the orientation might be restored to being parallel to the reference frame. Because both angles β and γ are coupled, the inverse kinematic equations for spherical coordinate systems are more complicated.

Once the position of the robot has been determined and reached, the next step will be to rotate the end effector to a desired orientation; this can only be accomplished if the rotation is done about the reference frame since rotating about the universal reference frame will cause the position to change as well. Depending on the robot wrist and the way the joints are assembled, two common configurations are explained by Niku [9]: the Roll, Pitch, Yaw (RPY) configuration and the Euler angles.

The Roll, Pitch, Yaw configuration consists of three rotations relative to the **a, o, n** axes, respectively, that will orient the robot hand to a desired position. This is done assuming that the current frame is parallel to the reference frame, thus the orientation is the same before the application of RPY. Since the position of the frame will change if the rotation happens relative to the universal reference frame, all matrices related to the orientation change due to RPY will be post-multiplied. The rotation about **a** is called Roll, the rotation about **o** is called Pitch and the rotation about **n** is called Yaw. The RPY representation in matrix form is:

$RPY(\emptyset_a\emptyset_o\emptyset_n) = Rot(a,\emptyset_a)Rot(o,\emptyset_o)Rot(n,\emptyset_n)$=

$$\begin{bmatrix} C\emptyset_a C\emptyset_o & C\emptyset_a S\emptyset_o S\emptyset_n - S\emptyset_a C\emptyset_n & C\emptyset_a S\emptyset_o C\emptyset_n + S\emptyset_a S\emptyset_n & 0 \\ S\emptyset_a C\emptyset_o & S\emptyset_a S\emptyset_o S\emptyset_n + C\emptyset_a C\emptyset_n & S\emptyset_a S\emptyset_o C\emptyset_n - C\emptyset_a S\emptyset_n & 0 \\ -S\emptyset_o & C\emptyset_o S\emptyset_n & C\emptyset_o C\emptyset_n & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{25}$$

Equation (25) represents the rotation of the object caused by the RPY only. The full positioning and orientation matrix is determined by combining the two matrices; for example a robot with spherical designed for spherical coordinates can be represented by $T_H^R = T_{sph}(r,\beta,\gamma)\ RPY(\emptyset_a\emptyset_o\emptyset_n)$.

The inverse kinematics solution of the RPY is more complex since sine and cosine information of the three individual angles is required. In order to solve for the sines and cosines the angles have to be decoupled. By pre-multiplying both sides of equation (25) by the inverse of $Rot(a,\emptyset_a)$:

$$Rot(a,\emptyset_a)^{-1}RPY(\emptyset_a\emptyset_o\emptyset_n) = Rot(o,\emptyset_o)Rot(n,\emptyset_n) \tag{26}$$

By assuming that the final orientation depicted by the RPY is represented by the (**n,o,a**) matrix, it can be seen that:

$$Rot(a,\emptyset_a)^{-1}\begin{bmatrix} n_x & o_x & a_x & 0 \\ n_y & o_y & a_y & 0 \\ n_z & o_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = Rot(o,\emptyset_o)Rot(n,\emptyset_n) \tag{27}$$

Multiplying;

$$\begin{bmatrix} n_x C\emptyset_a + n_y S\emptyset_a & o_x C\emptyset_a + o_y S\emptyset_a & a_x C\emptyset_a + a_y S\emptyset_a & 0 \\ n_y C\emptyset_a - n_x S\emptyset_a & o_y C\emptyset_a - o_x S\emptyset_a & a_y C\emptyset_a - a_x S\emptyset_a & 0 \\ n_z & o_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} C\emptyset_o & S\emptyset_o S\emptyset_n & S\emptyset_o C\emptyset_n & 0 \\ 0 & C\emptyset_n & -S\emptyset_n & 0 \\ -S\emptyset_o & C\emptyset_o S\emptyset_n & C\emptyset_o C\emptyset_n & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{28}$$

The n, o, a components in equation (27), represent the final desired values normally known. The RPY values are still unknown, so, by equating the different elements on both sides of equation (28) the following results are obtained:

From the 2,1 element;

$$n_y C\emptyset_a - n_x S\emptyset_a = 0, \rightarrow \emptyset_a = \arctan(n_x, n_y) = \arctan(-n_x, -n_y) \tag{29}$$

From 3,1 and 1,1;

$$S\emptyset_o = -n_z$$

$$C\emptyset_o = n_x C\emptyset_a + n_y S\emptyset_a \rightarrow \emptyset_o = \arctan[-n_z, (n_x C\emptyset_a + n_y S\emptyset_a)] \tag{30}$$

Finally from 2,2 and 2,3;

$$C\emptyset_n = o_y C\emptyset_a - o_x S\emptyset_a$$

$$S\emptyset_n = -a_y C\emptyset_a + a_x S\emptyset_a \rightarrow \emptyset_n = \arctan[(-a_y C\emptyset_a + a_x S\emptyset_a), (o_y C\emptyset_a - o_x S\emptyset_a)] \tag{31}$$

The Euler angle method is similar to the RPY method with the exception that the last rotation is about the **a** axis as well. Since the rotation must also be made along the current reference axis

in order to prevent position change the rotations representing the Euler angles will be ϕ for the rotation along the **a** axis, θ for the rotation along the **o** axis, and ψ for the second rotation along the **a** axis.

$$Euler(\emptyset,\theta,\psi) = Rot(a,\emptyset)Rot(o,\theta)Rot(a,\psi) = \begin{bmatrix} C\emptyset C\theta C\psi - S\emptyset S\psi & -C\emptyset C\theta S\psi - S\emptyset C\psi & C\emptyset S\theta & 0 \\ S\emptyset C\theta C\psi + C\emptyset S\psi & -S\emptyset C\theta S\psi + C\emptyset C\psi & S\emptyset S\theta & 0 \\ -S\theta C\psi & S\theta S\psi & C\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (32)$$

Equation (32) represents the orientation change caused by the Euler angles, where C and S are representations of cosines and sines respectively. The final location will be dictated by a combination of the translation matrix and this orientation matrix. Similar to the RPY method, the inverse kinematic solution for the Euler angles can be found by pre-multiplying the Euler matrix by the inverse of the rotation about the **a** axis then equating the elements of the two sides of the equation to each other.

Therefore, from the 2,3 elements;

$$-a_n S\emptyset + a_y C\emptyset = 0 \rightarrow \emptyset = \arctan(a_y, a_x)\ or\ \emptyset = \arctan(-a_y, -a_x) \quad (33)$$

With ϕ evaluated, the left hand side of the equation is known. Then from the 2,1 and 2,2 elements;

$$S\psi = -n_x S\emptyset + n_y C\emptyset \quad (34)$$

$$C\psi = -o_x S\emptyset + o_y C\emptyset \rightarrow \psi = \arctan[(-n_x S\emptyset + n_y C\emptyset), (-o_x S\emptyset + o_y C\emptyset)] \quad (35)$$

Once all the values are found, in order to construct the matrix that describes the translation and orientation, a combination of both matrices must be employed. This matrix varies depending on the robot configuration, for instance, if the robot is spherical and the joints are set as an RPY then the equation describing its full movement will consist of $T_H^R = T_{sph}(r, \beta, \gamma)\ RPY(\emptyset_a \emptyset_o \emptyset_n)$. The same principle is employed to the other configurations.

2.10.6 Denavit-Hartenberg convection

The Denavit-Hartenberg (D-H) convection is explained by both Spong et al. [10] and Craig et al. [13] as an approach to describe the joint and end effector orientation and position in terms of the main link parameters. It is important to understand that the Denavit-Hartenberg convection can obtained, in two ways are highly used and correct but the approach is relatively different. Spong et al. explains the D-H method, which has been presented by Richard Paul in 1981 [14]. The convention states that each homogeneous transformation is shown as the product of four basic transformations.

$$A_i = Rot_{z,\theta_i} Trans_{z,d_i} Trans_{x,a_i} Rot_{x,a_i} \tag{36}$$

$$= \begin{bmatrix} C\theta_i & -S\theta_i & 0 & 0 \\ S\theta_i & C\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C\alpha_i & -S\alpha_i & 0 \\ 0 & S\alpha_i & C\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} C\theta_i & -S\theta_i C\alpha_i & S\theta_i S\alpha_i & a_i C\theta_i \\ S\theta_i & C\theta_i C\alpha_i & -C\theta_i S\alpha_i & a_i S\theta_i \\ 0 & S\alpha_i & C\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Similarly, Craig et al.[13] developed a different approach to this,

$$T_i^{i-1} = \begin{bmatrix} C\theta_i & -S\theta_i & 0 & a_{i-1} \\ S\theta_i C\alpha_{i-1} & C\theta_i C\alpha_{i-1} & -S\alpha_{i-1} & Sa_{i-1}d_i \\ S\theta_i S\alpha_{i-1} & C\theta_i S\alpha_{i-1} & C\alpha_{i-1} & Ca_{i-1}d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{37}$$

The main difference between both approaches is the order on which the transformations are used as a result, the location of the origin of the coordinate frame for the ith link. In Paul's notation it is located at the end of the link, whereas in Craig's notation it is located at its base. As mentioned before, both approaches are valid, however it is important to understand the difference between the two to avoid confusion.

These four parameters $\theta_i, a_i, d_i, \alpha_i$ are **Joint Angle**, **Link Length**, **Link Offset** and **Link Twist** respectively. These terms derive from specific aspects of the geometric relationship between two coordinate frames. Since matrix A_i is a function of a single variable three of the quantities shown above are constant for a given link while the parameter $\theta_i$is a variable for a revolute joint.

2.10.7 Forward Kinematics for the Puma 560 robot

As detailed by Craig et al. [13], the information found in the robot parameters table is used to compute each of the link transformations.

Table 4 Craig's Parameters for the PUMA 560 robot

Joint	Parameters		
	a_{i-1} (m)	d_i(m)	α_{i-1}(°)
1	0	0	0
2	0	0	-90
3	a_2	d_3	0
4	a_3	d_4	-90
5	0	0	90
6	0	0	-90

For this example, Craig uses the parameters for the PUMA 560 robot listed in Table 4. Substituting these parameters in equation (37) results in the following transformations for each link:

$$T_1^0 = \begin{bmatrix} C\theta_1 & -S\theta_1 & 0 & 0 \\ S\theta_1 & C\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_2^1 = \begin{bmatrix} C\theta_2 & -S\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -S\theta_2 & -C\theta_2 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_3^2 = \begin{bmatrix} C\theta_3 & -S\theta_3 & 0 & a_2 \\ S\theta_3 & C\theta_3 & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_4^3 = \begin{bmatrix} C\theta_4 & -S\theta_4 & 0 & a_3 \\ 0 & 0 & -1 & d_4 \\ -S\theta_4 & -C\theta_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{38}$$

$$T_5^4 = \begin{bmatrix} C\theta_5 & -S\theta_5 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ S\theta_5 & C\theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_6^5 = \begin{bmatrix} C\theta_6 & -S\theta_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -S\theta_6 & -C\theta_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We now create theT_6^0, to do this each link's transformation is multiplied. While doing this, several sub results shall be used in order to simplify the inverse kinematics derived in the next section.

$$T_6^4 = T_5^4\, T_6^5 = \begin{bmatrix} C\theta_5 C\theta_6 & -C\theta_5 S\theta_6 & -S\theta_5 & 0 \\ S\theta_6 & C\theta_6 & 0 & 0 \\ S\theta_5 C\theta_6 & -S\theta_5 S\theta_6 & C\theta_5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{39}$$

$$T_6^3 = T_4^3 * T_6^4 = \begin{bmatrix} C\theta_4 C\theta_5 C\theta_6 - S\theta_4 S\theta_6 & -C\theta_4 C\theta_5 C\theta_6 - S\theta_4 C\theta_6 & -C\theta_4 C\theta_5 & a_3 \\ S\theta_5 C\theta_6 & -S\theta_5 S\theta_6 & C\theta_5 & d_4 \\ -S\theta_4 C\theta_5 C\theta_6 - C\theta_4 S\theta_6 & S\theta_4 C\theta_5 S\theta_6 - C\theta_4 C\theta_6 & S\theta_4 S\theta_5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{40}$$

$$T_3^1 = T_2^1 * T_3^2 = \begin{bmatrix} C\theta_{23} & -S\theta_{23} & 0 & a_2 C\theta_2 \\ 0 & 0 & 1 & d_3 \\ -S\theta_{23} & -C\theta_{23} & 0 & -a_2 S\theta_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{41}$$

Here $\boldsymbol{C\theta_{23}} = \boldsymbol{C\theta_2 C\theta_3 - S\theta_2 S\theta_3}$ and $\boldsymbol{S\theta_{23}} = \boldsymbol{C\theta_2 S\theta_3 + S\theta_2 C\theta_3}$ are cosine and sine transformations respectively.

Then the equation is left with;

$$T_6^1 = T_3^1 * T_6^3 = \begin{bmatrix} o_{11} & o_{12} & o_{13} & p_x \\ o_{21} & o_{22} & o_{23} & p_y \\ r_{31} & r_{32} & o_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{42}$$

Finally;

$$T_6^0 = T_1^0 * T_6^1 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{43}$$

Where:

$$r_{11} = C\theta_1[C\theta_{23}(C\theta_4 C\theta_5 C\theta_6 - S\theta_4 S\theta_6) - S\theta_{23} S\theta_5 C\theta_6] + S\theta_1(S\theta_4 C\theta_5 C\theta_6 + C\theta_4 S\theta_6)$$
$$r_{21} = S\theta_1[C\theta_{23}(C\theta_4 C\theta_5 C\theta_6 - S\theta_4 S\theta_6) - S\theta_{23} S\theta_5 C\theta_6] - C\theta_1(S\theta_4 C\theta_5 C\theta_6 + C\theta_4 S\theta_6)$$
$$r_{31} = -S\theta_{23}(C\theta_4 C\theta_5 C\theta_6 - S\theta_4 S\theta_6) - C\theta_{23} S\theta_5 C\theta_6$$

$$r_{12} = C\theta_1[C\theta_{23}(-C\theta_4 C\theta_5 S\theta_6 - S\theta_4 C\theta_6) + S\theta_{23} S\theta_5 S\theta_6] + S\theta_1(C\theta_4 C\theta_6 - S\theta_4 C\theta_5 S\theta_6)$$
$$r_{22} = S\theta_1[C\theta_{23}(-C\theta_4 C\theta_5 S\theta_6 - S\theta_4 C\theta_6) + S\theta_{23} S\theta_5 S\theta_6] - C\theta_1(C\theta_4 C\theta_6 - S\theta_4 C\theta_5 S\theta_6)$$
$$r_{32} = S\theta_{23}(-C\theta_4 C\theta_5 S\theta_6 - S\theta_4 C\theta_6) + C\theta_{23} S\theta_5 S\theta_6$$

$$r_{13} = -C\theta_1(C\theta_{23} C\theta_4 S\theta_5 + S\theta_{23} C\theta_5) - S\theta_1 S\theta_4 S\theta_5$$
$$r_{23} = -S\theta_1(C\theta_{23} C\theta_4 S\theta_5 + S\theta_{23} C\theta_5) + C\theta_1 S\theta_4 S\theta_5 \tag{44}$$
$$r_{33} = S\theta_{23} C\theta_4 S\theta_5 - C\theta_{23} C\theta_5$$

$$p_X = C\theta_1[a_2 C\theta_2 + a_3 C\theta_{23} - d_4 S\theta_{23}] - d_3 S\theta_1$$
$$p_y = S\theta_1[a_2 C\theta_2 + a_3 C\theta_{23} - d_4 S\theta_{23}] + d_3 C\theta_1$$
$$p_z = -a_3 S\theta_{23} - a_2 S\theta_2 - d_4 C\theta_{23}$$

These constitute the forward kinematics of the Puma 560 robot. They specify what would be the position and orientation of link 6 with respect to link 0 of the robot.

2.10.8 Algebraic Joint angle solution for the Puma 560 robot

The Puma 560 robot is a very effective and flexible robot. The algorithms used in this investigation will solve the inverse kinematic problem of this robot in order to simplify the comparison process. The algebraic solution for this robot configuration exists, and it will be

used in order to ensure that the algorithms are providing reliable results. Craig et al. [13] explains how to derive the algebraic solution of the PUMA 560 robot.

The solution

$$T_6^0 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{45}$$

$$= {}_1^0T(\theta_1) * {}_2^1T(\theta_2) * {}_3^2T(\theta_3) * {}_4^3T(\theta_4) * {}_5^4T(\theta_5) * {}_6^5T(\theta_6)$$

Of $\theta_i$when $T_6^0$is given as numeric values.

We restate the equation leaving $\theta_1$in the left side of the equation.

$$[{}_1^0T(\theta_1)]^{-1} * T_6^0 = {}_2^1T(\theta_2) * {}_3^2T(\theta_3) * {}_4^3T(\theta_4) * {}_5^4T(\theta_5) * {}_6^5T(\theta_6) \tag{46}$$

Thus inverting ${}_1^0T$

$$\begin{bmatrix} C\theta_1 & S\theta_1 & 0 & 0 \\ -S\theta_1 & C\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = {}_6^1T \tag{47}$$

Equating the (2,4) elements from both sides of equation (40) we have

$$-S\theta_1 p_x + C\theta_1 p_y = d_3 \tag{48}$$

We now make the trigonometric substitution;

$$p_x = \rho * \cos(\varphi) \tag{49}$$

$$p_y = \rho * \sin(\varphi)$$

Where;

$$\rho = \sqrt{{p_x}^2 + {p_y}^2} \tag{50}$$

$$\varphi = Atan2(p_y, p_x)$$

Substituting (49) into (48) we obtain

$$C\theta_1 S\varphi - S\theta_1 C\varphi = \frac{d_3}{\rho} \tag{51}$$

Using the difference of angles formula

$$\sin(\varphi - \theta_1) = \frac{d_3}{\rho} \tag{52}$$

Hence

$$\cos(\varphi - \theta_1) = \pm\sqrt{1 - \left(\frac{d_3}{\rho}\right)^2} \tag{53}$$

And so

$$\varphi - \theta_1 = Atan2\left(\frac{d_3}{\rho}, \pm\sqrt{1 - \left(\frac{d_3}{\rho}\right)^2}\right) \tag{54}$$

Finally,

$$\theta_1 = Atan2(p_x, p_x) - Atan2\left(d_3, \pm\sqrt{{p_x}^2 + {p_y}^2 - {d_3}^2}\right) \tag{55}$$

With this we have two possible solutions for θ_1 corresponding to the plus-minus sign in (55). Now that θ_1 is known, the left hand side of (40) is known. Thus equating the (1,4) and (3,4) elements from both sides of (40) we get;

$$C\theta_1 p_x + Sp_y = a_3 C\theta_{23} - d_4 S\theta_{23} + a_2 C\theta_2 \tag{56}$$

$$-p_z = a_3 S\theta_{23} + d_4 C\theta_{23} + a_2 S\theta_2$$

Squaring (40) and (48), and then adding the results, we obtain;

$$a_3 C\theta_3 + d_4 S\theta_3 = K \tag{57}$$

Where

$$K = \frac{{p_x}^2 + {p_y}^2 + {p_z}^2 - {a_2}^2 - {a_3}^2 - {d_3}^2 - {d_4}^2}{2a_2} \tag{58}$$

Now, solving by the same trigonometric substitution used in equation (48) we find;

$$\theta_3 = Atan2(a_3, d_4) - Atan2\left(K, \pm\sqrt{{a_3}^2 + {d_4}^2 - K^2}\right) \tag{59}$$

We can now rearrange equation (45) so that all the left hand side is a function of only known values and θ_2.

$$[{}^0_3T(\theta_2)]^{-1}\, T^0_6 = {}^3_4T(\theta_4)\, {}^4_5T(\theta_5)\, {}^5_6T(\theta_6) \tag{60}$$

$$\begin{bmatrix} C\theta_1 C\theta_{23} & S\theta_1 C\theta_{23} & -S\theta_{23} & a_2 C\theta_3 \\ -C\theta_1 S\theta_{23} & -S\theta_1 S\theta_{23} & -C\theta_{23} & a_2 S\theta_3 \\ -S\theta_1 & C\theta_1 & 0 & -d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = {}^3_6T \tag{61}$$

Where ${}^3_6T$ is given by equation (40) developed in the previous section. Equating the (1,4) and (2,4) elements from both sides we obtain;

$$C\theta_1 C\theta_{23} p_x + S\theta_1 C\theta_{23} p_y - S\theta_{23} p_z - a_2 C\theta_3 = a_3 \tag{62}$$

$$-C\theta_1 S\theta_{23} p_x - S\theta_1 S\theta_{23} p_y - C\theta_{23} p_z + a_2 S\theta_3 = d_4$$

These equations can be solved simultaneously to obtain $C\theta_{23}$ and $S\theta_{23}$;

$$S\theta_{23} = \frac{(-a_3 - a_2 C\theta_3) p_z + (C\theta_1 p_x + S\theta_1 p_y)(a_2 S\theta_3 - d_4)}{{p_z}^2 + (C\theta_1 p_x + S\theta_1 p_y)^2} \tag{63}$$

$$C\theta_{23} = \frac{(a_2 S\theta_3 - d_4) p_z - (-a_3 - a_2 C\theta_3)(C\theta_1 p_x + S\theta_1 p_y)}{{p_z}^2 + (C\theta_1 p_x + S\theta_1 p_y)^2}$$

Since the denominators are the same and greater than one, the equation can be solved for the sum of $\theta_2$and θ_3 as;

$$\theta_{23} = Atan2\big[(-a_3 - a_2 C\theta_3)p_z - \big(C\theta_1 p_x + S\theta_1 p_y\big)(d_4 - a_2 S\theta_3), (a_2 S\theta_3 - d_4)p_z - (a_3 + a_2 C\theta_3)\big(C\theta_1 p_x + S\theta_1 p_y\big)\big] \quad (64)$$

Equation (64) computes the values of θ_{23} according to the four possible combinations of solutions for θ_1 and θ_3. As a result, the four possible solutions for θ_2 can be obtained as;

$$\theta_2 = \theta_{23} - \theta_3 \quad (65)$$

The entire left side of equation (61) is now known. Thus, equating the (1,3) and the (3,3) elements from both sides, we get;

$$r_{13} C\theta_1 C\theta_{23} + r_{23} S\theta_1 C\theta_{23} - r_{33} S\theta_{23} = -C\theta_4 S\theta_5 \quad (66)$$

$$-r_{13} S\theta_1 + r_{23} C\theta_1 = S\theta_4 S\theta_5$$

As long as $S\theta_5 \neq 0$, the solution for θ_4 can be expressed as;

$$\theta_4 = Atan2(-r_{13} S\theta_1 + r_{23} C\theta_1, -r_{13} C\theta_1 C\theta_{23} - r_{23} S\theta_1 C\theta_{23} + r_{33} S\theta_{23}) \quad (67)$$

Considering equation (45), it can now be rewritten having the left hand side consisting only of known values and θ_4;

$$[{}^0_4T(\theta_4)]^{-1}\, T^0_6 = {}^4_5T(\theta_5){}^5_6T(\theta_6) \quad (68)$$

Where $[{}^0_4T(\theta_4)]^{-1}$ is given by;

$$\begin{bmatrix} C\theta_1 C\theta_{23} C\theta_4 + S\theta_1 S\theta_4 & S\theta_1 C\theta_{23} C\theta_4 - C\theta_1 S\theta_4 & -S\theta_{23} C\theta_4 & -a_2 C\theta_3 C\theta_4 + d_3 S\theta_4 - a_3 C\theta_4 \\ -C\theta_1 C\theta_{23} S\theta_4 + S\theta_1 C\theta_4 & -S\theta_1 C\theta_{23} S\theta_4 - C\theta_1 C\theta_4 & S\theta_{23} S\theta_4 & a_2 C\theta_3 S\theta_4 + d_3 C\theta_4 + a_3 S\theta_4 \\ -C\theta_1 S\theta_{23} & -S\theta_1 S\theta_{23} & -C\theta_{23} & a_2 S\theta_3 - d_4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (69)$$

And T_6^4 is given by equation (39) presented in the previous section. Now, equating the (1,3) and (3,3) elements from both sides of equation (68), yields;

$$r_{13}(C\theta_1 C\theta_{23} C\theta_4 + S\theta_1 S\theta_4) + r_{23}(S\theta_1 C\theta_{23} C\theta_4 - C\theta_1 S\theta_4) - r_{33}(S\theta_{23} C\theta_4) = -S\theta_5$$

$$r_{13}(-C\theta_1 S\theta_{23}) + r_{23}(-S\theta_1 S\theta_{23}) + r_{33}(-C\theta_{23}) = C\theta_5 \tag{70}$$

Thus, the equation can now be solved for θ_5 as;

$$\theta_5 = Atan2(S\theta_5, C\theta_5) \tag{71}$$

Where $C\theta_5$and $S\theta_5$ are given by equation (70).

Applying the same method, $[{}^0_5T]^{-1}$ can be computed and equation (45) is written as;

$$[{}^0_5T]^{-1}\, T_6^0 = {}^5_6T(\theta_6) \tag{72}$$

And as before, equating the (3,1) and (1,1) elements of this equation yields;

$$\theta_6 = Atan2(S\theta_6, C\theta_6) \tag{73}$$

Where

$$S\theta_6 = -r_{11}(C\theta_1 C\theta_{23} S\theta_4 - S\theta_1 C\theta_4) - r_{21}(S\theta_1 C\theta_{23} S\theta_4 + C\theta_1 C\theta_4) + r_{31}(S\theta_{23} S\theta_4)$$

$$C\theta_6 = r_{11}[(C\theta_1 C\theta_{23} C\theta_4 + S\theta_1 S\theta_4)C\theta_5 - C\theta_1 S\theta_{23} S\theta_5] + r_{21}[(S\theta_1 C\theta_{23} C\theta_4 - C\theta_1 S\theta_4)C\theta_5 - S\theta_1 S\theta_{23} S\theta_5] - r_{31}(S\theta_{23} C\theta_4 C\theta_5 + C\theta_{23} S\theta_5) \tag{74}$$

The difficulty of deriving the algebraic joint solution of a robot increases with the amount of degrees of freedom it has. As mentioned before, when the robot has 6 degrees of freedom, the solution becomes complex now that there are multiple solutions for each joint. A solution becomes even more difficult to reach once the robot possesses 7 or more degrees of freedom

now that each joint can have an infinite number of solutions that can satisfy the inverse kinematic problem. As a result, the use of algorithms to determine the inverse kinematic solution of a robot becomes a cost effective method at higher degrees of freedom.

2.11.0 Algorithms

The procedure for solving the problems in terms of the actions that are executed and the order in which they are executed is considered an algorithm. Deitel et al. [15] explain that an algorithm can be seen as a solution to any computing related problem that involves a series of steps or actions in a specified order. The order in which the algorithm is performed is very important. For instance, consider the following example: in order to drive a car; the driver must first get inside the car, then turn it on, then set the gear to forward drive, then release the break and finally press the gas pedal. This order of events ensures that the car will move. On the other hand, if the order consists of getting inside the car, then setting the gear on forward drive, then releasing the break and finally pressing the gas pedal, one can understand that the car will not move at all since it was never turned on. As can be seen in Wolfram MathWorld [16] there is an extensive list of algorithms for different purposes. For instance the Newton's method is an algorithm used to find roots of equations that uses the first terms of a Taylor series that lies in the vicinity of a potential root. Genetic algorithms mimic the scenario of natural selection by randomly changing a group of programs, then, it selects certain programs due to their performance and continues changing them until certain parameters are met. Another example is Bees algorithm, which mimics the food foraging behavior of honey bees for

optimization purposes. These algorithms are often edited in order to improve their effectiveness, this is known as software enhancement. There are two kinds of enhancement, commonly known as adaptive and perfective maintenance [17]. Adaptive maintenance refers to changes made to the software in response to changes in its environment such as new government regulations. Perfective maintenance refers to enhancing software in order to improve its effectiveness, making it perform faster or yield more accurate results. For instance, the work presented by Packianather et al. [18] in which the Bees algorithm was improved by simply changing the parameters that control how many search points are used in a specific search region, the algorithm was enhanced allowing it to show an average improvement of 41% in convergence speed. Similarly the method proposed by Hu Hang et al. [19] shows that changing the crossover parameter into an adaptive one and introducing independent genetic operations to a Genetic algorithm greatly improved its convergence speed and the calculation efficiency.

3 Methodology

The purpose of this study is to develop a method to compare and rate different algorithms that analyze the inverse kinematics of a 6R robotic manipulator. Once developed, it can be used as a tool to compare these algorithms and determine which one is the best to implement in a robot so it can obtain the final result more efficiently. To achieve this, two different inverse kinematic algorithms that can solve the inverse kinematic problem of a robot in a real-time fashion were selected. By researching and gathering enough information about the techniques the authors had developed, the algorithms were efficiently programmed and the leanest code was developed utilizing Matlab 7.5 as the code language. The programed codes were then executed 10 times at three different precision coefficients $(\boldsymbol{e})$, where $e^{-2} = 10^{-2\circ}$, $e^{-4} = 10^{-4\circ}$ and, $e^{-6} = 10^{-6\circ}$.these values signify the tolerance margin that the algorithm uses to reach the result is of the order of $10^{-2}, 10^{-4}$ or 10^{-6}. The time it took the algorithm to solve the Inverse Kinematic problem and the precision of the result was measured and recorded. The average time and precision was used for this comparison. In order to achieve the most reliable results, this experiment was executed in two computers that were equipped with the following processors:

- ***AMD Turion (tm) 64 X2 2.2GHz***
- ***Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz***

Using both processors will also help to understand if there is a considerable difference between the performance of the algorithms using these two architectures.

The Denavit Hatenberg parameters that will be used for this experiment were presented by Feng. et al. [4].

For a PUMA 560 robot the link parameters a_i, d_i, α_i are listed on Table 5:

Table 5 D-H Paul's Parameters for the PUMA 560 robot

Joint	**Parameters**		
	$\boldsymbol{a_i}$ **(m)**	$\boldsymbol{d_i}$**(m)**	$\boldsymbol{\alpha_i}$**(°)**
1	0	0.6604	-90
2	4.320	0.2000	0
3	0	-0.0505	90
4	0	0.4320	-90
5	0	0.0	90
6	0	0.0565	0

Once the algorithms were programmed and executed the aforementioned number of times, the algorithm with the best performance was determined. The data acquired by this study is organized in a way that shows the Inverse Kinematic algorithm, the architecture on which it was tested, the time the Inverse Kinematic Algorithm took to solve the problem and the precision of such result.

This investigation will provide to the engineering community an exhaustive comparison between two widely used inverse kinematic algorithm techniques using an experimental approach. This can become a valuable reference for future development of inverse kinematic algorithms now that it will provide insight on the performance level of the Newton-Raphson and the MDFP methods when used to solve the inverse kinematic problem of robots.

3.0.0 Algorithm selection.

The first research step was to select appropriate algorithms that focus on the characteristics that are going to be tested (robot degrees of freedom, robot type and link composition) in order for them to be rated and contrasted with each other and thus, test the capabilities of the test-bench. The degrees of freedom of the robot for which the algorithm is programmed will have to be six; this is considering that the processing required to solve the inverse kinematic algorithm for a seven or higher degrees of freedom robot is too large and these robots are not common in the industry. The algorithms selected and used for the purpose of this investigation are **A control Algorithm of General 6R Mechanic Arm Based on Inverse Kinematics** by Qu et al.[3], and **Inverse Kinematic Solution for Robot Manipulator Based on Electromagnetism-Like and Modified DFP Algorithms** by Feng et al. [4].

3.1.0 Testing environment

The second phase of the project consists on writing and executing these algorithms in order for them to be tested. The algorithms shall be written utilizing MatLab 7.5. This language, besides being relatively easy to learn is also a well tried language with a considerable amount of matrix operation subroutines and code. The program will be running on Windows 7 operating system with the minimal amount of additional processes running at the time, thus avoiding MatLab having to share processing power with othe5r processes on the computer. In order to achieve this, the computer will be operating in SAFE MODE which is a state that only essential files and drivers can be accessed. It also prevents unnecessary processes to run automatically, thus allowing Matlab 7.5 to perform at the best of its ability. Once the computer was started in SAFE

MODE only three processes were active alongside MatLab 7.5, these are; ***csrss.exe, winlogon.exe and ctfmon.exe.***

Robotic operations in the industry are normally control intensive, in addition, other executions such as sensor feedback need to be processed, and therefore a robust set of processors were selected as testing environments for the algorithms. An *AMD Turion (tm) 64 X2 2.2GHz* and an Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz processor will be used.

3.2.0 Features to be tested

As mentioned before, the purpose of the test-bench is to measure the time required by the algorithm to solve the inverse kinematic equation. This feature is often very important when working with a robotic manipulator since it is required for the robot to find its path as quickly as possible. The time required by the algorithm to solve the problem can be measured using a subroutine within the code, these kinds of subroutines are not uncommon, and can employ the internal clock of the processor in order to accurately determine the time required to reach the solution.

Another feature that will be tested is the precision of the algorithm. As mentioned before, the PUMA 560 robot has up to eight different solutions capable of reaching a desired position and orientation. In this experiment the algorithm was tasked with finding each of the solutions by giving it a starting point that is close to this solution. Feng et al. [4] determined all the solutions by utilizing the Electromagnetism-Like portion of the algorithm. These results were verified using the forward kinematics method presented by Spong et al. [10] and they all yielded the same result for the position and orientation of the manipulator. By using these solutions, a

relatively close approximation to the optimal result was introduced as the initial approximation criterion for the algorithms. Once these approximations were given to the algorithm, they were optimized and a transformation matrix that states the position and orientation that the optimized joint angle yield was determined. Then, this optimized transformation matrix value was compared with the to the desired (optimal) transformation matrix. The comparison was done by the following equation:

$$Error = (\| Resultant\ Transformation\ matrix - Desired\ Transformation\ matrix \|) \tag{75}$$

The algorithms used in this study use the Euclidean norm to determine the difference between each resulting vector, in a similar way, this equation helps to determine the difference between two transformation matrices. The Transformation matrix describes both position and orientation of the end effector, which means that it incorporates both units of orientation (degree) and position (meter). This equation takes the resulting transformation matrix obtained from the optimized joint angle and subtracts it from the transformation matrix of the desired position and orientation. The Euclidean norm of this matrix is then determined and used to quantify the difference between these two matrices. As a result, this value is used as a combined measure of error that incorporates the information of both the position and the orientation of the current result.

3.3.0 Initial values for iteration

Both the MDFP and the Newton Raphson algorithms have a similar mode of operation. Instead of searching the whole 6 dimensional spectrum for the optimal value (which is a lengthy process) an initial value for the joint angle is required to be given. Then, the algorithm optimizes this initial value to a desired precision.

In Chapter 1, Craig's derivation for the algebraic solution of the inverse kinematics of a Puma 560 robot was shown [13]. This option was considered to determine all the solutions for a given position and orientation hence a set of approximated initial values can be given to the algorithms. However, both algorithms were developed using Paul's notation, and it was later considered that applying Craig's notation to the algorithms would involve changing the algorithm sequence and this might affect the original algorithm's performance intended by the authors. As a result, using the algebraic solution for the PUMA 560 presented by Craig will not be viable for this experiment.

Feng et al.[4] used the full strength of the Electromagnetism-like algorithm to determine all the solutions for a given position and orientation of the robot. Since the experiment determined the precision of the algorithm by comparing the final position and orientation matrix, a comparison of joint angles was not required, thus it was decided to use all eight solutions determined by Feng et al to perform the experiment. By using these solutions, relatively close approximations were determined and used in the algorithms so that the approximated value could be optimized.

The approximation values for each solution are shown below;

$$\theta_{1\ FENG} = \begin{bmatrix} 14.9981 \\ 25.0043 \\ 34.9991 \\ 45.0663 \\ 54.9701 \\ 65.0246 \end{bmatrix} \qquad \theta_{1\ initial} = \begin{bmatrix} 10 \\ 20 \\ 30 \\ 40 \\ 50 \\ 60 \end{bmatrix}$$

$$\theta_{2\ FENG} = \begin{bmatrix} 14.9943 \\ 25.0045 \\ 35.0040 \\ -134.8443 \\ -54.9786 \\ -115.0963 \end{bmatrix} \qquad \theta_{2\ initial} = \begin{bmatrix} 10 \\ 20 \\ 30 \\ -130 \\ -50 \\ -110 \end{bmatrix}$$

$$\theta_{3\ FENG} = \begin{bmatrix} 14.9989 \\ -29.9990 \\ 144.9972 \\ -76.6554 \\ -36.5255 \\ 168.3681 \end{bmatrix} \qquad \theta_{3\ initial} = \begin{bmatrix} 10 \\ -30 \\ 140 \\ -70 \\ -30 \\ 160 \end{bmatrix}$$

$$\theta_{4\ FENG} = \begin{bmatrix} -142.5264 \\ -149.4014 \\ 34.3634 \\ 130.9456 \\ -15.9070 \\ 148.1551 \end{bmatrix} \qquad \theta_{4\ initial} = \begin{bmatrix} -140 \\ -140 \\ 30 \\ 130 \\ -10 \\ 140 \end{bmatrix}$$

$$\theta_{5\ FENG} = \begin{bmatrix} -142.9029 \\ -149.9997 \\ 35.0004 \\ -49.5765 \\ 18.4455 \\ -31.2045 \end{bmatrix} \qquad \theta_{5\ initial} = \begin{bmatrix} -140 \\ -140 \\ 30 \\ -40 \\ 10 \\ -30 \end{bmatrix}$$

$$\theta_{6\ FENG} = \begin{bmatrix} -142.9029 \\ 155.0002 \\ 145.0002 \\ 20.0620 \\ -44.5842 \\ -93.8781 \end{bmatrix} \qquad \theta_{6\ initial} = \begin{bmatrix} -140 \\ 150 \\ 140 \\ 20 \\ -40 \\ -90 \end{bmatrix}$$

$$\theta_{7\ FENG} = \begin{bmatrix} -142.9034 \\ 155.0008 \\ 144.9990 \\ -159.9225 \\ 44.5867 \\ 86.1114 \end{bmatrix} \qquad \theta_{7\ initial} = \begin{bmatrix} -140 \\ 150 \\ 140 \\ -150 \\ 40 \\ 80 \end{bmatrix}$$

$$\theta_{8\ FENG} = \begin{bmatrix} 14.9991 \\ -29.9968 \\ 144.9999 \\ 103.3375 \\ 36.5176 \\ 11.5898 \end{bmatrix} \qquad \theta_{8\ initial} = \begin{bmatrix} 10 \\ -30 \\ 140 \\ 100 \\ 30 \\ -10 \end{bmatrix}$$

3.4.0 Test Algorithms

3.4.1 Modified Davidon Fletcher Powell Algorithm (e.g. Feng et al. [4])

This algorithm was selected because of its unique design. It is composed of two algorithms that when combined, are able to determine the optimal joint angle values with no need for an approximated initial step to be provided. First the algorithm uses an Electromagnetic-like algorithm to determine a broad approximation of the optimal joint angles with an error of 10^{-1}, once this initial step has been selected, the DFP algorithm refines this joint value to an error of 10^{-6}. The algorithm begins utilizing the Electromagnetism-Like algorithm (e.g., Birbil et al. [20]) by first doing a wide spectrum search over a six dimensional hyper-plane to find the most suitable approximation to the optimal solution. This is accomplished by treating several randomly generated six dimensional points as charged particles in space and assigning charges depending on their proximity to the optimal value. This method is not capable of efficiently finding a result with an error smaller than 10^{-1} which is why the second section of the algorithm, called MDFP, uses this initial approximation to optimize it to an error of 10^{-2}, 10^{-4} and, 10^{-6} by applying a modified version of the Davidon Fletcher Powell algorithm. . The only information required is the D-H parameters of the robot (in this case the Puma 560) and the desired position and orientation of the end effector.

To determine the level of accuracy of the results, the algorithm utilizes a formula that embodies the error between the desired position and orientation and the position and orientation that

the current iteration yielded. These functions were developed by Wang et al.[21] and basically contain the sum of the differences between the position vectors (desired and current) and the orientation matrices (desired and current).

The electromagnetism-like section of this algorithm was developed by Birbil et al. [20]. This algorithm first develops a series of random points in a six dimensional hyper-plane. These points are treated as charged particles from this moment on and are given magnetic properties to allow the algorithm to work. Once the points have been determined a local search around each point is conducted to verify if any of them can be moved slightly closer to the optimal value. The next step in this algorithm is to compare each point and determine how close they are located with respect to the optimal location; a charge is then assigned to each point depending on its proximity to the optimal. Next, taking into consideration the value of the charge obtained in the previous step, a force with a direction is applied to a particle, this force is generated in such a way that a particle will be attracted to those particles that are closest to the optimal but repelled by those who are in a less favorable position. The resultant force is applied to the particle and a movement across that direction is determined as well. Once the iteration is complete, all particles would have moved closer to the optimal point. This process is repeated until the desired accuracy has been achieved. The algorithm provides the advantage of a wide spectrum search, capable of obtaining a decent result even when no initial guess is available; however, due to its nature it is not suitable for finding results that require a result with small errors ($e = 10^{-2},\ 10^{-4}, 10^{-6}$).

The next section of the algorithm is what the author calls the MDFP or Modified Davidon-Fletcher-Powell. This algorithm is considered "modified" since it differs from the original. Instead of following a procedure to determine a step size for the next iteration, the algorithm simply chooses a random number. The DFP algorithm determines step size using an additional algorithm, this process was considered by Feng. et al. in [4]. However, it was determined that the algorithm would be more computationally efficient if a different approach was made, thus, the MDFP determines this step size λ by selecting a random number between 0 and 1. Further research by Feng et al. determined that as long as this value is smaller than 2 ($\lambda < 2$), is enough for the algorithm to converge. For this reason, the code used for this research uses a constant step size of 0.01. The DFP algorithm, which is described in detail by Nocedal et al. [22], begins by determining a Hessian matrix H, in the case of the first iteration an identity matrix *I* is used. Next, an appropriate step size is selected. In step 3 the gradient vector g is created following the procedure mentioned by Wang et al. [21] and further detailed in [23]. Step 4 determines the search direction d, calculated by $d = -H * g$, and used to determine the next joint angle vector $x = x_k + \lambda * d_k$. Finally the current error is determined by $err = \|g\|$. This process is repeated until the error reaches the desired size, in this case, the precision coefficient ($\boldsymbol{e}$) . The new Hessian for any subsequent iteration is determined by $H_{k+1} = H_k + \frac{p_k * p^T{}_k}{q_k * p^T{}_k} - \frac{H_k * q_k * q^T{}_{k*} H_k}{q^T{}_k * H_k * q_k}$ where p and q are the difference between the current and previous joint angle and gradient vectors, respectively. In summary, the MDFP algorithm consists;

Step 1: receive initial value fi_k, error criteria ε and the step size λ.
Step 2: $H = I$
Step 3: Calculate the gradient $g_k = \nabla f(fi_k)$
Step 4: calculate the search direction $d = -H_k * g_k$

Step 5: set $k = 1$ and use $\|g_k\|$as the error criteria do:
Step 6: $fi_{k+1} = fi_k + \lambda * d_k$
Step 8: $q_k = g_{k+1} - g_k$
Step 9: $p_k = fi_{k+1} - fi_k$
Step 10: $H_{k+1} = H_k + \frac{p_k * p^T{}_k}{q_k * p^T{}_k} - \frac{H_k * q_k * q^T{}_k * H_k}{q^T{}_k * H_k * q_k}$
Step 11: $d_{k+1} = -H_{k+1} * g_{k+1}$
Step 12: $\boldsymbol{e} > \| g_{k+1}\|$?
Step 13: $k = k + 1$
Step 14: repeat from **Step 5** until error criterion ($\boldsymbol{e}$) is achieved.

It is important to understand that this algorithm has an exceptional capability of finding a solution without the need to provide an initial approximation to the real solution. However, the section of the algorithm which is in charge of finding this initial approximation, the Electromagnetism-like algorithm, takes a significant amount of time. To compare the performance of this algorithm with that of other algorithms that do require an initial approximation to be provided first becomes unfair, thus the Electromagnetism-like portion of this algorithm is not addressed in this research and only the MDFP portion of the algorithm is considered.

3.4.2 Newton Raphson Algorithm (e.g. Qu et al. [3])

This algorithm was selected because it possesses a design that works similar to the previous algorithm selected; however it works with an entirely different mechanic. The algorithm creates a transformation matrix that describes the position and orientation of the end effector at a given time called T_{cur}, this can be generated with the Denavit-Hartenberg convection. It then compares this matrix with the matrix that describes the desired position and orientation T_{end} by means of a differential kinematics equation $\Delta = (T_{cur})^{-1} * T_{end} - I$ where I is equal to the

identity matrix of the same dimensions as the transformation matrix. Once the equation has been solved, a differential movement vector is constructed with the components of Δ as seen in equation (76) and (77).

$$\Delta = \begin{bmatrix} 0 & -\delta_z & \delta_y & d_x \\ \delta_z & 0 & -\delta_x & d_y \\ -\delta_y & \delta_x & 0 & d_z \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{76}$$

$$D = \begin{bmatrix} d_x \\ d_y \\ d_z \\ \delta_x \\ \delta_y \\ \delta_z \end{bmatrix} \tag{77}$$

The next step is to determine the Jacobian (J) matrix. This was done by using a method presented by Spong (e.g. Spong et al. [10]). Once the Jacobian matrix has been developed, its pseudo inverse is calculated by using Singular Value Decomposition (SVD), using the pseudo inverse of a matrix helps avoid issues caused by singularity (e.g. Qu et al. [3]). Thus, by using the *s=svd(x)* sub routine embedded in Matlab7.5 the components of the pseudo inverse of the Jacobian (U, Σ, V) were determined. The SVD of the Jacobian will have the form seen in equation (78).

$$J = U\Sigma\, V^* \tag{78}$$

Where U is an *mxm* real or complex unitary matrix and Σ is a diagonal matrix and $V^*$is the conjugate transpose of Vwhich is also a real or complex unitary matrix. The pseudo inverse of the Jacobian is determined by equation (79).

$$J^{+} = V\Sigma^{+}\, U^{*} \tag{79}$$

Where Σ^{+} is the pseudo inverse of Σ and is determined by replacing every non-zero element on Σ by its reciprocal and then transposing the matrix. The next step in the sequence is to determine the joint angle increment as part of the Newton-Raphson iteration, which is determined by the following equation:

$$d\theta = J^{+}(\theta_{cur}) * D \tag{80}$$

Where, $J^{+}(\theta_{cur})$, is the pseudo inverse of the Jacobian for the current joint angle. Now that $d\theta$ has been determined. Next, $\|d\theta\|$ is inspected to verify if is larger than the precision coefficient $(\boldsymbol{e})$, if it is then $d\theta$ is added to the current θ_{cur} to begin a new iteration. In summary, the Newton Raphson Algorithm consists:

Step 1: determine an initial value θ_{cur}, and the error criteria ε
Step 2: determine T_{cur} using DH convection
Step 3: $\Delta = (T_{cur})^{-1} * T_{end} - I$
Step 4 : determine D
Step 5: Determine the Jacobian J
Step 6: $J^{+} = V\Sigma^{+}\, U^{*}$
Step 7: $d\theta = J^{+}(\theta_{cur}) * D$.
Step 8: if $\|d\theta\| > \boldsymbol{e}$, then let $\theta_{cur} = \theta_{cur} + d\theta$and continue from *step 2*, else, exit the iteration.

The Newton-Raphson algorithm operates similarly to the MDFP algorithm, they both iterate to optimize an initial joint angle to a certain precision and they are both able to be implemented for real-time operations. However, the Newton-Raphson algorithm possesses a unique procedure that makes it very different to the MDFP. For this reason, this algorithm was selected for this study, not only will be able to work from the same initial point as the MDFP but it will reach the solution using unique approach.

4 Results

The results are presented in the following tables. The results for the *AMD Turion (tm) 64 X2 2.2GHz* processor are presented in section 3.1 and the *Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz* are presented in section 3.2. There are two tables per page; each page shows the results for each solution (see Table 6), the first table in the page shows the error each algorithm presented for each precision coefficient ($10^{-2}, 10^{-4}, 10^{-6}$) and the second table shows the time for each precision coefficient ($10^{-2}, 10^{-4}, 10^{-6}$). The results for the 10 runs are presented in these tables and the average between the runs is presented in the bottom row. The Standard Deviation was determined for these results to ensure their reliability. It was determined that for both processors, the Standard Deviation for time was 10^{-17} while Standard Deviation for the error was 10^{-2} .

For easier reference, the results for the first solution for each processor are presented in this chapter the remainder of the result tables are located in the appendix section.

Table 6 Initial input parameters for the algorithms for each solution set

	Solution							
Joint Angle (˚)	1	2	3	4	5	6	7	8
θ_1	10	10	10	-140	-140	-140	-140	10
θ_2	20	20	-30	-140	-140	150	150	-30
θ_3	30	30	140	30	30	140	140	140
θ_4	40	-130	-70	-130	-40	20	20	100
θ_5	50	-50	-30	-10	10	-40	-40	30
θ_6	60	-110	160	140	-30	-90	-90	10

4.0.0 AMD Turion (tm) 64 X2 2.2GHz

4.0.1 Results for solution 1

Table 7 Solution 1 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.04900	0.20020	0.04900	0.00100	0.01220	0.00086
2	0.04900	0.20020	0.04900	0.00100	0.01220	0.00086
3	0.04900	0.20020	0.04900	0.00100	0.01220	0.00086
4	0.04900	0.20020	0.04900	0.00100	0.01220	0.00086
5	0.04900	0.20020	0.04900	0.00100	0.01220	0.00086
6	0.04900	0.20020	0.04900	0.00100	0.01220	0.00086
7	0.04900	0.20020	0.04900	0.00100	0.01220	0.00086
8	0.04900	0.20020	0.04900	0.00100	0.01220	0.00086
9	0.04900	0.20020	0.04900	0.00100	0.01220	0.00086
10	0.04900	0.20020	0.04900	0.00100	0.01220	0.00086
Average	**0.04900**	**0.20020**	**0.04900**	**0.00100**	**0.01220**	**0.00086**

Table 8 Solution 1 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.27370	0.21680	0.27480	1.64660	0.76610	2.53360
2	0.22630	0.17210	0.22630	1.60440	0.71270	2.4800
3	0.22410	0.17090	0.22640	1.60260	0.70940	2.47620
4	0.22340	0.17040	0.22570	1.60000	0.70980	2.47920
5	0.22280	0.17170	0.22440	1.60010	0.71030	2.47450
6	0.22300	0.17150	0.22420	1.59940	0.71100	2.47190
7	0.22330	0.17080	0.22430	1.59920	0.71100	2.46740
8	0.22280	0.17090	0.22430	1.59990	0.71190	2.47000
9	0.22310	0.17140	0.22430	1.59350	0.71070	2.46910
10	0.22340	0.17090	0.22470	1.59820	0.71190	2.46640
Average	**0.22859**	**0.17574**	**0.22994**	**1.60439**	**0.71648**	**2.47883**

4.1.0 Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz

4.1.1 Results for solution 1

Table 9 Solution 1 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.11580	0.20020	0.05770	0.00100	0.00980	0.00086
2	0.11580	0.20020	0.05770	0.00100	0.00980	0.00086
3	0.11580	0.20020	0.05770	0.00100	0.00980	0.00086
4	0.11580	0.20020	0.05770	0.00100	0.00980	0.00086
5	0.11580	0.20020	0.05770	0.00100	0.00980	0.00086
6	0.11580	0.20020	0.05770	0.00100	0.00980	0.00086
7	0.11580	0.20020	0.05770	0.00100	0.00980	0.00086
8	0.11580	0.20020	0.05770	0.00100	0.00980	0.00086
9	0.11580	0.20020	0.05770	0.00100	0.00980	0.00086
10	0.11580	0.20020	0.05770	0.00100	0.00980	0.00086
Average	**0.11580**	**0.20020**	**0.05770**	**0.00100**	**0.00980**	**0.00086**

Table 10 Solution 1 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.08460	0.76040	0.45190	2.50460	1.88700	3.82850
2	0.01210	0.26400	0.37980	2.43730	1.80710	3.76000
3	0.01190	0.26130	0.37870	2.43470	1.80630	3.76350
4	0.01210	0.26130	0.37820	2.43430	1.80670	3.76220
5	0.01200	0.26120	0.37710	2.43450	1.80640	3.76180
6	0.01200	0.26140	0.37690	2.43410	1.80730	3.75980
7	0.01200	0.26100	0.37690	2.43410	1.80710	3.76350
8	0.01210	0.26120	0.37730	2.43440	1.80690	3.76350
9	0.01200	0.26100	0.37670	2.43540	1.80730	3.76450
10	0.01190	0.26130	0.37680	2.43560	1.80700	3.76200
Average	**0.01927**	**0.31141**	**0.38503**	**2.44190**	**1.81491**	**3.76893**

4.2.0 Joint angle solutions

The following are the results for the optimized joint angle solutions generated after the algorithm finished the optimization routine on the initial approximation. As mentioned before, the algorithms were given an initial approximation value for each of the 8 solutions. These are shown below:

$$\theta_{1\ initial} = \begin{bmatrix} 10 \\ 20 \\ 30 \\ 40 \\ 50 \\ 60 \end{bmatrix} \qquad \theta_{5\ initial} = \begin{bmatrix} -140 \\ -140 \\ 30 \\ -40 \\ 10 \\ -30 \end{bmatrix}$$

$$\theta_{2\ initial} = \begin{bmatrix} 10 \\ 20 \\ 30 \\ -130 \\ -50 \\ -110 \end{bmatrix} \qquad \theta_{6\ initial} = \begin{bmatrix} -140 \\ 150 \\ 140 \\ 20 \\ -40 \\ -90 \end{bmatrix}$$

$$\theta_{3\ initial} = \begin{bmatrix} 10 \\ -30 \\ 140 \\ -70 \\ -30 \\ 160 \end{bmatrix} \qquad \theta_{7\ initial} = \begin{bmatrix} -140 \\ 150 \\ 140 \\ -150 \\ 40 \\ 80 \end{bmatrix}$$

$$\theta_{4\ initial} = \begin{bmatrix} -140 \\ -140 \\ 30 \\ 130 \\ -10 \\ 140 \end{bmatrix} \qquad \theta_{8\ initial} = \begin{bmatrix} 10 \\ -30 \\ 140 \\ 100 \\ 30 \\ -10 \end{bmatrix}$$

There are three columns per page representing each precision coefficient ($10^{-2}, 10^{-4}, 10^{-6}$). The first two sets represent the solutions for the Newton Raphson and the MDFP algorithms using the **AMD Turion (tm) 64 X2 2.2GHz.** The second pair of sets of are the results for the Newton Raphson and the MDFP algorithms using the **Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz** processor.

4.2.1 MDFP Joint angle results for the AMD Turion (tm) 64 X2 2.2GHz processor

10^{-2}

$$\theta_{1\ optimized} = \begin{bmatrix} 15.1324 \\ 25.1015 \\ 35.0041 \\ 43.7446 \\ 52.6528 \\ 64.2661 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 15.0357 \\ 25.1635 \\ 34.7467 \\ -135.7471 \\ -54.0171 \\ -116.0703 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 15.2514 \\ -30.4102 \\ 145.9542 \\ -70.0759 \\ -33.1626 \\ 161.5462 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -142.6678 \\ -149.5235 \\ 34.3079 \\ 135.7301 \\ -15.5320 \\ 145.5580 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -142.6934 \\ -149.8324 \\ 34.9494 \\ -45.4322 \\ 15.2814 \\ -35.6029 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.7706 \\ 154.3264 \\ 145.8375 \\ 18.9552 \\ -41.4148 \\ -91.6543 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -143.1443 \\ 154.7558 \\ 145.0109 \\ -153.9959 \\ 43.0342 \\ 78.9234 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 17.4971 \\ -27.1901 \\ 142.2824 \\ 102.7131 \\ 31.0158 \\ -9.9814 \end{bmatrix}$$

10^{-4}

$$\theta_{1\ optimized} = \begin{bmatrix} 15.1324 \\ 25.1015 \\ 35.0041 \\ 43.7446 \\ 52.6528 \\ 64.2661 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 15.0357 \\ 25.1635 \\ 34.7467 \\ -135.7471 \\ -54.0171 \\ -116.0703 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 15.2514 \\ -30.4102 \\ 145.9542 \\ -70.0759 \\ -33.1626 \\ 11.5462 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -142.6678 \\ -149.5235 \\ 34.3079 \\ 135.7301 \\ -15.5320 \\ 145.5580 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -142.6934 \\ -149.8324 \\ 34.9494 \\ -45.4322 \\ 15.2814 \\ -35.6029 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.7706 \\ 154.3264 \\ 145.8375 \\ 18.9552 \\ -41.4148 \\ -91.6543 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -143.1443 \\ 154.7558 \\ 145.0109 \\ -153.9959 \\ 43.0342 \\ 78.9234 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 15.2577 \\ -30.2755 \\ 145.1779 \\ 102.4782 \\ 32.3088 \\ -10.0194 \end{bmatrix}$$

10^{-6}

$$\theta_{1\ optimized} = \begin{bmatrix} 15.0355 \\ 25.0546 \\ 34.9009 \\ 44.4101 \\ 54.5181 \\ 65.2051 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 15.0121 \\ 25.0200 \\ 34.9570 \\ -135.2457 \\ -54.8533 \\ -114.9369 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 14.9374 \\ -29.9508 \\ 144.9055 \\ -77.0918 \\ -37.5752 \\ 168.8124 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -142.8101 \\ -149.9143 \\ 34.8147 \\ 134.5210 \\ -17.8536 \\ 144.6348 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -142.8097 \\ -149.9127 \\ 34.8098 \\ -45.4162 \\ 17.8603 \\ -35.2721 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.8291 \\ 154.9664 \\ 145.1150 \\ 18.5356 \\ -44.6395 \\ -92.7725 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -142.9714 \\ 155.0392 \\ 144.8955 \\ -158.3674 \\ 44.6794 \\ 84.9614 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 14.9551 \\ -29.9857 \\ 144.8770 \\ 101.9362 \\ 37.0962 \\ -10.6955 \end{bmatrix}$$

4.2.2 Newton Raphson joint angle results for the AMD Turion (tm) 64 X2 2.2GHz processor

10^{-2}

$$\theta_{1\ optimized} = \begin{bmatrix} 11.1830 \\ 23.8498 \\ 17.5135 \\ 39.1950 \\ 65.2405 \\ 73.4338 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 6.6029 \\ 23.6522 \\ 20.2792 \\ -132.1849 \\ -65.0274 \\ -106.7724 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 9.1391 \\ -36.9069 \\ 150.7935 \\ -75.9241 \\ -37.6114 \\ 165.6296 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -141.0444 \\ -147.7725 \\ 32.5856 \\ 135.3894 \\ -17.1742 \\ 143.7584 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -145.2585 \\ -150.8162 \\ 36.5668 \\ -55.3935 \\ 19.4684 \\ -25.4410 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.5133 \\ 154.4789 \\ 133.6499 \\ 24.6007 \\ -36.1784 \\ -97.5833 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -140.5797 \\ 154.1312 \\ 138.4885 \\ -157.1103 \\ 38.5794 \\ 83.8007 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 8.8399 \\ -32.2259 \\ 145.2406 \\ 101.0362 \\ 37.2389 \\ -9.9971 \end{bmatrix}$$

10^{-4}

$$\theta_{1\ optimized} = \begin{bmatrix} 15.0195 \\ 25.0360 \\ 34.8468 \\ 44.9214 \\ 55.0854 \\ 65.0913 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 15.0084 \\ 25.0358 \\ 34.8472 \\ -135.0672 \\ -55.0919 \\ -114.9163 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 15.0036 \\ -30.1301 \\ 145.1508 \\ -76.6285 \\ -36.5389 \\ 168.3224 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -142.8474 \\ -149.8909 \\ 34.8731 \\ 130.5527 \\ -18.4200 \\ 148.6365 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -142.8794 \\ -149.9158 \\ 34.9042 \\ -49.5304 \\ 18.4382 \\ -31.2869 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.8676 \\ 154.9764 \\ 145.0643 \\ 20.0149 \\ -44.6258 \\ -93.8527 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -142.8496 \\ 154.9747 \\ 145.0620 \\ -160.0037 \\ 44.6175 \\ 86.1637 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 14.9844 \\ -30.1156 \\ 145.1324 \\ 103.3537 \\ 36.5528 \\ -11.6642 \end{bmatrix}$$

10^{-6}

$$\theta_{1\ optimized} = \begin{bmatrix} 15.0245 \\ 25.0287 \\ 34.8916 \\ 44.9357 \\ 55.0574 \\ 65.0637 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 15.0244 \\ 25.0286 \\ 34.8916 \\ -135.0642 \\ -55.0575 \\ -114.9364 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 15.0244 \\ -30.0936 \\ 145.1084 \\ -76.6254 \\ -36.5216 \\ 168.3288 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -142.8621 \\ -149.9066 \\ 34.8918 \\ 130.5130 \\ -18.4287 \\ 148.6736 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -142.8624 \\ -149.9068 \\ 34.8921 \\ -49.4878 \\ 18.4289 \\ -31.3257 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.8623 \\ 154.9715 \\ 145.1076 \\ 19.9947 \\ -44.6590 \\ -93.8291 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -142.8622 \\ 154.9715 \\ 145.1075 \\ -160.0055 \\ 44.6589 \\ 86.1710 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 15.0242 \\ -30.0934 \\ 145.1082 \\ 103.3744 \\ 36.5217 \\ -11.6711 \end{bmatrix}$$

4.2.3 MDFP joint angle results for the Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz processor

10^{-2}

$$\theta_{1\ optimized} = \begin{bmatrix} 15.0939 \\ 27.6808 \\ 34.6478 \\ 41.0594 \\ 51.9589 \\ 61.3032 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 15.0076 \\ 28.5424 \\ 34.8529 \\ -130.5704 \\ -52.0958 \\ -110.1656 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 17.3689 \\ -26.8283 \\ 142.6039 \\ -70.3141 \\ -31.0774 \\ 160.2622 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -141.1470 \\ -143.4965 \\ 28.1875 \\ 130.3036 \\ -9.8507 \\ 140.3304 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -140.9404 \\ -143.3624 \\ 28.2814 \\ -40.2394 \\ 9.8433 \\ -30.2240 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -143.3150 \\ 158.0269 \\ 144.8250 \\ 19.9047 \\ -39.3771 \\ -90.1168 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -142.1453 \\ 154.2198 \\ 142.5222 \\ -150.0747 \\ 39.6554 \\ 79.9599 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 8.4379 \\ -33.1279 \\ 145.2001 \\ 100.0372 \\ 36.9323 \\ -9.9971 \end{bmatrix}$$

10^{-4}

$$\theta_{1\ optimized} = \begin{bmatrix} 15.1390 \\ 25.1610 \\ 34.7919 \\ 43.4802 \\ 52.5705 \\ 63.9616 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 14.9760 \\ 25.5556 \\ 34.2469 \\ -133.5570 \\ -53.0589 \\ -113.6444 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 15.2911 \\ -30.3768 \\ 145.9171 \\ -70.0802 \\ -33.1411 \\ 161.5308 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -142.6287 \\ -149.5741 \\ 34.3884 \\ 136.3000 \\ -15.6137 \\ 146.1062 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -142.7371 \\ -150.1133 \\ 35.3414 \\ -46.0068 \\ 15.6815 \\ -36.1836 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.7985 \\ 154.5280 \\ 145.4296 \\ 19.0971 \\ -41.1260 \\ -91.4152 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -143.1887 \\ 154.7576 \\ 144.9727 \\ -153.5203 \\ 42.6057 \\ 79.0982 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 15.0024 \\ -29.786 \\ 145.5373 \\ 102.9735 \\ 35.9538 \\ -11.5821 \end{bmatrix}$$

10^{-6}

$$\theta_{1\ optimized} = \begin{bmatrix} 15.0299 \\ 25.0292 \\ 34.9284 \\ 44.4098 \\ 54.7236 \\ 65.2338 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 15.0154 \\ 25.0422 \\ 34.9336 \\ -135.2091 \\ -54.6847 \\ -114.9641 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 14.9376 \\ -29.9505 \\ 144.9044 \\ -77.1097 \\ -37.5747 \\ 168.7359 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -142.8114 \\ -149.9183 \\ 34.8129 \\ 134.6499 \\ -17.9182 \\ 144.7652 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -142.8095 \\ -149.9131 \\ 34.8110 \\ -45.4197 \\ 17.8599 \\ -35.2667 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.8276 \\ 154.9651 \\ 145.1126 \\ 18.5197 \\ -44.5943 \\ -92.7592 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -142.9735 \\ 155.0531 \\ 144.8869 \\ -158.3626 \\ 44.8254 \\ 84.9975 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 15.0051 \\ -30.0997 \\ 145.1007 \\ 103.5641 \\ 36.0003 \\ -11.1727 \end{bmatrix}$$

4.2.4 Newton Raphson Joint angle results for the Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz processor

10^{-2}

$$\theta_{1\ optimized} = \begin{bmatrix} 11.1830 \\ 23.8498 \\ 17.5135 \\ 39.1950 \\ 65.2405 \\ 73.4338 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 6.6029 \\ 23.6522 \\ 20.2792 \\ -132.1849 \\ -65.0274 \\ -106.7724 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 9.1391 \\ -36.9069 \\ 150.7935 \\ -75.9241 \\ -37.6114 \\ 165.6296 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -141.0444 \\ -147.7725 \\ 32.5856 \\ 135.3894 \\ -17.1742 \\ 143.7584 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -145.2585 \\ -150.8162 \\ 36.5668 \\ -55.3935 \\ 19.4684 \\ -25.4410 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.5133 \\ 154.4789 \\ 133.6499 \\ 24.6007 \\ -36.1784 \\ -97.5833 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -140.5797 \\ 154.1312 \\ 138.4885 \\ -157.1103 \\ 38.5794 \\ 83.8007 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 8.6703 \\ -32.9284 \\ 144.2071 \\ 101.0381 \\ 36.2323 \\ -10.0341 \end{bmatrix}$$

10^{-4}

$$\theta_{1\ optimized} = \begin{bmatrix} 15.0195 \\ 25.0360 \\ 34.8468 \\ 44.9214 \\ 55.0854 \\ 65.0913 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 15.0084 \\ 25.0358 \\ 34.8472 \\ -135.0672 \\ -55.0919 \\ -114.9163 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 15.0036 \\ -30.1301 \\ 145.1508 \\ -76.6285 \\ -36.5389 \\ 168.3224 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -142.8474 \\ -149.8909 \\ 34.8731 \\ 130.5527 \\ -18.4200 \\ 148.6365 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -142.8794 \\ -149.9158 \\ 34.9042 \\ -49.5304 \\ 18.4382 \\ -31.2869 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.8676 \\ 154.9764 \\ 145.0643 \\ 20.0149 \\ -44.6258 \\ -93.8527 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -142.8496 \\ 154.9747 \\ 145.0620 \\ -160.0037 \\ 44.6175 \\ 86.1637 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 14.9954 \\ -30.1264 \\ 145.1563 \\ 103.3647 \\ 36.6710 \\ -11.7632 \end{bmatrix}$$

10^{-6}

$$\theta_{1\ optimized} = \begin{bmatrix} 15.0245 \\ 25.0287 \\ 34.8916 \\ 44.9357 \\ 55.0574 \\ 65.0637 \end{bmatrix}$$

$$\theta_{2\ optimized} = \begin{bmatrix} 15.0244 \\ 25.0286 \\ 34.8916 \\ -135.0642 \\ -55.0575 \\ -114.9364 \end{bmatrix}$$

$$\theta_{3\ optimized} = \begin{bmatrix} 15.0244 \\ -30.0936 \\ 145.1084 \\ -76.6254 \\ -36.5216 \\ 168.3288 \end{bmatrix}$$

$$\theta_{4\ optimized} = \begin{bmatrix} -142.8621 \\ -149.9066 \\ 34.8918 \\ 130.5130 \\ -18.4287 \\ 148.6736 \end{bmatrix}$$

$$\theta_{5\ optimized} = \begin{bmatrix} -142.8624 \\ -149.9068 \\ 34.8921 \\ -49.4878 \\ 18.4289 \\ -31.3257 \end{bmatrix}$$

$$\theta_{6\ optimized} = \begin{bmatrix} -142.8623 \\ 154.9715 \\ 145.1076 \\ 19.9947 \\ -44.6590 \\ -93.8291 \end{bmatrix}$$

$$\theta_{7\ optimized} = \begin{bmatrix} -142.8622 \\ 154.9715 \\ 145.1075 \\ -160.0055 \\ 44.6589 \\ 86.1710 \end{bmatrix}$$

$$\theta_{8\ optimized} = \begin{bmatrix} 15.01262 \\ -30.1054 \\ 145.1079 \\ 103.7240 \\ 36.5217 \\ -11.6356 \end{bmatrix}$$

5 Discussion of the results

In this chapter the results of this investigation are analyzed. The results for the **AMD Turion (tm) 64 X2 2.2GHz** and **Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz** are presented and analyzed. Two graphs have been created for each solution, one compares the error results between the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms and the other compares the time results for both algorithms for each precision coefficient, where; **e-2 = 10^{-2}**, **e-4 = 10^{-4}** and **e-6 = 10^{-6}**. These are presented in the following pages for clarity and ease of comparison, followed by a summary of these results. At the end of the chapter, a table summarizing the conclusions of each solution is presented.

5.0.0 Solution 1

For the first solution, using the AMD processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.04900 for 10^{-2}, 0.04900 for 10^{-4}, and 0.01220 for 10^{-6}, while the NR Algorithm yielded 0.20020 for 10^{-2}, 0.000100 for 10^{-4}, and 0.00086 for 10^{-6}. Also, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.22859s for 10^{-2}, 0.22994s for 10^{-4} and 0.71648s for 10^{-6}, while the NR algorithm measured 0.17574s for 10^{-2}, 1.60439s for $10^{-4}$and 2.47883s for 10^{-6}.

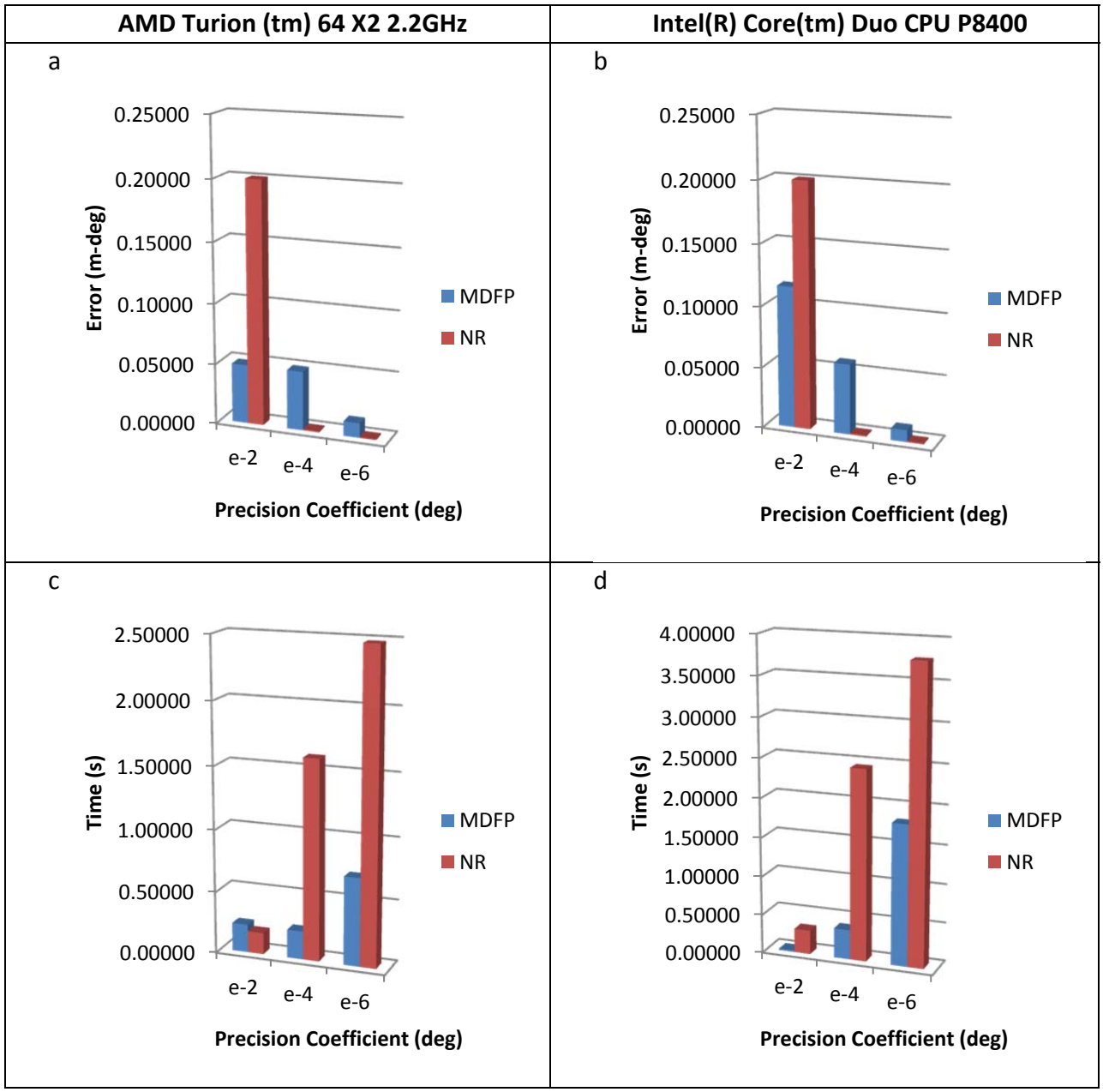


Figure 14 Results for Time and Error for Solution 1 on the AMD and Intel Processors. a) AMD Error results, b) Intel Error results, c) AMD Time results, d) AMD Time results (Where; e-2=10^{-2}, e-4=10^{-4}, e-6=10^{-6})

On the other hand, using the Intel processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.11580 for 10^{-2}, 0.05770 for 10^{-4}, and 0.00980 for 10^{-6}, while the NR Algorithm yielded 0.20020 for 10^{-2}, 0.00010 for 10^{-4}, and 0.00086 for 10^{-6}.In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.01927s for 10^{-2}, 0.38503s for 10^{-4} and 1.81491s for 10^{-6}, while the NR algorithm measured 0.31141s for 10^{-2}, 2.44190s for $10^{-4}$and 3.76893s for 10^{-6}.

Both algorithms followed a pattern in which for the 10^{-4} and 10^{-6} precision coefficients the NR algorithm took longer to reach a result than the MDFP, however this result was more accurate. This behavior was observed using both processors. For the 10^{-2} precision coefficient the result for NR was a less accurate than the MDFP for both processors. It is important to understand that the results found using the INTEL processor are similar to those found in the AMD processor, however all the calculation times were increased on the INTEL processor.

5.1.0 Solution 2

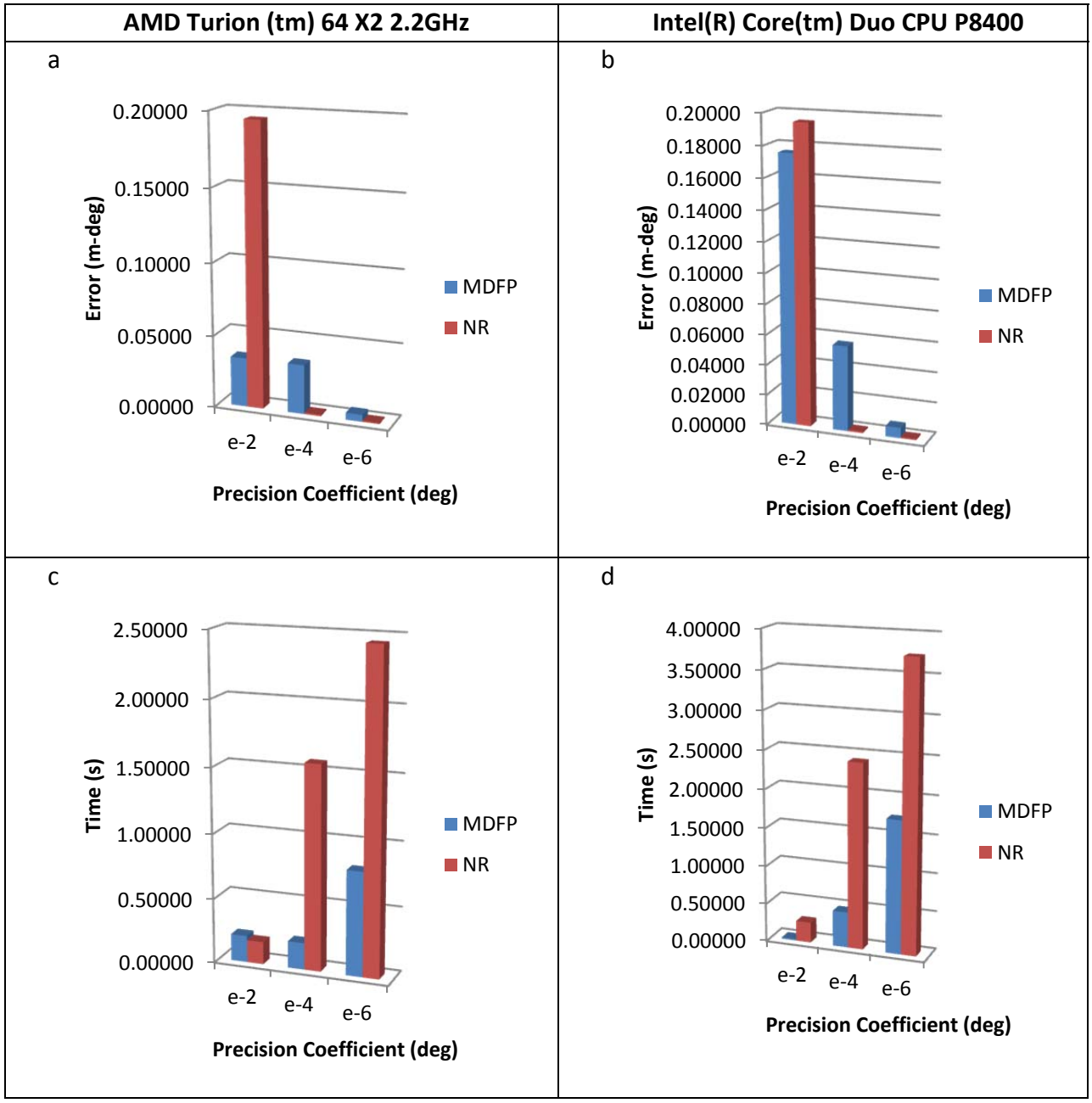


Figure 15 Results for Time and Error for Solution 2 on the AMD and Intel Processors. a) AMD Error results, b) Intel Error results, c) AMD Time results, d) AMD Time results (Where; e-2=10^{-2}, e-4=10^{-4}, e-6=10^{-6})

For the second solution, using the AMD processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.0341 for 10^{-2}, 0.03410 for 10^{-4}, and 0.00480 for 10^{-6}, while the NR

Algorithm yielded 0.19460 for 10^{-2}, 0.00094 for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.20698s for 10^{-2}, 0.20697s for 10^{-4} and 0.80878s for 10^{-6}, while the NR algorithm measured 0.17401s for 10^{-2}, 1.57015s for $10^{-4}$and 2.44554s for 10^{-6}.

For the Intel processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.17550 for 10^{-2}, 0.05640 for 10^{-4}, and 0.00690 for 10^{-6}, while the NR Algorithm yielded 0.19460 for 10^{-2}, 0.00094 for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.01925s for 10^{-2}, 0.46856s for 10^{-4} and 1.174237s for 10^{-6}, while the NR algorithm measured 0.26684s for 10^{-2}, 2.41461s for $10^{-4}$and 3.74195s for 10^{-6}.

Both algorithms followed a pattern in which for the 10^{-4} and 10^{-6} precision coefficients the NR algorithm took longer to reach a result than the MDFP, however this result was more accurate. This behavior was observed using both processors. For the 10^{-2} precision coefficient the result for NR was a less accurate than the MDFP for both processors. It is important to understand that the results found using the INTEL processor are similar to those found in the AMD processor, however all the calculation times were increased on the INTEL processor.

5.2.0 Solution 3

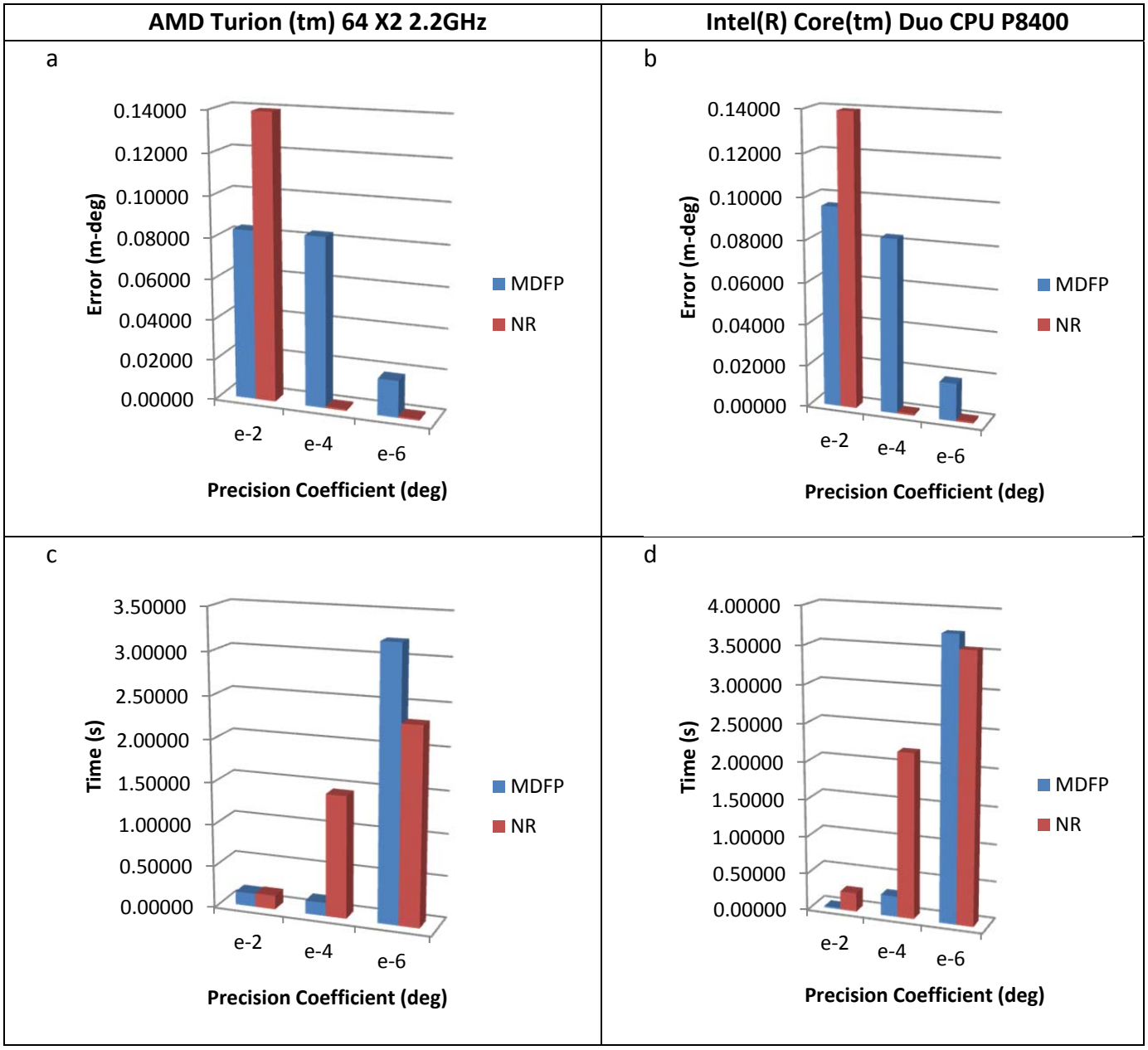


Figure 16 Results for Time and Error for Solution 3 on the AMD and Intel Processors. a) AMD Error results, b) Intel Error results, c) AMD Time results, d) AMD Time results (Where; e-2=10^{-2}, e-4=10^{-4}, e-6=10^{-6})

For the third solution, on the AMD processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.08330 for 10^{-2}, 0.08330 for 10^{-4}, and 0.01810 for 10^{-6}, while the NR Algorithm yielded 0.13930 for 10^{-2}, 0.00091 for 10^{-4}, and 0.00086 for 10^{-6}.

In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.15838s for 10^{-2}, 0.15882s for 10^{-4} and 3.20018s for 10^{-6}, while the NR algorithm measured 0.16665s for 10^{-2}, 1.44963s for $10^{-4}$and 2.32180s for 10^{-6}.

For the Intel processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.09570 for 10^{-2}, 0.08310 for 10^{-4}, and 0.01810 for 10^{-6}, while the NR Algorithm yielded 0.13930 for 10^{-2}, 0.00091 for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.01932s for 10^{-2}, 0.27931s for 10^{-4} and 3.73011s for 10^{-6}, while the NR algorithm measured 0.25261s for 10^{-2}, 2.20619s for $10^{-4}$and 3.54471s for 10^{-6}.

Both algorithms followed a pattern in which for the 10^{-4} and 10^{-6} precision coefficients where the result for the NR algorithm was more accurate, however NR took less time to reach this result for the 10^{-6} coefficient when compared to the MDFP. This behavior was observed using both processors. For the 10^{-2} precision coefficient the result for NR was a less accurate than the MDFP for both processors. It is important to understand that the results found using the INTEL processor are similar to those found in the AMD processor, however all the calculation times were increased on the INTEL processor.

5.3.0 Solution 4

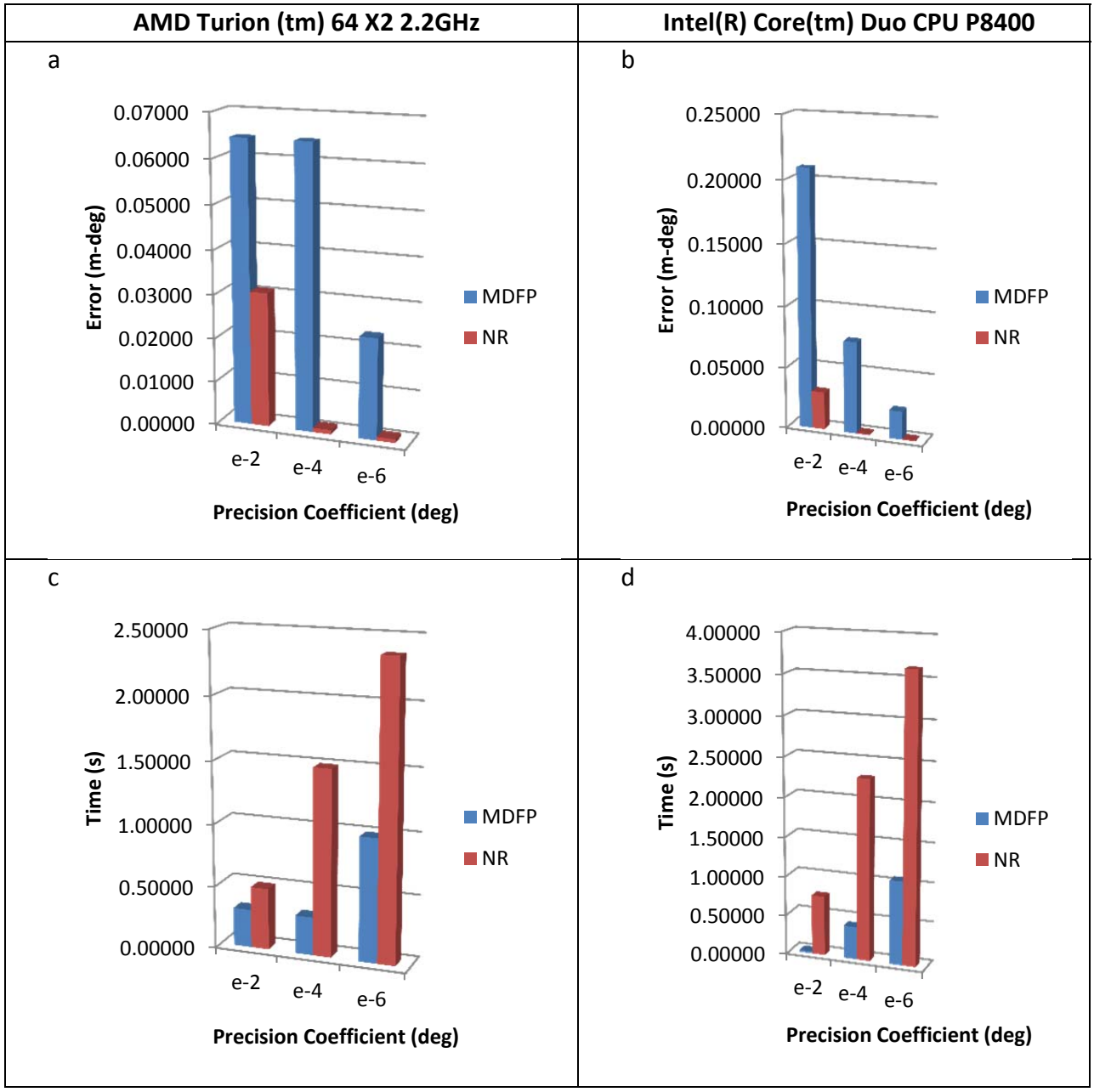


Figure 17 Results for Time and Error for Solution 4 on the AMD and Intel Processors. a) AMD Error results, b) Intel Error results, c) AMD Time results, d) AMD Time results (Where; e-2=10^{-2}, e-4=10^{-4}, e-6=10^{-6})

For the fourth solution, on the AMD processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.06450 for 10^{-2}, 0.06450 for 10^{-4}, and 0.02320 for 10^{-6}, while the NR

Algorithm yielded 0.03070 for 10^{-2}, 0.00100 for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.30653s for 10^{-2}, 0.30669s for 10^{-4} and 0.99297s for 10^{-6}, while the NR algorithm measured 0.49386s for 10^{-2}, 1.149241s for $10^{-4}$and 2.36219s for 10^{-6}.

Using the Intel processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.20940 for 10^{-2}, 0.0752 for 10^{-4}, and 0.0230 for 10^{-6}, while the NR Algorithm yielded 0.03070 for 10^{-2}, 0.00100for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.02321s for 10^{-2}, 0.41623s for 10^{-4} and 1.06661s for 10^{-6}, while the NR algorithm measured 0.75675s for 10^{-2}, 2.29193s for $10^{-4}$and 3.63059s for 10^{-6}.

Both algorithms followed a pattern in which for the 10^{-2} , 10^{-4} and 10^{-6} precision coefficients the NR algorithm took longer to reach a result than the MDFP, however this result was more accurate. This behavior was observed using both processors. It is important to understand that the results found using the INTEL processor are similar to those found in the AMD processor, however all the calculation times were increased on the INTEL processor.

5.4.0 Solution 5

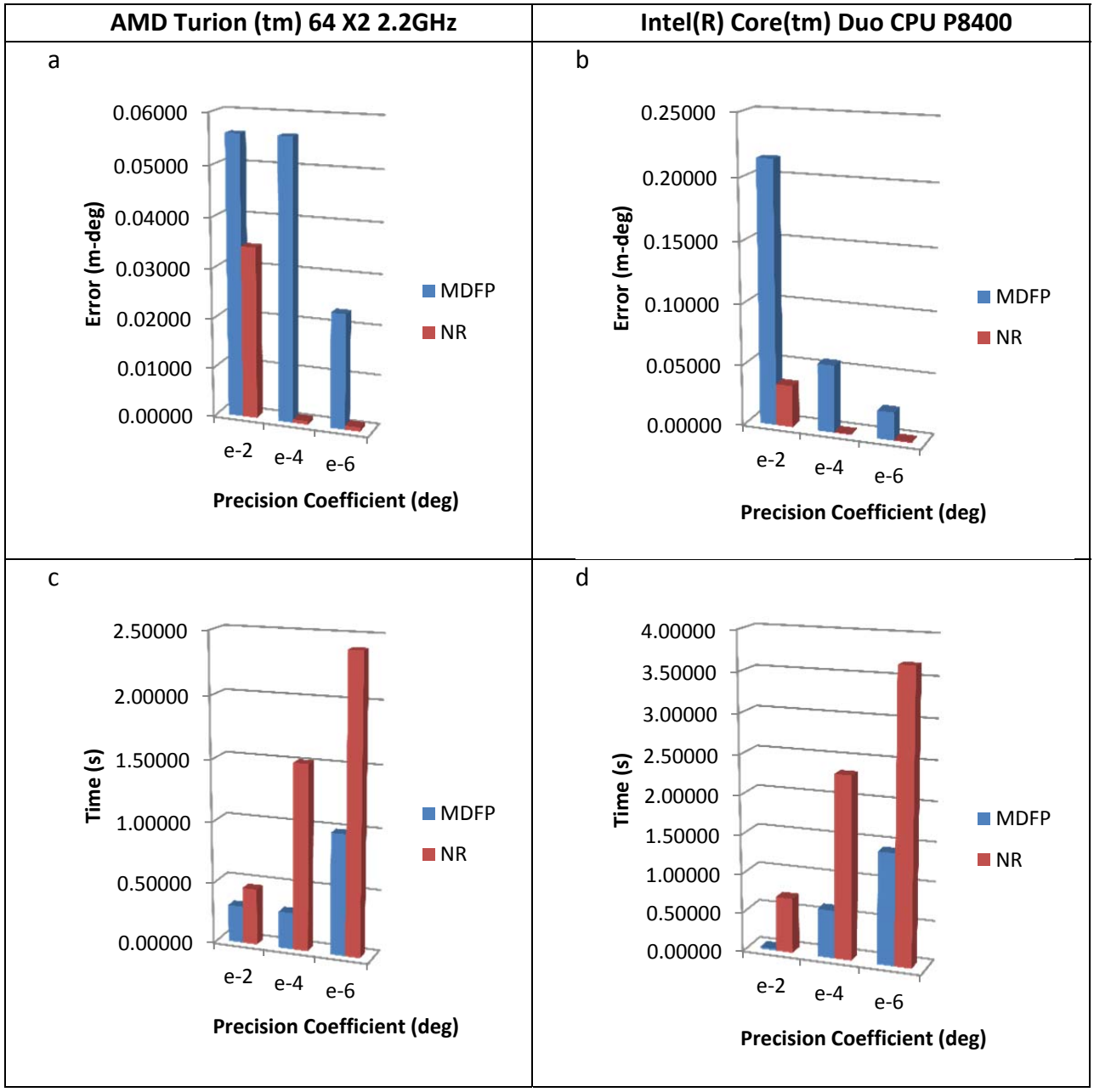


Figure 18 Results for Time and Error for Solution 5 on the AMD and Intel Processors. a) AMD Error results, b) Intel Error results, c) AMD Time results, d) AMD Time results (Where; e-2=10^{-2}, e-4=10^{-4}, e-6=10^{-6})

For the fifth solution, on the AMD processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The

MDFP yielded 0.05600 for 10^{-2}, 0.05600 for 10^{-4}, and 0.002310 for 10^{-6}, while the NR Algorithm yielded 0.03440 for 10^{-2}, 0.00072 for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.30245s for 10^{-2}, 0.30091s for 10^{-4} and 0.98834s for 10^{-6}, while the NR algorithm measured 0.45847s for 10^{-2}, 1.51178s for $10^{-4}$and 2.39778s for 10^{-6}.

For the Intel processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.21470 for 10^{-2}, 0.05550 for 10^{-4}, and 0.02310 for 10^{-6}, while the NR Algorithm yielded 0.03440 for 10^{-2}, 0.00072 for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.02330s for 10^{-2}, 0.61542s for 10^{-4} and 1.42671s for 10^{-6}, while the NR algorithm measured 0.70333s for 10^{-2}, 2.32599s for $10^{-4}$and 3.67206s for 10^{-6}.

Both algorithms followed a pattern in which for the 10^{-2} , 10^{-4} and 10^{-6} precision coefficients the NR algorithm took longer to reach a result than the MDFP, however this result was more accurate. This behavior was observed using both processors. It is important to understand that the results found using the INTEL processor are similar to those found in the AMD processor, however all the calculation times were increased on the INTEL processor.

5.5.0 Solution 6

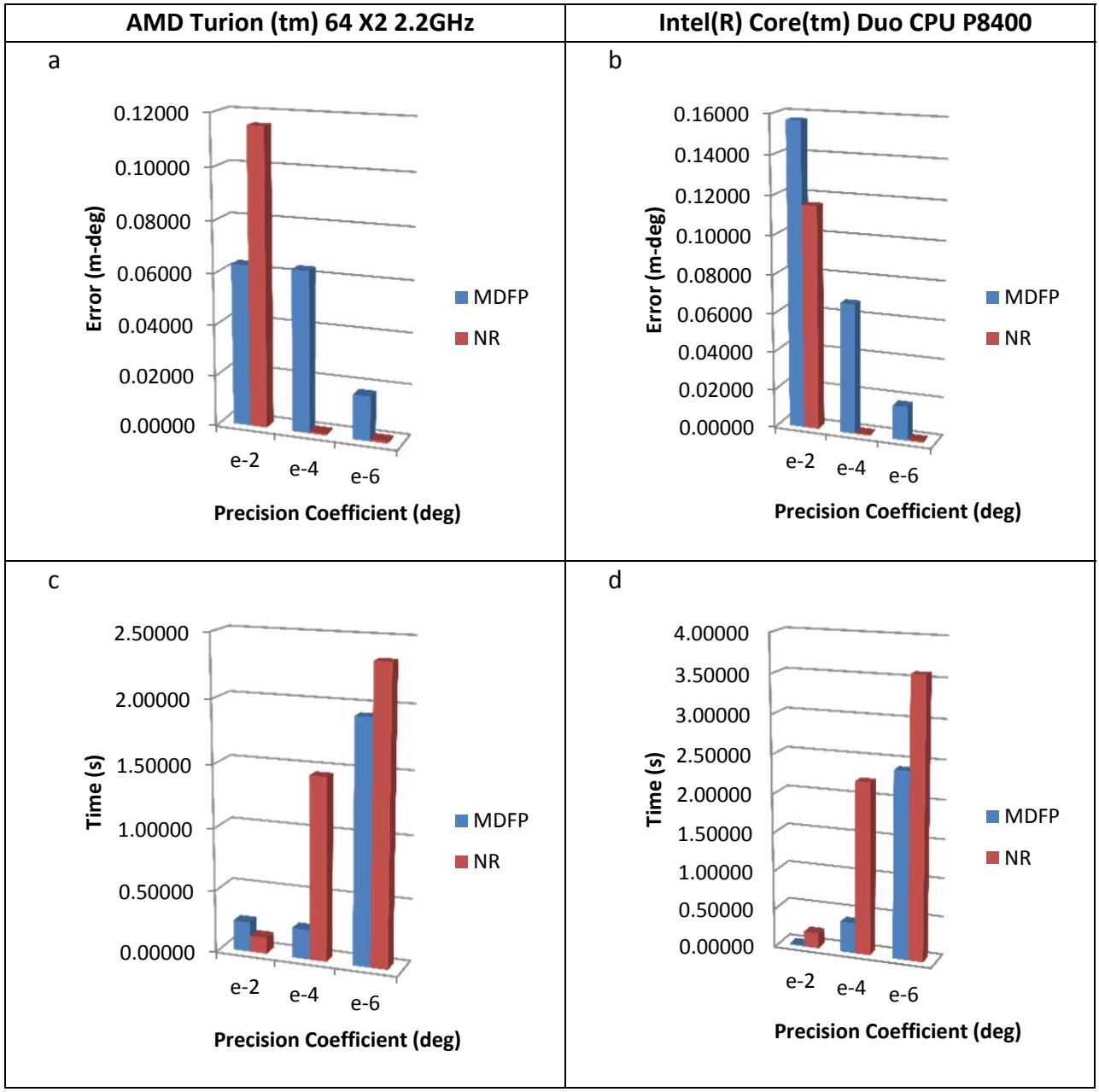


Figure 19 Results for Time and Error for Solution 6 on the AMD and Intel Processors. a) AMD Error results, b) Intel Error results, c) AMD Time results, d) AMD Time results (Where; e-2=10^{-2}, e-4=10^{-4}, e-6=10^{-6})

For the sixth solution, on the AMD processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.06320 for 10^{-2}, 0.06320 for 10^{-4}, and 0.01760 for 10^{-6}, while the NR

Algorithm yielded 0.11550 for 10^{-2}, 0.00072 for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.23961s for 10^{-2}, 0.23920s for 10^{-4} and 0.192892s for 10^{-6}, while the NR algorithm measured 0.13203s for 10^{-2}, 1.45471s for 10^{-4} and 2.33797s for 10^{-6}.

For Intel processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.15680 for 10^{-2}, 0.06760 for 10^{-4}, and 0.01780 for 10^{-6}, while the NR Algorithm yielded 0.11550 for 10^{-2}, 0.00072 for 10^{-4}, and 0.00086 for 10^{-6}.In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.01929s for 10^{-2}, 0.39713s for 10^{-4} and 2.41053s for 10^{-6}, while the NR algorithm measured 0.20163s for 10^{-2}, 2.2748s for $10^{-4}$and 3.58041s for 10^{-6}.

On the AMD processor both algorithms followed a pattern in which for the 10^{-4} and 10^{-6} precision coefficients the result for the NR algorithm was more accurate than the MDFP yet, it took longer to reach this result. On the other hand, this behavior was observed for all precision coefficients using the Intel processor. For the 10^{-2} precision coefficient the result for NR was less accurate than the MDFP on the AMD processor. It is important to understand that the results found using the INTEL processor are similar to those found in the AMD processor, however all the calculation times were increased on the INTEL processor.

5.6.0 Solution 7

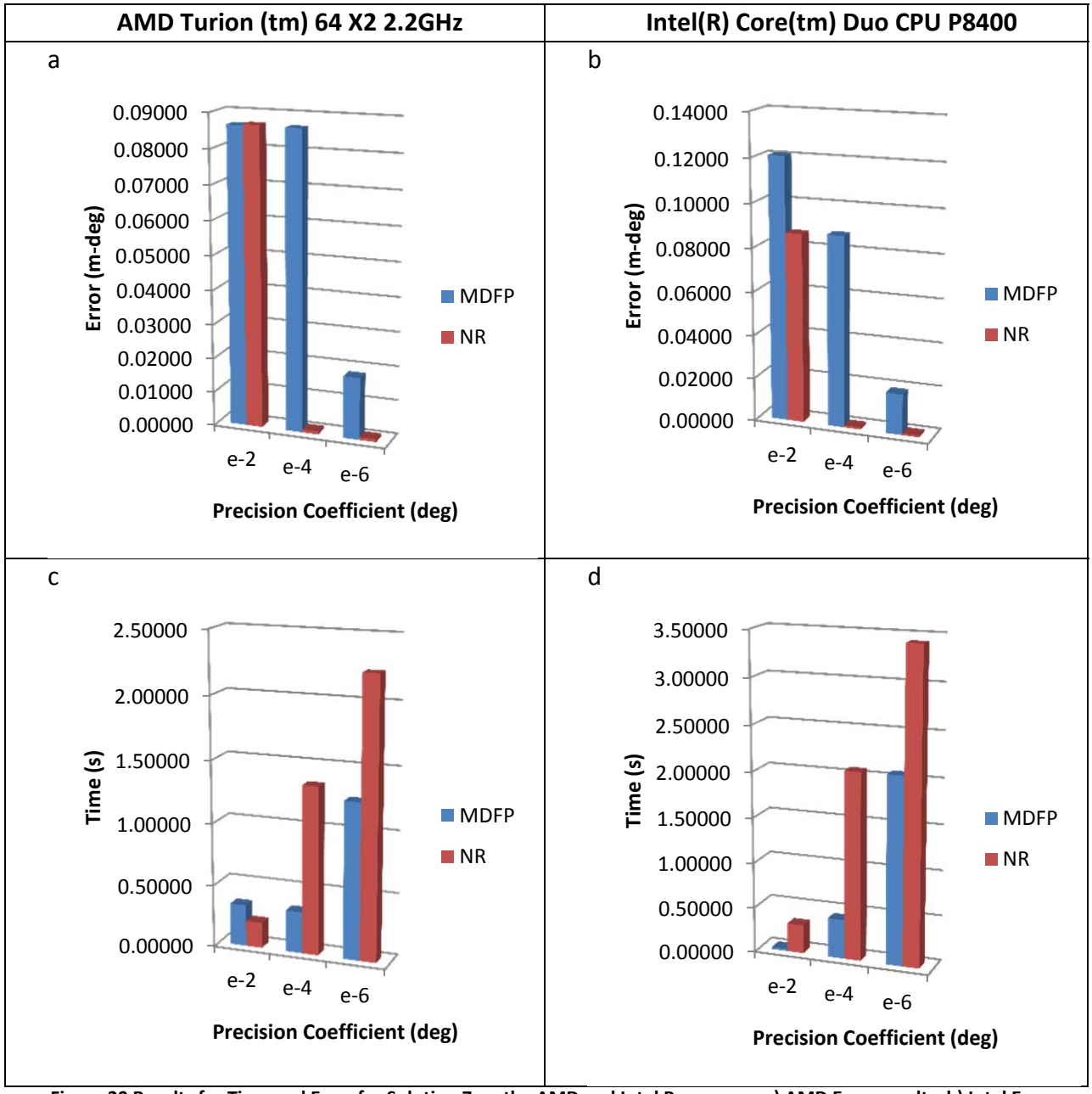


Figure 20 Results for Time and Error for Solution 7 on the AMD and Intel Processors. a) AMD Error results, b) Intel Error results, c) AMD Time results, d) AMD Time results (Where; e-2=10^{-2}, e-4=10^{-4}, e-6=10^{-6})

For the seventh solution, on the AMD processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.08650 for 10^{-2}, 0.08650 for 10^{-4}, and 0.01820 for 10^{-6}, while the NR

Algorithm yielded 0.08680 for 10^{-2}, 0.00088 for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.33526s for 10^{-2}, 0.33470s for 10^{-4} and 1.25492s for 10^{-6}, while the NR algorithm measured 0.20766s for 10^{-2}, 1.34587s for 10^{-4} and 2.22827s for 10^{-6}.

Using the Intel processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.12070 for 10^{-2}, 0.08760 for 10^{-4}, and 0.01880 for 10^{-6}, while the NR Algorithm yielded 0.08680 for 10^{-2}, 0.00088 for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.02321s for 10^{-2}, 0.43274s for 10^{-4} and 2.06695s for 10^{-6}, while the NR algorithm measured 0.31798s for 10^{-2}, 2.06634s for $10^{-4}$and 3.42332s for 10^{-6}.

On the AMD processor both algorithms followed a pattern in which for the 10^{-4} and 10^{-6} precision coefficients the result for the NR algorithm was more accurate than the MDFP yet, it took longer to reach this result. On the other hand, this behavior was observed for all precision coefficients using the Intel processor. For the 10^{-2} precision coefficient the result for NR was less accurate than the MDFP on the AMD processor but took less time to reach the result. It is important to understand that the results found using the INTEL processor are similar to those found in the AMD processor, however all the calculation times were increased on the INTEL processor.

5.7.0 Solution 8

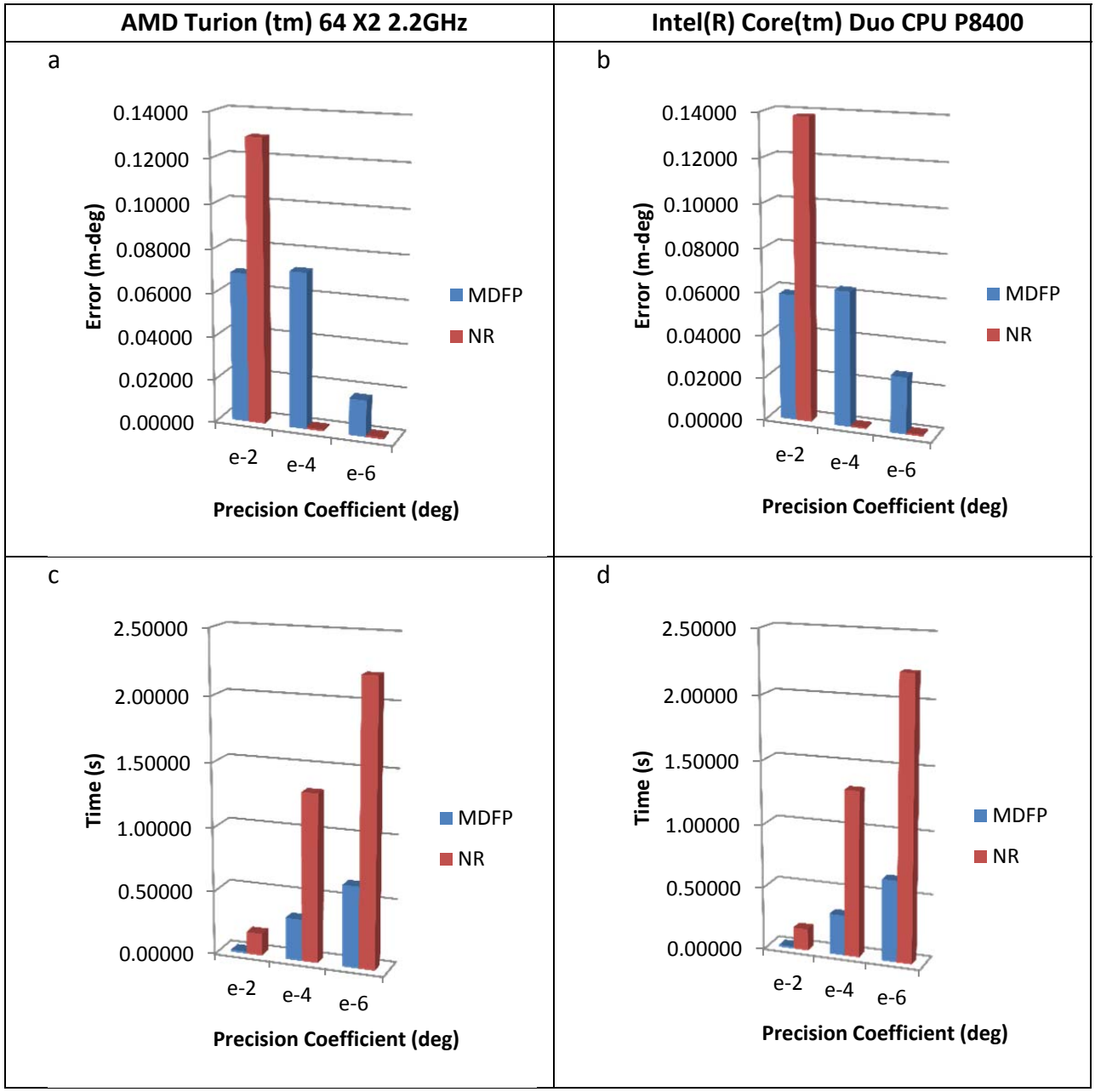


Figure 21 Results for Time and Error for Solution 8 on the AMD and Intel Processors. a) AMD Error results, b) Intel Error results, c) AMD Time results, d) AMD Time results (Where; e-2=10^{-2}, e-4=10^{-4}, e-6=10^{-6})

For the eighth solution, using the AMD processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.06880 for 10^{-2}, 0.07180 for 10^{-4}, and 0.01690 for 10^{-6}, while the NR

Algorithm yielded 0.12930 for 10^{-2}, 0.0007780 for 10^{-4}, and 0.00086 for 10^{-6}. In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.01650s for 10^{-2}, 0.32963s for 10^{-4} and 0.64148 for 10^{-6}, while the NR algorithm measured 0.17563s for 10^{-2}, 1.31808s for 10^{-4} and 2.21265s for 10^{-6}.

On the Intel processor, both the MDFP algorithm and the NR algorithm's Error values decreased as the algorithm's precision coefficient decreased. The MDFP yielded 0.05870 for 10^{-2}, 0.06280 for 10^{-4}, and 0.02650 for 10^{-6}, while the NR Algorithm yielded 0.13830 for 10^{-2}, 0.00070 for 10^{-4}, and 0.00085 for 10^{-6}.In contrast, the Calculation Time (measured in seconds *(s)*) for the MDFP and the NR algorithm precision increased as the precision coefficient decreased. The MDFP measured a solution time of 0.01623s for 10^{-2}, 0.32271s for 10^{-4} and 0.65058s for 10^{-6}, while the NR algorithm measured 0.17356s for 10^{-2}, 1.31500s for $10^{-4}$and 2.22314s for 10^{-6}.

Both algorithms followed a pattern in which for the 10^{-4} and 10^{-6} precision coefficients the NR algorithm took longer to reach a result than the MDFP, however this result was more accurate. This behavior was observed using both processors. For the 10^{-2} precision coefficient the result for NR was a less accurate than the MDFP for both processors. It is important to understand that the results found using the INTEL processor are similar to those found in the AMD processor, however all the calculation times were increased on the INTEL processor.

Table 11 Summary of the discussion of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for each processor

Solution	Processor	
	AMD	**Intel**
1	The NR $\mathbf{10^{-2}}$ yielded a larger error than the MDFP, however, NR Error for $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result for $\mathbf{10^{-2}}$ was shorter than MDFP however time for $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$ was longer than MDFP	The NR $\mathbf{10^{-2}}$ yielded a larger error than the MDFP, however, NR Error for $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$
2	The NR $\mathbf{10^{-2}}$ yielded a larger error than the MDFP, however, NR Error for $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result, was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$	The NR $\mathbf{10^{-2}}$ yielded a larger error than the MDFP, however, NR Error for $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$
3	The NR $\mathbf{10^{-2}}$ yielded a larger error than the MDFP, however, NR Error for $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result was longer than MDFP for $\mathbf{10^{-2}}$ and $\mathbf{10^{-4}}$. For $\mathbf{10^{-6}}$ the NR time to reach a result was shorter than MDFP.	The NR $\mathbf{10^{-2}}$ yielded a larger error than the MDFP, however, NR Error for $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result was longer than MDFP for $\mathbf{10^{-2}}$ and $\mathbf{10^{-4}}$. For $\mathbf{10^{-6}}$ the NR time to reach a result was shorter than MDFP.
4	The NR Error for $\mathbf{10^{-2}}$, $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result, was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$	The NR Error for $\mathbf{10^{-2}}$, $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$
5	The NR Error for $\mathbf{10^{-2}}$, $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result, was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$	The NR Error for $\mathbf{10^{-2}}$, $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$
6	The NR $\mathbf{10^{-2}}$ yielded a larger error than the MDFP. However, NR Error for $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result for $\mathbf{10^{-2}}$ was shorter than MDFP, however time for $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$ was longer than MDFP	The NR Error for $\mathbf{10^{-2}}$, $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$
7	The NR $\mathbf{10^{-2}}$ yielded a slightly larger error than the MDFP. However, NR Error for $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result for $\mathbf{10^{-2}}$ was shorter than MDFP. However time for $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$ was longer than MDFP	The NR Error for $\mathbf{10^{-2}}$, $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$
8	The NR $\mathbf{10^{-2}}$ yielded a larger error than the MDFP, however, NR Error for $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$	The NR Error for $\mathbf{10^{-2}}$, $\mathbf{10^{-4}}$and $\mathbf{10^{-6}}$ was smaller than the MDFP. NR time to reach a result was longer than MDFP for $\mathbf{10^{-2}}$ **,** $\mathbf{10^{-4}}$ and $\mathbf{10^{-6}}$

6 Conclusion

This research intended to develop a test procedure with the purpose of algorithm comparison. To locate the end effector of a robot in a specific position and orientation, the inverse kinematics of the robot's frame must be solved. Depending on the robot's configuration, finding the analytic solution of this inverse kinematic equation is often difficult. As a result, algorithms that numerically reach a desired solution are developed with the purpose of easing the computing power required and the response time of the robot. It is not unusual to have several algorithms that claim to solve the inverse kinematic equation for the same robot configuration; and as a result, there is a need to methodically compare each algorithm and select the most efficient one. In this study, two computers with different processor architectures were used to execute two algorithms. Two processor architectures were used; the *AMD Turion (tm) 64 X2 2.2GHz* and the *Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz*.

The algorithms that were selected take advantage of the optimization power of the Newton-Raphson method and a modified version of the DFP method. The time necessary for each algorithm to solve the inverse kinematic problem and the precision of such result was measured and then the information was used to select the best performer on a given computer architecture. The initial point from which the algorithms begin the iteration process was the same; this ensured that the testing conditions were similar for both algorithms. This study also intends to inspire future work in which other engineers and developers add performance data of more algorithms in different computer architectures, thus turning this into a database that would be an invaluable tool for the field of robotics.

After an exhaustive analysis of the information gathered by the two inverse kinematic algorithms it was possible to visualize a recurring pattern between the performances of both algorithms. In the *AMD Turion (tm) 64 X2 2.2GHz* architecture, the MDFP algorithm was able to find an inverse kinematic solution faster than the Newton Raphson Algorithm. The solution's Error ranged between 10^{-2} and 10^{-3} according to equation (75) which is a precise result. When compared to the Newton Raphson Algorithm, these results were acquired at a faster rate, the average solution time difference between both algorithms was 1.246179 seconds for the *AMD Turion (tm) 64 X2 2.2GHz processor* and 1.602579 seconds for the *Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz processor*. The Newton Raphson Algorithm took a longer time to reach the result when compared to the MDFP. However, the precision was greater for the Newton Raphson Algorithm than the one obtained with the MDFP since the Newton Raphson Algorithm was able to reach each solution with an error of 10^{-4} according to equation (75).

In order to expand the validity of the experiment and to address the effect of different processor architectures, the algorithms were also executed using the *Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz* architecture. The results gathered from this second architecture showed that the average difference between the solution times of both algorithms for each of the 8 solutions was similar to the average difference recorded on the previous processor. Thus, it can be concluded that the algorithms performed similarly in both processors. The pattern of the results did not vary with the architecture in any of the solutions, however, the Newton Raphson algorithm took longer to solve the inverse kinematic problem when executed in this architecture. Similarly, the MDFP took a longer time to reach the solution and the results varied

slightly for every solution when compared to the results for the AMD architecture. The information gathered in this research shows that the processor architecture plays a role in the performance of the algorithms. This has to be taken into account when developing a robot, now that even though the algorithms behaved similarly relative to each other in both architectures, the time taken to solution increased significantly for both algorithms in the *Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz processor* architecture.

In summary, both algorithms performed correctly when tasked to solve the inverse kinematic problem of a 6R robot. The results of this experiment can shed light to the decision of which algorithm will be the best choice for a robot when considering the application for which the algorithm will be subject to. If the robot is required to perform a task that requires speed at the expense of precision, then the MDFP algorithm will become the best choice. In contrast, if the process requires a high degree of precision and some computation time can be spared, then the best choice for the programming will be the Newton Raphson Algorithm. Judging from these results it is clear that the experiment was a success, not only was the algorithm's capabilities successfully measured but it was possible to compare their performance and determine which one would be the best option. This experiment was able to show that the computer architecture plays a role in the execution time of the algorithm.

References

[1] Courtney E Howard, “Cubic introduces compact robot for bomb disposal and similar military missions,” *Military & Aerospace Electronics*, vol. 19, no. 8, p. 10, Aug. 2008.

[2] D. Gerhardus, “Robot-assisted surgery: The future is here,” *Journal of Healthcare Management*, vol. 48, no. 4, pp. 242–51, Aug. 2003.

[3] Baida Qu and Baoguo Xu,2011, “A Control Algorithm Of General 6R Mechanic Arm Based On Inverse Kinematics,” Computer Science and Automation Engineering (CSAE), IEEE International Conference, vol. 1(1), pp. 327–330.

[4] F.Yin, Y.N. Wang, and S.-N. Wei,2011, “Inverse Kinematic Solution For Robot Manipulator Based On Electromagnetism-like And Modified DFP Algorithms,” Acta Automatica Sinica*,* vol. 37(1), pp. 74–82.

[5] F. L. Lewis, D. M. Dawson, and C. T. Abdallah,2003, *Robot Manipulator Control*, CRC Press, Boca Raton, FL, Chap. 1.

[6] Nathan Hodge, “In the Afghan War, a Little Robot Can Be a Soldier’s Best Friend,” *Wall Street Journal*, p. A.1, 14-Jun-2012.

[7] HT Media Ltd., “Japan uses robots to measure radiation at nuclear plant,” *The Financial Express, New Delhi*, 26-Apr-2011.

[8] Scott Powers, “Mars rover mission ‘Curiosity’ blasts off from cape,” *McClatchy - Tribune Business News, Sacramento, California*, 26-Nov-2011.

[9] S. B. Niku,2011, *Introduction to Robotics; Analisis, Control, Applications*, 2nd Edition,John Wiley & Sons, Hoboken, NJ.

[10] M. W. Spong,2006, S. Hutchinson, and M. Vidyasgar, *Robot Modeling and Control*, 1st Edition, John Wiley & Sons, Hoboken, NJ.

[11] L. Rainer, 2000, *Code Optimization Techniques for Embedded Processors; Methods, Algorithms and Tools*, 2nd Edition, Kluwer Academic Publishers, Norwell, MA.

[12] F. Cheng, H. Jiang, and H. Lou, 2008, in *Smart Structures: Innovative Systems for Seismic Response Control*, CRC Press, Boca Raton, FL, "Appendix A: MatLab".

[13] J. Craig, 1989, *Introduction to Robotics; Mechanics and Control*, 2nd Edition, Pearson Prentice Hall, Saddle River, NJ.

[14] R. Paul, 1982,*Robot Manipulators: Mathematics, Programming and Control*, MIT press, Cambridge, MA.

[15] H. Deitel and P. Deitel, 2004, *C How to Program*, 4th ed. Pearson Prentice Hall, Saddle River, NJ.

[16] W. MathWorld, “Algorithm - from Wolfram MathWorld,”. Available [Online]: http://mathworld.wolfram.com/Algorithm.html. [Accessed: 26-May-2011].

[17] S. R. Schach,1993, *Software Engineering*, 2nd ed. Aksen Associates Incorporated, Los Angeles, CA.

[18] M. S. Packianather, M. Landy, and D. T. Pham, 2009,“Enhancing the Speed of the Bees Algorithm Using Pheromone-Based Recruitment”, Industrial Informatics*, 7th IEEE International Conference*, pp. 789–794.

[19] H. Hang, Q. Weicheng, and F. Shuo, 2007, “Optimizing the Architecture of Planar Phased Array by Improved Genetic Algorithm”, Microwave, Antenna, Propagation and EMC Technologies for Wireless Communications International Symposium, pp. 676–679.
[20] S. I. Birbil and S.-C. Fang, 2003, “An Electromagnetism-Like Mechanism For Global Optimization,” Journal of Global Optimization, vol. 25(3), pp. 263–282.
[21] L.C. T. Wang and C. C. Chen, 1991, “A Combined Optimization Method for Solving the Inverse Kinematics Problems of Mechanical Manipulators”, Robotics and Automation, IEEE Transactions, vol. 7(4), pp. 489–499.
[22] J. Nocedal and S. J. Wright, 2006, *Numerical Optimization*, 2nd Edition, Springer, New York, NY.
[23] L. T. Wang and B. Ravani, 1985, “Recursive Computations Of Kinematic And Dynamic Equations For Mechanical Manipulators.,” *IEEE journal of robotics and automation,* pp. 124–131.

Appendix A : Matlab Codes

Appendix A1: Modified David Fletchell Powell (MDFP)

MDFP.m

```
clear all
clc
C=0;
while C<10
C=C+1;
tic
%initial guess
fi=[10;20;30;40;50;60];
size=6;
%%
Puma=[0       -90     0.6604 ;
      0.4318   0      0.2    ;
      0        90    -0.0505 ;
      0       -90     0.4320 ;
      0        90     0      ;
      0        0      0.0565];
%%
 Pw=[0.7351
     0.3856
     0.6818];
 Ow=[-0.727    0.317    0.610
      0.646    0.014    0.763
      0.233    0.948   -0.215];
H=eye(size);
la=1.5;
[g]=Grad(size,fi,Pw,Ow,Puma);
d=-(H*g);
while norm(g)>10e-6

    fin=fi+(la*d);
    [gn]=Grad(size,fin,Pw,Ow,Puma);
    q=gn-g;
    p=fin-fi;
    H=H+((p*p')/(q'*p))-((H*q*q'*H)/(q'*H*q));
    d=-(H*gn);
    g=gn;
    fi=fin;

end
MDFP_time(C,1)=toc;

%%%%%%%%%%%%%%%%%%%%%%%%%%
[T]=DH(fi,Puma,size);
Tend(1:3,1:3)=Ow;
Tend(1:3,4)=Pw;
Tend(4,1:4)=[0,0,0,1];
MDFP_pres(C,1)=norm(T-Tend);
%%%%%%%%%%%%%%%%%%%%%%%%%%
end
MDFP_pres
MDFP_time
fi=fi
```

Grad.M

```
function [g]=Grad(size,fi,Pw,Ow,Data)

x=[1;0;0];
y=[0;1;0];
z=[0;0;1];
Pij=[0; 0; 0];

 X(:,1)=x;
 Y(:,1)=y;
 Z(:,1)=z;
 pij(:,1)=Pij;

Data(:,4)=fi(:,1);
for i=1:size
  x=x*cosd(Data(i,4))+y*sind(Data(i,4));
  zo=z;
  z=z*cosd(Data(i,2))+(cross(x,z))*sind(Data(i,2));
  y=cross(z,x);
  P=(Data(i,3)*zo)+(Data(i,1)*x);
  Pij=Pij+P;

  X(:,i+1)=x;
  Y(:,i+1)=y;
  Z(:,i+1)=z;
  pij(:,i+1)=Pij;

end
sum=(dot(Ow(1:3,1),x)-1)*(cross(x,Ow(1:3,1)))+    (dot(Ow(1:3,2),y)-1)*(cross(y,Ow(1:3,2)))+    (dot(Ow(1:3,3),z)-
1)*(cross(z,Ow(1:3,3)));
P2a=Pw-Pij;

for i=1:size
P1=2*Z(:,i);
Pih=Pij-pij(:,i);
P2=cross(P2a,Pih)+sum;
g(i,1)=dot(P1,P2);
end
```

DH.M

```
function [AT,O,Z]=DH(r,Data,dof)

%Denavit-Hartenberg Convention
  O(1:3,1)=[0;0;0];
  Z(1:3,1)=[0;0;1];

Data(:,4)=r(:,1);
AT=1;

for j=1:dof
 %transformation matrix for link j

 A=[cosd(Data(j,4)), -sind(Data(j,4))*cosd(Data(j,2)), sind(Data(j,4))*sind(Data(j,2)), Data(j,1)*cosd(Data(j,4));
     sind(Data(j,4)), cosd(Data(j,4))*cosd(Data(j,2)), -cosd(Data(j,4))*sind(Data(j,2)), Data(j,1)*sind(Data(j,4));
     0, sind(Data(j,2)), cosd(Data(j,2)), Data(j,3);
     0, 0, 0, 1] ;
 AT=AT*A;
O((1:3),j+1)=AT((1:3),4);
Z((1:3),j+1)=AT((1:3),3);
End
```

Appendix A2: Newton-Raphson

Newton_Raphson.m

```
clear all
clc
C=0;
while C<10
C=C+1;
tic
I=eye(4);
Puma=[0       -90     0.6604 ;
      0.4318   0      0.2    ;
      0        90    -0.0505 ;
      0       -90     0.4320 ;
      0        90     0      ;
      0        0      0.0565];
 Pw=[0.7351
     0.3856
     0.6818];
 Ow=[-0.727    0.317    0.610
      0.646    0.014    0.763
      0.233    0.948   -0.215];
Tend(1:3,1:3)=Ow;
 Tend(1:3,4)=Pw;
 Tend(4,1:4)=[0,0,0,1];
%%
dof=6;
%initial guess
%fi=[10;20;30;40;50;60];
%fi=[10;20;30;-130;-50;-110];
%fi=[10;-30;140;-70;-30;160];
%fi=[-140;-140;30;130;-10;140];
%fi=[-140;-140;30;-40;10;-30];
%fi=[-140;150;140;20;-40;-90];
%fi=[-140;150;140;-150;40;80];
fi=[10;-30;140;103;30;-10];
iter=1;
delta=1;
while norm(delta)>10e-6
[T,O,Z]=DH(fi,Puma,dof);
Del=((Tend)*inv(T))-I;
[D]=Dcreate(Del);
[J]=jacobian(O,Z,dof);
[U,S,V]=svd(J);
[Sp]=Splus(S,dof);
delta=(U*Sp*ctranspose(V))'*D;
fi=delta+fi;
end
N_Raph_time(C,1)=toc;
N_Raph_pres(C,1)=norm(T-Tend);
%%%%%%%%%%%%%%%%%%%%%%%%%%
end
N_Raph_pres
N_Raph_time
fi
```

DH.m

```
function [AT,O,Z]=DH(r,Data,dof)

%Denavit-Hartenberg Convention
  O(1:3,1)=[0;0;0];
  Z(1:3,1)=[0;0;1];

Data(:,4)=r(:,1);
AT=1;

for j=1:dof
 %transformation matrix for link j

 A=[cosd(Data(j,4)), -sind(Data(j,4))*cosd(Data(j,2)), sind(Data(j,4))*sind(Data(j,2)), Data(j,1)*cosd(Data(j,4));
      sind(Data(j,4)), cosd(Data(j,4))*cosd(Data(j,2)), -cosd(Data(j,4))*sind(Data(j,2)), Data(j,1)*sind(Data(j,4));
      0, sind(Data(j,2)), cosd(Data(j,2)), Data(j,3);
      0, 0, 0, 1] ;
 AT=AT*A;
O((1:3),j+1)=AT((1:3),4);
Z((1:3),j+1)=AT((1:3),3);
End
```

Jacobian.m

```
function [J]=jacobian(O,Z,dof)

for i=1:dof
    J(1:3,i)=cross(Z(1:3,i),(O(1:3,dof+1)-O(1:3,i)));
    J(4:6,i)=Z(1:3,i);
End
```

Splus.m

```
function [Sp]=Splus(S,dof)

for i=1:dof
    for j=1:6
        if S(j,i)~= 0
            Sp(j,i)=1/S(j,i);
        else
            Sp(j,i)=S(j,i);
        end
    end
end
```

Dcreate.m

```
function [D]=Dcreate(Del)
D(1:3,1)=Del(1:3,4);
D(4,1)=Del(3,2);
D(5,1)=Del(1,3);
D(6,1)=Del(2,1);
```

Appendix B: Results

Appendix B1: AMD Turion (tm) 64 X2 2.2GHz

Results for solution 2

Table 12 Solution 2 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.03410	0.19460	0.03410	0.00094	0.00480	0.00086
2	0.03410	0.19460	0.03410	0.00094	0.00480	0.00086
3	0.03410	0.19460	0.03410	0.00094	0.00480	0.00086
4	0.03410	0.19460	0.03410	0.00094	0.00480	0.00086
5	0.03410	0.19460	0.03410	0.00094	0.00480	0.00086
6	0.03410	0.19460	0.03410	0.00094	0.00480	0.00086
7	0.03410	0.19460	0.03410	0.00094	0.00480	0.00086
8	0.03410	0.19460	0.03410	0.00094	0.00480	0.00086
9	0.03410	0.19460	0.03410	0.00094	0.00480	0.00086
10	0.03410	0.19460	0.03410	0.00094	0.00480	0.00086
Average	**0.03410**	**0.19460**	**0.03410**	**0.00094**	**0.00480**	**0.00086**

Table 13 Solution 2 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.25130	0.21500	0.25350	1.61150	0.85930	2.49940
2	0.20480	0.17080	0.20280	1.56870	0.80620	2.44630
3	0.20220	0.17040	0.20170	1.56870	0.80580	2.44150
4	0.20170	0.16950	0.20160	1.56450	0.80380	2.44000
5	0.20180	0.16970	0.20200	1.56210	0.80110	2.43700
6	0.20180	0.16910	0.20170	1.56460	0.80210	2.43870
7	0.20190	0.16970	0.20150	1.56320	0.80280	2.43600
8	0.20160	0.16820	0.20140	1.56440	0.80260	2.43850
9	0.20100	0.16920	0.20140	1.56980	0.80260	2.44100
10	0.20170	0.16850	0.20210	1.56400	0.80150	2.43700
Average	**0.20698**	**0.17401**	**0.20697**	**1.57015**	**0.80878**	**2.44554**

Results for solution 3

Table 14 Solution 3 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.08330	0.13930	0.08330	0.00091	0.01810	0.00086
2	0.08330	0.13930	0.08330	0.00091	0.01810	0.00086
3	0.08330	0.13930	0.08330	0.00091	0.01810	0.00086
4	0.08330	0.13930	0.08330	0.00091	0.01810	0.00086
5	0.08330	0.13930	0.08330	0.00091	0.01810	0.00086
6	0.08330	0.13930	0.08330	0.00091	0.01810	0.00086
7	0.08330	0.13930	0.08330	0.00091	0.01810	0.00086
8	0.08330	0.13930	0.08330	0.00091	0.01810	0.00086
9	0.08330	0.13930	0.08330	0.00091	0.01810	0.00086
10	0.08330	0.13930	0.08330	0.00091	0.01810	0.00086
Average	**0.08330**	**0.13930**	**0.08330**	**0.00091**	**0.01810**	**0.00086**

Table 15 Solution 3 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.20390	0.20600	0.20350	1.49060	3.24620	2.37080
2	0.15340	0.16270	0.15320	1.44510	3.20200	2.32000
3	0.15440	0.16210	0.15300	1.44590	3.19740	2.31310
4	0.15470	0.16330	0.15300	1.44560	3.19390	2.31550
5	0.15320	0.16320	0.15300	1.44560	3.19080	2.31710
6	0.15300	0.16430	0.15410	1.44700	3.19220	2.31510
7	0.15250	0.16200	0.15560	1.44520	3.19890	2.31730
8	0.15280	0.16070	0.15380	1.44600	3.19740	2.31660
9	0.15270	0.16090	0.15440	1.44160	3.18940	2.31880
10	0.15320	0.16130	0.15460	1.44370	3.19360	2.31370
Average	**0.15838**	**0.16665**	**0.15882**	**1.44963**	**3.20018**	**2.32180**

Results for solution 4

Table 16 Solution 4 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.06450	0.03070	0.06450	0.00100	0.02320	0.00086
2	0.06450	0.03070	0.06450	0.00100	0.02320	0.00086
3	0.06450	0.03070	0.06450	0.00100	0.02320	0.00086
4	0.06450	0.03070	0.06450	0.00100	0.02320	0.00086
5	0.06450	0.03070	0.06450	0.00100	0.02320	0.00086
6	0.06450	0.03070	0.06450	0.00100	0.02320	0.00086
7	0.06450	0.03070	0.06450	0.00100	0.02320	0.00086
8	0.06450	0.03070	0.06450	0.00100	0.02320	0.00086
9	0.06450	0.03070	0.06450	0.00100	0.02320	0.00086
10	0.06450	0.03070	0.06450	0.00100	0.02320	0.00086
Average	**0.06450**	**0.03070**	**0.06450**	**0.00100**	**0.02320**	**0.00086**

Table 17 Solution 4 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.35350	0.53480	0.35130	1.53420	1.04270	2.40160
2	0.30350	0.49320	0.30230	1.48930	0.99270	2.35670
3	0.30160	0.48980	0.30120	1.48780	0.98690	2.35650
4	0.30160	0.48840	0.30130	1.48630	0.98430	2.36430
5	0.30080	0.48880	0.30180	1.48710	0.98500	2.36130
6	0.30050	0.48970	0.30090	1.48760	0.98660	2.36280
7	0.30070	0.48990	0.30130	1.48680	0.98830	2.35810
8	0.30040	0.48850	0.30270	1.48780	0.98780	2.35240
9	0.30060	0.48780	0.30240	1.49110	0.98700	2.35260
10	0.30210	0.48770	0.30170	1.48610	0.98840	2.35560
Average	**0.30653**	**0.49386**	**0.30669**	**1.49241**	**0.99297**	**2.36219**

Results for solution 5

Table 18 Solution 5 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.05600	0.03440	0.05600	0.00072	0.02310	0.00086
2	0.05600	0.03440	0.05600	0.00072	0.02310	0.00086
3	0.05600	0.03440	0.05600	0.00072	0.02310	0.00086
4	0.05600	0.03440	0.05600	0.00072	0.02310	0.00086
5	0.05600	0.03440	0.05600	0.00072	0.02310	0.00086
6	0.05600	0.03440	0.05600	0.00072	0.02310	0.00086
7	0.05600	0.03440	0.05600	0.00072	0.02310	0.00086
8	0.05600	0.03440	0.05600	0.00072	0.02310	0.00086
9	0.05600	0.03440	0.05600	0.00072	0.02310	0.00086
10	0.05600	0.03440	0.05600	0.00072	0.02310	0.00086
Average	**0.05600**	**0.03440**	**0.05600**	**0.00072**	**0.02310**	**0.00086**

Table 19 Solution 5 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.34680	0.50040	0.34510	1.55710	1.03620	2.44390
2	0.29830	0.45710	0.29800	1.50860	0.98330	2.39600
3	0.29800	0.45450	0.29580	1.50720	0.98290	2.39040
4	0.29810	0.45420	0.29610	1.50450	0.98280	2.39610
5	0.29690	0.45300	0.29570	1.50720	0.98300	2.40380
6	0.29700	0.45310	0.29580	1.50780	0.98320	2.39040
7	0.29730	0.45330	0.29570	1.50830	0.98220	2.38980
8	0.29730	0.45330	0.29560	1.50490	0.98450	2.39480
9	0.29740	0.45340	0.29540	1.50470	0.98350	2.38960
10	0.29740	0.45240	0.29590	1.50750	0.98180	2.38300
Average	**0.30245**	**0.45847**	**0.30091**	**1.51178**	**0.98834**	**2.39778**

Results for solution 6

Table 20 Solution 6 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.06320	0.11550	0.06320	0.00072	0.01760	0.00086
2	0.06320	0.11550	0.06320	0.00072	0.01760	0.00086
3	0.06320	0.11550	0.06320	0.00072	0.01760	0.00086
4	0.06320	0.11550	0.06320	0.00072	0.01760	0.00086
5	0.06320	0.11550	0.06320	0.00072	0.01760	0.00086
6	0.06320	0.11550	0.06320	0.00072	0.01760	0.00086
7	0.06320	0.11550	0.06320	0.00072	0.01760	0.00086
8	0.06320	0.11550	0.06320	0.00072	0.01760	0.00086
9	0.06320	0.11550	0.06320	0.00072	0.01760	0.00086
10	0.06320	0.11550	0.06320	0.00072	0.01760	0.00086
Average	**0.06320**	**0.11550**	**0.06320**	**0.00072**	**0.01760**	**0.00086**

Table 21 Solution 6 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.28520	0.17220	0.28550	1.50910	1.97540	2.38890
2	0.23640	0.12950	0.23500	1.45750	1.92830	2.32970
3	0.23590	0.12780	0.23460	1.44850	1.92490	2.32810
4	0.23430	0.12790	0.23430	1.44830	1.92580	2.32510
5	0.23360	0.12730	0.23420	1.44800	1.92750	2.32700
6	0.23400	0.12750	0.23400	1.44930	1.92240	2.32790
7	0.23410	0.12690	0.23410	1.44460	1.92080	2.33550
8	0.23420	0.12700	0.23400	1.44720	1.92150	2.34080
9	0.23420	0.12700	0.23330	1.44490	1.92050	2.33680
10	0.23420	0.12720	0.23300	1.44970	1.92210	2.33990
Average	**0.23961**	**0.13203**	**0.23920**	**1.45471**	**1.92892**	**2.33797**

Results for solution 7

Table 22 Solution 7 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.08650	0.08680	0.08650	0.00088	0.01820	0.00086
2	0.08650	0.08680	0.08650	0.00088	0.01820	0.00086
3	0.08650	0.08680	0.08650	0.00088	0.01820	0.00086
4	0.08650	0.08680	0.08650	0.00088	0.01820	0.00086
5	0.08650	0.08680	0.08650	0.00088	0.01820	0.00086
6	0.08650	0.08680	0.08650	0.00088	0.01820	0.00086
7	0.08650	0.08680	0.08650	0.00088	0.01820	0.00086
8	0.08650	0.08680	0.08650	0.00088	0.01820	0.00086
9	0.08650	0.08680	0.08650	0.00088	0.01820	0.00086
10	0.08650	0.08680	0.08650	0.00088	0.01820	0.00086
Average	**0.08650**	**0.08680**	**0.08650**	**0.00088**	**0.01820**	**0.00086**

Table 23 Solution 7 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.38100	0.24740	0.38110	1.40010	1.30320	2.28540
2	0.33040	0.20400	0.33120	1.34270	1.25040	2.22500
3	0.33070	0.20420	0.32990	1.34680	1.25010	2.21950
4	0.33040	0.20340	0.32980	1.33940	1.25100	2.22190
5	0.33070	0.20280	0.32930	1.34040	1.25020	2.22240
6	0.33040	0.20410	0.32910	1.34010	1.24900	2.22110
7	0.33020	0.20410	0.32870	1.33640	1.24820	2.21960
8	0.33010	0.20240	0.32920	1.33800	1.24910	2.22180
9	0.33010	0.20210	0.32930	1.33810	1.24870	2.22290
10	0.32860	0.20210	0.32940	1.33670	1.24930	2.22310
Average	**0.33526**	**0.20766**	**0.33470**	**1.34587**	**1.25492**	**2.22827**

Results for solution 8

Table 24 Solution 8 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.06880	0.12930	0.07180	0.00078	0.01690	0.00086
2	0.06880	0.12930	0.07180	0.00078	0.01690	0.00086
3	0.06880	0.12930	0.07180	0.00078	0.01690	0.00086
4	0.06880	0.12930	0.07180	0.00078	0.01690	0.00086
5	0.06880	0.12930	0.07180	0.00078	0.01690	0.00086
6	0.06880	0.12930	0.07180	0.00078	0.01690	0.00086
7	0.06880	0.12930	0.07180	0.00078	0.01690	0.00086
8	0.06880	0.12930	0.07180	0.00078	0.01690	0.00086
9	0.06880	0.12930	0.07180	0.00078	0.01690	0.00086
10	0.06880	0.12930	0.07180	0.00078	0.01690	0.00086
Average	**0.06880**	**0.12930**	**0.07180**	**0.00078**	**0.01690**	**0.00086**

Table 25 Solution 8 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.09490	0.49790	0.37720	1.36050	0.68590	2.26550
2	0.00790	0.14080	0.32680	1.31700	0.64240	2.21490
3	0.00770	0.14000	0.32430	1.31300	0.63570	2.20660
4	0.00820	0.13950	0.32400	1.31360	0.63540	2.20220
5	0.00780	0.13960	0.32530	1.31190	0.63520	2.20450
6	0.00770	0.13960	0.32470	1.31350	0.63550	2.21080
7	0.00770	0.13920	0.32340	1.31420	0.63520	2.20360
8	0.00770	0.13980	0.32290	1.31140	0.63680	2.20440
9	0.00770	0.13980	0.32370	1.31270	0.63660	2.20760
10	0.00770	0.14010	0.32400	1.31300	0.63610	2.20640
Average	**0.01650**	**0.17563**	**0.32963**	**1.31808**	**0.64148**	**2.21265**

Appendix B2: Intel(R) Core(tm) Duo CPU P8400 @ 2.26GHz 2.27GHz

Results for solution 2

Table 26 Solution 2 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.17550	0.19460	0.05640	0.00094	0.00690	0.00086
2	0.17550	0.19460	0.05640	0.00094	0.00690	0.00086
3	0.17550	0.19460	0.05640	0.00094	0.00690	0.00086
4	0.17550	0.19460	0.05640	0.00094	0.00690	0.00086
5	0.17550	0.19460	0.05640	0.00094	0.00690	0.00086
6	0.17550	0.19460	0.05640	0.00094	0.00690	0.00086
7	0.17550	0.19460	0.05640	0.00094	0.00690	0.00086
8	0.17550	0.19460	0.05640	0.00094	0.00690	0.00086
9	0.17550	0.19460	0.05640	0.00094	0.00690	0.00086
10	0.17550	0.19460	0.05640	0.00094	0.00690	0.00086
Average	**0.17550**	**0.19460**	**0.05640**	**0.00094**	**0.00690**	**0.00086**

Table 27 Solution 2 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.08450	0.32170	0.53570	2.47590	1.81060	3.80210
2	0.01210	0.26320	0.46280	2.40650	1.73670	3.73840
3	0.01190	0.26180	0.46210	2.40830	1.74460	3.73590
4	0.01200	0.26090	0.46150	2.40780	1.73710	3.73530
5	0.01200	0.26040	0.46050	2.40750	1.73410	3.73660
6	0.01200	0.26070	0.46070	2.40800	1.73750	3.73490
7	0.01200	0.26070	0.46070	2.40740	1.73290	3.73390
8	0.01200	0.25980	0.46070	2.40840	1.73100	3.73360
9	0.01200	0.25960	0.46030	2.40900	1.72970	3.73390
10	0.01200	0.25960	0.46060	2.40730	1.72950	3.73490
Average	**0.01925**	**0.26684**	**0.46856**	**2.41461**	**1.74237**	**3.74195**

Results for solution 3

Table 28 Solution 3 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.09570	0.13930	0.08310	0.00091	0.01810	0.00086
2	0.09570	0.13930	0.08310	0.00091	0.01810	0.00086
3	0.09570	0.13930	0.08310	0.00091	0.01810	0.00086
4	0.09570	0.13930	0.08310	0.00091	0.01810	0.00086
5	0.09570	0.13930	0.08310	0.00091	0.01810	0.00086
6	0.09570	0.13930	0.08310	0.00091	0.01810	0.00086
7	0.09570	0.13930	0.08310	0.00091	0.01810	0.00086
8	0.09570	0.13930	0.08310	0.00091	0.01810	0.00086
9	0.09570	0.13930	0.08310	0.00091	0.01810	0.00086
10	0.09570	0.13930	0.08310	0.00091	0.01810	0.00086
Average	**0.09570**	**0.13930**	**0.08310**	**0.00091**	**0.01810**	**0.00086**

Table 29 Solution 3 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.08470	0.30890	0.34500	2.26760	3.79550	3.60910
2	0.01220	0.24880	0.27140	2.20320	3.72490	3.53600
3	0.01210	0.24700	0.27110	2.19900	3.72300	3.53530
4	0.01210	0.24640	0.27130	2.19860	3.72330	3.53940
5	0.01200	0.24640	0.27080	2.19900	3.72320	3.53670
6	0.01200	0.24630	0.27940	2.20040	3.72190	3.53980
7	0.01200	0.24610	0.27070	2.19960	3.72310	3.53780
8	0.01210	0.24540	0.27110	2.19910	3.72190	3.53760
9	0.01200	0.24540	0.27140	2.19850	3.72250	3.53820
10	0.01200	0.24540	0.27090	2.19690	3.72180	3.53720
Average	**0.01932**	**0.25261**	**0.27931**	**2.20619**	**3.73011**	**3.54471**

Results for solution 4

Table 30 Solution 4 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.20940	0.03070	0.07520	0.00100	0.02300	0.00086
2	0.20940	0.03070	0.07520	0.00100	0.02300	0.00086
3	0.20940	0.03070	0.07520	0.00100	0.02300	0.00086
4	0.20940	0.03070	0.07520	0.00100	0.02300	0.00086
5	0.20940	0.03070	0.07520	0.00100	0.02300	0.00086
6	0.20940	0.03070	0.07520	0.00100	0.02300	0.00086
7	0.20940	0.03070	0.07520	0.00100	0.02300	0.00086
8	0.20940	0.03070	0.07520	0.00100	0.02300	0.00086
9	0.20940	0.03070	0.07520	0.00100	0.02300	0.00086
10	0.20940	0.03070	0.07520	0.00100	0.02300	0.00086
Average	**0.20940**	**0.03070**	**0.07520**	**0.00100**	**0.02300**	**0.00086**

Table 31 Solution 4 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.08850	0.81520	0.48290	2.35580	1.13330	3.70410
2	0.01620	0.75390	0.40990	2.28630	1.05940	3.63570
3	0.01590	0.75020	0.40890	2.28770	1.05920	3.62710
4	0.01590	0.74970	0.40840	2.28430	1.05930	3.62350
5	0.01590	0.74940	0.40900	2.28320	1.05870	3.61860
6	0.01600	0.74990	0.40860	2.28420	1.05940	3.62260
7	0.01600	0.74980	0.40870	2.28460	1.05900	3.61940
8	0.01590	0.74980	0.40850	2.28360	1.05910	3.61670
9	0.01590	0.74950	0.40880	2.28500	1.05920	3.61830
10	0.01590	0.75010	0.40860	2.28460	1.05950	3.61990
Average	**0.02321**	**0.75675**	**0.41623**	**2.29193**	**1.06661**	**3.63059**

Results for solution 5

Table 32 Solution 5 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.21470	0.03440	0.05550	0.00072	0.02310	0.00086
2	0.21470	0.03440	0.05550	0.00072	0.02310	0.00086
3	0.21470	0.03440	0.05550	0.00072	0.02310	0.00086
4	0.21470	0.03440	0.05550	0.00072	0.02310	0.00086
5	0.21470	0.03440	0.05550	0.00072	0.02310	0.00086
6	0.21470	0.03440	0.05550	0.00072	0.02310	0.00086
7	0.21470	0.03440	0.05550	0.00072	0.02310	0.00086
8	0.21470	0.03440	0.05550	0.00072	0.02310	0.00086
9	0.21470	0.03440	0.05550	0.00072	0.02310	0.00086
10	0.21470	0.03440	0.05550	0.00072	0.02310	0.00086
Average	**0.21470**	**0.03440**	**0.05550**	**0.00072**	**0.02310**	**0.00086**

Table 33 Solution 5 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.08900	0.76220	0.68410	2.38220	1.49880	3.73240
2	0.01610	0.69950	0.60990	2.32060	1.42090	3.66790
3	0.01590	0.69680	0.60850	2.32030	1.41830	3.66450
4	0.01600	0.69660	0.60730	2.31990	1.41840	3.66560
5	0.01590	0.69620	0.60730	2.31950	1.41900	3.66350
6	0.01600	0.69610	0.60710	2.32040	1.41860	3.66480
7	0.01610	0.69670	0.60730	2.31920	1.41800	3.66520
8	0.01600	0.69600	0.60790	2.31920	1.41880	3.66440
9	0.01600	0.69690	0.60740	2.31870	1.41830	3.66600
10	0.01600	0.69630	0.60740	2.31990	1.41800	3.66630
Average	**0.02330**	**0.70333**	**0.61542**	**2.32599**	**1.42671**	**3.67206**

Results for solution 6

Table 34 Solution 6 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.15680	0.11550	0.06760	0.00072	0.01780	0.00086
2	0.15680	0.11550	0.06760	0.00072	0.01780	0.00086
3	0.15680	0.11550	0.06760	0.00072	0.01780	0.00086
4	0.15680	0.11550	0.06760	0.00072	0.01780	0.00086
5	0.15680	0.11550	0.06760	0.00072	0.01780	0.00086
6	0.15680	0.11550	0.06760	0.00072	0.01780	0.00086
7	0.15680	0.11550	0.06760	0.00072	0.01780	0.00086
8	0.15680	0.11550	0.06760	0.00072	0.01780	0.00086
9	0.15680	0.11550	0.06760	0.00072	0.01780	0.00086
10	0.15680	0.11550	0.06760	0.00072	0.01780	0.00086
Average	**0.15680**	**0.11550**	**0.06760**	**0.00072**	**0.01780**	**0.00086**

Table 35 Solution 6 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.08450	0.25750	0.46380	2.28850	2.48380	3.64040
2	0.01210	0.19750	0.39090	2.22020	2.40280	3.58100
3	0.01200	0.19580	0.38940	2.21820	2.40270	3.57560
4	0.01200	0.19560	0.38970	2.21900	2.40270	3.57120
5	0.01230	0.19510	0.38940	2.21980	2.40240	3.57260
6	0.01200	0.19540	0.38990	2.22010	2.40220	3.57460
7	0.01200	0.19510	0.38950	2.22020	2.40200	3.57470
8	0.01200	0.19510	0.38970	2.22020	2.40220	3.57080
9	0.01200	0.19480	0.38950	2.21870	2.40140	3.57010
10	0.01200	0.19440	0.38950	2.22990	2.40310	3.57310
Average	**0.01929**	**0.20163**	**0.39713**	**2.22748**	**2.41053**	**3.58041**

Results for solution 7

Table 36 Solution 7 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.12070	0.08680	0.08760	0.00088	0.01880	0.00086
2	0.12070	0.08680	0.08760	0.00088	0.01880	0.00086
3	0.12070	0.08680	0.08760	0.00088	0.01880	0.00086
4	0.12070	0.08680	0.08760	0.00088	0.01880	0.00086
5	0.12070	0.08680	0.08760	0.00088	0.01880	0.00086
6	0.12070	0.08680	0.08760	0.00088	0.01880	0.00086
7	0.12070	0.08680	0.08760	0.00088	0.01880	0.00086
8	0.12070	0.08680	0.08760	0.00088	0.01880	0.00086
9	0.12070	0.08680	0.08760	0.00088	0.01880	0.00086
10	0.12070	0.08680	0.08760	0.00088	0.01880	0.00086
Average	**0.12070**	**0.08680**	**0.08760**	**0.00088**	**0.01880**	**0.00086**

Table 37 Solution 7 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.08820	0.37500	0.50060	2.12910	2.13810	3.47990
2	0.01610	0.31460	0.42680	2.06130	2.05940	3.41680
3	0.01600	0.31160	0.42600	2.05910	2.05910	3.41450
4	0.01600	0.31200	0.42540	2.05960	2.05840	3.41630
5	0.01600	0.31180	0.42470	2.06020	2.05850	3.41770
6	0.01600	0.31080	0.42520	2.06010	2.05920	3.41780
7	0.01600	0.31090	0.42480	2.05820	2.05930	3.41760
8	0.01590	0.31090	0.42470	2.05870	2.05930	3.41740
9	0.01590	0.31140	0.42440	2.05870	2.05890	3.41860
10	0.01600	0.31080	0.42480	2.05840	2.05930	3.41660
Average	**0.02321**	**0.31798**	**0.43274**	**2.06634**	**2.06695**	**3.42332**

Results for solution 8

Table 38 Solution 8 Error Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.05870	0.13830	0.06280	0.00070	0.02650	0.00085
2	0.05870	0.13830	0.06280	0.00070	0.02650	0.00085
3	0.05870	0.13830	0.06280	0.00070	0.02650	0.00085
4	0.05870	0.13830	0.06280	0.00070	0.02650	0.00085
5	0.05870	0.13830	0.06280	0.00070	0.02650	0.00085
6	0.05870	0.13830	0.06280	0.00070	0.02650	0.00085
7	0.05870	0.13830	0.06280	0.00070	0.02650	0.00085
8	0.05870	0.13830	0.06280	0.00070	0.02650	0.00085
9	0.05870	0.13830	0.06280	0.00070	0.02650	0.00085
10	0.05870	0.13830	0.06280	0.00070	0.02650	0.00085
Average	**0.05870**	**0.13830**	**0.06280**	**0.00070**	**0.02650**	**0.00085**

Table 39 Solution 8 Time (seconds) Performance Comparison of the Modified Davidon Fletcher Powell (MDFP) and the Newton Raphson (NR) algorithms for different levels of precision: 10^{-2}, 10^{-4} and 10^{-6}

Run	10^{-2}		10^{-4}		10^{-6}	
	MDFP	NR	MDFP	NR	MDFP	NR
1	0.09420	0.49810	0.36820	1.35950	0.68180	2.25950
2	0.00790	0.13980	0.32680	1.29600	0.63940	2.22500
3	0.00770	0.14230	0.32430	1.30900	0.63270	2.30670
4	0.00720	0.13950	0.32400	1.30960	0.62540	2.20240
5	0.00780	0.13760	0.32530	1.29190	0.63520	2.21490
6	0.00750	0.13760	0.32470	1.29340	0.73950	2.21280
7	0.00750	0.13920	0.22340	1.29320	0.63520	2.19360
8	0.00750	0.12970	0.33270	1.29170	0.63680	2.20450
9	0.00750	0.12970	0.33370	1.35270	0.64670	2.20760
10	0.00750	0.14210	0.34400	1.35300	0.63310	2.20440
Average	**0.01623**	**0.17356**	**0.32271**	**1.31500**	**0.65058**	**2.22314**