IMPLEMENTING KINESTATIC CONTROL USING SIX DOF COMPLIANT PARALLEL MECHANISM

By

AKASH VIBHUTE

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2011

This thesis is dedicated in loving memory of my late grandfather Basavraj Vibhute.

ACKNOWLEDGMENTS

I would like to express profound gratitude to Dr. Carl Crane III for being my advisor and Committee Chair. I am deeply indebted to him for his in-depth teaching of the course Geometry of Robots, introducing me to this topic and guiding all the way through my research. His support and patience enabled me to complete the work successfully.

I would like to thank Dr. John Schueller for his time and willingness to serve on my committee, inspiring me to think rationally and work efficiently.

I am also grateful to Dr. Bo Zhang, Ognjen Sosa and Subrat Nayak whose work formed the foundation for my work. I would like to express my sincere thanks to Dr. Eric Schwartz and Shannon Ridgeway for helping me on the various aspects of my research.

I am especially thankful to my father Ajay Vibhute, who inspired me to be industrious and think in a scientific way towards each problem. I thank my mother Shakuntala Vibhute for her support and care which has made my life a joyful bliss. I am grateful in a special way to my late grandfather Basavraj Vibhute for his utmost love and affection for me. I would also like to thank my aunt Neela Shetti, uncle Shital Shetti and grandmother Indumati Vibhute for their encouragement. I wish to thank my uncle Shashikant Vibhute and aunt Geeta Vibhute for trusting me on whichever task I intended to do and lending me moral support in tough times. I would also like to thank my brother Aditya Vibhute and my sisters Akanksha Shetti, Tejaswini Vibhute and Amruta Vibhute for their patience and support.

Last but not the least, I would like to acknowledge the continuous support by Pete Calamore, Shane Ferreira, the faculty and staff at the College of Health and Human Performance for the past year and a half.

TABLE OF CONTENTS

page

APPENDIX

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

IMPLEMENTING KINESTATIC CONTROL USING SIX DOF COMPLIANT PARALLEL MECHANISM

By

Akash Vibhute

May 2011

Chair: Carl D. Crane, III
Major: Mechanical Engineering

This thesis presents the implementation of kinestatic control on a PUMA industrial robot. Serial robots have been in use for several decades for various applications like grinding, welding, machine tending, surgery, etc. All these applications involve the robot going from one point to another point. During this motion there is no measurement of the force being exerted by the tool mounted on the end-effector of the robot to the workpiece. This thesis focuses on controlling the force and torque (wrench) being acted by the tool to the workpiece.

To implement kinestatic control, a 6 DOF parallel mechanism was used between the robot end-effector and tool. This device is a passive parallel mechanism with 6 spring loaded legs connecting the top and bottom platforms. Each calibrated leg has an optical encoder to measure its length in terms of encoder counts. These encoder counts are received on a computer where the force in each of them is calculated by a specially derived formula taking into consideration all the non-linearities associated with a spring.

Once the leg lengths are known, the wrench acting on the workpiece can be calculated using screw theory. This contact wrench can be modified by moving the end effector of the robot in order to change the relative positions of the top and base of the

parallel platform. The required change in the platform can ne determined based on knowledge of the instantaneous compliance matrix which relates a change of contact wrench to a change in relative pose of the base and top platform.

This theory was implemented on Puma robot; the result of one application tested was that the torques in the effective wrench were eliminated to generate pure force along the line of action of the wrench.

The outcome of this research will allow future robots to perform functions that require simultaneous control of pose and contact wrenches. This will extend the range of application of industrial robots.

CHAPTER 1
INTRODUCTION

There have been tremendous developments in robotics and control technologies in the past few decades. Owing to this, robots now are able to perform complex tasks with a high degree of accuracy. Tasks which include assembly line operations or working in a hazardous environment which involves risk to human life can now be performed by robots easily. The majority of these operations can be performed simply by position control of the manipulator and with the repeatability and accuracy of manipulators being high, these operations can be performed with ease. Tasks which involve higher complexity, involving the manipulator or the tool mounted on a manipulator coming in contact with a workpiece, requires the robot to control these contact forces being generated.

The manipulator has rigid links and a very strong actuation. But if the tool on the end effector is misaligned even slightly, it could send the tool crashing on to the workpiece or have other adverse effects on the work environment. Hence it is necessary during such complex operations that, the force and torque being applied be limited in a specified tolerance range. The solution to this is to integrate force control algorithms into the position controller which constantly checks the force – torque feedback in a closed loop approach. A method to measure the spatial wrench is to use load cells. But load cells are very stiff which will give the manipulator very little time to react on sensing an out of tolerance wrench. Hence it is necessary to introduce compliance control. Consider the task of inserting a peg into a hole, this needs an ability to reposition and reorient the peg inline with the axis of the hole. A traditional method to solve this problem is to chamfer the hole, taper the end of the peg and use a Remote

Center Compliance (RCC) device, similar to that shown in Figure 1-1, between the robot end effector and the peg. Here, as the RCC has a high axial stiffness but low lateral stiffness, which combined with the chamfered hole, allows the peg to be inserted in the hole even with a slight positional misalignment.

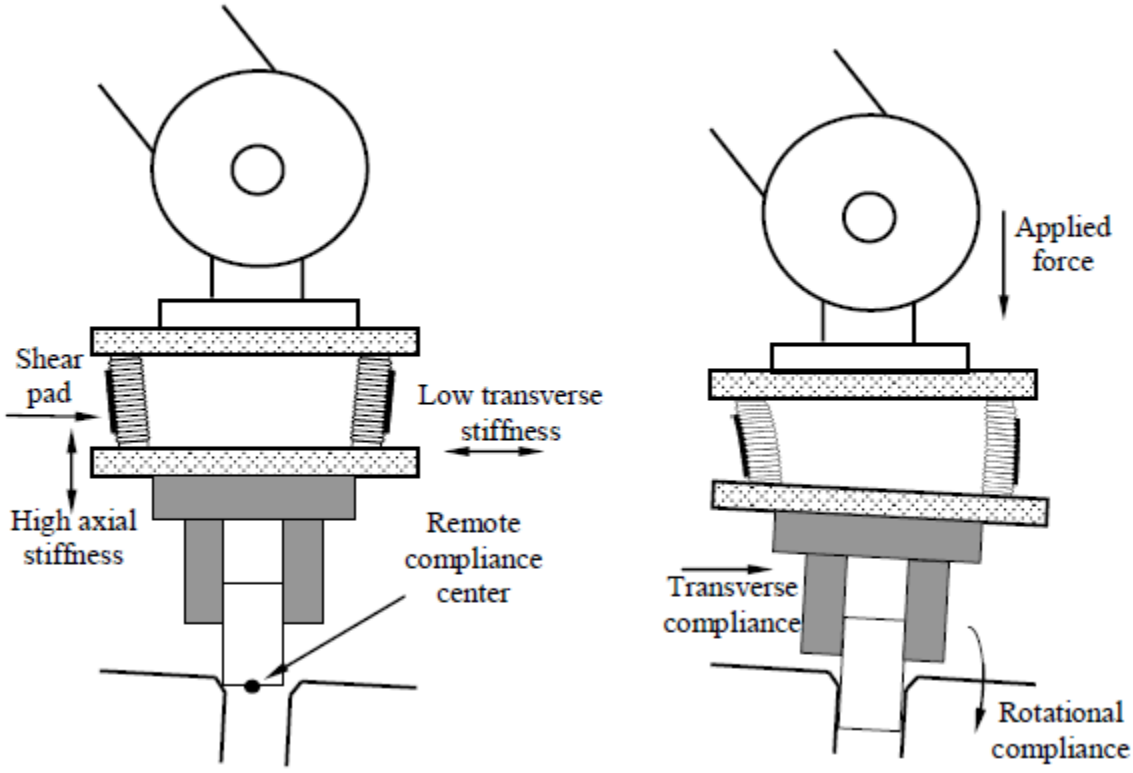


Figure 1-1. Remote Center Compliance Device [9]

Industrial robots generally have a serial manipulator configuration. A serial link configuration is an open-ended structure where each link is connected to the next one after the other. Thus serial manipulators have open kinematic loops. The end point of the manipulator can be taken to a specific position by varying the link parameters. A human arm is an example a serial manipulator. The kinematic diagrams of almost all industrial robots look similar to the one shown in Figure 1-2. It can be seen, that each of the joint actuator contributes to the effective wrench available at the end effector tool

mounting plate. Owing to the serial configuration, these manipulators have a low structural stiffness compared to a parallel mechanism.

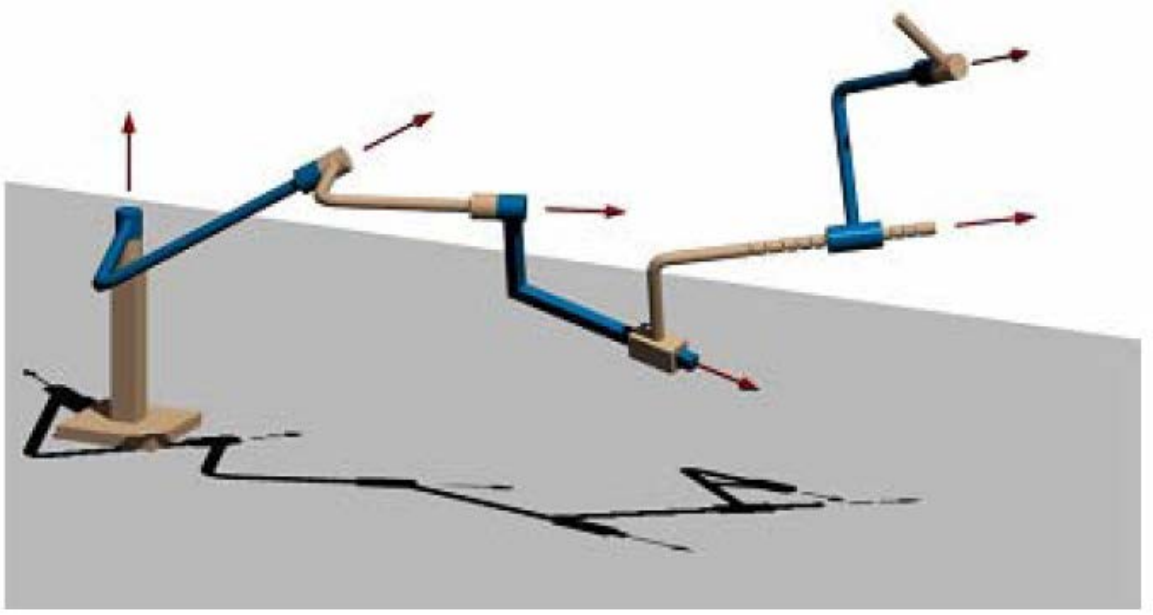


Figure 1-2. Kinematic structure of a serial industrial manipulator

A parallel manipulator is a closed-loop kinematic mechanism in which the top platform (also the end effector) is connected to the base platform by at least two independent kinematic chains as shown in Figure 1-3. These multiple closed loops improve the overall stiffness of the manipulator as the load is distributed among all the individual links. The end point positioning accuracy is also improved due to non accumulation of errors from one link to the other link. Thus these manipulators enjoy distinct advantages over serial manipulators in terms of size, stiffness, accuracy and load bearing capacity. But the major disadvantage of such manipulators is the high degree of complexity of the kinematic analysis and actuation limits of the actuators.

Thus the parallel mechanism can be used as a good counterpart by adding it to the serial manipulator, effectively utilizing the advantages of both the manipulators and reducing the effective disadvantages.

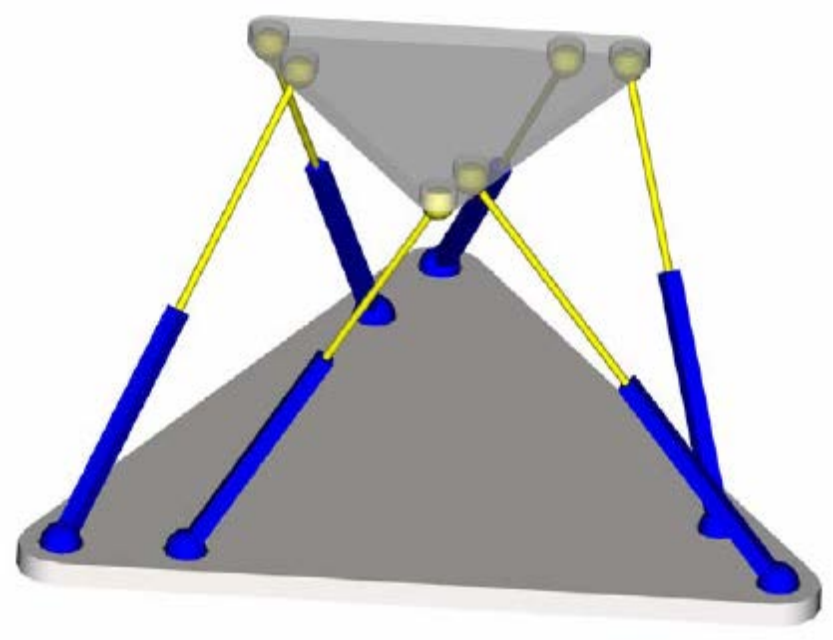


Figure 1-3. Parallel manipulator [9]

This type of a setup can be used for force control applications, demonstrated in this thesis. The force control method demonstrated in this thesis uses a commercially available serial industrial manipulator augmented with a compliant in-parallel platform attached on the end-effector to measure and thus control the contact forces. This setup is shown in the Figure 1-4.

The forward analysis of a manipulator is the process of finding the position and orientation of the end effector measured relative to a coordinate system fixed to the ground for a specified set of joint variables. The result can be represented as a transformation matrix. The reverse or inverse analysis deals with determining a set of joint variables that will position and orient the end effector as desired. The forward analysis of serial manipulators is quiet simple and straightforward while the reverse analysis is very complex and often requires the solution of multiple non-linear equations to obtain multiple solution sets [2]. But for a parallel mechanism, it is exactly the

opposite way. The reverse analysis being simple, while the forward analysis being complex.

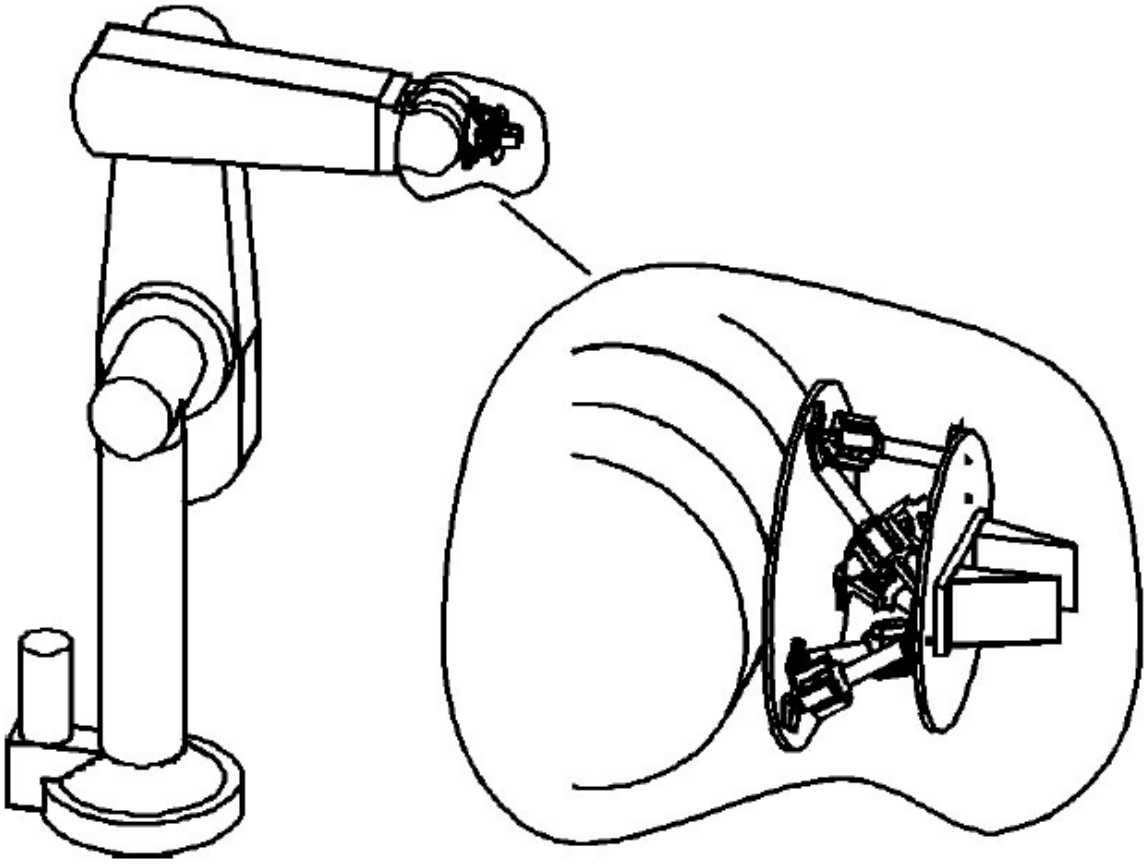


Figure 1-4. 6 DOF compliant parallel platform fitted between end-effector and tool mounting plate [9]

The first parallel spatial robot is credited to Pollard's five DOF parallel spray painting manipulator that had three branches. This manipulator was never actually built. In the late 1950s, Dr. Eric Gough invented the first well-known octahedral hexpod with six struts symmetrically forming an octahedron called the universal tire testing machine to respond to the problem of aero-landing loads. Then in 1965, Stewart published his paper of designing a parallel-actuated mechanisms as a 6 DOF flight simulator, which is different from the octahedral hexpod and widely referred to as the "Stewart platform".

Stewart's paper gained much attention and has had a great impact on the subsequent development of parallel mechanisms. Since then, much work has been done in the field of parallel geometry and kinematics such as geometric analysis, kinematics and statics, and parallel dynamics and controls. [9]

The most commonly used 6 DOF parallel mechanism consists of two rigid bodies connected through six identical leg connectors with six extensible legs each with spherical joints at both ends or with a spherical joint at one end and a universal joint at the other. This geometric arrangement is also known as a 6-6 Parallel Kinematic Mechanism (PKM) or "6-6 platform". The six joint points on the base platform are often but not necessarily located on one plane and are arranged in some symmetric pattern. Besides the general 6-6 parallel platform, there are some other configurations. One common configuration of a parallel mechanism is the 3-3 parallel manipulator as shown in Figure 1-5. The 3-3 parallel platform also has six connector legs, but each leg shares one joint point with another leg on the base platform and similarly on the top platform. The three shared joint points form a triangle on both the planer top platform. The three shared joint points form a triangle on both the planer top platform and the planar base platform. The mobility of a general spatial mechanism can be calculated using the following Equation 1-1 [14]

$$M = \lambda(n-1) - \sum_{i=1}^{j} (\lambda - f_i) \tag{1-1}$$

where,

M : Mobility or number of degree of freedom of the system

λ : Degrees of freedom of the space in which a mechanism is intended to function, for the spatial case, $\lambda = 6$

n : Number of links in the mechanism, including the fixed link

j : Numbers of joints in a mechanism, assuming that all the joints are binary

f_i : Degrees of relative motion permitted by joint i

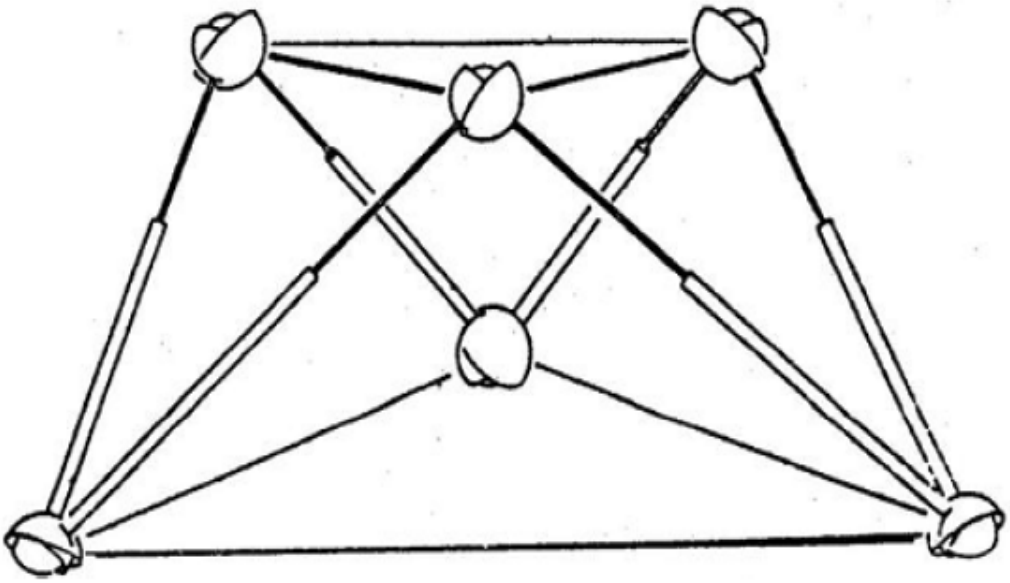


Figure 1-5. 3-3 parallel platform [6]

The mobility of the 3-3 platfrom shown in Figure 1-5 is 12 since in this case $\lambda = 6$, n=14 and there are 12 joints that allow 3 DOF and 6 joints that allow 1 DOF. Six of the degrees of freedom are trivial, i.e. each leg can rotate about its own axis due to having ball and socket joints at each end. Replacing the top leg joints with universal joints reduces the total mobility to six.

CHAPTER 2
LITERATURE REVIEW

2.1 Parallel Platform

2.1.1 Forward Kinematic Analysis

A parallel platform has legs which connect the top mobile platform to the base platform. For the case to be considered here, these legs are actuated by prismatic joints connected to the platform via a spherical or a universal joint. To control the position and orientation of the top platform, these leg lengths need to be measured and controlled. This process of finding the position and orientation of the top platform with respect to the bottom platform coordinates by measuring the leg lengths is called the forward kinematic analysis.

The forward kinematics of a structure with leg connectors sharing a common spherical joint on the top platform and similarly on the base platform, is easier to determine than for the general case. The simplest case being a 3-3 platform, in which the top platform and base platforms have 3 spherical joints each and a pair or leg connectors shares a joint. This is however is difficult to manufacture as the legs would collide with themselves. The forward analysis of this device was first solved by Griffis and Duffy [7] who showed how the position and orientation of the top platform could be established with respect to the base platform coordinates, given the geometry of the structure and the leg lengths. The solution for this is a closed-form solution based on the analysis of a three spherical four bar mechanism. There can be up to 16 real solutions, i.e. 8 solutions and their reflected images about the plane formed by the base connector points [7]

The geometrical method, patented by Duffy and Griffis [15] for determining the equivalent 3-3 parallel structure for a special 6-6 configuration mechanism has been used for the forward kinematic analysis of the parallel platform implemented in this thesis. The configuration of a special 6-6 parallel platform (Figure 2-1) was developed by Griffis and Duffy and fabricated by Bo Zhang [9] such that the forward kinematic analysis can use a method similar to the method for the 3-3 parallel platform. The legs on this structure are Spherical-Prismatic-Hooke (SPH) chains which connect the top platform with the bottom platform. The relationship between this Special 6-6 platform and its equivalent 3-3 platform was discovered by Griffis and Duffy.

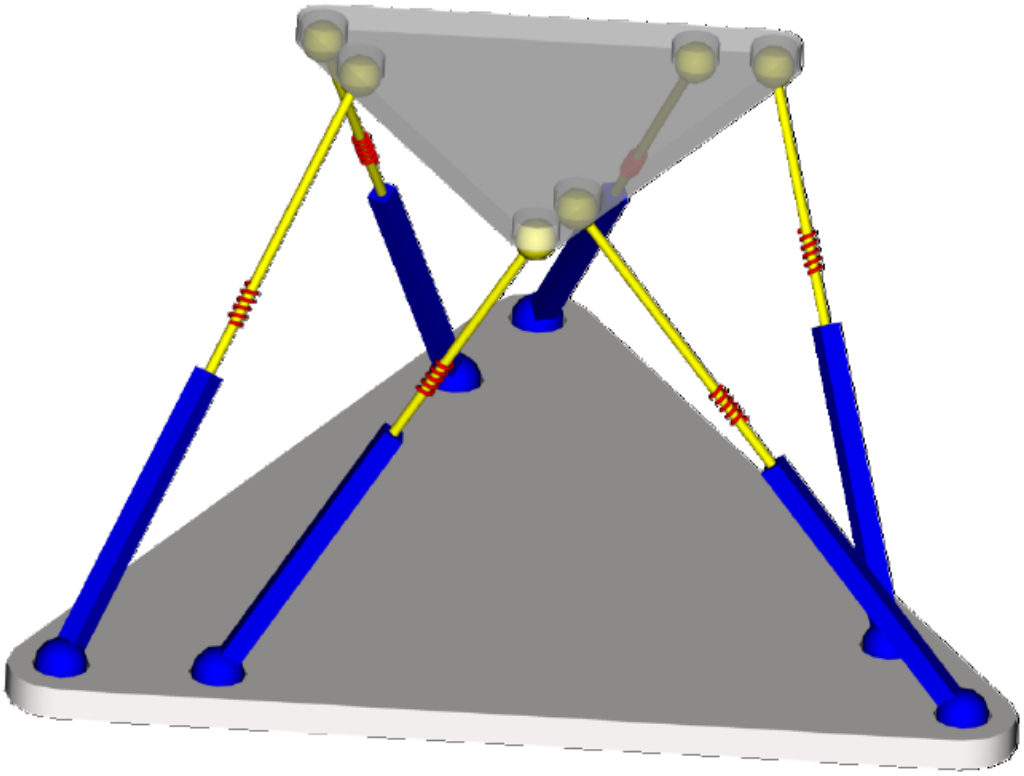


Figure 2-1. The Special 6-6 platform

A Special 6-6 platform is defined as the platform mechanism which can be geometrically reducible to an equivalent 3-3 platform. Figures 2-2 and 2-3 depict a

perspective and plan view of the 6-6 platform respectively where the leg connector points R_0, S_0 and T_0 lie along the edges of the triangle defined by points O_0, P_0 and Q_0 and the leg connector points O_1, P_1 and Q_1 lie along the edges of the triangle defined by the points R_1, S_1 and T_1. The objective is to determine the distance between the pairs of points O_0 - R_1, O_0 - S_1, P_0 - S_1, P_0 -T_1, Q_0 - T_1 and Q_0 - R_1. These distances are leg lengths for an equivalent 3-3 platform.[9]

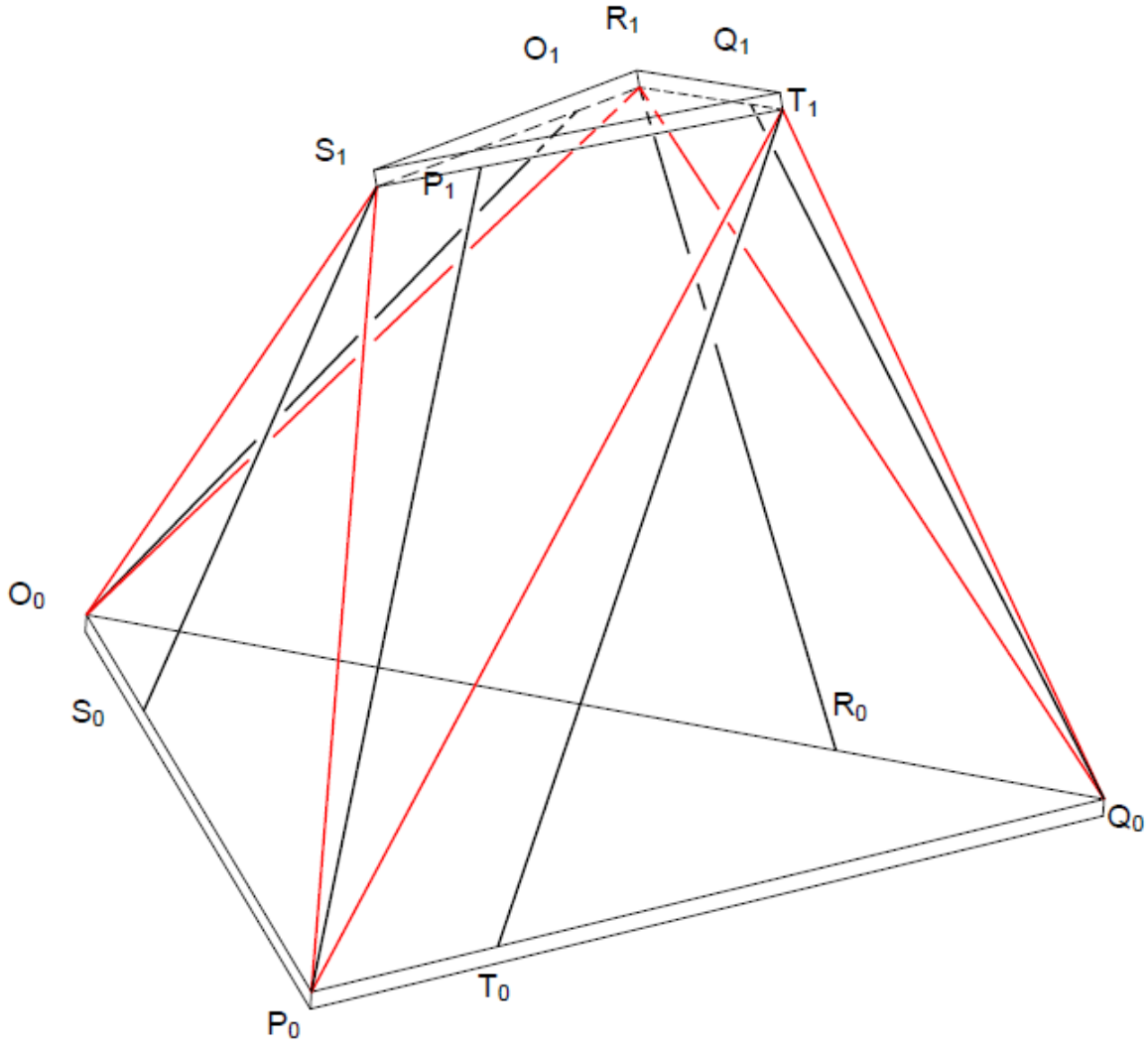


Figure 2-2. Perspective view of the special 6-6 platform [9]

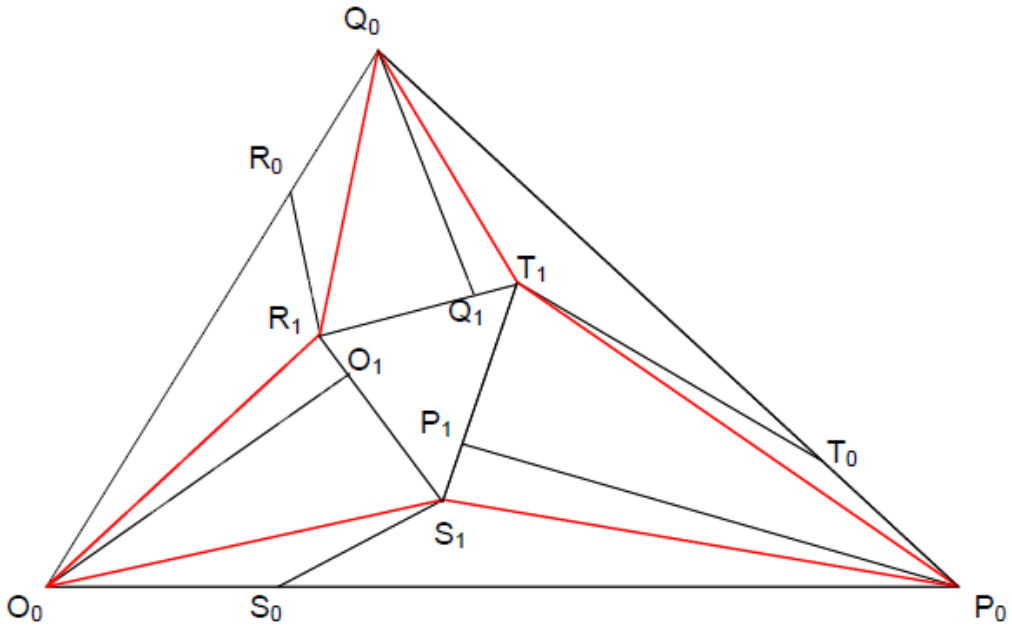


Figure 2-3. Plan view of special 6-6 platform [9]

Throughout this analysis, the notation $m_i n_j$ will be used to represent the distance between the two points M_i and N_j and the notation $\mathbf{m_i n_j}$ will represent the vector from point M_i to N_j. Using this notation, the problem statement is defined as:

Given: o_0o_1, p_0p_1, q_0q_1, r_0r_1, s_0s_1, t_0t_1 ; *connector lengths for Special 6-6 platform*

o_0p_0, p_0q_0, q_0o_0, o_0s_0, p_0t_0, q_0r_0 ; *base triangle parameters*

r_1s_1, s_1t_1, t_1r_1, r_1o_1, s_1p_1, t_1q_1 ; *top triangle parameters*

Find: o_0r_1, o_0s_1, p_0s_1, p_0t_1, q_0t_1, q_0r_1 ; *connector lengths for equivalent 3-3 platform*

The solution to this problem was developed by Griffis and Duffy [15].

Once the leg length dimensions of the equivalent 3-3 platform are determined, the forward analysis of the 3-3 device is used to determine the position and orientation of the top platform relative to the base.

2.1.2 Forward and Reverse Static Analysis

The concept of wrench from screw theory, which was introduced by Ball is employed to describe a force/torque applied to a body [1]. The forward static analysis is defined as computing the resultant wrench $\hat{w}$ = {**f** ; $\mathbf{m_0}$} due to the six forces generated in the legs acting upon the top platform. Now $\hat{w}$ can be expressed in the form:

$$\hat{w} = \{\mathbf{f_1} ; \mathbf{m_{01}}\} + \{\mathbf{f_2} ; \mathbf{m_{02}}\} + \ldots\ldots\ldots\ldots\ldots + \{\mathbf{f_6} ; \mathbf{m_{06}}\} \quad (2\text{-}2)$$

or

$$\hat{w} = f_1\{\mathbf{S_1} ; \mathbf{S_{0L1}}\} + f_2\{\mathbf{S_2} ; \mathbf{S_{0L2}}\} + \ldots\ldots\ldots\ldots\ldots + f_6\{\mathbf{S_6} ; \mathbf{S_{0L6}}\} \quad (2\text{-}3)$$

where {$\mathbf{S_i}$; $\mathbf{S_{0Li}}$}, i=1….6 are the Plücker coordinates of the line along the six legs. Refer to Appendix – A for coordinates of a line. It is convenient to express (2-2) in the form

$$\hat{w} = f_1\begin{bmatrix} S_1 \\ S_{0L1} \end{bmatrix} + f_2\begin{bmatrix} S_2 \\ S_{0L2} \end{bmatrix} + \ldots\ldots\ldots\ldots + f_6\begin{bmatrix} S_6 \\ S_{0L6} \end{bmatrix} \quad (2\text{-}4)$$

which may again be written as

$$\hat{w} = \mathbf{j}\,\lambda \quad (2\text{-}5)$$

where **j** is a 6 x 6 matrix called as the Jacobian matrix and is given as

$$\mathbf{j} = \begin{bmatrix} S_1 & S_2 & S_3 & S_4 & S_5 & S_6 \\ S_{0L1} & S_{0L2} & S_{0L3} & S_{0L4} & S_{0L5} & S_{0L6} \end{bmatrix} \quad (2\text{-}6)$$

and λ is a column vector given as

$$\lambda = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{bmatrix} \quad (2\text{-}7)$$

j maps the force magnitudes along each leg to the externally applied wrench.[1]

The geometry of the platform is solved first by knowing each of the leg lengths, thus the **j** matrix is known. Furthermore, the magnitude $f_i, i = 1...6$ i.e. force magnitude in each of the legs is known from the leg lengths and the spring constant and spring free length for each of the legs. Thus, the resultant wrench can be computed from (2-5). Clearly for equilibrium, an external wrench with an equal magnitude $|\mathbf{f}|$ and opposite direction must be applied to the platform.

Conversely, when the position and orientation of the top platform is known relative to the base and an external wrench or force is applied to the top platform, it is required to determine the magnitudes $f_i, i = 1...6$ of the connector forces. This is called the reverse or inverse static analysis. This is accomplished by solving for **λ** as

$$\lambda = j^{-1}\,\hat{w} \tag{2-8}$$

The point to note is, this computation cannot be performed when the rank of **j** is less than six for which the Plücker coordinates of the lines along the six legs become linearly dependant and the platform is said to be in a singular position [1].

2.1.3 Compliance Analysis

A derivative of the above Equation 2-8 will yield a relationship between the changes in individual leg forces to the change in the externally applied wrench. Griffis [8] extended this analysis to show how the change in the externally applied wrench could be mapped to the instantaneous motion of the top platform. According to the static force analysis presented above, a compliant parallel platform to detect forces and torques was designed by Bo Zhang [9]. Once the forward position analysis is completed to determine the current pose, with the known stiffness constant and free length of each

connector and measurement of the current leg lengths, the external wrench acting on the platform can be determined.

Based on this, Griffis and Duffy introduced the theory of kinestatic control to simultaneously control force and displacement for a certain constrained manipulator based on the general spatial stiffness of a compliant coupling. [9] In this theoretical analysis of the compliant control strategy, a model of a passive platform with compliant legs was utilized to describe the spatial stiffness of the parallel platform based wrench sensor. Compared to the open loop Remote Center Compliance (RCC) compensators that are commercially available (Figure 1-1, 2-4 and 2-5), the parallel-platform-based force-torque sensor can provide additional information about the external wrench to assist force and position control.

Figure 2-4. ATI industrial automation 9116 series RCC compensator (Reprinted with permission from ATI Industrial Automation)

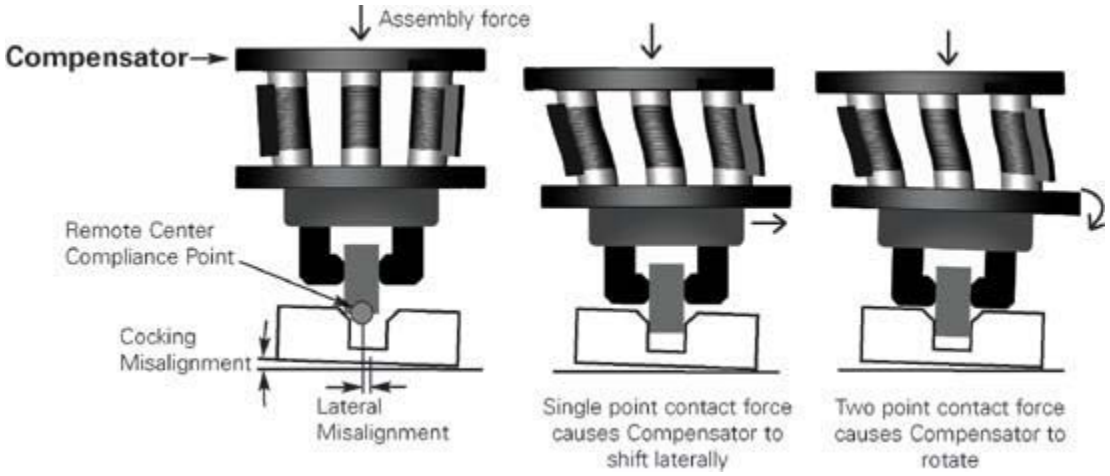


Figure 2-5. ATI industrial automation 9116 Series RCC compensator is designed to be used in "peg-in-hole"-type operations.

Dwarakanath and Crane studied a parallel platform based wrench sensor, using a LVDT (Linear Variable Displacement sensor) and using a potentiometer with a slider crank mechanism to convert axial deflection of the spring to angular deflection. [3]

Screw theory is a very powerful tool to investigate the compliance or stiffness characteristic of a compliant device. This theory was first introduced by Ball to describe the general motion of a rigid body. He presented the idea of the principal screw of a rigid body, where a wrench applied along a principal screw of inertia generates a twist on the same screw.[9] Duffy analyzed planer parallel spring systems theoretically. [4] Griffis and Duffy studied the non-symmetric stiffness behavior for a special octahedron parallel platform with 6 springs as the connectors and the stiffness mapping could be represented by a 6 x 6 matrix called the Stiffness Matrix. [9]

The basic property of a compliant component like the spring is its rigidity. Practically, no object is infinitely stiff; the only difference between a “compliant device” and a “rigid component” is that when the compliant device is compared to the rigid component, the stiffness of the compliant device is much lesser than the rigid device. A

spring is most commonly used to introduce compliance in a compliant device. The stiffness property of a spring is defined by the property of the spring called spring constant, is used to describe compliant devices.

2.1.3.1 Simple Planer Case Stiffness Analysis

Hooke's Law describes the fundamental relationship between an external force and the compliant displacement in static equilibrium.

$$F_s = -k\Delta l = -k\left(l - l_0\right) \tag{2-9}$$

where F_s is the force exerted on the spring (or more generally, the compliant component), *k* is the spring constant, l is the length of the spring when acted upon an external force, l_0 is the free length and Δl is the resultant displacement of spring from its free state.

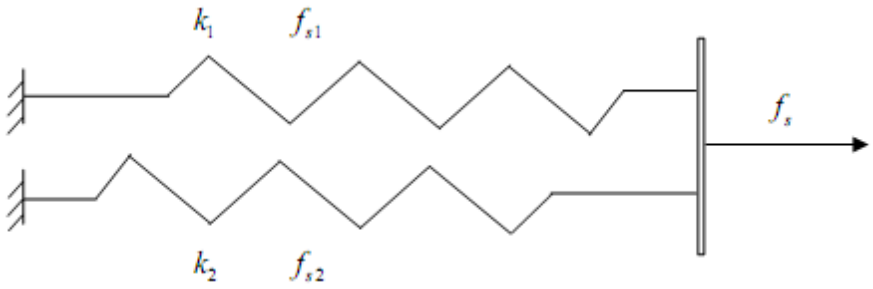


Figure 2-6. Planer in-parallel springs

Consider two springs connected in-parallel, with spring constants k_1 and k_2 respectively. An external force is applied on the right end of the two springs and in the direction along the axes of the springs. (Figure 2-6)

The force in each spring can be calculated as F_{s1}=-k_1 Δl_1 and F_{s2}=-k_2 Δl_2 but $\Delta l_1 = \Delta l_2 = \Delta l$. Hence, the total force applied is sum of the two spring forces and can be written as

$$F_s=F_{s1}+F_{s2}=-(k_1+k_2)\,\Delta l=-k_p\,\Delta l \qquad (2\text{-}10)$$

where k_p is the equivalent stiffness/spring constant of the in-parallel connected springs. It is the sum of the individual spring constants. Thus it is seen that when compliant components are connected in parallel, the resultant stiffness is greater than the stiffness of each of the independent components.

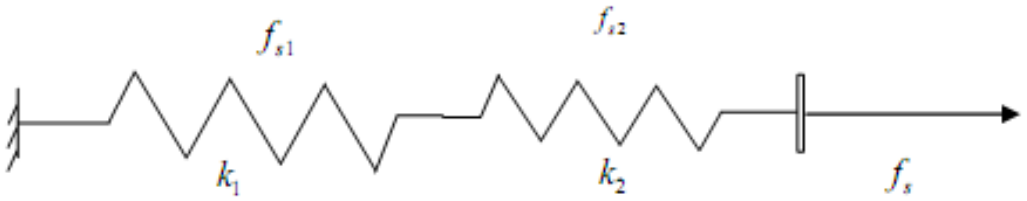


Figure 2-7. Planar serially connected springs

Consider a case where two springs are connected in series instead of parallel. The axes of the two springs are collinear and they are connected end to end, thus the direction of the external force applied on them is also along the axes of the springs as shown in Figure 2-7 above. The external force when applied to such a setup, is felt by each spring in an equal magnitude. $F_{s1}=-k_1\,\Delta l_1$ and $F_{s2}=-k_2\,\Delta l_2$. Here, $\Delta l = \Delta l_1 + \Delta l_2$. Hence, the total force can be written as

$$F_s=F_{s1}=F_{s2}=-k_s\,\Delta l=-k_s\,(\Delta l_1+\Delta l_2) \qquad (2\text{-}11)$$

Where k_s is the given by $\frac{1}{k_s}=\frac{1}{k_1}+\frac{1}{k_2}$ and is called as the equivalent stiffness or spring constant of the serially connected springs. Hence we see that when compliant components are connected serially, the overall equivalent stiffness is less than the stiffness of each of the independent compliant component.

For a case where $k_1 = k_2 = k$, $k_s = \frac{k}{2}$ and if $k_1 \ll k_2$, $k_s \cong k_1$ shows that serially connected components with widely different spring constants, have an equivalent stiffness, which is more dependent on the component with the smaller stiffness or spring constant.

From this result, it is sen that if one compliant component is connected to a relatively stiff bar, the equivalent stiffness of this two-component system is very close to the stiffness of the compliant component. The systems discussed above have one Degree of Freedom (DOF). A more general planar case would have two or more DOF. Figure 2-8 shows a planar 2 DOF spring case. In this spring system, two translational springs are connected at one end P and grounded separately at pivot points A and B respectively. Here translational springs behave like prismatic joints in revolute-prismatic-revolute (RPR) serial chains. In the X-Y plane, two such springs form a simple compliant coupling. The two-spring compliant coupling system is equivalent to a planar two-dimensional spring. The spring is two-dimensional because two independent forces act in the translational spring, and it is planar since the forces remain in a plane. The external force applied at point P is in static equilibrium with the forces acting in the springs. The two-dimensional spring remains in quasi-static equilibrium as the point P moves gradually. In order to analyze the two-dimensional force/displacement relationship or mapping of stiffness, it is necessary to decompose both the external force and displacements into standard Cartesian coordinate vectors. The locations of points A, B, and P, the initial and current lengths of AP and BP and the angles θ_1 and θ_2 are known. The free length of AP is l_{01} and the free length of BP is l_{02}. The spring constants are k_1 and k_2. The current length of AP and BP are l_1 and l_2 respectively. To

simplify the equations, dimensionless parameters $\rho_1 = \frac{l_{01}}{l_1}$ and $\rho_2 = \frac{l_{02}}{l_2}$ are introduced.

By definition, these two scalar values are always positive, and no negative spring lengths are allowed. When ρ > 1, the corresponding spring is elongated, and if ρ < 1, then it is compressed.

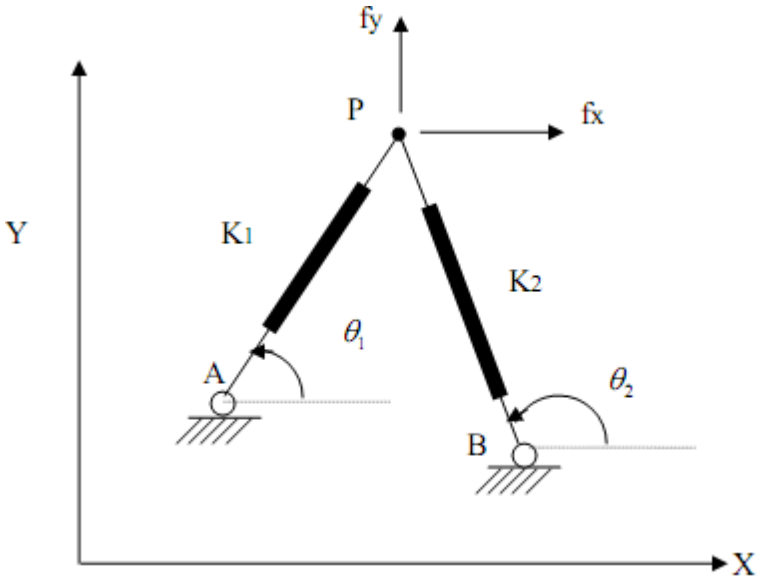


Figure 2-8. Planar 2 DOF spring

The external force applied on the spring system is given by:

$$\begin{bmatrix} f_x \\ f_y \end{bmatrix} = \begin{bmatrix} c_1 & c_2 \\ s_1 & s_2 \end{bmatrix} \begin{bmatrix} k_1(1-\rho_1)l_1 \\ k_2(1-\rho_2)l_2 \end{bmatrix} \tag{2-12}$$

Where $c_i = \cos(\theta_i)$ and $s_i = \sin(\theta_i)$. Differentiating the above Equation 2-12 will result in the following Equation 2-13

$$\begin{bmatrix} \delta f_x \\ \delta f_y \end{bmatrix} = [k] \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \tag{2-13}$$

Where [k] is the mapping of the stiffness of the system and according to Griffis can be written as: [6]

$$[k]=\begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}=\begin{bmatrix} c_1 & c_2 \\ s_1 & s_2 \end{bmatrix}\begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix}\begin{bmatrix} c_1 & s_1 \\ c_2 & s_2 \end{bmatrix} + \begin{bmatrix} -s_1 & -s_2 \\ c_1 & c_2 \end{bmatrix}\begin{bmatrix} k_1(1-\rho_1) & 0 \\ 0 & k_2(1-\rho_2) \end{bmatrix}\begin{bmatrix} -s_1 & c_1 \\ -s_2 & c_2 \end{bmatrix} \quad (2\text{-}14)$$

The Spring Stiffness Matrix $[k]$can be written in the form

$$[k]=[j][k_i][j]^T+[\delta j]\left[k_i(1-\rho_i)\right][\delta j]^T \quad (2\text{-}15)$$

where $[j]$is the static Jacobian matrix of the system, $[\delta j]$ is the differential matrix with respect to θ_1 and θ_2 while $[k]$ and $\left[k(1-\rho)\right]$ are 2 x 2 diagonal matrices. In general, $\theta_1 \neq \theta_2$ because if the two angles are equal then the spring matrix expression becomes singular. For such a case, the two springs are parallel and Equation 2-11 can be applied instead of Equation 2-14. [9]

The concept of twist from screw theory, which was introduced by Ball is employed to describe the small instantaneous displacement of a rigid body when acted upon by an external wrench. In order to maintain the system equilibrium of the structure, the top platform moves as the external wrench changes and aligns in the direction of this external wrench. Thus, the mapping of stiffness is a one-to-one correspondence that associates the twist describing the relative displacement between the bodies with the corresponding resultant wrench which interacts between them. This relationship is given by [14]

$$\delta\hat{w}=[K]\delta\hat{D} \quad (2\text{-}16)$$

The detailed derivation process was done by Duffy [4]; only the result has been shown here. [K] is the compliance matrix, $\delta\hat{w} = \left[\delta f \cdot \delta m\right]^T$ is the change of the wrench, and $\delta\hat{D} = \left[\delta X ; \delta\phi\right]^T$ is the change in position and orientation.

2.1.3.2 Stiffness Analysis for general spatial systems

Complaint Mechanisms can be considered as spatial springs with multiple DOF rather than one as linear spring have. It was first shown by Griffis that the compliance matrix for the special 6-6 parallel mechanism may be written as the sum of five matrices as [8]

$$\begin{aligned}\left[K\right] = {} & \left[j\right]\left[k_i\right]\left[j\right]^T + \left[\delta j_\theta\right]\left[k_i\left(1-\rho_i\right)\right]\left[\delta j_\theta\right]^T + \left[\delta j_\alpha\right]\left[k_i\left(1-\rho_i\right)\right]\left[\delta j_\alpha\right]^T \\ & + \left[\delta j_\theta\right]\left[k_i\left(1-\rho_i\right)\right]\left[V_\theta\right]^T + \left[\delta j_\alpha\right]\left[k_i\left(1-\rho_i\right)\right]\left[V_\alpha\right]^T\end{aligned} \tag{2-17}$$

where [j] is a 6 x 6 matrix whose columns are the Plücker coordinates of the lines along the six leg connectors and [k_i] is a diagonal 6 x 6 matrix whose diagonal elements are the spring constants of the six leg connectors. The term ρ_1 is defined as

$$\rho_i = \frac{l_{0i}}{l_i} \tag{2-18}$$

which is the ratio of the free length of connector i to the current length of the connector and the term $\left[k_i\left(1-\rho_i\right)\right]$ is a diagonal 6 X 6 matrix whose diagonal elements are given by $k_i\left(1-\rho_i\right)$.The terms $\left[\delta j_\theta\right]$ and $\left[\delta j_\alpha\right]$ are each 6 X 6 matrices whose columns are the derivatives of the Plücker coordinates of the lines along the leg connectors taken with respect to θ_i and α_i which are angles which define the direction of

the line along leg *i*. The remaining terms to be defined in Equation 2-17 are $[V_\theta]$ and $[V_\alpha]$.

Next, six unit vectors are defined as intermediate variables describing the directional information of the base platform triangle sides as

$$u_1 = \frac{-EA}{\| EA \|}, u_2 = \frac{AC}{\| AC \|}, u_3 = \frac{-AC}{\| AC \|}, u_4 = \frac{-EC}{\| EC \|}, u_5 = \frac{EC}{\| AC \|}, u_6 = \frac{EA}{\| EA \|} \quad (2\text{-}19)$$

where ***EA***, ***EC*** and ***AC*** are the vectors on the base platform triangle shown in Figure 2-9 V_i is given by the expression

$$V_i = \frac{u_i \times S_i}{\| u_i \times S_i \|} \quad (2\text{-}20)$$

$$\delta S_i^\theta = V_i \quad \text{and} \quad \delta S_i^\alpha = V_i \times S_i \quad (2\text{-}21)$$

$$[V_\alpha] = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ l_1 V_1 & l_2 V_2 & l_3 V_3 & l_4 V_4 & l_5 V_5 & l_6 V_6 \end{bmatrix} \quad (2\text{-}22)$$

$$[V_\theta] = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ l_1 S_1 \times V_1 & l_2 S_2 \times V_2 & l_3 S_3 \times V_3 & l_4 S_4 \times V_4 & l_5 S_5 \times V_5 & l_6 S_6 \times V_6 \end{bmatrix} \quad (2\text{-}23)$$

Substituting all the necessary components in Equation 2-17 yields the global stiffness matrix of the special 6-6 parallel passive mechanism via the stiffness mapping analysis.[9]

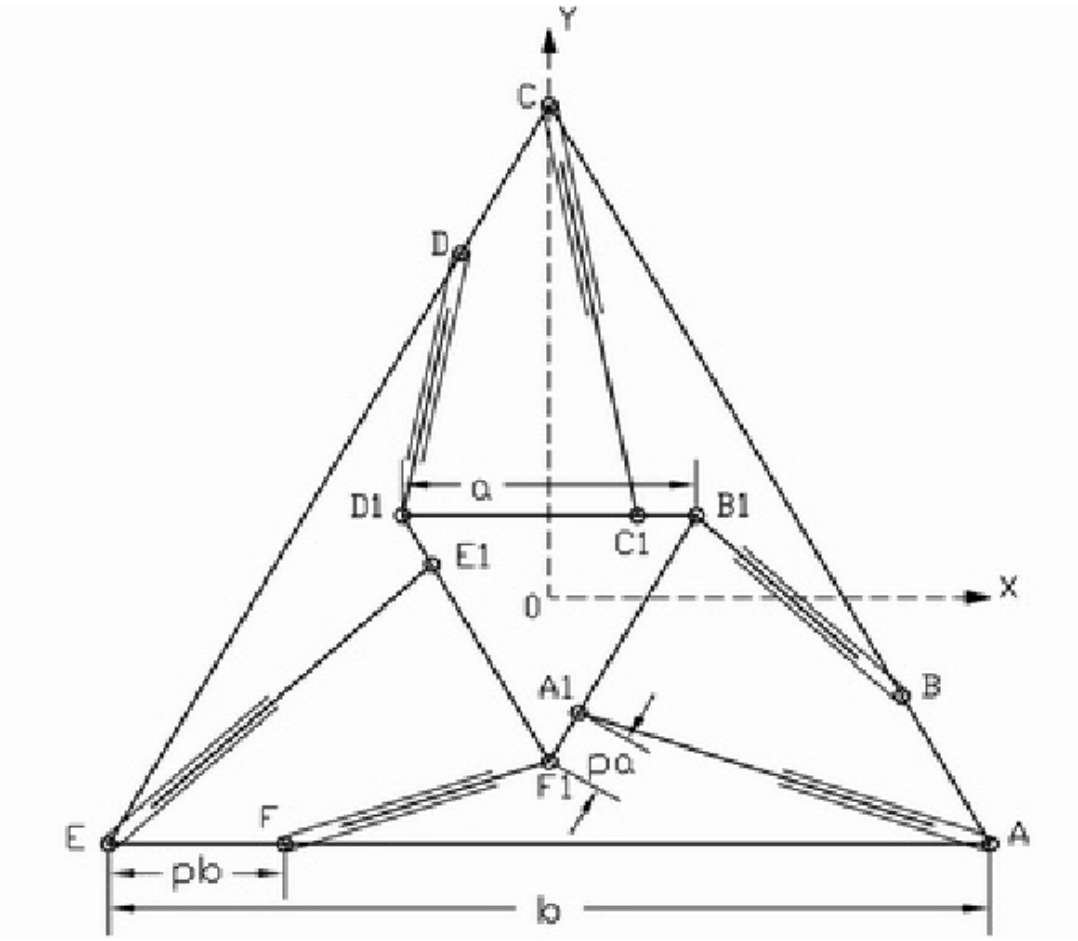


Figure 2-9. Top view of the special 6-6 configuration

2.1.3.3 Theory of Kinestatic Control

The theory of Kinestatic Control was proposed by Griffis and Duffy in 1991. [9] In general, the spatial stiffness of a compliant coupling that connects a pair of rigid bodies is used to map a small twist between the bodies into the corresponding interactive wrench. This mapping is based upon a firm geometrical foundation and establishes a positive-definite inner product (elliptic metric) that decomposes a general twist into a twist of freedom and a twist of compliance.

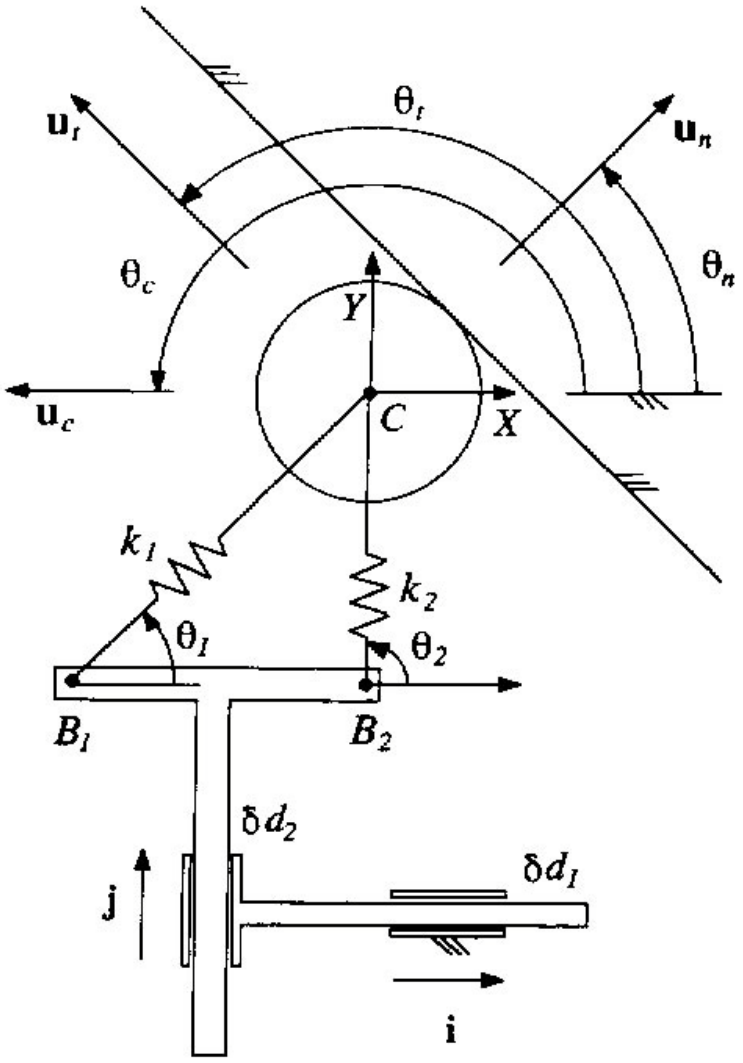


Figure 2-10. Passive compliance device for contact force control

A planar example is shown in Figure 2-10. A serial pair of actuated prismatic joints supports a wheel via a two-spring system. The actuated prismatic joints are controlled so that the wheel can maintain a desired contact with a rigid wall. [6] [4]. The objective of this problem is to control the contact force arising between the wheel and the wall when the wheel slides along the surface of the rigid wall. The serial pair of actuated prismatic joints supports the body B_1B_2, which connects with the wheel via two compliant connectors (B_1C and B_2C). The actuator drives the body with pure motion in

the *i* and *j* directions to adjust the position of the wheel along the surface and the contact force between the wheel and the surface.

Given the compliant proprieties of the two connectors (spring constants k_1 and k_2 and free lengths L_{01} and L_{02} for B_1C and B_2C respectively) and the geometry of the mechanism (angle θ_1 and θ_2), the control objective is to determine the displacement changes for the two prismatic actuators, i.e. δd_1 and δd_2 in order to achieve a desired position of the wheel and contact force between the wheel and the surface.

The result of this work was that Griffis and Duffy proved that the instantaneous compliance matrix which relates a change in the applied wrench (force for the planar example) to the relative displacement of the compliant mechanism was not symmetric when an external wrench was applied across the compliant mechanism. It was shown that the change in force is related to change in position by the following Equation 2-24

$$\begin{bmatrix} \delta f_x \\ \delta f_y \end{bmatrix} = [k] \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \tag{2-24}$$

Where [K] is the Compliance Matrix of the system and can be written as

$$[K] = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} = \begin{bmatrix} c_1 & c_2 \\ s_1 & s_2 \end{bmatrix} \begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix} \begin{bmatrix} c_1 & s_1 \\ c_2 & s_2 \end{bmatrix} \tag{2-25}$$

By knowing the Compliance Matrix, the manipulator can do a pure position move of the end effector in order to control the position of the wheel along the wall and the contact force with the wall.

To apply such a Kinestatic control to the general spatial case, it is necessary that the compliant mechanism being used possess six degrees of freedom. The 6-6 parallel platform discussed earlier can be used for this application. This device can be inserted

between the tool mounting plate of a regular 6 DOF industrial manipulator and the end effector tooling as shown in Figure 1-4. By measuring the instantaneous leg lengths of the device, the force in each connector and thus the wrench currently being applied to the top platform can be determined. The objective then is to determine the pose of the manipulator which would orient it in the direction of the external wrench being applied. The process which needs to be implemented in the robot position controller is shown in Figure 2-11.

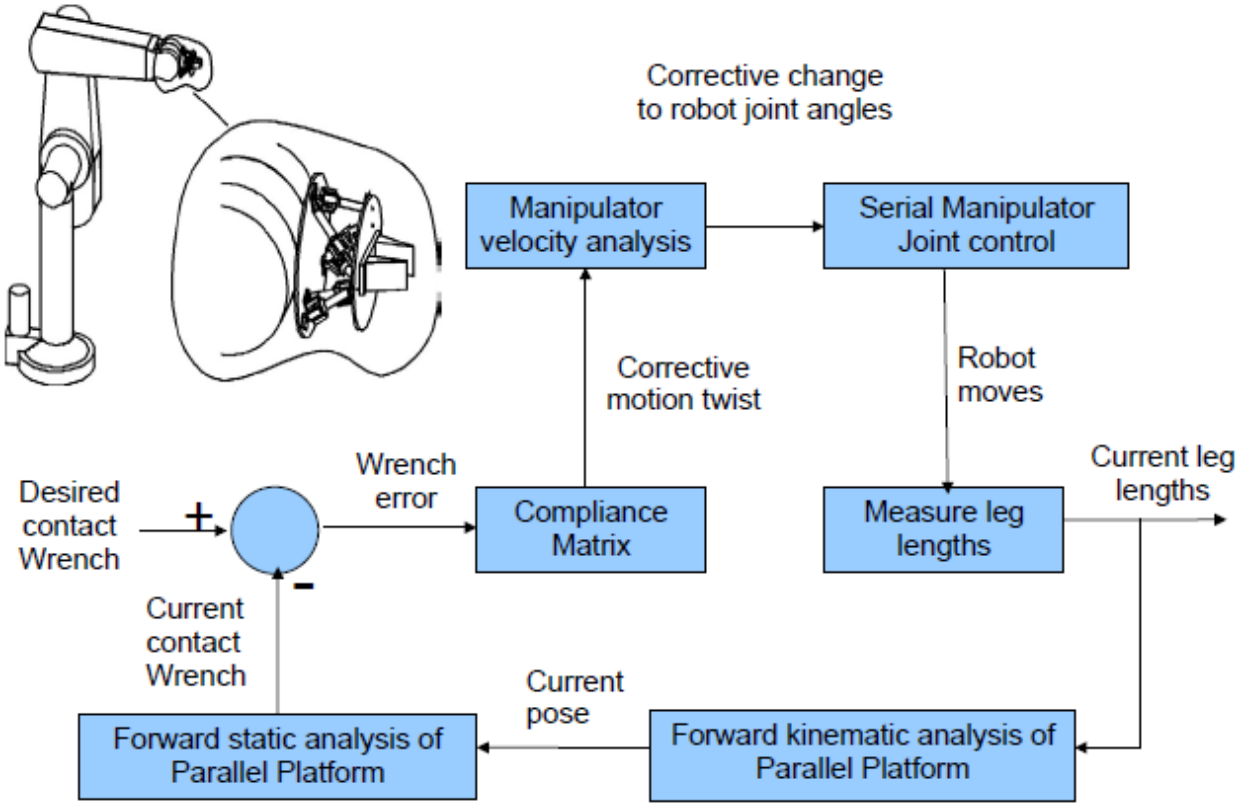


Figure 2-11. Kinestatic control: closed loop process scheme.[14]

2.2 Puma 762 Industrial Robot

The Puma robot arm can be compared to a human torso, shoulder, and wrist. It consists of members connected by six revolute joints, each defining an axis about which

the members rotate. The axes are equipped with limit-switch-shutoff-systems positioned 2 degrees past the software stops providing additional safety. The Puma 762 weighs 590 kg and has a maximum static payload of 20 kg. Table 2-1 lists useful mechanical specifications of the system.

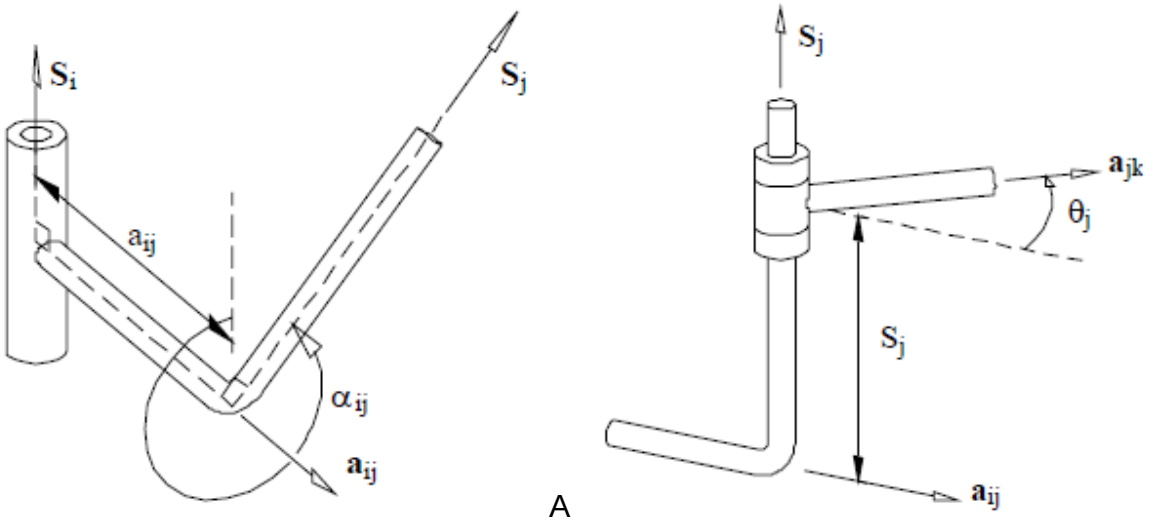
Figure 2-12 shows detailed definitions of vectors and parameters used to label the revolute joints. The mechanism parameters listed in Table 2-2 are used to create a kinematic chain shown in Figure 2-13. For a proper kinematic analysis of the manipulator, it is required that a coordinate system be assigned to each of the links. The coordinate system attached to link ij is called the i^{th} coordinate system [9]. Its origin is located at the intersection of vectors $\vec{S}_i$ and $\vec{a}_{ij}$; x-direction is along the vector $\vec{a}_{ij}$, and z-direction is along $\vec{S}_i$.[10]

Table 2-1. Mechanical specifications of the Puma 762 robot

Joint#	1	2	3	4	5	6
Software Movement Limits (deg)	320	220	270	532	220	532
Joint Angular Resolution (deg * 10^{-3})	5.0	3.5	4.5	12.5	6.2	13.4
Encoder Index Resolution equal to one motor revolution (deg)	4.0	2.78	3.57	6.26	6.26	13.4

Table 2-2. Mechanism parameters of the Puma 762 robot

Link Length, mm	Twist Angle, deg	Joint Offset, mm	Joint Angle, deg
$a_{12} = 0$	$\alpha_{12} = 90$		φ_1 = variable
$a_{23} = 650$	$\alpha_{23} = 0$	$S_2 = 190$	θ_2 = variable
$a_{34} = 0$	$\alpha_{34} = 270$	$S_3 = 0$	θ_3 = variable
$a_{45} = 0$	$\alpha_{45} = 90$	$S_4 = 600$	θ_4 = variable
$a_{56} = 0$	$\alpha_{56} = 90$	$S_5 = 0$	θ_5 = variable
			θ_6 = variable



A B

Figure 2-12. A) Manipulator linkage parameters for link ij B) Manipulator linkage parameters or revolute joint j [2]

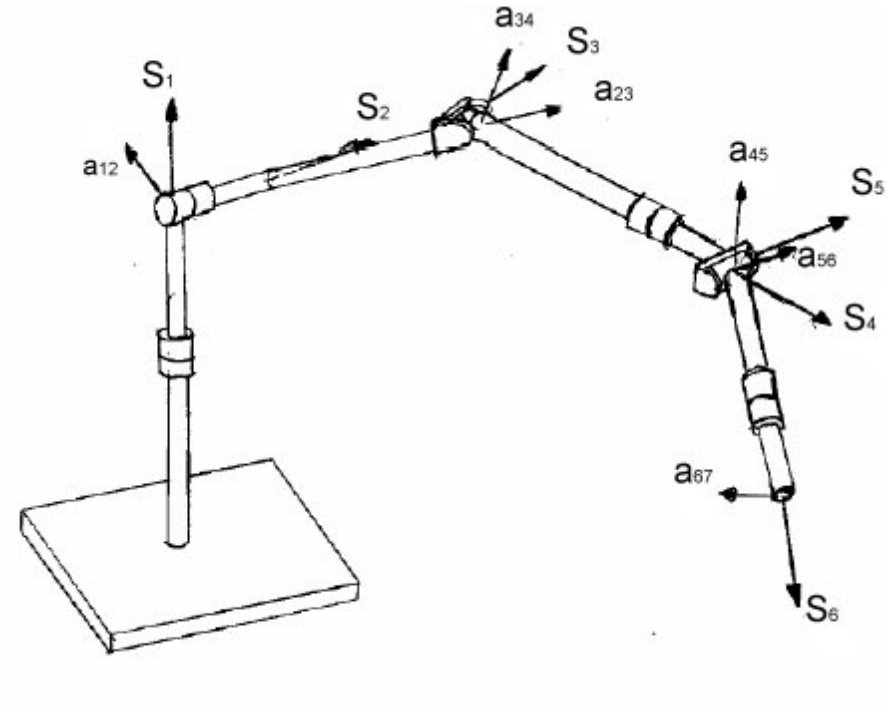


Figure 2-13. Labeled kinematic model of the Puma 762 manipulator [2]

The transformation matrix relating two of these coordinate systems in a three dimensional space is given by Equation 2-26.

$$ {}^{i}_{j}T = \begin{bmatrix} c_i & -s_j & 0 & a_{ij} \\ s_j c_{ij} & c_j c_{ij} & -s_{ij} & -s_{ij} S_j \\ s_j s_{ij} & c_j s_{ij} & c_{ij} & c_{ij} S_j \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-26} $$

The inverse of this transformation is very frequently used and is given by the following 4 x 4 transformation:

$$ {}^{j}_{i}T = \begin{bmatrix} c_j & s_j c_{ij} & s_j s_{ij} & -c_j a_{ij} \\ -s_j & c_j c_{ij} & c_j s_{ij} & s_j a_{ij} \\ 0 & -s_{ij} & c_{ij} & -S_j \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-27} $$

where a_{ij} represents a link length of a serial manipulator, s_i and c_i represent the sine and cosine of θ_i respectively. Furthermore, s_{ij} and c_{ij} represent the sine and cosine of α_{ij} . Finally, a fixed coordinate system is defined as having its origin coincident with the origin of the first coordinate system and its axis along the vector $\vec{S}_1$. The transformation that relates the first coordinate system and the fixed is given in the Equation 2-28.

$$ {}^{F}_{1}T = \begin{bmatrix} \cos\phi_1 & -\sin\phi_1 & 0 & 0 \\ \sin\phi_1 & \cos\phi_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-28} $$

General expressions determining the directions of joint vectors with respect to the first coordinate system are given in Equations 2-29 and 2-30.

$$ {}^{1}\vec{S}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad {}^{1}\vec{S}_2 = \begin{bmatrix} 0 \\ -s_{12} \\ c_{12} \end{bmatrix} \quad {}^{1}\vec{S}_3 = \begin{bmatrix} \overrightarrow{X_2} \\ \overrightarrow{Y_2} \\ \overrightarrow{Z_2} \end{bmatrix} \quad {}^{1}\vec{S}_n = \begin{bmatrix} X_{n-1,n-2,\ldots.2} \\ Y_{n-1,n-2,\ldots.2} \\ Z_{n-1,n-2,\ldots.2} \end{bmatrix} \tag{2-29} $$

$$ {}^{1}\vec{a}_{12} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad {}^{1}\vec{a}_{23} = \begin{bmatrix} c_2 \\ s_2 c_{12} \\ s_2 s_{12} \end{bmatrix} \quad {}^{1}\vec{a}_{nm} = \begin{bmatrix} W_{n-1,m-1,\ldots.2} \\ -U^{*}_{n-1,m-1,\ldots.2} \\ U_{n-1,m-1,\ldots.2} \end{bmatrix} \tag{2-30} $$

where the definitions of the X, Y, Z and W, U*, U are presented in [2]. Using the coordinate transformation given in Equation 2-28 and expressions from Equations 2-29 and 2-30, direction vectors can be obtained in terms of the fixed coordinate system and are shown in Equations 2-31 and 2-32.

$$ {}^{F}\vec{S}_i = {}^{F}T_1 \cdot {}^{1}\vec{S}_i \tag{2-31} $$

$$ {}^{F}\vec{a}_{ij} = {}^{F}T_1 \cdot {}^{1}\vec{a}_{ij} \tag{2-32} $$

Forward and Reverse Position Analysis

The first step in the analysis of serial manipulators is to determine the position and orientation of the end-effector for a specified set of joint angles. Following the method outlined by Crane and Duffy [2], the transformation that relates the end-effector coordinate system to the fixed coordinate system is determined. In the case of a 6-axis robot used in this application, the transformation in required is obtained as follows:

$$ {}^{F}_{6}T = {}^{F}_{1}T \cdot {}^{1}_{2}T \cdot {}^{2}_{3}T \cdot {}^{3}_{4}T \cdot {}^{4}_{5}T \cdot {}^{5}_{6}T \tag{2-33} $$

The coordinates of the end-effector tool point in the fixed coordinate system can be found from Equation 2-34

$$ {}^{F}T_{tool} = {}^{F}_{6}T \cdot {}^{6}T_{tool} \tag{2-34} $$

The terms used in Equation 2-33 are known mechanism parameters and are obtained from Table 2-2 for the Puma 762 manipulator.

The real problem arises when one tries to determine the set of joint angles which yield a required end-effector position and orientation. This procedure is called the reverse kinematic analysis and begins by closing the link-loop with a hypothetical member to form a closed loop kinematic chain instead of the serial open loop mechanism. For a 6-R (6-revolute-joint) manipulator such as the Puma 762, this results in a 1-degree-of-freedom 7-R spatial mechanism with the angle θ_7 known. Detailed derivation of the solution to this problem is provided by Crane and Duffy [2]. The C++ program source code written by Crane for this solution is provided in the Appendix D.

Owing to the complexity and length of the solution, only a summary of the derivation and the final solutions are discussed here.

The twist angle α_{71} are calculated using Equations 2-35 and 2-36

$$ c_{71} = {}^{F}\vec{S}_7 \cdot {}^{F}\vec{S}_1 \tag{2-35} $$

$$ s_{71} = \left({}^{F}\vec{S}_7 \times {}^{F}\vec{S}_1 \right) \cdot {}^{F}\vec{S}_1 \tag{2-36} $$

Joint angle θ_7 is found using Equations 2-37 and 2-38

$$ c_7 = {}^{F}\vec{a}_{67} \cdot {}^{F}\vec{a}_{71} \tag{2-37} $$

$$ s_7 = \left({}^{F}\vec{a}_{67} \times {}^{F}\vec{a}_{71} \right) \cdot {}^{F}\vec{a}_{71} \tag{2-38} $$

γ_1 is defined as the angle between vector $\vec{a}_{71}$ and the x-axis of the fixed coordinates.

This angle is obtained using Equations 2-39 and 2-40.

$$\cos\gamma_1 = {}^F\vec{a}_{71} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{2-39}$$

$$\sin\gamma_1 = \left({}^F\vec{a}_{71} \times \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) \cdot {}^F\vec{S}_1 \tag{2-40}$$

The distances S_7 , a_{71}, S_1 is derived by Equations 2-41, 2-42 and 2-43.

$$S_7 = \frac{\left({}^F\vec{S}_1 \times {}^F\vec{P}_{6orig} \right) \cdot {}^F\vec{a}_{71}}{s_{71}} \tag{2-41}$$

$$a_{71} = \frac{\left({}^F\vec{P}_{6orig} \times {}^F\vec{S}_1 \right) \cdot {}^F\vec{S}_7}{s_{71}} \tag{2-42}$$

$$S_1 = \frac{\left({}^F\vec{P}_{6orig} \times {}^F\vec{S}_7 \right) \cdot {}^F\vec{a}_{71}}{s_{71}} \tag{2-43}$$

Table 2-3. Closed-loop mechanism parameters of the Puma 762 robot

Link Length, mm	Twist Angle, deg	Joint Offset, mm	Joint Angle, deg
a_{12} = 0	α_{12} = 90		φ_1 = variable
a_{23} = 650	α_{23} = 0	S_2 = 190	θ_2 = variable
a_{34} = 0	α_{34} = 270	S_3 = 0	θ_3 = variable
a_{45} = 0	α_{45} = 90	S_4 = 600	θ_4 = variable
a_{56} = 0	α_{56} = 90	S_5 = 0	θ_5 = variable
a_{67} = 0	α_{67} = 90	S_6 =125	θ_6 = variable
a_{71} = C.L	α_{71} = C.L.	S_7 =C.L.	θ_7 = C.L.

Once the closed-loop parameters are obtained as above, the vector-loop equation for the mechanism is written as in Equation 2-44 and expanded as in Equation 2-45 using the direction cosines in [2].

$$S_1\overrightarrow{S_1} + S_2\overrightarrow{S_2} + a_{23}\overrightarrow{a_{23}} + S_4\overrightarrow{S_4} + S_6\overrightarrow{S_6} + S_7\overrightarrow{S_7} + a_{71}\overrightarrow{a_{71}} = \vec{0} \tag{2-44}$$

$$S_1\begin{bmatrix}0\\-s_{12}\\c_{12}\end{bmatrix} + S_2\begin{bmatrix}0\\0\\1\end{bmatrix} + a_{23}\begin{bmatrix}c_2\\-s_2\\0\end{bmatrix} + S_4\begin{bmatrix}X_{5671}\\Y_{5671}\\Z_{5671}\end{bmatrix} + S_6\begin{bmatrix}X_{71}\\Y_{71}\\Z_{71}\end{bmatrix} + S_7\begin{bmatrix}X_1\\Y_1\\Z_1\end{bmatrix} + a_{71}\begin{bmatrix}c_1\\s_1c_{12}\\U_{12}\end{bmatrix} = \vec{0} \tag{2-45}$$

This is further simplified using subsidiary spatial and polar-sine, sine-cosine and cosine laws from Equation 2-46

$$\begin{bmatrix}X_{5671}\\Y_{5671}\\Z_{5671}\end{bmatrix} = \begin{bmatrix}X_{32}\\-X^*_{32}\\\overline{Z}_3\end{bmatrix} \tag{2-46}$$

The representation in Equation 2-45 yields three equations. Further substation of the known terms and simplification of the Z component of the equation yields Equation 2-47.

$$\left[S_6Y_7 - S_7s_{71}\right]c_1 + \left[S_6X_7 + a_{71}\right]s_1 + S_2 = 0 \tag{2-47}$$

Solving this yields two solution angles θ_{1a} and θ_{1b} . Corresponding values of the angle φ_1 are calculated as (θ_{1a} - γ_1) and (θ_{1b} - γ_1).

Substituting all these values in Equation 2-45, one gets Equations 2-48 and 2-49

$$a_{23}c_2 + S_4X_{23} + S_6X_{71} + S_7X_1 + a_{71}c_1 = 0 \tag{2-48}$$

$$-S_1 - a_{23}s_2 + a_{34}V_{32} - S_4X^*_{23} + S_6Y_{71} + S_7Y_1 = 0 \tag{2-49}$$

This can be written as Equations 2-50 and 2-51

$$a_{23}c_2 + S_4 s_{2+3} = A \tag{2-50}$$

$$-a_{23}s_2 - S_4 c_{2+3} = B \tag{2-51}$$

Where, $A = -S_6 X_{71} - S_7 X_1 - a_{71}c_1,\ B = S_1 - S_6 Y_{71} - S_7 Y_1$, while s_{2+3} and c_{2+3} are the sine and cosine of $\theta_2 + \theta_3$ respectively. Squaring and adding both sides of Equation 2-50 and 2-51 yields Equation 2-52.

$$a_{23}^2 + S_4^2 + 2a_{23}S_4[s_2 s_{2+3} - c_2 s_{2+3}] = A^2 + B^2 \tag{2-52}$$

Simplifying this, yields Equation 2-53.

$$s_3[-2a_{23}S_4] + [a_{23}^2 + S_4^2 - A^2 - B^2] = 0 \tag{2-53}$$

Here, the only unknown is θ_3. For each value of θ_1 two values for θ_3 are obtained. Next, θ_5 is obtained using the relation $Z_{7123} = \overline{Z}_5$ in [2] which again yields two solutions. θ_4 is obtained by using the subsidiary spherical equations $X_{54} = X_{7123}$ and $X_{54}^* = -Y_{7123}$. Similarly the solution to θ_6 using the fundamental spherical sine and sine-cosine laws $X_{43217} = s_{56}s_6$ and $Y_{43217} = s_{56}c_6$. [2]The solution tree can be summarized as shown in Figure 2-14.

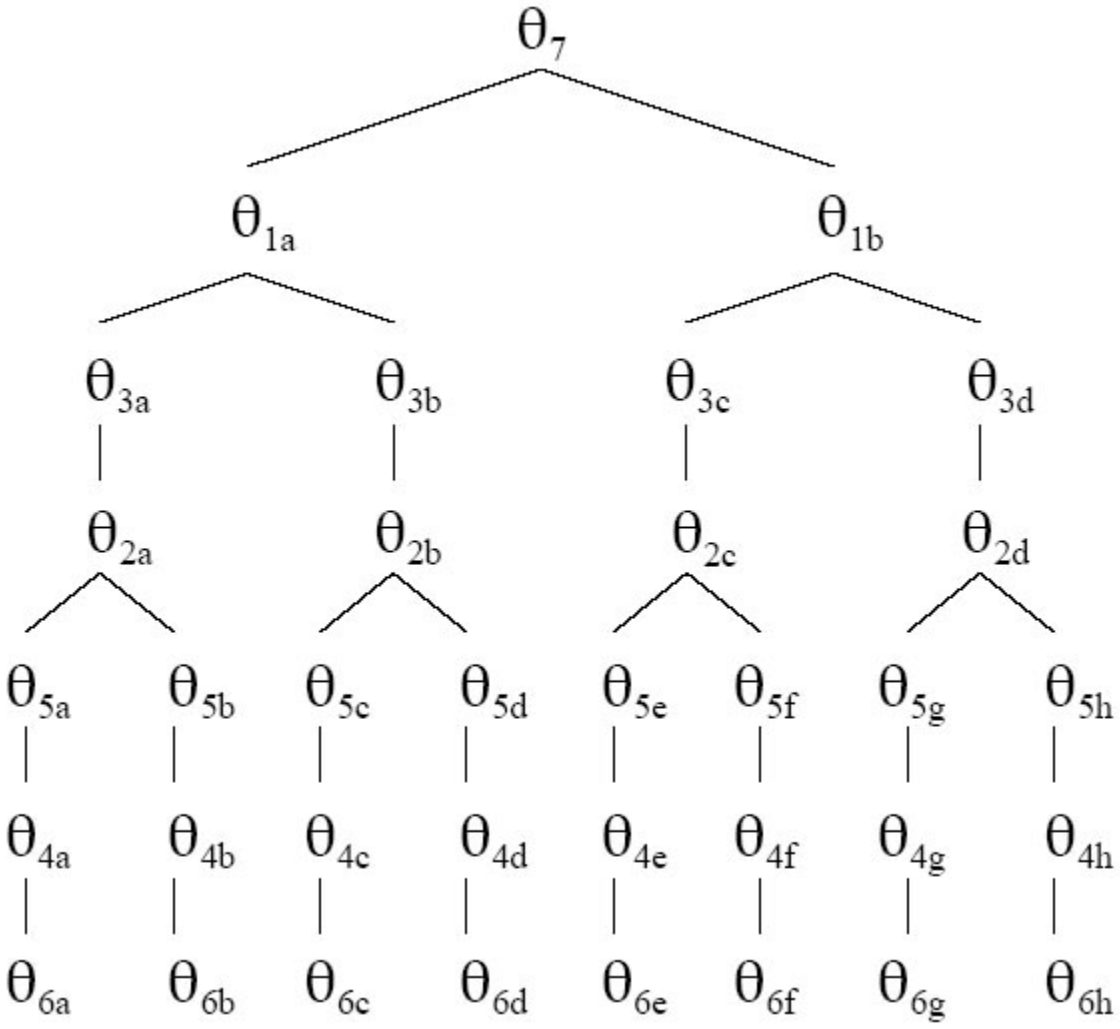


Figure 2-14. Solution tree to the reverse analysis of Puma 762 robot [13]

CHAPTER 3
6 DOF PARALLEL PLATFORM

This chapter details the mechanical and electrical design of the parallel platform.

3.1 Mechanical Design

The mechanical design of the prototype was done by Bo Zhang and this formed the basis for the electrical design and implementation. [9]

3.1.1 Design Specifications

The 6-6 Special parallel platform designed by Bo Zhang is a promising spatial compliant device. The platform structure uses 6 legs loaded with linear springs for passive compliance. One of the main advantages of the parallel platform due to its parallel kinematic geometry is that it can withstand a large payload (compared to serial geometries) with a relatively compact size. The detailed design specifications are presented in Table 3-1. Note that in Table 3-1, the direction of the z axis is referred to as perpendicular to the plane of the base platform. [9]

Table 3-1. Design objective specifications

Specifications	Range
Individual Connector Deflection	± 5mm
Maximum perpendicular load	50 N
Motion range in Z direction	± 4 mm
Motion range in X,Y directions	± 3.5 mm
Rotational range about Z axis	± 6°
Rotational range about X,Y axes	± 4°
Position measurement resolution	0.01 mm

3.1.2 Conceptual Design

A 3-3 octahedral structure was first considered for this platform due to its geometrical simplicity. The terminology 3-3 is due to the fact that the platform device has three links connecting the base platform to the top platform. The 3-3 parallel platform consists of 6 connectors with 6 concentric spherical joints, three on the base

platform and three on the top platform. But this is difficult to design and manufacture accurately, hence a general 6-6 platform was selected. This leads to a problem of the forward position analysis which would require solution of a 40th order polynomial. As a result, the Special 6-6 parallel platform developed by Griffis and Duffy was chosen. Here, the concentric ball and socket joints have been separated such that all the base platform connection points lie on a triangle and all the top platform connection points lie on a second triangle. The complexity of this device is similar to the 3-3 platform. In order to eliminate the six additional rotational freedoms due to the spherical joints, they were replaced by universal or Hooke joints.

Thus the mobility of the special 6-6 parallel platform is

$$M = 6(14-1) - \sum_{i=1}^{6}(6-3) - \sum_{i=1}^{6}(6-2) - \sum_{i=1}^{6}(6-1) = 78-18-24-30 = 6$$

The legs 1-6 are arranged in such a way that each of them is connecting to a corner connecting point on either the top or base platform triangle, while the other connecting point is in the midline of the triangle of the opposing platform. One computer generated model of these configurations is shown in Figure 3-1.

The best configuration for the top platform was found to be when the device is in its home unloaded configuration, this occurs in the configuration where the Plücker coordinates of the six leg connectors are ‘as far as possible’ without being linearly dependent. Lee [10] analyzed the 3-3 and Special 6-6 platforms to determine the ratio of the size of the top platform to the base as well as the height and pose of the top platform such that the determinant of the 6 × 6 matrix formed by the Plücker coordinates of the six leg connectors would be at its maximum value. It was shown that the maximum value would occur if the length of the side of the base triangle was twice that

of the top triangle. Further at the unloaded home position, the top platform was horizontal and rotated relative to the base as shown in Figure 3-1. The perpendicular distance between the base and the top platforms equaled the length of the side of the top platform triangle. [14]

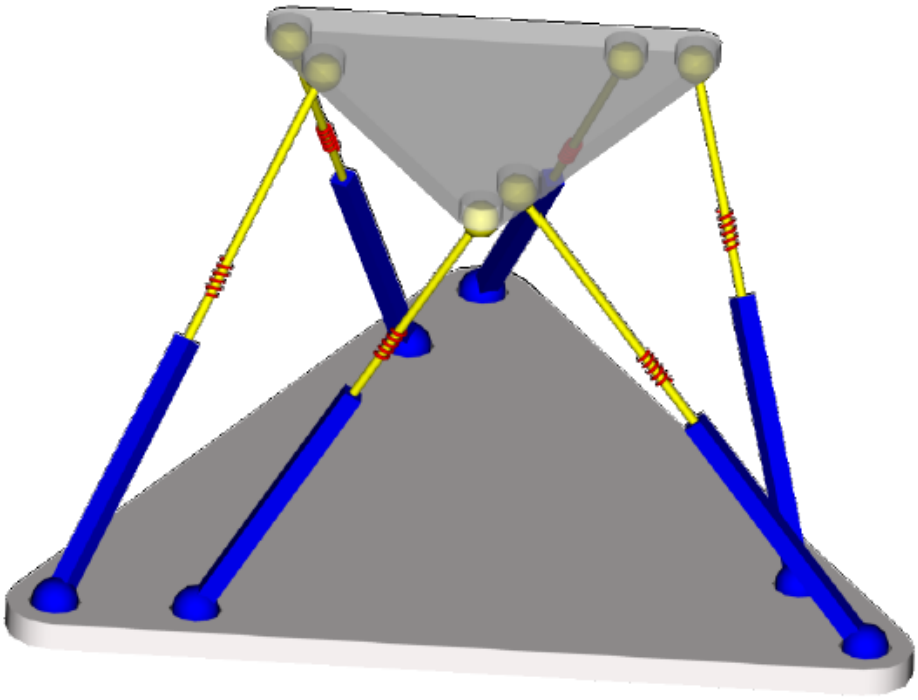


Figure 3-1. The special 6-6 platform

3.1.3 Prototype Design

The free lengths of the connectors are calculated by the following Equations 3-1 and 3-2:

$$l_{long} = \sqrt{\frac{1}{3}(3h^2 + a\left(3\rho^2 - 3\rho + 1\right) - \left(\cos\theta + \sqrt{3}\left(2\rho - 1\right)\sin\theta\right)ab + b^2} \qquad (3\text{-}1)$$

$$l_{short} = \sqrt{\frac{1}{3}(3h^2 + b\left(3\rho^2 - 3\rho + 1\right) - \left(\cos\theta + \sqrt{3}\left(2\rho - 1\right)\sin\theta\right)ab + a^2} \qquad (3\text{-}2)$$

As shown in Figure 3-2,

l : Free length of the legs, divided into two groups based on the fact that there are two different values for free length of the legs.

a : refers to the functional triangle edge length of the top platform.

b : refers to the functional triangle edge length of the base platform.

θ : refers to the rotation angle of the top platform about the z-axis.

$\rho\, a$, $\rho\, b$: refer to the distance between adjacent joint points on the base platform and top platform [14].

Zhang chose the parameters based on various factors like feasibility of manufacturing, encoders available in the market, necessary working space without interference, etc. [9] The selected values were

a=60mm, b=120mm, ρ = 28/120=0.2333, h=a=60mm,

L_{short}=68.021mm, L_{long}=80.969mm (3-3)

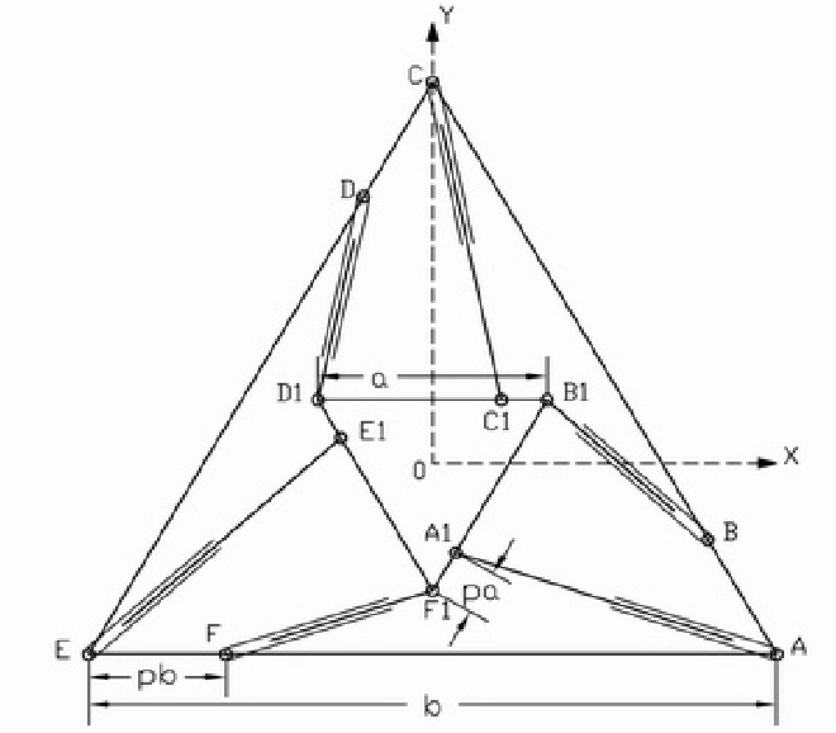


Figure 3-2. Plan view of the Special 6-6 platform

The prismatic joint on each leg is spring loaded. Precision linear springs were selected for use as the elastic component to provide elongation and compression of the leg connectors. After carefully studying and comparing different springs, Zhang chose cylindrical precision springs with spring constant 2.627 N/mm and the high linear coefficient (± 5%).[9]

The free length of each of the legs is measured by a linear optical encoder. Each of the legs is comprised of 2 parts, one attached to the top platform, the other attached to the base platform and both translating in a prismatically constrained manner. The encoder head is mounted on of the connectors and the optical track on the other. All the six connection points on top and base platform are in the same plane. The free length of these legs is defined as the distance between the connection points on the top and base platform when the platform is not loaded, i.e., it is free. The top end of the legs were chosen to have a Hooke joint while the bottom end were chosen to have spherical joints formed using combination of bearings and Hooke joints. [14]

A model of the platform is shown in Figure 3-3. Pictures of the platform are shown in Figures 3-4 through 3-6

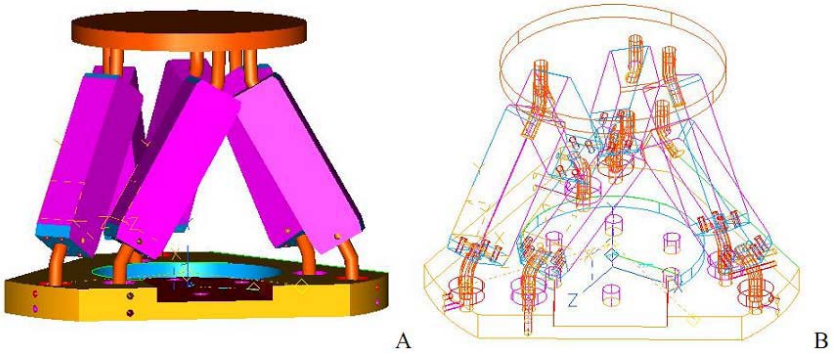


Figure 3-3. 3-D model of the special 6-6 parallel platform prototype. A) Solid model, B)Frame model. [Adapted from Nayak, S. 2009. Development of a 6 DOF Compliant Parallel Mechanism to Sense and Control the Force-Torque and

Displacement of a Serial Robot Manipulator. M.S. Thesis (Page 46 , Figure 3-5). University of Florida, Gainesville, Florida]

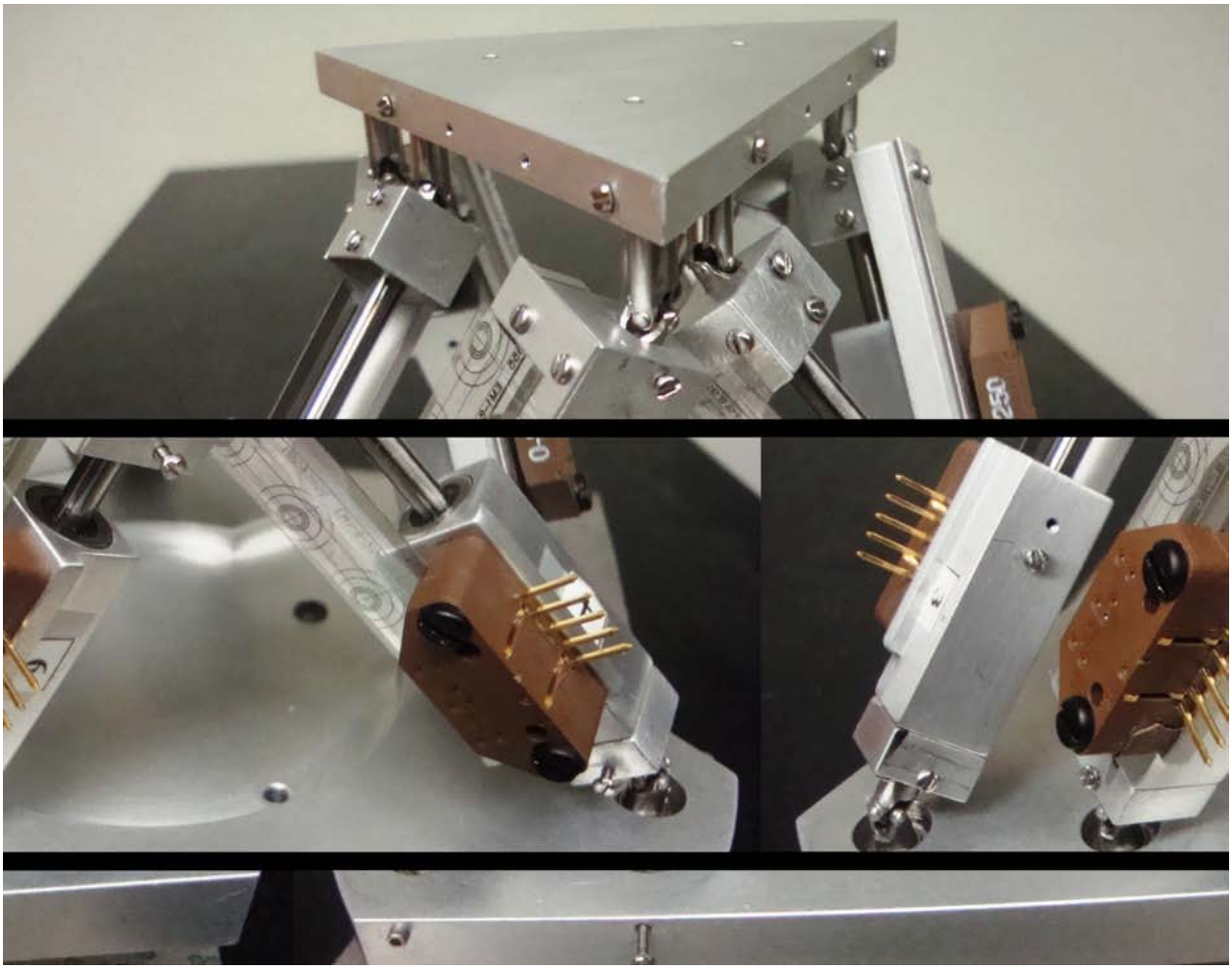


Figure 3-4. Photo of assembled prototype [Adapted from Nayak, S. 2009. Development of a 6 DOF Compliant Parallel Mechanism to Sense and Control the Force-Torque and Displacement of a Serial Robot Manipulator. M.S. Thesis (Page 46 , Figure 3-5). University of Florida, Gainesville, Florida]

Figure 3-5. Photo of base and top platform [Adapted from Nayak, S. 2009. Development of a 6 DOF Compliant Parallel Mechanism to Sense and Control the Force-Torque and Displacement of a Serial Robot Manipulator. M.S. Thesis (Page 47 , Figure 3-6). University of Florida, Gainesville, Florida]

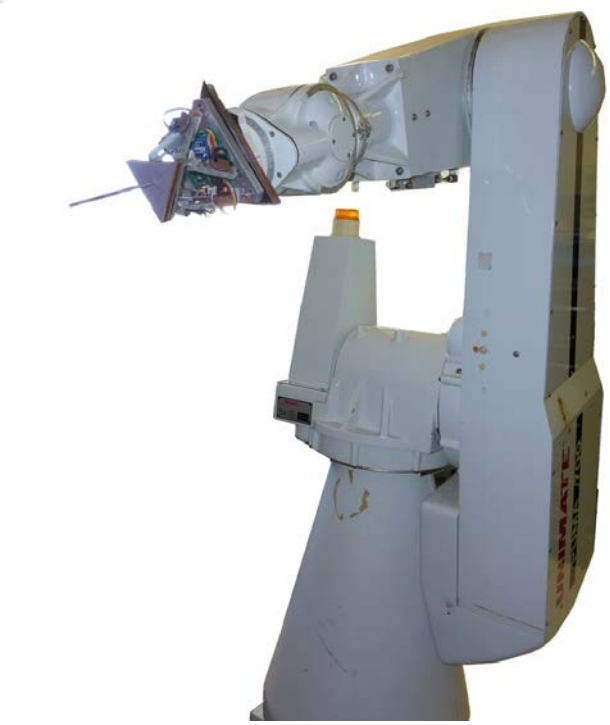


Figure 3-6. The Parallel platform mounted on the distal end of a PUMA industrial robot

3.2 Electrical Design

In this section, the electrical design of the prototype of the parallel platform is discussed. This involves position sensing with encoders, high speed simultaneous reading of the sensor, and sending it to the computer that manages the PUMA industrial robot controller.

3.2.1 Design Specifications

The following design specifications as laid out by Subrat Nayak in designing the electronics for the platform.

The position measurement device should be of a high resolution, accurate and immune to errors. It must also be compact, light weight and non-interfering with the normal functioning of the platform. The device must have an electrical output, thus eliminating the need for added interfacing coupler. The sensed data from the encoders needs to be sent to the control computer wirelessly to eliminate the wiring issues. Thus it also needs to be battery powered. The receiver must be compatible to be interfaced with current generation computers. [14]

3.2.2 Position Transducer

Subrat Nayak explored various position transducers and found encoders seemed to be the best suited for this application. These are digital sensors with very high resolution. The Optical type encoders were chosen due to higher resolution and feasibility. Absolute optical encoders were used which have an index sensor which demarcates the reference point to cope up with the problem of losing reference or accumulating error. Every time an index pulse shows up, the sensor is supposed to zero its position and then start counting the incremental pulses. Quadrature encoding with index forms the best absolute encoder was used as it can not only sense direction but

also provides 4 times higher resolution than an incremental encoder. The US Digital EM1 Transmissive Optical Encoders and US Digital LIN Transmissive Linear Strip with 250 counts per inch resolution were chosen. (Figure 3-7 and 3-8)

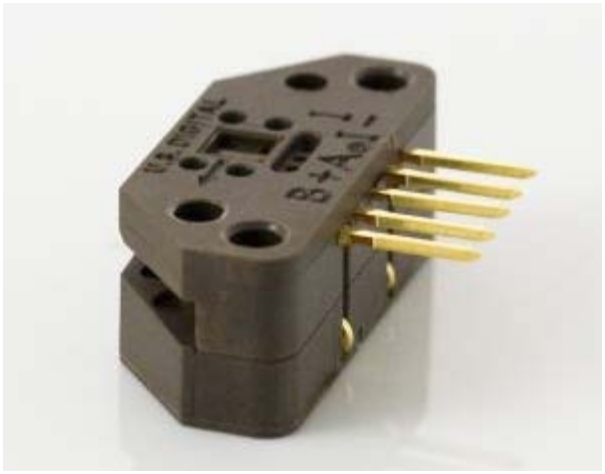


Figure 3-7. US digital EM1 transmissive optical encoder

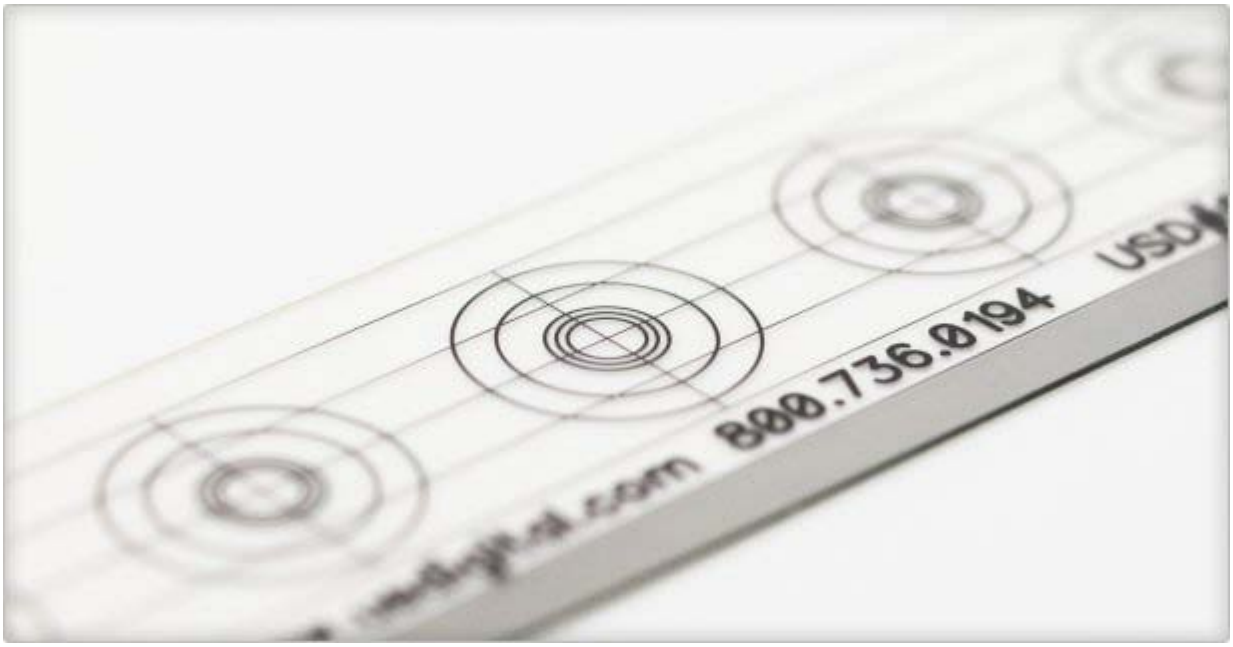


Figure 3-8. US digital LIN transmissive linear strip

3.2.3 Sensor data Capture and wireless transmission

Data from Six Encoders needs to be captured simultaneously. Hence Subrat Nayak implemented a FPGA (Altera cyclone II EP2C8T144C8) to read the encoders and stream the data continuously.

The data, now which is available in a parallel format, is multiplexed and a micro controller then forms and sends out packets of sensor data in asynchronous serial format to an Xbee wireless module in ASCII format. This data is received by another Xbee module and the data is available in serial format. A Serial – USB converter is used to feed this data to a control computer. [14]

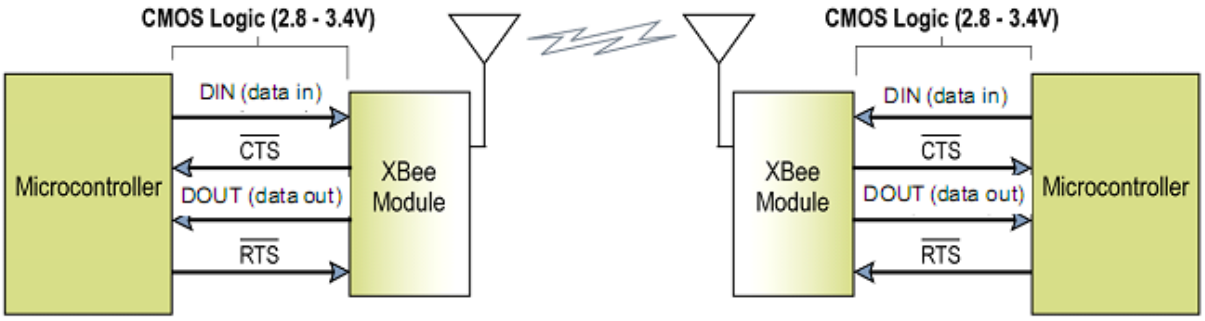


Figure 3-9. Two Xbee modules can replace a serial communication line

CHAPTER 4
IMPLEMENTATION OF KINESTATIC CONTROL

The implementation of kinestatic control was done on a Puma 762 industrial robot fitted with a Galil™ DMC-2160 motion controller as it was readily accessible and allowed for ease of control from a computer. For this application the Galil controller was commanded via a MATLAB™ program making the whole control procedure simple. The steps are discussed below.

Step 1: Input info for taking robot to a particular pose, generate the respective axes angles and select the best set of angles which will achieve the required pose. Then send these axes angles to the Galil™ controller. The required inputs are coordinates of:

$^{F}_{tool}P$ Tool point in the fixed coordinate system

$^{6}_{tool}P$ Tool point in the 6th or end effector coordinate system

$^{F}S^{6}$ 6th or end effector axis in the fixed coordinate system

a^{F}_{67} Orientation of the 6th axis in the fixed coordinate system

In this step, the user inputs the coordinates and the orientation for the robot to go to. This is done in the command window of the MATLAB™. Then the reverse analysis function *puma_rev_analysis.m* is called. This function generates the required set of axes angles to go to the required position and orientation: *command_pose*. This variable containing the axes coordinates is then fed to the program which handles the position contol for the robot: *galil_control.m*. Next the forward analysis of the robot is

perfomed by the program *puma_fwd_analysis.m* and the current values of ${}^{F}_{tool}P$, ${}^{\mathrm{F}}S^{6}$, a^{F}_{67} are recorded.

Step 2: Measure the platform leg lengths and calculate the equivalent wrench $\mathbf{w}_{\mathrm{eq}}$. on the platform. This ${}^{0}_{1}T$ matrix is used to calculate the coordinates of the six legs, the wrenches along the six legs is thus found. The equivalent wrench of these six legs is the sum of the six wrenches.

The process flow in this step can be described as follows. The program *serialdaq_plat.m* is called which acquires the leg lengths of the platform via serial port and are stored in the variable *leg_e*. These leg lengths, which are in encoder counts format are converted into usable leg lengths and input to another program *leg_to_T10.m* to genetrate the current ${}^{0}_{1}T$ matrix for the platform. This ${}^{0}_{1}T$ matrix is used to calculate the coordinates of the six legs.Next the wrenches along the six legs and thus the equivalent wrench $\mathbf{w}_{\mathrm{eq}}$ acting on the platform is calculated using the the program *eq_wrench.m.* It stores the equivalent wrench $\mathbf{w}_{\mathrm{eq}}$ in the variable *W_eq*.

Step 3: The current ${}^{\mathrm{F}}S^{6}$ is checked if it is in line with the equivalent wrench $\mathbf{w}_{\mathrm{eq}}$. If not, the ${}^{\mathrm{F}}S^{6}$ of the robot is then aligned with the direction of the equivalent wrench. During this process, the ${}^{F}_{tool}P$ is kept the same to retain the position of the tool. However, ${}^{6}_{tool}P$ is set to the original coordinates, i.e. when the platform is in an unloaded state. Thus when the alignment is complete, the tool axis will be in line with

${}^{F}S^{6}$, this eliminates the torque portion from the wrench leaving behind a pure force. This force can be regulated by continuously checking the error between force generated and tolerance force by adjusting the position of the robot along the ${}^{F}S^{6}$ which is now also in line with $\mathbf{w}_{eq}$.

The program flow for this step can be explained as follows. The ${}^{F}S^{6}$ generated in the previous step is compared to the direction of the wrench by the main kinestatic control program *ForceCorrection.m* , if it is not in line, the robot is moves to match the ${}^{F}S^{6}$ with the direction of the equivalent wrench $\mathbf{w}_{eq}$. Thus at this point, the effectice wrench on the tool is purely a force with the elimination of torques in the tool. Now the next step is to bring the force magnitude in the tool within the required limits. The magnitude of force is calculated from the user input variable *F_req,* the force vector (along which the ${}^{F}S^{6}$ was previously aligned). The tolerance to achieve this force is specified by user input variable *f_tol*. The robot is then moved closer to the material along the line ${}^{F}S^{6}$ if the force being generated now is less than the *F_req* or farther if it is more than required. This process again involves calling the *puma_rev_analysis.m* program to calculate the requied set of joint angles and *galil_control.m* to send these commands to the motion controller.

The Puma robot used to implement this application had an anomalous behavior on the 5th joint. Whenever the 4th joint was indexed, the 5th joint also moves by a constant ratio. This was due to some mechanical linkage or due to the mechanical design of the joint. To correct this behavior, a special program *Axis_4_5_Correction.m* was written.

This program detects the motion in the 4th joint and sends the corrected position command and the encoder counts the motion controller for the 5th joint to compensate the motion.

During the implementation of kinestatic control, it was found that the springs in the legs of the platform had non linearities. Hence each of the legs was individually calibrated on a precise load cell to map the change in free length of the spring to the corresponding force in the spring. Hence the method discussed in Chapter 2 where the wrench was calculated using a compliance matrix which in turn is dependant on the spring constants could not be used.

The results of implementation of Kinestatic Control on Puma 762 robot are detailed in the next chapter. The MATLAB™ programs to perform the kinestatic control are given in the Appendix E.

CHAPTER 5
RESULTS AND CONCLUSION

5.1 Results

The Kinestatic Control theory was implemented on a Puma 762 Industrial robot. This robot has been retrofitted with a Galil motion controller, thus making it easy to command it completely from a computer conneted to it via Ethernet. The Special 6-6 platform was mounted on the distal end of the robot and a metal tip was attached to the top mobile platform for the parallel mechanism.

For testing this theory, the robot was commanded to attack the wooden plate with a force of 4N in a direction specified by the variable $\boldsymbol{F\_req} = 0.9713\boldsymbol{i} + 0\boldsymbol{j} - 0.2375\boldsymbol{k}$. The tolerance for the force magnitude to be attained was defined to be ± 0.5N to account for the inaccuracies of mounting the platform and tool on the robot.

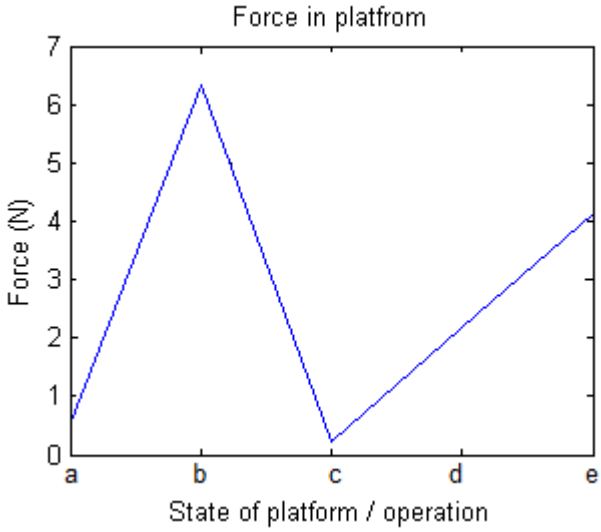
After the robot completed kinestatic correction, the wrench measurements in the tool yielded that there was negligible torque while the force magnitude was 3.6N. This was well within the tolerance limits and the robot had achieved its motto of kinestatic control. The photographs in Figure 5-1 below shows the Kinestatic Control in action during the test. Figure 5-1 (A) shows the robot aligned to $\boldsymbol{F\_req}$ ready to attack the wood plate. Next in Figure 5-1 (B), the robot touched the wood plate and reoriented itself in the direction of the wrench thus calculated, to convert wrenc into a pure force. In Figure 5-1 (C), the robot achieves its target force magnitude and stops. The graphs describing the contact wrench vs. state of platform / operation in the fixed coordinate system while robot was performing the wrench correction is shown in Figures 5-2. (A) thru (G).


A

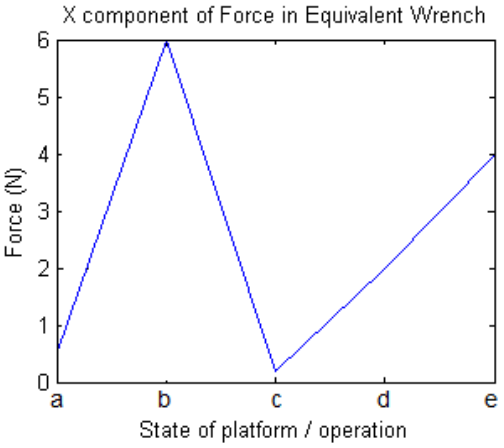

B


C

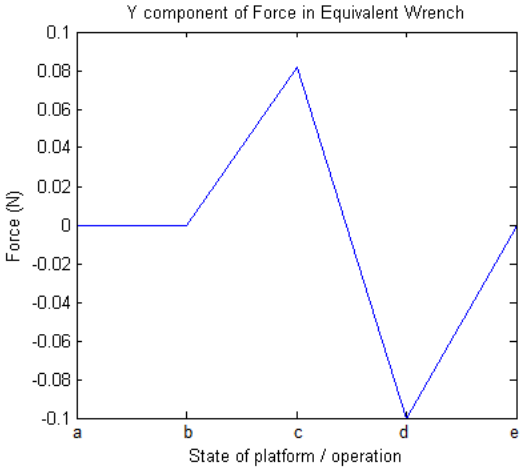
Figure 5-1. A) Robot moves to the required position and orientation B) Controller measures the wrench and reorients the robot C) Robot moves in the direction of the wrench to increase or decrease the force magnitude till the goal force is attained. (Photo courtesy - Akash Vibhute)


Force in platfrom
Force (N)
State of platform / operation
a
b
c
d
e
0
1
2
3
4
5
6
7

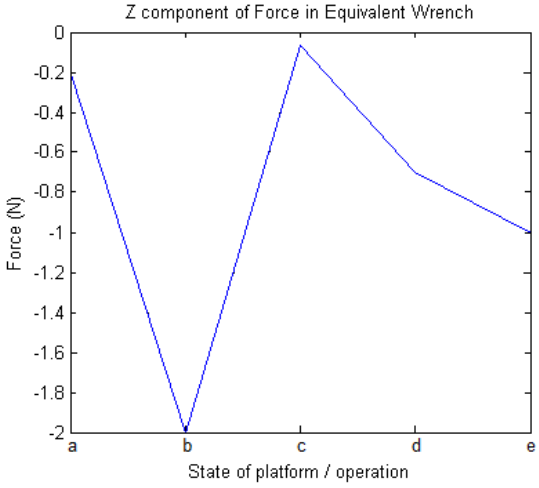
A


X component of Force in Equivalent Wrench
Force (N)
State of platform / operation
a
b
c
d
e
0
1
2
3
4
5
6

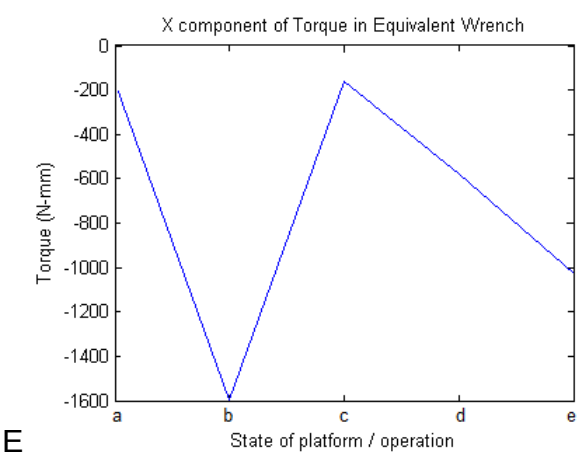
B


Y component of Force in Equivalent Wrench
Force (N)
State of platform / operation
a
b
c
d
e
0.1
0.08
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
-0.1

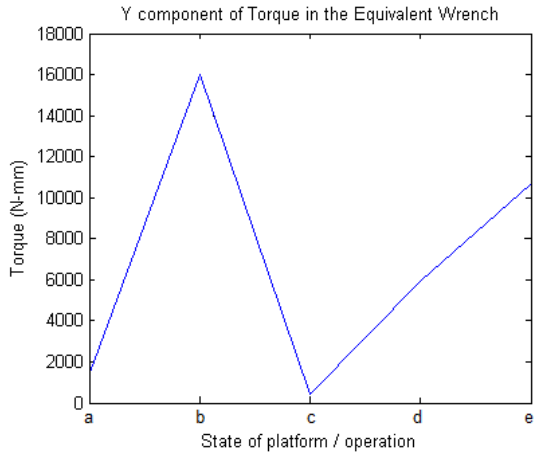
C


Z component of Force in Equivalent Wrench
Force (N)
State of platform / operation
a
b
c
d
e
0
-0.2
-0.4
-0.6
-0.8
-1
-1.2
-1.4
-1.6
-1.8
-2

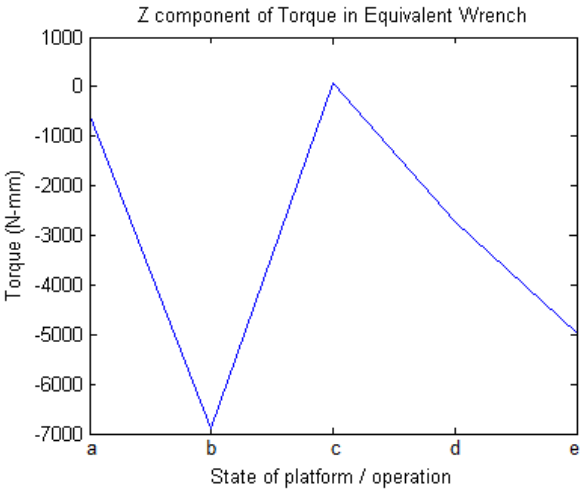
D



E



F



G

Figure 5-2. A) Force magnitude vs. State of platform
B) X component of Foce in the Effective Wrench vs. State of platform
C) Y component of Foce in the Effective Wrench vs. State of platform
D) Z component of Foce in the Effective Wrench vs. State of platform
E) X component of Torque in the Effective Wrench vs. State of platform
F) Y component of Torque in the Effective Wrench vs State of platform
G) Z component of Torque in the Effective Wrench vs. State of platform

The states of platform are,

a – Platfrom in rest, unloaded condition.

b – Tool has hit the workpiece, wrench genereated.

c – Robot reorients the tool in the direction of equivalent wrench.

d – Force in tool being controlled to achieve goal force.

e – Goal force attained, Kinestatic control complete.

5.2 Conclusion

This work will allow one to explore the applications and performance of the Theory of Kinestatic Control on a 6 axis industrial robot. Some of the prime applications of this theory are driving a peg into a hole whose axis is not known hence the robot cannot be aligned to drive the peg straight into it. Another major application being, while grinding a curved surface of a material which has been manufactured by processes which don not have very high accuracy like casting. While grinding such surfaces, the curvature of the material is not known and cannot be estimated, thus it cannot be programmed in the robot. This leads the tool to either eat into the material by overgrinding or leave it unfinished. But with the use of Kinestatic control, the tool on the robot can be reoriented in such a way, that the tool axis is parallel to the surface and the force being applied is kept constant even when the surface is uneven.

Apart from industrial applications, this technology can be applied even in the medical field where, during surgical operations, the force vector with of the procedure being done has to be highly accurate.

In the application discussed in this thesis, the kinestatic control was implemented using a force tolerance of ± 1N. This was done because no sophisticated closed loop control was implemented. But kinestatic control can be truly beneficial by the use of non-linear control methods. When such high end precise control methods would be implemented, the tolerance can be reduced to less than ± 0.1N. The results thus obtained can used to in implement real world situations.

APPENDIX A
COORDINATES OF A POINT AND LINE

The position vector to a point Q_1 from origin O is given by $r_1$and can be expressed in the form

$$r_1 = \frac{x_1 \mathrm{i} + y_1 j + z_1 k}{w_1} \tag{A-1}$$

Where $\mathbf{x}_1$, $\mathbf{y}_1$, $\mathbf{z}_1$ have units of length and $\mathbf{w}_1$ is dimensionless and they form the homogenous coordinates of point Q_1as ($\mathbf{w}_1$; $\mathbf{x}_1$, $\mathbf{y}_1$, $\mathbf{z}_1$)[1]

The join of two distinct points $\mathbf{r}_1$ and $\mathbf{r}_2$ determine a line. The vector $\mathbf{S}$ whose direction is along the line may be written as

$$\mathbf{S} = \mathbf{r}_2 - \mathbf{r}_1 \tag{A-1}$$

Direction is a unit less concept and thus, the elements of vector $\mathbf{S}$ are dimensionless. The vector $\mathbf{S}$ may be alternatively expressed as

$$\mathbf{S} = L\mathbf{i} + M\mathbf{j} + N\mathbf{k} \tag{A-3}$$

where L=$x_2 - x_1$, M=$y_2 - y_1$ and N=z_2-z_1 are defined as the dimensionless direction ratios. They are related to $|\mathbf{S}|$ by

$$L^2 + M^2 + N^2 = |\boldsymbol{S}|^2 \tag{A-4}$$

For $|\mathbf{S}|$=1, Equation (A-4) reduces to

$$L^2 + M^2 + N^2 = 1$$

If $\mathbf{r}$ designates a vector from the origin of the coordinate system to any general point on the line, then it's apparent that the vector $\mathbf{r} - \mathbf{r_1}$ is parallel to $\mathbf{S}$. Thus it may be written that,

$$(\mathbf{r} - \mathbf{r_1}) \times \mathbf{S} = 0 \tag{A-5}$$

This can be expressed in the form

$$\mathbf{r} \times \mathbf{S} = \mathbf{S_{OL}} \tag{A-6}$$

where

$$\mathbf{S_{OL}} = \mathbf{r_1} \times \mathbf{S} \tag{A-7}$$

is the moment of the line about origin and is clearly origin dependant. The elements of vector $\mathbf{S_{OL}}$ have units of length. Furthermore, since $\mathbf{S_{OL}} = \mathbf{r_1} \times \mathbf{S}$ the vectors $\mathbf{S}$ and $\mathbf{S_{OL}}$ are perpendicular and must satisfy the orthogonality condition.

$$\mathbf{S}\,\mathbf{S_{OL}} = 0 \tag{A-8}$$

The coordinates of the line will be written as $\{ \mathbf{S} ; \mathbf{S_{OL}} \}$ and will be referred to as the Plücker coordinates of the line. The semi-colon is introduced to signify the dimensions of $|\mathbf{S}|$ and $|\mathbf{S_{OL}}|$ are different. The coordinates $\{ \mathbf{S} ; \mathbf{S_{OL}} \}$ are homogenous since from (A-6) the coordinates $\{ \lambda \mathbf{S} ; \lambda \mathbf{S_{OL}} \}$ where λ is a non-zero scalar, determine the same line. Expanding (A-6) yields,

$$s_{0L} = \begin{vmatrix} \mathrm{i} & j & k \\ x_1 & y_1 & z_1 \\ L & M & N \end{vmatrix} \tag{A-9}$$

Which can again be expressed as

$\mathbf{S_{OL}} = P\mathbf{i} + Q\mathbf{j} + R\mathbf{k}$ where $P = y_1N - z_1M$, $Q = z_1L - x_1N$, $R = x_1M - y_1L$

Equation (A-8) becomes

$$LP + MQ + NR = 0 \tag{A-10}$$

Hence, $\{ \mathbf{S} ; \mathbf{S_{OL}} \}$ the Plücker coordinates of the line becomes $\{L, M, N; P, Q, R\}$ [1]

APPENDIX B
SOURCE CODE FOR KINEMATIC ANALYSIS OF PARALLEL MECHANISM

Source code of the C++ program written by Crane to perform the kinematic analysis of the special 6-6 parallel platform

main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define TRUE  1
#define FALSE 0

#ifndef D2R
#define D2R  0.01745329
#endif

#ifndef R2D
#define R2D 57.29577951
#endif

#define Sqrt(x)  sqrt((double)(x))
#define Fabs(x)  fabs((double)(x))

struct plat_params_struct
{
      double p0_x0 ;        // X coordinate of point P0 in the 0 coordinate
system (Y & Z values are 0)
      double q0_x0, q0_y0 ; // X and Y coordinates of point Q0 in 0
coordinate system (Z value is 0)
    double s1_x1 ;        // X coordinate of point S1 in the 1 coordinate
system (Y & Z values are 0)
      double t1_x1, t1_y1 ; // X and Y coordinates of point T1 in 1
coordinate system (Z value is 0)

      double o0s0 ;         // distance between points O0 and S0
      double p0t0 ;         // distance between points P0 and T0
      double q0r0 ;         // distance between points Q0 and R0
      double r1o1 ;         // distance between points R1 and O1
      double s1p1 ;         // distance between points S1 and P1
      double t1q1 ;         // distance between points T1 and Q1
} ;


void main()
{
      struct plat_params_struct plat_params ;

      void solve_georedux (struct plat_params_struct *params,
                                 double L_o0o1,double L_s0s1,double
L_p0p1,double L_t0t1,
                                 double L_q0q1,double L_r0r1,double Lsfor33[6])
;
      void solve_platform (int *pnum_solutions,
                  double T_2_1[8][4][4],
                  double p_x_1, double q_x_1, double q_y_1,
                  double s_x_2, double t_x_2, double t_y_2,
                  double L_or, double L_os, double L_ps,
                  double L_pt, double L_qt, double L_qr) ;
      void vecmult(double ans1[4], double matrix1[4][4], double vector1[4]);
```

```
      double dist (double pt1[4], double pt2[4]) ;


     /* input items */
            /* constant platform parameters */
            plat_params.p0_x0 = 120.0 ;     // mm
            plat_params.q0_x0 = 120.0/2.0 ;  plat_params.q0_y0 =
sin(60.0*D2R)*120.0 ;
            plat_params.s1_x1 =  60.0 ;
            plat_params.t1_x1 =  60.0/2.0 ;  plat_params.t1_y1 =
sin(60.0*D2R)*60.0 ;
            plat_params.o0s0 =  28.0 ;  //109.0 ;
            plat_params.p0t0 =  28.0 ;  //109.0 ;
            plat_params.q0r0 =  28.0 ;  //109.0 ;
            plat_params.r1o1 =  14.0 ;  // 47.5 ;
            plat_params.s1p1 =  14.0 ;  // 47.5 ;
            plat_params.t1q1 =  14.0 ;  // 47.5 ;

//*/
// Input variables
            /* leg lengths */
            //double L_oo = 97.625 ;   // distance between points O0 and O1
            //double L_ss = 88.125 ;   // distance between points S0 and S1,
etc.
            //double L_pp = 90.0 ;
            //double L_tt = 76.5 ;
            //double L_qq = 82.125 ;
            //double L_rr = 92.125 ;
//*/

            double L_oo = 79.6804 ;   // distance between points O0 and O1
            double L_pp = 81.8285 ;   // distance between points S0 and S1,
etc.
            double L_qq = 81.2669 ;
            double L_rr = 68.6500 ;
            double L_ss = 67.2037 ;
            double L_tt = 68.2854 ;

////*/
//// Input variables
//          /* leg lengths */
//          double L_oo = 90.0 ;   // distance between points O0 and O1
//          double L_ss = 76.5 ;   // distance between points S0 and S1, etc.
//          double L_pp = 82.125 ;
//          double L_tt = 92.125 ;
//          double L_qq = 97.625 ;
//          double L_rr = 88.125 ;
////*/


// Input variables
//          /* leg lengths */
//          double L_oo = 90.0 ;   // distance between points O0 and O1
//          double L_ss = 85.0 ;   // distance between points S0 and S1, etc.
//          double L_pp = 91.25 ;
//          double L_tt = 83.75 ;
//          double L_qq = 88.75 ;
```

```
//          double L_rr = 88.125 ;
//*/



//*/
// Output variables
      /* output items */
            double T_1_0[8][4][4];                        // all possible
poses of top platform

// (transformation matrix 1 to 0
            int num_solutions ;                           // number of real
solutions
//*/

      /* local variables */
            double L_or, L_os, L_ps, L_pt, L_qt, L_qr; //virtual 3-3 leg
lengths
            double Lsfor33[6] ;                        //virtual 3-3 leg
lengths passed out of georedux function

   solve_georedux(&plat_params, L_oo,L_ss,L_pp,L_tt,L_qq,L_rr,Lsfor33);

   // Rename the lengths of the equivalent 3-3

      L_or = Lsfor33[0] ;
      L_os = Lsfor33[1] ;
      L_ps = Lsfor33[2] ;
      L_pt = Lsfor33[3] ;
      L_qt = Lsfor33[4] ;
      L_qr = Lsfor33[5] ;

      solve_platform (&num_solutions, T_1_0, plat_params.p0_x0,
plat_params.q0_x0, plat_params.q0_y0,
                              plat_params.s1_x1, plat_params.t1_x1,
plat_params.t1_y1,
                           L_or, L_os, L_ps, L_pt, L_qt, L_qr) ;

      printf("num solutions = %d\n", num_solutions) ;

      FILE *fp ;
      fp = fopen("out.txt", "w") ;

      fprintf (fp, "INPUT ITEMS:\n") ;
    fprintf (fp, "\tpx0 = %8.4lf\tqx0 = %8.4lf\tqy0 = %8.4lf\n",
            plat_params.p0_x0, plat_params.q0_x0, plat_params.q0_y0) ;
    fprintf (fp, "\tsx1 = %8.4lf\ttx1= %8.4lf\tty1 = %8.4lf\n",
            plat_params.s1_x1, plat_params.t1_x1, plat_params.t1_y1) ;
      fprintf (fp, "\tdist_o0_s0 = %8.4lf\tdist_p0_t0 = %8.4lf\tdist_q0_r0 =
%8.4lf\n",
            plat_params.o0s0, plat_params.p0t0, plat_params.q0r0) ;
      fprintf (fp, "\tdist_r1_o1 = %8.4lf\tdist_s1_p1 = %8.4lf\tdist_t1_q1 =
%8.4lf\n",
            plat_params.r1o1, plat_params.s1p1, plat_params.t1q1) ;
      fprintf (fp, "\tLeg O length = %8.4lf\n", L_oo) ;
      fprintf (fp, "\tLeg P length = %8.4lf\n", L_pp) ;
```

```
      fprintf (fp, "\tLeg Q length = %8.4lf\n", L_qq) ;
      fprintf (fp, "\tLeg R length = %8.4lf\n", L_rr) ;
      fprintf (fp, "\tLeg S length = %8.4lf\n", L_ss) ;
      fprintf (fp, "\tLeg T length = %8.4lf\n", L_tt) ;

      fprintf (fp, "\nOUTPUTS:\n") ;
      fprintf (fp, "\t%d solutions for transformation matrix T_1_to_0\n",
num_solutions) ;
      int i, j ;
      for (i=0 ; i<num_solutions ; ++i)
         {fprintf (fp, "\nsolution %d\n", i) ;
            for (j=0 ; j<4 ; ++j)
          fprintf (fp, "%8.4lf\t%8.4lf\t%8.4lf\t%8.4lf\n", T_1_0[i][j][0],
T_1_0[i][j][1], T_1_0[i][j][2], T_1_0[i][j][3]) ;
       }

      // Check the solutions.
      double pt_r1_0[4], pt_s1_0[4], pt_t1_0[4] ;   // coordinates of points
R1, S1, and T1 in coord sys 0
      double pt_o1_0[4], pt_p1_0[4], pt_q1_0[4] ;   // coordinates of points
O1, P1, and Q1 in coord sys 0
      double dist_oo, dist_pp, dist_qq, dist_rr, dist_ss, dist_tt ;

      double pt_r1_1[4]  = {0.0, 0.0, 0.0, 1.0} ;
      double pt_s1_1[4]  = {plat_params.s1_x1, 0.0, 0.0, 1.0} ;
      double pt_t1_1[4]  = {plat_params.t1_x1, plat_params.t1_y1, 0.0, 1.0} ;
      double pt_o1_1[4]  = {plat_params.r1o1, 0.0, 0.0, 1.0} ;
      double pt_p1_1[4]  =
{plat_params.s1_x1+cos(120.0*D2R)*plat_params.s1p1,
sin(120.0*D2R)*plat_params.s1p1, 0.0, 1.0} ;
      double pt_q1_1[4]  =
{plat_params.t1_x1+cos(240.0*D2R)*plat_params.t1q1,
plat_params.t1_y1+sin(240.0*D2R)*plat_params.t1q1, 0.0, 1.0} ;

      double pt_o0_0[4]  = {0.0, 0.0, 0.0, 1.0} ;
      double pt_p0_0[4]  = {plat_params.p0_x0, 0.0, 0.0, 1.0} ;
      double pt_q0_0[4]  = {plat_params.q0_x0, plat_params.q0_y0, 0.0, 1.0} ;
      double pt_r0_0[4]  =
{plat_params.q0_x0+cos(240.0*D2R)*plat_params.q0r0,
plat_params.q0_y0+sin(240.0*D2R)*plat_params.q0r0, 0.0, 1.0} ;
      double pt_s0_0[4]  = {plat_params.o0s0, 0.0, 0.0 , 1.0} ;
      double pt_t0_0[4]  =
{plat_params.p0_x0+cos(120.0*D2R)*plat_params.p0t0,
sin(120.0*D2R)*plat_params.p0t0 , 0.0, 1.0} ;

      fprintf (fp, "\nCheck of solutions.\n\n") ;

      for (i=0 ; i< num_solutions ; ++i)
         {fprintf (fp, "Solution %d\n", i) ;
        vecmult(pt_r1_0, T_1_0[i], pt_r1_1) ;
        vecmult(pt_s1_0, T_1_0[i], pt_s1_1) ;
        vecmult(pt_t1_0, T_1_0[i], pt_t1_1) ;
            vecmult(pt_o1_0, T_1_0[i], pt_o1_1) ;
            vecmult(pt_p1_0, T_1_0[i], pt_p1_1) ;
            vecmult(pt_q1_0, T_1_0[i], pt_q1_1) ;

            dist_oo = dist(pt_o0_0, pt_o1_0) ;
```

```
            dist_pp = dist(pt_p0_0, pt_p1_0) ;
            dist_qq = dist(pt_q0_0, pt_q1_0) ;
            dist_rr = dist(pt_r0_0, pt_r1_0) ;
            dist_ss = dist(pt_s0_0, pt_s1_0) ;
            dist_tt = dist(pt_t0_0, pt_t1_0) ;

          fprintf (fp, "\tdist o0_o1 = %8.4lf\n", dist_oo) ;
            fprintf (fp, "\tdist p0_p1 = %8.4lf\n", dist_pp) ;
            fprintf (fp, "\tdist q0_q1 = %8.4lf\n", dist_qq) ;
            fprintf (fp, "\tdist r0_r1 = %8.4lf\n", dist_rr) ;
            fprintf (fp, "\tdist s0_s1 = %8.4lf\n", dist_ss) ;
            fprintf (fp, "\tdist t0_t1 = %8.4lf\n", dist_tt) ;
          }

      fclose(fp) ;
}

double dist(double pt1[4], double pt2[4])
{
      double mydist ;
      mydist = sqrt(pow(pt1[0]-pt2[0], 2) + pow(pt1[1]-pt2[1], 2) +
pow(pt1[2]-pt2[2], 2)) ;
      return(mydist) ;

}
```

plat_code.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
//#include <string.h>

#define TRUE  1
#define FALSE 0

#ifndef D2R
#define D2R  0.01745329
#endif

#ifndef R2D
#define R2D 57.29577951
#endif

#define Sqrt(x)  sqrt((double)(x))
#define Fabs(x)  fabs((double)(x))

struct plat_params_struct
{
      double p0_x0 ;        // X coordinate of point P0 in the 0 coordinate
system (Y & Z values are 0)
      double q0_x0, q0_y0 ; // X and Y coordinates of point Q0 in 0
coordinate system (Z value is 0)
    double s1_x1 ;        // X coordinate of point S1 in the 1 coordinate
system (Y & Z values are 0)
      double t1_x1, t1_y1 ; // X and Y coordinates of point T1 in 1
coordinate system (Z value is 0)

      double o0s0 ;         // distance between points O0 and S0
      double p0t0 ;         // distance between points P0 and T0
      double q0r0 ;         // distance between points Q0 and R0
      double r1o1 ;         // distance between points R1 and O1
      double s1p1 ;         // distance between points S1 and P1
      double t1q1 ;         // distance between points T1 and Q1
} ;

typedef struct Polyy{
      int deg;
      double coef[37];
      double eval(double x);
      } Poly ;
double Poly::eval(double x)
{
   int i ;
   double result = coef[0] ;
   double val ;

   if (deg > 0)
      result += coef[1]*x ;

   for (i=2 ; i<=deg ; i++)
     {
```

```
      val = pow (x, (double)i) ;
      result += val * coef[i] ;
     }

   return (result) ;
}

void pmult( Poly a, Poly b, Poly *c );
void psub( Poly a, Poly b, Poly *c );
void padd( Poly a, Poly b, Poly *c );
void pscale( Poly a, double s, Poly *as );

void solve_georedux (double L_o0o1,double L_s0s1,double L_p0p1,double L_t0t1,
                              double L_q0q1,double L_r0r1,double Lsfor33[6])
;
void solve_platform (int *pnum_solutions,
                 double T_2_1[8][4][4],
                 double p_x_1, double q_x_1, double q_y_1,
                 double s_x_2, double t_x_2, double t_y_2,
                 double L_or, double L_os, double L_ps,
                 double L_pt, double L_qt, double L_qr) ;

void matmult(double ans[4][4], double matrix1[4][4], double matrix2[4][4]);
void vecmult(double ans1[4], double matrix1[4][4], double vector1[4]);
double dotproduct(double vector1[3], double vector2[3]);
void crossproduct(double ans[3], double vector1[3], double vector2[3]);
double vecmag(double vector[3]);
int valuenear (double x, double goal, double tol) ;
void Inverse(double matdata[], int numcol, double *det,
                  double invary[]);
void MatSwap(double *s1, double *s2);
void Transpose(double *a, double *b, int m, int n);


/*
Function to reduce the special 66 platform geometry to the
 33 geometry in order to calculate the 33 leg lengths to send
 to the "solve_platform" function to do a forward analysis
*/
void solve_georedux(struct plat_params_struct *param, double L_o0o1,double
L_s0s1,double L_p0p1,
                              double L_t0t1,double L_q0q1,double L_r0r1,
                              double Lsfor33[6])
{
      double o0p0, o0s0, p0q0, p0t0, q0o0, q0r0, s0p0, t0q0, r0o0,
               r1s1, r1o1, s1t1, s1p1, t1r1, t1q1, o1s1, p1t1, q1r1,
               A, B, C, D, E, F, k1, k2, k3, k4, k5, k6,
               K0, K1, K2, K3, K4, K5, K6,
               m1, m2, m3, m4, m5, m6;

      o0p0 = param->p0_x0 ;
      o0s0 = param->o0s0  ;
      p0q0 = sqrt(pow(param->p0_x0-param->q0_x0,2) + pow(param->q0_y0,2)) ;
      p0t0 = param->p0t0 ;
      q0o0 = sqrt(pow(param->q0_x0,2) + pow(param->q0_y0,2)) ;
      q0r0 = param->q0r0 ;
      s0p0 = o0p0 - o0s0 ;
```

```
      t0q0 = p0q0 - p0t0 ;
      r0o0 = q0o0 - q0r0 ;
      r1s1 = param->s1_x1 ;
      r1o1 = param->r1o1 ;
      s1t1 = sqrt(pow(param->s1_x1-param->t1_x1,2) + pow(param->t1_y1,2)) ; ;
      s1p1 = param->s1p1 ;
      t1r1 = sqrt(pow(param->t1_x1,2) + pow(param->t1_y1,2)) ;
      t1q1 = param->t1q1 ;
      o1s1 = r1s1 - r1o1 ;
      p1t1 = s1t1 - s1p1 ;
      q1r1 = t1r1 - t1q1 ;

      A = o1s1/r1o1;
      B = p1t1/s1p1;
      C = q1r1/t1q1;
      D = s0p0/o0s0;
      E = t0q0/p0t0;
      F = q0r0/r0o0;

      k1 = (o1s1*r1o1) + (o1s1*o1s1);
      k2 = (p1t1*s1p1) + (p1t1*p1t1);
      k3 = (q1r1*t1q1) + (q1r1*q1r1);
      k4 = (s0p0*o0s0) + (s0p0*s0p0);
      k5 = (t0q0*p0t0) + (t0q0*t0q0);
      k6 = (r0o0*q0r0) + (q0r0*q0r0);

      m1 = r1s1/r1o1;
      m2 = s1t1/s1p1;
      m3 = t1r1/t1q1;
      m4 = o0p0/o0s0;
      m5 = p0q0/p0t0;
      m6 = q0o0/r0o0;

      K0 = (k6 - k3 + C*k5 - C*E*k2 + B*C*E*k4 - B*C*D*E*k1)/(F - A*B*C*D*E);
      K1 = (-B*C*D*E*m1)/(F - A*B*C*D*E);
      K2 = (B*C*E*m4)/(F - A*B*C*D*E);
      K3 = (-C*E*m2)/(F - A*B*C*D*E);
      K4 = (C*m5)/(F - A*B*C*D*E);
      K5 = (-m3)/(F - A*B*C*D*E);
      K6 = (m6)/(F - A*B*C*D*E);

      Lsfor33[0] = sqrt(fabs(K0 + K1*L_o0o1*L_o0o1 + K2*L_s0s1*L_s0s1 +
K3*L_p0p1*L_p0p1
                             + K4*L_t0t1*L_t0t1 + K5*L_q0q1*L_q0q1 +
K6*L_r0r1*L_r0r1));//or
      Lsfor33[1] = sqrt(fabs(k1 + m1*L_o0o1*L_o0o1 -
A*Lsfor33[0]*Lsfor33[0]));//os
      Lsfor33[2] = sqrt(fabs(k4 + m4*L_s0s1*L_s0s1 -
D*Lsfor33[1]*Lsfor33[1]));//ps
      Lsfor33[3] = sqrt(fabs(k2 + m2*L_p0p1*L_p0p1 -
B*Lsfor33[2]*Lsfor33[2]));//pt
      Lsfor33[4] = sqrt(fabs(k5 + m5*L_t0t1*L_t0t1 -
E*Lsfor33[3]*Lsfor33[3]));//qt
      Lsfor33[5] = sqrt(fabs(k3 + m3*L_q0q1*L_q0q1 -
C*Lsfor33[4]*Lsfor33[4]));//qr
```

```
/*    cout << "\n" << Lsfor33[0] << "\n" << Lsfor33[1] << "\n" << Lsfor33[2]
<< "\n" << Lsfor33[3] << "\n" << Lsfor33[4] << "\n" << Lsfor33[5] << "\n";*/
}

/*
Function to perform forward analysis of 33 stewart platform
*/
void solve_platform (int *pnum_solutions,
                 double T_2_1[8][4][4],
                 double p_x_1, double q_x_1, double q_y_1,
                 double s_x_2, double t_x_2, double t_y_2,
                 double L_or, double L_os, double L_ps,
                 double L_pt, double L_qt, double L_qr)
{
   int poly_solve(double root_r[], double root_c[], int d, double coeff[]) ;
   int i ;

   double p_1[3], q_1[3], vk[3];
   p_1[0] = p_x_1;
   p_1[1] = 0.0;
   p_1[2] = 0.0;
   q_1[0] = q_x_1;
   q_1[1] = q_y_1;
   q_1[2] = 0.0;
   vk[0]  = 0.0;
   vk[1]  = 0.0;
   vk[2]  = 1.0;

   double L_op, L_pq, L_oq ;
   L_op = vecmag(p_1);//sqrt(p_1[0]*p_1[0] + p_1[1]*p_1[1] +
p_1[2]*p_1[2]);//!p_1 ;
   L_oq = vecmag(q_1);//sqrt(q_1[0]*q_1[0] + q_1[1]*q_1[1] +
q_1[2]*q_1[2]);//!q_1 ;
   L_pq = sqrt(fabs((p_1[0] - q_1[0])*(p_1[0] - q_1[0])
              +(p_1[1] - q_1[1])*(p_1[1] - q_1[1])
              +(p_1[2] - q_1[2])*(p_1[2] - q_1[2])));//!(p_1 - q_1) ;


   double s_2[3], t_2[3];
   s_2[0] = s_x_2;
   s_2[1] = 0.0;
   s_2[2] = 0.0;
   t_2[0] = t_x_2;
   t_2[1] = t_y_2;
   t_2[2] = 0.0;

   double L_rs, L_rt, L_st ;
   L_rs = vecmag(s_2);//sqrt(s_2[0]*s_2[0] + s_2[1]*s_2[1] +
s_2[2]*s_2[2]);//!s_2 ;
   L_rt = vecmag(t_2);//sqrt(t_2[0]*t_2[0] + t_2[1]*t_2[1] +
t_2[2]*t_2[2]);//!t_2 ;
   L_st = sqrt(fabs((s_2[0] - t_2[0])*(s_2[0] - t_2[0])
              +(s_2[1] - t_2[1])*(s_2[1] - t_2[1])
              +(s_2[2] - t_2[2])*(s_2[2] - t_2[2])));//!(s_2 - t_2) ;

   double c41, s41, c34, s34, c12, s12, c23 ;
   double c41_o, s41_o ;
```

```
   double c41_p, s41_p ;
   double pxq[3];//cross product of p_1 and q_1

/* four bar at point O ////////////////////////////////////*/

   c41_o = c41 = dotproduct(p_1,q_1)/(L_op*L_oq);
   crossproduct(pxq,p_1,q_1);
   s41_o = s41 = (pxq[2]/(L_op*L_oq))*vk[2];

   c23 = (L_or*L_or + L_os*L_os - L_rs*L_rs) / (2.0*L_or*L_os) ;

   c34 = (L_os*L_os + L_op*L_op - L_ps*L_ps) / (2.0*L_os*L_op) ;
   s34 = sin(acos(c34)) ;

   c12 = (L_oq*L_oq + L_or*L_or - L_qr*L_qr) / (2.0*L_oq*L_or) ;
   s12 = sin(acos(c12)) ;

   /* First equation
      AA1 y^2 x^2 + BB1 x^2 + CC1 y^2 + DD1 x y + EE1 = 0  */
   double AA1, BB1, CC1, DD1, EE1 ;

   AA1 = s12 * (s41*c34 - c41*s34) + c12*(c41*c34+s41*s34) - c23 ;
   BB1 = s12 * (c41*s34 + s41*c34) + c12*(c41*c34-s41*s34) - c23 ;
   CC1 = s12 * (c41*s34 - s41*c34) + c12*(c41*c34+s41*s34) - c23 ;
   DD1 = 4.0 * s12 * s34 ;
   EE1 = -s12* (c41*s34 + s41*c34) + c12*(c41*c34-s41*s34) - c23 ;

/* four bar at point P ////////////////////////////////////*/

   double v_pq[3], v_po[3];
   v_pq[0] = q_1[0] - p_1[0];
   v_pq[1] = q_1[1] - p_1[1];
   v_pq[2] = q_1[2] - p_1[2];
   v_po[0] = -p_1[0];
   v_po[1] = -p_1[1];
   v_po[2] = -p_1[2];

   c41 = dotproduct(v_pq,v_po)/(vecmag(v_po)*vecmag(v_pq));
   crossproduct(pxq,v_pq,v_po);
   s41 = (pxq[2]/(vecmag(v_po)*vecmag(v_pq)))*vk[2];

   c41_p = -c41 ;
   s41_p =  s41 ;

   c23 = (L_pt*L_pt + L_ps*L_ps - L_st*L_st) / (2.0*L_pt*L_ps) ;

   c34 = (L_pq*L_pq + L_pt*L_pt - L_qt*L_qt) / (2.0*L_pq*L_pt) ;
   s34 = sin(acos(c34)) ;

   c12 = (L_op*L_op + L_ps*L_ps - L_os*L_os) / (2.0*L_op*L_ps) ;
   s12 = sin(acos(c12)) ;

   /* Third equation
      AA3 y^2 z^2 + BB3 y^2 + CC3 z^2 + DD3 y z + EE3 = 0 */
   double AA3, BB3, CC3, DD3, EE3 ;

   AA3 = s12 * (s41*c34 - c41*s34) + c12*(c41*c34+s41*s34) - c23 ;
```

```
   BB3 = s12 * (c41*s34 + s41*c34) + c12*(c41*c34-s41*s34) - c23 ;
   CC3 = s12 * (c41*s34 - s41*c34) + c12*(c41*c34+s41*s34) - c23 ;
   DD3 = 4.0 * s12 * s34 ;
   EE3 = -s12* (c41*s34 + s41*c34) + c12*(c41*c34-s41*s34) - c23 ;

/* four bar at point Q ////////////////////////////////////*/

   double v_qp[3], v_qo[3];
   v_qp[0] = - v_pq[0] ;
   v_qp[1] = - v_pq[1] ;
   v_qp[2] = - v_pq[2] ;
   v_qo[0] = - q_1[0] ;
   v_qo[1] = - q_1[1] ;
   v_qo[2] = - q_1[2] ;

   c41 = dotproduct(v_qo,v_qp)/(vecmag(v_qo)*vecmag(v_qp));
   crossproduct(pxq,v_qo,v_qp);
   s41 = (pxq[2]/(vecmag(v_qo)*vecmag(v_qp)))*vk[2];

   c23 = (L_qt*L_qt + L_qr*L_qr - L_rt*L_rt) / (2.0*L_qt*L_qr) ;

   c34 = (L_qr*L_qr + L_oq*L_oq - L_or*L_or) / (2.0*L_qr*L_oq) ;
   s34 = sin(acos(c34)) ;

   c12 = (L_pq*L_pq + L_qt*L_qt - L_pt*L_pt) / (2.0*L_pq*L_qt) ;
   s12 = sin(acos(c12)) ;

   /* Second equation
      AA2 z^2 x^2 + BB2 z^2 + CC2 x^2 + DD2 z x + EE2 = 0 */
   double AA2, BB2, CC2, DD2, EE2 ;

   AA2 = s12 * (s41*c34 - c41*s34) + c12*(c41*c34+s41*s34) - c23 ;
   BB2 = s12 * (c41*s34 + s41*c34) + c12*(c41*c34-s41*s34) - c23 ;
   CC2 = s12 * (c41*s34 - s41*c34) + c12*(c41*c34+s41*s34) - c23 ;
   DD2 = 4.0 * s12 * s34 ;
   EE2 = -s12* (c41*s34 + s41*c34) + c12*(c41*c34-s41*s34) - c23 ;

/* Form up the i/o equation.*/
   Poly a1, b1, c1, a2, b2, c2,
      temp1, temp2, a1a2, c1c2, a1c2, a2c1,
      b1b1, b2b2, b1b2, a2c2, c2c2, a1c1, a1a1, c1c1, a2a2,
      DD, p32, p33, p34, p35, p36, alpha, beta, rho1, rho2, ioeqn;

      a1.deg=2; a2.deg=2; c1.deg=2; c2.deg=2; b1.deg=1; b2.deg=1;

      a1.coef[0]=CC1;
      a1.coef[1]=0.0;
      a1.coef[2]=AA1;
      a2.coef[0]=BB2;
      a2.coef[1]=0.0;
      a2.coef[2]=AA2;
      c1.coef[0]=EE1;
      c1.coef[1]=0.0;
      c1.coef[2]=BB1;
      c2.coef[0]=EE2;
      c2.coef[1]=0.0;
      c2.coef[2]=CC2;
```

```
      b1.coef[0]=0.0;
      b1.coef[1]=0.5*DD1;
      b2.coef[0]=0.0;
      b2.coef[1]=0.5*DD2;

/*
      for(i = 0; i<3; ++i)
      {
            cout << "a1.coef[i] = " << a1.coef[i] << "\n";
            cout << "a2.coef[i] = " << a2.coef[i] << "\n";
            cout << "c1.coef[i] = " << c1.coef[i] << "\n";
            cout << "c2.coef[i] = " << c2.coef[i] << "\n";
      }
*/

      pmult( a1, a2, &a1a2 );
      pmult( c1, c2, &c1c2 );
      pmult( a2, c1, &a2c1 );
      pmult( a1, c2, &a1c2 );
      pmult( a2, c2, &a2c2 );
      pmult( c2, c2, &c2c2 );
      pmult( a2, a2, &a2a2 );
      pmult( a1, c1, &a1c1 );
      pmult( c1, c1, &c1c1 );
      pmult( a1, a1, &a1a1 );
      pmult( b1, b1, &b1b1 );
      pmult( b2, b2, &b2b2 );
      pmult( b1, b2, &b1b2 );

      pscale( a2c1, 2.0*AA3*BB3, &temp1);
      pmult(temp1, c1c2, &temp1);
      //pmult(temp1, c1c2, &p1);

/*    cout << AA3 << "\n";
      cout << BB3 << "\n";

      cout << a2c1.coef[0] << "  " << a2c1.coef[1] << "  " << a2c1.coef[2] <<
"  " << a2c1.coef[6] << "\n";
      cout << temp1.coef[0] << "  " << temp1.coef[1] << "  " << temp1.coef[2]
<< "  " << temp1.coef[6] << "\n";
      cout << c1c2.coef[0] << "  " << c1c2.coef[1] << "  " << c1c2.coef[2] <<
"  " << c1c2.coef[6] << "\n";
      cout << p1.coef[0] << "  " << p1.coef[1] << "  " << p1.coef[2] << "  "
<< p1.coef[6] << "\n";
*/
      pscale( c1c1, 4.0*AA3*BB3, &temp2);
      pmult(temp2, b2b2, &temp2);

      psub( temp1, temp2, &DD );

      pscale( a1c1, 2.0*AA3*CC3, &temp1);
      pmult(temp1, c2c2, &temp2);

      padd( DD, temp2, &DD );

      pscale( c2c2, 4.0*AA3*CC3, &temp1);
      pmult(temp1, b1b1, &temp2);
```

```
psub( DD, temp2, &DD );

pscale( a1a2, 2.0*AA3*EE3, &temp1);
pmult(temp1, c1c2, &temp2);

psub( DD, temp2, &DD );

pscale( a1c1, 4.0*AA3*EE3, &temp1);
pmult(temp1, b2b2, &temp2);

padd( DD, temp2, &DD );

pscale( a2c2, 4.0*AA3*EE3, &temp1);
pmult(temp1, b1b1, &temp2);

padd( DD, temp2, &DD );

pscale( b1b1, 8.0*AA3*EE3, &temp1);
pmult(temp1, b2b2, &temp2);

psub( DD, temp2, &DD );

pscale( c1c2, 2.0*AA3*DD3, &temp1);
pmult(temp1, b1b2, &temp2);

psub( DD, temp2, &DD );

pscale( a1a2, 2.0*BB3*CC3, &temp1);
pmult(temp1, c1c2, &temp2);

psub( DD, temp2, &DD );

pscale( a1c1, 4.0*BB3*CC3, &temp1);
pmult(temp1, b2b2, &temp2);

padd( DD, temp2, &DD );

pscale( a2c2, 4.0*BB3*CC3, &temp1);
pmult(temp1, b1b1, &temp2);

padd( DD, temp2, &DD );

pscale( b1b1, 8.0*BB3*CC3, &temp1);
pmult(temp1, b2b2, &temp2);

psub( DD, temp2, &DD );

pscale( a1a2, 2.0*BB3*EE3, &temp1);
pmult(temp1, a2c1, &temp2);

padd( DD, temp2, &DD );

pscale( a2a2, 4.0*BB3*EE3, &temp1);
pmult(temp1, b1b1, &temp2);

psub( DD, temp2, &DD );
```

```
pscale( a2c1, 2.0*BB3*DD3, &temp1);
pmult(temp1, b1b2, &temp2);

psub( DD, temp2, &DD );

pscale( a1a1, 2.0*CC3*EE3, &temp1);
pmult(temp1, a2c2, &temp2);

padd( DD, temp2, &DD );

pscale( a1a1, 4.0*CC3*EE3, &temp1);
pmult(temp1, b2b2, &temp2);

psub( DD, temp2, &DD );

pscale( a1c2, 2.0*CC3*DD3, &temp1);
pmult(temp1, b1b2, &temp2);

psub( DD, temp2, &DD );

pscale( a1a2, 2.0*DD3*EE3, &temp1);
pmult(temp1, b1b2, &temp2);

psub( DD, temp2, &DD );

pscale( a1a2, DD3*DD3, &temp1);
pmult(temp1, c1c2, &temp2);

psub( DD, temp2, &DD );

pscale( c1c1, AA3*AA3, &temp1);
pmult(temp1, c2c2, &temp2);

psub( DD, temp2, &DD );

pscale( a2a2, BB3*BB3, &temp1);
pmult(temp1, c1c1, &temp2);

psub( DD, temp2, &DD );

pscale( a1a1, CC3*CC3, &temp1);
pmult(temp1, c2c2, &temp2);

psub( DD, temp2, &DD );

pscale( a1a1, EE3*EE3, &temp1);
pmult(temp1, a2a2, &temp2);

psub( DD, temp2, &alpha );

pscale( b1b2, 4.0*AA3*EE3, &temp1);
pscale( c1c2, AA3*DD3, &temp2);
      padd(temp1, temp2, &beta);
pscale( a1a2, DD3*EE3, &temp1);
      padd(beta, temp1, &beta);
pscale( b1b2, 4.0*BB3*CC3, &temp1);
```

```
            psub(beta, temp1, &beta);
      pscale( a2c1, BB3*DD3, &temp1);
            psub(beta, temp1, &beta);
      pscale( a1c2, CC3*DD3, &temp1);
            psub(beta, temp1, &beta);

      psub(b1b1, a1c1, &rho1);
      psub(b2b2, a2c2, &rho2);

      pmult(alpha, alpha, &p32);
      pmult(beta, beta, &p33);
      pmult(rho1, rho2, &p34);
      pscale(p33, 4.0, &p35);
      pmult(p35, p34, &p36);
      psub(p32, p36, &ioeqn);
/*   for (i=0; i<17; ++i)
     {
           cout << "ioeqn[i] = " << ioeqn.coef[i] << "\n";
     }
*/

   double unitval, tempunitval ;
   unitval = ioeqn.coef[16] ;
   tempunitval = 1.0/unitval;
   pscale(ioeqn, tempunitval, &ioeqn);

   double coef2[9] ;
   for (i=8 ; i>=0 ; --i)
     {
      coef2[i] = ioeqn.coef[2*i] ;
/*      cout << "coef2[i] = " << coef2[i] << "\n";*/
     }

   double xsq_r[8], xsq_c[8] ;
   int OK ;
   OK = poly_solve(xsq_r, xsq_c, 8, coef2) ;
   if (OK != 1)
     {
//      cout << "\nERROR in poly_solve\n\n" ;
      exit(9) ;
     }

   int num_real = 0 ;
   double xx[8], yy[8], zz[8] ;
   for (i=0 ; i<8 ; ++i)
      if (valuenear(xsq_c[i], 0.0, 0.0001) && (xsq_r[i] >= 0.0))
      {
       xx[num_real] = sqrt(fabs(xsq_r[i])) ;
       num_real++ ;
      }

   *pnum_solutions = num_real ;

   /* Find corresponding values for thetay and thetaz.*/

   double y_candidate[2], z_candidate[2] ;
   double aa1, bb1, cc1, aa2, bb2, cc2 ;
```

```
   double aa3, bb3, cc3, dd3 ;
   double discr ;
   int badone[8]={0,0,0,0,0,0,0,0} ;
   double cand_value[4] ;
   double ang, cos_ang ;  // fold angles

   /* Get coordinates of points r, s, and t in the 1st coord. system
      (get the coordinates in the 1st system, then fold the triangles)
      get point s coordinates before folding */
   double sx_prefold, sy_prefold ;
   double sx, sy, sz ;
   cos_ang = (L_op*L_op + L_os*L_os - L_ps*L_ps) / (2.0*L_op*L_os) ;
   ang = acos(cos_ang) ;
   sx_prefold =   L_os*cos_ang ;
   sy_prefold = - L_os*sin(ang) ;

   /* get point r in xtra coordinate system before folding*/
   double rx_prefold, ry_prefold ;
   double rx, ry, rz ;
   cos_ang = (L_or*L_or + L_oq*L_oq - L_qr*L_qr) / (2.0*L_or*L_oq) ;
   ang = acos(cos_ang) ;
   rx_prefold =   L_or*cos_ang ;
   ry_prefold =   L_or*sin(ang) ;

   /* get point t in xtra2 coordinate system before folding*/
   double tx_prefold, ty_prefold ;
   double tx, ty, tz ;
   cos_ang = (L_pt*L_pt + L_pq*L_pq - L_qt*L_qt) / (2.0*L_pt*L_pq) ;
   ang = acos(cos_ang) ;
   tx_prefold =   L_pt*cos_ang ;
   ty_prefold = - L_pt*sin(ang) ;

   double thetax, thetay, thetaz ;
   double sin_x, cos_x, sin_y, cos_y, sin_z, cos_z ;

   double xvec[3], yvec[3], zvec[3], tempvec[3];
/*   cdc_vector xvec(3L), yvec(3L), zvec(3L), tempvec(3L) ;*/

   for (i=0 ; i< *pnum_solutions ; ++i)
     {
      aa1 = a1.eval(xx[i]) ;
/*            cout << "\naa1 = " << aa1;*/
      bb1 = b1.eval(xx[i]) ;
/*            cout << "\nbb1 = " << bb1;*/
      cc1 = c1.eval(xx[i]) ;
/*            cout << "\ncc1 = " << cc1;*/
      aa2 = a2.eval(xx[i]) ;
/*            cout << "\naa2 = " << aa2;*/
      bb2 = b2.eval(xx[i]) ;
/*            cout << "\nbb2 = " << bb2;*/
      cc2 = c2.eval(xx[i]) ;
/*            cout << "\ncc2 = " << cc2;*/
      discr = 4.0*bb1*bb1 - 4.0*aa1*cc1 ;
      if (discr < 0)
            {
                  badone[i] = TRUE ;
//                cout << "bady"<< discr << endl ;
```

```
                continue ;
        }
      y_candidate[0] = (-2.0*bb1 + sqrt(discr)) / (2.0*aa1) ;
      y_candidate[1] = (-2.0*bb1 - sqrt(discr)) / (2.0*aa1) ;

      discr = 4.0*bb2*bb2 - 4.0*aa2*cc2 ;
      if (discr < 0)
            {
                  badone[i] = TRUE ;
//                cout << "badz"<< discr <<endl ;
                  continue ;
            }
      z_candidate[0] = (-2.0*bb2 + sqrt(discr)) / (2.0*aa2) ;
      z_candidate[1] = (-2.0*bb2 - sqrt(discr)) / (2.0*aa2) ;

      aa3 = 4.0*AA3*bb1*bb2 + DD3*aa1*aa2 ;
      bb3 = 2.0*AA3*bb1*cc2 - 2.0*BB3*aa2*bb1 ;
      cc3 = 2.0*AA3*bb2*cc1 - 2.0*CC3*aa1*bb2 ;
      dd3 = AA3*cc1*cc2 + EE3*aa1*aa2 - BB3*aa2*cc1 - CC3*aa1*cc2 ;

      cand_value[0] = fabs(aa3*y_candidate[0]*z_candidate[0]
                + bb3*y_candidate[0] + cc3*z_candidate[0] + dd3) ;

      cand_value[1] = fabs(aa3*y_candidate[1]*z_candidate[0]
                + bb3*y_candidate[1] + cc3*z_candidate[0] + dd3) ;

      cand_value[2] = fabs(aa3*y_candidate[0]*z_candidate[1]
                + bb3*y_candidate[0] + cc3*z_candidate[1] + dd3) ;

      cand_value[3] = fabs(aa3*y_candidate[1]*z_candidate[1]
                + bb3*y_candidate[1] + cc3*z_candidate[1] + dd3) ;
      if ((cand_value[0] < cand_value[1]) && (cand_value[0] < cand_value[2])
             && (cand_value[0] < cand_value[3]))
            {
                  yy[i] = y_candidate[0] ;
                  zz[i] = z_candidate[0] ;
            }

      else if ((cand_value[1] < cand_value[0]) &&
         (cand_value[1] < cand_value[2]) && (cand_value[1] < cand_value[3]))
            {
                  yy[i] = y_candidate[1] ;
                  zz[i] = z_candidate[0] ;
            }

      else if ((cand_value[2] < cand_value[0]) &&
         (cand_value[2] < cand_value[1]) && (cand_value[2] < cand_value[3]))
            {
                  yy[i] = y_candidate[0] ;
                  zz[i] = z_candidate[1] ;
            }

      else if ((cand_value[3] < cand_value[0]) &&
         (cand_value[3] < cand_value[1]) && (cand_value[3] < cand_value[2]))
            {
                  yy[i] = y_candidate[1] ;
                  zz[i] = z_candidate[1] ;
```

```
      }
    thetax = 2.0*atan(xx[i]) ;
    thetay = 2.0*atan(yy[i]) ;
    thetaz = 2.0*atan(zz[i]) ;

    sin_x = sin(thetax) ;   cos_x = cos(thetax) ;
    sin_y = sin(thetay) ;   cos_y = cos(thetay) ;
    sin_z = sin(thetaz) ;   cos_z = cos(thetaz) ;

    sx =  sx_prefold ;
    sy =  cos_y * sy_prefold ;
    sz = -sin_y * sy_prefold ;

    rx = c41_o * rx_prefold - s41_o * cos_x * ry_prefold ;
    ry = s41_o * rx_prefold + c41_o * cos_x * ry_prefold ;
    rz = sin_x * ry_prefold ;

    tx = c41_p * tx_prefold - s41_p * cos_z * ty_prefold + L_op ;
    ty = s41_p * tx_prefold + c41_p * cos_z * ty_prefold ;
    tz = -sin_z * ty_prefold ;

    /* Enter origin of 2nd coord system as seen in 1st*/
    T_2_1[i][0][3] = rx ;
    T_2_1[i][1][3] = ry ;
    T_2_1[i][2][3] = rz ;
    T_2_1[i][3][3] = 1.0 ;

    /* Enter x axis of 2nd coord system as seen in 1st*/
    xvec[0] = sx - rx ;
    xvec[1] = sy - ry ;
    xvec[2] = sz - rz ;
      double tempmag;
      tempmag = vecmag(xvec);
      xvec[0] = xvec[0]/tempmag;
      xvec[1] = xvec[1]/tempmag;
      xvec[2] = xvec[2]/tempmag;
/*    xvec = ~xvec ;*/
    T_2_1[i][0][0] = xvec[0] ;
    T_2_1[i][1][0] = xvec[1] ;
    T_2_1[i][2][0] = xvec[2] ;
    T_2_1[i][3][0] = 0.0 ;

    /* Enter z axis of 2nd coord system as seen in 1st*/
    tempvec[0] = tx - rx ;
    tempvec[1] = ty - ry ;
    tempvec[2] = tz - rz ;
/*    zvec = xvec ^ tempvec;*/
      crossproduct(zvec,xvec,tempvec) ;
      tempmag = vecmag(zvec);
      zvec[0] = zvec[0]/tempmag;
      zvec[1] = zvec[1]/tempmag;
      zvec[2] = zvec[2]/tempmag;
/*    zvec = ~zvec ;*/
    T_2_1[i][0][2] = zvec[0] ;
    T_2_1[i][1][2] = zvec[1] ;
    T_2_1[i][2][2] = zvec[2] ;
    T_2_1[i][3][2] = 0.0 ;
```

```
      /* Enter y axis of 2nd coord system as seen in 1st*/

        crossproduct(yvec,zvec,xvec);
        tempmag = vecmag(yvec);
        yvec[0] = yvec[0]/tempmag;
        yvec[1] = yvec[1]/tempmag;
        yvec[2] = yvec[2]/tempmag;

/*      yvec = zvec ^ xvec ;*/
/*      yvec = ~yvec ;*/
      T_2_1[i][0][1] = yvec[0] ;
      T_2_1[i][1][1] = yvec[1] ;
      T_2_1[i][2][1] = yvec[2] ;
      T_2_1[i][3][1] = 0.0 ;
     }

}

/*Function to multiply two matrices and return the answer*/
void matmult(double ans[4][4],double matrix1[4][4], double matrix2[4][4])
{
   int i,j,k;

   for(i=0;i<4;i++)
   {
      for(j=0;j<4;j++)
      {
         for(k=0;k<4;k++)
       {
          ans[i][j]+=matrix1[i][k]*matrix2[k][j];
       }
      }
   }
}

/*Function to multiply a matrix times a vector and return the answer*/
void vecmult(double ans1[4], double matrix1[4][4], double vector1[4])
{
   int i,j;
   for(i=0;i<4;i++)
         ans1[i] = 0;

   for(i=0;i<4;i++)
   {
      for(j=0;j<4;j++)
      {
       ans1[i]+=matrix1[i][j]*vector1[j];
      }
   }
}

double dotproduct(double vector1[3], double vector2[3])
{
      double ans = 0;
      int i;
```

```
      for(i=0;i<3;i++)
      {
            ans += vector1[i]*vector2[i];
      }
      return ans;
}

void crossproduct(double ans[3], double vector1[3], double vector2[3])
{
      ans[0] = vector1[1]*vector2[2]-vector1[2]*vector2[1];
      ans[1] = vector1[2]*vector2[0]-vector1[0]*vector2[2];
      ans[2] = vector1[0]*vector2[1]-vector1[1]*vector2[0];
}

double vecmag(double vector[3])
{
      double ans;
      ans = sqrt(vector[0]*vector[0] + vector[1]*vector[1] +
vector[2]*vector[2]);
      return ans;
}

int valuenear (double val, double goal, double tol)
{
   if ((val > goal-tol) && (val < goal+tol))
      return 1 ;
   else
      return 0 ;
}

int _INT[100];
char _CHAR[50];


void Inverse(double matdata[], int numcol, double *det,
                  double invary[])
{
   int  *pivlst, i, j, k, l, lerow, lecol, l1;
   char *pivchk;
   double piv, t, leval;

   pivlst = &(_INT[0]); // (int *) calloc(50*2,2)  ;
   pivchk = &(_CHAR[0]); // (char *) calloc(50,1)  ;
   (*det) = 1.0;
   for ( i = 0; i <= numcol-1; ++i ) {
      pivchk[i] = 0;
       for ( j = 0; j <= numcol-1; ++j )
       invary[i*numcol+j] = matdata[i*numcol+j];
   }
   for ( i = 0; i <= numcol-1; ++i ) {
      leval = 0.0;
      for ( j = 0; j <= numcol-1; ++j ) {
       if ( ! (pivchk[j]) ) {
          for ( k = 0; k <= numcol-1; ++k ) {
             if ( ! (pivchk[k]) ) {
              if ( fabs(invary[j*numcol+k]) > leval ) {
                 lerow = j;
```

```
                lecol = k;
                leval = fabs(invary[j*numcol+k]);
             }
            }
         }
       }
      }
      pivchk[lecol] = 1;
      pivlst[i*2] = lerow;
      pivlst[i*2+1] = lecol;
      if ( lerow != lecol ) {
       (*det) = -(*det);
       for ( l = 0; l <= numcol-1; ++l ) {
              MatSwap(&invary[lerow*numcol+l],
                  &invary[lecol*numcol+l]);
       }
      }
      piv = invary[lecol*numcol+lecol];
      (*det) = (*det) * piv;
      if ( (*det) > 1.0e+30 ) {
       (*det) = 1;
      }
      invary[lecol*numcol+lecol] = 1.0;
      for ( l = 0; l <= numcol-1; ++l ) {
          invary[lecol*numcol+l] =invary[lecol*numcol+l]/piv;
      }
      for ( l1 = 0; l1 <= numcol-1; ++l1 ) {
       if ( l1 != lecol ) {
          t = invary[l1*numcol+lecol];
          invary[l1*numcol+lecol] = 0;
          for ( l = 0; l <= numcol-1; ++l ) {
                  invary[l1*numcol+l]=invary[l1*numcol+l] -
                                      invary[lecol*numcol+l]*t;
          }
       }
      }
   }
   for ( i = 0; i <= numcol-1; ++i ) {
      l = numcol - i -1;
      if ( pivlst[l*2] != pivlst[l*2+1] ) {
       lerow = pivlst[l*2];
       lecol = pivlst[l*2+1];
          for ( k = 0; k <= numcol-1; ++k )
             MatSwap(&invary[k*numcol+lerow],
                  &invary[k*numcol+lecol]);
      }
   }

//   free(pivlst); free(pivchk);

}//   Inverse( (double*)J, 6, &c, (double*)Jinv );

/*******************************************************/

void MatSwap(double *s1, double *s2)
{
   double temp;
```

```
   temp = (*s1);
   (*s1) = (*s2);
   (*s2) = temp;
}

/*********************************************************/

void Transpose(double *a, double *b, int m, int n)
{
   int i,j;
   for(i=0 ; i<m ; i++)
    for(j=0 ; j<n ; j++)
     *(b+(j*m)+i) = *(a+(i*n)+j);
}


/*********************************************************/
int poly_solve (double root_r[], double root_c[], int d, double xcof[])

/* This routine will evaluate the roots of a polynomial of
      degree "d" ("d" must be less than or equal to 36).
   "root_r" and "root_c" are the real and complex parts of the
      'd' solutions to the original equation.
   "xcof" is an array of coefficients, ordered from smallest
      to largest power.

   xcof[16] x^16 + xcof[15] x^15 + ... + xcof[1] x + xcof[0] = 0 */

{
   double coef[37], dis, X, Y, Z[37], X0, XX[40],YY[40],
       U, V, dUx, dUy, den , dX, dY, dXY, XY, C, B[40] ;
   int i, k, deg, cnt ;
   int lst, lflip, ltry ;

   if (d > 36)
      return (0) ;

   for (i=0 ; i<=d ; ++i)
      coef[i] = xcof[i] ;

   deg = d ;

   while (coef[deg] == 0.0)
      deg-- ;  /*The leading coefficient was zero.*/

   if (deg <1)
      return (-1) ;  /*The polynomial must be at least of degree 1.*/

   cnt = 0 ;  /*cnt keeps track of the number of roots found */

   if (deg == 1)
      goto solve_linear ;

   if (deg == 2)
      goto solve_quad ;
```

```
   /*************************/
   /*  Set initial values.   */
   /*************************/
L30:
   lst = 0 ;   /*lst counts the number of different starting values*/
   lflip = 0 ; /*lflip determines whether the inverse polynomial is
             being considered */

   X = 0.00608 ;
   Y = 0.2939 ;

L35:
   X0 = X ;
   X  = -5.0*Y ;
   Y  =  2.0*X0 ;

   ltry = 0 ;  /*ltry counts the # of interations for a starting value*/

   lst++ ;

L38:
   XX[0] = 1.0 ;
   YY[0] = 0.0 ;

   for (i=1 ; i<=deg ; ++i)
     /*Evaluate x^16, x^15, etc where x is complex*/
     {XX[i] = X * XX[i-1] - Y * YY[i-1] ;
      YY[i] = X * YY[i-1] + Y * XX[i-1] ;   /*line 40*/
     }
   U = coef[0] ;
   V = 0.0 ;

   for (i=1 ; i<=deg ; ++i) /*Evaluate the polynomial. */
     {U += coef[i] * XX[i] ;
      V += coef[i] * YY[i] ;
     }

   dUx = 0.0 ;
   dUy = 0.0 ;

   for (i=1 ; i<=deg ; ++i)
     {dUx += i*coef[i] * XX[i-1] ;
      dUy -= i*coef[i] * YY[i-1] ;  /*line 60*/
     }
   den = dUx*dUx + dUy*dUy ;

   dX =  (V*dUy - U*dUx)/den ;
   dY = -(U*dUy + V*dUx)/den ;

   X += dX ;   /*Next try for root. */
   Y += dY ;

   if (Fabs(X) < 40.0)
     {dXY = Sqrt(dX*dX + dY*dY) ;
      XY =  Sqrt(X*X + Y*Y) ;

      if (Fabs(dXY/XY) > 0.0000000002)   /*was 0.0000001 */
```

```
      {ltry++ ;
       if (ltry<400)     /*was 300*/
          goto L38 ;
       else
          goto flip_poly ;
       }
       else
        goto reduce_poly ;
      }
flip_poly:
   lflip++ ;
   ltry = 0 ;

   for (k=0 ; k<=deg ; ++k)
      Z[k] = coef[deg-k] ;

   for (k=0 ; k<=deg ; ++k)
      coef[k] = Z[k] ;

   if (lflip ==1)
     {X = 0.189 ;
      Y = -0.132 ;
      goto L38 ;
     }

   if (lflip ==2)
      if (lst < 4 )
         goto L35 ;
      return (-300) ;  /*A solution was not found for 300 iterations
                         for 4 starting values. */

   /*************************/
reduce_poly:

   if (Fabs(Y) < 0.000006)  /*was 0.0000005*/
      Y = 0.0 ;
   cnt++ ;

   if (lflip ==1)
     {for (k=0 ; k<=deg ; ++k)  /*flip it back*/
       Z[k] = coef[deg-k] ;
      for (k=0 ; k<=deg ; ++k)
         coef[k] = Z[k] ;

      den = X*X + Y*Y ; /*The root to the orig. eqn is 1/(X+iY)*/
      root_r[cnt-1] = X = X/den ;
      root_c[cnt-1] = Y = Y/den ;
     }

   else
     {root_r[cnt-1] = X ;
      root_c[cnt-1] = Y ;
     }

   if (Y==0.0)
     {/*Reduce the equation by one degree.*/
```

```
      C = X ;
      B[deg] = 0.0 ;
      for (k=deg-1 ; k>=0 ; --k)
       B[k] = coef[k+1] + C * B[k+1] ;  /*115*/

      deg-- ;  /*Reduce the degree of the polynomial by 1*/

      for (k=0 ; k<=deg ; ++k)
         coef[k] = B[k] ;

      if (deg ==2)
       goto solve_quad ;
      else if (deg ==1)
         goto solve_linear ;

      else
       goto L30 ;
     }

   else
     {/*Reduce the equation by the complex conjugates.*/
      cnt++ ;
      root_r[cnt-1] =  X ;
      root_c[cnt-1] = -Y ;

      B[deg-2] = coef[deg] ;
      B[deg-3] = coef[deg-1] + 2.0* X * B[deg-2] ;

      for (k=deg-4 ; k>=0 ; --k)
      {B[k] = coef[k+2]  - (X*X+Y*Y) * B[k+2] + 2.0 * X * B[k+1] ;
        }
      deg -= 2 ;

      for (k=0 ; k<=deg ; ++k)
       coef[k] = B[k] ;

      if (deg==2)
         goto solve_quad ;
      if (deg==1)
       goto solve_linear ;
      else
         goto L30 ;
     }


   /*************************/
solve_quad:
   dis = coef[1]*coef[1] - 4.0*coef[2]*coef[0] ;

   X = -coef[1] / (2.0*coef[2]) ;

   if (dis>= 0.0)
     {Y = Sqrt(dis) / (2.0*coef[2]) ;
      root_r[cnt] = X+Y ;
      root_r[cnt+1] = X-Y ;
      root_c[cnt] = root_c[cnt+1] = 0.0 ;
     }
```

```
   else
     {Y = Sqrt(-dis)/ (2.0*coef[2]) ;
      root_r[cnt] =   root_r[cnt+1] = X ;
      root_c[cnt] = -(root_c[cnt+1] = Y) ;
     }
   return (1) ;

solve_linear:
   root_r[cnt] = -coef[0] / coef[1] ;
   root_c[cnt] = 0.0 ;
   return (1) ;
}

/*****************************************************/

/*******************************************************

multiplies two Polynomials:

      a[]={ a0, a1, a2, ..., a(da) }
      b[]={ b0, b1, b2, ..., b(db) }
      ab[]={ ab0, ab1, ab2, ..., ab(da+db) }

*******************************************************/

void pmult( Poly A, Poly B, Poly *AB )
{
      int i, j, da, db;
      double *a, *b, *ab;

      da=A.deg;
      db=B.deg;
      a=A.coef;
      b=B.coef;
      ab=AB->coef;
      AB->deg=da+db;

      for(i=0; i<da+db+1; i++)
            ab[i]=0.;

      for(i=0; i<=da; i++)
            for(j=0; j<=db; j++)
                  ab[i+j] += a[i] * b[j];
}

/*******************************************************
subtract two Polys:
*******************************************************/

void psub( Poly A, Poly B, Poly *A_B )
{
      int i, ds, db;
      double *a=A.coef, *b=B.coef, *a_b=A_B->coef;

      if( A.deg > B.deg ) {
            db=A.deg;
```

```
            ds=B.deg;
            for(i=0; i<=db; i++) {
                  if( i<=ds )
                        a_b[i] = a[i] - b[i];
                  else
                        a_b[i] = a[i];
                  }
            }
      else {
            db=B.deg;
            ds=A.deg;
            for(i=0; i<=db; i++) {
                  if( i<=ds )
                        a_b[i] = a[i] - b[i];
                  else
                        a_b[i] = -b[i];
                  }
            }

      A_B->deg=db;

}

/*********************************************************
adds two Polys:

      ba= b + a;

*********************************************************/

void padd( Poly A, Poly B, Poly *A_B )
{
      int i, ds, db;
      double *a=A.coef, *b=B.coef, *a_b=A_B->coef;

      if( A.deg > B.deg ) {
            db=A.deg;
            ds=B.deg;
            for(i=0; i<=db; i++) {
                  if( i<=ds )
                        a_b[i] = a[i] + b[i];
                  else
                        a_b[i] = a[i];
                  }
            }
      else {
            db=B.deg;
            ds=A.deg;
            for(i=0; i<=db; i++) {
                  if( i<=ds )
                        a_b[i] = a[i] + b[i];
                  else
                        a_b[i] = b[i];
                  }
            }

      A_B->deg=db;
```

```
}

/*********************************************************
scales a Poly:
*********************************************************/

void pscale( Poly A, double s, Poly *AS )
{
      int i;
      double *a=A.coef, *as=AS->coef;

      for(i=0; i<=A.deg; i++)
            as[i] = s*a[i];

      AS->deg=A.deg;
}
```

APPENDIX C
SOURCE CODE FOR REVERSE KINEMATIC ANALYSIS OF PUMA 762 ROBOT

The source code of the C++ program written by Crane to perform the reverse kinematic analysis of Puma 762 robot is given below:

main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "vmath.h"

// constant mechanism parameters for Puma robot  (mm)
#define a23 650.0  //mm
#define a34 0.0
#define S2  190.0
#define S4  600.0

void  reverse_puma (cdc_vector P_tool_6, cdc_vector P_tool_F,
                    cdc_vector S6_F,     cdc_vector a67_F, double S6,
                    int *pvalid, double angles_rad[6][8]) ;

FILE *fp ;

void main ()
{
   fp = fopen ("out.txt", "w") ;
   int i, j ;

// PUMA example
   cdc_vector P_tool_6( 0.0,0.0,0.0) ; //127, 76.2, 177.8
   cdc_vector P_tool_F(-725.0,190.0,650.0) ;
   cdc_vector S6_F (-1.0,0.0,0.0) ;
   cdc_vector a67_F(0.0,0.0,-1.0) ;


   int valid=0 ;        // will be set equal to the number of valid answers

   double angles_rad[6][8] ;  // solution angles in radians

   for (i=0 ; i<8 ; ++i)
      for (j=0 ; j<6 ; ++j)
         angles_rad[j][i] = 0.0 ;

   S6_F  = ~S6_F ;    // unitize the orientation vectors
   a67_F = ~a67_F ;

   reverse_puma (P_tool_6, P_tool_F, S6_F, a67_F, S6,
                 &valid, angles_rad) ;

   fprintf (fp, "\nINPUTS:\n") ;
   fprintf (fp, "\tS6 = %8.3lf\n", S6) ;
   fprintf (fp, "\tP_tool_6:\t %10.3f  %10.3f   %10.3f\n",
                 P_tool_6[0], P_tool_6[1], P_tool_6[2]) ;
   fprintf (fp, "\tP_tool_F:\t %10.3f  %10.3f   %10.3f\n",
                 P_tool_F[0], P_tool_F[1], P_tool_F[2]) ;
   fprintf (fp, "\tS6_F:\t %10.3f  %10.3f   %10.3f\n",
                 S6_F[0], S6_F[1], S6_F[2]) ;
   fprintf (fp, "\ta67_F:\t %10.3f  %10.3f   %10.3f\n",
                 a67_F[0], a67_F[1], a67_F[2]) ;
```

```
   fprintf (fp, "\nOUTPUTS:\n") ;
   fprintf (fp, "\tvalid solutions = %d\n", valid) ;

   if (valid > 0)
      fprintf (fp, "\n      phi1       th2         th3         th4
th5         th6    degrees\n\n") ;
   else
      fprintf (fp, "\nNo solutions exist.\n") ;

   for (i=0 ; i<valid ; ++i)
     {
      fprintf (fp, "%10.3lf  %10.3lf  %10.3lf  %10.3lf  %10.3lf
%10.3lf\n\n",
          angles_rad[0][i]*R2D, angles_rad[1][i]*R2D, angles_rad[2][i]*R2D,
          angles_rad[3][i]*R2D, angles_rad[4][i]*R2D, angles_rad[5][i]*R2D);
     }

   fclose (fp) ;
}


void  reverse_puma (cdc_vector P_tool_6, cdc_vector P_tool_F,
                    cdc_vector S6_F,     cdc_vector a67_F, double S6,
                    int *pvalid, double angles_rad[6][8])

{
   double angles_loc[6][8] ;  // A local array to hold the solutions.
   int valid_array[8] ;

   double th1_rad[8] ;

   int valid_loc ;

   int i, j, k ;

   // Close the loop variables.
   double S7, a71, S1 ;
   double th7_rad, al71_rad, gam1_rad ;

   CloseLoop (P_tool_6, P_tool_F, S6_F, a67_F,
              &S7, &a71, &S1, &th7_rad, &al71_rad, &gam1_rad) ;

   double s71, c71 ;
   double s7,  c7  ;

fprintf (fp,"S7 = %lf, a71 = %lf, S1 = %lf\n", S7, a71, S1) ;
fprintf (fp,"th7= %lf, al71= %lf, gam = %lf  degrees\n", th7_rad*R2D,
al71_rad*R2D,
           gam1_rad*R2D) ;

   s7  = sin (th7_rad) ;
   c7  = cos (th7_rad) ;
   s71 = sin (al71_rad) ;
   c71 = cos (al71_rad) ;

   //       SOLVE FOR PHI1  ////////////////////////////////////////////
```

```
   double X7, Y7, Z7 ;

   X7 = s7 ;
   Y7 = -c71*c7 ;
   Z7 = -s71*c7 ;
   double A, B, D ;
   A = S6*Y7 - S7*s71 ;  // equation (10.8)
   B = S6*X7 + a71 ;
   D = S2 ;
fprintf (fp,"A = %lf  B = %lf  D = %lf\n", A ,B, D) ;

   int OK = 0 ;
   double ang1a, ang1b ;

   OK = solve_trig(&ang1a, &ang1b, A, B, D) ;

   if (!OK)
     {*pvalid = 0 ;   // If not OK, then there will be no solutions.
      return ;
     }

   for (i=0 ; i<8 ; ++i)
      valid_array[i] = 1 ;

   for (i=0 ; i<4 ; ++i)
     {
      angles_loc[0][i]   = ang1a - gam1_rad ;
      angles_loc[0][i+4] = ang1b - gam1_rad ;
      th1_rad[i]   = ang1a ;
      th1_rad[i+4] = ang1b ;
     }

   //       SOLVE FOR TH3   ////////////////////////////////////////////

   double s1, c1 ;
   double X1, Y1 ;
   double AA, BB, DD ;
   double ang3a, ang3b ;

   double X71, Y71 ;

   for (i=0 ; i <2 ; ++i)
     {
      if (!valid_array[i])
         continue ;

      s1 = sin(th1_rad[i*4]) ;
      c1 = cos(th1_rad[i*4]) ;

      X1 = s71*s1 ;
      Y1 = -c71 ;

      X71 = X7*c1 - Y7*s1 ;
      Y71 = -Z7 ;

      A = -S6*X71 - S7*X1 - a71*c1 ;         // equation 10.13
      B =  S1 - S6*Y71 - S7*Y1 ;             // equation 10.14
```

```
   AA = 2.0*a23*a34 ;
   BB = -2.0*a23*S4 ;
   DD = a23*a23 + a34*a34 + S4*S4 - A*A - B*B ;

   OK = solve_trig (&ang3a, &ang3b, AA, BB, DD) ;

   if (!OK)
     {if (i==0)
        for (j=0 ; j<4 ; ++j)
           valid_array[j] = 0 ;
      else
        for (j=0 ; j<4 ; ++j)
           valid_array[j+4] = 0 ;
     }
   else
     {
      if (i==0)
        {
         angles_loc[2][0] = angles_loc[2][1] = ang3a ;
         angles_loc[2][2] = angles_loc[2][3] = ang3b ;
        }
      else
        {
         angles_loc[2][4] = angles_loc[2][5] = ang3a ;
         angles_loc[2][6] = angles_loc[2][7] = ang3b ;
        }
     }
  }

//       SOLVE FOR TH2   ////////////////////////////////////////////
double s3, c3 ;
double coef1, coef2, coef3, coef4 ;
double s2, c2 ;

for (i=0 ; i <8 ; i+=2)
  {
   if (!valid_array[i])
      continue ;

   s1 = sin(th1_rad[i]) ;
   c1 = cos(th1_rad[i]) ;

   s3 = sin(angles_loc[2][i]) ;
   c3 = cos(angles_loc[2][i]) ;

   X1 = s71*s1 ;
   Y1 = -c71 ;

   X71 = X7*c1 - Y7*s1 ;
   Y71 = -Z7 ;

   A = -S6*X71 - S7*X1 - a71*c1 ;         // equation 10.13
   B =  S1 - S6*Y71 - S7*Y1 ;             // equation 10.14

   coef1 = a23 + a34*c3 - S4*s3 ;
   coef2 = -a34*s3 - S4*c3 ;
```

```
      coef3 = -coef1 ;
      coef4 = coef2 ;


      solve_pair (&c2, &s2, coef1, coef2, A, coef4, coef3, B, &OK) ;

//cout << "\n1st equation:  " << coef1 << " c2 + " << coef2 << " s2 = " <<A ;
//cout << "\n2nd equation:  " << coef4 << " c2 + " << coef3 << " s2 = " <<B ;
//cout << "\n\tc2 " << c2 << "     s2 = " << s2 << endl ;
      if (!OK)
        {valid_array[i] = valid_array[i+1] = 0 ;
         continue ;
        }

      angles_loc[1][i] = angles_loc[1][i+1] = atan2(s2, c2) ;

     }


   //       SOLVE FOR TH5   ////////////////////////////////////////
   double Z7123 ;
   double X712, Y712 ;

   for (i=0 ; i<8 ; i+=2)
     {
      if (!valid_array[i])
         continue ;

      s1 = sin(th1_rad[i]) ;
      c1 = cos(th1_rad[i]) ;

      s3 = sin(angles_loc[2][i]) ;
      c3 = cos(angles_loc[2][i]) ;

      s2 = sin(angles_loc[1][i]) ;
      c2 = cos(angles_loc[1][i]) ;

      X71 = X7*c1 - Y7*s1 ;
      Y71 = -Z7 ;

      X712 = X71*c2 - Y71*s2 ;
      Y712 = X71*s2 + Y71*c2 ;

      Z7123 = -(X712*s3 + Y712*c3) ;
fprintf(fp, "th1 = %8.4lf\tth2 = %8.4lf\tth3 = %8.4lf\n", th1_rad[i],
angles_loc[1][i], angles_loc[2][i]) ;
fprintf(fp, "X712 = %8.5lf\tY712 = %8.5lf\tZ7123 = %8.4lf\n\n", X712, Y712,
Z7123) ;

      if (fabs(Z7123) > 1.0)
        {
         valid_array[i]= valid_array[i+1] = 0 ;
         continue ;
        }

      angles_loc[4][i] = acos (-Z7123) ;
      angles_loc[4][i+1] =  -angles_loc[4][i] ;
```

```
  }

//       SOLVE FOR TH4   ////////////////////////////////////////
double X7123, Y7123 ;
double s5, c5 ;
double Z71, Z712 ;

for (i=0 ; i<8 ; ++i)
  {
   if (!valid_array[i])
      continue ;

   s1 = sin(th1_rad[i]) ;
   c1 = cos(th1_rad[i]) ;

   s3 = sin(angles_loc[2][i]) ;
   c3 = cos(angles_loc[2][i]) ;

   s2 = sin(angles_loc[1][i]) ;
   c2 = cos(angles_loc[1][i]) ;

   s5 = sin(angles_loc[4][i]) ;
   c5 = cos(angles_loc[4][i]) ;

   if (valuenear (s5, 0.0, 0.0001) )
     {valid_array[i] = 0 ;
      continue ;
     }

   X71 = X7*c1 - Y7*s1 ;
   Y71 = -Z7 ;
   Z71 = X7*s1 + Y7*c1 ;

   X712 = X71*c2 - Y71*s2 ;
   Y712 = X71*s2 + Y71*c2 ;
   Z712 = Z71 ;

   X7123 = X712*c3 - Y712*s3 ;
   Y7123 = Z712 ;

   angles_loc[3][i] = atan2(-Y7123/s5, X7123/s5) ;
  }

//       SOLVE FOR TH6   ////////////////////////////////////////

double s4, c4 ;
double X43217, Y43217 ;
double X4321, Y4321, Z4321 ;
double X432, Y432, Z432 ;
double X43, Y43, Z43 ;
double X4B, Y4B, Z4B ;

for (i=0 ; i<8 ; ++i)
  {
   if (!valid_array[i])
      continue ;
```

```
   s1 = sin(th1_rad[i]) ;
   c1 = cos(th1_rad[i]) ;

   s3 = sin(angles_loc[2][i]) ;
   c3 = cos(angles_loc[2][i]) ;

   s2 = sin(angles_loc[1][i]) ;
   c2 = cos(angles_loc[1][i]) ;

   s5 = sin(angles_loc[4][i]) ;
   c5 = cos(angles_loc[4][i]) ;

   s4 = sin(angles_loc[3][i]) ;
   c4 = cos(angles_loc[3][i]) ;

   X4B = s4 ;
   Y4B = 0.0 ;
   Z4B = c4 ;

   X43 = X4B*c3 - Y4B*s3 ;
   Y43 = X4B*s3 + Y4B*c3 ;
   Z43 = Z4B ;

   X432 = X43*c2 - Y43*s2 ;
   Y432 = -Z43 ;
   Z432 = X43*s2 + Y43*c2 ;

   X4321 = X432*c1 - Y432*s1 ;
   Y4321 = c71*(X432*s1 + Y432*c1) - s71*Z432 ;
   Z4321 = s71*(X432*s1 + Y432*c1) + c71*Z432 ;

   X43217 = X4321*c7 - Y4321*s7 ;
   Y43217 = -Z4321 ;

   angles_loc[5][i] = atan2(X43217, Y43217) ;

  }

//       WRAPUP          ////////////////////////////////////////

// Count the number of valid solutions.
*pvalid = 0 ;
valid_loc = 0 ;
for (i=0 ; i<8 ; ++i)
  {
   if (valid_array[i])
      valid_loc++ ;
  }
*pvalid = valid_loc ;

// If there are no solutions, or if there are 8 solutions, then exit.
if (*pvalid == 0)
   return ;

if (*pvalid == 8)
  {for (i=0 ; i<8 ; ++i)
```

```
         for (j=0 ; j<6 ; ++j)
            angles_rad[j][i] = angles_loc[j][i] ;
       return ;
      }


   // Shift the valid solutions to the front of the angles_rad array.
   j = 0 ;
   for (i=0 ; i<8 ; ++i)
     {if (valid_array[i])
        {
         for (k=0 ; k<6 ; ++k)
           {angles_rad[k][j] = angles_loc[k][i] ;
           }
         j++ ;
        }
     }

}
```

APPENDIX D
MATLAB CODES FOR KINESTATIC CONTROL

The matlab codes written for implementing kinestatic control are detailed in this chapter

D.1 Main kinestatic correction program

ForceCorrection.m

```
function [ W_eq_corrected  ] = ForceCorrection_B( P_tool_F,S6_F,a67_F, F_req,
f_tol )
% This function will do the following steps:
%  Step 1: Take in the info for taking the robot to a particular
% position and pose - these are P_tool_F , P_tool_6 , S6_F , a67 and
% generate command pose. (puma_rev_analysis.m and galil_control.m)
%  Step 2: Measure W_eq by acquiring the leg lengths (leg_to_T10.m and
% eq_wrench.m)
%  Step 3: Take in the required force and direction and compare it with
% those generated. Find the excess/under force (magnitude).
%  Step 4: Align the robot's S6F with the direction of W_eq
% keeping the P_tool_F the same as in previous step. In this case the
% P_tool_6 is that when the plat is free of any load (free state
% basically). This step will eliminate the tool torque leaving behind only
% pure force behind. (puma_rev_analysis.m , galil_control.m and intial
% (fixed) P_tool_F)
%  Step 5: Move the robo along the new S6F (W_eq's direction)
% forward or reverse by comparing the force generated now to the user
% required force. (puma_rev_analysis.m , eq_wrench.m , leg_to_T10.m)

%%  Step 1: Take in the info for taking the robot to a particular
% position and pose - these are P_tool_F , P_tool_6 , S6_F , a67 and
% generate command pose.
%Initial location of tool point when the tool is not constrained.
P_tool_6=[0.0;0.0;235.0];

%puma_rev_analysis.m
[ command_pose ] = puma_rev_analysis( P_tool_6,P_tool_F,S6_F,a67_F );

%galil_control.m
[ current_pose ] = galil_control( command_pose );

 %% Step 2: Measure W_eq by acquiring the leg lengths
%serialdaq_plat.m
[ leg_e ] = serialdaq_plat( );

%leg_to_T10.m
[ T10, legf, legl ] = leg_to_T10(
leg_e(1),leg_e(2),leg_e(3),leg_e(4),leg_e(5),leg_e(6) );

%puma_fwd_analysis.m
[~, ~, ~, T6_F] = puma_fwd_analysis( P_tool_6 );

%eq_wrench.m
[ W_eq ] = eq_wrench( legl, legf, T10, T6_F );

%% Step 3: Take in the required force and direction and compare it with
% those generated. Find the excess/under force (magnitude).
%F_req is the Force vector required at tool, input by user. f_tol is the
%tolerence in force permissible, this again is a user input value
```

```
F_gen = [W_eq(1);W_eq(2);W_eq(3)];
F_gen_mag = norm(F_gen);

f_mag_err= norm(F_req) - F_gen_mag;

if ( f_mag_err > f_tol || f_mag_err < -f_tol )
   f_mag_err;
else
   f_mag_err=0;
end


%% Step 4: Align the robot's S6F with the direction of W_eq
% keeping the P_tool_F the same as in previous step. In this case the
% P_tool_6 is that when the plat is free of any load (free state
% basically). This step will eliminate the tool torque leaving behind only
% pure force behind. (

%serialdaq_plat.m
[ leg_e ] = serialdaq_plat( );

%leg_to_T10.m
[ T10, legf, legl ] = leg_to_T10(
leg_e(1),leg_e(2),leg_e(3),leg_e(4),leg_e(5),leg_e(6) );

%puma_fwd_analysis.m
[~, ~, ~, T6_F] = puma_fwd_analysis( P_tool_6 );

%eq_wrench.m
[ W_eq ] = eq_wrench( legl, legf, T10, T6_F );

 %New S6_F is direction of W_eq
S6_Fv=[W_eq(1),W_eq(2),W_eq(3)];
S6_F= S6_Fv/norm(S6_Fv);

%P_tool_F and a67_F is same as in step 1

%P_tool_6 is the default value
P_tool_6=[0.0;0.0;235.0];

%puma_rev_analysis.m
[ command_pose ] = puma_rev_analysis( P_tool_6,P_tool_F,S6_F,a67_F );

%galil_control.m
[ current_pose ] = galil_control( command_pose );


%% Step 5 (Method B): Move the robo along the new S6F (W_eq's direction)
% forward or reverse by comparing the force generated now to the user
% required force. (puma_rev_analysis.m , eq_wrench.m , leg_to_T10.m)

%serialdaq_plat.m
[ leg_e ] = serialdaq_plat( );
```

```
%leg_to_T10.m
[ T10, legf, legl ] = leg_to_T10(
leg_e(1),leg_e(2),leg_e(3),leg_e(4),leg_e(5),leg_e(6) );

%puma_fwd_analysis.m
[~, ~, ~, T6_F] = puma_fwd_analysis( P_tool_6 );

%eq_wrench.m
[ W_eq ] = eq_wrench( legl, legf, T10, T6_F );

 %F_req is the Force vector required at tool, input by user. f_tol is the
%tolerance in force permissible, this again is a user input value

F_gen = [W_eq(1);W_eq(2);W_eq(3)];
F_gen_mag = norm(F_gen);

f_mag_err= norm(F_req) - F_gen_mag;

if ( f_mag_err > f_tol || f_mag_err < -f_tol )
   f_mag_err;
else
   f_mag_err=0;
end

while ( f_mag_err ) %repeat process till force comes back within tolerance
%limits

%P_tool_6 will will again be the default value in this step
P_tool_6=[0.0;0.0;235.0];

%P_tool_F will be the corrected point (P_tool_F from previous step +
%f_mag_err)
[S6_F, a67_F, P_tool_F]=puma_fwd_analysis( P_tool_6 );

P_tool_F = P_tool_F + S6_F*1*(f_mag_err/abs(f_mag_err)); %This will move the
%robot in a direction opposite to the current (ie -S6_F) by 1mm

%puma_rev_analysis.m
[ command_pose ] = puma_rev_analysis( P_tool_6,P_tool_F,S6_F,a67_F );

%galil_control.m
[ current_pose ] = galil_control( command_pose );

%serialdaq_plat.m
[ leg_e ] = serialdaq_plat( );

%leg_to_T10.m
[ T10, legf, legl ] = leg_to_T10(
leg_e(1),leg_e(2),leg_e(3),leg_e(4),leg_e(5),leg_e(6) );

%puma_fwd_analysis.m
[~, ~, ~, T6_F] = puma_fwd_analysis( P_tool_6 );
```

```
%eq_wrench.m
[ W_eq ] = eq_wrench( legl, legf, T10, T6_F );
 %F_req is the Force vector required at tool, input by user. f_tol is the
%tolerence in force permissible, this again is a user input value

F_gen = [W_eq(1);W_eq(2);W_eq(3)];
F_gen_mag = norm(F_gen);


f_mag_err= norm(F_req) - F_gen_mag;

if ( f_mag_err > f_tol || f_mag_err < -f_tol )
   f_mag_err;
else
   f_mag_err=0;
end

end % end while loop


end
```

D.2 Puma reverse analysis in matlab

puma_rev_analysis.m

```
function [ command_pose ] = puma_rev_analysis( P_tool_6,P_tool_F,S6_F,a67_F )
% function below can be used for debugging
%function [ command_pose , axes_angles, axes_angles_valid, total] =
puma_rev_analysis( P_tool_6,P_tool_F,S6_F,a67_F )
%Function to generate the reqired axes angles to reach goal (in relative
%coordinates)

%% Run reverse analysis of PUMA and generate possible set of axes angles
[a,b,c,d,e,f,g,h,valid]=puma_rev_ana(P_tool_6,P_tool_F,S6_F,a67_F);


%% Compensate the kinematic correction angles
Pos_A_KCorrection=180;
Pos_B_KCorrection=0;
Pos_C_KCorrection=0;
Pos_D_KCorrection=0;


    a(1)=a(1) +Pos_A_KCorrection;
    b(1)=b(1) +Pos_A_KCorrection;
    c(1)=c(1) +Pos_A_KCorrection;
    d(1)=d(1) +Pos_A_KCorrection;
    e(1)=e(1) +Pos_A_KCorrection;
    f(1)=f(1) +Pos_A_KCorrection;
    g(1)=g(1) +Pos_A_KCorrection;
    h(1)=h(1) +Pos_A_KCorrection;

    a(2)=a(2) +Pos_B_KCorrection;
    b(2)=b(2) +Pos_B_KCorrection;
    c(2)=c(2) +Pos_B_KCorrection;
    d(2)=d(2) +Pos_B_KCorrection;
    e(2)=e(2) +Pos_B_KCorrection;
    f(2)=f(2) +Pos_B_KCorrection;
    g(2)=g(2) +Pos_B_KCorrection;
    h(2)=h(2) +Pos_B_KCorrection;


    a(3)=a(3) +Pos_C_KCorrection;
    b(3)=b(3) +Pos_C_KCorrection;
    c(3)=c(3) +Pos_C_KCorrection;
    d(3)=d(3) +Pos_C_KCorrection;
    e(3)=e(3) +Pos_C_KCorrection;
    f(3)=f(3) +Pos_C_KCorrection;
    g(3)=g(3) +Pos_C_KCorrection;
    h(3)=h(3) +Pos_C_KCorrection;

    a(4)=a(4) +Pos_D_KCorrection;
    b(4)=b(4) +Pos_D_KCorrection;
    c(4)=c(4) +Pos_D_KCorrection;
    d(4)=d(4) +Pos_D_KCorrection;
```

```
    e(4)=e(4) +Pos_D_KCorrection;
    f(4)=f(4) +Pos_D_KCorrection;
    g(4)=g(4) +Pos_D_KCorrection;
    h(4)=h(4) +Pos_D_KCorrection;

%%
for i=1:6
%To optimize the total motion, trim >360deg command_pose
%Along +ve angles
    if (a(i) >= 360)
        a(i)= a(i) - 360;
    end
%Along -ve angles
    if (a(i) <= -360)
        a(i)= a(i) + 360;
    end

   axes_angles(1,i)=a(i);
end



for i=1:6
    if (b(i) >= 360)
        b(i)= b(i) - 360;
    end

    if (b(i) <= -360)
        b(i)= b(i) + 360;
    end

   axes_angles(2,i)=b(i);
end



for i=1:6
    if (c(i) >= 360)
        c(i)= c(i) - 360;
    end

    if (c(i) <= -360)
        c(i)= c(i) + 360;
    end

   axes_angles(3,i)=c(i);
end



for i=1:6
    if (d(i) >= 360)
        d(i)= d(i) - 360;
    end

    if (d(i) <= -360)
        d(i)= d(i) + 360;
```

```
    end

   axes_angles(4,i)=d(i);
end


for i=1:6
    if (e(i) >= 360)
       e(i)= e(i) - 360;
    end

    if (e(i) <= -360)
        e(i)= e(i) + 360;
    end

   axes_angles(5,i)=e(i);
end


for i=1:6
    if (f(i) >= 360)
        f(i)= f(i) - 360;
    end

    if (f(i) <= -360)
        f(i)= f(i) + 360;
    end

   axes_angles(6,i)=f(i);
end


for i=1:6
    if (g(i) >= 360)
        g(i)= g(i) - 360;
    end

    if (g(i) <= -360)
        g(i)= g(i) + 360;
    end

   axes_angles(7,i)=g(i);
end


for i=1:6
    if (h(i) >= 360)
        h(i)= h(i) - 360;
    end

    if (h(i) <= -360)
        h(i)= h(i) + 360;
    end
```

```
   axes_angles(8,i)=h(i);
end


%%
for i=1:6
%To optimize the total motion, trim >180deg command_pose
%Along +ve angles
    if (a(i) >= 180)
        a(i)= a(i) - 180;
    end
%Along -ve angles
    if (a(i) <= -180)
        a(i)= a(i) + 180;
    end

   axes_angles(1,i)=a(i);
end


for i=1:6
    if (b(i) >= 180)
        b(i)= b(i) - 180;
    end

    if (b(i) <= -180)
        b(i)= b(i) + 180;
    end

   axes_angles(2,i)=b(i);
end


for i=1:6
    if (c(i) >= 180)
        c(i)= c(i) - 180;
    end

    if (c(i) <= -180)
        c(i)= c(i) + 180;
    end

   axes_angles(3,i)=c(i);
end


for i=1:6
    if (d(i) >= 180)
        d(i)= d(i) - 180;
    end

    if (d(i) <= -180)
        d(i)= d(i) + 180;
```

```
    end

   axes_angles(4,i)=d(i);
end


for i=1:6
    if (e(i) >= 180)
       e(i)= e(i) - 180;
    end

    if (e(i) <= -180)
        e(i)= e(i) + 180;
    end

   axes_angles(5,i)=e(i);
end


for i=1:6
    if (f(i) >= 180)
        f(i)= f(i) - 180;
    end

    if (f(i) <= -180)
        f(i)= f(i) + 180;
    end

   axes_angles(6,i)=f(i);
end


for i=1:6
    if (g(i) >= 180)
        g(i)= g(i) - 180;
    end

    if (g(i) <= -180)
        g(i)= g(i) + 180;
    end

   axes_angles(7,i)=g(i);
end


for i=1:6
    if (h(i) >= 180)
        h(i)= h(i) - 180;
    end

    if (h(i) <= -180)
        h(i)= h(i) + 180;
    end
```

```
   axes_angles(8,i)=h(i);
end

%% Find valid solution by set axes limits
% Limits of axes as per manufacturer
A(1)=160; A_(1)=-160;
A(2)=110; A_(2)=-110;
A(3)=135; A_(3)=-135;
A(4)=266; A_(4)=-266;
A(5)=100; A_(5)=-100;
A(6)=266; A_(6)=-266;

k=0;
if (valid)
for i=1:valid
    number_of_angles_ok=0;
    for j=1:6
      if (axes_angles(i,j) < A(j) && axes_angles(i,j) > A_(j))
      number_of_angles_ok = number_of_angles_ok+1;
      end
    end

    if (number_of_angles_ok == 6)
        k=k+1;
    end

    if (k) %to check if the number of angles ok greater than 0 exist
    for j=1:6
      if (number_of_angles_ok == 6)
      axes_angles_valid(k,j)=axes_angles(i,j);
      end
    end
    end

end
end

%
%%
%Query Galil controller for current robot position

%The following 2 and delete(g) lines will not be needed when the PUMA is
commanded from a
%central m file with all m files integrated and g object already created
g = actxserver('galil');%set the variable g to the GalilTools COM wrapper
g.address = '192.168.0.84';%Connect to Galil controller at IP 192.168.0.84

%Encoder scales to convert encoder counts to degree(radian?)
ENC_DEG_1=0.00125;   ENC_DEG_2=0.00086875;   ENC_DEG_3=0.001115625;
ENC_DEG_4=0.00626;   ENC_DEG_5=0.00626;   ENC_DEG_6=0.01337;


%Return the current position to matlab
```

```
PosE_A=str2num(g.command('MG _TPA')); PosE_B=str2num(g.command('MG _TPB'));
PosE_C=-str2num(g.command('MG _TPC'));
PosE_D=str2num(g.command('MG _TPD')); PosE_E=str2num(g.command('MG _TPE'));
PosE_F=str2num(g.command('MG _TPF'));

Pos_A=PosE_A*ENC_DEG_1; Pos_B=PosE_B*ENC_DEG_2; Pos_C=PosE_C*ENC_DEG_3;
Pos_D=PosE_D*ENC_DEG_4; Pos_E=PosE_E*ENC_DEG_5; Pos_F=PosE_F*ENC_DEG_6;
current_pose=[Pos_A,Pos_B,Pos_C,Pos_D,Pos_E,Pos_F];
delete(g);%delete g object and close connection with controller

%Find the set of angles which yield the lowest total axes movement
if (k)
for m=1:k
    total(m)=0;

    for n=1:6
        total(m)=total(m)+abs(axes_angles_valid(m,n));
    end
end
end

%I is the index of the lowest
if(k)
[~,I]=min(total);
end

%Set the final pose (in absolute coordinates)that the robot needs to go
if(k)
for i=1:6
    final_pose(i)=axes_angles_valid(I,i);
end
end

%We will be using absolute coordinates for the application
    if(k)
    command_pose=final_pose;
    end

    if(k == 0)%if number of angles ok is 0 then leave manipulator where it is
    command_pose = current_pose;
    end
end
```

D.3 Puma forward analysis

puma_fwd_analysis.m

```
function [S6_F, a67_F, P_tool_F, T6_F]=puma_fwd_analysis( P_tool_6 )

% This function does the forward analysis of a PUMA robot and returns S6_F,
% a67_F, P_tool_F. It also computes T6F if there is a need for it. The
% inputs to this function are the six axis angles: phi1, th2, th3, th4,
% th5, th6 in degrees. It also needs the coordinates of P_tool_6.

%Query Galil controller for current robot position

g = actxserver('galil');%set the variable g to the GalilTools COM wrapper
g.address = '192.168.0.84';%Connect to Galil controller at IP 192.168.0.84

%Encoder scales to convert encoder counts to degree
ENC_DEG_1=0.00125;   ENC_DEG_2=0.00086875;   ENC_DEG_3=0.001115625;
ENC_DEG_4=0.00626;   ENC_DEG_5=0.00626;   ENC_DEG_6=0.01337;


%% Read galil and return the current position to matlab
PosE_A=str2num(g.command('MG _TPA')); PosE_B=str2num(g.command('MG _TPB'));
PosE_C=-str2num(g.command('MG _TPC'));
PosE_D=str2num(g.command('MG _TPD')); PosE_E=str2num(g.command('MG _TPE'));
PosE_F=str2num(g.command('MG _TPF'));

Pos_A=PosE_A*ENC_DEG_1 ; Pos_B=PosE_B*ENC_DEG_2 ; Pos_C=PosE_C*ENC_DEG_3 ;
Pos_D=PosE_D*ENC_DEG_4 ; Pos_E=PosE_E*ENC_DEG_5 ; Pos_F=PosE_F*ENC_DEG_6 ;

delete(g);%delete g object and close connection with controller

%Kinematic correction angles
Pos_A_KCorrection=180;
Pos_B_KCorrection=0;
Pos_C_KCorrection=0;
Pos_D_KCorrection=0;

%% The axes angles have been modified for kinematic correction. (This is
% needed only for forward analysis, reverse analysis DOES NOT NEED IT.
% These angles were found by actual experiments on the robot.
%  The correction values are:
phi1=(Pos_A +Pos_A_KCorrection )*pi/180; th2=(Pos_B +Pos_B_KCorrection
)*pi/180; th3=(Pos_C +Pos_C_KCorrection)*pi/180;
th4=(Pos_D +Pos_D_KCorrection)*pi/180;  th5=(Pos_E )*pi/180; th6=(Pos_F
)*pi/180; %degree to radians, and K2 correction
%
%% CONSTANT MECHANISM PARAMETERS (obtained from PUMA reference manual, sosa
% thesis)
%

a12=0;  % in mm
a23=-650; %
a34=0; % 0
```

```
a45=0;
a56=0;


al12=90*pi/180; % in degrees
al23=0*pi/180;
al34=270*pi/180;
al45=90*pi/180;
al56=90*pi/180;


S2=190; % in mm
S3=0;
S4=-600;
S5=0;
S6=125;


%%
%Shorthand Notation

c12=cos(al12);
c23=cos(al23);
c34=cos(al34);
c45=cos(al45);
c56=cos(al56);

s12=sin(al12);
s23=sin(al23);
s34=sin(al34);
s45=sin(al45);
s56=sin(al56);

c2=cos(th2);
c3=cos(th3);
c4=cos(th4);
c5=cos(th5);
c6=cos(th6);

s2=sin(th2);
s3=sin(th3);
s4=sin(th4);
s5=sin(th5);
s6=sin(th6);


%Transformation Matrices
T1F=[cos(phi1) -sin(phi1) 0 0; sin(phi1) cos(phi1) 0 0; 0 0 1 0; 0 0 0 1];
T21=[c2 -s2 0 a12; s2*c12 c2*c12 -s12 -s12*S2; s2*s12 c2*s12 c12 c12*S2; 0 0
0 1];
T32=[c3 -s3 0 a23; s3*c23 c3*c23 -s23 -s23*S3; s3*s23 c3*s23 c23 c23*S3; 0 0
0 1];
T43=[c4 -s4 0 a34; s4*c34 c4*c34 -s34 -s34*S4; s4*s34 c4*s34 c34 c34*S4; 0 0
0 1];
```

```
T54=[c5 -s5 0 a45; s5*c45 c5*c45 -s45 -s45*S5; s5*s45 c5*s45 c45 c45*S5; 0 0
0 1];
T65=[c6 -s6 0 a56; s6*c56 c6*c56 -s56 -s56*S6; s6*s56 c6*s56 c56 c56*S6; 0 0
0 1];
T6F=T1F*T21*T32*T43*T54*T65;
T6_F=T6F;

S6_F=[T6F(1,3);T6F(2,3);T6F(3,3)];
a67_F=[T6F(1,1);T6F(2,1);T6F(3,1)];

P_tool_6=[P_tool_6;1];
P_tool_F=T6F*P_tool_6;

%% Eliminate 4th row
P_tool_F(4,:)=[];

end
```

D.4 Power on and initialize Puma

startup_initialize.m

```
%%Matlab program to execute the Power on cycle and Initialization of
%%the PUMA Robot
g = actxserver('galil');%set the variable g to the GalilTools COM wrapper
g.address = '192.168.0.84';%Connect to Galil controller at IP 192.168.0.84


%%Power on Cycle
g.command('MO');%Motor off command
g.command('OP0');%Set digital outputs to 0
g.command('AF 0,0,0,0,0,0');%Set all motors to follow digital feedback

pause on;%Turn on puse function in matlab

%Sequence to turn ON VAL Controller Power
IN_3=str2double(g.command('MG @IN[3]'));%Query Input 3 and store in IN_3
while (IN_3)%Loop till IN_3 is greater than 0 with a pause of 5 sec
    disp('No VAL Power - Switch VAL Controller ON!!');
    pause (2);
    IN_3=str2double(g.command('MG @IN[3]'));
end
disp('VAL Controller Power is ON');

g.command('SH');%Start accepting commands and enable servo control
g.command('OP1');%Set digital outputs to 1

%Sequence to turn ON Arm power
IN_1=str2double(g.command('MG @IN[1]'));%Query Input 1 and store in IN_1
while (IN_1)%Loop till IN_1 is greater than 0 with a pause of 5 sec
    disp('Switch ON Arm Power');
    pause (2);
    IN_1=str2double(g.command('MG @IN[1]'));
end
disp('Arm is now powered ON');


%%Intialization Cycle
g.command('DP 0,0,0,0,0,0');%Define current position 0,0,0,0,0,0
g.command('AF 0,0,0,0,0,0')%Set all motors to digital feedback

%Variables for calculations
READYPOT_1=2.500;   READYPOT_2=2.525;   READYPOT_3=2.566;
READYPOT_4=2.538;   READYPOT_5=2.512;   READYPOT_6=2.512;
POT_ENCSCALE_1=77220;   POT_ENCSCALE_2=75188;   POT_ENCSCALE_3=75301;
POT_ENCSCALE_4=23866;   POT_ENCSCALE_5=24079;   POT_ENCSCALE_6=24036;
TEMP_1=0;   TEMP_2=0;   TEMP_3=0;   TEMP_4=0;   TEMP_5=0;   TEMP_6=0;

%Get average value of analog inputs for 100 values
for n=1:100
    TEMP_1=TEMP_1+str2double(g.command('MG @AN[1]'));
    TEMP_2=TEMP_2+str2double(g.command('MG @AN[2]'));
```

```
    TEMP_3=TEMP_3+str2double(g.command('MG @AN[3]'));
    TEMP_4=TEMP_4+str2double(g.command('MG @AN[4]'));
    TEMP_5=TEMP_5+str2double(g.command('MG @AN[5]'));
    TEMP_6=TEMP_6+str2double(g.command('MG @AN[6]'));
end
POTVAL_1=TEMP_1/100; POTVAL_2=TEMP_2/100; POTVAL_3=TEMP_3/100;
POTVAL_4=TEMP_4/100; POTVAL_5=TEMP_5/100; POTVAL_6=TEMP_6/100;

disp(POTVAL_1); disp(POTVAL_2); disp(POTVAL_3);
disp(POTVAL_4); disp(POTVAL_5); disp(POTVAL_6);

%Coarse error distances, rounded the converted to string
ERR_1=num2str(round((READYPOT_1 - POTVAL_1) * POT_ENCSCALE_1));
ERR_2=num2str(round((READYPOT_2 - POTVAL_2) * POT_ENCSCALE_2));
ERR_3=num2str(round((READYPOT_3 - POTVAL_3) * POT_ENCSCALE_3));
ERR_4=num2str(round((READYPOT_4 - POTVAL_4) * POT_ENCSCALE_4));
ERR_5=num2str(round((READYPOT_5 - POTVAL_5) * POT_ENCSCALE_5));
ERR_6=num2str(round((READYPOT_6 - POTVAL_6) * POT_ENCSCALE_6));

%Coarse error correction
g.command('JG 5000,5000,5000,1000,1000,1000');%Set in Jog mode with slew
speed as given
g.command('SH');%Servo here
g.command('AC 2000,2000,2000,400,400,400');%Set accelaration of motors
g.command('SP 5000,5000,5000,1000,1000,1000');%Set speed of motors
g.command(strcat('PR',ERR_1,',',ERR_2,',',ERR_3,',',ERR_4,',',ERR_5,',',ERR_6
));%Send the variable values to Galil
disp(strcat('PR',ERR_1,',',ERR_2,',',ERR_3,',',ERR_4,',',ERR_5,',',ERR_6));
g.command('BG');%Begin motion

pause (0.01)
BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
disp(BState);
while ( BState>0 )
    BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
    disp(BState);
    pause (.01)
end

g.command('JG 1000,1000,1000,200,200,200');%Set in Jog mode with slew speed
as given
g.command('SP 1000,1000,1000,200,200,200');%Set speed of motors
g.command('FI')%Find index and stop motion and zero at the point where
transition is found
g.command('BG');%Begin motion

pause (0.01)
BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
disp(BState);
while ( BState>0 )
```

```
    BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
    disp(BState);
    pause (.01)
end

g.command('PA -7700, -9650, -11500, -3400, -3000, -1700')%Move these
absoulute distances
g.command('BG');%Begin motion

pause (0.01)
BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
disp(BState);
while ( BState>0 )
    BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
    disp(BState);
    pause (.01)
end

%Fine positioning to required coordinates and defining final pose as zero
g.command('JG 500,500,500,100,100,100');%Set in Jog mode with slew speed as
given
g.command('FI')%Find index and stop motion and zero at the point where
transition is found
g.command('BG');%Begin motion

%g.command('MC ABCDEF');%Hold further commands till the motion is complete

pause (0.01)
BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
disp(BState);
while ( BState>0 )
    BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
    disp(BState);
    pause (.01)
end

disp('Found Index');

g.command('PA -4450, 0, -2000, -810, -975, 2500')%Move these absoulute
distances
g.command('BG');%Begin motion

pause (0.01)
```

```
BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
disp(BState);
while ( BState>0 )
    BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
    disp(BState);
    pause (.01)
end

g.command('DP 0,-103597,80672,0,0,0');%Define current position 0,-90,90,0,0,0
disp('PUMA is now Home');

pause off;

delete(g);%delete g object and close connection with controller
```

D.5 Send control commands to Galil™ controller

galil_control.m

```
function [ current_pose ] = galil_control( command_pose )
%Function to pass motion arguments in Absolute (PA) mode to Galil controller
from matlab environment
%IMP: TO BE ABLE TO RUN THIS FUNCTION, THE PUMA SHOULD BE STARTED UP AND
%INITIALIZED, RUN startup_initialize.m TO DO THIS

Deg_A=command_pose(1); Deg_B=command_pose(2);
Deg_C=-command_pose(3); Deg_D=command_pose(4);
Deg_E=command_pose(5); Deg_F=command_pose(6);

% Connect to the controller
g = actxserver('galil');%set the variable g to the GalilTools COM wrapper
g.address = '192.168.0.84';%Connect to Galil controller at IP 192.168.0.84
%%
% Axis 4-5 Anomalous motion correction
[ Actual_Deg_E, Actual_PosE_E, Apparant_PosE_E  ] = Axis_4_5_Correction(
command_pose );
%%

g.command('MO');%Motor off command
g.command('SH');%Start accepting commands and enable servo control

%Encoder scales to convert encoder counts to degree
ENC_DEG_1=0.00125;   ENC_DEG_2=0.00086875;   ENC_DEG_3=0.001115625;
ENC_DEG_4=0.00626;   ENC_DEG_5=0.00626;   ENC_DEG_6=0.01337;

En_A=round(Deg_A/ENC_DEG_1); En_B=round(Deg_B/ENC_DEG_2);
En_C=round(Deg_C/ENC_DEG_3);
En_D=round(Deg_D/ENC_DEG_4); En_E=round(Deg_E/ENC_DEG_5);
En_F=round(Deg_F/ENC_DEG_6);

PosE_A=str2num(g.command('MG _TPA')); PosE_B=str2num(g.command('MG _TPB'));
PosE_C=str2num(g.command('MG _TPC'));
PosE_D=str2num(g.command('MG _TPD')); PosE_E=str2num(g.command('MG _TPE'));
PosE_F=str2num(g.command('MG _TPF'));

%%
En_E=round(Actual_PosE_E); %Corrected 5th Axis motion in encoder counts
%%

%Max speeds on the axes and variable speed calculation
Sp_max_A=8000;  Sp_max_B=11511;  Sp_max_C=8964;
Sp_max_D=3194;  Sp_max_E=0.46*3194;  Sp_max_F=1496;

if (abs(PosE_A-En_A)/4 < Sp_max_A)
    Sp_A=ceil(abs(PosE_A-En_A)/4);
    if (Sp_A < 10)
        Sp_A = 10;
    end
else
```

```
    Sp_A=Sp_max_A;
end

if (abs(PosE_B-En_B)/4 < Sp_max_B)
    Sp_B=ceil(abs(PosE_B-En_B)/4);
    if (Sp_B < 10)
        Sp_B = 10;
    end
else
    Sp_B=Sp_max_B;
end

if (abs(PosE_C-En_C)/4 < Sp_max_C)
    Sp_C=ceil(abs(PosE_C-En_C)/4);
    if (Sp_C < 10)
        Sp_C = 10;
    end
else
    Sp_C=Sp_max_C;
end

if (abs(PosE_D-En_D)/4 < Sp_max_D)
    Sp_D=ceil(abs(PosE_D-En_D)/4);
    if (Sp_D < 10)
        Sp_D = Sp_max_D;
    end
else
    Sp_D=Sp_max_D;
end

if (abs(PosE_E-En_E)/4 < Sp_max_E)
    Sp_E=ceil(abs(PosE_E-En_E)/4);
    if (Sp_E < 10)
        Sp_E = Sp_max_D;
    end
else
    Sp_E=Sp_max_E;
end

if (abs(PosE_F-En_F)/4 < Sp_max_F)
    Sp_F=ceil(abs(PosE_F-En_F)/4);
    if (Sp_F < 10)
        Sp_F = 10;
    end
else
    Sp_F=Sp_max_F;
end

Sp_A = num2str(Sp_A);
Sp_B = num2str(Sp_B);
Sp_C = num2str(Sp_C);
Sp_D = num2str(Sp_D);
Sp_E = num2str(Sp_E);
Sp_F = num2str(Sp_F);
```

```
%% Check for very small motions ( < 5 encoder counts)
if (abs(PosE_A-En_A) < 5)
    En_A=En_A+5;
end
if (abs(PosE_B-En_B) < 5)
    En_B=En_B+5;
end
if (abs(PosE_C-En_C) < 5)
    En_C=En_C+5;
end
if (abs(PosE_D-En_D) < 5)
    En_D=En_D+5;
end
if (abs(PosE_E-En_E) < 5)
    En_E=En_E+5;
end
if (abs(PosE_F-En_F) < 5)
    En_F=En_F+5;
end
%%
En_A = num2str(En_A);
En_B = num2str(En_B);
En_C = num2str(En_C);
En_D = num2str(En_D);
En_E = num2str(En_E);
En_F = num2str(En_F);


%Motion Commands
g.command('AC 8000,11511,8964,3194,3194,1496');%Set Accelaration of motors
g.command('DC 8000,11511,8964,3194,3194,1496');%Set Decelaration of motors
g.command(strcat('SP',Sp_A,',',Sp_B,',',Sp_C,',',Sp_D,',',Sp_E,',',Sp_F));%Se
t speed of motors
g.command(strcat('PA',En_A,',',En_B,',',En_C,',',En_D,',',En_E,',',En_F));%Mo
ve robot to relative coordinates
g.command('BG');%Begin motion

pause on;

pause (0.01)
BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
%disp(BState);
while ( BState>0 )
    BState = str2double(g.command('MG _BGA'))+str2double(g.command('MG
_BGB'))+str2double(g.command('MG _BGC'))+str2double(g.command('MG
_BGD'))+str2double(g.command('MG _BGE'))+str2double(g.command('MG _BGF'));
    %disp(BState);
    pause (.01)
end

pause off;

%%
Apparant_PosE_E = num2str(Apparant_PosE_E);
```

```
g.command(strcat('DP',',,,,',Apparant_PosE_E,','));%Corrected 5th Axis
encoder value
%%

disp('Command completed');

%Return the current position to matlab
PosE_A=str2num(g.command('MG _TPA')); PosE_B=str2num(g.command('MG _TPB'));
PosE_C=-str2num(g.command('MG _TPC'));
PosE_D=str2num(g.command('MG _TPD')); PosE_E=str2num(g.command('MG _TPE'));
PosE_F=str2num(g.command('MG _TPF'));
Pos_A=PosE_A*ENC_DEG_1; Pos_B=PosE_B*ENC_DEG_2; Pos_C=PosE_C*ENC_DEG_3;
Pos_D=PosE_D*ENC_DEG_4; Pos_E=PosE_E*ENC_DEG_5; Pos_F=PosE_F*ENC_DEG_6;

current_pose=[Pos_A,Pos_B,Pos_C,Pos_D,Pos_E,Pos_F];

delete(g);%delete g object and close connection with controller
end
```

D.6 Acquire platform leg length data from serial port

serialdaq_plat.m

```
function [ leg_e ] = serialdaq_plat( )
        xbee = serial('COM2');
        set(xbee,'BaudRate',115200,'Parity','none');
        %might have to replace the terminator 0D0A by
        fopen(xbee);

        %Take 10 readings and use the last accurate reading to account for
        %missed data packets
 for i=1:10

        data_raw=fgetl(xbee);%reads one line of text from the device
%connected to the serial port object, xbee, and returns the data to data_raw
        data=dec2hex(unicode2native(data_raw));%ASCII (unicode) to decimal ->
%decimal to hex
        [m,n]=size(data);
        if m==23&&n==2
        X=data;

           if
(X(4,1)=='4'&&X(4,2)=='1'&&X(7,1)=='4'&&X(7,2)=='2'&&X(10,1)=='4'&&X(10,2)=='
3'&&X(13,1)=='4'&&X(13,2)=='4'&&X(16,1)=='4'&&X(16,2)=='5'&&X(19,1)=='4'&&X(1
9,2)=='6')
            leg2_h=[X(5,1),X(5,2),X(6,1),X(6,2)];
            leg1_h=[X(8,1),X(8,2),X(9,1),X(9,2)];
            leg6_h=[X(11,1),X(11,2),X(12,1),X(12,2)];
            leg5_h=[X(14,1),X(14,2),X(15,1),X(15,2)];
            leg4_h=[X(17,1),X(17,2),X(18,1),X(18,2)];
            leg3_h=[X(20,1),X(20,2),X(21,1),X(21,2)];
           end

        leg1_e=hex2dec(leg1_h);
        leg2_e=hex2dec(leg2_h);
        leg3_e=hex2dec(leg3_h);
        leg4_e=hex2dec(leg4_h);
        leg5_e=hex2dec(leg5_h);
        leg6_e=hex2dec(leg6_h);

        end

 end
        leg_e=[leg1_e,leg2_e,leg3_e,leg4_e,leg5_e,leg6_e];
        fclose(xbee);
        delete(xbee);


end
```

D.7 Forward analysis of platform in MATLAB™

leg_to_T10.m

```
function [ T10, legf, legl ] = leg_to_T10(
leg1_e,leg2_e,leg3_e,leg4_e,leg5_e,leg6_e )

%M file to determine the transformation matrix T10 by knowing
%six leg lengths of the stewart platform. T10 is a (6,4,4) matrix

leg1_l= -0.025771* leg1_e + 91.9216;
leg2_l= -0.025455* leg2_e + 79.2948;
leg3_l= -0.027115* leg3_e + 92.7287;
leg4_l= -0.025419* leg4_e + 80.5882;
leg5_l= -0.025657* leg5_e + 93.5823;
leg6_l= -0.026074* leg6_e + 81.2437;

legl=[leg1_l,leg2_l,leg3_l,leg4_l,leg5_l,leg6_l];

[T10_1,T10_2,T10_3,T10_4,T10_5,T10_6]=
plat_leglen_to_T10(leg1_l,leg3_l,leg5_l,leg6_l,leg2_l,leg4_l);

%T10_1 to T10(1,m,n)
    for i=1:16
        if (i<5)
            m=1;
            n=i;
        end
        if (i>4 && i<9)
            m=2;
            n=i-4;
        end
        if (i>8 && i<13)
            m=3;
            n=i-8;
        end
        if (i>12)
            m=4;
            n=i-12;
        end

    T10(1,m,n)=T10_1(i);
    end

    %T10_2 to T10(2,m,n)
    for i=1:16
        if (i<5)
            m=1;
            n=i;
        end
        if (i>4 && i<9)
            m=2;
            n=i-4;
        end
        if (i>8 && i<13)
```

```
        m=3;
        n=i-8;
    end
    if (i>12)
        m=4;
        n=i-12;
    end

T10(2,m,n)=T10_2(i);
end


%T10_3 to T10(3,m,n)
for i=1:16
    if (i<5)
        m=1;
        n=i;
    end
    if (i>4 && i<9)
        m=2;
        n=i-4;
    end
    if (i>8 && i<13)
        m=3;
        n=i-8;
    end
    if (i>12)
        m=4;
        n=i-12;
    end

T10(3,m,n)=T10_3(i);
end


%T10_4 to T10(4,m,n)
for i=1:16
    if (i<5)
        m=1;
        n=i;
    end
    if (i>4 && i<9)
        m=2;
        n=i-4;
    end
    if (i>8 && i<13)
        m=3;
        n=i-8;
    end
    if (i>12)
        m=4;
        n=i-12;
    end

T10(4,m,n)=T10_4(i);
```

```
end

%T10_5 to T10(5,m,n)
for i=1:16
    if (i<5)
        m=1;
        n=i;
    end
    if (i>4 && i<9)
        m=2;
        n=i-4;
    end
    if (i>8 && i<13)
        m=3;
        n=i-8;
    end
    if (i>12)
        m=4;
        n=i-12;
    end

T10(5,m,n)=T10_5(i);
end

%T10_6 to T10(6,m,n)
for i=1:16
    if (i<5)
        m=1;
        n=i;
    end
    if (i>4 && i<9)
        m=2;
        n=i-4;
    end
    if (i>8 && i<13)
        m=3;
        n=i-8;
    end
    if (i>12)
        m=4;
        n=i-12;
    end

T10(6,m,n)=T10_6(i);
end

%Force in each of the legs
leg1_f= 0.029796 * leg1_e - 14.25;
leg2_f= 0.028449 * leg2_e - 13.407;
leg3_f= 0.027919 * leg3_e - 11.325;
leg4_f= 0.029123 * leg4_e - 14.103;
leg5_f= 0.027136 * leg5_e - 13.12;
leg6_f= 0.030975 * leg6_e - 14.81;
  legf=[leg1_f,leg2_f,leg3_f,leg4_f,leg5_f,leg6_f];
end
```

D.8 Determine the equivalent wrench acting on platform

eq_wrench.m

```
function [ W_eq ] = eq_wrench( legl, legf, T10_t, T6_F )
%This function uses leg_to_T10.m to calculate the equivalent length.
%Initially the coordinates of the points on the top plat are transformed to
%the bot plat coordinates using the T10 matrix selected experimentally.
%Then the wrench in each of the legs is calculated. The equivalent is thus
%computed by summing all the six individual wrenches.

leg1=legl(1); leg2=legl(2); leg3=legl(3);
leg4=legl(4); leg5=legl(5); leg6=legl(6);

n=3; %Select nth matrix of the 6 generated matrices
for i=1:4
    for j=1:4
        T10_(i,j)=T10_t(n,i,j);
    end
end

%Constant parameters
%Coordinates in the top plat
O1_1=[14.0;0;0];
P1_1=[53.0;12.1244;0];
Q1_1=[23.0;39.8371;0];
R1_1=[0.0;0.0;0];
S1_1=[60.0;0.0;0];
T1_1=[30.0;51.9615;0];
%Coordinates in the bottom plat
O0_0=[0.0;0.0;0];
P0_0=[120.0;0.0;0];
Q0_0=[60.0;103.9230;0];
R0_0=[46.0;79.6743;0];
S0_0=[28.0;0.0;0];
T0_0=[106.0;24.2487;0];

%Coordinate transform
O1_0=T10_*[O1_1;1];
P1_0=T10_*[P1_1;1];
Q1_0=T10_*[Q1_1;1];
R1_0=T10_*[R1_1;1];
S1_0=T10_*[S1_1;1];
T1_0=T10_*[T1_1;1];

%Coordinate transform to fixed coordinate system
O1_F=T6_F*O1_0;
P1_F=T6_F*P1_0;
Q1_F=T6_F*Q1_0;
R1_F=T6_F*R1_0;
S1_F=T6_F*S1_0;
T1_F=T6_F*T1_0;

O0_F=T6_F*[O0_0;1];
P0_F=T6_F*[P0_0;1];
```

```
Q0_F=T6_F*[Q0_0;1];
R0_F=T6_F*[R0_0;1];
S0_F=T6_F*[S0_0;1];
T0_F=T6_F*[T0_0;1];

 %Eliminate 4th row
O1_F(4,:)=[];
P1_F(4,:)=[];
Q1_F(4,:)=[];
R1_F(4,:)=[];
S1_F(4,:)=[];
T1_F(4,:)=[];

O0_F(4,:)=[];
P0_F(4,:)=[];
Q0_F(4,:)=[];
R0_F(4,:)=[];
S0_F(4,:)=[];
T0_F(4,:)=[];


%Find S vector
S_O=(O1_F-O0_F)/norm(O1_F-O0_F);
S_P=(P1_F-P0_F)/norm(P1_F-P0_F);
S_Q=(Q1_F-Q0_F)/norm(Q1_F-Q0_F);
S_R=(R1_F-R0_F)/norm(R1_F-R0_F);
S_S=(S1_F-S0_F)/norm(S1_F-S0_F);
S_T=(T1_F-T0_F)/norm(T1_F-T0_F);

%Find Moment vector S0L
S0L_O=cross(O0_F,S_O);
S0L_P=cross(P0_F,S_P);
S0L_Q=cross(Q0_F,S_Q);
S0L_R=cross(R0_F,S_R);
S0L_S=cross(S0_F,S_S);
S0L_T=cross(T0_F,S_T);

%Wrench in each leg
W1=legf(1)*[S_O;S0L_O];
W2=legf(3)*[S_P;S0L_P];
W3=legf(5)*[S_Q;S0L_Q];
W4=legf(6)*[S_R;S0L_R];
W5=legf(2)*[S_S;S0L_S];
W6=legf(4)*[S_T;S0L_T];

%Equivalent wrench
W_eq=W1+W2+W3+W4+W5+W6;


end
```

D.9 4th and 5th axis linked motion correction

Axis_4_5_Correction.m

```
function [ Actual_Deg_E, Actual_PosE_E, Apparant_PosE_E  ] =
Axis_4_5_Correction( command_pose )
% This function corrects the anomalous motion of axis 5 when axis 4 is
% moved.
% The function will have to generate positions for Axis 5 when Axis 4 is
% moved and reset the encoder as if Axis 5 is not moved at all
% The point to be noted is that, this anomalous motion of 5th axis exists
% only when 4th axis moved, not the vice versa way. Also the actual encoder
% reading does not change even when the 5th axis is moving.
% We also need to reset encoder

%%
%Motion in 5th axis / Motion in 4th axis (in encoder counts)
Compensation_Factor = (-11200/23962);

%Encoder scales to convert encoder counts to degree
ENC_DEG_4=0.00626;   ENC_DEG_5=0.00626;



%%
g = actxserver('galil');%set the variable g to the GalilTools COM wrapper
g.address = '192.168.0.84';%Connect to Galil controller at IP 192.168.0.84

%%

%Return the current position to matlab
PosE_D=str2num(g.command('MG _TPD'));
PosE_E=str2num(g.command('MG _TPE'));
delete(g);

%Commanded position to axis 4 and 5
Apparant_Deg_D=command_pose(4);
Apparant_Deg_E=command_pose(5);

%%
%Corrected position to axis 5
Actual_PosE_E = (Apparant_Deg_E/ENC_DEG_5)-(PosE_D-
Apparant_Deg_D/ENC_DEG_4)*Compensation_Factor;
Actual_Deg_E = Actual_PosE_E*ENC_DEG_5;

%Reset Encoder to the Apparant value after motion (Value of encoder before
%additional motion for correction)
Apparant_PosE_E = round(Apparant_Deg_E/ENC_DEG_5);

end
```

LIST OF REFERENCES

1. Crane, C.D., Rico, J.M. and Duffy, J., 2001, EML6282 Geometry of Robots II Class notes: *Screw Theory and its Application to Spatial Robot Manipulators*, MAE, University of Florida, Gainesville, FL.

2. Crane, C.D. and Duffy, J., 1998, Kinematics Analysis of Robot Manipulators, Cambridge University press, USA.

3. Dwarakanath, T., and Crane, C. 2000, “In-parallel Passive Compliant Coupler for Robot Force Control,” *Proceedings of the ASME Mechanisms Conference*, Baltimore, MD, pp. 214-221.

4. Duffy, J, 1996, *Statics and Kinematics with Applications to Robotics*, Cambridge University press, New York, USA.

5. Griffis, M., and Duffy, J., 1989, “A Forward Displacement Analysis of a Class of Stewart Platforms,” Journal of Robotic Systems, Vol. 6(6), pp. 703-720.

6. Griffis, M., 1991, “Kinestatic control: A Novel Theory for Simultaneously Regulating Force and Displacement”, Ph.D. Dissertation, University of Florida, Gainesville, FL

7. Griffis, M., and Duffy, J., 1991, “Kinestatic Control: A Novel Theory for Simultaneously Regulating Force and Displacement,” Trans. ASME Journal of Mechanical Design, Vol. 113, No. 4, pp. 508-515.

8. Lee, J., 1996, “An Investigation of A Quality Index for The Stability of In-Parallel Planar Platform Devices,” Master’s Thesis, University of Florida, Gainesville, FL.

9. Zhang, B., 2005, "Design and Implementation of a 6 DOF Parallel Manipulator with Passive Force Control," Ph.D. Dissertation, University of Florida, Gainesville, FL.

10. Unimate Industrial Robots, PUMA 700 Series Mark III – VAL II Equipment Manual, Unimation Inc, Danbury, Connecticut, May 1986.

11. Galil MC, DMC-2x00 Manual Rev. 1.3, Galil Motion Control Inc., Rocklin, California, June 2001.

12. Galil MC, DMCWin32 Galil Windows API toolkit Rev 2.2 for PCI/ISA/Ethernet Controllers, Galil Motion Control Inc., Rocklin, California, June 2006.

13. Sosa, O., 2004, “Design and Implementation of a Modular Manipulator Architecture”, M.S. Thesis, University of Florida, Gainesville, FL

14. Nayak, S., 2009, “Development of a 6 DOF Compliant Parallel Mechanism to Sense and Control the Force-Torque and Displacement of a Serial Robot Manipulator”, M.S. Thesis, University of Florida, Gainesville, FL

15. Griffis, M., and Duffy, J., “Method and Apparatus for Controlling Geometrically Simple Parallel Mechanisms with Distinctive Connections,” United States Patent, Patent Number 5,179,525, Jan.12, 1993.

BIOGRAPHICAL SKETCH

Akash Vibhute was born in Maharashtra, India in 1987. He received his Bachelor of Engineering in Mechanical Engineering degree from Government College of Engineering Aurangabad, India in July 2009 and Master of Science in Mechanical and Aerospace Engineering from University of Florida in the spring of 2011. He was always fascinated about cutting edge technology and would try to go hands on whenever possible. His school time projects include Hydro-Pneumatic powered rocket, FM radio transmitter, homemade Bio-Diesel reactor and many more. Having hands on experience of building so many projects, he got attracted to the idea of process automation. During his College days he built his first automation project, an ATMega8 controlled line follower robot, following this he developed many microcontroller applications for local bussinesses. He was first thoroughly introduced to CNC machining centers at the company owned by his father. Here he got the liberty and opportunity to explore the in and outs of the machine. Intrigued by this, he decided to build his own 3 – axis CNC PCB drilling machine and successfully did so in 2008. By this time he was gaining interest in industrial robots, having seen many of them in various trade shows. He got the novel opportunity to be a part of the robot training team at a industry in Aurangabad. Mesmerized by this robot he decided to build a small 4-axis SCARA robot for machine tending application. This was successfully completed in 2009.

Passionate about acquiring knowledge and learing more about robotics, he decided to pursue Master of Science in Mechanical Engineering at University of Florida, USA. Here he met Dr. Carl D. Crane, III who willingly accepted him to work in Center for Intelligent Machines ad Robotics. His first project there was to develop a protocol to control a Puma 762 industrial robot which was retrofitted with a Galil motion controller.

During this he came across the Special 6-6 platform and took the course Geometry of Robots -2 enabling him to start working on implementing the theory of kinestatic control on this robot.