Evaluation of Galil DMC-4080 as a Controller of Stewart Platform

Christer Engblom

Department of Automatic Control

Msc Thesis
ISRN LUTFD2/TFRT--5946--SE
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

Evaluation of the Galil DMC–4080 as a closed-loop controller of a stepper-motor driven Stewart Platform with μm- and μrad- precision and resolution. The project involves a full implementation and optimization of kinematic equations, implementing modes of motion & controller structure, as well the use of MATLAB simulations. The 4080 proves successful in static positioning but have limitations in time-performances with the kinematics as well as smoothness of motion due to error handling.

Contents

1 Introduction

SOLEIL is a French synchrotron radiation facility situated just outside Paris in France. For the better part of a year now, SOLEIL has been in the process of evaluating which controller is to replace one of its most used ones; the facility has since its inauguration in 2006 been using the Galil DMC-2182 for most of its controller needs. The DMC-4080, the newest and best performing unit from Galil, is one of the controllers being considered. The purpose of this thesis is to evaluate how well the Galil DMC-4080 controls a hexapod robot design called *Stewart Platform*. This platform can be used to position mirror systems (such as monochromators) with many degrees of freedom and high precision.

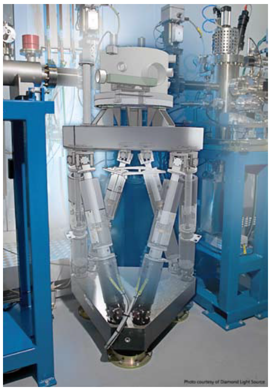

Figure 1.1 Example of monochromator on Stewart Platform

2

Background

2.1 Stewart Platform

The *Stewart Platform* is a six-degrees-of-freedom (DOF) platform that, by utilizing prismatic actuators in various designs, is capable of moving in three linear directions (x, y, z) and three angular directions (θ (roll), ϕ (pitch) and ψ (yaw))independently or in any combination of these[1]. The design that will be used in this project is the so called 3-6 Stewart Platform; this design connects the six actuators to each three points on a base- and positional- plate[1] (see Figure 2.1). This system is of a parallel

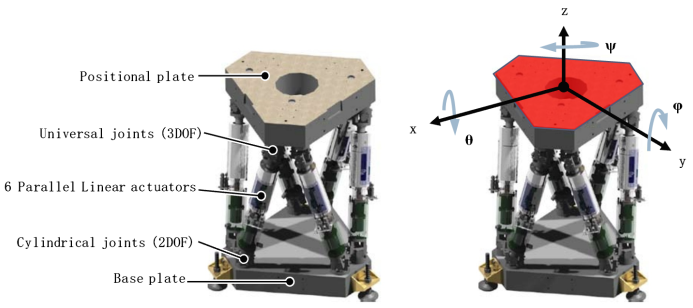

Figure 2.1 Stewart Platform setup to the left. The image to the right illustrate the 6 possible degrees of freedom of the positional plate.

manipulator sort meaning that the movable platform is in direct contact (via joints) to all of its leg actuators. This particular setup comes with certain advantages and disadvantages that will be covered in this section.

Structural Rigidity, System Precision & Construction Size

The rigidity of a system is defined as its ability to resist flex during loads. Parallel systems are known to have good rigidity compared to the number of actuators in the system[2]. This comes from the mechanical setup; the flexibility of one actuator is limited by being braced against all other axes connected to the same joint of the platform[2]. In the example of Stewart Platform, with a heavy load situated in the center of the positional plate, the load will be evenly distributed on *all* axes which results in a minimal amount of flex in the system[2][4]. The system precision is for the same reason very good for parallel systems; positional errors in the joints are in fact roughly averaged by the error in all actuators connected to the same joint[2][4].

The good structural rigidity that comes from parallel systems also makes for comparatively small and lightweight constructions compared to a serial manipulator system with similar rigidity and same amount of DOF[2][4].

Kinematic Equations

To control the hexapod we need to be able to convert virtual axes coordinates (see Figure 2.1) to leg lengths. These kinematic conversions are called the *Reverse Kinematics* and are defined in Equation 2.1, where **L** denotes the non-linear virtual state-to leg-length conversion.

$$[l_1, l_2, l_3, l_4, l_5, l_6]^T = \mathbf{L}(x, y, z, \theta, \phi, \psi) \tag{2.1}$$

The Reverse Kinematics are essential in controlling the robot and one only needs these kinematics when doing so in open loop. To run the robot in a closed-loop feedback is required. Feedback can be acquired by mounting sensors on the positional plate (multiple interferometers for an example), or – if you have a good enough model of the system – use the leg lengths to estimate the states of the virtual axes. These calculations are called the *Forward Kinematics* and are defined in Equation 2.2, where **F** denotes a (in this project) numerical approximative conversion from leg-lengths to virtual states.

$$[\hat{x}, \hat{y}, \hat{z}, \hat{\theta}, \hat{\phi}, \hat{\psi}]^T = \mathbf{F}(l_1, l_2, l_3, l_4, l_5, l_6) \tag{2.2}$$

Note that even though Equations 2.1 and 2.2 seem relatively straightforward and simple at first glance they are not. The Forward Kinematics of Stewart Platform are especially known to be quite complicated and difficult to solve[2]. These equations will be revisited later.

Singularities & Workspace

The mechanical setup that gives Stewart Platform such good rigidity and precision also limits the workspace of the robot[2]. It is of interest to define a workspace in which full controllability is possible due to the lack of singular points within. These singularities can come from mechanical limitations (the leg actuators might collide with each other for certain platform moves) or kinematic limitations[2][3]; for certain platform positions (x_s, y_s, z_s, θ_s, ϕ_s ,ψ_s), where the index s denotes a singular point, the system will loose its controllability due to kinematic limitations. What this means from a mathematical point of view is that the Jacobian matrix defined by Equation 2.3 has either none or multiple solutions – which is true when Equation 2.4 is true[3].

$$\mathbf{J}_L(x,y,z,\theta,\phi,\psi) = \begin{bmatrix} \frac{\delta l_1}{\delta x} & \frac{\delta l_1}{\delta y} & \frac{\delta l_1}{\delta z} & \frac{\delta l_1}{\delta \theta} & \frac{\delta l_1}{\delta \phi} & \frac{\delta l_1}{\delta \psi} \\ \frac{\delta l_2}{\delta x} & \frac{\delta l_2}{\delta y} & \frac{\delta l_2}{\delta z} & \frac{\delta l_2}{\delta \theta} & \frac{\delta l_2}{\delta \phi} & \frac{\delta l_2}{\delta \psi} \\ \frac{\delta l_3}{\delta x} & \frac{\delta l_3}{\delta y} & \frac{\delta l_3}{\delta z} & \frac{\delta l_3}{\delta \theta} & \frac{\delta l_3}{\delta \phi} & \frac{\delta l_3}{\delta \psi} \\ \frac{\delta l_4}{\delta x} & \frac{\delta l_4}{\delta y} & \frac{\delta l_4}{\delta z} & \frac{\delta l_4}{\delta \theta} & \frac{\delta l_4}{\delta \phi} & \frac{\delta l_4}{\delta \psi} \\ \frac{\delta l_5}{\delta x} & \frac{\delta l_5}{\delta y} & \frac{\delta l_5}{\delta z} & \frac{\delta l_5}{\delta \theta} & \frac{\delta l_5}{\delta \phi} & \frac{\delta l_5}{\delta \psi} \\ \frac{\delta l_6}{\delta x} & \frac{\delta l_6}{\delta y} & \frac{\delta l_6}{\delta z} & \frac{\delta l_6}{\delta \theta} & \frac{\delta l_6}{\delta \phi} & \frac{\delta l_6}{\delta \psi} \end{bmatrix} \tag{2.3}$$

$$|\mathbf{J}_L(x_s,y_s,z_s,\theta_s,\phi_s,\psi_s)| = 0, \text{ where } (x_s,y_s,z_s,\theta_s,\phi_s,\psi_s) \text{ is a singular point} \tag{2.4}$$

It is therefore important to properly define the workspace of the system prior to usage.

2.2 Stepper Motors

Steppers are electric motors designed to rotate in discrete steps. They consist of a main rotor with jagged edges, which may be of simple iron or a permanent magnet, that is slightly misaligned by surrounding electromagnets. By putting a charge to one of the electromagnets the rotor will make sure to align itself accordingly. A rotary step is then accomplished by switching the charge to the next coil and hence force the rotor to turn[5]. Figure 2.2 shows how a two-phase stepper would take one full step; current would first go into phase one, followed by two, and then back to one again. Figure 2.3 shows the current over time for the two phases. This method of rotation is called *Wave Drive*; note that there are better methods[5] (ex: Full Step-drive, Microstepping).

The number of full steps the motor does per revolution is determined by its mechanical construction and differs from product to product. In the case of this project, 200 full steps are taken per revolution (see Table 7.1, Appendix 7.1) which corresponds to 1.8°. *Microstepping* will enable a better resolution.

Microstepping using Bipolar Signals

Microstepping is a drive method to get rotation steps smaller than a full step. This is done by utilizing sinusoidal waves in the phases (instead of the rectangular signals of the Wave drive) and have the signals in 90° out of phase with each other. Figure 2.4 shows the current over all two phases using microstepping (when taking one full step). Note that the current sometimes changes direction (negative values); allowing the current to change direction, the current is said to be *bipolar*, makes for a more powerful motor than if only using current in one direction[5].
The smaller steps (aka microsteps) from using microstepping should in theory become infinitely small if the current inputs are truly sinusoidal like in Figure 2.4. In

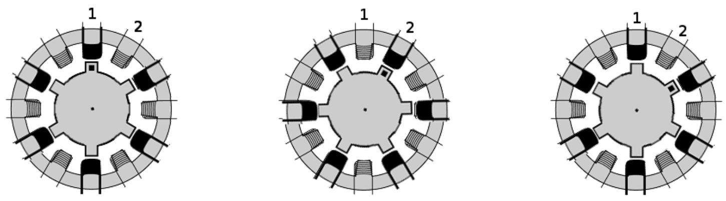

Figure 2.2 Conceptual model of of the workings of a two-phase stepper with wavedrive. Here one full step has been completed.

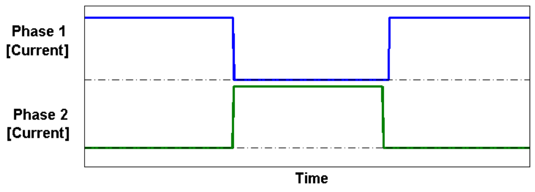

Figure 2.3 Current-Time plot for Wave Drive of a two-phase step. The dotted black lines are zero.

reality this is not possible; mechanical drawbacks such as backlash will limit the size of the microstep. It can also be hard to produce a smooth current curve for the motor[6].

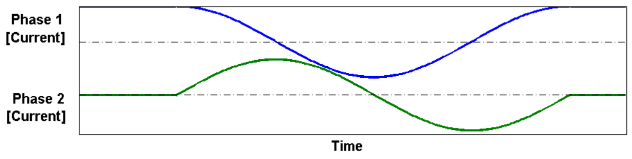

Figure 2.4 Current-Time plots for Microstepping (with bipolarity) of a two-phase step. The dotted black lines are zero for respective phase.

Rotary Encoders

Steppers are usually equipped with rotary encoders that provides angular feedback. Encoders are either *absolute* or *incremental*[7]. The absolute encoder maintains angular positions on a power-reboot. The incremental encoder records only changes in position and does not maintain angular positions on a power-reboot[7]. The incremental encoders are therefore in need of a *homing procedure* on every reboot to find an initial position.

A common method to record angular changes (in incremental encoders) is the *quadrature method*[7]; this method utilizes a disc connected to the driveshaft. The disc generates, when the motor rotates, two sets of pulses that are in 90 degrees out of phase which each other. By comparing the data between the two sets one can extrapolate direction and magnitude of changes in angular position.

2.3 Hardware

This section will provide the reader with a quick overview of the hardware involved in the project.

The FMB Oxford M3

The FMB Oxford M3 is a mirror positioning system designed for good precision, resolution, and repeatibility[9]. The system utilizes the Stewart Platform for a complete 6 DOF movement giving the user full control within the workspace of the robot. The linear actuators used in the system are 2-phase, bipolar, parallel stepper motors (Appendix 7.1, Table 7.1) which are positioned between two thick polymer concrete blocks (base plate, positional plate). These motors produce linear movement by converting rotational movements with a corkscrew. The steppers use incremental encoders of quadrature types. Figure 2.1 depicts the same FMB Oxford model as described here.
The resolutions and repeatabilities in all DOF are in the low μrad- and μm- range as can be seen in Table 7.1 in Appendix 7.1. The workspace is relatively small; $\pm 2°$ in the rotary DOF and somewhere between $\pm$ 40 mm for the positional DOF.

Initial Control Layout, MCS-8 The FMB Oxford platform comes with its own control system. This system, the MCS-8, consists of 3 main components; the Delta Tau PMAC2 for motion control, Phyton ZMX+ stepper drivers to provide power and current signals for the stepper motors, and an embedded Linux computer for ethernet communications to and from the Delta Tau unit. Figure 2.6 shows how all these components line up.

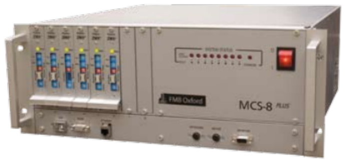

Figure 2.5 The MCS-8 – the control system that comes with the FMB Oxford robot.

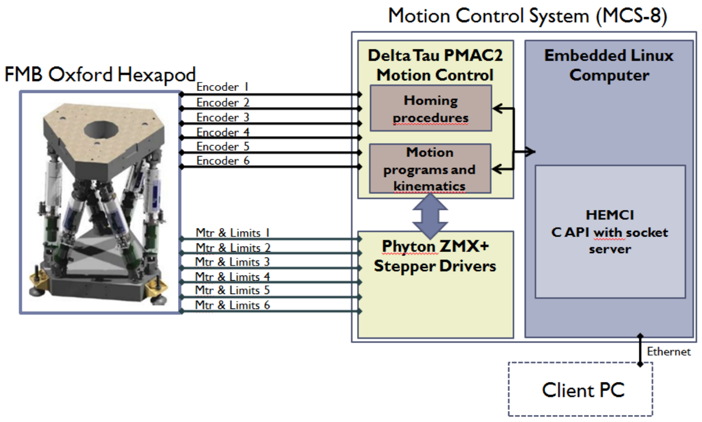

Figure 2.6 Control Layout of the MCS-8 with the FMB Oxford M3.

Galil DMC-4080 & Stepper Drivers

The DMC-4080 is Galils highest performing stand-alone controller[8]. Up to eight axes can be controlled independently and it supports both servo- and stepper- motors. It is designed to solve complex motion problems and the unit comes with several modes of motion and functions in solving these. A few interesting modes that are relevant to the project are for an example: *Point-to-Point positioning*, *PVT- mode* and *Electronic CAM* – what these functions do exactly will be explained in Section 2.4.
Stepper drivers are also needed to control the hexapod. As long as the drivers are capable of producing a bi-directional current of up to approximately 2 A and also of producing *small enough* microsteps as to not exceed the readhead resolution of the encoder (see Table 7.1,7.1, Appendix 7.1) they should suffice.
The standard 4080- unit comes with built-in stepper drivers. We will in this project only use the controller-board of the 4080 meaning that the stepper drivers are not included; six separate stepper drivers have therefore been connected (Figure 2.7).

Control Layout, Galil DMC-4080 Exchanging the control system of MCS-8 with the 4080 (and its own stepper drivers) would yield a similar control layout. The most notable change would be the lack of an embedded computer as the 4080 comes with its own ethernet handling system. Figure 2.8 depicts the control layout to be tested in this project.

Stepper Control & Motion with the DMC-4080 The 4080 can control stepper motors in a closed-loop; there are however some limitations – the 4080 does not support (for steppers) the standard closed-loop controllers such as PID[8]. How the 4080 handles errors between feedback and reference values will be covered in Section 3.1. The following two subsections explains two standard modes of motion that

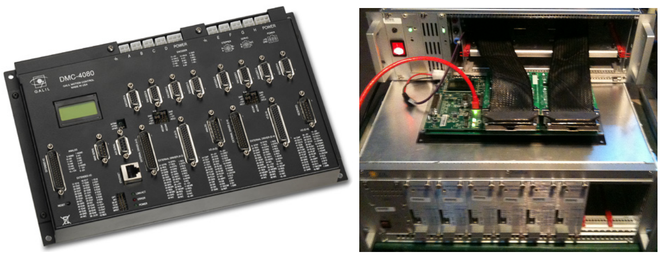

Figure 2.7 Left – standard 4080-unit with built-in stepper drivers. Right – Controller-board of the 4080 connected to six stepper drivers (used in the project).

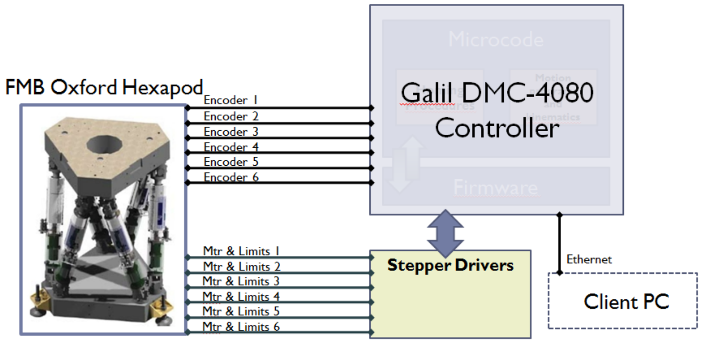

Figure 2.8 Possible control layout of the Galil DMC-4080 with the FMB Oxford M3.

the 4080 will use for the stepper motors.

Point-to-Point Positioning When doing a standard point-to-point positioning all steppers will follow either a triangular or trapezoidal velocity profile[8]. The shape of the profile is defined by the allowed maximum velocity v_{max}, acceleration a_a, and deceleration a_d of the motor. These values have to be defined before the move is initiated. Figure 2.9 illustrates how the position-velocity-acceleration profiles would look like.

Jogging Mode When jogging is initiated the motor will accelerate (with acceleration a_a) up to maximum allowed velocity v_{max} and hold this until stopped by a limit switch or by command from the user[8]. Figure 2.10 shows the motion profile of the mode.

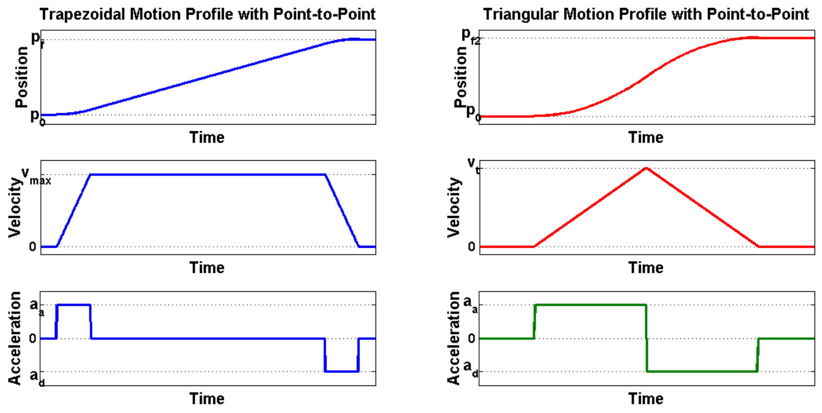

Figure 2.9 Left – Trapezoidal motion profile with maximum velocity v_{max} and final position p_f. Right – Triangular motion profile with maximum velocity v_t and final position p_{f2}. Note that $v_t \leq v_{max}$ and $p_{f2} \leq p_f$.

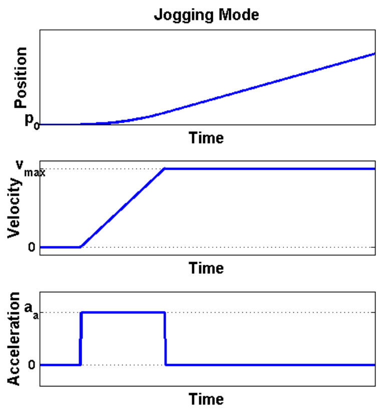

Figure 2.10 Jogging mode; Very similar motion profile to the point-to-point positioning.

2.4 Working in the Microcode of the DMC-4080

A rough architecture diagram of the 4080 can be seen in Figure 2.11. The firmware is here more low-level of the two and is also the fastest; code can be altered and/or added to the firmware but this can only be done via Galil (the company that makes the controller). The microcode, which is slower than the firmware, is however completely open for the user to edit or add new code. We will in this project only be working at the microcode level.

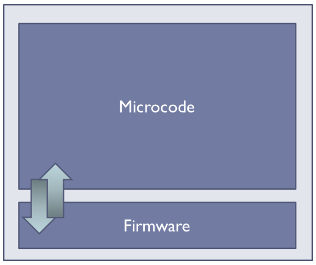

Figure 2.11 Architecture diagram of the DMC-4080.

Generic Microcode of the DMC-2182

SOLEIL Synchrotron has been using the DMC-2182, a predecessor of the 4080, for several years and has during this time developed a *Generic Microcode* for handling different modes of motion, error- and limit switches and communication with other devices. The microcode does this by utilizing existing modes and functions in the firmware. Some of the most useful and relevant functions (to this project) of the generic microcode from the DMC-2182 will now be listed and briefly explained:

Stepper Closed-Loop This can roughly be explained as a loop (in the microcode level) with the microcode comparing encoder-feedback with reference values every iteration. The 2182 can only (like the 4080) do correctional moves when the motor is at a stop. So rather than having the motor stop and handle every error, the system only do this when an error limit (defined by the user) is exceeded. If an error limit is exceeded during a move, the system activates *Dynamic Error Handling*. If an error limit is found to be exceeded directly after a move, the system activates *Static Error Handling*.

Dynamic Error Handling Stops the motor and does a correctional move. If error is within limit the motor continues towards the commanded position. If not; the system tries again. If error diverges and still does so after a certain number of tries, the system will abort and immediately stop the motor.

Static Error Handling Does a correctional move. If error is within limit the positioning is flagged as complete. If not; the system tries again. If error diverges and still does so after a certain number of tries, the system will abort and immediately stop the motor.

Limit Switch Handling Makes sure to stop the motor. If the motor is engaged in a multiple-axis move all relevant motors are stopped.

Backlash Compensating Since no positional corrections can be made during a move, backlash can not be dynamically compensated for. The generic microcode has however implemented point-to-point backlash compensation; if the user knows the magnitude of the backlash the positioning can overcompensate and hence make sure the final position will be correct (and thus minimize the use of static error handling).

Procedure for Finding Encoder Index This procedure jogs the motor (see Section 2.3) at a speed and direction defined by the user while listening for a transition (low$\leftrightarrow$high) on the encoder index channel. If a transition is detected the motor comes to an abrupt stop.
If the user still wants a higher precision (i.e position the axis closer to the edge of the transition) the process can be repeated but in the reverse direction and with a slower speed.

Homing Procedure Similar procedure as for finding encoder indexes; the only difference is that the homing sequence listens for a transition on the home input.

Multiple-Axis Modes of Motion The microcode enables synchronized moves for multiple axes (up to four motors at a time) for point-to-point positioning or jogging (see Section 2.3). The system also supports *electronic gearing*; a mode of motion can move several slave axes with a gear ratio towards a common master axis.

3

Method

3.1 Microcode Migration & Advanced Functions Implementation for the 4080

The generic microcode from the DMC-2182 was migrated to the 4080 and *extended* to incorporate new and relevant functions[8].

Switch-Deceleration Command

The Switch-Deceleration (SD) command automatically changes the motor deceleration value (to a value set by the user) when a limit switch is activated.

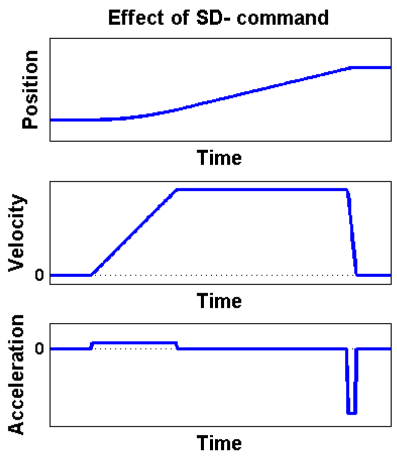

Figure 3.1 Effects of the SD- command when motor activates a forward limit switch. Here it sets the deceleration to a value 10 times higher than the acceleration.

Find-Index Command

The procedure for finding the encoder index described in Section 2.4 was completely exchanged with this more efficient and precise sequence:

The FI-command will start a sequence to find the edge of next index pulse of the encoder. This sequence is a 2-step method: first the axis jogs in a direction defined by the user while listening for an index pulse on the encoder index channel. When a transition from low to high is detected, the system will record the axis position and have the axis stopped. The second part of the sequence is that the axis is commanded to reverse and reach the previously recorded position.

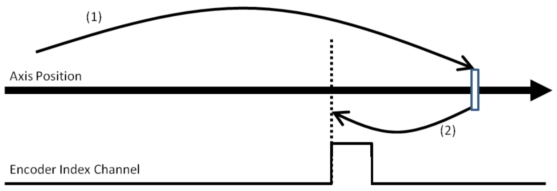

Figure 3.2 The workings of the FI- command. The procedure consists of a two-stage method to find the edge of the encoder index pulse.

Stepper Position Maintenance Mode, Stepper Closed-Loop at Firmware-Level

The generic microcode of the 2182 requires a software loop at the microcode level for running the steppers in a closed loop (see Stepper Closed-Loop in Section 2.4). This same loop already exists at the firmware level for the 4080; this means a faster and more reliable updating interval between loop iterations and therefore a better monitoring on the stepper errors. This mode is called the Stepper Position Maintenance (SPM) Mode.

The microcode method of closing the loop has therefore been altered to instead utilize the firmware SPM-mode. If error limits are exceeded the same microcode *Dynamic Error Handler*- and *Static Error Handler*- methods descibed in Section 2.4 will be launched.

PVT-Mode

This mode of motion allows the user to create complex and arbitrary motion trajectories for one or several axes. The user does so by defining a series of Position-Velocity-Time coordinates they should traverse during their respective trajectories. Up to 256 coordinates can be buffered per axis (and up to 8 axes can run simultaneously in PVT); these points are removed one by one from the buffer during motion. Additional points can be added to the buffer during motion which allows for essentially infinite trajectory lengths.

The mode interpolates between points with a third degree polynomial:

$$P(t) = a \cdot t^3 + b \cdot t^2 + c \cdot t + d \tag{3.1}$$

where $P(t)$ denotes the axis position at time t and where a, b, c and d are calculated from the boundary values (and initial axis position) of position-velocity-time coordinates inputted by the user.

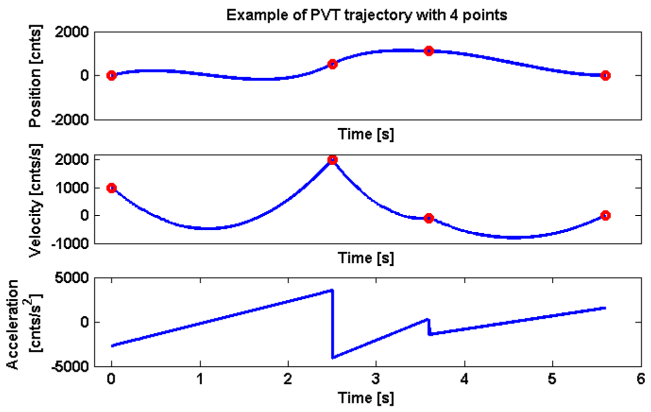

Figure 3.3 Example of PVT trajectory (1 axis) with 4 points. Red circles show the point coordinates the user defined, the blue lines are the resulting third degree interpolation between the points.

ECAM

ECAM (Electronic Cam) is a mode of motion that focuses on periodic synchronization of multiple axes in motion. It does this by defining a *master axis* and *slave axes*. All slaves are synchronized to the master through a table- based relationsship which the user will have to build; this table will contain the positions of all slaves at corresponding master positions. Linear interpolation is done between tabular points. Figure 2.7 illustrates how it works.

The slave motions in Figure 3.4 are entirely dictated by the motion of the master (the user commands the master to move which in turn moves the slaves). Basically any complex trajectory can be defined via the table-based relationsship. The ECAM table can hold up to 256 segments per axis and can control up to eight axes. Data can not be added to the table during motion which means that trajectories are finite and limited within its 256 segments.

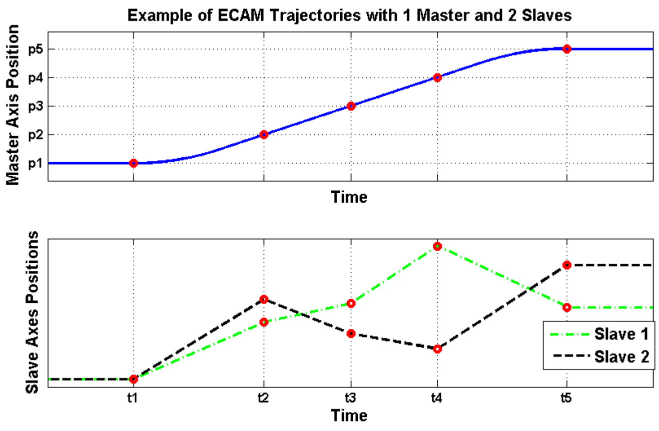

Figure 3.4 Example of ECAM trajectory with one master axis (top) and two slave axes (bottom). Five points of synchronization have been defined (red circles); this means that when the master axis reaches these points the slaves should be at their respective positions.

3.2 Extracting the Kinematic Equations & Homing Procedure

As was previously mentioned in Section 2.1; the reverse- and forward kinematics are both needed to control the hexapod in a closed loop. SOLEIL Synchrotron have these available but they are written in the coding format of the Delta Tau PMAC2. They have therefore been extracted and will in this section be presented and explained.

Reverse Kinematics

The reverse kinematics are given by the expression in Equation 3.2.

$$
\begin{aligned}
\mathbf{L} = \mathbf{L}(x,y,z,\theta,\phi,\psi) = \begin{pmatrix} l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{pmatrix}^T &= \\
= \begin{pmatrix} |\mathbf{P_1}| & |\mathbf{P_2}| & |\mathbf{P_3}| & |\mathbf{P_4}| & |\mathbf{P_5}| & |\mathbf{P_6}| \end{pmatrix}^T & \\
\text{where: } \mathbf{P_i} = \mathbf{T} + \mathbf{R}\cdot\mathbf{p}_i - \mathbf{b}_i = \begin{pmatrix} x \\ y \\ z+h \end{pmatrix} + \mathbf{R}\cdot\begin{pmatrix} x_{pi} \\ y_{pi} \\ z_{pi} \end{pmatrix} - \begin{pmatrix} x_{bi} \\ y_{bi} \\ z_{bi} \end{pmatrix} &
\end{aligned}
$$

$$
\mathbf{R} = \begin{pmatrix} \cos\psi\sin\theta & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \sin\psi\cos\theta & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{pmatrix}
$$

$$
\begin{aligned}
i = 1,2,3,4,5,6 \text{ – leg number} \\
h\text{– height of positional plate over base plate at home position} \\
(l_1,l_2,l_3,l_4,l_5,l_6) \text{ – corresponding leglengths to input data} \\
(x_{bi},y_{bi},z_{bi})\text{– universal leg coordinates on base plate} \\
(x_{pi},y_{pi},z_{pi})\text{– local leg coordinates on positional plate}
\end{aligned} \tag{3.2}
$$

Interpretation The reverse kinematics presented in Equation 3.2 can be broken down into a series of coordinate transformations and vector geometry. Each leg length is the norm of the vector $\mathbf{P}_i$ which in turn is calculated from vectors $\mathbf{T}$, $\mathbf{b}_i$, and multiplication between transformational-rotational matrix $\mathbf{R}$ and vector $\mathbf{p}_i$. Note that the vector $\mathbf{p}_i$ (positional joints for all legs) are the ones undergoing coordinate transformations (see Figure 3.5).

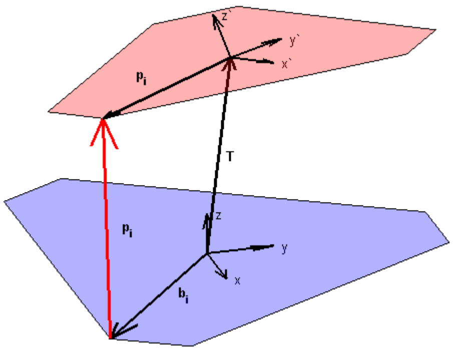

Figure 3.5 Coordinate transformation and vector geometry gives leg lengths.

Forward Kinematics

The forward kinematics are used to estimate the states of the virtual axes using only the leg lengths of the robot. The kinematics extracted from the Delta Tau unit does this iteratively; calculating estimates each iteration until errors are within limits.
We have the state estimations, $\hat{\mathbf{x}}_k = \begin{pmatrix}\hat{x}_k & \hat{y}_k & \hat{z}_k & \hat{\theta}_k & \hat{\phi}_k & \hat{\psi}_k\end{pmatrix}^T$, where index k represents the current number of iterations, for the corresponding actual leg lengths $\mathbf{L} = \begin{pmatrix}l_1 & l_2 & l_3 & l_4 & l_5 & l_6\end{pmatrix}^T$. The kinematics always start with an initial estimate $\hat{\mathbf{x}}_0$:

$$\hat{\mathbf{x}}_0 = \begin{pmatrix}\hat{x}_0 & \hat{y}_0 & \hat{z}_0 & \hat{\theta}_0 & \hat{\phi}_0 & \hat{\psi}_0\end{pmatrix}^T \tag{3.3}$$

From this we can utilize the expression used in Equation 3.2 to get a corresponding leg length estimation $\hat{\mathbf{L}}_k$ and leg length error $\tilde{\mathbf{L}}_k$:

$$\hat{\mathbf{L}}_k = \begin{pmatrix}\hat{l}_{k1} & \hat{l}_{k2} & \hat{l}_{k3} & \hat{l}_{k4} & \hat{l}_{k5} & \hat{l}_{k6}\end{pmatrix}^T = \mathbf{L}(\hat{x}_k, \hat{y}_k, \hat{z}_k, \hat{\theta}_k, \hat{\phi}_k, \hat{\psi}_k) \tag{3.4}$$

$$\tilde{\mathbf{L}}_k = \hat{\mathbf{L}}_k - \mathbf{L}, \text{ where } \mathbf{L} \text{ are the current leglengths} \tag{3.5}$$

If the expression in Equation 3.6 is true then no further calculations are needed and the state estimations $\hat{\mathbf{x}}_k$ are within error limits. If not – the estimation is not good enough and needs further adjusting.

$$|\tilde{\mathbf{L}}_k| < d, \text{ where d is a threshold value} \tag{3.6}$$

This is done by linear approximation; Equation 3.7 shows the relationship between real leg lengths and its leg length- and state estimations.

$$\mathbf{L} = \hat{\mathbf{L}}_k + \hat{\mathbf{J}}_{\hat{L}_k} \cdot (\mathbf{x} - \hat{\mathbf{x}}_k) + \mathscr{O}(||\mathbf{x} - \hat{\mathbf{x}}_k||) \approx \hat{\mathbf{L}}_k + \hat{\mathbf{J}}_{\hat{L}_k} \cdot (\mathbf{x} - \hat{\mathbf{x}}_k) \tag{3.7}$$

$$\text{where: } \hat{\mathbf{J}}_{\hat{L}_k} = \mathbf{J}_{\hat{L}_k}(\hat{x}_k, \hat{y}_k, \hat{z}_k, \hat{\theta}_k, \hat{\phi}_k, \hat{\psi}_k) = \begin{bmatrix}
\frac{\delta \hat{l}_{k1}}{\delta \hat{x}_k} & \frac{\delta \hat{l}_{k1}}{\delta \hat{y}_k} & \frac{\delta \hat{l}_{k1}}{\delta \hat{z}_k} & \frac{\delta \hat{l}_{k1}}{\delta \hat{\theta}_k} & \frac{\delta \hat{l}_{k1}}{\delta \hat{\phi}_k} & \frac{\delta \hat{l}_{k1}}{\delta \hat{\psi}_k} \\
\frac{\delta \hat{l}_{k2}}{\delta \hat{x}_k} & \frac{\delta \hat{l}_{k2}}{\delta \hat{y}_k} & \frac{\delta \hat{l}_{k2}}{\delta \hat{z}_k} & \frac{\delta \hat{l}_{k2}}{\delta \hat{\theta}_k} & \frac{\delta \hat{l}_{k2}}{\delta \hat{\phi}_k} & \frac{\delta \hat{l}_{k2}}{\delta \hat{\psi}_k} \\
\frac{\delta \hat{l}_{k3}}{\delta \hat{x}_k} & \frac{\delta \hat{l}_{k3}}{\delta \hat{y}_k} & \frac{\delta \hat{l}_{k3}}{\delta \hat{z}_k} & \frac{\delta \hat{l}_{k3}}{\delta \hat{\theta}_k} & \frac{\delta \hat{l}_{k3}}{\delta \hat{\phi}_k} & \frac{\delta \hat{l}_{k3}}{\delta \hat{\psi}_k} \\
\frac{\delta \hat{l}_{k4}}{\delta \hat{x}_k} & \frac{\delta \hat{l}_{k4}}{\delta \hat{y}_k} & \frac{\delta \hat{l}_{k4}}{\delta \hat{z}_k} & \frac{\delta \hat{l}_{k4}}{\delta \hat{\theta}_k} & \frac{\delta \hat{l}_{k4}}{\delta \hat{\phi}_k} & \frac{\delta \hat{l}_{k4}}{\delta \hat{\psi}_k} \\
\frac{\delta \hat{l}_{k5}}{\delta \hat{x}_k} & \frac{\delta \hat{l}_{k5}}{\delta \hat{y}_k} & \frac{\delta \hat{l}_{k5}}{\delta \hat{z}_k} & \frac{\delta \hat{l}_{k5}}{\delta \hat{\theta}_k} & \frac{\delta \hat{l}_{k5}}{\delta \hat{\phi}_k} & \frac{\delta \hat{l}_{k5}}{\delta \hat{\psi}_k} \\
\frac{\delta \hat{l}_{k6}}{\delta \hat{x}_k} & \frac{\delta \hat{l}_{k6}}{\delta \hat{y}_k} & \frac{\delta \hat{l}_{k6}}{\delta \hat{z}_k} & \frac{\delta \hat{l}_{k6}}{\delta \hat{\theta}_k} & \frac{\delta \hat{l}_{k6}}{\delta \hat{\phi}_k} & \frac{\delta \hat{l}_{k6}}{\delta \hat{\psi}_k}
\end{bmatrix} \tag{3.8}$$

Setting $\tilde{\mathbf{x}}_k = \hat{\mathbf{x}}_k - \mathbf{x}$ and solving for it in Equation 3.7 gives:

$$\begin{aligned} &\mathbf{L} \approx \hat{\mathbf{L}}_k + \hat{\mathbf{J}}_{\hat{L}_k} \cdot (\mathbf{x} - \hat{\mathbf{x}}_k) = \hat{\mathbf{L}}_k - \hat{\mathbf{J}}_{\hat{L}_k} \cdot \tilde{\mathbf{x}}_k \iff \\ &\iff \tilde{\mathbf{x}}_k \approx \hat{\mathbf{J}}_{\hat{L}_k}^{-1} \cdot (\hat{\mathbf{L}}_k - \mathbf{L}) = \hat{\mathbf{J}}_{\hat{L}_k}^{-1} \cdot \tilde{\mathbf{L}}_k \end{aligned} \tag{3.10}$$

The new state estimations are then given by:

$$\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{x}}_k - \tilde{\mathbf{x}}_k \approx \hat{\mathbf{x}}_k - \hat{\mathbf{J}}_{\hat{L}_k}^{-1} \cdot \tilde{\mathbf{L}}_k \tag{3.11}$$

The result from Equation 3.11 is then tested by inputting it into Equation 3.4 and calculating the resulting error. If error is still beyond the threshold d (Equation 3.5) new estimates will be produced; Figure 3.6 gives the workflow of the process.

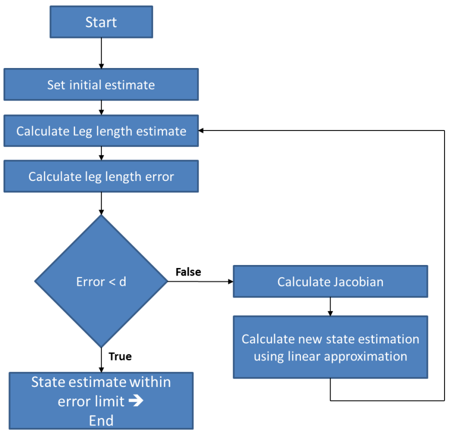
Start
Set initial estimate
Calculate Leg length estimate
Calculate leg length error
Error < d
False
Calculate Jacobian
Calculate new state estimation using linear approximation
True
State estimate within error limit ➔ End

Figure 3.6 Flowchart of the forward kinematics process.

Initial Estimates & Threshold Value The forward kinematics require a quite large amount of calculations especially when determining the Jacobian (see Equation 3.8) and its inverse. These calculations are repeated every iteration when producing the state estimates; for good efficiency it is therefore of interest to do this with the minimal amount of iterations as possible.

Equation 3.11 shows how new state estimates $\hat{\mathbf{x}}_\mathbf{k}$ are produced from initial estimate $\hat{\mathbf{x}}_\mathbf{0}$ by a series of linear approximations. These linear approximations will contain errors as the Stewart Platform is a non-linear process; choosing a good initial estimate $\hat{\mathbf{x}}_\mathbf{0}$ will directly affect the number of iterations needed.

The choice of threshold d is a matter of weighing efficiency against precision: the kinematics should be calculated fast but keep within the systems resolution.

Homing Procedure

As was previously mentioned in Section 2.3; the encoders of the linear actuators are incremental. This means that the length of the legs are no longer known when rebooting the electronics and we have to run a sequence to find the home position. The system must be able to find the home position from any point inside the workspace and should do so without violating the physical restraints or by exceeding the workspace of the robot. The homing procedure extracted from the MCS-8 is illustrated in Figure 3.7.

This procedure relies on three key aspects:

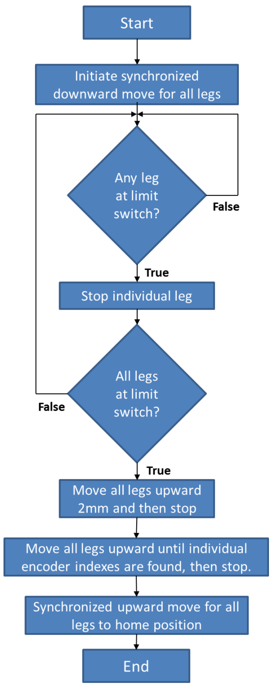
Start
Initiate synchronized downward move for all legs
Any leg at limit switch?
False
True
Stop individual leg
All legs at limit switch?
False
True
Move all legs upward 2mm and then stop
Move all legs upward until individual encoder indexes are found, then stop.
Synchronized upward move for all legs to home position
End

Figure 3.7 Flowchart of the homing sequence.

1. Synchronized moves on all six legs.

   The system must find the home position for all six legs simultaneously the reason being the parallel nature of the process (all legs are connected to the same plate).

2. Backward limit switches.

   Stopping all axes at their respective backward limit switches ensures (if one assumes all legs are built similarly) that they all have roughly the same distance to their individual encoder indexes.

3. Encoder indexes and offsets.

   Once the encoder indexes have been found, the position of the axes are known and they need only to move a certain distance towards their home positions.

3.3 Matlab Simulation

The kinematics and homing procedure in Section 3.2 were tested in a Matlab SimMechanics/Simulink environment (see Figure 3.8). The physical process was mod-

eled using setup data from the FMB Oxford documentation (such as leg coordinates on the base- and platform plates, leg lengths at the home positions, etc). All six actuators were set to follow positional reference values (output from the reverse kinematics) and feed their respective leg lengths into the forward kinematics. No controller was used in this simulation (all actuators were simply *set* to their positions,velocities and accelerations), the system was run in open-loop and did so in a continuous manner.
The point of this simulation was therefore only to test and observe that:

1. Reference values fed through the reverse kinematics would result in a correct response from the physical model (Example: positive reference value for z results in an upward direction of the positional plate).

2. The forward kinematics would, with a good enough threshold value (see Equation 3.6), yield almost identical values as those fed into the reverse kinematics.

3. Confirm that the homing procedure is functional.

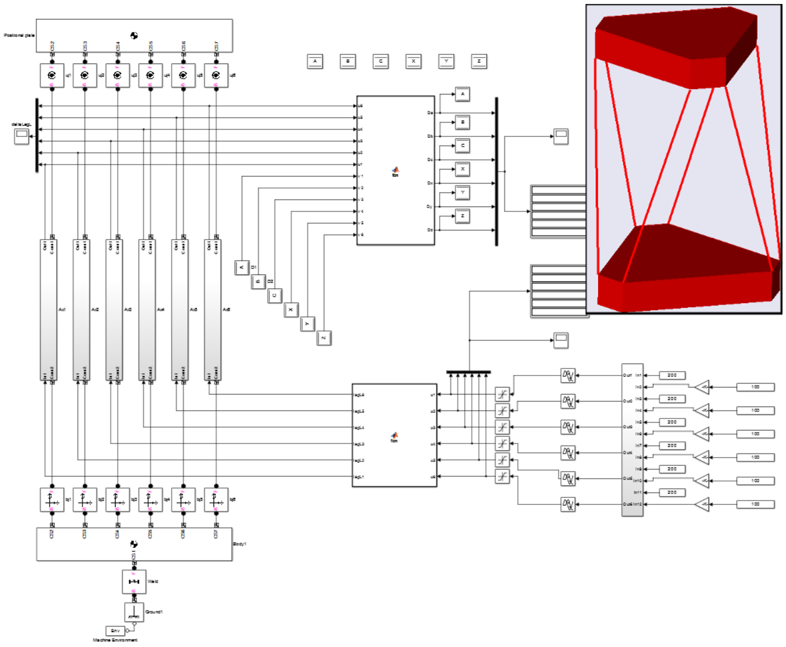

Figure 3.8 Simulation of the process using SimMechanics/Simulink. Image to the upper right corner is a 3D model of the process.

Confirming the Kinematics

Figure 3.9 presents the results of a simulation where all virtual axes $(x, y, z, \theta, \phi, \psi)$ are headed one by one towards their range limits (see Appendix 7.1,Table 7.1). The second graph in the figure presents the virtual axes errors with a threshold value of $d = 4.4721 \cdot 10^{-9}$ m (see Equation 3.6). From this the data in Table 3.1 and 3.2 are calculated.

	x (m)	y (m)	z (m)
Standard Deviation, σ	$3.22 \cdot 10^{-10}$	$2.82 \cdot 10^{-10}$	$5.19 \cdot 10^{-11}$
Maximum Deviation, σ_m	$2.67 \cdot 10^{-9}$	$3.00 \cdot 10^{-9}$	$5.76 \cdot 10^{-10}$

Table 3.1 Standard- and maximum deviations of the forward kinematics with (x, y, z) with threshold $d = 4.4721 \cdot 10^{-9}$ m.

	θ (Degrees)	ϕ (Degrees)	ψ (Degrees)
Standard Deviation, σ	$2.32 \cdot 10^{-8}$	$2.97 \cdot 10^{-8}$	$1.92 \cdot 10^{-8}$
Maximum Deviation, σ_m	$5.26 \cdot 10^{-7}$	$1.47 \cdot 10^{-7}$	$2.13 \cdot 10^{-7}$

Table 3.2 Standard- and maximum deviations of the forward kinematics with (θ, ϕ, ψ) with threshold $d = 4.4721 \cdot 10^{-9}$ m.

Comparing all deviations in Tables 3.1 and 3.2 with the required resolutions found in Appendix 7.1, Table 7.1 we see that the precision of the forward kinematics are well within limits (errors are $\sim 10^3 - 10^4$ times smaller). The response of the model was also confirmed to be correct by observing the movements of the 3D model (see Figure 3.8) during the simulation.

Additional simulations have been run; the results of these can be found in Appendix 7.2 and will be used a reference when implementing the controller on the real process.

Confirming the Homing Procedure

Figure 3.10 shows a simulation of a slightly altered homing sequence from the one defined in Figure 3.7; the encoder index was here assumed to be in the backward limit switch and the 2mm move was therefore not needed. The homing was initiated from an arbitrary initial position which in this case was $(x, y, y, \theta, \phi, \psi) = (10, 5, 0, 0, 0, 0)$ and ended up at the home position of $(x, y, y, \theta, \phi, \psi) = (0, 0, 0, 0, 0, 0)$.

The lower graph (when $t < 60$) in Figure 3.10 shows how the states x, y, θ, ϕ and ψ remain constant during the initial synchronized move downwards. These states go towards zero during the second part of the homing; when all leg actuators reach their respective limit switches and stop individually. The final stage of

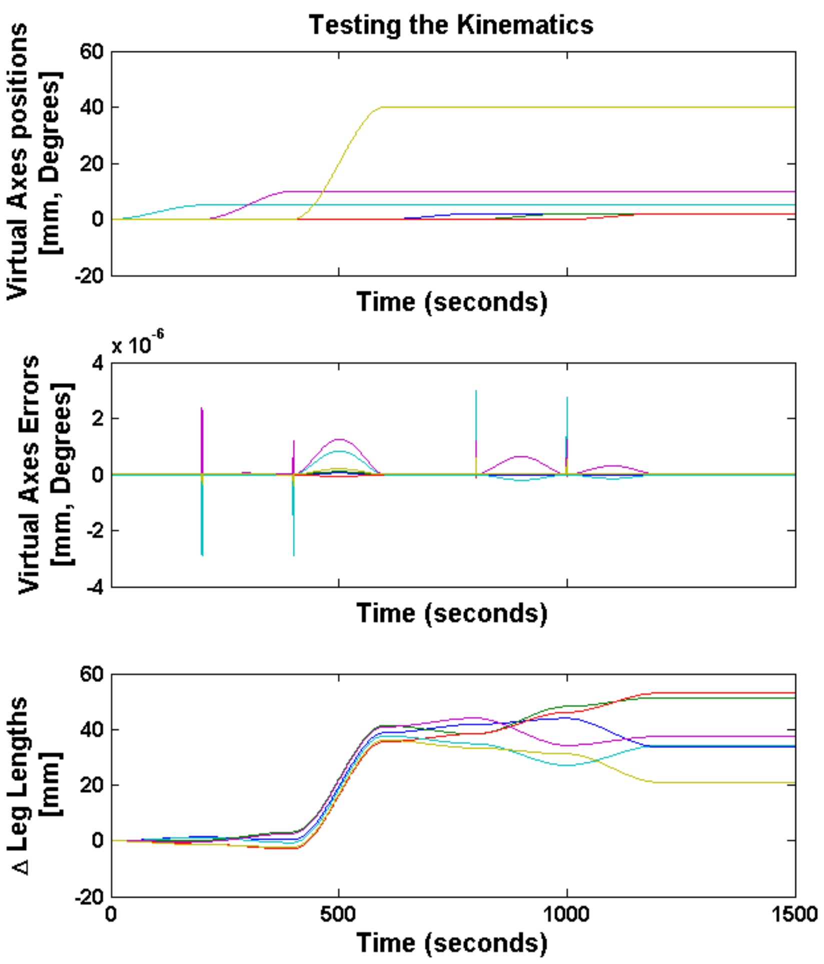

Figure 3.9 Testing the kinematics in Matlab simulation; All virtual axes $(x, y, z, \theta, \phi, \psi)$ go towards their range limits.

the sequence, a synchronized move upwards for all actuators, sets the last non-zero state z to zero.

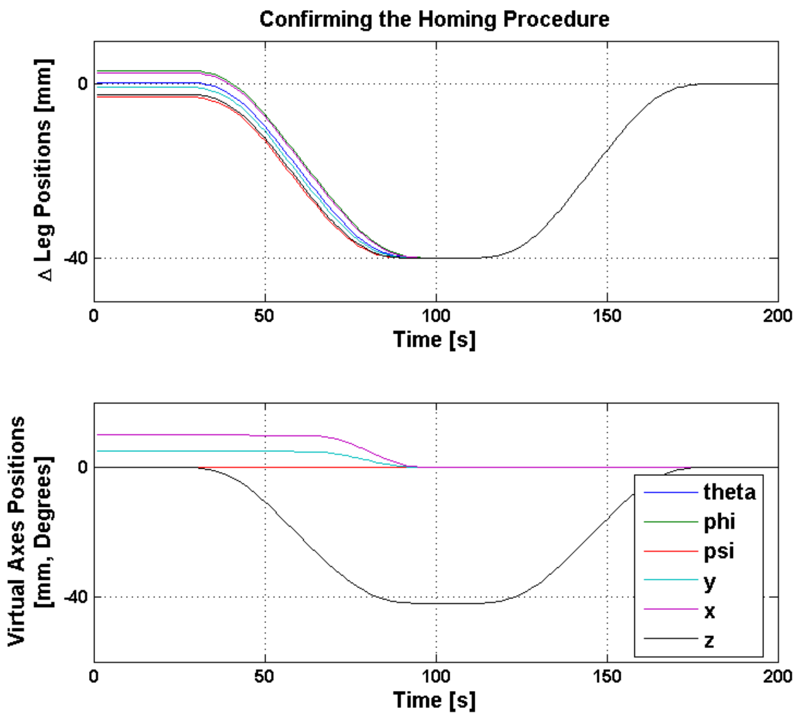

Figure 3.10 Confirming the homing sequence; All leg lengths decrease until they hit their individual limit switches then go towards their home positions.

3.4 Implementing the Controller

Having prepared the microcode, extracted the kinematics, and extracted the homing procedure the controller could now be implemented.

Kinematics into Microcode

The kinematics presented in Section 3.2 were implemented into the microcode format and adjusted to be as efficient as possible without violating the state $(x, y, z, \theta, \phi, \phi)$ resolutions defined in Appendix 7.1, Table 7.1. The following factors had to be addressed in the implementation:

Matrix Calculations & Inverses The microcode format does not have any built-in functions to handle matrix- or vector multiplication or to calculate matrix inverses. A standard matrix-multiplication algorithm was therefore utilized. Equation 3.12 shows an example of a 2x2 matrix multiplication; the algorithm would simply calculate the elements of **C** directly as according to the equation.

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B}, \text{ where } \mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$
$$\text{and } \mathbf{C} = \begin{pmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} & a_{11} \cdot b_{12} + a_{12} \cdot b_{22} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} \end{pmatrix} \quad (3.12)$$

Matrix inverses were calculated using the *Gauss-Jordan* algorithm (a version of Gauss-elimination). Equation 3.13 shows how the process would (with row operations on an augmented matrix) take place for arbitrary 3x3 matrices.

We have matrix **D** where $\mathbf{D}^{-1}$ is calculated from:

$$\left[\begin{array}{ccc|ccc} d_{11} & d_{12} & d_{13} & 1 & 0 & 0 \\ d_{21} & d_{22} & d_{23} & 0 & 1 & 0 \\ d_{31} & d_{32} & d_{33} & 0 & 0 & 1 \end{array}\right] \xrightarrow{\text{Row operations}} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & i_{11} & i_{12} & i_{13} \\ 0 & 1 & 0 & i_{21} & i_{22} & i_{23} \\ 0 & 0 & 1 & i_{31} & i_{32} & i_{33} \end{array}\right]$$
$$\text{where } \mathbf{D}^{-1} = \begin{bmatrix} i_{11} & i_{12} & i_{13} \\ i_{21} & i_{22} & i_{23} \\ i_{31} & i_{32} & i_{33} \end{bmatrix} \quad (3.13)$$

Fixed-Point Variables Microcode variables are fixed-point and use 6 bytes for storage. They are stored in the **Q31.16** format meaning 31 integer bits, 16 fractional bits and one bit reserved for sign. This format is set and can not be changed. To

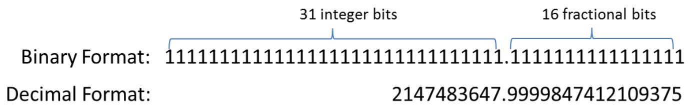
31 integer bits
16 fractional bits
Binary Format: 1111111111111111111111111111111.1111111111111111
Decimal Format: 2147483647.9999847412109375

Figure 3.11 Illustration of the Q31.16 format. The image shows the maximum value a microcode variable can hold and its decimal counterpart.

maximize both precision and range of the kinematic calculations it is important that all variables utilize as much as possible of the 6 bytes available. This is done by evaluating the variables range and (if need be) multiply its value with a scalar to utilize as many integer and fractional bits as possible. To illustrate this, imagine the following pseudo-program code:

```
a=100*b;
```

if in this case $(-10^5 < b < 10^5)$ then $(-10^7 < a < 10^7)$. A better way of utilizing all the bits of the Q31.16 format would be to *multiply* the values of a with a factor. A better way of storing the value a would then be:

```
a=(100*b)*100;
```

the range of a is now: $(-10^9 < a < 10^9)$ which would use up more available bits.

Implementing Custom Trigonometric Functions Table 7.1 in Appendix 7.1 state that the ranges of the rotational states are within $\pm 2°$ and that the resolutions are at their smallest approximately $2.865 \cdot 10^{-3}$ Degrees. The firmware does provide with native sine- and cosine- functions that the microcode can use. These are however very limited when it comes to resolution; the smallest angle increment that they use are $\sim 0.5 \cdot 10^{-3}$ Degrees which is more than 15 times worse than needed. Custom sine- and cosine- functions were implemented with Taylor/Maclaurin series:

$$\sin(\alpha) = \sum_{k=1}^{\infty} \frac{(-1)^{k+1} \cdot \alpha^{2k-1}}{(2k-1)!} = \alpha - \frac{\alpha^3}{3!} + \mathcal{O}_s(\alpha^5) \approx \alpha - \frac{\alpha^3}{3!} \tag{3.14}$$

$$\cos(\alpha) = \sum_{k=0}^{\infty} \frac{(-1)^{k} \cdot \alpha^{2k}}{(2k)!} = 1 - \frac{\alpha^2}{2} + \frac{\alpha^4}{4!} + \mathcal{O}_c(\alpha^6) \approx 1 - \frac{\alpha^2}{2} + \frac{\alpha^4}{4!} \tag{3.15}$$

where α in this case is any angle in the range of $\pm 2°$. To get an estimate of the error we can calculate the average value of the ordo-notations over the angle range. Estimates of the ordo- notations were calculated in MATLAB with the following equations:

$$\mathcal{O}_s(\alpha^5) \approx \sum_{k=3}^{20} \frac{(-1)^{k+1} \cdot \alpha^{2k-1}}{(2k-1)!}, \; \mathcal{O}_c(\alpha^6) \approx \sum_{k=3}^{20} \frac{(-1)^{k} \cdot \alpha^{2k}}{(2k)!} \tag{3.16}$$

where the mean values from $\sim 1.3 \cdot 10^5$ uniformly chosen values of α within $\pm 2°$ were calculated to be:

$$\bar{\mathcal{O}}_s = -1.052 \cdot 10^{-15}, \; \bar{\mathcal{O}}_c = -3.589 \cdot 10^{-13} \tag{3.17}$$

Forward Kinematics – Initial Estimates & Thresholds At this point the kinematics were fairly functional and just needed tweaking to yield values with acceptable precision and a good enough calculation time. The importance of choosing a good initial estimate $\hat{\mathbf{x}}_0$ and threshold value d was mentioned in Section 3.2.

INITIAL ESTIMATES: The MCS-8 (see Section 2.3), the unit from which the kinematics were extracted from, sets the initial estimate $\hat{\mathbf{x}}_0$ (current position) to the state estimate used in the previous position. This choice makes sense if the forward kinematics are calculated fast in relation to the process changes (this would mean that the previous position would be fairly close to the current one). Running the forward kinematics on the 4080 would however suggest a different approach. Initially the calculation time to produce one set of acceptable state estimations (before any major tweaking was done) landed on 1.5 seconds or more. The initial estimate was therefore instead set to the state reference values; trusting that the good repeatability of the stepper motors would yield somewhat similar reference values to the actual states.

THRESHOLD: 300 data-points consisting of all six leg-lengths during a positive z-move from the home position $(x_0, y_0, z_0, \theta_0, \phi_0, \psi_0)$ were generated for testing purposes. This set of data was fed through the implemented 4080 forward kinematics with a set threshold value d and compared with the reference values. This process was repeated for 9 different threshold values. Figure 3.12 show the results

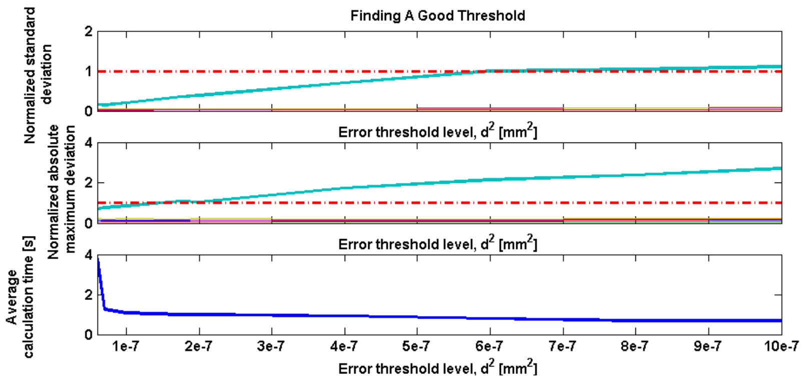

Figure 3.12 Evaluating different thresholds.

of running the forward kinematics with different thresholds. The top graph depicts the standard deviation of the state errors and the middle shows the maximum deviation. Both graphs contain each 6 curves depicting the respective error of each state $(x, y, z, \theta, \phi, \psi)$. All curves in both graphs are normalized to their respective resolutions (Table 7.1, Appendix 7.1); if any curve goes beyond the red dotted line it means the error was greater than the required state resolution. The bottom graph show the average calculation time (over 300 calculations) to produce one set of acceptable state estimates. The average calculation time can be very high (4 seconds or more) for very low thresholds; at this level round-off errors are dominant and the system has a hard time in finding good estimates. The kinematics should be calculated as fast as possible but without errors that exceed the state resolutions. With this in mind the threshold was set to $d^2 = 1.1 \cdot 10^{-7}$ mm$^2 \Rightarrow d \approx 1.0488 \cdot 10^{-7}$ m; as high as it can be without any maximum deviations or standard deviations exceeding the state resolutions.

Hexadecimal- vs Decimal Format Microcode variables are normally displayed in the decimal format. The system does however support hexadecimals as an alternative; this format offers several advantages over the decimal:

BINARY CONVERSIONS: All computerized systems use the binary format which has a base of 2. The hexadecimal format uses a base of 16 which happen to be an even multiplier from the binary system. What this effectively means is that the conversion between the two formats will be simpler and require less operations than a binary-decimal conversion would. An example is given in Equation 3.19.

$$\begin{gathered}11110010110110111101_2 = 1048576_{10} + 262144_{10} + 131072_{10} + 65536_{10} + \\ + 8192_{10} + 2048_{10} + 1024_{10} + 256_{10} + 128_{10} + 32_{10} + 8_{10} + 4_{10} + 1_{10} = 994749_{10} \\ 11110010110110111101_2 = 1111\ 0010\ 1101\ 1011\ 1101_2 = F2DBD_{16}\end{gathered} \tag{3.19}$$

Equation 3.19 illustrates how the conversion to decimal (in this case) needs to map 13 binary numbers to decimal and then add them together. In the case of binary-hexadecimal, 5 groups-of-four of binary numbers are directly mapped to hexadecimals.

FRACTIONAL REPRESENTATION: Equation 3.19 presents an example of an ordinary integer conversion. If the variable also has a fractional component the binary-decimal conversion can give off truncation errors; Equation 3.20 shows how the truncation error would present itself when displaying the variable on the 4080. It can only display decimal fractional numbers up to 4 digits.

$$0.0100100100000000_2 = 0.28515625_{10} \xrightarrow{\text{trunc}} 0.2851_{10} \tag{3.20}$$

No truncation error will be present in the hexadecimal format and all 16 fractional bits will always be displayed:

$$0.0100100100000000_2 = 0.4900_{16} \tag{3.21}$$

Implementing the Controller of Virtual Axes

With kinematics implemented into the microcode the next step is constructing a controller for moving the virtual axes x, y, z, θ, ϕ, and ψ. Control will be accomplished by combining point-to-point positioning and jogging (see Section 2.3), advanced functions and modes of motion (see Section 3.1) and kinematic equations.

Backlash Eventhough point-to-point backlash compensating was available in the Generic Microcode from the DMC-2182 (see Section 2.4) it was *not* implemented for the Stewart Platform controller; the reason was that the trajectories produced by the kinematics might be too complex (coupled with eventual error handling) to estimate the final positional backlash compensation. Even so; the backlash of all leg-motors were measured by running a synchronized move in one direction, followed by a synhchronized move in the opposite direction. The difference between the starting- and final- positions would be the backlash of the corresponding legs:

$$\mathbf{b}_r = \begin{bmatrix}0.00613 & 0.005073 & 0.005700 & 0.00444 & 0.00718 & 0.00622\end{bmatrix} \text{ mm} \tag{3.22}$$

Virtual Axes Setup It has been previously mentioned that the 4080 supports independent control of up to eight axes. The 4080 also includes two virtual motors; these are controlled like any other real motor but differ in that they lack encoders. The virtual states x, y, z, θ, ϕ and ψ will utilize the virtual motors of the 4080. One apparent limitation is the *lack* of virtual motors available on the 4080; we have six virtual states and the 4080 only provides with two. The following method was therefore used:

ONE VIRTUAL MOTOR $\longrightarrow$ SIX VIRTUAL STATES: When making a move from any initial position $(x_i, y_i, z_i, \theta_i, \phi_i, \psi_i)$ to an arbitrary position $(x_f, y_f, z_f, \theta_f, \phi_f, \psi_f)$ the virtual motor will be set to move the state that that does the furthest positional move. The speed, acceleration and deceleration of the virtual motor are then set to state-specific values and the motor is commanded to do a point-to-point positioning. The positions of the other five virtual states will set in linear proportion to the virtual motor position. Figure 3.13 shows an example

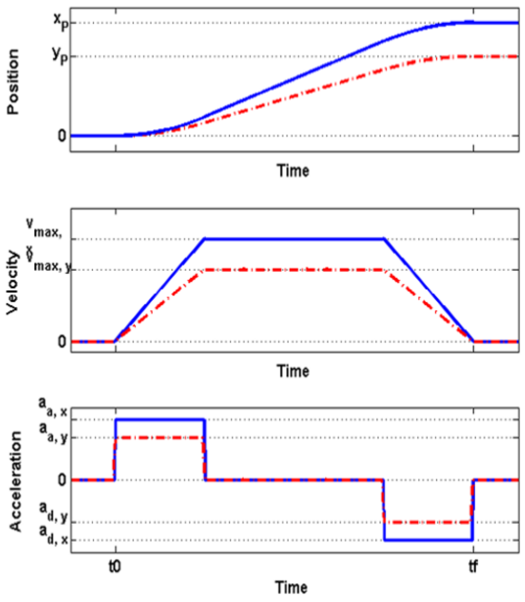

Figure 3.13 Example of two virtual states x and y driven by one virtual motor. Blue solid line represents state x and the red dotted line y.

of two virtual states driven by one virtual motor. Here the states x and y start at $(x_i, y_i) = (0, 0)$ and finish in (x_f, y_f). State x does the furthest positional move of the two and is driven by the virtual motor in a point-to-point positioning (with

$v_{max,x}$,$a_{a,x}$ and $a_{d,x}$) from 0 to x_f. During this move state y is defined as:

$$y = y_i + \frac{y_f}{x_f} \cdot x \tag{3.23}$$

thus ensuring that both x and y reach their final positions at the same time. Figure 3.13 also show that the maximum velocity, acceleration and deceleration are lower for state y than x; these values are in fact defined by:

$$v_{max,y} = \left(\frac{y_f}{x_f}\right) \cdot v_{max,x},\ a_{a,y} = \left(\frac{y_f}{x_f}\right) \cdot a_{a,x},\ a_{d,y} = \left(\frac{y_f}{x_f}\right) \cdot a_{d,x} \tag{3.24}$$

Reverse Kinematics – Connecting Virtual Axes to System Actuators ...

PRE-BUFFERING: All virtual axes will have the trapezoidal or triangular motion profile as defined in Section 2.3; this effectively means that all virtual axis trajectories can be calculated and buffered prior to any move (as long as initial and final positions are known). PVT and ECAM, modes of motion available to the 4080, both offer pre-buffering and interpolation between points.

PVT VS ECAM: The criterias for choosing mode of motion comes down to how they handle the following points:

1. Buffering
2. Interpolation Between Points
3. Multi-Axis Synchronization
4. Error Handling
5. Ease of Use

ECAM turns out to be more suitable for the task, especially when it comes to points 3-5. The fact that one master axis controls six slaves fits well into the design method mentioned in Section 3.4. This particular setup is also well suited for multi-axis synchronization. The main feature that makes ECAM the preferable choice is its ease of use compared to PVT; we only need the corresponding slave positions when buffering with ECAM. PVT needs positions, velocity and time for all six slave axes which is much harder to come by.

Forward Kinematics The forward kinematics will provide, by converting current leg-lengths to virtual states, feedback to the system. The two main objectives with the implementation of the forward kinematics is optimization (to get the calculation time as low as possible) and precision.

Control Scheme

Figure 3.14 shows the control scheme used in the project where: $\mathbf{x}_f = (x_f, y_f, z_f, \theta_f, \phi_f, \psi_f)$ is any destinational point inputted by the user, $\mathbf{x}_R = (x_R, y_R, z_R, \theta_R, \phi_R, \psi_R)$ are trajectory reference points generated from point-to-point positioning (Section 2.3), $\mathbf{L}_R = (l_{R1}, l_{R2}, l_{R2}, l_{R4}, l_{R5}, l_{R6})$ being the reference points for the legs, $\mathbf{L} = (l_1, l_2, l_2, l_4, l_5, l_6)$ the actual leg lengths, and $\hat{\mathbf{x}} = (\hat{x}, \hat{y}, \hat{z}, \hat{\theta}, \hat{\phi}, \hat{\psi})$ the estimated virtual axis positions.

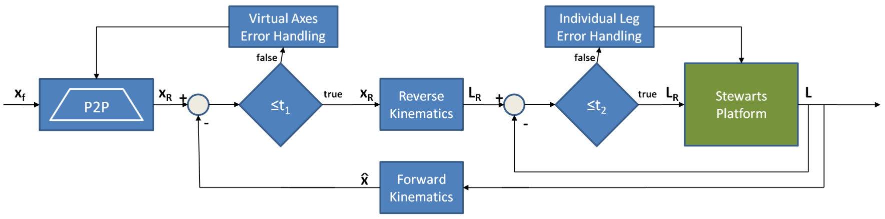

Figure 3.14 Control Scheme

The scheme involves two feedback loops; the inner loop checks and corrects that all individual leg lengths are within threshold limits (here portrayed by array $\mathbf{t}_2$), the outermost loop checks that all virtual states are within threshold limits $\mathbf{t}_1$. Arrays $\mathbf{t}_1$ and $\mathbf{t}_2$ each switches between two different sets of values depending on how far along the trajectory the process is. If the trajectory has reached its final position then Equation 3.26 is true otherwise Equation 3.28 is.

$$\begin{aligned} \mathbf{t}_1 &= \begin{pmatrix} 1\mu m & 1\mu m & 1\mu m & 28.63\mu Degrees & 57.3\mu Degrees & 28.63\mu Degrees \end{pmatrix} \\ \mathbf{t}_2 &= \begin{pmatrix} 0.1\mu m & 0.1\mu m & 0.1\mu m & 0.1\mu m & 0.1\mu m & 0.1\mu m \end{pmatrix} \end{aligned} \tag{3.26}$$

$$\begin{aligned} \mathbf{t}_1 &= \begin{pmatrix} 100\mu m & 100\mu m & 100\mu m & 8mDegrees & 8mDegrees & 8mDegrees \end{pmatrix} \\ \mathbf{t}_2 &= \begin{pmatrix} 22\mu m & 22\mu m & 22\mu m & 22\mu m & 22\mu m & 22\mu m \end{pmatrix} \end{aligned} \tag{3.28}$$

The dynamic error thresholds in Equation 3.28 are much larger than the sets in Equation 3.26; fluidity and speed of motion (minimize the use of dynamic error handling) was prioritized over precision during motion.

The virtual axis error handling is pretty straightforward; it will make a synchronous stop for all legs, recalculate the trajectories from current position and then restart the move. The individual error handling works exactly as dynamic error handling as was described in Section 2.4; all legs will do a synchronous stop, the error will be corrected, and the trajectory will be resumed.

4 Results

The control scheme in Figure 3.14 proved successful; the 4080 was able to perform point-to-point positioning and could do so with one or multiple states simultaneously. Running the reverse kinematics took on average 24 ms per try and about 600-800 ms per forward-kinematic-conversion.

Forty-nine tests were executed and the leg-length data was post-processed through the kinematics used in the MATLAB simulation in Section 3.3; Figure 4.1 shows the homing procedure and Figures 4.2–4.7 show some results when running the virtual states one by one towards set values (while keeping all other virtual states zero). Error handling is visible in some of the graphs; Figure 4.4 for an example show individual leg error handling at $t \approx 3.1 \cdot 10^4 - 3.5 \cdot 10^4$ ms and virtual axis error handling later at $t \approx 5.8 \cdot 10^4$ ms. Parasitic movements (moving states affecting other states) were also observable; Figures 4.8 and 4.9 presents two occurences where parasitic movements arose from z-moves, Figure 4.10 shows parasitic movements during a ψ-move. Table 4.1 shows how much the states deviated from their commanded positions and reference trajectories during the actual moves. Time comparisons were made using the 4080 vs MCS-8 in controlling the hexapod; these results can be found in Table 4.2.

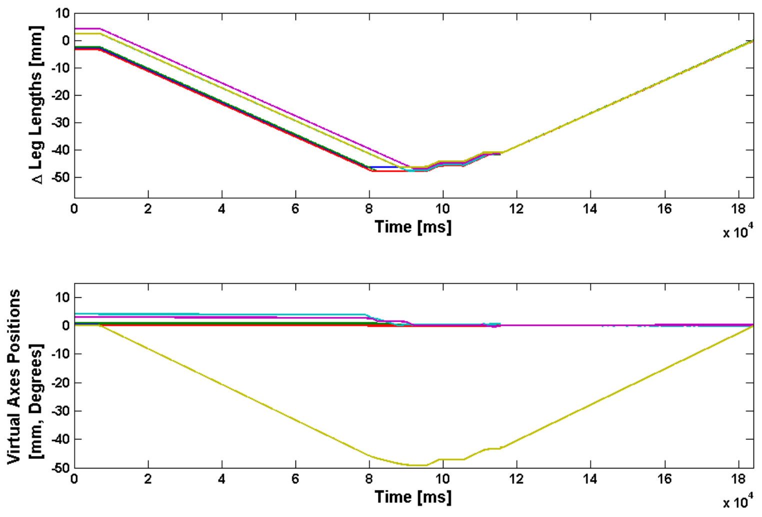

Figure 4.1 Results from running the homing procedure from an arbitrary position.

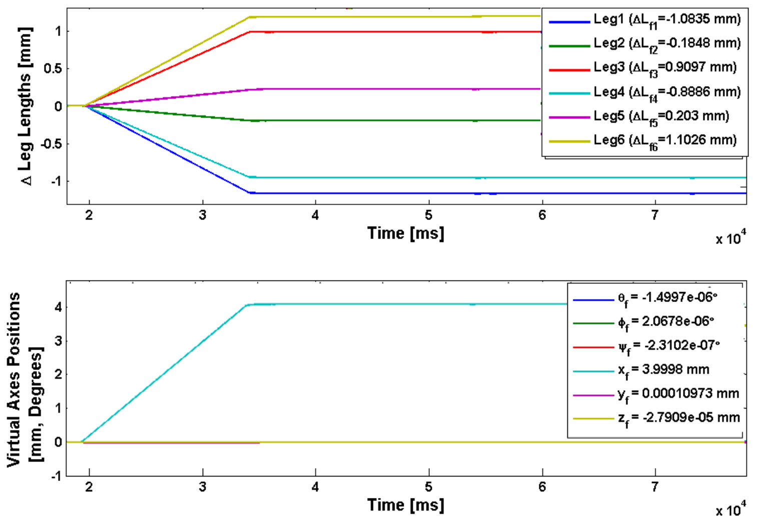

Figure 4.2 Results from running state *x* from 0 to 4 mm whilst keeping all other virtual states zero.

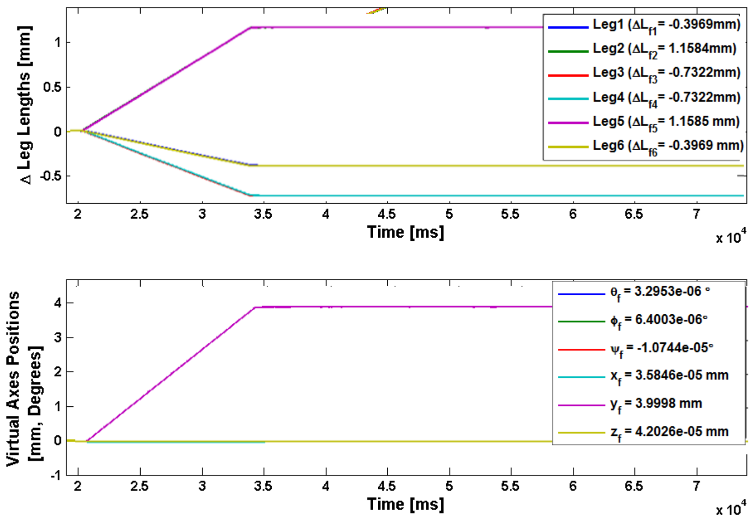

Figure 4.3 Results from running state y from 0 to 4 mm whilst keeping all other virtual states zero.

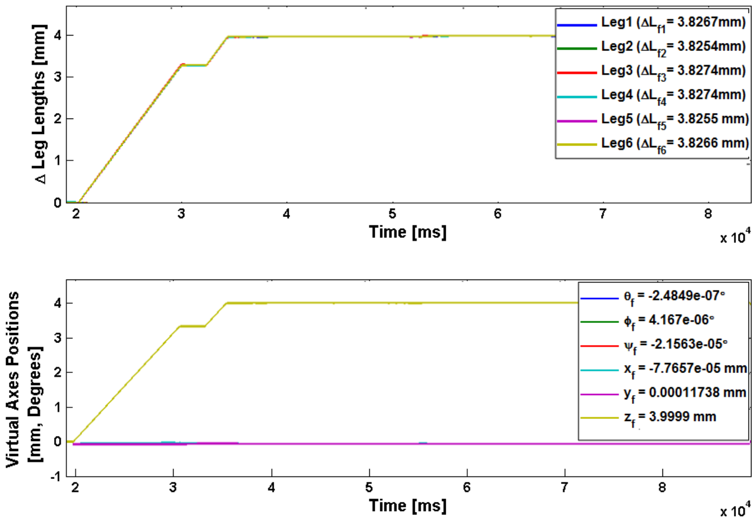

Figure 4.4 Results from running state z from 0 to 4 mm whilst keeping all other virtual states zero.

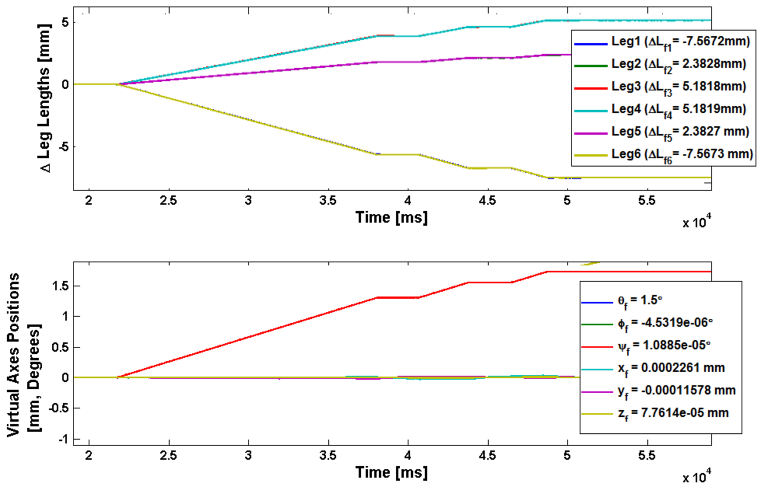

Figure 4.5 Results from running state θ from 0 to 1.5° whilst keeping all other virtual states zero.

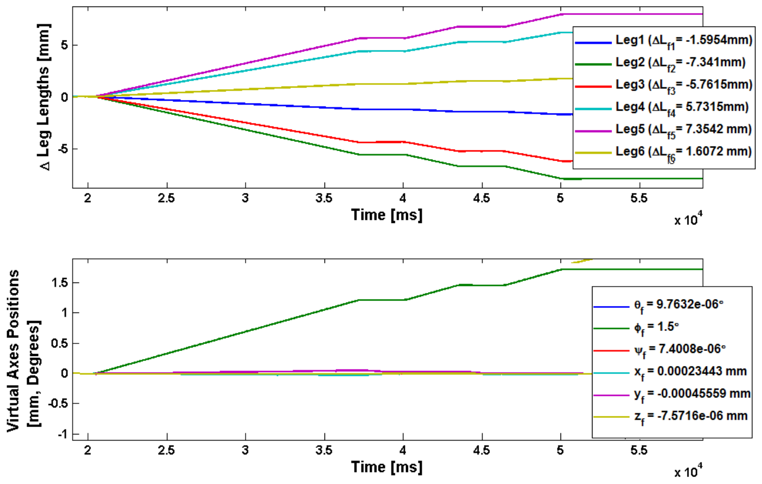

Figure 4.6 Results from running state ϕ from 0 to 1.5° whilst keeping all other virtual states zero.

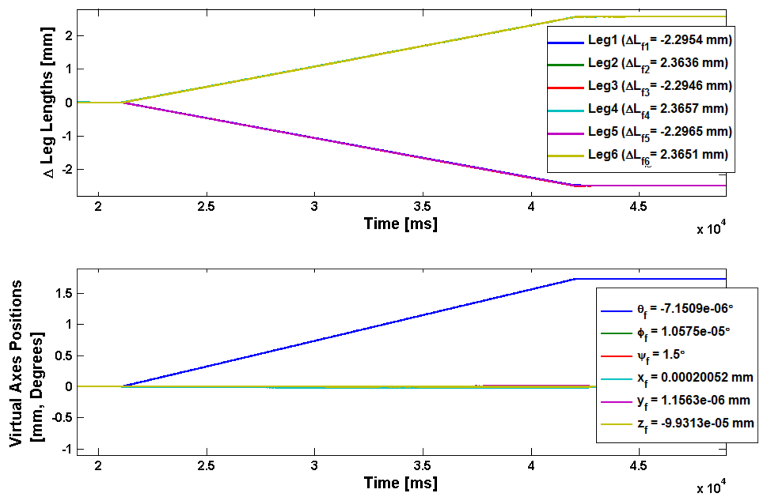

Figure 4.7 Results from running state ψ from 0 to 1.5° whilst keeping all other virtual states zero.

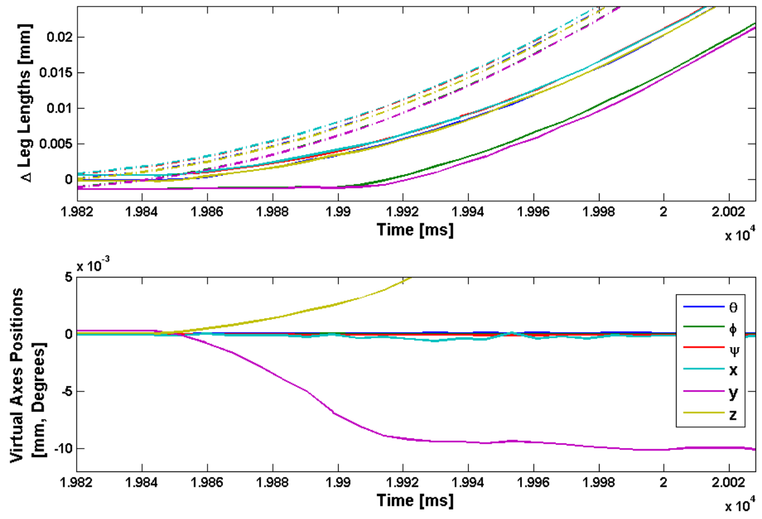

Figure 4.8 Parasitic movements of states x, y, θ, ϕ, ψ (they should be zero) whilst z is moving. The dotted lines in the upper graph represents the reference values.

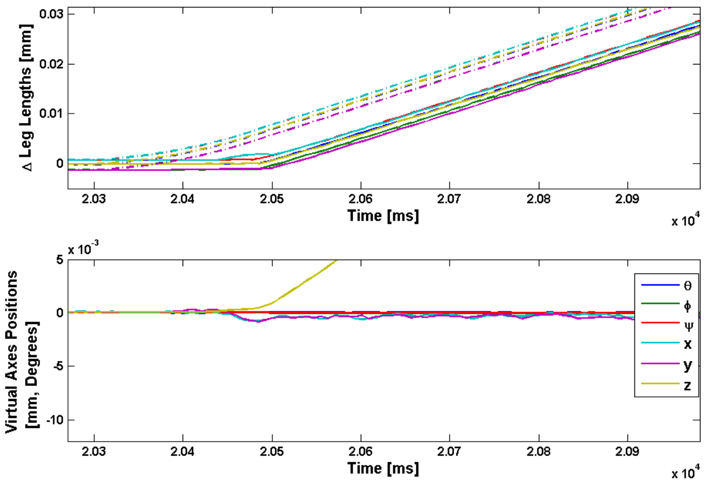

Figure 4.9 Parasitic movements of states x, y, θ, ϕ, ψ (they should be zero) whilst z is moving. The dotted lines in the upper graph represents the reference values.

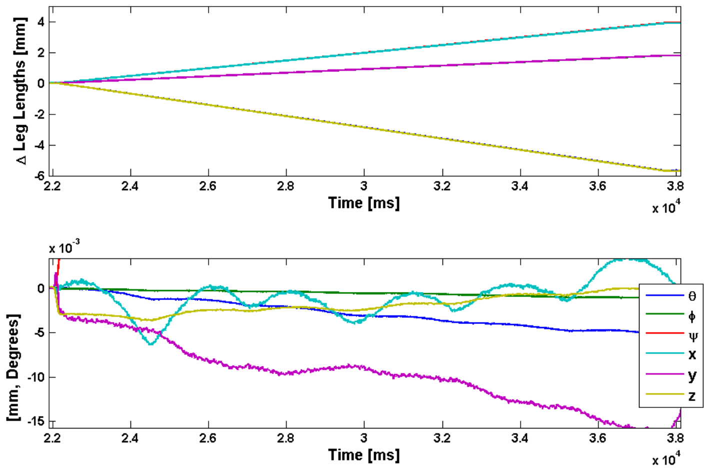

Figure 4.10 Parasitic movements of states x, y, z, θ, ϕ (they should be zero) whilst ψ is moving. The dotted lines in the upper graph represents the reference values.

	St Dev, σ	Max Dev, σ_m	Norm σ	Norm σ_m
x_f	$3.15 \cdot 10^{-7}$ m	$1.00 \cdot 10^{-6}$ m	0.32	1.00
y_f	$2.41 \cdot 10^{-7}$ m	$9.00 \cdot 10^{-7}$ m	0.24	0.90
z_f	$1.72 \cdot 10^{-7}$ m	$6.00 \cdot 10^{-7}$ m	0.17	0.60
θ_f	$9.50 \cdot 10^{-6}$ Degrees	$2.38 \cdot 10^{-5}$ Degrees	0.33	0.83
ϕ_f	$8.66 \cdot 10^{-6}$ Degrees	$2.38 \cdot 10^{-5}$ Degrees	0.15	0.42
ψ_f	$1.53 \cdot 10^{-5}$ Degrees	$2.52 \cdot 10^{-5}$ Degrees	0.53	0.88
x_d	$9.88 \cdot 10^{-6}$ m	$2.48 \cdot 10^{-5}$ m	9.88	24.80
y_d	$1.29 \cdot 10^{-5}$ m	$3.38 \cdot 10^{-5}$ m	12.90	33.80
z_d	$1.21 \cdot 10^{-6}$ m	$3.63 \cdot 10^{-6}$ m	1.21	3.63
θ_d	$2.71 \cdot 10^{-4}$ Degrees	$7.29 \cdot 10^{-4}$ Degrees	9.46	25.45
ϕ_d	$2.29 \cdot 10^{-4}$ Degrees	$7.71 \cdot 10^{-4}$ Degrees	4.00	13.46
ψ_d	$1.1 \cdot 10^{-3}$ Degrees	$3.90 \cdot 10^{-3}$ Degrees	38.39	136.13

Table 4.1 Results – Commanded & Dynamic Positions Deviations. $(x_f, y_f, z_f, \theta_f, \phi_f, \psi_f)$ denotes the final positions while $(x_d, y_d, z_d, \theta_d, \phi_d, \psi_d)$ denotes dynamical positions (positions during a move). The last two columns depicts the deviations normalized to the required state resolutions found in Table 7.1, Appendix 7.1.

	Move Length	DMC 4080	MCS-8
Small Moves	$\leq \sqrt{3} \cdot 0.1 \approx 0.17$ mm	16.2 s	2.1 s
Medium Moves & Ang displ	$\sim \sqrt{3} \approx 1.73$ mm	40.0 s	16.2 s
Big Moves	4 mm	59.3 s	9.9 s

Table 4.2 Time Comparisons (average); DMC-4080 vs MCS-8. Time was measured from when move-command was issued until controller flagged the positioning as complete.

5 Discussion – Analyzing the Results

The results confirm what the MATLAB simulations produced in Section 3.3 and Appendix 7.2. Comparing the upper graphs in Figures 4.2–4.7 with the ones found in Appendix 7.2 one can see a clear resemblance that only seem to differ when it comes to three key factors:

1. Velocities and Accelerations
2. Error Handling
3. Parasitic Movements

The first of these points simply comes down to that leg velocities and accelerations were chosen in the simulation to produce esthetically suitable curves, while maximum leg velocity in the real process had to be well within the limit presented in Table 7.1, Appendix 7.1. Error handling and parasitic movements will be discussed in the following subsections.

5.1 Static & Dynamic Behaviour

Table 4.1 presents the static and dynamic precisions of the system. We can see that the static deviations (standard- or maximum deviation) never exceed the required state resolutions presented in Table 7.1, Appendix 7.1, as the normalized deviations never exceeded 1. The dynamic deviations are much larger; in some cases the maximum deviation were up to 136.13 times larger than the required state resolution.

Both static and dynamic precisions holds well within the virtual state error limits presented in Equations 3.26-3.28, Section 3.4. As was previously mentioned in Section 3.4 the dynamic error limits were set to greater values than the static error limits. The main reason for doing so was to uphold a certain level of fluidity in motion and avoid (as much as possible) the stop-correct-resume method of the error handler. Also, since calculation time of the forward kinematics varies between 600-800 ms per conversion, the speed of virtual state error-checking will be limited to this conversion-time. It should be possible to run the system with a dynamic precision equal to the static precision; this would require a change in error limits as well as lower leg- velocities and accelerations; the process changes of the platform would have to be slowed down to match the long-time lag of the virtual state error checking.

5.2 Parasitic Movements

Figures 4.8–4.10 shows how some virtual states change during trajectories when they should in fact stay still. These changes are caused by the movement of other states, hence the name *parasitic movements*. One can observe three different sources of errors occuring:

1. Error Drifts (Figures 4.8,4.10)
2. Error Offsets (Figures 4.8,4.10)
3. Errors of Periodical Nature (Figure 4.10)

The drifts could be explained by ratio round-off errors with the method explained in Section 3.4; a faulty ratio would for an example in Equation 3.23 cause an error of linear nature.
Figures 4.8 and 4.9 both show similar moves but with different levels of parasitic errors. Figure 4.8 shows an error offset of magnitude $\sim 10^{-2}$, something which Figure 4.9 does not portray. One can observe a delay between reference trajectories and actual leg movements in both figures; this delay seems to be roughly the same for all legs in Figure 4.9 but slightly different for the legs in Figure 4.8. It is this mismatch of delays between the legs that cause a slight de-synchronization and hence error offsets. *Backlash* is a viable source of these delays which is confirmed by measuring the positional errors of the legs at the moment they start to move:

$$\mathbf{b}_1 = \begin{bmatrix} 0.001438 & 0.005052 & 0.001442 & 0.001495 & 0.005082 & 0.001496 \end{bmatrix} \text{ mm} \tag{5.1}$$

$$\mathbf{b}_2 = \begin{bmatrix} 0.00605 & 0.005673 & 0.005682 & 0.004724 & 0.006693 & 0.006122 \end{bmatrix} \text{ mm} \tag{5.2}$$

$\mathbf{b}_1$ is here the positional errors (at the moment of leg movement) measured in Figure 4.8 while $\mathbf{b}_2$ is from Figure 4.9. We see that these values are equal to or less than the measured backlash in Equation 3.22, Section 3.4. Also one can see that the legs from Figure 4.9 are similarily aligned to their respective backlash while the legs from Figure 4.8 are not.
Virtual state x in Figure 4.10 presents an error of periodic nature. Being periodic

would suggest that it stems from a rotational element and since the leg motors produce linear motion by converting rotational motion with a corkscrew it would be feasible that this is the source of error (misaligned mechanical parts for an example).

5.3 System Limitations

Closed-Loop Control Frequency Due to the calculation- speed limitation of the forward kinematics the system is only able to handle low frequency closed-loop control. The maximum frequency would be $f_{max} = \frac{1}{T_{max}} \approx \frac{1}{0.8} = 1.25Hz$.

Load Weight, Movements & System Flexibility As was previously mentioned in Section 2.1; the $(\hat{x}, \hat{y}, \hat{z}, \hat{\theta}, \hat{\phi}, \hat{\psi})$ estimations are acquired through kinematic conversions from the leg-lengths feedback $(l_1, l_2, l_3, l_4, l_5, l_6)$. These estimations (see Section 3.2) presume a non-flexible system and will thus yield estimation errors (in function of the applied weight load and movement) and thus affecting overall system precision. There are two possible types of weight loads to be considered:

1. **Static Loads,** Non-fluctuating load movements and weight
2. **Dynamic Loads,** Fluctuating load movements and/or weight

The error estimations could be decreased with an improved kinematic model of the system; this would however require a much more detailed knowledge of the system setup than is provided by the FMB Oxford documentation (in addition to load weight and/or movements). The best way to avoid these errors would be get $(x, y, z, \theta, \phi, \psi)$- feedback directly from the positional plate; this would eliminate the need of forward kinematics and thus eliminate estimation errors.

5.4 Summary & Conclusion

The Galil DMC–4080 has successfully been used as a controller of a stepper-motor driven Stewart Platform with μm- and μrad- static precision and resolution. Control was done point-to-point, and could do so freely in the workspace of the virtual axes. Closed-loop control was achievable but with certain limitations such as big time-lags in some of the feedback (due to 600-800 ms calculations of forward kinematics). Another limitation is how the DMC-4080 makes error corrections with stepper-motors; this results in a so called stop-correct-resume type of error-handler which utilizes error-limits (to have some flow in movement). System precision might also suffer due to system flexing due to loads; these errors arise from a kinematic model that doesn't take the flexing into account and sensors far from the positional plate. It is possible to have as high dynamic precision as is used in static positioning but due to the nature of the error handler and the long time-lags in feedback the time-costs of doing so would be very high. It is therefore preferable, if one can afford it, to have a less strict demand for dynamic precision. In the context of the stepper-driven Stewart Platform, the Galil DMC-4080 is a suitable closed-loop controller if time efficiency and smoothness of motion are not the driving factors. What truly limits the performance is calculation time of the kinematics (maximum closed-loop control frequency is approximately 1.25 Hz); the reader should bear in mind that these were (in this project) written in the microcode-level – if these were instead implemented into the *firmware* the controller should get a better time- performance since the kinematics would be running faster. Smoothness of motion would of course still be an issue since the controller would still utilize the stop-correct-resume method of error handling.

6

References

1. D. Stewart, "A platform with six degrees of freedom", *Proceedings of the Institution of Mechanical Engineers, Volume 180, No 15, Year 1965-1966, pp.371-386*

2. L.J. du Plessis, "An Optimization Approach to the Determination of Manipulator Workspaces", *etd-06012009-145209, University of Pretoria, Pretoria, Year 1999, pp.1-39*

3. T. Charters, R. Enguica, P. Freitas, "Detecting Singularities of Stewart Platform", *Mathematics-in-Industry Case Studies Journal, Vol 1, Year 2009, pp.66-80*

4. S. Kizir, Z. Bingul, "Position Control and Trajectory Tracking of the Stewart Platform", *Serial and Parallel Robot Manipulators - Kinematics, Dynamics, Control and Optimization, Dr. Serdar Kucuk (Ed.), ISBN: 978-953-51-0437-7*, Year 2012

5. Ericsson, "Stepper Motor Basics", *Industrial Circuits Application Note, Solarbotics Ltd.*, viewed 2013-05-01 <*http://www.solarbotics.net/library/pdflib/pdf/motorbas.pdf*>

6. R. Laidman, "Stepper Motors and Control: Part IV - Microstepping of Stepper Motors.", *Stepperworld.com*, viewed 2013-05-01 <*http://www.stepperworld.com/Tutorials/pgMicrostepping.htm*>

7. Michael B. Histand, David G. Alciatore, "Introduction to Mechanics and Measurement Systems", *Tata McGraw-Hill Publishing Company Limited, New Dehli, ISBN-13: 978-0-07-064814-2, Year 2007*

8. "DMC-40x0, Manual Rev. 1.0m", *Galil Motion Control Inc., viewed 2013-02-01 <http://www.galilmc.com/support/manuals.php>*

9. "Mirror Systems 2009", *FMB Oxford Ltd., viewed 2013-02-01 <http://www.fmb-oxford.com/uploads/files/MirrorSystems09.pdf>*

7

Appendix

7.1 FMB Oxford M3, Technical Data

M3 Motion	**Parameter**	**Specification**
Pitch (X-Rotation)	Drive	Linear actuators w/ limit switches & encoder
	Range	$\pm 2\,^\circ$
	Resolution	0.5 μrad $\approx$ 0.02865 milliDegrees
	Repeatability	1.0 μrad $\approx$ 0.05730 milliDegrees
Roll (S-Rotation)	Drive	Linear actuators w/ limit switches & encoder
	Range	$\pm 2\,^\circ$
	Resolution	1 μrad $\approx$ 0.05730 milliDegrees
	Repeatability	2.5 μrad $\approx$ 0.14324 milliDegrees
Yaw (Z-Rotation)	Drive	Linear actuators w/ limit switches & encoder
	Range	$\pm 2\,^\circ$
	Resolution	0.5 μrad $\approx$ 0.02865 milliDegrees
	Repeatability	1.0 μrad $\approx$ 0.05730 milliDegrees
Vertical (Z Direction)	Drive	Linear actuators w/ limit switches & encoder
	Range	$\pm$ 40 mm
	Resolution	<1.0 μm
	Repeatability	<2.0 μm
Lateral (X Direction)	Drive	Linear actuators w/ limit switches & encoder
	Range	$\pm$ 10 mm
	Resolution	<1.0 μm
	Repeatability	<2.0 μm
Longitudinal (S Direction)	Drive	Linear actuators w/ limit switches & encoder
	Range	$\pm$ 5 mm
	Resolution	<1.0 μm
	Repeatability	<2.5 μm

Table 7.1 Virtual Axes Motion

7.2 Matlab Simulation Results

The results presented in this section were produced in a Matlab SimMechanics/Simulink environment. All six plots show the change in all leg lengths when running the states x, y, z, θ, ϕ and ψ to set values whilst keeping all other states at zero. The variables ΔL_{fi} where $i = 1, 2, 3, 4, 5, 6$ denotes the total change in position for leg i (in mm).

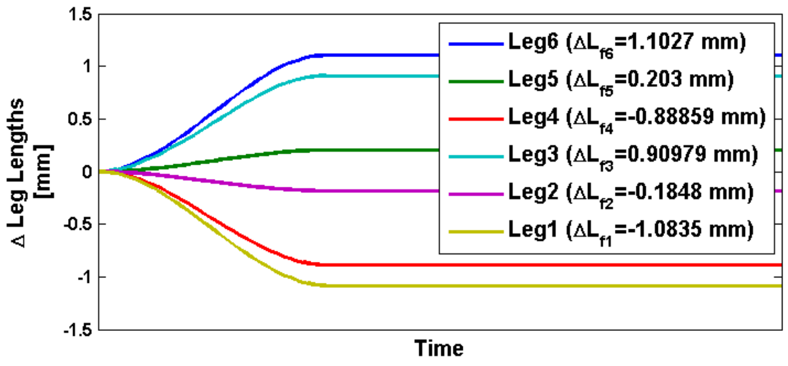

Figure 7.1 Changes in leg lengths when state $x \to 4$ mm from $(x_0, y_0, z_0, \theta_0, \phi_0, \psi_0) = (0,0,0,0,0,0)$

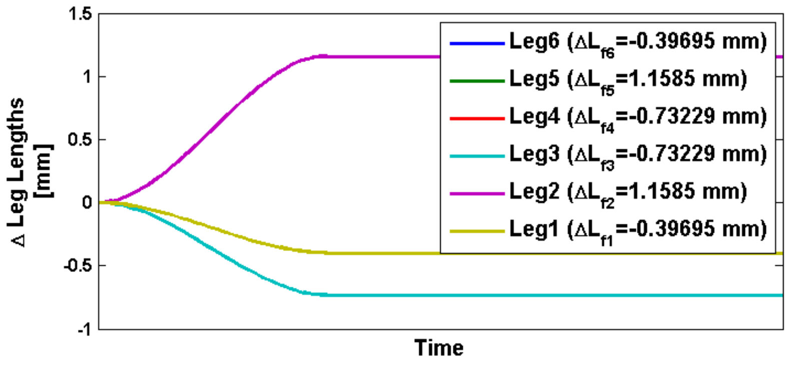

Figure 7.2 Changes in leg lengths when state $y \to 4$ mm from $(x_0, y_0, z_0, \theta_0, \phi_0, \psi_0) = (0,0,0,0,0,0)$

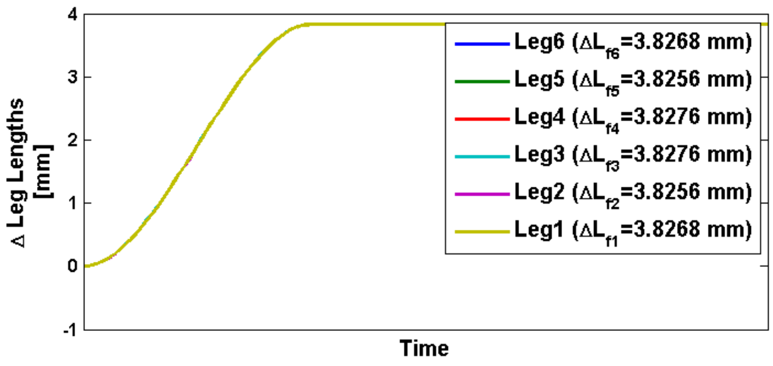

Figure 7.3 Changes in leg lengths when state $z \rightarrow 4$ mm from $(x_0, y_0, z_0, \theta_0, \phi_0, \psi_0) = (0,0,0,0,0,0)$

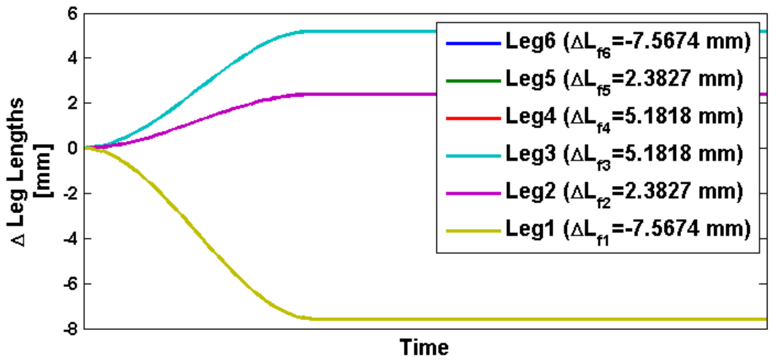

Figure 7.4 Changes in leg lengths when state $\theta \rightarrow 1.5^\circ$ from $(x_0, y_0, z_0, \theta_0, \phi_0, \psi_0) = (0,0,0,0,0,0)$

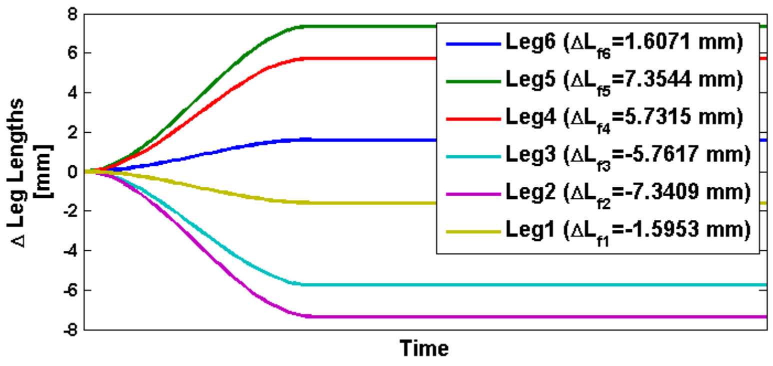

Figure 7.5 Changes in leg lengths when state $\phi \rightarrow 1.5^\circ$ from $(x_0, y_0, z_0, \theta_0, \phi_0, \psi_0) = (0,0,0,0,0,0)$

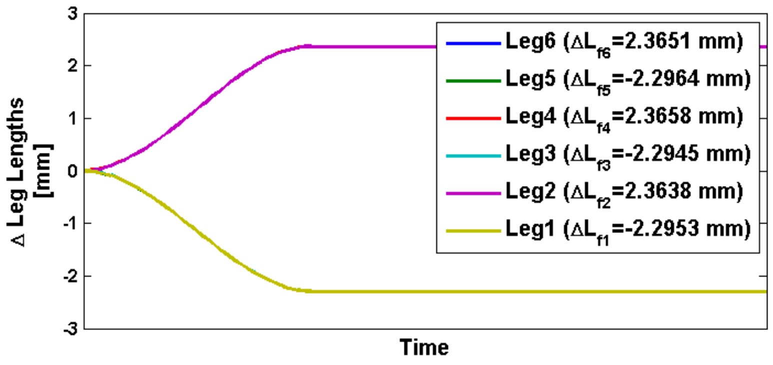

Figure 7.6 Changes in leg lengths when state $\psi \rightarrow 1.5^\circ$ from $(x_0, y_0, z_0, \theta_0, \phi_0, \psi_0) = (0,0,0,0,0,0)$

Motor Type	McLennan Type 23HS 309 halfstep, 2-phase, bi-polar, parallel stepper motor
Holding torque	80 Ncm
Step angle (full step drive)	1.8° or 200 steps/rev
Driving current	2 A
No. of μsteps/rev	4000 (w/ MCS-8, 20 μsteps/step)
Maximum velocity	180000 μsteps/sec (20 μsteps/step)
Gear box ratio	25:1
Resolution per μstep	5 nm (w/ MCS-8, 20 μsteps/step)
Maximum movement per actuator	95 mm
Maximum holding load (power off)	200 kg
Connector	12-way Trim Trio

Table 7.2 Linear Actuators

Readhead type	Renishaw - RGH24Y
Readhead resolution	0.1 μm
Scale type	RGS20-S
Precision reference type	RGM245S
Connector	15-way sub D

Table 7.3 Encoder Data

Switch type	Type V4 Roller
Max supply voltage	250 VAC (28 VDC)
Max switching current	10A AC (non-inductive)(5A DC)
Switching funcion	Normally closed
Connector	12-way Trim Trio

Table 7.4 Limit Switches

7.3 Kinematics in Microcode Format

```
'************************************************************************
'Kinematic Equations for FMB Oxfords Stewarts Platform (M3)
'By: Christer Engblom, 2013-01-17
'************************************************************************
' This file contains the forward and reverse kinematics for the FMB
' Oxford Stewarts Platform of type M3.
' Note that the file only contains the kinematics and nothing else.
'
' In this file:
'run=1 ==> Runs Reverse Kinematics once
'run=2 ==> Runs Forward Kinematics once
'
' The user sets the virtual axes coordinates with Vcoord[x]:
'X Axis - Vcoord[0], mm (Range: +/- 10mm)
'Y Axis - Vcoord[1], mm (Range: +/- 5mm)
'Z Axis - Vcoord[2], mm (Range: +/- 40mm)
'A Axis, Pitch - Vcoord[3], milliDegrees (Range: +/- 2000 milliDegrees)
'B Axis, Roll  - Vcoord[4], milliDegrees (Range: +/- 2000 milliDegrees)
'C Axis, Yaw   - Vcoord[5], milliDegrees (Range: +/- 2000 milliDegrees)
```

```
'
' Running the reverse kinematics will use the values in Vcoord[x] to
' calculate the leg-length distance from the home position:
'delta_l1 - LegL[0], counts (10000 counts/mm)
'delta_l2 - LegL[1], counts (10000 counts/mm)
'delta_l3 - LegL[2], counts (10000 counts/mm)
'delta_l4 - LegL[3], counts (10000 counts/mm)
'delta_l5 - LegL[4], counts (10000 counts/mm)
'delta_l6 - LegL[5], counts (10000 counts/mm)
'
' Running the forward kinematics will use the values in LegL[x] to
' calculate the virtual axes coordinates:
'X Axis - xFor0, mm (Precision: <10^-3 mm)
'Y Axis - yFor0, mm (Precision: <10^-3 mm)
'Z Axis - zFor0, mm (Precision: <10^-3 mm)
'A Axis, Pitch - aFor0, milliDegrees (Precision: ~10^-2 mDegrees)
'B Axis, Roll  - bFor0, milliDegrees (Precision: ~10^-2 mDegrees)
'C Axis, Yaw   - cFor0, milliDegrees (Precision: ~10^-2 mDegrees)
' The variable 'thresh2' sets the precision of the forward kinematics. A
' lower number on this variable increases the precision (has to be positive).
'**********************************************************************
#AUTO
DA*[];'
DM Vcoord[6];'
'0: X position (in millimeter)
'1: Y position (in millimeter)
'2: Z position (in millimeter)
'3: Yaw position (in millidegrees)
'4: Roll position (in millidegrees)
'5: Pitch position (in millidegrees)
DM LegL[6];' current commanded leg lengths in counts (relative home position)
DM LegL2[6];' current commanded leg lengths in counts (relative home position)
DM JointP[36];'
'0-17: base joints (mm)
'3*k+0, x coordinates
'3*k+1, y coordinates
'3*k+2, z coordinates
'18-35: plate joints (mm)
'3*k+18+0, x coordinates
'3*k+18+1, y coordinates
'3*k+18+2, z coordinates
'where k=0,1,2,3,4,5.
DM Rot[45];'
'0-8: Rotation matrix (3x3) for reverse kin - million times its real value
'9-17: Rotation matrix (3x3) for forward kin - million times its real value
'18-26: partial derivative (3x3) for forward kin - million times its real value
'27-35: partial derivative (3x3) for forward kin - million times its real value
'36-44: partial derivative (3x3) for forward kin - million times its real value
DM KinFv[72];' variables that forward kinematics will use
'0-2: Xs1, Ys1, Zs1 - 10000 times its real value
'3-5: Xs2, Ys2, Zs2 - 10000 times its real value
'6-8: Xs3, Ys3, Zs3 - 10000 times its real value
'9-11: Xs4, Ys4, Zs4 - 10000 times its real value
'12-14: Xs5, Ys5, Zs5 - 10000 times its real value
'15-17: Xs6, Ys6, Zs6 - 10000 times its real value
'18-23: l1,l2,l3,l4,l5,l6 - 10000 times its real value
'24-29: deltal1, deltal2, deltal3, deltal4, deltal5, deltal6  - 10000 times its real value
'30-65: Jacobian matrix  - 10000 times its real value
'66-71: delta_a, delta_b, delta_c, delta_x, delta_y, delta_z  - 1000000 times its real value
DM Trig[12];'
'0-2: sin(a), sin(b), sin(c) - REVERSE KINEMATICS
'3-5: cos(a), cos(b), cos(c) - REVERSE KINEMATICS
'6-8: sin(a), sin(b), sin(c) - FORWARD KINEMATICS
'9-11: cos(a), cos(b), cos(c) - FORWARD KINEMATICS
run=0;'
Vcoord[3]=0;Vcoord[4]=0;Vcoord[5]=0;'
Vcoord[0]=0;Vcoord[1]=0;Vcoord[2]=0;'
JS#INIT;'
'
#MLOOP
JS#KINR,run=1;'
JS#KINF,run=2;'
JP#MLOOP;'
```

```
'
EN
'
#INIT
' Setting base joints (mm)
JointP[0]=63.95;JointP[1]=-301.93;JointP[2]=0;' Leg 1
JointP[3]=293.45;JointP[4]=95.58;JointP[5]=0;' Leg 2
JointP[6]=229.5;JointP[7]=206.34;JointP[8]=0;' Leg 3
JointP[9]=-229.5;JointP[10]=206.34;JointP[11]=0;' Leg 4
JointP[12]=-293.45;JointP[13]=95.58;JointP[14]=0;' Leg 5
JointP[15]=-63.95;JointP[16]=-301.93;JointP[17]=0;' Leg 6
'
' Setting plate joints (mm)
JointP[18]=270;JointP[19]=-225.17;JointP[20]=0;' Leg 1
JointP[21]=330;JointP[22]=-121.24;JointP[23]=0;' Leg 2
JointP[24]=60;JointP[25]=346.41;JointP[26]=0;' Leg 3
JointP[27]=-60;JointP[28]=346.41;JointP[29]=0;' Leg 4
JointP[30]=-330;JointP[31]=-121.24;JointP[32]=0;' Leg 5
JointP[33]=-270;JointP[34]=-225.17;JointP[35]=0;' Leg 6
'
lenZ=754;' Leg length (mm) at zero motor pos
height=721.226;' height of PCS over BCS (mm)
lgScal=10000;' amount of cnts per millimeter
thresh1=0.005;' for forward kinematics.
thresh2=1;' for forward kinematics: lower value increases precision.
'
' Setting variables to zero before use.
LegL[0]=0;LegL[1]=0;LegL[2]=0;'
LegL[3]=0;LegL[4]=0;LegL[5]=0;'
aFor0=0;bFor0=0;cFor0=0;xFor0=0;yFor0=0;zFor0=0;'
Rot[24]=0;Rot[25]=0;Rot[26]=0;'
Rot[36]=0;Rot[39]=0;Rot[42]=0;'
EN
'
' ============= Forward Kinematics =============
#KINF
' Storing delta-leg length values and then calculating leg lengths.
LegL2[0]=LegL[0];LegL2[1]=LegL[1];LegL2[2]=LegL[2];'
LegL2[3]=LegL[3];LegL2[4]=LegL[4];LegL2[5]=LegL[5];'
l1=lgScal*lenZ+LegL2[0];l2=lgScal*lenZ+LegL2[1];l3=lgScal*lenZ+LegL2[2];'
l4=lgScal*lenZ+LegL2[3];l5=lgScal*lenZ+LegL2[4];l6=lgScal*lenZ+LegL2[5];'
'
' If average leg-length is below threshold value:
'   set initial xFor,yFor,zFor,aFor,bFor,cFor coord to zero
' If not: set initial x,y,z,a,b,c coord to previous value
IF(@ABS[1-((l1+l2+l3+l4+l5+l6)/60000/lenZ)]*100000<thresh1);'
aFor=0;bFor=0;cFor=0;xFor=0;yFor=0;zFor=0;'
ELSE;aFor=aFor0;bFor=bFor0;cFor=cFor0;xFor=xFor0;yFor=yFor0;zFor=zFor0;ENDIF;'
notDone=1;iter=1;'
'
#KINFL
' Calculating cos(.), sin(.) and storing them.
JS#SINF(aFor);Trig[6]=_JS;' sin(a)
JS#SINF(bFor);Trig[7]=_JS;' sin(b)
JS#SINF(cFor);Trig[8]=_JS;' sin(c)
JS#COSF(aFor);Trig[9]=_JS;' cos(a)
JS#COSF(bFor);Trig[10]=_JS;' cos(b)
JS#COSF(cFor);Trig[11]=_JS;' cos(c)
'
' Building rotation matrix.
Rot[9]=Trig[9]*Trig[10];'
Rot[10]=(Trig[9]*Trig[7]*Trig[8]/1000)-(Trig[6]*Trig[11]);'
Rot[11]=(Trig[9]*Trig[7]*Trig[11]/1000)+(Trig[6]*Trig[8]);'
Rot[12]=Trig[6]*Trig[10];'
Rot[13]=(Trig[6]*Trig[7]*Trig[8]/1000)+(Trig[9]*Trig[11]);'
Rot[14]=(Trig[6]*Trig[7]*Trig[11]/1000)-(Trig[9]*Trig[8]);'
Rot[15]=-Trig[7]*1000;'
Rot[16]=Trig[10]*Trig[8];'
Rot[17]=Trig[10]*Trig[11];'
'
' This section does 3 things:
' 1. Calculate leg coordinates and leg lengths from variables xFor,yFor,zFor,aFor,bFor,cFor
' 2. Calculate differences between actual leg lengths and leg lengths calculated (1)
' 3. Use values calculated from (2) in comparison to threshold (thres2) to decide if
'    current values of xFor,yFor,zFor,aFor,bFor,cFor are precise enough.
iKf=0;tmpF2=0;'
#KINFL1
kKf=0;sKf=0;iKf18=iKf+18;iKf24=iKf+24;'
#KINFL2
mKf=0;tmpF=0;indxAf=9+(3*kKf);indxBf=18+(3*iKf);'
#KINFL3
tmpF=tmpF+(Rot[indxAf]*JointP[indxBf]);' Rotation matrix multiplied by plate-joint coord
indxAf=indxAf+1;indxBf=indxBf+1;'
mKf=mKf+1;'
JP#KINFL3,mKf<3;'
indxBf=(3*iKf)+kKf;'
IF(kKf=0);tmpF=(tmpF/100)+(10000*(xFor-JointP[indxBf]));ELSE;'x coordinate of leg
IF(kKf=1);tmpF=(tmpF/100)+(10000*(yFor-JointP[indxBf]));ELSE;' y coordinate of leg
tmpF=(tmpF/100)+(10000*(zFor-JointP[indxBf]+height));ENDIF;ENDIF;' z coordinate of leg
KinFv[indxBf]=tmpF;' Storing leg length coordinates
sKf=sKf+((tmpF/1000)*(tmpF/1000));' calculating: (x^2+y^2+z^2)
kKf=kKf+1;'
JP#KINFL2,kKf<3;'
KinFv[iKf18]=1000*@SQR[sKf];' leg lengths
KinFv[iKf24]=KinFv[iKf18]-LegL2[iKf]-(lenZ*10000);' Calculating difference in leg-lengths
IF(@ABS[KinFv[iKf24]]>=@SQR[thresh2]);tmpF2=thresh2;ENDIF;'
IF(tmpF2<thresh2);'
tmpF2=tmpF2+(KinFv[iKf24]*KinFv[iKf24]);' Calculating precision in this iteration.
ENDIF;'
iKf=iKf+1;'
JP#KINFL1,(iKf<6);'
'
' The following section runs if precision isnt good enough for this iteration.
IF(tmpF2<thresh2);notDone=0;ENDIF;'
IF(notDone=1);'
' Calculating partial derivatives of rotation matrix.
Rot[18]=-Trig[6]*Trig[10];'
Rot[19]=-(Trig[6]*Trig[7]*Trig[8]/1000)-(Trig[9]*Trig[11]);'
Rot[20]=-(Trig[6]*Trig[7]*Trig[11]/1000)+(Trig[9]*Trig[8]);'
Rot[21]=Trig[9]*Trig[10];'
Rot[22]=(Trig[9]*Trig[7]*Trig[8]/1000)-(Trig[6]*Trig[11]);'
Rot[23]=(Trig[9]*Trig[7]*Trig[11]/1000)+(Trig[6]*Trig[8]);'
'
Rot[27]=-Trig[9]*Trig[7];'
Rot[28]=Trig[9]*Trig[10]*Trig[8]/1000;'
Rot[29]=Trig[9]*Trig[10]*Trig[11]/1000;'
Rot[30]=-Trig[6]*Trig[7];'
Rot[31]=Trig[6]*Trig[10]*Trig[8]/1000;'
Rot[32]=Trig[6]*Trig[10]*Trig[11]/1000;'
Rot[33]=-Trig[10]*1000;'
Rot[34]=-Trig[7]*Trig[8];'
Rot[35]=-Trig[7]*Trig[11];'
'
Rot[37]=(Trig[9]*Trig[7]*Trig[11]/1000)+(Trig[6]*Trig[8]);'
Rot[38]=-(Trig[9]*Trig[7]*Trig[8]/1000)+(Trig[6]*Trig[11]);'
Rot[40]=(Trig[6]*Trig[7]*Trig[11]/1000)-(Trig[9]*Trig[8]);'
Rot[41]=-(Trig[6]*Trig[7]*Trig[8]/1000)-(Trig[9]*Trig[11]);'
Rot[43]=Trig[10]*Trig[11];'
Rot[44]=-Trig[10]*Trig[8];'
'
' Building Jacobian Matrix
iKf=0;'
#KINFL4
kKf=0;iKf18=iKf+18;iKfx3=iKf*3;'
#KINFL5
IF(kKf<3);'
indxAf=18+(9*kKf);indxBf=18+iKfx3;indxCf=iKfx3;'
tmpF=JointP[indxBf]*Rot[indxAf];'
indxAf=indxAf+1;indxBf=indxBf+1;'
tmpF=tmpF+(JointP[indxBf]*Rot[indxAf]);'
indxAf=indxAf+1;indxBf=indxBf+1;'
tmpF=tmpF+(JointP[indxBf]*Rot[indxAf]);'
'
tmpF2=(tmpF/1000)*(KinFv[indxCf]/10000);'
indxCf=indxCf+1;'
'
```

```
indxAf=indxAf+1;indxBf=18+iKfx3;'
tmpF=JointP[indxBf]*Rot[indxAf];'
indxAf=indxAf+1;indxBf=indxBf+1;'
tmpF=tmpF+(JointP[indxBf]*Rot[indxAf]);'
indxAf=indxAf+1;indxBf=indxBf+1;'
tmpF=tmpF+(JointP[indxBf]*Rot[indxAf]);'
'
tmpF2=tmpF2+((tmpF/1000)*(KinFv[indxCf]/10000));'
indxCf=indxCf+1;'
'
indxAf=indxAf+1;indxBf=18+iKfx3;'
tmpF=JointP[indxBf]*Rot[indxAf];'
indxAf=indxAf+1;indxBf=indxBf+1;'
tmpF=tmpF+(JointP[indxBf]*Rot[indxAf]);'
indxAf=indxAf+1;indxBf=indxBf+1;'
tmpF=tmpF+(JointP[indxBf]*Rot[indxAf]);'
'
tmpF2=tmpF2+((tmpF/1000)*(KinFv[indxCf]/10000));'
'
indxAf=30+(6*iKf)+kKf;'
KinFv[indxAf]=(tmpF2/(KinFv[iKf18]/10000))*10;'
ELSE;'
indxAf=33+(6*iKf);indxBf=iKfx3;'
KinFv[indxAf]=(KinFv[indxBf]/KinFv[iKf18])*10000;'
indxAf=indxAf+1;indxBf=indxBf+1;'
KinFv[indxAf]=(KinFv[indxBf]/KinFv[iKf18])*10000;'
indxAf=indxAf+1;indxBf=indxBf+1;'
KinFv[indxAf]=(KinFv[indxBf]/KinFv[iKf18])*10000;'
ENDIF;'
kKf=kKf+1;'
JP#KINFL5,kKf<4;'
iKf=iKf+1;'
JP#KINFL4,iKf<6;'
'
' This subsection does two things:
' 1. Gaussian elimination on Jacobian Matrix
' 2. Using factors while computing (1) to further 'adjust' differences
'    between actual leg-lengths and leg-lengths calculated from xFor,yFor,zFor,aFor,bFor,cFor.
iKf=1;'
#KINFL6
kKf=iKf+1;'
#KINFL7
indxAf=29+((kKf-1)*6)+iKf;'
indxBf=29+((iKf-1)*6)+iKf;'
tmpF=(KinFv[indxAf]/KinFv[indxBf])*10000;' Calculating factor used in (1) and (2).
KinFv[indxAf]=0;' set cell to zero.
mKf=iKf+1;'
#KINFL8
indxAf=29+((kKf-1)*6)+mKf;indxBf=29+((iKf-1)*6)+mKf;'
KinFv[indxAf]=KinFv[indxAf]-((tmpF/10000)*KinFv[indxBf]);' Gaussian elim.
mKf=mKf+1;'
JP#KINFL8,mKf<=6;'
indxAf=23+kKf;indxBf=23+iKf;'
KinFv[indxAf]=KinFv[indxAf]-((tmpF/10000)*KinFv[indxBf]);' adjusting delta-leg lengths
kKf=kKf+1;'
JP#KINFL7,kKf<=6;'
iKf=iKf+1;'
JP#KINFL6,iKf<6;'
'
' Calculating how much xFor,yFor,zFor,aFor,bFor,cFor should differ in next iteration.
iKf=6;'
#KINFL9
indxAf=23+iKf;'
tmpF=KinFv[indxAf];'
kKf=6;'
IF(kKf>iKf);'
#KINFL0
indxAf=29+((iKf-1)*6)+kKf;indxBf=65+kKf;'
tmpF=tmpF-(KinFv[indxAf]*(KinFv[indxBf]/1000000));'
kKf=kKf-1;'
JP#KINFL0,kKf>iKf;'
ENDIF;'
indxAf=29+((iKf-1)*6)+kKf;indxBf=65+iKf;'
```

```
KinFv[indxBf]=(tmpF*100)/(KinFv[indxAf]/10000);'
iKf=iKf-1;'
JP#KINFL9,iKf>0;'
'
' Calculating new values of xFor,yFor,zFor,aFor,bFor,cFor to be tried in next iteration
aFor=aFor-(KinFv[66]*57.2958/1000);'
bFor=bFor-(KinFv[67]*57.2958/1000);'
cFor=cFor-(KinFv[68]*57.2958/1000);'
xFor=xFor-(KinFv[69]/1000000);'
yFor=yFor-(KinFv[70]/1000000);'
zFor=zFor-(KinFv[71]/1000000);'
ENDIF;'
aFor0=aFor;bFor0=bFor;cFor0=cFor;'
xFor0=xFor;yFor0=yFor;zFor0=zFor;'
iter=iter+1;'
JP#KINFL,(notDone=1)&(iter<100);' cont until prec is good enough (up to max 100 iter)
run=0;'
CFA;MG "___FORWARD_CALC___";'
CFA;MG "Calculated with",iter," iterations.";'
CFA;MG "x=",xFor0," ,y=",yFor0," ,z=",zFor0;'
CFA;MG "a=",aFor0," ,b=",bFor0," ,c=",cFor0;'
EN
'
' ============== Reverse Kinematics ==============
#KINR
' Calculating cos(.) and sin(.) and storing them.
JS#SINR(Vcoord[3]);Trig[0]=_JS;' sin(a)
JS#SINR(Vcoord[4]);Trig[1]=_JS;' sin(b)
JS#SINR(Vcoord[5]);Trig[2]=_JS;' sin(c)
JS#COSR(Vcoord[3]);Trig[3]=_JS;' cos(a)
JS#COSR(Vcoord[4]);Trig[4]=_JS;' cos(b)
JS#COSR(Vcoord[5]);Trig[5]=_JS;' cos(c)
'
' Building rotation matrix.
Rot[0]=Trig[3]*Trig[4];'
Rot[1]=(Trig[3]*Trig[1]*Trig[2]/1000)-(Trig[0]*Trig[5]);'
Rot[2]=(Trig[3]*Trig[1]*Trig[5]/1000)+(Trig[0]*Trig[2]);'
Rot[3]=Trig[0]*Trig[4];'
Rot[4]=(Trig[0]*Trig[1]*Trig[2]/1000)+(Trig[3]*Trig[5]);'
Rot[5]=(Trig[0]*Trig[1]*Trig[5]/1000)-(Trig[3]*Trig[2]);'
Rot[6]=-Trig[1]*1000;'
Rot[7]=Trig[4]*Trig[2];'
Rot[8]=Trig[4]*Trig[5];'
'
' Function to calculate all 6 leg lengths.
' This is done by first calculating the leg coordinates
' and then the vector length they span up.
iKr=0;'
#KINRL;'
kKr=0;sKr=0;'
#KNRSL1
mKr=0;tmpR=0;indexA=3*kKr;indexB=18+(3*iKr);'
#KNRSL2
tmpR=tmpR+(Rot[indexA]*JointP[indexB]);' Rotation matrix multiplied by plate-joint coord
indexA=indexA+1;indexB=indexB+1;'
mKr=mKr+1;'
JP#KNRSL2,mKr<3;'
indexB=(3*iKr)+kKr;'
IF(kKr<2);tmpR=(tmpR/100)+(10000*(Vcoord[kKr]-JointP[indexB]));' x and y coordinate of leg
ELSE;tmpR=(tmpR/100)+(10000*(Vcoord[kKr]-JointP[indexB]+height));ENDIF;' z coordinate of leg
sKr=sKr+((tmpR/1000)*(tmpR/1000));' calculating: (x^2+y^2+z^2)
kKr=kKr+1;'
JP#KNRSL1,kKr<3;'
LegL[iKr]=((lgScal/10)*@SQR[sKr])-(lgScal*lenZ);' Calc leg-length (in counts) from home pos.
iKr=iKr+1;'
JP#KINRL,(iKr<6);'
run=0;'
CFA;MG "___INVERSE_CALC___";'
CFA;MG "dl1=",LegL[0],", dl2=",LegL[1],", dl3=",LegL[2];'
CFA;MG "dl4=",LegL[3],", dl5=",LegL[4],", dl6=",LegL[5];'
EN
'
' ========================================================
' Subroutines called to calculate sin(.) and cos(.). This is done using Taylor Series.
' When angle is between +/- 2 degrees, max error is:
' sin(.) => ~10^-6 degrees, cos(.) => ~10^-10 degrees
'
' Called from Reverse Kinematics. Input: millidegrees, Output: 1000*sin(.) or 1000*cos(.)
#SINR
mradsr=^a*3141.5927/180000;'
^c=mradsr-(mradsr*mradsr*mradsr/6000000);'
EN,,^c
#COSR
mradcr=^a*3141.5927/180000;'
^c=1000-(mradcr*mradcr/2000)+(mradcr*mradcr/1000*mradcr*mradcr/24000000);'
EN,,^c
'
' Called from Forward Kinematics. Input: millidegrees, Output: 1000*sin(.) or 1000*cos(.)
#SINF
mradsf=^a*3141.5927/180000;'
^c=mradsf-(mradsf*mradsf*mradsf/6000000);'
EN,,^c
#COSF
mradcf=^a*3141.5927/180000;'
^c=1000-(mradcf*mradcf/2000)+(mradcf*mradcf/1000*mradcf*mradcf/24000000);'
EN,,^c
```

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER´S THESIS *Date of issue* June 2014 *Document Number* ISRN LUTFD2/TFRT--5946--SE
Author(s) Christer Engblom	*Supervisor* Mirjam Lindberg, Max-lab Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden Rolf Johansson, Dept. of Automatic Control, Lund University, Sweden (examiner) *Sponsoring organization*

Title and subtitle

Evaluation of Galil DMC-4080 as a Controller of Stewart Platform

Abstract

Evaluation of the Galil DMC–4080 as a closed-loop controller of a stepper-motor driven Stewart Platform with mm- and mrad- precision and resolution. The project involves a full implementation and optimization of kinematic equations, implementing modes of motion & controller structure, as well the use of MATLAB simulations. The 4080 proves successful in static positioning but have limitations in timeperformances with the kinematics as well as smoothness of motion due to error handling.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 1-62	*Recipient's notes*
Security classification		

http://www.control.lth.se/publications/