Blog Post

Ryan Carter,
William Ellett,
England Kwok,
Robert Smith,
Ambar Srivastava

June 1, 2019

Contents

1 Introduction **1**

2 Robot and Task Integration **1**
2.1 Motor and Sensor Selection . . . 1
2.2 System Architecture . . . 1
2.2.1 Motion Planning Structure . . . 1
2.2.2 Communication . . . 2
2.2.3 Embedded Software . . . 2
2.3 Calibration . . . 3
2.4 Automation . . . 3
2.4.1 Motion Planning . . . 3
2.4.2 Robot Automation . . . 4

3 Discussion **5**
3.1 Design of Manipulator . . . 5
3.1.1 Design Goals . . . 5
3.1.2 Base . . . 6
3.1.3 Arm . . . 6
3.1.4 End Effector . . . 6
3.1.5 Base Plane . . . 7
3.1.6 Embedded Components . . . 7
3.2 Results . . . 7
3.2.1 Full Jenga Tower Build Time . . . 7
3.2.2 Control Parameter Tuning . . . 7
3.3 Possible Improvements . . . 9

4 Conclusion **10**

1 Introduction

The goal of this project was to design, build and program a robotic manipulator to automate stacking a Jenga tower. The essential requirements of the task were for it to be capable of picking up Jenga blocks from a fixed loading bay, and then placing them to produce a stacked Jenga tower. Additionally, it was essential for the manipulator to have a user interface in which the loading bay location and stack location could be changed.

This task required at minimum a five degree of freedom system in order to stack a tower. Three translation degrees of freedom (x, y, z), one rotational degree of freedom (θ_z in frame 0) and an actuation degree of freedom for gripping the block. To achieve these task space requirements, the team developed the concept for a robotic arm. This robotic arm has six motors, and hence its position is described by six generalised coordinates. Four of these generalised coordinates are rotational degrees of freedom: q_1, q_2, q_3 and q_5. q_4 is a constrained rotational coordinate. This constraint was derived from the task space definition of the Jenga block, which excludes θ_x and θ_y rotations in frame 0; in other words, the block always remains horizontal. q_6 defines the state of the gripper.

Speed and accuracy were the two major metrics by which the team defined performance. This report will discuss hardware selection, motion planning, control and system architecture, in particular the discussion will be centred around how these design elements impacted system performance. A discussion on results and potential improvements is also included.

2 Robot and Task Integration

2.1 Motor and Sensor Selection

The selection of motors and sensors had an impact on both speed and accuracy. The motors selected were Dynamixel XL430 and XL320 servomotors with in built motor control units (MCUs). The XL430's were used to meet the higher torque requirements of the main arm, with smaller XL320's on the end effector to reduce end effector mass. Although more expensive than many other available servos, the MCUs provide the advantage of outputting position and velocity feedback. Additionally, the MCUs have position control and velocity control modes which allow more flexibility in control system implementation. The motor encoders had sufficiently high resolution (0.088°) to meet the teams position error requirements. Using forward kinematics, the maximum absolute error due to the encoder resolution was found to be 0.9mm.

2.2 System Architecture

The system architecture consists of two main subsystems: Matlab, running on a laptop or PC, and an OpenCM controller board. Matlab is connected to the OpenCM board via USB and they communicate with serial. The computing power of Matlab is used to create the motion plan and send the resulting polynomial coefficients to the OpenCM.

2.2.1 Motion Planning Structure

Trajectory generation software is built using an object-oriented structure in Matlab with several layers of inheritance. This allows for the generation of motion plans at a high level of abstraction, see section 2.4.1.

At the lowest level, trajectories are generated as piecewise cubic polynomials with an arbitrary number of degrees of freedom, and an arbitrary number of via points. This generalised approach allows for fast tuning of a motion plan by the ability to use any number of continuous via points, which may be easily added or removed. The robot trajectory class inherits from this trajectory generation class, constraining the trajectory with properties specific to the robot design. This class provides functionality to convert trajectory data into a form as described in section 2.2.2 for communication with the OpenCM controller board, as well as plotting commands which may be used for validation of trajectories and debugging.

All robot motion commands inherit from the robot trajectory class. This allows different commands and movement strategies to retain the same communication protocols and basic structure whilst changing the prescribed motion. Different motion strategies were able to be tested using this approach by simply changing the calculation methods used to determine via point locations and times.

2.2.2 Communication

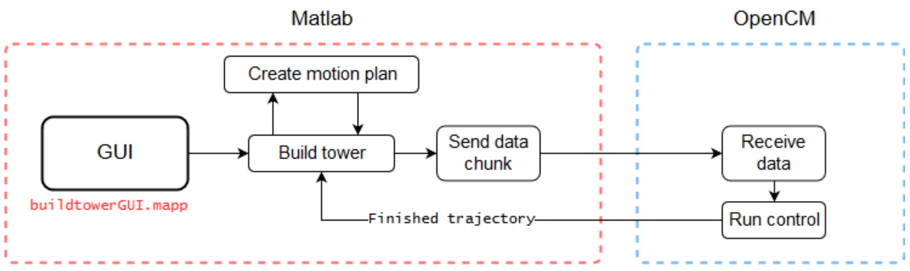

Figure 1: Overview of system architecture

Sending a single cubic polynomial requires sending it's 4 coefficients and time domain (t_0 and t_f). This is sent as an ASCII string with the format `a3 a2 a1 a0 t0 tf`. A large motion plan is sent in chunks, where a chunk is any number of polynomials less than the maximum constrained by memory on the OpenCM. This results in arrays of polynomial coefficients plus their time domains storing the trajectories for $x(t)$, $y(t)$, $z(t)$, $\theta(t)$ and grip(t) (see section **??** for detail on how these are used). The OpenCM will use these to generate the trajectory during run time for motor commands. The generation of these trajectories can be validated by having the OpenCM simulate operation without actuating the motors, instead sending the generated data over serial to Matlab for plotting and validation.

2.2.3 Embedded Software

The OpenCM board keeps an internal state machine to organise operation. This enables the system to remain running, and to run a variety of operations without needing to reboot. Typical operation has the system in `WAITING` until a command is sent using the Matlab GUI, which causes a state change to run the chosen command until it's complete and the state returns to `WAITING`. The states are summarised in table 1.

State name	Description
`WAITING`	Initial state, listens to serial for a command and changes state
`RECEIVING`	Receiving polynomial data: `a3 a2 a1 a0 t0 tf`
`PLOTTING`	Sending generated trajectories from stored polynomials over serial for plotting with Matlab
`RUN_CONTROL`	Run currently stored trajectories with position or velocity control
`READ`	With motors disabled continuously send measured joint angles
`CALIBRATE`	Like `READ`, but can also calculate error from a reference
`SEND_CURRENT`	Send current task space vector X, using measured joint angles and forward kinematics

Table 1: OpenCM states summary

2.3 Calibration

Calibration of the manipulator was critical in order to improve the end effector position accuracy. In order to achieve this, a regression analysis was completed on 350 data points, taking known end effector positions and using inverse kinematics to calculate expected q values. Comparing the calculated values to measured resulted in a distribution of calibration values, shown for q_3 in figure 2. The result of this analysis was a linear fit for each motors inbuilt encoder in the form:

$$q_{\text{calculated}} = \text{SCALE} \times q_{\text{measured}} + \text{OFFSET}$$

As can be seen in figure 2, the scale factor for q_3 is 0.9875, whilst the offset is +0.0112rad. The data gathered displayed a high correlation with the linear relation, with q_3 displaying $R^2 = 0.9976$, indicating that the assumed linear approximation is appropriate for this analysis.

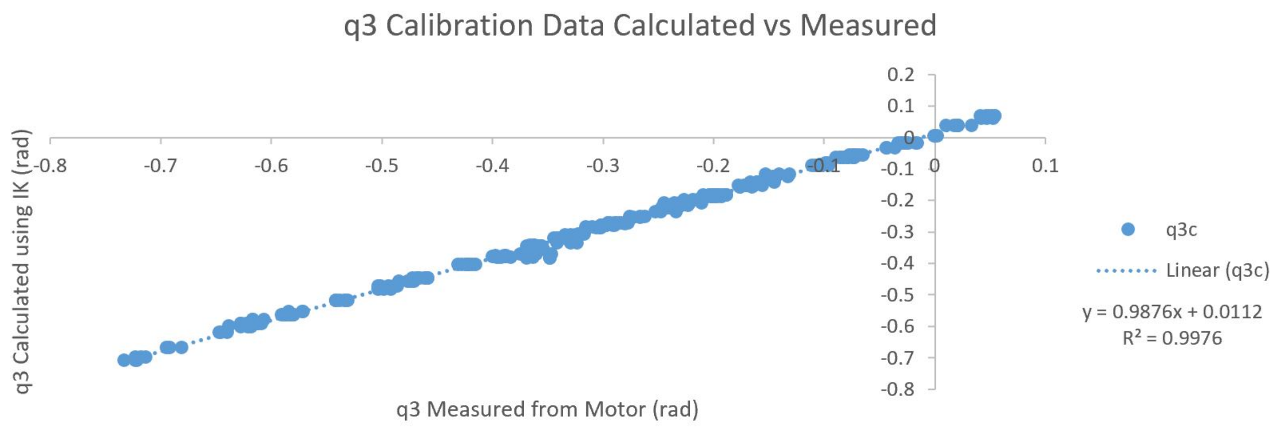

Figure 2: Calculated q_3 angle using inverse kinematics VS Calibration Data collected for q_3 actuator from 350 data points

2.4 Automation

2.4.1 Motion Planning

A class was developed for determining the build sequence for a Jenga tower from a given build location and orientation. The available build space is broken up into regions to determine the order and direction of block placement satisfying constraints on the end effector position and orientation. This ensures the generation of feasible build sequences within the robot's mechanical constraints, with a flexibility of tower location and orientation. Parametric via points are generated from every block in the tower build for positions to drop the block, approach the tower, and withdraw from the tower.

A motion plan can be constructed as a sequence of robot commands, each calculated as a task space trajectory to send to the robot, as detailed in section 2.2.1. Each movement command is calculated from a series of via points generated from the associated block and loading bay coordinates. Via point times in each command are calculated parametrically from start time and duration of the command. This parametric approach enables every trajectory to be generated automatically for any tower position and orientation in the build space, and allows for time-scaling to provide different speeds, prioritising minimum build time or accuracy of block placement.

2.4.2 Robot Automation

Both control architectures mentioned in section **??** have similar automation procedures which are described in listing 1 and 2.

```
# for every polynomial stored
while i < nPolys:
    # evaluate until reached end of current polys time domain
    t = currentTime()
    tf = polys[i].tf
    while t < tf:
        # reference task space position
        Xr = cubicEvaluate(polys[i], t)
        # reference joint space
        Qr = inverseKinematics(Xr)
        # send position commands to motors
        writeQToServos(Qr)
        t = currentTime()
    # finished this poly, move onto next
    i += 1
```

Listing 1: Position Control Pseudocode

```
    # for every polynomial stored
    while i < nPolys:
        # evaluate until reached end of current polys time domain
        t = currentTime()
        tf = polys[i].tf
        while t < tf:
            # reference task space position and velocity
            Xr = cubicEvaluate(polys[i], t)
            Xdr = quadEvaluate(polys[i], t)
            # reference joint space
            Qr = inverseKinematics(Xr)
            # measured joint and task space
            Qm = readQfromServos()
            Xm = forwardKinematics(Qm)
            # PD velocity control using error in task space
            Xdc = velocityControl(Xr, Xm)
            # convert task space velocity control to joint space velocity
            Qdc = inverseJacobian(Xdc, Qm)
            writeQdToServos(Qdc)
            t = currentTime()
        # finished this poly, move onto next
        i += 1
```

Listing 2: Velocity Control Psuedocode

3 Discussion

3.1 Design of Manipulator

3.1.1 Design Goals

In order to minimise the influence of mechanical components on the automation side of the project and adhere to the top level project goals to maximise both precision and accuracy and minimise build time, specific design goals were set for mechanical component design, those goals being:

- Minimise system mass & moment of inertia
- Maximise system rigidity (minimise compliance and slop)
- Minimise system friction
- Maximise system robustness to tolerance error

Minimising system mass was extremely important, as higher mass manipulators require higher motor torques in order to support the static weight of the system. Furthermore, the inertia inherent to moving systems is dependant on system mass, and directly influences the actuators ability to accelerate that mass in a controllable and desirable way. System rigidity is dependant on each components resistance to mechanical compliance, that being the deflection of each member under load, in addition to mechanical slop, relating to ill-constrained joints between system parts. Friction in rotating and sliding joints is extremely detrimental to system performance, as this increases the torque required to accelerate the system. In addition, it introduces issues with repeatability, wherein the system responds differently to actuator inputs due to inconsistent/varying friction within each joint (as a result of slight differences in slop position and compliance). Maximising robustness relates primarily to the systems ability to perform to an acceptable standard despite small differences in position/speed.

3.1.2 Base

The Base of the manipulator was designed to be tall and wide, utilising a 140mm slew bearing in order to facilitate the rotation of q_1, aiding in L_1 rigidity. 3D printing was utilised in order to ensure manufacture accuracy. A removable 3D printed electronics pod was implemented, to improve serviceability of all critical electronic components. This also provided reliability, by ensuring these components were safe from rough handling.

3.1.3 Arm

The manipulator arm was built using a parallelogram kinematic design. The links were constructed from carbon fibre tubes to minimise linkage masses and moment of inertia, and maximise link rigidity. Joints were manufactured from 3d printed resin parts using a poly-jet 3d printer to minimise manufacture tolerance error.

Whilst arm weight was relatively low, a spring assist was placed on the joint q_2 in order to reduce the static torque requirement on this joint. It was decided to utilise extension springs, as torsion springs proved costly due to the lack of stock components able to meet specifications. To ensure sufficient assisting torque, the static torque requirement on q_2 was calculated at positions bounding the usable travel of the manipulator. As can be seen in figure 3, those points are at the top and bottom of the tower at a 300mm and 200mm radius. Taking into consideration the changing moment arm length as q_2 rotates, and the lower mounting point of the spring, an available spring was selected. As can be seen in figure 3, a stiffness of 0.666N/mm (2x0.333N/mm springs in parallel) resulted in a close fit to the critical points. With this design, the majority of static load on the q_2 joint was countered by the spring, resulting in better utilisation of q_2 motor torque.

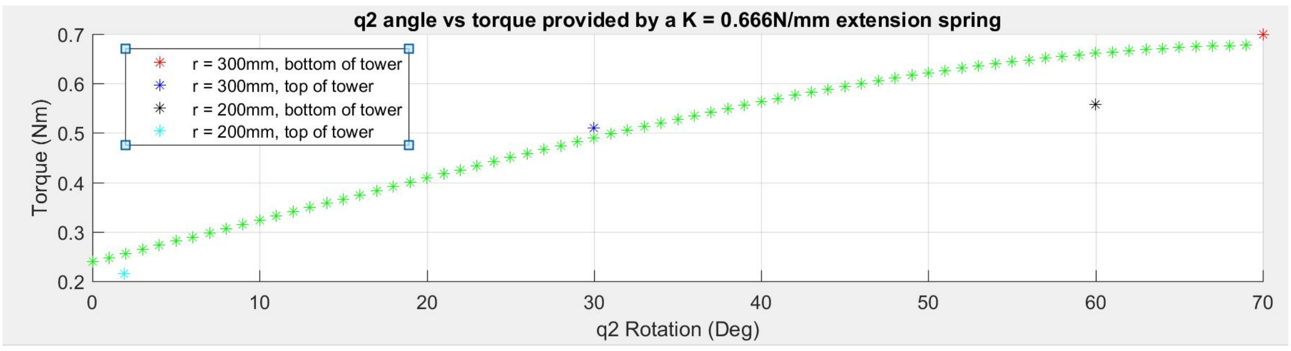

Figure 3: Spring Design for q_2 assist, displaying points at four extreme end effector positions

3.1.4 End Effector

The end effector was designed utilising a double rack and pinion mechanism, allowing linear movement of both gripper sides with the rotation of one actuator. These grippers were guided by rails and linear bearings in order to reduce sliding friction. Mass optimisation was carried out on the main end effector bracket, with the second iteration eliminating 40% of the initial end effector weight, resulting in less static torque applied to the q2 actuator. The grippers were designed to maximise system robustness. Mechanically limiting the gripper such that a block could not get stuck in an undesirable position, coupled with a top and back wall on the gripper meant that the manipulator could be pushed hard up against the block, making the system more robust to end effector errors in the pickup sequence. The addition of a laser pointer in the centre of the end effector allowed for easy position calibration, coupled with a Cartesian grid engraved in the base plane to improve the ease of actuator calibration.

3.1.5 Base Plane

The base plane was laser cut from MDF, with laser engraving of a Cartesian coordinate system. The laser engraving provided precise markings on the board to be used as an integrated calibration table. Line spacing for the calibration table was set at 12.5mm as this provided a good match for the 25mm x 75mm jenga blocks. Two in-set loading bays were implemented 3mm below the base plane surface, allowing each block to be pushed up against its bay walls ensuring accurate positioning in the end effector grippers.

3.1.6 Embedded Components

For the embedded system controller, an OpenCM9.04 board was used. This board was chosen for ease of integration, as it was supplied with connectors to interface with the Dynamixel motors via the TTL protocol. The high clock speed of 72Mhz was also considered as a beneficial factor over some other embedded controllers as this allowed a sufficiently high clock speed for fast motion control.

3.2 Results

3.2.1 Full Jenga Tower Build Time

The fastest full tower build achieved was 3:04min, shown in figure 4. Reaching this timing requires sweeping approximately 60° in 1s for each movement between the loading bay and tower. At these speeds, the base joint (q_1) overshoots noticeably on the blocks furthest from the loading bay that require the fastest q_1 movement. This effect decreases for each vertical layer as more time is allocated to allow for the increased height, thereby lowering the required q_1 velocity. Slower trajectories eliminate this problem, as seen in figure 5.

3.2.2 Control Parameter Tuning

Joint-space position control was tuned using a test trajectory that incorporated angular speeds around 40°/s, similar to what was used in the fast tower build mode. Pauses were used between movements to observe settling behaviour. The results are shown in figure 6, using the position control gains in table 2.

Motor/Control	P	I	D
XL430 Position	640	100	9000
XL430 Velocity	100	-	1000
XL320 Position	32	0	0

Table 2: Motor (joint-space) gains used

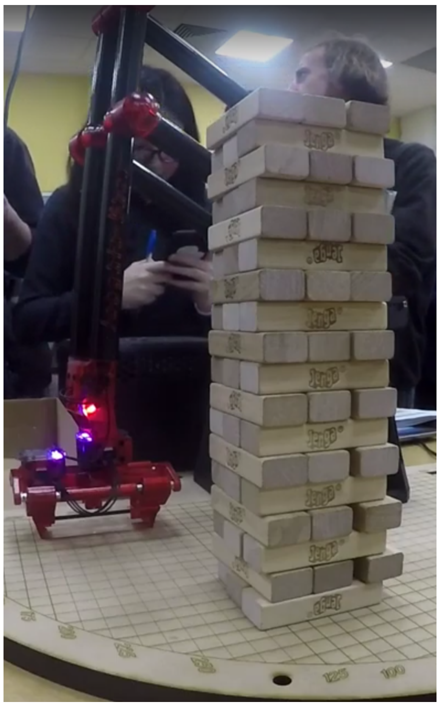

Figure 4: Fast build in 3:04min

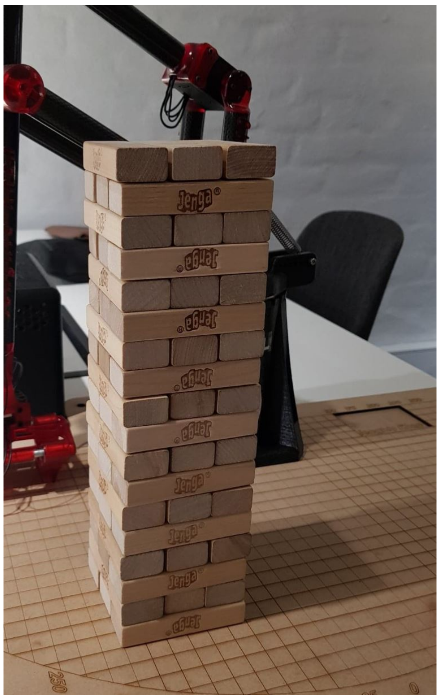

Figure 5: Accurate build in 6:00min

We were unable to tune task-space velocity control to even a minimum performance specification that would allow us to stack a tower at a reasonable speed. Example results in figure 7 show the issues we had with significant oscillation whilst testing a large range of PID values.

Axis	P	I	D
x	5	5	0
y	7	5	0
z	5	1	1

Table 3: Velocity control task-space PID gains

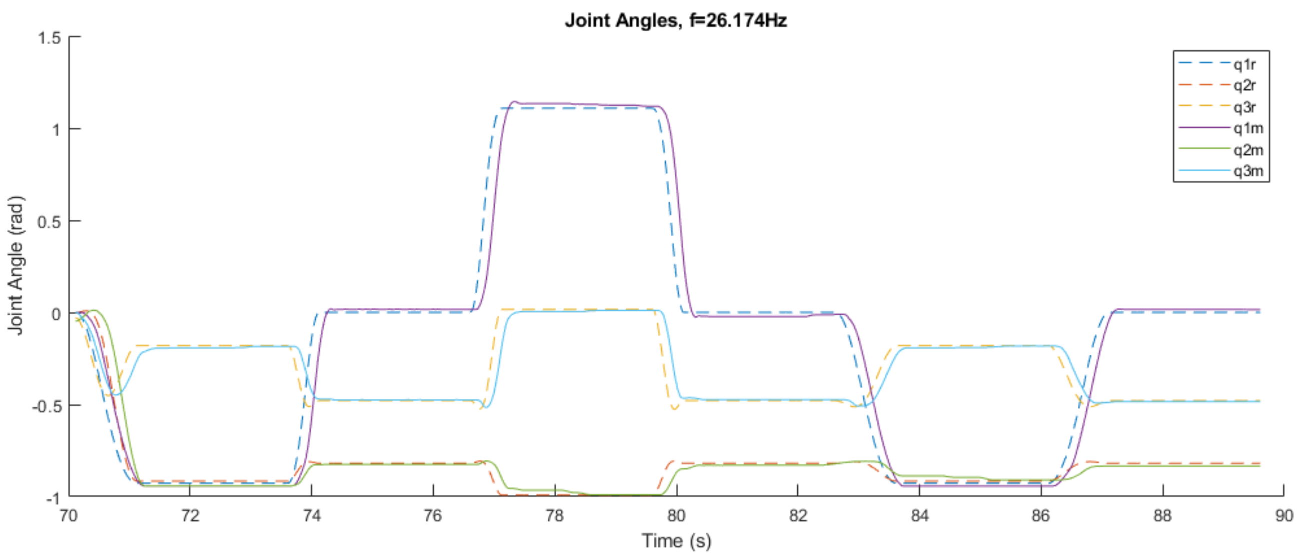

Figure 6: Position control on a test trajectory

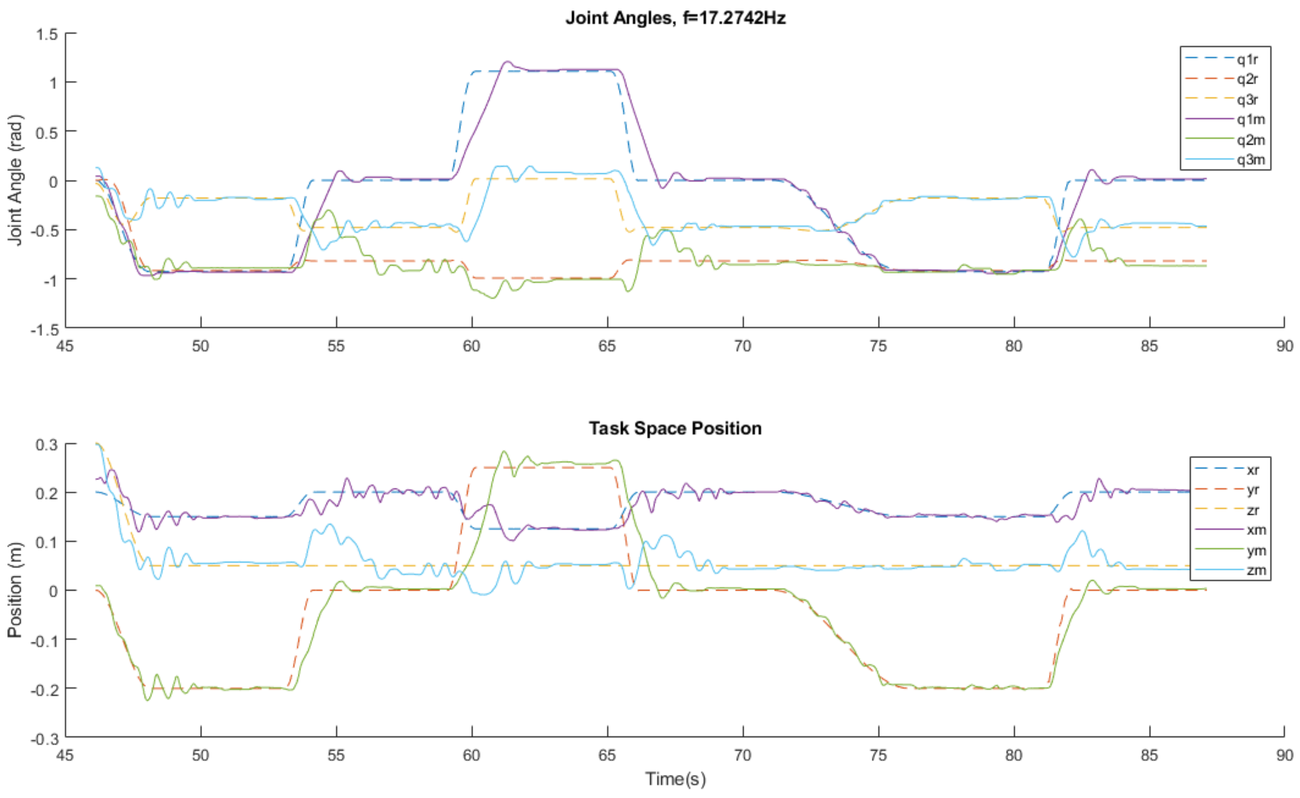

Figure 7: Velocity control on a test trajectory

3.3 Possible Improvements

While the results of this robot design have been in general quite good, a number of small short-comings have been identified as areas for possible improvement.

Robust constraint-based motion planning: although separating the available build space into regions provided a feasible build sequence for many scenarios; using a constraint based solver for motion planning could allow the range of the build space to be robustly extended, and enable path optimisation.

q_5 joint limits: the q_5 joint (end effector orientation) is mechanically constrained, such that it cannot achieve a full 360°of rotation. In several cases this requires the robot to build the tower from the opposite side (from the loading bay). Increasing the range of q_5 could reduce the constraints on the motion plan and allow for more optimal path planning. This advantage should be compared against the relative cost in mechanical design of extending the range of this joint.

Time optimisation with torque constraints: using torque constraints of each motor, the time of each trajectory segment may be optimised to use available motor torque. This could provide a more optimal time for every trajectory segment, improving the balance between speed and accuracy.

SD card to front-load serial communication delays: During the build tower sequence, the arm needs to pause at the loading before picking up the next block whilst the trajectory data is sent. Each pause with the motion planning implemented was an average of 0.4s, and occurs 54 times in a complete tower build. Using a SD card would allow all the trajectory data to be transmitted up front, but would mean a pause of at least 20s before operation would start.

Velocity control: Tuning of the PID controllers was unsuccessful due to the unclear starting point, large number of parameters and unknown system model. By finding a system model of the robot using fast Fourier transforms, PID controllers can be designed and simulated in Matlab or Simulink then fine-tuned on the robot. This could prove a more productive strategy.

4 Conclusion

All essential criteria of the task were met. The great manipulator Jenghis Khan can construct a full tower in a time of 3:04min. Matlab was used to complete the motion planning, which could build a tower at any location and orientation within the robots reachable work-space. The robustness of the motion plan strategy also allowed the user to choose the build time of the tower, depending on whether they preferred a fast or accurate build. While task-space velocity control was explored extensively, this was unable to achieve the same performance as joint space position control.