The DoodleBot

a project report presented
by
Anthony Evans and Rahul Victor
to
The Department of Electrical and Electronic Engineering

in partial fulfillment of the requirements
for the degree of
Master of Electrical Engineering

The University of Melbourne
Melbourne, Victoria
October 2013

Contents

1 Introduction

1.1 Abstract

The DoodleBot Project was a year long undertaking completed by Rahul Victor and Anthony Evans for the Engineering Capstone Project subject at the University of Melbourne. The produced device was a functional 2.1 axis CNC machine capable of autonomously reproducing bitmap images or photos approximated by B-splines with a pen. The device incorporates open-loop time-optimal control that attempts to complete the drawing accurately in minimum time. As an industry partner, NHP Electrical Engineering Products Pty Ltd proposed the project and provided a significant portion of the hardware for the project. This document will outline the project requirements, detail the engineered solution, justify the design and discuss performance and limitations.

1.2 Project Requirements

The DoodleBot project had two key stakeholders with their own independent requirements to address and satisfy.

1.2.1 The University of Melbourne, School of Engineering Requirements

As part of the Bachelor of Engineering (Electrical) and Master of Engineering (Electrical) courses, students are required to complete a year long subject called the Electrical Capstone Project. Students work on completing a major engineering project designed to develop and showcase practical engineering skills and theory. The expectations of the University were outlined in the available marking criteria and subject guidelines.

The Melbourne School of Engineering requires the following deliverables:

- A final technical report documenting the design, justification and performance of the project (this report)
- An oral presentation showcasing the design, justification and performance of the project
- A demonstration of the project at the annual Endeavour Exhibition to the general public and industry representatives

Assessment of these is based on the following criteria:

- Demonstration of technical skill and understanding of engineering concepts
- Understanding relevant literature in the field
- Quality of design and development
- Quality of implementation, experimentation/testing and results
- Quality of presentation (for each deliverable)

1.2.2 NHP Electrical Engineering Products Pty Ltd Requirements

The DoodleBot is an industry partnered project with NHP Electrical Engineering Products Pty Ltd (henceforth referred to as NHP). NHP's requirements for the project were agreed over several meetings and detailed in a Project Charter produced by the DoodleBot Team.

The agreed deliverables, to be handed over at the conclusion of the year, were:

- A functional device (including all hardware and software)
- Two promotional videos (one with a technical focus and one with a marketing focus) showcasing the features of the device

The specification of the product are:

- A CNC machine implemented on their specified hardware capable of 'drawing' specified programs on paper
- Capable of discrete 2 axis control
- Two state operation in third axis (on-off)
- Input incorporating image/photo recognition
- Software to be implemented on PLC and PC, with communication via a ethernet interface;

1.3 Project Scope

The responsibility of the DoodleBot team was to design, implement and construct the following components:

- Control problem formulation, solution design and software implementation. Understanding of relevant control concepts
- Design and software implementation of image/photo input process
- Design and software implementation of user interface
- PC-PLC Network Interface design and software implementation
- Z-axis design and construction
- Wiring loom construction and assembly of components

NHP's role in the project was to:

- Provide an assembled CNC frame with mechanical components for two axis movement;
- Provide a programmable logic controller (PLC), 2x stepper motors and 2x stepper motor PLC modules
- Provide adequate software and licenses for PLC programming

1.4 Performance Indicators

In addition to the requirements, the DoodleBot Team identified several key performance indicators to design for:

- How accurately the mechanical device is able to reproduce the desired input
- How fast the mechanical device is able to draw the desired input
- How well the system is able to create a bicolour representation of the original image

2

System Design

2.1 Overview

The Project Scope, detailed in the previous chapter, requires that the DoodleBot Team design and implement a system that can take an input in the form of a bitmap image and reproduce this image on paper using the provided hardware.

In addition to the requirements, the DoodleBot team designed the system to another key performance indicator - minimizing for the machine to draw.

2.1.0.1 User Interface

The main Graphical User Interface (GUI) is written in the Java programming language. This language was chosen for the simplicity and level of library support that it gives graphical interfaces, as well as the power that the language has to easily create servers and to interface with other programming languages. The desired result for this module was to allow for user input and manipulation of curve objects that would then be drawn by the printer at the users request. This was achieved through use of the Swing package, which comes with features that are ready made to allow for the drawing of different types of curves. The user directly manipulates the control points of a curve, with the shape of the curve updating in real time.

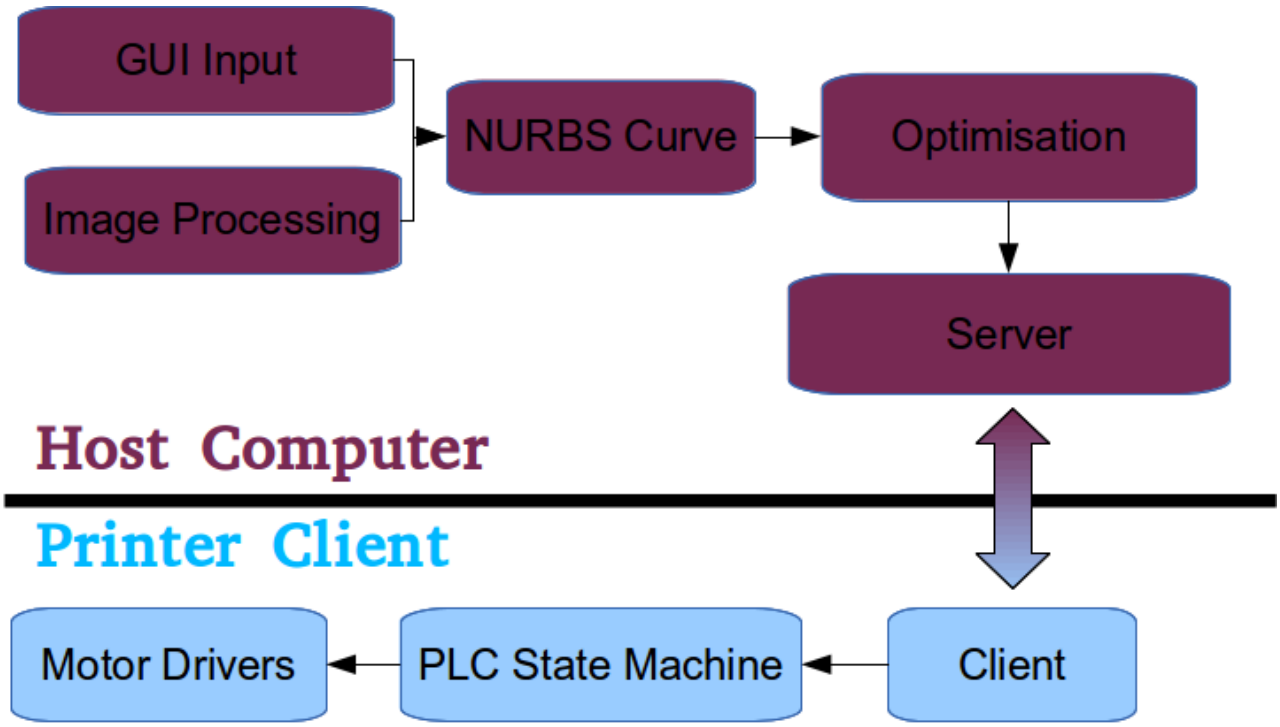

Figure 2.1.1

2.1.0.2 Image Processing

The image input method consists of a MATLAB® script that the Java GUI can call, which processes a given image and returns a series of URBS curves. These can then be displayed with the main GUI. The MATLAB® script performs this task by applying some initial image equalisation and filtering before running a Canny edge detector. The resulting output has all of the detected edges collated into 'edge objects' - line segments that do not split into two separate lines. These edge objects are then simplified by approximation by straight lines, before being fitted to a NURBS curve in a way that tries to minimise error.

2.1.0.3 URBS Curve

The key data structure that most modules on the Host utilise is a 3rd order Uniform Rational Basis Spline (URBS) curve. An URBS curve is a cousin of the better known NURBS curve, with the only difference being that the knot vector of the URBS curve has its knots uniformly spaced. An URBS curve is a method of representing an arbitrary curve through a space with a minimal set of data values. Curves can subsequently have any number of points on their surface calculated to arbitrary precision. There are a few major benefits to the restriction of input lines to 3rd order URBS curves which result in nice properties for our subsequent optimisation task; namely continuous velocity across the domain of the curve and uniform division of the path space resulting in an even division of optimisation between each pair of control points.

2.1.0.4 Optimisation

This module was solved with the application of a MATLAB script that programs the given URBS curve into a discrete structure that can be solved by SeDuMi. The script performs a change of variables in order to linearise the physical constraints of the system. The change of variables retains the convexity of the optimisation cost for which we wish to solve. The script also restricts the otherwise infinite dimensionality of the function we wish to solve for to a piecewise approximation. This means the result of our solution will be an approximation of the true and infinite dimensional optimum for our given cost. The problem we wish to solve is; given a path

$$\mathbf{q}(s) = \begin{bmatrix} x(s) \\ y(s) \end{bmatrix}$$

We would like to determine the maximal $\frac{ds(t)}{dt}$ for each point in time, subject to the actuator limitations of our plant. This maximal $\frac{ds(t)}{dt}$ will result in the traversal of the path $\mathbf{q}(s)$ in the minimum possible time. The key restriction that our model faces is the selection of the path velocity $\frac{ds(t)}{dt} = \dot{s}(t)$ that will allow the curve's trajectory to be within the actuator limitations for both axes. The torque required by each actuator at a given point on a given curve is dependent on the shape of the curve and the path velocity $\dot{s}(t)$. Due to the Bang-Bang nature of the time optimal problem we wish to solve, one actuator will be limiting the maximum $\dot{s}(t)$ at each point along the curve. The optimisation algorithm will determine the limiting actuator and generate a piecewise path velocity that obeys that actuator constraint.

2.1.0.5 Server

The implementation of the transmission of the optimal path from Server to Client was given to be a transmission of the velocity instructions that the PLC should give to its motors at a given uniform distribution of sample times. The PLC could linearly accelerate the motors to those velocities to achieve a good approximation of the curve to be followed. This was implemented as a result of limitations on the frequency of velocity instructions given by the PLC and also to simplify the code required to be written on the PLC side. This technique was simulated to result in a high accuracy for a sufficiently high instruction frequency.

2.1.0.6 Client

2.1.0.7 PLC State Machine

2.1.0.8 Motor Drivers

2.2 Uniform Rational Basis Splines

2.2.1 Introduction to URBS

An Uniform Rational Basis Spline (URBS) is a way of expressing a complex path with a small data set. An URBS curve consists of two main components; control points and basis splines. Control points are a set of locations in space which are individually weighted and summed in order to give the path that the curve travels. They are represented mathematically by the symbol $\mathbf{N}_i^n$, where n represents the dimension of the control point and i is an indexing identifier. Basis splines calculate the weightings that each control point will be given at each point along the curve. As the curve progresses, each spline rises and falls in magnitude. At any point on the curve, the total sum of the magnitudes of all the splines is one. The higher a control point is weighted, the closer the curve will approach to its location. The first and the last control points are the only points ever to have a weighting of one - meaning that their positions are actually on the curve. Every other control point is weighted in an identical fashion with its influence beginning at zero, rising quadratically to a maximum of 0.75, before again fading to zero. Weightings at specific points along the curve are represented mathematically by the symbol $\mathbf{W}_i^d$ where d represents the order of the polynomial that defines the weighting curve and i is an indexing identifier.

In order to display the power of an URBS curve, a demonstrative example is shown in Fig. 2.2.1. The curve can be seen to touch the first and the last control points, but does not ever reach the middle two. The point halfway along the curve is a special point known as a knot. A longer curve might have many more of these knot points. Each knot point is a place where a control point's weighting becomes zero and its position will no longer have an effect on the path that the curve takes. As this knot exits, so another enters, with another control point beginning from zero weighting and growing in weighting as the curve continues. At these point the two control points with non-zero weighting are weighted evenly and as such the curve lies directly between them.

A point along the horizontal axis of the spline graph seen in Figure 2.2.1 corresponds to a point on the curve. Since in our application we desire to travel along a path characterised by a URBS curve, we use a parametrisation for each of these points denoted by s. Henceforth we often desire to measure the progress made along the path over time. This makes it natural to have s become a function of time $s(t)$. In all instances,

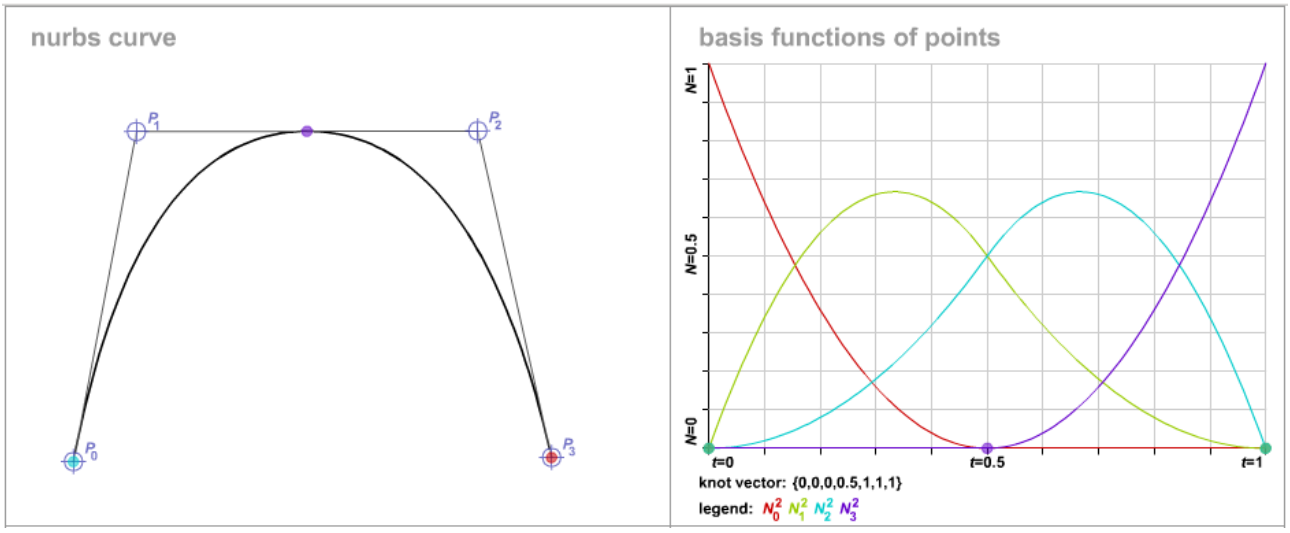

Figure 2.2.1: The splines in the right section represent the weighting and mixing of each of the four control points in order to form the curved line seen alongside those points in the left section.[5]

s begins a curve at zero and completes a curve when s equates to one. Thus;

$$s(t) \in [0, 1]$$
$$s(t_0) = 0$$
$$s(t_f) = 1$$

Where t_0 denotes the starting time for the travelling of a path and t_f denotes the finishing time.

2.2.2 Design Rationale

The decision to use URBS as a method of restricting and structuring the input curve type was made for several reasons.

Firstly the structure that a spline based representation gives to the input data gives natural compression for that data, as compared to storing the location of each individual point upon the desired curve. This effect is easily seen when compared to an array based method of storing curve data for single given line. A number of integers on the order of 20 can store the same amount of data as thousands of bits. However, this effectiveness goes down when approximating a high density of curves per unit of area. In our application, the average density of curve was to be very low. Thus, an URBS curve fit well for data compression.

This paradigm also allows for easy storage and manipulation of input curves by the GUI program. It also decouples the size and form of the curve from any functions that interact with it, as the curves all have a standard form and all data only represents objects that are of interest. This means that computation interactions were easier to design and were intrinsically flexible to alterations in the makeup of the URBS curves. With an array based method altering something as simple as the relative size of a data set to be

printed would create problems and require patches to most of the functions that make up the pathway from data to printing commands.

The restriction of the curves to splines of the 3rd order ensures that the parametrised curve has the property of being twice differentiable;

$$\mathbf{q}(s) \in \mathbb{C}^2$$

Where $\mathbf{q}(s)$ represents the position of each axis with respect to a point along the curve. If this were not the case, the path velocity could not be be guaranteed to be continuous. The plant would then require infinite acceleration at those discontinuities in order to travel the path at any speed. Thus a 3rd order URBS curve can be guaranteed to be traversable by a system with finite force input without any halt to its movement. Each curve can be swept in a single motion in a continuous manner. This results in fast traversal of the path, allowing the mechanism to complete its tasks in a short amount of time.

Finally, the parametrisation of the curve $\mathbf{q}(s)$, allows us to easily retrieve the path velocity $\mathbf{q}'(s) = \frac{dq(s)}{ds}$ and path acceleration $\mathbf{q}''(s) = \frac{d^2q(s)}{ds^2}$ unambiguously at any point on the curve. This is vital in calculating the forces required for a path traversal at any given $\dot{s}(t)$. The requirements of the path can thus be analysed with a view to perfect traversal accuracy. In comparison, an array based method would have a path velocity which would need to be estimated based on discrete position data. The method used to extract the path velocities would be inherently inaccurate.

The URBS curve has a few limitations in implementation, particularly with complexities involved the creation of a spline to represent a target image. However when a movement command is in a spline format it guarantees properties which result in easy application of optimisation calculations and the overall accuracy of command output.

2.2.3 URBS Analysis

Given their benefits, 3rd order URBS were used as the curve data structure for the DoodleBot.

The general weighting structure of an URBS is the same for each and every curve, given the uniformity of their knots. This means that we can easily calculate the basis functions for an URBS given only the number of control points. Thus an URBS curve is completely defined by the number and location of its control points.

A knot vector is a set of numbers which define the locations of all of the knots that a curve contains. An URBS contains only evenly spaced knots. The knot vector defines the knot spans of a curve. Over each span, only three control points will have non-zero weighting. At the end of each span, one control point's weighting goes to zero and is substituted by another control point, whose weighting begins from zero also. The numeric values of the knots within the vector are unimportant; the curve parametrising variable s starts from the first knot and ends at the last and all knots are evenly spaced. We have arbitrarily defined the knots such that all lie between zero and one.

The knot vector defines weighting coefficients for each of the control points. The values of these weights quadratically shifts across the domain of $s \in [0, 1]$. This brings consecutive control points into prominence evenly as the path parameter is traversed. Each point on the curve consists of a linear combination of the control points that are non-zero for the knot span. This is demonstrated in the following equation

$$q_i(s) = w_i \mathbf{N}_i^2 + w_{i-1} \mathbf{N}_{i-1}^2 + w_{i-2} \mathbf{N}_{i-2}^2$$

Where $q_i(s)$ is a point on the path with s inside the ith knot span and N_i^2 is the ith control point.

When the s parameter passes a knot value such as $s = 0.5$ in Fig 2.2.1, it changes the knot span it is within. This signifies that the lowest indexed control point no longer has influence over the position of the path and the control point corresponding to the index of the new knot span will begin to rise in weighting.

In this way each point on the curve is defined by a linear combination of three control points and each control point has weighted influence over a similar sized domain on s. The weightings for each control point can be determined given the knot vector.

The knot vector can be obtained by applying the following algorithm, where N is defined as the number of control points.

$$\begin{aligned}
K_3 &= [0, 0, 0, 1, 1, 1] \\
K_4 &= \left[0, 0, 0, \frac{1}{2}, 1, 1, 1\right] \\
K_5 &= \left[0, 0, 0, \frac{1}{3}, \frac{2}{3}, 1, 1, 1\right] \\
K_N &= \left[0, 0, 0, \frac{1}{N-2}, \frac{2}{N-2} \cdots, \frac{N-2}{N-2}, 1, 1\right]
\end{aligned}$$

subject to

$$\begin{aligned}
N &\geq 3 \\
N &\in \mathbb{Z}
\end{aligned}$$

Note that there are three repeated knots at the beginning and end of a curve. The knot span of these points each has a width of zero, meaning that the curve does not exist inside these spans. These knots make sure that the curve terminates at the location of the first and last control point. They also ensure that the weighting functions have defined values throughout the recursive calculations.

The knot vector can be calculated when one is given the number of control points of a curve. Given the knot vector we can use the recursive definition of URBS basis functions to calculate the the control point

weightings for any given $s \in [0, 1]$

$$W_i^d(s) = \frac{s - K_i}{K_{i+d} - K_i} W_i^{d-1}(s) + \frac{K_{i+d+1} - s}{K_{i+d+1} - K_{i+1}} W_{i+1}^{d-1}(s)$$

$$W_i^0(s) = \begin{cases} 1 & \text{if } K_i \leq s < K_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

Where K_i is the knot at index i and d is the dimension of the URBS curve.

For the cases where the multiplying fractions result in $\frac{0}{0}$, we take the result to be 0.

When the URBS curve is of dimension 2 the line equation for $q_i(s) = q(s) \Big|_{K_i \leq s < K_{i+1}}$ results in the following definition [1];

$$q_i(s) = \mathbf{N}_i^2 W_i^2(s) + \mathbf{N}_{i-1}^2 W_{i-1}^2(s) + \mathbf{N}_{i-2}^2 W_{i-2}^2(s)$$

Where $\mathbf{N}_i^2 = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$ is the ith control point and $W_i^2(s)$ is the second order basis weighting corresponding to that control point. For details of the derivation and closed form of $W_i^2(s)$, please see Appendix A.

Given the piecewise form of the line, one can easily take the derivative of $\mathbf{q}(s)$ with respect to s and obtain the path velocity $\frac{d\mathbf{q}(s)}{ds} = \mathbf{q}'(s)$ and path acceleration $\frac{d^2\mathbf{q}(s)}{ds^2} = \mathbf{q}''(s)$. This results in the derivative of $W_i^2(s)$ with respect to s alongside the same control points and knot indexes, calculating the result in a manner exactly the same as for the position.

2.3 Control System and Optimisation

2.3.1 Open Loop Control

The DoodleBot uses two stepper motors in open loop to control the position and movement of the drawing head (open loop justification in Section 2.3.1.1). The input to the system is the frequency of voltage pulses and the output is the velocity of the drawing head.

Both the input and output operate in 'step' units (see Section 3.2 for more detail on what this unit is in the real system).

For this system to operate as required, there are constraints on the maximum acceleration the system can run at.

2.3.1.1 Stepper Motors and Open Loop Justification

Stepper Motors are brushless DC motors and are used as the primary drivers for X and Y movement in the DoodleBot. Where a standard DC motor creates an output torque proportional to input current, a stepper motor rotates at a speed that matches the frequency of input voltage steps.

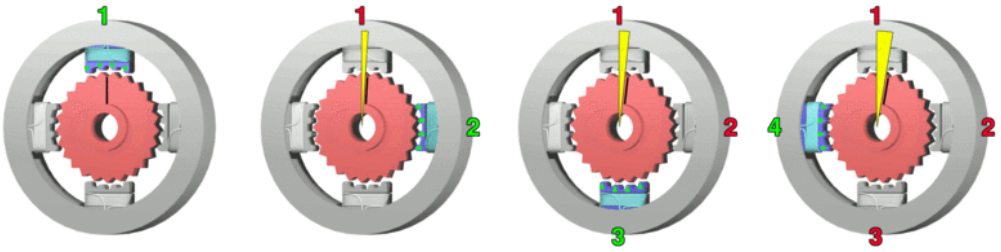

Figure 2.3.1: Operation of a simplified stepper motor over four steps

As shown in Figure 2.3.1, stepper motors consist of several electromagnets arranged in a equally spaced pattern around a gear-shaped rotor. Each electromagnet is shaped to have its own teeth that can align with the rotor's teeth.

When the stepper motor driver receives a single voltage step, it provides a constant current to the first electromagnet, creating magnetic force that rotates the rotor a small amount (a single step) such that the teeth on the rotor gear align with the teeth on the magnet. Damping in the mechanical system prevent oscillations about this point. As long as the input voltage pulse is maintained, this electromagnet will remain active.

When the first electromagnet is aligned with the rotor teeth, the second electromagnet will be out of alignment. When a second input voltage pulse is received, the stepper motor driver will activate the second electromagnet, causing the rotor to make another rotation step such that it is now in alignment with the second electromagnet. This process continues sequentially across numerous electromagnets around the rotor.

The important thing to note is that the speed of rotation is proportional to the frequency in which the input voltage pulses are received. As long as the magnetic force provided by the electromagnet is enough for the rotor to make the required step within the time the electromagnet is active, the mechanical load on the rotor does not effect the average speed at all. A lighter load may allow the individual step to be quicker, but as the second step will not be made until the next input pulse is received, the differences between a heavy and light load are negligible.

This behaviour means that within certain torque constraints, the stepper motor is a robust and predictable device that we can run in open loop. Since the loads on the system do not vary significantly between possible states (the gravitational force is always the same orientation for all states and mass is constant) the torque constraint can be approximated by an acceleration constraint.

2.3.2 Path Velocity Optimisation

A machine that follows prescribed paths has an inherent trade-off between obeying the physical restrictions of the device and following a path at the fastest possible speed. There is a wealth of information surrounding the topic of the optimisation of such physical systems, seen in [2], [3], [4], [6], [7] and [8]. Each study was relevant to the optimisation that we had in mind, but used differing strategies and overcame different limitations to achieve a trajectory that minimised some particular cost. The method that we decided was the most relevant to our problem and was the process that we ended up implementing was covered in [7]. A comparison of the different optimisation strategies is instructive and can be seen in Appendix B.

The physics model we chose to construct the limitations of our plant was a pair of double integrators. Given that we can determine the limits on $\ddot{\mathbf{q}}(t) = \begin{bmatrix} \frac{d^2x(t)}{dt^2} \\ \frac{d^2y(t)}{dt^2} \end{bmatrix} = \begin{bmatrix} \ddot{x}(t) \\ \ddot{y}(t) \end{bmatrix}$, we can construct an equality that will impose limits on the trajectory of each of the two axes in following the path. We can separate the specifics of the path and the speed of its traversal via the path parametrisation s.

$$\mathbf{q}(s) = \sum_{i=1}^{N} \mathbf{N}_i^2 W_{i,2}(s) = \begin{bmatrix} x(s) \\ y(s) \end{bmatrix}$$
$$\mathbf{q}(t) = \mathbf{q}\left(s(t)\right)$$

Here we can see the mechanism of the separation of concerns. We construct a function $s(t)$ which represents our progress along the path as a function of time. We have before stated that

$$s(t_0) = 0$$
$$s(t_f) = 1$$
$$s(t) \in [0, 1]$$

Which means that the plant starts at the beginning of the path $s = 0$ at t_0 and completes its movement at the end of the path $s = 1$ at t_f. We can find the derivatives of $\mathbf{q}(t)$ with respect to time in terms of the properties of the path $\mathbf{q}(s)$ and the time derivatives of a traversal trajectory $s(t)$ as follows;

$$\dot{\mathbf{q}}(t) = \frac{d\mathbf{q}}{ds}\frac{ds}{dt}$$
$$\ddot{\mathbf{q}}(t) = \frac{d}{dt}\left(\frac{d\mathbf{q}}{ds}\right)\frac{ds}{dt} + \frac{d\mathbf{q}}{ds}\frac{d^2s}{dt^2}$$
$$= \frac{d^2\mathbf{q}}{ds^2}\left(\frac{ds}{dt}\right)^2 + \frac{d\mathbf{q}}{ds}\frac{d^2s}{dt^2}$$
$$\ddot{\mathbf{q}}(t) = \mathbf{q}''(t)\dot{s}^2(t) + \mathbf{q}'(t)\ddot{s}(t)$$

Where $\mathbf{q}''(t) = \frac{d^2\mathbf{q}(s)}{ds^2}$, $\mathbf{q}'(t) = \frac{d\mathbf{q}(s)}{ds}$, $\dot{s}(t) = \frac{ds}{dt}$ and $\ddot{s}(t) = \frac{d^2s}{dt^2}$.

Our problem can be simply stated as choosing $\dot{s}(t)$ such that

$$\ddot{\mathbf{q}}_{min} \leq \ddot{\mathbf{q}}(t) \leq \ddot{\mathbf{q}}_{max}$$

This can be stated intuitively as attempting to minimise the time taken for the completion of a specific path without exceeding the torque limitations of a plant.

The problem faced in minimising the time taken is that there are points known as switching points in the trajectory of $s(t)$ which occur when the path velocity $\dot{s}(t)$ needs to decrease in order to be able to clear a section of the curve without breaking the constraints. In order to minimise the time taken to traverse the curve, there are switching points in the trajectory of $s(t)$ when a difficult section of curve is completed. At these points the path velocity should be increasing as much as possible within the bounds of the constraints. This result of maximum acceleration and deceleration is known as 'Bang Bang' control and is typical in the result of time-optimal calculations. This result is intuitive when one considers that the actuators are performing at their limits at all stages of the curve.

To find the solution that minimises the completion time these key acceleration and deceleration switching points need to be identified. The identification of these points gives the positions where the system should begin to accelerate and decelerate along the path. In order to find these switching points to obtain the minimum time of traversal a few options are available.

One could attempt to determine the limiting axis at each point on the curve and then determine the optimal points for switching between accelerating and decelerating the path velocity via iterative shooting methods to meet the imposed limitations. This result will give the global optimum for the minimum time of path traversal, given the constraints. A drawback of this problem is that it is a difficult problem to solve.

Another simpler option is to break the path into discrete segments, calculate the maximum $\mathbf{q}'(s)$ and $\mathbf{q}''(s)$ for each segment and accelerate along the path such that these maximum values are satisfied at the commencement of and during each of these discrete divisions along the path. This method is a discrete approximation of the continuous method outlined above. As such the answer that results is an approximation of the global optimum. Limiting sections of the path that might be confined to a small area spread out with the discretisation of the path. Hence the problem that is solved is not an accurate representation of the true problem. However we are able to guarantee that the constraints of the system will not be exceeded and the system will minimise the time of completion for the approximated system. Thus the approximated solution will approach the global solution as the granularity of the discrete approximation increases.

Once all of the switching points are identified, we can easily calculate the optimal path trajectory $s^*(t)$, from which we can calculate the desired velocities and positions along $x^*(t)$ and $y^*(t)$ with ease.

Formally, the problem to be solved is expressed by;

$$\begin{aligned}
&\min_{T,\, s(\cdot)} \int_0^T 1 dt \\
\text{subject to} \quad &\ddot{\mathbf{q}}(t) = \mathbf{q}'(t)\ddot{s}(t) + \mathbf{q}''(t)\dot{s}^2(t) \\
&s(0) = 0 \\
&s(T) = 1 \\
&\dot{s}(0) = 0 \\
&\dot{s}(T) = 0 \\
&\dot{s}(t) \geq 0 \\
&\ddot{\mathbf{q}}_{min} \leq \ddot{\mathbf{q}}(t) \leq \ddot{\mathbf{q}}_{max} \\
&\text{for } t \in [0, T]
\end{aligned}$$

2.3.3 Pontryagin's Maximum Principle

We set about attempting to solve the optimal control problem utilising Pontryagin's Maximum Principle. Despite our best efforts, we found the problem to be intractable to solve in the general case. The major confounding factor we had in finding a continuous and completely optimal solution to our problem lies in the construction of a representation for the interaction of the path constraints on the separate axes. One can construct a Hamiltonian

2.3.4 Convex Optimisation Approach

The method used to solve the problem involved breaking the curve down into a discrete number of sections and solving the continuous problem in a piecewise manner, which allows us to more easily take into consideration the restrictions imposed.

The problem is now to apply a numerical solving method in order to find the approximated optimum. Using the technique outlined in [7], the problem was reformulated such that the non-linear constraint $\ddot{\mathbf{q}}(t) = \mathbf{q}'(t)\ddot{s}(t) + \mathbf{q}''(t)\dot{s}^2(t)$ becomes linear for the solution variables, whilst the cost function remains convex. Enforcing linearity in the solution variables allows for the solution to be found by efficient solvers, as well as ensuring that a solver will not be able to get caught in a local minima. The key to this linearisation is representing the non-linear portion of the constraint $\dot{s}^2(t)$ with the solution variable $b(s)$ as so;

$$\begin{aligned}
a(s) &= \ddot{s}(s) \\
b(s) &= \dot{s}^2(s)
\end{aligned}$$

Such that the non-linear constraint becomes

$$\ddot{\mathbf{q}}(s) = \mathbf{q}'(s)a(s) + \mathbf{q}''(s)b(s)$$

Note that the time dependence of the path parametrisation is neglected. This is because cost function is reformulated with a change of variables such that its dependence on t is replaced by a dependence on s. Thus the solution that minimises the cost function will minimise the time taken implicitly without direct reference to time as shown here;

$$\begin{aligned} J(T) &= \int_0^T 1 \, dt \\ J(s(\cdot)) &= \int_{s(0)}^{s(T)} \frac{1}{\dot{s}} ds \\ &= \int_0^1 \frac{1}{\dot{s}} ds \end{aligned}$$

The cost function is trying to minimise the inverse of $\dot{s}(t)$, which is the equivalent of attempting to maximise the path velocity. Given that we are attempting to solve for $b(s) = \dot{s}^2(s)$, we can formulate this cost in terms of our solution variable;

$$J(s(\cdot)) = \int_0^1 \frac{1}{\sqrt{b(s)}} ds$$

The change of variables gives us a cost that remains convex and is in terms of one of the solution variables.

2.3.5 Discretisation of Path

The results of this alteration to our problem are far reaching. Firstly, it can be seen that our problem is now linear in its constraints. This results in our viable solution space taking up a contiguous range, meaning that for the solver no isolated solutions are unreachable. Secondly, the cost integral that integrates over the trajectory s is convex in the solution variable $b(s)$, so that a solver will always be able to determine the correct direction to move the state such that it converges to the local optimum. With the combination of these two properties, it can be seen that a solver can be employed to correctly solve for the optimal trajectory $\dot{s}^*(s)$, through the solution of $b^*(s)$

In order to form a discrete problem that a computer can iterate over and solve, the path is discretised and the trajectory is solved over each path segment. As the control input, $a(s)$ is given as being piecewise constant for each discrete segment of s. Thus, as it is finite and constrained by the system constraints. $\sqrt{b(s)}$ is piecewise linear and continuous, meaning $b(s)$ is piecewise non-linear and continuous.

2.3.6 SeDuMi

The discrete optimisation problem is reformulated into a second order cone program, so as to be solved with numerical efficiency. For each segment, the bounds on $b(s)$ and $a(s)$ are defined with respect to the acceleration constraints and with calculation of $\mathbf{q}''(s)$ and $\mathbf{q}'(s)$. The starting and finishing values of $b(s)$ are anchored to zero. Finally, two solution variables are introduced in order to enable the efficiency of a second order cone program. $c(s)$ is introduced as $\sqrt{b(s)}$ and $d(s)$ is introduced as the increment of the cost integral which now, thanks to the discretisation process is a cost sum- $\frac{1}{c(k\Delta s)+c((k+)\Delta s}$. The entire solving functionality was automated and wrapped in a function that could solve the optimal trajectory for a curve given simply an URBS representation and the torque constraints.

2.3.7 Results

Here, the process that a curve undergoes from initialisation to the generation of the final final velocity profile will be demonstrated, drawing upon the theoretical underpinnings of the previous sections. Firstly, a curve is obtained through plotting or by extraction from image features. The curve that will be taken through the optimisation process can be seen in Figure 2.3.2. The curve has eight control points and has a complex shape that will incur several changes in the limiting axis, demonstrating the ability of our optimisation process to drive the machine to the limits of its physical limitations.

The spline is decomposed into its X and Y axis coordinates, giving the positions that each axis should achieve at each point on the curve. At this level of zoom the error in the final curve that our system produces is nearly imperceptible and is included on the position profiles seen in Fig.2.3.3. The deviation from the correct path is not an artefact of the optimisation process. The velocity profile sampling which is required to pass the movement information across to the plant discretises the velocity information, which causes accumulating integration error.[1] However it can be seen in Fig.2.3.3 that the final effect is slight. The effects are also reset at the culmination of each curve. Thus the introduced error was considered to be within allowable bounds for our purposes.

Given the path, our algorithm can extract the path velocity and acceleration ($\mathbf{q}'(t)$ and $\mathbf{q}''(t)$) at each discretisation point for the SeDuMi solver. The curves for the path in the example spline seen in Fig 2.3.2 can be seen in Fig 2.3.4. These values are extracted by taking the derivative of the URBS weighting equations with respect to s.

Given this construction of the problem, SeDuMi can solve for the optimal discrete $\dot{s}^*(s)$. The solution given for the example curve can be seen in Fig. 2.3.5. The final calculation to achieve the desired $s^*(t)$ requires that

[1] Further work would include an interpolation scheme based on the sampling rate that would adjust these profiles such that the machine would be back on the curve after each sampling period. This implementation is not trivial as the adjustments to the velocity discretisation may cause the acceleration constraints to become exceeded.

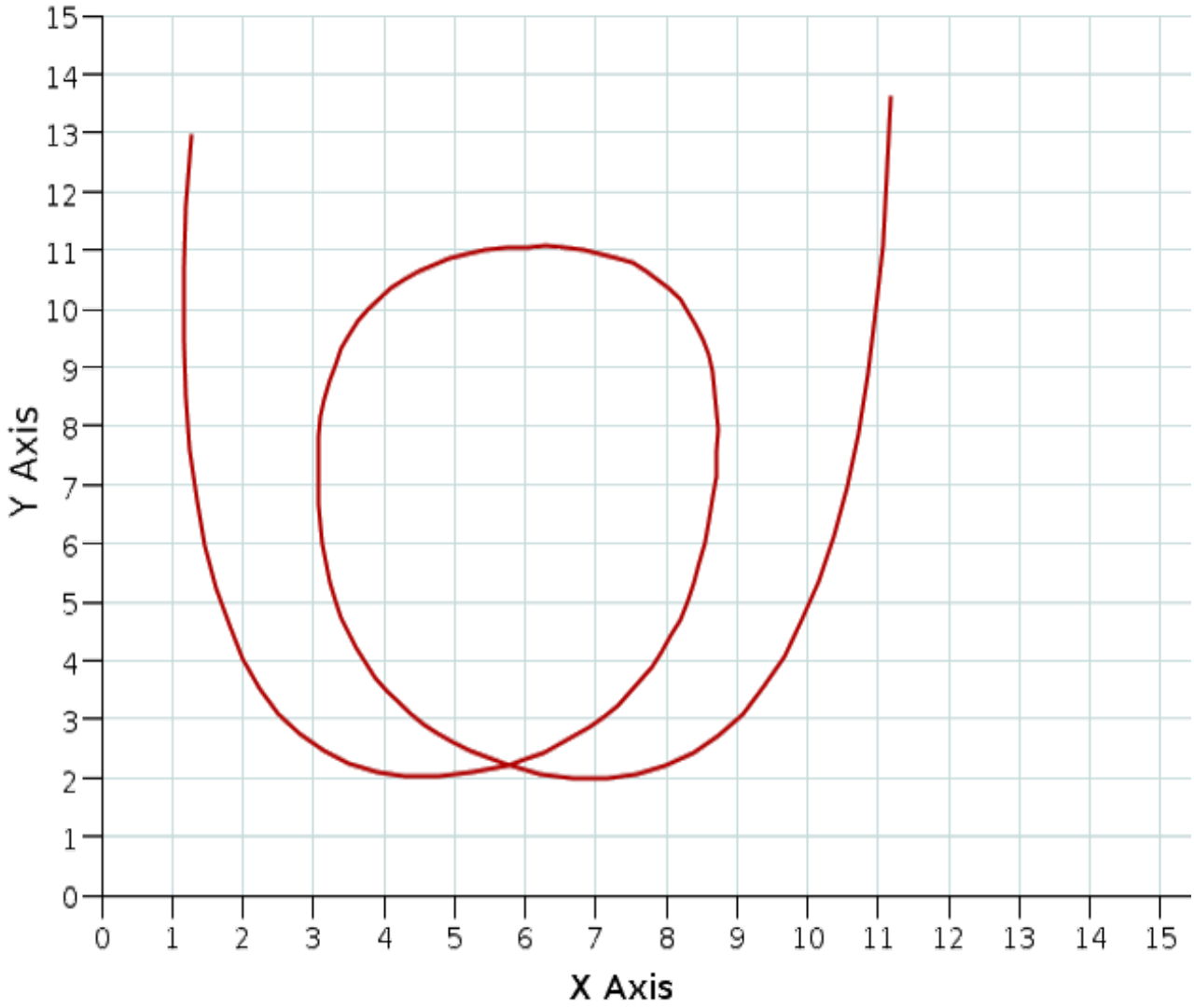

Figure 2.3.2: This spline has 7 control points and 10 knots. This example will be used to demonstrate the properties of URBS curves and the optimisation process.

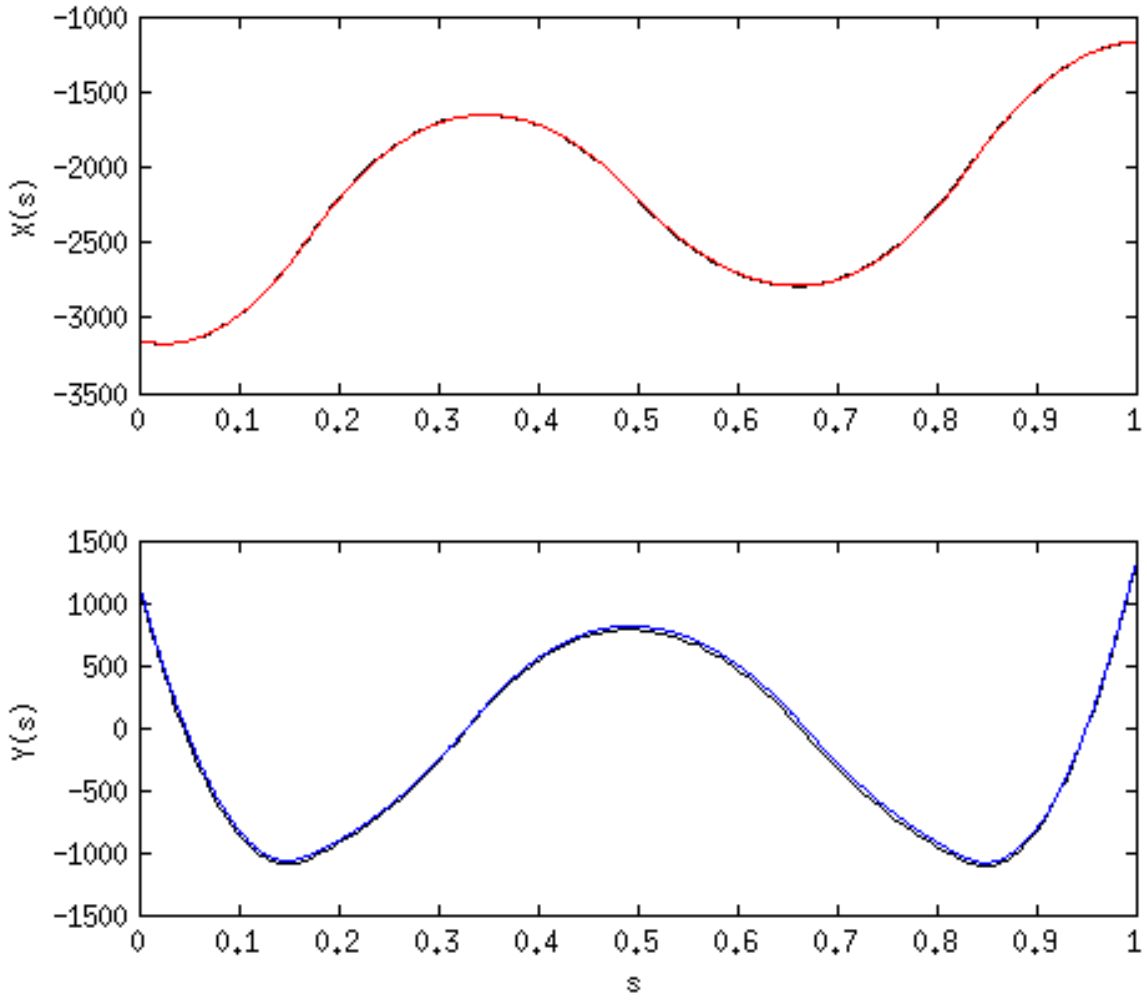

Figure 2.3.3: The X axis (TOP) and Y axis (BOTTOM) displaying the correct trajectory in colour and the approximated path that will be followed in black, given a velocity profile that is sampled at 25Hz.

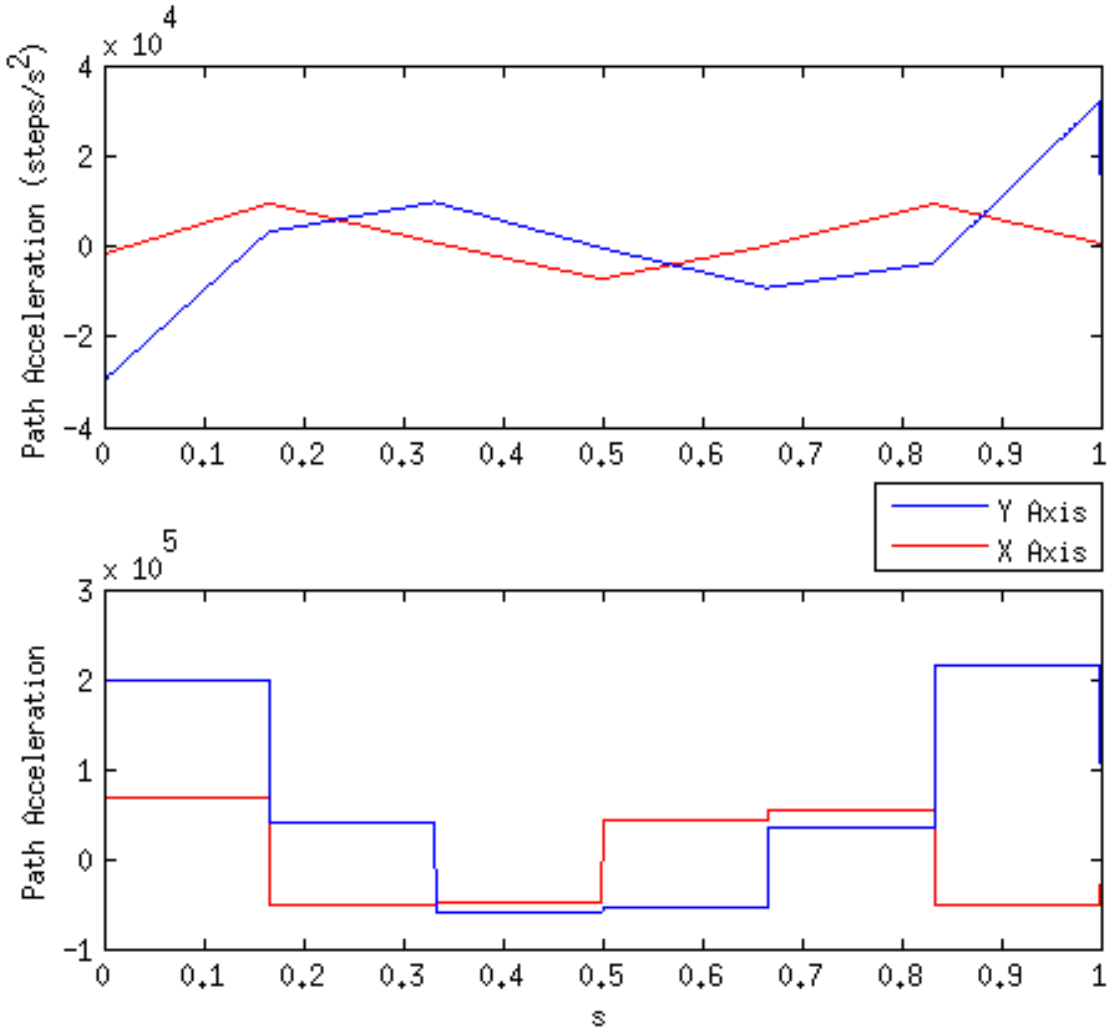

Figure 2.3.4: The curve velocity (TOP) and acceleration (BOTTOM) for the example curve. Note the discrete jumps in level for the acceleration and the corresponding changes in gradient for the velocity.

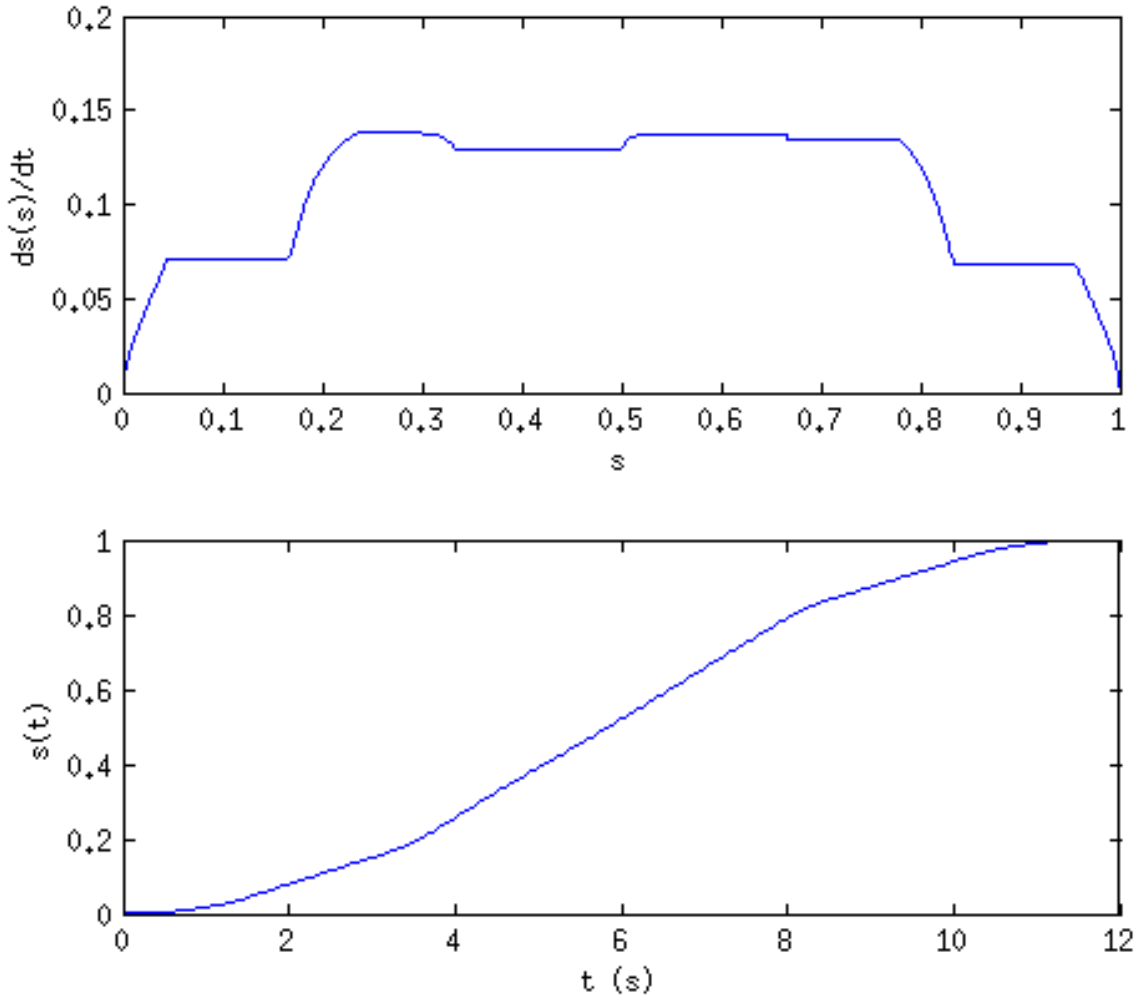

Figure 2.3.5: The optimal curve velocity (TOP) and the retrieved path trajectory (BOTTOM) for the example curve.

we compute the expected time t for each s via the inverse relation

$$t(s) = \int_{\circ}^{s} \frac{1}{\dot{s}^*(s)} du$$

Since $\dot{s}^*(s)$ is defined in a piecewise non-linear fashion due to its relation to the piecewise linear trajectory $b(s)$, we obtain a mapping between t and s for arbitrary t via the algorithm outlined in Appendix C. The output $s(k\Delta T)$ can be seen in Fig. 2.3.5. Though the solution can be solved into a piecewise closed form, an algorithm that solves for a given discrete sampling rate $k\Delta T$ was used instead. This allows us to specify a desired velocity profile sampling frequency and hence use the subsequently calculated values of $s(k\Delta T)$ to

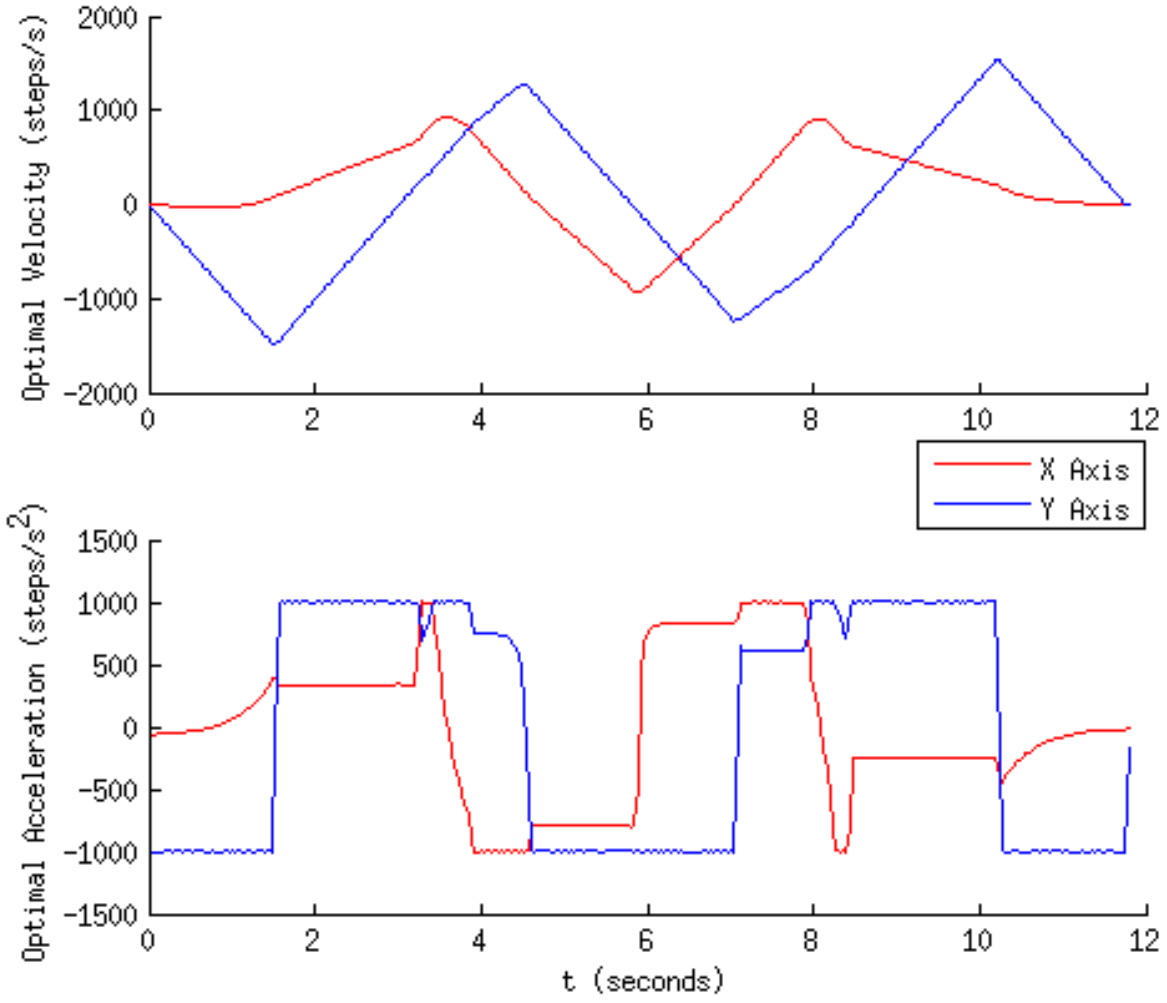

Figure 2.3.6: The optimal axis velocity (TOP) and the optimal path acceleration (BOTTOM) for the example curve.

generate the sampled velocity profile via

$$\frac{d\mathbf{q}\left(k\Delta T\right)}{dt} = \frac{\mathbf{q}\left(s(k\Delta T)\right)}{ds}\frac{ds(k\Delta T)}{dt}$$

Hence, we have generated the optimal axis velocity curves that can be sent to the unit in order to faithfully follow a URBS curve with a time optimal trajectory. An output of the optimal trajectory $\dot{\mathbf{q}}(k\Delta T)$ and $\ddot{\mathbf{q}}(k\Delta T)$ can be seen in Fig. 2.3.6.

2.4 Image Interpolation

3

Implementation

3.1 CNC Machine

3.2 Mechanical

3.2.1 Frame

3.2.1.1 Motors

3.2.1.2 Limit Switches

3.2.2 Drawing Head - Z-axis

3.3 Electrical and Wiring

The wiring diagram on the following page(Figure 3.3.1) shows electrical layout of the system. The DoodleBot has two major groups of components - the electronics and control components sit in the DoodleBot Control Enclosure and the electromechanical components are mounted to the DoodleBot CNC

Frame - that are interfaced by a 12-wire loom and a 2-wire loom broken by two connectors.

3.3.1 Connectors and Loom

3.3.2 Voltage Dividers

The direction signals are produced from the PLC's embedded output on pins 0 and 1 at 24V. The AMCI SMD23 needs to read these signals as a +5V differential signal across its terminals.

Because the AMCI SMD23 direction input is a digital signal (ie, high impedance input), we can use a simple voltage divider circuit to produce the correct voltage. The PLC embedded output can produce a maximum of 500mA per pin and requires a minimum of 1mA per pin while in the on-state to operate at 24V (going below this current limit causes the voltage to quickly drop to 0V).

Choosing resistors of 820”Ωand 220”Ωin the voltage dividers gives a 5.08V output across the 220”Ωresistor. The voltage divider draws 2.3mA, which is within the operating specification of the embedded output.

3.3.3 Solenoid Driver

The pull type solenoid used in the Z-axis operation (drawing head) is designed to operate at 12V. It has a resistance of 58”Ωand an unknown inductance. In steady state operation it draws 207mA and consumes 6W. The solenoid can be modelled as an R-L circuit: an off-on transition will have current start at 0 and slowly ramp up to 207mA. In an on-off transition, there will be a spikes of negative current which gradually reduces to zero.

The PLC embedded output can produce a maximum of 500mA per pin and requires a minimum of 1mA per pin while in the on-state to operate at 24V (going below this current limit causes the voltage to quickly drop to 0V).

Due to the transient behaviour of the solenoid, a voltage divider cannot be used. Instead an STMicroelectronics L7812 Linear Voltage Regulator in TO220 packaging is used to perform a 24V-12V DC-DC conversion. 6W is lost as heat in the L7812, so a heatsink is used.

Due to the minimum current requirement of the PLC output, a TIP31C bipolar junction transistor (BJT) is used as the solenoid switch (a field effect transistor (FET) would not work since it would not draw enough current to allow the PLC to operate normally).

There is an additional requirement that the minimum current through the base of the BJT is enough to drive 207mA through the solenoid. The TIP31C datasheet shows that with a collector-emitter current of 1A, the minimum DC current gain is 25. So at least 8.28mA is required through the base. A more conservative value of 20mA is chosen by placing a 1.2k”Ωresistor in between the PLC output and BJT base.

To allow the inductor to discharge quickly (and without damaging other components), a flyback diode is put

across the Solenoid Driver Output in reverse orientation. In the on-off transition, the reverse current will have an alternate path to follow and energy will dissipate as heat in the solenoid and diode.

3.3.4 Kill Switch

The DoodleBot features a kill switch (emergency stop switch) mounted to Control Enclosure that disconnects the power supply's GND output from all the powered components. It should be noted that the power supply itself continues to operate and certain reactive components may retain their charge and as such, the kill switch should only be used in emergencies.

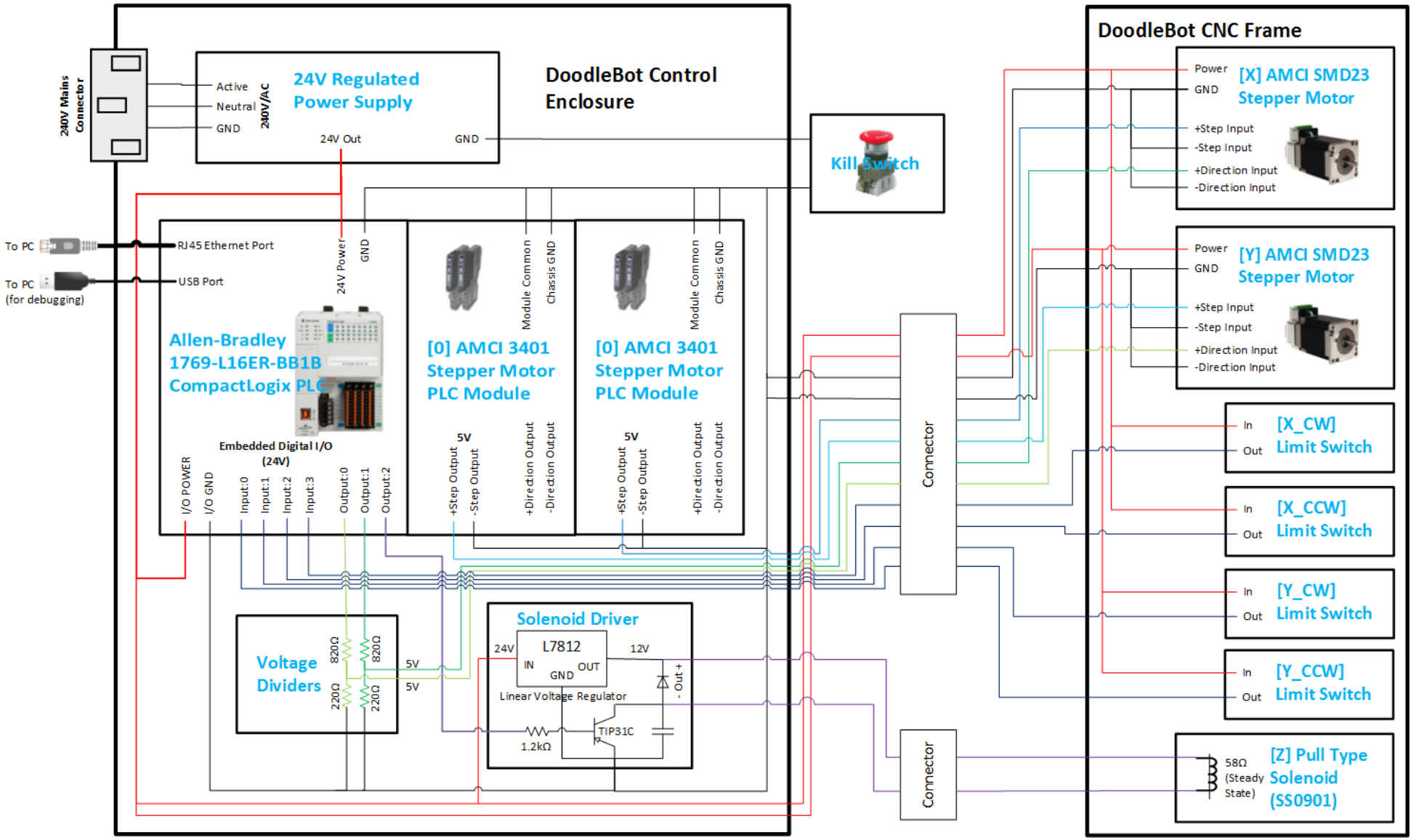

Figure 3.3.1: Wiring Diagram

3.4 PLC Control Software

3.4.1 Introduction to Programmable Logic Controllers

3.4.2 PLC Architecture

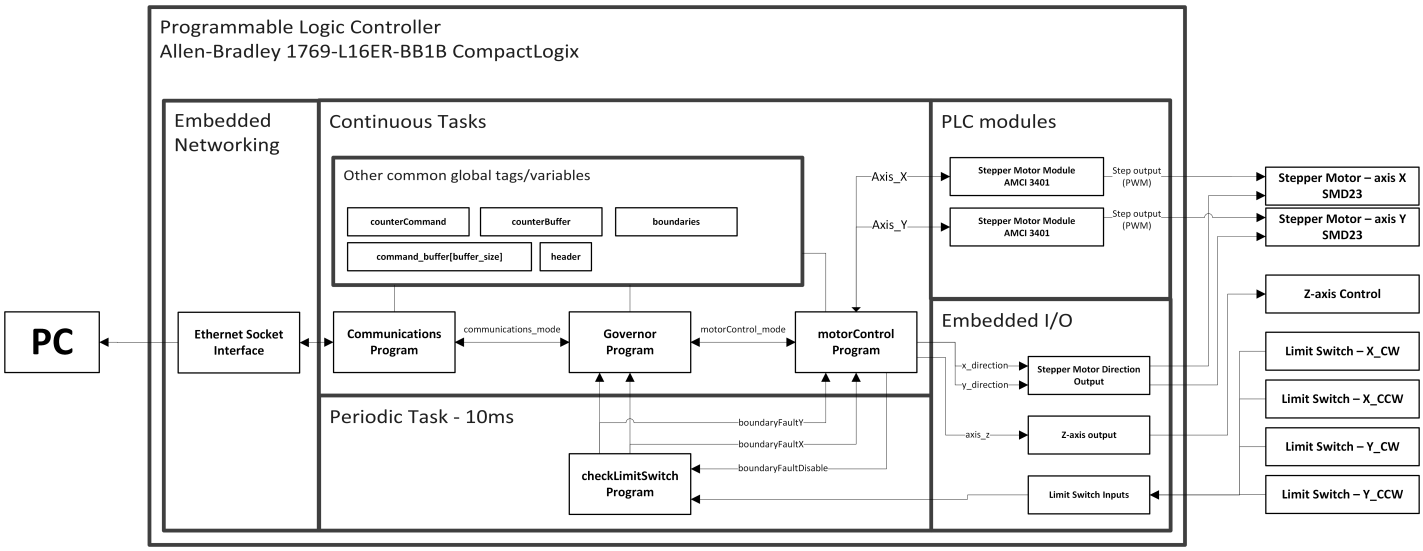

Figure 3.4.1: The program and I/O architecture of the PLC

3.4.2.1 Programs

The programmable logic controller features four programs, shown visually in 3.4.1

Governor

The Governor is a state machine that sets the overall state of the DoodleBot controller and controls Communications and motorControl.

Communications

The Communications state machine is in charge of managing the UDP socket interface, all network communications and the populating of the command list.

motorControl

The motorControl state machine handles all stepper motor control.

checkLimitSwitch

The checkLimitSwitch program periodically monitors the limit switch input and triggers an emergency stop if the switches are triggered when not expected at a period of 10ms.

3.4.2.2 Intraprogram Interfaces and Global Data

The Governor program interfaces with the Communications and motorControl programs via the two global scoped variables, *motorControl_mode* and *communications_mode* which act as intraprogram interfaces. On each state transition, the Governor sets these variables to certain values, determining what state to move the other programs into. The other programs can then set these variables to different values, informing the Governor on their own state changes and status which the Governor then uses as state transition conditions. Table 3.4.1 shows the possible values of the intraprogram interfaces.

Code	motorControl_mode	communications_mode	Set By
00	Wait	Wait	Governor
10	Initialize	Ready	Governor
11	Initialization complete	Header received	motorControl/communications
20	Operate commands	Buffer command list	Governor
21	All buffered commands complete	Buffering complete	motorControl/communications
22	N/A	Buffer full	communications

Table 3.4.1: Codes for interface between Governor, Communications and motorControl programs

In addition to locally scoped variables, there are also several key global scope variables that are used by two or more program that determine the behaviour and input of the system. These are shown in Figure 3.4.2

Variable Name	Element	Description
counterCommand	.counter	Index number of the command currently being operated on by motorControl (also, how many commands have been processed)
	.pointer	Index of where the command referred to by counterCommand.counter resides in the commandBuffer array
counterBuffer	.counter	Index number of the command currently being buffered by Communications (also, how many commands have been buffered)
	.pointer	Index of where the command referred to by counterBuffer.counter resides in the commandBuffer array
commandBuffer[]	.command_no	Command number of what the current array element stores
	.type	Determines the command type. 0 is a 'draw' and 1 is a 'move'.
	.x_speed	For 'draw' operations, determines the X target speed for this sample. For 'move' operations, determines the target position for X to move to.
	.y_speed	For 'draw' operations, determines the Y target speed for this sample. For 'move' operations, determines the target position for Y to move to.
BUFFER_SIZE	CONSTANT	Size of the command buffer
PC_IP	CONSTANT	IPv4 address of the PC producing the command list
boundaries	.X_CW	The number of clockwise steps from the center until the boundary in the X positive direction
	.X_CCW	The number of clockwise steps from the center until the boundary in the X negative direction
	.Y_CW	The number of clockwise steps from the center until the boundary in the Y negative direction
	.Y_CCW	The number of clockwise steps from the center until the boundary in the Y positive direction
header	.delta_time	Period of velocity profile samples
	.no_commands	Total number of commands to be received and executed on
	.max_accel_x	Max acceleration in the X axis
	.max_accel_y	Max acceleration in the Y axis
	.move_speed_x	Target speed for 'move' operations in the X axis
	.move_speed_y	Target speed for 'move' operations in the Y axis

Table 3.4.2: Global scope variables used by multiple programs

3.4.3 Governor

The Governor is a state machine that control the operation of the other programs in the DoodleBot system.

POWER_ON DoodleBot waits for the user to press one of the Y limit switches to state A to start initialization.

A_Init The DoodleBot checks the boundaries and centres itself on a 'home' position. After this is complete it progresses to state B.

B_ready The DoodleBot starts sending the 'ready' packet periodically to the PC, while waiting for a header to arrive. If a valid header arrives, it transitions to state C. If one of the Y limit switches are pressed, it returns to state A to run through the initialization routine once more.

C_buffer The DoodleBot begins requesting commands from the PC, filling the commandBuffer. When either the commandBuffer is full or all commands have been received, it progresses to state D.

D_operating The DoodleBot begins executing the commands in the commandBuffer. If there are still commands to received, it will continue to buffer when slots are freed up after being executed. After all commands have been processed the system returns to state B, ready for another header.

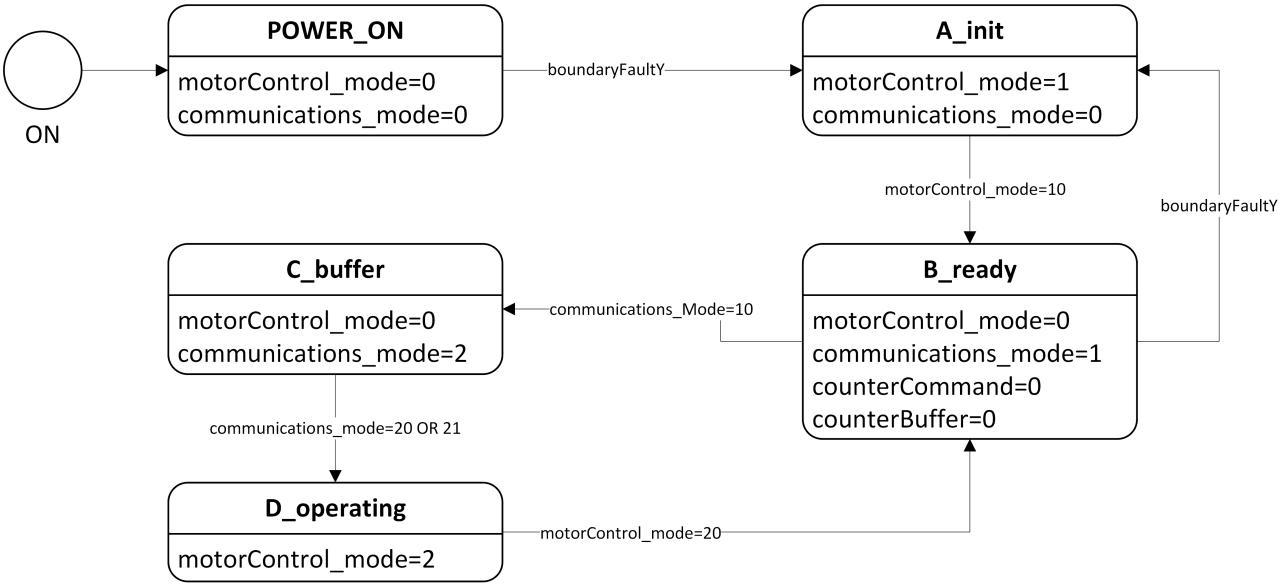

Figure 3.4.2: The Governor State Machine

3.4.4 Communications

The communications program has the role of communicating with the PC to populate the commandBuffer with the output of the PCs optimization stage. Figure 3.4.3 shows the operation of the program.

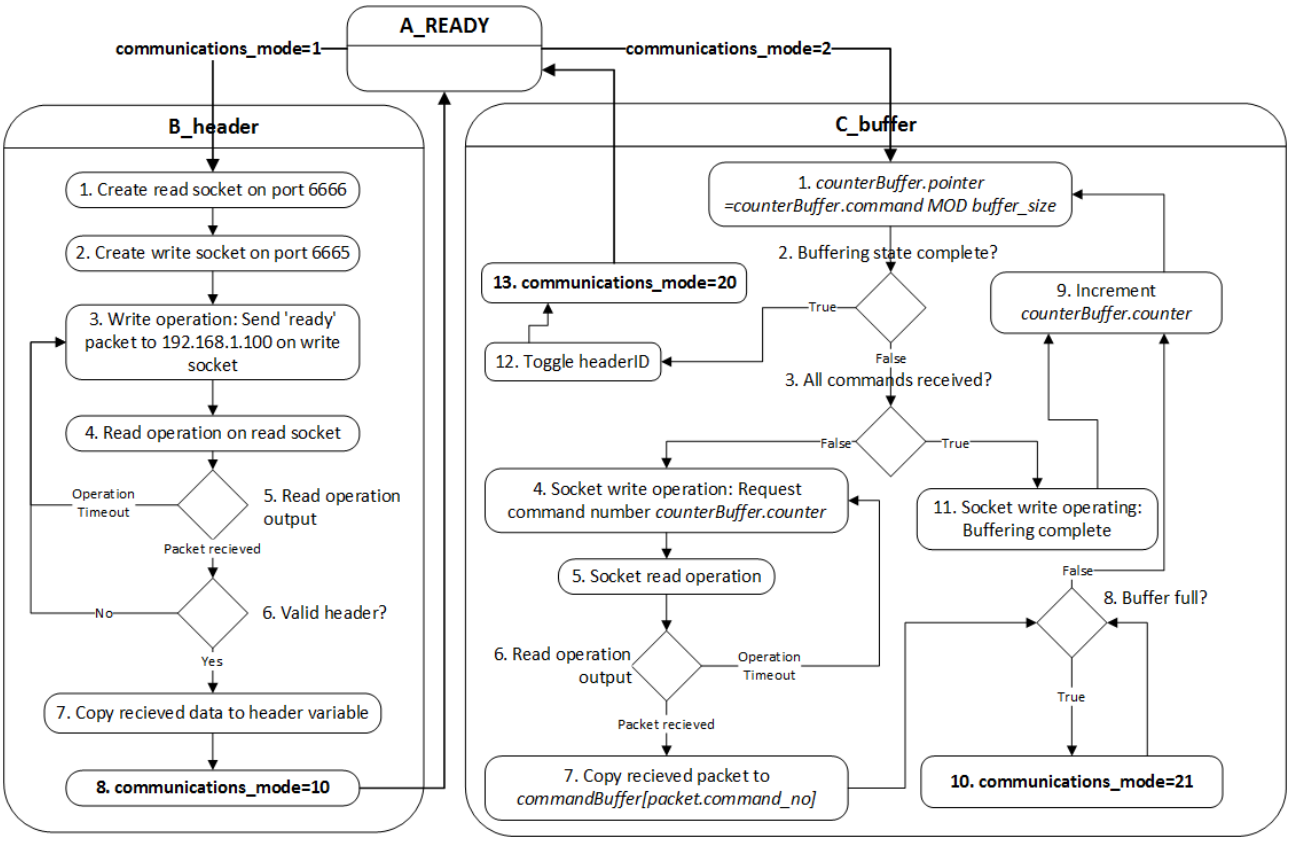

Figure 3.4.3: Communications program: States and State Actions

3.4.5 motorControl

3.4.5.1 Introduction to the AMCI 3401 Stepper Module

The main purpose of the motorControl program is to communicate with the two AMCI 3401 Stepper Motor Modules. Their primary role is to send two types of control signals to the stepper motor modules:

Step Output

A $5V$ square pulse output, with each rising edge signifying a single step of the motor.

Direction Output

A $5V$ boolean output that defines which direction the stepper motors should turn.

The modules takes its input by asynchronously monitoring a 16-byte command word on the PLC and mirror it internally. All commands given by the motorControl program involving modifying this command word. In addition, the module monitors its relative position and operating state by providing a 16-byte status word. The modules feature a broad range of commands, but the those used by the DoodleBot system are:

Absolute Move

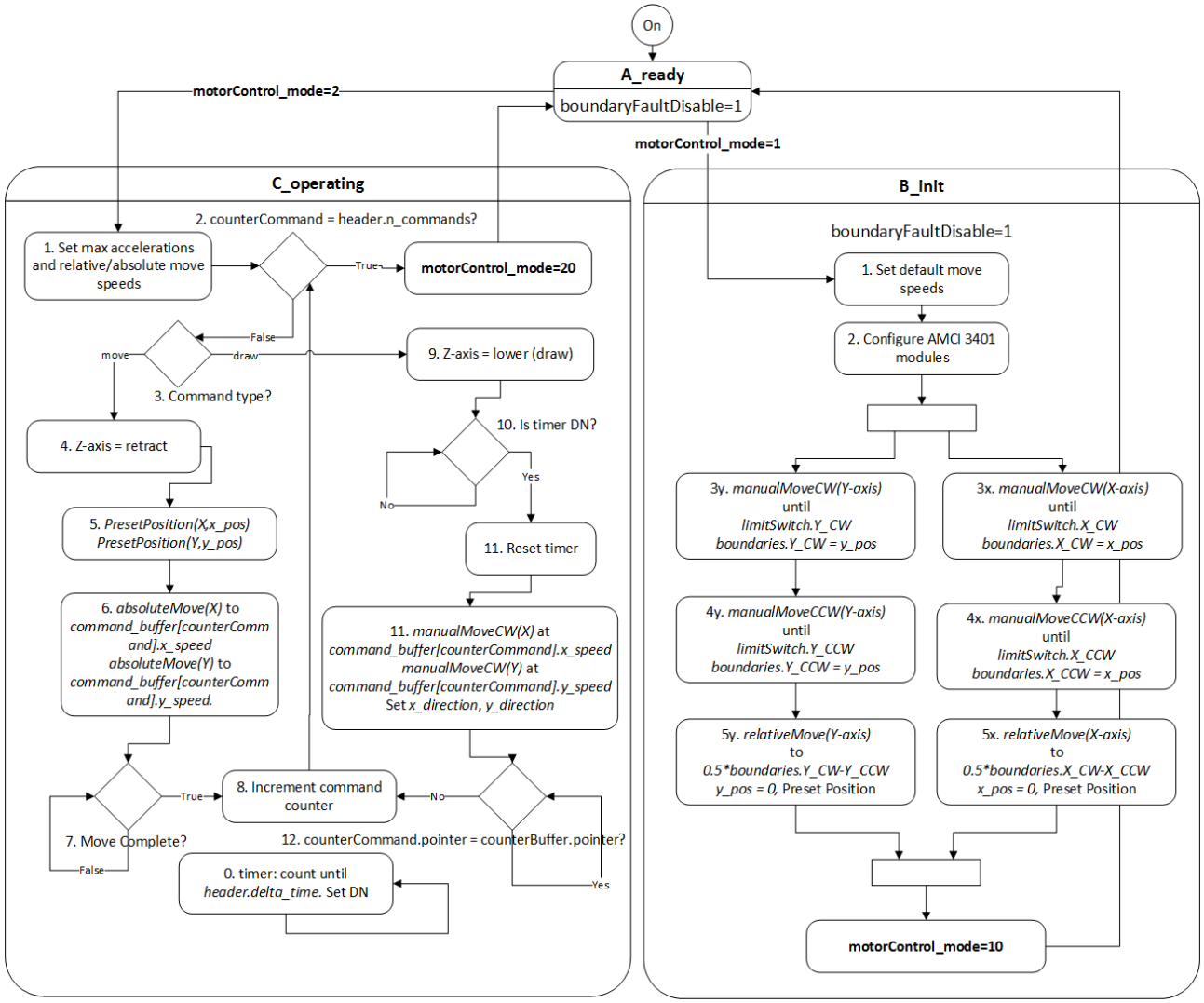

Figure 3.4.4: motorControl Program: States and State Actions

Given a position in space (relative to the preset origin position), the AMCI module creates the appropriate velocity profile to get there (within the acceleration/deceleration/top speed parameter limits).

Relative Move

Given a distance to travel (from the current position), the AMCI module creates the appropriate velocity profile to get there (within the acceleration/deceleration/top speed parameter limits).

Manual Move

Manual type moves are actually classified as two separate commands defining the direction of the move - manual move clockwise, and manual move counterclockwise. These moves accelerate to the programmed speed at the acceleration rate and travel until stopped. While moving in this state, the programmed speed can be changed without having to stop and restart.

Immediate Stop

An immediate stop command stops all current motion.

Preset Current Position

Sets the internal position memory of the module to a position defined in the command word.

3.4.5.2 Stepper Motor Direction Management

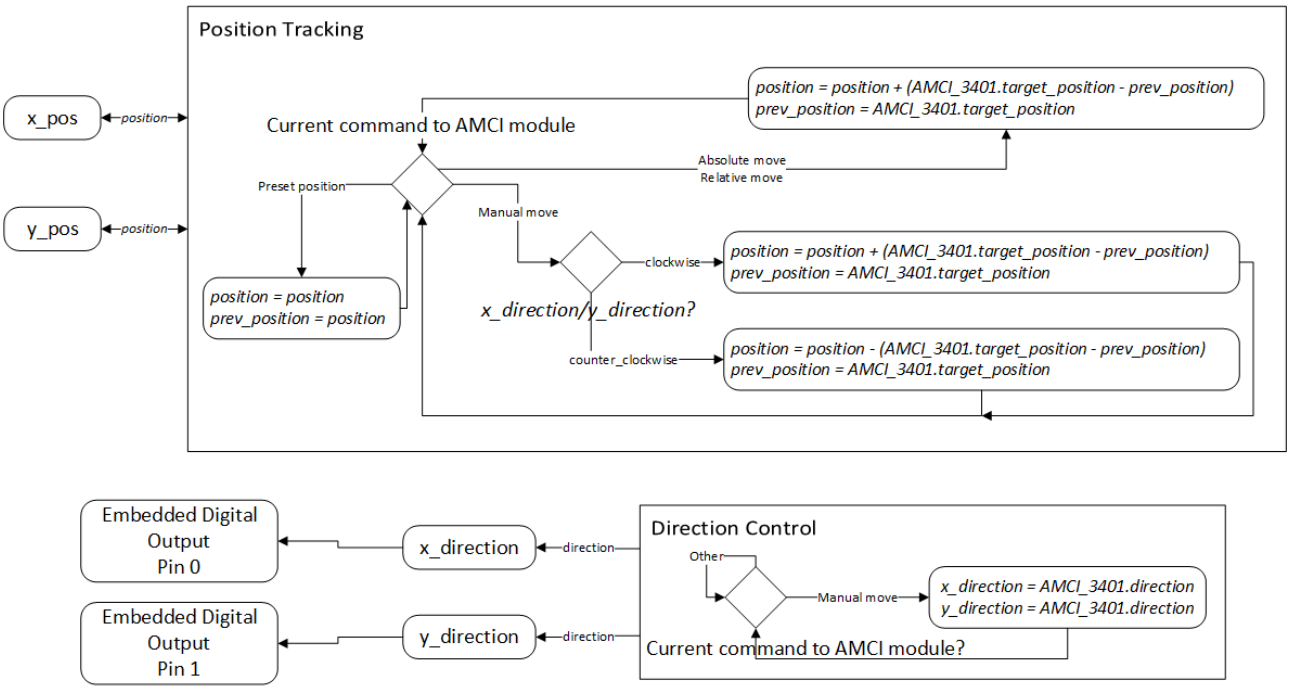

Figure 3.4.5: The two routines for managing direction and position

Empirical observation of the AMCI 3401 modules in operation showed a significant delay (in the order of 2-3 seconds) in between distinct commands. This delay is also present when switching between the Manual Move Clockwise and the Manual Move Counterclockwise states. In comparison, changing the programmed speed while in each state has negligible delay.
This behaviour is significant since the optimized velocity profiles require both axes to track the commands within the period of sampling (ideally set to 40-150ms for good results) to follow the required path.
To overcome this problem, the DoodleBot system controls the direction output via the PLC embedded DC output pins 0 and 1, and set manually through the motorControl program. To continue using Absolute Moves, this requires extra software functionality to keep track of the current position.

Absolute Move, Relative Move

Relative Moves are treated by the rest of the code as usual. Absolute Moves must be preceded by a Preset Current Position command with the position set to x_pos and y_pos. <blah> function monitors the directional state of the AMCI 3401 modules and mirrors this onto the direction output pins. <gah> function measures the change in position since last scan and adds this to x_pos and y_pos.

Manual Move

All manual moves, whether clockwise or anticlockwise, are sent to the AMCI 3401 modules as Manual Move Clockwise at the correct speed. The x_direction and y_direction bits need to be explicitly set

with each Manual Move command to operate in the required direction. Since the AMCI 3401 module thinks it is always moving clockwise (even when it's not), it's internal position state is no longer correct. Function <gah> measures the change in position since last scan and depending on the direction of movement, adds or subtracts this value to x_pos and y_pos.

Preset Current Position

Change the value of x_pos and y_pos to mirror the position state in the AMCI 3401 controller.

3.4.6 checkLimitSwitch

The checkLimitSwitch program is run in a Periodic Task of 10ms and monitors the four limit switches. Upon detecting a rising edge from one of these limit switches, it sets the appropriate boundaryFault value (boundaryFaultX for the X limit switches and boundaryFaultY for the Y limit switches).

If the boundaryFaultDisable is NOT set, it will assume that unintended operation has occurred and an immediate stop command to the stepper motor modules halting all operation. If boundaryFaultDisable IS set then this operation does not happen (used in a situation where motorControl or the Governor are expecting a limit switch input. Eg, boundary checking or waiting for user to manually toggle the switch).

3.5 User Interface

Intro

3.6 Java Program

The Java code is the backbone that interfaces between the separate modules of the Host system namely; the Graphical User Inferface (GUI), the MATLAB® optimisation, the profile server and the MATLAB® image processing. Each module has an associated class and the result of their interactions generates a desired sequence of velocity profiles to be enacted by the CNC machine.

3.7 MATLAB® Image Processing

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."

Edsger Dijkstra

4

Performance and Testing

Appendices

A

URBS Weight Derivation

The following [1] gives a series of derivations of the general case of the weights for URBS splines up to the third order. In general;

$$W_{i,d}(s) = \frac{s - K_i}{K_{i+d} - K_i} W_{i,d-1}(s) + \frac{K_{i+d+1} - s}{K_{i+d+1} - K_{i+1}} W_{i+1,d-1}(s)$$

$$W_{i,0}(s) = \begin{cases} 1 & \text{if } K_i \leq s < K_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

Using this recursive formula, we can build up results for first order URBS given $K_i \leq s < K_{i+1}$;

$$W_{i-2,0}(s), W_{i+1,0}(s), W_{i-1,0}(s) = 0$$
$$W_{i,0}(s) = 1$$

Second order;

$$
\begin{aligned}
W_{i,1} &= \frac{s - K_i}{K_{i+1} - K_i} W_{i,0}(s) + \frac{K_{i+2} - s}{K_{i+2} - K_{i+1}} W_{i+1,0}(s) \\
&= \frac{s - K_i}{K_{i+1} - K_i}
\end{aligned}
$$

$$
\begin{aligned}
W_{i-1,1}(s) &= \frac{s - K_{i-1}}{K_i - K_{i-1}} W_{i-1,0}(s) + \frac{K_{i+1} - s}{K_{i+1} - K_i} W_{i,0}(s) \\
&= \frac{K_{i+1} - s}{K_{i+1} - K_i}
\end{aligned}
$$

Third order;

$$
\begin{aligned}
W_{i,2}(s) &= \frac{s - K_i}{K_{i+2} - K_i} W_{i,1}(s) + \frac{K_{i+3} - s}{K_{i+3} - K_{i+1}} W_{i+1,1}(s) \\
&= \frac{s - K_i}{K_{i+2} - K_i} \frac{s - K_i}{K_{i+1} - K_i} \\
&= \frac{(s - K_i)^2}{(K_{i+2} - K_i)(K_{i+1} - K_i)}
\end{aligned}
$$

$$
\begin{aligned}
W_{i-1,2}(s) &= \frac{s - K_{i-1}}{K_{i+1} - K_{i-1}} W_{i-1,1}(s) + \frac{K_{i+2} - s}{K_{i+2} - K_i} W_{i,1}(s) \\
&= \frac{s - K_{i-1}}{K_{i+1} - K_{i-1}} \frac{K_{i+1} - s}{K_{i+1} - K_i} + \frac{K_{i+2} - s}{K_{i+2} - K_i} \frac{s - K_i}{K_{i+1} - K_i}
\end{aligned}
$$

$$
\begin{aligned}
W_{i-2,2}(s) &= \frac{s - K_{i-2}}{K_i - K_i} W_{i-2,1}(s) + \frac{K_{i+1} - s}{K_{i+1} - K_{i-1}} W_{i-1,1}(s) \\
&= \frac{K_{i+1} - s}{K_{i+1} - K_{i-1}} \frac{K_{i+1} - s}{K_{i+1} - K_i} \\
&= \frac{(K_{i+1} - s)^2}{(K_{i+1} - K_{i-1})(K_{i+1} - K_i)}
\end{aligned}
$$

These weightings define the linear combination of the control points creating a point on the curve;

$$
P(s) = \mathbf{N}_i^2 W_{i,2}(s) + \mathbf{N}_{i-1}^2 W_{i-1,2}(s) + \mathbf{N}_{i-2}^2 W_{i-2,2}(s)
$$

for $K_i \leq s < K_{i+1}$.

B

Derivation of $s^*(t)$ given $s^*(s)$

$$s(t) = \sum_{k=1}^{s_k < s(t_k)} \frac{2\Delta s}{\dot{s}_k^2 - \dot{s}_{k-1}^2}\left(\dot{s}_k - \dot{s}_{k-1}\right) + \left(\left(\frac{(t - t_{last})(\dot{s}_{next}^2 - \dot{s}_{last}^2)}{2\Delta s} + \dot{s}_k\right)^2 - \dot{s}_k^2\right)\frac{\Delta s}{(\dot{s}_{next}^2 - \dot{s}_{last}^2)}$$

$$\Delta t_k(s) = \int_{s_k}^{s} \frac{1}{\sqrt{\frac{\dot{s}(s_{k+1}) - \dot{s}(s_k)}{s_{k+1} - s_k}u + \dot{s}(s_k)}} du$$

References

[1] Markus Altmann. About nonuniform rational b-splines - nurbs, 1995. URL http://web.cs.wpi.edu/~matt/courses/cs563/talks/nurbs.htmf.

[2] A. Bardine, S. Campanelli, P. Foglia, and C.A. Prete. Nurbs interpolator with confined chord error and tangential and centripetal acceleration control. In *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 International Congress on*, pages 489–496, 2010. doi: 10.1109/ICUMT.2010.5676592.

[3] Shyh-Leh Chen and Kai-Chiang Wu. Contouring control of smooth paths for multiaxis motion systems based on equivalent errors. *Control Systems Technology, IEEE Transactions on*, 15(6):1151–1158, 2007. ISSN 1063-6536. doi: 10.1109/TCST.2007.899719.

[4] Dong-Soo Choi, Seung-Jean Kim, Yang-O Kim, and In-Joong Ha. A computationally efficient approach to time-optimal control of robotic manipulators along specified paths. In *Industrial Electronics, 2001. Proceedings. ISIE 2001. IEEE International Symposium on*, volume 3, pages 1569–1574 vol.3, 2001. doi: 10.1109/ISIE.2001.931940.

[5] Jan Foretnik. Nurbs demo, 2012. URL http://geometrie.foretnik.net/files/NURBS-en.swf.

[6] Sheng-Jung Tseng, Kuan-Yuan Lin, Jiing-Yih Lai, and Wen-Der Ueng. A nurbs curve interpolator with jerk-limited trajectory planning. *Journal of the Chinese Institute of Engineers*, 32(2):215–228, 2009. doi: 10.1080/02533839.2009.9671498. URL http://www.tandfonline.com/doi/abs/10.1080/02533839.2009.9671498.

[7] D. Verscheure, B. Demeulenaere, J. Swevers, J. De Schutter, and M. Diehl. Time-optimal path tracking for robots: A convex optimization approach. *Automatic Control, IEEE Transactions on*, 54(10): 2318–2327, 2009. ISSN 0018-9286. doi: 10.1109/TAC.2009.2028959.

[8] S.-S. Yeh and P.-L. Hsu. The speed-controlled interpolator for machining parametric curves. *Computer-Aided Design*, 31(5):349 – 357, 1999. ISSN 0010-4485. doi: http://dx.doi.org/10.1016/S0010-4485(99)00035-4. URL http://www.sciencedirect.com/science/article/pii/S0010448599000354.