

Redundant Kinematics Solution for a Combined 6DOF Robotic Manipulator and 2DOF Part Positioner in a WAAM Application

A Thesis Presented for the
Master of Science
Degree

The University of Tennessee, Knoxville

Ethan Vals

May 2022

© by Ethan Vals, 2022
All Rights Reserved.

To my parents for the constant encouragement and for always believing in me.

Acknowledgements

I would like to thank my family for always supporting me. Thanks to my mom for homeschooling me and giving me a great educational background so I could pursue engineering. Thanks to my dad for helping me with all my latest projects or hobbies. Thanks to Dr. Bennett for the guidance as I started engineering school and for helping me find an undergraduate research position. Thank you to Dr. Hamel for being my advisor and mentoring me through graduate school. Thanks to my committee for taking the time to advise me on my thesis. Thanks to Logan McNeil and Matthew Lamsey for all the time spent discussing my thesis topic, and also to Sam Robertson, Joshua Penney, and Shems-Eddine Belhout for supporting me through my research.

Abstract

A typical wire arc additive manufacturing (WAAM) robot cell consists of a 6 DOF robot manipulator and a 2 DOF part positioner. Since the WAAM process requires a minimum of 5 DOFs, there are three redundant DOFs in the system that can be utilized to improve the robot manipulator positioning during part printing. In this thesis, the redundant kinematics of a manipulator and part positioner robot system are solved and then implemented on an actual robot system. The inverse kinematics of the manipulator and part positioner are solved as a kinematic chain using the pseudo-inverse Jacobian method. The two DOFs in the part positioner are solved using a gravity aligned constraint function, which ensures that the welding torch is always vertically aligned and welding normal to the part path. The last DOF, the arbitrary z-rotation of the weld torch, is solved using the gradient projection method, which projects a cost function onto the nullspace of the Jacobian. A singularity avoidance cost function and a joint limit avoidance cost function were tested and compared. Additionally, the kinematics were solved with the z-rotation of the weld torch set at a fixed angle relative to the world base, effectively minimizing manipulator movement during the path. Comparing the solved joint angles shows that the singularity avoidance cost function struggled to create a smooth and collision free part path. The fixed torch rotation joint solutions gave the best robot position results, which were then tested on the actual robots and compared to the solutions found by a separate robot programming software.

Table of Contents

1	Introduction	1
2	Background	5
2.1	Kinematic Redundancy in Robotic Systems	5
2.2	Robot System Kinematics	7
2.3	Current Robot Programming Process	9
2.3.1	Path Planning Software	9
2.3.2	Robot Programming	9
2.3.3	Exported Robot Code in KUKA Robot Language	11
3	Iterative Inverse Kinematics Method for Solving Redundant DOFs	16
3.1	Problem Approaches	16
3.2	Kinematics Model of Robot Cell	17
3.2.1	Forward Kinematics of Manipulator and Part Positioner	17
3.2.2	Kinematic Chain of Robot Cell	22
3.2.3	Solving the Jacobian	22
3.3	Iterative Method with Newton's Method	26
3.4	Gravity Aligned Constraint	29
3.5	Solving Arbitrary Torch Rotation with the Jacobian Null Space	32
4	Implementation	34
4.1	Kinematic Solver MATLAB Program	34
4.1.1	Program Overview	34

4.1.2	KUKA Robot Language Exporter	37
4.1.3	Kinematic Solver Graphical User Interface	39
4.1.4	Inverse Kinematic Solver Effectiveness	40
4.2	Comparison of Kinematic Solver Solutions	40
4.2.1	Singularity Avoidance	44
4.2.2	Joint Limit Avoidance	44
4.2.3	Combined Singularity and Joint Limit Avoidance	46
4.2.4	Fixed Torch Rotation	49
4.3	Program Testing on KUKA Manipulator and Part Positioner	49
5	Conclusions and Future Work	55
5.1	Conclusions	55
5.2	Future Work	55
	Bibliography	57
	Appendices	60
A	Kinematic Solver MATLAB Code	61
B	Kinematic Functions MATLAB Code	70
	Vita	80

List of Tables

3.1	DH table for KUKA KR50	21
3.2	DH table for part positioner	21

List of Figures

1.1	KUKA KR50 manipulator and KUKA KP-2 part positioner in the University of Tennessee's hybrid manufacturing workcell	4
2.1	Redundant degrees of freedom in a manipulator and part positioner robot cell in a WAAM application	6
2.2	Forward kinematics versus inverse kinematics of a robot manipulator	8
2.3	MATLAB path planning program with an imported part	10
2.4	Robot manipulator and part positioner in Octopuz simulation	12
2.5	A manipulator with a spherical wrist can have the same inverse kinematic solution for a point with a wrist up position and a wrist down position	15
3.1	Joint coordinate frames for manipulator	19
3.2	Joint coordinate frames for part positioner	20
3.3	Kinematic chain of the complete robot cell	23
3.4	Gravity aligned constraint for the gravity vector and point normal vector	30
4.1	MATLAB inverse kinematic solver block diagram	35
4.2	MATLAB inverse kinematics solver GUI	41
4.3	Manipulator and part positioner figures in the MATLAB inverse kinematics solver	42
4.4	Propeller blade path contours	43
4.5	Robot joint solutions using singularity avoidance cost function	45
4.6	Robot joint solutions using joint limit avoidance cost function	47

4.7	Robot joint solutions using combined singularity avoidance and joint limit avoidance	48
4.8	Robot joint solutions with a fixed torch rotation	50
4.9	Octopuz robot program joint values recorded with RSI	51
4.10	MATLAB inverse kinematics program joint values recorded with RSI	54

Chapter 1

Introduction

Wire arc additive manufacturing (WAAM) is an emerging manufacturing method in which molten wire is deposited in layers using the MIG welding process. The WAAM process is categorized as a direct energy deposition (DED) additive manufacturing method, in which a metal wire is fed into a molten weld pool. Compared to other DED processes, the WAAM process is relatively simple to set up, as only a manipulator and a standard welding station using a Gas Metal Arc Welding (GMAW) or Gas Tungsten Arc Welding (GTAW) power source is required (13). The WAAM process provides several manufacturing advantages, such as increased part complexity, faster part completion times, and less material waste.

Robot manipulators are commonly used in WAAM applications, with the MIG welding torch attached to the end-effector of the robot. The manipulator provides a large workspace suitable for creating large scale additively manufactured parts. Additionally, a part positioner system is often used to orient the part while it is welded. The part positioner usually has two axis that are used to rotate the part so that the manipulator can approach from an optimal welding position. The part positioner is often used to orient the part so that only gravity aligned welding occurs, where the welding torch is kept vertically aligned throughout the part path.

Robot manipulators greatly increase the flexibility of the WAAM process, allowing larger and more complex geometries to be fabricated. The manipulator also allows non-gravity aligned welding to be performed, where the torch welds the part at an angled position. However, with the added flexibility there are added complexities in the robot

programming process, specifically in how to account for the redundant DOFs in the robot system. Redundancy occurs when the number of DOF required to define the robot position in the joint space is greater than those required to define an end-effector position in the operational space. A robot system with redundant DOFs, such as a manipulator and part positioner system, can be solved so that the geometric configuration of the system is optimized for a specific application, but this requires that the redundant DOFs first be solved.

Several methods have been developed for solving redundancies in a robotic system. Siciliano (10) discusses several of these methods, such as the gradient projection method and the task space augmentation method. Chang (2) discusses solving the inverse kinematics of a redundant system using the Lagrangian multiplier method while also applying cost functions to solve the redundant DOFs.

Several methods have been applied to specifically solving the redundant DOFs in a system using a part positioner. Xu and Yao (14) explain a process for solving the part positioner joint angles independently of the manipulator configuration, and Boz and Lazoglu (1) use a similar method for solving the part positioner configuration used in a five-axis milling machine. These methods solve the part positioner configuration using geometric, closed form inverse kinematic equations with constraints, such as the gravity aligned welding constraint, built into the equations. While this method is ideal for some applications, a more flexible solver is beneficial for some systems where different constraints will be used on the part positioner. This would also allow for the modification of the inverse kinematics solver in which non-gravity aligned welding is used and the manipulator and part positioner are not independent of each other. Cook (Cook et al.) explains a method for solving a multiple robot system in which the kinematics of both robots are solved simultaneously.

When solving the inverse kinematics for a redundant system, cost or constraint functions must be defined in order to direct the solver to a solution. (Cook et al.) describes a method for applying the gravity aligned welding constraint in which the Jacobian is extended to include additional constraint functions. Cost functions can also be used to account for redundant DOFs. Cost functions can be applied to the gradient projection method or the task space augmentation method described by (10) and also to the lagrangian method

discussed by (2). (15) developed the manipulability measure which can be used as a singularity avoidance cost function. (7) implements a joint limit avoidance cost function with the gradient projection method. (8) discusses a cost function for minimizing the rate of change of the Jacobian.

For this thesis, the inverse kinematics of a redundant manipulator and part positioner system used in a WAAM application are solved and then implemented. An inverse kinematics program is developed in MATLAB which can solve robot positions using different constraint and cost functions. The inverse kinematics of the robot system and the part positioner constraint function are solved using the method described in (Cook et al.), and various cost functions are used with the gradient projection method described by (10). The solved robot positions for a specific part plan are then implemented for the University of Tennessee's hybrid manufacturing cell, which includes a KUKA KR50 manipulator and a KUKA KP-2 part positioner system used for the WAAM process.

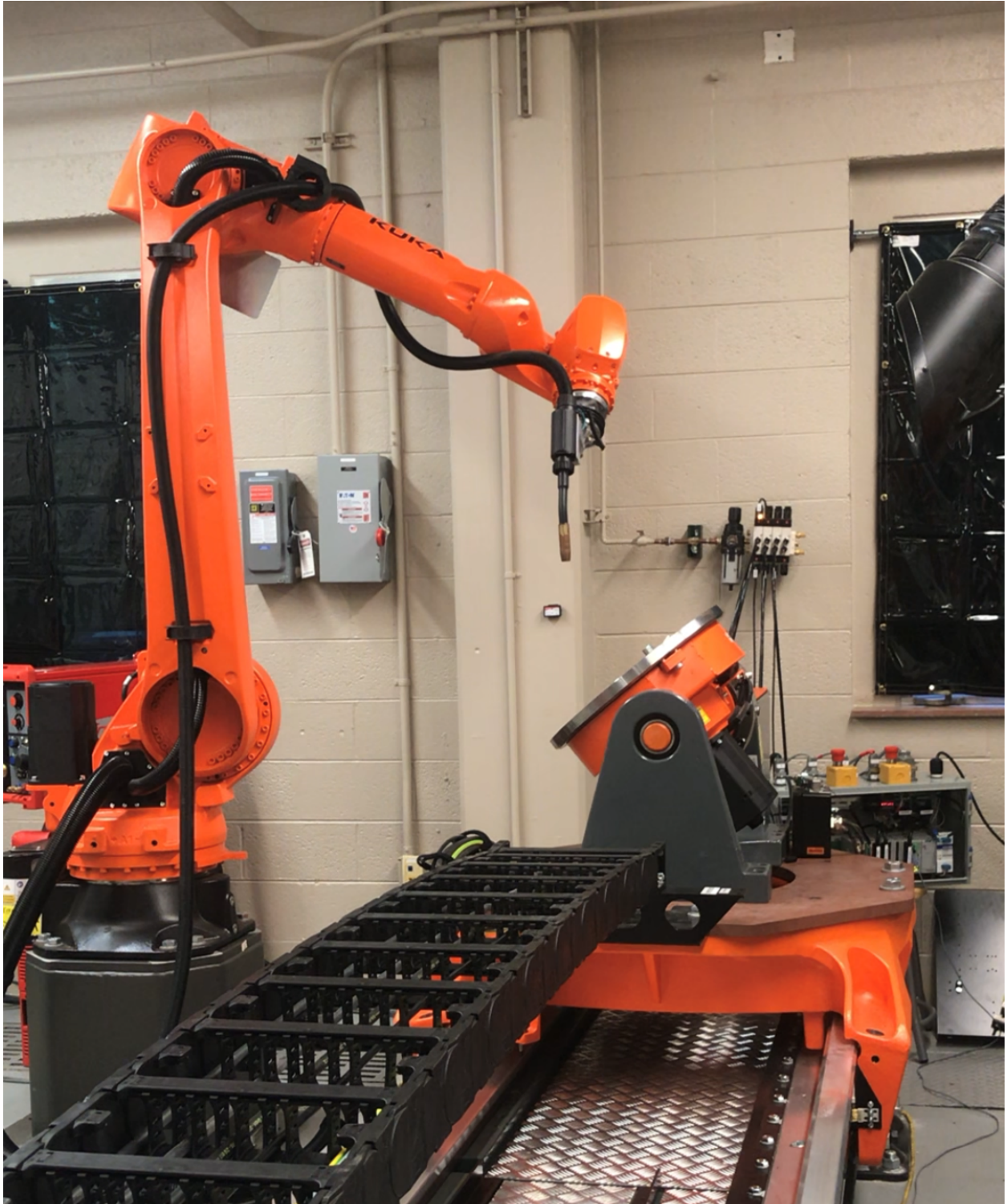


Figure 1.1: KUKA KR50 manipulator and KUKA KP-2 part positioner in the University of Tennessee's hybrid manufacturing workcell

Chapter 2

Background

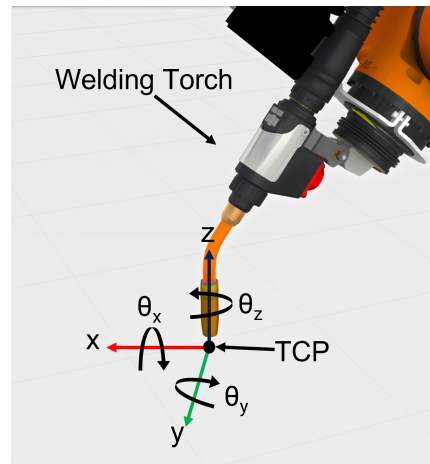
2.1 Kinematic Redundancy in Robotic Systems

A manipulator is kinematically redundant if the manipulator has a higher number of DOFs than the task requires. If n is the manipulator dimensions in the joint space, m is the manipulator dimensions in the operational space, and r is the task dimensions in the operational space, then a manipulator is intrinsically redundant if $m < n$ and redundant with respect to the task if $r < m$ (11). In order to position a point in space, six DOF are required and $r = 6$. The six operational space dimension in the task space are the translations, x, y, z , and the rotations about the three axis A, B, C . A typical six DOF industrial manipulator has $n = m$ and is not intrinsically redundant, so it is able to position a point in the operational space without redundancy.

Considering the six DOF manipulator and two DOF part positioner system, both robots have $n = m$ so the system has a total of eight DOF in the operational space. For most WAAM processes, $r = 5$ since the z-rotation of the torch is arbitrary in a welding application. Therefore $r < m$ and the robot system is redundant and has three redundant DOFs. For some WAAM processes, such as with a dual wire feeding torch, the torch must be oriented with respect to the path. Consequently, the rotation of the torch is not arbitrary. For this case, there are only two redundant DOFs.



$$6 \text{ DOF Manipulator} + 2 \text{ DOF Part Positioner} = 8 \text{ DOF}$$



$$6 \text{ DOF} - \text{Arbitrary } \theta_z = 5 \text{ DOF}$$

$$8 \text{ DOF} - 5 \text{ DOF} = 3 \text{ Redundant DOF}$$

Figure 2.1: Redundant degrees of freedom in a manipulator and part positioner robot cell in a WAAM application

2.2 Robot System Kinematics

To accurately position a robot, the relationship between the robot joint variables, q , and the robot end position and orientation, x_e , with p_e being the translation vector and ϕ_e being the orientation vector, must be found.

$$q = \begin{bmatrix} q_1 \\ \vdots \\ q_n \end{bmatrix} \quad (2.1)$$

$$x_e = \begin{bmatrix} p_e \\ \phi_e \end{bmatrix} \quad (2.2)$$

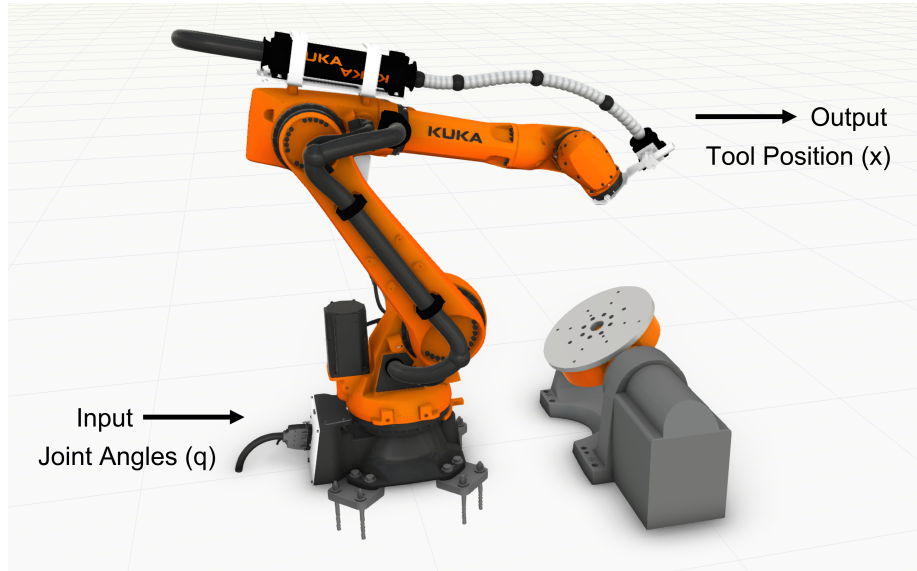
The direct kinematics function, $k(q)$, can be calculated from the kinematic transformation matrices between the robot joints and can be used to calculate the end position in the operational space based on the current joint values.

$$x_e = k(q) \quad (2.3)$$

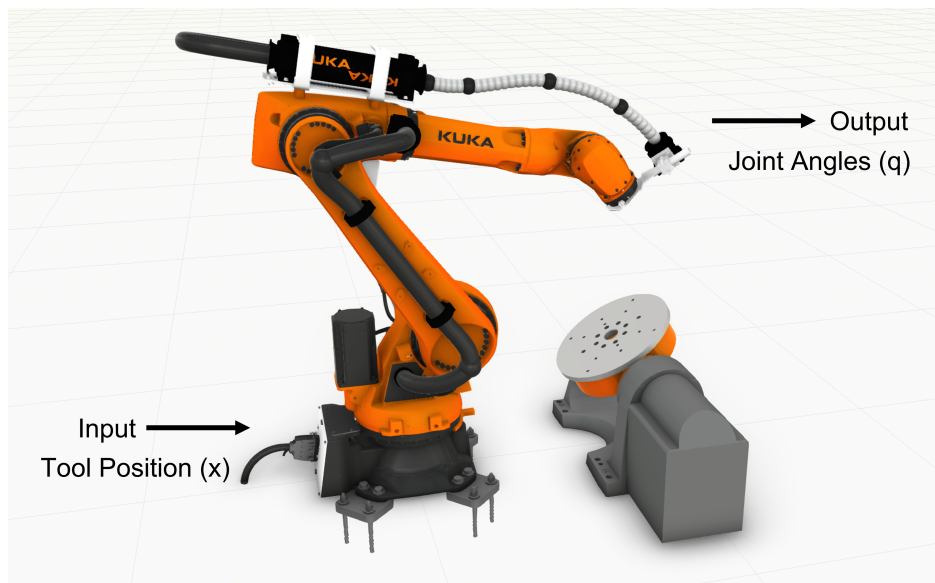
Typically, the inverse kinematics problem is of more value, where the joint angle values are found for a desired end position and orientation. The inverse kinematics problem is more complex than the forward kinematics problem, as it may not be possible to find a closed form solution and there are usually multiple or even infinite many possible solutions, as in the case of a redundant manipulator.

The inverse kinematics problem in principle can be solved using either a closed form solution or a numerical solution. A closed form solution can be found either algebraically from the direct kinematic functions or geometrically from the structure of the manipulator. A method must then be developed to choose one of the multiple solutions calculated. In the case of a redundant manipulator with infinite many solutions, constraints must be applied to the joint positions in order to find a possible solution.

The numerical solution to the inverse kinematics problem can be applied to any manipulator without utilizing closed form methods. The solution is found iteratively and



(a) Forward Kinematics



(b) Inverse Kinematics

Figure 2.2: Forward kinematics versus inverse kinematics of a robot manipulator

converges on a solution based on the initial conditions and solution criteria used in the numerical method. While this method is more computationally expensive than the closed form solution, it allows more flexibility when solving a redundant kinematic system in an optimization framework, as the desired cost and constraint functions can be applied to find the most desirable solution.

2.3 Current Robot Programming Process

The purpose of this thesis is to simplify the current robot programming process for creating WAAM parts at the University of Tennessee. The current process consists of creating the part path plan, using a robot simulation program to output robot code for the path, and then testing and running the robot code on the actual robots.

2.3.1 Path Planning Software

First, the path plan for the WAAM part is created. Contour layers are created for the desired part and then imported into a custom MATLAB path planning software, shown in Figure 2.3. The software runs through each point of the contours and assigns them a movement order with respect to the other points. Different settings can be applied to the part, such as combining collinear moves or alternating contour direction. Additionally, the torch orientation can be set to be at either a fixed orientation or normal to the path build direction. Once the desired settings are applied, the part path is exported as a PATH text file which contains the position, orientation, and velocity values as well as the movement type (linear or point-to-point) for each point.

2.3.2 Robot Programming

The PATH file is then imported into Octopuz[™] (9), a robot simulation program. The path is placed in the virtual environment and connected to the robot manipulator. The simulation can then be run, and the robot will move through the imported path at the input orientations and velocities. It must be checked that no robot collisions occur and that the robot maintains

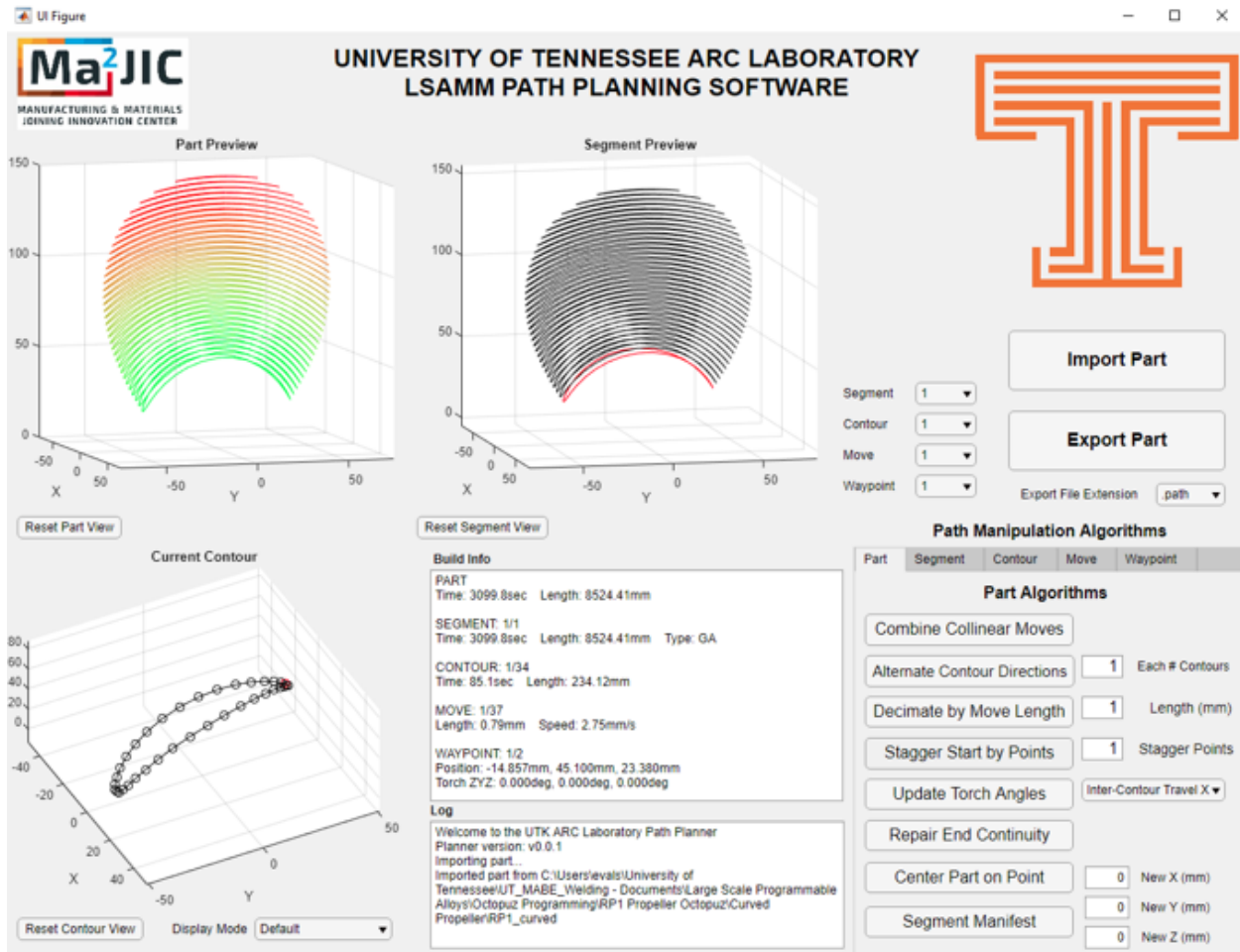


Figure 2.3: MATLAB path planning program with an imported part

a desirable orientation throughout the path. If the robot moves into an undesirable position, that point must be manually adjusted and then re-tested. The weld settings, base data, tool data, and time delay between contours must all be applied as well.

The robot manipulator can also be connected with a part positioner in Octopuz. Different movement settings can be applied to the part positioner. For example, the part positioner can be programmed to keep the torch vertically aligned throughout the path. The torch rotation used with the part positioner can also be modified to either follow the input point orientation, stay at a fixed angle with respect to the world base, or rotate freely to minimize robot movement.

2.3.3 Exported Robot Code in KUKA Robot Language

Finally, Octopuz exports the programmed robot motions as KUKA Robot Language (KRL), which can be uploaded to the KUKA controller and run on the actual robots. A KRL program consist of moves between robot positions. These moves are defined as either linear or point-to-point moves. A linear move moves the robot tool center point (TCP) at a constant set velocity in a straight line between two points. A point-to-point move takes the shortest path in the joint space to the next point. The velocity of a point-to-point move is set as a percentage value of the maximum speed it will take the robot to complete the move between points.

Two types of robot position types can be sent to the robot in KRL. First, the E6AXIS KRL position type sends the joint angle values directly to the robot. The E6AXIS position type can only be used for point-to-point moves. Second, the E6POS KRL position type consists of the x, y, z, A, B, C values that the robot tool center point (TCP) must move to, as well as the two external axis joint angles for the part positioner. The E6POS position type can be used for either linear or point-to-point type moves.

The benefit of the E6POS position type is that it positions the robot based on the set base position. The base position is manually set on the robot controller and represents the coordinate frame that the path points are referenced to. If the base position set on the robot controller and the base position set on the robot position solver (Octopuz in this case) are slightly different, then the robot can account for this change in position because it solves



Figure 2.4: Robot manipulator and part positioner in Octopuz simulation

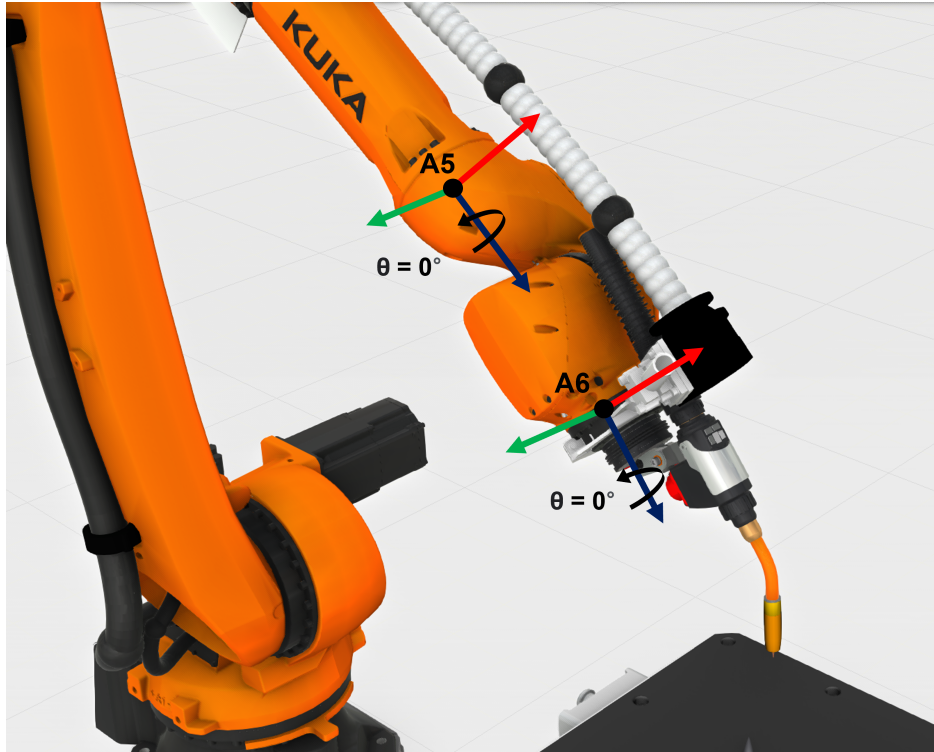
the inverse kinematics itself for the input x, y, z, A, B, C values with respect to the set base on the robot controller. If the axis joint values for every point are sent to the robot as the E6AXIS position type, then the robot will move to the set joint angles and disregard the base position.

The current process Octopuz uses for creating KRL code uses E6POS position types for all the path points. The approach, retract, and in-between contour points are all point-to-point motions, and the path points within the part contour are all linear moves. For a welding application, the velocity of the weld torch must be controlled, so only linear moves are used on the part path.

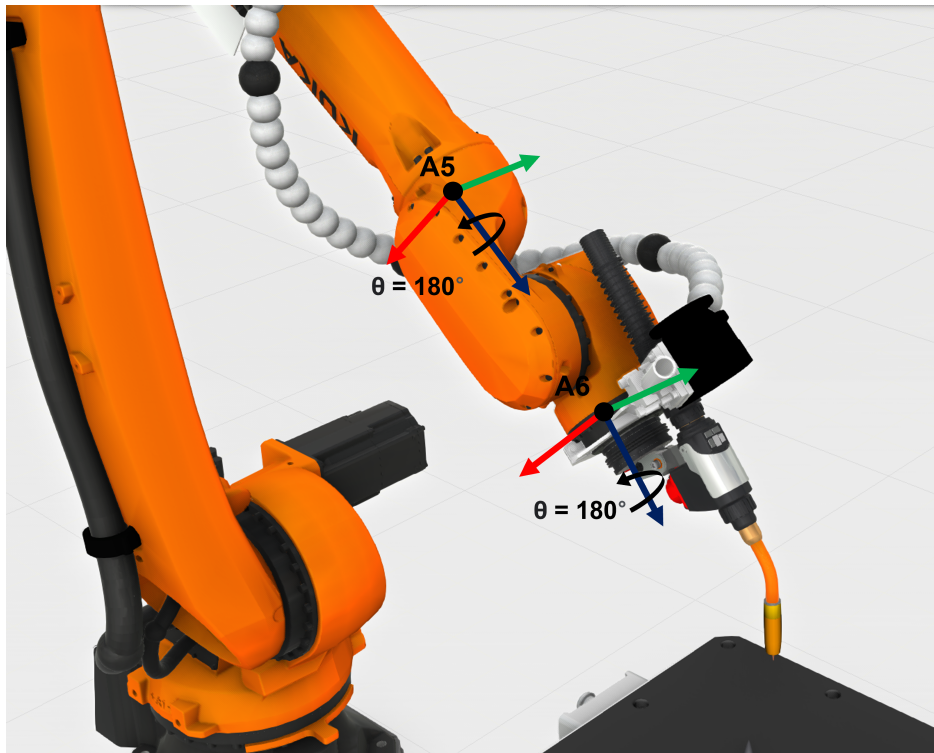
Since the KUKA robot receives only E6POS type positions, the KUKA controller must solve the inverse kinematics itself while running the program on the physical robot, and since the Octopuz and the KUKA controller do not use the exact same inverse kinematics solver, it is not guaranteed that the solved robot position on the physical robot will be the same as the Octopuz simulation robot position. Both inverse kinematics solvers take into account the joint limit constraints of the robot and will generally give identical solutions with a standard elbow up and forward facing robot configuration. For linear moves, the torch tip position is always maintained and the robot will not deviate from its initial configuration. However, with point-to-point moves the wrist orientation of the robot is not always consistent. For some part geometries, the robot wrist axes become close to their joint limits, resulting in the wrist of the robot flipping over on the in-between contour, point-to-point moves, in order to move the wrist to a more central joint axis position. This can result in unexpected robot motion that could potentially cause a collision between the robot and the part. Therefore, it is required to test run the path on the actual robot at a lower velocity with welding off to insure the robot wrist does not flip over between contours.

The custom inverse kinematics solver discussed in this thesis will be implemented into the path planner software code in MATLAB so that the robot positions and robot code can be generated automatically. The program will generate the KRL code based on the solved robot positions and part point values. In order to prevent unexpected robot motion, such as the wrist flipping over, the point-to-point moves between contours will be set as E6AXIS type positions, where the pre-calculated robot joint angles are sent directly to the robot. The

linear moves on the part path will then be sent as E6POS position types and will maintain the initial configuration from the previous point-to-point move.



(a) Wrist Up Position



(b) Wrist Down Position

Figure 2.5: A manipulator with a spherical wrist can have the same inverse kinematic solution for a point with a wrist up position and a wrist down position

Chapter 3

Iterative Inverse Kinematics Method for Solving Redundant DOFs

3.1 Problem Approaches

For the manipulator and part positioner WAAM system, a numerical method for solving the inverse kinematics is the ideal choice to support off-line programming for WAAM part trajectories. It is possible to calculate the joint angles for a manipulator and part positioner geometrically, as discussed by (14) and (1). However, this method calculates the manipulator and part positioner joint angles in two steps to insure the torch angle is always vertical for gravity aligned welding. First the part positioner joints are solved for, and then the manipulator kinematics are solved for the new part positioner position. This reduces the flexibility of the inverse kinematics solver, as it cannot be used for more complicated part welds (such as very large parts) where it might be desirable for the torch to be non-gravity-aligned. Additionally, this method does not take advantage of the extra DOF provided by the rotation of the torch on the z-axis. The numerical method allows the manipulator and part positioner to be solved simultaneously as a combined kinematic system. Simultaneous solution allows for custom cost functions and constraint functions to be applied to the whole system that can vary the desired torch angle and rotation for the desired application.

A popular numerical approach to the inverse kinematics problem makes use of the pseudo-inverse of the Jacobian to iteratively find the joint displacements from the joint velocity values

until the desired end-effector position is achieved. The pseudo-inverse of the Jacobian can be used for the inverse Jacobian in the case of a redundant manipulator that has a non-square Jacobian, as suggested by Whitney in (12). The psuedo-inverse can be calculated using (3.1).

$$J^\dagger = J^T(JJ^T)^{-1} \quad (3.1)$$

(10) describes using the pseudo-inverse Jacobian to solve redundant robot system, and (Cook et al.) describes this method in detail for a multiple robot system. For this thesis, the pseudo-inverse Jacobian method is used to solve the manipulator and part positioner system inverse kinematics, as it is relatively simple to modify the solution to incorporate different constraint and cost functions for the desired application. For use in a WAAM application, different part geometries might require different robot kinematic solutions, so it is important that the cost and constraint functions used in the solver be easily modified. For example, for some parts, the z-axis rotation of the torch is arbitrary, whereas some parts may use a dual wire weld torch, where the z-axis rotation of the torch is not arbitrary.

3.2 Kinematics Model of Robot Cell

Before solving the inverse kinematics of the robot system, a kinematic model for the robots and the transformations between them must be found. The kinematic model contains the transformation matrices between each joint of the robots, as well as the translations between the robots to the base position and the desired point position. The Denavit-Hartenberg convention is used to find the forward kinematics matrices for the manipulator and part positioner, and then transformation matrices are used to relate the robot positions and form a kinematic chain of the entire robot cell. Finally, the Jacobian is found for the complete kinematic chain.

3.2.1 Forward Kinematics of Manipulator and Part Positioner

The DH convention was used to find the forward kinematic equations for the manipulator and the part positioner. The DH convention provides a systematic method for assigning

coordinate frames to robot joints and defining the transformations between them and is commonly used in describing industrial robot kinematics (11). The coordinate frames are defined for each joint using the following steps.

1. Choose the z-axis to be aligned with the joint axis.
2. Choose the origin to be at the intersection of the common normal of the current z-axis and the previous z-axis.
3. Choose the x-axis to be along the common normal between the two z-axis, pointing in the direction of the current z-axis.
4. Choose the y-axis so that a right-handed frame is created.

Using these steps, the coordinate frames were assigned to the joints of the manipulator and part positioner as shown in Figure 3.1 and Figure 3.2.

The transformation matrices between the joint coordinate frames must then be found by defining the following parameters.

- a_i is the distance from the previous origin to the current origin along the new x-axis.
- d_i is the distance from the previous origin to the current origin along the previous z-axis.
- α_i is the angle between the old and new z-axis about the new x-axis.
- θ_i is the angle between the old and new x-axis about the old z-axis.

The DH parameters for the manipulator and part positioner are formulated in DH tables as shown in Table 3.1 and Table 3.2.

Since all of the joints in the robot system are revolute, then θ_i is the joint variable. The final transformation matrices are found from the composition of the two rotation matrices (3.3) and (3.4).

$$A_i^{i-1}(q_i) = A_{i'}^{i-1} A_i^{i'} \quad (3.2)$$

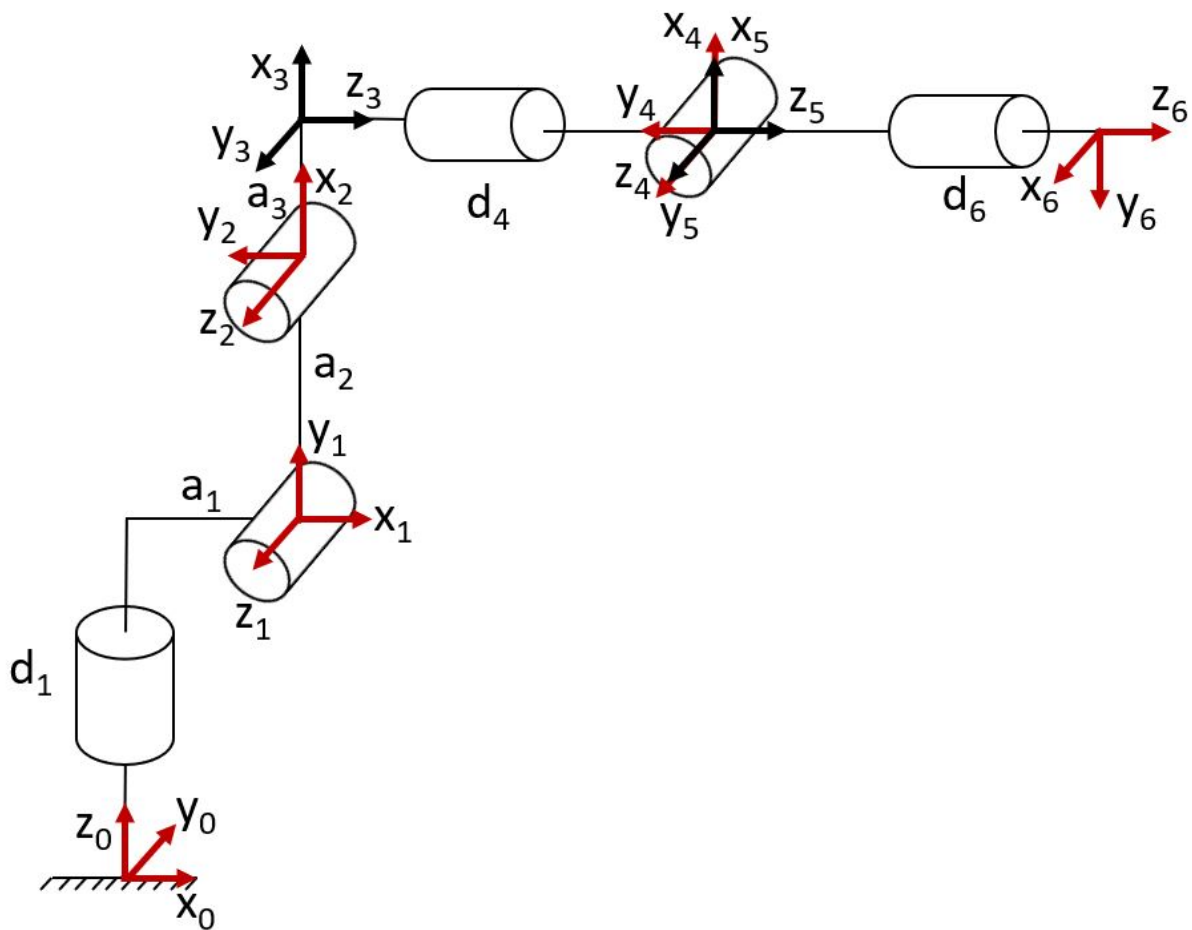


Figure 3.1: Joint coordinate frames for manipulator

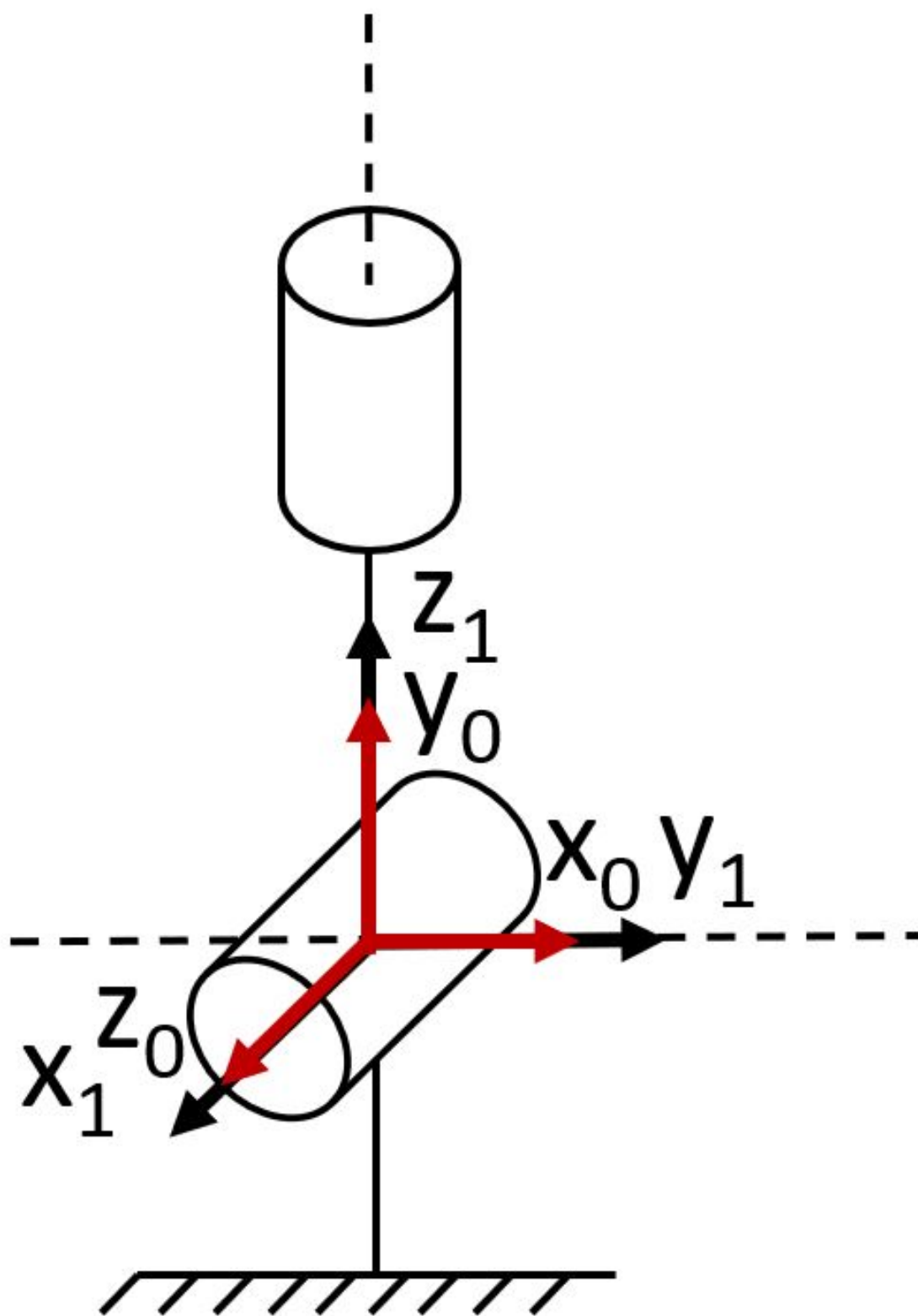


Figure 3.2: Joint coordinate frames for part positioner

Table 3.1: DH table for KUKA KR50

Link	a_i	α_i	d_i	θ_i
1	a_1	$\pi/2$	d_1	θ_1
2	a_2	0	0	θ_2
3	a_3	$\pi/2$	0	θ_3
4	0	$-\pi/2$	d_4	θ_4
5	0	$\pi/2$	0	θ_5
6	0	0	d_6	θ_6

Table 3.2: DH table for part positioner

Link	a_i	α_i	d_i	θ_i
1	0	$-\pi/2$	0	θ_1
2	0	0	0	θ_2

$$A_{i'}^{i-1} = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & 0 \\ \sin(\theta_i) & \cos(\theta_i) & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.3)$$

$$A_i^{i'} = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & \cos(\alpha_i) & -\sin(\alpha_i) & 0 \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.4)$$

3.2.2 Kinematic Chain of Robot Cell

The kinematics of the robot manipulator and part positioner and the transformations between them can be combined to form a closed kinematic chain with respect to positions on the workpiece being considered. The transformations are illustrated in Figure 3.3, where $T_R(i)$ is the forward kinematics of the manipulator, E is the torch translation, Z_P is the base translation to the part positioner, $T_P(i)$ is the part positioner forward kinematics, G is the workpiece translation, $P(i)$ is the point translation, and Err is the error translation between the torch tip and the desired point.

3.2.3 Solving the Jacobian

The Jacobian of a robot system describes the linear relationship between the joint velocities and the end-effector linear and angular velocities, as shown in (3.5). The columns of the Jacobian matrix describe the contributions of each joint velocity to the end-effector velocities, which are described by the rows of the Jacobian. Therefore, a non-redundant manipulator will have a square Jacobian with an equal number of joint DOFs as required operational space DOFs, and a redundant manipulator will have a Jacobian with more columns than rows.

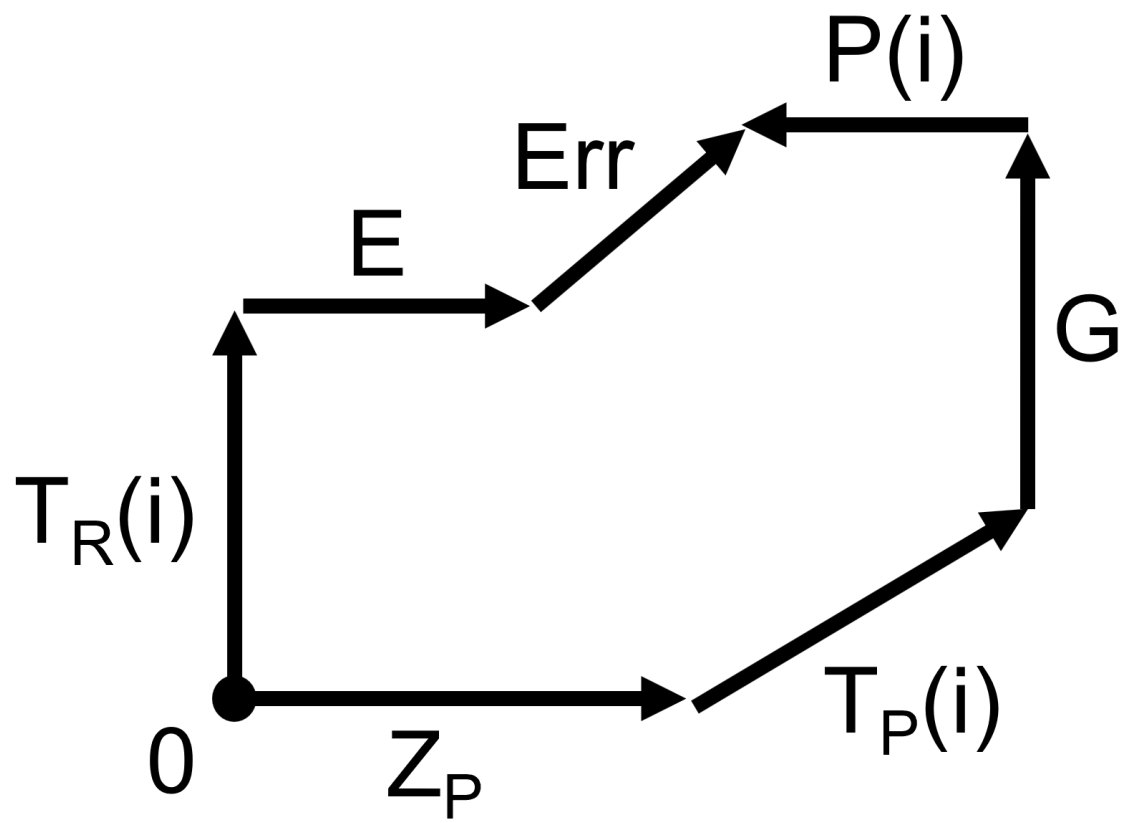


Figure 3.3: Kinematic chain of the complete robot cell

$$v_e = \begin{bmatrix} \dot{p}_e \\ \omega_e \end{bmatrix} = J(q)\dot{q} \quad (3.5)$$

The Jacobian can be calculated using the following equations depending on if the robot joint is prismatic or revolute. For the manipulator and part positioner problem, all joints are revolute. The translation and rotational rows of the Jacobian are solved separately and then combined to form the complete Jacobian.

$$\begin{bmatrix} J_{Pi} \\ J_{Oi} \end{bmatrix} = \begin{cases} \begin{bmatrix} z_{i-1} \\ 0 \end{bmatrix} & \text{for a prismatic joint} \\ \begin{bmatrix} z_{i-1} \times (p_e - p_{i-1}) \\ z_{i-1} \end{bmatrix} & \text{for a revolute joint} \end{cases} \quad (3.6)$$

The z_{i-1} value is found as the first three rows of the third column of the rotation matrix, the p_{i-1} value is the first three rows of the fourth column of the translation matrix, and the p_e value is the first three rows of the fourth column of the transformation matrix to the end-effector, as shown in (3.7), (3.8), and (3.9) respectively, where $z_0 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$ selects the third column of the rotation matrix and $\tilde{p}_0 = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T$ select the fourth column of the transformation matrix.

$$z_{i-1} = R_1^0(q_1) \dots R_{i-1}^{i-2}(q_{i-1})z_0 \quad (3.7)$$

$$\tilde{p}_{i-1} = A_1^0(q_1) \dots A_{i-1}^{i-2}(q_{i-1})\tilde{p}_0 \quad (3.8)$$

$$\tilde{p}_e = A_1^0(q_1) \dots A_n^{n-1}(q_n)\tilde{p}_0 \quad (3.9)$$

The Jacobian can then be found for the entire kinematic chain, incorporating both the manipulator and the part positioner, using (3.6). The Jacobian for the robot system will be a $[6 \times 8]$ matrix using all eight joint angles. The first two columns of the Jacobian are the

part positioner joint velocity contributions to the end-effector velocity, and the third column to the eighth column are the six manipulator joint velocity contributions to the end-effector velocity.

$$J = \begin{bmatrix} v_1 & v_2 & \dots & v_8 \\ w_1 & w_2 & \dots & w_8 \end{bmatrix} \quad (3.10)$$

$$v_1 = -OG^{-1}A_{2P}^{-1} \cdot [w \times (A_{1P}^{-1}Z_P^{-1}T_R(i)Er)] \quad (3.11)$$

$$v_2 = -OG^{-1}A_{2P}^{-1}A_{1P}^{-1} \cdot [w \times (Z_P^{-1}T_R(i)Er)] \quad (3.12)$$

$$v_3 = OG^{-1}T_P^{-1}(i)Z_P^{-1} \cdot [w \times (A_{1R}A_{2R}A_{3R}A_{4R}A_{5R}A_{6R}Er)] \quad (3.13)$$

$$v_4 = OG^{-1}T_P^{-1}(i)Z_P^{-1} \cdot A_{1R} \cdot [w \times (A_{2R}A_{3R}A_{4R}A_{5R}A_{6R}Er)] \quad (3.14)$$

$$v_5 = OG^{-1}T_P^{-1}(i)Z_P^{-1} \cdot A_{1R}A_{2R} \cdot [w \times (A_{3R}A_{4R}A_{5R}A_{6R}Er)] \quad (3.15)$$

$$v_6 = OG^{-1}T_P^{-1}(i)Z_P^{-1} \cdot A_{1R}A_{2R}A_{3R} \cdot [w \times (A_{4R}A_{5R}A_{6R}Er)] \quad (3.16)$$

$$v_7 = OG^{-1}T_P^{-1}(i)Z_P^{-1} \cdot A_{1R}A_{2R}A_{3R}A_{4R} \cdot [w \times (A_{5R}A_{6R}Er)] \quad (3.17)$$

$$v_8 = OG^{-1}T_P^{-1}(i)Z_P^{-1} \cdot A_{1R}A_{2R}A_{3R}A_{4R}A_{5R} \cdot [w \times (A_{6R}Er)] \quad (3.18)$$

$$w_1 = -OG^{-1}A_{2P}^{-1}w \quad (3.19)$$

$$w_2 = -OG^{-1}A_{2P}^{-1}A_{1P}^{-1}w \quad (3.20)$$

$$w_3 = OG^{-1}T_P^{-1}(i)Z_P^{-1}w \quad (3.21)$$

$$w_4 = OG^{-1}T_P^{-1}(i)Z_P^{-1}A_{1R}w \quad (3.22)$$

$$w_5 = OG^{-1}T_P^{-1}(i)Z_P^{-1}A_{1R}A_{2R}w \quad (3.23)$$

$$w_6 = OG^{-1}T_P^{-1}(i)Z_P^{-1}A_{1R}A_{2R}A_{3R}w \quad (3.24)$$

$$w_7 = OG^{-1}T_P^{-1}(i)Z_P^{-1}A_{1R}A_{2R}A_{3R}A_{4R}w \quad (3.25)$$

$$w_8 = OG^{-1}T_P^{-1}(i)Z_P^{-1}A_{1R}A_{2R}A_{3R}A_{4R}A_{5R}w \quad (3.26)$$

$$w = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^T \quad (3.27)$$

$$r = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T \quad (3.28)$$

$$O = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (3.29)$$

3.3 Iterative Method with Newton's Method

The error translation between the end-effector position of the welding torch and the desired workpiece point position can be calculated from the kinematic chain.

$$Err = E^{-1}T_R^{-1}(i)Z_PT_P(i)GP(i) \quad (3.30)$$

The resulting transformation matrix can be simplified using the first terms of the Taylor series expansion to provide linear approximations for the sin and cosine terms. The position errors and rotation errors can then be represented by vectors as shown in (3.32) and (3.33).

$$Err = \begin{bmatrix} 1 & -d\theta_z & d\theta_y & dx \\ d\theta_z & 1 & -d\theta_x & dy \\ -d\theta_y & d\theta_x & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.31)$$

$$u_{err} = \begin{bmatrix} dx \\ dy \\ dz \\ 1 \end{bmatrix} \quad (3.32)$$

$$\phi_{err} = \begin{bmatrix} d\theta_x \\ d\theta_y \\ d\theta_z \\ 0 \end{bmatrix} \quad (3.33)$$

When both the error vectors are zero, then the torch will be aligned with the desired point. Therefore, a forcing function, or objective function, can be created from the position and orientation error vectors. The vectors are transformed to the point frame using (3.34), and the forcing function is found as a 6x1 matrix in (3.35), where O is the same as (3.29) and is used to remove the constant terms from the error vectors.

$$Act = G^{-1}T_P^{-1}(i)Z_P^{-1}T_R(i)E \quad (3.34)$$

$$f6_{perr} = \begin{bmatrix} OActu_{err} \\ OAct\phi_{err} \end{bmatrix} \quad (3.35)$$

The forcing function (3.35) can then be used with Newton's method to iteratively solve for the joint angle values were the forcing function becomes zero.

$$\theta_{j_1} = \theta_j - [\frac{\partial}{\partial \theta} f6p_{err}(\theta_j)]^{-1} f6p_{err}(\theta_j) \quad (3.36)$$

Since the derivative of the positions and orientations with respect to the joint angles is the Jacobian, the equation can be rewritten as (3.38). Since the Jacobian is a 6×8 matrix, the pseudo-inverse is taken instead of the inverse. The psuedo-inverse Jacobian is multiplied by a scalar, h , to determine the iteration step sizes. The algorithm converges when the error meets a convergence criteria, ϵ , using (3.39), where K is a weighting matrix.

$$\frac{\partial}{\partial \theta} f6p_{err}(\theta_j) = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} \cdots \frac{\partial x}{\partial \theta_8} \\ \frac{\partial y}{\partial \theta_1} \cdots \frac{\partial y}{\partial \theta_8} \\ \frac{\partial z}{\partial \theta_1} \cdots \frac{\partial z}{\partial \theta_8} \\ \frac{\partial \phi_x}{\partial \theta_1} \cdots \frac{\partial \phi_x}{\partial \theta_8} \\ \frac{\partial \phi_y}{\partial \theta_1} \cdots \frac{\partial \phi_y}{\partial \theta_8} \\ \frac{\partial \phi_z}{\partial \theta_1} \cdots \frac{\partial \phi_z}{\partial \theta_8} \end{bmatrix} = J \quad (3.37)$$

$$\theta_{j_1} = \theta_j - hJ^\dagger f6p_{err}(\theta_j) \quad (3.38)$$

$$[f6p_{err}(\theta_j)]^T K [f6p_{err}(\theta_j)] < \epsilon \quad (3.39)$$

Using Newton's method to solve the inverse kinematics problem is equivalent to using the psuedo-inverse Jacobian method. This method uses the relationship between the Jacobian, the joint velocities, and the end-effector velocities, as shown in (3.40). By estimating the velocities to be position difference values, the small change in joint positions can be iteratively calculated and added to the current joint values until the solution converges. Comparing (3.40), which can be used to solve for the change in joint angle, to (3.38), it can be seen that the pseudo-inverse Jacobian method is the same as the right side of the Newton's method equation, which is just the pseudo-inverse of the Jacobian multiplied by the error vector, which can be represented as the change in position of the x, y, z, A, B, C end-effector values. Therefore, the iterative methods are equivalent.

$$\dot{q} = J^\dagger(q)\dot{x} \quad (3.40)$$

3.4 Gravity Aligned Constraint

Now that the inverse kinematics algorithm has been found for the manipulator and part positioner system, additional constraints and cost functions can be added to take advantage of the extra three DOFs. First, a constraint will be added to insure only gravity aligned welding occurs throughout the part path plan. This means that the path normal vector will always be aligned with the gravity vector and as a result the torch will always remain vertical to the path. Weld beads deposited in the gravity aligned direction generally have a more consistent geometry, as the gravity force will cause "drooping" and undesirable bead geometry for a non-gravity aligned weld without modified weld parameters (4). Using the part positioner to insure only gravity aligned welding enhances the successful creation of the desired part geometry.

In order to enforce the gravity aligned constraint, a forcing function term must be created that moves towards zero as the constraint is satisfied. One option is to take the cross product between the vector normal to the path and the gravity vector. When these two vectors, shown in Figure 3.4, are aligned, then their cross product will be zero. The gravity vector, g , will be a constant vector pointed in the base z-direction for this case, and the path normal vector, n_g , is calculated from the path points in the path planner.

$$g = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (3.41)$$

$$f_g = n_g(i) \times g \quad (3.42)$$

Now that the forcing function is found for the gravity aligned constraint, it can be added to the forcing function matrix. However, since the gravity vector is always aligned with the z-axis, the resulting cross product between the gravity vector and the path normal vector will

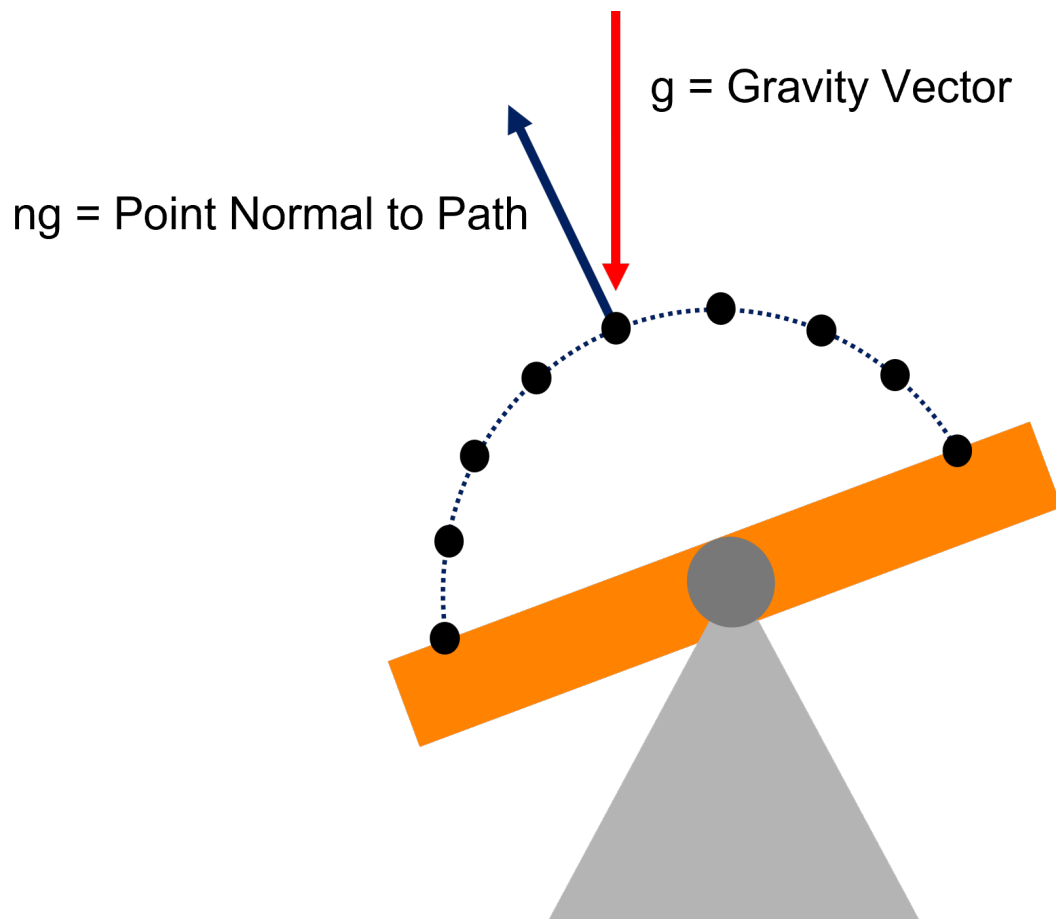


Figure 3.4: Gravity aligned constraint for the gravity vector and point normal vector

always be in the xy-plane. Therefore, the z-component of the cross product can be removed and only two rows are added to the forcing function vector. The 2×1 gravity aligned forcing function is calculated below. The R matrix is used to remove the z-component of the vector, and the path normal vector is transformed to the part frame. The new, augmented forcing function is given by (3.45).

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad (3.43)$$

$$f2_{g-err} = R([Z_P T_P G n_g(i)] \times g) \quad (3.44)$$

$$f8_{t-err} = \begin{bmatrix} f6_{p-err} \\ f2_{g-err} \end{bmatrix} \quad (3.45)$$

The Jacobian must also be augmented to include the new gravity aligned constraint. Since the alignment of the path normal vector with the gravity vector is only dependent on the part positioner joint angles, only the first two columns of the Jacobian have elements in the added two rows. The w and R in (3.47) and (3.48) are the same values as (3.27) and (3.43) respectively.

$$J = \begin{bmatrix} v_1 & v_2 & v_3 & \dots & v_8 \\ w_1 & w_2 & w_3 & \dots & w_8 \\ d_1 & d_2 & 0 & \dots & 0 \end{bmatrix} \quad (3.46)$$

$$d_1 = R Z_P A_{1P} w \quad (3.47)$$

$$d_2 = R Z_P w \quad (3.48)$$

3.5 Solving Arbitrary Torch Rotation with the Jacobian Null Space

Since for some applications the z-rotation of the welding torch is arbitrary, there is still one redundant DOF in the operational space of the system that can be used to enhance results. The path planning program assigns all points an A, B, C rotation that is either in a fixed direction or aligned with the path direction. In order to remove the assigned z-rotation and make it arbitrary, the sixth row of the Jacobian matrix is removed, which corresponds to the z Euler angle rotation.

With the z-rotation removed from the Jacobian, a cost function can be used to assign the torch rotation value. One method for adding additional constraints to the solved joint variables is the gradient projection method described in (10). This method adds a term to the psuedo-inverse Jacobian method solution that projects a constraint onto the null-space of the Jacobian. This maintains the joint and end-effector constraints but modifies the internal motion of the manipulator.

$$\dot{q} = J^\dagger(q)\dot{x} + [I - J^\dagger(q)J(q)]\dot{q}_0 \quad (3.49)$$

The $\dot{q}_0$ term is a joint velocity vector and can be found using (3.50) which maximizes a cost function, $w(q)$, where k_0 is a scaling factor.

$$\dot{q}_0 = k_0 \left(\frac{\partial w(q)}{\partial q} \right)^T \quad (3.50)$$

The complete iterative equation can then be written as (3.51).

$$\theta_{j1} = \theta_j - hJ^\dagger f_{8_{t-err}}(\theta_j) + [I - J^\dagger(q)J(q)]\dot{q}_0 \quad (3.51)$$

$$[f_{8_{t-err}}(\theta_j)]^T K [f_{8_{t-err}}(\theta_j)] < \epsilon \quad (3.52)$$

The chosen cost function depends on the desired application. For this thesis, two main cost functions were explored. First, the manipulability measure, developed by (15), can be

used to avoid singularities in the manipulator configuration. Since the Jacobian becomes rank deficient at a singularity, the value of the manipulability measure, (3.53), approaches zero close to a singularity. The cost function will therefore maximize the manipulability measure, effectively maximizing the manipulator "distance" from a singularity.

$$w(q) = \sqrt{\det(J(q)J^T(q))} \quad (3.53)$$

Since the Jacobian matrix has elements with both rotational units and translational units, the weights of the rotational and translational components are not equal when calculating the manipulability measure value. Therefore, the Jacobian matrix components were scaled and then set to a ratio value, as shown in (3.54) and (3.55), before the manipulability measure was calculated.

$$\frac{J_O}{1} = \frac{J_{O,S}}{10} \quad (3.54)$$

$$\frac{J_P}{1000} = \frac{J_{P,S}}{10} \quad (3.55)$$

The second cost function used maximizes the individual joint distances from joint limits, as shown in (3.56), where q_{iM} is the maximum joint value, q_{im} is the minimum joint value, and $\bar{q}_i$ is the middle value of the joint (7). This cost function is useful to add to the solver even for cases with a set weld torch rotation value, as it will prevent undesirable robot position solutions where the joint limits are exceeded.

$$w(q) = -\frac{1}{2n} \sum_{i=1}^n \left(\frac{q_i - \bar{q}_i}{q_{iM} - q_{im}} \right)^2 \quad (3.56)$$

Chapter 4

Implementation

4.1 Kinematic Solver MATLAB Program

4.1.1 Program Overview

The MATLAB kinematic solver program was written to be easily modifiable for different robot parameters, initial conditions, constraints, and cost functions. The block diagram in Figure 4.1 illustrates the iterative process for calculating the inverse kinematics positions for an entire part path.

The first step in the program is to initialize the kinematic properties and kinematic functions classes. The kinematic properties class consists of all the robot geometry values, including the DH parameters, the transformation distances between the manipulator and part positioner, and the weld torch dimension values. It also holds the initial joint values, the current joint values, the matrix of all solved joint values, the step size value, and the convergence criteria. Finally, the cost functions and constraint functions used in the solving process are determined in the kinematic properties class.

The kinematic functions class stores all of the kinematic transformation matrices for the manipulator, part positioner, torch translation, part translation, and base translations. Additionally, the Jacobian, error vector, and null-space cost function values are stored.

When the kinematic solver iterates through solutions, the joint angles are updated in the kinematic properties class. Then, a function is called which updates all of the kinematic

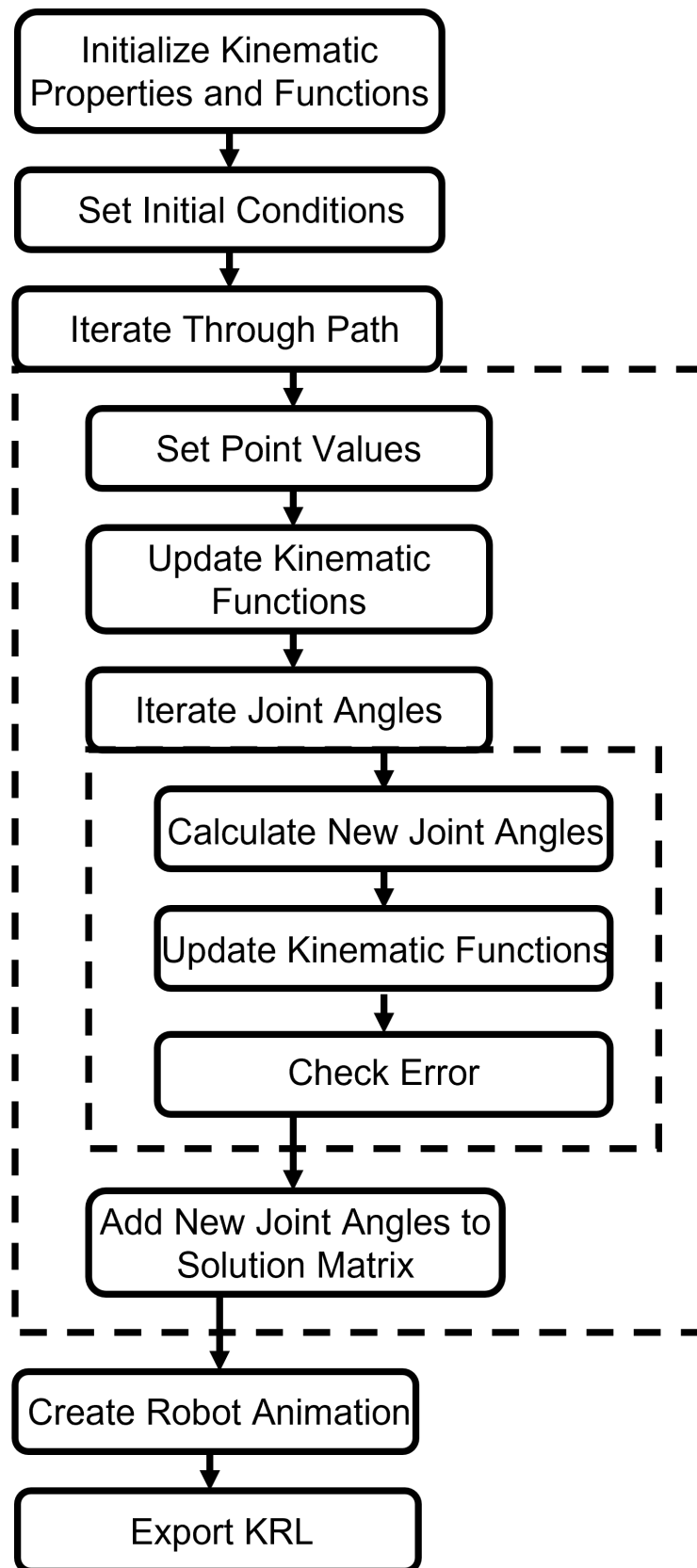


Figure 4.1: MATLAB inverse kinematic solver block diagram

functions based on the kinematic property values and new joint angles. Depending on the specified cost and constraint functions, the Jacobian, error vector, and null-space cost function value are modified.

After the kinematic properties and functions classes are initialized, the initial joint values are set to the current joint values. The initial joint values are only used for solving the first path point, and future points take the solved joint values from the previous point as the initial joint values for the next point.

Next, the program begins iterating through the path points using the part data output by the path planner program. In the path planner, the part is divided into segments, which consist of contours, which consist of moves, which consist of waypoints. The moves consist of the current waypoint and the next waypoint, so only the current waypoint is selected for solving. Therefore, the kinematic solver iterates through each segment, solving for the waypoints in each contour. The solved joint values are then stored in a matrix and organized by segment, contour, and point order.

Before a point is solved, the x, y, z data and the orientation rotation matrix is extracted from the part data and stored in the kinematic properties class. Additionally, the path normal vector used for the gravity aligned welding constraint is found from the third column of the orientation rotation matrix. The kinematic functions class is updated for the initial joint angles, and the iterative kinematics solving process is run.

The iterative kinematic solving process goes through three main steps. First, the new joint angles for the current iteration are solved for using the iterative joint function. The new joint angles are set in the kinematic properties class, and then the kinematic functions class is updated for the new joint values. Finally, the error value is compared to the convergence criteria using. The process is repeated until the error value is sufficiently small and the solution has converged for that path point. The solved joint angles are then stored in the solved joint angle matrix and the final error values for that point are displayed. This process is repeated until the joint values for each point in the part plan are solved. A plot is created to view an animation of the solved robot positions.

The KUKA robots use different coordinate frames than the solver does to represent joint rotations, so some joint angles need to be incremented or have their rotation signs

flipped. The solved joint angles are run through a function that converts them to the rotation representation used by KUKA using the equations below. The rotation angles are also converted from radians to degrees in this step. Finally, the matrix of joint solutions is ready to be exported and converted to KUKA Robot Language (KRL).

$$\theta_{1,KUKA} = \theta_{1,S} \quad (4.1)$$

$$\theta_{2,KUKA} = \theta_{2,S} \quad (4.2)$$

$$\theta_{3,KUKA} = -\theta_{3,S} \quad (4.3)$$

$$\theta_{4,KUKA} = -\theta_{4,S} \quad (4.4)$$

$$\theta_{5,KUKA} = \frac{\pi}{2} - \theta_{5,S} \quad (4.5)$$

$$\theta_{6,KUKA} = -\theta_{6,S} \quad (4.6)$$

$$\theta_{7,KUKA} = -\theta_{7,S} \quad (4.7)$$

$$\theta_{8,KUKA} = -\theta_{8,S} \quad (4.8)$$

4.1.2 KUKA Robot Language Exporter

Once the inverse kinematics are solved for each of the path points, the robot positions can be exported to KUKA Robot Language (KRL) to be run on the physical KUKA manipulator and part positioner. In order to generate the KRL, it is necessary to understand the different point and move types that the KUKA controller recognizes.

A KRL program can consist of linear moves and point-to-point moves. For a linear move, the robot tool center point (TCP) will follow a straight path between the two specified points. For a point-to-point move, the robot moves the TCP to the next point using the shortest path in the joint space. Therefore, the path between the points is not considered. In terms of velocity, a linear move takes an input linear velocity. A point-to-point move takes a percentage value of the maximum joint speed it will take to reach the next point. So if a point-to-point move is given a velocity of 100%, then the robot will move to the next point as fast as its joint velocities will allow it (the robot can move with a linear velocity of up to $2m/s$, so a 100% joint speed will be very fast). However, for a point-to-point move, it is impossible to predict the joint paths and relative joint velocities between two points, as this is information built into the KUKA controller kinematic position solver. This is an important consideration for a welding application where the velocity of the weld torch must be controlled.

A robot position in KRL is either defined by the point position, x, y, z, A, B, C or by the individual joint axis of the robots. In KRL, a point defined robot position is defined by an E6POS value type. This value can be used in either point-to-point or linear moves. A point using joint angles in KRL is defined by an E6AXIS value type, which can only be used for point-to-point moves. Since the WAAM application of the robots requires a constant linear velocity, the path points must be defined by their point position, or the E6POS KRL value type, which can be used with linear moves. This means that the KUKA controller will be solving the inverse kinematics of these points. In order to keep the robot positions solved for in the MATLAB inverse kinematics program, the first move of the path can be defined as a point-to-point move using the E6AXIS KRL position type, where the solved joint angles are sent directly to the robot. Then the robot will start the path in the solved position and will maintain the position throughout the path linear moves. In summary, the first point of the part path will use the solved joint angles, and then the rest of the points in the path will use the part x, y, z, A, B, C values with the inverse kinematics solved by the KUKA controller.

The KRL program itself is generated in MATLAB and consist of an SRC file and a DAT file. The DAT file defines all of the point types and values, and the SRC file runs through the path itself, commanding the robot to move to each position. For the E6AXIS joint defined

KRL position types, the solved joint angles were input directly. The E6POS point defined KRL position types take the x, y, z values defined for the part in the path planner. The A, B, C values are then calculated as xyz current (or intrinsic) frame Euler angles from the solved robot position rotation matrix (since the z -rotation of the torch is solved differently from the z -rotation defined by the path planner). The A, B, C rotations of the torch are calculated and then input into the defined point in the DAT file with the x, y, z values.

4.1.3 Kinematic Solver Graphical User Interface

A graphical user interface (GUI), shown in Figure 4.2, was created for the inverse kinematics solver so that it could be implemented with the path planner GUI. The part path generated in the path planner GUI is sent to the kinematics GUI, and then the position can be solved for each point in the path. Before running the solver, different settings can be applied for the kinematics solutions. First, the user specifies if the part positioner joint angles should be solved or whether the part positioner will remain stationary throughout the path. This allows simple gravity aligned parts that do not require the part positioner to be solved. If using the part positioner is selected, then the user must select whether the gravity aligned welding constraint will be used on the part positioner. If gravity aligned welding is not selected, the solution will only depend on the initial condition values and will not be constrained to a specific solution. If the gravity aligned welding constraint is selected, then the user can specify what cost function will be applied to the solution. The options are to use the singularity avoidance cost function, the joint limit avoidance cost function, the combined singularity and joint limit cost functions, or no cost functions. If none is selected, the position will be solved to the orientation given by the imported path from the path planner. If the path planner is used to orient the points to be normal to the path, then the orientation of the torch will be to face the direction of the path. Additionally, there is an option to keep the torch rotation fixed relative to the world base throughout the entire path, and the fixed rotation angle value can be set.

The kinematics solver button starts the iterative solver process. Once the path is solved, a specific point of the path can be selected and the solved joint angles, torch position, and error values are displayed for that point. The joint angles for the entire path are plotted, and

a robot model figure of the manipulator and part positioner are plotted at the solved joint angles. Finally, the robot path kinematics can be exported as KRL code to be uploaded to the physical robot system.

4.1.4 Inverse Kinematic Solver Effectiveness

For a solution point with a given x, y, z, A, B, C value, the complete inverse kinematics solver could generally converge on a solution within at most 2000 iterations, and the time to solve one contour consisting of 39 points took approximately 20 seconds. The part positioner orientation using the gravity aligned constraint was usually found very quickly in a few hundred iterations.

The convergence criteria used for the error function was $\epsilon = 0.001$. All position errors converged to a value with an error less than $0.05mm$ and all orientation errors to a value with an error less than 0.05° .

Using a cost function increases the convergence time to up to two minutes a contour, depending on how strict the weight of the cost function used are. If a point took more than 10,000 iterations to solve a point with a cost function, the point was solved to the orientation of the previous point but the translation values of the new point.

4.2 Comparison of Kinematic Solver Solutions

In order to test the kinematic solver, the points of the first contour of a propeller blade path were run through. The propeller blade is an ideal shape for testing, as each point on the path has a different normal vector and requires a new part positioner orientation to keep the gravity aligned constraint. The kinematic solver was tested with the singularity avoidance cost function, the joint limit cost function, and with a combination of both. Additionally, a fixed torch rotation method was developed and tested to minimize robot movement during the path.

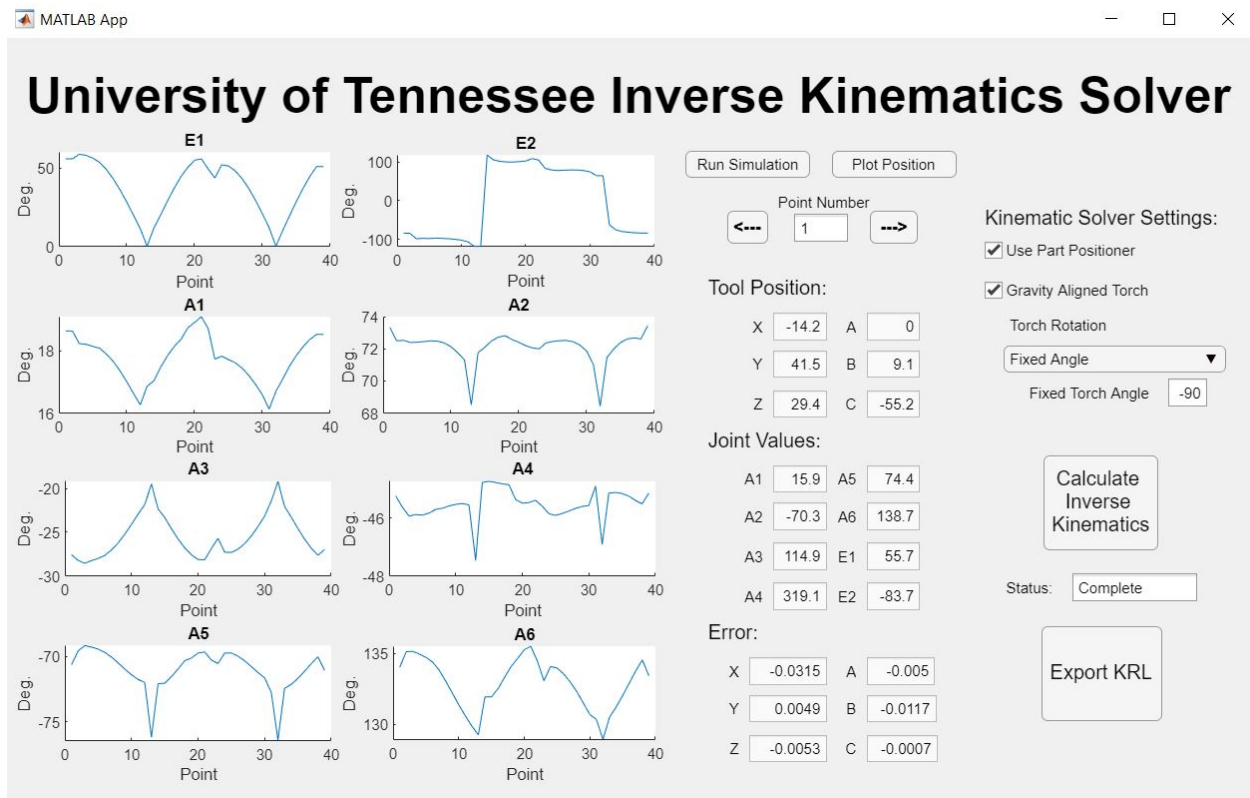


Figure 4.2: MATLAB inverse kinematics solver GUI

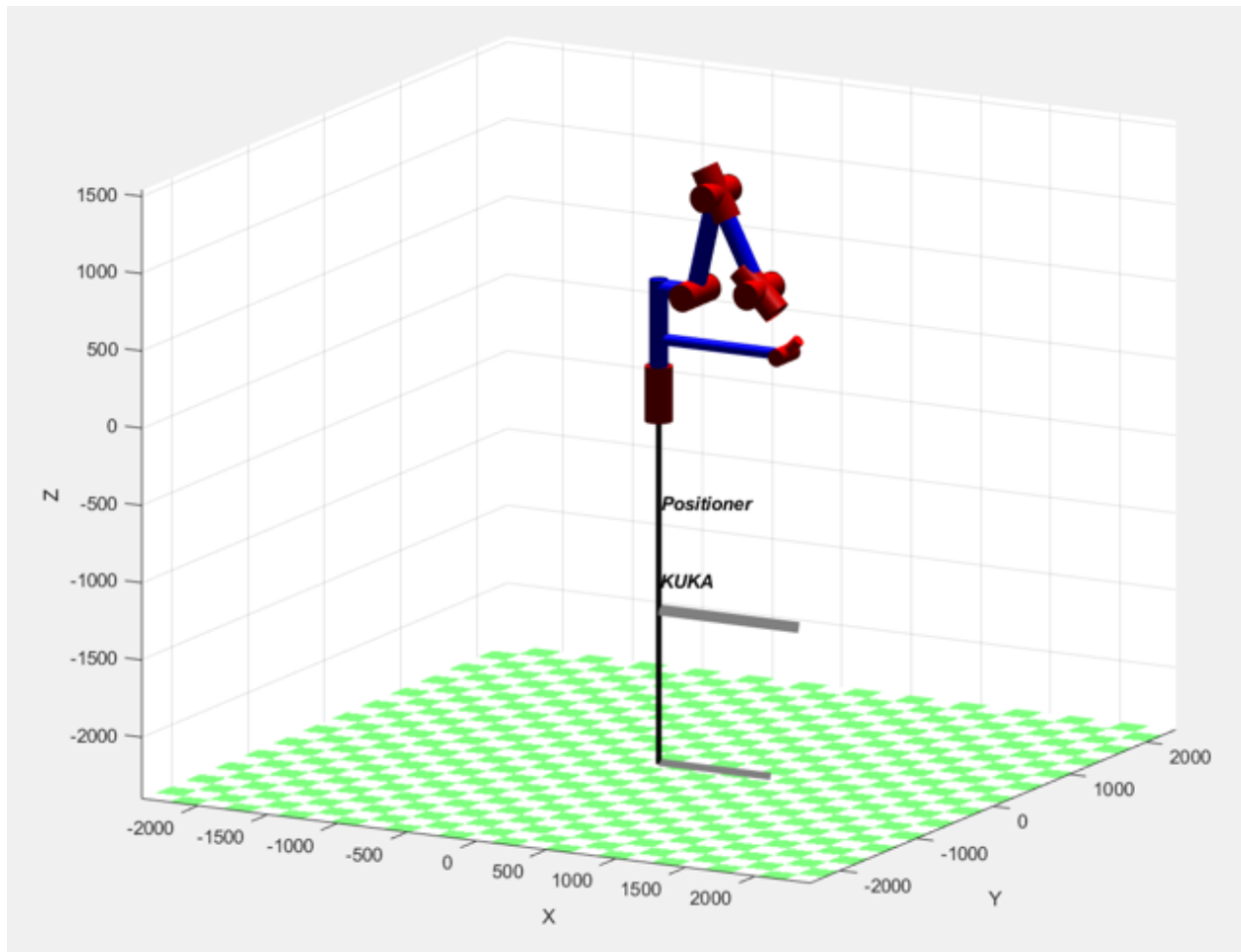


Figure 4.3: Manipulator and part positioner figures in the MATLAB inverse kinematics solver

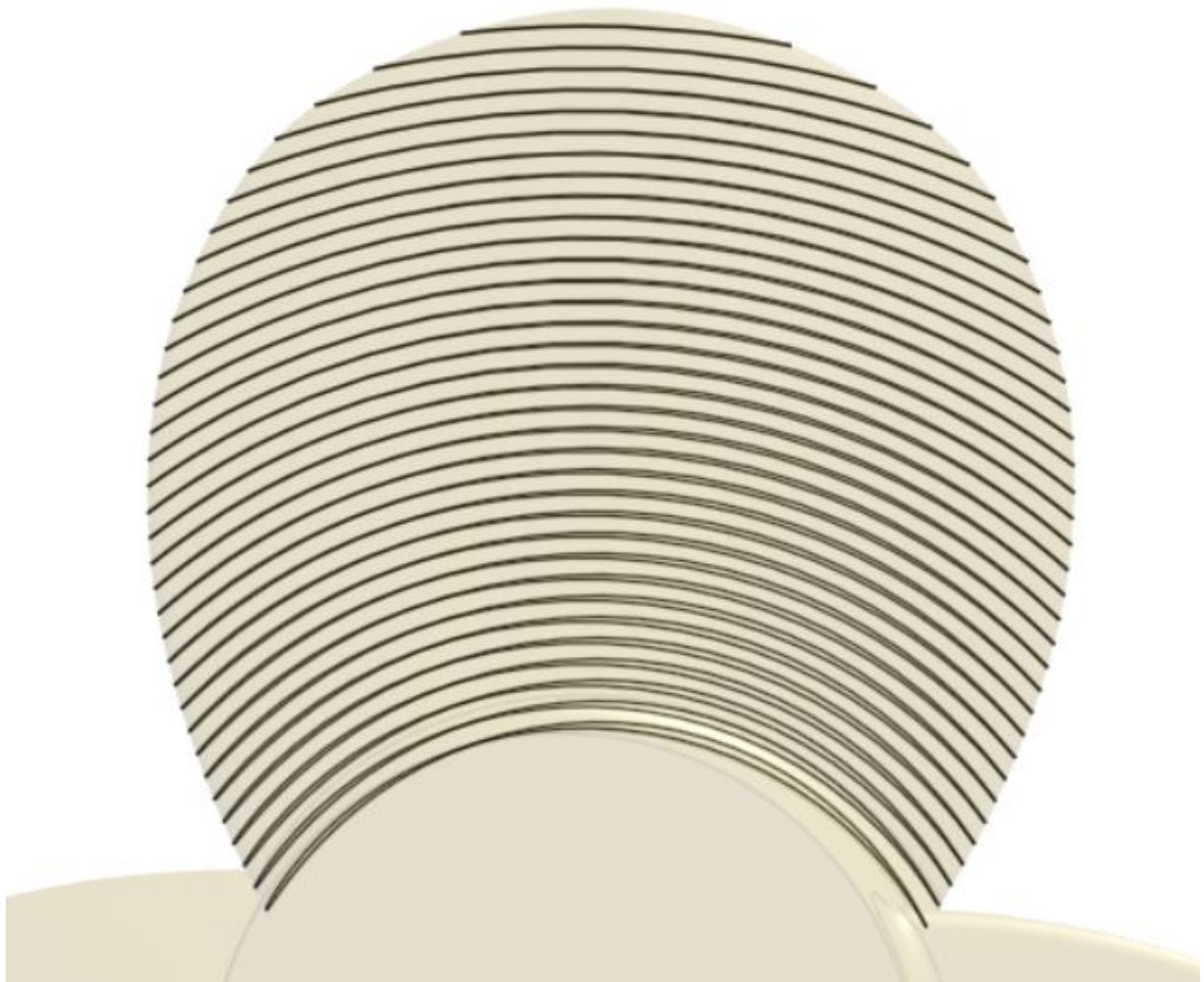


Figure 4.4: Propeller blade path contours

4.2.1 Singularity Avoidance

Singularity avoidance is important while planning a robot path, as a singularity can cause high joint velocities which may cause torque errors in the robot and prevent the program from running. Therefore, the singularity cost function was tested with the inverse kinematics solver program. However, running the program with the singularity avoidance cost function on the first contour of the propeller blade path gave several undesirable robot positions, such as with the robot elbow down (colliding with the floor) or with axis one flipped 180° backwards. From the solved joint angles shown in Figure 4.5, it is also seen that there are large jumps between different robot configurations. This implies that between points the robot is moving to the position with the largest manipulability measure value regardless of its initial conditions at the previous point. Additionally, in order to avoid the robot wrist singularity, the robot is forced into configurations where the wrist axis is bent as far from its zero position as possible, resulting in several undesirable robot positions. Therefore, for the singularity cost function to be usable, some additional constraints must be used to control the robot position.

4.2.2 Joint Limit Avoidance

The joint limit avoidance cost function insures that the solved robot positions are within the specified joint limits for each of the robot axis. In order to simplify the computation time, only four joint limits were specified. Axis one was constrained between positive and negative 90° in order to prevent the robot from becoming backwards oriented, axis two was constrained between 0° and 180° to prevent the robot from moving to the elbow down position, and axis five was constrained between 0° and -90° to prevent the wrist from flipping over during the path.

The second axis of the part positioner was also constrained between 0° and 180° in order to minimize the motion of the part while it is being welded. Without the joint constraint on the part positioner, the first axis of the part positioner will typically stay forward facing with a joint angle above 0° . This can cause a sudden change in orientation of the second part positioner axis in order to maintain the gravity aligned constraint. The joint limit constraint

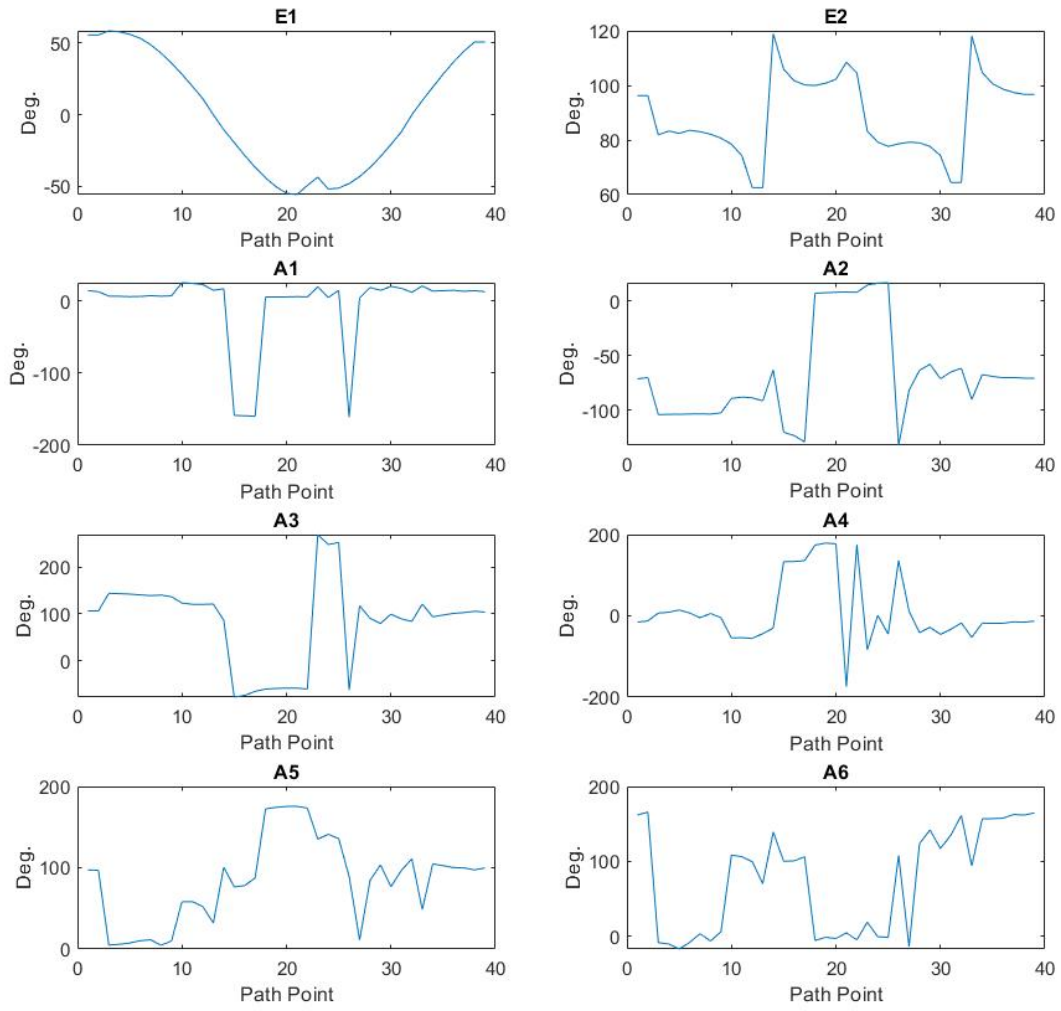


Figure 4.5: Robot joint solutions using singularity avoidance cost function

on the second axis forces the part positioner to be solved for both positive and negative first axis joint values while also minimizing the change in position of the second axis.

The solved joint angles for the first contour of the propeller blade with the joint limit cost function, shown in Figure 4.6, were much smoother overall and did not have any large changes in position. Additionally, the joint limit applied to the fifth axis prevented the fourth and sixth axis from becoming aligned and creating a singularity. While running through the path, the first axis of the robot remains positive and the torch is oriented to the right of the part. The change in rotation of the torch is also relatively small. Overall, the joint limit cost function gave desirable robot position solutions for the propeller blade part path.

4.2.3 Combined Singularity and Joint Limit Avoidance

A combination of the singularity avoidance cost function and the joint limit cost function was then tested. While the joint limit cost function gave desirable joint solutions for the propeller blade part, the addition of the singularity cost function would increase the flexibility of the solver, insuring desirable, singularity free solutions for a wider range of part geometries. However, upon testing the combined cost functions, the solver struggled to give a desirable solution. The solved joint angles shown in Figure 4.7 show that the joint limit constraints were not sufficiently followed, as the first axis flips 180° backwards and the robot moves to the elbow down position. The singularity cost function weight was then decreased until the joint limit cost function was sufficiently followed. However, the singularity cost function value was then too low and did not significantly affect the solution results.

The main issue with combining the singularity and joint limit avoidance cost functions is that they work against each other to a degree, making convergence to a solution difficult. The joint limits are defined to keep the robot positioned towards the center of the part while also keeping the torch positioned in front of the wrist. This centered position also corresponds to the position where a wrist singularity is most likely to occur. Therefore, the singularity cost function is trying to move the robot away from this position to a point where the wrist joint has a larger angle.

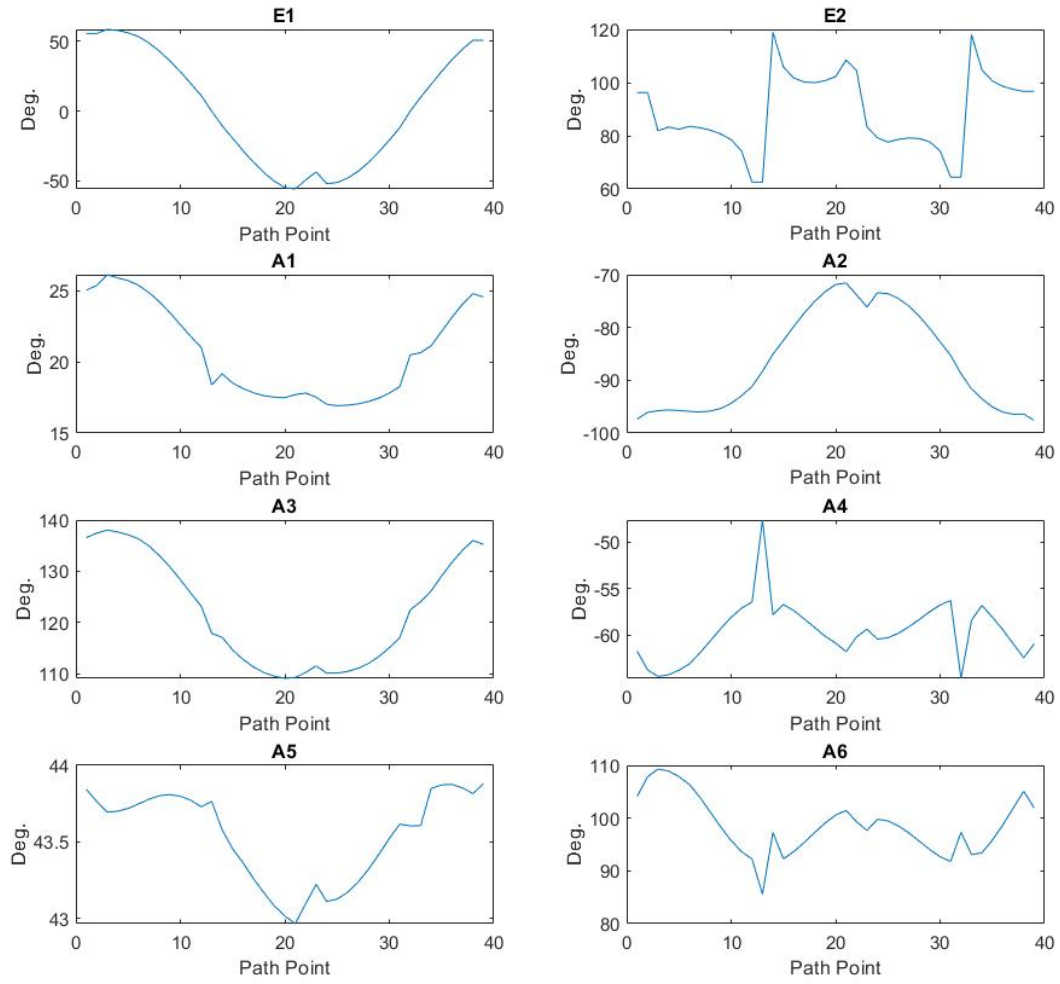


Figure 4.6: Robot joint solutions using joint limit avoidance cost function

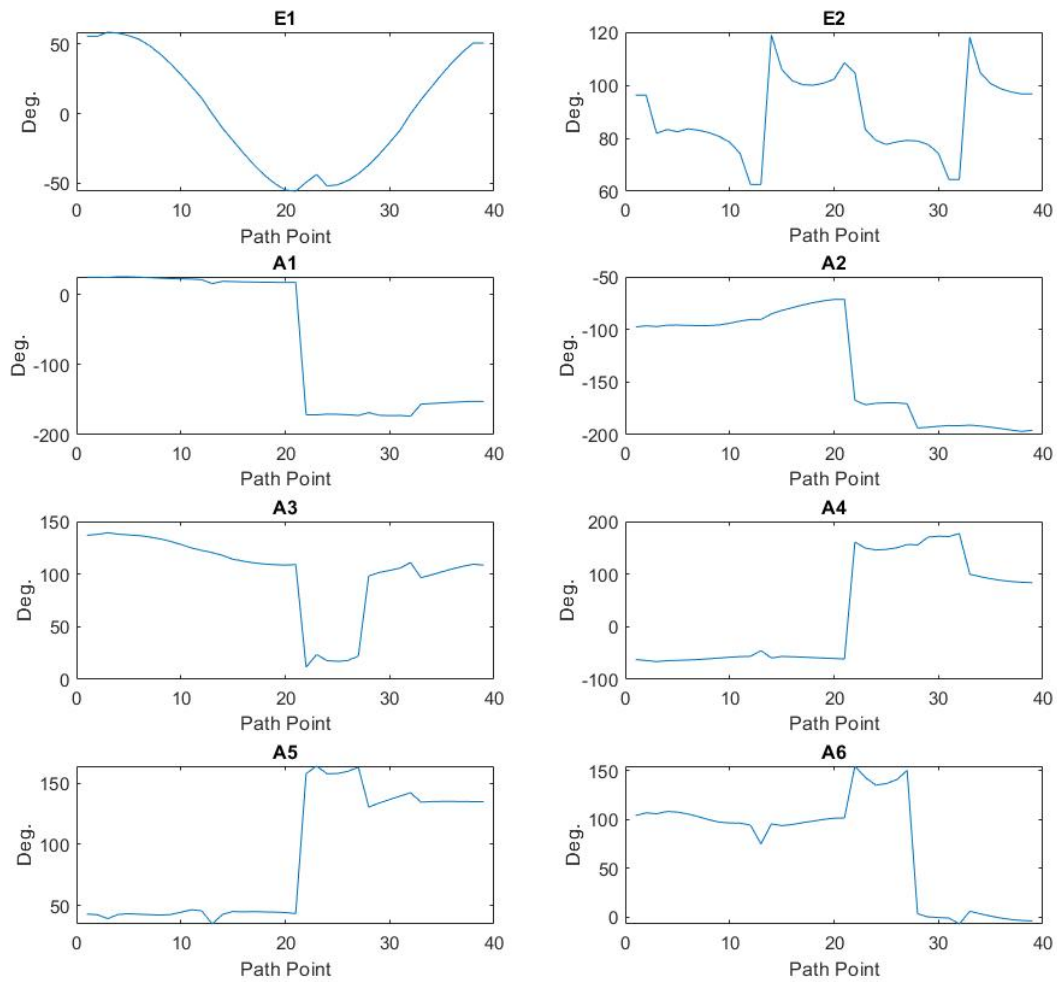


Figure 4.7: Robot joint solutions using combined singularity avoidance and joint limit avoidance

4.2.4 Fixed Torch Rotation

Using a fixed torch rotation for the duration of the part path is the most reliable solving method for a part using the gravity aligned constraint. The fixed torch rotation reduces overall movement of the manipulator and prevents unexpected, fast robot movement. The robot position for a fixed torch rotation is solved in two steps. First, the manipulator and part positioner kinematics are solved for a point with the z-rotation of the torch specified to be oriented along the direction of the path. Once the point is solved, the positioner is in the correct gravity aligned orientation and is kept fixed. Then the manipulator position is re-solved (with the positioner kept fixed) to the desired fixed z-rotation with respect to the world coordinate frame. The joint limit cost function was also used for both steps when solving for the fixed torch rotation. This removed undesirable robot positions and also prevented the wrist from flipping over.

The solved joint angles are shown in Figure 4.6. There are no large changes in axis positions for the manipulator, so the robot position stays smooth through the path. Additionally, since the torch rotation is fixed, the robot never approaches the wrist singularity.

4.3 Program Testing on KUKA Manipulator and Part Positioner

A KRL robot program was created for a single layer of the propeller blade part in both Octopuz and the MATLAB inverse kinematics solver so that the resulting joint values run on the actual KUKA manipulator and part positioner could be compared. While the programs ran, the joint values were recorded using Robot System Integration (RSI), which recorded the joint values for the six manipulator axes.

Both the robot programs used a fixed welding torch orientation of -90° . The KRL program created in Octopuz consists of all linear moves, meaning that only the x, y, z, A, B, C values of the path points with respect to the part positioner base are exported and that the

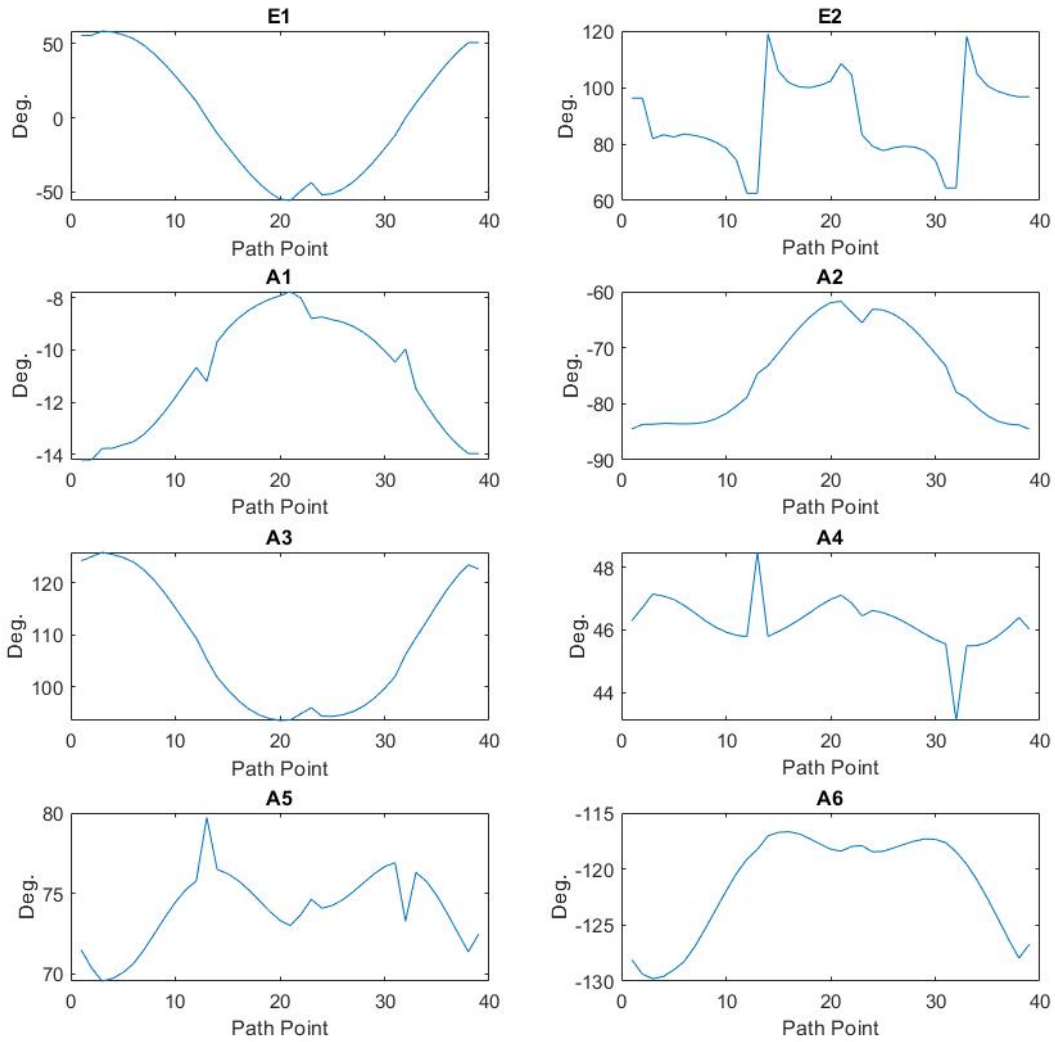


Figure 4.8: Robot joint solutions with a fixed torch rotation

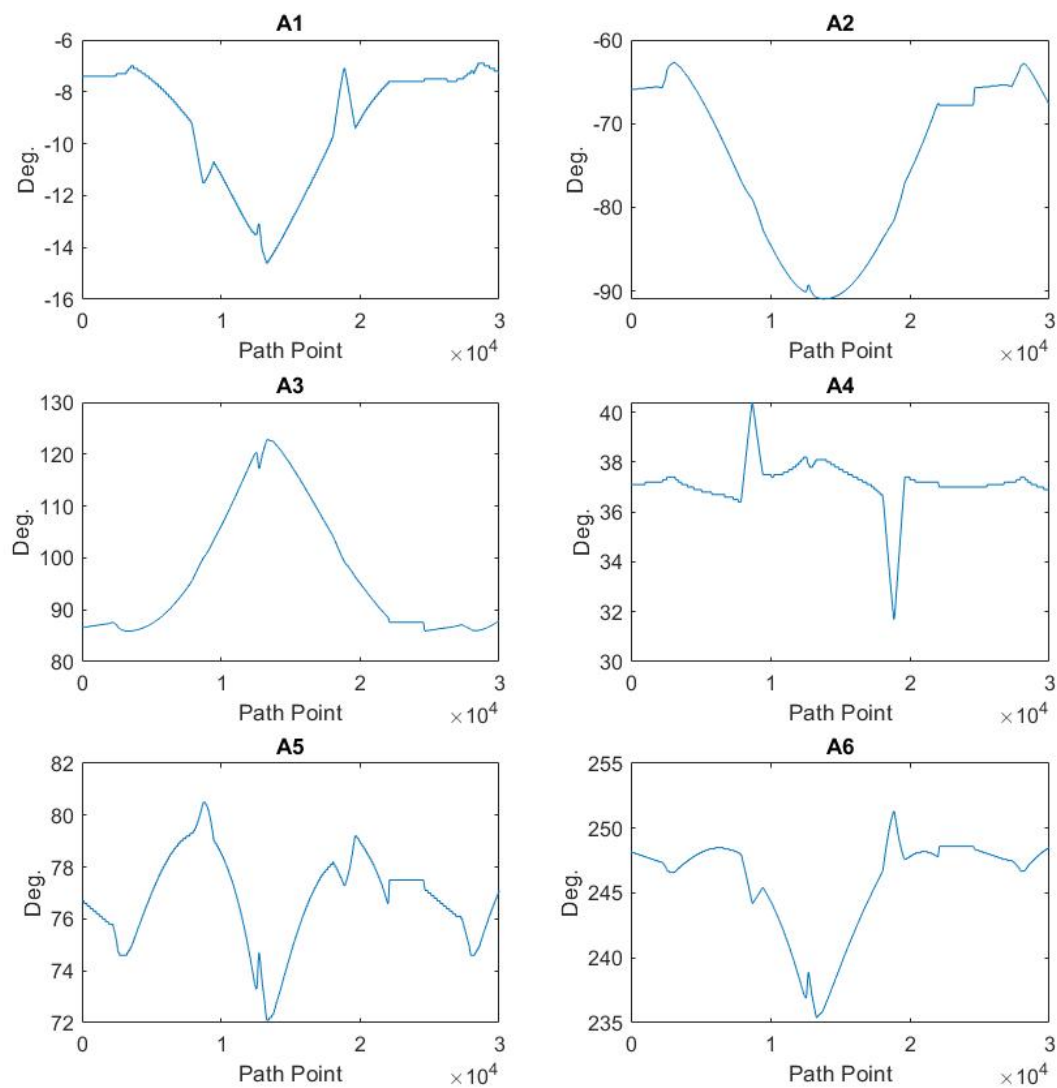


Figure 4.9: Octopuz robot program joint values recorded with RSI

KUKA controller solves the inverse kinematics for each point. The resulting robot motions were all smooth, as seen in Figure 4.9, and solved exactly like the simulation in Octopuz.

The next test was to use the inverse kinematic solver solutions to export and run KRL which consisted of an initial E6AXIS position type point-to-point move and then all linear moves for the rest of the contour. The E6AXIS type position move sends the solved joint values directly to the robot, while the linear moves are solved from the input x, y, z, A, B, C values. The solved joint angles used to create the program are the same as shown in Figure 4.8 and the recorded RSI joint values are shown in Figure 4.10.

As illustrated by the figures, the solved joint angles and the recorded joint angles line up fairly well. Most of the differences between the joint angles are due to the part positioner base position not being perfectly calibrated to a position that can be added to the inverse kinematics solver, so the two programs use slightly different base positions. In order to accurately calibrate the base position, a "touch off" program must be used on the robot, where the weld torch wire tip touches off on points of the metal base plate that will be welded on and uses an electric current to find the exact center of the weld plate. Since the robot cell is still under development, the touch off program has not been created yet, but a rough estimate of the base position gives close enough joint angle values so that they can be compared.

During the test run of the MATLAB inverse kinematics KRL program, the weld torch did not stay perfectly fixed at -90° while following the part path. As can be seen in Figure 4.10, there is a large spike in the sixth axis value. This point corresponds to where the second axis of the part positioner had a large joint value change of approximately 60° . While running the program, the robot is kinematically connected to the part positioner. So as the second axis value quickly changed, the KUKA controller maintained the orientation of the torch with respect to the base position. The torch orientation was then quickly adjusted to the -90° rotation value at the start of the next point of the path.

There are several solutions to the torch misalignment issue. First, the part path can be segmented so that the solution points are closer together. This would insure that the robot corrects its torch orientation more frequently. More strict constraints could also be used on the part positioner second axis to prevent large joint displacements. The OCTOPUZ test

joint solutions did not have a misalignment issue because the changes in part positioner joint angles were not as large.

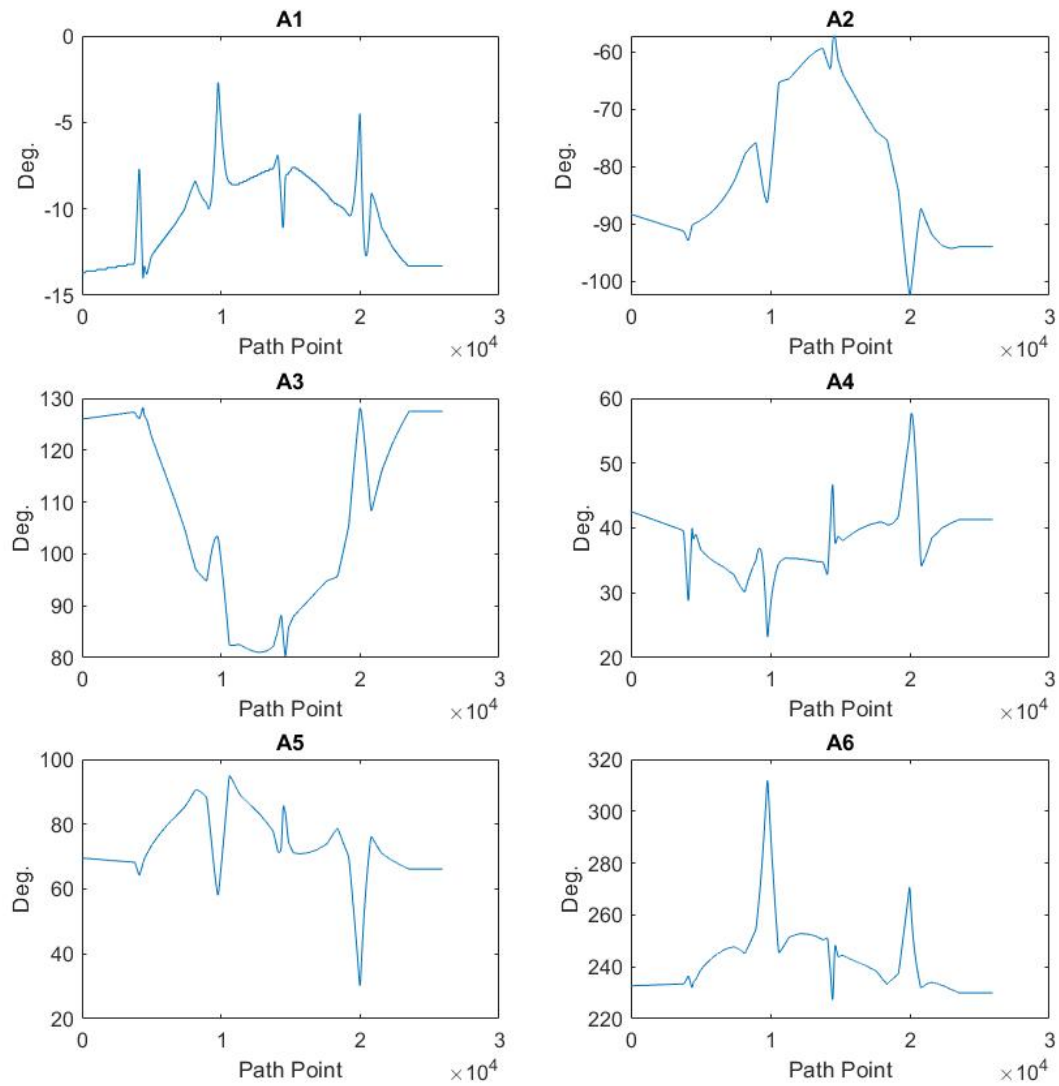


Figure 4.10: MATLAB inverse kinematics program joint values recorded with RSI

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The MATLAB inverse kinematics solver program has greatly simplified the robot programming process for fabricating a WAAM part. The part path plan can be directly sent to the kinematics solver from the path planning program, and then once the kinematics are solved for each point, the path can quickly be reviewed by checking the joint angle plots and running the robot figure animation. The solved joint angles can also be input directly to Octopuz where they can be viewed before the program is run on the actual robots.

Currently the most reliable method for solving a path to be run on the actual WAAM robots is to keep the torch rotation at a fixed angle to minimize robot motion while also using the gravity aligned part positioner constraint. Additionally, since the part positioner can be kept stationary, simple gravity aligned part paths can also be reliably solved. The singularity avoidance cost function, the joint limit cost function, and the combination of the two have not been reliable as a generalized solver, but they could prove to be useful given a specific part geometry that requires more complex robot motion.

5.2 Future Work

Several aspects of the inverse kinematics solver program could be improved in the future. First, one of the main problems with the joint solutions found using the singularity avoidance

cost function and the joint limit avoidance cost function was large jumps in joint angles which would result in high joint velocities when run on the actual robots. The kinematic solver solves each point of the path separately and does not take into account the robot position for the last point, besides the initial joint positions. This can result in the solver jumping the robot to an undesirable location if it cannot immediately find a solution close to its initial conditions. The large position change of the robot results in high joint accelerations and therefore high torque values. Several methods exist for minimizing joint velocity, acceleration, and torque, as discussed by (6), (5), and (16).

The constraint equation used to solve the part positioner orientation could also be improved. Currently, the part positioner is solved so that every point on the part path deposits the weld bead in a gravity aligned orientation. For larger parts with points far away from the part positioner, a small change in the path normal vector can result in a large change in the joint values, resulting in a high torque. A better part positioner constraint would have the welding torch remain gravity aligned when possible but have a limit of how far off vertical the torch can move when a torque limit could occur.

While some of the solver settings require further work, this thesis provides the foundation for further developments in controlling the robot position in a redundant system for specific applications in the future. The solver currently provides a more efficient method for creating the robot program for a part path, and with some improvements to the cost functions used, it would be a great tool for solving a wide variety of parts with different requirements.

Bibliography

- [1] Boz, Y. and Lazoglu, I. (2013). A postprocessor for table-tilting type five-axis machine tool based on generalized kinematics with variable feedrate implementation. *The International Journal of Advanced Manufacturing Technology*, 66. 2, 16
- [2] Chang, P. (1987). A closed-form solution for inverse kinematics of robot manipulators with redundancy. *IEEE Journal on Robotics and Automation*, 3(5):393–403. 2, 3
- [Cook et al.] Cook, G., Andersen, K., Zein-Sabattou, S., and Fernandez, K. Multiple-robot programming for coordinated motion, end-effector calibration, and part localization. IEEE. 2, 3, 17
- [4] Ghosh, P. K., Gupta, S. R., and Randhawa, H. S. (2000). Characteristics of a pulsed-current, vertical-up gas metal arc weld in steel. *Metallurgical and Materials Transactions A*, 31:2247–2259. 29
- [5] Guo, D. and Zhang, Y. (2014). Simulation and experimental verification of weighted velocity and acceleration minimization for robotic redundancy resolution. *IEEE Transactions on Automation Science and Engineering*, 11:1203–1217. 56
- [6] Hollerbach, J. and Suh, K. (1987). Redundancy resolution of manipulators through torque optimization. *IEEE Journal on Robotics and Automation*, 3:308–316. 56
- [7] Liegeois, A. (1977). Automatic supervisory control of the configuration and behavior of multibody mechanisms. *IEEE Transactions on Systems, Man, and Cybernetics*, 7:868–871. 3, 33
- [8] Mayorga, R. and Wong, A. (1988). A singularities avoidance approach for the optimal local path generation of redundant manipulators. pages 49–54. IEEE Comput. Soc. Press. 3
- [9] Octopuz (2021). <https://octopuz.com/>. [Online; accessed 23-November-2021]. 9
- [10] Siciliano, B. (1990). Kinematic control of redundant robot manipulators: A tutorial. *Journal of Intelligent and Robotic Systems*, 3. 2, 3, 17, 32

- [11] Siciliano, B., Sciavicco, L., Villani, L., and Oriolo, G. (2009). *Robotics*. Springer London. 5, 18
- [12] Whitney, D. (1969). Resolved motion rate control of manipulators and human prostheses. *IEEE Transactions on Man Machine Systems*, 10:47–53. 17
- [13] Williams, S. W., Martina, F., Addison, A. C., Ding, J., Pardal, G., and Colegrove, P. (2016). Wire + arc additive manufacturing. *Materials Science and Technology*, 32:641–647. 1
- [14] Xu, P., Yao, X., Chen, L., Liu, K., and Bi, G. (2020). Heuristic kinematics of a redundant robot-positioner system for additive manufacturing. IEEE. 2, 16
- [15] Yoshikawa, T. (1985). Manipulability of robotic mechanisms. *The International Journal of Robotics Research*, 4. 3, 32
- [16] Zhang, Y., Guo, D., and Ma, S. (2013). Different-level simultaneous minimization of joint-velocity and joint-torque for redundant robot manipulators. *Journal of Intelligent Robotic Systems*, 72:301–323. 56

Appendices

A Kinematic Solver MATLAB Code

Listing 1: Below is the MATLAB program used to solve the inverse kinematic robot positions for a part path. The functions iterate through each point in the path file and then iterate the robot joint values until the solution is found. Additionally, the functions for converting the solved joint angles to values that can be run by the KUKA robots are included.

```
1  classdef InverseKinematicsSolverFunctions
2      methods(Static)
3          function [f,k] = SolveInverseKinematicsForPart(active_part)
4              tic
5              k = KinematicProperties;
6              f = KinematicFunctions;
7
8              k.q = k.q0;
9              f.z_rot_old = 0;
10             InverseKinematicsSolverFunctions.SolvePartPoints(active_part,k,f);
11             InverseKinematicsSolverFunctions.SetJointsWithinJointLimits(k,f);
12             InverseKinematicsSolverFunctions.DisplaceJointAnglesForKuka(k,f);
13             toc
14
15             KinematicsPlot.PlotKinematics(k,f);
16         end%SolveInverseKinematicsForPart
17
18         function SolvePartPoints(active_part,KinematicProperties,KinematicFunctions)
19             k = KinematicProperties;
20             f = KinematicFunctions;
21             k.q_vals{1} = k.q0;
22             segment_length = length(active_part.segments);
23             contour_num=1;
24             for segment=1:segment_length
25                 contour_length = length(active_part.segments{segment}.contours);
26                 for contour=1:contour_length
27                     move_length = length(active_part.segments{segment}.contours{contour}.
28                         moves);
29                     move_num = 1;
30                     for move=1:move_length
31                         if move==1 %first move
32                             InverseKinematicsSolverFunctions.SetRetractedPointValues(f,
33                                 active_part,segment,contour,move);
34                             CalculateKinematicFunctions.UpdateKinematicFunctions(k,f);
35
36                             f = InverseKinematicsSolverFunctions.IterateJointAngles(k,f);
```

```

35
36     if k.torch_rotation(1) == 'f'
37         CalculateKinematicFunctions.
38             UpdateKinematicFunctionsForFixedTorchRotation(k,f);
39         f = InverseKinematicsSolverFunctions.IterateJointAngles(k,f)
40         ;
41     end
42
43     k.q_vals{contour_num}(:,move_num) = k.q;
44
45     move_num
46     Err_calc = inv(f.E)*inv(f.TR)*f.ZP*f.TP*f.G*f.P
47
48     move_num = move_num+1;
49     InverseKinematicsSolverFunctions.SetPointValues(f,active_part ,
50         segment , contour , move);
51
52     CalculateKinematicFunctions.UpdateKinematicFunctions(k,f);
53     f = InverseKinematicsSolverFunctions.IterateJointAngles(k,f);
54
55     if k.torch_rotation(1) == 'f'
56         CalculateKinematicFunctions.
57             UpdateKinematicFunctionsForFixedTorchRotation(k,f);
58         f = InverseKinematicsSolverFunctions.IterateJointAngles(k,f)
59         ;
60     end
61
62     k.q_vals{contour_num}(:,move_num) = k.q;
63
64     move_num
65     Err_calc = inv(f.E)*inv(f.TR)*f.ZP*f.TP*f.G*f.P
66     move_num=move_num+1;
67 elseif move==move_length %last move
68     InverseKinematicsSolverFunctions.SetPointValues(f,active_part ,
69         segment , contour , move);
70
71     CalculateKinematicFunctions.UpdateKinematicFunctions(k,f);
72     f = InverseKinematicsSolverFunctions.IterateJointAngles(k,f);
73
74     if k.torch_rotation(1) == 'f'
75         CalculateKinematicFunctions.
76             UpdateKinematicFunctionsForFixedTorchRotation(k,f);
77         f = InverseKinematicsSolverFunctions.IterateJointAngles(k,f)
78         ;
79     end
80

```

```

72
73     k.q_vals{contour_num}(:,move_num) = k.q;
74
75     move_num
76     Err_calc = inv(f.E)*inv(f.TR)*f.ZP*f.TP*f.G*f.P
77     move_num=move_num+1;
78
79     InverseKinematicsSolverFunctions.SetRetractedPointValues(f,
80         active_part,segment,contour,move);
81
82     CalculateKinematicFunctions.UpdateKinematicFunctions(k,f);
83     f = InverseKinematicsSolverFunctions.IterateJointAngles(k,f);
84
85     if k.torch_rotation(1) == 'f'
86         CalculateKinematicFunctions.
87             UpdateKinematicFunctionsForFixedTorchRotation(k,f);
88         f = InverseKinematicsSolverFunctions.IterateJointAngles(k,f)
89         ;
90     end
91
92     k.q_vals{contour_num}(:,move_num) = k.q;
93
94     move_num
95     Err_calc = inv(f.E)*inv(f.TR)*f.ZP*f.TP*f.G*f.P
96
97     move_num = move_num+1;
98
99     else
100     InverseKinematicsSolverFunctions.SetPointValues(f,active_part,
101         segment,contour,move);
102     CalculateKinematicFunctions.UpdateKinematicFunctions(k,f);
103     f = InverseKinematicsSolverFunctions.IterateJointAngles(k,f);
104
105     if k.torch_rotation(1) == 'f'
106         CalculateKinematicFunctions.
107             UpdateKinematicFunctionsForFixedTorchRotation(k,f);
108         f = InverseKinematicsSolverFunctions.IterateJointAngles(k,f)
109         ;
110     end
111
112     k.q_vals{contour_num}(:,move_num) = k.q;
113
114     move
115     Err_calc = inv(f.E)*inv(f.TR)*f.ZP*f.TP*f.G*f.P
116     move_num = move_num+1;
117
118 end

```

```

111         end
112         contour_num=contour_num+1;
113     end
114 end
115 end%SolvePartPoints
116
117 function SetPointValues(kinematic_functions, active_part, segment, contour, move)
118     f = kinematic_functions; % shorthand
119
120     x = active_part.segments{segment}.contours{contour}.moves{move}.point1.x;
121     y = active_part.segments{segment}.contours{contour}.moves{move}.point1.y;
122     z = active_part.segments{segment}.contours{contour}.moves{move}.point1.z;
123     f.P = active_part.segments{segment}.contours{contour}.moves{move}.point1.R;
124     f.P(4,:) = [0 0 0];
125     f.P(:,4) = [x; y; z; 1];
126     f.ng = f.P(1:3,3);
127 end%func SetPointValues
128
129 function SetRetractedPointValues(kinematic_functions, active_part, segment, contour,
130     move)
131     f = kinematic_functions; % shorthand
132
133     retracted_point = PathFileWriter.GenerateRetractedWaypoint(active_part.segments{
134         segment}.contours{contour}.moves{move}.point1);
135     x = retracted_point.x;
136     y = retracted_point.y;
137     z = retracted_point.z;
138     f.P = retracted_point.R;
139     f.P(4,:) = [0 0 0];
140     f.P(:,4) = [x; y; z; 1];
141     f.ng = f.P(1:3,3);
142 end%func SetRetractedPointValues
143
144 function f = IterateJointAngles(kinematic_properties, kinematic_functions)
145     k = kinematic_properties; % shorthand
146     f = kinematic_functions;
147
148     k.h = 0.1;
149     i=0;
150     part_frame = f.ZP*f.TP*f.G*f.P;
151     f.z_rot_old = atan2(part_frame(2,1), part_frame(1,1));
152     while (f.err_matrix.')*(f.err_matrix) >= k.convergence_criteria
153         dq = k.h*pinv(f.J)*f.err_matrix - f.null_val;

```

```

154         if dq(1)>0.001
155             dq(1)=0.001;
156         end
157
158         q_new = k.q - dq;
159
160         while q_new(2)<2*pi
161             q_new(2) = q_new(2) + 2*pi;
162         end
163         while q_new(2)>0
164             q_new(2) = q_new(2) - 2*pi;
165         end
166
167         f.J_old = f.J;
168         k.q_old = k.q;
169
170         k.q = q_new;
171
172         %Limit Part Positioner Movement
173         if k.q(1) > pi/2
174             k.q(1) = -k.q(1)+0.1;
175         end
176         if k.q(1) < -pi/2
177             k.q(1) = -k.q(1)-0.1;
178         end
179         if k.q(1) > pi/2
180             k.q(1) = 0;
181         end
182         if k.q(1) < -pi/2
183             k.q(1) = 0;
184         end
185
186         if k.q(2) > 0
187             k.q(2) = -pi/2;
188         end
189         if k.q(2) < -pi
190             k.q(2) = -pi/2;
191         end
192
193         if f.ng(3)>0.99
194             k.q(1)=0;
195             k.q(2)=k.q_vals{end}(2,end);
196         end
197
198         CalculateKinematicFunctions.UpdateKinematicFunctions(k,f);

```

```

199
200         if i > 5000
201             fprintf('i = %f      err = %f\n',i,(f.err_matrix.')(f.err_matrix))
202         end
203
204         %Edit step size
205         if i == 2500
206             k.h = 0.01;
207         end
208         %Set torch rotation to last torch rotation value
209         if i == 5000
210             k.h = 0.1;
211             if k.torch_rotation(1) ~= 'f'
212                 if size(k.q_vals)~=0
213                     k.q = k.q_vals{end}(:,end);
214                 else
215                     k.q = k.q0;
216                 end
217                 k.torch_fixed_rotation_value = -pi+2*f.z_rot_old;
218                 CalculateKinematicFunctions.
219                 UpdateKinematicFunctionsForFixedTorchRotation(k,f);
220             end
221             %Edit step size
222             if i == 6000
223                 k.h = 0.01;
224             end
225             %Reset initial conditions
226             if i == 10000
227                 k.q = k.q0;
228                 CalculateKinematicFunctions.
229                 UpdateKinematicFunctionsForFixedTorchRotation(k,f);
230             end
231             i=i+1;
232         end%while
233     end%func IterateJointAngles
234
235     function SetJointsWithinJointLimits(kinematic_properties,kinematic_functions)
236         k = kinematic_properties; % shorthand
237         f = kinematic_functions;
238
239         for i = 1:length(k.q_vals)
240             for j = 1:size(k.q_vals{i},2)
241                 k.q_vals{i}(1,j) = rem(k.q_vals{i}(1,j),2*pi);

```



```

242     if k.q_vals{i}(1,j) > pi
243         k.q_vals{i}(1,j) = k.q_vals{i}(1,j) - 2*pi;
244     end
245     if k.q_vals{i}(1,j) < -pi
246         k.q_vals{i}(1,j) = 2*pi + k.q_vals{i}(1,j);
247     end
248     k.q_vals{i}(2,j) = rem(k.q_vals{i}(2,j),2*pi);
249     if k.q_vals{i}(2,j) > pi
250         k.q_vals{i}(2,j) = k.q_vals{i}(2,j) - 2*pi;
251     end
252     if k.q_vals{i}(2,j) < -pi
253         k.q_vals{i}(2,j) = 2*pi + k.q_vals{i}(2,j);
254     end
255     k.q_vals{i}(3,j) = rem(k.q_vals{i}(3,j),2*pi);
256     if k.q_vals{i}(3,j) > pi
257         k.q_vals{i}(3,j) = k.q_vals{i}(3,j) - 2*pi;
258     end
259     if k.q_vals{i}(3,j) < -pi
260         k.q_vals{i}(3,j) = 2*pi + k.q_vals{i}(3,j);
261     end
262     k.q_vals{i}(4,j) = rem(k.q_vals{i}(4,j),2*pi);
263     if k.q_vals{i}(4,j) > pi
264         k.q_vals{i}(4,j) = k.q_vals{i}(4,j) - 2*pi;
265     end
266     if k.q_vals{i}(4,j) < -pi
267         k.q_vals{i}(4,j) = 2*pi + k.q_vals{i}(4,j);
268     end
269     k.q_vals{i}(5,j) = rem(k.q_vals{i}(5,j),2*pi);
270     if k.q_vals{i}(5,j) > pi
271         k.q_vals{i}(5,j) = k.q_vals{i}(5,j) - 2*pi;
272     end
273     if k.q_vals{i}(5,j) < -pi
274         k.q_vals{i}(5,j) = 2*pi + k.q_vals{i}(5,j);
275     end
276     k.q_vals{i}(6,j) = rem(k.q_vals{i}(6,j),2*pi);
277     if k.q_vals{i}(6,j) > pi
278         k.q_vals{i}(6,j) = k.q_vals{i}(6,j) - 2*pi;
279     end
280     if k.q_vals{i}(6,j) < -pi
281         k.q_vals{i}(6,j) = 2*pi + k.q_vals{i}(6,j);
282     end
283     k.q_vals{i}(7,j) = rem(k.q_vals{i}(7,j),2*pi);
284     if k.q_vals{i}(7,j) > pi
285         k.q_vals{i}(7,j) = k.q_vals{i}(7,j) - 2*pi;
286     end

```

```

287         if k.q_vals{i}(7,j) < -pi
288             k.q_vals{i}(7,j) = 2*pi + k.q_vals{i}(7,j);
289         end
290         k.q_vals{i}(8,j) = rem(k.q_vals{i}(8,j),2*pi);
291         if k.q_vals{i}(8,j) > pi
292             k.q_vals{i}(8,j) = k.q_vals{i}(8,j) - 2*pi;
293         end
294         if k.q_vals{i}(8,j) < -pi
295             k.q_vals{i}(8,j) = 2*pi + k.q_vals{i}(8,j);
296         end
297         if k.q_vals{i}(7,j) > 0
298             k.q_vals{i}(7,j) = -k.q_vals{i}(7,j);
299             k.q_vals{i}(6,j) = k.q_vals{i}(6,j)-pi;
300             k.q_vals{i}(8,j) = k.q_vals{i}(8,j)+pi;
301         end
302     end%for
303 end%for
304 end%func SetJointsWithinJointLimits
305
306 function DisplaceJointAnglesForKuka(kinematic_properties , kinematic_functions)
307     k = kinematic_properties; % shorthand
308     f = kinematic_functions;
309
310     k.q_vals_KUKA = k.q_vals;
311
312     for i = 1:length(k.q_vals_KUKA)
313         for j = 1:size(k.q_vals_KUKA{i},2)
314             k.q_vals_KUKA{i}(1,j) = k.q_vals_KUKA{i}(1,j);
315             k.q_vals_KUKA{i}(2,j) = k.q_vals_KUKA{i}(2,j)+pi;
316             if k.q_vals_KUKA{i}(2,j)>pi
317                 k.q_vals_KUKA{i}(2,j)=-(2*pi-k.q_vals_KUKA{i}(2,j))
318             end
319             k.q_vals_KUKA{i}(3,j) = -k.q_vals_KUKA{i}(3,j);
320             k.q_vals_KUKA{i}(4,j) = -k.q_vals_KUKA{i}(4,j);
321             k.q_vals_KUKA{i}(5,j) = pi/2 - k.q_vals_KUKA{i}(5,j);
322
323             k.q_vals_KUKA{i}(6,j) = -k.q_vals_KUKA{i}(6,j);
324             if k.q_vals_KUKA{i}(6,j) > pi
325                 k.q_vals_KUKA{i}(6,j) = k.q_vals_KUKA{i}(6,j) - 2*pi;
326             end%if
327             if k.q_vals_KUKA{i}(6,j) < -pi
328                 k.q_vals_KUKA{i}(6,j) = k.q_vals_KUKA{i}(6,j) + 2*pi;
329             end%if
330
331             k.q_vals_KUKA{i}(7,j) = -k.q_vals_KUKA{i}(7,j);

```

```

332
333         k.q_vals_KUKA{i}(8,j) = -k.q_vals_KUKA{i}(8,j);
334         if k.q_vals_KUKA{1}(8,j) > pi
335             k.q_vals_KUKA{1}(8,j) = k.q_vals_KUKA{1}(8,j) - 2*pi;
336         end%if
337         if k.q_vals_KUKA{1}(8,j) < -pi
338             k.q_vals_KUKA{1}(8,j) = k.q_vals_KUKA{1}(8,j) + 2*pi;
339         end%if
340     end%for
341 end%for
342 end%func DisplaceJointAnglesForKuka
343 end%Static Methods
344 end%class InverseKinematicsSolverFunctions

```

B Kinematic Functions MATLAB Code

Listing 2: Below is the MATLAB program used to solve the kinematic functions used in the iterative solver. The transformation matrices for the manipulator, part positioner, and the transformations between them are calculated, as well as the Jacobian and the error vector.

```
1 classdef CalculateKinematicFunctions
2     properties
3         Property1
4     end
5
6     methods(Static)
7         function UpdateKinematicFunctions(kinematic_properties, kinematic_functions)
8             CalculateKinematicFunctions.CalculateTransformationMatrices(kinematic_properties
9                                     , kinematic_functions);
10            CalculateKinematicFunctions.CalculateJacobian(kinematic_properties,
11                                                           kinematic_functions);
12            CalculateKinematicFunctions.CalculateErrorVector(kinematic_properties,
13                                                             kinematic_functions);
14            CalculateKinematicFunctions.CalculateNullSpaceValue(kinematic_properties,
15                                                                kinematic_functions);
16        end%func UpdateKinematicFunctions
17
18        function CalculateTransformationMatrices(kinematic_properties, kinematic_functions)
19            CalculateKinematicFunctions.CalculateManipulatorKinematics(kinematic_properties,
20                                                                           kinematic_functions);
21            CalculateKinematicFunctions.CalculatePositionerKinematics(kinematic_properties,
22                                                                           kinematic_functions);
23            CalculateKinematicFunctions.CalculatePositionerBaseTranslation(
24                kinematic_properties, kinematic_functions);
25            CalculateKinematicFunctions.CalculateTorchTranslation(kinematic_properties,
26                                                                     kinematic_functions);
27            CalculateKinematicFunctions.CalculateWorkpieceBaseTranslation(
28                kinematic_properties, kinematic_functions);
29        end%func CalculateTransformationMatrices
30
31        function UpdateKinematicFunctionsForFixedTorchRotation(kinematic_properties,
32                                                                kinematic_functions)
33            k = kinematic_properties; % shorthand
34            f = kinematic_functions;
35
36            torch_rotation_value = k.torch_fixed_rotation_value*(pi/180);
37
38            part_frame = f.ZP*f.TP*f.G*f.P;
39            z_rot = -atan2(part_frame(2,1), part_frame(1,1))+pi+torch_rotation_value;
```

```

30     P_rot = f.P*[cos(z_rot) -sin(z_rot) 0 0; sin(z_rot) cos(z_rot) 0 0; 0 0 1 0; 0 0
31         0 1];
32
33     f.P(1:3,1:3) = P_rot(1:3,1:3);
34
35     CalculateKinematicFunctions.UpdateKinematicFunctions(k,f);
36
37     delete_rows = size(f.J,1)-6;
38     for i = 1:delete_rows
39         f.J(end,:) = [];
40         f.err_matrix(end) = [];
41     end
42 end%func UpdateKinematicFunctionsForFixedTorchRotation
43
44 function CalculateManipulatorKinematics(kinematic_properties, kinematic_functions)
45     k = kinematic_properties; % shorthand
46     f = kinematic_functions;
47
48     t1R=k.q(3); t2R=k.q(4); t3R=k.q(5); t4R=k.q(6); t5R=k.q(7); t6R=k.q(8);
49
50     f.A1R = [cos(t1R), 0, sin(t1R), k.a1*cos(t1R)
51             sin(t1R), 0, -cos(t1R), k.a1*sin(t1R)
52             0, 1, 0, k.d1
53             0, 0, 0, 1];
54     f.A2R = [cos(t2R), -sin(t2R), 0, k.a2*cos(t2R)
55             sin(t2R), cos(t2R), 0, k.a2*sin(t2R)
56             0, 0, 1, 0
57             0, 0, 0, 1];
58     f.A3R = [cos(t3R), 0, sin(t3R), k.a3*cos(t3R)
59             sin(t3R), 0, -cos(t3R), k.a3*sin(t3R)
60             0, 1, 0, 0
61             0, 0, 0, 1];
62     f.A4R = [cos(t4R), 0, -sin(t4R), 0
63             sin(t4R), 0, cos(t4R), 0
64             0, -1, 0, k.d4
65             0, 0, 0, 1];
66     f.A5R = [cos(t5R), 0, sin(t5R), 0
67             sin(t5R), 0, -cos(t5R), 0
68             0, 1, 0, 0
69             0, 0, 0, 1];
70     f.A6R = [cos(t6R), -sin(t6R), 0, 0
71             sin(t6R), cos(t6R), 0, 0
72             0, 0, 1, k.d6
73             0, 0, 0, 1];
74     f.TR = f.A1R * f.A2R * f.A3R * f.A4R * f.A5R * f.A6R;
75 end%func CalculateManipulatorKinematics

```

```

74
75 function CalculatePositionerKinematics(kinematic_properties, kinematic_functions)
76     k = kinematic_properties; % shorthand
77     f = kinematic_functions;
78
79     t1P=k.q(1); t2P=k.q(2);
80
81     rotx = [1 0 0 0; 0 0 -1 0; 0 1 0 0; 0 0 0 1];
82
83     A1P_1 = [cos(t1P) -sin(t1P) 0 0; sin(t1P) cos(t1P) 0 0; 0 0 1 0; 0 0 0 1];
84     A1P_2 = [1 0 0 0; 0 cos(-pi/2) -sin(-pi/2) 0; 0 sin(-pi/2) cos(-pi/2) 0; 0 0 0
85             1];
86     f.A1P = rotx*A1P_1*A1P_2;
87     A2P_1 = [cos(t2P) -sin(t2P) 0 0; sin(t2P) cos(t2P) 0 0; 0 0 1 0; 0 0 0 1];
88     A2P_2 = [1 0 0 0; 0 cos(0) -sin(0) 0; 0 sin(0) cos(0) 0; 0 0 0 1];
89     f.A2P = A2P_1*A2P_2;
90
91     f.TP = f.A1P * f.A2P;
92 end%func CalculatePositionerKinematics
93
94 function CalculatePositionerBaseTranslation(kinematic_properties, kinematic_functions
95 )
96     k = kinematic_properties; % shorthand
97     f = kinematic_functions;
98
99     f.ZP = [1 0 0 k.pp_base(1); 0 1 0 k.pp_base(2); 0 0 1 k.pp_base(3); 0 0 0 1];
100 end%func CalculatePositionerBaseTranslation
101
102 function CalculateTorchTranslation(kinematic_properties, kinematic_functions)
103     k = kinematic_properties; % shorthand
104     f = kinematic_functions;
105
106     PE = [sqrt(2)/2 0 -sqrt(2)/2 0; 0 1 0 0; sqrt(2)/2 0 sqrt(2)/2 k.dtcp; 0 0 0 1];
107     RE = [-1 0 0 0; 0 1 0 0; 0 0 -1 0; 0 0 0 1];
108     f.E = PE*RE;
109 end%func CalculateTorchTranslation
110
111 function CalculateWorkpieceBaseTranslation(kinematic_properties, kinematic_functions)
112     k = kinematic_properties; % shorthand
113     f = kinematic_functions;
114
115     f.G = [1 0 0 k.workpiece_base(1); 0 1 0 k.workpiece_base(2); 0 0 1 k.
116           workpiece_base(3); 0 0 0 1];
117 end%func CalculateWorkpieceBaseTranslation

```

```

116 function CalculateJacobian(kinematic_properties, kinematic_functions)
117     k = kinematic_properties; % shorthand
118     f = kinematic_functions;
119
120     O = [1 0 0 0; 0 1 0 0; 0 0 1 0];
121     w = [0 0 1 0].';
122     r = [0 0 0 1].';
123
124     v1_cross = cross(O*w, O*(inv(f.A1P)*inv(f.ZP)*f.TR*f.E*r));
125     v1_cross(4) = 0;
126     v1 = -O*inv(f.P)*inv(f.G)*inv(f.A2P)*v1_cross;
127
128     v2_cross = cross(O*w, O*(inv(f.ZP)*f.TR*f.E*r));
129     v2_cross(4) = 0;
130     v2 = -O*inv(f.P)*inv(f.G)*inv(f.A2P)*inv(f.A1P)*v2_cross;
131
132     v3_cross = cross(O*w, O*(f.A1R*f.A2R*f.A3R*f.A4R*f.A5R*f.A6R*f.E*r));
133     v3_cross(4) = 0;
134     v3 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*v3_cross;
135
136     v4_cross = cross(O*w, O*(f.A2R*f.A3R*f.A4R*f.A5R*f.A6R*f.E*r));
137     v4_cross(4) = 0;
138     v4 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.A1R*v4_cross;
139
140     v5_cross = cross(O*w, O*(f.A3R*f.A4R*f.A5R*f.A6R*f.E*r));
141     v5_cross(4) = 0;
142     v5 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.A1R*f.A2R*v5_cross;
143
144     v6_cross = cross(O*w, O*(f.A4R*f.A5R*f.A6R*f.E*r));
145     v6_cross(4) = 0;
146     v6 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.A1R*f.A2R*f.A3R*v6_cross;
147
148     v7_cross = cross(O*w, O*(f.A5R*f.A6R*f.E*r));
149     v7_cross(4) = 0;
150     v7 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.A1R*f.A2R*f.A3R*f.A4R*v7_cross;
151
152     v8_cross = cross(O*w, O*(f.A6R*f.E*r));
153     v8_cross(4) = 0;
154     v8 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.A1R*f.A2R*f.A3R*f.A4R*f.A5R*
        v8_cross;
155
156     Jp = [v1 v2 v3 v4 v5 v6 v7 v8];
157
158     w1 = -O*inv(f.P)*inv(f.G)*inv(f.A2P)*w;
159     w2 = -O*inv(f.P)*inv(f.G)*inv(f.A2P)*inv(f.A1P)*w;

```

```

160 w3 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*w;
161 w4 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.A1R*w;
162 w5 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.A1R*f.A2R*w;
163 w6 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.A1R*f.A2R*f.A3R*w;
164 w7 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.A1R*f.A2R*f.A3R*f.A4R*w;
165 w8 = O*inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.A1R*f.A2R*f.A3R*f.A4R*f.A5R*w;
166
167 Jo = [w1 w2 w3 w4 w5 w6 w7 w8];
168
169 f.J = [Jp; Jo];
170
171 if k.allow_positioner_movement == 0
172     k.q(1)=0;
173     k.q(2)=0;
174 end%if
175 if k.torch_rotation(1) ~= 'n' && k.torch_rotation(1) ~= 'f'
176     f.J(6,:)=[];
177 end%if
178 if k.enable_gravity_aligned_constraint == 1
179     if k.allow_positioner_movement == 0
180         fprintf('Cannot enable GA constraint if allow_positioner_movement=0')
181         return;
182     end%if
183     R = [1 0 0 0; 0 0 1 0];
184     w = [0 0 1 0].';
185     d1 = R*f.ZP*f.A1P*w;
186     d2 = R*f.ZP*w;
187
188     f.J(end+1,:)=zeros(1,8);
189     f.J(end+1,:)=zeros(1,8);
190     f.J(end-1:end,1)=d1;
191     f.J(end-1:end,2)=d2;
192 end%if
193 end%func CalculateJacobian
194
195 function CalculateErrorVector(kinematic_properties, kinematic_functions)
196     k = kinematic_properties; % shorthand
197     f = kinematic_functions;
198
199     O = [1 0 0 0; 0 1 0 0; 0 0 1 0];
200
201     Err_calc = inv(f.E)*inv(f.TR)*f.ZP*f.TP*f.G*f.P;
202     dx = Err_calc(1,4); dy = Err_calc(2,4); dz = Err_calc(3,4);
203     z_rot = atan2(Err_calc(2,1),Err_calc(1,1));
204     y_rot = atan2(-Err_calc(3,1),sqrt((Err_calc(3,2))^2+(Err_calc(3,3))^2));

```



```

205     x_rot = atan2(Err_calc(3,2),Err_calc(3,3));
206
207     if x_rot > pi
208         x_rot = -(2*pi-x_rot)
209     end
210     if x_rot < -pi
211         x_rot = 2*pi + x_rot
212     end
213     if y_rot > pi
214         y_rot = -(2*pi-y_rot)
215     end
216     if y_rot < -pi
217         y_rot = 2*pi + y_rot
218     end
219     if z_rot > pi
220         z_rot = -(2*pi-z_rot)
221     end
222     if z_rot < -pi
223         z_rot = 2*pi + z_rot
224     end
225
226     if dx > k.tlim
227         dx = k.tlim;
228     end
229     if dy > k.tlim
230         dy = k.tlim;
231     end
232     if dz > k.tlim
233         dz = k.tlim;
234     end
235     if dx < -k.tlim
236         dx = -k.tlim;
237     end
238     if dy < -k.tlim
239         dy = -k.tlim;
240     end
241     if dz < -k.tlim
242         dz = -k.tlim;
243     end
244
245     u_err = [dx dy dz 1].';
246     phi_err = [x_rot y_rot z_rot 0].';
247     Act = inv(f.P)*inv(f.G)*inv(f.TP)*inv(f.ZP)*f.TR*f.E;
248
249     f.err_matrix = [O*Act*u_err; -O*Act*phi_err];

```

```

250
251     if k.torch_rotation(1) ~= 'n' && k.torch_rotation(1) ~= 'f'
252         f.err_matrix(6) = [];
253     end%if
254     if k.enable_gravity_aligned_constraint == 1
255         if k.allow_positioner_movement == 0
256             fprintf('Cannot enable GA constraint if allow_positioner_movement=0')
257             return;
258         end%if
259         w = [0 0 1 0].';
260         g = [0; 0; 1];
261         Rcross = [1 0 0; 0 1 0];
262         cross_val = f.ZP*f.A1P*f.A2P*f.G;
263         cross_val(:,4) = [];
264         cross_val(4,:) = [];
265
266         f2g_err = Rcross*(cross(cross_val*f.ng,g));
267         f.err_matrix = [f.err_matrix; f2g_err];
268
269         if f.ng(3) > 0.99
270             f.err_matrix(end) = 0;
271             f.err_matrix(end-1) = 0;
272         end%if
273     end%if
274 end%func CalculateErrorVector
275
276 function CalculateNullSpaceValue(kinematic_properties, kinematic_functions)
277     k = kinematic_properties; % shorthand
278     f = kinematic_functions;
279
280     if k.torch_rotation(1) == 'j'
281         t1 = k.q(1); t2 = k.q(2); t3 = k.q(3); t4 = k.q(4); t5 = k.q(5); t6 = k.q(6)
282             ; t7 = k.q(7); t8 = k.q(8);
283         k0 = 5;
284         qd0 = [0, 0, -t3/(3*pi^2), -(2*t4 - pi)/(6*pi^2), 0, 0, -(2*(2*t7 + pi)/2))
285             /(3*pi^2), 0].';
286         f.null_val = (eye(8) - pinv(f.J)*f.J)*k0*qd0;
287
288     elseif k.torch_rotation(1) == 's'
289         joint_vals_current = k.q;
290         joint_vals_old = k.q_old;
291         qd0 = zeros(length(joint_vals_current),1);
292
293         for i=1:length(joint_vals_current)
294             %Calculate Jacobian with new individual joint value

```

```

293     joint_vals_independent = joint_vals_old;
294     joint_vals_independent(i,1) = joint_vals_current(i,1);
295     k.q = joint_vals_independent;
296
297     CalculateKinematicFunctions.CalculateTransformationMatrices(k,f);
298     CalculateKinematicFunctions.CalculateJacobian(k,f);
299     CalculateKinematicFunctions.CalculateErrorVector(k,f);
300
301     %Scale Jacobian
302     scale_val = [(1/100) (1/100) (1/100) 1 1 1].';
303     scale_val_old = [(1/100) (1/100) (1/100) 1 1 1].';
304     if size(f.J,1)==7
305         scale_val(7)=1;
306     end%if
307     if size(f.J_old,1)==7
308         scale_val_old(7)=1;
309     end%if
310     if size(f.J,1)==8
311         scale_val(7)=1;
312         scale_val(8)=1;
313     end%if
314     if size(f.J_old,1)==8
315         scale_val_old(7)=1;
316         scale_val_old(8)=1;
317     end%if
318     J_scale = scale_val.*f.J;
319     J_old_scale = scale_val_old.*f.J_old;
320
321     w = sqrt(det(J_scale*J_scale.'));
322     w_old = sqrt(det(J_old_scale*J_old_scale.'));
323
324     qd0(i,1) = ((w-w_old)/(0.001));
325 end%for
326 k0 = 0.00001;
327 f.null_val = (eye(8) - pinv(f.J)*f.J)*k0*qd0;
328 f.null_val;
329
330 k.q = joint_vals_current;
331 %CalculateKinematicFunctions.UpdateKinematicFunctions(k,f);
332 CalculateKinematicFunctions.CalculateTransformationMatrices(k,f);
333 CalculateKinematicFunctions.CalculateJacobian(k,f);
334 CalculateKinematicFunctions.CalculateErrorVector(k,f);
335
336 elseif k.torch_rotation(1) == 'b'

```

```

337     t1 = k.q(1); t2 = k.q(2); t3 = k.q(3); t4 = k.q(4); t5 = k.q(5); t6 = k.q(6)
        ; t7 = k.q(7); t8 = k.q(8);
338
339     k0_1=0.1;
340
341     qd0_1 = [0, 0, -t3/(3*pi^2), -(2*t4 - pi)/(6*pi^2), 0, 0, -(2*(2*t7 + pi/2))
        /(3*pi^2), 0].';
342
343
344     joint_vals_current = k.q;
345     joint_vals_old = k.q_old;
346
347     qd0 = zeros(length(joint_vals_current),1);
348
349     for i=1:length(joint_vals_current)
350         %Calculate Jacobian with new individual joint value
351         joint_vals_independent = joint_vals_old;
352         joint_vals_independent(i,1) = joint_vals_current(i,1);
353         k.q = joint_vals_independent;
354
355         %CalculateKinematicFunctions.UpdateKinematicFunctions(k,f);
356         CalculateKinematicFunctions.CalculateTransformationMatrices(k,f);
357         CalculateKinematicFunctions.CalculateJacobian(k,f);
358         CalculateKinematicFunctions.CalculateErrorVector(k,f);
359
360         %Scale Jacobian
361         scale_val = [(1/100) (1/100) (1/100) 1 1 1].';
362         scale_val_old = [(1/100) (1/100) (1/100) 1 1 1].';
363         if size(f.J,1)==7
364             scale_val(7)=1;
365         end%if
366         if size(f.J_old,1)==7
367             scale_val_old(7)=1;
368         end%if
369         if size(f.J,1)==8
370             scale_val(7)=1;
371             scale_val(8)=1;
372         end%if
373         if size(f.J_old,1)==8
374             scale_val_old(7)=1;
375             scale_val_old(8)=1;
376         end%if
377         J_scale = scale_val.*f.J;
378         J_old_scale = scale_val_old.*f.J_old;
379
380         w = sqrt(det(J_scale*J_scale.'));
381         w_old = sqrt(det(J_old_scale*J_old_scale.'));

```

```

380         qd0_2(i,1) = ((w-w_old)/(0.001));
381     end%for
382     k0_2 = 0.0000001;
383     k.q = joint_vals_current;
384
385     CalculateKinematicFunctions.CalculateTransformationMatrices(k,f);
386     CalculateKinematicFunctions.CalculateJacobian(k,f);
387     CalculateKinematicFunctions.CalculateErrorVector(k,f);
388
389     f.null_val = (eye(8) - pinv(f.J)*f.J)*(k0_1*qd0_1+k0_2*qd0_2);
390 else
391     t1 = k.q(1); t2 = k.q(2); t3 = k.q(3); t4 = k.q(4); t5 = k.q(5); t6 = k.q(6)
        ; t7 = k.q(7); t8 = k.q(8);
392     k0=1000;
393     qd0 = [0, 0, -t3/(3*pi^2), -(2*t4 - pi)/(6*pi^2), 0, 0, -(32*(2*t7 + pi/8))
        /(3*pi^2), 0].';
394     f.null_val = (eye(8) - pinv(f.J)*f.J)*k0*qd0;
395 end%if
396 end%func CalculateNullSpaceValue
397 end%Static Methods
398 end%class InverseKinematicsSolverFunctions

```

Vita

Ethan Vals first began pursuing engineering after discovering his love for math and science in high school while taking dual enrollment classes at Pellissippi State Community College. He began engineering school at the University of Tennessee and quickly settled on pursuing mechanical engineering. The summer after his freshman year, Ethan began working as an undergraduate research assistant at the University of Tennessee under Dr. William R. Hamel, with his research focused on large-scale additive manufacturing of metals. After graduating with his Bachelors in mechanical engineering in only three and a half years, Ethan began his Masters in mechanical engineering with a focus on robotics. Ethan's work experience includes multiple design projects, MATLAB programming of a path planning program, robot programming and code generation, robot operation and welding, and the implementation of a custom robotics inverse kinematics program. In his free time, Ethan enjoys spending time with his friends and family and pursuing his many hobbies, including writing, playing piano, backpacking, or building his latest 3D printed project.