UNIVERSITY OF CAMBRIDGE
Department of Engineering

Conformal Geometric Algebra for Robot Kinematics

Author Name: Zach Lambert
Supervisor: Professor Joan Lasenby
Date: 30/05/2021

I hereby declare that, except where specifically indicated, the work submitted herin is my own original work.

Signed _____ *date* 30/05/21

Conformal Geometric Algebra for Robot Kinematics

Zach Lambert Pembroke College

Technical Abstract

Robot kinematics is the problem of relating the joint positions of a robot to the position and orientation of an end effector, collectively called its pose. This is necessary for controlling the end effector of a robot as well as knowing the position of all its links, for visualisation and collision avoidance.

Conventionally, this is done with standard algebraic techniques, linear algebra and quaternions. Conformal geometric algebra provides an alternative method. In general, geometric algebra (GA) is a type of algebra that it well suited to geometric operations. Conformal geometric algebra (CGA) is a specific type of geometric algebra that can be used to perform intersection of geometric primitives and represent rigid body transformations, among other things.

This makes CGA well suited to many robotics applications, such as computer vision and kinematics. This project set out to investigate the practical use of CGA for kinematics by controlling a delta and serial robot. Kinematics functions were implemented using conventional methods and CGA for comparison, then used by the robots to perform a manipulation tasks.

An introduction to geometric algebra and conformal geometric algebra is given first, explaining key concepts and results required for kinematics. In particular, the use of conformal geometric algebra to represent and intersect geometric primitives is presented, to be used for the delta robot kinematics.

Following this, solutions to the kinematics of the delta robot and serial robot are given, for forward kinematics and inverse kinematics. Solutions are first given using conventional methods, then using CGA. Prior to solving the serial robot kinematics, an overview of representing rigid body transformations is given, for both methods.

To organise the software, the robotics operating system (ROS) will be used to control the robots. ROS facilitates distributed robotics software, where different components are run as separate processes, called nodes. ROS acts as a middleware between these nodes, allowing communication between nodes via a pub/sub protocol or client/server protocol.

A number of pieces are code are required for the final result. The first is a *C++* CGA library. To allow for fine tuning of this library, a custom CGA library was developed. The bulk of the library provides data structures and operations for performing CGA operations, generated using a custom code generator, written in *Python*. The remainder of the library provides functions for geometry and rigid body transformations.

Following the CGA library, a standalone robotics library was developed, independent from ROS. This provides a standard interface to a base robot class for performing various kinematics operations. Child classes were implemented for the

delta and serial robot. By using a compile-time flag, the compilation could select between different source files, to select a conventional or CGA implementation.

Finally, a ROS package was developed. This contained descriptions of the kinematic structures of the delta and serial robot, specified by `URDF` files. For each robot type, a state publisher and controller node were written. The state publisher node would perform the forward kinematics of the robot, allowing the end effector pose to be calculated and the robot visualised. The controller node would listen to commands to follow a particular velocity or execute a trajectory to a given goal.

The end goal was to allow the robots to be manually controlled, saving a series of waypoints to follow, then having it automatically follow these waypoints. A commander node was written to process user input, store saved waypoints and send appropriate commands to the controller.

The control code was first applied to simulated robots using the Gazebo robotics simulator, allowing for faster development. Following this, it was able to be applied to real robots, making use of a standard control architecture provided by ROS such that the robot is always controlled the same way, simulated or real.

To evaluate the viability of CGA for robot kinematics, a comparison of execution time with conventional methods was given, then a qualitative overview of its ability to perform different tasks.

Execution times for kinematic functions using CGA were consistently slower, but not significantly so. For functions with closed-form solutions, execution times were still of the order of micro-seconds, with little variation, making them sufficiently fast.

For the inverse kinematics of the serial robot, the only function using numerical methods, execution were far longer, dependent on convergence time, although the CGA function still had larger execution times.

For both the delta and serial robot, using CGA made programming easier, given that the programmer understands CGA. The geometrical nature of the delta robot kinematics was much easier to solve using CGA, as opposed to the tedious algebraic solution using standard algebra. For the serial robot, the kinematics were quite similar, except that CGA could be used for all calculations, providing consistency, whereas conventional methods often had to work with a number of algebras, such as homogeneous matrices and quaternions, often requiring conversions.

CGA was able to be used when manually controlling the two robots, or having them perform tasks. Both robots were able to pick and place objects, as well as stack objects on top of one another. Any limitations were due to simulation or hardware limitations.

This shows that CGA is a viable method for performing kinematics in practical control applications. Further work could extend this to the kinematics of other robot topologies; investigate the use of CGA for more advanced control methods, such as constrained control; or integrate computer vision into control, such as with visual servoing, which can also be solved using CGA.

1 Introduction

Robot kinematics is the problem of relating the joint positions of a robot structure to the configuration of the robot, particular the position and orientation of the end effector: the gripper or tool which the robot is designed to move.

There are four problems that need to be solved:

- **Forward kinematics**:
  For a particular set of joint positions, what is the end effector pose (position and orientation)?
- **Inverse kinematics**:
  For a desired end effector pose, what are the required joint positions?
- **Instantaneous forward kinematics**:
  For a particular set of joint positions and velocities, what is the end effector twist (linear and angular velocity)?
- **Instantaneous inverse kinematics**:
  For a set of joint positions and desired end effector twist, what are the required joint velocities?

A robot consists of a number of rigid bodies, called *links*. These links are connected by joints, which constrain their relative motion. The configuration of links and joints is called the kinematic structure, or topology of the robot.

The topology of a robot will determine the nature of these solutions. The broadest classification is between serial and parallel robots.

1.1 Serial robots

With a serial robot, each link only has two joints, connecting it to its two neighbours, except for the first and last links which only have a single joint. The result is that the kinematic structure forms a serial chain, as shown in Figure 1.

With a serial robot, the forward kinematics can be calculated easily. However, the inverse kinematics doesn't have a closed-form solution in general, requiring numerical methods.

The exception to this is industrial robots: serial robots with a specific structure such that the inverse kinematics has a closed-form solution.

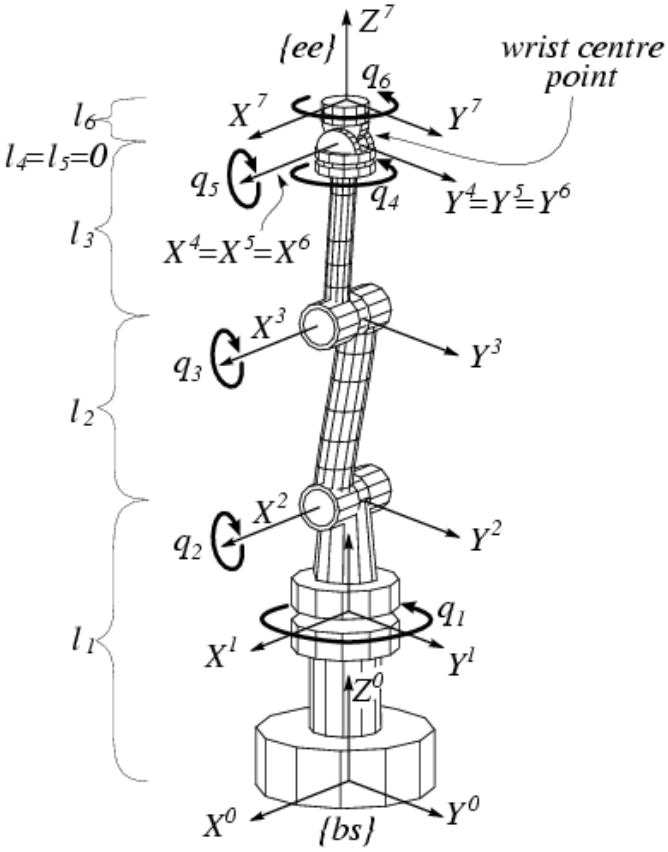

Figure 1: The kinematic structure of a serial robot[1].

1.2 Parallel robots

A parallel robot is defined by the presence of loops in the kinematic structure. This means that multiple serial chains connect to a common link. A good example is the delta robot, shown in Figure 2. It has three serial chains, consisting of a rocker arm and connecting rod, which meet at a base plate.

A parallel robot imposes the constraint that each serial chain meet at the end effector, making forward kinematics non-trivial. Often numerical methods are required.

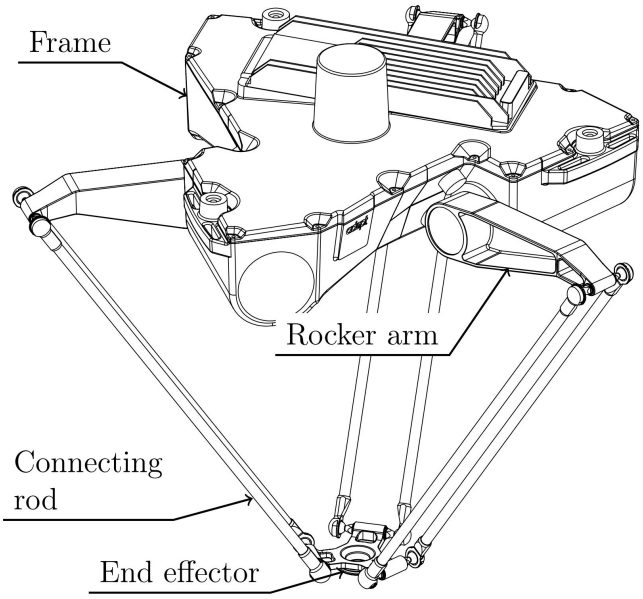

Figure 2: Structure of a delta robot[2], an example of a parallel robot. It has three degrees of freedom, such that the position of the end effector is fully defined by the three rocker arm joint positions.

On the other hand, inverse kinematics only needs to be solved for each serial chain individually. These serial chains always have simple structures themselves, with low degrees of freedom; otherwise the forward kinematics would be under-constrained. This makes the inverse kinematics simple.

The delta robot is an example of a parallel robot where the forward and inverse kinematics both have closed form solutions.

Due to the symmetry and use of ball joints on the connecting rods, the end effector is constrained to lie flat. The three controlled joints, connecting the frame to the rocker arms fully define the end effector position.

1.3 Motivation

Conventionally, robot kinematics is solved using a combination of linear algebra and quaternions, used for kinematics problems both with closed form solutions and numerical solutions.

Geometric algebra (GA) is a type of algebra that can be applied to geometrical problems more naturally than other methods. One specific type of geometric algebra called conformal geometric algebra (CGA) extends this functionality to support a wider range of operations, such as representing geometrical primitives and performing rigid body transformations.

This makes CGA well suited to robotics and has seen success for a range of applications, including computer vision[3] and kinematics.

1.4 Project objectives

This project aims to investigate the use of CGA for kinematics in a practical robotics application. Control will be implemented for both the delta robot and a general serial robot, using the robotics operating system (ROS)[4] to organise the software.

This follows from work by Wei L in the previous year[5] where the kinematics of the delta robot was solved using CGA then applied to the control a delta robot in the Unity game engine[6].

The kinematics of the serial robot will also be solved using CGA, then solutions to both kinematics will be implemented in a *C++* library to use in ROS.

To understand how CGA compares to conventional methods, the kinematics library will provide implementations using conventional and CGA methods. The only difference will be how kinematics are solved, allowing benchmarks of execution times to be measured and provide a fair comparison.

Finally, the end goal is to apply this to a control task. The ROS package developed will allow for the manual control of the two robots and planning out of a task by setting waypoints. The robot will then be able to automatically move between these waypoints to perform the task.

2 Geometric algebra (GA)

2.1 Overview of GA

Geometric algebra originates from work by Grassmann and Clifford and experienced increased interest in recent history after Hestenes presented Clifford algebra as a new "unified language for mathematics and physics."[7]

Geometric algebra is essentially the same as Clifford algebra, but is used for geometric applications. For simplicity, the name "Geometric algebra" will be used.

Note that throughout this report, vectors in geometric algebra are *not* written using bold font. This is because in geometric algebra, all objects, including scalars and vectors, belong to the same algebra. Bold font will be used when looking at vectors in standard vector algebra.

2.1.1 Vectors and planes

A vector $x \in \mathbb{R}^n$ has the following properties:

- Attitude: The subspace spanned by the vector. All vectors $x = \lambda a$ have the same attitude.
- Handedness: The sign of the vector. Vectors $x = a$ and $x' = -a$ have the opposite handedness.
- Size: The *weight* of the vector. Vector $x = 2a$ has double the size of vector $x' = a$.

A plane is defined as the subspace spanned by two linearly independent basis vectors. However, we can also consider an *oriented plane* which like a vector, has an attitude, handedness and size.

These oriented planes are called *bivectors*, and are formed by the *outer product* of two vectors:

$$B = a \wedge b$$

Consider the following three bivectors:

$$B_1 = a \wedge b$$
$$B_2 = 2a \wedge b$$
$$B_3 = a \wedge (-b)$$

All these bivectors have the same attitude: a plane spanned by a and b. However, bivector B_2 has double the size of B_1 and B_3 and bivector B_3 has the opposite handedness of bivectors B_1 and B_2.

2.1.2 The outer product, blades and multivectors

The outer product is a new operation, which as seen earlier *combines* vectors. It has the following properties:

- Associative: a∧(b∧c) = (a∧b)∧c = a∧b∧c
- Distributive: a∧(b + c) = a∧b + a∧c
- Anti-commutative: a∧b = -b∧a
- a∧a = 0

The final property makes sense. A plane can't be defined by two linearly dependent vectors. Following this, the outer product is shown to only care about the orthogonal component. For b' orthogonal to a:

$$a \wedge b = a \wedge (b' + \lambda a) = a \wedge b'$$

Blades and multivectors

The outer product of r basis vectors is called an *r-blade*, or a blade with the *grade* of r.

A multivector is a linear combination of blades of any grade. When all blades have the same grade r, it is called a homogeneous multivector of grade r.

Homogeneous multivectors of grade 2 are also referred to as bivectors, as seen earlier. Similarly, the names trivector and quadvector are also used.

Grade selection

For a general multivector, the grade r part can be extracted with:

$$A_r = \langle A \rangle_r$$

For the grade 0 (scalar) part, the subscript is dropped:

$$\langle A \rangle_0 = \langle A \rangle$$

Interpreting blades and multivectors

Mixed grade multivectors don't have a geometric interpretation, although are used for some operations as seen later.

Bivectors represent oriented planes, trivectors oriented volumes, and higher grade homogeneous multivectors, oriented hyper-volumes.

Blades act as *basis bivectors*, and so on. For example, consider the following bivectors:

$$\begin{aligned} A &= b \wedge c \\ B &= c \wedge a \\ C &= a \wedge b \\ X &= (2a + b) \wedge (c - a) \\ &= 2a \wedge c + b \wedge c - b \wedge a \\ &= -2c \wedge a + b \wedge c + a \wedge b \\ &= -2B + A + C \end{aligned}$$

A, B and C are blades, since they can be expressed as the outer product of vectors. All bivectors can be expressed in terms of these blades, such as X.

2.1.3 The inner product

The inner product between vectors $a \cdot b$ is equivalent to the standard scalar product between vectors.

The inner product does exist between any two multivectors. However, defining this requires the *geometric product*.

2.1.4 The geometric product

Both the inner and outer product can be expressed in terms of the geometric product. This operation acts as the foundation of geometric algebra.

The geometric product between a and b is denoted ab and has the properties:

- Associative: $a(bc) = (ab)c = abc$
- Distributive: $a(b + c) = ab + ac$
- Neither commutative nor anticommutative
- $aa = a^2 \in \mathbb{R}$

The geometric product between vectors relates to the inner and outer products with:

$$\begin{aligned} ab &= \frac{1}{2}(ab + ba) + \frac{1}{2}(ab - ba) \\ &= a \cdot b + a \wedge b \end{aligned}$$

where the inner and outer products are defined in terms of the geometric product as:

$$a \cdot b = \frac{1}{2}(ab + ba) \tag{1}$$

$$a \wedge b = \frac{1}{2}(ab - ba) \tag{2}$$

This separation derives from the fact that $ab + ba \in \mathbb{R}^n$, so captures the scalar part of ab while the other component of ab captures the *non-scalar* part:

$$\begin{aligned} a \cdot b &= \frac{1}{2}(ab + ba) \\ &= \frac{1}{2}((a + b)^2 - a^2 - b^2) \in \mathbb{R}^n \end{aligned}$$

Useful results

For orthogonal vectors, the outer and geometric product are equivalent:

$$a \cdot b = 0 \implies ab = a \wedge b$$

The geometric product can be reversed with either of the following, using eq. (1) and eq. (2):

$$ab = 2a \cdot b - ba \qquad ab = 2a \wedge b + ba \tag{3}$$

2.1.5 Canonical basis of geometric algebra

It is useful to express all multivectors using a set of basis blades. For the basis vectors $a_1, a_2, \ldots, a_n$, this gives basis blades $a_1 \wedge a_2, a_1 \wedge a_3, \ldots, a_1 \wedge a_2 \wedge a_3, \ldots$.

This doesn't place any restriction on the values $a_i \cdot a_j$ between basis vectors. Instead, it is more useful to consider a set of orthonormal basis vectors $e_1, \ldots e_n$ where:

$$e_i \cdot e_j = \begin{cases} \pm 1 & i = j \\ 0 & i \neq j \end{cases}$$

The result is that the geometric product between basis vectors is:

$$e_i e_j = \begin{cases} e_i^2 = \pm 1 & i = j \\ e_i \wedge e_j = -e_j e_i & i \neq j \end{cases}$$

The geometric product between different products now *anticommutes*, which is more useful than the geometric product between two general vectors, which neither commutes nor anticommutes.

Signature

e_i^2 is equal to $+1$ or -1. A canonical basis has signature $\mathcal{G}(p, q)$, where $p + q = n$ and:

$$e_i^2 = \begin{cases} +1 & 1 \leq i \leq p \\ -1 & p + 1 \leq i \leq p + q \end{cases}$$

In other words, p basis vectors square to $+1$ and q basis vectors square to -1.

Expressing multivectors using basis vectors

Any multivector can be expressed as a linear combination of blades, formed by the outer product of vectors.

For a canonical basis, any multivector can be expressed by the linear combination of blades, formed by the *geometric product* of basis vectors.

Canonical basis

The canonical basis of a geometric algebra is the set of basis blades provided. Using orthonormal basis vectors $e_1, \ldots, e_n \in \mathbb{R}^n$, this gives the canonical basis:

$$\begin{aligned} \{&1, \\ &e_1, e_2, \ldots, e_n \\ &e_{12}, e_{13}, \ldots, e_{23}, e_{24}, \ldots, e_{(n-1)n} \\ &e_{123}, e_{124}, \ldots \\ &\ldots \\ &e_{12\ldots n}\} \end{aligned}$$

where e_{12} denotes $e_1 e_2$ and so on.

For grade r, there are $\binom{n}{r}$ basis blades, giving a total dimension of the canonical basis as:

$$\sum_{r=0}^{n} \binom{n}{r}$$

2.1.6 The geometric product between multivectors

Any multivector can be expressed as a linear combination between basis blades. Since the geometric product is distributive, the geometric product between multivectors is simply found by summing the geometric product between all components.

To evaluate the result of the geometric product between two basis blades, this uses the anticommution property to remove repeated vectors.

Example 1, no repeated vectors:

$$e_{123} e_{45} = e_1 e_2 e_3 e_4 e_5 = e_{12345}$$

Example 2, repeated e_2 with $e_2^2 = +1$:

$$e_{123} e_2 = e_1 e_2 e_3 e_2 = -e_1 e_2 e_2 e_3 = -e_{12}$$

Example 3, repeated e_2 with $e_2^2 = -1$:

$$e_{123} e_2 = e_1 e_2 e_3 e_2 = -e_1 e_2 e_2 e_3 = +e_{12}$$

Whenever two vectors are swapped, this flips the sign.

Grade of the geometric product between blades

Blades of grade s and grade t will produce a blade of grade r, where

$$|s-t| \leq r \leq \min(n, s+t)$$

If all vectors of the lower grade blade are contained in the other, this gives the minimum possible blade of grade $|s-t|$. If no vectors are shared between the blades, the maximum grade $s+t$ is achieved. There are only n basis vectors, so the maximum number of distinct vectors between the two blades is n and if $s+t > n$, there must be some shared vectors.

For homogeneous multivectors of grades s and t, denoted A_s and B_t, the resultant multivector $C = A_s B_t$ will contain blades ranging from grade $|s-t|$ to grade $\min(n, s+t)$:

$$\begin{aligned} C &= A_s B_t \\ &= \langle A_s B_t \rangle_{|s-t|} + \langle A_s B_t \rangle_{|s-t|+1} \\ &\quad + \cdots + \langle A_s B_t \rangle_{\min(n, r+s)} \end{aligned}$$

2.1.7 Generalising the inner and outer product

For vectors, the inner and outer products are expressed using the geometric product:

$$a \cdot b = \frac{1}{2}(ab + ba) \qquad a \wedge b = \frac{1}{2}(ab - ba)$$

$$ab = a \cdot b + a \wedge b$$

Here, the inner product extracts the *minimum grade* component of the geometric product, and the outer product, the *maximum grade* component of the geometric product.

Therefore, for general homogeneous multivectors A_s and B_t:

$$A_s \cdot B_t = \langle AB \rangle_{|s-t|} \tag{4}$$

$$A_s \wedge B_t = \langle AB \rangle_{s+t} \quad \text{for } s+t \leq n \tag{5}$$

which extends to general multivectors since the operators are distributive.

Specific expressions for the inner and outer product

Consider the geometric product between a vector a and homogeneous multivector of grade r, B_r:

$$\begin{aligned} aB_r &= \langle aB_r \rangle_{r-1} + \langle aB_r \rangle_{r+1} \\ B_r a &= \langle B_r a \rangle_{r-1} + \langle B_r a \rangle_{r+1} \\ &= \begin{cases} \langle aB_r \rangle_{r-1} - \langle aB_r \rangle_{r+1} & r \text{ odd} \\ -\langle aB_r \rangle_{r-1} + \langle aB_r \rangle_{r+1} & r \text{ even} \end{cases} \end{aligned}$$

where the sign changes between $\langle aB_r \rangle_{r+1}$ and $\langle B_r a \rangle_{r+1}$ if r is odd, because an odd number of swaps are required to move a from the front to back. For the lower grade part, there is one less swap since at some point a swaps places with its basis vector, which commutes.

Using $a \cdot B_r = \langle aB_r \rangle_{r-1}$ and $a \wedge B_r = \langle aB_r \rangle_{r+1}$:

$$a \cdot B_r = \begin{cases} \frac{1}{2}(aB + Ba) & r \text{ odd} \\ \frac{1}{2}(aB - Ba) & r \text{ even} \end{cases} \tag{6}$$

$$a \wedge B_r = \begin{cases} \frac{1}{2}(aB - Ba) & r \text{ odd} \\ \frac{1}{2}(aB + Ba) & r \text{ even} \end{cases} \tag{7}$$

with commutative properties:

$$a \cdot B_r = \begin{cases} B_r \cdot a & \text{odd} \\ -B_r \cdot a & r \text{ even} \end{cases}$$

$$a \wedge B_r = \begin{cases} -B_r \wedge a & \text{odd} \\ B_r \wedge a & r \text{ even} \end{cases}$$

These expressions can be found for vectors, since the geometric product will always be split between two different grades, that have different sign changes when reversing the order of operations. For other combinations of homogeneous multivectors, neat expressions don't exist.

Interpreting the inner and outer products

A homogeneous multivector of grade r represents a subspace of dimension r in a vector space $\mathbb{R}^n$, called the attitude, with a size and orientation.

The outer product between two blades finds the *union* of these two subspaces, multiplying their

sizes, with an attitude that depends on the order of basis vectors.

The inner product between two blades will find the *orthogonal complement*. ie: A blade with all basis vectors in the larger grade blade, that aren't contained in the smaller grade blade, is returned. Again, the size is multiplied and the attitude depends on the order of basis vectors.

The interpretation will be made clearer when looking specifically at geometric algebra for 3D space.

2.1.8 Pseudoscalar and duality

For a geometric algebra of a vector space with dimension n, the largest possible grade blade is $e_{12\ldots n}$ and there is only a single blade of this grade.

This blade is called the *pseudoscalar*, denoted I, since like the scalar, there is only one blade of its grade. The pseudoscalar always satisfies $I^2 = \pm 1$, the sign dependent on the dimension and signature.

Taking the geometric product with a multivector is called the *duality transform*, and the resultant multivector is called the *dual*. For example the dual A^* of A is formed by:

$$A^* = AI$$

Note that it post-multiplies by the pseudoscalar. The duality transform commutes or anticommutes for homogeneous multivectors, dependent on the grade and dimension. For a homogeneous multivector of grade r, with dimension n, moving the pseudoscalar to the front requires $r(n-1)$ swaps, meaning:

$$IA_r = (-1)^{r(n-1)} A_r I$$

The duality product takes the complement of a blade. For example, for $\mathbb{R}^3$:

$$e_1 I = e_{23} \quad e_2 I = e_{31} \quad e_3 I = e_{12}$$

Swapping the inner and outer product

Since the pseudoscalar extracts the complement, it can also be used to swap the inner and outer products, making use of the general definitions of the inner and outer production in eq. (4) and eq. (5).

Assuming $r + s \leq n$:

$$\begin{aligned} A_r \cdot (B_s I) &= A_r \cdot \langle B_s I \rangle_{n-s} \\ &= \langle A_r B_s I \rangle_{n-s-r} \\ &= \langle A_r B_s I \rangle_{n-(s+r)} \\ &= \langle A_r B_s I \rangle_{s+r} I \\ A_r \cdot (B_s I) &= (A_r \wedge B_s) I \end{aligned} \tag{8}$$

2.1.9 Reverse

The reverse $\widetilde{A}$ of a multivector A is defined as the result when the order of basis vectors are reversed.

For example:

$$\begin{aligned} A = e_1 + e_{12} + e_{123} \quad \Longrightarrow \quad \widetilde{A} &= e_1 + e_{21} + e_{321} \\ &= e_1 - e_{12} - e_{123} \end{aligned}$$

For an individual blade of grade r, the sign change follows the pattern:

r	num swaps	sign change
1	0	+1
2	1	-1
3	3	-1
4	6	+1
6	15	-1
⋮	⋮	⋮

2.1.10 Norm

For a homogeneous multivector A_r, the norm is defined as:

$$|A_r|^2 = |A_r \cdot A_r| = |\langle A_r^2 \rangle| \tag{9}$$

This is simply the sum of the square of the components.

For a vector:

$$\begin{aligned} a &= a_1 e_1 + a_2 e_2 + a_3 e_3 \\ a \cdot a &= a_1^2 + a_2^2 + a_3^2 \\ |a|^2 &= a \cdot a \end{aligned}$$

For a bivector:

$$\begin{aligned} B &= B_1 e_{23} + B_2 e_{31} + B_3 e_{12} \\ B \cdot B &= -(B_1^2 + B_2^2 + B_3^2) \\ |B|^2 &= -B \cdot B \end{aligned}$$

And so on.

2.2 Using GA for geometric operations

This section focus on the application to 3D geometry. Therefore, it uses the geometric algebra with signature $\mathcal{G}(3,0)$ and canonical basis:

$$\{1, e_1, e_2, e_3, e_{23}, e_{31}, e_{12}, I\}$$

For consistency, the selected 2-blades are the duals of the basis vectors.

2.2.1 Projection

For a normalised vector u, the inner product $u \cdot x$ will project x onto this direction.

The inner product with a bivector produces a similar result. Define a bivector $B = b_1 \wedge b_2 = b_1 b_2$, with $b_1 \cdot b_2 = 0$. The dot product with a vector a gives:

$$\begin{aligned} a \cdot B &= \frac{1}{2}(aB - Ba) \\ &= \frac{1}{2}(ab_1b_2 - b_1b_2a) \\ &= \frac{1}{2}((2a \cdot b_1 - b_1 a)b_2 - b_1(2a \cdot b_2 - b_2 a)) \\ &= (a \cdot b_1)b_2 - (a \cdot b_2)b_1 - \frac{1}{2}b_1 a b_2 + \frac{1}{2}b_1 a b_2 \\ &= (a \cdot b_1)b_2 - (a \cdot b_2)b_1 \end{aligned} \tag{10}$$

which uses eq. (3) to reverse the order of the geometric product.

This shows that $a \cdot B$ projects the vector a onto the plane defined by B and rotates by 90 degrees anticlockwise. $B \cdot a = -a \cdot B$ so would rotate clockwise instead.

2.2.2 Reflections

A reflection of a in a plane with unit normal n ($n^2 = 1$) is given by:

$$\begin{aligned} a' &= a - 2(a \cdot n)n \\ &= a - (an + na)n \\ &= a - a - nan \\ &= -nan \end{aligned} \tag{11}$$

By sandwiching a vector between another (and negating), this gives the reflection.

Reflecting other multivectors

This can be generalised to the reflection of any blade and therefore any multivector.

Consider a blade $B_r = b_1 b_2 \dots b_r$, where $\{b_i\}$ are some selection of basis vectors, and reflect along some basis vector n.

If b_i is parallel to n, then $b_i n = n b_i$.
If b_i is orthogonal, then $b_i n = -n b_i$.

If the blade doesn't contain n, meaning it is orthogonal to n, then:

$$nB_r n = n(-1)^r n B_r = (-1)^r B_r$$

If the blade does contain n, meaning it has a component in n, then:

$$nB_r n = n(-1)^{r-1} n B_r = -(-1)^r B_r$$

When reflecting a homogeneous multivector M_r along n, blades containing n should be reflected (change sign), whereas blades that don't contain n should not. If $M_r = M_\perp + M_\parallel$, then following from the above two equations:

$$nM_r n = nM_\perp n + nM_\parallel n = (-1)^r (M_\perp - M_\parallel)$$

Therefore, to reflect a homogeneous multivector of grade r, the reflection is given by:

$$M'_r = (-1)^r n M_r n \tag{12}$$

which is a generalisation of eq. (11).

2.2.3 Rotations

A rotation can be formed by two reflections:

$$a' = mnanm = Ra\widetilde{R} \tag{13}$$

where $R = mn$ is defined as a *rotor*.

A more useful expression for R is found by considering the vectors n and m necessary to rotate an angle θ about a particular direction, defined by the bivector B.

This is achieved by setting:

$$m = \cos(\theta/2)n + \sin(\theta/2)nB$$

where B is the unit plane containing m and n such that $n \cdot B = nB$ is perpendicular to n. This also means $nB = -Bn$ such that $nBn = -B$.

Therefore, the rotor is given by:

$$\begin{aligned} R &= mn \\ &= \cos(\theta/2)n^2 + \sin(\theta/2)nBn \\ &= \cos(\theta/2) - \sin(\theta/2)B \end{aligned} \tag{14}$$

Since B is a unit plane ($B^2 = -1$), this can be expressed in exponential notation:

$$R = \exp(-B\theta/2) \tag{15}$$

2.3 Conformal geometric algebra (CGA)

2.3.1 Foundation of CGA

If euclidean space $\mathbb{E}^n$ is represented by a GA with canonical basis $\mathcal{G}(n, 0)$, conformal geometric algebra (CGA) uses a canonical basis of $\mathcal{G}(n+1, 1)$ to represent conformal space. This extends the vector space with two additional basis vectors e_+ and e_- where $e_+^2 = +1$, $e_-^2 = -1$.

Null vectors

A null vector X satisfies $X^2 = 0$. CGA defines two specific null vectors, that act as more useful basis vectors than e_+ and e_-:

$$n_0 = \frac{1}{2}(e_+ + e_-) \qquad n_\infty = (e_- - e_+) \tag{16}$$

These null vectors are anticommuting with other basis vectors, but don't commute or anticommute with themselves:

$$\begin{aligned} n_0 n_\infty &= \frac{1}{2}(e_+ + e_-)(e_- - e_+) \\ &= \frac{1}{2}(2e_+e_- - 2) \\ &= -1 + E \end{aligned}$$

where E is defined as the *Minkowski plane*.

$$\begin{aligned} n_0 \cdot n_\infty &= -1 \\ n_0 \wedge n_\infty &= e_+ \wedge e_- = E \end{aligned}$$

For a vector x that doesn't contain e_+ or e_- (ie: lies in euclidean space), $x \cdot n_0 = 0$ and $x \cdot n_\infty = 0$.

Conformal space

A point x in euclidean space maps to a vector in conformal space with the mapping:

$$X = \lambda(n_0 + x + \frac{1}{2}x^2 n_\infty) \tag{17}$$

where the free variable λ indicates that the scale of X is irrelevant.

The reason for this mapping is that it results in the following:

$$\begin{aligned} X \cdot Y &\propto (n_0 + x + \frac{1}{2}x^2 n_\infty) \cdot (n_0 + y + \frac{1}{2}y^2 n_\infty) \\ &\propto (-\frac{1}{2}(x^2 + y^2) + x \cdot y) \\ &\propto (x^2 + y^2 - xy - yx) \\ &\propto (x - y)^2 \end{aligned}$$

The inner product between two conformal vectors is proportional to the euclidean distance between the corresponding vectors in euclidean space.

One result of this is that X must be a null vector, since:

$$X^2 = -\frac{1}{2}|x - x|^2 = 0$$

2.3.2 Normalised conformal vectors and inverse mapping

A normalised conformal vector is given by:

$$X = n_0 + x + \frac{1}{2}x^2 n_\infty \tag{18}$$

For an unnormalised vector X', since $n_\infty \cdot n_0 = -1$, it is normalised with:

$$X = \frac{X'}{-n_\infty \cdot X'}$$

allowing the euclidean point x can be extracted with:

$$x = (X' \cdot e_1)e_1 + (X' \cdot e_2)e_2 + \cdots$$

When the conformal vectors are normalised, the inner product specifically gives:

$$X \cdot Y = -\frac{1}{2}(x - y)^2$$

2.3.3 Interpreting vectors in CGA

With the above mapping, a vector represents a point. However a general vector in conformal space represents a sphere. Specifically, a *dual sphere*, but this distinction will be explained later. For all results below, X is the normalised conformal point for position x using eq. (18).

A sphere Σ^* at position x and radius r is represented by:

$$\Sigma^* = \lambda(X - \frac{1}{2}r^2 n_\infty) \tag{19}$$

where a point y lies on this sphere if $Y \cdot \Sigma^* = 0$, which is only true if $|x - y|^2 = r^2$.

For a sphere at position $x = (r + \rho)n$, with radius r, this represents a plane Π^* with unit normal n at a distance ρ from the origin, when at the limit $r \to \infty$:

$$\begin{aligned} \Sigma^* &= \lambda(n_0 + n(r + \rho) + \frac{1}{2}(\rho^2 + 2r\rho)n_\infty) \\ \Pi^* &= \lim_{r\to\infty}(\Sigma/r) \\ &= \lambda(n + \rho n_\infty) \end{aligned} \tag{20}$$

using the fact that vectors can be freely scaled, while representing the same object.

Like the sphere, $Y \cdot \Pi^* = 0$ if y lies on the plane, since:

$$\begin{aligned} Y \cdot \Pi^* &\propto (n_0 + y + \frac{1}{2}x^2 n_\infty) \cdot (n + \rho n_\infty) \\ &= -\rho + y \cdot n \end{aligned}$$

which is equal to 0 if $y \cdot n = \rho$.

In summary, all vectors X represent spheres, from $r = 0$ (a point) to $r = \infty$ (a plane) and for a conformal vector Y, if $Y \cdot X = 0$, the point y lies on the point/sphere/plane.

2.3.4 Forming geometric primitives by intersection

A (dual) circle is formed by the intersection of two vectors Σ_1^* and Σ_2^*, representing spheres or planes:

$$C^* = \Sigma_1^* \wedge \Sigma_2^* \tag{21}$$

Using eq. (10), the inner product with a conformal vector Y is:

$$Y \cdot C^* = \Sigma_2^*(Y \cdot \Sigma_1^*) - \Sigma_1^*(Y \cdot \Sigma_2^*)$$

In other words, $Y \cdot C^* = 0$ only if $Y \cdot \Sigma_1^* = 0$ and $Y \cdot \Sigma_2^* = 0$, meaning y lies on both spheres and therefore lies on the intersection of the spheres.

Therefore a conformal bivector always represents a circle or a line (when both X_1 and X_2 are planes), representing a circle at infinity.

Note, that this line doesn't have to pass through the origin.

This usage of intersection extends further. A trivector represents the intersection of three spheres, which is a point pair.

2.3.5 Dual and direct geometric primitives

The geometric primitives given previously: spheres, planes, circles, lines and point-pairs, were given in their *dual* form.

For example, $C^* = X_1 \wedge X_2$ is a *dual circle*. The *direct circle* is related by $C = C^* I$ using the duality transform.

Direct geometric primitives can conveniently be found by the outer product of points that lie on the shape.

A direct sphere is a quadvector, given by:

$$\Sigma = X_1 \wedge X_2 \wedge X_3 \wedge X_4$$

for four points that lie on the sphere.

A direct plane is the same, but with one point at infinity, represented by n_∞:

$$\Pi = X_1 \wedge X_2 \wedge X_3 \wedge n_\infty$$

A direct circle is given by:

$$C = X_1 \wedge X_2 \wedge X_3$$

for three points that lie on the circle.

And a direct line:

$$L = X_1 \wedge X_2 \wedge n_\infty$$

Finally, a direct point pair is:

$$P = X_1 \wedge X_2$$

Direct geometric primitives are denoted Σ, Π, etc, while the dual forms are denoted Σ^*, Π^*, etc.

2.3.6 Intersecting direct geometric primitives with the meet operator

By using eq. (8) to use the dual to swap the inner and outer product, it can be shown that for direct geometric primitives X and Y, the intersection is given by:

$$X \vee Y = (X^* \wedge Y^*)^* \tag{22}$$

where $\vee$ is called the meet operator.

2.3.7 Interpreting intersection results

The intersection result of interest in this project is the point pair, so only the method of interpreting this will be given.[8]

Firstly, for the dual point pair T^*, the dual midplane Π^* is found by:

$$\Pi^* = T^* \cdot n_\infty$$

Secondly, a projection operator (a rotor) is created:

$$P = 1 + \frac{T^*}{\sqrt{(T^*)^2}}$$

Finally, the two points of the point pair are extracted by applying the transformation P or it's inverse (the reverse):

$$X^{(1)} = P\Pi\widetilde{P} \qquad X^{(2)} = \widetilde{P}\Pi P$$

3 Kinematics

This section will outline how to solve the kinematics of the delta and serial robot, first using conventional methods[9], then CGA.

An overview of the representation of rigid body transformations using both methods will also be given, necessary for the kinematics of serial robots.

Finally, the theory required for moving robots through a trajectory is given.

3.1 Delta robot

Figure 3 shows the structure of the delta robot and notation used for the various points, lengths and angles.

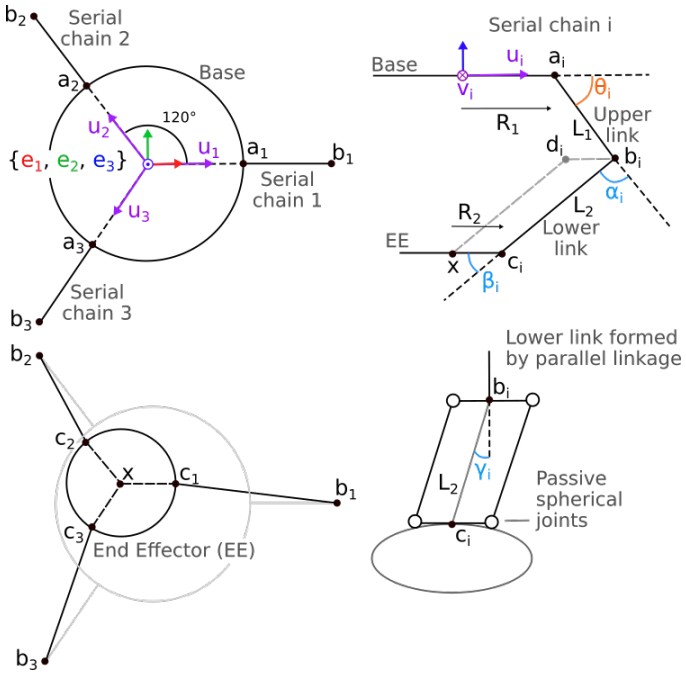

Figure 3: Structure of the delta robot.

The forward kinematics solution is the intersection of three spheres.

The inverse kinematics solution can be found for each joint individually, and involves finding the joint angle θ_i that gives a pseudo-elbow position a distance L_2 from the end effector position.

3.1.1 Forward kinematics with conventional methods

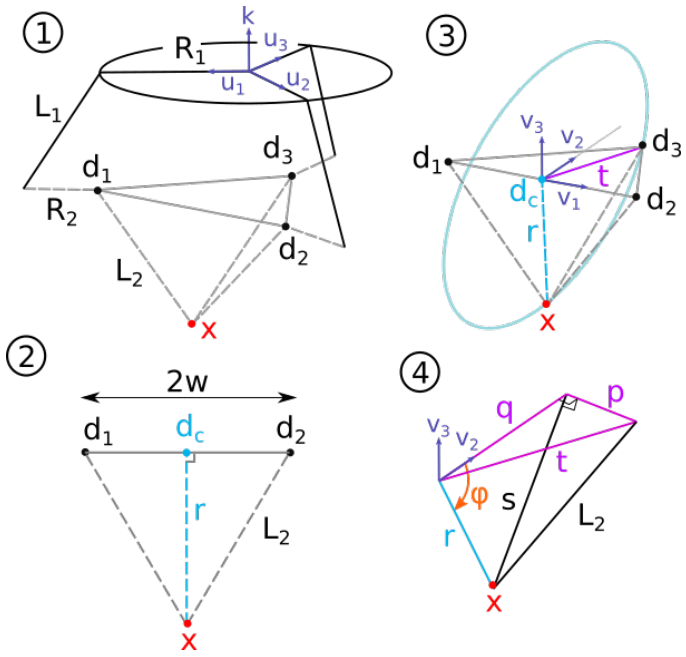

Figure 4: Notation used for solving the forward kinematics of the delta robot conventionally.

Using the notation outlined in Figure 4, the points $\boldsymbol{d}_i$ for each joint i are evaluated:

$$\boldsymbol{d}_i = (R_1 - R_2 + L_1\cos(\theta_i))\boldsymbol{u}_i - L_1\sin(\theta_i)\boldsymbol{k} \quad (23)$$

The objective is then to solve for $\boldsymbol{x}$ such that:

$$|\boldsymbol{x} - \boldsymbol{d}_i| = L_2 \quad \forall i$$

This is equivalent to intersecting three spheres of length L_2 about the three points $\boldsymbol{d}_i$. The first step is to find the circle formed by the intersection of the first two points:

$$\boldsymbol{d_c} = \frac{1}{2}(\boldsymbol{d}_1 + \boldsymbol{d}_2)$$
$$\boldsymbol{v}_1 = \frac{\boldsymbol{d}_2 - \boldsymbol{d}_1}{|\boldsymbol{d}_2 - \boldsymbol{d}_1|}$$
$$\boldsymbol{v}_3 = \frac{\boldsymbol{v}_1 \times (\boldsymbol{d}_3 - \boldsymbol{d}_1)}{|\boldsymbol{n}_1 \times (\boldsymbol{d}_3 - \boldsymbol{d}_1)|}$$
$$\boldsymbol{v}_2 = \boldsymbol{v}_3 \times \boldsymbol{v}_1$$

$$w = \frac{1}{2}\left|\boldsymbol{d}_2 - \boldsymbol{d}_1\right|$$
$$r = \sqrt{L_2^2 - w^2}$$

With this, the solution is solved by finding the angle ϕ such that $L_2^2 = \left|\boldsymbol{x} - \boldsymbol{d}_3\right|^2$, with:

$$\boldsymbol{x} = \boldsymbol{d}_c + r(\boldsymbol{v}_2\cos(\phi) - \boldsymbol{v}_3\sin(\phi))$$
$$\boldsymbol{d}_3 = \boldsymbol{d}_c + p\boldsymbol{v}_1 + q\boldsymbol{v}_2$$

Letting $p = (\boldsymbol{d}_3 - \boldsymbol{d}_c)\cdot\boldsymbol{v}_1$, $q = (\boldsymbol{d}_3 - \boldsymbol{d}_c)\cdot\boldsymbol{v}_2$ and $t^2 = p^2 + q^2 = \left|\boldsymbol{d}_3 - \boldsymbol{d}_c\right|^2$, the final solution for phi comes out as:

$$\alpha = \cos^{-1}\left(\frac{t^2 - w^2}{2qr}\right)$$

Which gives two possible solutions for $\boldsymbol{x}$:

$$\boldsymbol{x} = \boldsymbol{d}_c + r(\boldsymbol{v}_2\cos\alpha \pm \boldsymbol{v}_3\sin\alpha)$$

There are two valid configurations for a given set of joint angles. Either the end effector is above the base or below.

3.1.2 Inverse kinematics with conventional methods

For inverse kinematics, each angle θ_i is solved independently to satisfy:

$$\left|\boldsymbol{d}_i - \boldsymbol{x}\right|^2 = L_2^2$$

Using the same definition of $\boldsymbol{d}_i$ as in eq. (23), this relationship can be re-arranged into the form:

$$P\cos\theta_i + Q\sin\theta_i = k$$

with:

$$P = 2L_1(R_1 - R_2 - \boldsymbol{x}^T\boldsymbol{u}_i)$$
$$Q = 2L_1(\boldsymbol{x}^T\boldsymbol{k})$$
$$k = L_2^2 - L_1^2 - (R_1 - R_2)^2 + 2(R_1 - R_2)\boldsymbol{x}^T\boldsymbol{u}_i - \left|\boldsymbol{x}\right|^2$$

With $R^2 = P^2 + Q^2$ and $\phi_i = \tan^{-1}(Q/R)$, this gives the solution:

$$\theta_i = \phi_i \pm \cos^{-1}\left(\frac{k}{R}\right)$$

or, more conveniently:

$$\psi_i = \cos^{-1}(k/R)$$
$$\theta_i = \phi_i \pm \psi_i$$

Each joint has two solutions, either side of ϕ_i, corresponding to an inverted solution for $\theta_i = \phi_i - \psi_i$ and a non-inverted solution for $\theta_i = \phi_i + \psi_i$.

3.1.3 Forward kinematics with CGA

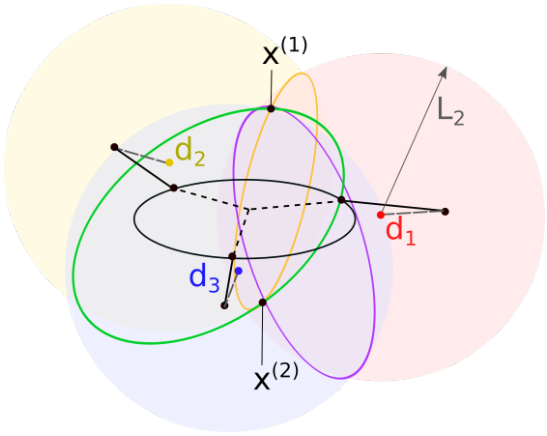

Figure 5: Notation used for solving the forward kinematics of the delta robot with CGA. The two possible solutions $x^{(1)}$ and $x^{(2)}$ correspond to the point pair formed by the intersection of three spheres about the points d_i.

Following the notation given in Figure 5, dual spheres can be defined using eq. (19). These are then intersected using the outer product to give the intersection point pair T:

$$d_i = (R_1 - R_2 + L_1\cos(\theta_i))u_i - L_1\sin(\theta_i)e_3$$
$$D_i = n_0 + d_i + \frac{1}{2}d_i^2 n_\infty$$
$$\Sigma_i^* = D_i - \frac{1}{2}L_2^2 n_\infty$$
$$T = I_5(\Sigma_1^* \wedge \Sigma_2^* \wedge \Sigma_3^*)$$

The two solutions $X^{(1)}$ and $X^{(2)}$ are found with Section 2.3.7, then normalised allowing the euclidean positions $x^{(1)}$ and $x^{(2)}$ to be extracted.

The delta robot used in this project has the end effector below the base, so the solution with the smaller z component $x \cdot e_3$ is selected.

3.1.4 Inverse kinematics with CGA

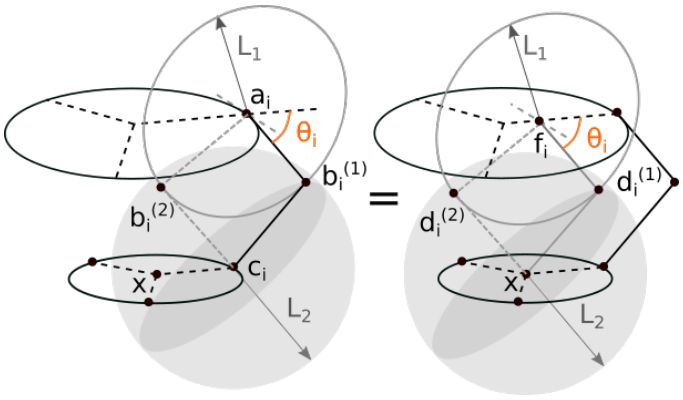

Figure 6: Notation used for solving the inverse kinematics of the delta robot using CGA. For each joint θ_i, there are two possible solutions, corresponding to two possible locations of the pseudo-elbow positions, $d_i^{(1)}$ or $d_i^{(2)}$.

Inverse kinematics is simpler, as each chain can be solved independently. This requires finding the displacement $b_i - a_i$, from which θ_i can be extracted with simple trigonometry. This is equivalent to finding the displacement $d_i - e_i$ as shown in Figure 6.

A dual sphere Σ_x^* is defined about the end effector the same way as before and intersected with dual circles C_i^* about each point f_i. The circles are formed using eq. (21) by intersecting the appropriate dual sphere and plane:

$$
\begin{aligned}
X &= n_0 + x + \frac{1}{2}x^2 n_\infty \\
\Sigma_x^* &= X_i - \frac{1}{2}L_2^2 n_\infty \\
f_i &= (R_1 - R_2)u_i \\
F_i &= n_0 + f_i + \frac{1}{2}f_i^2 n_\infty \\
C_i^* &= u_i \wedge e_3 \wedge (F_i - \frac{1}{2}L_1^2 n_\infty) \\
T_i &= I_5(\Sigma_x^* \wedge C_i^*)
\end{aligned}
$$

The two solutions, $d_i^{(1)}$ and $d_i^{(2)}$ are extracted. The solution with a larger component in u_i, $u_i \cdot d_i$ is selected and allows θ_i to be calculated:

$$\theta_i = \sin^{-1}\left(\frac{(d_i - f_i)\cdot(-e_3)}{L_1}\right)$$

3.1.5 Jacobian

The Jacobian is found the same way for conventional and CGA methods. One method of finding the Jacobian is to differentiate the inverse kinematic solution, but a simpler method is to consider velocity constraints.

The velocity at each pseudo elbow is $\boldsymbol{v}_i = \theta_i \hat{\boldsymbol{v}}_i$, with:

$$\hat{\boldsymbol{v}}_i = -L_1(\cos\theta_i \boldsymbol{k} + \sin\theta_i \boldsymbol{u}_i)$$

The direction of the lower link is given by the displacement from the psuedo-elbow to end effector with:

$$\boldsymbol{n}_i = \frac{\boldsymbol{x} - \boldsymbol{d}_i}{|\boldsymbol{x} - \boldsymbol{d}_i|}$$

The end effector velocity, $\dot{\boldsymbol{x}}$ must satisfy:

$$\dot{\boldsymbol{x}} \cdot \boldsymbol{n}_i = \boldsymbol{v}_i \cdot \boldsymbol{n}_i$$
$$\boldsymbol{n}_i^T \dot{\boldsymbol{x}} = \boldsymbol{n}_i^T \hat{\boldsymbol{v}}_i \theta_i$$

or more generally, in terms of small changes:

$$\boldsymbol{n}_i^T \boldsymbol{\Delta x} = \boldsymbol{n}_i^T \hat{\boldsymbol{v}}_i \Delta\theta_i$$

Combining all joints, this gives:

$$J_x \boldsymbol{\Delta x} = J_\theta \boldsymbol{\Delta\theta}$$

with:

$$J_x = \begin{bmatrix} \leftarrow \boldsymbol{n}_1^T \rightarrow \\ \leftarrow \boldsymbol{n}_2^T \rightarrow \\ \leftarrow \boldsymbol{n}_3^T \rightarrow \end{bmatrix} \qquad J_\theta = \begin{bmatrix} \boldsymbol{n}_1^T \hat{\boldsymbol{v}}_1 & 0 & 0 \\ 0 & \boldsymbol{n}_2^T \hat{\boldsymbol{v}}_2 & 0 \\ 0 & 0 & \boldsymbol{n}_3^T \hat{\boldsymbol{v}}_3 \end{bmatrix}$$

such that the forward and inverse Jacobian have the form:

$$J = J_x^{-1} J_\theta \qquad J^{-1} = J_\theta^{-1} J_x$$

Singularities occur when J is non-invertible. This occurs when $|J_\theta| = 0$, when any $\boldsymbol{n}_i^T \hat{\boldsymbol{v}}_i = 0$. This corresponds to the case when the upper and lower links are in the same plane perpendicular to the pseudo-elbow position, meaning that the chain is fully extended and has reduced mobility.

3.1.6 Dependent joints

To fully define the robot state, the dependent joints α_i, β_i and γ_i must be supplied. This will allow for visualisation of the robot model.

For both conventional and CGA methods, once the points $\boldsymbol{x}$ and $\boldsymbol{d}_1, \boldsymbol{d}_2, \boldsymbol{d}_3$ have been found, the dependent joint positions are found with:

$$\begin{aligned}\beta_i &= \text{atan2}((\boldsymbol{x}-\boldsymbol{d}_i)\cdot(-\boldsymbol{k}),\\ &\qquad (\boldsymbol{x}-\boldsymbol{d}_i)\cdot(-\boldsymbol{u}_i))\\ \alpha_i &= \pi-\theta_i-\beta_i\\ \gamma_i &= \sin^{-1}\left(\frac{(\boldsymbol{x}-\boldsymbol{d}_i)\cdot\hat{\boldsymbol{v}}_i}{L_2}\right)\end{aligned}$$

3.2 Representing rigid body transformations conventionally

3.2.1 Rotations

The rotation matrix R defining a rotation about an axis $\hat{\boldsymbol{\omega}}$ by an angle θ is equal to:

$$R = I + \sin(\theta)S(\hat{\boldsymbol{\omega}}) + (\cos(\theta)-1)S(\hat{\boldsymbol{\omega}})^2 \quad (24)$$

where the skew-symmetric matrix $S(\hat{\boldsymbol{\omega}})$ gives the cross product:

$$\hat{\boldsymbol{\omega}}\times\boldsymbol{x} = S(\hat{\boldsymbol{\omega}})\boldsymbol{x} = \begin{bmatrix}0 & -\omega_3 & \omega_2\\ \omega_3 & 0 & -\omega_1\\ -\omega_2 & \omega_1 & 0\end{bmatrix}\boldsymbol{x} \quad (25)$$

Matrix exponential parameterisation

This has the matrix exponential parameterisation:

$$R = \exp(S(\hat{\boldsymbol{\omega}})\theta) = \exp(S(\hat{\omega}\theta)) \quad (26)$$

where the vector $\hat{\boldsymbol{\omega}}\theta$ is called the exponential coordinates.

For a constant rotation axis $\hat{\boldsymbol{\omega}}$:

$$\frac{dR}{d\theta} = S(\hat{\boldsymbol{\omega}})R = RS(\hat{\boldsymbol{\omega}}) \quad (27)$$

And:

$$\begin{aligned}R(\theta+\delta\theta) &= R(\theta) + R(\theta)S(\hat{\boldsymbol{\omega}})\delta\theta + \mathcal{O}(\delta\theta^2)\\ &= R(\theta)(I + S(\hat{\boldsymbol{\omega}})\delta\theta)\\ &= R(\theta)\delta R\\ \text{or } &= R(\theta) + S(\hat{\boldsymbol{\omega}})R(\theta)\delta\theta\\ &= \delta R R(\theta)\end{aligned}$$

for an incremental rotation of $\delta\theta$:

$$\delta R = I + S(\hat{\boldsymbol{\omega}})\delta\theta + \mathcal{O}(\delta\theta^2) \quad (28)$$

Angular velocity

If parametrising R with respect to time, for a time-varying rotation axis $\hat{\boldsymbol{\omega}}(t)$, there are two ways to parameterise this:

$$R(t+\delta t) = R(t)\delta R \quad \text{or} \quad \delta R R(t)$$

For the first case, with $\delta\theta = \dot{\theta}(t)\delta t$:

$$\begin{aligned}R(t+\delta t) &= R(t)\delta R\\ &= R(t)(I + S(\hat{\boldsymbol{\omega}}(t))\dot{\theta}(t)\delta t + \mathcal{O}(\delta t^2))\\ \dot{R} &= \lim_{\delta t\to 0}\left(\frac{R(t+\delta t)-R(t)}{\delta t}\right)\\ &= R(t)S(\hat{\boldsymbol{\omega}}(t))\dot{\theta}(t)\\ &= R(t)S(\boldsymbol{\omega}^{(1)}(t))\end{aligned}$$

Similarly, for the second case:

$$\dot{R} = S(\boldsymbol{\omega}^{(2)}(t))R(t)$$

In both cases, the angular velocity $\boldsymbol{\omega}(t) = \hat{\boldsymbol{\omega}}(t)\dot{\theta}(t)$, about an instantaneous rotation axis $\hat{\boldsymbol{\omega}}(t)$. However, these angular velocities correspond to different reference frames.

The *first* definition, $\boldsymbol{\omega} = \boldsymbol{\omega}^{(1)}$ is more useful so will be used, giving:

$$\dot{R}(t) = R(t)S(\boldsymbol{\omega}(t)) \quad (29)$$

For a fixed point $\boldsymbol{x}$ transformed to $\boldsymbol{x}' = R\boldsymbol{x}$, the

velocity in the moving frame is equal to:

$$\begin{aligned}\boldsymbol{v} &= R^T \boldsymbol{v}' \\ &= R^T \frac{d}{dt}(R\boldsymbol{x}) \\ &= R^T R S(\boldsymbol{\omega})\boldsymbol{x} \\ &= S(\boldsymbol{\omega})\boldsymbol{x} \\ &= \boldsymbol{\omega} \times \boldsymbol{x}\end{aligned}$$

as expected.

3.2.2 Rigid body transformations

A rigid body transformation T has the structure:

$$T = \begin{bmatrix} R & \boldsymbol{p} \\ \boldsymbol{0}^T & 1 \end{bmatrix} \tag{30}$$

which gives a transformed vector:

$$\begin{bmatrix} \boldsymbol{x}' \\ 1 \end{bmatrix} = T \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$
$$\boldsymbol{x}' = \boldsymbol{p} + R\boldsymbol{x}$$

The transformed vector is rotated by R then translated by $\boldsymbol{p}$.

Screw transformations

For a rotation, rotating about an axis gave the exponential parameterisation. Similarly for a rigid body transformation, a screw transformation leads to the exponential parameterisation. A screw transform consists of a rotation about a line and a translation along that line, as shown in Figure 7.

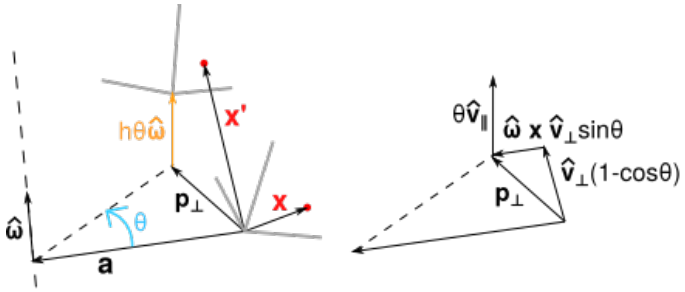

Figure 7: A screw transformation.

with a screw transformation, the transformed vector $\boldsymbol{x}'$ is:

$$\begin{aligned}\boldsymbol{x}' &= -\boldsymbol{a} + h\theta\hat{\boldsymbol{\omega}} + R(\boldsymbol{x} - \boldsymbol{a}) \\ &= R\boldsymbol{x} + (R - I)\boldsymbol{a} + h\theta\hat{\boldsymbol{\omega}} \\ &= R\boldsymbol{x} + \boldsymbol{p}\end{aligned}$$

R is the rotation of the twist transformation, but $\boldsymbol{p}$ is formed by the translation caused by this rotation and the translation along the twist axis.

$$\begin{aligned}\boldsymbol{p} &= \theta\hat{\boldsymbol{v}}_{\parallel} + \hat{\boldsymbol{v}}_{\perp}(1 - \cos\theta) + \hat{\boldsymbol{\omega}} \times \hat{\boldsymbol{v}}_{\perp} \sin\theta \\ &= \theta\hat{\boldsymbol{v}}_{\parallel} + \hat{\boldsymbol{v}}_{\perp}(1 - \cos\theta) + S(\hat{\boldsymbol{\omega}})\hat{\boldsymbol{v}}_{\perp} \sin\theta \end{aligned} \tag{31}$$

Matrix exponential parameterisation

Analogous to $R = \exp(S(\hat{\boldsymbol{\omega}})\theta)$ for rotations, the rigid body transformation T has an exponential parameterisation.

$$T = \exp(\xi(\hat{\boldsymbol{u}})\theta) = \exp(\xi(\hat{\boldsymbol{u}}\theta)) \tag{32}$$

with:

$$\hat{\boldsymbol{u}} = \begin{bmatrix} \hat{\boldsymbol{\omega}} \\ \hat{\boldsymbol{v}} \end{bmatrix} \quad \xi(\hat{\boldsymbol{u}}) = \begin{bmatrix} S(\hat{\boldsymbol{\omega}}) & \hat{\boldsymbol{v}} \\ \boldsymbol{0}^T & 0 \end{bmatrix} \tag{33}$$

This can be shown to give the correct transformation for the screw transformation defined by θ, $\hat{\boldsymbol{\omega}}$ and $\hat{\boldsymbol{v}}$.

The vector $\hat{\boldsymbol{u}}\theta$ is the exponential coordinates of the rigid body transformation.

Additionally, the matrix ξ is called the twist and the vector $\hat{\boldsymbol{u}}$ alone is called the twist coordinates.

For constant twist coordinates $\hat{\boldsymbol{u}}$:

$$\frac{dT}{d\theta} = \xi(\boldsymbol{u})T = T\xi(\hat{\boldsymbol{u}}) \tag{34}$$

And:

$$T(\theta + \delta\theta) = T(\theta)\delta T \quad \text{or} \quad \delta T T(\theta)$$

for an incremental transform by $\delta\theta$:

$$\delta T = I + \xi(\hat{\boldsymbol{u}})\delta\theta + \mathcal{O}(\delta\theta^2) \tag{35}$$

Spatial velocity

Like rotation, T can be parameterised by time to give:

$$\dot{T} = T\xi(\boldsymbol{u}^{(1)}) \text{ or } \xi(\boldsymbol{u}^{(2)})T$$

for the spatial velocity $\boldsymbol{u} = \hat{\boldsymbol{u}}\dot{\theta}$. Again, the first definition, $\boldsymbol{u} = \boldsymbol{u}^{(1)}$ is more useful, so will be used, giving:

$$\dot{T}(t) = T(t)\xi(\boldsymbol{u}) \tag{36}$$

The spatial velocity contains both angular and linear velocity:

$$\boldsymbol{u} = \begin{bmatrix} \hat{\boldsymbol{\theta}}\dot{\theta} \\ \boldsymbol{u}\dot{\theta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{u} \end{bmatrix} \tag{37}$$

With this, the velocity of a point P is:

$$\begin{aligned} \begin{bmatrix} \boldsymbol{v}_P \\ 0 \end{bmatrix} &= T^{-1} \begin{bmatrix} \boldsymbol{v}'_P \\ 0 \end{bmatrix} \\ &= T^{-1} \frac{d}{dt} \left(T \begin{bmatrix} \boldsymbol{x}_P \\ 1 \end{bmatrix} \right) \\ &= T^{-1} T \xi(\boldsymbol{u}) \begin{bmatrix} \boldsymbol{x}_P \\ 1 \end{bmatrix} \\ &= \xi(\boldsymbol{u}) \begin{bmatrix} \boldsymbol{x}_P \\ 1 \end{bmatrix} \\ \boldsymbol{v}_P &= \boldsymbol{v} + \boldsymbol{\omega} \times \boldsymbol{x}_P \end{aligned}$$

as expected.

3.3 Serial robot kinematics using conventional methods

3.3.1 Forward kinematics

An N degree of freedom serial robot consist of a fixed base and N links. Reference frame 0 is attached to the fixed base, and reference frames $1, 2, \ldots, N$ fixed to each link.

The pose of frame j, within frame i is defined by the rigid transformation ${}^{j}T_i$ such that a fixed point ${}^{j}\boldsymbol{x}$ in frame j transforms to the point ${}^{i}\boldsymbol{x}$ in frame i with:

$$\begin{bmatrix} {}^{i}\boldsymbol{x} \\ 1 \end{bmatrix} = {}^{i}T_j \begin{bmatrix} {}^{j}\boldsymbol{x} \\ 1 \end{bmatrix}$$

as shown in Figure 8.

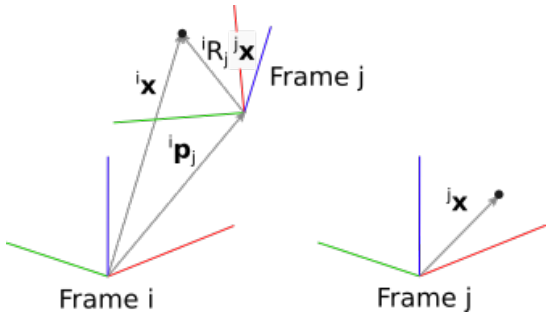

Figure 8: Notation for reference frames and the transformation of the position of a point between reference frames.

For a n-link serial chain, the pose of the end effector is given by concatenating each transform between links:

$$T = {}^{0}T_n = {}^{0}T_1 {}^{1}T_2 \cdots {}^{n-1}T_n \tag{38}$$

Joints and DH parameters

Joints between links parameterise the rigid body transformations ${}^{i-1}T_i$. Simple joints have one degree of freedom, denoted q_i, corresponding to an angle (revolute joints) or a distance (prismatic joint). Compound joints, which have more than one degree of freedom, can be expressed using multiple simple joints.

A common parameterisation, which is used in this project, are the Denavit-Hartenberg parameters (DH parameters), shown in Figure 9, which can represent any revolute joint.

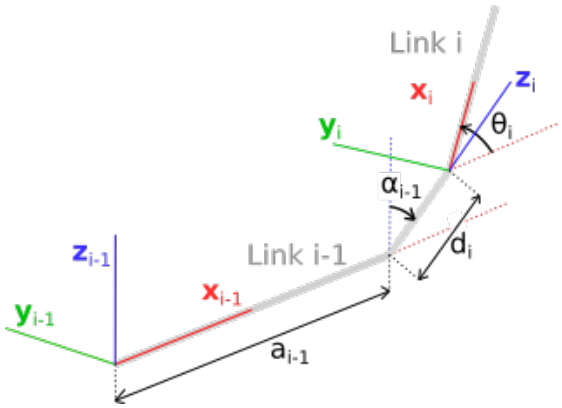

Figure 9: DH parameters convention for defining the transformation between two links that are connected with a revolute joint.

For joint i, parametrising the transformation from link $i-1$ to link i, the transformation consists of the following steps:

1. Rotate by θ_i about the z-axis.
2. Translate by d_i along the z-axis.
3. Rotate by α_{i-1} about the x-axis.
4. Translate by a_{i-1} along the x-axis.

All parameters are constant, defined by the robot geometry, except for θ_i, set equal to q_i, which is variable.

If the DH parameters of the robot and the joint positions $\{q_i\}$ are known, the final end effector pose given in eq. (38) can be found. From this, the position and orientation can be read, performing forward kinematics.

3.3.2 Spatial velocity and the Jacobian

For a fixed point ${}^i\boldsymbol{x}_A$ in frame i, the spatial velocity ${}^i\boldsymbol{u}$ of the reference frame gives the velocity experienced by this point as:

$$\begin{bmatrix} {}^i\boldsymbol{v}_A \\ 0 \end{bmatrix} = \xi({}^i\boldsymbol{u}) \begin{bmatrix} {}^i\boldsymbol{x}_A \\ 1 \end{bmatrix}$$
$${}^i\boldsymbol{v}_A = {}^i\boldsymbol{v} + {}^i\boldsymbol{\omega} \times {}^i\boldsymbol{x}_A$$

as seen earlier, for spatial velocity defined by eq. (37), satisfying eq. (36).

For joint i, the spatial velocity of link i is equal to ${}^i\boldsymbol{u}_i = {}^i\hat{\boldsymbol{u}}_i\dot{q}_i$, for constant twist coordinates:

$${}^i\hat{\boldsymbol{u}}_i = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

which has a simple structure due to the use of DH parameters.

Spatial velocity of the end effector

The goal is to find the spatial velocity of the end effector frame, denoted ${}^n\boldsymbol{u}$, where the derivative of the end effector transform satisfies:

$$\frac{d}{dt}({}^0T_n) = {}^0T_n\xi({}^n\boldsymbol{u})$$

Evaluating the derivative with the product rule and eq. (36) gives:

$$\begin{aligned}
\frac{d}{dt}({}^0T_n) &= \frac{d}{dt}\left({}^0T_1 \cdots {}^{n-1}T_n\right) \\
&= \sum_{i=1}^{n} \left({}^0T_1 \cdots {}^{i-1}T_i\xi({}^i\hat{\boldsymbol{u}}_i) \cdots {}^{n-1}\right)\dot{q}_i \\
&= {}^0T_n \left[\sum_{i=1}^{n} {}^iT_n^{-1}\xi({}^i\hat{\boldsymbol{u}}_i){}^iT_n\dot{q}_i\right] \\
&= {}^0T_n \left[\sum_{i=1}^{n} \xi({}^n\hat{\boldsymbol{u}}_i\dot{q}_i)\right] \\
&= {}^0T_n\xi\left(\sum_{i=1}^{n} {}^n\boldsymbol{u}_i\dot{q}_i\right)
\end{aligned}$$

which defines the end effector spatial velocity as:

$${}^n\boldsymbol{u} = \sum_{i=1}^{n} {}^n\hat{\boldsymbol{u}}_i\dot{q}_i = J\dot{\boldsymbol{q}}$$

for $\dot{\boldsymbol{q}} = \begin{bmatrix} \dot{q}_1 & \cdots & \dot{q}_n \end{bmatrix}^T$ and:

$$J = \begin{bmatrix} \uparrow & & \uparrow \\ {}^n\hat{\boldsymbol{u}}_1 & \cdots & {}^n\hat{\boldsymbol{u}}_n \\ \downarrow & & \downarrow \end{bmatrix}$$

Intuitively, this is summing the contribution to spatial velocity from each joint, after transforming the spatial velocity to the end effector frame.

Transforming spatial velocity

The vector ${}^n\boldsymbol{u}_i$ represent the contribution to the spatial velocity of the end effector from joint i. To find this, the spatial velocity of link i from joint i (${}^i\boldsymbol{u}_i$) is transformed to the end effector.

When transforming from i to j:

$$\xi({}^j\boldsymbol{u}) = {}^iT_j^{-1}\xi({}^i\boldsymbol{u}){}^iT_j$$

which corresponds to:

$${}^j\boldsymbol{u} = {}^jX_i{}^i\boldsymbol{u}$$

$${}^jX_i = \begin{bmatrix} {}^jR_i^T & 0 \\ -{}^jR_i^T S({}^j\boldsymbol{p}_i) & {}^jR_i^T \end{bmatrix}$$

for a spatial velocity transform matrix ${}^jX_i$.

This relates the linear and angular velocity between the two links as:

$$
\begin{aligned}
{}^{j}\boldsymbol{\omega} &= {}^{j}R_i^T {}^{i}\boldsymbol{\omega} \\
{}^{j}\boldsymbol{v} &= {}^{j}R_i^T({}^{i}\boldsymbol{v} + {}^{i}\boldsymbol{\omega} \times {}^{j}\boldsymbol{p}_i)
\end{aligned}
$$

This shows that the spatial velocity of frame j is formed by adding $\boldsymbol{p} \times {}^{i}\boldsymbol{\omega}$ to the linear velocity, then rotating both the linear and angular velocity by ${}^{j}R_i^T$ to refer to link j.

Finally, to transform a spatial velocity to the end effector:

$$
{}^{n}\hat{\boldsymbol{u}}_i = {}^{n}X_{n-1} \cdots {}^{i+1}X_i \, {}^{i}\hat{\boldsymbol{u}}_i
$$

3.3.3 Inverse kinematics

The Jacobian can be used to define incremental changes to the end effector transform with:

$$
\delta T \approx I + \xi(\boldsymbol{\delta\xi}) = I + \xi(J\boldsymbol{\delta q})
$$

where $\delta\xi$ is the change in exponential coordinates $\hat{\boldsymbol{u}}\delta\theta$.

This is useful for inverse kinematics, where:

$$
T(\boldsymbol{q} + \boldsymbol{\delta q}) \approx T(\boldsymbol{q})\delta T = T(I + \xi(\boldsymbol{\delta q})) \tag{39}
$$

With this, Newton's method can be used to numerically solve the inverse kinematics.

Denote the current joint positions $\boldsymbol{q}$, with corresponding pose $T(\boldsymbol{q})$. The target pose is $T_* = T(\boldsymbol{q}_*)$ and the goal is to find $\boldsymbol{q}_*$.

Let $\boldsymbol{q}_* = \boldsymbol{q} + \boldsymbol{\delta q}$, and by using eq. (39):

$$
\begin{aligned}
T_* &= T(\boldsymbol{q} + \boldsymbol{\delta q}) \\
&\approx T(\boldsymbol{q})\delta T \\
&= T(\boldsymbol{q})(I + \xi(J\boldsymbol{\delta q}))
\end{aligned}
$$

This corresponds to solving the least squares problem:

$$
\boldsymbol{\delta q} = J(\boldsymbol{q})^{+} \boldsymbol{\xi}^{-1} \left[T^{-1}(\boldsymbol{q}) T_* - I \right]
$$

where $J(\boldsymbol{q})^{+}$ is the pseudoinverse of the Jacobian and $\boldsymbol{\xi}^{-1}[\cdot]$ is the operation of solving $\xi(\boldsymbol{\xi}) = (\cdot)$ which simply involves reading the appropriate matrix elements.

The pseudoinverse can be found using the SVD, allowing for removing singular values that are below a certain value. Denoting the solution to this as $\boldsymbol{\delta q}(\boldsymbol{q})$, the final solution is reached by iteratively applying:

$$
\boldsymbol{q}_{k+1} = \boldsymbol{q}_k + \boldsymbol{\delta q}(\boldsymbol{q}_k)
$$

until the norm of the change in twist coordinates, $|J\boldsymbol{\delta q}|$ falls below a particular threshold.

3.4 Representing rigid body transformations using CGA

3.4.1 Rotations

Starting with GA in three dimensions (not extending to CGA yet), a rotation is defined by a rotor R, such that a rotated vector is given using eq. (13), $x' = Rx\widetilde{R}$.

Matrix exponential parameterisation

As seen in eq. (14) and eq. (15), the rotor has an exponential parameterisation for rotating about an axis:

$$
R = \exp(-B\theta/2) = \cos(\theta/2) - B\sin(\theta/2)
$$

For a constant B:

$$
\frac{dR}{d\theta} = -\frac{1}{2}BR \quad \text{or} \quad -\frac{1}{2}RB \tag{40}
$$

For an incremental rotation:

$$
\delta R = 1 - \frac{1}{2}B\delta\theta + \mathcal{O}(\delta\theta^2) \tag{41}
$$

Angular velocity

If parametrising a rotor as $R(t+\delta t) = R(t)\delta R$, for time varying rotation axis B, where:

$$
\delta R = 1 - \frac{1}{2}B(t)\dot{\theta}(t)\delta t + \mathcal{O}(\delta t^2)
$$

then:

$$
\dot{R} = -\frac{1}{2}R\Omega \tag{42}
$$

where $\Omega(t) = B(t)\dot{\theta}(t)$ is a bivector representing the angular velocity about the instantaneous rotation axis $B(t)$.

With this a fixed point x experiences velocity:

$$\begin{aligned} v &= \widetilde{R} v' R \\ &= \widetilde{R} \frac{d}{dt}\left(Rx\widetilde{R}\right) R \\ &= -\frac{1}{2}\widetilde{R}\left(R\Omega x\widetilde{R} + Rx\widetilde{\Omega}\widetilde{R}\right) R \\ &= \frac{1}{2}(x\Omega - \Omega x) \\ &= x \cdot \Omega \end{aligned}$$

which is equivalent to $\boldsymbol{v} = \boldsymbol{\omega} \times \boldsymbol{x}$.

3.4.2 Rigid body transformations

Similar to how homogeneous matrices are conventionally used for rigid body transformations, requiring an additional component to allow translations, GA allows for translations and rigid body transformations by using CGA.

Rotations in CGA

Rotors apply to conformal points in the same way:

$$\begin{aligned} X' &= RX\widetilde{R} \\ &= n_0 R\widetilde{R} + Rx\widetilde{R} + \frac{1}{2}x^2 n_\infty R\widetilde{R} \\ &= n_0 + x' + \frac{1}{2}(x')^2 \end{aligned}$$

using $x'^2 = Rx^2\widetilde{R} = x^2$, so rotation doesn't affect the norm as expected.

Versors, translations and rigid body transformations

Rotors are not the only operators that are applied using $Rx\widetilde{R}$. A *versor* V is any object applied in this way, all formed by an even number of reflections, using eq. (12), such that a transformation is always applied with:

$$x' = Vx\widetilde{V} \tag{43}$$

without a sign dependent on the grade of x.

A rotor is a type of versor, containing a scalar and bivector. However, a multivector with scalar, bivector and quadvector is also a versor, but not a rotor.

Rotors which contain null vectors component will no longer perform rotations of conformal vectors. One example is the translation rotor:

$$R_t = 1 + \frac{1}{2}n_\infty p = \exp\left(\frac{1}{2}n_\infty p\right) \tag{44}$$

Using this, a rigid body transformation versor, denoted T, is defined by combining rotation and translation rotors:

$$T = (1 + \frac{1}{2}n_\infty p)(\cos(\theta/2) - B\sin(\theta/2)) \tag{45}$$

Screw transformations

A rigid body transformation versor has an exponential parametrisation, analogous to the conventional case, where:

$$T = \exp(-\frac{1}{2}\hat{U}\theta) \tag{46}$$

where:

$$\hat{U} = B + \hat{V} \qquad \hat{V} = \hat{v}n_\infty \tag{47}$$

$\hat{U}$ is a bivector representing a twist coordinates, where B contains the angular component an V the linear component.

The incremental transform is:

$$\delta T = 1 - \frac{1}{2}\hat{U}\delta\theta + \mathcal{O}(\delta\theta^2) \tag{48}$$

Spatial velocity

Using eq. (48) with $\delta\theta = \dot{\theta}(t)\delta t$, the derivative of T with respect to time is:

$$\dot{T} = -\frac{1}{2}T(t)U \tag{49}$$

for spatial velocity bivector $U(t) = \hat{U}(t)\dot{\theta}(t)$ with instantaneous twist coordinates bivector $\hat{U}(t)$, defined by eq. (47).

The spatial velocity bivector contains the angular and linear velocity with:

$$U = B\dot{\theta} + \hat{V}\dot{\theta} = \Omega + V \tag{50}$$

where Ω is the angular velocity bivector and $V = vn_\infty$ is a bivector representing the linear velocity v.

3.4.3 Velocity with CGA

Consider $X = n_0 + x + \frac{1}{2}x^2 n_\infty$, where the point x moves with velocity $\dot{x} = v$. Differentiating the conformal point gives:

$$\begin{aligned}\dot{X} &= \dot{x} + \frac{1}{2}(x\dot{x} + \dot{x}x)n_\infty \\ &= \dot{x} + x \cdot \dot{x}n_\infty \\ &= v + x \cdot vn_\infty\end{aligned}$$

A bivector $V = vn_\infty$ representing the velocity of this point is extracted with:

$$V = \dot{X}n_\infty = vn_\infty \tag{51}$$

This is identical to how linear velocity is represented in the spatial velocity bivector in eq. (50).

With this, the velocity of a fixed point P can be found:

$$\begin{aligned}V_P &= \widetilde{R}V_P'R \\ &= \widetilde{R}\frac{d}{dt}\left(TX_P\widetilde{T}\right)n_\infty R \\ &= -\frac{1}{2}\widetilde{R}\left(TUX_P\widetilde{T} + TX_P\widetilde{U}\widetilde{T}\right)n_\infty R \\ &= \widetilde{R}T\frac{1}{2}(X_PU - UX_P)\widetilde{T}n_\infty R \\ &= \widetilde{R}T(X_P \cdot U)\widetilde{T}n_\infty R \\ &= \widetilde{R}R_tR(v + x_P \cdot \Omega)\widetilde{R}\widetilde{V}_tn_\infty R \\ &= (v + x_P \cdot \Omega)n_\infty\end{aligned}$$

where R_t commutes with $R(\cdots)\widetilde{R}$ and disappears because there is no n_0 term.

This gives $v_P = v + x_P \cdot \Omega$ as expected.

3.5 Serial robot kinematics using CGA

3.5.1 Forward kinematics

When using CGA, versors represent the pose of reference frames in the same way as rigid body transformation matrices.

A point ${}^jx$ in frame j transforms to the position ${}^ix$ in frame i with:

$${}^jX = {}^jT_i{}^iX{}^j\widetilde{T}_i$$

for the corresponding conformal points X.

For an n-link serial chain, the pose of the end effector is represented by:

$$T = {}^0T_n = {}^0T_1 \cdots {}^{n-1}T_n \tag{52}$$

analogous to eq. (38) for conventional methods.

Similarly, the versor for joint i is parametrised by q_i using DH parameters and the spatial velocity bivector of frame i is ${}^iU$, defined by eq. (50), satisfying eq. (49).

After evaluating the end effector versor, the position vector and orientation rotor can be extracted. Details of this given in Appendix A.

3.5.2 Spatial velocity and the Jacobian

Like with conventional methods, the goal is to find ${}^nU$ such that:

$$\frac{d}{dt}\left({}^0T_n\right) = {}^0T_n{}^nU$$

The spatial velocity for a given joint is ${}^iU_i = {}^i\hat{U}_i\dot{q}_i$. When using DH parameters, the twist coordinates are equal to:

$${}^i\hat{U}_i = e_{12}$$

Using the product rule and eq. (49), the derivative of the transform is evaluated:

$$\begin{aligned}\frac{d}{dt}\left({}^0T_n\right) &= \frac{d}{dt}\left({}^0T_1 \cdots {}^{n-1}T_n\right) \\ &= {}^0T_n \left[\sum_{i=1}^{n} {}^i\widetilde{T}_n \, {}^iU_i \, {}^iT_n \dot{q}\right] \\ &= {}^0T_n \left[\sum_{i=1}^{n} {}^nU_i \dot{q}\right]\end{aligned}$$

Therefore:

$${}^nU = \sum_{i=1}^{n} {}^nU_i \dot{q}_i$$

which is equivalent to a vector representation:

$${}^n\boldsymbol{u} = \sum_{i=1}^{n} {}^n\boldsymbol{u}_i \dot{q}_i = J\dot{\boldsymbol{q}}$$

with:

$$J = \begin{bmatrix} \uparrow & & \uparrow \\ {}^n\hat{\boldsymbol{u}}_1({}^nU_1) & \cdots & {}^n\hat{\boldsymbol{u}}_n({}^nU_n) \\ \downarrow & & \downarrow \end{bmatrix}$$

where ${}^n\boldsymbol{u}_i({}^nU_i)$ indicates the vector equivalent of the spatial velocity bivector. This simply involves reading the components along e_{23}, e_{31}, etc.

Transforming spatial velocity

Transforming from ${}^iU$ to ${}^jU$ uses:

$${}^jU = {}^j\widetilde{T}_i \, {}^iU \, {}^jT_i$$

noting that the reverse of the transform is applied instead.

To show that this gives the expected result:

$$\begin{aligned}{}^jU &= {}^j\widetilde{R}_i(1 - \frac{1}{2}n_\infty {}^jp_i)({}^i\Omega + {}^iv n_\infty)(1 + \frac{1}{2}n_\infty {}^jp_i)R \\ &= \widetilde{R}(\Omega + ({}^iv + {}^jp_i \cdot {}^i\Omega)n_\infty)R \\ &= \widetilde{R}^i\Omega R + \left(\widetilde{R}\left({}^iv + {}^jp_i \cdot {}^i\Omega\right)R\right)n_\infty \\ &= {}^j\Omega + {}^jv n_\infty\end{aligned}$$

which transforms the linear and angular velocity in the same was as seen earlier.

With this, the twist coordinates of each joint are referred to the end effector directly by the joint transform versors, giving:

$${}^n\hat{U}_i = {}^n\widetilde{T}_{n-1} \cdots {}^{i+1}\widetilde{T}_i \, {}^i\hat{U}^{i+1}T_i \cdots {}^nT_{n-1}$$

3.5.3 Inverse kinematics

Denoting $U(\boldsymbol{u})$ the bivector equivalent of $\boldsymbol{u}$:

$$\delta T \approx 1 - \frac{1}{2}\delta U = 1 - \frac{1}{2}U(J\boldsymbol{\delta q})$$

This means that an incremental change in joint positions gives:

$$T(\boldsymbol{q} + \boldsymbol{\delta q}) \approx T(\boldsymbol{q})(1 - \frac{1}{2}U(J\boldsymbol{\delta q})) \tag{53}$$

Denote the current joint positions $\boldsymbol{q}$. The target pose is represented by versor $T_* = T(\boldsymbol{q}_*)$.

Let $\boldsymbol{q}_* = \boldsymbol{q} + \boldsymbol{\delta q}$ and by using eq. (53):

$$\begin{aligned}T_* &\approx T(\boldsymbol{q})(1 - \frac{1}{2}U(J\boldsymbol{\delta q})) \\ U(J\boldsymbol{\delta q}) &= 2(1 - \widetilde{T}T_*)\end{aligned}$$

This corresponds to the least squares problem:

$$\boldsymbol{\delta q} = J(\boldsymbol{q})^+ U^{-1}\left[2(1 - \widetilde{T}T_*)\right]$$

where $\boldsymbol{u} = U^{-1}[\cdot]$ denotes the operation of solving $U(\boldsymbol{u}) = (\cdot)$, which requires reading the appropriate components from the bivector.

If the solution to the above is denoted $\boldsymbol{\delta q}(\boldsymbol{q})$, the final solution is converged on with:

$$\boldsymbol{q}_{k+1} = \boldsymbol{\delta q}(\boldsymbol{q}_k)$$

3.6 Control

This section looks at how the kinematic solutions provided for the delta and serial robot can be applied to manual control of the robot and having it move through a trajectory between poses[10].

3.6.1 Manual control

Instantaneous forward kinematics uses the Jacobian to find the end effector twist for a given set of joint velocities $\dot{\boldsymbol{q}}$:

$$\boldsymbol{u} = J\dot{\boldsymbol{q}}$$

For the delta robot, this is only contains linear velocity, while for the serial robot it contains angular and linear velocity.

In the case of the serial robot, the spatial velocity is given in the end effector frame. For manual control, it is easier for the linear velocity to be expressed in the fixed frame, while keeping the angular velocity defined in the end effector frame. Denoting this adjusted spatial velocity as $\boldsymbol{u}_*$:

$$\boldsymbol{u}_* = J_*\dot{\boldsymbol{q}}$$

with a modified Jacobian:

$$J_* = \begin{bmatrix} I & 0 \\ 0 & {}^0R_n \end{bmatrix} J \tag{54}$$

When using CGA, the component of the twist vector corresponding to linear velocity can be rotated by ${}^0R_n$ to give the same effect.

To perform instantaneous inverse kinematics, ie: choose the joint velocities to give a desired twist, the inverse relationship needs to be solved:

$$\dot{\boldsymbol{q}} = J_*^+\boldsymbol{u}_*$$

The pseudoinverse can be found using SVD to threshold the singular values. This avoids the pseudoinverse having large singular values, which leads to undesired large joint velocities.

Using this, the end effector can be commanded to move with a desired twist. To avoid getting close to singularities a constraint on the allowable Jacobian singular values can be added. When the minimum singular value falls below a certain value, mark that state as invalid and stop the robot from continuing with that twist.

3.6.2 Interpolating trajectories

The second method of control required is to interpolate and execute trajectories between the current pose T_0 and target pose T_1.

Let u denote the normalised "progress" between the two poses and $\tau = t/T$ the normalised time, for total travel time T. A cubic polynomial $u(\tau)$ is fitted such that:

$$u(0) = 0 \quad u(1) = 1 \quad \dot{u}(0) = 0 \quad \dot{u}(1) = 0$$

which has the form:

$$u(\tau) = 3\tau^2 - 2\tau^3 \tag{55}$$

Interpolating position is the same for conventional and CGA:

$$\begin{aligned} \delta p &= p_1 - p_0 \\ p(u) &= p_0 + u\delta p \end{aligned} \tag{56}$$

To interpolate a rotor:

$$\begin{aligned} \delta R &= \widetilde{R}_0 R_1 \\ &= \exp(-\frac{1}{2}B\delta\theta) \\ R(u) &= R_0 \exp(-\frac{1}{2}Bu\delta\theta) \end{aligned} \tag{57}$$

The conventional method uses SLERP to interpolate quaternions, which is identical to how rotors are interpolated. Note, that like with SLERP, when $\theta > \pi$, the value $\theta - 2\pi$ is interpolated instead in order to rotate about the shorter angle.

The trajectory has a target linear velocity and angular velocity, selecting the limiting choice. Additionally, there is a maximum joint speed which shouldn't be exceeded.

At any point during this trajectory, the linear and angular velocity are given as:

$$\begin{aligned}\dot{p} &= \frac{6}{T}\tau(\tau-1)\delta p \\ \boldsymbol{\omega} &= \frac{6}{T}\tau(\tau-1)\hat{\boldsymbol{\omega}}\delta\theta \\ \Omega &= \frac{6}{T}\tau(\tau-1)B\delta\theta \end{aligned} \tag{58}$$

The maximum linear and angular speeds are $1.5T|\delta p|$ and $1.5T\delta\theta$ such that for maximum linear and angular speeds $v_{\max}$ and $\omega_{\max}$, the trial trajectory time T is equal to:

$$T = \max\left(\frac{v_{\max}}{1.5|\delta p|}, \frac{\omega_{\max}}{1.5\delta\theta}\right) \tag{59}$$

which ensures both linear and angular velocity are less than or equal to their maximum allowed value.

For a constant time-step size δT, the number of data-points is:

$$N = (T/\Delta T) + 1 \tag{60}$$

Finally, the trajectory is interpolated and a trajectory of corresponding joint positions is returned. The maximum joint speed is also found, such that the trajectory time can be scaled if this is larger than a target maximum joint speed.

The algorithm for fitting the trajectory is given in Algorithm 1.

Algorithm 1 Interpolate a trajectory

$\dot{q}_{\max} \leftarrow 0$
$q_0 \leftarrow$ Current Joint Positions
for $n = 1, 2, \ldots, N$ **do**
  $\tau_n \leftarrow (n+1)/N$
  $t_n \leftarrow \tau_n \times T$
  $u_n \leftarrow 3\tau^2 - 2\tau^3$
  $q_n \leftarrow$IK(q_{n-1}, $p(u_n)$, $R(u_n)$)
  $v_n \leftarrow (6/T)\tau(1-\tau)\delta p$
  $\omega_n \leftarrow (6/T)\tau(1-\tau)\hat{\omega}\delta\theta$
  $\dot{q}_n \leftarrow$ VEL_IK(q_n, v_n, ω_n)
  $\dot{q}_{\max} \leftarrow \max(\dot{q}_{\max}, |\dot{q}_n|)$
end for
if $\dot{q}_{\max} > \dot{q}_{\text{target}}$ **then**
  for $n = 1, 2, \ldots, N$ **do**
    $t_n \leftarrow t_n \times (\dot{q}_{\text{target}}/\dot{q}_{\max})$
  end for
  return $\{q_n\}_{n=1}^N$, $\{t_n\}_{n=1}^N$, $\dot{q}_{\max}$
end if

where:

- IK(q_0, p, R) solves the inverse kinematics for position p and orientation R, starting from initial joint positions q_0.
- VEL_IK(q, v, ω) solves the instantaneous inverse kinematics for current joint positions q, target linear velocity v and target angular velocity ω.

This could easily return the joint speeds as well, which could be used in joint control. However, for this project only the position will be used for simplicity.

4 Code

In order to demonstrate that these methods can be used for control of robots, they will be used for a practical control task, on simulated and real robots.

The task used to demonstrate this is the following:

- Allow the end effector to be manually controlled using an Xbox controller, including the position, orientation and angle of a gripper.
- Provide a real-time visualisation of the robot state.
- Allow the user to record a set of waypoints, and have the robot automatically move between these.

Robotics software typically consists of a large number of components, operating at a range of rates, with varying levels of complexity. The result is that robotics software is best written as distributed software, where different components run on different processes and communicate via a message passing protocol.

To facilitate this, a *middleware* is required. In this case, the robotics operating system (ROS) will be used, due to being open source and having a well-developed ecosystem.

ROS calls individual components *nodes*. Nodes run as separate (possibly multithreaded) processes. Communication is done via a pub-sub protocol, organised using *topics*, or via a client/server protocol, organised using *services*.

The following code will be required:

- A dedicated CGA library, used to perform fundamental CGA operations as well as provide functions for geometry and transformations.
- A kinematics library for defining a robot model and performing kinematics calculations.
- A ROS package that defines robot models for the delta and serial robots and nodes to visualise and control these, able to control simulated or real robots.

ROS nodes can be written using *C++* or *Python*, but because kinematics usually needs to performed quickly, *C++* was chosen.

This section will give an overview of the CGA and kinematics libraries along with the ROS package that makes use of them. For more details of code, GitHub links are given in Appendix C.

For simulation, the Gazebo robotics simulator[11] was used. Information on the hardware used is given in Appendix D.

4.1 CGA library

A number of C++ CGA libraries exist already. However, it was decided to write a custom CGA library. This would allow for finer control of the data structures and functions provided to focus on kinematics. Compared to other libraries which aim to provide a general library, supporting a range of different algebras, a custom library would be simpler to use and likely more efficient.

The CGA library directory contains the following:

```
cga/
    generator/
        main.py
        structs.py
        operations.py
        write.py
    include/cga/
        cga.h
        constants.h
        geometry.h
        transform.h
    src/
        cga.cpp
```

The library consists of a code generator under `generator/`, written in *Python*; and the *C++*

source code under the `include/` and `src/` directories. This includes the `cga.h` and `cga.cpp` files written by the code generator and a number of other header files to provide specific functions.

4.1.1 Code generator

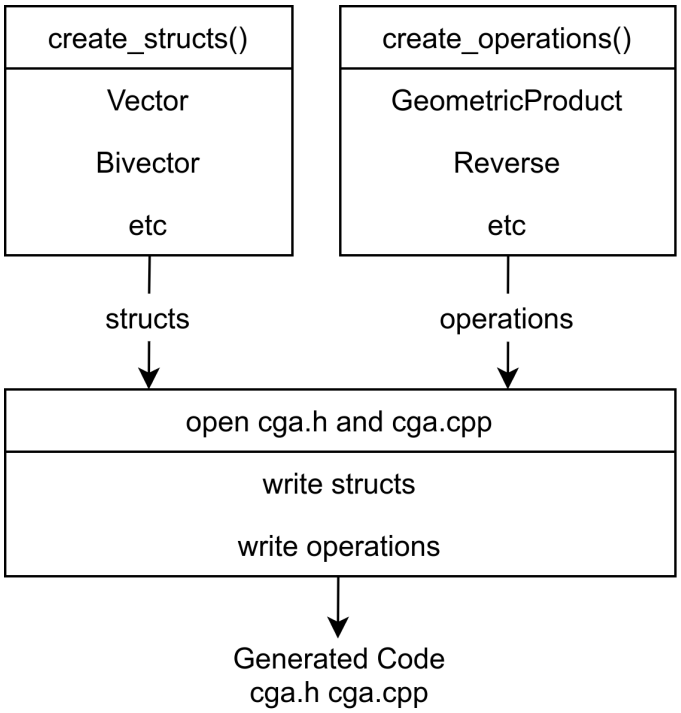

Figure 10: Overview of how the CGA code generator works.

The code generator writes the `cga.h` and `cga.cpp` source files, which define a `struct` for each CGA object and a number of operations between them. Within the generator, classes are used to represent structs and objects, which are then used to write the code, as shown in Figure 10.

Symbolic algebra

The `sympy` and `galgebra` python packages are used to provide symbolic algebra for conformal geometric algebra.

Figure 11 shows how the `galgebra` package is used to define a number of basis vectors. These are later used when defining structs.

```
from galgebra.ga import Ga
G = Ga("e_1 e_2 e_3 e_4 e_5",
       g=[1, 1, 1, 1, -1])
e1, e2, e3, e4, e5 = G.mv()
eo = 0.5*(e4 + e5)
ei = (e5 - e4)
s = e1*e1
```

Figure 11: Code used to define basis vectors for CGA. Creates the geometric algebra by providing names of basis vectors and a signature, then defines two null vectors, `eo` and `ei` for n_0 and n_∞.

Symbolic algebra allows for creating and manipulating algebraic expressions, consisting of some combination of geometric algebra objects and variable names.

If two arguments for an operation are given as symbolic algebra expressions, with specific variable names, the final expression will also be in terms of these variable names.

Structs

All structs are represented by the `Struct` class.

Figure 12 shows how a `Vector` struct is created. The object is first instantiated with just a name. It is fully defined by adding variables. These are the variables that will appear as members in the struct itself, as shown in Figure 13.

```
Vector = Struct("Vector")
Vector.add_variable("e1", e1, e1)
Vector.add_variable("e2", e2, e2)
Vector.add_variable("e3", e3, e3)
Vector.add_variable("eo", eo, -ei)
Vector.add_variable("ei", ei, -eo)
```

Figure 12: Creation of a `Vector` struct, by giving it a name and defining variables.

```
struct Vector {
    double e1;
    double e2;
    double e3;
    double eo;
    double ei;
    // Constructors, methods, etc ...
};
```

Figure 13: The member variables within the Vector struct, as an example of how variables map to struct members.

Creation of a variable requires three arguments:

1. Name: Name of the variable within the struct.
2. Expression: A symbolic algebra expression for the variable.
3. Extractor: For a CGA expression M, taking the inner product of the expression with the extractor x (specifically $x \cdot M$) will give the component corresponding to that variable.

For a vector v, the symbolic algebra expression generated for the vector is:

$$(\text{v.e1})e_1 + (\text{v.e2})e_2 + \cdots$$

where (v.e1) is the reference to the corresponding variable within the struct and e_1 is the CGA vector.

The expression can be freely manipulated, while maintaining references to the variables within the struct. This means that the expression returned is in terms of these variable names.

In addition to simple structs such as the vector, *compound structs* are defined, such as the `Rotor` which contains a scalar (`double`) variable `s` and a `Bivector` variable `b`. With this, the e_{12} component of the rotor `R` is evaluated with `R.b.e12`.

Structs also contain information for writing constructors and conversions between types.

Operations

Operations are defined by their own classes, such as `GeometricProduct`, `OuterProduct`, etc.

Each operation class provides two functions, `write_individual` and `write`.

The `write_individual` writes a function for computing the operation between a given operand or pair of operands. For a binary operation, this has the form:

```
def write_inviddidual(self,
                      op1,
                      op2,
                      available,
                      f_h,
                      f_cpp):
    # Function body
```

The function does the following:

- Compute an expression for the result of applying the operation between the two operands `op1` and `op2`, such as between a vector and bivector.
- Find the most compact available struct for the return type, from `available`, a list of structs available as return types.
- Write the function declaration to the header file `f_h`.
- Write the function definition to the source file `f_cpp`, where each component of the return object is set using an expression given by the extractor.
- Write the same operation with the operands flipped, if they are different operands, such as $a \wedge B$ vs $B \wedge a$, exploiting commution or anticommution where possible.

The `write` function writes operations for all structs (for unary operations) or all pairs of structs (for binary operations). There is flexibility in defining which the pairs for which functions are computed for. Certain combinations are less useful, so may not be worth generating.

Example operation

The inner product between a 3D vector and 3D bivector will be shown as an example.

The function declaration is:

```
Vector3 inner(const Vector3 &lhs,
              const Bivector3 &rhs);
```

where the `Vector3` is on the left-hand side, the `Bivector3` on the right-hand side.

Expressions for each struct are equal to:

$$v = (\text{lhs.e1})e_1 + (\text{lhs.e2})e_2 + (\text{lhs.e3})e_3$$
$$B = (\text{rhs.e23})e_{23} + (\text{rhs.e31})e_{31} + (\text{rhs.e12})e_{12}$$

Evaluating $a = v \wedge B$ gives the result:

$$\begin{aligned} a = \quad & [(\text{lhs.e3})(\text{rhs.e31}) - (\text{lhs.e2})(\text{rhs.e12})]e_1 \\ & + [(\text{lhs.e1})(\text{rhs.e12}) - (\text{lhs.e3})(\text{rhs.e23})]e_2 \\ & + [(\text{lhs.e2})(\text{rhs.e23}) - (\text{lhs.e1})(\text{rhs.e31})]e_3 \end{aligned}$$

The most compact struct for this result is the `Vector3` struct.

The expressions for each variable within the return struct are:

$$e_1 \cdot a = (\text{lhs.e3})(\text{rhs.e31}) - (\text{lhs.e2})(\text{rhs.e12})$$
$$e_2 \cdot a = (\text{lhs.e1})(\text{rhs.e12}) - (\text{lhs.e3})(\text{rhs.e23})$$
$$e_3 \cdot a = (\text{lhs.e2})(\text{rhs.e23}) - (\text{lhs.e1})(\text{rhs.e31})$$

Figure 14 gives the generated function body.

```
Vector3 inner(const Vector3 &lhs,
              const Bivector3 &rhs)
{
    Vector3 result;
    result.e1 = lhs.e3*rhs.e31
                - lhs.e2*rhs.e12;
    result.e2 = lhs.e1*rhs.e12
                - lhs.e3*rhs.e23;
    result.e3 = lhs.e2*rhs.e23
                - lhs.e1*rhs.e31;
    return result;
}
```

Figure 14: The function body for the inner product between a 3D vector and 3D bivector, showing how the value of each variable of the return object is set.

4.1.2 Library source code

The `cga.h` and `cga.cpp` source code provides structs for representing CGA objects and functions for CGA operations, all written by the generator.

The library also contains header files with inline functions for a number of other specific operations.

Constants

The `constants.h` header file contains useful constants. This includes the basis vectors and pseudoscalars. Some examples are:

```
static Vector3 e1(1, 0, 0);
static Vector eo(0, 0, 0, 1, 0);
static Pseudoscalar I5(1);
```

Geometry

The `geometry.h` header file contains functions for creating geometric primitives, intersecting them and interpreting the results. These function are primarily used by the delta robot kinematics.

Examples are given here for creating a conformal point (Figure 15) and creating a dual sphere (Figure 16). The code for intersecting three dual spheres and interpreting the result is given in Figure 31 in Appendix B.

```
inline Vector make_point(
    const Vector3 &x)
{
    Vector result = x;
    result.eo = 1;
    result.ei = 0.5*norm2(x);
    return result;
}
```

Figure 15: C++ function for creating a conformal point for a given position, using eq. (17).

```
inline Vector make_dual_sphere(
    const Vector3& position,
    double radius)
{
    Vector result = position;
    result.eo = 1;
    result.ei = 0.5*(norm2(position)
                     - radius*radius);
    return result;
}
```

Figure 16: C++ function for creating a dual sphere for a given position and radius, using eq. (19).

Transformations

The `transforms.h` header file provides functions for creating translations, rotations and rigid body transformations. These function are primarily used by the serial robot kinematics.

For convenience, functions are also provided to apply the transformations. Figure 17 shows a function used to transform a vector with an arbitrary versor using eq. (43).

```
inline Vector transform_vector(
    const Vector &x, const Versor &T)
{
    return (T*x*reverse(T)).v;
}
```

Figure 17: Function used to transform a vector using a versor by evaluating $Tx\widetilde{T}$. On evaluating Tx or $x\widetilde{T}$ this expands to a non-vector expression, but the final expression will only contain a vector component due to the structure of T. Therefore, only the vector component is returned.

4.2 Kinematics library

The kinematics library was called `cbot`, for CGA robotics. The library compiles to produce a dynamic library `cbot.so`, where a compilation flag is set to choose conventional or CGA implementations of the kinematics functions, as outlined in Figure 18.

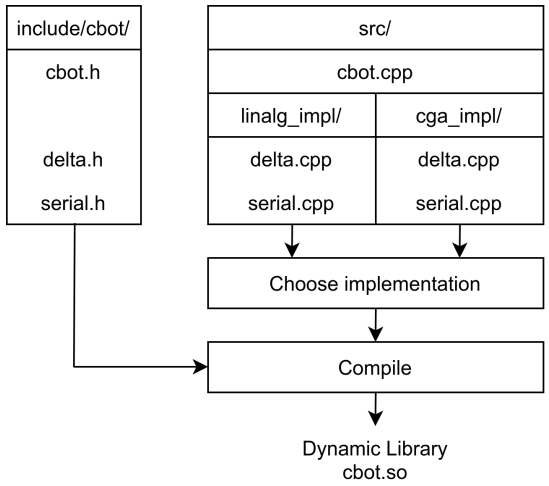

Figure 18: Overview of how the kinematics library is structured. Header files are provided independent of a particular implementation. On compiling, a compile flag selects source files from the *linalg* (for linear algebra, conventional methods) or *cga* to select how the kinematics are implemented.

4.2.1 Header files

The header files provide the interface to the library, primarily in the form of classes for the two robot types, `Delta` and `Serial`.

Data structures for kinematics

Within `cbot.h` a number of data structures are defined for kinematics. These are used for arguments and return types from kinematic functions.

The structure and naming of these are chosen to match those provided by ROS to make conversion easy.

The data structures provided are:

- `Pose`: Contains position and orientation, with orientation represented as a quaternion.
- `Twist`: Twist, or spatial velocity, containing linear and angular velocity.
- `Joint`: Position and velocity of a single joint. Also has a *dependent* flag to indicate if the joint is independent or dependent.
- `Joints`: An unordered map of joint name to joint state, used to represent all joint states for a robot.
- `JointTrajectoryPoint`: A single point in a joint trajectory, containing a list of joint positions and a time.
- `JointTrajectory`: A list of joint trajectory points and a list of joint names.
- `TrajectoryConstraints`: A data structure defining maximum speed constraints, taken into account when interpolating a trajectory.

Robot classes

Also within `cbot.h` is a base class `Robot`. The `Delta` and `Serial` classes, in `delta.h` and `serial.h` respectively inherit from this class.

`Robot` defines the interface for a robot class, by defining pure virtual functions that child classes must implement.

The most important of these functions are the following:

- `bool update_pose()`
  Compute the forward kinematics of the robot, by updating the robot state and end effector pose from the joint positions.
- `bool update_twist()`
  Compute the instantaneous forward kinematics of the robot, by updating the end effector twist from the joint velocities.
- `bool update_joint_positions()`
  Compute the inverse kinematics of the robot, by setting the joint positions to give the desired end effector pose.
- `bool update_joint_velocities()`
  Compute the instantaneous inverse kinematics of the robot, by setting the joint velocities to give the desired end effector twist.
- `bool calculate_trajectory(const Pose &goal)`
  Calculate and store the joint trajectory to travel from the current robot state to the goal pose.
- `bool update_dependent_joints()`
  Calculate the dependent joint positions.

All functions return a `bool` to indicate whether the calculation was successful.

The robot class itself will store the robot state, including joint states, end effector pose and twist and the calculated trajectory. Other function are provided for writing and reading these.

The `Delta` and `Serial` classes are similar to the base class. However, they also define some extra data structures for specifying the dimensions of the robot.

Joints are referenced by name, such that they can be looked up in the `Joints` data structure. Since an unordered map is used (implemented using a hash table in C++), lookup is fast.

Figure 19 shows how a delta robot object is created and the forward kinematics is solved, including finding the dependent joint positions and reading the solution.

```
cbot::Delta::Dimensions dim;
dim.r_base = 0.15;
// Set other dimensions ...

cbot::Delta::JointNames joint_names;
joint_names.theta.push_back("theta1");
// ...
joint_names.alpha.push_back("alpha1");
// ...

cbot::Delta delta(dim, joint_names);
delta.set_joint_position("theta1", 0.3);
delta.set_joint_position("theta2", -0.1);
delta.set_joint_position("theta3", 0.7);

if (!delta.update_pose()) {
    return 1; // Failed, exit
}

// Read result of forward kinematics
cbot::Pose pose = delta.get_pose();

if (!delta.update_dependent_joints()) {
    return 1; // Failed, exit
}

// Read dependent joint values
double alpha1 =
    delta.get_joint_position("alpha1");
// ...
```

Figure 19: Code used to create a delta robot object, set the joint positions, perform forward kinematics and read the result.

4.2.2 Conventional implementations

Implementations of the ksinematics functions are defined in `linalg/delta.cpp` and `linalg/serial.cpp`.

The `Eigen` library is used to perform linear algebra, both for kinematics and other operations such as computing the SVD.

Delta robot kinematics

The forward and inverse kinematics are calculated using Section 3.1.1 and Section 3.1.2, using the series of calculations outlined.

Instantaneous forward and inverse kinematics use the Jacobian, which once the forward kinematics has been calculated, is calculated using Section 3.1.5.

Serial robot kinematics

For the forward kinematics, the transform for each joint is first calculated, then concatenated to give the end effector transform as in Section 3.3.1, shown in Figure 32 in Appendix B.

On the other hand, inverse kinematics uses Newton's method, as outlined in Section 3.3.3. The core part of this function is shown in Figure 20, which makes use of a function for calculating the approximate twist to reach the goal transform and a function for checking when the twist is close enough to zero (by it's norm).

```
// ee_transform is the current end
   effector transform and
// goal_transform is the supplied goal
   transform
while (i < max_iter) {
    update_pose();
    twist = get_twist_coord_change(
        ee_transform, goal_transform);
    if (twist_is_zero(norm)) break;

    update_jacobian();
    j_svd.compute(J, flags);

    delta_q = j_svd.solve(twist);
    // Add delta_q to joint positions ...

    i++;
}
```

Figure 20: Code for the inverse kinematics of the serial robot using conventional methods or CGA. Performs forward kinematics to update the pose. Calculates the approximate twist to reach the end effector. Adds to the joint positions appropriately. Stops when the twist norm falls below a threshold. The difference between conventional and CGA implementations is the variable types and the `get_twist_coord_change(...)` and `twist_is_zero(...)` functions.

The instantaneous forward and inverse kinematics uses the Jacobian found using Section 3.3.2.

4.2.3 CGA implementations

Implementations of the kinematics functions are defined in `cga/delta.cpp` and `cga/serial.cpp`.

The `cga` library is used for CGA operations to implement kinematics, but `Eigen` is still used for the SVD and related operations.

Delta robot kinematics

The delta robot implementations makes use of geometry functions provided by the CGA library. Forward kinematics follows Section 3.1.3 and inverse kinematics follows Section 3.1.4.

The code for the forward kinematics is much simpler than the conventional function and is shown in Figure 33 in Appendix B.

The Jacobian is found the same was as conventionally, following Section 3.1.5.

Serial robot kinematics

Forward kinematics is applied the same was as conventionally, first creating versors for each joint, then multiplying together, as shown in Section 3.3.1. The code is identical to that given in Figure 32 but with versors for transforms and a different `get_dh_transform(...)` function.

Similarly, the inverse kinematics when using CGA follows Section 3.5.3 and the implementation is identical to Figure 20 except using different variable types. The functions `get_twist_coord_change(...)` and `twist_is_zero(...)` are what change.

The Jacobian is found using Section 3.5.2, transforming the spatial velocity from each joint to the end effector. However, unlike conventional methods, it doesn't need to calculate spatial velocity transforms to use and instead re-uses the joint transforms.

4.3 ROS package

Figure 21 and Figure 22 show an overview of the nodes and topics used to control the delta and serial robots. Both robots use similar architectures except that the delta robot is parallel and therefore a node is needed to perform the forward kinematics to completely define the joint positions, fully defining the robot state for visualisation.

This section will look at the different components in more detail.

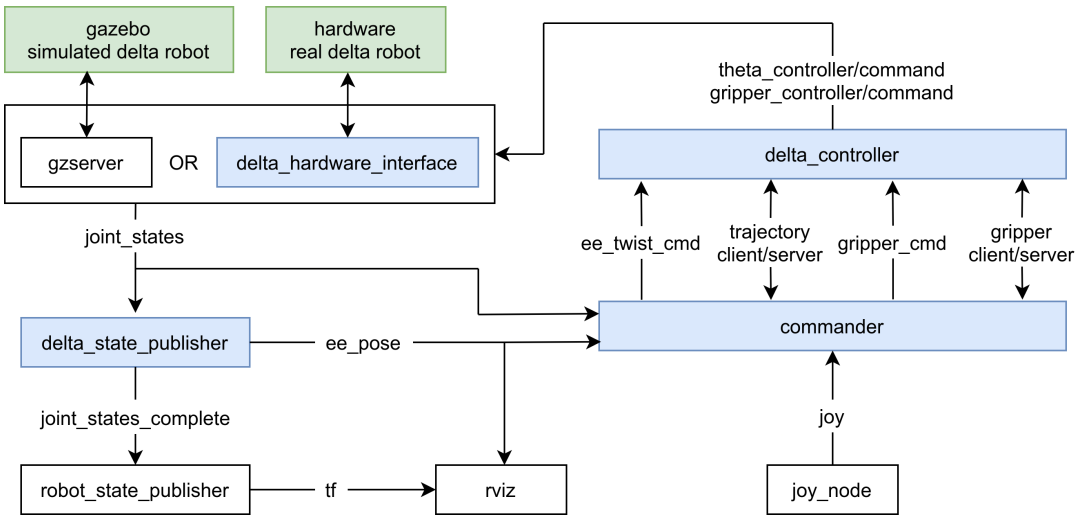

Figure 21: Graph of the nodes and topics used to control the delta robot and visualise its state. Non-ROS components are highlighted in green and custom nodes are highlighted in blue.

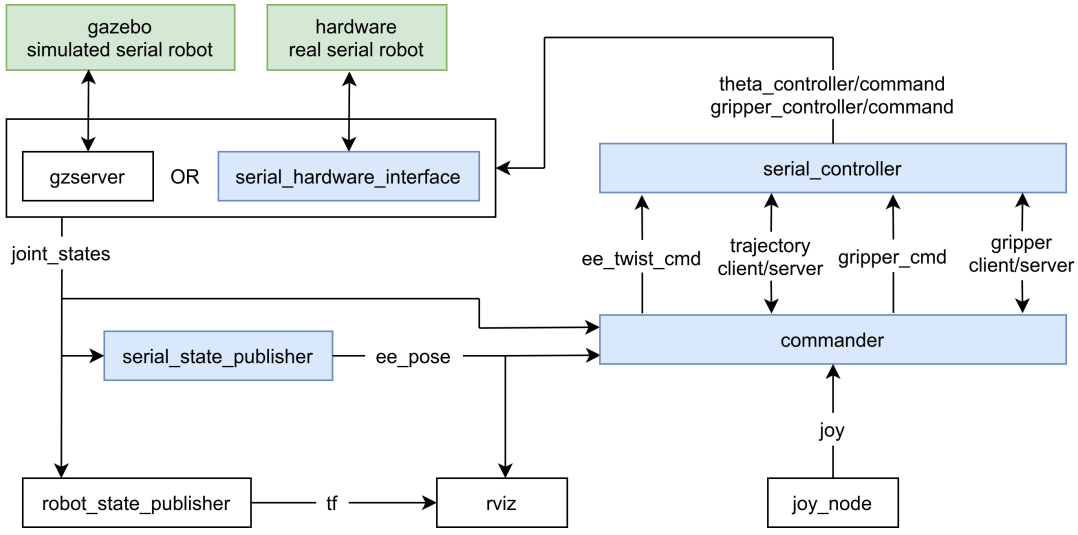

Figure 22: Graph of the nodes and topics used to control the serial robot and visualise its state. Non-ROS components are highlighted in green and custom nodes are highlighted in blue.

4.3.1 Modelling

The first step is to specify a description of the robot model for visualisation and forward kinematics. This is done by creating `URDF` files for the delta and serial robot, standing for Unified Robot Description Format. These files have an `XML` format, containing elements defining links, joints and some other details.

The problem encountered with the delta robot is that `URDF` files do not support parallel robots. The kinematic structure must form a tree. The solution is to remove joints which close the loop and manually set these in order to close the loop.Figure 23 shows the robot model for the delta robot with and without correct dependent joints set.

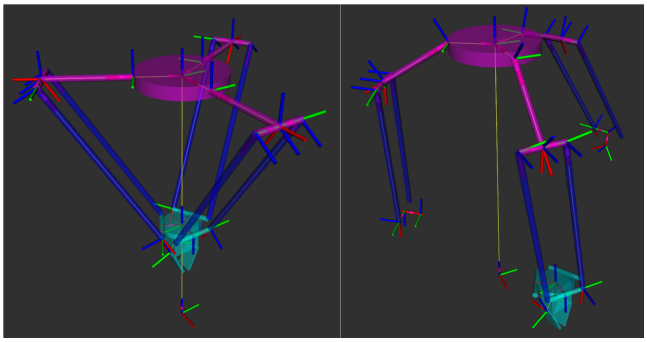

Figure 23: Screenshot of the `URDF` model of the delta robot, visualised in `rviz`. On the left, the dependent joints have been set correctly using forward kinematics, while on the right they have been left at arbitrary values.

4.3.2 Control of simulation and hardware

The `gzserver` node within the `gazebo_ros` package is responsible for running the robot simulations. It provides topics for writing joint commands and reading joint states. Similarly, for actual hardware, a custom hardware interface node is written, providing the same topics.

To standardise the topics, the `ros_control` package is used to organise joint control. For both robots, two *controllers* are created, one for the main joints that set the end effector pose, called `theta_controller` and one for the gripper joint called `gripper_controller`. Commands are written to the joints under the `theta_controller/command` topic for example, specifying all joint position targets. Another controller called the `joint_state_controller` is created for providing feedback of the joint states under the `joint_states` topic.

4.3.3 Commander node

The commander node is responsible for processing the `joy` topic from the `joy_node` node, which provides input from a gamepad such as an Xbox controller.

Two modes of control are implemented: manual and automatic.

Manual control

The gamepad inputs are used to set the linear and angular velocity of the end effector. For the delta robot, angular velocity is ignored. A `ee_twist_cmd` topic is published, which the robot controller then executes.

This also includes manual control of the gripper joint.

When in manual control mode, the `A` button is used to store the current pose, while the `X` button is used to store the current gripper angle. These will be used in automatic control.

Automatic control

Whenever the current pose or gripper angle is saved, this creates a task and places it on a task queue.

When the `Y` button is pressed, the commander switches into an automatic control mode. It will then execute the saved tasks automatically. For a *pose task* it will automatically move from the current pose to the saved goal pose. For a *gripper task* it will automatically move from the current gripper angle to the saved goal gripper angle.

The commander doesn't execute these tasks itself. Instead, it communicates with the controller node via action servers in order to initiate the tasks and

monitor their progress. The two action servers provided are called `trajectory` and `gripper` for pose trajectory tasks and gripper tasks.

4.3.4 Controller

The two controller nodes, `delta_controller` and `serial_controller` both use the same code. They differ in which robot class is used, either `Delta` or `Serial`.

The controller uses the `robot.update_joint_positions()` function to perform instantaneous inverse kinematics. This finds the joint velocities required to execute the desired end effector twist, which is received over the `ee_twist_cmd` topic. This is then executed simply by integrating the joint positions on a loop.

The controller also starts a trajectory server. When the commander sends a request, it switches to trajectory mode. The `robot.calculate_trajectory(goal)` function calculates the trajectory to the requested pose from the current state, making use of inverse kinematics to calculate the required joint positions at each step of the trajectory. The controller then reads the joint positions off the trajectory over time, until the end of the trajectory is reached.

4.3.5 State publisher and visualisation

The two state publisher nodes, `delta_state_publisher` and `serial_state_publisher` also use similar code, using the `Delta` or `Serial` robot classes.

Both use `robot.update_pose()` to determine the current end effector pose. This is set by listening to the `joint_states` topic which provides the joint positions. The pose is published as `ee_pose` and to be displayed in `rviz`.

The delta robot is parallel, so must also use forward kinematics to determine the dependent joint positions. The delta robot also has the `delta.update_dependent_joints()` function to do this. The complete set of joint states is published as the `joint_states_complete` topic.

The `rviz` program can display the robot model and markers for expected pose and twist. To display the model, `rviz` must know the robot structure (using a provided `URDF` file) and know the *transforms* of link, provided under the `tf` topic.

The link transforms (poses) could have been manually calculated by the robot classes themselves. However, it seemed more sensible to make use of the `robot_state_publisher` node which will calculate the transforms.

This also makes it easier to validate that the forward kinematics is accurate, by checking that the end effector pose marker matches the position and orientation of the end effector on the robot model.

5 Results

5.1 Execution times for kinematics functions

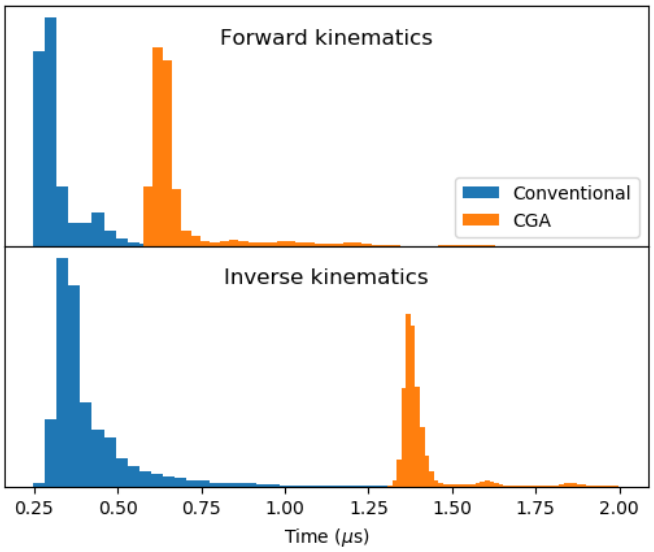

Figure 24: Histogram of execution times for performing the forward and inverse kinematics of the delta robot with conventional or CGA methods.

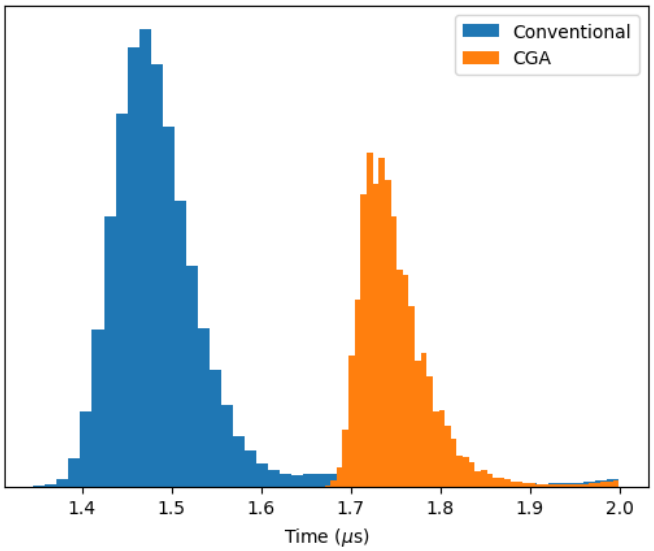

Figure 25: Histogram of execution times for performing the forward kinematics of the serial robot with conventional or CGA methods.

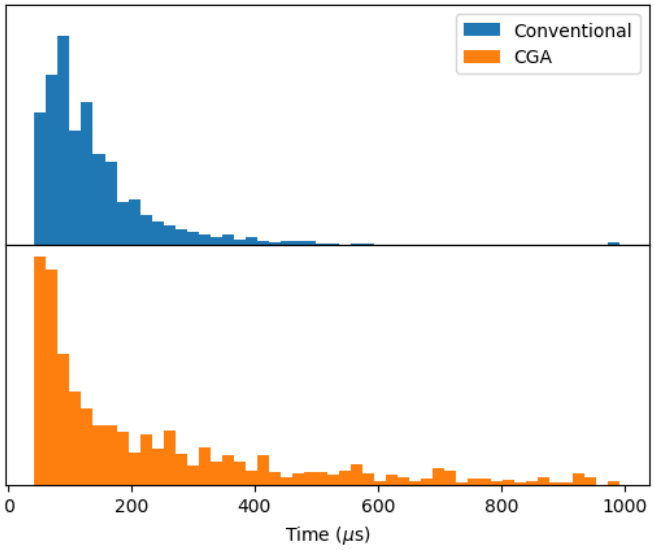

Figure 26: Histogram of execution times for performing the inverse kinematics of the serial robot with conventional or CGA methods.

5.2 Control tasks

Videos are available for all tasks mentioned here, available at:
https://www.youtube.com/playlist?list=PLHSZpbJPMrmbw1spEI9z1c3tPFYf5xKDF

In all cases, CGA was used for the kinematics and `rviz` was open to visualise the robot state alongside the simulation or hardware. The red arrow shown in `rviz` indicates the end effector pose estimate from the state publisher, offset to lie in the centre of the gripper.

5.2.1 Delta robot

The delta robot could be manually controlled in simulation and with hardware.

For the simulated robot, it was able to automatically move an object from one place to another, shown in Figure 27. It was occasionally unsuccessful due to slip between the gripper and cube.

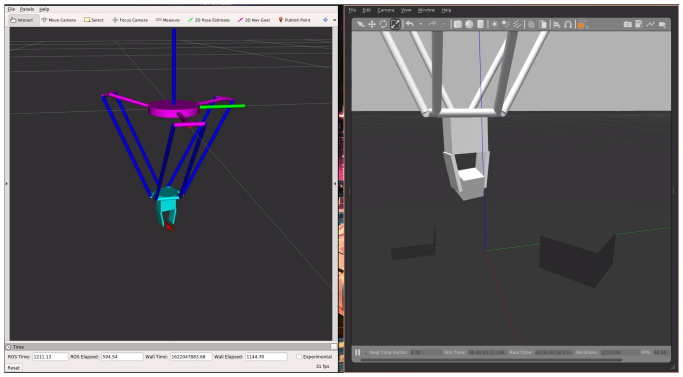

Figure 27: Screenshot of the simulated delta robot automatically moving a cube from one position to another.

The real delta robot was able to automatically move between saved positions, shown in Figure 28, although there was insufficient time to setup the gripper control.

The motion was slightly jittery due to using quickly written firmware, but could made smoother with more time (eg: using micro-stepping on stepper motors).

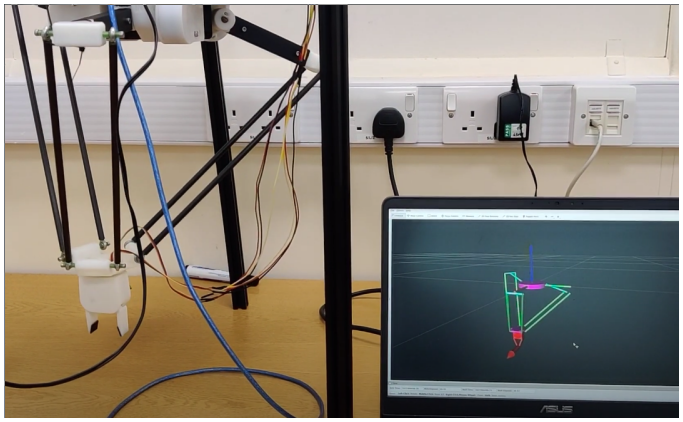

Figure 28: Photo of the real delta robot automatically moving between saved positions.

5.2.2 Serial robot

Likewise, manual control worked well with the serial robot, both in simulation and with hardware.

For the simulated robot, it was able to be commanded to stack two objects on top of one another, shown in Figure 29. Again, issues occurred when there was slip between the cube and gripper, but other than that, it was able to reproduce the desired behaviour accurately.

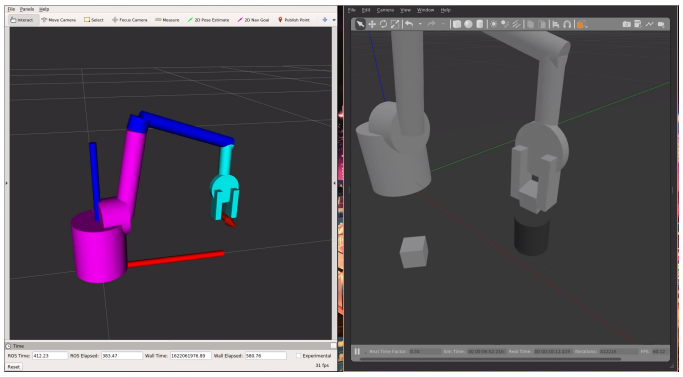

Figure 29: Screenshot of the simulated serial robot automatically stacking cubes.

The real robot was first commanded to move an object from one position to another. Following this, it was able to be setup to automatically stack three objects, shown in Figure 30, after carefully planning it's motion.

The main issue encountered with the real serial robot was the limited accuracy inherent in a low-cost robot kit like this, but it was sufficient for this project.

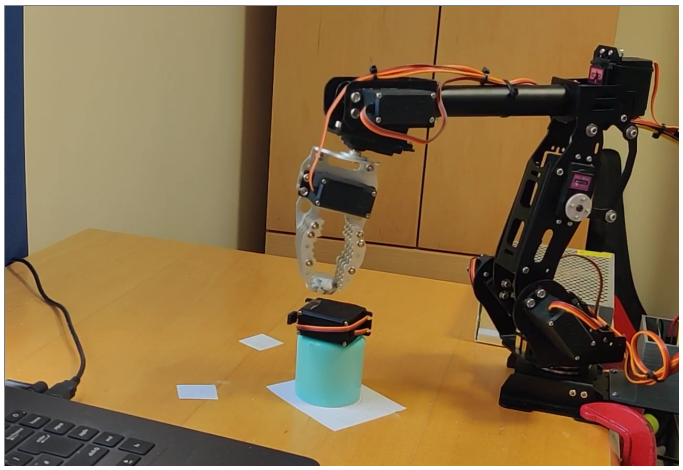

Figure 30: Photo of the real serial robot automatically stacking a number of objects.

6 Discussion

6.1 Using CGA to solve delta robot kinematics

Execution times

For the delta robot, execution times were longer for the forward and inverse kinematics, as shown in Figure 24. The inverse kinematics was particularly slower since the standard algebraic solution was simple, such that using intersection of geometric primitives gave an unnecessary overhead.

However, because solutions are found analytically, execution times are still consistently small, making it irrelevant that CGA is slower if it is still sufficiently fast.

Advantages and disadvantages of CGA

The advantage of using CGA is that for someone who understands CGA, writing code to perform geometric operations becomes much easier. The conventional solution required an explicit parameterisation of the spheres and it was easy to make a mistake when solving or implementing.

The only disadvantage is the increased execution time and the less intuitive method for someone unfamiliar with CGA. However, a carefully designed library eliminates this issue, especially if functions are available for interpreting all intermediate intersection results, such as circles, removing the need for a library user to understand CGA.

6.2 Using CGA to solve serial robot kinematics

Execution times

Like with the delta robot kinematics, because the forward kinematics of the serial robot had a closed form solution, execution times were consistently small, shown in Figure 25. Again, CGA gave slightly larger execution times, but was still sufficiently fast.

The inverse kinematics of the serial robot differs to previous functions in that it is solved numerically. Previous functions were of the order of microseconds, while the serial robot inverse kinematics is of the order of 100s of microseconds, as shown in Figure 26.

Execution time is primarily determined by how many steps are required to converge, although CGA execution times tended by be longer, indicating a longer time per step. However, the performance of both implementations mostly depends on how close the initial joint positions are to the solution and what accuracy is required.

For example, this implementation used a threshold of 1μm and 10^{-6} radians to threshold the error in linear and angular displacement. Due to the solution converging asymptotically if this accuracy could be smaller, the execution times would be significantly smaller.

Additionally, in this application, inverse kinematics is only used for points along a trajectory. In every case, the step from current to goal pose is very small, meaning inverse kinematics can be performed very quickly.

Advantages and disadvantages of CGA

The two methods are very similar, only differing in how pose and twist are represented. However, the advantage of CGA is that it is more consistent. Rigid body transformations, twist, orientation are all represented within the same algebra. This makes writing code simpler.

On the other hand, conventional methods requires a range of objects: joint transformations as homogeneous matrices, spatial velocity transformations as 6×6 matrices, orientations as quaternions. Conversions are often required between these different objects.

In addition to the increased execution times, a disadvantage of CGA is that it still requires the use of linear algebra for defining and using the Jacobian, requiring some conversions.

6.3 Task performance

All videos recorded of the robot performing tasks used the CGA implementations of the kinematics functions.

As shown, the delta and serial robots were both able to be manually controlled and accurately perform the planned tasks. Any limitations were down to limitations in simulation or hardware, not CGA as a method for kinematics.

6.4 Improvements to CGA library

Although the CGA library performs well, there is room for a few improvements.

Firstly, functions could be generated for special operations such as the versor product $Vx\widetilde{V}$. When the versor V has a particular structure, this expression will always return an object of the same grade as the argument x. When evaluating it via two geometric products, the geometric products have no concept of this, so calculate all components. The final result will always have most of it's components be non-zero, making the full intermediate calculations inefficient. This would make the serial robot inverse kinematics significantly faster.

Secondly, the code generator generates a large number of data structures and functions between them, which bloats the library. This doesn't affect execution times and is not a problem if running on a modern computer. However, if CGA was to be used for kinematics on embedded devices, which have more limited memory, this is a lot of wasted program memory. Instead, the generator could be configured to only generate the necessary data structures and functions. For example, for a serial robot, it may only need rotors, versors, vectors and bivectors.

6.5 Improvements to other code

For the robotics library, providing two possible implementations added a fair bit of complexity and duplication of code. If taking the project further, it may make more sense to focus on CGA and only use this.

The control code could be taken further by using planning methods such as RRT*, allowing the robot to plan trajectories under a larger number of constraints, where it can't simply interpolate a straight path. This could take into account joint limits, self-collisions and collisions with the environment. Additionally a cost function could be used to search for minimum energy paths.

6.6 Further applications of CGA

The kinematics library could be extended to support a number of other robot topologies such as the steward platform and agile eye, with kinematics implemented with CGA.

CGA can also be used for geometrical control tasks, such as constraining the end effector to move about the surface of a sphere.

Finally, since CGA can be used for computer vision, this could be integrated with control tasks. An example is *visual servoing* where a camera views the workspace of a manipulator and provides feedback to the controller of the distance between the end effector and a target objective, such as when trying to pick up an object.

7 Conclusion

CGA was shown to be a viable alternative to conventional methods for performing kinematics operations and was able to be applied to the practical control of robots:

- The execution times when using CGA to solving kinematics problems was longer than when using conventional methods, but still sufficiently fast. This could be improved by further optimising the code.
- For someone familiar with CGA, writing code to solve kinematics tends to be simpler due to simple geometric operations and the use of a consistent algebra.
- Practical usage was demonstrated by using CGA for kinematics functions in the control of a delta and serial robot. Both were able to be manually controlled and setup to automatically perform manipulation tasks.

Further work could investigate the kinematics of other types of robot, such as the agile eye; use CGA for more specialised control tasks such as constraining motion about the surface of a sphere; or integrating computer vision into control, such as with visual servoing, which can also use CGA.

References

[1] Diagram of a serial robot. Available at: https://orocos.org/kdl_old (Accessed: 31 May 2021)

[2] Diagram of a delta robot. Available at: http://www.multibody.net/teaching/msms/students-projects-2019-2/delta-robot/attachment/1-20/ (Acdessed: 31 May 2021)

[3] Wareham R., Cameron J., Lasenby J. (2005) *Applications of Conformal Geometric Algebra in Computer Vision and Graphics* In: Li H., Olver P.J., Sommer G. (eds) Computer Algebra and Geometric Algebra with Applications. IWMM 2004, GIAE 2004. Lecture Notes in Computer Science, vol 3519. Springer, Berlin, Heidelberg.

[4] ROS (2020) Available at: https://www.ros.org/ (Accessed: 31 May 2021)

[5] Hadfield H., Wei L., Lasenby J. (2020) *The forward and inverse kinematics of a delta robot* In: Magnenat-Thalmann N. et al. (eds) Advances in Computer Graphics. CGI 2020. Lecture Notes in Computer Science, vol 12221. Springer, Cham.

[6] Unity game engine (2020) Available at: https://unity.com/ (Accessed: 31 May 2021)

[7] Hestenes D., Sobczyk G. (1987) *Clifford Algebra to Geometric Calculus, a Unified Language for Mathematics and Physics*

[8] Lasenby A., Lasenby J., Wareham R. (2004) *A covariant approach to geometry using geometric algebra*

[9] Siciliano B., Khatib O. (2016) *Springer handbook of robotics, 2nd edition* Chapter 2

[10] Siciliano B., Sciavicco L., Villani L., Oriolo G. (2008) *Robotics - Modelling, Planning and Control* Chapter 4

[11] Gazebo (2020) Available at: http://gazebosim.org/ (Accessed: 31 May 2021)

A Additional theory

Expression for a rigid body transformation using DH parameters

Expressions are provided for a homogeneous matrix and a CGA versor. In both cases, the result can be calculated more easily by multiplying the translation and rotation operators (as is done in this project), but the following expressions could be evaluated directly for better efficiency.

For conventional methods, the homogeneous matrix is:

$$
\begin{aligned}
{}^{i-1}T_i &= \mathrm{Trans}_X(a_{i-1})\mathrm{Rot}_X(\alpha_{i-1})\mathrm{Trans}_Z(d_i)\mathrm{Rot}_Z(\theta_i) \\
&= \begin{bmatrix} 1 & 0 & 0 & a_{i-1} \\ 0 & \cos\alpha_{i-1} & -\sin\alpha_{i-1} & 0 \\ 0 & \sin\alpha_{i-1} & \cos\alpha_{i-1} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_i \\ \cos\alpha_{i-1}\sin\theta_i & \cos\alpha_{i-1}\cos\theta_i & -\sin\alpha_{i-1} & -d_i\sin\alpha_{i-1} \\ \sin\alpha_{i-1}\sin\theta_i & \sin\alpha_{i-1}\cos\theta_i & \cos\alpha_{i-1} & d_i\cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

For CGA methods, the versor is:

$$
\begin{aligned}
{}^{i-1}T_i &= \mathrm{Trans}_X(a_{i-1})\mathrm{Rot}_X(\alpha_{i-1})\mathrm{Trans}_Z(d)\mathrm{Rot}_Z(\theta_i) \\
&= \left[\cos\left(\frac{\alpha_{i-1}}{2}\right) - \frac{1}{2}e_{23}\sin\left(\frac{\alpha_{i-1}}{2}\right) + \frac{1}{2}n_\infty e_1 a_{i-1}\cos\left(\frac{\alpha_{i-1}}{2}\right) - \frac{1}{2}n_\infty e_{123}a_{i-1}\sin\left(\frac{\alpha_{i-1}}{2}\right)\right] \\
&\quad \left[\cos\left(\frac{\theta_i}{2}\right) - \frac{1}{2}e_{12}\sin\left(\frac{\theta_i}{2}\right) + \frac{1}{2}n_\infty e_3 d_i\cos\left(\frac{\theta_i}{2}\right) - \frac{1}{2}n_\infty e_{123}d_i\sin\left(\frac{\theta_i}{2}\right)\right] \\
&= \cos\left(\frac{\alpha_{i-1}}{2}\right)\cos\left(\frac{\theta_i}{2}\right) \\
&\quad - e_{23}\left[\sin\left(\frac{\alpha_{i-1}}{2}\right)\cos\left(\frac{\theta_i}{2}\right)\right] + e_{31}\left[\sin\left(\frac{\alpha_{i-1}}{2}\right)\sin\left(\frac{\theta_i}{2}\right)\right] - e_{12}\left[\cos\left(\frac{\alpha_{i-1}}{2}\right)\sin\left(\frac{\theta_i}{2}\right)\right] \\
&\quad + \frac{1}{2}n_\infty e_1\left[a_{i-1}\cos\left(\frac{\alpha_{i-1}}{2}\right)\cos\left(\frac{\theta_i}{2}\right) - d_i\sin\left(\frac{\alpha_{i-1}}{2}\right)\sin\left(\frac{\theta_i}{2}\right)\right] \\
&\quad - \frac{1}{2}n_\infty e_2\left[a_{i-1}\cos\left(\frac{\alpha_{i-1}}{2}\right)\sin\left(\frac{\theta_i}{2}\right) + d_i\sin\left(\frac{\alpha_{i-1}}{2}\right)\cos\left(\frac{\theta_i}{2}\right)\right] \\
&\quad + \frac{1}{2}n_\infty e_3\left[d_i\cos\left(\frac{\alpha_{i-1}}{2}\right)\cos\left(\frac{\theta_i}{2}\right) - d_{i-1}\sin\left(\frac{\alpha_{i-1}}{2}\right)\sin\left(\frac{\theta_i}{2}\right)\right] \\
&\quad - \frac{1}{2}n_\infty e_{123}\left[a_{i-1}\sin\left(\frac{\alpha_{i-1}}{2}\right)\cos\left(\frac{\theta_i}{2}\right) + d_i\cos\left(\frac{\alpha_{i-1}}{2}\right)\sin\left(\frac{\theta_i}{2}\right)\right]
\end{aligned}
$$

Extracting the position and orientation from a rigid body transformation versor

A point at the origin isn't affected by the rotation, so:

$$P = Tn_0\widetilde{T} \qquad p = \frac{(P \cdot e_1)e_1 + (P \cdot e_2)e_2 + (P \cdot e_3)e_3}{-P \cdot n_\infty}$$

Then construct the corresponding translation and extract the rotor, since $T = R_t R$:

$$R_t = 1 + \frac{1}{2}n_\infty p \qquad R = \widetilde{R}_t T$$

B Larger pieces of code

```
inline PointPair intersect(
    const Vector spheres[3])
{
    PointPair intersection;
    cga::Bivector T = dual(outer(spheres[0], outer(spheres[1], spheres[2]))
    );

    if (inner(T, T) < 0) {
        intersection.valid = false;
    } else {
        intersection.valid = true;
        cga::Rotor P(1, T/std::sqrt(inner(T, T)));
        cga::Vector Y = transform_vector(inner(T, cga::ei), P);
        intersection.point1 = describe(Y).point.position;
        Y = transform_vector(inner(T, cga::ei), reverse(P));
        intersection.point2 = describe(Y).point.position;
    }
    return intersection;
}
```

Figure 31: A C++ function used to intersect three spheres and interpret the result using Section 2.3.7, returning a `PointPair` object that contains the two intersections and a valid flag.

```
for (std::size_t i = 0; i < joints.size(); i++) {
    dim.dh_parameters[i].set_q(joints[joint_names[i]].position);
    transforms[i] = get_dh_transform(dim.dh_parameters[i]);
}

ee_transform = transforms[0];
for (std::size_t i = 1; i < joints.size(); i++) {
    ee_transform = ee_transform * transforms[i];
}
```

Figure 32: The C++ code for the forward kinematics of the serial robot using either conventional methods or CGA. The two implementations differ in the variable used for the transforms and the `get_dh_transform(...)` function.

```
// Member variables
bool Delta::update_pose()
{
    // Member variables set:
    //     cga::Vector3 d[3];
    //     cga::Vector3 x;

    cga::Vector D_sphere[3];
    double theta_i;
    for (int i = 0; i < 3; i++) {
        // Get joint position
        theta_i = joints.at(joint_names.theta[i]).position;
        // Define position of pseuo-elbow
        d[i] = u[i] * (dim.r_base + dim.l_upper * std::cos(theta_i) - dim.r_ee)
               - cga::e3 * (dim.l_upper*std::sin(theta_i));
        // Create dual sphere
        D_sphere[i] = cga::make_dual_sphere(d[i], dim.l_lower);
    }
    auto result = cga::intersect(D_sphere);
    if (!result.valid) return false;

    x = (result.point1.e3 < result.point2.e3 ? result.point1 : result.point2);

    // Update the pose member variable, which is returned
    // by the get_pose() function.
    // ...
}
```

Figure 33: Code used to perform the forward kinematics of the delta robot using CGA. It creates points for the three pseudo-elbow positions, forms dual spheres, intersects them and selects the appropriate point.

C Code repositories

Repository for the CGA and CGA robotics libraries:
https://github.com/zachlambert/cga-robotics

Repository for the ROS package:
https://github.com/zachlambert/cga-robotics-ros

Repository for the custom firmware used on the delta robot:
https://github.com/zachlambert/delta-x-custom-firmware

D Hardware

Figure 34 shows a photo of the delta robot kit used. It uses three stepper motors, driven by a RepRap stepper controller (commonly used in 3D printers) and controlled by an Arduino mega. This allowed for custom firmware to be written to the device.

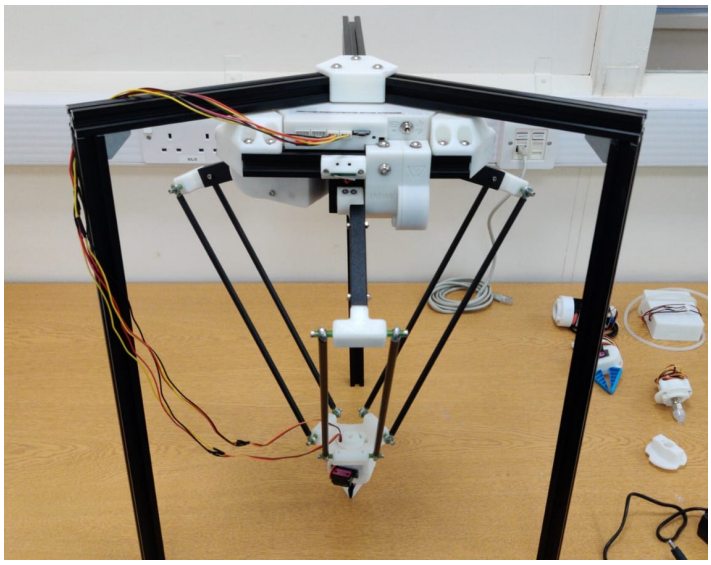

Figure 34: Photo of the assembled delta robot kit with a gripper attached.

Figure 35 shows the serial robot kit used and the electronics required to control it. The serial robot didn't come with a controller, so instead a USB serial controller, was used to drive the 8 servos (using two servos for joint 2). A switching power regulator was used to supply the 5V to the servos, connected with a piece of strip board to ensure it could support the current required.

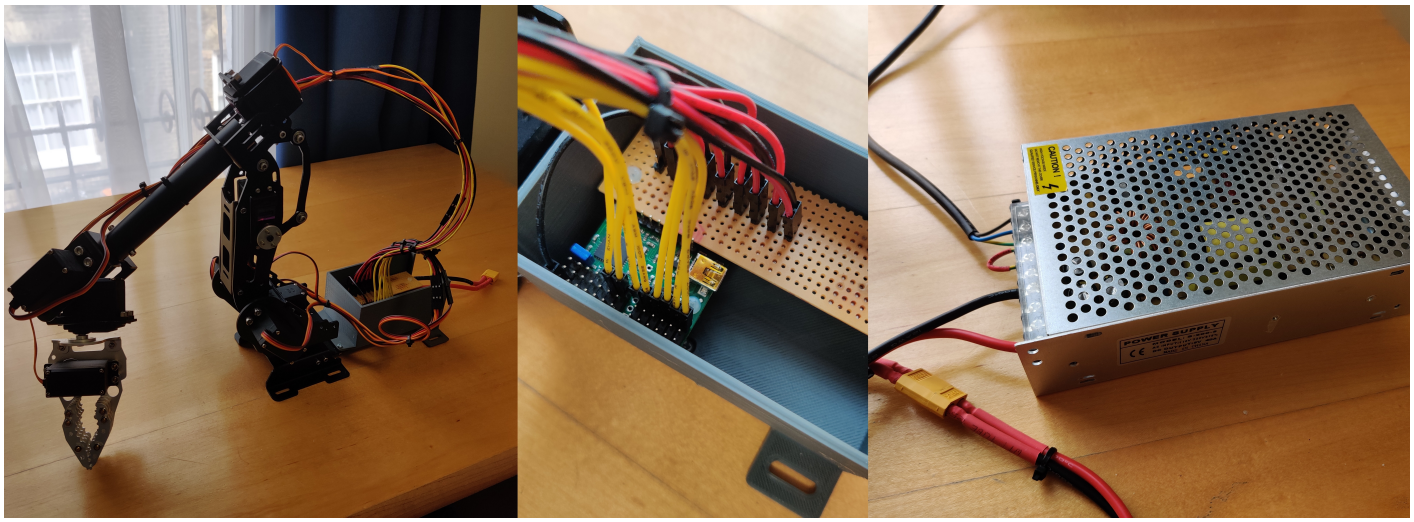

Figure 35: Photo of the assembled serial robot kit (left), USB servo controller (centre) and power supply(right).

Delta robot	https://store.deltaxrobot.com/products/delta-x-basic-kit
Arduino board	https://store.arduino.cc/arduino-mega-2560-rev3
Serial robot	https://www.vvdoit.com/SZDOIT-8DOF-Metal-Robotic-Arm-8-Axis-Mechanical-Arm-With-Gripper-Kit-ABB-Industrial-Robot-Model-360-degree-Rotating-Base-Motors-p2755804.html
USB servo controller	https://www.pololu.com/product/1352

E Covid-19 disruption

Covid-19 prevented testing with the delta robot kit during Lent term, leaving just a few days in Easter to get some code working. However, this didn't affect the majority of the project since a delta robot simulation could be used. Using simulations was the contingency plan, so this worked well.

Final word count = 11,985, including headers, captions and appendices, excluding technical abstract.