Inverse kinematics of the Lynxmotion AL5B arm

Cory Walker
Electrical Engineering and Computer Science
University of Tennessee
Knoxville, TN 37996-2250

May 4, 2016

Contents

1 Introduction

When we ordered the Lynxmotion AL5B arm, we simply get an arm and no software to make it useful. A key part of using any robotic arm is understanding the kinematics of it so that it can be used with dexterity. The servos operate by receiving PWM signals from a microcontroller to indicate a desired goal angle. When the arm is powered on, each joint servo plays a role in the movement, and we need to be able to calculate where the arm gripper will be with certain servo positions. Perhaps more importantly, we need to be able to calculate what we should set the servo values to if we want the gripper to be at a certain position.

Figure 1: Lynxmotion AL5B

For this year's robot, after loading a set of blocks, we would have an entire hopper full of blocks, 8 rows wide and 2 layers high.

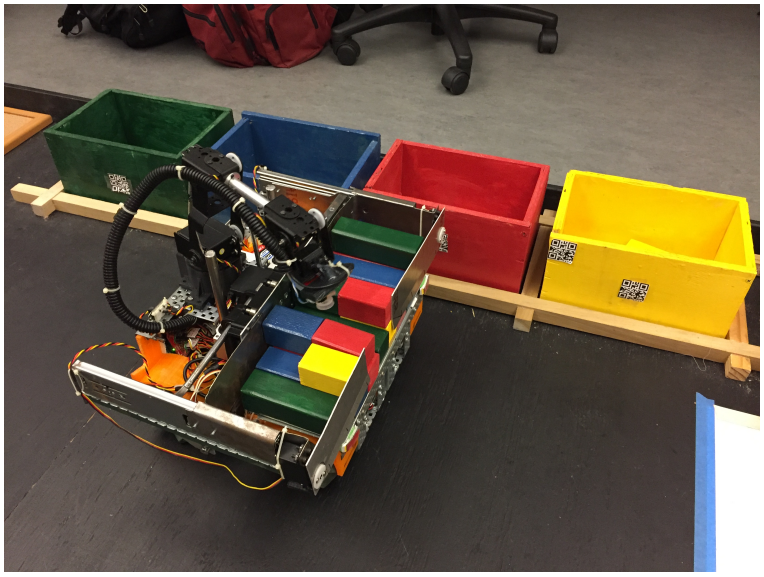


Figure 2: This year's robot

Our arm, with a vacuum-powered suction cup mounted, needed to support picking and placing of these blocks into the colored bins seen above. The blocks on either level can either be full length blocks or half length blocks. Manually coding in the arm angles would require a lookup table of $8 * 3 * 3 = 72$ arm positions. Since each pick and place location requires 3 intermediate locations, we would actually need $72 * 3 = 216$ hard coded arm positions. Because each arm position requires setting 4 servo values, we would need to make a total of $216 * 4 = 864$ servo angle measurements to build our lookup table. This would be incredibly tedious, and if we ever needed to change the length of an arm joint or move the base of the arm (which happened often) we would need to make all these measurements all over again.

Calculating the X, Y, and Z positions of all the blocks is trivial because they have defined positions and are arranged in a grid. If we could simply tell the arm what X, Y, and Z coordinate to go to, we can skip all of these manual servo angle calibrations. This is why understanding the inverse kinematics of the arm is important.

2 Formulation of coordinate system

We will use centimeters and radians throughout our calculations. Here are a few diagrams that will help explain our coordinate system:

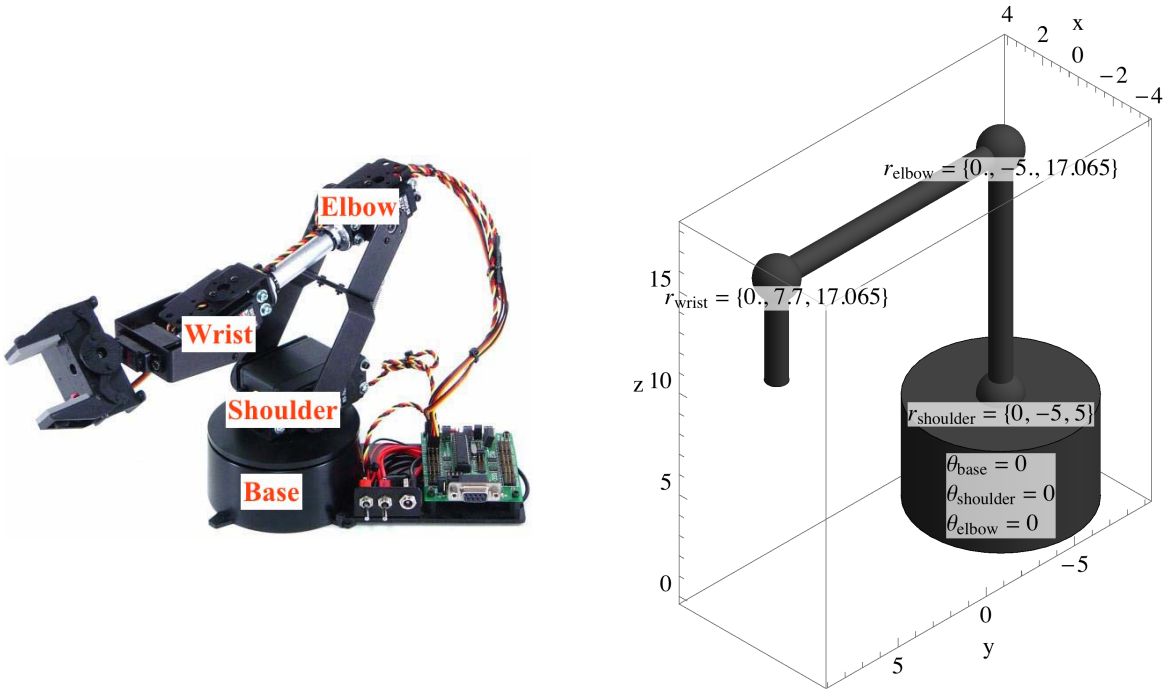


Figure 3: Actual and model arm with labels.

The figure to the left designates some helpful terms for important points on the arm. A human analogy is helpful here, so we have named the joints after shoulders, elbows, and wrists. The figure to the right is our simplified model of the arm, along with annotated points and angles. The pose given above is the reset pose, where all θ values are set to zero. A $\theta_{base} = 0$ means that the arm points perfectly forward, and a $\theta_{base} > 0$ means that the arm moves clockwise from zero when viewed from above. A $\theta_{shoulder} = 0$ means that the first segment of the arm points directly upwards. Increasing this value will rotate the entire arm forward, with the wrist moving closer to the ground:

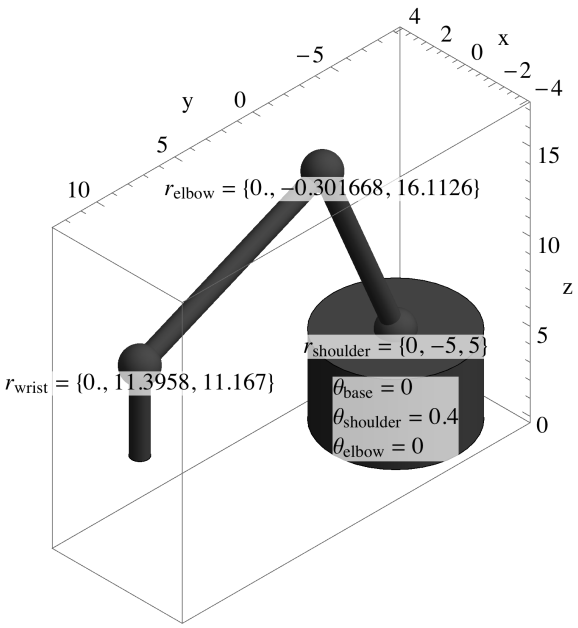


Figure 4: Arm with $\theta_{shoulder} = 0.4$.

Notice how the wrist angle looks rotated here in order to still point directly downwards. You should also notice that θ_{wrist} is not included in the diagram. This is because by ignoring θ_{wrist}, we can simplify our calculations significantly. For our robot, we almost always want the wrist to point directly downwards for optimal suction abilities. Knowing this, we can simply tell our arm to travel to a destination 5cm above and set θ_{wrist} to the angle that points it down.

Moving on to the final θ value, a $\theta_{elbow} = 0$ means that the second length of the arm is perfectly perpendicular to the first length of the arm. Increasing θ_{elbow} will angle the second length of the arm towards the ground:

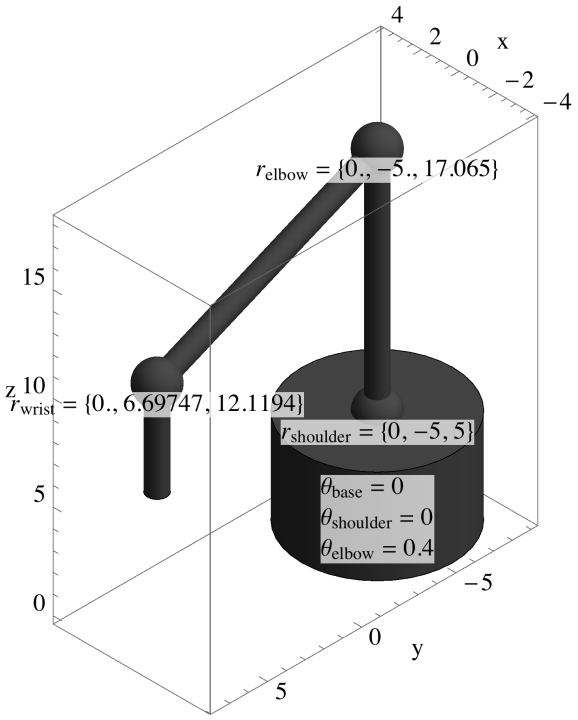


Figure 5: Arm with $\theta_{elbow} = 0.4$.

If we want to mimic the pose shown in Figure 3, we can set the θ values to the following:

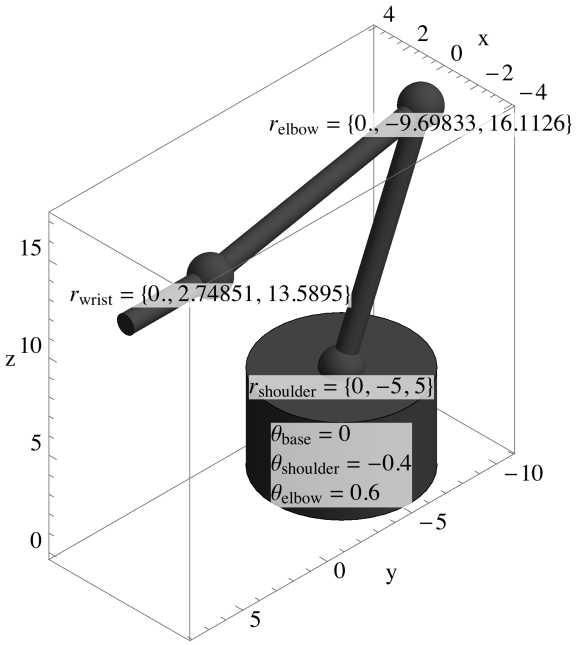


Figure 6: Model arm that recreates pose in the actual picture of arm.

It is important to note that we can define the position of the arm either by supplying r_{wrist} or supplying $\theta = \{\theta_{base}, \theta_{shoulder}, \theta_{elbow}\}$. There is no reason to specify both, because by specifying

θ, we can calculate r_{wrist}, known as forward kinematics. By specifying r_{wrist}, we can calculate θ, known as reverse or inverse kinematics.

2.1 Predefined constants

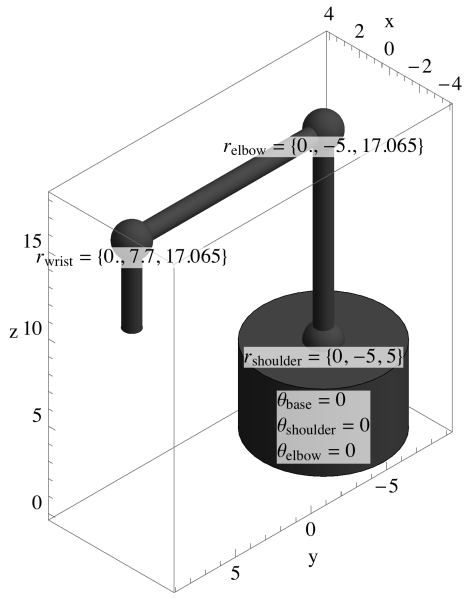


Figure 7: Reset position of the arm

In our system, we will have three imporant constants. The first predefined constant in our calculations is $r_{shoulder}$. For our robot this year, I decided on $r_{shoulder} = \{0, -5, 5\}$. I chose this constant because the center of the shoulder joint was very close to 5cm behind and 5cm above the front bottom of the arm assembly. We could have also been fine by setting $r_{shoulder} = \{0, 0, 0\}$, but $r_{shoulder} = \{0, -5, 5\}$ seems much more intuitive to me. We end up subtracting this value as soon as possible in our inverse kinematics equations.

The second constant is the shoulder to elbow distance, which is defined to be 12.065 cm. As a vector, this could be $\overrightarrow{r_s r_e} = \{0, 0, 12.065\}$. Notice the abbreviation of $r_{shoulder}$ and r_{elbow}.

The third constant is the elbow to wrist distance, which is defined to be 12.7 cm. As a vector, this could be $\overrightarrow{r_e r_w} = \{0, 12.7, 0\}$. Notice the abbreviation of r_{wrist}.

Collectively, these constants can allow us to determine r_{wrist} if $\theta = \{0, 0, 0\}$:

$$r_w = r_s + \overrightarrow{r_s r_e} + \overrightarrow{r_e r_w} = \{0, -5, 5\} + \{0, 0, 12.065\} + \{0, 12.7, 0\} = \{0, 7.7, 17.065\} \tag{1}$$

Which can be verified in Figure 7 above. Of course, if $\theta \neq \{0, 0, 0\}$, our calculations are more complicated, but we will address these calculations below.

3 Choice of tool and technical representation of coordinates

I knew that solving such a system of equations would involve large equations so I knew that the best tool for the job was Mathematica, a symbolic math package. Mathematica allows for the automation of mathematical operations.

4 Solve for forward kinematic equations

For the forward kinematic equations, we are supplied with $\{\theta_{base}, \theta_{shoulder}, \theta_{elbow}\}$ and we must calculate r_{wrist}. By the end of this section, we will have an equation for r_{wrist}.

4.1 Rotation matrices

For this section and the remaining sections, we will make use of Mathematica's RotationMatrix function. According to the documentation,

$$\text{RotationMatrix}(\theta, w) \tag{2}$$

gives the 3D rotation matrix for a counterclockwise rotation around the 3D vector w. For example:

$$\text{RotationMatrix}(\theta, \{1,0,0\}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix} \tag{3}$$

Thus to rotate any given 3D vector v by θ radians counter clockwise around $\{1,0,0\}$, we simply do

$$v \cdot \text{RotationMatrix}(\theta, \{1,0,0\}) \tag{4}$$

Where the $\cdot$ operator is standard matrix multiplication. Here is another example of an expanded rotation matrix:

$$\text{RotationMatrix}(\theta, \{0,0,-1\}) = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5}$$

4.2 Without base component

The first step to solving the forward kinematics is to consider what would happen if we had no base at all. How does this simplify our situation? Well, we no longer have an offset for $r_{shoulder}$ and we also no longer have any element of θ_{base}. The parameters that remain are $\theta_{shoulder}$ and θ_{elbow}. Given these parameters, we can calculate $r'_{wrist} - r'_{shoulder}$ with:

$$r'_w - r'_s = \text{RotationMatrix}(\theta_e + \theta_s, \{-1,0,0\}) \cdot \overrightarrow{r_e r_w} + \text{RotationMatrix}(\theta_s, \{-1,0,0\}) \cdot \overrightarrow{r_s r_e} \tag{6}$$

The prime notation is to designate that these values are not actually r_w and r_s, but what these values would be without any base component. After we have found $r'_w - r'_s$, we can incorporate the base component using a process described in the next section.

4.3 With base component

We can incorporate base rotation with the following equation:

$$(r_w - r_s) = (r'_w - r'_s) \cdot \text{RotationMatrix}(\theta_{base}, \{0, 0, -1\}) \tag{7}$$

And finally to incorporate the base offset, or shoulder position r_s:

$$r_w = (r_w - r_s) + r_s \tag{8}$$

At this point we have found the final r_{wrist} using only our θ values and our predefined constants, thus completing our solution of the forward kinematics.

5 Solve for reverse kinematic equations, eliminating multiple solutions

Not surprisingly, in order to solve for the reverse kinematic equations, we should take the forward kinematic process and move backwards. This time, we will start with the desired $r_{wrist} = \{x, y, z\}$ and solve for the $\theta = \{\theta_{base}, \theta_{shoulder}, \theta_{elbow}\}$ values.

5.1 Removing base component

The last process in the forward kinematics process was to incorporate the base component. Our first step in the reverse kinematics process will be to remove this base component. The base component includes the $r_{shoulder}$ and θ_{base}.

A crucial requirement for solving for the reverse kinematic equation is to understand that the base rotation can be solved for without any complicated math. If our destination point is directly in front of us, we know that our base angle will need to be zero. We can calculate our base angle by using only x and y because z does not matter for us. Of course, if x and y are both zero, then we can arbitrarily set the base angle. Here is our equation for the base angle:

$$\theta_{base} = -\left(\tan^{-1}(x - x_s, y - y_s) - \frac{\pi}{2}\right) \tag{9}$$

Notice that we subtract the (x, y) position of the shoulder before applying $\tan^{-1}$. We use a sign change and a $\pi/2$ offset to get the ranges that we expect. Our next step is to remove this rotation from the input by rotating our input by θ_{base} back towards the center.

$$r'_w = r'_s = \text{RotationMatrix}(-\theta_{base}, \{0, 0, -1\}) \cdot (r_w - r_s) \tag{10}$$

Notice that this is essentially the same as Equation 7, only with the prime notation switched and the negation of θ_{base}. We are undoing the changes made before. Here is an expanded version of the rotation matrix:

$$r'_w - r'_s = \begin{pmatrix} \cos(\theta_{base}) & -\sin(\theta_{base}) & 0 \\ \sin(\theta_{base}) & \cos(\theta_{base}) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot (r_w - r_s) \tag{11}$$

The x component of $(r'_w - r'_s)$ will always be zero, since removing the base rotation and offset will remove any x component.

After removing the base component, we have a 2D problem where we need to find θ_s and θ_e. Here is a diagram of the perspective we will consider:

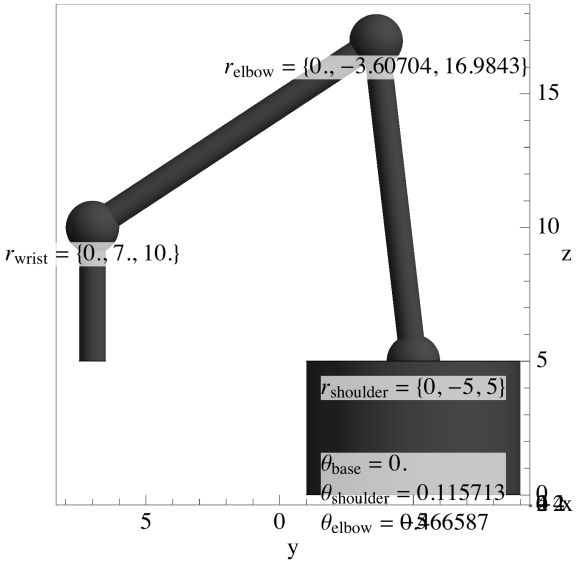


Figure 8: Arm from the side, simulating the removal of the base component.

5.2 Solving for θ_s and θ_e

Our next step is to solve for the θ_s and θ_e present in Equation 6. Here is Equation 6:

$$r'_w - r'_s = \text{RotationMatrix}(\theta_e + \theta_s, \{-1, 0, 0\}) \cdot \overrightarrow{r_e r_w} + \text{RotationMatrix}(\theta_s, \{-1, 0, 0\}) \cdot \overrightarrow{r_s r_e}$$

We've already found $(r'_w - r'_s)$ from Equation 10. $\overrightarrow{r_e r_w}$ and $\overrightarrow{r_s r_e}$ are just constant vectors defined in Section 2.1. Let us say $h = |\overrightarrow{r_s r_e}|$ and $u = |\overrightarrow{r_e r_w}|$. Here is Equation 6 in expanded form:

$$(r'_w - r'_s)_x = 0 \tag{12}$$
$$(r'_w - r'_s)_y = h\sin(\theta_s) + u\cos(\theta_e + \theta_s) \tag{13}$$
$$(r'_w - r'_s)_z = h\cos(\theta_s) - u\sin(\theta_e + \theta_s) \tag{14}$$

What we really want to do is to solve for θ_s and θ_e. We should now have enough information to solve for our target variables. This is not a linear system of equations, and there will not be a

unique solution for many values. There will not be a solution at all for some target positions. An example of this is attempting to reach out of our range. No matter what we set θ_e and θ_s to, we will never be able to reach further than $(h + u)$ cm away from our shoulder.

If we put this system of equations into Mathematica and instruct it to solve for θ_s and θ_e, we get the following result:

$$a = \sqrt{-h^2y^2\left(u^4 - 2u^2\left(h^2 + y^2 + z^2\right) + \left(-h^2 + y^2 + z^2\right)^2\right)}$$

$$\theta_s = \tan^{-1}\left(\frac{-u^2hz + a + h^3z + hy^2z + hz^3}{h^2\left(y^2 + z^2\right)}, \frac{-u^2hy^2 - za + h^3y^2 + hy^4 + hy^2z^2}{h^2y\left(y^2 + z^2\right)}\right)$$

$$\theta_e = \tan^{-1}\left(\frac{a}{uh^2y}, \frac{u^2 + h^2 - y^2 - z^2}{uh}\right)$$

In the above equations, $y = (r'_w - r'_s)_y$ and $z = (r'_w - r'_s)_z$. For the final result listed above, I have removed some extraneous solutions that are not applicable to our use case. For example, we do not care that we can add an arbitrary $k \cdot 2\pi$ to our θ values.

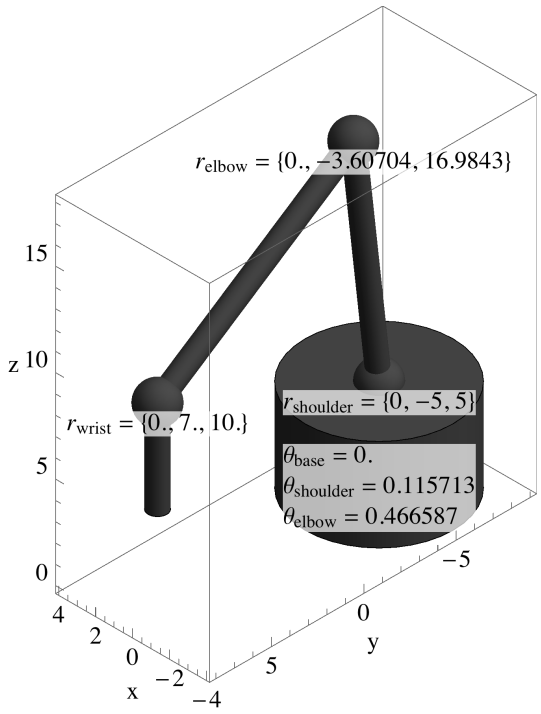
At this point we have found $\theta = \{\theta_{base}, \theta_{shoulder}, \theta_{elbow}\}$ using only r_{wrist} as an input, meaning we have solved for the reverse kinematics equations for our arm.

6 Testing using forward kinematic equations

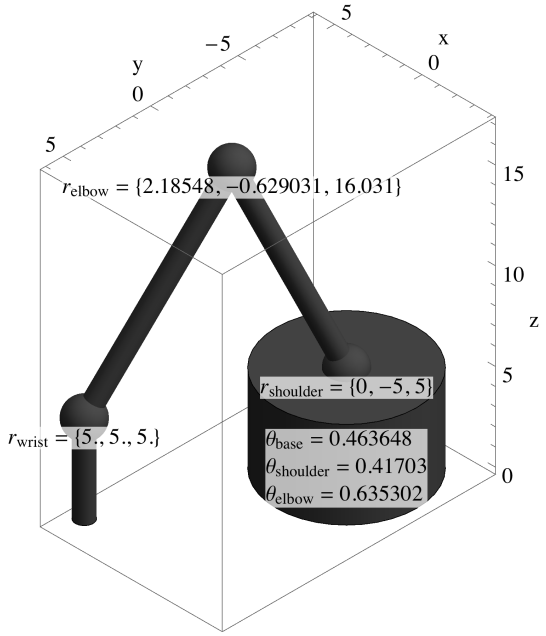
In order to test the reverse kinematic equations, I can plug the result of the forward kinematics directly into the reverse kinematics to ensure that I get the same result as an output. For example, I can ensure that:

$$\text{revkin}(\text{fwdkin}(\{.5, 0, -1\})) = \{.5, 0, -1\}$$

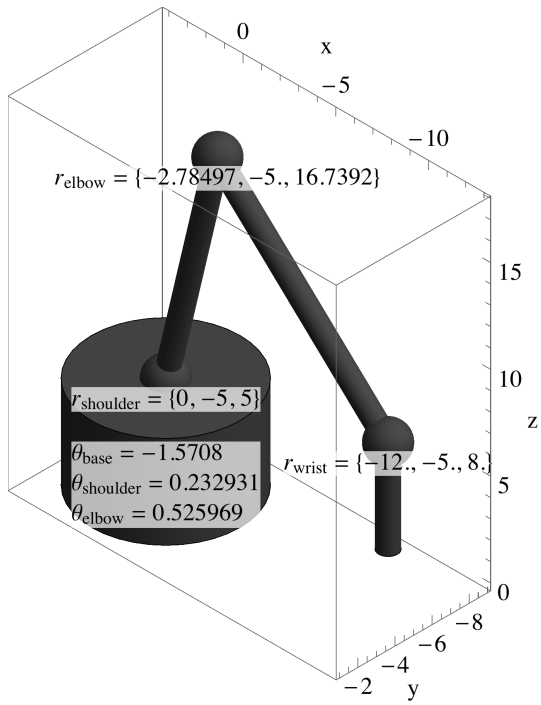
I can also run the reverse kinematics equations on some test values to ensure that the θ values make sense.



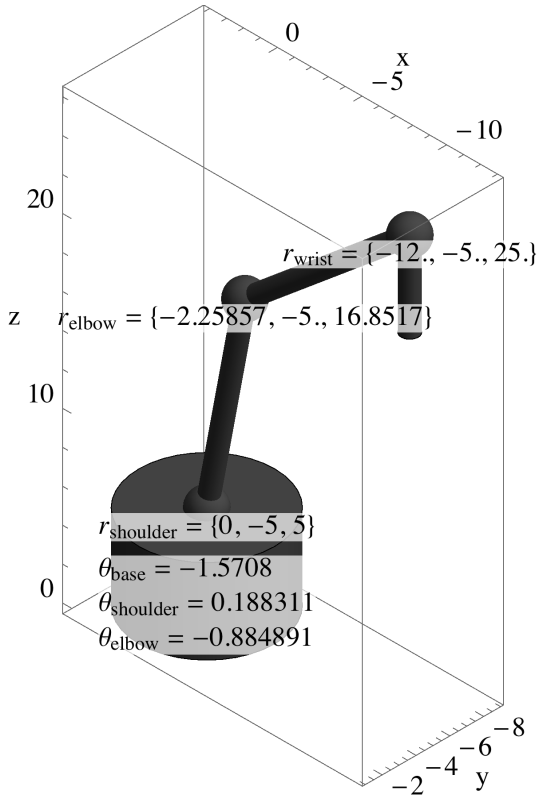
(a) Calculated θ and pose for $r_{wrist} = \{0, 7, 10\}$



(b) Calculated θ and pose for $r_{wrist} = \{5, 5, 5\}$



(a) Calculated θ and pose for $r_{wrist} = \{-12, -5, 8\}$



(b) Calculated θ and pose for $r_{wrist} = \{-12, -5, 25\}$

7 Integration with robot and calibration, interpolation

In order for the reverse kinematics to run on the robot, I copied and pasted the equations for θ above into a Python module and ran some regex substitutions to convert the Mathematica syntax into Python. The results of this are listed below in the Appendix.

Knowing the values for θ is not enough for the code to properly run on the robot. The servos in the arm expect PWM values (set through values ranging 0-255) and the actual radians that certain values correspond to depends on the mounting of the horn and the throw of the servo arms. I created a mapping for each arm servo that mapped radian values to servo PWM values. I could then feed the θ values through that function to get three servo values as an output.

After testing the inverse kinematics on the actual arm, I found that the arm was moving accurately to the (x, y, z) position that I was telling it to go to. This meant that my reverse kinematics equations were doing the job.

My final step was to create interpolation code to handle smooth movement. If the arm's servo values are currently $(10, 206, 100)$ and we wish to go to $(10, 205, 250)$, I would use a sigmoid-based transition that lasts t seconds in order to smoothly transition from the start to the end destination.

8 Appendix

Listing 1: ../kinematics.py

```
from Vec3d import Vec3d
from math import atan2, pi, degrees, sqrt

shoulderToElbow = 4.75 * 2.54
elbowToWrist = 5.00 * 2.54
shoulderPos = Vec3d(0, -4.6, 7)


def fwdkin(rot):
    humerus = Vec3d(0, 0, shoulderToElbow)
    forearm = Vec3d(0, elbowToWrist, 0)
    baseTheta = rot[0]
    shoulderTheta = rot[1]
    elbowTheta = rot[2]
    pos = humerus.rotated_around_x(-degrees(shoulderTheta))
    pos += forearm.rotated_around_x(-degrees(elbowTheta + shoulderTheta))
    pos.rotate_around_z(-degrees(baseTheta))
    pos += shoulderPos
    return pos


def revkin(pos):
    npos = pos - shoulderPos
    # Atan2 parameters are in the reverse order from Mathematica
    baseTheta = -(atan2(npos[1], npos[0]) - pi / 2)
    npos = npos.rotated_around_z(degrees(baseTheta))
    y = npos[1]
    z = npos[2]

    shoulderTheta = atan2(
    (1 * (-(elbowToWrist ** 2 * shoulderToElbow * y ** 2) +
    shoulderToElbow ** 3 * y ** 2 + shoulderToElbow * y ** 4 +
    shoulderToElbow * y ** 2 * z ** 2 -
    z * sqrt(-(shoulderToElbow ** 2 * y ** 2 * (elbowToWrist ** 4 +
    (-shoulderToElbow ** 2 + y ** 2 + z ** 2) ** 2 -
    2 * elbowToWrist ** 2 * (shoulderToElbow ** 2 + y ** 2 +
    z ** 2)))))) / (shoulderToElbow ** 2 * y *
    (y ** 2 + z ** 2)),
    (1 * (-(elbowToWrist ** 2 * shoulderToElbow * z) +
    shoulderToElbow ** 3 * z + shoulderToElbow * y ** 2 * z +
    shoulderToElbow * z ** 3 +
    sqrt(-(shoulderToElbow ** 2 * y ** 2 * (elbowToWrist ** 4 +
    (-shoulderToElbow ** 2 + y ** 2 + z ** 2) ** 2 -
    2 * elbowToWrist ** 2 * (shoulderToElbow ** 2 + y ** 2 +
    z ** 2)))))) / (shoulderToElbow ** 2 * (y ** 2 + z ** 2)))

    elbowTheta = atan2(
    (elbowToWrist ** 2 + shoulderToElbow ** 2 - y ** 2 - z ** 2) /
    (elbowToWrist * shoulderToElbow),
    sqrt(-(shoulderToElbow ** 2 * y ** 2 * (elbowToWrist ** 4 +
```

```
    (-shoulderToElbow ** 2 + y ** 2 + z ** 2) ** 2 -
    2 * elbowToWrist ** 2 * (shoulderToElbow ** 2 + y ** 2 +
    z ** 2)))) / (elbowToWrist * shoulderToElbow ** 2 * y))

    return Vec3d(baseTheta, shoulderTheta, elbowTheta)

# print fwdkin(Vec3d(.5, 0, -1))
# print revkin(Vec3d(3.28974, 1.02183, 27.7517))
```

Listing 2: ../arm.py

```
from __future__ import print_function

from math import pi, exp
import time
import operator
import logging

from head.spine.kinematics import revkin
from head.spine.Vec3d import Vec3d
from head.imaging.block_detector import block_detector

logger = logging.getLogger(__name__)

wristToCup = 10  # Distance in centimeters from wrist center to cup tip

# Servo configuration
BASECENTER = 85  # more positive moves to the right
BASERIGHT = BASECENTER + 85
BASELEFT = BASECENTER - 85


def base_r2p(r):
    return BASECENTER + (r / (pi / 2)) * (BASERIGHT - BASECENTER)
SHOULDERCENTER = 95  # more positive moves backward
SHOULDERDOWN = SHOULDERCENTER - 79


def shoulder_r2p(r):
    return SHOULDERCENTER + (r / (pi / 2)) * (SHOULDERDOWN -
        ↪ SHOULDERCENTER)
ELBOWCENTER = 126  # more positive moves down
ELBOWUP = ELBOWCENTER - 90


def elbow_r2p(r):
    return ELBOWCENTER - (r / (pi / 2)) * (ELBOWUP - ELBOWCENTER)

WRISTCENTER = 30 + 82  # more positive flexes up
WRISTDOWN = WRISTCENTER - 82


def wrist_r2p(r):
    return WRISTCENTER - (r / (pi / 2)) * (WRISTDOWN - WRISTCENTER)
```

```
# Probably no calibration needed
WRISTROTATECENTER = 90
SUCTIONCENTER = 90

# Starts at base
PARKED = [180, 170, 180, 60, 180]

# CENTER is defined as 0 radians


def to_servos(cuppos, wrist, wristrotate):
    wristpos = cuppos + Vec3d(0, 0, wristToCup)
    rot = revkin(wristpos)
    wrist += -pi / 2
    # If positive wrist flexes up, positive shoulder and elbow rotations
       ↪ will
    # add directly to wrist
    wrist += rot[1] + rot[2]
    servo = [base_r2p(rot[0]), shoulder_r2p(rot[1]), elbow_r2p(rot[2]),
             wrist_r2p(wrist), wristrotate]
    servo = [max(int(round(p)), 0) for p in servo]
    # print servo
    return servo


def interpolate(f, startargs, endargs, seconds, smoothing):
    def linear(x):
        return x

    def rawsigmoid(a, x):
        return 1 / (1 + exp(-(x - .5) / a))

    def sigmoid(a, x):
        return (rawsigmoid(a, x) - rawsigmoid(a, 0)) / (rawsigmoid(a, 1) -
           ↪  rawsigmoid(a, 0))
    start_time = time.time()
    difference = map(operator.sub, endargs, startargs)
    curr_time = time.time()
    iters = 0
    while (curr_time - start_time) < seconds:
        elapsed = curr_time - start_time
        fraction = elapsed / seconds
        if smoothing == 'linear':
            def sfunc(x):
                return linear(x)
        elif smoothing == 'sigmoid':
            def sfunc(x):
                return sigmoid(0.13, x)
        toadd = [v * sfunc(fraction) for v in difference]
        currargs = map(operator.add, startargs, toadd)
        f(*currargs)
        curr_time = time.time()
        iters += 1
```

```
    logger.info("Arm interpolation iterations: %d", iters)
    f(*endargs)


class get_arm:

    def __init__(self, s):
        self.s = s

    def __enter__(self):
        self.arm = Arm(self.s)
        return self.arm

    def __exit__(self, type, value, traceback):
        for i in range(2):
            for devname, port in self.s.ports.iteritems():
                self.s.ser[devname].flushOutput()
                self.s.ser[devname].flushInput()
            time.sleep(0.1)
        self.arm.park()


class Arm(object):

    def __init__(self, s):
        self.s = s
        self.servos = PARKED

    # Wrist is the amount of up rotation, from straight down, in radians
    # cuppos assumes that wrist is set to 0 radians
    # This is a raw function - use move_to instead
    def set_pos(self, cuppos, wrist, wristrotate):
        self.servos = to_servos(cuppos, wrist, wristrotate)
        self.s.set_arm(self.servos)

    def set_servos(self, *args):
        self.servos = args
        self.s.set_arm(list(args))

    def move_to(self, cuppos, wrist, wristrotate, seconds=1, smoothing='
       ↪ sigmoid'):
        '''Wrist measurement is in radians!!!'''
        assert wrist < pi
        startargs = self.servos
        endargs = to_servos(cuppos, wrist, wristrotate)
        interpolate(lambda *args: self.set_servos(*args),
                    startargs, endargs, seconds, smoothing)
        # interpolate(lambda *args: print(repr(args)), startargs, endargs,
           ↪  seconds, smoothing)

    def move_to_abs(self, servopos, seconds=1, smoothing='sigmoid'):
        startargs = self.servos
        endargs = servopos
        interpolate(lambda *args: self.set_servos(*args),
```

```
                    startargs, endargs, seconds, smoothing)

    def park(self, seconds=2):
        self.move_to_abs(PARKED, seconds)
        self.s.detach_arm_servos()

    def detect_blocks(self, level):
        bd = block_detector(self.s)
        # First move away from rails
        self.move_to(Vec3d(11, -1, 10), 0, 180)
        self.move_to(Vec3d(-3.5, 4, 17), 0.04 * 3.14, 180)
        bd.grab_left_frame()
        self.move_to(Vec3d(3, 4, 17), 0.04 * 3.14, 180)
        bd.grab_right_frame()
        istop = level == 'top'
        return bd.get_blocks(top=istop, display=False)
```