Robot Arm Documentation

QUT Robotics Club: Workshop Robot Arm Project

Written by Andrew Razjigaev - President for 2018

Email: qutroboticsclub@gmail.com Website: https://qutrobotics.com/

TABLE OF CONTENTS

OVERVIEW OF THE DESIGN

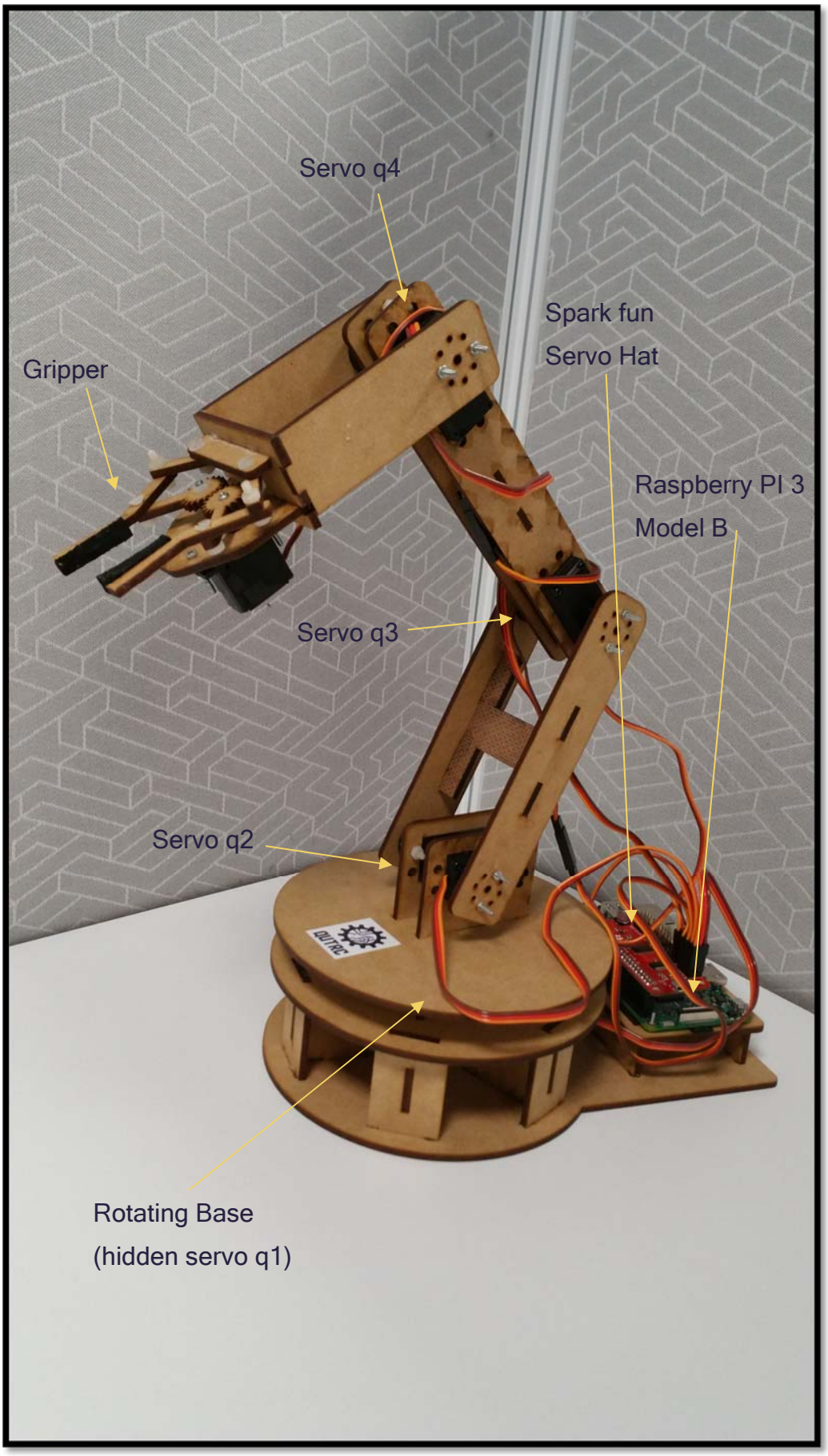
Purpose of Robot Arm Project

The workshop is inspired by the Introduction to Robotics subject in the Robotics minor. From the view of some students, the content was great but it lacked practical understanding or the problem-solving with the challenges in the application of that theory. As a result, people lose the confidence to fully understand the theory and struggle with subjects like Advanced robotics and in other project work in third and fourth year engineering. Therefore, the workshop is meant to address this gap by providing a practical experience that would provide an advantage for students before attempting Introduction to Robotics.

The Systems and features of the Robot Arm Design

The robot arm consists of a few systems:

- **The Mechanical Design**: The robot arm consists of 4 revolute joints (RRRR), 3 planar joints with 1 rotation base. All of these joints are physical servos of the same model. It also contains a gripper with a micro servo that actuates a four-bar linkage gripper mechanism. The gripper has foam to improve its grasping ability. The structure consists of laser-cut MDF or HDF and a variety of M2 and M3 screws that connects the servos to the structure.
- **The Electrical Design**: The arm has a Raspberry PI 3 model B with a Spark fun Raspberry PI Servo hat. The hat is a red PCB that can manage up to 16 servos using the i2c communication line with the PI. Power is delivered from a charging source to the wall. HDMI is used to display the Raspberry PI desktop and programming environment onto a monitor and mice and keyboards are used to interact with the PI operating system.
- **The Software Design**: The software consists of three Python modules that support an overall control code. They are the Servo Module, the Kinematics module and the Utilities module. All servo control functions are in the Servo module while all Kinematics equations are functions in the Kinematics module. The Utilities module manages other aspects like the keyboard inputs and functions that control the servos as an overall arm.


Servo q4
Spark fun
Servo Hat
Gripper
Raspberry PI 3
Model B
Servo q3
Servo q2
Rotating Base
(hidden servo q1)

MOTOR CONTROL

About the Servos

The primary servos used for the robot arm segments are the model Turnigy TGY-S901D purchased from Hobby King: https://hobbyking.com/en_us/turnigytm-tgy-s901d-ds-mg-robot-servo-13kg-0-14sec-58g.html?wrh_pdp=6

Servo Specifications:

Model: TGY-S901D

Operating Voltage: 6.0V / 7.2V

Operating Speed: 0.16sec.60º/ 0.14sec.60º

Stall Torque: 12.5kg.cm /13kg.cm

Size: 40.8X20.1X37.6mm

Weight: 58g

Motor: Coreless Type

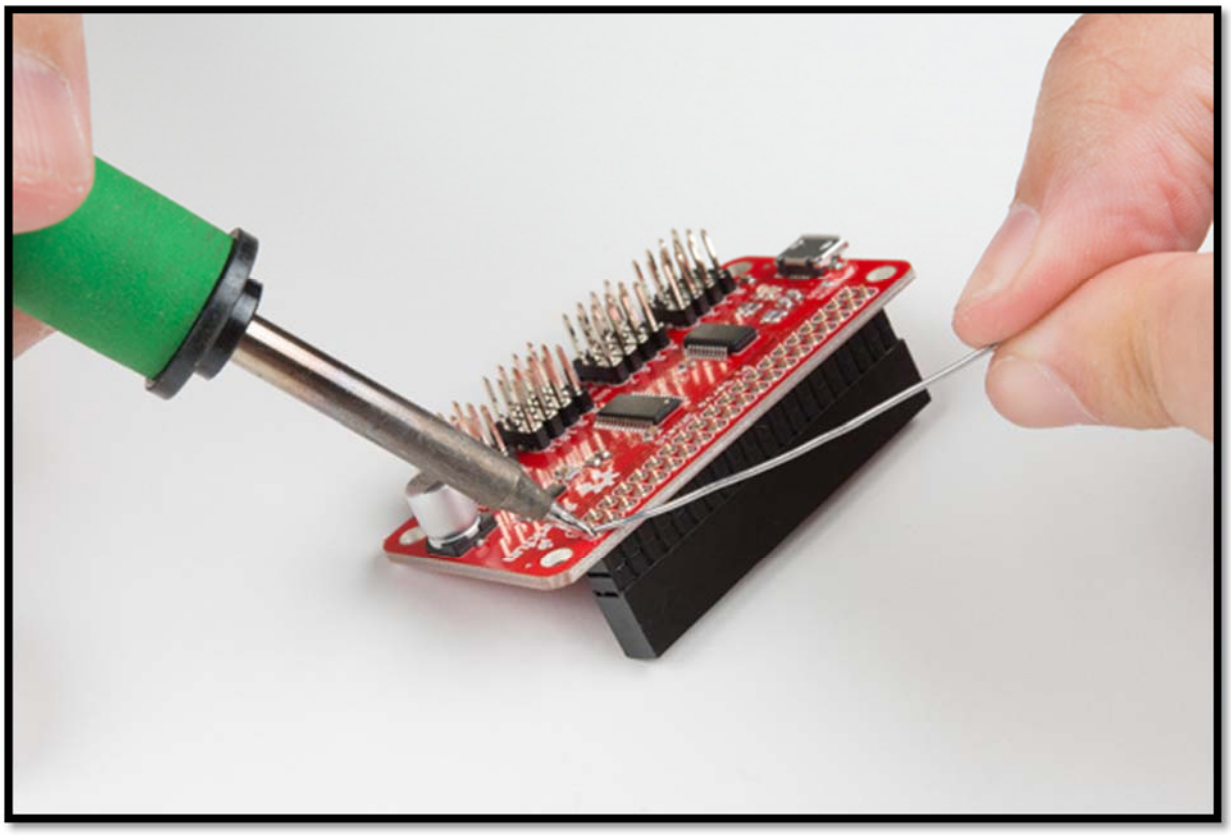
Ball bearing: 2BB

Gear: Metal

Plug: JR Type

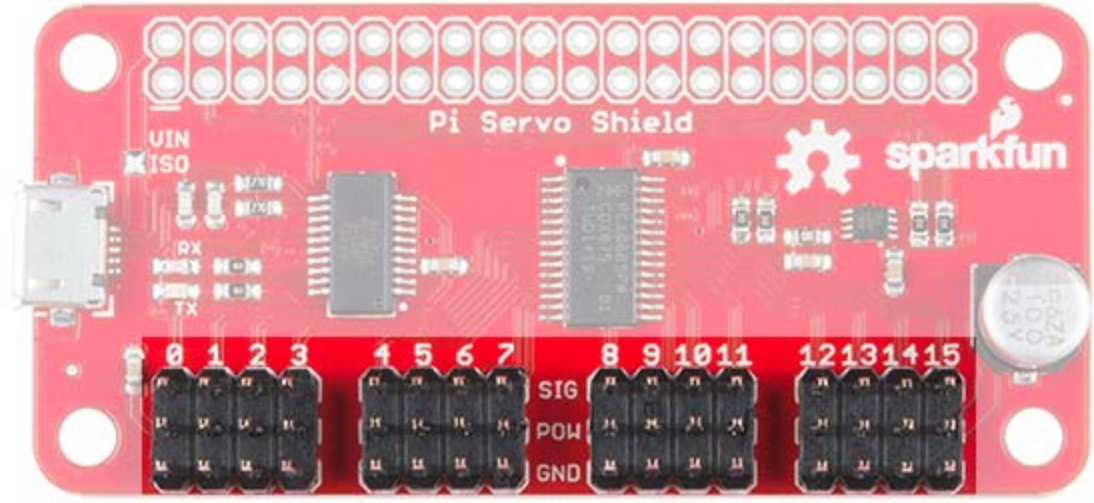
Spline count: 25

About the Servo HAT



Each servo is controlled using a Spark fun Servo hat described by the hookup guide: https://learn.sparkfun.com/tutorials/pi-servo-hat-hookup-guide#introduction

The SparkFun Pi Servo Hat allows your Raspberry Pi to control up to 16 servo motors via I2C connection. It requires soldering the female headers onto the PCB, the headers allow the PCB to be stacked onto the Raspberry PI pins. Each servo connects to the channels 0 to 15 where the servo connection is hooked up as so: the yellow cable connects to the signal line, the red cable to the power line and the brown cable to the ground line.



Once the servos and hat are connected to the PI, it is possible to then run python or even C++ code (according to Spark fun) to control the servos. For the workshops, all the Raspberry PIs are programmed with python and are accessed directly using HDMI to a monitor and a keyboard and mouse to program with. The PI uses the NOOBS SD card to boot the Desbian distribution of Raspbian (the Operating System). When giving the PI power, the desktop appears and it is possible to open and edit a python script and run code from the top left menu icon.

Low-Level Servo Control

Provided that i2c is enabled in PI-configuration, you may run code that uses the i2c communication which is the SMBus module. To control the servos first import that module and declare a bus connection.

- import smbus
- bus = smbus.SMBus(1)

Then provide the part address for the connection to go to which is 0x40 by default:

- addr = 0x40

Then initialize the settings for the communication by enabling the PWM chip and tell it to automatically increment addresses after a write (that lets us do single-operation multi-byte writes)

- bus.write_byte_data(addr, 0, 0x20)
- bus.write_byte_data(addr, 0xfe, 0x1e)

With this initialized, now the servos can be told to move to certain positions via Pulse Width Modulation (a method of changing the power delivered to a motor by changing the average voltage in the pulses). The yellow cable of the servo connector carries the control signal, used to tell the motor where to go. This control signal is a specific type of pulse train. Servo motors get their control signal from that pulse width. We can write commands to the servo that tell how big that pulse is, for example:

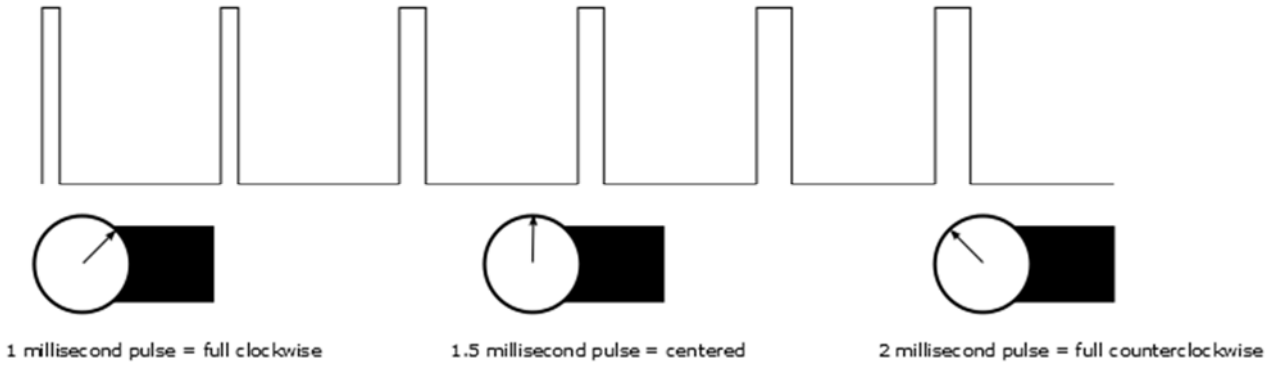
- bus.write_word_data(addr, 0x06, 0)
- bus.write_word_data(addr, 0x08, 1250)

The first write is to the “start time” register for channel 0 (indicated by using 0x06). By default, the PWM frequency of the chip is 200Hz, or one pulse every 5ms. The start time register determines when the pulse goes high in the 5ms cycle. All channels are synchronized to that

cycle. Generally, this should be written to 0 i.e. the start of the period should be high. This only needs to be done once.

The second write is to the “stop time” register, and it controls when the pulse should go low (indicated by using 0x08 for channel 0). The range for this value is from 0 to 4095, and each count represents one slice of that 5ms period (5ms/4095), or about 1.2us. Thus, the value of 1250 written above represents about 1.5ms of high time per 5ms period.

Generally speaking, a pulse width of 1.5ms yields a “neutral” position, halfway between the extremes of the motor’s range. 1.0ms yields approximately 45 degrees off center, and 2.0ms yields -45 degrees off center. In practice, those values may be slightly more or less than 45 degrees, and the motor may be capable of slightly more or less than 45 degrees of motion in either direction. Common servos rotate over a range of 90° as the pulses vary between 1 and 2ms – they should be at the center of their mechanical range when the pulse is 1.5ms:



https://learn.sparkfun.com/tutorials/hobby-servo-tutorial#servo-motor-background

To achieve 1ms pulses, use a stop time of 836, likewise, to get 2ms pulses use a stop time of 1664. Using the module time to pause the code so that the servo has time to adjust to its position:

- time.sleep(2) # pause for two seconds
- bus.write_word_data(addr, 0x08, 836) # chl 0 end time = 1.0ms
- time.sleep(2)
- bus.write_word_data(addr, 0x08, 1664) # chl 0 end time = 2.0ms

Essentially, the servo position is controlled by the stop time of the pulse of the PWM which is a value that ranges from 0 (no power) to 4095 (Full power). Keeping it safe, the range used is from 836 to 1664 which allows a good 90 degrees of movement (acceptable for the arm to pick and place objects in its specific workspace).

See Appendix A for the start and stop addresses for using other channels on the PI hat.

For channel 0 in the example, the start address is 0x06 and the stop address is 0x08.

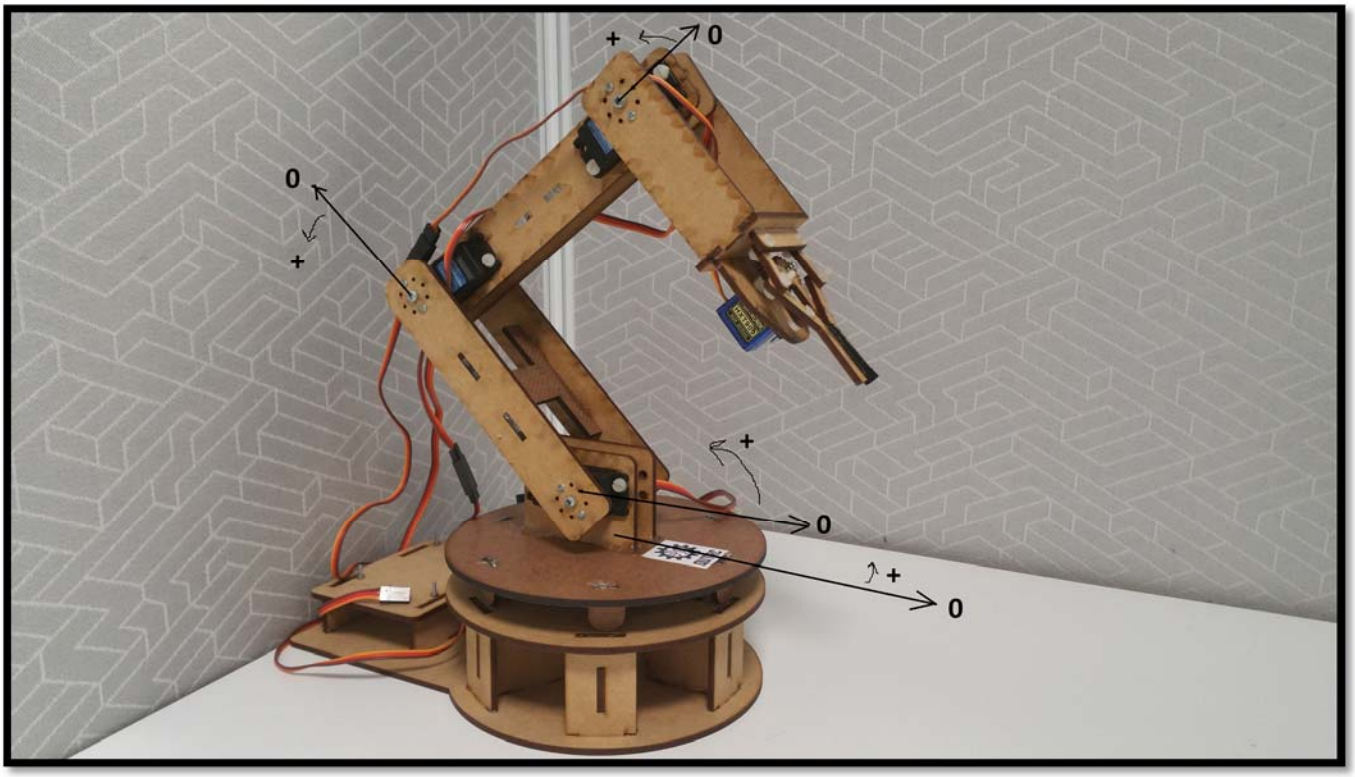
High Level Control

The low level control showed that the position of the servo can by controlled by PWM pulses governed by integers from the range of 0 to 4095 with a mechanical working range of 836 to 1664. For the interest of understanding the kinematics of the arm, it is necessary to map that PWM to physical degrees or radians of rotation. For that, calibration is done where one should measure the angle output from each joint that undergoes the PWM values of 836, 1250 and 1664. The relationship is proportional and hence a linear relationship is required to map PWM to angle output.

Using the calibration data, one can model the servo as a class object in python that has variables like joint limits and functions that can move the servo to a certain degrees or radians or read the current position of the joint. The functions for the PI servo class is in the Robotarm_ServoModule.py file in GitHub:

Measuring Convention of the Joints of the Arm:

To measure the joints consistently for each arm, the standard for placing the origin for the angles is proposed based on the right hand rule about the axis of rotation and the coordinate frames later on. To get the idea, the robot in the picture has the revolute base at 0, the next joint is at 135, the joint above that is at -100 and the joint above that one is at -90 degrees.



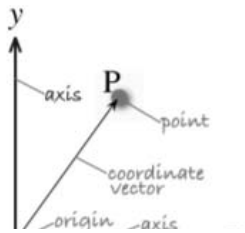
KINEMATIC MODELLING

The next two sections refer to Peter Corke's Textbook [1]. His content is also available as videos in the Robot Academy website: https://robotacademy.net.au/

Understanding 3D space using Homogeneous Transforms

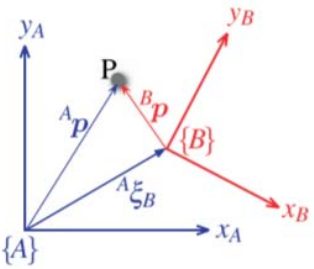
In robotics, we tend represent the world as a 3D Cartesian space where points are coordinate vectors with components on each axis:

$$^{A}P = [T_x \quad T_y \quad T_z]$$



Each point in space is measured from a Cartesian coordinate frame and has a letter on top that tells you what frame it's measured from.

A similar set-up is done when you have multiple coordinate frames. A transform from one frame to another is represented as a pose (position and orientation). The letter at the top left of it tells you the frame it's measuring from and the bottom right letter tells you where the transform goes to:



$$^{A}\xi_B = transform\ from\ frame\ A\ to\ frame\ B$$

This is useful when you want to find where one measured point is to another frame e.g.:

$$^{A}P = {}^{A}\xi_B\, {}^{B}P$$

This means that the transform from frame A to B of the point measured in frame B becomes the same point measured in frame A. This transform can be cascaded to transform from many more coordinate frames:

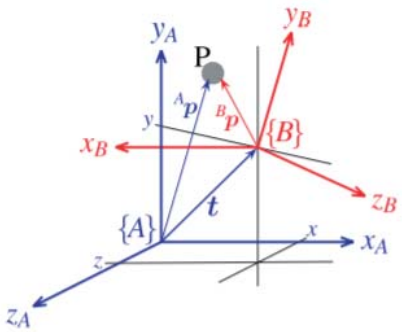
$$^{A}P = {}^{A}\xi_B\, {}^{B}\xi_C\, {}^{C}P$$

For the robot arm project we'll use this theory to find where the gripper is from the base:

$$^{Base}P_{Gripper} = {}^{Base}\xi_{Joint1}\, {}^{Joint1}\xi_{Joint2}\, {}^{Joint2}\xi_{Joint3}\, {}^{Joint3}P_{Gripper}$$

We can represent the transform as a matrix, like so:

$$^{A}\xi_B \rightarrow {}^{A}T_B = \begin{bmatrix} \dots & \dots & \dots & T_x \\ \dots & {}^{A}R_B & \dots & T_y \\ \dots & \dots & \dots & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

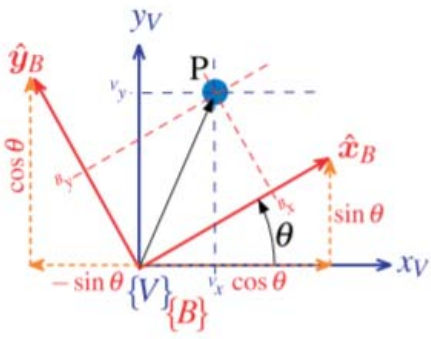


The matrix is 4 by 4 with the top left 3 by 3 being a 3D rotation Matrix and the forth column having the translational components $t = [T_x \quad T_y \quad T_z]$ of the transform in each axis. The bottom row is what makes it possible to use the transforms in this way:

$$^{Base}T_{Gripper} = {}^{Base}T_{Joint1} \times {}^{Joint1}T_{Joint2} \times {}^{Joint2}T_{Joint3} \times {}^{Joint3}T_{Gripper}$$

Where you can use matrix multiplication (dot product) to cascade the transforms.

To understand the rotation matrix think of it as an array of components of each axis. Note: every axis is a unit vector. Based on the figure to the right, you'll see that new axis is split into components measured on the old axis:



$$
{}^{V}R_{B} \rightarrow \begin{matrix} \\ along\ x_V \\ along\ y_V \\ along\ z_V \end{matrix} \overset{*}{\begin{matrix} x_B & y_B & z_B \end{matrix}} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}
$$

The matrix means that for some x distance along x_B is measured as that times $\cos\theta$ along x_V

And that for some y distance along y_B is measured as that times $-\sin\theta$ along x_V

And so on for each 3 components on each of the 3 axes … note that the z-axis in the figure is unchanged as theta rotates the frame about the z-axis, so therefore the z-axes components 'resemble' the identity matrix. A rotation matrix with no rotation is an identity matrix. All rotation matrix components must range between -1 and 1 and the determinant is always 1 to be a valid rotation matrix. A point can be rotated using matrix multiplication like this:

$$
R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}
$$

$$
R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}
$$

$$
R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}
$$

$$
{}^{V}P = {}^{V}R_{B}\,{}^{B}P
$$

$$
{}^{V}P = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{B}T_X \\ {}^{B}T_Y \\ {}^{B}T_Z \end{bmatrix}
$$

Rotation in 3D space means that a transform can not only translate in the x, y and z directions but rotate about the x, y and z axes. 3D rotation is commonly described by Euler angles roll pitch yaw. Rotation about the x-axis is roll, rotation about the y-axis is pitch and rotation about the z-axis is yaw.

$$
R = R_X(\theta_{roll}) \times R_Y(\theta_{pitch}) \times R_Z(\theta_{yaw})
$$

Putting this together a transformation matrix is used like this:

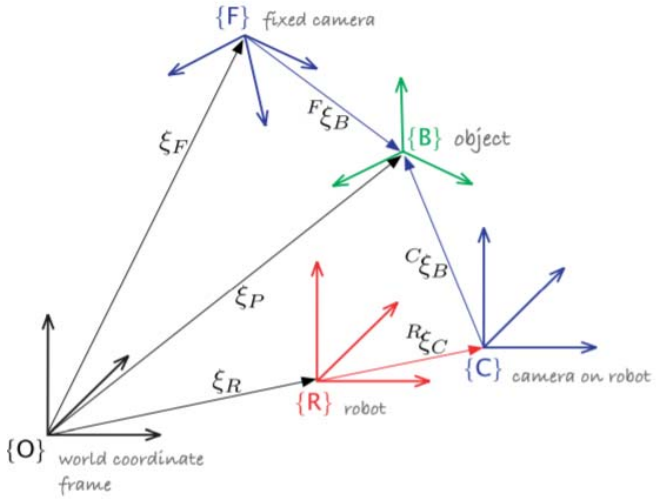
Let's say that ${}^{A}R_{B}$ is z-axis rotation: $R_Z(\theta)$ and translation vector is $t = [5, 10, -5]$

$$
{}^{A}P = {}^{A}T_{B}\,{}^{B}P
$$

$$
{}^{A}P = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 5 \\ \sin\theta & \cos\theta & 0 & 10 \\ 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{B}T_X \\ {}^{B}T_Y \\ {}^{B}T_Z \\ 1 \end{bmatrix}
$$

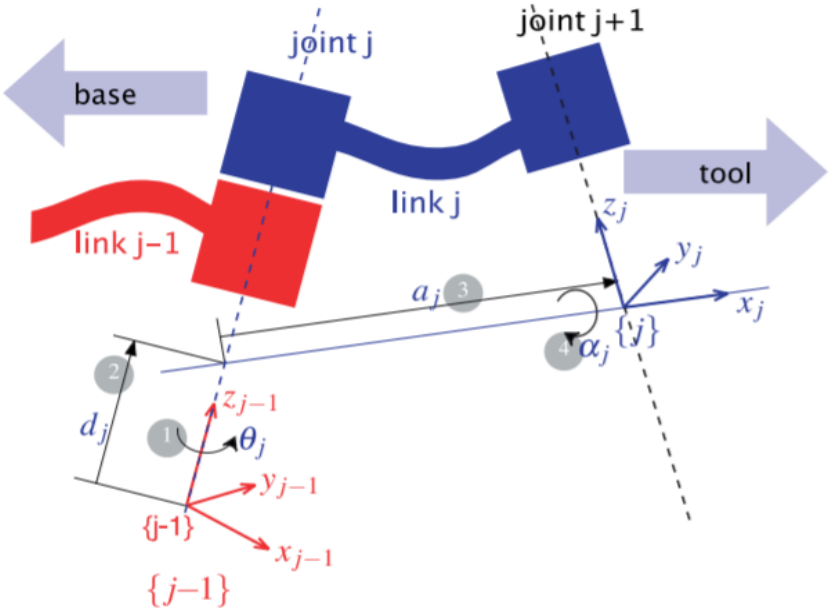
Which transforms the point ${}^{B}P$ to the same point but measured from the A frame

So overall, in 3D space, any object with a coordinate frame can be represented as a transform matrix from the world coordinate frame to it. The matrix consists of a 3 by 3 rotation matrix consisting of 3 parameters and a 3 by 1 translation vector consisting of 3 parameters. A transform is translation and rotation of an object in space based on 6 parameters: x, y, z, roll, pitch and yaw



DH Parameters in the Transform Matrix

In the previous section, the transform from one coordinate frame to another in 3D space is described by 6 parameters. Denavit and Hartenburg said that for 3D transforms between revolute joints (ones that rotate only in one axis) can be simplified to only need 4 parameters.



If you place the coordinate frames in a certain way (mainly placing the z axis as the axis of rotation and the x-axis as an axis to measure between axes), you can have the transform represented by only rotations and translations about the x and z axes:

$$^{j-1}\mathbf{A}_j(\theta_j, d_j, a_j, \alpha_j) = \mathbf{T}_{Rz}(\theta_j)\mathbf{T}_z(d_j)\mathbf{T}_x(a_j)\mathbf{T}_{Rx}(\alpha_j)$$

So from frame j-1 to frame j, the transform is described by parameters θ, d, a, α

This yields a standard transformation matrix that we can plug and use for every joint:

$$ {}^{j-1}\boldsymbol{A}_j = \begin{pmatrix} \cos\theta_j & -\sin\theta_j\cos\alpha_j & \sin\theta_j\sin\alpha_j & a_j\cos\theta_j \\ \sin\theta_j & \cos\theta_j\cos\alpha_j & -\cos\theta_j\sin\alpha_j & a_j\sin\theta_i \\ 0 & \sin\alpha_j & \cos\alpha_j & d_j \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

So every time you want to compute the transformation matrix among the joints, you need to find 4 parameters:

Joint angle	θ_j	the angle between the x_{j-1} and x_j axes about the z_{j-1} axis	revolute joint variable
Link offset	d_j	the distance from the origin of frame $j-1$ to the x_j axis along the z_{j-1} axis	prismatic joint variable
Link length	a_j	the distance between the z_{j-1} and z_j axes along the x_j axis; for intersecting axes is parallel to $\hat{z}_{j-1} \times \hat{z}_j$	constant
Link twist	α_j	the angle from the z_{j-1} axis to the z_j axis about the x_j axis	constant

Solving the Forward Kinematics

Forward kinematics is the problem of finding out where the gripper is from the base of the robot. It can be represented as a function K of joint values to gripper pose:

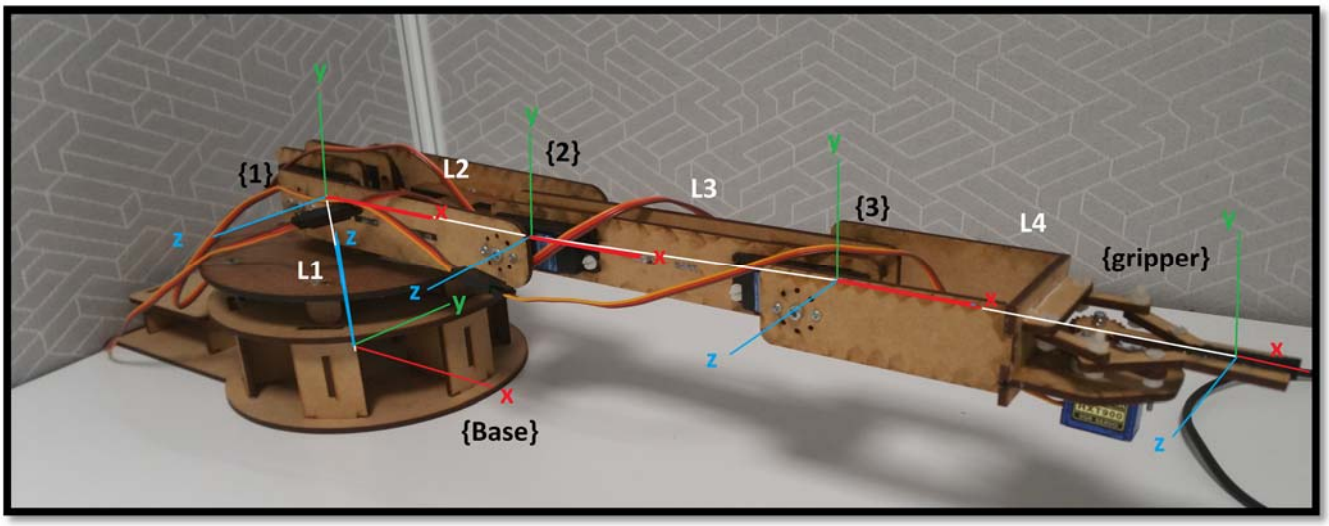
$$\boldsymbol{x} = K(\boldsymbol{q})$$

Where **x** is the tool point pose vector and **q** is the joint values vector of the robot arm

Finding the pose of the gripper from the base can be solved using the transformation matrices learnt before:

$$ {}^{Base}T_{Gripper} = {}^{Base}T_{Joint1} \times {}^{Joint1}T_{Joint2} \times {}^{Joint2}T_{Joint3} \times {}^{Joint3}T_{Gripper} $$

Using the transform matrix ${}^{Base}T_{Gripper}$ we can extract the overall translation. The next figure shows how the coordinate frames have been set to find the 4 DH parameters for each joint:



Note we will need the arm lengths:

They are: $L1 = 100mm, L2 = 120mm, L3 = 140mm, L4 = 140mm$

All translation is measured in mm

Each joint does a rotation in radians.

The variable angle produced by each joint is

$$q_1, q_2, q_3, q_4$$

Computing the DH parameters yields:

DH Parameters	$\boldsymbol{\theta}$	$\boldsymbol{d}$	$\boldsymbol{a}$	$\boldsymbol{\alpha}$
${}^{Base}T_{Joint1}$	q_1	$L1$	0	$\frac{\pi}{2}$
${}^{Joint1}T_{Joint2}$	q_2	0	$L2$	0
${}^{Joint2}T_{Joint3}$	q_3	0	$L3$	0
${}^{Joint3}T_{Gripper}$	q_4	0	$L4$	0

We then find the tool point coordinate by multiplying the matrices:

$${}^{Base}T_{Gripper} = {}^{Base}T_{Joint1} \times {}^{Joint1}T_{Joint2} \times {}^{Joint2}T_{Joint3} \times {}^{Joint3}T_{Gripper}$$

And extracting the translation vector out of ${}^{Base}T_{Gripper}$

The other value that is of interest to know is the grasping angle of the gripper (pitch from the base frame). That value in intuitively the sum of the joints:

$$grasp\ angle = \theta_P = q_2 + q_3 + q_4$$

From this, we have a way of knowing where the gripper is in the base frame using the servo joint angles. This is our forward kinematics function:

$$\boldsymbol{x} = K(\boldsymbol{q})$$

$$[x, y, z, \theta_P] = K([q_1, q_2, q_3, q_4])$$

We can find the tool point position and its grasping angle based on the given joint values. The kinematics module in the GitHub does this calculation nicely.

Since the pose outputs match the number of joint inputs, the robot arm is considered to be fully actuated. If we used less inputs than outputs, we'd have an under-actuated robot arm. If we use more inputs than outputs then we'd have an over-actuated robot arm. It is nicer to work with a fully actuated robot.

INVERSE KINEMATICS CONTROLLER

This inverse kinematics problem is that we want the robot to move the gripper to some location but we don't know the joint values. It's the opposite of the forward kinematics problem and has more useful applications in practice:

$$\boldsymbol{x} = K(\boldsymbol{q}) \quad is \quad [x, y, z, \theta_P] = K([q_1, q_2, q_3, q_4]) \quad \rightarrow \quad \boldsymbol{q} = K^{-1}(\boldsymbol{x})$$

This is a major challenge as it is difficult to invert the forward kinematics equations as each variable is interdepended on each other – in essence it's a lot of work but the velocity controller method allows a generic solution to be used to iteratively find the joint values q_1, q_2, q_3, q_4 that achieve the correct forward kinematics x, y, z, θ_P.

The Jacobian Velocity Relationship

The Jacobian is a matrix version of a multidimensional derivative:

$$J = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

It is found by taking the derivative of the forward kinematics using the chain rule:

$$\frac{d}{dt}\boldsymbol{x} = \frac{d}{dt}K(\boldsymbol{q})$$

$$\frac{d\boldsymbol{x}}{dt} = J(\boldsymbol{q}) \times \frac{d\boldsymbol{q}}{dt}$$

$$\dot{\boldsymbol{x}} = J(\boldsymbol{q})\, \dot{\boldsymbol{q}}$$

This relates the velocity of the tool point $\dot{\boldsymbol{x}} = [\dot{x}, \dot{y}, \dot{z}, \dot{\theta}_P]$ to the velocity of the joints: $\dot{\boldsymbol{q}} = [\dot{q_1}, \dot{q_2}, \dot{q_3}, \dot{q_4}]$ By a matrix of partial derivatives called the Jacobian $J(\boldsymbol{q})$. This matrix is usually really big and has lots of trigonometry computing sines and cosines of the current joint values $\boldsymbol{q} = [q_1, q_2, q_3, q_4]$. For our robot, we will use the discrete version for every while loop that takes about a constant Δt in time.

$$\Delta \boldsymbol{x} = J(\boldsymbol{q})\, \Delta \boldsymbol{q}$$

This Jacobian relationship looks like this:

$$\begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta\theta_P \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \frac{\partial x}{\partial q_3} & \frac{\partial x}{\partial q_4} \\ \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \frac{\partial y}{\partial q_3} & \frac{\partial y}{\partial q_4} \\ \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \frac{\partial z}{\partial q_3} & \frac{\partial z}{\partial q_4} \\ \frac{\partial \theta_P}{\partial q_1} & \frac{\partial \theta_P}{\partial q_2} & \frac{\partial \theta_P}{\partial q_3} & \frac{\partial \theta_P}{\partial q_4} \end{bmatrix} \begin{bmatrix} \Delta q_1 \\ \Delta q_2 \\ \Delta q_3 \\ \Delta q_4 \end{bmatrix}$$

Those partial derivatives were computed using the symbolic toolbox in MATLAB and written as a function in the kinematics module. All that code is available on the GitHub.

Velocity Controller to Solve the Inverse Kinematics

The point of the Jacobian matrix is that it provides us with a means of solving the inverse kinematics in real time. This is done by determining a velocity (change in position per while loop) to reach the desired gripper pose:

This can be considered as the tool point error:

$$\Delta \boldsymbol{x} = \boldsymbol{x}_{\boldsymbol{desired}} - \boldsymbol{x}_{\boldsymbol{current}}$$

Using the inverse of the Jacobian to lets you determine the joint velocity (the change in joint values per while loop).

$$\Delta \boldsymbol{q} = J^{-1}(\boldsymbol{q})\, \Delta \boldsymbol{x}$$

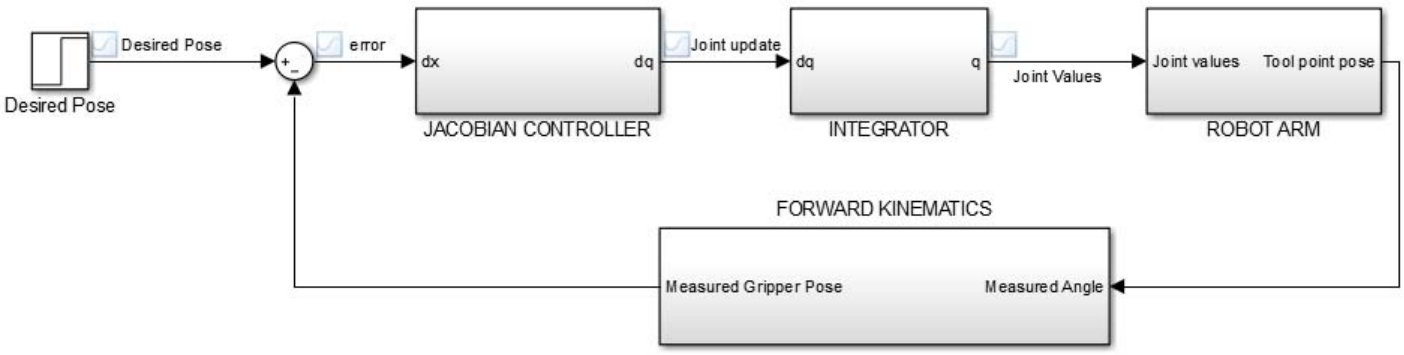
Integrating that in each time step gives the joint position:

$$\boldsymbol{q}_{\boldsymbol{updated}} = \boldsymbol{q}_{\boldsymbol{current}} + \Delta \boldsymbol{q}$$

This needs to be run over many while loops until $\boldsymbol{q}_{\boldsymbol{updated}}$ becomes $\boldsymbol{q}_{\boldsymbol{desired}}$ i.e.

$$\boldsymbol{q}_{\boldsymbol{desired}} = K^{-1}(\boldsymbol{x}_{\boldsymbol{desired}})$$

The controller can be visualized by this figure as an iterative process of forward kinematics and inverting the Jacobian:



A last note is that the velocity should have a speed limit per while loop as that keeps things safe by returning a smaller $\Delta \boldsymbol{q}$. You can do this by restricting the magnitude of $\Delta \boldsymbol{x}$:

$$\Delta \boldsymbol{x}_{\boldsymbol{safe}} = \boldsymbol{speedLimit} \times \frac{\Delta \boldsymbol{x}}{\boldsymbol{norm}(\Delta \boldsymbol{x})}$$

User input by keyboard

In the implementation of determining $\boldsymbol{x}_{\boldsymbol{desired}}$ in the controller is up to the user. For the workshop, the point $\boldsymbol{x}_{\boldsymbol{desired}}$ is updated and controlled using the keyboard. Using the getch module, instantaneous keypresses can be read and determined to see which parameter in $\boldsymbol{x}_{\boldsymbol{desired}}$ should be updated. In the utilities module on GitHub, this function is used to move the point $\boldsymbol{x}_{\boldsymbol{desired}}$ by 5mm in each x, y and z coordinate and by 5 degrees in the grasping angle.

Advanced Improvements to the Velocity Controller Method

This goes off-track from the Introduction to Robotics content but is interesting in a practical sense…

Although the Velocity Controller method works, it lacks a variety of practical aspects and poses a few problems:

- What do we do if the Jacobian has no inverse i.e. $det(J) = 0$
- What do we do if the solution of $\Delta \boldsymbol{q}$ actually drives the robot to a joint limit
- Is the solution the solver chooses optimal i.e. is it the minimum amount of $\Delta \boldsymbol{q}$ steps required to reach $\boldsymbol{x}_{\boldsymbol{desired}}$

There are variety of ways of improving the controller to optimize it and get it to consider things like joint limits. This idea of optimizing the controller is present in more advanced engineering control subjects.

The following improvements are given in the kinematics module as the damped least squares function. The damped least squares is a method of putting a line of best fit for the inverse Jacobian such that $\Delta \boldsymbol{q}$ is minimized and the joint limits are avoided as much as possible (Described by reference [2]). This method is used primarily in human body modelling of the kinematics in simulations and human body-like animations.

$$\Delta \boldsymbol{q} = J^T (J^T J + D(\lambda)^2 I)^{-1} \Delta \boldsymbol{x}$$

The new inverse is guaranteed to have a solution. The identity is in there to minimize the joint motion, $D(\lambda)$ is actually a penalty function matrix that gives ‘pain’ to the robot for moving close

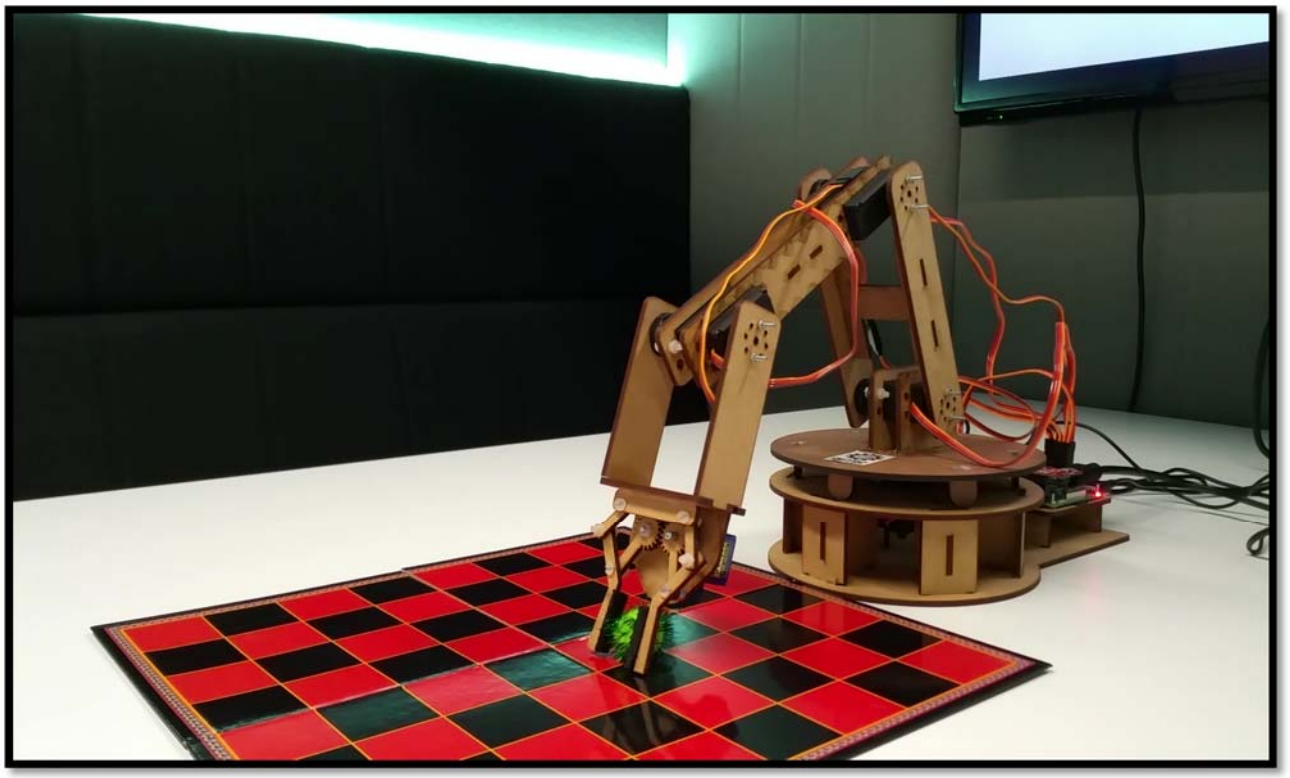
to its joint limit. It tries to minimize the quadratic penalty function as much as possible to stay 'comfortable' as it moves.

$$D(\lambda) = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 \\ 0 & 0 & 0 & \lambda_4 \end{bmatrix}$$

$$\lambda_i = c\left[\frac{2q_i - q_{imax} - q_{imin}}{q_{imax} - q_{imin}}\right]^p + w$$

The constants $c, p \; and \; w$ are weightings to tune the controller. By default, they are all 1 except $p = 2$ in the GitHub code.

Using this implementation, we can get a robot arm to automatically avoid joint limits, minimize its path of motion and solve the inverse kinematics which makes it nice as we can now just move the tool point and expect the robot to respond accordingly.



SUMMARY

Overall, the workshop robot arms have a variety of systems and features that makes an interesting demonstration for students throughout the semester. It covered how to use the Raspberry PI to control a bunch of servos. It covered how to use python to control the servos and how to model the arm mathematically. The workshops touch content form introduction to robotics such as DH parameters and kinematics. This gives students a heads up for the content in more challenging robotics subjects later on in their degrees.

Limitations or areas for future development:

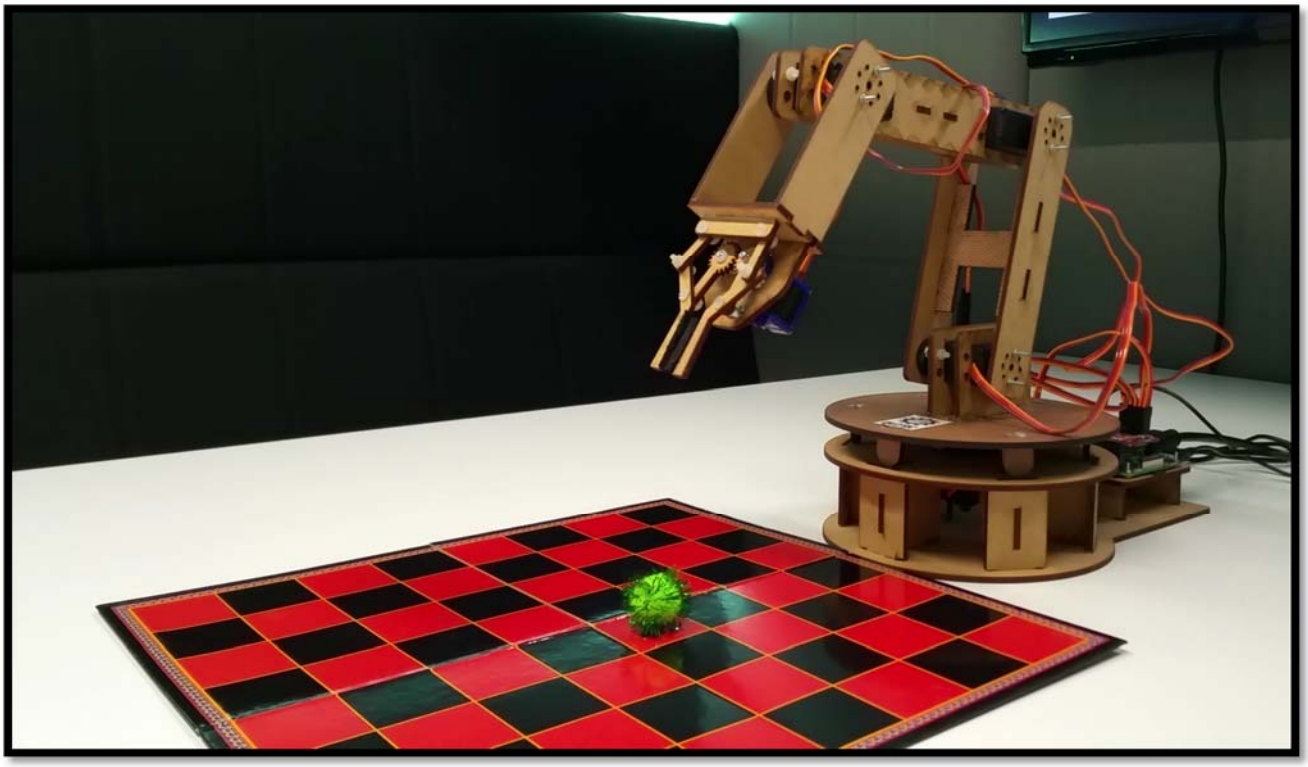
There are a few limitations with the current approach:

1. Gripper weakness, the gripper grasps weakly and micro servos tend to make too much noise alongside it.
2. Friction in the revolute base: the base may need ball bearings to improve its design
3. No battery supply: to remove the dependency on a power socket, we should implement remote power in the future. Not just for the arm but for other potential PI related projects in the future that would be mobile: e.g. hexapods, walking robots and camera based soccer robots
4. Keyboard input is a shaky way of controlling the desired pose. As the keyboard gives discrete steps, the velocity changes discontinuously making the robot arm jerk a bit. A better tool for controlling the robot may be explored later on
5. The robot lacks sensors. An interesting sensor to attach is a PI camera where one can do computer vision to decide what objects the robot can pick fully autonomously
6. Restrictions in remote control of the PI. As the nature of using Wi-Fi on campus, it becomes a bit trickier to get a group of PIs connected to the internet to communicate to the robots remotely in workshops.

REFERENCES:

[1] Corke, Peter. (2013) Robotics, Vision and Control: Fundamental Algorithms in MATLAB (1st Ed.). Springer Publishing Company, Incorporated.

[2] Na, Meng & Yang, Bin & Jia, Peifa. (2008). Improved Damped Least Squares Solution with Joint Limits, Joint Weights and Comfortable Criteria for Controlling Human-like Figures. 1090 - 1095. 10.1109/RAMECH.2008.4681441.



APPENDICES:

Appendix A: The Channel Addresses

For the Robot arm, generally channels 0 to 4 are used for the whole arm. An easy pattern to find the addresses is that each hexadecimal increments by 4 for every channel:

Example:

Ch.0 start = 0x06, Ch.1 start = 0x06 + 4*1 = 0x0A, Ch.2 start = 0x06 + 4*2 = 0x0E

Ch.0 stop = 0x08, Ch.1 stop = 0x08 + 4*1 = 0x0C, Ch.2 stop = 0x08 + 4*2 = 0x10

Channel #	Start Address	Stop Address
Ch. 0	0x06	0x08
Ch. 1	0x0A	0x0C
Ch. 2	0x0E	0x10
Ch. 3	0x12	0x14
Ch. 4	0x16	0x18
Ch. 5	0x1A	0x1C
Ch. 6	0x1E	0x20
Ch. 7	0x22	0x24
Ch. 8	0x26	0x28
Ch. 9	0x2A	0x2C
Ch. 10	0x2E	0x30
Ch. 11	0x32	0x34
Ch. 12	0x36	0x38
Ch. 13	0x3A	0x3C
Ch. 14	0x3E	0x40
Ch. 15	0x42	0x44