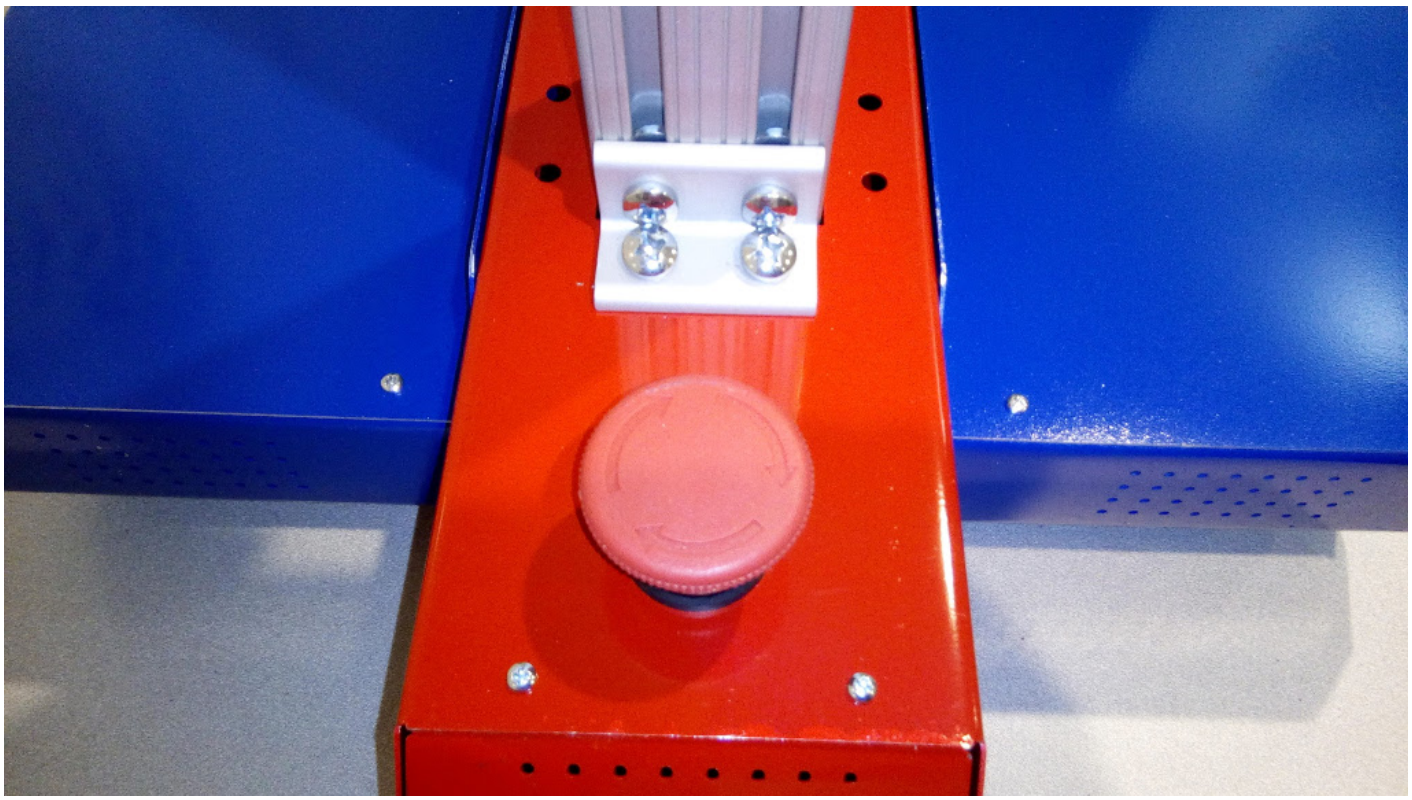
MMM Guide

Yosser D'Avanzo June 2016. (This article is deprecated.)

Getting to know the robot

The MMM can be divided into four main sections: the base, the chest, the on board computers, and the MMM.py file. We will learn how to work more closely with each of these different parts through this guide.

The base contains the two batteries, two motors, and the power switch. To power this section up, simply twist the big red switch. This should cause the switch to pop out. While in this state, the motors are free to receive electricity from the batteries. To turn off the base, simply push the switch in until it pops back into place. This can be done at any time, and in no way damages the robot. Any code running will continue to run, the motors just wont move.



In case of RUN AWAY ROBOT push BIG RED BUTTON!

The chest contains the arms, 5 spare servos per arm, ultrasonic range finders, and will be where our imaginations play with many marvelous mechanisms. To power this section up, look for the black I/O switch on the back of the chest, and press the I half in. This can be flipped off at any point without damaging the robot. Code will continue to run without the servos moving, but retrieving info from a sensor will fail because, well, they're not on.

The computer-y bits are the brain of the robot. The MMM is designed to operate with a main on-board computer, like a laptop or tablet, placed on the robot and connected to the central arduino. This section is on while the computer is on and the arduino is plugged in.

The MMM.py file is the given API for the MMM robot. It contains functions for all of the robot's basic movements and sensory information. Each of the functions have straightforward names and annotations summarizing what they do. For our purposes, we shouldn't worry about HOW these functions work; we should instead trust that they do, and use them as building blocks for bigger and cooler functions!

The "Hello World" Dance!

Like any first-time programming experience, we need to get some Hello World function up and running to understand the basics. To start, we need to connect to the robot. For this, we need to add in all the functions from the MMM.py API and the built in python "time" library.

```
from MMM import *
import time
```

We will be coming back to this section of code to add some more tools, but for now we have everything we need to connect to the robot. To use functions that move the MMM, like setWheelVelocity(), we need an MMM object. The constructor function, defined by def **init**() tells us that we need the portName to initialize an instance of the MMM class. portName is the serial port the arduino is connected to inputted as a string. Calling MMM(portName) will return the desired object which we should save under a variable name. In this example, we will attach it to COM5. After sending the command to create the MMM instance, we should wait approximately 5 seconds to allow the connection to be secured.

```
johnny = MMM( 'COM5' )
time.sleep( 5 )
```

We can now tell the robot what to do! The robot will default to its standard posture when booted up. To make johnny move, we start changing his variables using functions from MMM.py and then sending the package. We can call update() after every variable change or change a lot of variables at a time then call update().

```
johnny.setWheelVelocity( .18 , .18 )
johnny.update()
johnny.extendArms( 0 , .127 )
johnny.update()
```

OR

```
johnny.setWheelVelocity( .18 , .18 )
johnny.extendArms( 0 , .127 )
johnny.update()
```

You might notice that making the robot move this way can be quite unreliable. Delays and misinterpretations will happen regularly. In order to fix this, we need to add threading. The method above is like sending a postcard to the robot telling it what to do: this takes a long time and the card can get damaged. Threading allows us to have a more open and stable connection to the arduino, akin to a phone call where communications can clearly and quickly be dispatched. It does so by allowing the computer to run your code, telling the arduino what to do via update() and listening to what the arduino is saying via parseData() all at the same time.
Import the built-in 'threading' module. Now, we need to write an updateRobot() function for the threading to run. We need to make sure that all the variable changes we're about to send are within their restrictions by calling clampAll(), sending them via update(), giving the arduino a moment to process it and send us back info by calling sleep() for about

a tenth of a second, and then recieving the arduino's information via parseData(). This function should do this a bunch of times so we have continuous communication. Like an infinite loop number of times.

```
def updateRobot():
  while True:
    johnny.clampAll();
    johnny.update();
    time.sleep(0.1);
    johnny.parseData();
```

Then, to make the thread start, simply copy the next section and paste it under the above function.

```
# Begin Thread
thread = threading.Thread(target=updateRobot, args=())
thread.daemon = True
thread.start()
```

Now any calls in the code after this point will be run in parallel with the communications. We no longer need to call update() at any point because this code will automatically do so every .1 seconds. Up to this point is the recommended setup for all python 2.7 code communicating with the MMM.
The "Hello World" dance protocol varies from programmer to programmer, so feel free to be creative! Here, I'll make him turn clockwise, wait 1 second, turn counterclockwise, wait 1 second, then reset to his original parameters.
Note: functions within the MMM.py file can be called by using johnny.functionFromMMM(args). Functions made outside that file require the MMM obj to be handed in such as functionOutsideMMM(johnny, args)

```
def helloWorldDance( MMM ):
  MMM.setWheelVelocity( -.18 , .18 )
  time.sleep( 1 )
  MMM.setWheelVelocity( .18 , -.18 )
  time.sleep( 1 )
  MMM.reset()
helloWorldDance( johnny )
```

And there you have it! A robot showing you the dance of his people. He'd make Claptrap proud.

Base Movement

We will now focus on getting the robot to a preselected location that is a certain distance that is "x" meters (m) away from the robot. The problem we face here is that the inputs "v" for the setWheelVelocity() are in terms of meters per second (m/s). The solution to this problem is to set the wheel velocity, and then wait "t" seconds until we travel the required distance. We could try guessing how long we have to wait for, or use good old math instead. Distance divided by speed tells us the exact time we need to travel.
Note: to make the robot travel straight, set both wheel velocities to the same value.

```
t = x / v
johnny.setWheelVelocity( v , v )
time.sleep( t )
MMM.setWheelVelocity( 0, 0 )
```

Now let's expand the problem a little more. The previous exercise is great if the robot is facing the right direction but what if it isn't? Moving to the desired angle is a similar problem. Instead of speed in meters per second, we need to find the rate the robot turns at in degrees per second. There are a couple of ways to do this, however we will focus on an empirical method. Using a stopwatch, we can time how long it takes for the robot to perform one rotation. To make the robot rotate about its center, set the wheel velocities to equal magnitudes but opposite signs. Notice that which is negative only affects the direction it turns in, but not the rate it turns at.

```
johnny.setWheelVelocity( 1.8 , -1.8 )
```

One rotation, or 360 degrees (deg), should be approximately equal to 9 seconds. This means that the robot rotates at a rate of 40 deg/s when one wheel is 1.8 and the other is -1.8. To move "d" degrees clockwise, we wait "t" seconds of turning where "t" is calculated by dividing "d" by the turning rate

```
turningRate = 40
t = d / turningRate
johnny.setWheelVelocity( 1.8 , -1.8 )
time.sleep( t )
MMM.setWheelVelocity( 0, 0 )
```

Using these two methods, the robot can be hardcoded to follow a predetermined path. You can have it deliver things between two people on other sides of the room or building, or attach a paintbrush to the base and have it paint the floor with intricate art. There's plenty to do with just this.

We should now formalize formalize our moveForward() function from last time. We should get something like the following:

```
def moveForward( MMM, dist ):
  time_delay = dist / 1.8
  johnny.setWheelVelocity( 1.8 , 1.8 )
  time.sleep( time_delay )
  MMM.setWheelVelocity( 0, 0 )
```

To formalize turning, we need to be a little more clever. The concept is the same, only instead of inputting the amount to turn, we simply input the current direction and the desired direction and move the difference. This system has many more useful applications and will make future projects more intuitive. Angle goes up in the clockwise direction.

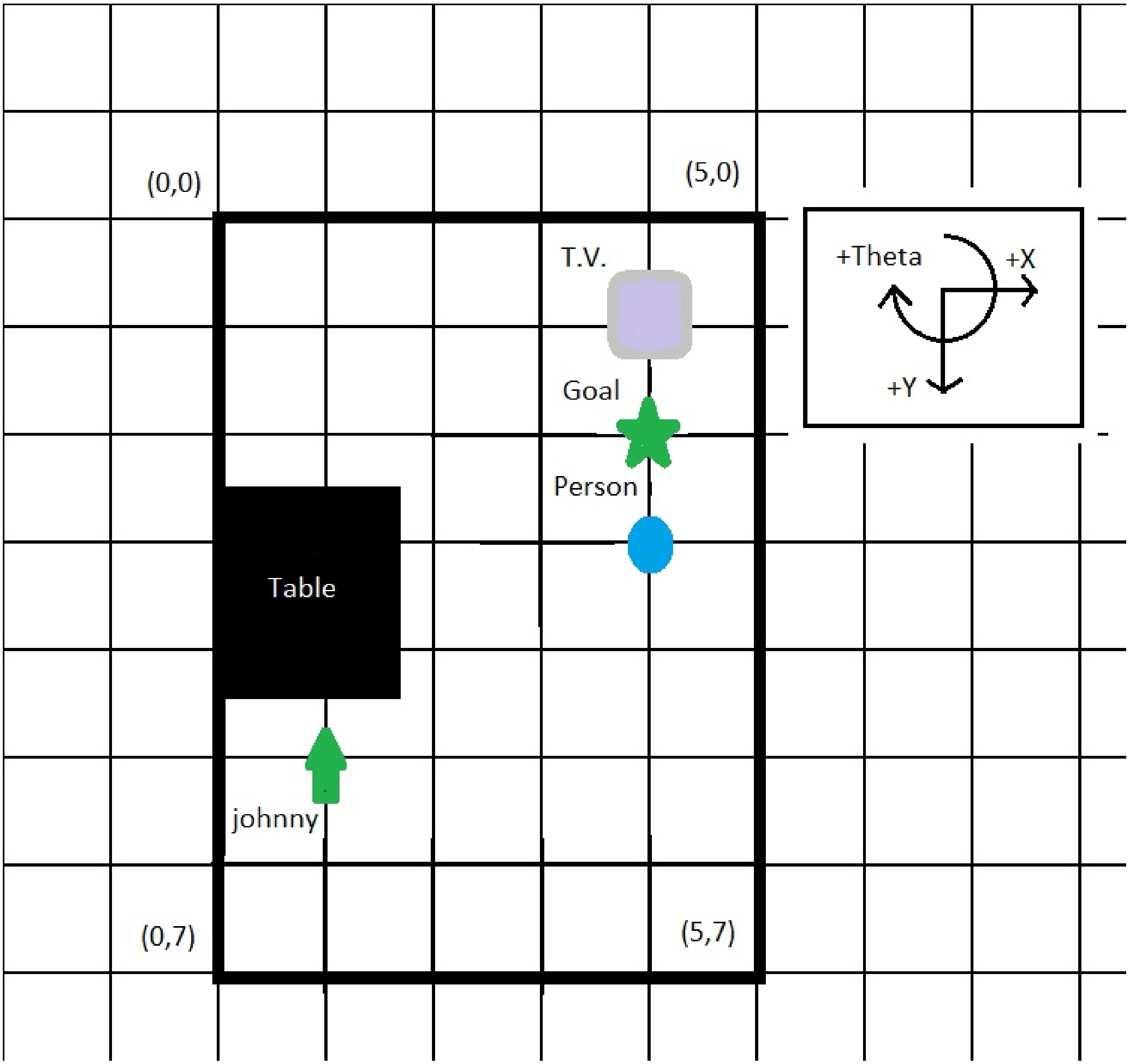
```
def turnTo( MMM, current_theta, desired_theta ):
  theta = (desired_theta - current_theta) % 360
  turningRate = 40
  time_delay = theta / turningRate
  johnny.setWheelVelocity( 1.8 , -1.8 )
  time.sleep( time_delay )
  MMM.setWheelVelocity( 0, 0 )
```

Disclaimer: The methods used above is known as dead reckoning and works on the assumption that there is no slipping while the robot moves. Since this guide is working with the standard issue MMM, we do not have any sensors that could help us account for slipping. Considering the robot's weight and wheel material, this should not be an issue on hard, clean floors.

navigate()

Now let's consolidate what we've learned so far into a function called navigate(). It is defined to make the robot move to a given a graph, its current coordinate on the graph, and a destination on the graph. The graph will represent the robot's actual surroundings as an n x m list of integers where every entry is either a 0 or a 1. 0's represent open space in the room, and 1 represents an obstacle at that coordinate. Every interval of this graph will represent .5 meters of actual space. For simplicity's sake, the robot can only travel on the intervals--no diagonal crossing.

0 degrees theta is pointed in the positive-y direction and is positive in the clockwise direction.




```
graph = [ [ 1, 1, 1, 1, 1, 1,  1, 1 ] , [ 1, 0, 0, 1, 1, 0,  0, 1 ] , [ 1, 0, 0, 0, 0, 0,  0, 1 ] ,  [ 1, 0, 0

robot = ( 1 , 5 )
goal =  ( 4, 2 )
```

```
def navigate( MMM, robo_loc, initial_theta, goal_loc, graph ):
  current_theta = initial_theta
  interval_size = .5
```

We can also define directions to make writing the code more comprehensive. Also, we should save the previous move to avoid undoing the most recent move. At first, our previous move was not moving, (0 , 0).

```
  up = ( 0 , -1 )
  down = ( 0 , 1 )
  left = ( -1 , 0 )
  right = ( 1, 0 )
  prev = ( 0 , 0 )
```

These definitions will also help us guide the robot in an intuitive manner. We will add a nested function to navigate() called go() which takes a move, orients the robot in the correct direction and moves it for us.

```
  def go( move ):
    if( move == up ): turnTo( MMM, current_theta, 180 )
    if( move == down ): turnTo( MMM, current_theta, 0 )
    if( move == left ): turnTo( MMM, current_theta, 90 )
    if( move == right ): turnTo( MMM, current_theta, 270 )
    moveForward( MMM, interval_size )
```

We also need to be able to check the graph if the next move is legal. It should return true if the given move will land us on a 0. Because this function is nested within navigate() it has access of all other variables and doesn't need them as inputs.

```
  def is_legal_move( move ):
    check = ( move(0) + robo_loc(0) , move(1) + robo_loc(1) )
    if( graph( check(0) , check(1) ) == 0 ): return True
    else: return False
```

To figure out what direction to go, we need to first know where the desired location is relative to the robot's current position. The previous move will be saved in a tuple to make sure that the robot doesn't end up moving right back to where it was. At first, there were no previous moves.

```
  x = goal_loc(0) - robo_loc(0)
  y = goal_loc(1) - robo_loc(1)
```

So long as the robot is not at its goal, we want to loop over logic to move it. The logic will try to decide a move. It will try checking what y-direction it wants to go based on its relative position, then check if that is a legal and non-previous move. If all this is satisfied, it will carry out the move and update the relative position.

```
  while( x !=0 and y != 0 ):
    if( y != 0 ):
      if( y > 0 ): next = up
      else: next = down
      if( is_legal_move( next ) and next != prev ):
```

```
        go(next)
        robo_loc = robo_loc(1) + next(1)
        y = goal_loc(1) - robo_loc(1)
```

Now repeat the same logic for the x-direction within the same loop.

```
    if( x != 0 ):
      if( x > 0 ): next = right
      else: next = left
      if( is_legal_move( next ) and next != prev ):
        go(next)
        robo_loc = robo_loc(0) + next(0)
        x = goal_loc(0) - robo_loc(0)
```

The robot now has a rudimentary program to navigate arbitrary environments! This method is based on the mountain-climber approach: you only move if it will immediately gets you closer to the goal. There is little planning ahead except for keeping track of your previous move. For most simple floor layouts this algorithm will work well. navigate() is also scalable for any graph size. So the intervals can be bigger or smaller, just as there can be more or fewer rows and columns.

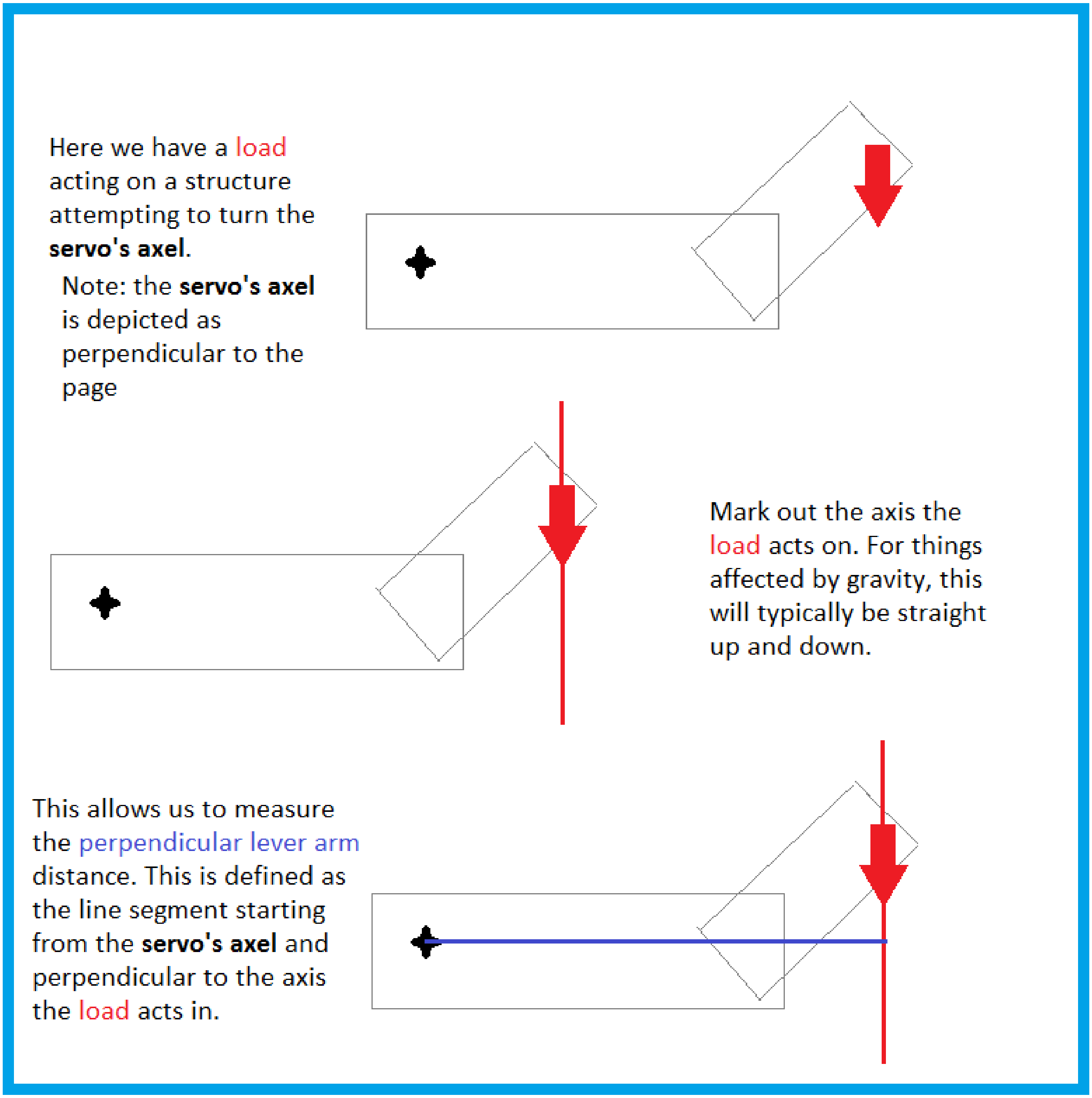
Torso Movement

This section focuses on learning how to maneuver the arms of the MMM robot. For nomenclature sake, we will refer to the joint closest to the chest and which moves the entire arm forwards and back, parallel to the ground, the shoulder. The next joint moves the forearm up and down and will be called the elbow. The forearm consists of the linear actuator that can extend and contract will simply be referred to as the arm. Each end point of the arm comes equipped with an Ultrasonic Range Finder and five (5) servo motor connections. Before we move on, we should first cover some basics about strength limitations about these joints.

Does it even lift, Bro?

One very important difference in the mechanics of the base and the arms is that the base works on motors that allow for continuous 360 motion while the arms function on servos that can only access a limited range of degrees. Servos also have the added benefits of being able to turn to a specific angle on command and are capable of resisting being turned while powered on--this is known as being not back-drivable. In practice, this means you can tell servo A to turn to angle X and trust that it will stay at that position as long as power is provided to the servo and the load is within the servo's rating. These two key features make servos much more useful in more delicate and precise work. On Top of this, human arms get along just fine without being able to turn every joint 360 degrees!

Every commercial servo comes with a rating for how much you can safely expect them to handle (Rating). This top turning motion is typically measured in Newton-meters (N*m) or pound-inches (lb*in). For the MMM, please consult the specifications manual for details on each joint's limit. We will use the perpendicular lever arm technique to check if the load (load), measured in Newtons or pounds, is safely held by the servo. This is accomplished by making sure that the load times the perpendicular lever arm distance (lever_arm) does not exceed the safety rating.




```
Rating > ( load * lever_arm )
```

This is very important for selecting which tasks can and cannot be performed by a given mechanism. Servos with higher safety ratings tend to be bulkier and more expensive, making servo selection a very important part of designing a robot. Due to its laser cut design and robust nature, the MMM is ideally suited for user-designed upgrades. If you want to do something with different servos or mechanisms, go right ahead and build it on!

Are we there yet?

At the end of each arm, there is an Ultrasonic Range Finder. Similarly to bats, the robot emits a pulse and receives information on depth. In this case, however, the depth information is of how far the sensor is from the solid surface it's being pointed at. Calling the getLeftRange() or getRightRange() functions returns this is measured in centimeters up to

. If the next surface is further than cm away, then the function returns -1. The sensor is ideally placed to work in tandem with the linear actuator on the arm to bring the end of the robot flush with some contact surface.

hiFive()

Let’s help johnny make friends. Friends like hi-fives right? Yeah they do. Admittedly, the standard stance he takes with a call to johnny.reset() lacks that hand-waving-in-the-air openness of a high-five. Within the first few lines of this function, lets go ahead and define a better stance. For the purpose of this example, the code will pertain to a right-handed high-five but can easily be adapted for either or both of his hands. The exact positioning doesn’t really matter, so feel free to be creative!

```
def hiFive( MMM ):
   MMM.rotateShoulders( 90 , 0 )
   MMM.rotateElbows( -60, 60 )
```

Cool poses are only half the battle when making new friends. The sensing someone else’s hand is what it’s really about. Let’s say that if someone’s hand is with 10 cm of johnny’s, then they’re accepting his high-five. That would mean while there’s something further than 10 cm or too far away for a good reading we should give it another very hopeful second.

```
   while( MMM.getRightRange() > .1 or MMM.getRightRange() == -1 ):
      time.sleep( 1 )
```

This infinite loop very patiently makes johnny wait until someone slaps him some skin. Currently there is no way for the person to know that the high five was well received since the termination of the loop doesn’t do anything but end the function. For the peace of mind of johnny’s new friend, let's add a reset() command so that something happens after the loop.

```
   MMM.reset()
```

There it is. Our robot has successfully delivered one unit of high-five and achieved the “friendship” directive. Honestly this task has made johnny as boring as a lamp. A closer examination of oh how Pixar managed to make a boring lamp an iconic persona can shed some light on what johnny is missing. Among other things, he’s missing secondary action. His main action is holding his hand up for a friendly high-five, but he carries it out so robotically that it may not seem welcoming or obvious to a passer-by. Secondary action is all the little things you do while carrying out the primary task like having your arm slowly droop down then quickly pick back up after holding it up for a long time, or maybe fidgeting in place because of anxiety or excitement. Let’s check the time, wake him up by getting rid of sleep(1) and help add some personality().

```
   start_time = time.time()
   while( MMM.getRightRange() > .1 or MMM.getRightRange() == -1 ):
      personality( MMM, start_time )

   MMM.reset()

def personality( MMM, started ):
```

At our disposal now is the ability to draw up a personality for this machine. It may seem like a silly task at first: why can't the robot just wait for a high-five like a statue so we move on? That's a completely acceptable approach to this situation because there are no right answers. This very artistic task is just as important to robotics as a whole as being able to navigate a room or calculating if a servo can handle a particular load. Basically any robot that will interact with humans should have some secondary action to its movements to ensure that users interacting with the robot are receiving the right physical cues just as they may if a person were doing the task. In this example, the robot's arm will appear to tire with time, slowly dropping until it jumps back up.

```
def personality( MMM, started ):
   time_passed = time.time() - started
   secs_passed = int(time_passed) % 7
   if( secs_passed != 0 ):
      MMM.rotateElbows( -60, 60 - secs_passed*5 )
   else:
      MMM.rotateElbows( -60, 60 )
```

The example above is cyclical. Another strategy for secondary action is to have random action driven by a random number generator (RNG). In this case, We'll generate a random number between 0 and 9 and multiply it by how many steps (degrees) we want the elbows to turn for, pause (seconds), then turn back to the original position and wait until repeating.

```
def personality( MMM, started ):
   step = 5
   random_move = randint( 0,9 )*step
   init_position  = MMM.rightElbow
   MMM.rotateElbows( -60 , init_positon + random_move )
   time.sleep( 3 )
   MMM.rotateElbows( -60, init_position )
   time.sleep( 5 )
```

As an added bonus, within hiFive(), right before the reset, lets have the arm extend out as far as the range finder says the hand is so it looks more like a real hi-five.

```
def hiFive( MMM ):
   MMM.rotateShoulders( 90 , 0 )
   MMM.rotateElbows( -60, 60 )
   start_time = time.time()
   while( MMM.getRightRange() > .1 or MMM.getRightRange() == -1 ):
      personality( MMM, start_time )
   MMM.extendArms( 0, MMM.getRightRange() )
   MMM.reset()
```

Before adding secondary action, johnny wasn't the best high-fiver. It was hard to understand what he wanted when he would raise his hand up and not react even when he was high fived. The first step was to give the user feedback that he/she did something by having johnny bring his hand back down. Then by adding random gestures and movements around the point of interest, johnny can direct the user towards the goal while communicating emotions like urgency or playfulness that would help the user become more comfortable interacting with him. Designing for easy human interaction can be the difference between a functional robot and a useful robot.

Bring it all in

We have worked with every part of the robot at this point making us familiar with each section of the robot: base, torso, and grippers. These parts can now all come together to complete a more complicated task, fetch(). We will be attempting to fetch a bottle from a table. The robot will begin at some arbitrary position from the table denoted by an x (meters), y (meters), and theta (degrees). We will be using moveForward() and turnTo() from the “Hello World” Dance, and also the gripper() function from the Firmly Grasp It section.

```
def fetch( MMM, x, y, theta ):
   johnny.reset()
```

Problems in Space

As described before, moveForward() and turnTo() work on the assumption that the wheels will not slip on the floor. This was because we did not have the means to account for the event. Sure, special wheels could be added to the robot or a dustpan and mop could be added to automatically clean the floor in front of the robot but then this doesn’t then account for very steep inclines, obstacles in the way, tipping over, or people bumping into it. There are so many directions that the robot could fail in that it would take nothing short of sentience to account for any and all problems. The set of these issues that a particular solution can handle is the problem space of that solution. Instead of creating the basis for the robot uprising, what we have done is reduce that problem space to only worry about moving the robot assuming that nothing else is affecting its locomotion. Doing so clarified the problem, provided insight to what methods could and couldn’t work, and made the issue more approachable. We can always go back and account for more issues building on previous work knowing that one problem was already taken care of.

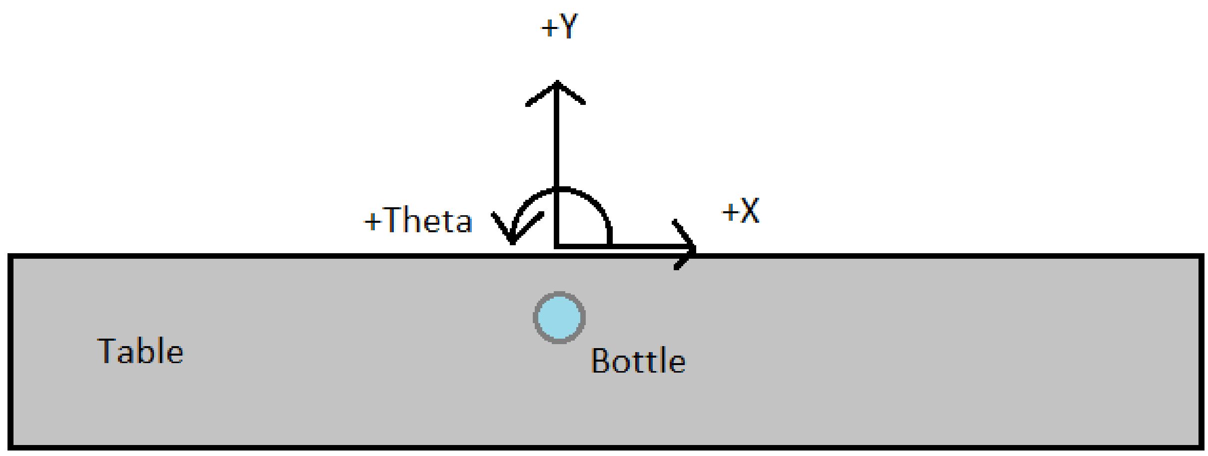
For simplicity’s sake, we will reduce the problem space for moving the robot in this task to assuming that the robot will not slip on the floor, that there are no obstacles other than the table, and that the robot can only begin with a positive y-coordinate value. This is to avoid having to work with mapping like in navigate(). That being said, navigate can be used to bring the robot to a position that satisfies all these parameters, and then have the robot perform fetch().

The moveForward() function at the beginning served its purpose in simple forwards locomotion. We can now go back and build on it, adding modifications to allow for negative inputs that will make the robot move in reverse. This can be done by saving the sign of dist in a variable, dir, making the time delay the absolute value of dist/1.8, and multiplying both of setWheelVelocity()’s inputs by dir to have them move in the proper direction.

```
def moveForward( MMM, dist ):
  dir = sign( dist )
  time_delay = abs( dist / 1.8 )
  johnny.setWheelVelocity( dir * 1.8 , dir * 1.8 )
  time.sleep( time_delay )
  MMM.setWheelVelocity( 0, 0 )
```

Coordinating Coordinates

The coordinate system is arbitrary and can be placed wherever. This gives us the ability to choose whichever suits us best. In order to narrow down the best coordinate system, we should consider the task; the easiest way to interface with a table is to approach it perpendicular to the edge you’ll be working. Approaching from any other angle means the robot will have to turn while at the table, which can cause a snag or slip. This example will be working with a coordinate system where the edge of the table closest to the bottle is the origin, this same edge is the x-axis, and the line perpendicular to this and containing the bottle is the y-axis. From the top view, the x-axis is positive towards the right, the y-axis is positive away from the table, theta starts on the x-axis and is positive counterclockwise. Setting up the coordinates in this way makes it easy to properly approach the table by zeroing out the x-coordinate, then traveling along the y-axis. We can go ahead and store the location of the tableside the robot will interact with.




```
loc = ( 0 , 0 )
```

With the location of our destination and with johnny's location relative to this, we can begin positioning ourselves. The first step is to make johnny get to the y-axis. This requires that he move to 0 deg if he is to the left of loc or 180 deg if he is to the right of loc. We should update our theta to move around properly, but not our x nor y in order to find our way back.

```
if( x < loc(1) ):
   turnTo( johnny, theta, 0 )
   theta = 0
else if( x > loc(1) ):
   turnTo( johnny, theta, 180 )
   theta = 180
```

Once in position, we want to move x meters in that direction, then turn towards the table, which should be towards 270 deg theta and record the change.

```
moveForward( johnny, x )
turnTo( johnny, theta, 270 )
theta = 270
```

Notice that if johnny begins on the y-axis, then the code will skip over any orientation in the x-direction and point johnny towards the table. By this point johnny should be able to move forward y meters to be flush with the table.

```
moveForward( johnny, y )
```

Johnny will now scan the table with the ultrasonic rangefinder while sweeping his shoulder. Using the extension arm and gripper, he will reach out and grab the bottle and retract the arm. Problem space reductions will concern sensing and engaging the bottle. We will assume that the bottle is at a height approximately equal to the height of johnny's shoulder joint so that it can be found by sweeping, that the bottle is reachable by extending the arm, and that the gripper will be able to hold the bottle in the air.

Sweeping the arm can be done by starting the arm at one extreme, then having a for loop increment the angle until reaching the other end. The smaller the step, the longer it will take, however big steps risk missing the bottle entirely. Note that the code is run fast enough that the steps can get very small without taking more than a few seconds in total. Here we will start by setting the left arm to the upper limit of 180 deg, and taking 3 deg steps.

```
johnny.rotateShoulders( 180, 0 )
for step in  xrange( 180, 0, -3 ) :
   johnny.rotateShoulders( step, 0 )
```

After moving the shoulders to the current step in the for loop, we should check that the ultrasonic range finder returns values within the distance we can reach with the extension arm. If it is, we should extend the arm to that distance and clamp the gripper, and return the hand. Note that the extension arm takes inputs in meters, and the range finder returns data in centimeters. Also note that the range finder returns -1 for out of reach or invalid distances.

```
   distance = johnny.getLeftRange() / 100.0
   if( distance != -1 and distance < .127 ):
      johnny.extendArms( distance, 0 )
      johnny.setLeftGrippers( 255, 0, 0, 0, 0 )***
      break
```

Once the bottle is safely at hand, we can back out up to distance y. Depending on if x was positive, negative, or zero, we can then face 0, 180 or 90, respectively and travel x forward.

```
moveForward( johnny, -1*y )
if( x > 0 ):
   turnTo( johnny, theta, 0 )
else if( x < 0 ):
   turnTo( johnny, theta, 0 )
else:
   turnTo( johnny, theta, 90 )
moveForward( johnny, x )
```

Finally, we need to know when to let go of the bottle in the left hand. The only input we can get from the user that they are ready to receive the bottle is through the ultrasonic range finder. Considering how great of a friend johnny is for bringing the user the bottle, he deserves a hiFive(). At the end of hiFive(), we need it to return True to tell us that that he was indeed hi-fived. In which case the left arm will be raised again, and will promptly release the bottle.

```
if( hiFive() ):
   johnny.rotateElbows( 90, 0 )
   johnny.setLeftGrippers( 0, 0, 0, 0, 0 )***
```

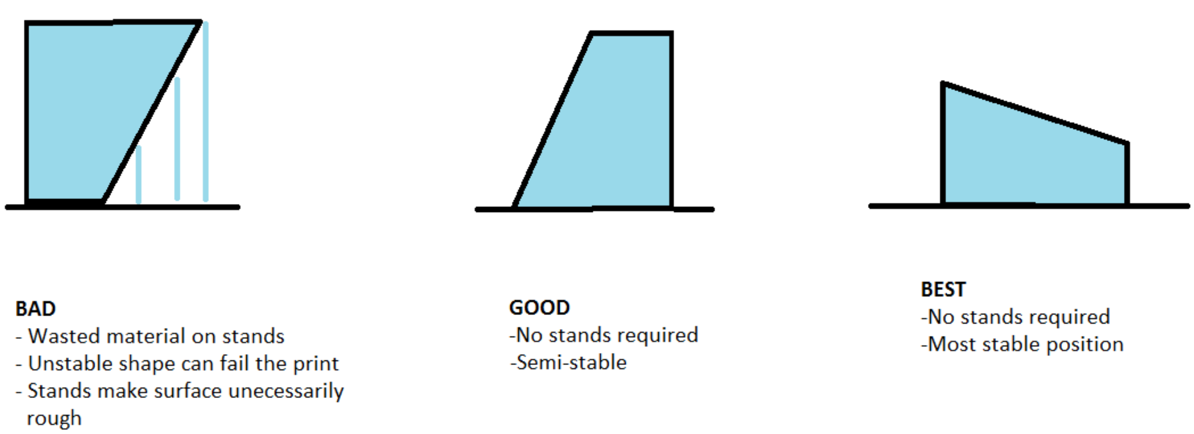
Handy Advice

It is highly encouraged to explore new gripper designs. The easiest way to rapidly produce complicated mechanisms and moving parts are by laser cutting or 3D printing designs. Access to these technologies makes creating toolings for specific cases very easy, and to make it easier, the MMM comes equipped with a versatile connector for easy additions. Each side of the MMM comes equipped with five slots for servos. Each servo can be mapped to a different variable in setLeftGrippers() and setRightGrippers().

Most commercial servos take in a value from 0 to 255. Functions like rotateShoulders() and extendArms() then map more intuitive values i.e. degrees, meters, into a number the servo can use. The three main types of servos are positional rotation, linear, and continuous rotation. Positional rotation servos are the most common and can have varying rotational ranges they can turn to. This type of servo drives all torso joints. Linear servos extend outwards and are used on the extension arm. The final type of servos are servos modified to act as motors allowing them full range of continuous turning. All three are supported for the gripper connectors.

Designing grippers is very case specific: no one gripper can do every task. That being said there are a few tricks to help make certain designs more functional.

1. Grabbing objects from the sides is usually a lot harder than finding some way to hook them. It isn’t always possible to do this, like picking up a ball, but when possible should be done. Remember that servos are non-backdrivable which means that quite a bit of weight can be hooked onto one before it will give.
2. More joints gives you more flexibility, but more room to fail. A very dexterous hand with all six servos acting in a new axis of motion would allow for extremely detailed and fine work. The big problem is that that also means that there will be six axes that the hand could fail in. A solid piece of aluminum acting as a hook is nowhere near as versatile, but much less likely to suffer a critical failure.
3. Use leverage. Servos come equipped with gear boxes, but that doesn’t mean that mechanical advantages like gears and pulleys can’t be used to multiply that power.
4. Make easily printable designs. The easier and safer it is to print a design, the more iterations can be made and thusly the more progress can be done on the project. If a part requires a lot of stands to be printed along with it, for example a hollow sphere, it may just be better to print it in flatter parts and assemble it separately. Never forget that something as simple as rearranging a part can have a huge impact on materials per print and odds of failing the print.



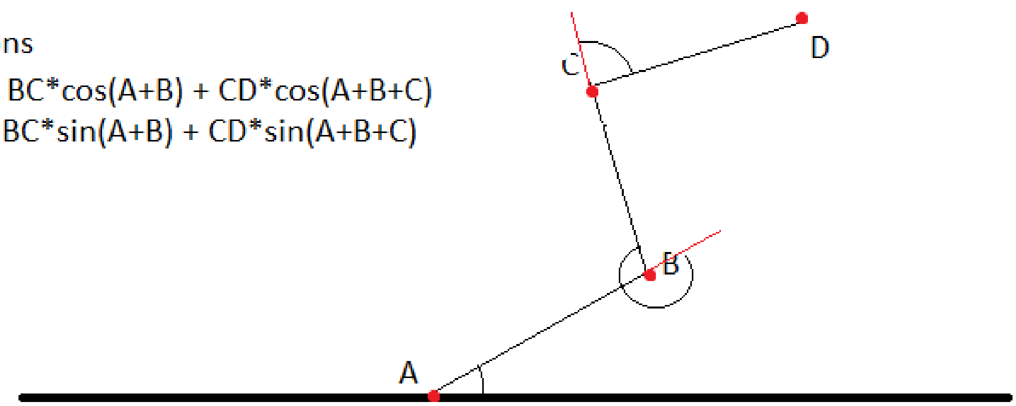
5. Springs can really lift your spirits. They come in many types, sizes, shapes, and designs. They can be bought specifically for extension, compression, and torsion and tend to be cheap. These specifications can make a weak

servo perform a much more difficult task by providing extra force like on the MMM or can be used as a clutch in a claw to provide force control based on extension.

6. Simple 2D kinematics can be very useful. It provides information on any joint in a mechanism moving on a plane. Using these equations can allow the MMM to raise a glass of water without spilling a drop by keeping the final theta constant while all other joints move. This also plays a huge role in fine spatial manipulation by making it very easy to move the tip of the joints to the exact position desired.

Kinematic equations

$$X = AB*\cos(A) + BC*\cos(A+B) + CD*\cos(A+B+C)$$
$$Y = AB*\sin(A) + BC*\sin(A+B) + CD*\sin(A+B+C)$$
$$D = A + B + C$$



7. Find it on the internet. If nothing else, there are many established and well-stocked websites that offer all sorts 3D and 2D models for free that can save a lot of time when brainstorming for ideas or looking to make a harder tooling.