This simulation is made for the purpose of simulating the behaviour of a quatruped robot dog. The robots and the simulations code different languages C++ for the robot and python for the simulation. The simulation is made to be as similar to the behaviour of the robot as possible, exluding the statemachine behaviour of the robot. The codes naming scheme and myAsyncServo class are based on github page in source [1] and main structure of the code and inversekinematics for legs is based on information from source [2]. This code was created as digital testing ground for the upgrades made for it, goal of these updates is to be able to control the robots movement and walking in a way that would make it possible to use in all possible situations.

1 Code structure

The structure of the code is shown image 1. Code can be divided in two sections, controlling that is made of the initialization and the command phase and the control structure of the robots parts. Control structure is divided to body, four legs and three joints for each legs. All layers of the code are responsible of driving the parts of the functionality those are assigned to. Body layer takes inputs and controls the behavior of the whole body by relaying information to the legs if needed. Legs are controlled indivudually by the leg layer and its mainly controlling the position of legs and calculating angles for the joints. Servo layer is responsible of controlling each individual joint and executing the movement given by the legs.

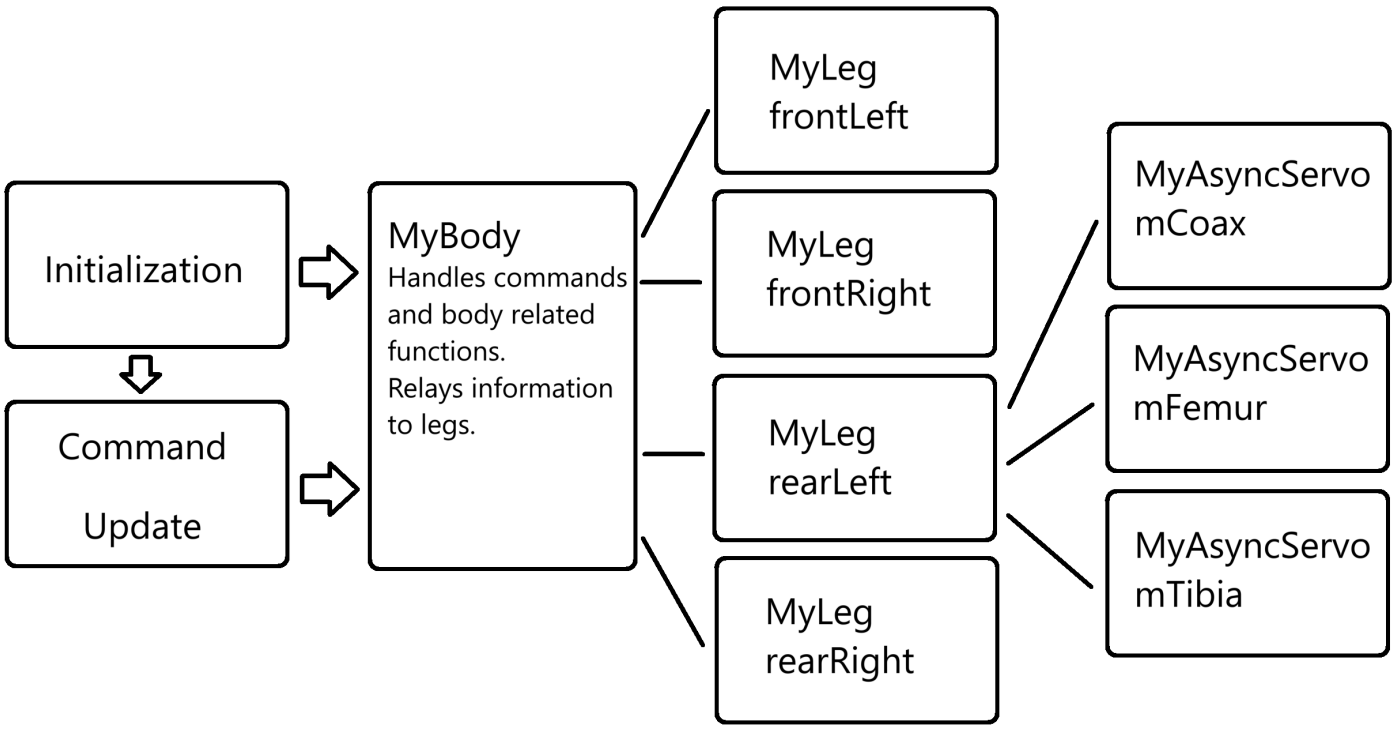

Figure 1: Parent child structure of the code.

1.1 Initialization

Initialization of the code consist of attaching the legs and servomotors by defining home and zero position for each leg. When first booting up it is important to make sure that all servomotors are correctly defined in the MyAsyncServo, mainly the motors degree range needs to be changed if it doesnt match the servo in use. In my setup I have only servomotors with 270 degree range. After everything is correct it is time to calibrate the robots home and zero position. This can be done by starting using the middle degree value of robots servos

and using those as home positions in the initialisation. Code used in this first stage is in the python version of the code is,

```
body = MyBody()
body.mRightFront.mTibia.Attach("SERVO_35KG",220,0,90)
body.mRightFront.mFemur.Attach("SERVO_20KG", 100.0, 0.0,175.0)
body.mRightFront.mCoax.Attach("SERVO_20KG", 135.0, 0.0,135.0)
body.mLeftFront.mTibia.SetLimits(80.0, 270.0)
body.Home()
```

This is repeated for every leg and joint. First argument defines the type of servo used in the joint (teensy version has the channel of motor first), second argument is the home position in degrees and next comes the offset of the home. Home could be calibrated using same degrees for every joint and changing the offset or by tuning every motor only using the home position and keeping the offset as 0. Last argument is the zero position of the leg joint in degrees. At the first calibration only the home position value is used to find the zero position. After the zero position is found the limits should llse be found and inputed for the setlimits command. Finally the home position can be calibrated.

The robots zero position is the position where the robot is standign legs straight down as shown in the image 2. This position is used as the name states as a zero point. All movement is calculated from this position. In the image there are all the global coordinatesystems and points of the robot. Legs of the robots are in local coordinate system defined as $x, y, z = 0, 0, 0$ at the yellow point. Coax joints turn at the black point, femur at the yellow point and tibia from the blue point. Leg coordinate system is positive Y down (blue line), positive X at the robots front (green line) and positive Z at the left of the robot (red line). Bodys coordinate system is at the center distance between leg origins. The red dot offset from the pody center is the center of mass (COM). Measuremets of the body, offset of COM and all general parameters are set in myconfig. In physical robot the legs are straight when line from the femur axle center through the tibia joint axel and contact point of the paw is perpenticular to the frame.

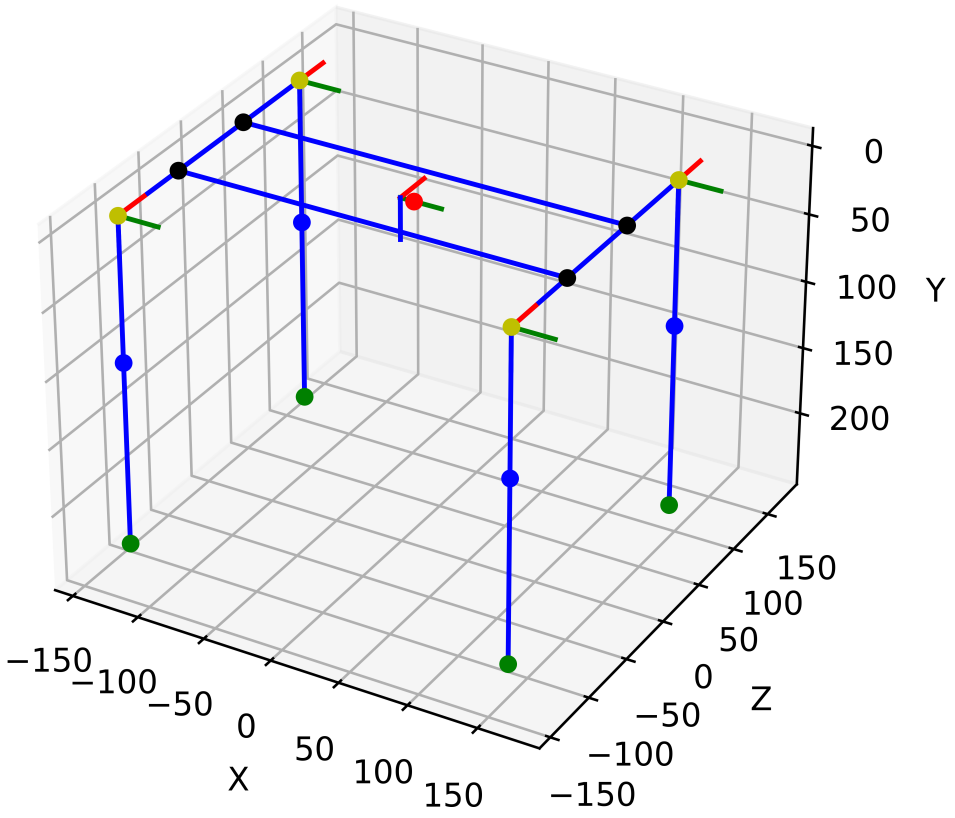

Figure 2: Zero position of the robot in simulation.

After the zero and limit degrees are calibrated the home position can be calibrated to a position that can be freely chosen. Home position is a position that is only used to start the motors at known place and the rest of the code doesnt know the coordinate values of it. I chose to use homeposition where the robot is as low as it can and it is shown in the image 3.

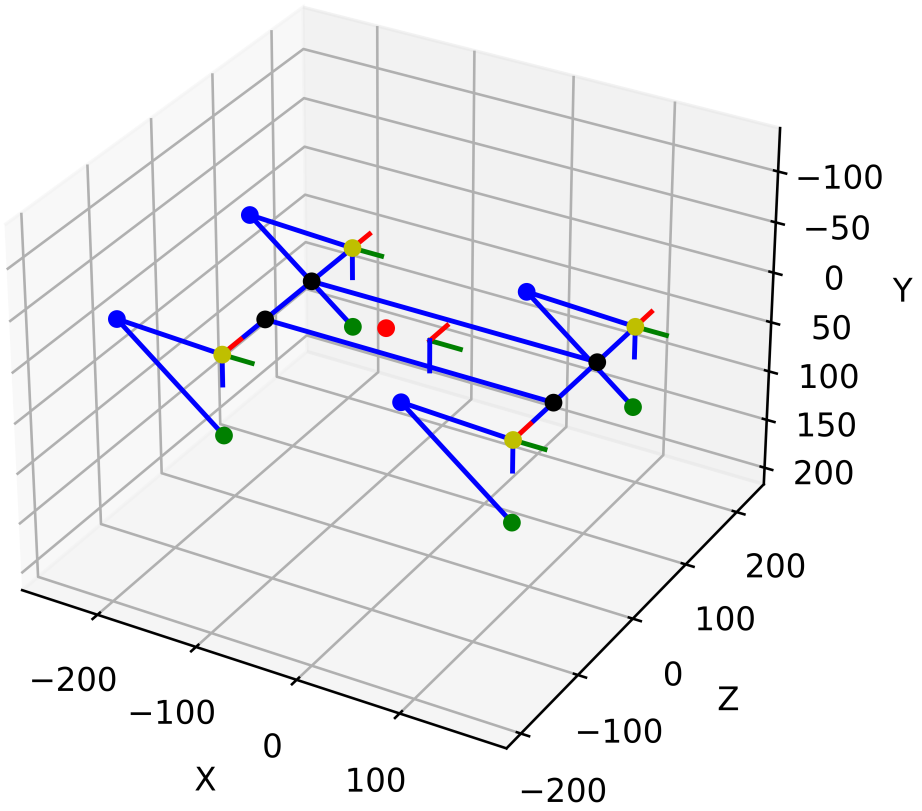

Figure 3: Home position of the robot.

After these steps are done the robot is ready to take commands, first command should allways be to move the robot to walking position. This will initialise all variables for positions so that the robot can be controlled properly. The initial walking position is sert up using myconfig but after the initial step the walking position is used as robots normal position and it is changed when center of mass is moved or the robots body position changed. My initial walking position is shown in the image 4.

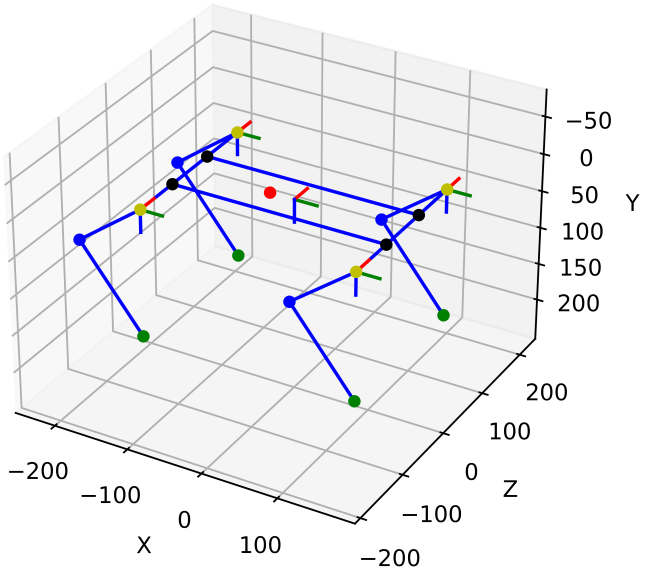

Figure 4: Initial walking position of the robot.

1.2 Command and update

After the robot is initialized its state can be changed. The robots position can be changed and it can walk. To make the robot walk walking gate and steering parameters should be set first by using the lines,
body.setGait("GAIT_WALK")
body.WalkDirection(0.2, 0, 0, 0, 0, 0, 0, 0.16)

Gait is basicly the delay between legs during single walk trajectory, walk direction is the steering parameters. Arguments of the steering go as velocity in m/s, rotation over X axis, rotation over Y axis, rotation over Z axis, movement X direction, movement Y direction and movement Z direction. Final value is ramp up rate. The steering parameters can be updated when ever needed but to those take effect the command,
body.PositionUpdate()

Needs to ba called. This goes allso for the position changes. Steering inputs are values ranging from 1 to -1.

Python version of the code updates everytime the update command is given but the teensy version only changes direction when the last cycle has finished.

2 Python functions

As stated before, the codes should be similar in structure, but there are some differences. In this documentation Im coing through the python version.

2.1 MyBody setGait(mode)

Sets the delays for every foot debending on the chosen walking cycle. Mode is the name of the gait. Delays are set from 0 to 1, where 1 is a full walk cycle. Delays are set to variable self.mFootDelays.

2.2 MyBody Home()

Calls homing command for each leg.

2.3 MyBody getLegPoints(leg)

Return the coordinates for leg in body coordinates. Because of how the body is controlled the coordinates are compensated to include bodys wire frame parts that represent the position of coax attachment point in every corner (self.wfRightRear for right rear corner). Argument leg can be LF (left front), RF (right front), LR (left rear) and RR (right rear). Used for example when the support triange is calculeted.

2.4 MyBody WalkDirection(velocity , degree_x , degree_y, degree_z, x , y, z, ramp)

This sets the steering information that is used during the walking and position changes to variables (self.sVelocity).

2.5 MyBody getSupportTriange(perc_LF,perc_RF,perc_LR,perc_RR)

Inputs are the current states of every leg in one walk cycle from 0 to 1. It returns points of triange when walk gait is used that represents the support triange formed by paw currently at the ground as shown in the image 5.

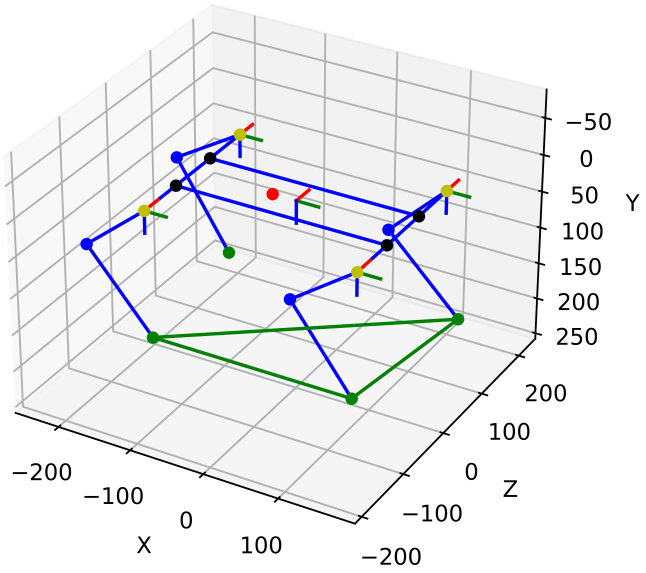

Figure 5: Support triange durimg walk cycle.

2.6 MyBody COMUpdateXY()

takes the points of every tibia joint in x z plane and calculates the movement of COM relative the leg movement. Then the wire frames COM is updated to different position. Center of mass is calibrated in the config file as offset from the body origin, when the legs move to a different position the mass of the legs produce changes for the mass distribution. This is approximated by considering the leg as massles beam with a weight at the end. The formula that is used for this is of form,

$$x_0 = \frac{m_1 x_1 + m_2 x_2}{m_1 + m_2}. \tag{1}$$

Where x_0 is the position of COM in this system from the measurement point, x_1 and x_2 are the distances of masses from the measurement point. Masses corresponding the measurements are m_1 and m_2. This formula is changed to work with four legs and taking note of the COM position relative to the body center. Distances used are given using body origin as measurement point and weights are as defined in configuration. The center of mass is taken in account by defining extra Weight that makes the formula to give 0 when legs are in zero position resulting to formula that is used for x and z axis seperately.

$$x_0 = \frac{m\ x_{LF} + m\ x_{RF} + m\ x_{RR} + m\ x_{LR} - 4m\ x_{COMoffset}}{8m} \tag{2}$$

$$z_0 = \frac{m\ z_{LF} + m\ z_{RF} + m\ z_{RR} + m\ z_{LR}}{4m} \tag{3}$$

Resulting values are then added to self.wfCOM to update position of the COM.

2.7 MyBody rotateBody(angle_x,angle_z)

This function takes inputs as form of angles in degrees. Angles are used to define unit quaternions over x and z axis as

$$q_x = (\frac{\cos(\theta)}{2}, \frac{\sin(\theta)}{2}, 0, 0) \tag{4}$$

$$q_x = (\frac{\cos(\theta)}{2}, 0, 0, \frac{\sin(\theta)}{2}) \tag{5}$$

Degree angles are turned to radians by using a constant as multiplier. Next the corner points of wire frame are taken from the wf variables. Both rotations are done to all 4 points of wire frame so the rotation is done in loop by using quaternion functions that are the same as in the legs. Both functions are explained later. Rotation using quaternions is done by applying the formula

$$v' = q^* v q \tag{6}$$

where q^* is the conjugate of unit quaternion q, and the resulting v' is the coordinates in new position. This equation is applied again with the other unit quaternion before updating the positions of one corner. Last the COM point is allso rotated to mach current body position. Take note that legs and body have different quaternion equations.

2.8 MyBody PositionUpdate()

This handles the movement of the robot. Walk cycle is divided in to 50 points in the curve making the stance and swing part of one cycle. Given velocity is used to get the duration of single step cycle and that is divided by 50 to get the duration of one step in the walking cycle.

Next it divides to different functionality debending from the action taken. During walking confition
self.sDirection_x != 0 or self.sDirection_z != 0 or self.sDegree_y != 0
is at effect, this translates to linear movement in x axis, linear movement in z axis or rotation over y axis. Combination of these is allso possible. Next step is to calculate the progression of each leg in walking cycle debending of the progress of the cycle and foot delays. Next the rotation point is defined if there is rotation over y axis. Next every leg is commanded to do the movement needed for walking and support triangle and COM is updated and wal step incremented.

MOVEMENT OF COM IS DONE HERE TO MAKE THE ROBOT NOT LOSE ITS BALANCE. CURRENTLY UN DONE

Next condition is true if there is no walking going on but the pody position is changed. Body position is changed by changing the standing position of the robot.
The function takes inputs from the three remaining steering inputs. It handles y axis height changes, rotation over x axis and rotation over z axis by applying those changes to the current standing position in steps. The step chise is based to a constant that corresponds the amount of movement or rotation in one walk cycle period. Rotations are done by quaternion turning as done in legs.
Last condition is to go back to current standing position.

2.9 MyBody GoWalkPosition()

Commands the legs to current standing position and updates the COM.

2.10 MyLeg getPoints()

All leg coordinates were in leg coordinate system where origin of the system is coax lenght away from the bodys wire frame. This function returns the position of all leg joints as body compatible format by moving the origin to a point corresponding the wirerames corner.

2.11 MyLeg setName(name) setReverse(value)

These two are used to set direction and naming identifier for every leg.

2.12 MyLeg quaternion_multiply(v1,v2)

This is a general function to do quaternion multiplication between two quaternions. Quaternions are 4 dimensional vectors containing 4 real numbers and imaginary unit vectors. Basic quaternion can be show in forms,

$$a+bi + cj + dk \tag{7}$$
$$(a,bi, cj, dk). \tag{8}$$

Quaternion multiplication is defined using defined rules and results in quaternion. Rules come from the imaginary units of the quaternions and can be shown as

$$i^2 = j^2 = k^2 = -1 \tag{9}$$
$$ij = k \tag{10}$$
$$ji = -k \tag{11}$$
$$jk = i \tag{12}$$
$$kj = -i \tag{13}$$
$$ki = j \tag{14}$$
$$ik = -j. \tag{15}$$

And the multiplication of two quaternions is then

$$v_1 = (q_0, q_1 i, q_2 j, q_3 k) \tag{16}$$
$$v_2 = (p_0, p_1 i, p_2 j, p_3 k) \tag{17}$$
$$v_1 v_2 = v_3 \text{where} \tag{18}$$
$$v_3 = (q_0 p_0 + q_1 p_1 ii + q_2 p_2 jj + q_3 p_3 kk, \tag{19}$$
$$q_1 p_0 i + q_0 p_1 i + q_3 p_2 jk + q_2 p_3, \tag{20}$$
$$q_2 p_0 j + q_3 p_1 ki + q_0 p_2 j + q_1 p_3 ik, \tag{21}$$
$$q_3 p_0 k + q_2 p_1 ji + q_1 p_2 ij + q_0 p3_k) \tag{22}$$

and using the rules of the imaginary parts it results to new quaternion

$$v3 = (g_0, g_1 i, g_2 j, g_3, k) \tag{23}$$

2.13 MyLeg quaternion_conjugate(q)

This function is used to get congugate of a quaternion. Because unit quaternions are used for coming functions and the quaternions contain 3 imaginary parts the quaternion and its conjugate can be shown as

$$v = (p_0, p_1 i, p_2 j, p_3 k) \tag{24}$$

$$v^* = (p_0, -p_1 i, -p_2 j, -p_3 k) \tag{25}$$

2.14 MyLeg rotateAxis()

Rotates the points of leg according to the angles of coax and femur. Position of the paw is updated using the other functions. For every leg the zero position was set and it is seen as the position where leg is straight. By knowing the angle of coax and femur it is possible to rotate the position of knee and shoulder points to correspond the current leg position. This is done by using unit quaternions as rotators. Unit quaternion is a quaternion is of form

$$q = (\frac{\cos(\theta)}{2}, \frac{\sin(\theta)}{2} X, \frac{\cos(\theta)}{2} Y, \frac{\cos(\theta)}{2} Z) \tag{26}$$

where X, Y, Z are unitvectors corresponging the axle that rotation is done by, in this case 0 or 1. And angle θ is the rotation in radians. Then the multiplication propertios of the quaternions can be used to transfer some vector v that is in this case vector from origin to the point we want to rotate to new positions multiplying it with the unit quaternion and its conjugate

$$v' = qvq^* \tag{27}$$

The rotation is done by constructing the unit quaternions with the angles and using the zero position of the point as vector. The resulting vector is the point in the new position.

2.15 MyLeg home() go_center()

Home is used to drive the leg in to the defined home position and go center is used to drive the leg position corresponding the middle of the whole movement of servomotor.

2.16 MyLeg IKSolver(x,y,z)

Variable MAX_LEN is the y axis measurement of the end of the leg. Legs z coordinates are tranformed to body frame corner. Using the coordinates given the angles for all leg servomotors are calculated using inverse cinematics equations. The code is based to implementation from source [2] and moustly corresponds the equations from source [3]. In the image 6 the distances and angles corresponding the equations are shown.

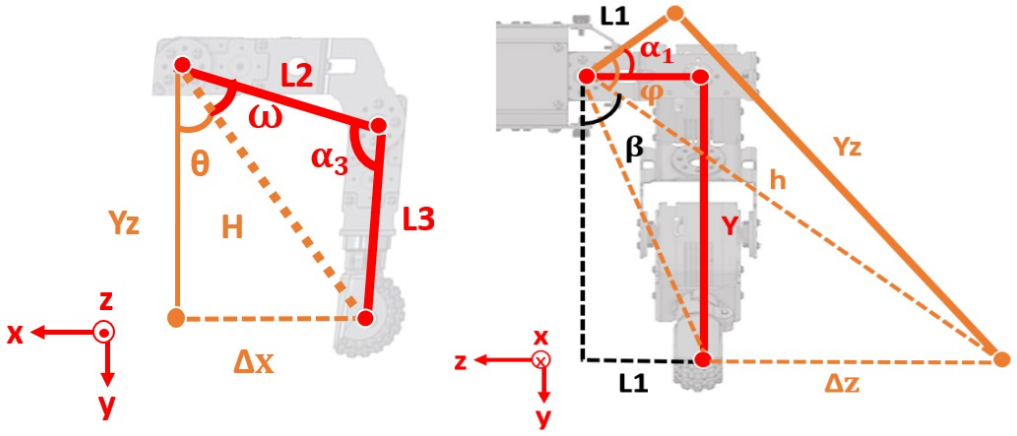

Figure 6: Visualisation of inverse kinematic angles and lenghts [3].

In the image 7 the resulting equations are shown.

$\beta = \tan^{-1}\left(\frac{L1+\Delta z}{Y}\right)$	$\theta = \tan^{-1}\left(\frac{\Delta x}{Yz}\right)$
$h = \frac{Y}{\cos(\beta)}$	$H = \frac{Yz}{\cos(\theta)}$
$\varphi = \cos^{-1}\left(\frac{L1}{h}\right)$	$\omega = \cos^{-1}\left(\frac{L2^2-L3^2+H^2}{2*L2*H}\right)$
$Yz = h * \sin(\varphi)$	$\alpha_2 = 90° - (\omega + \theta)$
$\alpha_1 = \beta + \varphi - 90°$	$\alpha_3 = \cos^{-1}\left(\frac{L2^2+L3^2-H^2}{2*L2*L3}\right)$

Figure 7: Inverse kinematicks equations where the used equations are based on [3].

The equations used are modified to work so that all angles are deltas from the zero position. The function result these angles.

2.17 MyLeg step_curve(percentage,rotation, y_rot,x,z)

This function produces the change in angles needed to move leg in the current walk cycle step. Argument percentage is the current stage of walk cycle for the leg, rotation is the center point of rotation curve, y rot is the rotation over y axis, x is movement in x axis and z is movement in z axis.

One step cycle is defined in 2 stages shown in image 8. One whole cycle is defined by using x_0, x_1, x_2, x_3 points for stance and swing stages and by using these points and interpolation the rest of the 50 points needed for a walk cycle are optained.

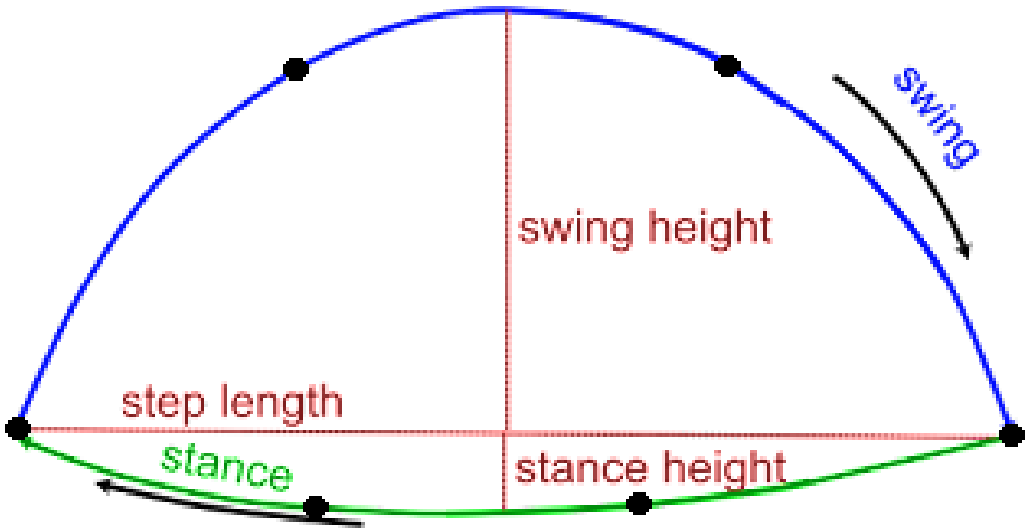

Figure 8: Walking cycle for one leg [2].

The function is based to code from source [2] and is modified to function with linear x and z movement and controlled turning over set point.

At the start lenght of step is defined and it debends on the movement wanted. If movement is linear the step lenght is calculated by pythagoras theorem, but if rotation is allso effecting the step lenght is then first defined for the middle of body and then adjusted relative to change in turning radius.

The location to turning radius is defined by x, z movement inputs and y rotation input. The components of rotation are shown in the image 9. Every leg will get the next point to move to with the functionality adabted from the code from source. In the original code walking was done only to x direction with turning by shortening the step size of one step.

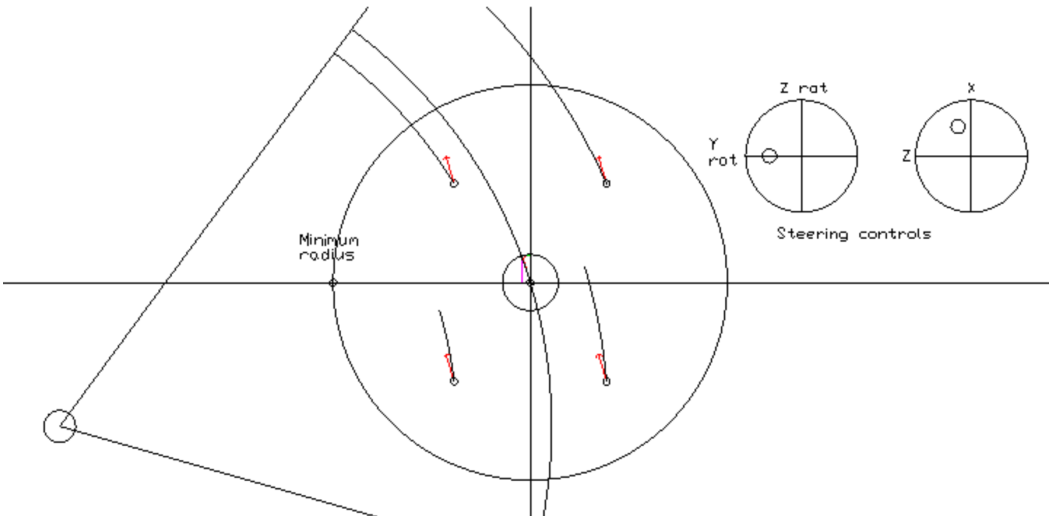

Figure 9: Controls and functionality of turning the robot while walking.

In the new walk cycle the directions are based to unit circle vector from the x and z steering inputs shown in the image as the right control instrument. With the linear vector in the body origin linear movement can be done in all x, z directions if no rotation is present by taking the step lenght from configuration file and

distriputing it to x and z directions.

When rotation is at play the distance of the rotation radius in z direction is controlled by the y rotation instruments input value. Rotation is centered perpentigular to the linear movement vector and there by its offset in x direction is effected by the value of x and z movements. Rotation center point is located between set maximum and set minimum when movement in x,z direction is involved and when those are 0 the center point is at the body origin. All step vectors are rotated by using unit circle and trigonometric functions by using the compensated step lenght when linearry moving or by the intensity of y rotation when it is not.

These values are then used in the interpolating as the goal points and it results in delta coordinates that the leg needs to move to produce the movement.

2.18 MyLeg GoPosition(x,y,z)

This function is used to move paw of leg to position with out taking in to account the current position of the leg. Sets the mPosition variable that is the coordinates of paw in standing position and MPositionPaw that is the current temperary position of the paw.

2.19 MyLeg BackToPosition()

This function is used to return the leg to its current standing position.

2.20 MyLeg movePositionWithDelta(delta)

The leg position is presumed to be known. This is used in walking and other temperary movement.

2.21 MyLeg WalkWithDelta(x,y,z)

Sets the temporary position of the paw during walking. The step curve function outputs deltas from x=0 and z=0 position so this is used during walking. Functions are allso separated between getting coordinates and moving servos for example COM position updating.

2.22 MyLeg addToWalk()

This is used to add offset after the step coordinates have been updated. Used in COM positioning.

2.23 MyLeg updateWalkPosition()

After all coordinates are included in the temperary coordinates this is the function that gets called when motor movement is needed.

2.24 MyLeg PositionWithDelta(x,y,z)

With this function it is possible to change current standing position. For example tilting is a action that changes this.

2.25 MyLeg updateStandPosition()

This function applies the changes in standing position by actually moving motors to the desired position.

2.26 MyLeg DoStep(percentage, rot_point,y_rot,x,z)

This is the function that finds the walking coordinates and updates the temperary position of one leg.

3 Results of the simulation

Code managed to achieve all set goals. It behaves and clearly shows how the robots code works. It isnt supposed to be ful physical simulation and there by it excludes gravity and other physical factors on the whole system. The code run in the simulator is still missing the COM position compensation that was found to be too long project to do properly for this time period. The work for COM offsetting will continue in the future, after witch the code is translated to c++ and used in real system.

System works as it should with the current limitations of matplotlib library. The code can be run live using the onscreen controls and only found foult states were the situations when robot was driven over limits, the maximum and minimum angles of each joint is functionality that is implemented in the real statemachine version of the code and as sutch it was not needed in the simulation enviroment. The simulation is allso a bit CPU heawe because of how matplotlib works. There was a effort to try to use vispy library for faster rendering but the effort didnt produce any results.

References

[1] https://github.com/cguweb-com/Arduino-Projects/tree/main/Nova-SM3

[2] https://leon70sml.blogspot.com/

[3] https://community.robotshop.com/blog/show/mechdog-kinematics-trajectory-synchronization-gaits