12-DOF Quadrupedal Robot: Inverse Kinematics

Adham Elarabawy

05/19/20

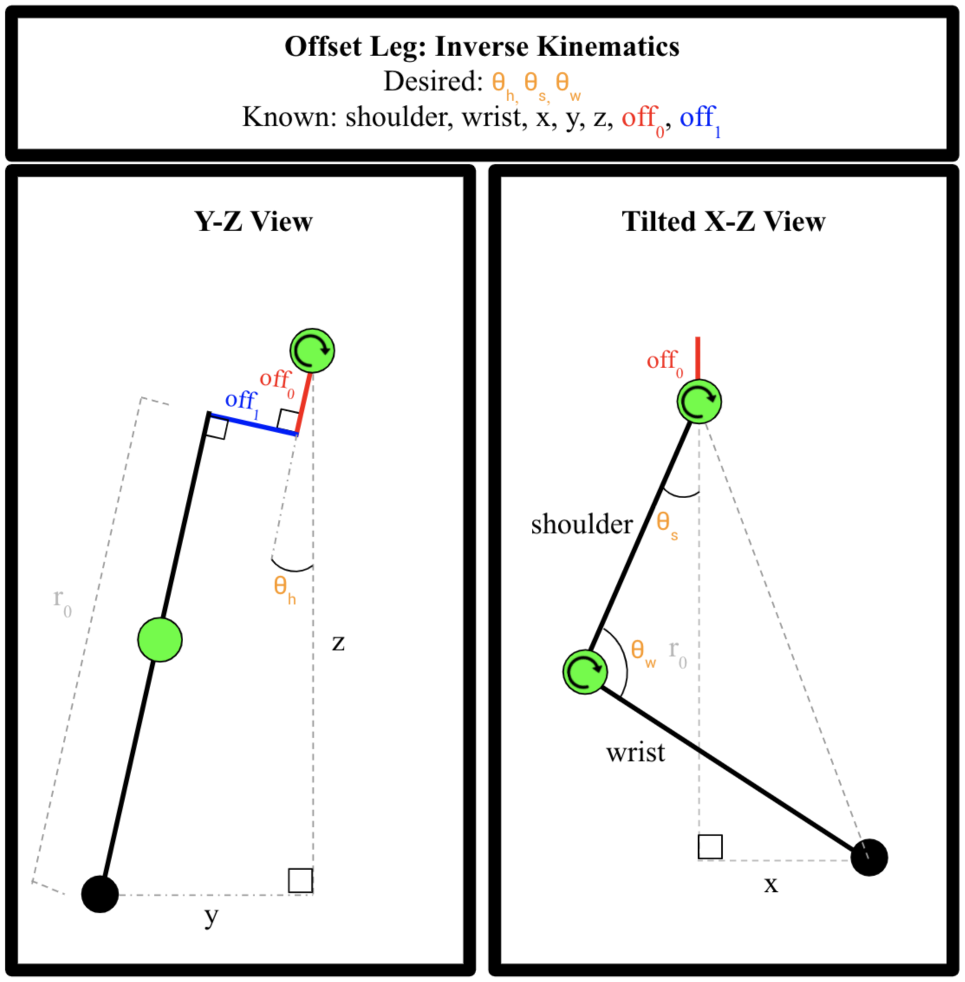
1 Introduction

This whitepaper is part of ongoing effort to establish a robust control system for the OpenQuadruped 3D-Printed Quadruped project that is accurate, fast, easily accessible, and properly abstracted. This paper is divided into 6 distinct sections that attempt to address the above ideals: **Introduction, Math, Usage, Visualization, Performance, Optimizations**.

In essence, an Inverse Kinematics (IK) model attempts to convert some intuitive domain, like xyz cartesian coordinate system, into directly useful values, like motor angles. Although at the time of this writing, some IK models already exist online for generic joints, none properly address both offsets present in the hip joint, which can make a noticeable impact on stability. Therefore, unlike existing implementations, my IK model accounts for both possible joint offsets present in most quadruped robots.

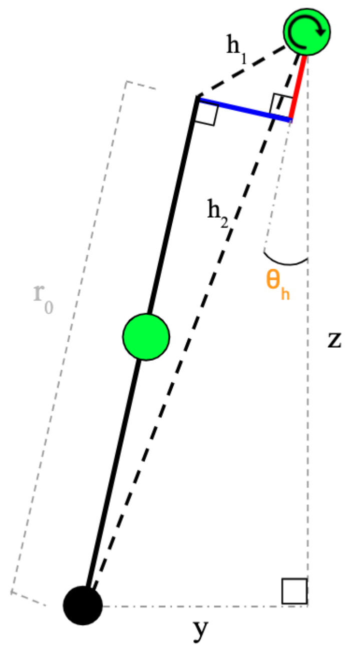
2 Math

First, let's understand the explicit objectives of the model:


Offset Leg: Inverse Kinematics
Desired: θ_h, θ_s, θ_w
Known: shoulder, wrist, x, y, z, off_0, off_1
Y-Z View
off_0
off_1
r_0
θ_h
z
y
Tilted X-Z View
off_0
shoulder
θ_s
θ_w
r_0
wrist
x

It is important to draw attention to the fact that the view on the right is a *tilted* X-Z view, which is tilted with respect to r_0. This allows us to carry over the r_0 from our calculations from the Y-Z view. Let's get started:

We will start with the Y-Z view–the messier of the two.

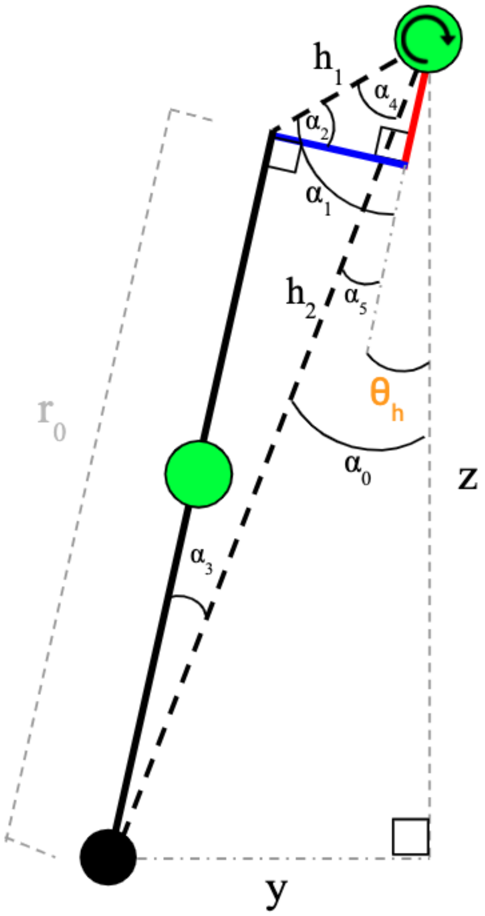


First, we calculate h_1 h_2 using simply Pythagorean Theorem:

$h_1 = \sqrt{off_0{}^2 + off_1{}^2}$ (ref the first image for offset positions)

$h_2 = \sqrt{z^2 + y^2}$

Ok, now let's add some intermediary angles that we can compute so that we figure out θ_h and r_0.



$\alpha_0 = \arctan\left(\frac{y}{z}\right)$

$\alpha_1 = \arctan\left(\frac{off_1}{off_0}\right)$

$\alpha_2 = \arctan\left(\frac{off_0}{off_1}\right)$

$\alpha_3 = \arcsin\left(\frac{h_1 * \sin(\alpha_2 + 90)}{h_2}\right)$ (law of sines)

$\alpha_4 = 90 - (\alpha_3 + \alpha_2)$

$\alpha_5 = \alpha_1 - \alpha 4$

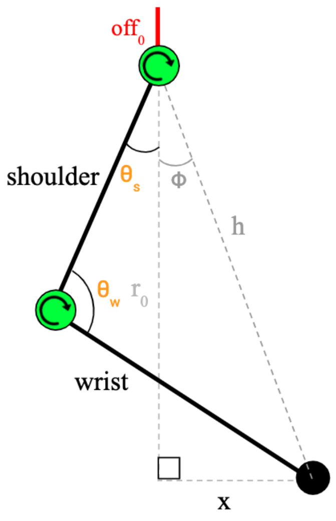
And finally: $\theta_h = \alpha_0 - \alpha_5$

We can then use the Law of Sines to derive r$_0$:

$$\frac{r_0}{\sin \alpha_4} = \frac{h_1}{\sin \alpha_3}$$

$$r_0 = \frac{h_1 * \sin \alpha_4}{\sin \alpha_3}$$

Now that we've gotten our r$_0$ value, we can move on to the *Tilted X-Z view*:



We already know: shoulder, wrist, x, and r$_0$. The first step in this view is to compute the third leg's value (the rightmost leg labeled h):

$$h = \sqrt{r_0^2 + x^2}$$

Now, compute ϕ through simple trigonometry:

$$\sin\phi = \frac{x}{h}$$

$$\phi = \arcsin(\frac{x}{h})$$

All sides of the visible triangle are now explicitly defined. Use Law of Cosines to get the θ_s (shoulder angle):

Note that s & w correspond to shoulder & wrist lengths, respectively.

$$\cos(\theta_s + \phi) = \frac{h^2+s^2-w^2}{2h*s}$$

$$\theta_s = \arccos(\frac{h^2+s^2-w^2}{2h*s}) - \phi$$

Now that the θ_s (shoulder angle) is computed, the only value that remains is θ_w (wrist angle) using the Law of Cosines:

$$\theta_w = \arccos(\frac{w^2+s^2-h^2}{2*w*s})$$

At this point, we now have the inverse kinematics for a 3 DOF leg completely solved. This means that given any xyz coordinate (within the bounds of the control range of the actuator), we can determine the exact joint angles–and thus the motor angles–that we need to go to.

3 Usage

Now that we can effectively command the quadruped's legs to any xyz position, this section explores the possible applications. There are two main ways we can utilize the leg IK: 6-axis body pose translation and rotation **or** moving the legs to certain positions in order to walk/trot/gallop according to a higher control structure.

The latter option is trivial on this end of the control structure. The complexity comes from planning gaits and generating leg trajectories, whose time-parameterized positions are then fed into this IK model for each leg. Therefore, I will leave this application to a future paper where I discuss gait planning and leg trajectory generation.

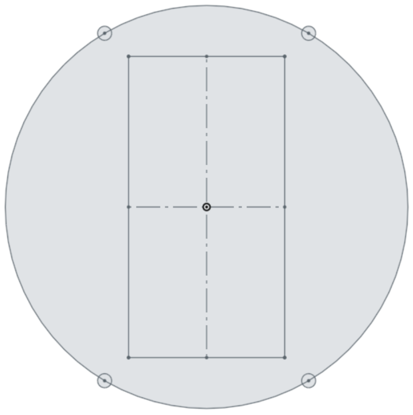
The former option requires further inverse kinematics, but in the scope of the quadruped's body rather than just the leg scope. To clarify, the 6-axis body pose translation and rotation refers to translating the body along the x, y, z axis and rotating about the Euler Angles: yaw, pitch, and roll.

Body Translation
Body translation along the x, y, z axis is actually relatively trivial as well. In order to translate the body along a given axis, you simply translate each leg in the same direction by the same magnitude. For instance, to translate the body 20mm down, I would simply lower the z input to each leg by 20mm. The same principle applies to all of the directions, and multiple translations can be applied commutatively.

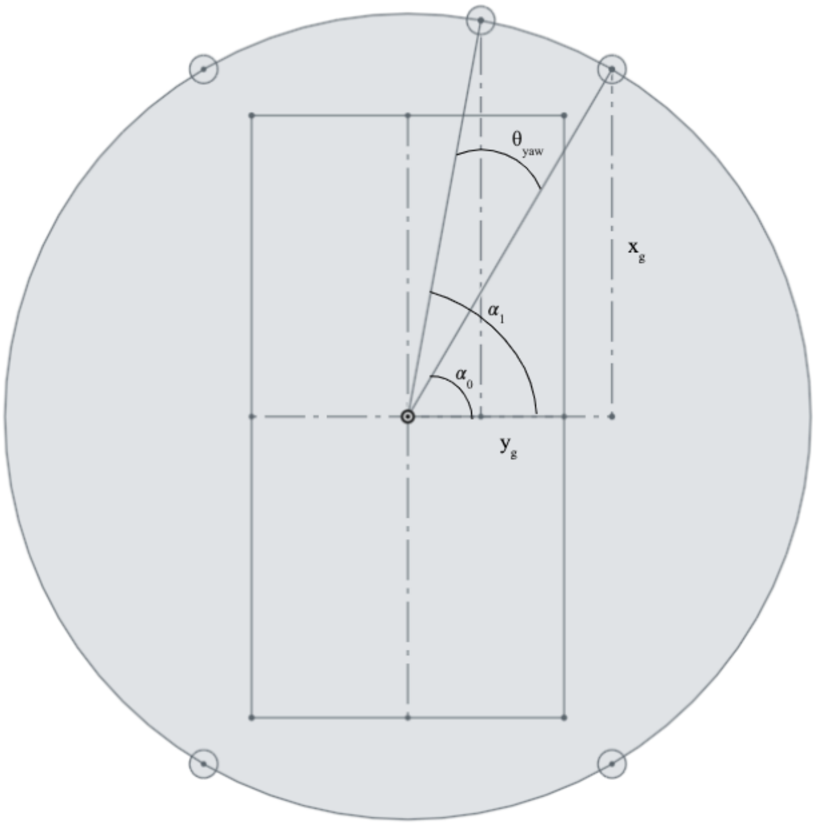
Body Rotation

Body rotation about the Euler Angles is a bit more difficult.

Let's start with yaw. Intuitively, when we increment yaw, we expect the body to rotate about the z axis. If we traced an arbitrary point during the yaw rotation, we would expect the resultant drawing to be a circular arc. Noting the relationship between the yaw translation and the intuitive circle arc actually yields our solution.



The frame above is from the perspective right above the dog facing down. The rectangle represents the quadruped's body, the small circles represent the feet, and the large circle represents the path that the feet will follow. The fundamental notion of the yaw rotation is that by moving the feet xy positions constrained by the large circle, the body will rotate in the opposite direction by the same amount.



So let's walk through one of the legs. In the diagram above, you should note that the large circle (clearly a constant radius) is defined by the xy euclidean distance from the origin of the quadruped to each leg. Since the circle's radius is maintained throughout the transformation, we can use this as a starting point for our calculations.

$$d = \sqrt{(x_g{}^2) + (y_g{}^2)}$$
$$\alpha_0 = \arctan \frac{x_g}{y_g}$$
$$\alpha_1 = \theta_{yaw} + \alpha_0$$

We can then get our final global (body scope) x and y values for the given leg through simple trigonometry:

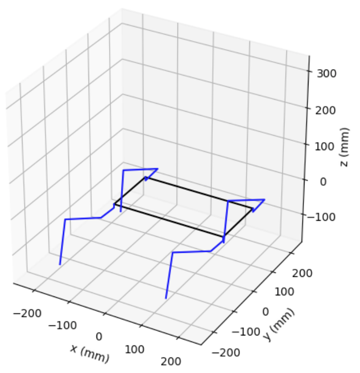
$$x_{new} = d * \sin \alpha_1$$
$$y_{new} = d * \cos \alpha_1$$

Before just plugging in these equations into your IK solver, you should note that the vertical flip of this math requires an operation change–meaning this math isn't perfectly symmetrical. The front two joints (FL & FR) use the above equations, whereas the back two legs (BL & BR) flip the cos and sin functions so that the new x value is computed using the cos function instead of sin and vice versa for the new y value. A final note is that these new values are the x & y positions of the leg in terms of the body scope. You can't just feed these values directly into the leg IK solver until you convert it back to the hip joint scope, which can be done by subtracting half of the body dimensions from the global x & y values.

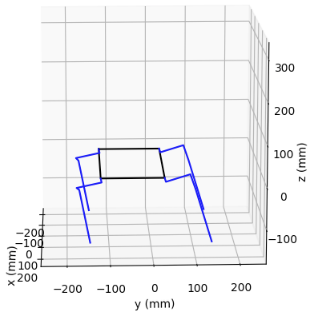
Note: Pitch and Roll will be added in a future revision of this paper.

4 Visualization

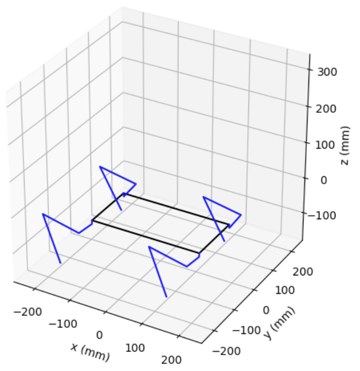
You can find instructions for running the visualization on my github page for this project:
https://github.com/adham-elarabawy/OpenQuadruped



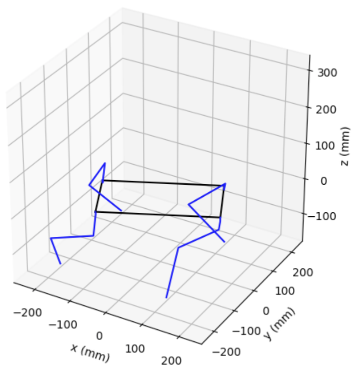
Body X Translation



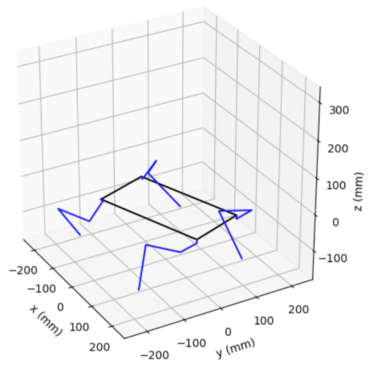
Body Y Translation



Body Z Translation



Body Yaw Rotation



XYZ Translation + Yaw Rotation

5 Performance

Leg IK computations for 4 legs was tested on a teensy 4.0 & 3.6 and formal performance metrics weren't profiled because the limiting factor in speed was blocking serial communication. However, with 2-way serial communication between one of the teensy and a RPi 4B+, performance upwards of 240+ fps was observed. The observed fps increased proportionally to serial baud rate, indicating that the serial comms, NOT the 4 leg IK computations, were the limiting factor. For most quadruped robot platforms, 240fps+ Inverse Kinematics is much higher than needed, and any further improvement in performance will not result in significant improvements.

6 Optimizations

The computations are currently all floating-point without any of the standard floating-point optimizations (i.e. scaling by a factor of 10 and rounding to int). If more performance is desired, one could perform these optimizations.