Agnim Agarwal, Krishna Pochana, Neil Janwani, Shiva Sreeram

SCRAT: A Squirrel Based Quadruped

Introduction

Quadruped locomotion presents a challenge in that it requires manipulating and coordinating four independent appendages which combine in a robot with a large number of DOFs. Such a task expands on concepts we learned in class this term, such as driving lone arms, dealing with rigid obstacles, and using secondary tasks to manipulate within the null space of our kinematic chain. This project seeks to implement and control a quadruped robot within ROS's *Rviz* to walk with a generalized gait and maneuver in-place. In this paper, we create a URDF for our quadruped, formulate our kinematic chains through which to control the robot, and ultimately explore our goals of walking and maneuvering using various trajectories while visualizing in *Rviz*.

Figure 1: Solidworks model of Scrat

Robot

First, let us look at the robot itself. All parts were designed in Solidworks, with the front and hind legs modeled differently. Different length ratios and joint styles are used in the front versus the rear, as can be seen in Figure 1. This was done to more closely model the anatomy of a squirrel, but also to see how our control methods were affected by the different structures. We did indeed see differences, with the front legs being able to maneuver with smaller joint movements due to the smaller joint sizes; however, the hind legs were less prone to multiplicities due to their larger bend at the elbows and elbow joint design. The assembly was then exported to a URDF through the Solidworks exporter, and with minor modification loaded into *Rviz*, as seen below. The Solidworks URDF generator works by making Solidworks-defined joints the joints in the URDF. It is possible to add inertias and masses as well to the file, however this was not necessary as *Rviz* is a purely visualization tool.

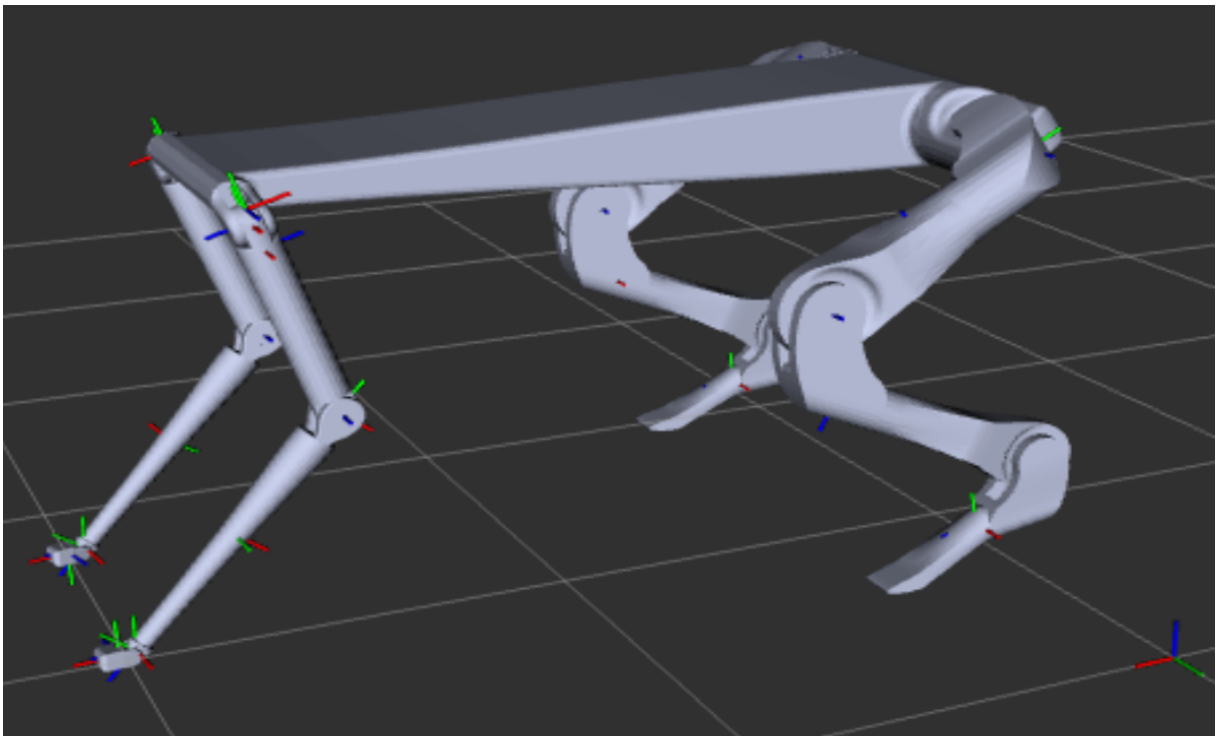

Figure 2: *Rviz* representation of the robot

This robot was designed to be a quadruped with realistic proportions that allow for natural movements in its space. As such, we have designed it to have eight degrees of freedom for each of the front legs and seven for each of the back legs. Thus, there are thirty in total. Specifically going into what each of these are, the front legs have: shoulder rotation forward and back, shoulder rotation in and out, rotation of the upper leg, elbow rotation, rotation of the lower leg, ankle rotation side to side, angle rotation in and out, and foot rotation. For the back legs, rotation forward and back at the hip, rotation in and out at the hip, rotation of the upper leg, knee rotation, rotation of the lower leg, ankle rotation up and down, and ankle side to side rotation.

As for the workspace of the robot, the torso is set at 1 meter to start, the front arms, when fully extended, are 1.4 meters long, and the back legs, when fully extended, are 1.72 meters long. When using *viewurdf.rviz*, the joints can be manipulated to move anywhere within the radius of the length of the leg from where it is attached to the torso but this isn't the objective for this project. We do not move through the entire possible workspace, as the goal of the project is to handle walking gaits and deal with singularities. Thus, in general the tip frame is kept below the torso of the robot, and we have an effective workspace of overlapping hemispheres below the torso.

Discussing the particular features, the most important one is the moving base. As we are performing walking motions, we need this to do more than just walk in place. In addition, notice that we have extra DOFs in all legs—these are used to make the walking motion appear natural, to keep the feet flat on the ground unless nearing a singularity (even when maneuvering), and other secondary tasks. These secondary tasks also allow us to deal with redundancies, such as dealing with the elbow being in or out. Given that we want to have the walking motion appear natural, we already have a desired way to handle the redundancies. More detail on the secondary tasks is given in the sections below.

Task

We are handling each leg individually so we are working with four tips (the feet), each of which has a dimensionality of six (x,y, z, roll, pitch, and yaw). Given their independence, the legs are coordinated to achieve the objective for natural motion. Since all of the legs are handled individually, the task is handled separately for each of them. The forward kinematics is handled the exact same way as the 7 DOFs arm we worked with in the class. Given that it is generalized, we simply call the function for both types of legs even though they have a different amount of DOFs. The overall task works fairly similarly to the 7 DOFs arm–identify target positions, find the current position using forward kinematics, find the error, and lastly perform inverse kinematics where we use the Jacobian (which is either 6 by 8 or 6 by 7 depending on the leg) to optimize where to move. For the Jacobian, there is a secondary task involved: setting centers for each of the joints where each type of leg has its own set of centers. This is how we handle the aforementioned redundancy. Not every task is necessarily achievable as one could tell the leg to move to a position outside of its range of motion (reach a singularity). Because of this possibility, we have prepared a type of demo to showcase handling singularities. We will go into more detail regarding all of these topics in the following section.

Algorithm, Implementation & Particular Features

-Moving the root

Normally in *Rviz,* the root is a static ground to the robot (as it was the robot arms created in class). However, this root can actually be 'moved' by transforming the world frame. This can give the illusion that the robot is actually walking. While the vector transforms necessary for the legs and root are discussed later, syntactically completing this objective is fairly simple. First, the vertical 3-dimensional position vector and 3x3 rotation matrix must be defined for the root transformation. Then, by creating a *T-matrix* one can generate a transform to be included in the command message that will be published to the running instance of *Rviz*. Adding on the timestamp to this message, publishing the command will enable the root (or world frame) to be transformed to wherever desired.

- Walking

A walking gait can be considered as a periodic motion and within each period is a set of partially overlapping motions with a smaller period, equal to the number of legs. Each leg goes

through the same motion, and to generate a walking gait, we only need to generate one general walking gait, then change the times at which each leg moves according to the gait. We also need to consider the fact that each leg does not start at the same position; it is only their position relative to the original start position that can be described by the same periodic motion. Thus, when the robot first spawns into *Rviz*, before beginning any motion, the gait generator records the starting positions of all four leg tips. Then, the general gait is phase shifted for each leg relative to the overall period of the leg so that there is a percentage overlap between consecutive legs. Thus, by connecting overall periods together, we can make a walking motion for arbitrarily long time lengths.

Looking at the general walking gait, we see that we need to generate motion in two main directions: in the xy-plane and in the z-direction (thus in task space). We could define closed loop paths for this, but sine and cosine functions make the velocity calculation much easier, and lend themselves to easy manipulation. Thus, we have one sine function for motion in the xy-plane and one for motion in the z-direction. The period of the general gait is controlled by the sine function for vertical motion, as we always want the leg to start touching the ground and return to this same position at the end of the leg's motion. The period of the vertical sine function can then be chosen according to the desired gait motion. We chose to run the horizontal gait for a half period and the vertical motion for a quarter period such that the leg would start at the ground, go up to a maximum, and then return to the ground, while going from the start position to the maximum amplitude by the end of the leg's motion. To generate a continuous walking gait, each leg moves in one step the same horizontal distance the torso moves in an overall period, and thus at the start of each overall period, each leg is at the same position relative to the torso. Lastly, we take the dot product of the xy-sine function with the unit vector of the direction the robot heads in, so that the robot can walk in any direction. The functions for the gait generation are given below:

Let R_z be the rotation matrix of the root frame relative to the world, $\hat{x}$ be the unit vector representing the direction the torso moves in relative to the root frame, A_{xy}, T_{xy} be the amplitude and period of planar motion, and A, T be the amplitude and period of the normal motion. The gait generation is a set of sin functions given by:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = (-R_z\hat{x}) \cdot A_{xy} \sin\left(\frac{2\pi t}{T_{xy}}\right) + \begin{pmatrix} 0 \\ 0 \\ A\sin(\frac{2\pi t}{T}) \end{pmatrix} \tag{1}$$

One last thing must be considered: the motions of the torso and the legs not in their walking periods such that the overall body looks as if it is walking while the legs not in motion look to be stationary. The motion of the torso is given by a velocity that is set by the user multiplied by the unit direction vector, and thus if a leg is not supposed to be moving, it will be, by default, moved along with the robot since the robot's torso is moved. To account for this, we drive the leg with the same velocity in the opposite direction, so that the stationary legs look as expected. Thus, we can generate a walking gait in any direction.

Once these points are determined, the velocities at those points and angular velocities are computed and fed through either a pseudo or weighted inverse (see demonstration video to view the difference between each inverse method) of the Jacobian to generate our joint velocities. The joint angles are determined through forward kinematics and integrating to the updated joint angles. Of importance is that we overactuated our legs (8 DOFs in front, 7 DOFs in rear) compared to the 6 needed to satisfy the three spatial and three rotational components of the tip frame. Thus, we used secondary tasks in null space to center the legs so that the elbow of the front legs did not hit the same joint in the rear legs, and to center the legs. The equations for integrating the secondary task through the inverse jacobian are given below:

Let λ be the gain of the secondary centering task, $\hat{\theta}$ be the vector containing all goal joint angles, and θ be the vector containing the actual joint values. Our secondary task integration then looks as follows:

$$\begin{aligned} \dot{\beta} &= \lambda(\hat{\theta} - \theta) \\ \dot{\theta} &= \dot{\beta} + J_{winv}(\dot{v_p} - J\dot{\beta}) \end{aligned} \tag{2}$$

$$J_{winv} = J^T(JJ^T + g^2 I) \tag{3}$$

- Singularity Detection

To detect singularities, we recognize that once a singularity occurs in a given leg, that leg will begin to drag and thus not be able to follow the desired position generated from our walking gait. Thus, the distance between the position and the desired position will become much greater at a singularity. To do our detection, we use the following algorithm:

1. From our walking gait, determine an estimate for the desired position, rotation, linear velocity, and angular velocity at time $t + \Delta t$ for each leg. Increasing Δt allows for 'sooner' singularity detection, while a small Δt will detect singularities right before they happen.
2. Compute θ-dot using a weighted inverse, with a secondary task centering at the defined root position
3. From θ-dot, compute theta at $t_1 = t + \Delta t$ and use forward kinematics to get the estimated Cartesian tip position for each leg at t_1
4. For each leg, calculate the distance between the estimated future position and the desired future position. If this distance exceeds our pre-defined bound of 0.1, signal a singularity in that leg. Increasing this number allows singularity detection to happen later, while decreasing this number does the opposite.

This method offers several advantages. We are identifying when the robot is extending past the task space, which can be identified when the determinant of the Jacobian matrix is close to zero. However, given the high dimensionality of this matrix for our robot of 30 DOFs, it is not

easy to calculate this without slowing down the program significantly. In comparison, the calculation of the tip position in this method is more straightforward given our use of fkin to calculate position from a list of joint values.

Once a singularity is detected in a leg, that leg steps out of the singularity and in the direction calculated from the path using the walking gait. Once a singularity is detected in a leg, the base link (the body of the robot) will freeze until the step is complete. In this case, the leg is set to step using the aforementioned walking code. By defining some 2D amplitude in the *xy*-plane, the robot may step some magnitude in the direction of the desired path.

We do this stepping out motion such that it keeps other legs stationary, so upon detecting a singularity, the stepping motion occurs while keeping all other legs at their same position. In our simulations, without a singularity, the robot will keep all leg tip positions constant and move the root frame in the direction of the path, so we generally expect the motion involves 1) the body stretching in the direction of the path followed by 2) leg stepping when a singularity is detected.

Since the root frame is moving in the direction of our path, moving the leg back to this position will allow the robot to continuously move in the path without any change in the robot's legs' relative position at rest. Furthermore, the robot may now follow any 2D path requested by the user. This is simply due to the fact that the robots swaying direction may be controlled, and singularity detection can enable the robot to step in the direction of the sway.

Analysis

https://youtu.be/49BmwsE4ZXc

Going through the demos, we see many of the applications mentioned above in the algorithms and features section. In the diagonal walking demo, we see how the individual leg gaits are indeed identical, and simply phase shifted relative to each other. We also see the periods stitch together to make a continuous gait that we can stretch for an arbitrary amount of time. Lastly, we see that we can move in arbitrary directions, in the shown case the direction is [1,1].

The next demo we are taking a look at is where we force the body to move to the side. The robot sways by following an $x(t)$ motion (while moving the leg positions by $-x(t)$) for a bit. After a certain point, the legs reach a singularity and the legs appear to lose all control. This is

due to the fact that only a pseudo-inverse of the Jacobian was used. This is really an overdefined system, so when a solution can not be found the following control of the leg is completely lost.

Correcting the motion outside the task space, we add weighting to the Jacobian pseudo-inverse (and make it square). This allows us to achieve a much better result. There is much less crazy motion, but toward the end we'd still like to see some singularity correction that prevents the robot from moving outside the task space.

In the singularity correction demo, we see that we can detect singularities using the methods presented in the last section, and once we are within a certain threshold distance from the singularity, we use the walking gait motion to move a set of legs (front and/or back as needed) to get back within the normal joint space of each leg. This threshold can be tuned to be as close or as far from singularity as desired. We can also see in this section the ability to keep the lower foot flat to the ground as we maneuver around due to our extra DOFs, as compared to something like the MIT Cheetah which has spherical point-like contacts and lower DOF legs that cannot control this.

As you may have noticed in the previous demo, the knees start moving in a strange way: too far under the robot. To correct for this, we apply a secondary task of centering code where we have chosen the predetermined angles for the joints. This effect isn't extremely strong but when seeing the demo, it allows for the legs to be in a configuration that is more closely aligned with the expectation of legs relative to the body.

Combining these features allowed us to implement both pushup and crouching demos where we prevent moving outside the task space in our previously mentioned redundancies, such as dealing with the elbow being in or out as well as intersecting joints. Integrating our centering method and our singularities detection provides a much smoother, natural motion for these experiments.

The last demo then gives us an interesting combination of all the aspects we explored, as we can maneuver around without regards for whether our point is reachable. In the case that our the point is unreachable with the legs in their original positions, we can simply correct leg positions to avoid singularities until we get to the goal point.