Implementing 2D SIMBICON Using PyBullet

Kaitlin Thachuk

A thesis submitted in partial fulfillment for the degree of Bachelors of Science (Honours)

in the

Faculty of Computer Science

University of British Columbia

Supervisor: Prof. Michiel van de Panne

April 2018

Abstract

This project explores locomotion of a biped in two dimensions using the SIMBICON (simple biped locomtion control) framework. The project is completed in a newer physics engine PyBullet and the character is represented using MuJoCo XML. The SIMBICON framework includes implementing a basic four state finite state machine which drives the walking motion of the character. Balance feedback, torso control using the stance leg torque, and decoupling the swing foot placement from the torso pitch all contribute to enabling the character to walk in a balanced manner. Torques for all the joints are calculated using proportional-derivative (PD) controllers. The results of this project are a set of target angles, feedback parameters, and gain parameters which are utilized in the main program

Contents

List of Figures

List of Tables

1 Introduction

1.1 Previous Works

The field of physics-based character animation has seen lots of improvement in the past ten years. Physics-based character animation is characterized by controlling the character through external forces and torques, meaning the global position and alignment of the character is not directly controllable. This is contrasted by the much more popular in industry kinematics-based animation which concerns itself only with motion trajectories on which the characters move. The main problem with kinematics-based animation, and the main motivation for research into physics-based animation is that characters on a motion trajectory have no concept of mass, forces, or collision/object interaction dynamics. All of these shortcomings are of course easily rectified if the character exists in a physics-based environment where interactions would naturally be in accordance with the laws of physics. Of course physics-based animation is not without its' own flaws and limitations. The major ones that have held it back thus far are the fact that controlling the characters is quite complex, requires a lot of manual tuning, and is prone to exhibiting erratic behaviours if the character loses control.

Thomas Geijtenbeek summarized previous work in this field quite nicely in his 2012 paper [3] which I will abbreviate here. Physics simulators, which are used to produce physics-based character animation, at a high level perform three basic steps. The first is collision detection, which is looking for intersections in the geometries of the objects. Collision detection itself is a vast field of study which is outside the scope of this project. For interested readers Fares and Hamam [2] wrote a state-of-the-art review on collision dynamics for rigid bodies which nicely summarizes some of the techniques in the field. The second step is forward dynamics simulation where for each object present the linear and angular acceleration is computed. Finally the last step is numerical integration. Using the previously computed accelerations the positions, orientations and velocities of the objects are updated. Again there has been some effort put into finding robust integration methods, one of which is the Runge Kutta mehods. Some examples of physics simulators/engines include Bullet and it's python wrapper PyBullet [1], MuJoCo [6], and Open Dynamics Engine (ODE) [5].

Moving on to the character itself, often the character models are constrained rigid bodies. This indicates that their bodies are non-penetrable and non-elastic and that their motion is constrained through limits on the joints connecting the links. The joint constraints are meant to mimic the real life joint constraints experienced in humanoids which the characters are often modelled after. The links themselves are often represented by simple geometries such as rectangles or cylinders, for which collision detection is easier, with given densities or masses which will be used by the forward dynamics step. The joints can be represented in many different ways but the most common joints are hinge joints which could be used to model a knee or elbow joint and a ball-and-socket joints which would be useful for shoulders and hips.

Now that we have a basic understanding of how the character has been put together and the environment it will be in we can begin to discuss the real meat of the problem: motion control. A motion controller is responsible for helping the character achieve its' task like walking for example. There are three common design approaches to motion control: dynamics-based optimization control, stimulus-response network control and pose-driven feedback control. Dynamics-based optimization control tries to use the equations of motion for the constrained dynamics to find joint torque vales that optimize a high level objective. Stimulus-response network control has roots in AI and biological systems where a generic control network tries to achieve motion control by constructing relations between the input sensors and the output joint torques. The design technique that will be used in this project is pose-driven feedback where the character has a set of target angles that form a pose and feedback controllers are used to compute torques that try to minimize the difference between the current pose and the target pose.

Feedback controllers are another area where there a quite a few different methods being used to try and control the character. For the sake of brevity, the focus will be on the ones used in this project although a more extensive list can be found in Geijtenbeek's paper [3]. The control technique utilized in this paper is called proportional-derivative (PD) control, which computes a torque that is linearly proportional to the difference between the current orientation and velocity and the desired orientation and velocity. The equation governing this control method is

$$\tau = k_p(\Theta_d - \Theta) + k_d(\dot{\Theta}_d - \dot{\Theta}) \tag{1}$$

where the d subscript indicates desired values and k_p, k_d are controller gain values.

To actually get a character to perform an action such as walking we need to move through a progression of poses. The most general way this can be achieved is by using a finite state machine in which each state is associated with a key target pose. Common triggers for state transitions are time limits and foot contacts. It is important to note that these target poses might not actually be reached when simulating and that some exaggeration might be necessary to achieve the desired motion. While this strategy is effective for 2D characters when moving in 3D some additional control is required for dealing with balance.

1.2 Overview of Approach

The purpose of this project was to reimplement the work done by Yin, Loken and van de Panne in their 2007 paper [7] on simple biped locomotion control (SIMBICON) using a newer physics simulator, PyBullet. SIMBICON aims to take on the problem of walking bipeds being generally unstable and create balance and feedback control to correct this. SIMBICON is a framework for controling a biped in both 2D and 3D using a pose-control finite state machine with four states. This allows for a large flexibility in the amount of gaits possible and makes it possible to transition between gaits smoothly. In each state we apply torques to the joints that are calculated using PD controllers. Finally balance control is provided in the the form of a feedback strategy that attempts to place the foot appropriately given the position of the body's center of mass.

2 Methods

Some of the methods (Subsections 2.1-2.3) described in this paper are techniques taken from the Yin et al. paper [7].

2.1 Finite State Machine

As mentioned above the character is controlled using a finite state machine. Each state has its own target pose which is given as a set of target joint angles for the stance and swing legs. The stance leg is the leg that remains in contact with the ground and provides stability whereas the swing leg is in motion driving towards the desired pose. These joint angles can be found in Table 3 and the joint torques are calculated using these values and Equation 1. The states are symmetric and are given visually in Figure 1 and in more detail in Table 1.

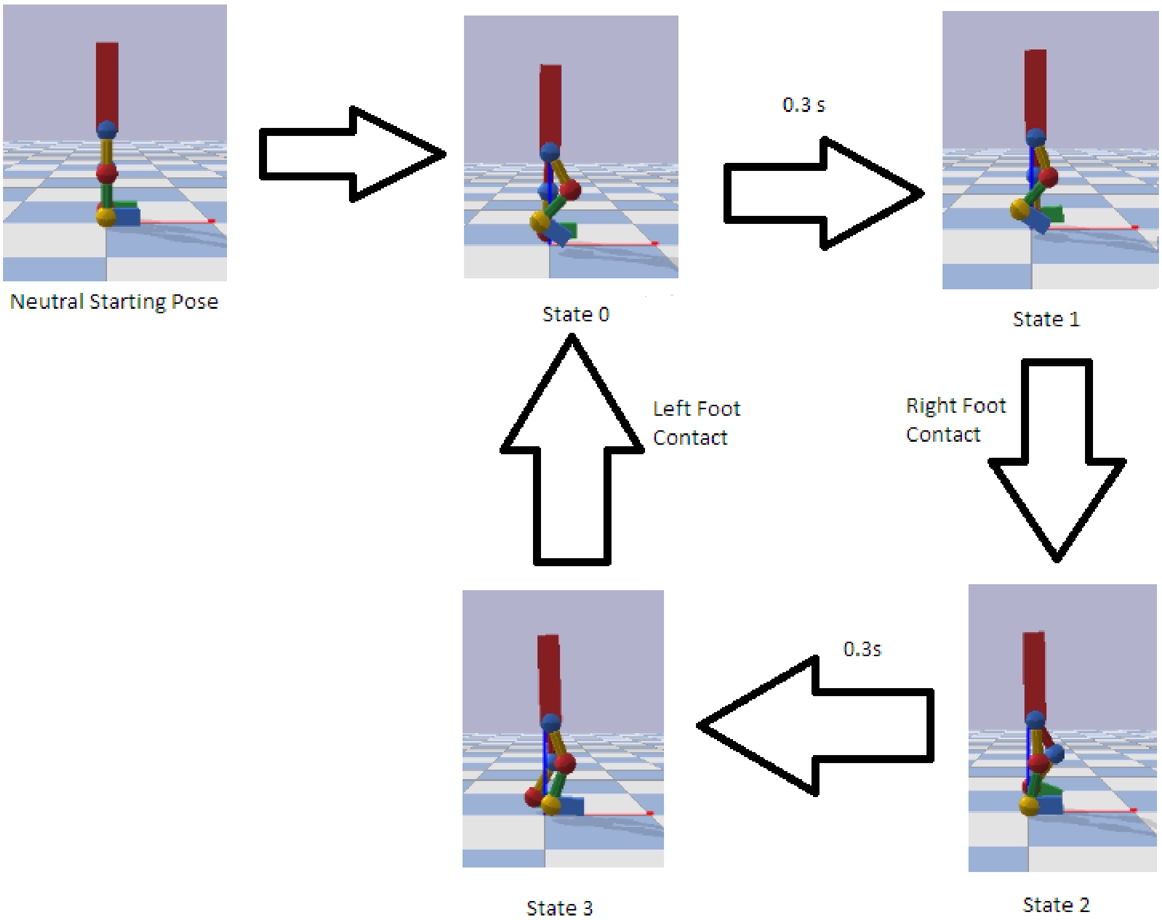

Figure 1: Finite state machine for walking character.

Table 1: State Descriptions for the Finite State Machine

State	Swing Leg	Stance Leg	State Transition
0	Right	Left	Δ t = 0.3s
1	Right	Left	right foot makes contact with the ground
2	Left	Right	Δ t = 0.3s
3	Left	Right	left foot makes contact with the ground

2.2 Torso and Swing-hip Control

There is a natural intuition that while walking it is desirable to try to keep the character's torso as upright as possible with respect to the world frame. This can be achieved by using a virtual PD controller to compute a net torso torque τ_{torso} in the world frame. It also follows that the swing foot placement should not be affected by the current pitch of the torso. This also results in a virtual PD controller operating with respect to the world coordinates to calculate the swing hip torque τ_{swing}. Since we cannot apply any external torques to the character both of these torques have to be the result of internal torques only. This can be achieved by utilizing the fact that the swing and stance hip torques are calculated separately. We can then require that the net torque seen by the torso is in fact a result of the swing and stance hip torques

$$\tau_{torso} = -\tau_{stance} - \tau_{swing} \tag{2}$$

and can consequently use Equation 2 to calculate the torque for the stance hip as

$$\tau_{stance} = -\tau_{torso} - \tau_{swing} \tag{3}$$

This ensures that the torques remain internal and satisfies the desired behaviour.

2.3 Balance Feedback

A key component to the control strategy is the balance feedback, whose aim is to try and help the character balance as it walks. The main strategy for achieving this is employ a feedback law that is going to affect the placement of the swing foot. Intuitively we would like to see balance behaviour where if the character was falling forward a larger step would be taken forward and if falling backwards, a smaller step. We can achieve this by applying the following feedback term to the swing hip

$$\Theta_d = \Theta_{d0} + c_d d + c_v v \tag{4}$$

In this equation we calculate a new target angle for the swing hip based on the original desired angle Θ_{d0}, the horizontal distance between the stance ankle and the torso's center of mass d, and the forward velocity of the center of mass v. Two more gain parameters are introduced into the model as a result of this feedback, c_d and c_v. When the character is moving slowly c_d has a more significant impact on the balance control and consequently c_d is often much larger than c_v. Both d and v are needed since it is the combination of these parameters that fully describes the current position of the character. Including both allows for balance control that is responsive to the torso pitching a large distance slowly and pitching quickly but over perhaps a smaller distance.

2.4 Character Representation in Robotic Languages

There are a few different options available for the format one can use to represent a character for a simulation. This project explored two: Unified Robot Description Format (URDF) [4] and MuJoCo [6]. Both of these formats are supported by PyBullet for parsing/loading, and both come from sources that integrate these formats into their own physics simulators. While both of these formats accomplish the same goal of providing a representation for the character they are different in subtle ways. The difference most relevant to this project is their treatment of constraining a body to 2D. In URDF you cannot specifiy constraints on the body in the global frame using the XML. URDF therefore places the task of constraining the body onto the physics simulator itself. This became a problem because the Bullet functionality for constraining characters is not exposed in the PyBullet implementation. PyBullet does provide some functionality for creating constraints but it is limited to prismatic, fixed, point to point, and gear joints none of which provide the desired behaviour of a hinge joint. This means it was not immediately clear how to constrain the character to 2D using URDF and PyBullet together and after a large amount of frustration it became clear that switching the character representation was going to be more beneficial. MuJoCo was the next choice and the format that was eventually used in this project because it allows the body to be constrained to 2D in the XML specification which PyBullet could parse in and adhere to. In the XML limits on the joints, the geometry of the links, and the masses of each link are defined. For the character used in the project the link masses are in Table 2 and the joint limits are given in Appendix B with the rest of the detailed code documentation.

Table 2: Link Masses

Link	Mass(kg) for each
Torso	15
Thigh	7
Shin	5
Foot	1
Hips, Knees, Ankles	0.5

2.5 PyBullet and Debugging

As mentioned in the previous section PyBullet is a python wrapper for th C++ Bullet library. As a result of this only some components of the Bullet library are available to PyBullet users. This makes it extremely difficult to find resources when trying to debug an issue or weird behaviour. Often forums will be focused on using Bullet only and the solutions provided by other users requires Bullet functionality not exposed in PyBullet. Furthermore Bullet is not popular enough that all problems encountered have solutions available online. As an example, if you make the time step of the simulation larger the character floats up off the ground and into the air, I suspect this has something to do with the integrators being unable to produce valid approximations but there was no solution or explanation of this phenomenon findable anywhere so it remains a mystery. Excluding obvious incorrect behaviour debugging these systems are extremely tedious and frustrating. For the most part trying to access and understand the behaviour you are seeing in your character is done visually and through print statements. PyBullet does have some debugging capabilities such as being able to put lines and text in the GUI, as well as change the camera, and visualizations. The most "useful" debugging ability is to create a slider which allows you to dynamically change a parameter. Personally I did not use any of these debugging tools since there are a large number of parameters that are all interdependent and it's easier to modify things manually. One of the aspects of projects like this that is generally under-reported in academic papers is the amount of time that goes into manually trying to find there parameters. Debugging these systems boils down to running simulations repeatedly and trying to fine tune based off of what appeared to go wrong. Of course base starting values can be approximated by using angles that seem reasonable in the real world but all of the fine tuning is done by hand. Furthermore, the fine tuning gets more complicated when you include the more macro level parameters like k_p,k_d, c_d, and c_v. When trying to achieve the most optimal motion theses parameters can also play a huge part in the behaviour you are seeing. Overall, PyBullet does not seem to provide any sort of meaningful structure to make this process any easier and manually tuning parameters appears to be the most common strategy.

3 Results and Discussion

Tables 3 and 4 contain the main results of this project, the target angles and parameter values used. Using these parameters the figure is capable of walking until it falls off the edge of the plane. The balance feedback helps it move but was not strenuously tested by the environment. For example the plane it walks on is flat and there are no physical obstacles that the character encounters. The values for k_p and k_d were selected by hand based off the fact that they allowed for large enough movement of the links to get walking behaviour. The balance feedback terms were chosen by first trying to optimize the target angles and PD parameters as much as possible and then manually tuning the balance terms such that they improved the motion. Weird behaviour is seen if the character falls or gets out of the pattern of motion. Often times the character will fall and flail about on the ground but other times the character will freeze in an odd position where it seems like neither the simulation nor gravity are actually working. As mentioned above trying to debug this problem lead to an unresolved issue with PyBullet where the simulator was behaving weirdly and I couldn't figure out how to resolve it.

4 Conclusion

In conclusion this project shows that the SIMBICON framework can be reimplemented in a newer physics simulator with the same results. Further it shows that the techniques for creating a biped that moves in two dimensions in a stable, balanced way are valid and an excellent place to start in building more complex motions in higher dimensions.

Table 3: Target Angles

State	Target Angles	Angle(rad)
0,1,2,3	Torso	0
0,2	Swing Hip	0.7
	Swing Knee	-1.3
	Swing Ankle	0.2
	Stance Knee	-0.05
	Stance Ankle	0.2
1,3	Swing Hip	-0.7
	Swing Knee	-0.05
	Swing Ankle	0.2
	Stance Knee	-0.1
	Stance Ankle	0.2

Table 4: Parameters

State	Parameter	Value
0,1,2,3	k_p	800 Nm/rad
0,1,2,3	k_d	80 Nm/rad
0,2	c_d	0
	c_v	0.2
1,3	c_d	2.2
	c_v	0

5 Further Works

The simple biped locomotion control presented in this paper can easily be expanded into many avenues for further works. The first and most obvious of which is to control the motion in 3D instead of the 2D model presented here. Removing the constraint on the body would introduce another dimension of balance control that would need to be implemented and optimized, allowing for steps to the side to help keep the figure upright. When working with this 2D model one could use optimization routines to try and find the set of parameters which produces the longest upright walking time. Stochastic policy search, cyclic coordinate decent and covariance matrix adaptation are all methods that could be applied to finding the particular balance parameters, cd and cv for optimal walking. Another expansion of this model is different gaits and walking terrain. For example running, walking with high knees, jumping, and even more complex movements could be implemented using the finite state machine framework. Furthermore inclined or bumpy terrain as well as external forces acting on the body could be used to further push the balance and locomotion control. Finally models like these can be used to bridge the gap between simulation and reality. The locomotion control can be applied to real life robots and these models can help guide the way in which their locomotion is developed.

6 Acknowledgements

First and foremost thank you to Prof. Michiel Van de Panne for taking me on and sharing his time and knowledge to introduce me to this topic.

A special thank you to Kristen Ruhnke without whom this thesis would probably not have been written in any sort of reasonable time frame; your support and motivation have made all the difference.

Finally thank you to my parents who have offered endless encouragement over the years.

A Code

A.1 Character Representation

```
<mujoco model="legs">
<compiler angle="degree" coordinate="global" />
<default>
<joint stiffness="5" damping="5"/>
</default>
<worldbody>
<body name="base">
<joint armature="0" axis="1 0 0" damping="0" limited="false"
    name="ignorex" pos="0 0 0" stiffness="0" type="slide"/>
<joint armature="0" axis="0 0 1" damping="0" limited="false"
    name="ignorez" pos="0 0 0" stiffness="0" type="slide"/>
<joint armature="0" axis="0 1 0" damping="0" limited="false"
    name="ignorey" pos="0 0 0" stiffness="0" type="hinge"/>
<geom name="base" size="0.1 0.2 0.4" type="box" mass="15"/>

<!-- Beginning of right leg -->
<body name="right_hip">
```

```
<geom name="right_hip_geom" size="0.1" type="sphere" pos="0 -0.15 -0.4" mass="0.5"/>

<body name="right_thigh">
<joint axis="0 1 0" name="right_hip_to_right_thigh" pos="0 -0.15 -0.4" limited="true" range="-90 90" type="hinge"/>
<geom name="right_thigh_geom" fromto="0 -0.15 -0.5 0 -0.15 -0.8" size="0.05" type="capsule" mass="7"/>

<body name="right_knee">
<geom name="right_knee_geom" size="0.1" type="sphere" pos="0 -0.15 -0.8" mass="0.5"/>

<body name="right_shin">
<joint axis="0 1 0" name="right_knee_to_right_shin" pos="0 0 -0.8" limited="true" range="0 90" type="hinge"/>
<geom name="right_shin_geom" fromto="0 -0.15 -0.9 0 -0.15 -1.2" size="0.05" type="capsule" mass="5"/>

<body name="right_ankle">
<geom name="right_ankle_geom" size="0.1" type="sphere" pos="0 -0.15 -1.2" mass="0.5"/>

<body name="right_foot">
<joint axis="0 1 0" name="right_ankle_to_right_foot" pos="0 0 -1.2" limited="true" range="-15 90" type="hinge"/>
<geom name="right_foot_geom" pos="0.15 -0.15 -1.2" size="0.15 0.05 0.075" type="box" mass="1"/>

</body> </body> </body> </body> </body> </body>
<!-- Beginning of left leg -->
<body name="left_hip">
<geom name="left_hip_geom" size="0.1" type="sphere" pos="0 0.15 -0.4" mass="0.5"/>

<body name="left_thigh">
<joint axis="0 1 0" name="left_hip_to_left_thigh" pos="0 0.15 -0.4" limited="true" range="-90 90" type="hinge"/>
<geom name="left_thigh_geom" fromto="0 0.15 -0.5 0 0.15 -0.8" size="0.05" type="capsule" mass="7"/>

<body name="left_knee">
<geom name="left_knee_geom" size="0.1" type="sphere" pos="0 0.15 -0.8" mass="0.5"/>

<body name="left_shin">
<joint axis="0 1 0" name="left_thigh_to_left_shin" pos="0 0 -0.8" limited="true" range="0 90" type="hinge"/>
<geom name="left_shin_geom" fromto="0 0.15 -0.9 0 0.15 -1.2"
```

```
size="0.05" type="capsule" mass="5"/>

<body name="left_ankle">
<geom name="left_ankle_geom" size="0.1" type="sphere" pos="0 0.15 -1.2" mass="0.5"/>

<body name="left_foot">
<joint axis="0 1 0" name="left_ankle_to_left_foot" pos="0 0 -1.2" limited="true" range="-15 90" type="hinge"/>
<geom name="left_foot_geom" pos="0.15 0.15 -1.2" size="0.15 0.05 0.075" type="box" mass="1"/>

</body> </body> </body> </body> </body> </body> </body> </worldbody> </mujoco>
```

A.2 Main Program

```python
import pybullet as p
import pybullet_data
import time

physicsClient = p.connect(p.GUI)  # or p.DIRECT for non-graphical version
p.setAdditionalSearchPath(pybullet_data.getDataPath())  # used by loadURDF
planeId = p.loadURDF("plane.urdf")
legsStartPos = [0, 0, 2]
p.setGravity(0, 0, -10)
legsStartOrientation = p.getQuaternionFromEuler([0, 0, 0])
legsID = p.loadMJCF("legs.xml")[0]
p.resetBasePositionAndOrientation(legsID, legsStartPos, legsStartOrientation)

# start camera view perpendicular to figure
p.resetDebugVisualizerCamera(cameraDistance=4, cameraYaw=0, cameraPitch=0, cameraTargetPosition=[0, 0, 1])


# joint and link numbers
base_link = 3

right_hip_joint = 5
right_knee_joint = 8
right_ankle_joint = 11
right_ankle_link = 10
right_foot_link = 12

left_hip_joint = 14
left_knee_joint = 17
```

```
left_ankle_joint = 20
left_ankle_link = 19
left_foot_link = 21

# target angles
base_angle = 0

swing_hip_0_2 = 0.7
swing_knee_0_2 = -1.3
swing_hip_1_3 = -0.7
swing_knee_1_3 = -0.05
swing_ankle_0_2 = 0.2
swing_ankle_1_3 = 0.2

stance_knee_0_2 = -0.05
stance_ankle_0_2 = 0.20
stance_knee_1_3 = -0.10
stance_ankle_1_3 = 0.2

# PD controller constants
kp = 800
kd = 80
cd_0_2 = 0
cv_0_2 = 0.2
cd_1_3 = 2.2
cv_1_3 = 0

# simulation parameters
time_step = 0.01
p.setTimeStep(time_step)
foot_contact_tol = 0.0001


def stance_leg_torque(swing_hip_torque):
    body = p.getJointState(legsID, base_link)
    tau_body = kp*(-base_angle - body[0]) - kd * body[1]
    tau_stance = -tau_body - swing_hip_torque
    return tau_stance


def balance_feedback(target_angle, stance_ankle_link, cd, cv):
    body = p.getLinkState(legsID, 3, computeLinkVelocity=1)
    body_position = body[0]
    body_velocity = body[6][0]
    ankle_position = p.getLinkState(legsID, stance_ankle_link)
        [0]

    d = body_position[0] - ankle_position[0]

```

```
    return target_angle + cd*d + cv*body_velocity


def in_contact(contact_points, contact_links):
    links_in_contact = []
    for contact_point in contact_points:
        links_in_contact.append(contact_point[4])

    for link in contact_links:
        if link in links_in_contact:
            return 1

    return 0


def set_torque_0():
    # swing leg
    right_hip = p.getJointState(legsID, right_hip_joint)

    # with feedback
    swing_hip_angle = balance_feedback(-swing_hip_0_2, left_ankle_link, cd_0_2, cv_0_2)
    tau_r_hip = kp * (swing_hip_angle - right_hip[0]) - kd * right_hip[1]

    # without
    # tau_r_hip = kp * (swing_hip_0_2 - right_hip[0]) - kd * right_hip[1]

    right_knee = p.getJointState(legsID, right_knee_joint)
    tau_r_knee = kp * (-swing_knee_0_2 - right_knee[0]) - kd * right_knee[1]

    right_ankle = p.getJointState(legsID, right_ankle_joint)
    tau_r_ankle = kp * (-swing_ankle_0_2 - right_ankle[0]) - kd * right_ankle[1]

    # stance leg
    tau_l_hip = stance_leg_torque(tau_r_hip)

    left_knee = p.getJointState(legsID, left_knee_joint)
    tau_l_knee = kp * (-stance_knee_0_2 - left_knee[0]) - kd * left_knee[1]

    left_ankle = p.getJointState(legsID, left_ankle_joint)
    tau_l_ankle = kp * (-stance_ankle_0_2 - left_ankle[0]) - kd * left_ankle[1]

    # update forces
```

```python
    p.setJointMotorControlArray(legsID, [right_hip_joint, right_knee_joint, right_ankle_joint, left_hip_joint,
        left_knee_joint, left_ankle_joint], controlMode=p.TORQUE_CONTROL, forces=[tau_r_hip, tau_r_knee,
        tau_r_ankle, tau_l_hip, tau_l_knee, tau_l_ankle])
    p.stepSimulation()


def set_torque_1():
    # swing leg
    right_hip = p.getJointState(legsID, right_hip_joint)
    # with feedback
    swing_hip_angle = balance_feedback(-swing_hip_1_3, left_ankle_link, cd_1_3, cv_1_3)
    tau_r_hip = kp * (swing_hip_angle - right_hip[0]) - kd * right_hip[1]

    # without
    # tau_r_hip = kp * (swing_hip_1_3 - right_hip[0]) - kd * right_hip[1]

    right_knee = p.getJointState(legsID, right_knee_joint)
    tau_r_knee = kp * (-swing_knee_1_3 - right_knee[0]) - kd * right_knee[1]

    right_ankle = p.getJointState(legsID, right_ankle_joint)
    tau_r_ankle = kp * (-swing_ankle_1_3 - right_ankle[0]) - kd * right_ankle[1]

    # stance leg
    tau_l_hip = stance_leg_torque(tau_r_hip)

    left_knee = p.getJointState(legsID, left_knee_joint)
    tau_l_knee = kp * (-stance_knee_1_3 - left_knee[0]) - kd * left_knee[1]

    left_ankle = p.getJointState(legsID, left_ankle_joint)
    tau_l_ankle = kp * (-stance_ankle_1_3 - left_ankle[0]) - kd * left_ankle[1]

    # update forces
    p.setJointMotorControlArray(legsID, [right_hip_joint, right_knee_joint, right_ankle_joint, left_hip_joint,
        left_knee_joint, left_ankle_joint], controlMode=p.TORQUE_CONTROL, forces=[tau_r_hip, tau_r_knee,
        tau_r_ankle, tau_l_hip, tau_l_knee, tau_l_ankle])
    p.stepSimulation()


```

```
def set_torque_2():
    # swing leg
    left_hip = p.getJointState(legsID, left_hip_joint)
    # with feedback
    swing_hip_angle = balance_feedback(-swing_hip_0_2,
        right_ankle_link, cd_0_2, cv_0_2)
    tau_l_hip = kp * (swing_hip_angle - left_hip[0]) - kd *
        left_hip[1]

    # without
    # tau_l_hip = kp * (swing_hip_0_2 - left_hip[0]) - kd *
        left_hip[1]

    left_knee = p.getJointState(legsID, left_knee_joint)
    tau_l_knee = kp * (-swing_knee_0_2 - left_knee[0]) - kd *
        left_knee[1]

    left_ankle = p.getJointState(legsID, left_ankle_joint)
    tau_l_ankle = kp * (-swing_ankle_0_2 - left_ankle[0]) - kd *
         left_ankle[1]

    # stance leg
    tau_r_hip = stance_leg_torque(tau_l_hip)

    right_knee = p.getJointState(legsID, right_knee_joint)
    tau_r_knee = kp * (-stance_knee_0_2 - right_knee[0]) - kd *
        right_knee[1]

    right_ankle = p.getJointState(legsID, right_ankle_joint)
    tau_r_ankle = kp * (-stance_ankle_0_2 - right_ankle[0]) - kd
         * right_ankle[1]

    # update forces
    p.setJointMotorControlArray(legsID, [right_hip_joint,
        right_knee_joint, right_ankle_joint, left_hip_joint,
         left_knee_joint, left_ankle_joint], controlMode=p.
             TORQUE_CONTROL, forces=[tau_r_hip, tau_r_knee,
         tau_r_ankle, tau_l_hip, tau_l_knee, tau_l_ankle])
    p.stepSimulation()


def set_torque_3():
    # swing leg
    left_hip = p.getJointState(legsID, left_hip_joint)
    # with feedback
    swing_hip_angle = balance_feedback(-swing_hip_1_3,
        right_ankle_link, cd_1_3, cv_1_3)
    tau_l_hip = kp * (swing_hip_angle - left_hip[0]) - kd *
        left_hip[1]
```

```python

    # without
    # tau_l_hip = kp * (swing_hip_1_3 - left_hip[0]) - kd * left_hip[1]

    left_knee = p.getJointState(legsID, left_knee_joint)
    tau_l_knee = kp * (-swing_knee_1_3 - left_knee[0]) - kd * left_knee[1]

    left_ankle = p.getJointState(legsID, left_ankle_joint)
    tau_l_ankle = kp * (-swing_ankle_1_3 - left_ankle[0]) - kd * left_ankle[1]

    # stance leg
    tau_r_hip = stance_leg_torque(tau_l_hip)

    right_knee = p.getJointState(legsID, right_knee_joint)
    tau_r_knee = kp * (-stance_knee_1_3 - right_knee[0]) - kd * right_knee[1]

    right_ankle = p.getJointState(legsID, right_ankle_joint)
    tau_r_ankle = kp * (-stance_ankle_1_3 - right_ankle[0]) - kd * right_ankle[1]

    # update forces
    p.setJointMotorControlArray(legsID, [right_hip_joint, right_knee_joint, right_ankle_joint, left_hip_joint,
        left_knee_joint, left_ankle_joint], controlMode=p.TORQUE_CONTROL, forces=[tau_r_hip, tau_r_knee,
        tau_r_ankle, tau_l_hip, tau_l_knee, tau_l_ankle])
    p.stepSimulation()


# state definitions
def state0():
    t = time.time()
    while time.time() - t < 0.3:
        set_torque_0()
    state1()


def state1():
    no_contact = 1
    while no_contact:
        #swing leg right leg, stance left
        set_torque_1()
        contact_points = p.getContactPoints(bodyA=planeId, bodyB=legsID)
        #print("contact points: " + str(contact_points))
```

```
        if len(contact_points) > 0 and in_contact(contact_points, [right_foot_link, right_ankle_link]):
            no_contact = 0

    state2()


def state2():
    t = time.time()
    while time.time() - t < 0.3:
        set_torque_2()
    state3()


def state3():
    no_contact = 1
    while no_contact:
        # swing leg left leg, stance right
        set_torque_3()
        contact_points = p.getContactPoints(bodyA=planeId, bodyB=legsID)
        #print("contact points: " + str(contact_points))
        if len(contact_points) > 0 and in_contact(contact_points, [left_foot_link, left_ankle_link]):
            contact_distance = contact_points[0][8]
            if contact_distance < foot_contact_tol:
                no_contact = 0
    state0()


# start the FSM with a small time delay to allow character to fall into place
t = time.time()
while time.time() - t < 0.5:
    p.stepSimulation()
state0()
```

B Code Documentation

B.1 Character Representation

Lines 8-10 are the most important in this representation since they are the ones that constrict the body to 2D. Here we construct the contraint out of three separate joints, one for each dimension. Along the x and z axis the worldbody is constrained with a slide joint, which gives the worldbody one degree of freedom along each axis, which results in unconstrained motion in the 2D x,z plane. Along the y axis we constrain the worldbody to a hinge joint, which gives the worldbody one degree of rotational freedom around the y axis. The overall effect of these joint constraints is that the worldbody is constrained to 2D motion where it can still fall forward and backward but not to the sides.

MuJoCo uses a nested representation to define their kinematic tree of parent/child relationships. Here the parent/child relationship is enforced by the child relationship already present in XML. So to be a child of a link means you must be a child of your parents' link tag which means you must be a nested XML element. In this way we can define two children to be at the same level, for example left and right hips, while still allowing each of those children to have children of their own, for example left and right thighs. In contrast to URDF where each link is defined separately and you must specify joints to link them together, if no joint is specified between a parent and its' child a fixed joint is generated by default. At the highest level is the worldbody which in our case is the torso, now constrained to 2D and is the only part of the character that exists in relation to the world frame directly.

The `<joint>` tag: In this tag we can define which axis the rotation occurs around and where the joint is located. It is here that we also limit the joints, with the joint limits given in degrees. The limits were selected to mirror the true limits those joints experience on the human body.

The `<geom>` tag: here we finally define the geometries that make up the visual components of the character. It is very important to note that capsules must be used to make up the main portions of the legs rather than rectangles. Initially they were made of rectangles and there was very weird behaviour in the legs due to internal collisions between the legs and sphere "joints" which were only resolved when the geometry was changed. The feet could however remain rectangles since they did not seem to experience this behaviour.

B.2 Main Program

Throughout this piece of code $getJointState(legsID, joint_index)$ returns an array of which we are interested in: index 0: the current velocity of the queried joint, and index 1: the current position.

The first 15 lines of code are necessary for loading the character and plane in as MuJoCo and URDF files respectively. As well as starting the legs slightly above the ground otherwise it loads the character midway through the plane and you have weird contact collisions happen right at the beginning of the simulation. Finally we start the camera perpendicular to the character to better see its' motion when walking.

Lines 21-31 are static values for the link and joints labels needed for the simulation. Figures 2 and 3 show how these are labelled for character used. These values can be determined dynamically by iterating over all values from 0 to getNumJoints(legsID) and calling getJointInfo() for each joint, or getLinkState() for each link although in the link case there is no way to get the total number of links in the character. I determined these values via guessing and checking and decided to simply store them statically since the character's representation does not change.

Lines 33-54 are all the parameters needed to run the simulation, they are stored in variables so that modifying them is easy as there is only one place in the code to change the values. The parameters correspond to the values in Tables 3 and 4.

Lines 56-59 contain parameters used in the simulation. The time step is used by the integrators and the $foot_{c}ontact_{t}ol$ is used as a tolerance value for determining if a foot has made contact with the plane.

`def stance_leg_torque(swing_hip_torque)` is the first helper function whose job is to calculate the torque for the stance hip using the method discussed in Subsection 2.2 and the implementation closely follows Equation 3.

`def balance_feedback(target_angle, stance_ankle_link, cd,cd)` in this helper function we are computing the balance feedback as given in Equation 4. Here we approximate d by the difference in the x coordinates of the base and the ankle's center of mass as the positions are given in 3D coordinate space.

`def in_contact(contact_points, contact_links)` this is a simple helper function who takes in a list of links currently in contact with the plane and outputs true if any of the links are are interested in $(contact_{l}inks)$ is in contact with the plane. This function is used to determine if the foot or ankle is in contact with the plane for states 1 and 3.

`def set_torques_0()` is the main function in charge of calculating and setting all the torques for state 0. `def set_torques_1(), def set_torques_2()` `def set_torques_3()` perform the same functions for their respective states so for brevity only state 0 will be talked about in depth. Torques are calculated for each of the joints by getting the current state of the joint and using Equation 1. The final steps are to update the torques being applied to the various joints and stepping the simulation to apply them.

`def state0() def state2()` these two functions control the finite state machine for states 0 and 2. In these states the transition to the next state is based on an interval of time passing. During this time, the torques are repeatedly updated to move the character.

`def state1() def state3()` these two functions control the finite state machine for states 1 and 3. In these states the state transition is governed by the foot making contact with the plane. Here we check to see if either the ankle or the foot is in contact with the plane and if yes we move on to the next state, otherwise we keep updating the torques. We check for both the ankle and the foot since there were issues with it not always recognizing the foot was in contact with the ground. These states are ones in which the simulation often gets stuck since it will continuously wait for the foot contact which may not always occur.

Line 264-267 Finally we are at the point in the program where the finite state machine is started. There is a small loop before we call $state0()$. This is to allow the character to fall until it comes in contact with the plane. This is because we initialized the character to load in above the plane, however if we immediately start the walking process while the character is in the air it has the potential to land on the ground in a funny position and the entire walking cycle is thrown off.

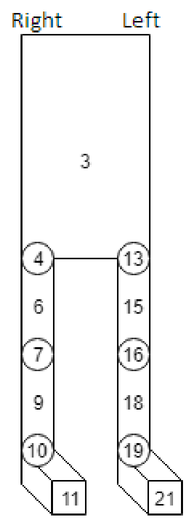

Figure 2: Diagram of how the links are labelled on the character.

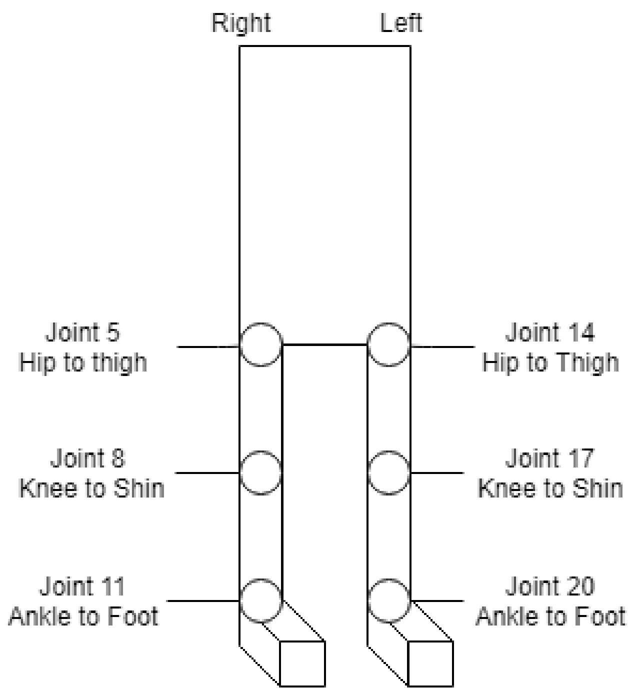

Figure 3: Diagram of how the joints are labelled on the character.

References

[1] Erwin Coumans and Yunfei Bai. Pybullet and bullet. Available at `https://pybullet.org/wordpress/` (2018/03/25).

[2] Charbel Fares and Yskandar Hamam. Collision detection for rigid bodies: A state of the art review. *GraphiCon 2005*, 2005.

[3] T. Geijtenbeek and N. Pronost. Interactive character animation using simulated physics: A state of the art review. *Computer Graphics Forum*, 31(8):2492–2515, 2012.

[4] Open Source Project. Unified robot description format (urdf). Available at `http://wiki.ros.org/urdf` (2018/04/01).

[5] Russell Smith. Open dynamics engine. Available at `http://www.ode.org/` (2018/03/25).

[6] Emo Todorov. Mujoco. Available at `http://www.mujoco.org/` (2018/03/25).

[7] KangKang Yin, Kevin Loken, and Michiel van de Panne. Simbicon: Simple biped locomotion control. *ACM Trans. Graph.*, 26(3):Article 105, 2007.