Simulation Development of a Bipedal Robot

March 13, 2018

Technion - Israel Institute of Technology

Faculty of Mechanical Engineering

Final Project

Yarden As
305201360
yardenas@campus.technion.ac.il
Nadav Mailhot
302288865
mailhotn@campus.technion.ac.il

Contents

Introduction

After writing about the assumptions we made in order to develop a simulation for a Kneed Walker robot in the midterm report, we've implemented a simulation that takes care of the hybrid dynamics of the robot, a generic *Controller* class and a *Terrain* class and a *Simulation* class that integrates the former classes. As said, further theoretical information about how we derived the equations of motion can be found in the midterm report.

To help future generations of the SMILe lab use this simulation of the Kneed Walker for further research, we've written this report that serves as a thorough documentation and explains the details of our implementation.

1 Getting Started

1.1 Recommended method

The easiest way to manage different versions of the code is using git, therefore:

```
git clone https://github.com/YardenAs/Kneed-Walker.git <Working DIR>
```

One can also choose to work with a git GUI (our favorite is GitKraken) and clone the Kneed-Walker repository.

1.2 The non-git method

Go to the Kneed-Walker GitHub repository and download the Kneed-Walker Repository to your working directory.

1.3 Some words about the branches

The Kneed-Walker repository holds 3 different branches:

1. master branch - this branch is the stable version of the Kneed-Walker.
2. CompassBiped - this branch essentially constrains the knees and makes sure that the swing leg doesn't hit the floor when it shouldn't. Although this branch **is stable** and can be trusted, it switches the coordinates of the swing and stance leg at each impact (as in CoordinateSwitch) instead of toggling the constraints (as in master branch) so the state vector has different meaning: x and y coordinates denote the position of the stance leg and not the position of the left leg with respect to the walking direction as in master branch.
3. CoordinateSwitch - in this branch, instead of toggling the ground contact constraints from the left and the right leg, at each ground impact the state values (angles and angular velocities) are switched between the stance leg and the swing leg. This branch **is not stable** and most most likely has some bugs. We decided to keep this branch because some of the work found in the literature (mainly Grizzle) uses this method.

2 The *KneedWalker* Class

The *KneedWalker* class essentially holds all of the inertial and geometric information of the Kneed-Walker.

2.1 The *KneedWalker* class properties

sh This is a 1×3 vector that holds (with this specific order) the shank mass, length and moment of inertia.

th Thigh mass, length and moment of inertia

to Torso mass, length and moment of inertia

grav acceleration of gravity

Order State space order (14)

FallThresh A threshold that helps identify if the robot fell

Support The current stance leg (left of right)

BadImpulse This is a flag that rises when the impulse of the detaching leg does not point outwards from the ground

BadLiftoff This is a flag that rises when the detaching leg's velocity does not point outwards from the ground

Torques The current torques at the Kneed-Walker joints

nEvents The number of $ode45$ termination events

link_width Render model link's width

link_color Render model link's color

RenderObj Render model object

LinkRes Render model resolution

LineWidth Render model link's width

2.2 The $KneedWalker$ class methods

2.2.1 $KneedWalker$ constructor

This function creates a new instance of a *KneedWalker* class.

KneedWalker() creates a new instance of a *KneedWalker* object out of default properties.

KneedWalker(sh, th, to) creates a new instance of a *KneedWalker* object out of the given **sh, th** and **to** vectors.

2.2.2 Forward kinematics

GetPos is used to get the forward kinematics of various points on the robot.

Pos = GetPos(KW, X, which) Pos is a 1×2 vector of the wanted point, KW is a *KneedWalker* class instance and which tells the function the wanted point (Lankle, Lknee, Rankle, Rknee, TorsoCOM, TorsoEnd and Hip.) X is the state vector of the robot at a specific time.

2.2.3 Velocities

GetVel calculates the velocity of a certain point on the robot.

Vel = GetVel(KW, X, which) Vel is a 1×2 vector that holds the velocity of the wanted point in the x-y directions. X is the state vector of the robot at a specific time.

2.2.4 Dynamic equations

DynEq is a function that computes the numeric values of the Kneed-Walker dynamic equations given the state vector X at a specific time. As stated in the midterm report, the dynamic equations are represented in a matrix form.

[M, B, G, W, Wdot, Fq] = DynEq(KW, X) where M is the inertia matrix, B is the centripetal and Coriolis forces vector, G is the conservative forces (only gravity in our case) vector, W and Wdot are the constraints matrices and Fq are the external generalized forces, namely motors torques.

2.2.5 Ground reaction forces

GetReactionForces is a function that calculates the ground reaction forces with the exception of impacts using the state vector X of the Kneed-Walker.

F = GetReactionForces(KW, X) F are the forces in the x-y directions, X is a state vector at a specific time of the Kneed-Walker.

2.2.6 State derivatives

Derivative function is used by the *Simulation* class and later on by $ode45.m$ to calculate the derivatives of the state given the time t and the state vector X. This function uses *DynEq* function to get the numerical values of the dynamics of the robot according to the state and translating it to the form $\dot{X} = f\left(X\right)$ needed by $ode45.m$

Xdot = Derivative(KW, t, X) where t is the time sample, X is the state vector at time t and KW is an instance of a *KneedWalker* class.

2.2.7 Simulation terminating events

In order to detect ground impact, falling of the robot and other situations where $ode45.m$ needs to be terminated (this does not necessarily mean that the simulation stops), we use *Events* function which is called by the *Simulation* class and $ode45.m.$ the current events are (with their event index):

1. leg contact - swing leg impacts the ground.
2. robot fell - for more info on how this is implemented visit the Events function.
3. $|\alpha| \geq {}^{\pi}/_{6}$ - torso rotated too much
4. Left knee lock - we assumed that the knees cannot lock and the $ode45.m$ should terminate when this occurs but it is possible to modify the code and let the knees remain locked.
5. Right knee lock
6. Left leg 45° - left leg spreads too much
7. Right leg 45°

Note that is is possible (and sometimes recommended) to add more events that suits your needs. As said before, all events terminate the current $ode45.m$ run **but** the only event that doesn't stop the simulation is event #1 (after leg contact with the ground the support leg constraints are swapped.) Therefore, if you do add **simulation terminating** events, make sure that their event index is larger than 1.

[value, isterminal, direction] = Events(KW, X, Floor) Floor is an instance of *Terrain* class used to detect ground impact, X is the state vector of the robot and KW is an instance of a *KneedWalker* class. For further information about the output of this function, go to mathworks documentation.

2.2.8 Impact calculations

At each leg impact with the ground, *HandleEvents* function calls *CalcImpact* which in turn calculates the state vector of the robot after the impact given the state vector before the impact. For further information about the dynamic model behind this calculation, see the midterm report.

[Xf, Lambda] = CalcImpact(KW, Xi) calculates the state vector Xf after the impact along with the ground impact impulses in the x and y directions given the state Xi of the robot prior the impact.

2.2.9 Events handling and decision making

After each event the simulation needs to determine what to do next, therefore at each *ode*45.*m* termination (which occurs at each event) we call through *Simulation* to *HandleEvents*. If the event is a ground contact event, *HandleEvents* calls *CalcImpact*, toggles between the support leg and rises **BadImpulse** or **BadLiftoff** flags as needed.

Xf = HandleEvents(KW, iEvent, Xi, Floor) handles the event using an instance of a *Terrain* class, the state vector Xi prior the event, the index of the event iEvent and an instance of a *KneedWalker* class.

2.2.10 Energy calculation

E = GetEnergy(KW, X) calculates the energy of the robot using the state vector X of the robot and an instance of a *KneedWalker* class.

2.2.11 Setting the motors torques

KW = SetTorques(KW, T) is an encapsulating function that sets the motors torques using torques vector T and an instance of a *KneedWalker* class. usually this function is called from *Simulation* that receives T as an output from the *Controller*.

3 The *Simulation* Class

The *Simulation* class essentially manages the simulation in terms of handling both the *Controller* and *KneedWalker* events, transferring the control output from the *Controller* to *KneedWalker* , the state of the robot from *KneedWalker* to *Controller* and the *Terrain* height to *KneedWalker*.

3.1 The *Simulation* class properties

Mod is an instance of *KneedWalker* class

Con is an instance of *Controller* class

Env is an instance of *Terrain* class

ModCo is a 1 : *Model Order* vector

ConCo is a 1 : *Controller Order* vector

nEvents is the total (*Controller* and *KneedWalker*) number of events

ModEv is the number of *KneedWalker* events

ConEv is the number of *Controller* events

3.2 The *Simulation* class methods

3.2.1 The *Simulation* constructor

This function creates a new instance of *Simulation* class.

Simulation(Mod, Con, Env) creates a new *Simulation* object based on Mod (a *KneedWalker* object), Con (a *Controller* object) and Env (a *Terrain* object.)

Simulation() creates a new *Simulation* object based on the default constructors of *KneedWalker*, *Controller* and *Terrain*.

3.2.2 Derivative function

Xt = Derivative(sim, t, X) is a function used by *ode45.m* to calculate the next state vector using the current state vector. X is the state of the controller and the robot at each time sample t. the *Derivative* function also sets the robot's motors torques based on the control output using *KneedWalker*'s *SetTorques* method.

3.2.3 Events function

[value, isterminal, direction] = Events(sim, t, X) is a function used by *ode45.m* as a listener that detects if any of the given events occurred. This function uses the *KneedWalker* and *Controller Events* functions.

4 The *Controller* Class

The *Controller* class is in charge of getting the state of the robot and calculating the controller output.

Currently the *Controller* architecture is based on two pulses at each step the Kneed-Walker takes. Although this architecture will obviously change in further research, the current implementation can be used as a starting point for future implementations.

4.1 The *Controller* class properties

Period is a vector of normalized pulse widths for each motor. We normalize the periods with the time it takes the robot finish a step.

Amp is a vector of amplitudes for each pulse

Phase is the normalized pulse starting time (0 is the beginning of the step, 1 is the end of the step)

omega is the step frequency

nEvents is the number of events the controller has

Order is the order of the differential equation that governs the controller

Kp is the proportional constant for the torso's PD controller.

Kd is the differentiator constant for the torso's PD controller

4.2 The *Controller* class methods

4.2.1 The *Controller* constructor

This function creates new instance of *Simulation* class.

Controller(omega, Amp, Phase, Period, PD_Con) creates a new *Controller* object out of omega, Amp vector, Phase vector, Period vector and PD_Con vector.

4.2.2 Derivative function

To make the controller work in a periodic manner we used *ode45.m* to integrate the state of the controller until it reaches 1 (*Events* function detects this event). The integration speed is governed by omega $\dot{X} = \omega \Rightarrow X = \omega t$.

Xdot = Derivative(C, t, X) is used by *ode45.m* through *Simulation* to make the controller work on a periodic way.

4.2.3 Events function

[value, isterminal, direction] = Events(C, t, X) is a function used by *ode45.m* as a listener that detects if the state of the controller has reached the value 1, if so, we call to *HandleEvents* through *Simulation*.

4.2.4 Handling the events

Xa = HandleEvent(C, iEvent, Xi, ConEv) sets the state X of the controller back to 0 at each cycle (that ends with *Event* function when the state reaches to 1). C is an instance of *Controller* class, iEvent is the index of event, Xi is the state before the event occured and ConEv is used to determine if the event is a *Controller* event or a *KneedWalker* event.

4.2.5 Broadcasting the output torques

Torques = Output(C, t , X) is used to calculate the torques based on the time sample t and the robot's state X.

5 The *Terrain* Class

We used the *Terrain* class written by Jonathan Spitz and used it to generate the floor for the robot. The *Terrain* class can generate various types of floor:

1. Inclined plane - we used only this type of floor.
2. Sinusoidal
3. Infinite parabola
4. Finite parabola

We urge the reader to read the very well documented *Terrain.m* file.

6 *RunScript.m*

This script does the following steps in order to run the simulation:

1. Initialize a *KneedWalker*, *Controller*, *Simulation* and *Terrain* objects.
2. Run an *ode45.m* session until the first event occurred.
3. If the event is not ground impact or a controller event - terminate the simulation and animate.
4. If the event is ground impact of a controller event - handle the event (i.e. switch legs etc) and continue simulating until the EndCond flag is up.
5. If EndCond flag is up - terminate the simulation and animate.

This framework can be used for future simulations and can be merged as a method of the *Simulation* class if wanted.

Furthermore, we used this framework in *GAScript.m* and *GA_Sim_KW.m* which is also an “off the shelf” script for optimizing a control signal that uses a two pulses per step profile for each motor.

7 Future Work

Currently the Kneed-Walker is fully working and provides a good starting point for further research at different research topics (reinforcement learning, genetic algorithms etc.)

Nevertheless, the code can evolve in various directions:

1. Releasing the “No Knee Lock” constraints assumption. Our simulation stops if one of the knees extends to it’s fully open position. Obviously a more realistic simulation can take care of this by adding some more states to the simulation (support knee locked and swing kneed free, swing knee locked and support leg free etc.)

2. Our simulation assumes that at impact the support leg instantaneously released from the floor and therefore there can be only one leg touching the floor each moment. A more realistic behavior can be achieved if a two-legs-touching-the-floor state is added. Furthermore, adding this dynamic state will eliminate the need to check the liftoff velocity and contact impulse (as in BadLiftoff and BadImpulse.)