Notes during AI Project:

* Using Python 2.7 version
* Installed the anaconda software
* Had to understand what the anaconda software is and why it is needed
  + “Anaconda software **helps you create an environment for many different versions of Python and package versions**… Furthermore, you may use Anaconda to deploy any required project with a few mouse clicks.”
  + “Anaconda is a distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment.”
* As things were outdated and the link to get miniconda 2.7

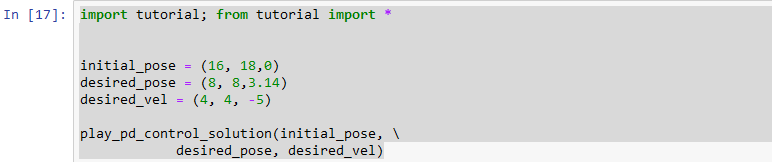
New notes:

Setup

* Using python 2.7 version
* Downloaded miniconda2 version 4.8.3
* Then created virtual environment which included packages:
  + **pyBox2D**: performs constrained rigid body simulation. It can simulate bodies composed of convex polygons, circles, and edge shapes.
  + **NumPy**: used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices.
  + **SciPy**: useful for solving many mathematical equations and algorithms. It is designed on the top of the Numpy library that gives more extension of finding scientific mathematical formulae like Matrix Rank, Inverse, polynomial equations, LU Decomposition, etc.
  + **MatPlotLab**: creating static, animated, and interactive visualizations in Python. Matplotlib makes easy things easy and hard things possible. Create publication quality plots. Make interactive figures that can zoom, pan, update.
  + **Jupyter**:interactive development environment for notebooks, code, and data. This allows us to open a jupyter environment using that has access to all of the libraries mentioned.
  + PyGame: the development of multimedia applications like video games using Python. It uses the Simple DirectMedia Layer library and several other popular libraries to abstract the most common functions, making writing these programs a more intuitive task

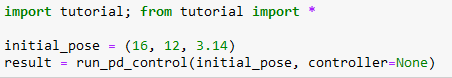
**PD CONTROL**

**PD - Control of a robot**

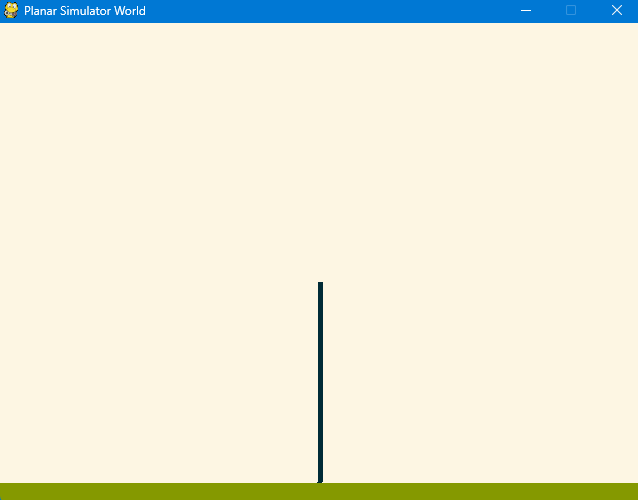
* We want to be able to control the paddle in any way that we want
* We are provided:   
  
* From this we learn how the robot or paddle moves in the environment, heres what each line tells us:
  + 1. Import tutorial; from tutorial import \*
    - This line imports the tutorial module, making all functions, classes, and variables available in the current scope. The from tutorial import \* part imports everything from tutorial, allowing direct access to its functions without needing to prefix them with tutorial..
  + 2. Initial\_pose = (x, y, theta)
    - Here, the variable initial\_pose is defined as a tuple representing the robot's initial position and orientation. First integer is the x coordinate (horizontal position) of the robot, the second is the y coordinate (vertical position) of the robot, and theta will represent the initial orientation angle (in radians) of the robot.
    - If its 16, 18, 0, then it will start near the center of the simulated space and will lay flat or horizontally.
  + 3. Desired\_pose = (x, y, theta)
    - The desired\_pose is another tuple defining the robot's target position and orientation. Like initial pose, the parameters serve the same purpose, however, the robot is now being directed to move to this x,y,theta position.
    - So, following the example from before, if the desired position is lets say (8,8,3.14), then the robot will go to the bottom left part of the screen with a full rotation of pi radians.
  + 4. Desired\_vel = (x\_vel, y\_vel, a\_vel)
    - x\_vel: horizontal velocity (meters per second)
    - Y\_vel: vertical velocity (meters per second)
    - A\_vel: angular velocity in radians per second, indicating that the robot should rotate counter clock wise
  + 5. play\_pd\_control\_solution (initial\_pose, desired\_pose, desired\_vel)
    - Initial pose: starting position and orientation
    - Desired pose: target position
    - Desired velocity: target velocities

**PD Control Part 1: running the simulation and accessing the robot state information**

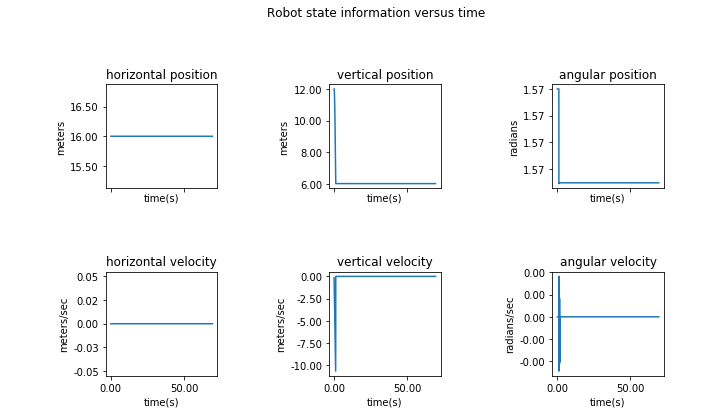
* Now we learn about how to utilize gravity and how objects can fall.



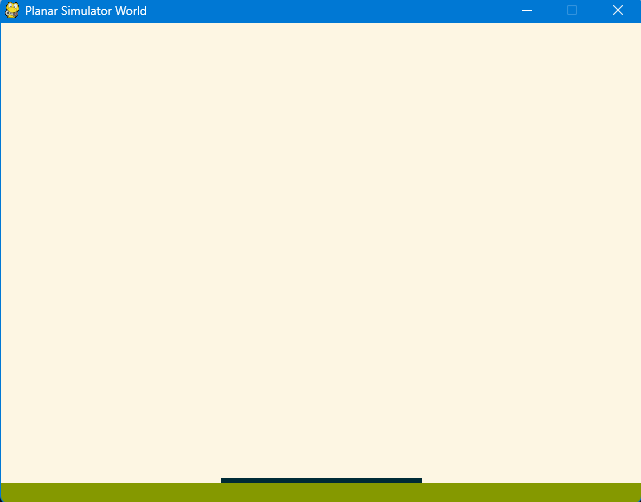
* + We can change the orientation of the initial\_pose by changing the last variable, 3.14/2 is given, this makes the robot stand vertically and then fall. If we can get it to just 3.14 then we will get it horizontally.
* After that, now we are able to rerun the simulation and plot the state of the robot as well.
  + I will show the differences of states when the orientation is changed as well
* First orientation: initial\_pose = (16, 12, 3.14/2.). I will provide the picture and the corresponding states. In this state, the vertical line just falls to the position that is capture in the picture below



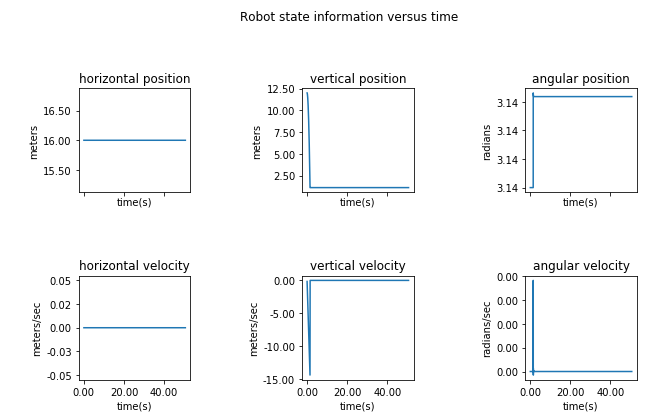
* Corresponding states:



* Second orientation: initial\_pose = (16, 12, 3.14)

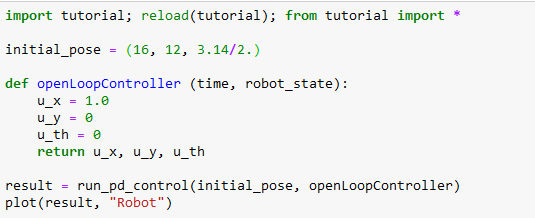


* Corresponding states:



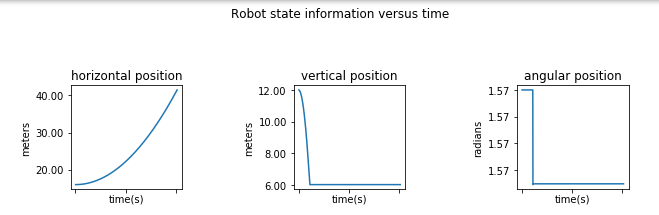
**PD Control Part 2: open loop control of the robot**

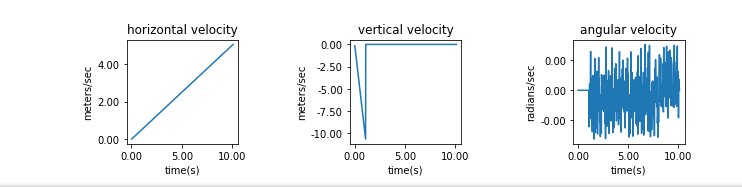
* We are also capable of moving the robot using open loop control. This can be done by applying force to the center of mass in the x or y direction. On top of that, we can apply angular torque about the center of mass.
* One of the inputs to the run\_pd\_control is currently set to None, in the following example we are going to show ho wto write a controller that gets run at ever time step.
* The output of the controller is u\_x, u\_y, u\_th, which is the amount of force applied in the x direction, y direction, and angular torque applied. The force is applied to the robot’s center of mass.

The following code:   


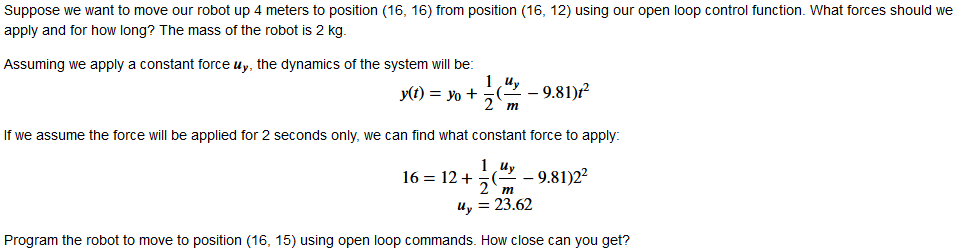
What happens: robot is vertical and falls, to the position seen previously. Instead of staying static in that position like the vertical example, it then begins to move to the right as if it there was a magnet pulling it. This is due to the u\_x being set to 1.0.

Corresponding states to the code after it is ran:



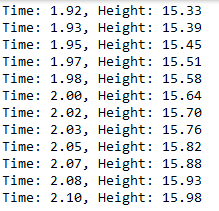


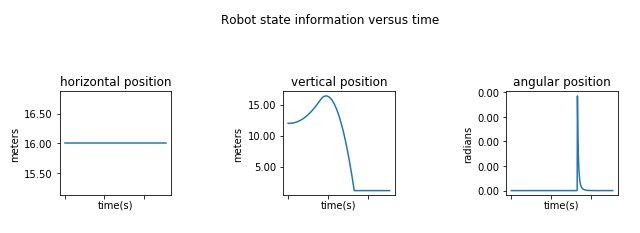
Note: even when the u\_y variable is increased, it does not do anything as there is gravity involved. So when it is set to 10, it has no visible affect. However, when u\_th is changed then it has the ability to rotate the robot and make it fall over, changing it from vertical to horizontal and then dragging across the floor due to the x axis.

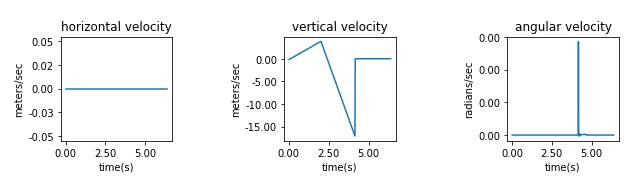
**Answering the following problem:   
**

We are given the following code as well:   


**This code gives us two things:**

1. The first thing the code does is generate our robot in the environment. The robot is spawned in horizontally (flat), it then floats up vertically for approximately 1 second to which it then just sinks to the ground.
2. Here is the corresponding Robot State information vs. time graphs:   
   



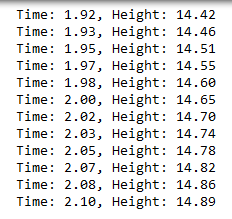


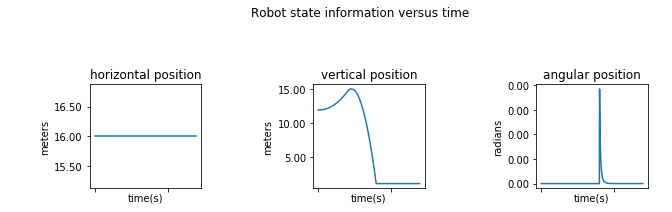
**Answer to the problem:**

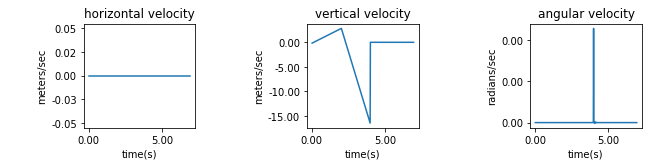
* The robot starts at (16,12) and should move to (16,15), with the given dynamics. We need to find the correct force to reach y=15 after applying the force for a specific amount of time. The code specifies a constant force of u\_y = 23.62 for 2 seconds to reach the (16,16) mark. Since we are trying to get y = 15, the following changes will be made to get us closer to that target.
* The first thing we will change to the code is the force, we need it to be slightly lower than 23.62. To determine this we will use the given formula and plug in 15 instead of 16. We can find this in the following way:
  + Given y(t) = y\_0 + ½(u\_y/m - 9.81) \* t^2, as we are aiming to move the robot upward by ~3 meters (12 to 15 instead of 16) let's solve for the required force (u\_y) when the force is applied for t = 2 seconds.
  + First lets compute the required acceleration: s = ½ \* a\*t^2 -> 3 = 1/2 \* a \* (2)^2 -> a = 1.5m/s^2
  + Then we should compute the required force, g is gravity and holds the value of 9.81: a = u\_y/m - g (gravity) -> u\_y = m(a + 9.81) -> u\_y = (1.5 + 9.81) = 22.62 is our corrected constant force.

This gives us the following results:

1. Robot does the same thing as described before, however it is at a slightly lower height to which it then drops to the floor.
2. Corresponding robot state information versus time:

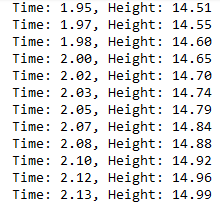


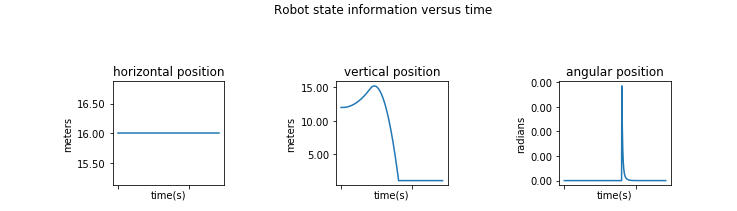


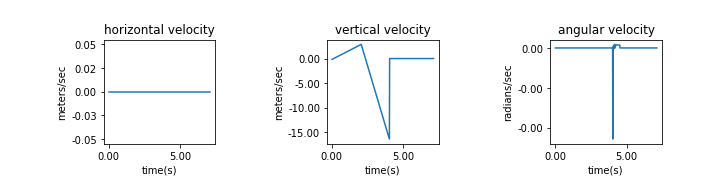


**Final Answer:**

We want to stop precisely at 15 meters, not 14.89… so we need to adjust the time that the force is applied for as well if we want to get even closer! Therefore, through trial and error I came ot the conclusion that with the constant force of 22.62, if we apply it for 2.035 seconds then we will get the closest possible result of 14.99. The code and results are listed below:

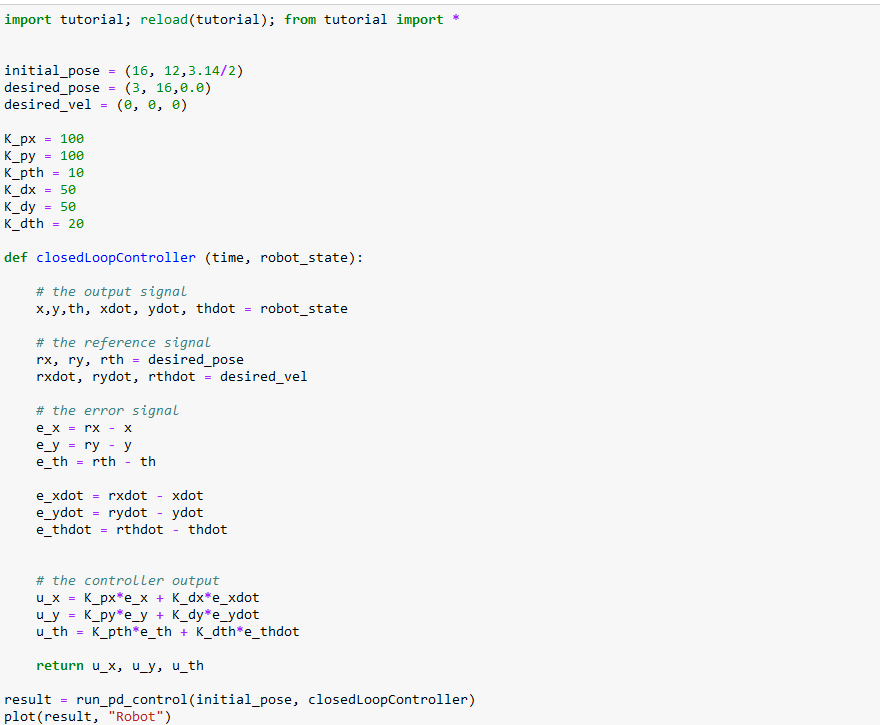






PD Control Part 3: feedback control of the robot (continued)

Given the following code:



* Observation of the given code: Horizontal robot moving from the center to the top left of the screen turning clockwise.
* Explanation of the code:

1. We have the initial pose, desired pose, and desired velocity as we have seen before
2. Then we have K\_px and K\_py, they are going to affect the robots movement along the x and y axes, respectively. They will determine how strongly the robot is reacting to errors in position along the x and y axes. K\_pth will control the robots rotational movement towards the desired orientation. Higher K\_pth will result in faster rotation to align with the target orientation.
3. K\_dx and K\_dy are the derivative gain, they will adjust the robots control inputs based on its velocity eros along the x and y axes. If the value of K\_d is higher it will dampen the robot’s linear motion, preventing it from moving too quickly and overshooting. K\_dth adjusts the rotational control input based on the error in angular velocity. It will prevent the robot from rotating too quickly past the desired orientation.
4. Quick summary of the closedLoopController: it calculates the control inputs u\_x, u\_y, and u\_th to reduce the error between the current and the desired state.
5. Problem: Try using different gains. See if you can observed system response behavior, such as:

Definitions:

* 1. **Under damped** - The system oscillates around the setpoint before settling.
  2. **Damped** - The system returns to the setpoint as quickly as possible without oscillating.
  3. **Overdamped** - The system returns to the setpoint without oscillating but slower than in the critically damped case

**Underdamped Behavior Example - An underdamped system responds quickly but overshoots the target and oscillates before settling.**

Changed the gains to:

* K\_px = 150,
* K\_py =150,
* K\_pth = 10,
* K\_dx = 10 ,
* K\_dy = 10,
* K\_dth = 5

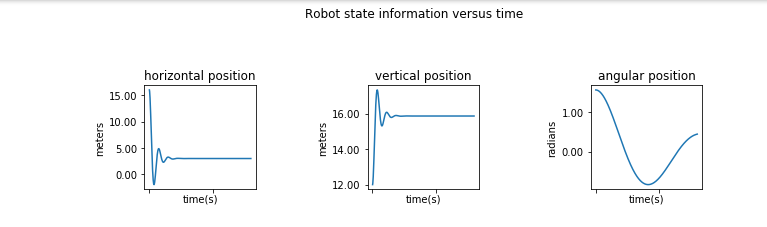
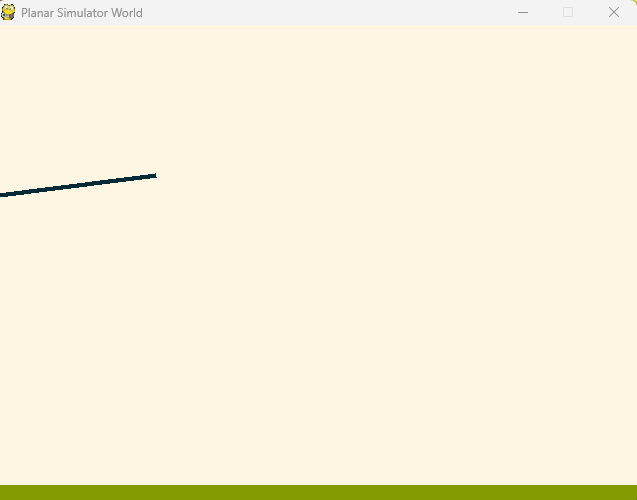
What are we doing:

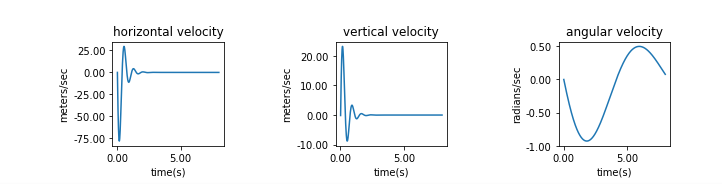
* Increased proportional gains (K\_px and K\_py) to 150
* Decreased derivative gains (K\_dx and K\_dy) to 10
* Slightly reduced K\_dth to 5.

What we want to see:

* Robot moves rapidly towards the desired position
* It is overshooting significantly, robot is oscillating around the set point before it eventually settles.

Results: robot in the top right corner rocking oscillating back and forth





**Critically Damped Behavior - A critically damped system reaches the setpoint in the shortest time without oscillating.**

* Changed the gains:
* K\_px = 100,
* K\_py =100,
* K\_pth = 20,
* K\_dx = 40 ,
* K\_dy = 40,
* K\_dth = 20

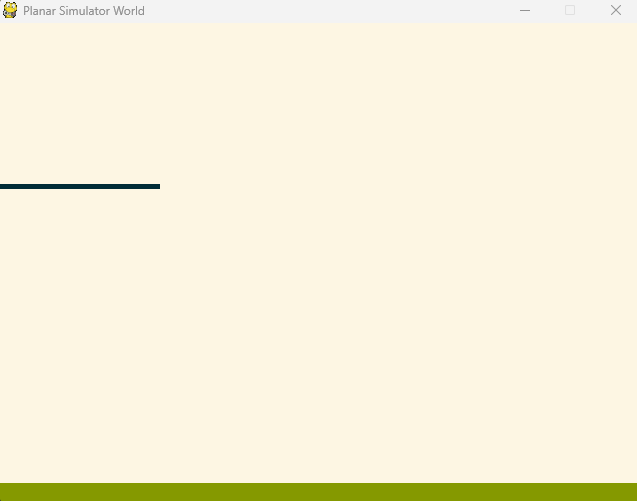
What is happening:

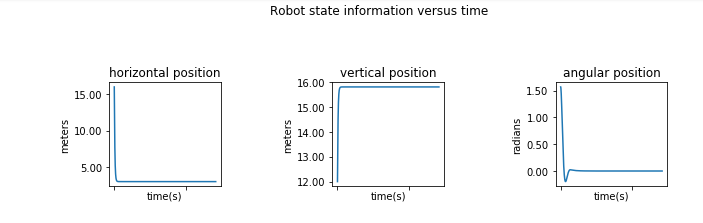
* Increased derivative gains (K\_dx and K\_dy) to 40.
* K\_p gains stay at 100

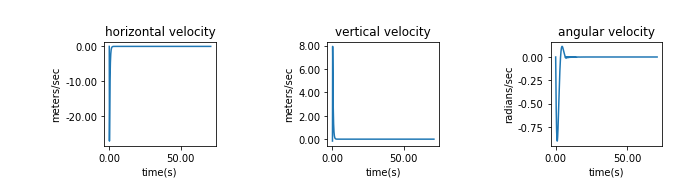
What we want to see:

* The robot moves swiftly towards the desired position without overshooting.
* No oscillations are observed.
* Minimized settling time

Results: robot in the top right corner, it settled in less than 2 seconds.







**Overdamped Behavior**

* Changed the gains:
* K\_px = 50,
* K\_py =50,
* K\_pth = 5,
* K\_dx = 100,
* K\_dy = 100
* K\_dth = 50

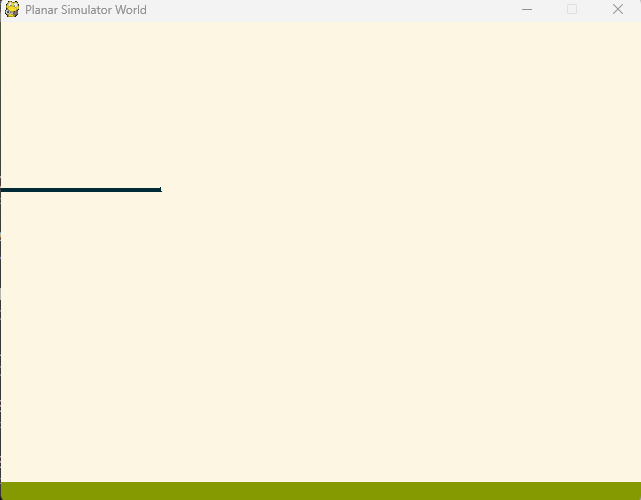
What is happening:

* Reduced K\_p gains to 50
* Significantly increased K\_d gains to 100.

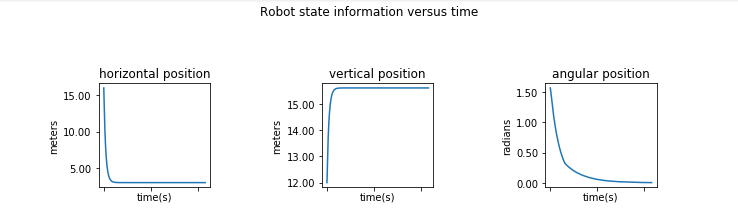
What we want to see happen:

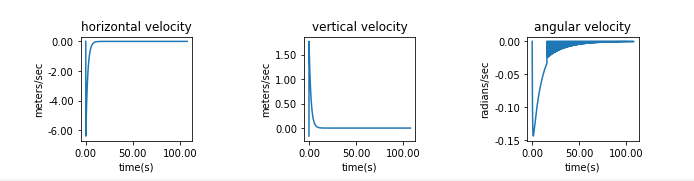
* Robot moves very slow towards the desired position
* We do not overshoot or oscillate
* Takes forever to settle because of all the damping

**Results:**

****

Moved to the top right of the screen slowly and took forever to settle as seen in the graphs, i took nearly 100 seconds.

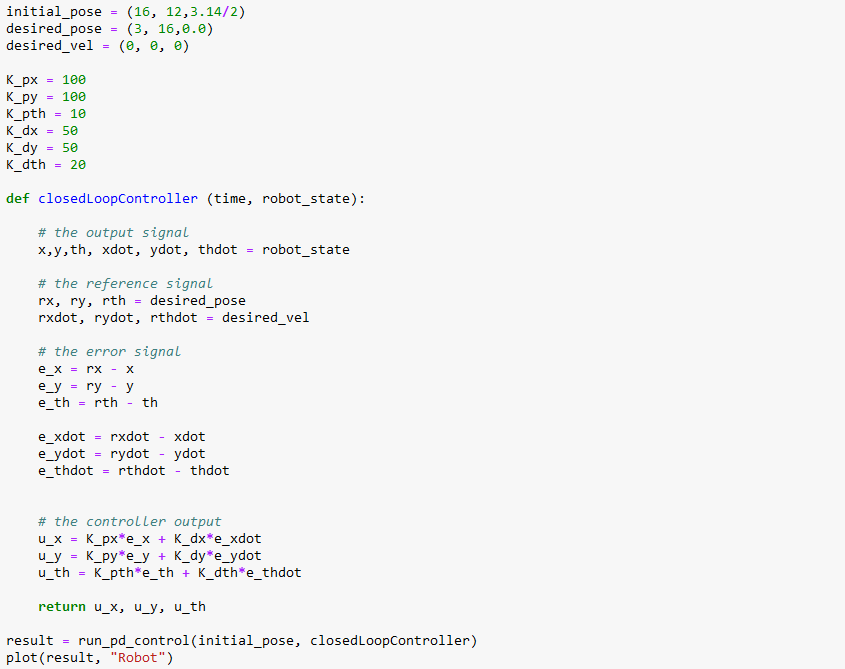




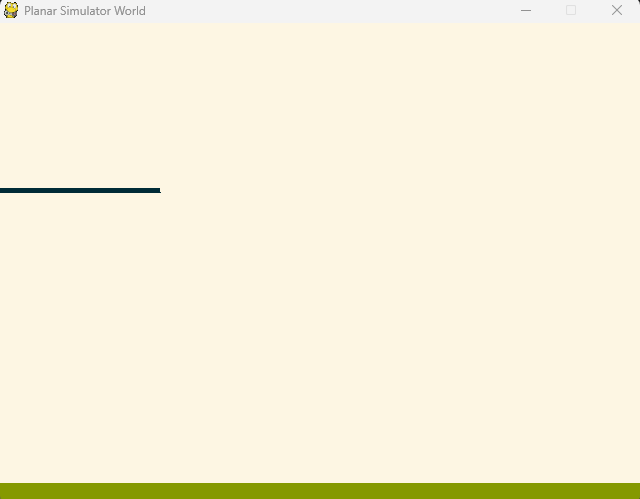
1. Problem - Improve upon your controller by adding a derivative term. In this case the reference signal for the derivative terms should be equal to 0.

* Answer: Given the formula: u = K\_pose \* (X\_desired - X\_current) + K\_d \* (X\_desired - X\_current), where u is the output, K\_d is the derivative gain and X\_desired is our reference signal (0 in this case). We will have such integers rxdot, rydot, and rthdot = 0, 0, 0.
  + In short, we have a proportional controller, we want to improve its performance by adding a derivative term, making it a proportional-derivative controller. The derivative term will help reduce overshooting. It will also improve the stability of the system by considering the rate of of change of error.

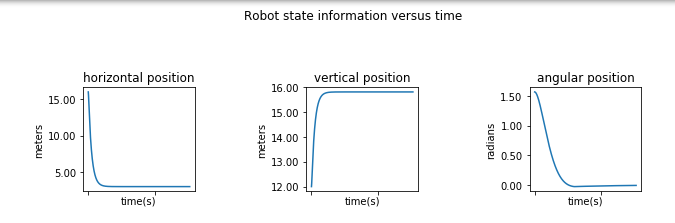
Code: improved controller to make it a Proportional-Derivative Controller by the addition of the u\_x, u\_y, and u\_th variables as the derivative terms. We therefore enhanced the controllers responsiveness to the changes in the system’s state.

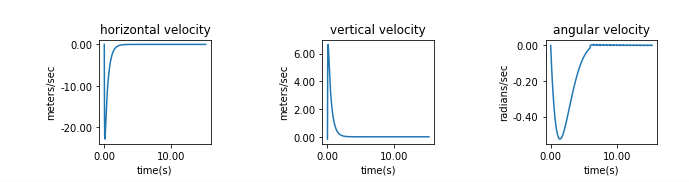


**Results:**



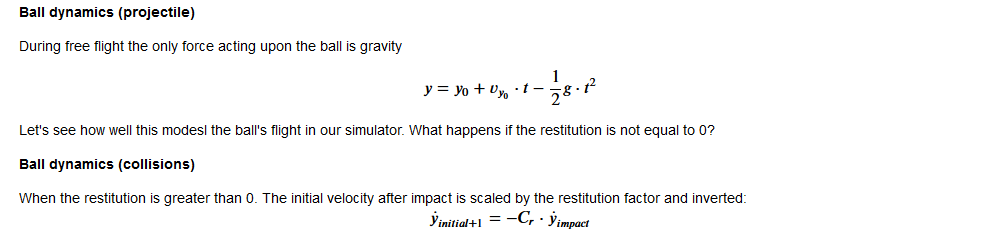
Observation: Quickly moved to desired location, and was very fast in settling.



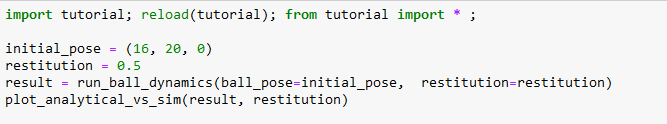
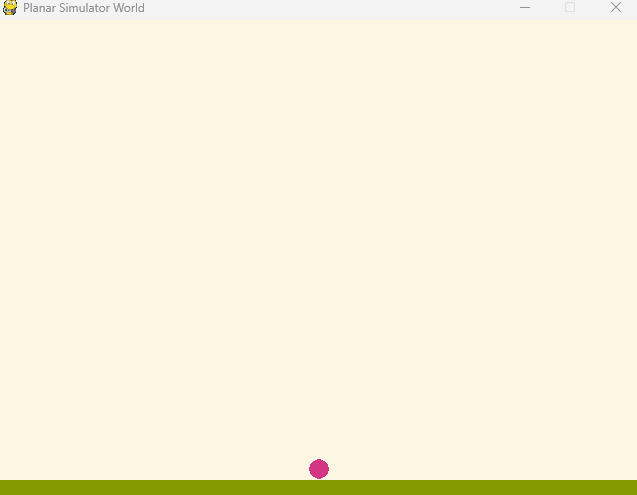
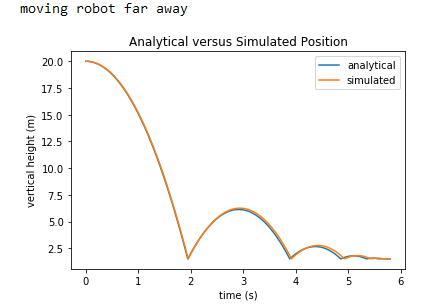


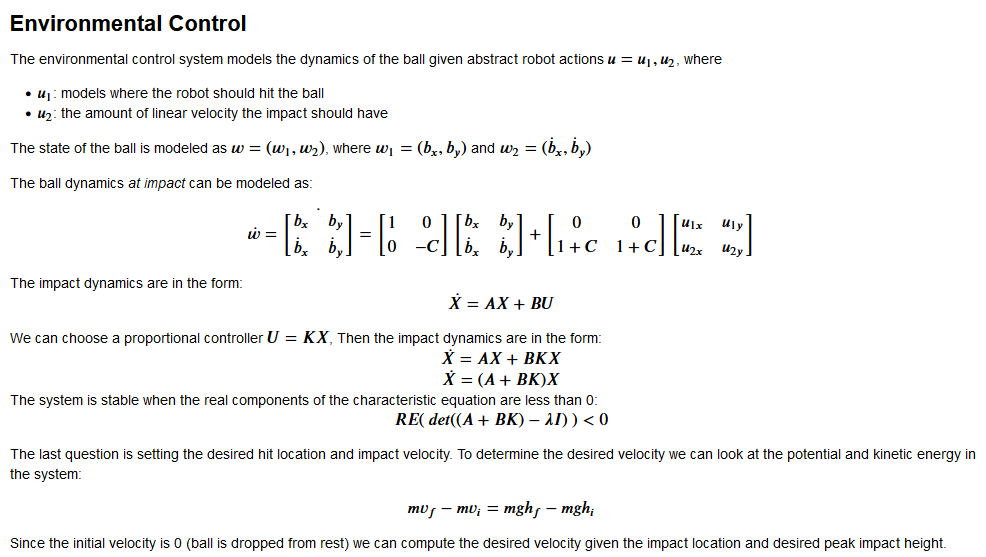
**Robot Juggling**

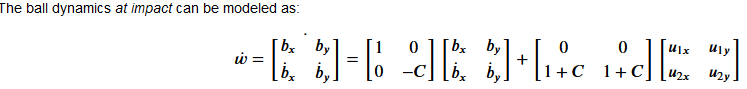
Now that we have a controller for the robot we are now going to focus on the problem of robot juggling.

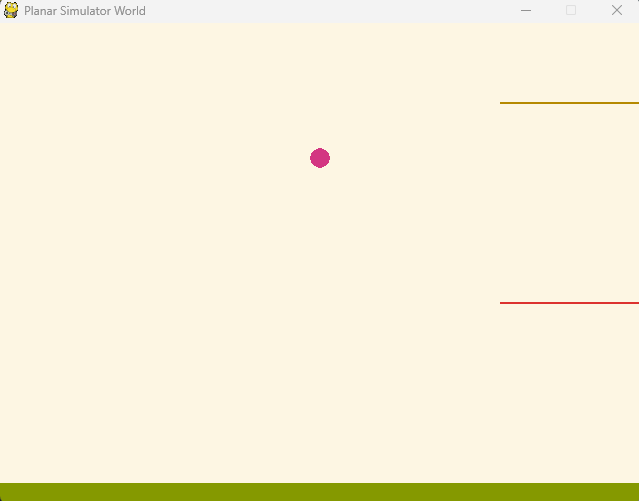


Given code

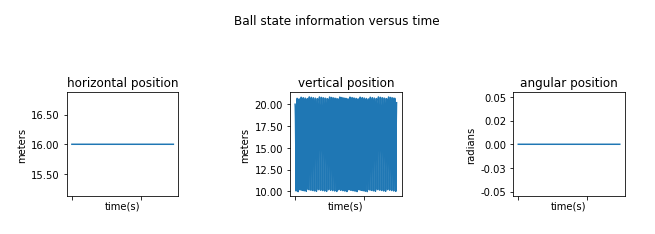
Observation of code that was ran: ball dropped, bounced, and then settled, ended at the position below:   
  
Results after executing the code:   


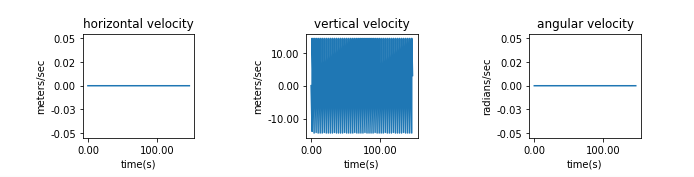


Given code:   
  
Observation of after running the given code: now seeing the environment, we can clearly see that the ball is bouncing between the two heights. We can once again look at the ball dynamics at impact model that was given:   
  
  
Result of Code:



Graph Information after executing code:

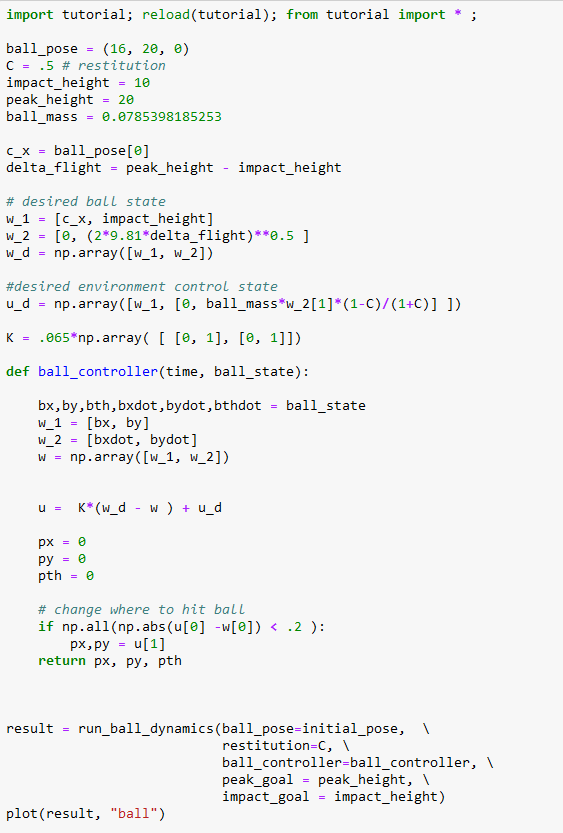


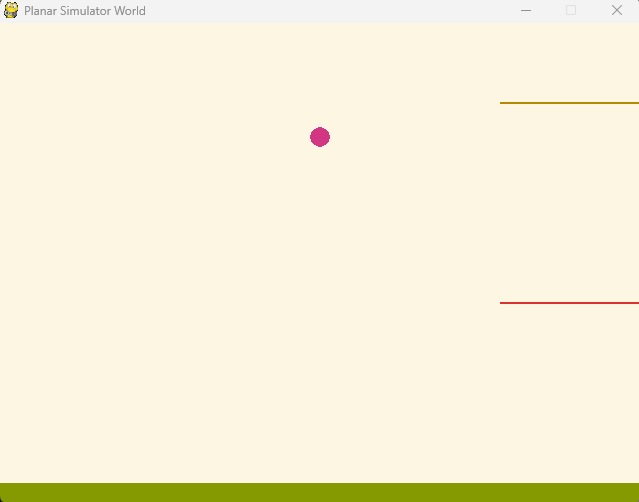


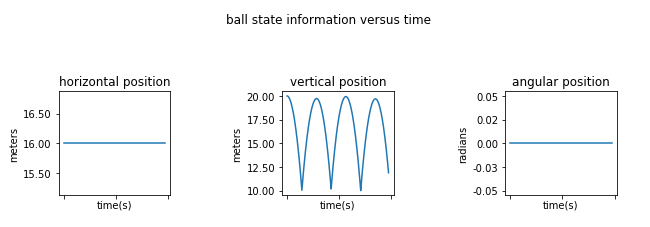
**Environmental Control with a Reference Signal**

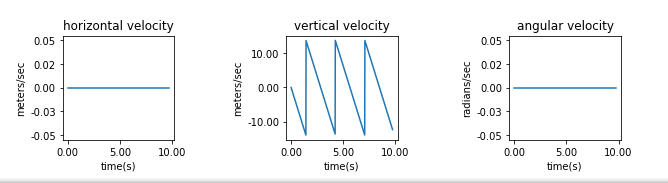
**Why? - We can augment our controller to prevent steady state error. To do this we provide a reference signal for the desired environmental control state.**

Code:

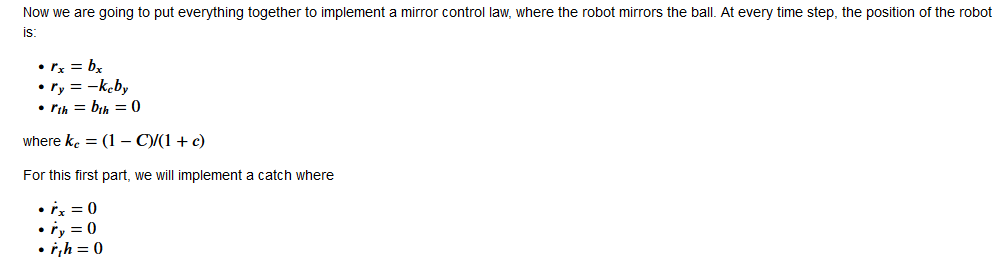






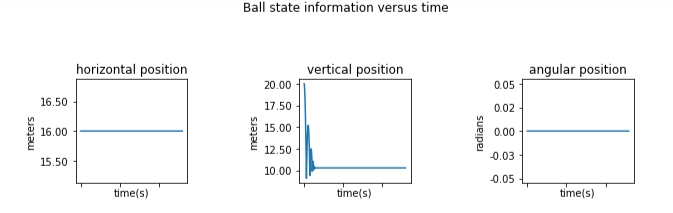


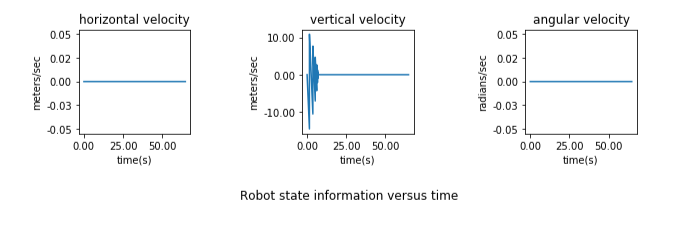
**A Simple Robot Catch**

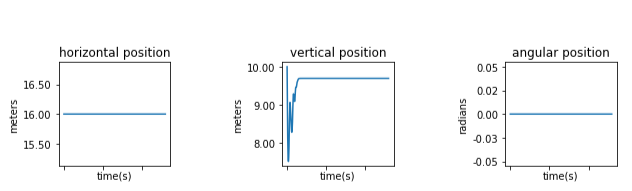
****

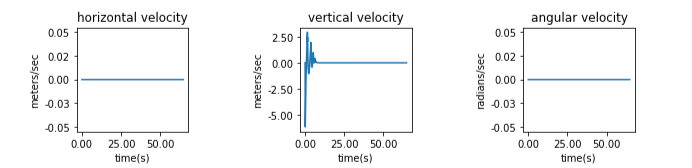
**-> video recording**

**Graph Results:**

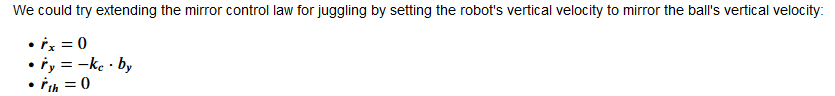
****

****

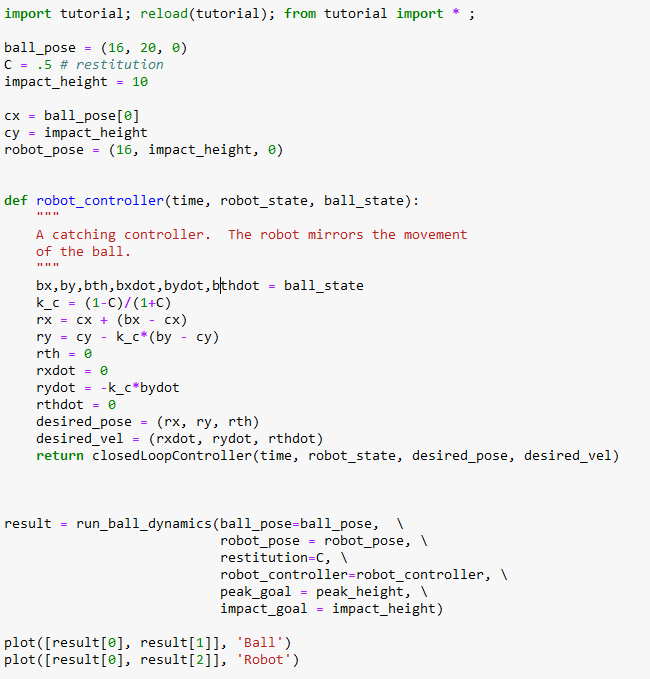
****

****

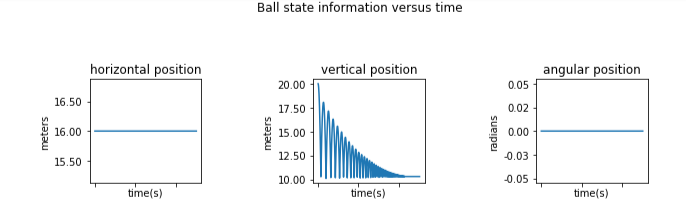
**A Simple robot Juggler:**

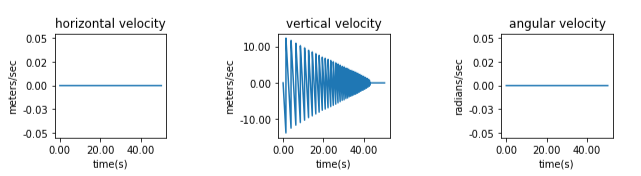
****

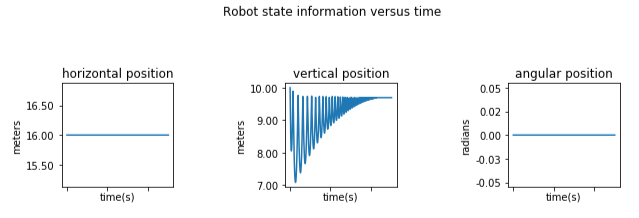
**Code given:**

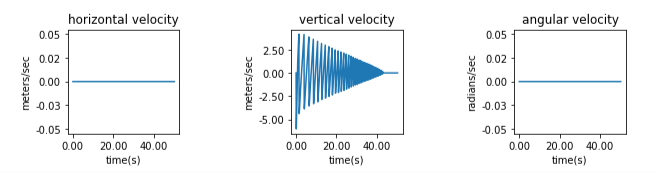
****

**Results:**

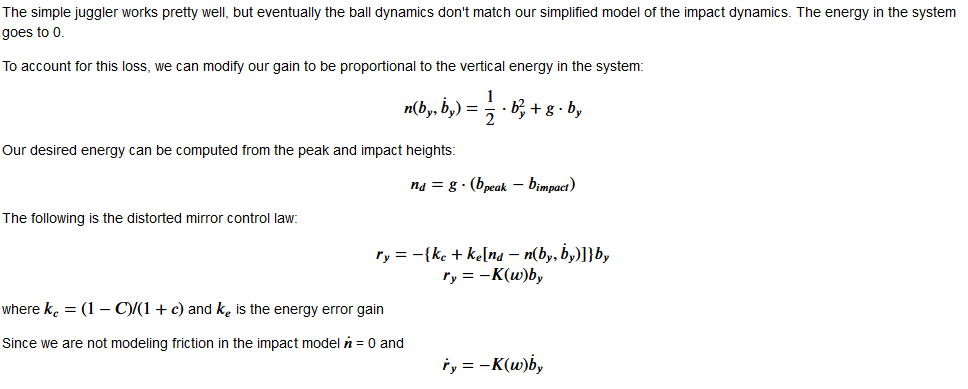
**-> video   
**

****

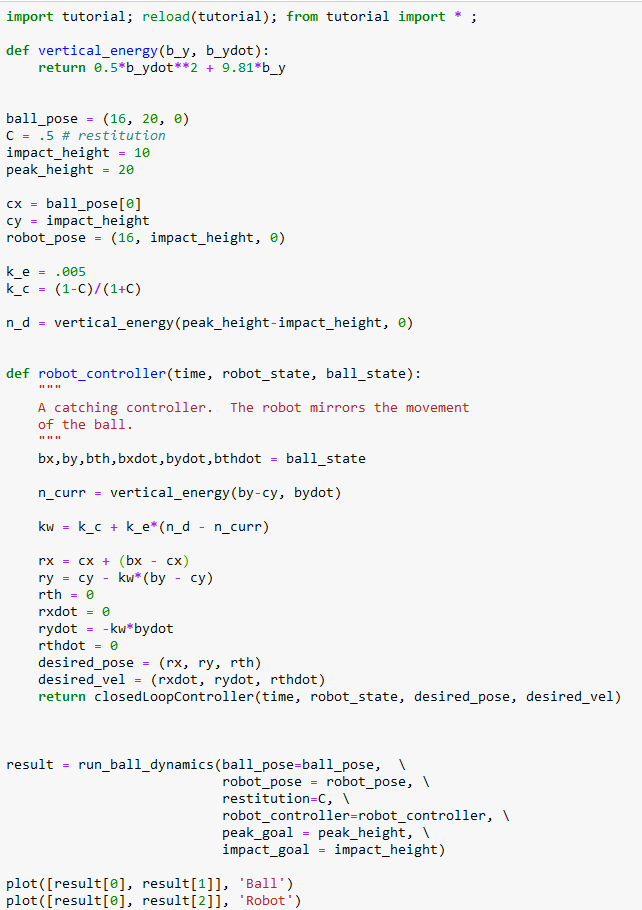
****

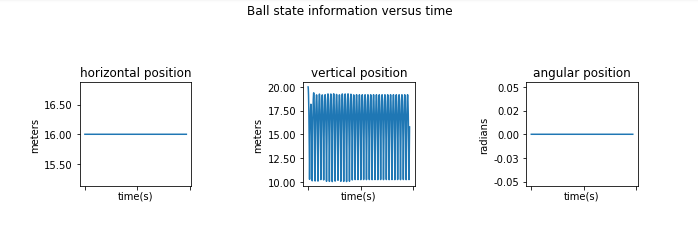
****

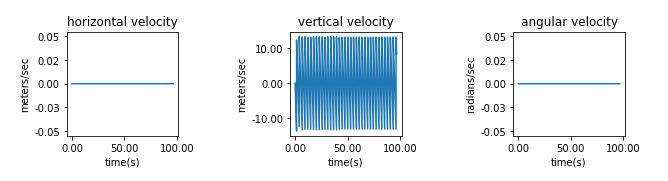
**An Improved Robot Juggler**

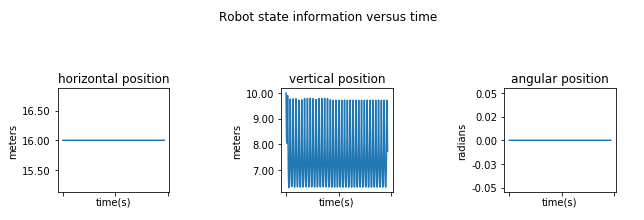
****

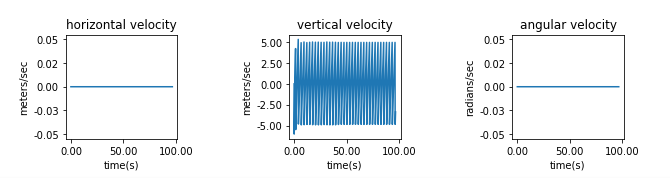
**Video ->>**

**Given Code:   
**

****

****

****

****