MECA 482 - Control Systems Final Project

California State University, Chico

CHICO STATE ENGINEERING

Thomas Allen: Mechanical Engineer (5182)

Aaron Fisher: Mechatronic Engineer (5182)

Kate Gordon: Mechanical Engineer (5182)

Stuart Matthews: Mechatronic Engineer (5182)

December 16, 2019

Department of Mechanical and Mechatronic Engineering and Sustainable Manufacturing

California State University, Chico, CA 95929-0789

Introduction:

Unstable systems and controls are important to simulate because some advanced control systems prove dangerous to test in real world applications. By simplifying the systems that exist in the real world, changes and behavior can be studied with little risk. The simple system known as, “balancing a ball on a beam” is an example of a small risk, simplified system. The cantilever in this project is a beam, usually anchored at one end and the other end is free to move, changing the angle of freedom of the system. The purpose of this project is to code a PID controller to balance something, either a toy car or a ball, on the beam by the use of sensors and code. The PID controller relates angle of the beam to the position of the rolling object and from this equation, the system can stabilize using feedback equations. The feedback signal, to control the beam’s angle, is derived from the ball’s current position to where the system is commanding the ball to move. This simplified system has many real-world applications, such as balancing an airplane wing in strong wind situations, although here, there is little error for adjustment and the situation is much more sensitive and dangerous.

There are several different variations and methods that are often experimented with on systems like the ball balancer project due to the knowledge of how the system should behave. For example, I. Petkoviç, M. Brezak, and R. Cupec built the system with an artificial vision based test [1]. Due to the preexisting results and knowledge of the system behavior, they were able to determine the rolling objects position with a camera. Similarly, P. Dadios and R. Baylon used camera based vision systems with a study on fuzzy logic to balance the ball on the beam [2].

This document reflects the controls systems project of a ball balancer system shown in the below *figures 1* and *2,* with the use of a small toy car instead of a ping pong ball (toy car not shown). The balancing system is comprised of a tall support and shorter support in which the beam is placed in between. The small support is connected to a lever which is attached to a wheel. As the wheel turns, this controls the height and angle of the beam in turn causing the toy car to roll along the beam. In theory, the end result of this project will balance the toy car in the center of the beam after a short duration of back and forth movements of the system due to the PID controller working properly. With adjustment and fine tuning the system should be able to balance the toy car in a timely manner.

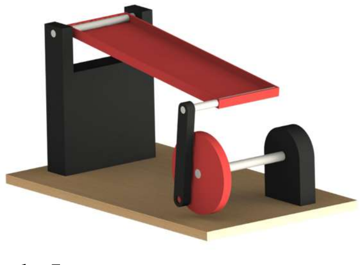


Fig 1: Isometric view

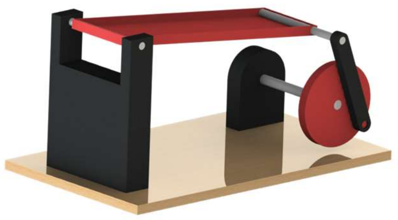


Fig 2: balancer in a rotated view.

Modeling:

The physical system actually consists of an electromechanical system and a mechanical system. The first system contains a DC servo motor that take a signal from the controller and then translates that into mechanical action, or rotation displacement. The second system part is the cantilever that receives the rotation of the servo and converts it into linear displacement of the rolling ball.

Ball and Beam system model:

A beam contains the ball where it can roll the length of the cantilever, as shown in *figure 3*. The free end of the beam is attached to a lever arm which connects to a servo gear. The beams angle changes as the servo gear turns. The ball will resultantly roll down the beam due to gravity and the new angle. The controllers design manipulates the balls position based on beam angle.

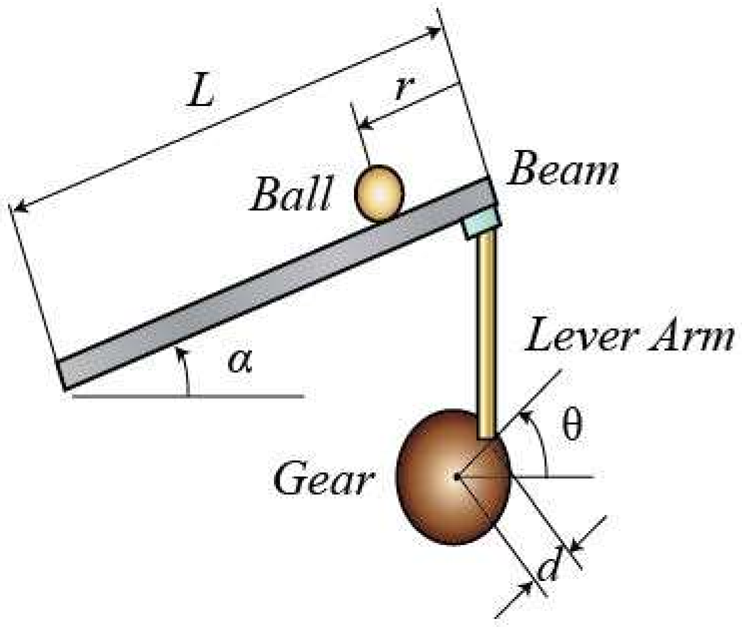


Figure 3: Ball on Beam Model

From this model the transfer function relating the distance of the rolling ball to the angle of the servo rotation is shown in Equation (1).

$$\frac{R(s)}{\theta(s)} = \frac{mgd}{L(\frac{J}{R2}+m)}\frac{1}{s^2}\;[\frac{m}{rad}] \qquad (1)$$

Electromechanical Model:

The DC motor is a common actuator in controls as it can easily be coupled and provides either rotary or translational motion. Either wheels, drums or cables are easily attached to motors for a variety of system modeling. The electric armature and equivalent circuit of the motor can be found in *figure 4*.

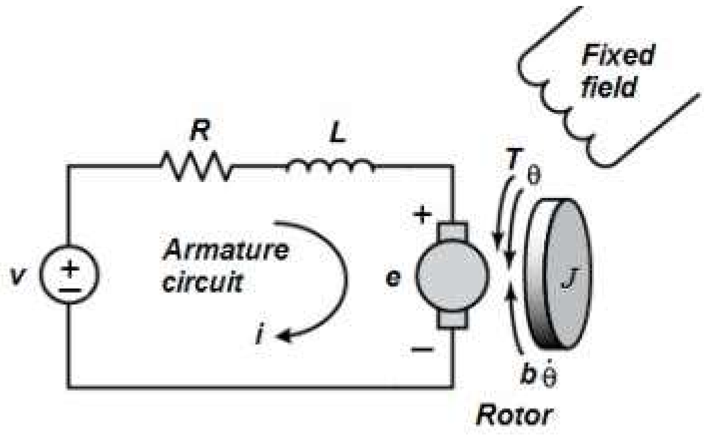


Figure 4: Equivalent DC motor Circuit.

The transfer function for this circuit shows the relationship between the rotation angle and voltage of the circuit, and can be found in equation (2).

$$\frac{\theta(s)}{V(s)} = \frac{k+kg}{s((Js+b)(Las+Ra)+K^2)} \left[\frac{rad}{V}\right] \qquad (2)$$

Sensor Calibration:

Controller Design:

The design of the controller for the overall systems proved to be difficult. To simplify the system as a whole, it was divided into different feedback loops which can be seen in *figure 5*. The first is the inner loop, which controls the gear angle position so the angle tracks the reference signal. The second loop, or the outer loop, controls the car's position based on the feedback of the inner loop.

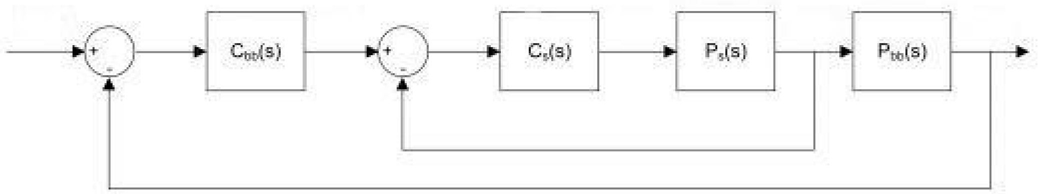


Figure 5: Two loop or cascaded control system design

Inner Loop Design:

The inner loop utilizes a PD controller to correctly determine the position of the servo gear. This control feedback system can be seen in *figure 6*. In order to meet the performance desired, the proportional and velocity constants, or "gains", had to be determined. This can be seen in equation (3).

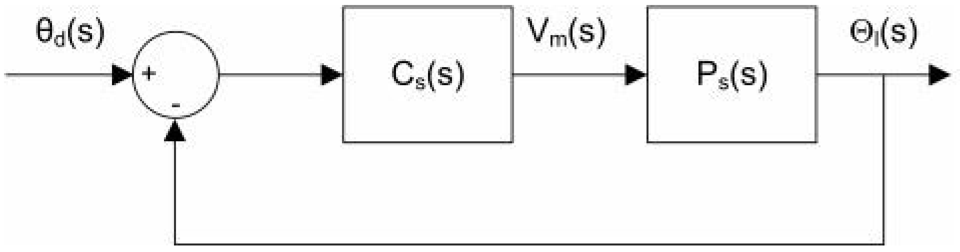


Figure 6: Inner control system

$$C_s(s) = K_p + \frac{K_i}{s} + K_d s = \frac{K_d s^2 + K_p s + K_i}{s} \qquad (3)$$

The proportional controller is used from above and if the system proves to be unstable a derivative or integral term will be added.

Outer Loop Design:

The outer loop contains a compensator term in the control design. This is due to the knowledge of a zero in the forward path increasing the bandwidth of the closed loop feedback system. On the contrary, adding a pole increases overshoot and rise time making the overall design less stable. In the practical sense, the overshoot must be minimized and bandwidth maximized. In *figure 7*, the inner loop is replaced with a single block G(s).

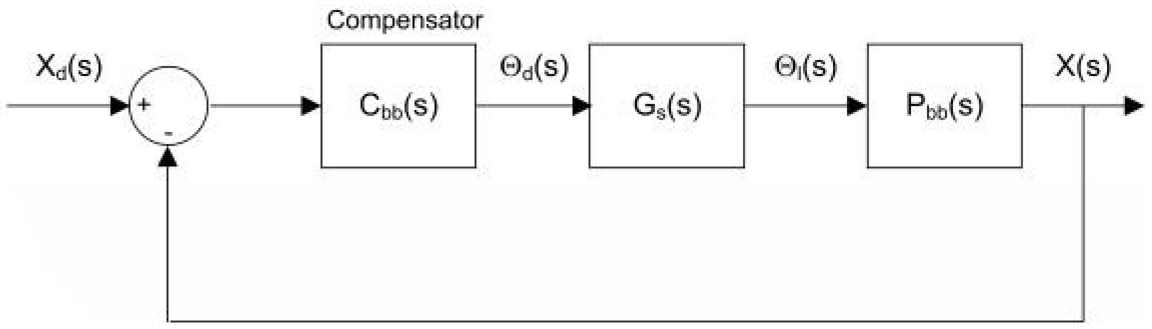


Figure 7: Outer loop control system

Simulations:

To start the design of simulation, the proportional gain term was set to one hundred. The systems response was then recorded by using a step input of .25 m. The systems response curve can be seen below in *figure 8*.

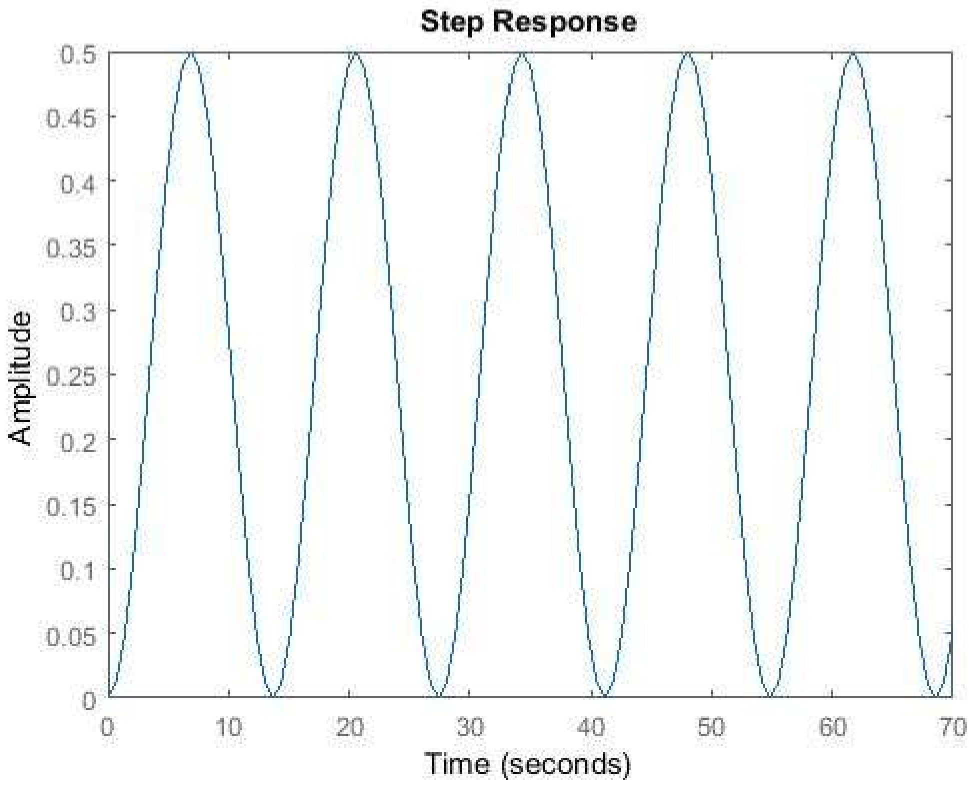


Figure 8: The systems response is unstable

The system response remains unstable no matter the value of the proportional constant. The derivative term was then added and the response was plotted as seen in *figure 9*.

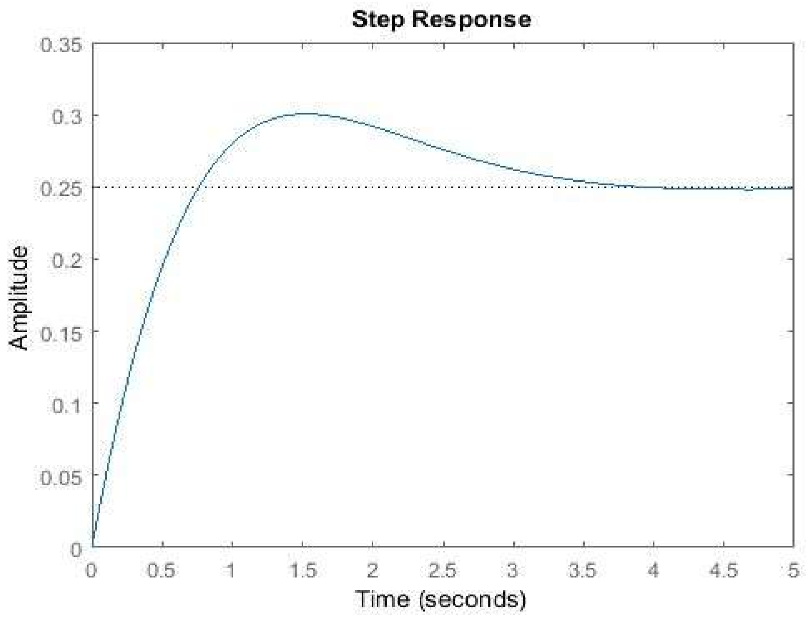


Figure 9: Stable system response with Kp and Kd term

After introducing the derivative term to the system the response shows stability. However, the overshoot was much too large and the settling time must drop. From simple PID controller characteristics, tuning the derivative term to a larger value the overshoot was lowered and the settling time was reduced. Then by increasing the proportional gain slightly the settling time came down to achieve the desired response. The system's step response can be seen in *figure 10*.

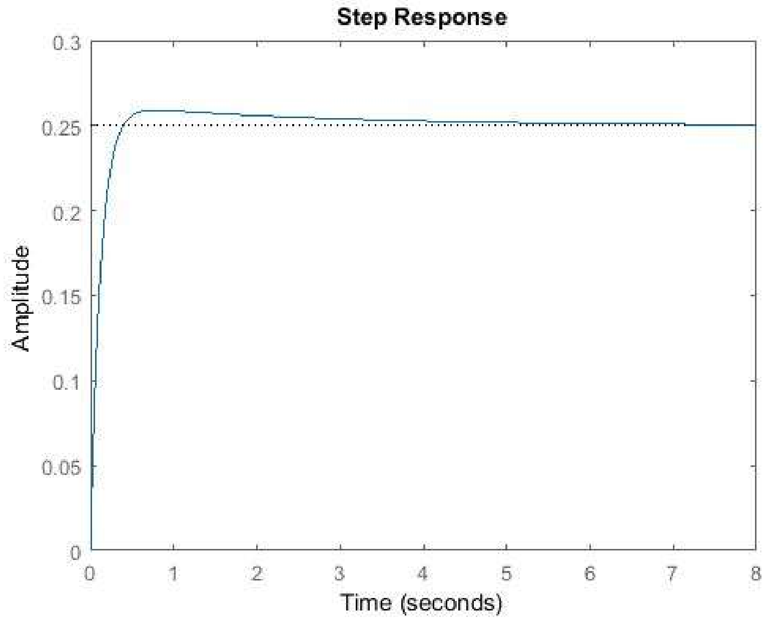


Figure 10: Desired system response with tuned Kp and Kd terms

Simulink:

In order to correctly simulate this project, simulink files were also required. Simulink is a graphical representation of the transfer function and control loops. These simulations were then required to be linked with vrep, a modeling software that creates simulated results of a physical system. The first simulink file, as seen in Figure 11, shows the mathematical model of the transfer function and control loop.

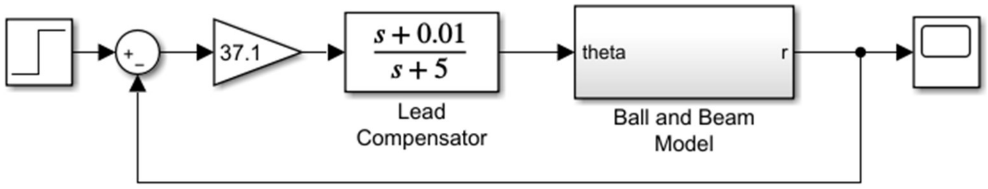


Figure 11: Simulink of transfer function control loop

The second simulink simulation that was created represents the forces and geometric constraints of the real system. However, to model all of the forces can be difficult, so an equation of motion was used directly in the model. This can be seen in figure 12.

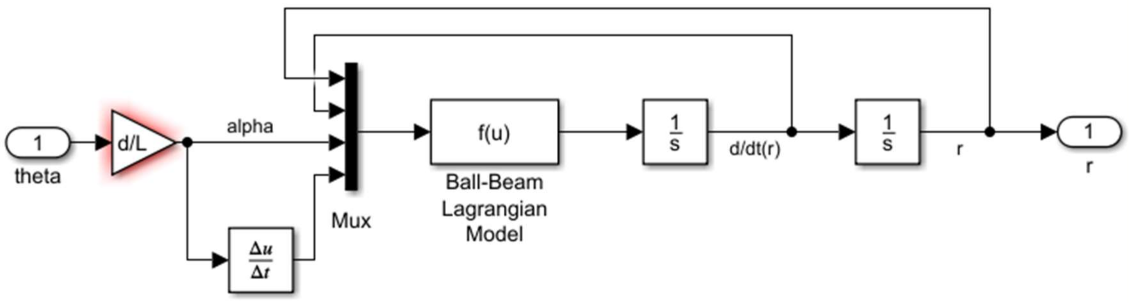


Figure 12: Simulink model of the system

Vrep:

The model built in Vrep is a direct representation of the model that would be built physically. By applying joints, sensors and API's the model can move directly, simulating real world results with the simulink representations built. Figure 13 shows the model created in Vrep, with the red cylindrical shapes representing the joints of the ball and beam balancer. The small red dot at the end of the balancer simulated the proximity sensor that gives the model feedback. The lever arm was built in vrep to the specifications that were set in the matlab simulation, keeping all the dimensions the same.

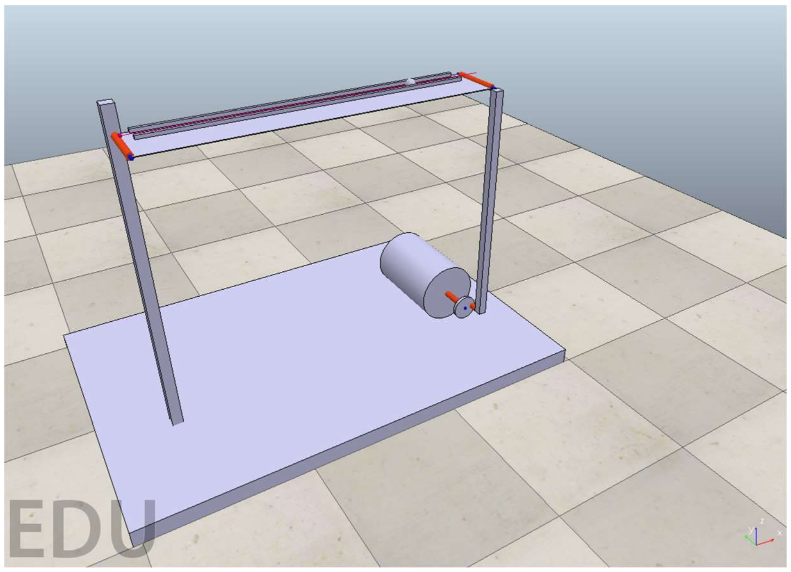
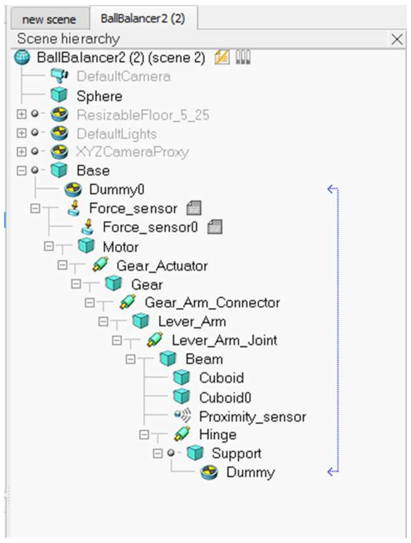


Figure 13: Vrep model simulating the physical system with simple shapes

To correctly simulate the results in vrep, the sensors, joints and shapes had to be placed in the hierarchy tree in a specific order. This hierarchy correlates the motion to the desired parts and simulates the behavior as it would react in the real world. This breakdown can be seen in Figure 14. The tree is expanded in the figure to show the parent shapes were built first. After the simple shape model was created the joints were implemented as children along with the sensors. The last added components were the two dummies added at the top and bottom of the hierarchy tree. These two dummies were linked to make the model all linked as a structure similar to a feedback system. The entire working model can be found as an attached file.



```
new scene | BallBalancer2 (2)
Scene hierarchy
BallBalancer2 (2) (scene 2)
  DefaultCamera
  Sphere
  ResizableFloor_5_25
  DefaultLights
  XYZCameraProxy
  Base
    Dummy0
    Force_sensor
      Force_sensor0
      Motor
        Gear_Actuator
          Gear
            Gear_Arm_Connector
              Lever_Arm
                Lever_Arm_Joint
                  Beam
                    Cuboid
                    Cuboid0
                    Proximity_sensor
                    Hinge
                      Support
                        Dummy
```

Figure 14: Vrep Hierarchy tree

Conclusion:

The ball and beam model proved to be a basic representation of a control system. After studying the behaviors and feedback of a simulation, the physical model implementation becomes far easier.

Appended Notes:

While building this model the team wanted to document the process for linking programs together to simulate in real time. There were three separate programs used to simulate the overall system: Matlab, Simulink, and Vrep. Matlab was used to generate the transfer function and step response of the plant. Simulink was used to model the feedback and control loop of the system. Lastly Vrep was used to model the entire system.

In order to simulate, the model in vrep with sensors must be interfaced with matlab to calculate the real time sensors and output. Along with matlab, simulink also must be interfaced with vrep so that the control system can be applied properly. The steps and code used to make this work are as follows.

<u>Interfacing Matlab with Vrep:</u>

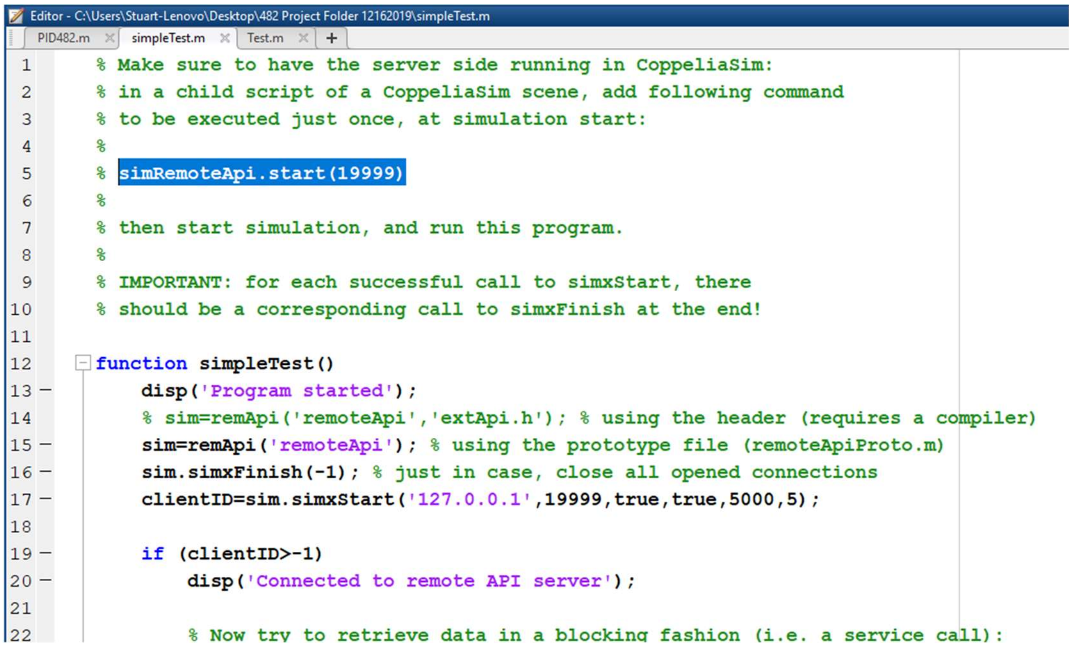
In order to interface Vrep and Matlab, there must first be a set of files pulled from the Vrep folder location and put into the project folder. For this demonstration, Windows 10 64-bit was used. Begin by opening the file location of Vrep (CoppeliaSim) and proceed through the following folders:

CoppeliaSimEdu > programming > remoteApiBindings > matlab > matlab

From here copy the files: *remApi.m, remoteApiProto.m* and *simpleTest.m* to the project folder. Backtrack to the remoteApiBindings folder and proceed through the following folders:

remoteApiBindings > lib > lib > Windows

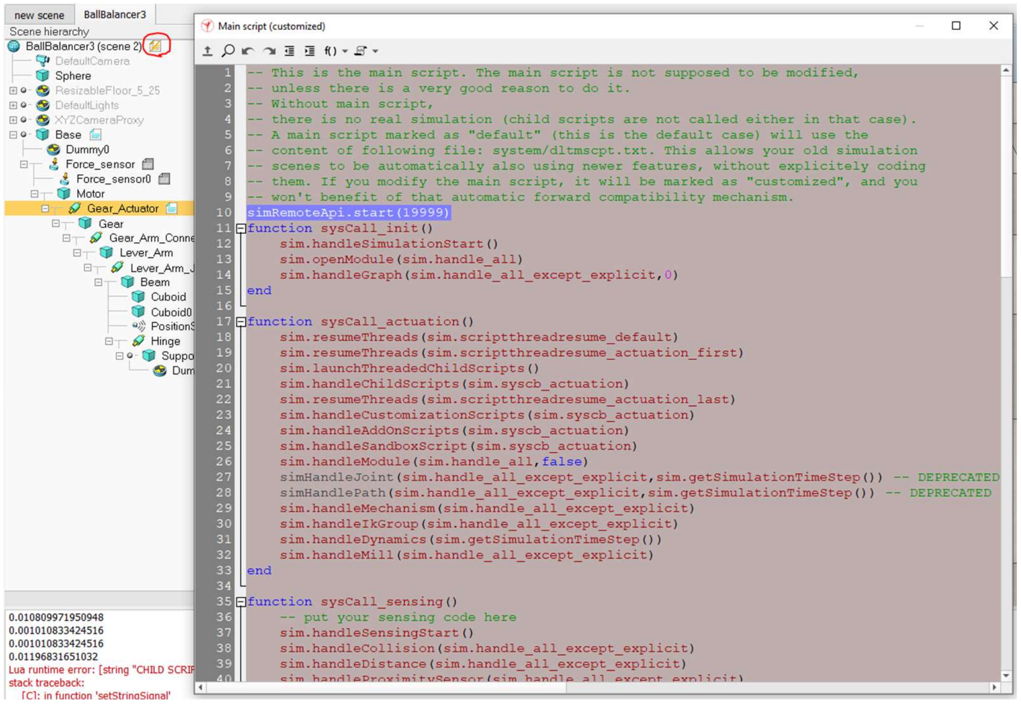
Add *remoteApi.dll* to the project folder. From here, open CoppeliaSim and Matlab. Open the scene and the *simpleTest.m* file. Copy the line *simRemoteApi.start(19999)* and open the main scene script in CoppeliaSim (Figures 15 & 16).



Editor - C:\Users\Stuart-Lenovo\Desktop\482 Project Folder 12162019\simpleTest.m
PID482.m
simpleTest.m
Test.m

```
% Make sure to have the server side running in CoppeliaSim:
% in a child script of a CoppeliaSim scene, add following command
% to be executed just once, at simulation start:
%
% simRemoteApi.start(19999)
%
% then start simulation, and run this program.
%
% IMPORTANT: for each successful call to simxStart, there
% should be a corresponding call to simxFinish at the end!

function simpleTest()
    disp('Program started');
    % sim=remApi('remoteApi','extApi.h'); % using the header (requires a compiler)
    sim=remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
    sim.simxFinish(-1); % just in case, close all opened connections
    clientID=sim.simxStart('127.0.0.1',19999,true,true,5000,5);

    if (clientID>-1)
        disp('Connected to remote API server');

        % Now try to retrieve data in a blocking fashion (i.e. a service call):
```

Figure 15: Location of *simRemoteApi.start()* in simpleTest.m



```
0.010809971950948
0.001010833424516
0.001010833424516
0.01196831651032
Lua runtime error: [string "CHILD SCRIF
stack traceback:
   [C]: in function 'setStringSignal'
```

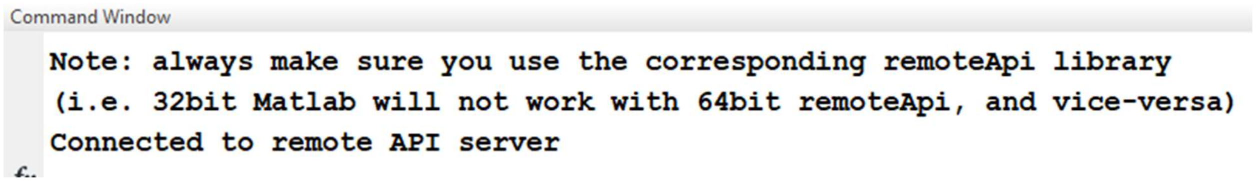
```
-- This is the main script. The main script is not supposed to be modified,
-- unless there is a very good reason to do it.
-- Without main script,
-- there is no real simulation (child scripts are not called either in that case).
-- A main script marked as "default" (this is the default case) will use the
-- content of following file: system/dltmscpt.txt. This allows your old simulation
-- scenes to be automatically also using newer features, without explicitely coding
-- them. If you modify the main script, it will be marked as "customized", and you
-- won't benefit of that automatic forward compatibility mechanism.
simRemoteApi.start(19999)
function sysCall_init()
    sim.handleSimulationStart()
    sim.openModule(sim.handle_all)
    sim.handleGraph(sim.handle_all_except_explicit,0)
end

function sysCall_actuation()
    sim.resumeThreads(sim.scriptthreadresume_default)
    sim.resumeThreads(sim.scriptthreadresume_actuation_first)
    sim.launchThreadedChildScripts()
    sim.handleChildScripts(sim.syscb_actuation)
    sim.resumeThreads(sim.scriptthreadresume_actuation_last)
    sim.handleCustomizationScripts(sim.syscb_actuation)
    sim.handleAddOnScripts(sim.syscb_actuation)
    sim.handleSandboxScript(sim.syscb_actuation)
    sim.handleModule(sim.handle_all,false)
    simHandleJoint(sim.handle_all_except_explicit,sim.getSimulationTimeStep()) -- DEPRECATED
    simHandlePath(sim.handle_all_except_explicit,sim.getSimulationTimeStep()) -- DEPRECATED
    sim.handleMechanism(sim.handle_all_except_explicit)
    sim.handleIkGroup(sim.handle_all_except_explicit)
    sim.handleDynamics(sim.getSimulationTimeStep())
    sim.handleMill(sim.handle_all_except_explicit)
end

function sysCall_sensing()
    -- put your sensing code here
    sim.handleSensingStart()
    sim.handleCollision(sim.handle_all_except_explicit)
    sim.handleDistance(sim.handle_all_except_explicit)
    sim.handleProximitySensor(sim.handle_all_except_explicit)
```

Figure 16: Where to Paste *simRemoteApi.start()* in CoppeliaSim

Now create a new .m file and add the following lines to it from *simpleTest.m.* When the new file is run it will display whether or not it was able to connect to the API. *sim.simxGetStringSignal()* will be used for the next part of the code and is not mandatory at this time

```
>sim=remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
>sim.simxFinish(-1); % just in case, close all opened connections
>clientID=sim.simxStart('127.0.0.1',19999,true,true,5000,5);
>
>if (clientID>-1)
>       disp('Connected to remote API server');
>       sim.simxGetStringSignal(clientID,'distance',sim.simx_opmode_streaming);
>       while(1)
>
>       end
> else
>           disp('Failed connecting to remote API server');
>end
>sim.delete(); % call the destructor!
>
>disp('Program ended');
```

Run the CoppeliaSim scene, and then while that is under way, run the matlab script. If done correctly, the tex *Connected to remote API server* should display (Figure 17).



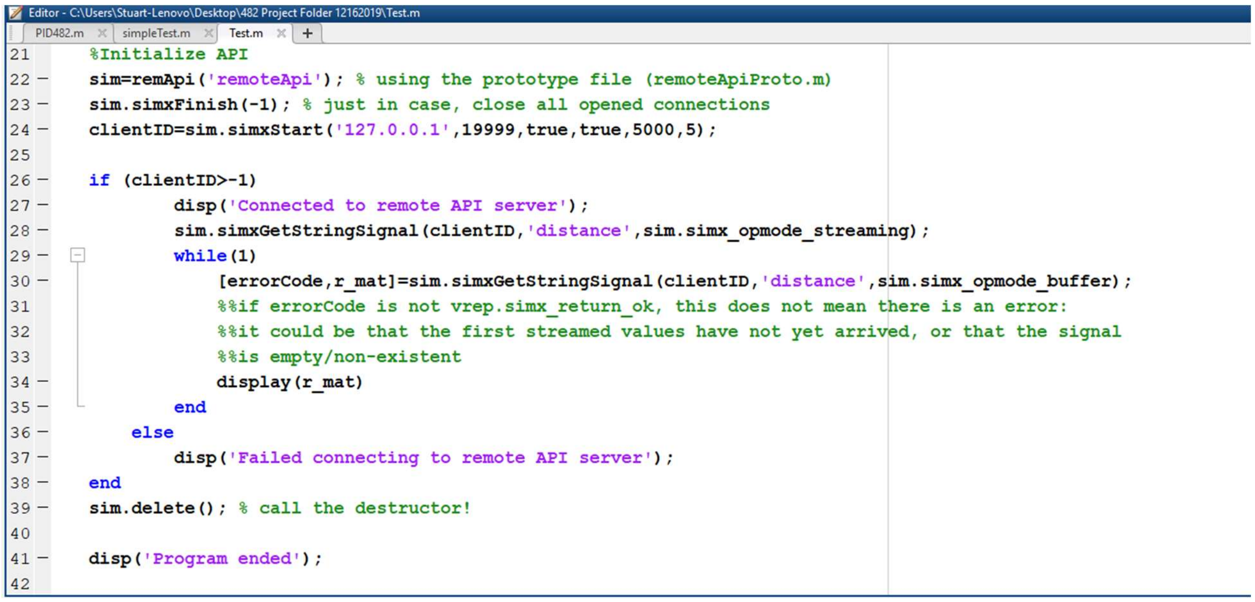
```
Note: always make sure you use the corresponding remoteApi library
(i.e. 32bit Matlab will not work with 64bit remoteApi, and vice-versa)
Connected to remote API server
```

Figure 17: Successful Connection to the remote API.

Now add the following lines within the while loop of the Matlab code.

```
>[errorCode,r_mat]=sim.simxGetStringSignal(clientID,'distance',sim.simx_opmode_buffer);
>display(r_mat)
```

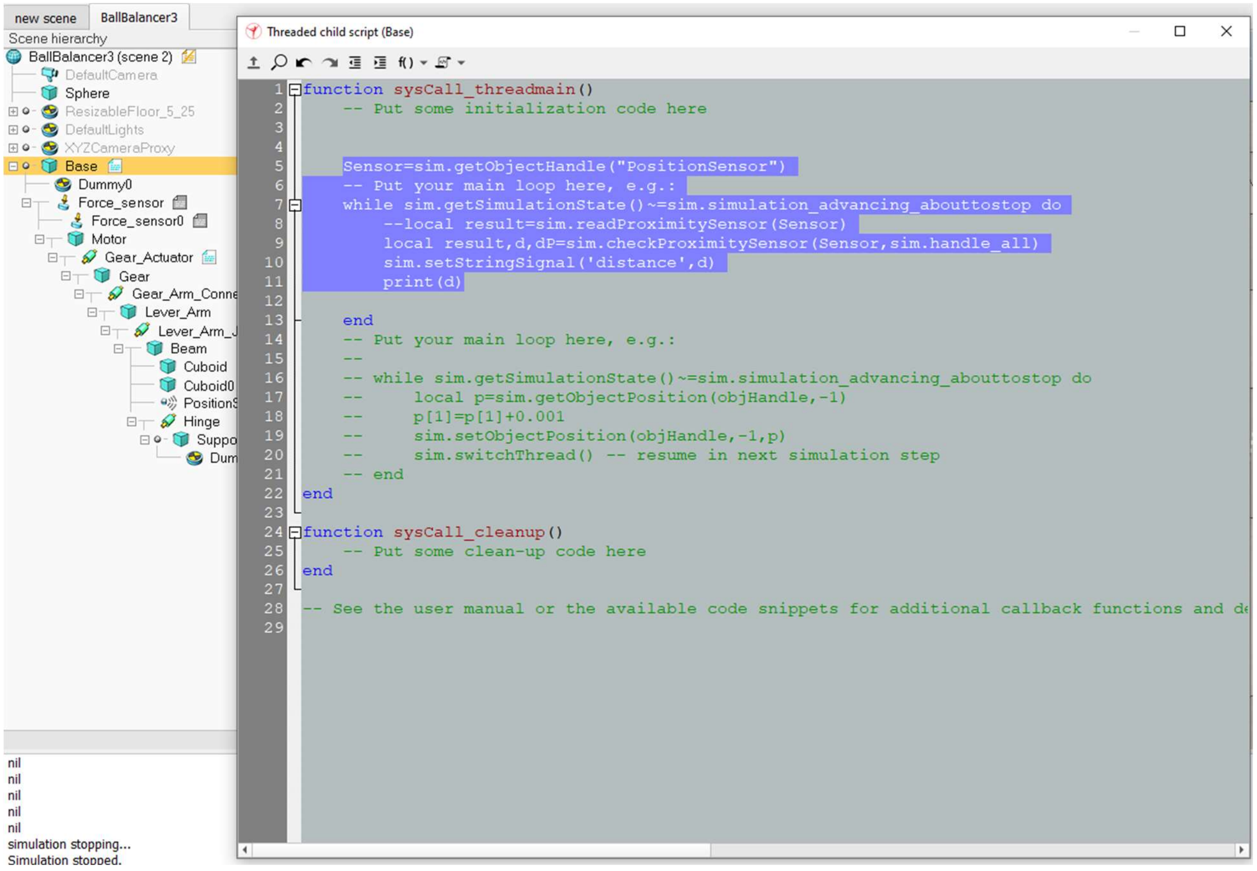
These lines, along with the *sim.simxGetStringSignal()* previously mentioned will stream, in the case of this project, the distance of the ball from the sensor. The Matlab file should now appear similar to the following (Figure 18).



```
Editor - C:\Users\Stuart-Lenovo\Desktop\482 Project Folder 12162019\Test.m
PID482.m    simpleTest.m    Test.m
%Initialize API
sim=remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
sim.simxFinish(-1); % just in case, close all opened connections
clientID=sim.simxStart('127.0.0.1',19999,true,true,5000,5);

if (clientID>-1)
        disp('Connected to remote API server');
        sim.simxGetStringSignal(clientID,'distance',sim.simx_opmode_streaming);
        while(1)
            [errorCode,r_mat]=sim.simxGetStringSignal(clientID,'distance',sim.simx_opmode_buffer);
            %%if errorCode is not vrep.simx_return_ok, this does not mean there is an error:
            %%it could be that the first streamed values have not yet arrived, or that the signal
            %%is empty/non-existent
            display(r_mat)
        end
    else
        disp('Failed connecting to remote API server');
end
sim.delete(); % call the destructor!

disp('Program ended');
```

Figure 18: Code for the Matlab Side of Integration

Moving back to CoppeliaSim, go to the top level structure of the hierarchy and add a threaded child code (Figure 19). Add the lines into the code and repeat what was done to check that the remote API was connected. There should now be the distance values displayed in Vrep showing up in Matlab.



```
function sysCall_threadmain()
    -- Put some initialization code here


    Sensor=sim.getObjectHandle("PositionSensor")
    -- Put your main loop here, e.g.:
    while sim.getSimulationState()~=sim.simulation_advancing_abouttostop do
        --local result=sim.readProximitySensor(Sensor)
        local result,d,dP=sim.checkProximitySensor(Sensor,sim.handle_all)
        sim.setStringSignal('distance',d)
        print(d)

    end
    -- Put your main loop here, e.g.:
    --
    -- while sim.getSimulationState()~=sim.simulation_advancing_abouttostop do
    --     local p=sim.getObjectPosition(objHandle,-1)
    --     p[1]=p[1]+0.001
    --     sim.setObjectPosition(objHandle,-1,p)
    --     sim.switchThread() -- resume in next simulation step
    -- end
end

function sysCall_cleanup()
    -- Put some clean-up code here
end

-- See the user manual or the available code snippets for additional callback functions and de
```

```
nil
nil
nil
nil
nil
simulation stopping...
Simulation stopped.
```

Figure 19: Code and Location to allow for data streaming to Matlab from Vrep.

Sensor=sim.getObjectHandle(“”) assigns the object a variable in the code. In this case it is the proximity sensor. *Local result,d,dP=sim.checkProximitySensor(Sensor,sim.handle_all)* checks the trigger status, distance and distancePoint of the proximity sensor and the object it is detecting. *sim.setStringSignal(‘’,)* links the distance variable *d* to the string *distance*. This is used by the lines s*im.simxGetStringSignal(clientID,'distance',sim.simx_opmode_streaming);* and *[errorCode,r_mat]=sim.simxGetStringSignal(clientID,'distance',sim.simx_opmode_buffer);* in Matlab to get the value stored in the string and convert it to a variable to be displayed in the command window of Matlab.

Resources

[1] Ali, A. T., Ahmed, A. M., Almahdi, H. A., Taha, O. A., & Naseraldeen, A. (2017). Design and implementation of ball and beam system using pid controller. *MAYFEB Journal of Electrical and Computer Engineering*, *1*.

[2] Dadios, E. P., Baylon, R., De Guzman, R., Florentino, A., Lee, R. M., & Zulueta, Z. (2000, October). Vision guided ball-beam balancing system using fuzzy logic. In *2000 26th Annual Conference of the IEEE Industrial Electronics Society. IECON 2000. 2000 IEEE International Conference on Industrial Electronics, Control and Instrumentation. 21st Century Technologies* (Vol. 3, pp. 1973-1978). IEEE.