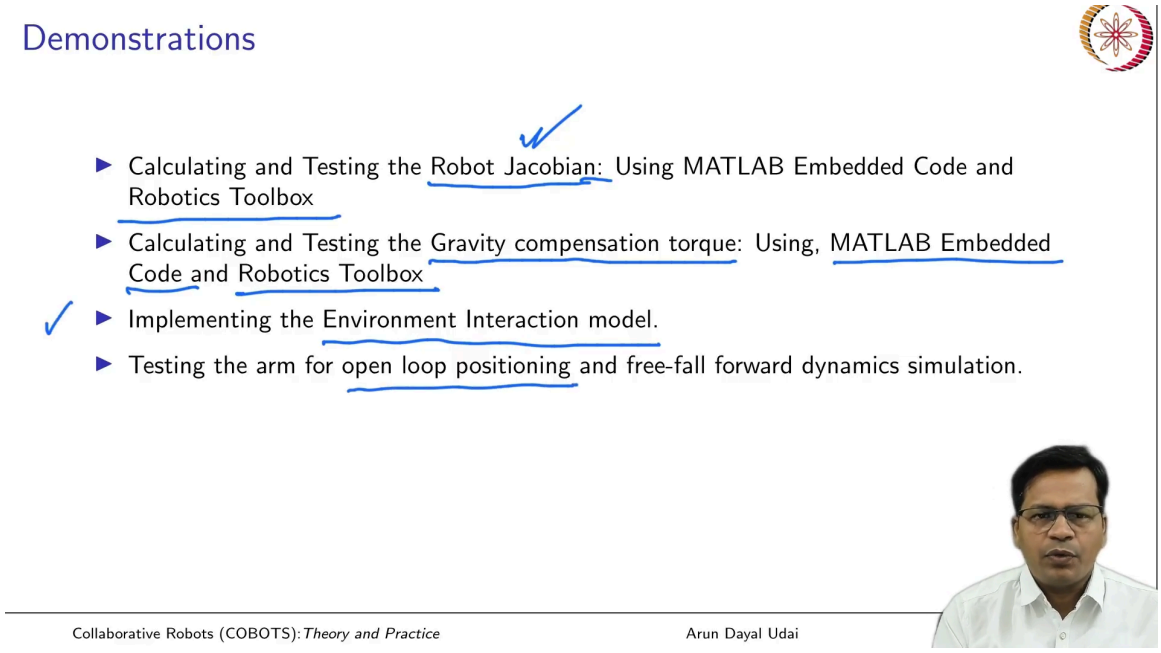
NPTEL Online Certification Courses
COLLABORATIVE ROBOTS (COBOTS): THEORY AND PRACTICE
Dr Arun Dayal Udai
Department of Mechanical Engineering
Indian Institute of Technology (ISM) Dhanbad
Week: 08
Lecture: 34

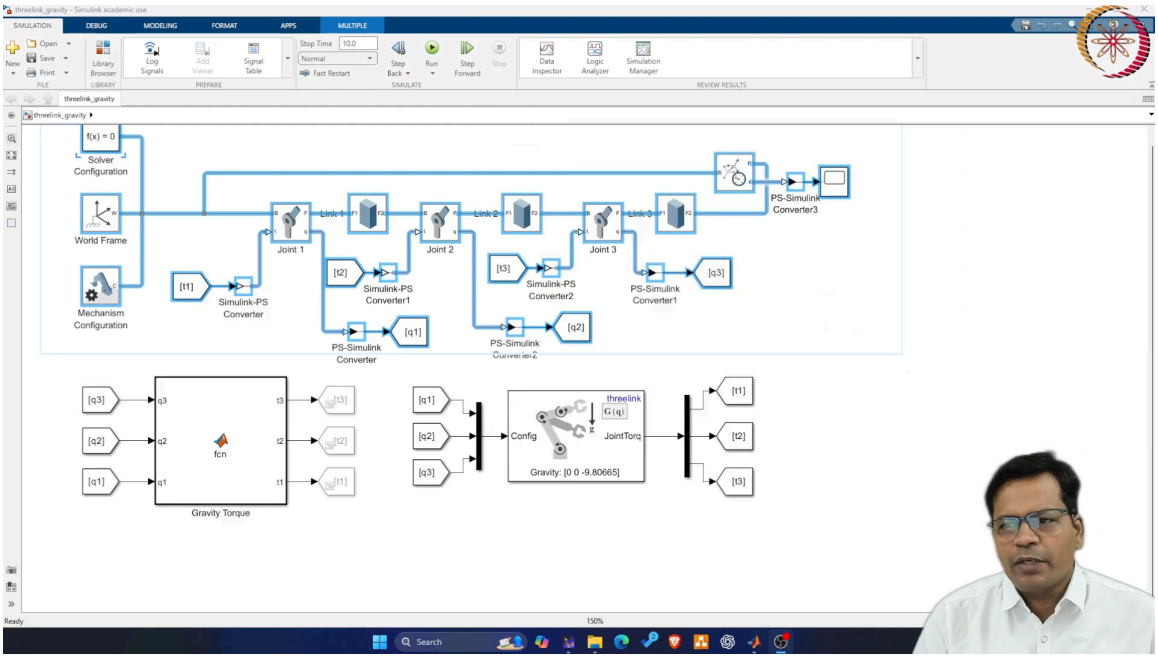
MATLAB Demonstrations- Preparing the 3R Spatial Arm for Force Control



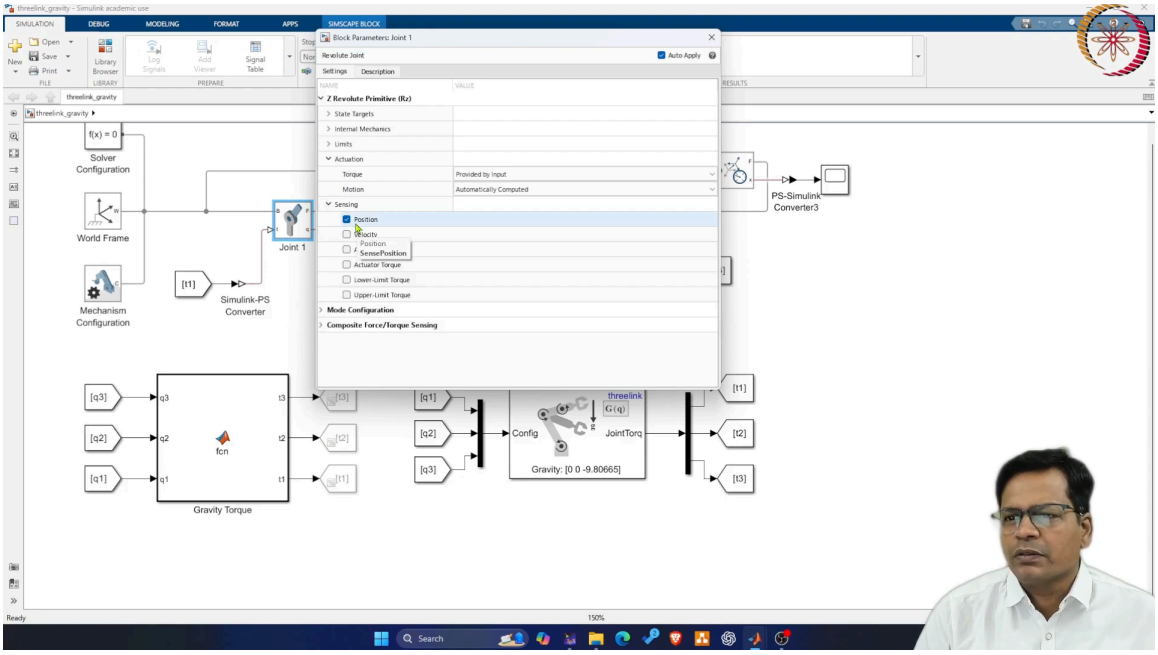
Welcome back to the MATLAB Demonstrations. We will continue preparing the 3R Spatial Arm that we worked on in the last lecture for force control, with a few more elements that we will add. So, here we go. So, these are the demonstrations. We will calculate the robot Jacobian and implement it into the MATLAB Embedded Code in our Simulink model. We will do it otherwise also using the Robotics Toolbox. We will calculate the Jacobian and test whether it is correct or incorrect to ensure we use the correct Jacobian model in the actual simulation when applying it to the force control algorithm. So, this Jacobian needs to be verified. This is very important if you have a customised Jacobian and are not using the Jacobian toolbox.

Similarly, we will develop embedded MATLAB code in the MATLAB Simulink environment for gravity compensation torque. We will do it through MATLAB embedded code. Again, we will test it. If we are not using the Robotics Toolbox, we need to do it using MATLAB-embedded code. We will do it and test it before actually using it in our force control algorithm.

We will implement the environment interaction model that we discussed earlier in the previous lecture. The same model we will be using it here again, and we will test it, and we will finally test the arm for open-loop positioning and for free-fall forward dynamic simulation. So, let us continue with the MATLAB environment. So, here we go.



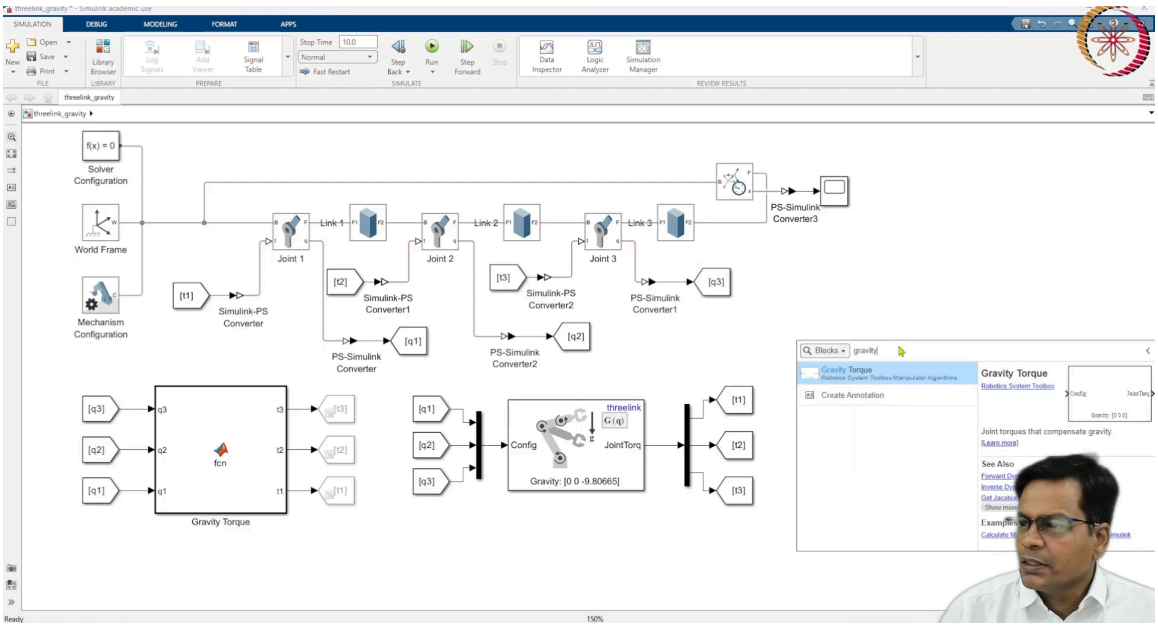
So, we will again pick up the same model that we used earlier. So, you just see that. So, this is the model. So, from here to here, all of these block sets were developed earlier. If you remember, we have a 3R special arm, as you have seen here also. So, it is the same one. We developed these blocks in the previous lecture.



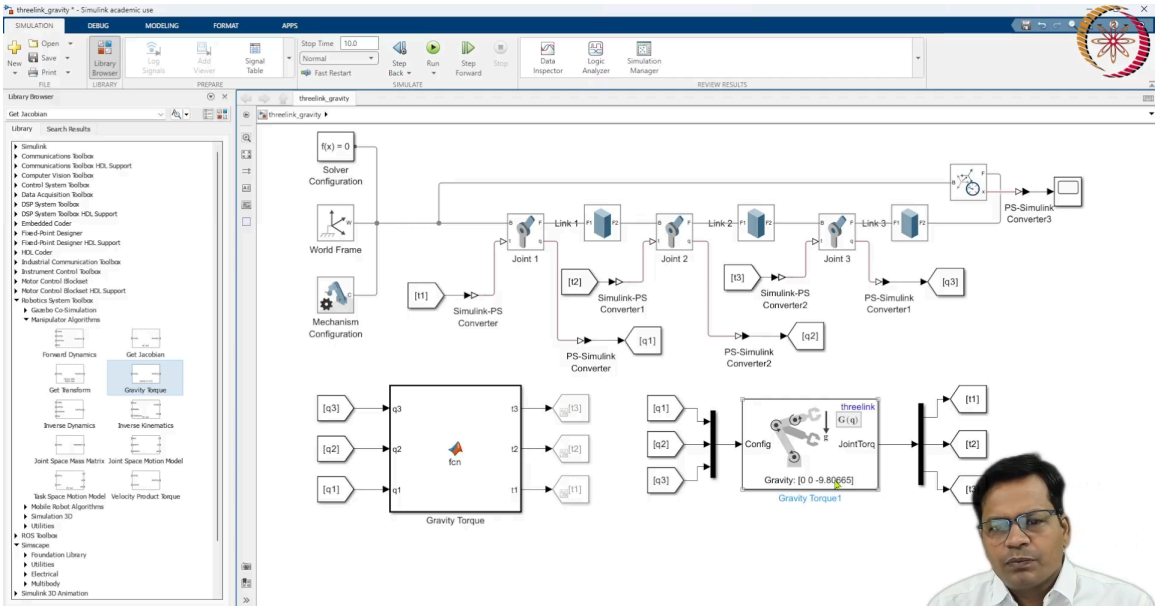
Now, additionally, we have put a few more elements into the joints. First is the actuation. So, torque is now provided by the input, and motion will be computed automatically. So, what essentially are we doing? Essentially, we are doing forward dynamics. So, torque is provided by the input, the motion will be computed automatically, and we will be sensing the position. Joint position, as the robot moves or remains stationary, will be continuously tracked. So, these two will additionally implement this and this over here, which can be connected and stored. So, whatever the joint angle is that we are sensing is converted using a PS to Simulink converter and will be saved here to the variable named Q1. That is the joint angle 1, and similarly, the input block over here is t1. t1 is the runtime variable, which is the outcome of the gravity compensation torque. That torque will put it to the joint.

Again, we will be using a Simulink to PS converter here. That is the physical system. These are the physical systems, so we need to convert the data types here from Simulink to the physical system. So, for each of the joints, I did that. The independent torques are t1, t2, and t3. Those are the outcomes of the gravity compensation torque and the runtime variables. The joint angles are stored in q1, q2, and q3. They become the input to the gravity torque calculation block.

So now, the whole robot system is ready with actuation and sensing inputs and outputs, and the rest of the robot remains the same. Now, gravity compensation torque can be calculated using different ways. The first one I'll discuss here is using the Robotics Toolbox.

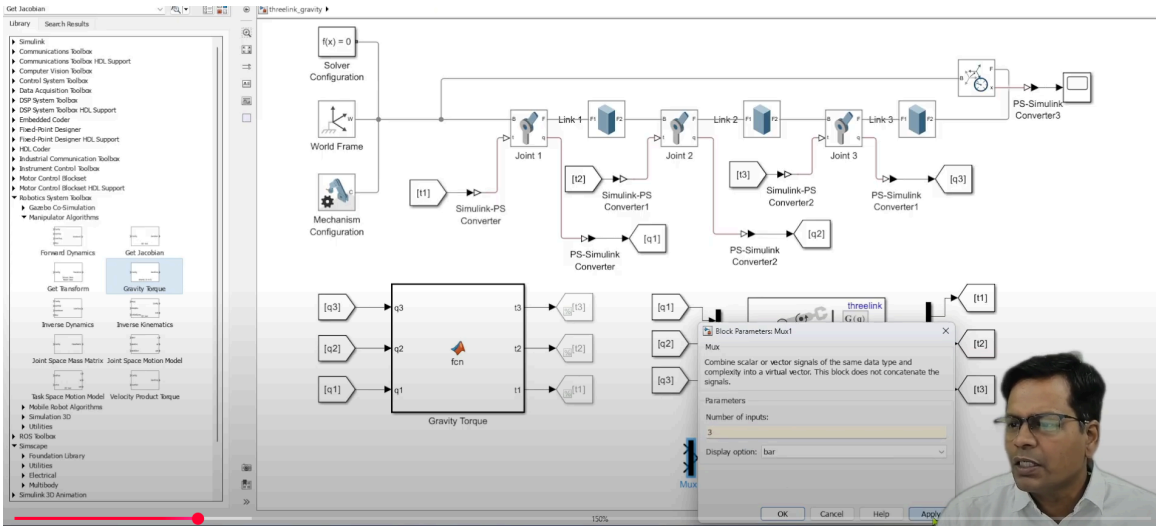


So, here is your Robotics Toolbox. You can directly double-click here and input gravity torque, which is included in the Robotic System Toolbox under Manipulator Algorithms. If you go, let me just show you.

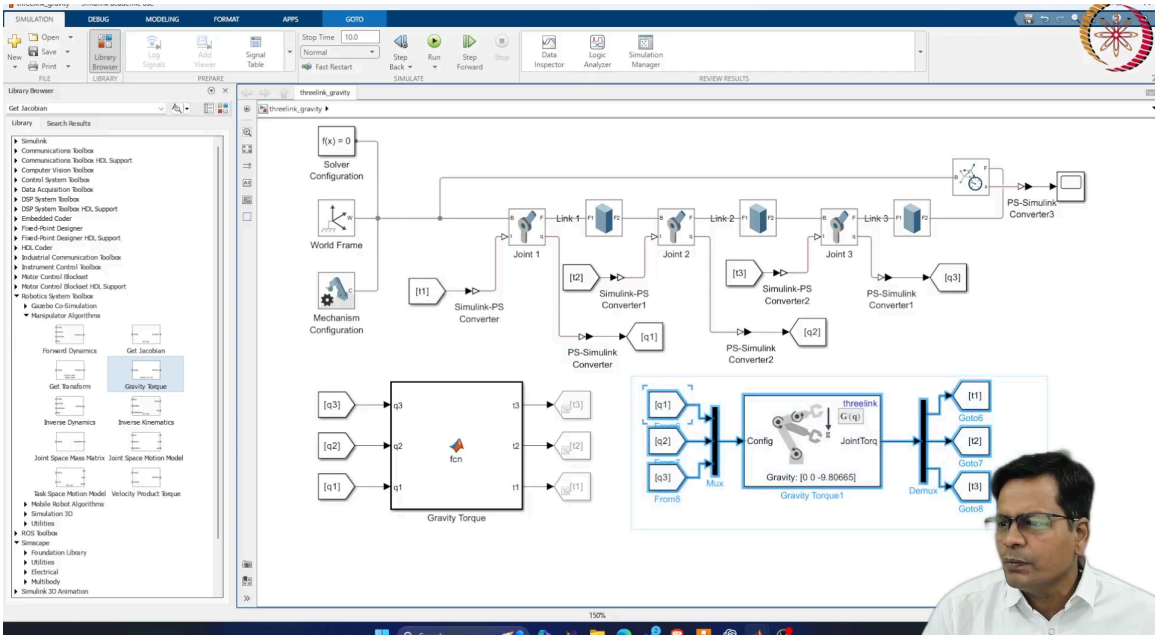


So, here is your Simscape. This is your Robotic System Toolbox. Under Manipulator Algorithms, you have get Jacobian, forward Dynamics, and gravity torque. So here it is. You can directly drag and drop it here to get the block here. So, this is what I have done earlier.

So, I have just shown you. So, this also includes gravity of [0, 0, -9.8065]. So, as per our convention, gravity is directed downward along the negative z-axis of the base frame. So, that is the gravity that, by default, it takes. The configuration of the robot is continuously taken up by the joint angles, that is, q1, q2, and q3. As you can see, it is getting highlighted in the model itself. q1, q2, and q3 go as input through a MUX channel. MUX basically takes scalar variables and puts them in vector form. So, you can directly go here and do this.

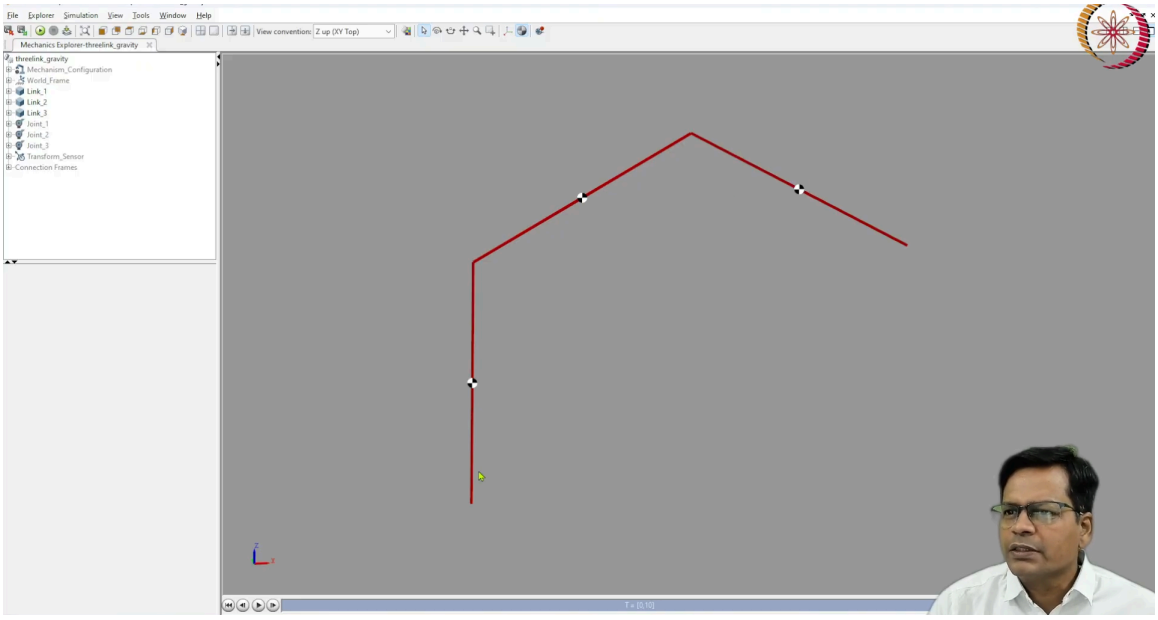


This is the standard Simulink block set. That is the smaller Simulink block set. And by default, it has two channels. You can increase it to three channels here. I am double-clicking, entering the variable here. The number of inputs is three, and it becomes like this. Make it a little bigger so that it becomes easier to connect the lines. So, yes, this is your mark.

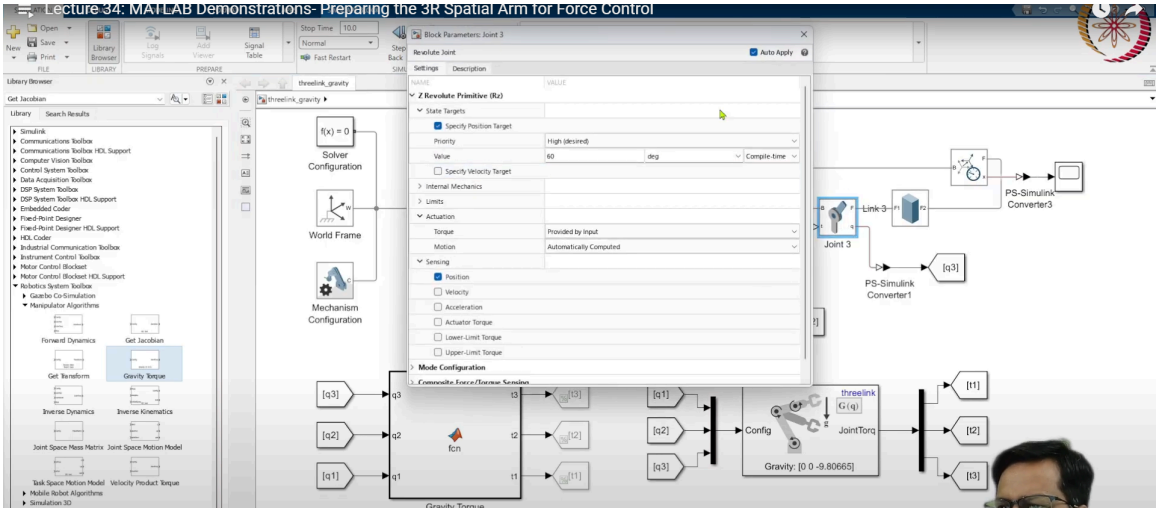


Similarly, this gravity compensation block will give you a torque output. That is the joint torque output for a particular pose. The pose is defined by these three inputs. So, with that joint torque again, I will split them because it comes out as a vector. I will split them using a Demux. That again, is a Simulink standard block. Again, I will make it three.

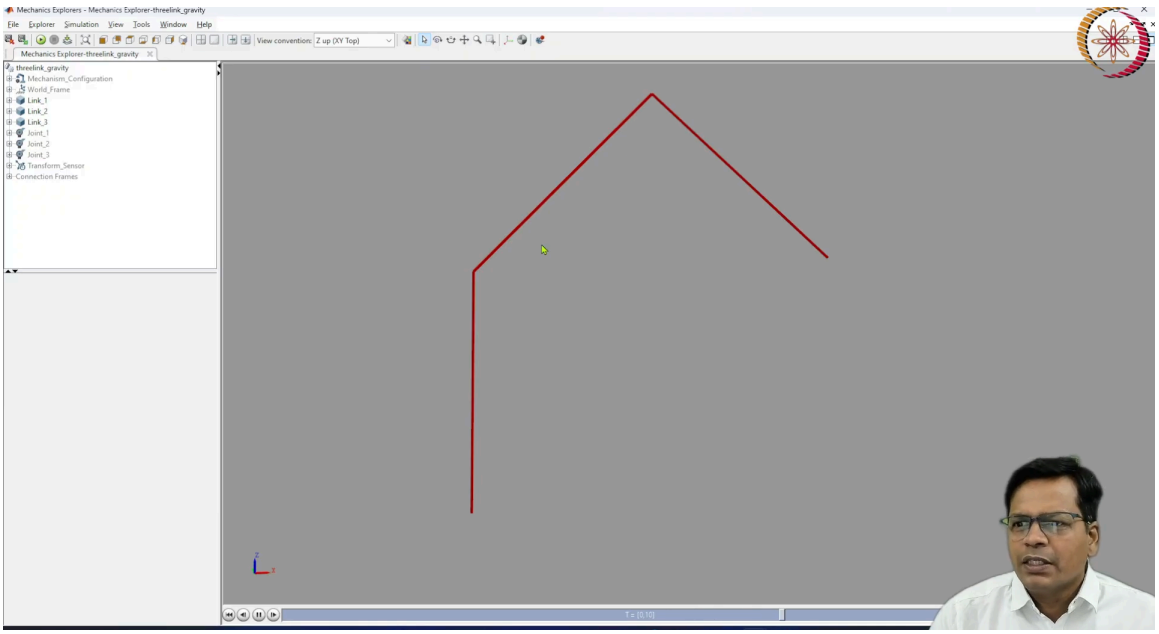
That way, I have done. So, this is it. Now, it will take in the vector and split it into scalar variables that are the joint talks. So, t1, t2 and t3 that will finally go to this. You can directly disconnect it from here, or instead of storing it in a temporary variable and passing it here, it looks very clean, though. You can directly delete this t1 block, go to and from the block, and connect it directly from here to here. In that case whole of this Simulink scheme becomes too much clumsy because all the wires will go from here to there for all three joints. So, in order to avoid that, make the system very clean. I have stored it in a t1 variable, and I have passed it to this. So now, this model is ready as it is, and it is ready to run. So, you can directly try it out for the gravity compensation if it is working or not.



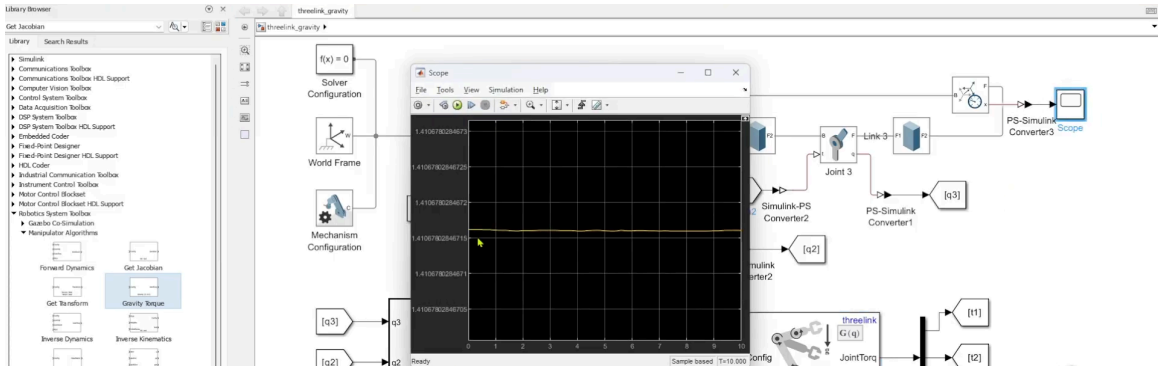
So, as soon as I execute it, it will compile and it will get executed. So, you see it is this robot is not any more falling back. It is not freely becoming a double pendulum kind of thing. So, it is not oscillating, it is remaining stationary, it is it remains stationary in its place. These are the gravity centre of gravity locations for each of the links.



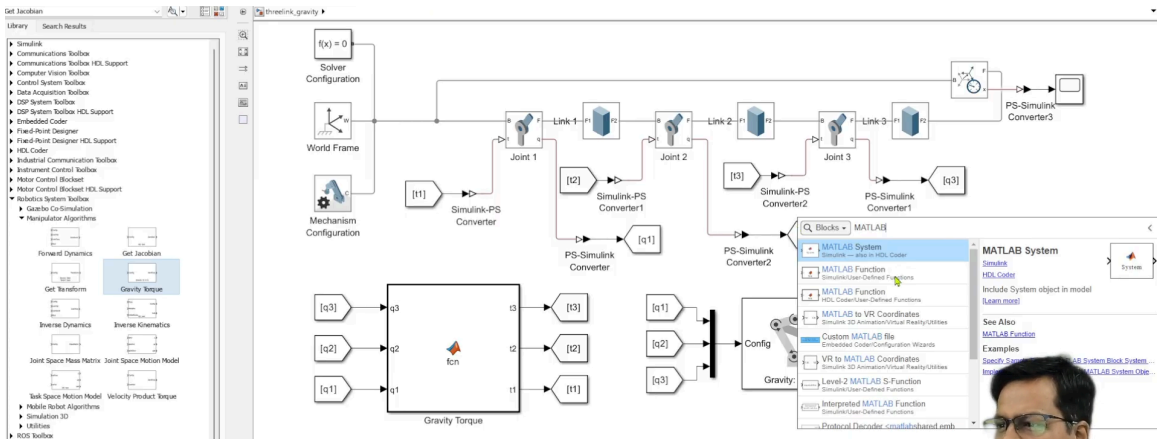
So, the initial state here I have defined, as you see for joint 2, is minus 30. For joint 1, it was 0. For joint 3, it is plus 60. That makes the robot configuration, the initial configuration, and it is not getting changed, you see. If I change it again to joint 2, I am making it minus 45, and again here, I am making it plus 90.



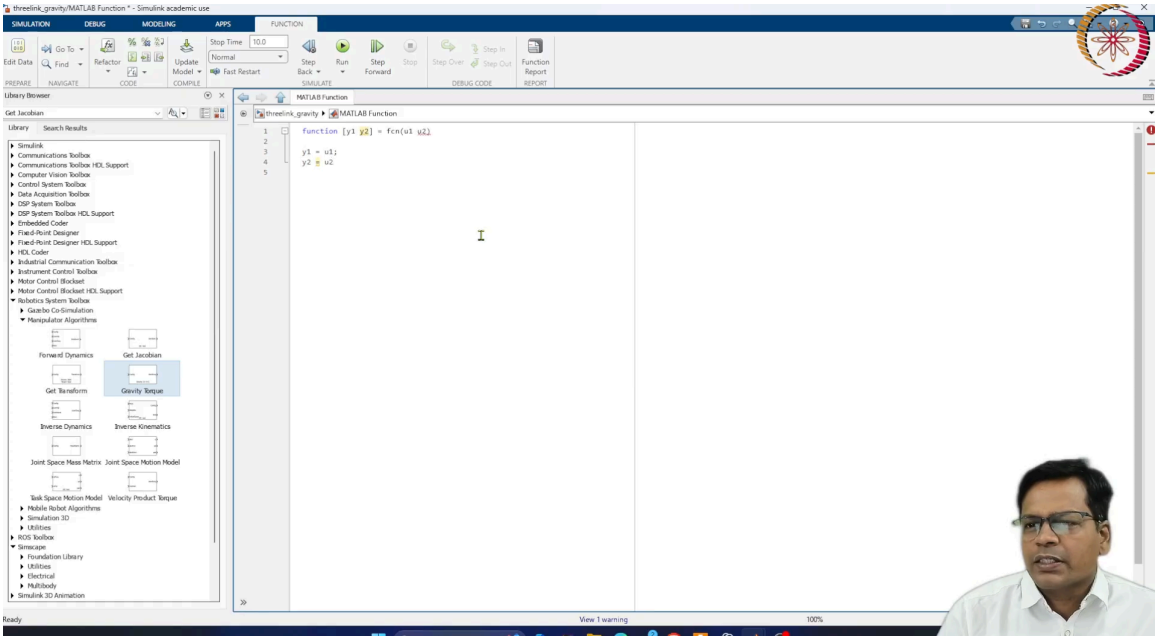
So, again, I will run it. You see, it is like this, and it is again stable. It is not falling on its own because you are providing it with a gravitational torque, which is necessary to maintain this force that is known as the gravity compensation torque that is being applied here. So, that becomes my model.



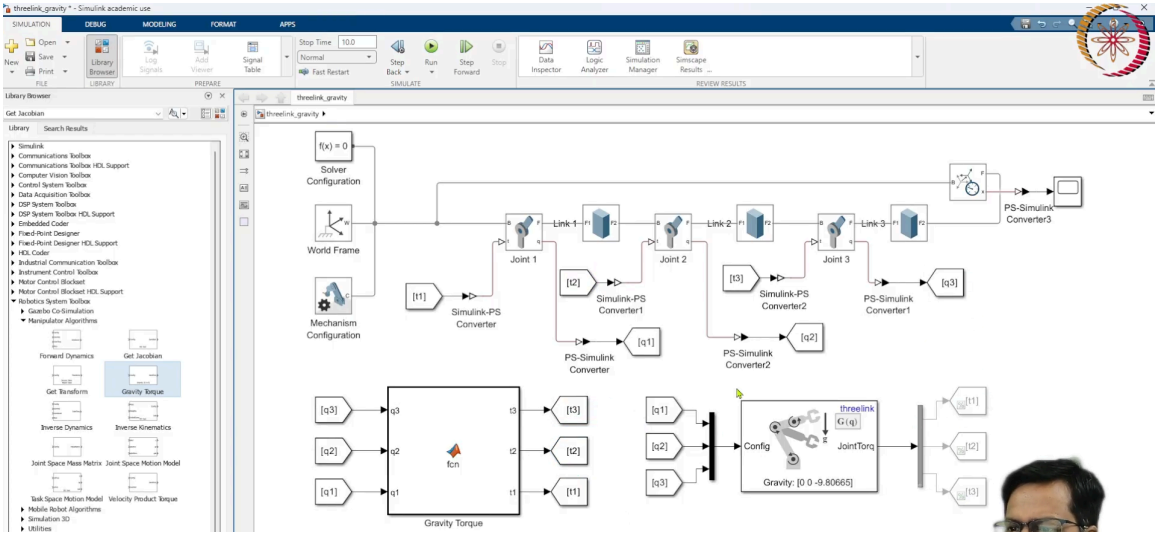
So, I will put it back to some value, and that should remain as it is. So this way, you can try it out with different angles, and you can see if it runs. So, this is what is the meaning of gravity compensation torque and you can use this torque block directly in your force control algorithm. So, this can be used. This is the end effector position that remains stationary. It is not getting changed. End effector X. You can plot for all three. End effector X, Y and Z. All the plots can be seen. The robot does not change its position.



Now I will make this under remark. I will put it in a comment now. I will use this dot. This is what? This is my MATLAB-coded block, which can be inserted from here. This becomes a MATLAB function, a user-defined function. So that can be put here. So, you see, this is the block. You double-click it.

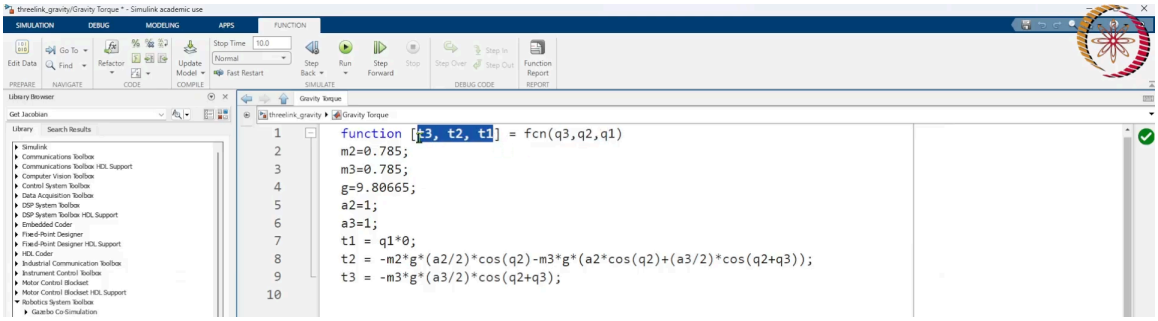


So, this is the default function. So, y is the output from this function. You can put it in vector form as y1 and y2. Similarly, input could be u1 and u2. In that case, you can return y as y1 is equal to some value. I will just demonstrate it using y2 is equal to u2. I am just taking the value and returning it. You can have a predefined function for input and output here; u1 and u2 are taken here. Now, if you go back and see, it can take two inputs, u1 and u2, and give you two outputs, y1 and y2.

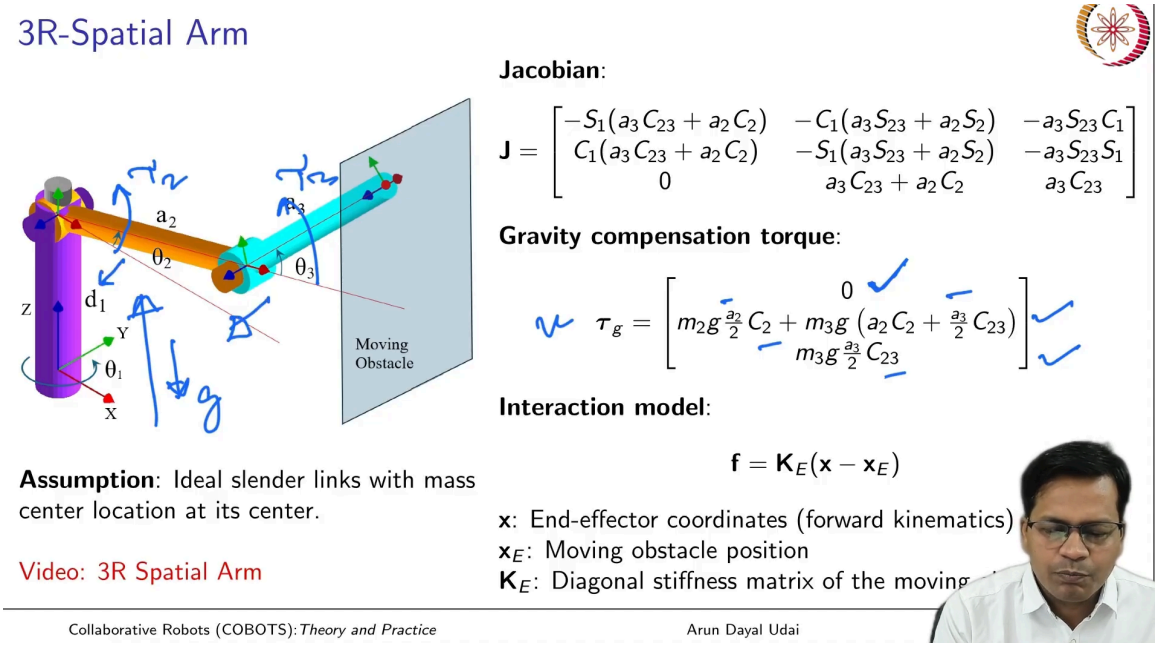


So, this is how you define a MATLAB function that is embedded code. You can put it the way you code it in MATLAB; you can put all those here. So, using that, I have defined

this gravity torque vector here. It will take in all the joint angles, q1, q2, and q3, and give you tau 1, tau 2, and tau 3. Those are directly connected here.



Let us now peek in here to see what is inside. So, I will double-click here. I will enlarge it a bit so that it becomes very clear. So, I have put the masses here. That is mass for the first link, the second link, and the third link. The first link is not getting affected by gravity because it is perpendicular to the ground. So, it is not coming into the picture. So, if you see your gravity torque, how was that?

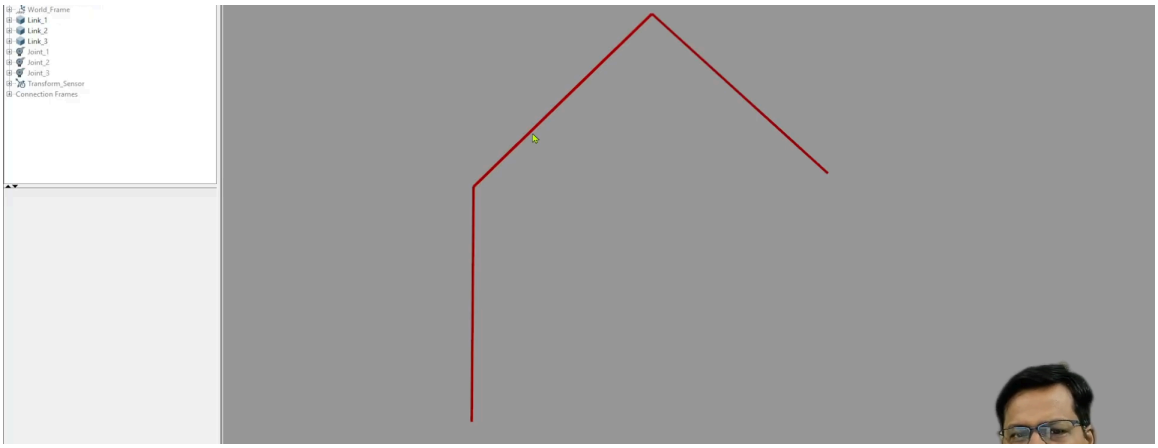


It is like this. So, it is exactly like this. So, you see, you have zero, that is tau g. For tau 1, it is 0; tau 2 is this, and tau 3 is this. They all depend on link lengths a2, a3 and joint angle theta 2 and theta 3. So, it is independent of joint angle theta 1 again because the first link you see is perpendicular to the ground, and gravity is directed like this. So, this

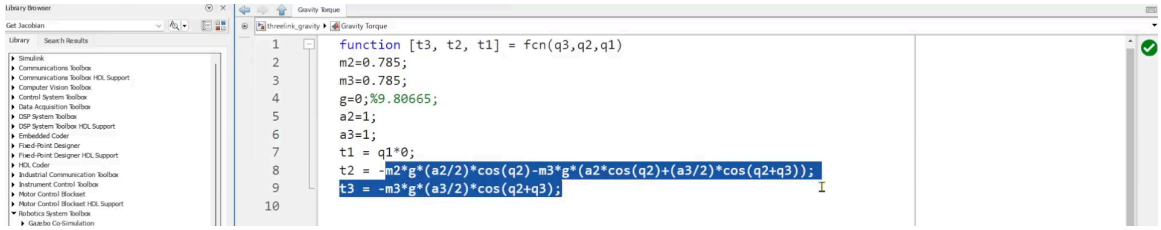
is my gravity compensation torque. Exactly this: I have coded it inside, and taken care of the sign of the torque. Torque positive is like this. So, this is your tau 2. This is your tau 3. If this gives you in plus, maybe it is in minus. You need to take care of how you have put your axis. In my case, I have put the axis like this, like this, but in Matlab, it may be different. So, take care of that and put the signs correctly. That is what I have done. So, I have exactly inserted. The masses you can directly get from here link mass, inertia properties, you get the mass from here, density and the mass. So, if you just go back to custom, it shows you the mass. So, exactly this one, you can take it and put it here.

m2 and m3, g is the acceleration due to gravity that I have taken. Link lengths are a2 and a3. Those are here. This is exactly what I have shown you over here. Exactly this torque is coded here in the metal equation.

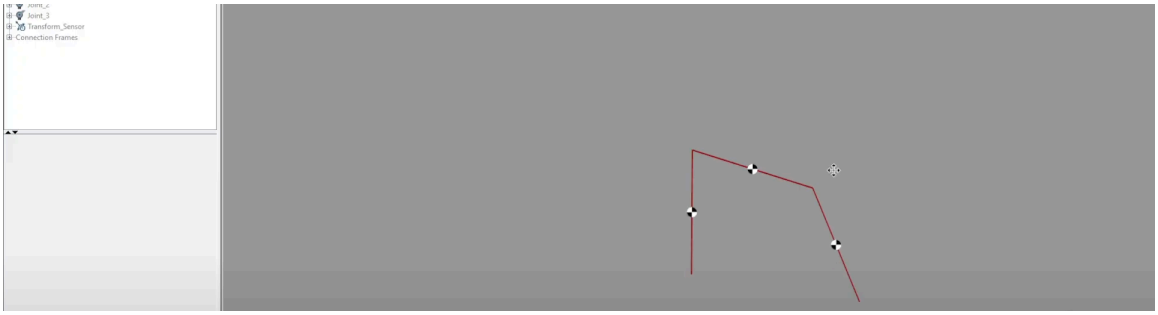
So, the input variables are joint angles q1, q2 and q3. Output variables are t1, t2 and t3. That is passed through this.



Now, let us see how it works. This is again stable. The initial state was whatever is the initial state was. Accordingly, my function calculated the joint torques and fed them to the robot joints, and the robot was stationary in its place.

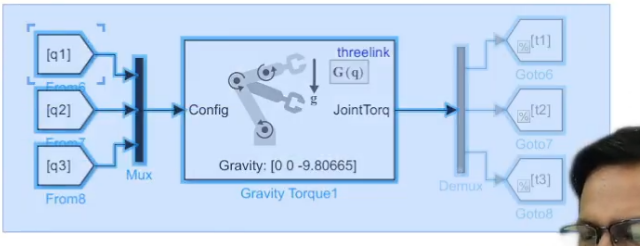


Now, let me try something. If I put g is equal to 0 here. So, if gravity is 0, this should behave accordingly. So, in this case, I am in the code assuming gravity is equal to 0. So, this should give me an output torque, which is equal to 0 because all these contain g. So, all the torques are 0. So, that means it is no longer supporting the links in its place. So, it should fall on its own.

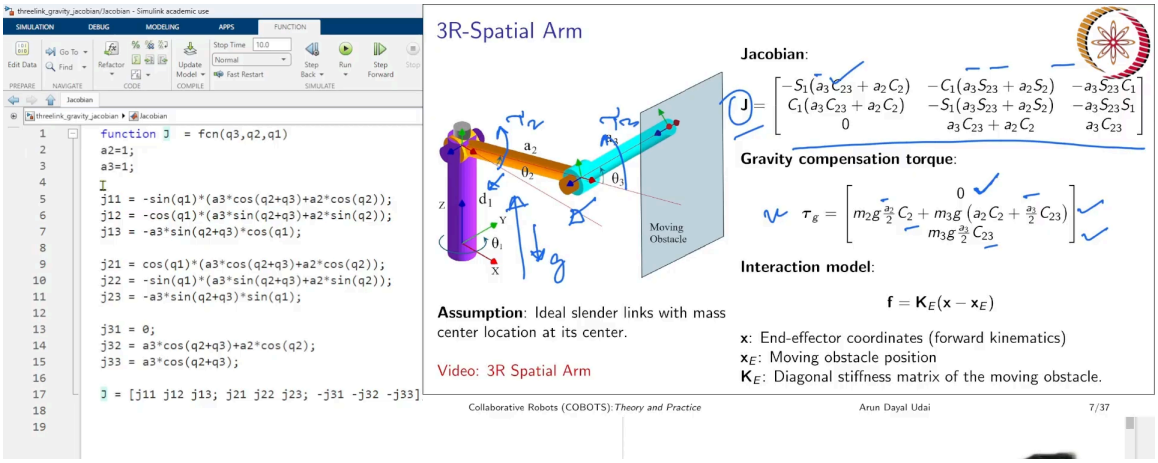


So, as soon as I compile it, it falls like a pendulum. Did you see that? So, this is how it will freely swing, and it will do this motion. Got it? These are the link mass locations. So, this gravity compensation, as soon as I activate it once again, will come into action. So, I will put it back the way it was. So, I have taken 9. You can make it go on the mass. If you take gravity as 9.81 by 6, so it is gravity, I am assuming that is 9.81 by 6, which is on maybe the moon or somewhere. So, you can change the acceleration due to gravity here. Accordingly, it will support it, and now, if I run it, it behaves a little differently. It falls, but not that fast because it is getting compensated a bit, not completely. So, that becomes my gravity compensation torque.

So, this is the physical significance of having a gravity compensation torque. So, it makes you feel as if there is no gravity at all. So, as long as you keep on compensating it, the robot behaves equally in all directions. So, the robot comes into space. So, this is gravity compensation. So, I will leave it as a remark.



I will not be using this from the robotics toolbox; rather, I will be using the gravity torque that I have coded myself. At least for three links, it is very simple, but for more than three links, So, you may have gravity compensation torque, which is calculated using a ready-made toolbox, the robotic system toolbox. You can use the gravity torque block. But in my case, I am keeping it as it is. So, this is gravity compensation. So, I have designed it. I have designed it using the MATLAB toolbox. I have designed it using the Robotics Toolbox MATLAB embedded code, and I have tried both.



Now, I will move ahead with the robot. So, I will do the Jacobian calculation once again. So, gravity is over. I will not play with this anymore. Now, I have coded the Jacobian. Inside this, again, I have taken input as q1, q2, and q3. Those are the joint angles. You know, the Jacobian is a function of joint angles, if you see here.

Jacobian:

$$\mathbf{J} = \begin{bmatrix} -S_1(a_3C_{23} + a_2C_2) & -C_1(a_3S_{23} + a_2S_2) & -a_3S_{23}C_1 \\ C_1(a_3C_{23} + a_2C_2) & -S_1(a_3S_{23} + a_2S_2) & -a_3S_{23}S_1 \\ 0 & a_3C_{23} + a_2C_2 & a_3C_{23} \end{bmatrix}$$

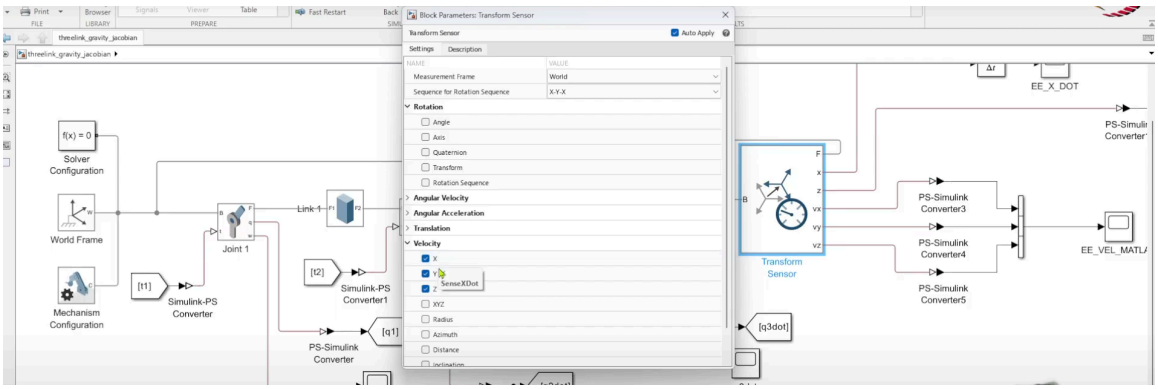
So, it takes in the joint angles and link length. So, Jacobian, this is exactly coded here, okay? So, you have all the elements visible here as j11, j12, and j13, so all the elements they are there here. So, exactly these elements are coded here. I am returning that as a matrix here, okay? I am clubbing all j11, j12, and j13.

The next row is j21, j22, j23. Next row is j31, j32, and j33. Again, I am considering the signs, which should be properly put. That is the reason when you code, you need to check, you need to cross-verify your joint angle direction, and axis direction, and accordingly, you need to do some changes and verify. So, this is your gravity compensation.

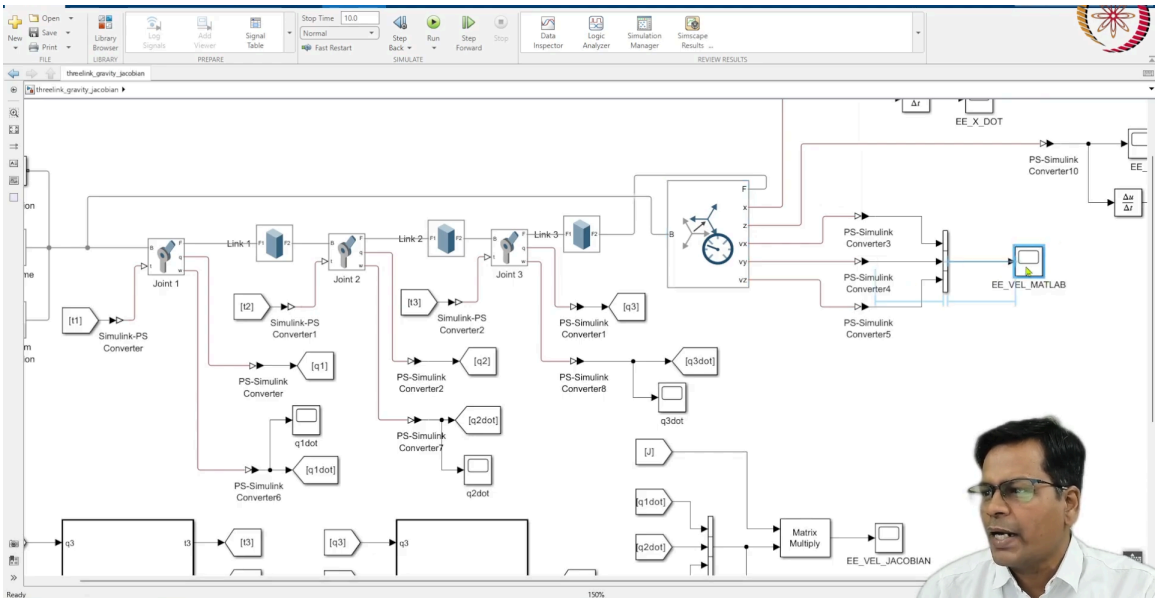
Now, let me check how you will check it. So, in order to check that, what I will do is remember my equation for the Jacobian. What it does basically is you can directly use the equation J theta dot is equal to end effector velocity.

$$\dot{x} = J\dot{\Theta}$$

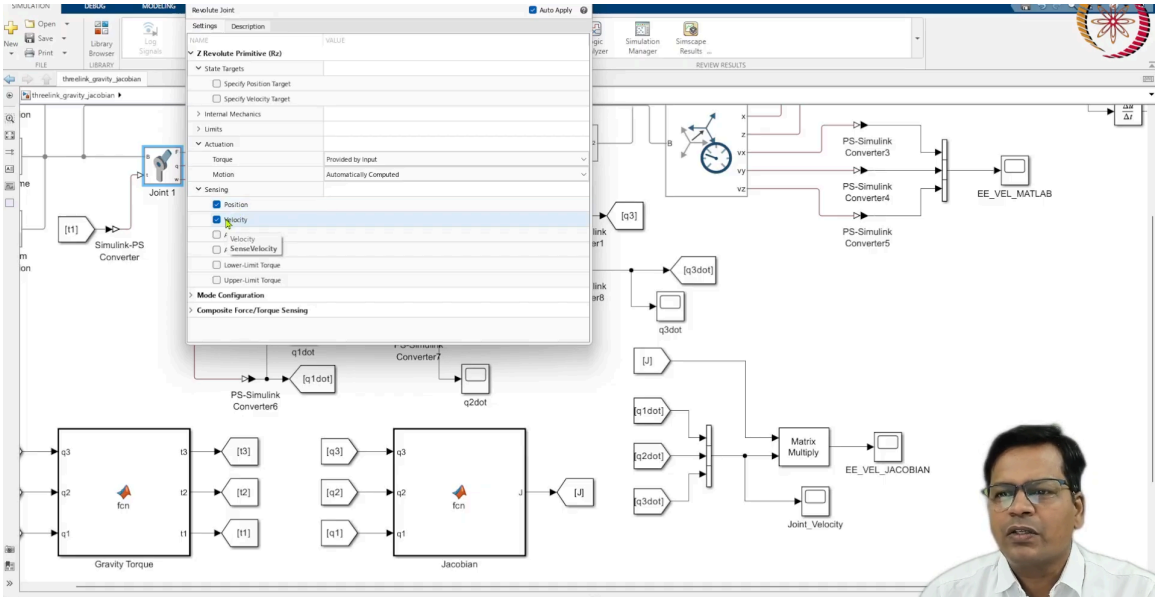
So, this is the standard equation that I will be using. So, I will directly take input from the joints and multiply with the Jacobian, and I will obtain the x dots, and all the end effector velocities. End effector velocity I can also obtain using the end effector transformation block. Transform sensor here, and you know, I can directly obtain the velocities from here.



So, velocities can be obtained velocity along x, velocity along y, velocity along z; all three can be obtained, okay? And I'll compare that. I'll store it first.

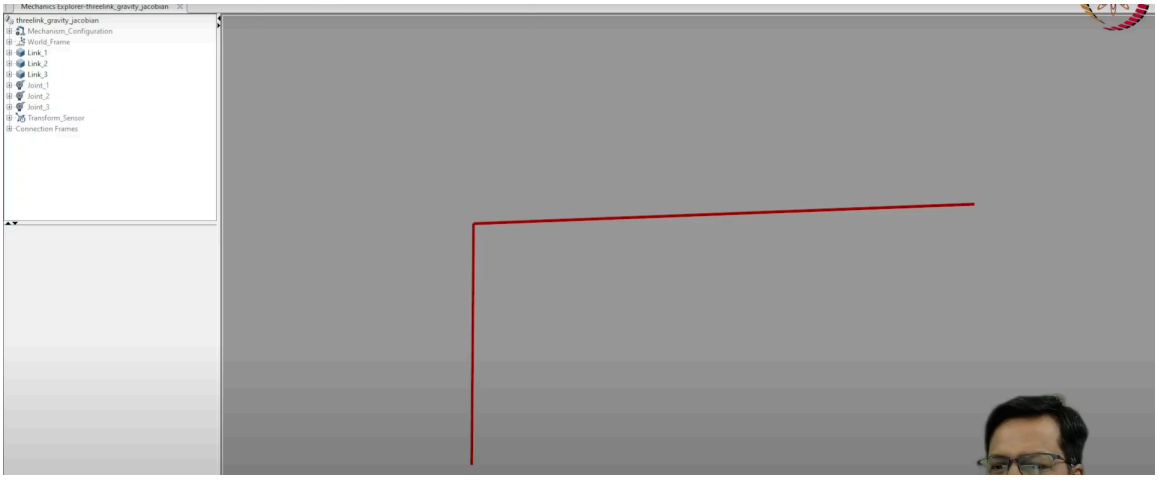


I will plot it here. End effector velocity from MATLAB. That is using the transform sensor. I will plot it here. So, I have used the PS to Simulink converter for all the velocities that come from the physical system. Converted it to Simulink and clubbed them together, and plotted it in a single plot scope.

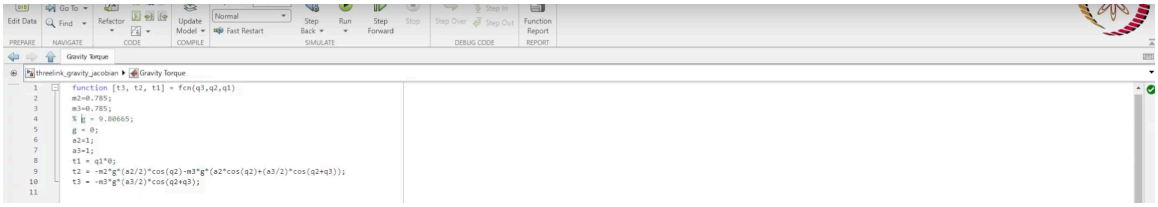


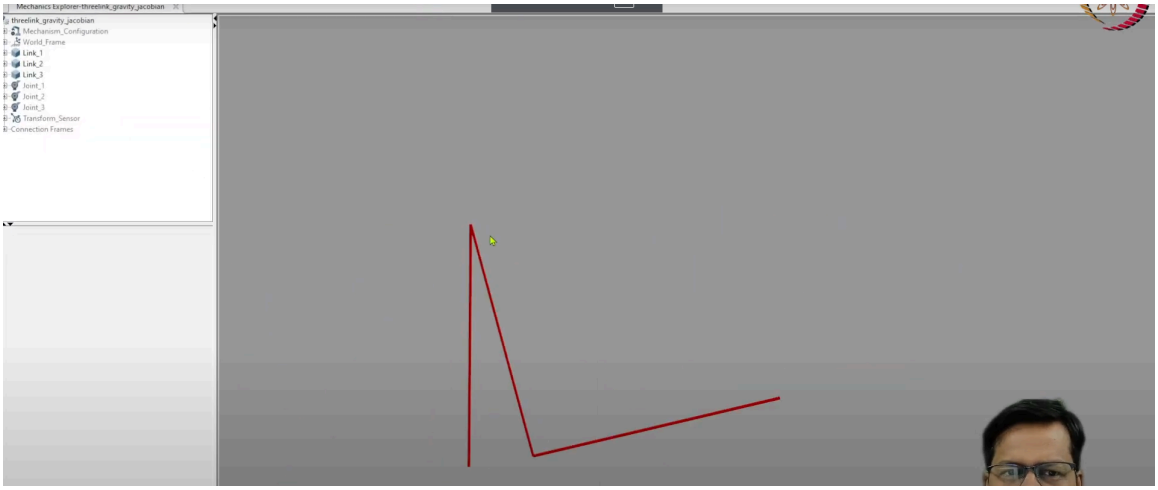
Now again, I have used the joint angles here: q1 dot, q2 dot, q3 dot. So those are the additional outputs. Now, I have taken it from here, so all the joints are not only sensing the position but also sensing the joint angular velocity. So that velocity again is saved as q1 dot, q2 dot, okay? q1 dot here, q2 dot here, and q3 dot here. So, those are the input

here, converted it to the vector form, multiplied by the Jacobian. The Jacobian that I have obtained through MATLAB, and I have obtained the end effector velocity using the Jacobian. So, using the Jacobian, I have calculated the end effector velocity and using the MATLAB transform sensor, I have calculated the velocity. Both should match, and that will verify my Jacobian if it is correct or incorrect.

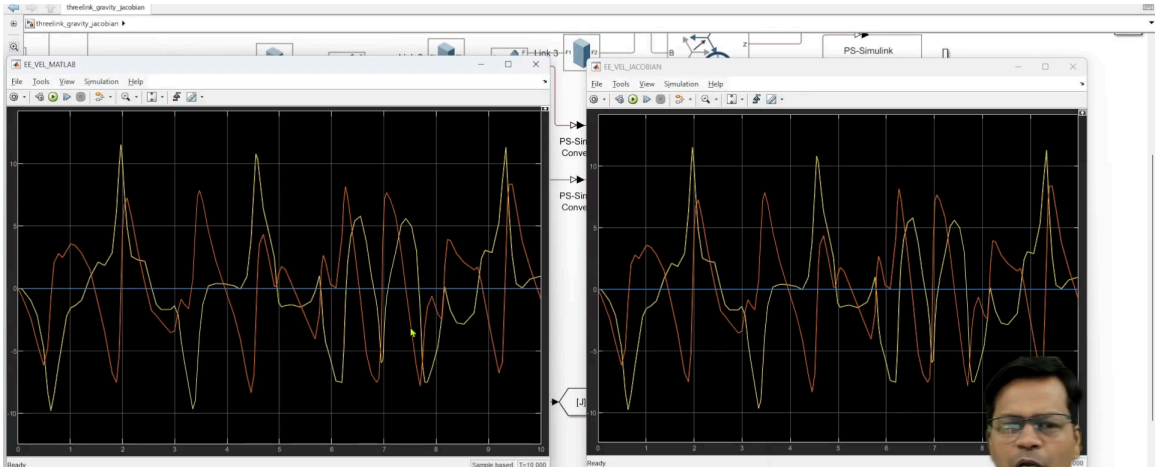


So, let me run this. Okay, now it is stationary, so it is not moving. What I should be doing here is I'll make the gravity equal to zero so that it falls, as to see some changes in the end effector velocity.

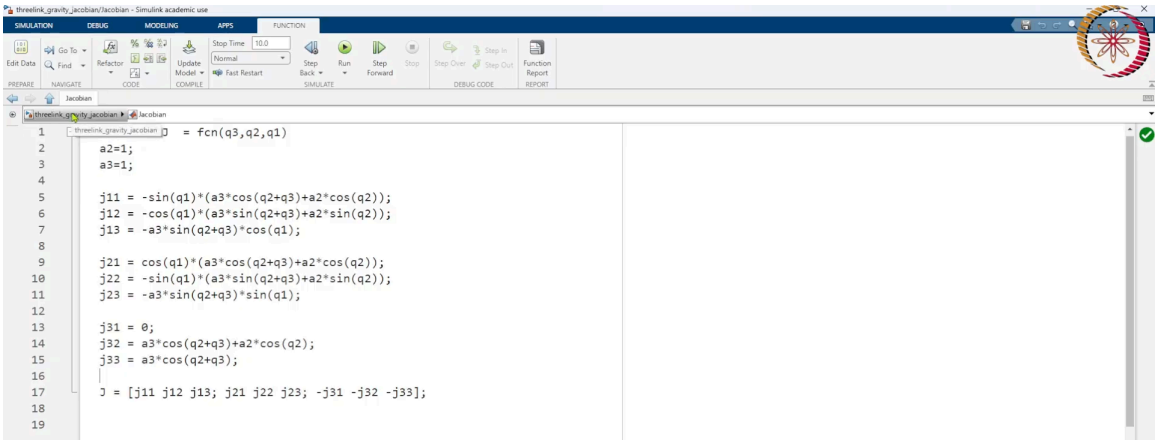




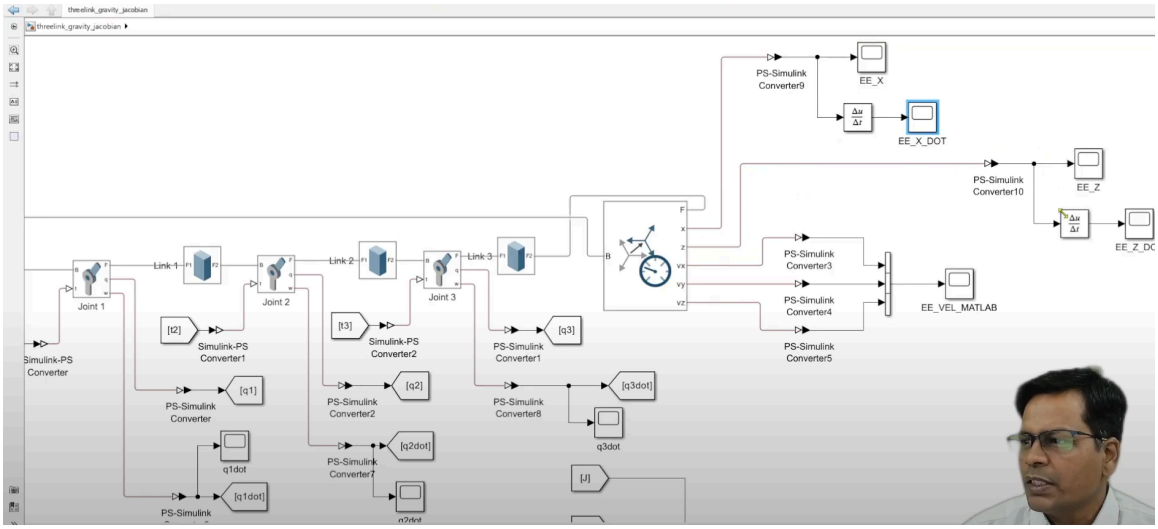
So, I'll just remove my gravity compensation here. I'll make g equal to zero. g is equal to zero, so that it falls on its own. There is no supporting gravity compensation torque. So, it will fall. It falls like this. As soon as the simulation is over, I will check the plots. I will check if they are correct or not. So, this is how it has been executed.



Now, I will check the plots. The end effector velocity from MATLAB is this. The end effector velocity from my own calculation is this. So, these are the two plots. You see, both are exactly the same: end effector velocity from MATLAB and end effector velocity from my own Jacobian. Both the velocity along x and the velocity along y are plotted, and they are perfectly matching. That means my Jacobian is correct.

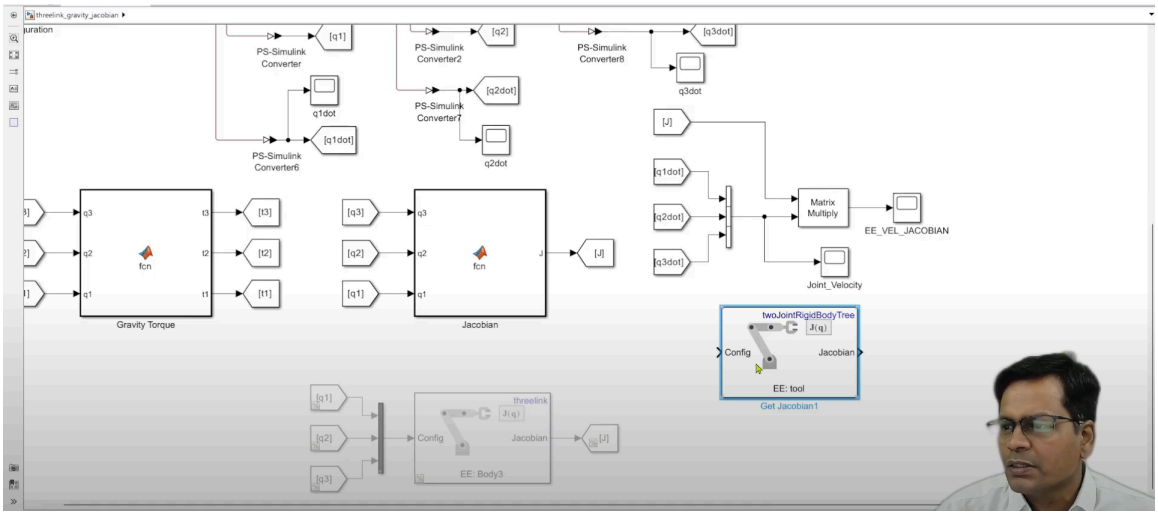


So, you must have noticed I have made some changes to my Jacobian calculation. I have adjusted some positive and negative signs accordingly. You see, I have put negative, negative, and negative here because that joint angle was not at the axis of how I calculated my Jacobian analytically. So, while putting it here, you have to check your axis direction before you actually place it here. Only then will it match. So, you need to make those changes.

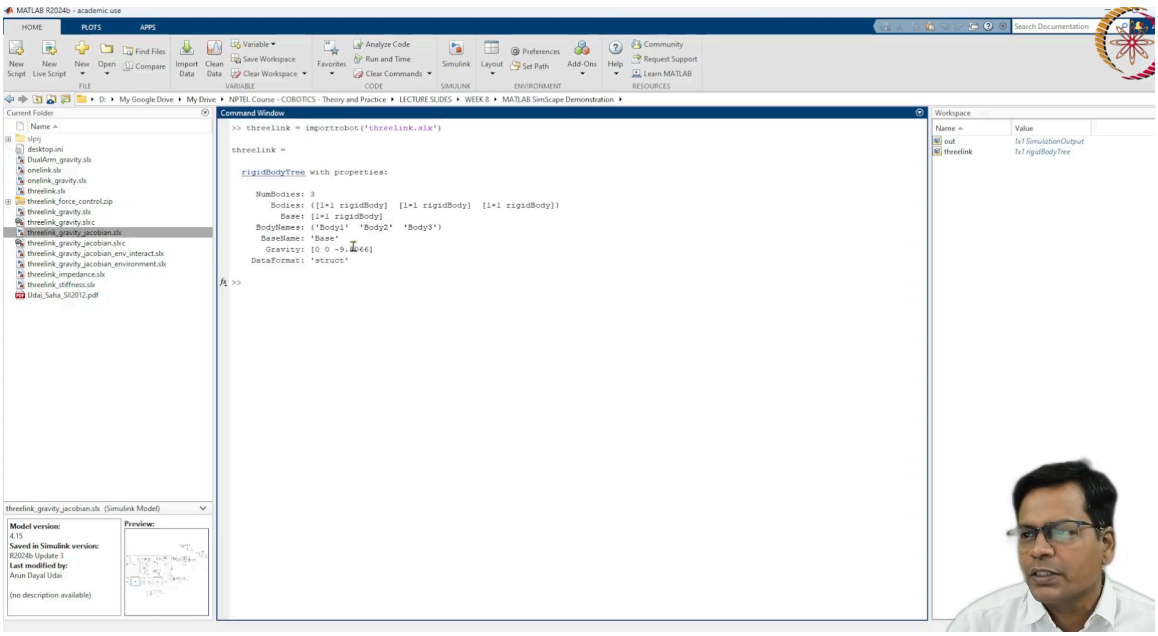


So, those changes were made here, and I have found them correct. So, this is how you can verify, and the end effector velocity can also be obtained in multiple ways. You can directly check the displacement, take the derivative, and get the velocity along x. Similarly, velocity along z. So, you can directly take the value here, take the derivative, and find out the velocity. That is another way to calculate velocity. This way, you can

directly obtain the velocity, or you can obtain it using your Jacobian. So, that way, you validate your Jacobian. So, the validation of the Jacobian is done.

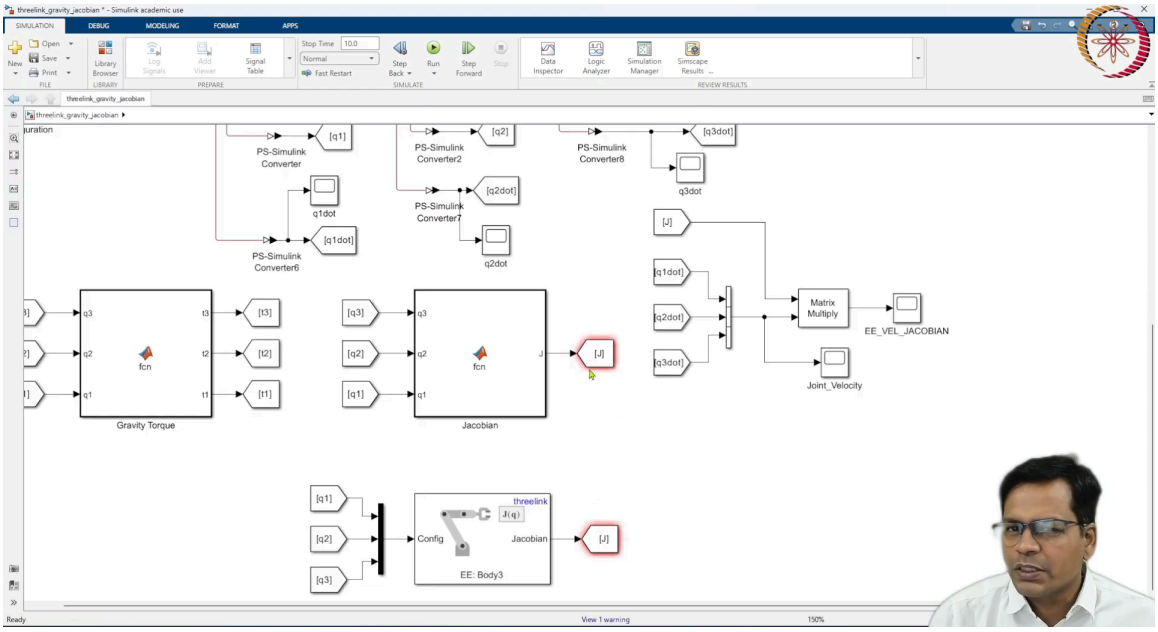


Again, you can directly use the robotic system toolbox to get the Jacobian. So, in the robotic system toolbox, you double-click here and go to Jacobian, and you can directly get the toolbox called 'get Jacobian. This uses the rigid body tree, as it is written here.



So, you need to execute the same in case of gravity. Also, it uses the rigid body tree, and you already know how we can import that here. So, I will call it treelink is equal to import robot in quotes. I have taken the model name as three-link-dot-slx. So, when you do this, you can directly import that as an object in your workspace. So, that has to be

included here. So, when you do it here, you write three links. Three links here, and this becomes complete. So now, it can take in the configuration as you have taken it in your Jacobian. This is how it is used. So, I have already done it for you. I'll uncomment this, and this makes my Jacobian using robotics toolbox.



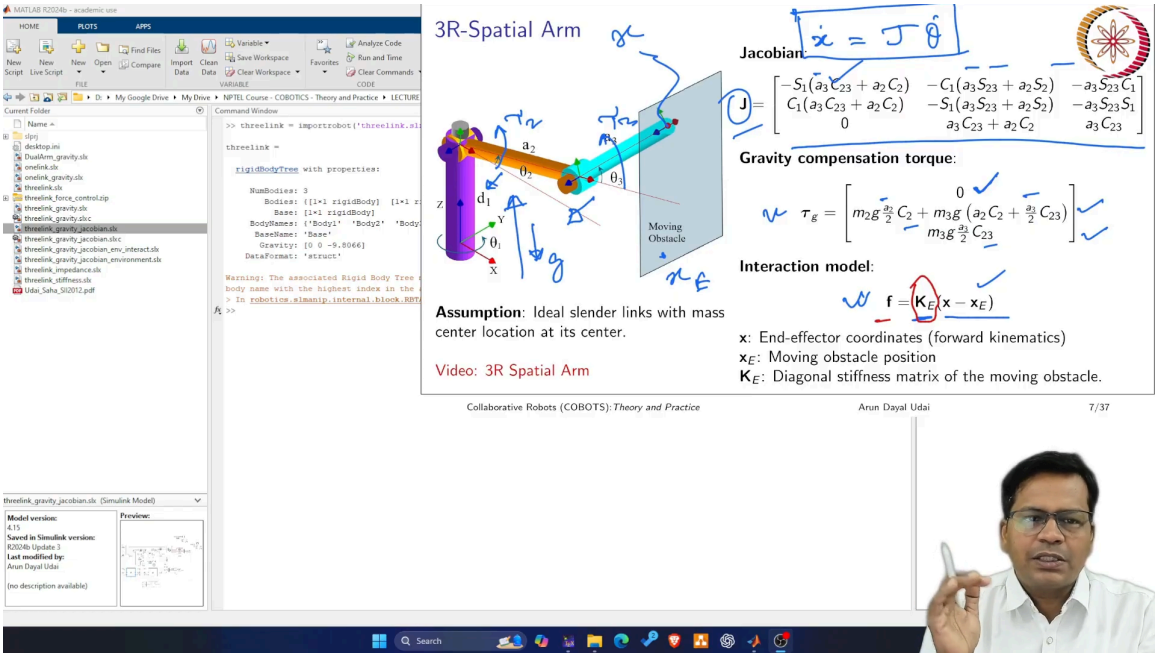
It is shown as red because you cannot save the same variable from two different sources. So, you have to make one of them in the remark. If you put this in a comment, then this becomes okay.

So, this is how you can calculate the Jacobian using the Robotics System toolbox. But I will be using as because it is a three-link manipulator. So, it is very simple. I will be using my own block that I have made using this code.

```
function J  = fcn(q3,q2,q1)
a2=1;
a3=1;

j11 = -sin(q1)*(a3*cos(q2+q3)+a2*cos(q2));
j12 = -cos(q1)*(a3*sin(q2+q3)+a2*sin(q2));
j13 = -a3*sin(q2+q3)*cos(q1);

j21 = cos(q1)*(a3*cos(q2+q3)+a2*cos(q2));
j22 = -sin(q1)*(a3*sin(q2+q3)+a2*sin(q2));
j23 = -a3*sin(q2+q3)*sin(q1);

j31 = 0;
j32 = a3*cos(q2+q3)+a2*cos(q2);
j33 = a3*cos(q2+q3);

J = [j11 j12 j13; j21 j22 j23; -j31 -j32 -j33];
```

So, just like gravity block it is it can be used from the robotic system toolbox, even

Jacobian can be used, but I am not going to use this for my future calculations and other force simulations. So, I will stick to this gravity and this Jacobian strictly.

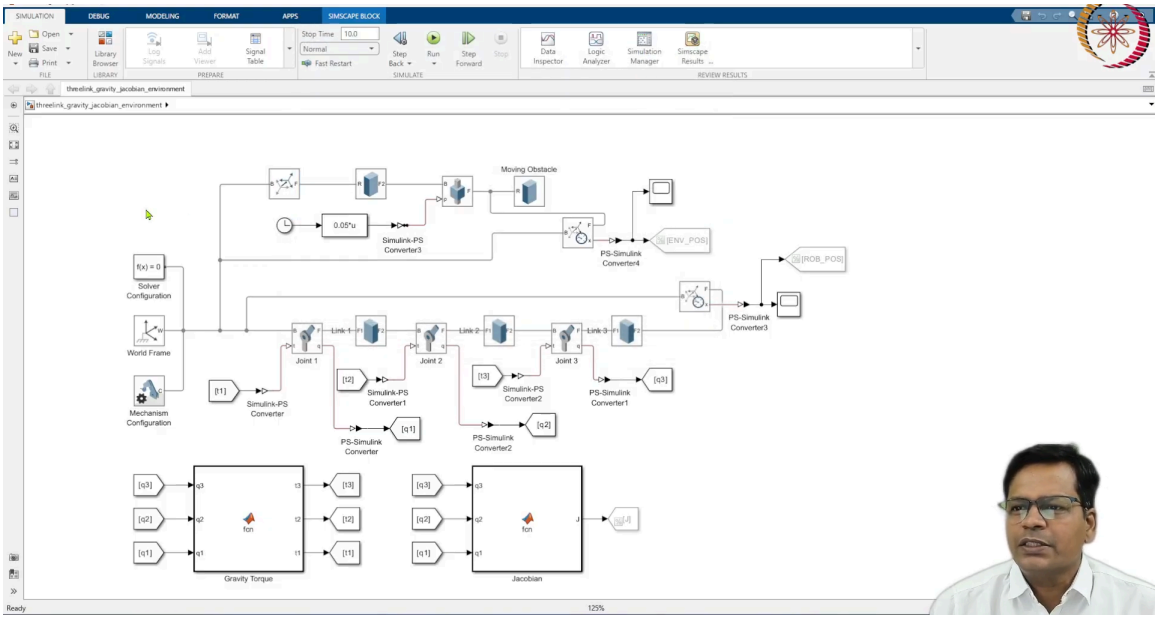


Now, I will create the environment so as to create a plane that moves towards the robot and does something. This is what I wanted to do. So, you see, there is a plane that moves towards the static robot, and I'll see if this plane makes my robot flex or not. Upon interaction, it should become soft or it should behave like a spring. So, that collision is detected using environment modelling. As you have seen, f is equal to K_E into x minus x_E. x is the end effector position of the robot. This is your x. x_E is the position of this moving obstacle. When this collides with the robot, So, there is a difference in position. It moves towards the robot. So, wherever you have commanded your robot to go, it gets into the robot's position, and the robot penetrates into that object.

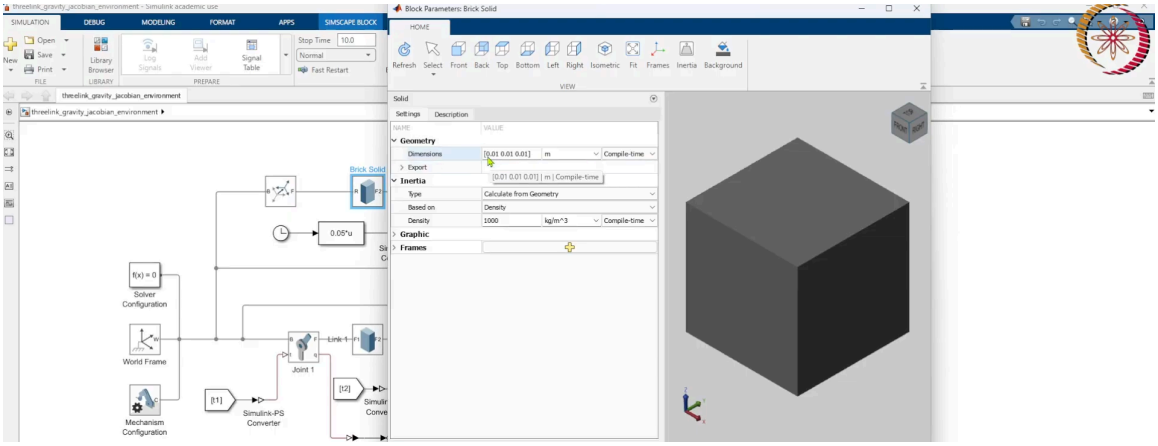
So, if there is any penetration, and because of the stiffness of this object, which is there. It creates a force. A surface normal force that will actually resist the robot to getting into this surface. So, this is my model. So, what I have done here is x minus xE into ke should give me the force vector.

$$f = K_E(x - x_E)$$

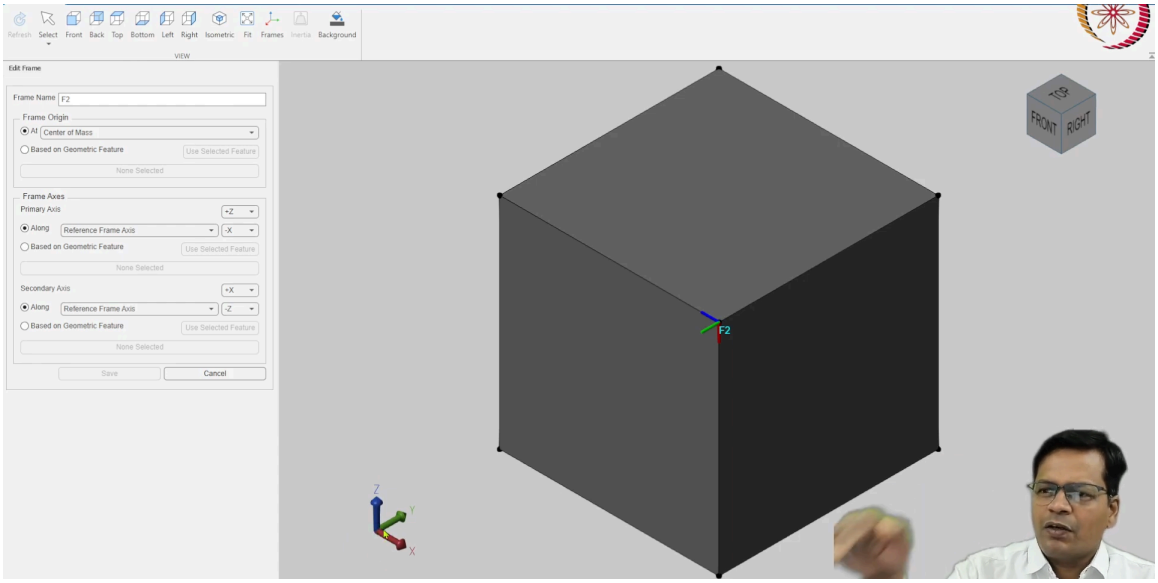
I have ignored any frictional forces along the wall surfaces; I have just considered the normal forces that will come out of this moving obstacle. So, this is my environment, I want to code into my MATLAB Simulink environment, so this is my model. Now I am ready with everything. I am ready with the gravity torque. I am ready with the Jacobian. I have already validated my Jacobian and the gravity torque. So this is my 3R robot, which we have already done.



So now, I need to create a moving obstacle. How to do that? Basically, what I have done here is that with the same world frame, I have created a different system here. So, basically, this is your rigid body transform. I am first making a frame that is in a space about which this obstacle will move. So, this is that stationary frame. This is situated in the robot workspace somewhere. So, I have done that transformation along the z-axis, and some translation offset I have put here. I have kept that at two meters. I know my link lengths are the Second link and the third link are 1 meter and 1 meter. So, in any case, this should start from a position that is greater than 2 meters and start moving towards the robot. That offset is there. That becomes the base of this moving obstacle. So, this is my rigid body transformation.

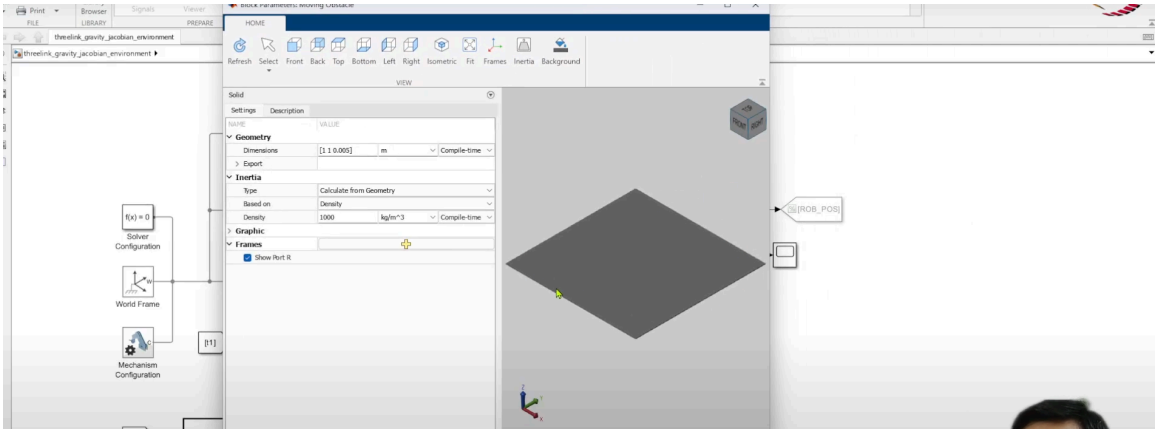


This is a brick solid element, a small element of some density, does not matter. It is a very small reference frame that is located at that location; x is equal to 2 meters, 0 and 0, and this is of negligible dimensions. That is not very important.



Now, I have put a prismatic joint. The prismatic joint will actually make this robot move towards the object. So, now you have frames here again. This is frame is put like this because I want to move along this direction. The negative of the x-axis should be my new z-axis, as you know because my robot is directed along the forward x-axis. So, as because I want my obstacle to move along this direction. So, this is my reference frame F2 is my reference frame now. So, about F2, I will have a moving object. which moves about F2 along this direction that is the negative x-axis. So, that is my this. So, this is your

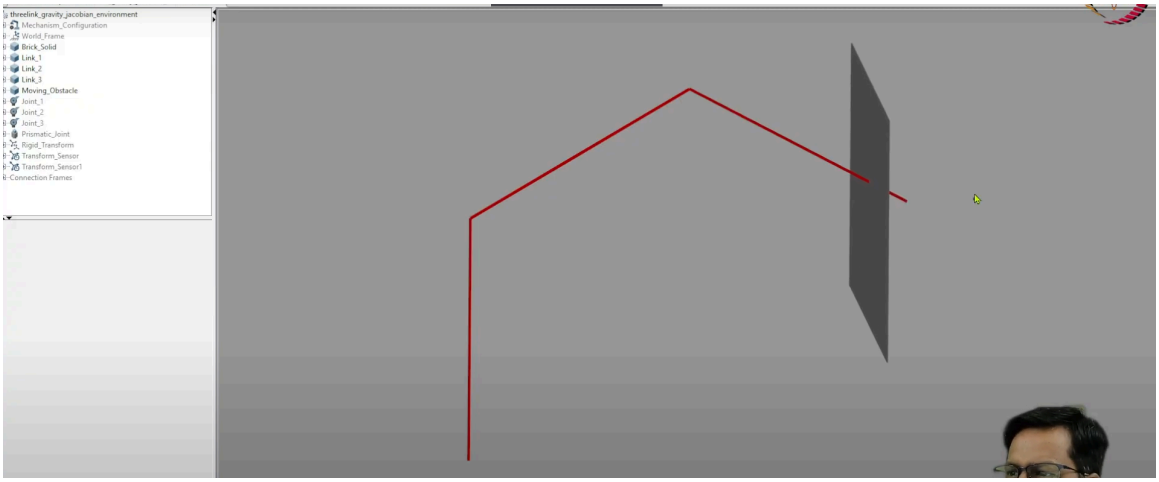
prismatic joint. This will be aligned along the negative of the x-axis, and this is your moving obstacle.



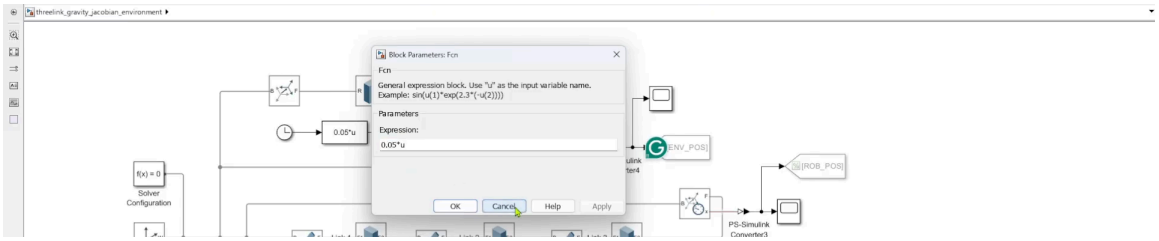
It is nothing, but it is like a plane. Mass doesn't matter, but it has some geometry of 1 meter into 1 meter and a negligible thickness of 0.005 meters. That is how I have taken and read nothing.

I will connect this to the follower to the prismatic joint. So, this is my obstacle. So, now this obstacle should move with time. How? The value that I am taking in. So, this is the clock function. This simply creates clock ticks corresponding to the simulation time. So, this is the clock tick generator, and this is the function block. If you just type fcn, this is the user-defined function block. It can take in some value and give you an output depending on the function that you defined here. So, now I have taken in the clock input. v is equal to, so I have kept it as velocity multiplied by time.

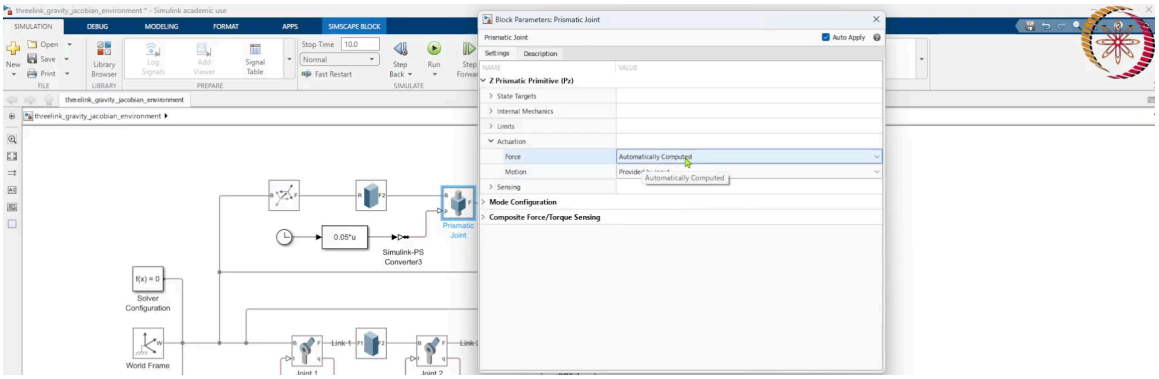
So, this becomes my velocity. Velocity is 0.05 meters per second, and time is the input to this function block. Velocity is multiplied by time. So, this will create a uniformly moving system obstacle toward the robot. Let us see how it behaves, and then you will understand it more closely.



So, you see, it is moving at a, so this is my stationary frame, which is here, about which this obstacle is moving toward the robot at a velocity of 0.05 meters per second toward the robot. Over here, it is velocity multiplied by time. Displacement is equal to velocity multiplied by time.

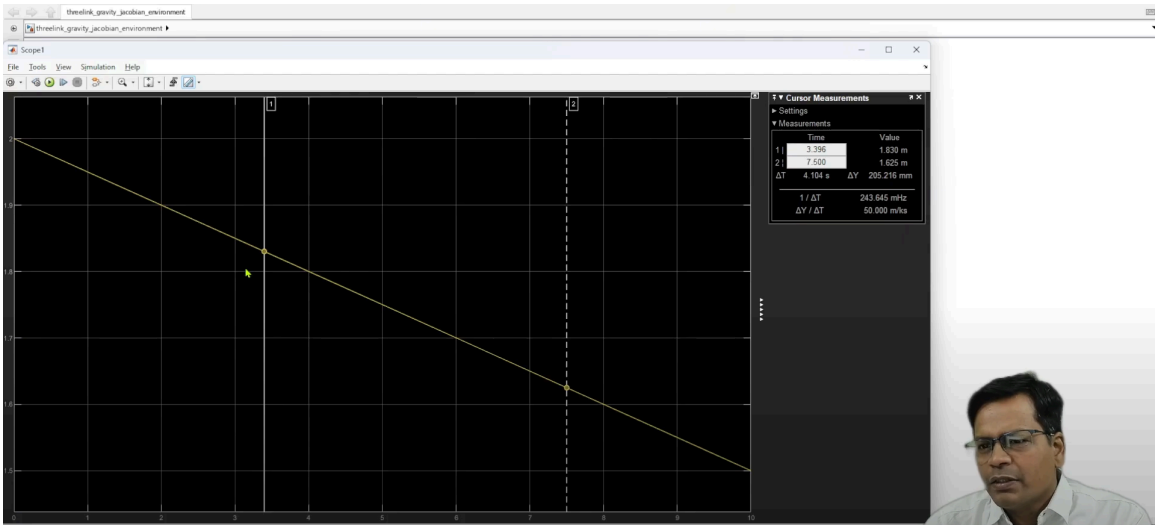


That is what I have taken here. This is a simple equation that becomes my function, fed it to the prismatic joint with actuation as motion provided by the input, and the force is automatically computed. That is irrelevant here. I just want to move it.

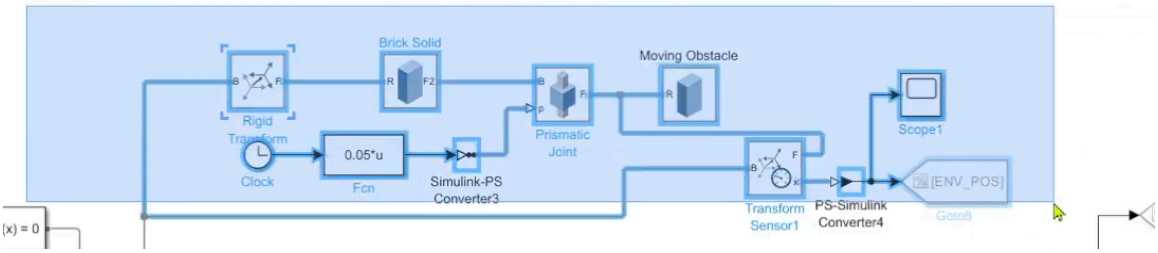


So, now the position of the wall is continuously monitored using a transform sensor. It is doing forward kinematics for this moving obstacle with respect to the base, what is my

current location? So, in the same frame, I have connected as a follower. This is the base; this will give me the displacement along the x-axis.

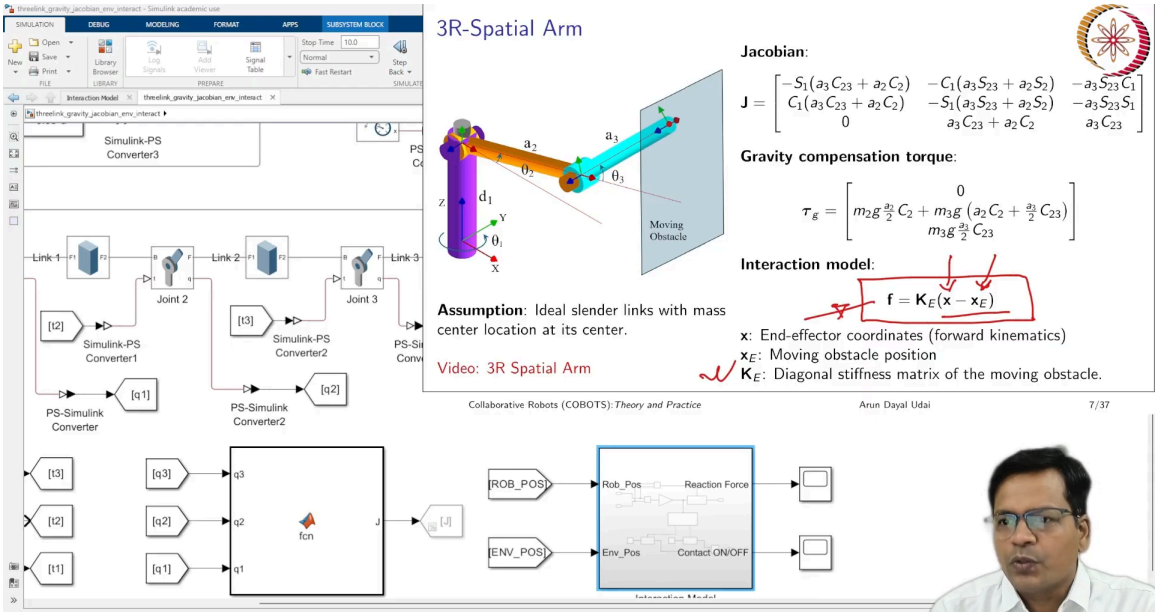


I have stored that here as an environment position block, and I have plotted it here also. You see, it was moving with time. It moves continuously starting from the location 2 0 0. x is equal to 2 0 0 and moving with the constant velocity of 0.05 meters per second. Displacement is continuous towards the robot. So, this is how this moving obstacle is created.

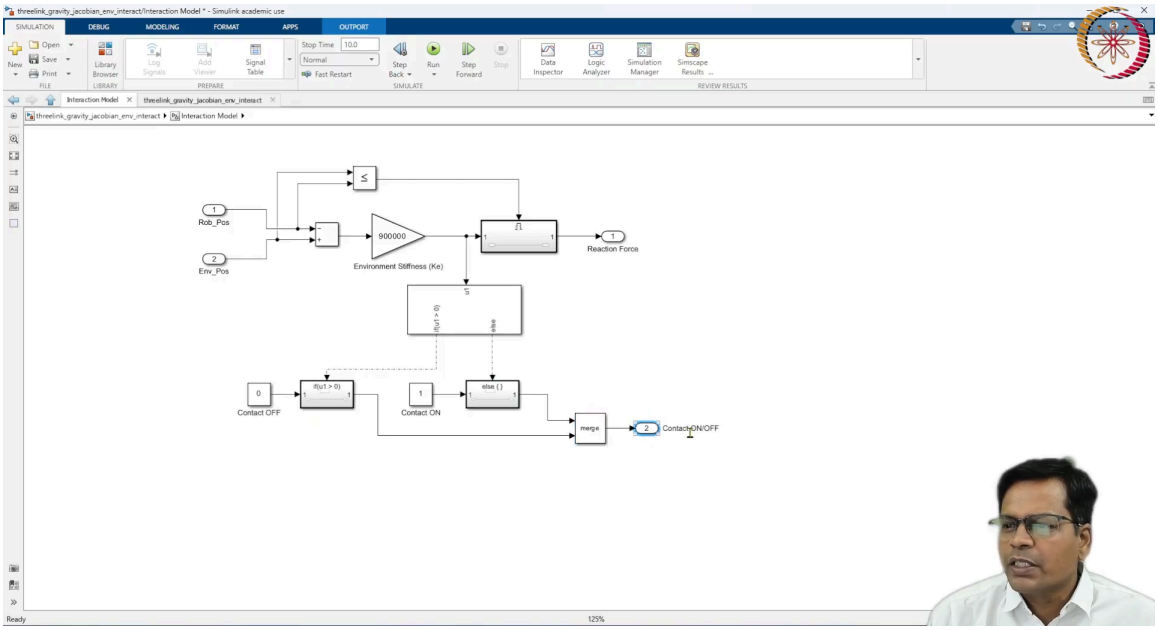


This will be used for simulation only. It has got nothing to do with the force control algorithm here. So, I am saving the environment position; I am saving the robot position. So, using these two, I will calculate the error, and I will multiply it by the stiffness matrix of the environment to generate the forces that come on the robot tip when it interacts with the wall. Currently as because there is no interaction model thus, this robot is in its position, it is not getting deflected, and it penetrates through the wall. So, this is it.

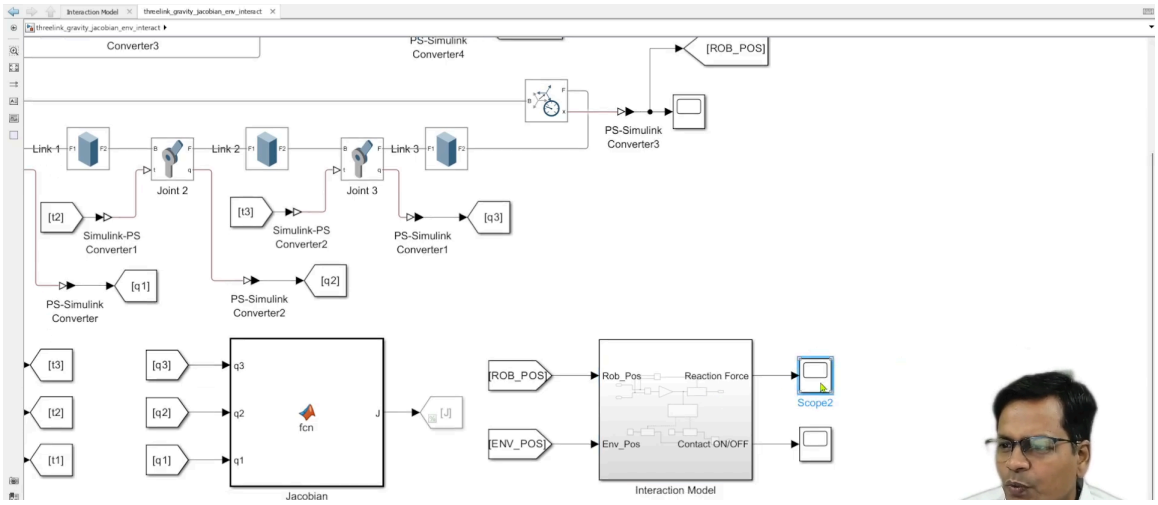
So, now, a moving wall is created. I have saved the environment position and the robot position. I will try to create the environment interaction model.



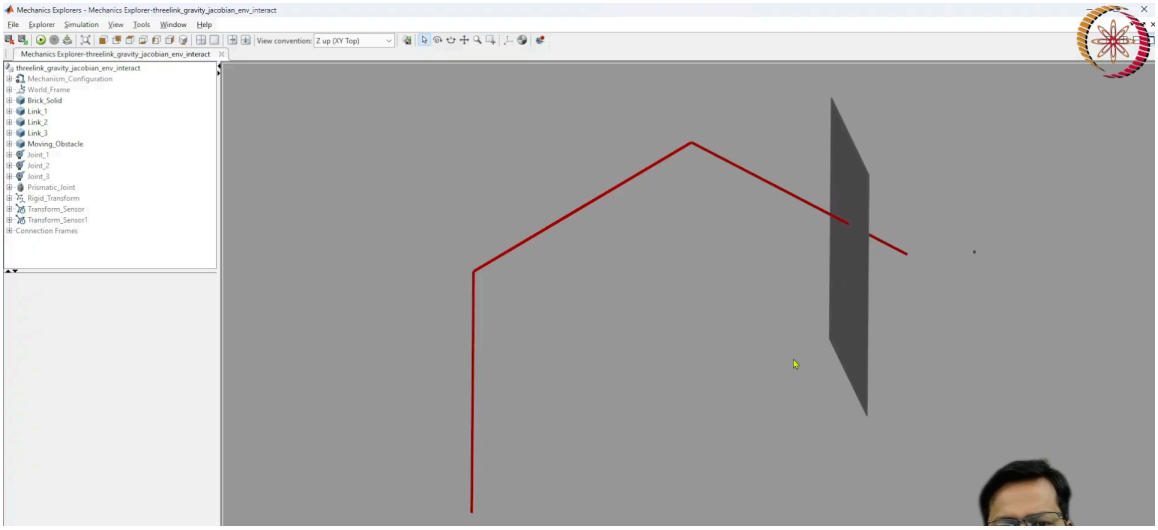
So, it is an environment interaction model. So, this time, this is my interaction model. It takes in the input of the robot's end effector position, the robot's position that comes from here and the environment position that comes from the wall location, moving wall. It was moving towards the robot. So, I am saving both positions. I am inserting it here. Essentially, I am going to code in this equation here using Simulink blocks. So, I will take the robot's position, I will take the environment position, take the difference, multiply it with the stiffness matrix, and the diagonal stiffness matrix of the moving obstacle, and it will generate the force.



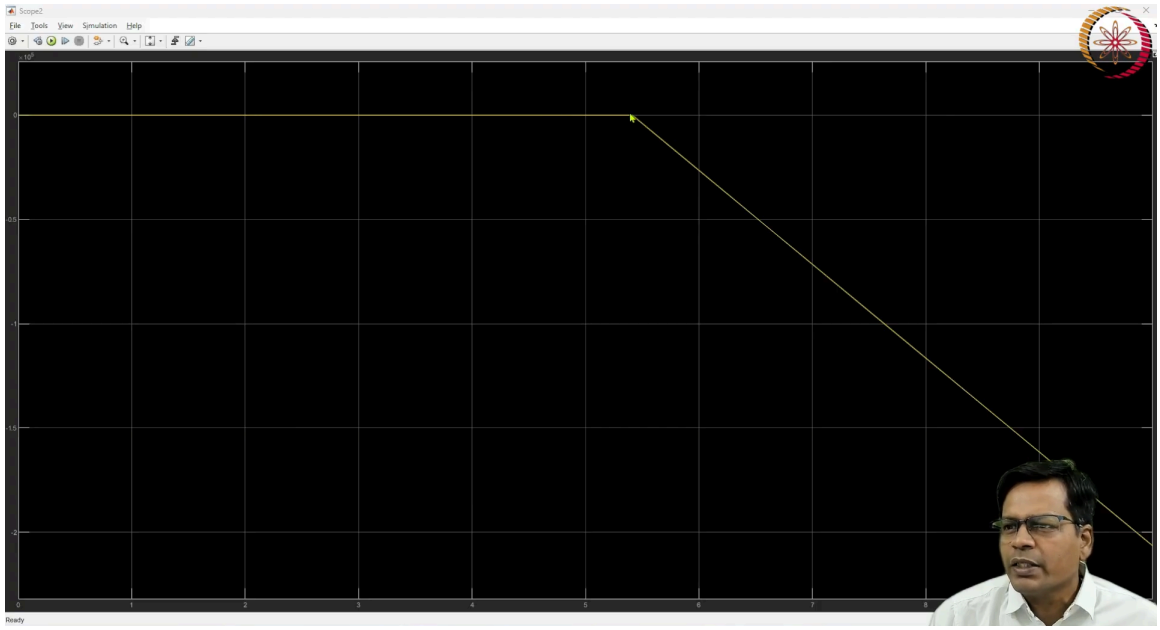
Let us see how I have implemented it in my blocks. So, this is my block. So, the robot's position comes from here. The environmental position comes from here. I am taking the difference, and multiplying it with the environment stiffness to calculate the force. Force comes out when the robot actually gets into the wall. So, the robot position and environment position when it to have some positive value, which means it has penetrated into the wall. In that case, this is an enabled system. These blocks you can find it in your MATLAB Simulink because they are all Simulink blocks. With nothing to do with the robotics toolbox or Simscape Sim Mechanics block. So, this will give you an output of the reaction force. Force is calculated only when the robot penetrates into the wall, okay. Otherwise, when there is no contact, force should not be calculated, and it becomes equal to zero. So, that job is being done over here.



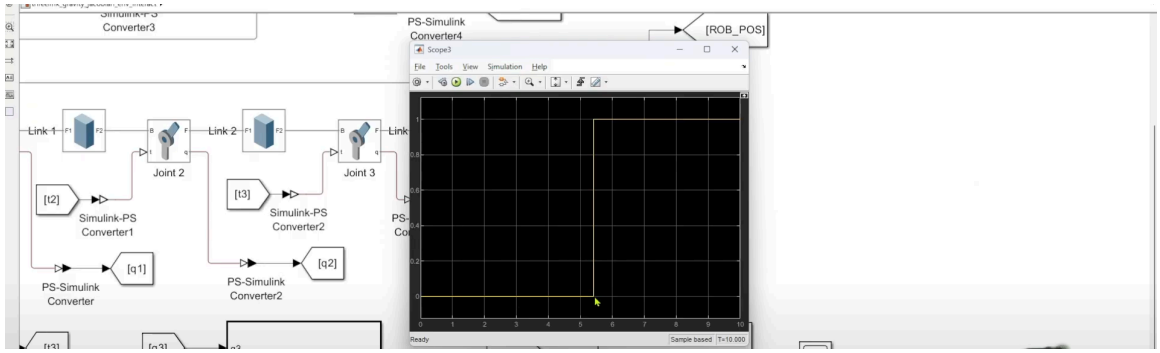
Now, I am calculating whether the contact is happening or it is not happening. So, contact on and contact off is passed through through this. If-then-else block, they are merged to a single output. So, this contact on or off will be given by this environment interaction block.



So, now let us run this once again. So, this is my interaction model. So, this should give me the reaction force as well as a contact on and off situation, status of that. So, if I run this, so as soon as it establishes contact, it won't deflect the robot still because I haven't applied any force control algorithm here, but it should give me the force.



So, it is moving towards the robot. So, you see what it is giving. It was giving zero four, zero four, and zero force up till here when the contact actually have happened, and gradually, it has calculated the force. Gradually, it is increasing because it is continuously moving into the wall. The robot gets into that. As the the robot gets into the wall, the force is being calculated and continuously increased because it is penetrating more and more, and this is the situation of contact.



So, contact has happened here, and it gave 1. So, this block will be used as an interaction block. Other blocks we have already designed and verified, including the Jacobian and gravity torque. So, now we are ready for any force control algorithm to be developed on this. So, I have also created the moving obstacle.

Demonstrations

- Calculating and Testing the Robot Jacobian: Using MATLAB Embedded Code and Robotics Toolbox
- Calculating and Testing the Gravity compensation torque: Using, MATLAB Embedded Code and Robotics Toolbox
- Implementing the Environment Interaction model.
- Testing the arm for open loop positioning and free-fall forward dynamics simulation.

So, what are all the things that I have demonstrated now? I have calculated and tested the robot Jacobian using MATLAB embedded code. This is what we will be using, and we also did it using the Robotics Toolbox. We won't be using this. Calculated and tested the gravity compensation torque using MATLAB Embedded code and the Robotics Toolbox. Again, I will be using this.

We have Implemented an Environment Interaction model to do any force calculation and implement any force control algorithm on top of that. I have checked the robot for open-loop positioning, and we have also seen free fall under forward dynamic simulation. We have just removed the gravity compensation torque, and we saw the robot falling on its own with zero torque at the input, of how the robot behaves. So, that was the forward dynamic simulation of the 3R Spatial manipulator we have done.

So, in the next lecture, I will discuss Hybrid and Stiffness Force Controllers. That is all for this lecture. Thanks a lot.