

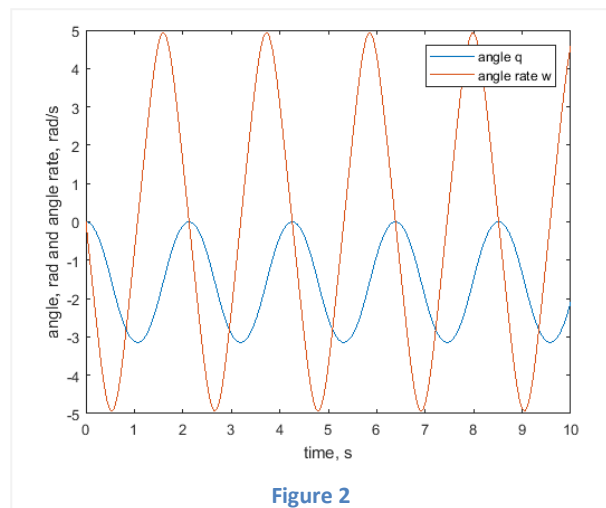
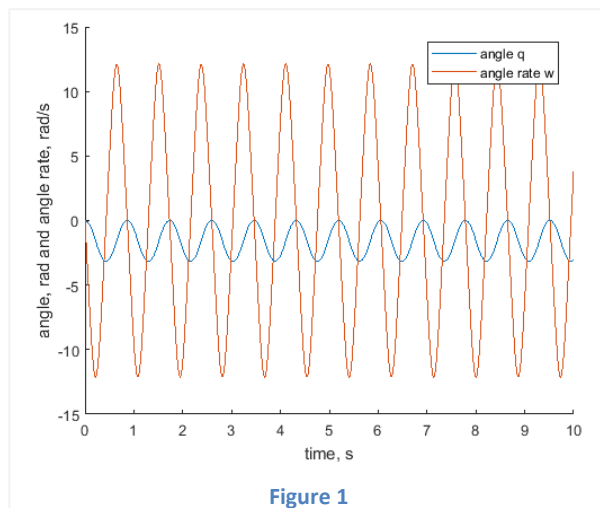
# Exercise I: Introduction to Simscape Multibody

## Problem 1 (Install Matlab)

Make sure you have a working MATLAB installation. As a student at the University of Stuttgart, you can obtain a free license here: <http://www.stud.uni-stuttgart.de/dienste/software/matlab.html>. Make sure that you install both the **Simscape Multibody** and the **Symbolic Math** Toolbox. To check whether you have these, try typing `help mech` and `help symbolic` in the MATLAB Command Window and check the version information that is displayed. This tutorial is based on MATLAB 2018a, Simscape Multibody 5.0, and Symbolic Math Toolbox 7.2. Other versions should be compatible but have not been tested.

## Problem 2 (Get Started with Simscape Multibody)

Carefully work yourself through the basic Matlab Simscape Multibody documentation. To access this documentation, either search for “Getting Started with Simscape Multibody” in the MATLAB help, or type `web(fullfile(docroot, 'physmod/sm/getting-started-with-simmechanics.html'))` in your MATLAB command window. Read the seven articles “About Multibody Modelling”.



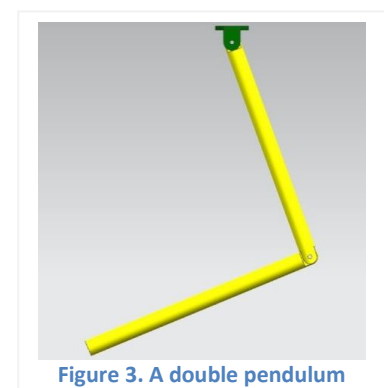
Complete the four tutorials to build and analyze a simple pendulum. When you are ready:

- Make sure that each output of your system matches the output provided in the tutorial.
- Set the torque input and damping in your system to zero and change the model such that it represents a rod made from high-density polyethylene (HDPE). Plot the angle and angle rate; your result should look like in Figure 1. Does the motion change when you change the density of the material?
- Starting from your result in the previous question, simulate the same system on the Moon. You should see the result as shown in Figure 2.

## Problem 3 (Use Simscape Multibody)

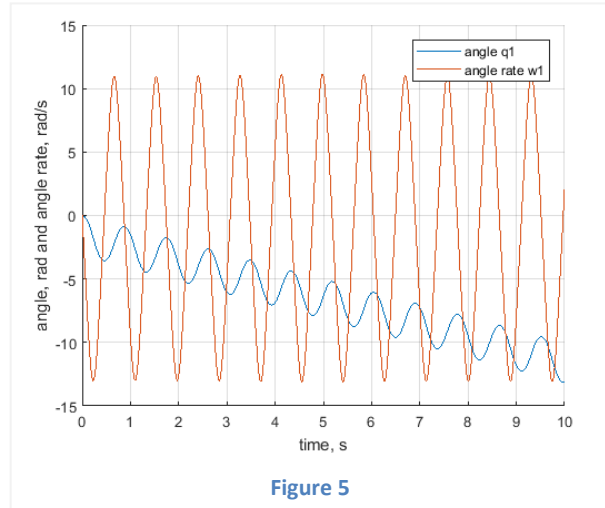
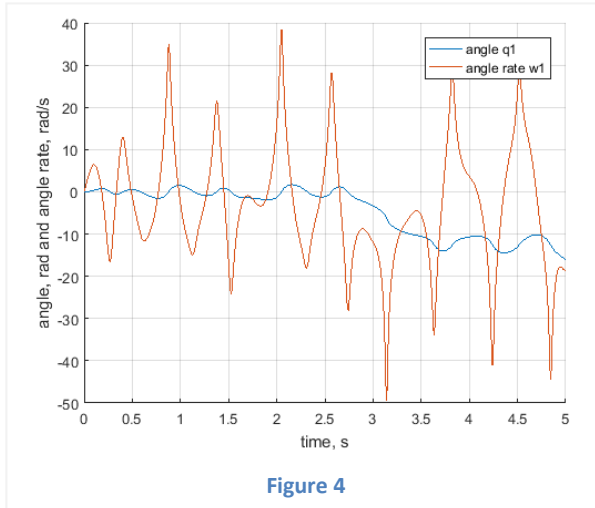
Extend your model of the single pendulum created in **Problem 2b** (no torque input, no damping, HDPE) to represent a double pendulum with two identical links (Figure 3).

- Plot the angle and angular rate for the joint between the links. Your output for the first five seconds should look like in Figure 4.
- For both solids in the pendulum, set the property *Inertia-Type* to 'Custom' and manually compute the inertia properties (*Mass*, *Center of Mass*, *Moments of Inertia*, and *Products of Inertia*) based



on the values of  $L$ ,  $H$ ,  $W$ , and  $\rho$ . If you do this right, the motion of the pendulum should remain identical.

- (c) Add a kinematic motion input to the first joint such that the first link moves with a constant angular rate ( $q = \omega t$ ,  $\omega = 1$ ). To do this, you will need to change the property *Actuation-Motion* of the joint and provide an appropriate input to the joint. Then, run the system and plot the angle and angular rate for the joint between the links. You should get a plot as in Figure 5.

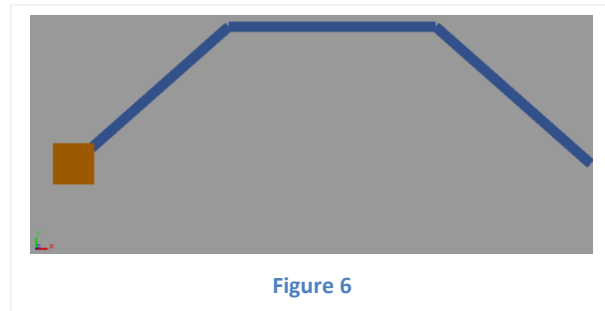


⚠ For the **Problem 3a** and Problem 3b above, we provided only five seconds of the output to check and compare the solutions. Why did we not provide the full 10 seconds of the output?

#### Problem 4 (Four bar linkage in Simscape)

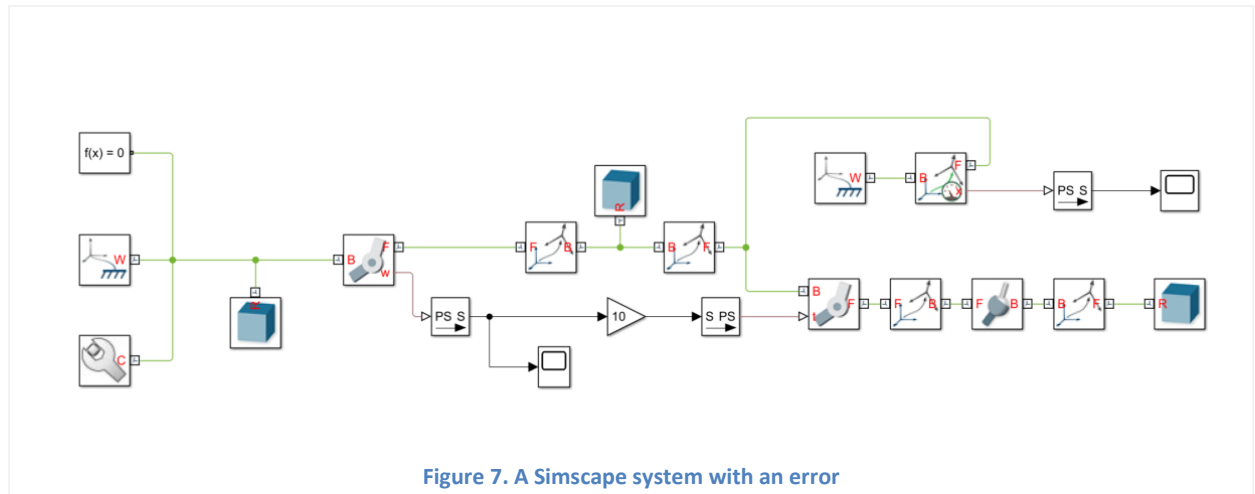
🔧 A four bar linkage is a triple pendulum both ends of which are connected through revolute joints to fixed frames. Extend the double pendulum you created in **Problem 3** to model a four bar linkage. Set the new end point of the linkage to be fixed at position  $[0.5; 0; 0]$ . You can do so, by adding a *Rigid Transform* to the *World Frame* and setting the transformation to *Cartesian*. Connect the new fixed frame with another rotational joint to the frame at the end of your third link.

- Set state targets in all joints to active (high priority) and zero. Run the model. Explain the warning/error you get.
- Disable all state targets but one and find a balanced configuration of the system as shown in Figure 6, in which the four bar linkage will remain at rest (at least in initially).
- If you set the *translation* of the *Rigid Transform* to  $[1; 0; 0]$  and try to run the model you get an error. Why?



#### Problem 5 (Debugging Simscape)

- ☑ Find and describe the error in the Simscape model shown in Figure 7. How would you fix it? Is this an energy conserving system?



## Exercise II: CoSys, Transformations, Derivatives

### Problem 6 (Euler Angles, Tait-Bryan/Cardan convention)

☞ [NOTE: in order for your code to work, you must include the folder with the class files in your MATLAB path as described in the header of *Example\_3\_SingleTransformations.m*). Alternatively, you can copy all class-files and your MATLAB-scripts into the same folder.]

Study the provided example scripts *Example\_3\_SingleTransformations.m* and *Example\_4\_DoubleTransformations.m* which visualize single and double rotation transformations. They use the graphics framework that is the base for all the code we will write in this class. Then, extend *Example\_4* to create your own MATLAB-script that creates the following coordinate systems (Figure 8):

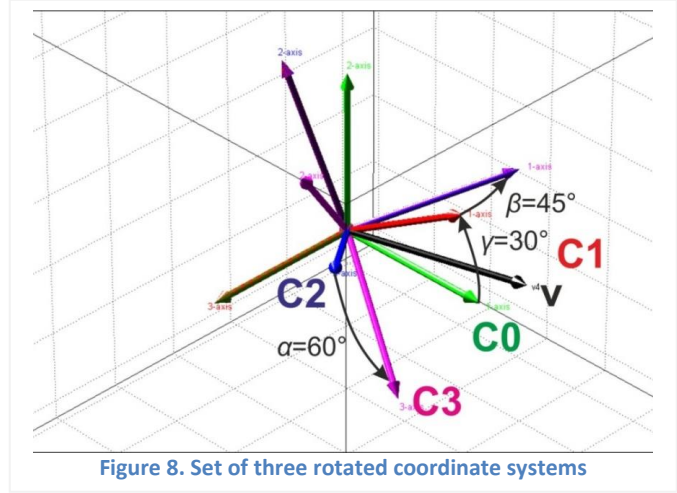


Figure 8. Set of three rotated coordinate systems

- $I$ , which is identical to the coordinate system of the graphical output
- $C_1$ , which is rotated with respect to  $I$  by  $\gamma = 30$  deg about the 3-axis of  $I$
- $C_2$ , which is rotated with respect to  $C_1$  by  $\beta = 45$  deg about the 2-axis of  $C_1$
- $C_3$ , which is rotated with respect to  $C_2$  by  $\alpha = 60$  deg about the 1-axis of  $C_2$

Additionally define a vector  $\mathbf{v}$  that is given in  $C_3$  by the coordinates  ${}_{C_3}\mathbf{v} = (1 \ 1 \ 1)^T$ . (Background: This series of rotations is the main convention used to describe the 3D motion of Airplanes. The three rotations are called 'yaw' (3-axis), 'pitch' (2-axis), and 'roll' (1-axis). In contrast to the way how we depicted the coordinate systems in the lecture notes and also in contrast to the graphical output, the axis convention in aerospace is that the 1-axis is pointing in the direction of flight, the 2-axis is pointing to the right, and the 3-axis is pointing downwards. Matching this with the graphical output could be done easily by adding yet another transformation, but this is not the scope of this problem.)

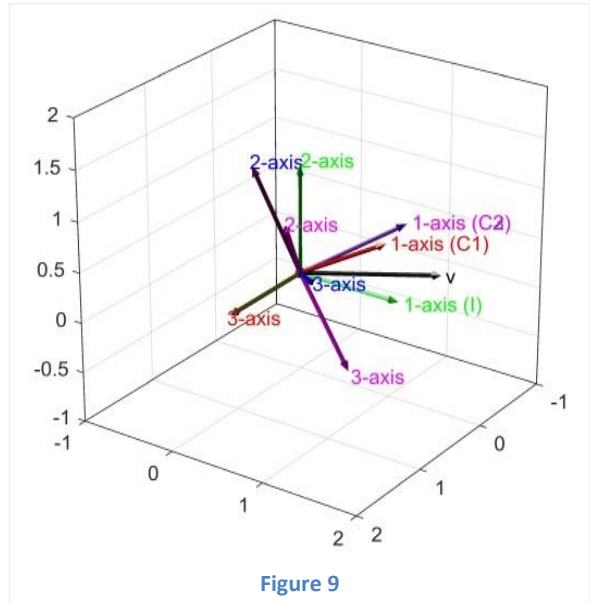


Figure 9

- (a) Make a screenshot of your output; it should look like in Figure 9. The best way of doing this in MATLAB is by executing the following line (or adding it to your script): `print(gcf, '-r600', '-djpeg', 'Problem_6_Output.jpg');`

- (b) Check that your numerical values for:  ${}_I\mathbf{v}$ ,  ${}_{C_1}\mathbf{v}$ ,  ${}_{C_2}\mathbf{v}$ ,  ${}_{C_3}\mathbf{v}$  are:

$$\begin{aligned} {}_I\mathbf{v} &= (1.6319, 0.5195, 0.2588)^T, & {}_{C_1}\mathbf{v} &= (1.6730, -0.3660, 0.2588)^T, \\ {}_{C_2}\mathbf{v} &= (1, -0.3660, 1.3660)^T, & {}_{C_3}\mathbf{v} &= (1, 1, 1)^T. \end{aligned}$$

### Problem 7 (Derivatives of Transformations)

☞ Use the template file *Template\_7\_EulerAngles\_Derivatives.m* to implement the Cardan-convention rotations from **Problem 6** in analytical form for time varying coordinate systems. The code should compute the following transformations analytically:

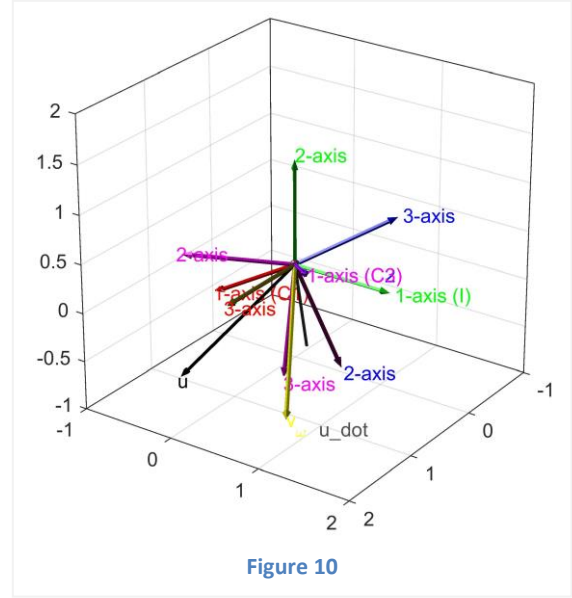
- $\mathbf{A}_{IC_1}$  (between  $C_1$  and the inertial frame  $I$ )
- $\mathbf{A}_{IC_2}$  (between  $C_2$  and the inertial frame  $I$ )
- $\mathbf{A}_{IC_3}$  (between  $C_3$  and the inertial frame  $I$ )
- $\mathbf{A}_{C_3I}$  (between the inertial frame  $I$  and  $C_3$ )

It should also generate the skew symmetric matrix  ${}_{C_3}\tilde{\omega}_{IC_3}$  and extract from it the vector of angular velocities  ${}_{C_3}\boldsymbol{\omega}_{IC_3}$ .

- (a) Complete the template to learn how to use the symbolic math toolbox. Your analytical result for the vector of angular velocities  ${}_{C_3}\boldsymbol{\omega}_{IC_3}$  should be:

$${}_{C_3}\boldsymbol{\omega}_{IC_3} = \begin{pmatrix} \dot{\alpha} - \dot{\gamma} \sin \beta \\ \dot{\beta} \cos \alpha + \dot{\gamma} \cos \beta \sin \alpha \\ \dot{\gamma} \cos \alpha \cos \beta - \dot{\beta} \sin \alpha \end{pmatrix}$$

The second part of the template uses the `matlabFunction` command and the analytical expressions for  $\mathbf{A}_{IC_1}$ ,  $\mathbf{A}_{IC_2}$ ,  $\mathbf{A}_{IC_3}$ ,  ${}_{C_3}\tilde{\omega}_{IC_3}$ ,  ${}_{C_3}\boldsymbol{\omega}_{IC_3}$  to automatically create Matlab functions ('.m' files) that compute these quantities. Then, the script uses the created functions to generate the graphical output of **Problem 6** by setting  $\alpha = 60^\circ$ ,  $\beta = 45^\circ$ , and  $\gamma = 30^\circ$ . Starting from this configuration, the script also generates an animation over  $n=100$  time steps, where  $\Delta t = \pi/n$  (i.e.,  $t=0..\pi$ ) and  $\dot{\alpha} = \dot{\beta} = \dot{\gamma} = 1 \text{ rad/s}$ .



- (b) A screenshot of the last frame of the animation should look like in Figure 10.

### Problem 8 (Integrating Angular Velocities)

In **Problem 7**, the animation (or 'simulation', 'integration') of rotations was based on three independent rotations around the principal axis of subsequent coordinate systems (or simply: 'Cardan angles'). As you saw, the resulting motion is all but uniform and the global rotation axis is continuously changing. In the following, we thus seek to examine two different ways to create a uniform rotation about a given axis.

To this end, complete the first part of the provided file `Template_8a_IntegratingAngularVelocities.m` to do the following:

- Define two coordinate systems  $I$  and  $C(t)$  that are rotated about the 3-axis by 90deg at time  $t=t_0$

$$\text{(i.e., } \mathbf{A}_{IC}(t_0) = \begin{bmatrix} +\cos(\frac{\pi}{2}) & -\sin(\frac{\pi}{2}) & 0 \\ +\sin(\frac{\pi}{2}) & +\cos(\frac{\pi}{2}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{)}$$

- Create a graphical environment that shows these two coordinate systems.
- Define a vector of angular velocity between the two frames which is constant and defined in **inertial** coordinates as  ${}_I\boldsymbol{\omega}_{IC} = \left( \sqrt{\frac{1}{3}} \quad \sqrt{\frac{1}{3}} \quad \sqrt{\frac{1}{3}} \right)^T$ .
- Show this vector in the graphical environment

(a) To implement the **first approach**, use appropriate transformations and the definition of the angular velocity  ${}_C\tilde{\omega}_{IC} = \mathbf{A}_{CI} \cdot \dot{\mathbf{A}}_{IC}$  to derive an expression that computes  $\dot{\mathbf{A}}_{IC}(t)$  from  $\mathbf{A}_{IC}(t)$  and  ${}_I\omega_{IC}$  in the appropriate coordinates. Make use of the second part of *Template\_8a\_IntegratingAngularVelocities.m*. Your script should create an animation which uses the expression for  $\dot{\mathbf{A}}_{IC}(t)$  in an Euler forward integration scheme ( $\mathbf{A}_{IC}(t + \Delta_t) = \mathbf{A}_{IC}(t) + \Delta_t \cdot \dot{\mathbf{A}}_{IC}(t)$ ) with:

- $\mathbf{A}_{IC}(t_o) = \begin{bmatrix} +\cos(\frac{\pi}{2}) & -\sin(\frac{\pi}{2}) & 0 \\ +\sin(\frac{\pi}{2}) & +\cos(\frac{\pi}{2}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$

- $n = 100$

- $\Delta_t = 2\pi/n$

☒ The last frame of your animation should look like in Figure 11.

(b) In the **second approach**, we use the Cardan angles from **Problem 6** and **Problem 7**. To this end, we need to establish a relationship between the rates of the Cardan angles  $\dot{\alpha}, \dot{\beta}, \dot{\gamma}$  and the vector of the angular velocity  ${}_C\omega_{IC}$ . The easiest way of doing this, is by rewriting the angular velocity as a linear equation  ${}_C\omega_{IC} = \mathbf{B} \cdot (\dot{\alpha} \ \dot{\beta} \ \dot{\gamma})^T$  where the matrix  $\mathbf{B}$  can be extracted manually from your answer to **Problem 7**. From this we can obtain:

$$\begin{pmatrix} \dot{\alpha} \\ \dot{\beta} \\ \dot{\gamma} \end{pmatrix} = (\mathbf{A}_{IC} \cdot \mathbf{B})^{-1} \cdot {}_I\omega_{IC}$$

So, instead of integrating  $\mathbf{A}_{IC}$  directly, we integrate the Cardan angles  $\alpha, \beta, \gamma$  (as in **Problem 7**) and then compute the new transformation  $\mathbf{A}_{IC}$  from these values.

The first part of the template file *Template\_8b\_IntegratingAngularVelocities.m* is a solution to **Problem 7** and derives the expression for  ${}_C\omega_{IC}$ . Use this expression to extract the matrix  $\mathbf{B}$  and complete the provided template. Your script should create an animation that uses this integration approach with:

- $\alpha(t_o) = \beta(t_o) = 0; \ \gamma(t_o) = \frac{\pi}{2}$

- $n = 100$

- $\Delta_t = 2\pi/n$

☒ The last frame of your animation should look like in Figure 12.

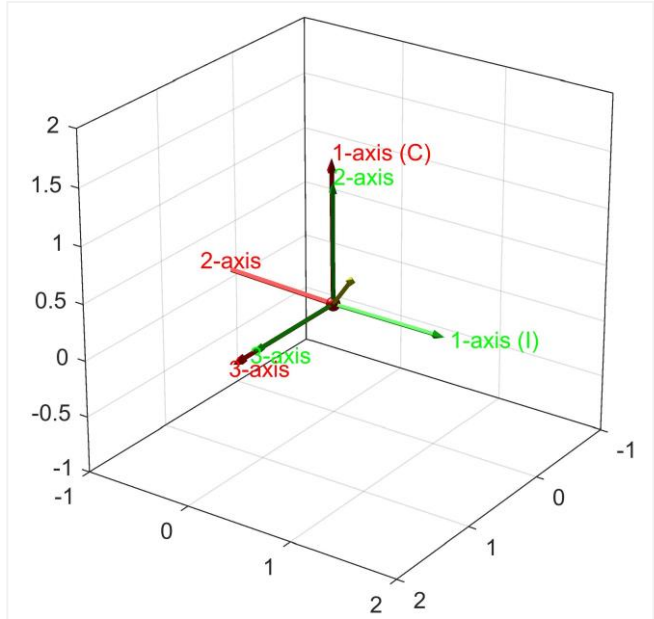


Figure 11

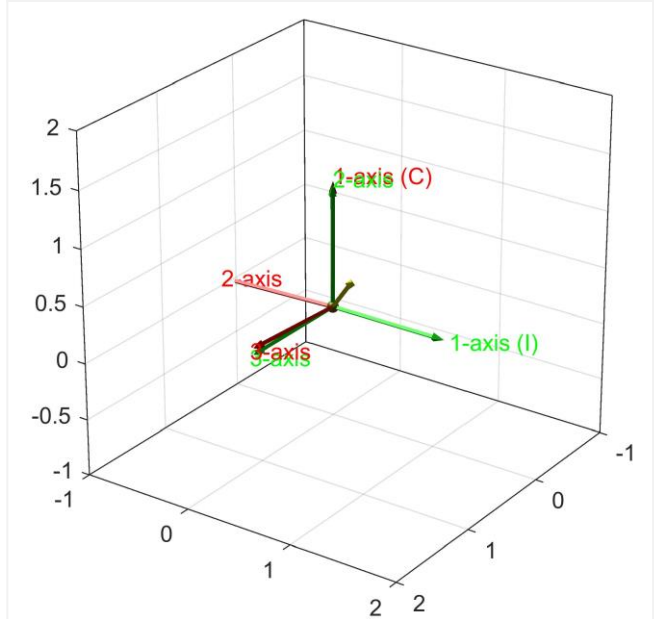


Figure 12

(c) When looking at both methods, we can see that they do not really create a full 360 deg. rotation in which the two coordinate frames are aligned again at the end of the animation. Numerical errors are introduced by using the simple Euler-forward integration method. Describe qualitatively the difference of this error-accumulation between the two methods and state which method seems better suited to describe angular motions.

💡 How can you create an absolutely accurate rotation by changing only **one** line in the code from part (a)?

### Problem 9 (BetterCoSysClass)

💻 Write a better version of the class `CoSysCLASS`, which allows using the constructor with three arguments instead of two: `C = BetterCoSysCLASS(env, A_BC, B)`. In this case, the third argument is another, already existing coordinate system. The transformation `A_BC` is then not interpreted as being from the inertial frame to `C`, but between the two frames `B` and `C`. Also write the public functions `C.setTransform(A_BC, B)` and `A_BC = C.getTransform(B)` which allow setting such a relative transformation at a later point and getting the transformation matrix `A_BC` with respect to another, arbitrary coordinate system. You can use the template file `Template_9_betterCoSysCLASS.m` or adapt `CoSysCLASS.m`.

☑ Use the file `TestFile_9_BetterCoSysCLASS.m` to use your class `BetterCoSysCLASS.m`. The output of the script should look the same as in Figure 9.

### Problem 10 (Understanding Rotation Matrices)

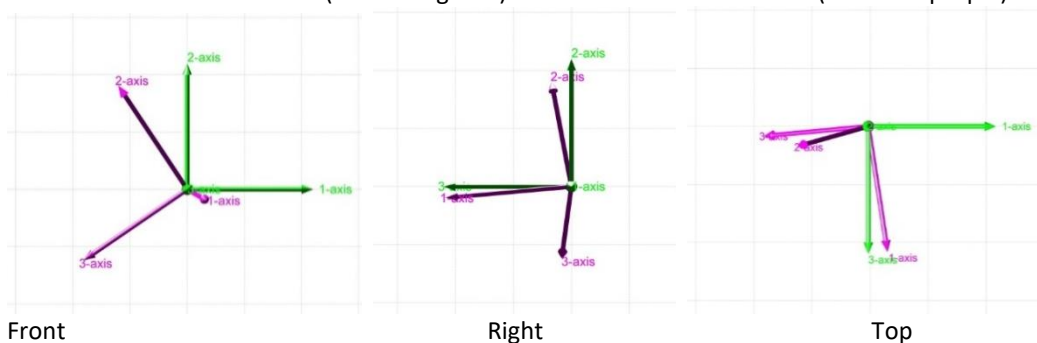
(a) Which one of the following matrices **DOES NOT** represent an orthonormal transformation? Why?

$$\begin{aligned} (1) \mathbf{A}_{IC} &= \begin{bmatrix} +0.354 & -0.573 & +0.739 \\ +0.612 & +0.739 & +0.280 \\ -0.707 & +0.354 & +0.612 \end{bmatrix} & (2) \mathbf{A}_{IC} &= \begin{bmatrix} +0.500 & -0.750 & +0.433 \\ +0.866 & +0.433 & -0.250 \\ 0 & +0.500 & +0.866 \end{bmatrix} \\ (3) \mathbf{A}_{IC} &= \begin{bmatrix} +0.354 & +0.612 & -0.707 \\ -0.573 & +0.739 & +0.354 \\ +0.739 & +0.280 & +0.612 \end{bmatrix} & (4) \mathbf{A}_{IC} &= \begin{bmatrix} +0.354 & +0.612 & -0.707 \\ -0.573 & +0.739 & +1.354 \\ +0.739 & +0.280 & +0.612 \end{bmatrix} \end{aligned}$$

(b) Which one of the following matrices **DOES** represent a rotation? Why?

$$\begin{aligned} (1) \mathbf{A}_{IC} &= \begin{bmatrix} +0.866 & -0.500 & +0.433 \\ -0.500 & +0.433 & +0.750 \\ +0.433 & +0.750 & +0.500 \end{bmatrix} & (2) \mathbf{A}_{IC} &= \begin{bmatrix} +0.866 & -0.250 & +0.433 \\ +0.500 & +0.433 & -0.750 \\ 0 & +0.866 & +0.500 \end{bmatrix} \\ (3) \mathbf{A}_{IC} &= \begin{bmatrix} +0.866 & +0.250 & +0.433 \\ +0.500 & -0.433 & -0.750 \\ 0 & -0.866 & +0.500 \end{bmatrix} & (4) \mathbf{A}_{IC} &= \begin{bmatrix} +0.866 & +0.500 & +0.433 \\ +0.500 & -0.433 & -0.750 \\ +0.433 & -0.750 & +0.500 \end{bmatrix} \end{aligned}$$


(c) Write down the approximate transformation matrix for  $\mathbf{A}_{IC}$  that represents the rotation between the inertial frame 'I' (shown in green) and the reference frame 'C' (shown in purple).





## Exercise III: Rigid Body Motion

### Problem 11 (Bound Vectors and Coordinate Systems)

 Familiarize yourself with the newly provided classes *BoundVectorCLASS.m* and *BoundCoSysCLASS.m* (you can find them in the material for Exercise III). They represent vectors and coordinate systems that are not drawn at the origin of the graphical coordinate system but with some offset. NOTE: this does *not* change the values of the vectors or the coordinate systems, which physically do not have an 'origin'.

The offset is given for the bound coordinate system in components of the inertial reference frame, and for the bound vectors in the same components as the vectors themselves. The classes are only provided as templates, in which some of the 'set'-functions are not yet implemented.

- ☒ Complete the class files to learn how to use set functions in Matlab and to do the necessary transformations. To test them, you can run the file *TestFile\_11\_BoundVectorAndCoSys.m*. It should get you the graph in Figure 13.

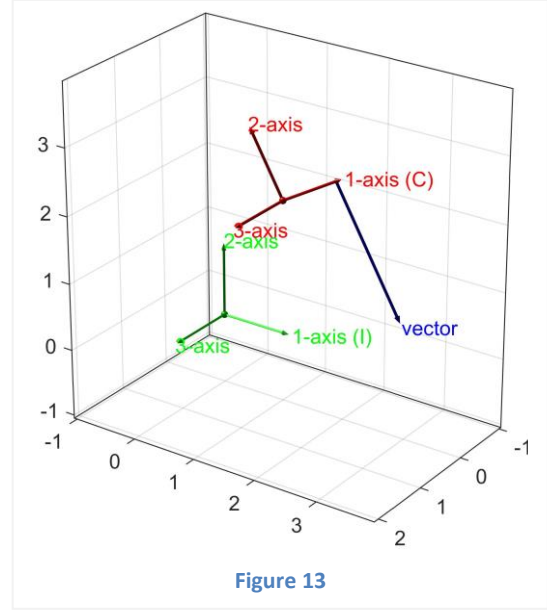



Figure 13

### Problem 12 (Rigid Body Kinematics)

 The class *RigidBodyKinematicsCLASS.m*, (again only provided as a template), defines a rigid body with all its kinematic values and dynamic properties. (At the moment we're only interested in the kinematic values.) Familiarize yourself with the class, which is basically just a collection of the following values and corresponding objects for their graphical visualization:

$$\begin{array}{ll} {}_B\mathbf{r}_{IB} & {}_B\mathbf{A}_{IB} \\ {}_B\mathbf{v}_B & {}_B\vec{\Omega}_B \\ {}_B\mathbf{a}_B & {}_B\dot{\vec{\Omega}}_B \end{array}$$

In the materials is also a file *TestFile\_12\_RigidBodyKinematics.m* that can be used to test your class. If you run the first part of this file, it will create the output shown in Figure 14.

The coordinate system  $B$  (given by  ${}_B\mathbf{A}_{IB}$ ) is shown in black, the relative position vector  ${}_B\mathbf{r}_{IB}$  in grey. The linear velocity  ${}_B\mathbf{v}_B$  is shown in red and the linear acceleration  ${}_B\mathbf{a}_B$  in blue. The angular velocity  ${}_B\vec{\Omega}_B$  is yellow, and the angular acceleration  ${}_B\dot{\vec{\Omega}}_B$  is magenta.

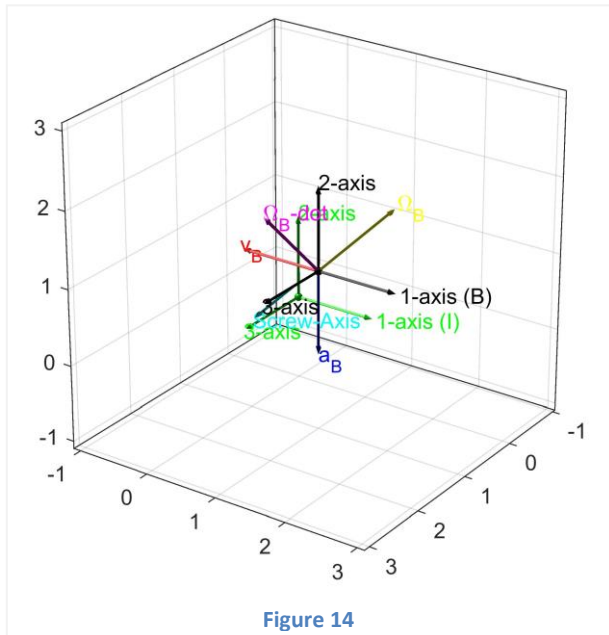


Figure 14

(a) The class *RigidBodyKinematicsCLASS.m* should provide a function `B.integrationStep(delta_t)` which performs an internal integration step of size 'delta\_t'. This means, based on the currently set values for velocities and accelerations, all positions and velocities are updated via Euler-forward integration, while the rotation matrix is updated via a matrix exponential. I.e., multiple calls to `integrationStep` will compute how the body is moving over time.



Complete this function and test it with accelerations that are **constant in body-fixed coordinates**. Simulate over 100 steps with a length of 0.01 seconds. This means, you set all parameters of B to their desired values and call `integrationStep` 100 times. The code in the second part of `TestFile_12_RigidBodyKinematics.m` will do this for you once you enter the correct initial conditions. Run it for the following cases and check  ${}_B\mathbf{A}_{IB}(t)$  as well as  ${}_B\mathbf{r}_{IB}(t)$  at the end of each run. In all cases,  ${}_B\mathbf{A}_{IB}(t_o)$  is initially the identity matrix, and  ${}_B\mathbf{r}_{IB}(t_o)$  is initially a zero-vector.

1)  ${}_B\mathbf{v}_B(t_o) = (0 \ 0 \ 0)^T$ ,  ${}_B\ddot{\mathbf{Q}}_B(t_o) = (0 \ 1.5 \ 0)^T$ ,  ${}_B\mathbf{a}_B(t) \equiv (0 \ 0 \ 0)^T$ ,  ${}_B\dot{\ddot{\mathbf{Q}}}_B(t) \equiv (0 \ 0 \ 0)^T$

☑ Check your result:

$${}_B\mathbf{A}_{IB} = \begin{bmatrix} 0.0707 & 0 & 0.9975 \\ 0 & 1 & 0 \\ -0.9975 & 0 & 0.0707 \end{bmatrix}, {}_B\mathbf{r}_{IB} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

2)  ${}_B\mathbf{v}_B(t_o) = (1.5 \ 0 \ 0)^T$ ,  ${}_B\ddot{\mathbf{Q}}_B(t_o) = (0 \ 1.5 \ 0)^T$ ,  ${}_B\mathbf{a}_B(t) \equiv (0 \ 0 \ 0)^T$ ,  ${}_B\dot{\ddot{\mathbf{Q}}}_B(t) \equiv (0 \ 0 \ 0)^T$

☑ Check your result:

$${}_B\mathbf{A}_{IB} = \begin{bmatrix} 0.0707 & 0 & 0.9975 \\ 0 & 1 & 0 \\ -0.9975 & 0 & 0.0707 \end{bmatrix}, {}_B\mathbf{r}_{IB} = \begin{pmatrix} 0.1301 \\ 0 \\ 1.5112 \end{pmatrix}$$

3)  ${}_B\mathbf{v}_B(t_o) = (0 \ 0 \ 0)^T$ ,  ${}_B\ddot{\mathbf{Q}}_B(t_o) = (0 \ 0 \ 0)^T$ ,  ${}_B\mathbf{a}_B(t) \equiv (1.5 \ 0 \ 0)^T$ ,  ${}_B\dot{\ddot{\mathbf{Q}}}_B(t) \equiv (0 \ 1.5 \ 0)^T$

☑ Check your result:

$${}_B\mathbf{A}_{IB} = \begin{bmatrix} 0.7368 & 0 & 0.6761 \\ 0 & 1 & 0 \\ -0.6761 & 0 & 0.7368 \end{bmatrix}, {}_B\mathbf{r}_{IB} = \begin{pmatrix} 0.6041 \\ 0 \\ 0.4219 \end{pmatrix}$$

(b) Implement the functions

- `I_r_IQ = positionOfPoint(obj, B_r_BQ)`
- `I_v_Q = velocityOfPoint(obj, B_r_BQ)`
- `I_a_Q = accelerationOfPoint(obj, B_r_BQ)`

which return the position, velocity, and acceleration of a body-fixed point Q. Run the third part of the script `TestFile_12_RigidBodyKinematics.m` which runs the simulation with

$${}_B\mathbf{v}_B(t_o) = (0 \ 0 \ 0)^T,$$

$${}_B\ddot{\mathbf{Q}}_B(t_o) = (0 \ 0 \ 4)^T,$$

${}_B\mathbf{a}_B(t) \equiv (0 \ 3 \ 0)^T$ ,  ${}_B\dot{\ddot{\mathbf{Q}}}_B(t) \equiv (0 \ 0 \ 0)^T$  and the point Q located at  $(0 \ 3/16 \ 0)^T$ . Your output should look like in Figure 15.

- ☑ Give a physical example of something that behaves like the body above. What are the points B and Q, respectively?

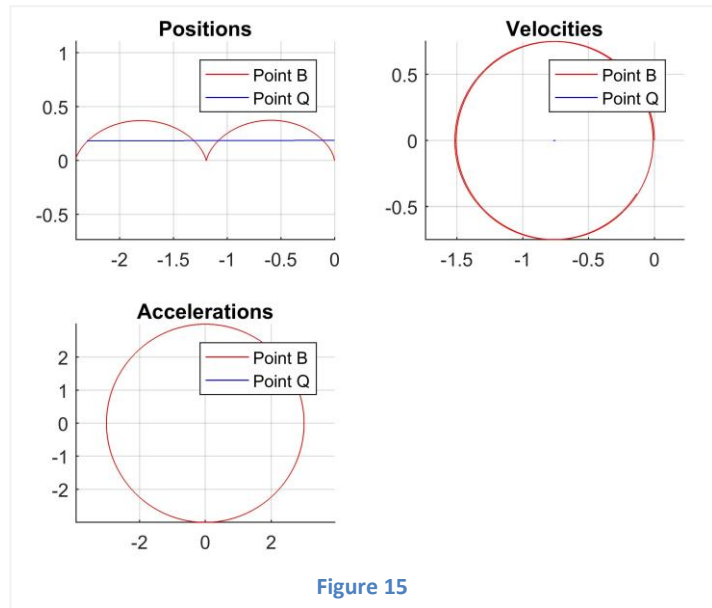


Figure 15

### Problem 13 (Twists and Screw Axis)

✍ Here, we try to represent an arbitrary rigid body motion as a twist; i.e., as a motion along an axis while the body is simultaneously performing a rotation about this axis.

With the velocity of an arbitrary point on the body being defined as  ${}_B\mathbf{v}_Q = {}_B\mathbf{v}_B + {}_B\tilde{\boldsymbol{\Omega}}_B \cdot {}_B\mathbf{r}_{BQ}$  establish a way to find a point  $T$  on the body which is only moving along the axis of  $\tilde{\boldsymbol{\Omega}}_B$ . This point will not be unique. How did you select yours? Why?

### Problem 14 (Visualizing Twists)

📁 Using your result from **Problem 13**, extend the class *RigidBodyKinematicsCLASS.m* such that it also shows the screw axis (and the velocity along it) as a bound vector  ${}_B\mathbf{v}_T$  that is attached at the point  $T$ . Run a simulation of your rigid body starting from the following values (The script *TestFile\_14\_VisualizingTwists.m* will do this for you):

- ${}_B\mathbf{r}_{IB} = (0 \ 0 \ 0)^T$
- $A_{IB} = \text{diag}(1 \ 1 \ 1)$
- ${}_B\mathbf{v}_B = (0 \ 0 \ 0)^T$
- ${}_B\tilde{\boldsymbol{\Omega}}_B = (0 \ 1.5 \ 1.5)^T$
- ${}_B\mathbf{a}_B = (1.5 \ 1.5 \ 0)^T$
- ${}_B\dot{\tilde{\boldsymbol{\Omega}}}_B = (0 \ 0 \ 0)^T$

Your last frame should look like Figure 16.

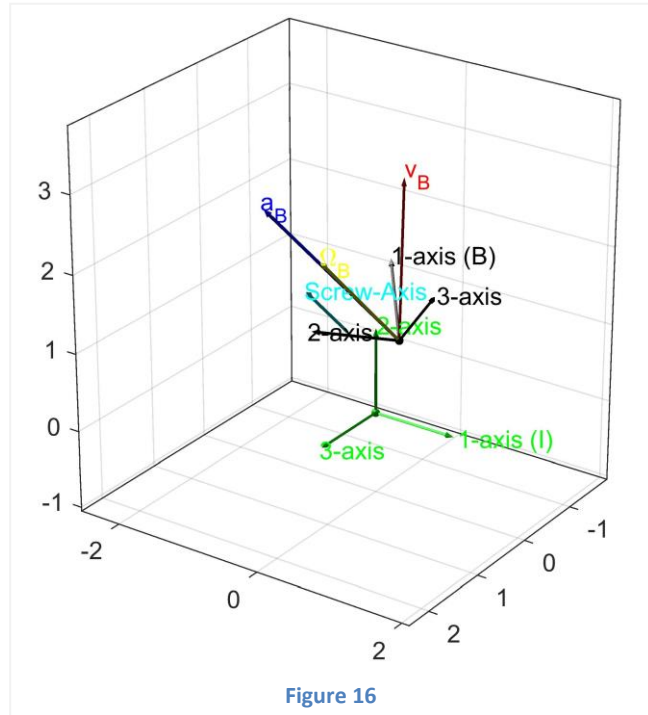


Figure 16

### Problem 15 (Rigid Body Dynamics)

📁 Here, we examine the motion of a single rigid body that is not subject to any external forces or moments. Based on the code from *RigidBodyKinematicsCLASS.m* that you developed in **Problem 12**, we want to create a dynamic simulation that assumes no external forces or moments are acting on the body (i.e., the body is floating in zero-gravity). The code that integrates accelerations is already in place, so all we have to do is to write a new function `computeNaturalDynamics()` that computes the accelerations. This function does not take any input arguments (since the external forces and moments are assumed to be zero) nor does it create any output (since the resulting accelerations  ${}_B\mathbf{a}_B$  and  ${}_B\dot{\tilde{\boldsymbol{\Omega}}}_B$  are part of the class and stored there). It thus should look like this:

```
function computeNaturalDynamics(obj)
    % ADD YOUR CODE HERE

    % Compute accelerations from the equations of motion of a rigid
    % body:
    obj.B_a_B =
    obj.B_omegaDot_B =
end
```

You can use the provided template *Template\_15\_RigidBodyDynamicsCLASS.m* (which is essentially the solution of **Problem 12** and **Problem 14**).

**(a)** Complete the class. To test your classes, you can use the provided file *TestFile\_15\_RigidBodyNaturalDynamics.m*. Run it for the following cases and check out your output for  ${}_B\mathbf{A}_{IB}$  and  ${}_B\mathbf{r}_{IB}$  at the end of each run:

- 1)  ${}_B\mathbf{v}_B = (1 \ 1 \ 1)^T$ ,  ${}_B\vec{\Omega}_B = (0 \ 0 \ 0)^T$
- 2)  ${}_B\mathbf{v}_B = (0 \ 0 \ 0)^T$ ,  ${}_B\vec{\Omega}_B = (1 \ 0 \ 0)^T$
- 3)  ${}_B\mathbf{v}_B = (0 \ 0 \ 0)^T$ ,  ${}_B\vec{\Omega}_B = (1 \ 1 \ 1)^T$
- 4)  ${}_B\mathbf{v}_B = (0 \ 0 \ 0)^T$ ,  ${}_B\vec{\Omega}_B = (1 \ 0.01 \ 0.01)^T$
- 5)  ${}_B\mathbf{v}_B = (0 \ 0 \ 0)^T$ ,  ${}_B\vec{\Omega}_B = (0.01 \ 1 \ 0.01)^T$
- 6)  ${}_B\mathbf{v}_B = (0 \ 0 \ 0)^T$ ,  ${}_B\vec{\Omega}_B = (0.01 \ 0.01 \ 1)^T$

In all cases,  ${}_B\mathbf{A}_{IB}$  is initially the identity matrix, and  ${}_B\mathbf{r}_{IB}$  is a zero-vector. The mass  $m$  is 1 Kg, and the inertia matrix is diagonal and given as

$${}_B\mathbf{I}_B = \begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2.5 \end{bmatrix}.$$

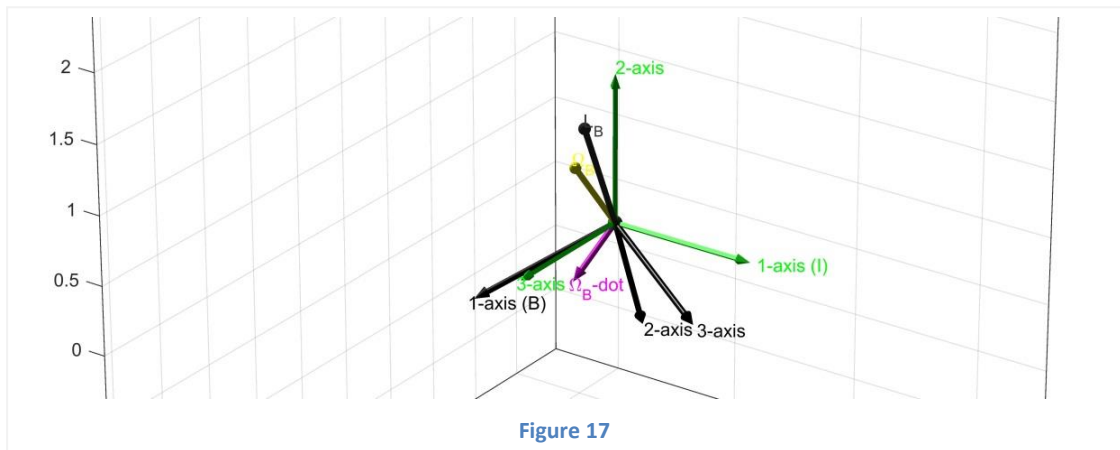
The step-size is chosen to be 1ms, and the integration time 30 s. The provided code has all these quantities already implemented. Your final output for the case (6) should be

$${}_B\mathbf{A}_{IB} = \begin{bmatrix} 0.1564 & 0.9877 & 0.0004 \\ -0.9876 & 0.1564 & 0.0145 \\ 0.0142 & -0.0026 & 0.9999 \end{bmatrix}, \quad {}_B\mathbf{r}_{IB} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

- ☒ Qualitatively, describe the difference of what you observe for the cases (4), (5), and (6).

**(b)** Extend the class 'RigidBodyDynamicsCLASS' such that it also shows the angular momentum about the center of gravity as a bound vector  ${}_B\vec{\mathbf{L}}_B$ .

- ☒ Make a screenshot from the end of your run of case (3) that shows  ${}_B\vec{\mathbf{L}}_B$ ; it should look like this:



- ☒ Since the angular momentum doesn't change if no external forces or torques are applied, this can provide a good way of debugging your code from part (a).

### Problem 16 (Computing Inertia Matrices)

✎ (a) Derive the inertia matrix  ${}_B\mathbf{I}_G$  for the following rigid objects:

- 1) A rectangular cuboid with homogenous density and mass  $m$ . It is defined in body fixed coordinates as well as centered and aligned with the axes of this coordinate system. Its length along the 1-axis is given as  $a$ , along the 2-axis as  $b$ , and along the 3-axis as  $c$ . See Figure 18.
- 2) An ellipsoid with homogenous density and mass  $m$ . It is defined in body fixed coordinates as well as centered and aligned with the axes of this coordinate system. The lengths of the principle axes are  $a$  along the 1-axis,  $b$  along the 2-axis, and  $c$  along the 3-axis. See Figure 19.
- 3) 6 particles of mass  $\frac{m}{6}$ , each of them situated in the center of a face of the cuboid of the first question. They are shown as red dots in Figure 18 and their coordinates are given as:

$$r_{1,2} = \begin{bmatrix} \pm \frac{a}{2} & 0 & 0 \end{bmatrix}$$

$$r_{3,4} = \begin{bmatrix} 0 & \pm \frac{b}{2} & 0 \end{bmatrix}$$

$$r_{5,6} = \begin{bmatrix} 0 & 0 & \pm \frac{c}{2} \end{bmatrix}$$

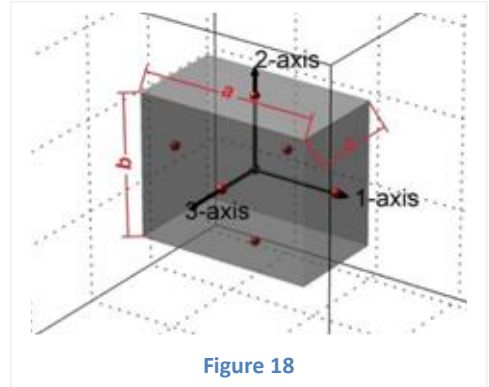


Figure 18

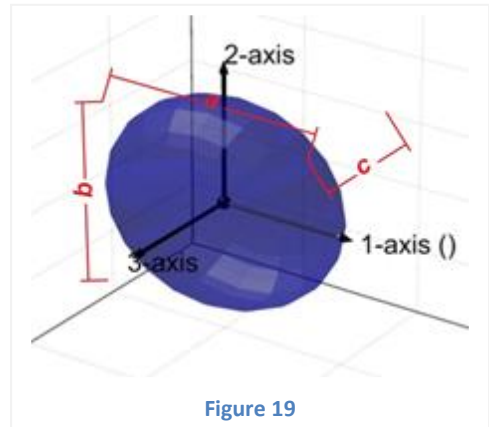


Figure 19

✎ (b) Now we want to do the reverse of part (a). Suppose that the following diagonal matrix is the inertia matrix of one of the rigid objects considered above:

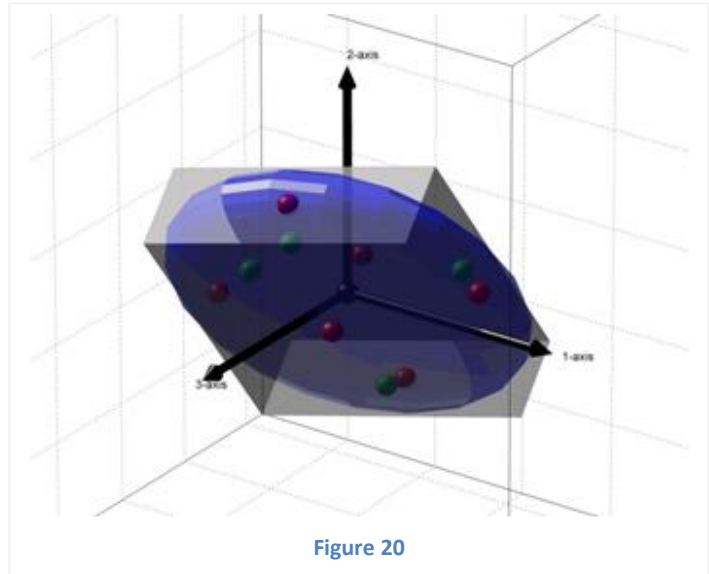
$${}_B\mathbf{I}_G = \begin{bmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{bmatrix}$$

For each of the three objects, find corresponding expressions for the parameters  $a$ ,  $b$ , and  $c$  in terms of  $I_1$ ,  $I_2$ , and  $I_3$ .

### Problem 17 (Visualize Inertia Matrices)

📖 We want to use the results of **Problem 16** to visualize arbitrary inertia matrices. To do so, finalize the open 'switch'-cases in the template *Template\_17\_ShowInertia.m* by replacing the currently set values for all points (currently set to -1, 0, and +1 as values) with appropriate expressions. Start with the switch cases 1, 3, and 4 which are easier; the case 2 is more challenging and is provided as an extra question below. For each of the switch case, use the provided file *TestFile\_17\_ShowInertia.m* to test the following four inertia matrices:

$$\begin{aligned}
 1) \quad {}_B\mathbf{I}_G &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
 2) \quad {}_B\mathbf{I}_G &= \begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2.5 \end{bmatrix} \\
 3) \quad {}_B\mathbf{I}_G &= \begin{bmatrix} +1.56 & -0.16 & +0.23 \\ -0.16 & +1.95 & -0.50 \\ +0.23 & -0.50 & +2.5 \end{bmatrix} \\
 4) \quad {}_B\mathbf{I}_G &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}
 \end{aligned}$$



The mass  $m$  is  $10 \text{ Kg}$  in all cases and the

physical units of inertia are  $\text{Kg} \cdot \text{m}^2$ . Figure 20 shows the output you should get (possibly with different color shades) for the third (non-diagonal) inertia matrix.

- 🔧 Complete the switch-case 2 in *Template\_17\_ShowInertia.m* by replacing the currently set values for all points with appropriate expressions. Then, running *TestFile\_17\_ShowInertia.m* for this case and the third (non-diagonal) inertia matrix above should produce the four green points in Figure 20.
- 🔧 Extend *RigidBodyDynamicsCLASS.m* such that it visualizes the mass and inertia properties of the body it represents. You might want to use the code from *Template\_17\_ShowInertia.m* or write your own routines.

### Problem 18 (Acceleration of a Point on a Rigid Body)

✍ The angular velocity of a rigid body that is moving in a plane is given by  ${}_B\boldsymbol{\Omega}_B = \omega \cdot [0 \ 0 \ 1]^T$  and its angular acceleration by  ${}_B\dot{\boldsymbol{\Omega}}_B = \dot{\omega} \cdot [0 \ 0 \ 1]^T$ . I.e., the body is rotating about the 3-axis with rate  $\omega$  and acceleration  $\dot{\omega}$ . Point  $B$  is located on the body and has an acceleration of  ${}_B\mathbf{a}_B = [a_x \ a_y \ 0]^T$ . In addition to  $B$ , a second point  $Q$  is located on the body with the relative position  ${}_B\mathbf{r}_{BQ} = [r_x \ r_y \ 0]^T$ . All values are given in body-fixed coordinates.

Compute possible values of  $\omega$  and  $\dot{\omega}$  such that the point  $Q$  is not accelerating (i.e.,  ${}_B\mathbf{a}_Q = \mathbf{0}$ )

### Problem 19 (Calculating Inertia Matrices)

✎ What is the inertia about the 3-axis of the homogeneous object with mass  $m$  shown in Figure 21? Give your answer as a multiple of  $1/120$ . The dashed lines are spaced 1 meter apart.

NOTE: A solid cuboid of height  $h$ , width  $w$ , and depth  $d$ , and mass  $m$  that is rotating around its center of gravity has an inertia of:

$$I_h = \frac{1}{12} m (w^2 + d^2)$$

$$I_w = \frac{1}{12} m (h^2 + d^2)$$

$$I_d = \frac{1}{12} m (h^2 + w^2)$$

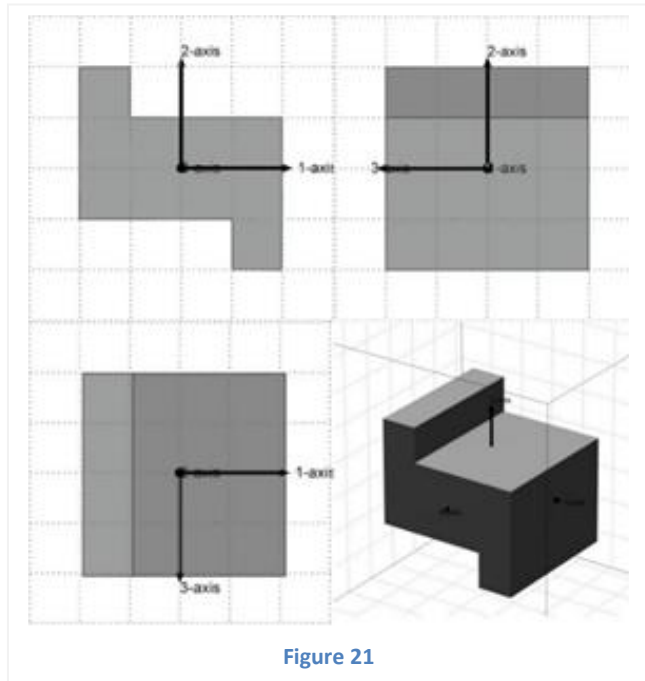


Figure 21

### Problem 20 (Understanding Inertia Matrices)

(a) Which one of the following matrices DOES NOT represent an inertia matrix? Why?

$$(1) {}_B \mathbf{I}_G = \begin{bmatrix} +2.359 & -0.532 & +0.478 \\ -0.532 & +2.266 & -0.077 \\ +0.478 & -0.077 & +1.875 \end{bmatrix} \text{Kg} \cdot \text{m}^2 \quad (2) {}_B \mathbf{I}_G = \begin{bmatrix} +2.186 & +0.650 & +0.108 \\ +0.108 & +2.063 & +0.375 \\ +0.650 & +0.375 & +2.250 \end{bmatrix} \text{Kg} \cdot \text{m}^2$$

$$(3) {}_B \mathbf{I}_G = \begin{bmatrix} +3.313 & 0.758 & -0.650 \\ 0.758 & +2.438 & -0.375 \\ -0.650 & -0.375 & +1.750 \end{bmatrix} \text{Kg} \cdot \text{m}^2 \quad (4) {}_B \mathbf{I}_G = \begin{bmatrix} +2.484 & +0.118 & -0.821 \\ +0.118 & +2.641 & -0.828 \\ -0.821 & -0.828 & +1.375 \end{bmatrix} \text{Kg} \cdot \text{m}^2$$

(b) Which one of the following statements is NOT necessarily correct for the object (solid, homogeneous density) shown in Figure 22?

- (1)  $c > a$
- (2)  $e < a$
- (3)  $f = 0 \text{ Kg} \cdot \text{m}^2$
- (4)  $d = 0 \text{ Kg} \cdot \text{m}^2$

The inertia of the object is given by a matrix of the form:

$${}_B \mathbf{I}_G = \begin{bmatrix} a & d & e \\ d & b & f \\ e & f & c \end{bmatrix} \text{Kg} \cdot \text{m}^2$$

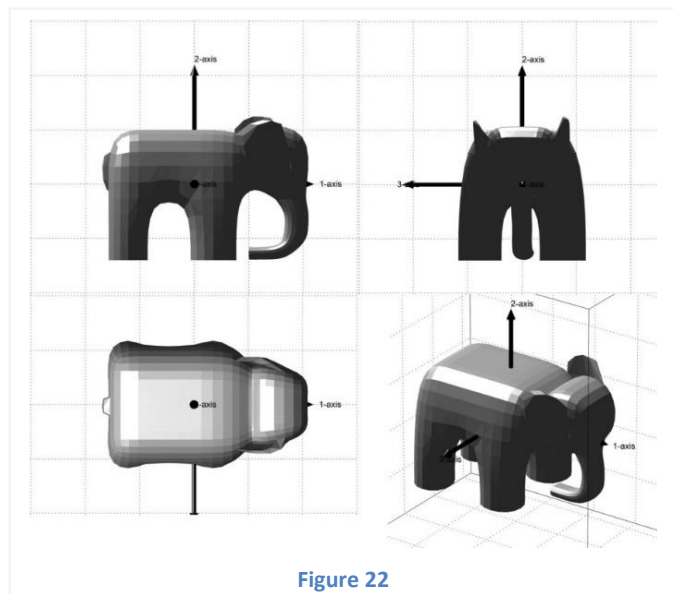


Figure 22

## Exercise IV: Recursive Kinematics

### Problem 21 (2D Kinematics)

✎ Let's consider the motion of a planar robotic arm that is subject to explicit constraints as shown in Figure 23. For each of the two examples shown, compute the position and orientation of the robot at the points  $P_1(x_1 \ y_1 \ \gamma_1)$ ,  $P_2(x_2 \ y_2 \ \gamma_2)$  and of the end-effector at  $P_e(x_e \ y_e \ \gamma_e)$  as a function of the joint coordinates  $q_1 \dots q_4$  and the length of the final link  $d$ . The reference-frame I is located at the origin of the robot, the orientation of the robot segments  $\gamma$  is measured with respect to the vertical, and all angles are assumed to be positive in a counter clockwise direction. This means, in the figure,  $P_1$  is given by  $(0 \ q_1 \ 0^\circ)$ , the value of  $q_2$  is  $-45^\circ$ , and the value of  $q_4$  is  $+45^\circ$ .

State the constraint functions  $f_a$  and  $f_b$  for the Cartesian coordinate vectors  $\mathbf{x}_a$  and  $\mathbf{x}_b$ , and

analytically express the motion subspaces  $\mathbf{J}_a = \frac{\partial f_a}{\partial \mathbf{q}_a}$  and  $\mathbf{J}_b = \frac{\partial f_b}{\partial \mathbf{q}_b}$ .

- $\mathbf{x}_a = [x_1 \ y_1 \ \gamma_1 \ x_e \ y_e \ \gamma_e]^T = f_a(\mathbf{q}_a)$
- $\mathbf{x}_b = [x_1 \ y_1 \ \gamma_1 \ x_2 \ y_2 \ \gamma_2 \ x_e \ y_e \ \gamma_e]^T = f_b(\mathbf{q}_b)$ ,

where  $\mathbf{q}_a = [q_1 \ q_2]^T$  and  $\mathbf{q}_b = [q_1 \ q_2 \ q_3 \ q_4]^T$ . Check the numerical values for

- 1)  $\mathbf{q}_a = [0.1m \ -45^\circ]$ ,  $\mathbf{q}_b = [0.1m \ -45^\circ \ 0.2m \ +45^\circ]$ , and  $d = 0.15m$
- 2)  $\mathbf{q}_a = [0.2m \ +30^\circ]$ ,  $\mathbf{q}_b = [0.2m \ +30^\circ \ 0.1m \ +30^\circ]$ , and  $d = 0.15m$

For case (2) and the joint coordinates  $\mathbf{q}_b$ , your end-effector coordinates should be  $(-0.1799, 0.3616, 1.0472)$ . For case (1) which represents the configurations in Figure 23, draw the components of  $\mathbf{J}_a$  and  $\mathbf{J}_b$  that correspond to the translations of the points into the figure.

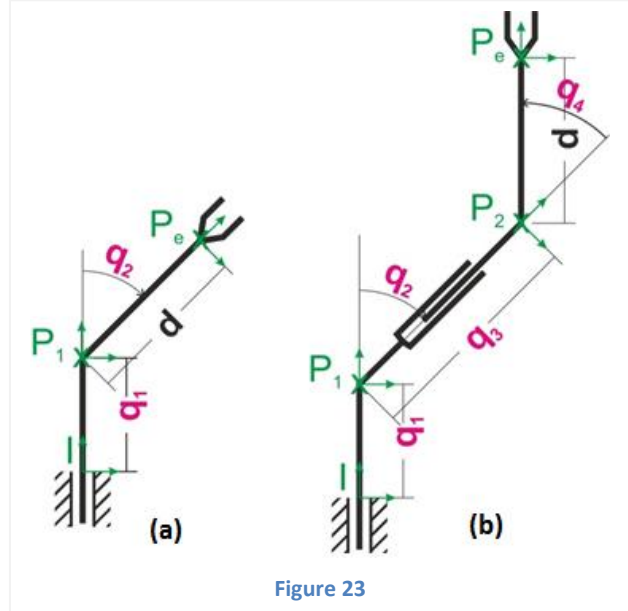


Figure 23

### Problem 22 (Recursive Kinematics)

📖 Note that for the robot in **Problem 21**, case (b) is essentially made from two robots of case (a) mounted on top of each other. Now imagine doing this repeatedly and you will get a robot that has  $2n$  joints which are alternating prismatic and rotational. They define the position of  $n$  points  $P_k$  between the base and the end-effector. See Figure 24 for an example with 12 joints.

Write a piece of Matlab code that computes the position of the end-effector as a function of  $2n$ -long vector of joint variables and the

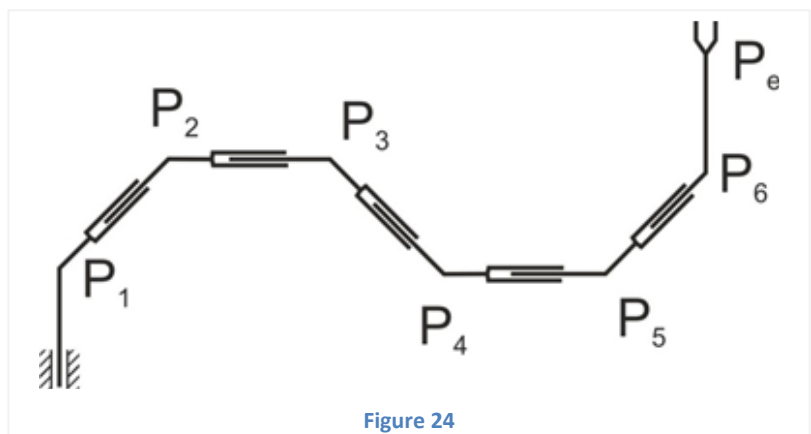


Figure 24



parameter  $d$ . **Do not** try to do this analytically. Instead think of a procedure that could compute the position of point  $P_2$  from the position of  $P_1$  (as they are given in case (b) above) and try to generalize it, such that you can derive the position of any point  $P_k$  from the position of its predecessor  $P_{k-1}$ . Then all you have to do is to write a loop that walks along the joints starting with the origin. You can use the provided template *Template\_22\_RecursiveKinematics.m*.

(a) Compute the position and coordinates of the end-effector for the following cases:

$$\mathbf{q}_1 = [0.1m \quad -45^\circ \quad 0.2m \quad +45^\circ]^T, \quad d = 0.15m$$

$$\mathbf{q}_2 = [0.1m \quad 25^\circ \quad 0.15m \quad -35^\circ \quad 0.05m \quad 33^\circ \quad 0.2m \quad 130^\circ]^T, \quad d = 0.15m$$

$$\mathbf{q}_3 = [0.1m \quad -45^\circ \quad 0.2m \quad -45^\circ \quad 0.2m \quad -45^\circ \quad 0.2m \quad +45^\circ \quad 0.2m \quad +45^\circ \quad 0.2m \quad +45^\circ]^T, \quad d = 0.15m$$

Your numerical result in the last case should be (0.8243, 0.3914, 0).

✂ In the provided template *Template\_22\_RecursiveKinematics.m*, coordinates of the points  $P_k$  are computed iteratively within a for-loop. Change the code such that they are computed with a recursive function call.

### Problem 23 (Linked Lists in Object Oriented Programming)

📖 Read the Matlab documentation on how to implement a linked list in [Object Oriented Programming](#). In particular, you need to read the following chapter. You can search for it in your local documentation, or just click on the link to read online.

- [Class to Implement Linked Lists](#)

### Problem 24 (Rotational Joint and Recursive Forward Kinematics)

📖 Write a class *LinkedRigidBodyDynamicsCLASS.m* that introduces a linked rigid body such that it stores a handle to its parent object (predecessor) and children objects (successors) to which the body is attached. You can use the provided template *Template\_24\_LinkedRigidBodyDynamicsCLASS.m*. Then, write a class *LinkedRotationalJointCLASS.m* that allows introducing a rotational joint between a predecessor body and a successor body. Use the provided template *Template\_24\_LinkedRotationalJointCLASS.m*.

To test your classes use the routine *TestFile\_24\_RotationalJoint.m* which implements a triple pendulum. The resulting output should look as in Figure 25. The second part of the test file generates an animation of the triple pendulum based on the data from a Simscape simulation.

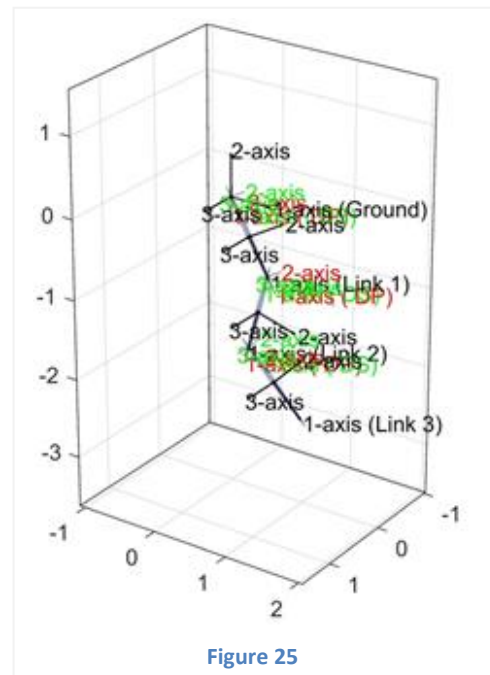


Figure 25

### Problem 25 (Rolling Contact Joint)

📖 Write a class *LinkedRollingContactJointCLASS.m* that allows introducing a rolling contact (without slippage) between a predecessor body which is assumed to be flat and a successor body

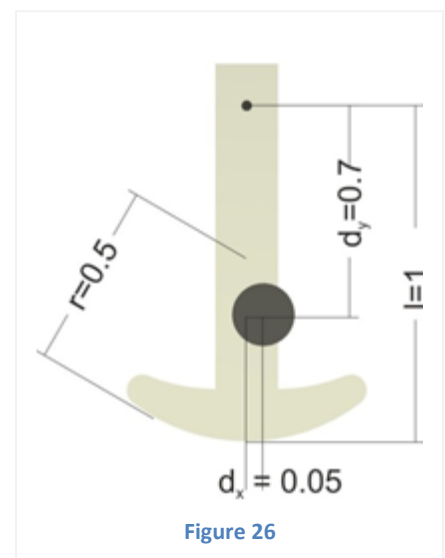


Figure 26

which is circular with a radius of 'r'. Use the provided template *Template\_25\_LinkedRollingContactJointCLASS.m*.

To test this class, we look at the simple kinematic example of a Passive Dynamic Walker (PDW) shown in Figure 26 and Figure 27. This mechanism consists of two legs with circular feet that are connected by a common hinge joint at the hip. Since the legs are lightweight, the position of their COGs is mostly determined by the position of lead weights on the legs (positioned at  $d_x$  and  $d_y$ ). Rather than using meters as units, all dimensions of the PDW are normalized to the length of the legs 'l':

- The leg length is 1.0
- The foot radius is 0.5
- The COG of each leg is positioned
  - 0.7 below the hip axis
  - 0.05 in front of the centerline

The generalized coordinates of this system are the stance leg angle  $\alpha$  (counterclockwise with respect to the vertical) and the angle between the two legs  $\gamma$  (positive if the swing leg is ahead of the stance leg).

Rather than using a virtual 3DOF joint in a floating-base description, we assume that in this particular example one of the two legs (the 'stance leg') is on the ground and model the connection of the ground and the stance leg as a rolling contact. Complete the MATLAB script *Template\_25\_TestRollingContactJoint.m* that defines the PDW and handles its forward kinematics. The script does the following, but is not complete:

- (a) It defines three bodies for 'ground', 'stance leg', and 'swing leg'.
- (b) It defines the two joints for 'ground contact' and 'hip'. You need to fill in the values for the position and orientation of the joints.
- (c) It numerically computes the position and orientation of the two COGs of the legs (as the vector  $\mathbf{x} = [x_{\text{stance}} \ y_{\text{stance}} \ \gamma_{\text{stance}} \ x_{\text{swing}} \ y_{\text{swing}} \ \gamma_{\text{swing}}]^T$ ) for the following values of the generalized coordinates:

$$\circ \quad \mathbf{q} = [\alpha \ \gamma]^T = [-30^\circ \ 60^\circ]^T$$

- (d) It uses the '`recursiveGraphics()`' function to create a visual representation of the PDW. It should produce the picture shown in Figure 28. The second part creates a short animation of the Passive Dynamic Walker.
- (e) It uses symbolic variables to find analytical expressions for the position and orientation of each leg's COG. It also symbolically computes the Jacobian of the COGs. You need to fill in the missing values and variable declarations.



Figure 27

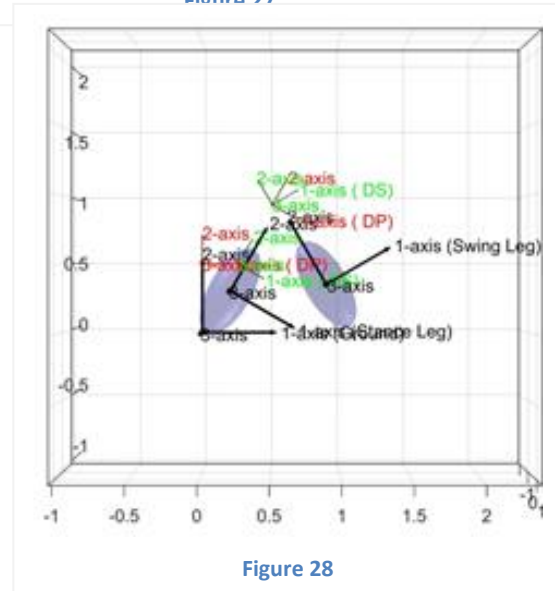


Figure 28

## Problem 26 (Computing Jacobians)

- ☑ The pictures in Figure 29 show a chairplane and its abstraction in a three axis view.

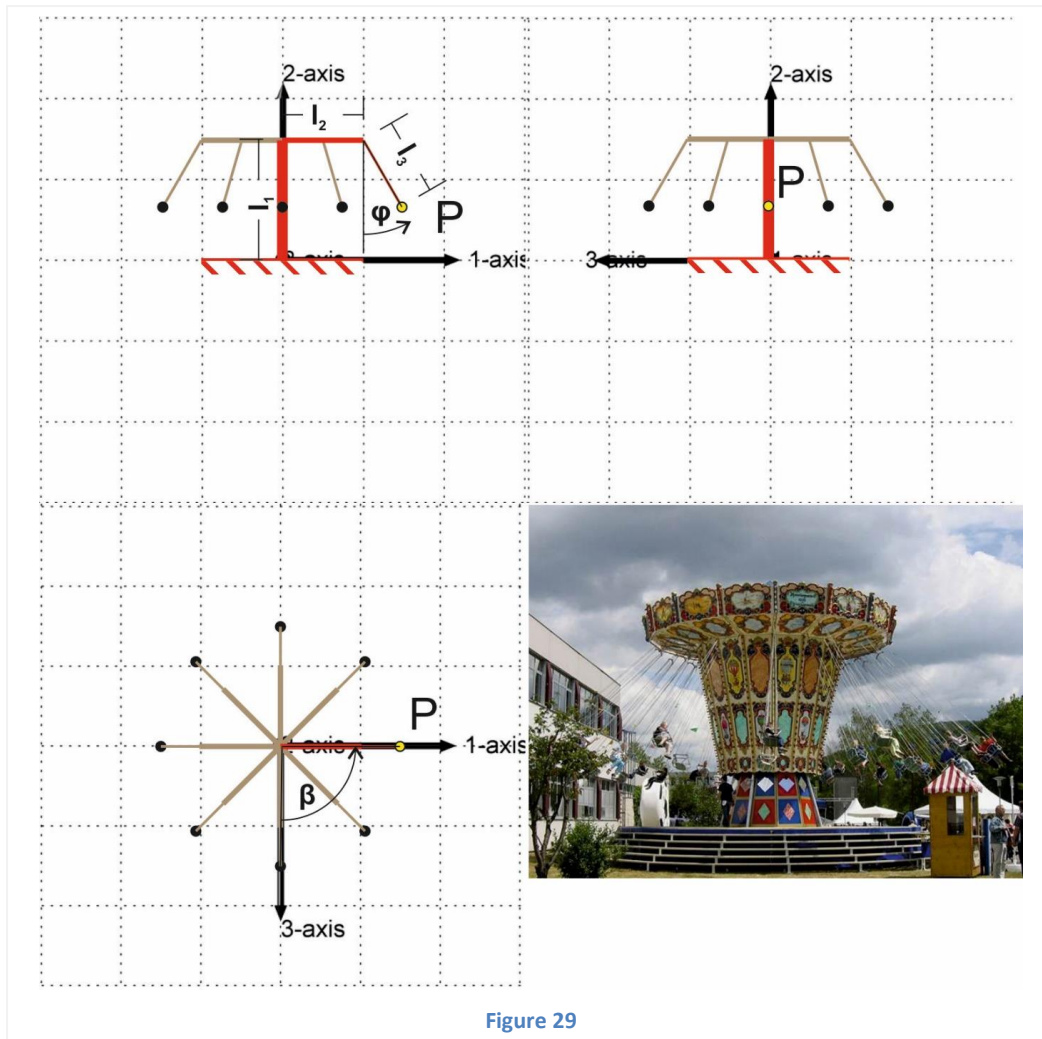


Figure 29

(a) Compute the position  $\vec{x} = [x_1 \ x_2 \ x_3]^T$  of the point  $P$  as a function of the generalized coordinates  $\vec{q} = [\beta \ \varphi]^T$  and the geometric parameters  $l_1$ ,  $l_2$  and  $l_3$ .

$$\vec{x} = f(\vec{q}) = \begin{bmatrix} \rule{10cm}{0.4pt} \\ \rule{10cm}{0.4pt} \\ \rule{10cm}{0.4pt} \end{bmatrix}$$

(b) Compute the constraint Jacobian  $\mathbf{J}_f$  such that  $\dot{\vec{x}} = \mathbf{J}_f \cdot \dot{\vec{q}}$ :

$$\mathbf{J}_f = \begin{bmatrix} \rule{10cm}{0.4pt} & \rule{10cm}{0.4pt} \\ \rule{10cm}{0.4pt} & \rule{10cm}{0.4pt} \\ \rule{10cm}{0.4pt} & \rule{10cm}{0.4pt} \end{bmatrix}$$

(c) Sketch the possible directions of motion (vector length not necessarily drawn to scale) in all three views of Figure 29.

### Problem 27 (Understanding Jacobians)

- (a) Which one of the following matrices does represent the constraint Jacobian  $\mathbf{J}_f = \frac{\partial f}{\partial \mathbf{q}}$  for the system shown in Figure 30? We are only considering the position and orientation of the end effector  $P_e$ . That is, the Cartesian coordinates are  $\mathbf{x} = [x_e \ y_e \ \gamma_e]^T = f_c(\mathbf{q})$ . All units are meters, the gridlines are spaced at 1m, and all angles are assumed to be positive in a counter clockwise direction (i.e.,  $q_1$  and  $q_3$  have negative values).

$$1) \mathbf{J}_f = \begin{bmatrix} +5.17 & +0.97 & +3.35 \\ -2.83 & +0.26 & +3.78 \\ 0 & 0 & 1 \end{bmatrix}$$

$$2) \mathbf{J}_f = \begin{bmatrix} +1.82 & +0.97 & +3.35 \\ -6.76 & +0.26 & +3.78 \\ 1 & 0 & 1 \end{bmatrix}$$

$$3) \mathbf{J}_f = \begin{bmatrix} -6.76 & +0.26 & +3.78 \\ +1.82 & +0.97 & +3.35 \\ 1 & 0 & 1 \end{bmatrix}$$

$$4) \mathbf{J}_f = \begin{bmatrix} -2.83 & +0.26 & +3.78 \\ +5.17 & +0.97 & +3.35 \\ 1 & 0 & 1 \end{bmatrix}$$

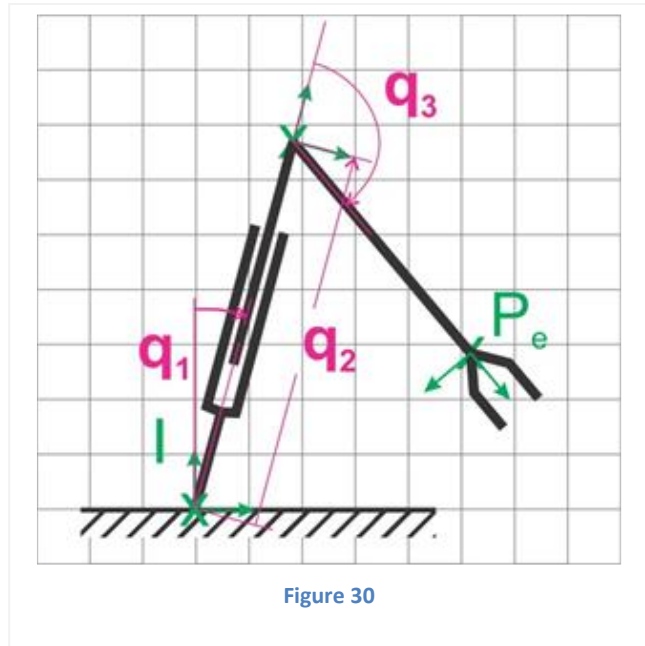


Figure 30

- (b) Which one of the following statements can be used to compute the Cartesian velocity  ${}_I \dot{\mathbf{x}}$  for the specified constraint type? All constraints are explicit equality constraints.
- 1)  ${}_I \dot{\mathbf{x}} = {}_I \mathbf{J} \cdot \dot{\mathbf{q}} + \frac{\partial f}{\partial t}$ , for holonomic constraints
  - 2)  ${}_I \dot{\mathbf{x}} = {}_I \mathbf{J} \cdot \dot{\mathbf{q}} + \frac{df}{dt}$ , for rheonomic constraints
  - 3)  ${}_I \dot{\mathbf{x}} = {}_I \mathbf{J} \cdot \dot{\mathbf{q}} + \frac{\partial f}{\partial t}$ , for non-holonomic constraints
  - 4)  ${}_c \dot{\mathbf{x}} = {}_c \mathbf{J} \cdot \dot{\mathbf{q}} + \frac{df}{dt}$ , for scleronomic constraints
- (c) Which one of the following statements is **NOT** correct for the bias acceleration  ${}_I \sigma_c$  of an explicit **rehonomic** constraint  ${}_I \mathbf{x} = f_c(\mathbf{q}, t)$ ?
- 1)  ${}_I \sigma_c = {}_I \dot{\mathbf{J}}_c \cdot \dot{\mathbf{q}} + {}_I \ddot{\mathbf{x}}$
  - 2)  ${}_I \sigma_c = {}_I \ddot{\mathbf{x}} - {}_I \mathbf{J}_c \cdot \ddot{\mathbf{q}}$
  - 3)  ${}_I \sigma_c = {}_I \dot{\mathbf{J}}_c \cdot \dot{\mathbf{q}} + \frac{d}{dt} \left( \frac{\partial f_c}{\partial t} \right)$
  - 4)  ${}_I \sigma_c = \frac{d^2 f_c}{dt^2} - {}_I \mathbf{J}_c \cdot \ddot{\mathbf{q}}$
- (d) Figure 31 shows a hobbyist's implementation of a flyball governor. In the configuration shown, the angle of the axis is  $\varphi = 0$  and the angle of the linkage is  $\gamma = 30^\circ$ .

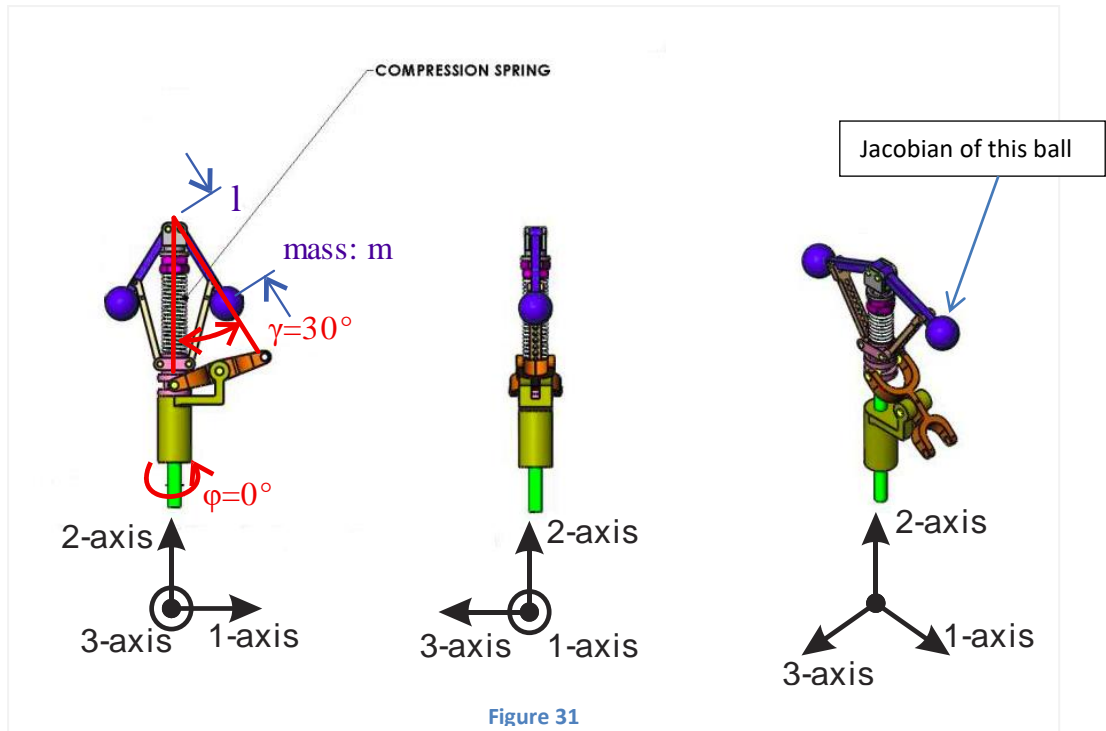


Figure 31

Which one of the following Jacobians describes the velocity of the ball on the right  ${}_I \dot{\mathbf{x}} = {}_I \mathbf{J} \dot{\mathbf{q}}$  with  $\mathbf{q} = (\varphi, \gamma)^T$  for the configuration shown?

$$(1) {}_I \mathbf{J} = \begin{bmatrix} \frac{l\sqrt{3}}{2} & \frac{l\sqrt{3}}{2} \\ \frac{l}{2} & 0 \\ 0 & -\frac{l}{2} \end{bmatrix} \quad (2) {}_I \mathbf{J} = \begin{bmatrix} \frac{l\sqrt{3}}{2} & 0 \\ 0 & \frac{l}{2} \\ -\frac{l}{2} & 0 \end{bmatrix} \quad (3) {}_I \mathbf{J} = \begin{bmatrix} 0 & \frac{l\sqrt{3}}{2} \\ 0 & \frac{l}{2} \\ -\frac{l}{2} & 0 \end{bmatrix} \quad (4) {}_I \mathbf{J} = \begin{bmatrix} 0 & \frac{l\sqrt{3}}{2} \\ \frac{l}{2} & 0 \\ 0 & -\frac{l}{2} \end{bmatrix}$$

(e) Which one of the following statements is **NOT** correct? (Note: 'I' is a stationary point and coordinate system, 'B' is an arbitrary body-fixed point and coordinate system, and 'G' is the center of gravity).

- 1)  ${}_B \dot{\vec{r}}_{IB} = {}_B \mathbf{v}_B - {}_B \tilde{\boldsymbol{\omega}}_B \cdot {}_B \vec{r}_{IB}$
- 2)  ${}_I \ddot{\vec{r}}_{IB} = {}_I \mathbf{a}_B - {}_I \tilde{\boldsymbol{\omega}}_{IB} \times {}_I \mathbf{v}_B$
- 3)  ${}_B \dot{\vec{L}}_G = {}_B (\dot{\vec{L}}_G) - {}_B \tilde{\boldsymbol{\omega}}_{IB} \times {}_B \vec{L}_G$
- 4)  ${}_B \dot{\vec{P}} = {}_B (\dot{\vec{P}}) - {}_B \tilde{\boldsymbol{\omega}}_B \cdot {}_B \vec{P}$

## Problem 28 (Understanding Recursion)

(a) The code below defines a Matlab class (in the file 'ColorCLASS.m') and a script (in the file 'main.m') that uses this class. State the output that you get when running the script 'main.m':

```
% In the file ColorCLASS.m
classdef ColorCLASS < handle
    properties
        child = [];
        color = 'r';
        isLeaf = false;
    end
end
```

```

methods
    % Constructor
    function obj = ColorCLASS(color)
        obj.color = color;
    end
    % Rainbow function
    function Rainbow(obj)
        if obj.isLeaf
            disp('Sandwich');
        else
            disp(obj.color);
            obj.child.Rainbow();
        end
    end
end
end

% In the file main.m
green = ColorCLASS('green');
blue  = ColorCLASS('blue');
purple = ColorCLASS('purple');
red    = ColorCLASS('red');
yellow = ColorCLASS('yellow');

green.child = purple;
blue.child  = green;
yellow.child = blue;
red.child   = yellow;

green.isLeaf = true;

clc
disp('Rainbow:')
red.Rainbow();

```

## Exercise V: Inheritance, Velocities, Accelerations

### Problem 29 (Inheritance and Recursion)

Read the Matlab documentation on how to implement a linked list in [Object Oriented Programming](#). In particular, you need to read the following sections. You can search for them in your local documentation, or just click on the links to read online.

- [Role of Classes in MATLAB](#)
- [Hierarchies of Classes — Concepts](#)
- [Subclass Syntax](#)
- [Modify Inherited Methods](#)

### Problem 30 (Inheritance to Structure Code)

In the class files for this exercise, you will find new versions (...CLASS\_v2) of all the classes that we need to implement a kinematic tree in Matlab. They can be tested with the file *TestFile\_30\_RotationalJoint\_v2.m* that computes the kinematics of a triple pendulum. This script is exactly the same as the test file for Problem 24 (*TestFile\_24\_RotationalJoint.m*), with the only difference that it uses the CLASSES *LinkedRigidBodyDynamicsCLASS\_v2* and *LinkedRotationalJointCLASS\_v2*.

None of the functionality has changed in the new classes, but we use inheritance to take advantage of the fact that *RigidBodyDynamicsCLASS* extends the functionality of *RigidBodyKinematicsCLASS*, that *LinkedRigidBodyDynamicsCLASS* extends *RigidBodyDynamicsCLASS* and *LinkedBodyObjectCLASS*, and that *LinkedRotationalJointCLASS* extends *LinkedJointObjectCLASS*.

- Establish a CLASS diagram that shows how these classes (and the new class *LinkedGenericJointCLASS\_v2*) depend on each other, and which class inherits from which other class(es).
- Verify that the functionality is indeed the same, just distributed over different files. A good way to do so is the MATLAB code comparison tool (Figure 32) which you can use to compare the original CLASS files from Exercise IV with the new versions of Exercise V.
- Why did we change certain properties from being “private” to being “protected”?
- What does the following line do in the constructor of *LinkedRotationalJointCLASS\_v2.m*:  
`obj = obj@LinkedGenericJointCLASS_v2(env, predBody, sucBody) ?`
- Why do we first define a *LinkedGenericJointCLASS\_v2* (that inherits from *LinkedJointObjectCLASS\_v2*) and then inherit from it to get the *LinkedRotationalJointCLASS\_v2* rather than directly have *LinkedRotationalJointCLASS\_v2* inherit from *LinkedJointObjectCLASS\_v2*?

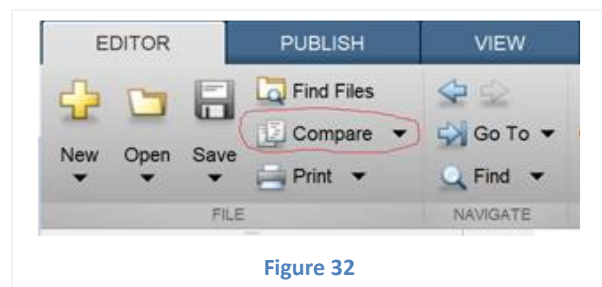


Figure 32

### Problem 31 (Kinematics with Velocities, Accelerations, and Jacobians)

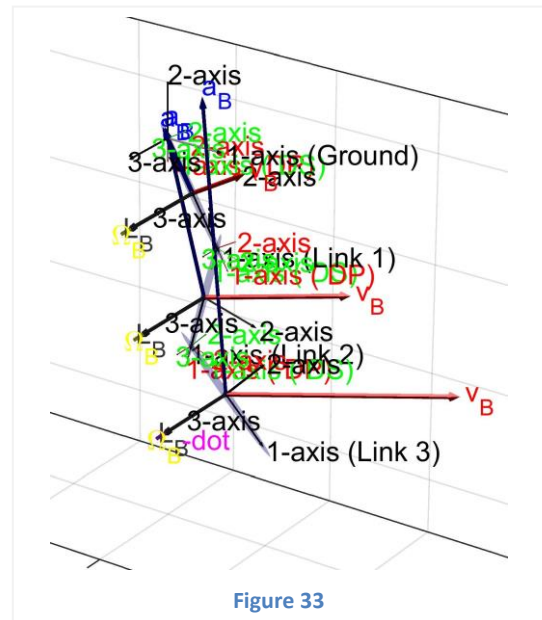
Based on the concept of inheritance, we will now implement three new classes that allow us to not only compute positions/orientations in a complex kinematic tree, but also to compute velocities, accelerations, and Jacobians. These classes are called VeAcJa classes. To this end, we need to write three new classes (provided as templates):

- *LinkedRigidBodyDynamicsVeAcJaCLASS* which inherits most properties from *LinkedRigidBodyDynamicsCLASS\_v2* and extends it with variables to store and compute Jacobians. It also implements a new `recursiveForwardKinematicsVeAcJa` function that recursively computes positions/orientations, velocities, accelerations, and Jacobians.



- *LinkedGenericJointVeAcJaCLASS* which inherits most properties from *LinkedRigidBodyDynamicsCLASS\_v2* and extends it with a new function *recursiveForwardKinematicsVeAcJa* that recursively computes positions/orientations, velocities, accelerations, and Jacobians across the joint. This is a generic joint class that does not implement a specific behavior, but that serves as a template for all different types of joints.
- *LinkedRotationalJointVeAcJaCLASS* finally inherits from *LinkedGenericJointVeAcJaCLASS* and implements the specifics of a rotational joint.

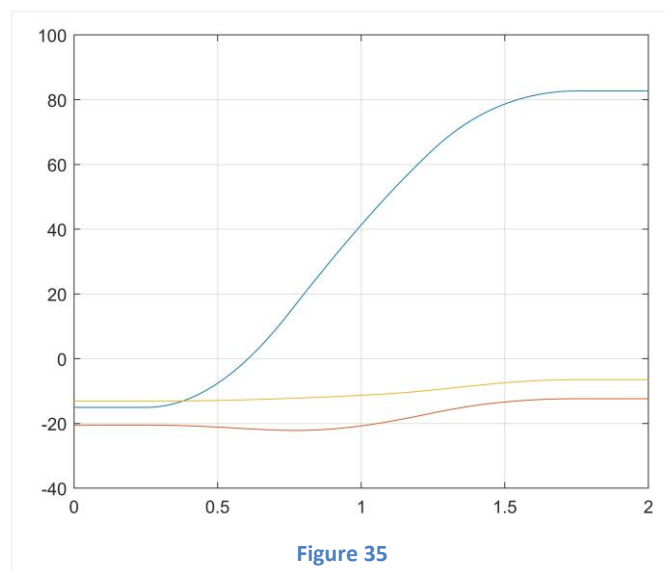
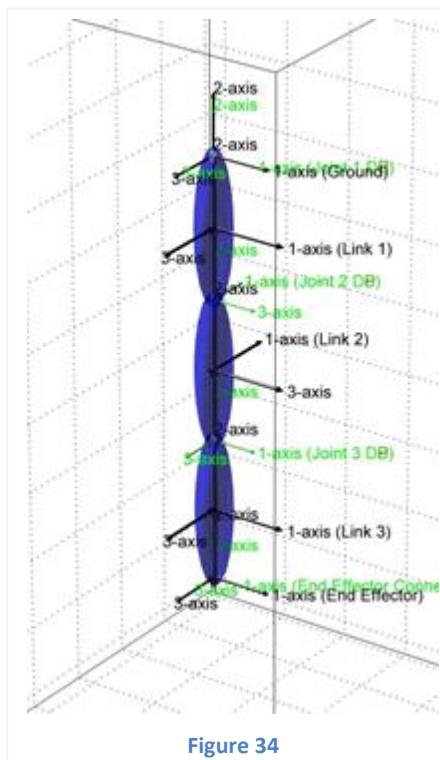
Finalize the three classes and test them with the script *TestFile\_31\_RotationalJointVeAcJa.m*. It should produce the output shown in Figure 33.

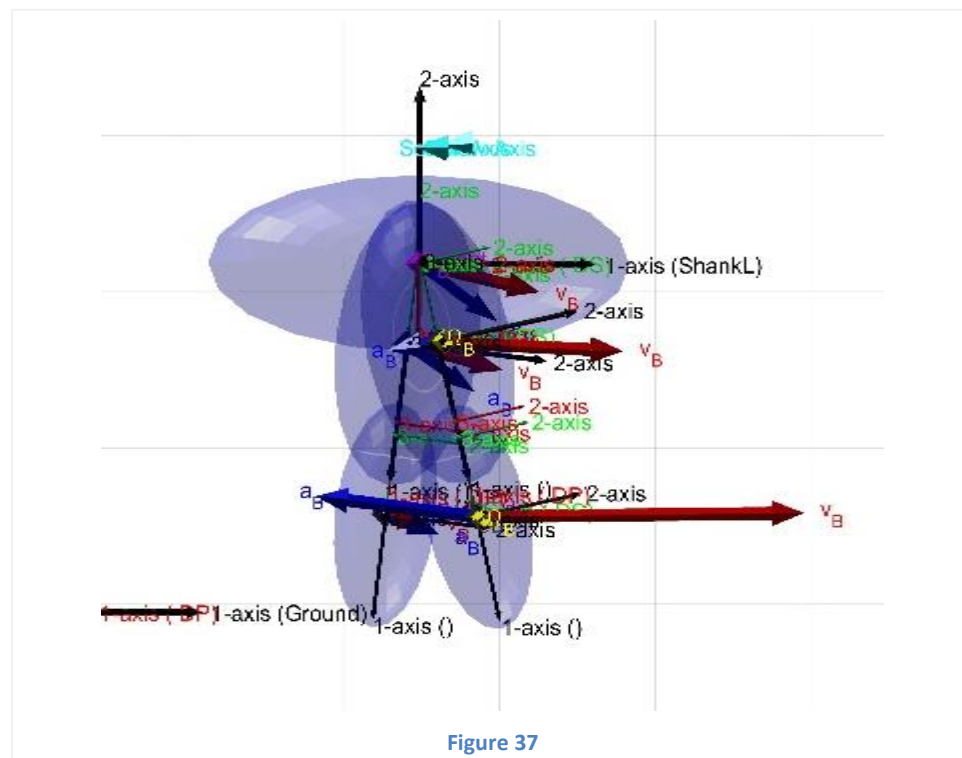


### Problem 32 (Robotic Arm. 1 – Gravity compensation)

In this and some of the following exercises, we study the kinematics and dynamics of a three segment robotic arm (Figure 34). The robot consists of three identical segments or ‘links’ and an end-effector. The first link is attached to a stationary ground base.

In the initial configuration (i.e., when all joint angles are measured to be 0), the robot is pointing straight down, with the rotational joints either aligned along the positive 3-axis (joint 1 and joint 3), or the positive 1-axis (joint 2). The ground reference frame is located where the first link of the robot is connected to the ground. For simplicity, we assume that the center of gravity of each link is located at the center between the joint coordinate systems. I.e., in the initial configuration, all COGs are aligned along a straight line. Each link is 1 m long and weighs 3 Kg. The end-effector has no mass and inertia and does not influence the dynamics of the robot, yet it provides a coordinate system that defines the orientation and position of the end-effector. I.e., the work space motion of the robot is defined for this coordinate system.





### Problem 35 (Specific Joint Functions)

✍ Naturally, the computation of the orientation, translation, velocities, accelerations, and Jacobians across the joint depends on the type of joint used. State these expressions for the four types of joints that we discussed in class:

(a) Rotational joint, where  $\mathbf{q} = [\gamma]$

$$\begin{aligned}
 \mathbf{A}_{D_p D_S}(\mathbf{q}) &= \begin{bmatrix} \_ & \_ & \_ \\ \_ & \_ & \_ \\ \_ & \_ & \_ \end{bmatrix} \\
 {}_{D_p} \mathbf{r}_{D_p D_S}(\mathbf{q}) &= [\_ \_ \_]^T \\
 {}_{D_p} \boldsymbol{\omega}_{D_p D_S}(\mathbf{q}, \dot{\mathbf{q}}) &= [\_ \_ \_]^T \\
 {}_{D_p} \dot{\mathbf{r}}_{D_p D_S}(\mathbf{q}, \dot{\mathbf{q}}) &= [\_ \_ \_]^T \\
 {}_{D_p} \dot{\boldsymbol{\omega}}_{D_p D_S}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) &= [\_ \_ \_]^T \\
 {}_{D_p} \ddot{\mathbf{r}}_{D_p D_S}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) &= [\_ \_ \_]^T \\
 R(\mathbf{q}) &= \begin{bmatrix} 0 & 0 & \_ & 0 & 0 \\ 0 \cdots 0 & \_ & 0 \cdots 0 & & \\ 0 & 0 & \_ & 0 & 0 \end{bmatrix} \\
 S(\mathbf{q}) &= \begin{bmatrix} 0 & 0 & \_ & 0 & 0 \\ 0 \cdots 0 & \_ & 0 \cdots 0 & & \\ 0 & 0 & \_ & 0 & 0 \end{bmatrix}
 \end{aligned}$$

(b) Translational joint, where  $\mathbf{q} = [\Delta x]$

$$\begin{aligned}
 \mathbf{A}_{D_p D_S}(\mathbf{q}) &= \begin{bmatrix} \_ & \_ & \_ \\ \_ & \_ & \_ \\ \_ & \_ & \_ \end{bmatrix} \\
 {}_{D_p} \mathbf{r}_{D_p D_S}(\mathbf{q}) &= [\_ \_ \_]^T \\
 {}_{D_p} \boldsymbol{\omega}_{D_p D_S}(\mathbf{q}, \dot{\mathbf{q}}) &= [\_ \_ \_]^T \\
 {}_{D_p} \dot{\mathbf{r}}_{D_p D_S}(\mathbf{q}, \dot{\mathbf{q}}) &= [\_ \_ \_]^T \\
 {}_{D_p} \dot{\boldsymbol{\omega}}_{D_p D_S}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) &= [\_ \_ \_]^T \\
 {}_{D_p} \ddot{\mathbf{r}}_{D_p D_S}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) &= [\_ \_ \_]^T \\
 R(\mathbf{q}) &= \begin{bmatrix} 0 & 0 & \_ & 0 & 0 \\ 0 \cdots 0 & \_ & 0 \cdots 0 & & \\ 0 & 0 & \_ & 0 & 0 \end{bmatrix} \\
 S(\mathbf{q}) &= \begin{bmatrix} 0 & 0 & \_ & 0 & 0 \\ 0 \cdots 0 & \_ & 0 \cdots 0 & & \\ 0 & 0 & \_ & 0 & 0 \end{bmatrix}
 \end{aligned}$$

(c) 3DOF joint, where  $\mathbf{q} = [\Delta x \quad \Delta y \quad \gamma]^T$

$$\begin{aligned}
 \mathbf{A}_{D_P D_S}(\mathbf{q}) &= \begin{bmatrix} \_ & \_ & \_ \\ \_ & \_ & \_ \\ \_ & \_ & \_ \end{bmatrix} \\
 {}_{D_P} \mathbf{r}_{D_P D_S}(\mathbf{q}) &= [\_ \quad \_ \quad \_]^T \\
 {}_{D_P} \boldsymbol{\omega}_{D_P D_S}(\mathbf{q}, \dot{\mathbf{q}}) &= [\_ \quad \_ \quad \_]^T \\
 {}_{D_P} \dot{\mathbf{r}}_{D_P D_S}(\mathbf{q}, \dot{\mathbf{q}}) &= [\_ \quad \_ \quad \_]^T \\
 {}_{D_P} \dot{\boldsymbol{\omega}}_{D_P D_S}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) &= [\_ \quad \_ \quad \_]^T \\
 {}_{D_P} \ddot{\mathbf{r}}_{D_P D_S}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) &= [\_ \quad \_ \quad \_]^T \\
 R(\mathbf{q}) &= \begin{bmatrix} 0 & 0 & \_ & \_ & \_ & 0 & 0 \\ 0 \cdots 0 & \_ & \_ & \_ & \_ & 0 \cdots 0 \\ 0 & 0 & \_ & \_ & \_ & 0 & 0 \end{bmatrix} \\
 S(\mathbf{q}) &= \begin{bmatrix} 0 & 0 & \_ & \_ & \_ & 0 & 0 \\ 0 \cdots 0 & \_ & \_ & \_ & \_ & 0 \cdots 0 \\ 0 & 0 & \_ & \_ & \_ & 0 & 0 \end{bmatrix}
 \end{aligned}$$

(d) Rolling contact joint, where  $\mathbf{q} = [\gamma]$

$$\begin{aligned}
 \mathbf{A}_{D_P D_S}(\mathbf{q}) &= \begin{bmatrix} \_ & \_ & \_ \\ \_ & \_ & \_ \\ \_ & \_ & \_ \end{bmatrix} \\
 {}_{D_P} \mathbf{r}_{D_P D_S}(\mathbf{q}) &= [\_ \quad \_ \quad \_]^T \\
 {}_{D_P} \boldsymbol{\omega}_{D_P D_S}(\mathbf{q}, \dot{\mathbf{q}}) &= [\_ \quad \_ \quad \_]^T \\
 {}_{D_P} \dot{\mathbf{r}}_{D_P D_S}(\mathbf{q}, \dot{\mathbf{q}}) &= [\_ \quad \_ \quad \_]^T \\
 {}_{D_P} \dot{\boldsymbol{\omega}}_{D_P D_S}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) &= [\_ \quad \_ \quad \_]^T \\
 {}_{D_P} \ddot{\mathbf{r}}_{D_P D_S}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) &= [\_ \quad \_ \quad \_]^T \\
 R(\mathbf{q}) &= \begin{bmatrix} 0 & 0 & \_ & 0 & 0 \\ 0 \cdots 0 & \_ & 0 \cdots 0 \\ 0 & 0 & \_ & 0 & 0 \end{bmatrix} \\
 S(\mathbf{q}) &= \begin{bmatrix} 0 & 0 & \_ & 0 & 0 \\ 0 \cdots 0 & \_ & 0 \cdots 0 \\ 0 & 0 & \_ & 0 & 0 \end{bmatrix}
 \end{aligned}$$

### Problem 36 (Understanding Inheritance)

The code below defines three Matlab classes (in three different files 'WonderlandCharacter.m', 'TweedleDum.m', and 'TweedleDee.m') and a script (in the file 'main.m') that uses these classes.

```
% In the file WonderlandCharacter.m
classdef WonderlandCharacter < handle
    properties (Access = public)
        name = []; % A string
        foe = []; % Another WonderlandCharacter
    end
    methods
        function obj = WonderlandCharacter(name, foe) % Constructor
            obj.foe = foe;
            obj.name = name;
        end
    end
end

% In the file TweedleDum.m
classdef TweedleDum < WonderlandCharacter
    methods
        function obj = TweedleDum() % Constructor
            obj = obj@WonderlandCharacter('TweedleDum', []);
            tweedleDee = TweedleDee('TweedleDee', obj);
            obj.foe = tweedleDee;
        end
        function TellStory(obj) % Story telling function
            disp([obj.name, ' and ', obj.foe.name]);
            disp('    Agreed to have a battle');
            obj.foe.TellStory();
        end
    end
end

% In the file TweedleDee.m
classdef TweedleDee < WonderlandCharacter
    methods
        function obj = TweedleDee(name, foe) % Constructor
            obj = obj@WonderlandCharacter(name, foe);
        end
        function TellStory(obj) % Story telling function
            disp(['For ', obj.name, ' said ', obj.foe.name]);
            disp('    Had spoiled his nice new rattle');
        end
    end
end

% In the file main.m
%% PART I
redQueen = WonderlandCharacter('The Red Queen', []);
alice = WonderlandCharacter('Alice', redQueen);
redQueen.foe = alice; % <----
disp([alice.name, ' is fighting ', redQueen.name]);
disp([alice.foe.name, ' is fighting ', redQueen.foe.name]);
%% PART II
tweedleDum = TweedleDum();
tweedleDum.TellStory();
```

- (a) Show the hierarchy of the three classes in a diagram.
- (b) What is the output of 'PART I'?

- (c) Is there a way of getting rid of the third line of 'PART I' (`redQueen.foe = alice`) by changing the first line `redQueen = WonderlandCharacter('The Red Queen', [])`? If yes, how? If not, why not?
- (d) What is the output of 'PART II'?

## Exercise VI: Multibody Dynamics

### Problem 37 (Robotic Arm. 2 – Controller Evaluation)

In this problem, we will simulate the dynamics of the robotic arm that was introduced in Problem 32. The configuration of this robot including all joint parameters, link lengths, masses, and inertia is provided in the code template *Template\_37\_ControllerEvaluation.m* which serves as the basis for our analysis.

In this exercise, we first implement the EOM using the projected Newton Euler Equations and then evaluate three joint-level controllers that force the robot to follow a desired set of joint trajectories. Each controller computes a set of active motor torques  $\tau$ , that are a result of the desired joint values  $(\mathbf{q}_{des}, \dot{\mathbf{q}}_{des}, \ddot{\mathbf{q}}_{des})$  and the actual joint values  $(\mathbf{q}, \dot{\mathbf{q}})$  at a given time  $t$ . To this end, the function `ODE()` calls the function `tau = controller(t_, q_, d_q_)` during the forward dynamic simulation. Since this is a nested function, it has access to the variables `q_des`, `d_q_des`, and `dd_q_des`.

- (a) Let's first finish the code necessary for the dynamic simulation. In particular, we introduce a new class *LinkedRigidBodyDynamicsMfgCLASS.m*, which extends *LinkedRigidBodyDynamicsVeAcJaCLASS.m* with a function that recursively computes the terms  $\mathbf{M}$ ,  $\mathbf{f}$ , and  $\mathbf{g}$  of the equations of motion. Complete the provided template *Template\_37\_LinkedRigidBodyDynamicsMfgCLASS.m* by implementing the code that computes the contribution of this specific body to  $\mathbf{M}$ ,  $\mathbf{f}$ , and  $\mathbf{g}$ .

*Controller #1* is already implemented. It is a simple PD-controller in which the motor torques are computed according to:  $\tau = P \cdot (\mathbf{q}_{des} - \mathbf{q}) + D \cdot (\dot{\mathbf{q}}_{des} - \dot{\mathbf{q}})$ . That is, if the actual joint angles and the actual generalized joint velocities deviate from the desired trajectories, corrective torques are sent to the motor that force the joint trajectories back onto the desired trajectories. The obvious drawback is that this controller only applies a torque if a position and/or velocity error is present. Since we already know that motor torques are required to generate accelerations and to overcome gravity, this is probably not the best solution. You can run *Template\_37\_ControllerEvaluation.m* to see how this controller performs and check whether you have correctly implemented the necessary code in *LinkedRigidBodyDynamicsMfgCLASS.m*. Your output should look like in Figure 38:

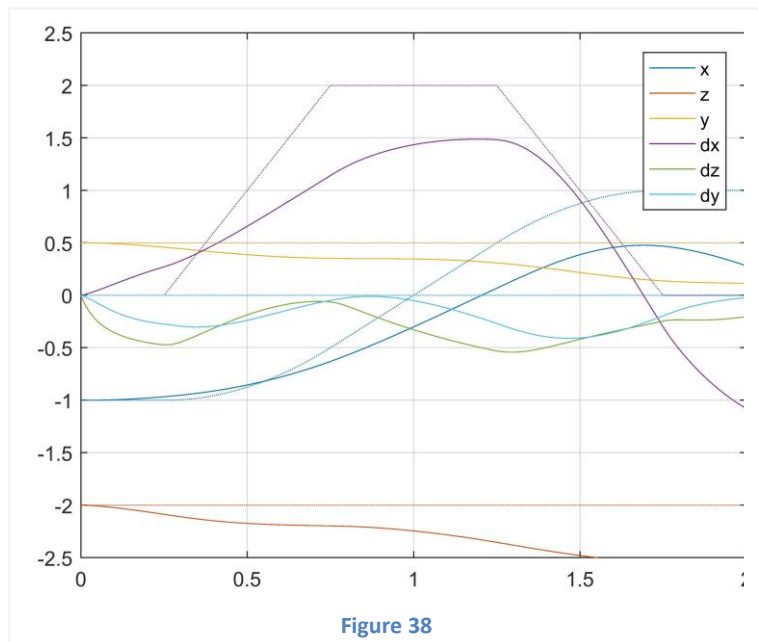


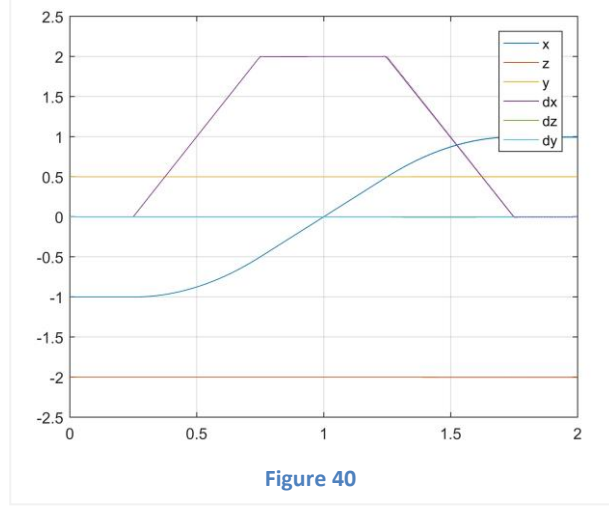
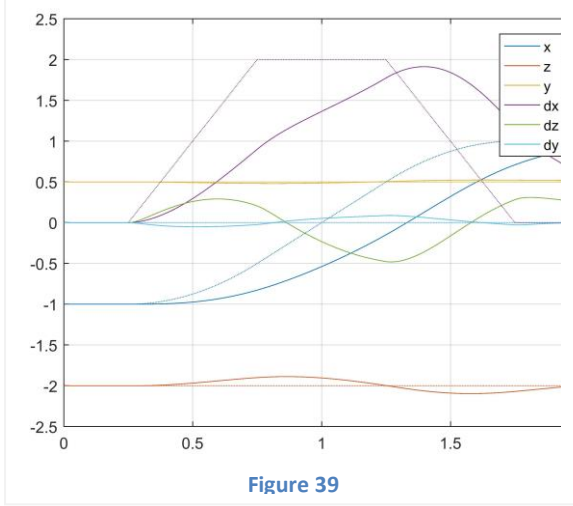
Figure 38

- (b) Let's now implement two better controllers that include a feedforward component and take advantage of the fact that we know the dynamic model.


*Controller #2* implements a gravity compensation term. Knowing the current values of  $\mathbf{q}$ , project the active (gravitational) forces into the general coordinate space and use them to compensate for gravity when computing  $\tau$ . As long as the robot is not commanded to move (i.e., for the first 0.25 seconds of the trajectory), the robot should remain at rest with this controller (and not collapse). However, we still get deviations from the desired trajectory while we are moving, as we do not account for torques that are necessary to accelerate the robot.

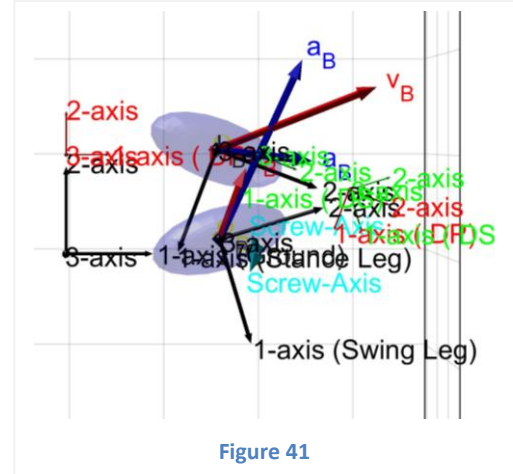


To fix this, *Controller #3* essentially runs a full inverse dynamics computation each time the controller is called. Knowing the current values of  $\mathbf{q}$  and  $\dot{\mathbf{q}}$  you can compensate for gravity, Coriolis, and centrifugal forces. And knowing the desired accelerations  $\ddot{\mathbf{q}}_{des}$ , you can compute motor torques  $\boldsymbol{\tau}$  that ensure that the joints accelerate in this way. This is called the method of *computed torques*. Implement both controllers in the template and select them by sequentially uncommenting lines 167 and 168, the outputs should look like in Figure 39 and Figure 40.




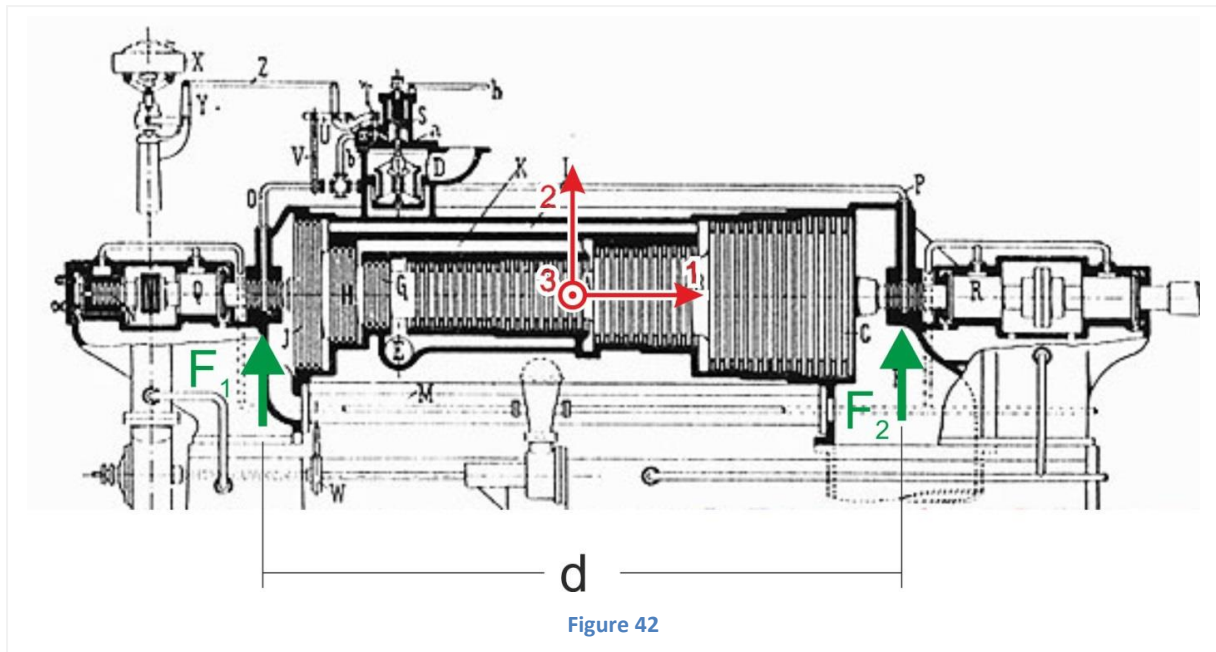
### Problem 38 (Passive Dynamic Walker)

 In this problem, we will simulate the dynamics of the Passive Dynamic Walker introduced in Problem 25 and Problem 33. *Template\_38\_PassiveDynamicWalker.m* contains the description of this system and the necessary code to run a simulation. However, you still need to complete the ODE function, where you need to compute the generalized accelerations using the recursive methods provided by our framework. You can use the code of Problem 37 as an example. After correct implementation, the last frame of the animation should look like in Figure 41.



### Problem 39 (Turbine)

 A steam turbine shown in Figure 42 (with roughly a length of 10 m and a rotor-radius of about 1 m) weighs 100 metric tons. The turbine has radial symmetry with respect to its rotation axis, and the inertia about this axis is  $50'000 \text{ Kg} \cdot \text{m}^2$ . The inertia about any axis that is perpendicular to the rotation axis (and that goes through the center of gravity) is  $450'000 \text{ Kg} \cdot \text{m}^2$ . During regular operation, the turbine is rotating with a constant angular velocity of 1'500 rotations per minute,  $\vec{\omega} = (1500 \text{ rpm} \ 0 \ 0)^T$ .



(a) Complete the inertia matrix  $I_1$  of the turbine, such that its entries match the figure and the values given above.

$$I_1 = \begin{bmatrix} \underline{\hspace{2cm}} & \underline{\hspace{2cm}} & \underline{\hspace{2cm}} \\ \underline{\hspace{2cm}} & \underline{\hspace{2cm}} & \underline{\hspace{2cm}} \\ \underline{\hspace{2cm}} & \underline{\hspace{2cm}} & \underline{\hspace{2cm}} \end{bmatrix}$$

(b) Due to a fabrication error, the rotation axis of the turbine is misaligned by an angle of  $1/1000$  radians with its axis of radial symmetry. Compute the inertia matrix  $I_2$  of this defect turbine, if –at this instance– the misalignment corresponds to a positive rotation about the 3-axis. Use the approximations for small angles:  $\sin(\alpha) = \alpha$  and  $\cos(\alpha) = 1$ .

$$I_2 = \begin{bmatrix} \underline{\hspace{2cm}} & \underline{\hspace{2cm}} & \underline{\hspace{2cm}} \\ \underline{\hspace{2cm}} & \underline{\hspace{2cm}} & \underline{\hspace{2cm}} \\ \underline{\hspace{2cm}} & \underline{\hspace{2cm}} & \underline{\hspace{2cm}} \end{bmatrix}$$

(c) Compute the forces  $F_1$  and  $F_2$  (including signs, see Figure 42 for force direction) that the bearings must withstand at this instance when the defect turbine is forced to rotate along a constant axis and with a constant angular rate  $\vec{\omega} = (1500 \text{ rpm} \ 0 \ 0)^T$ , with  $\dot{\vec{\omega}} = \vec{0}$ . Do take into account both gravitational and dynamic loads. The bearings are spaced at a distance  $d$  of 10 meters. For this question, approximate:  $\pi = \sqrt{10}$ ,  $g = 10$ .

$$\vec{F}_1 = \underline{\hspace{2cm}}$$

$$\vec{F}_2 = \underline{\hspace{2cm}}$$

### Problem 40 (Equations of Motion for a System of Particles)

✍ For the double pendulum that we covered in class and that is shown in Figure 43, analytically derive the equations of motion using the method of projected Newton-Euler equations. I would suggest using the symbolic math toolbox in Matlab. *Example\_9\_2DOFManipulator.m* might be a good starting point.

My results were:

$$\mathbf{M}(\mathbf{q}) \cdot \ddot{\mathbf{q}} - \mathbf{f}(\mathbf{q}, \dot{\mathbf{q}}) - \mathbf{g}(\mathbf{q}) = \mathbf{0}$$

With:

$$\mathbf{M}(\mathbf{q}) = \begin{bmatrix} m_1 \cdot l_1^2 + m_2 \cdot (l_1^2 + 2 \cdot l_1 \cdot l_2 \cdot \cos(q_2) + l_2^2) & m_2 \cdot l_2 \cdot (l_2 + l_1 \cdot \cos(q_2)) \\ m_2 \cdot l_2 \cdot (l_2 + l_1 \cdot \cos(q_2)) & m_2 \cdot l_2^2 \end{bmatrix}$$

$$\mathbf{f}(\mathbf{q}, \dot{\mathbf{q}}) = m_2 \cdot l_1 \cdot l_2 \cdot \sin(q_2) \cdot \begin{bmatrix} +\dot{q}_2 \cdot (2 \cdot \dot{q}_1 + \dot{q}_2) \\ -\dot{q}_1^2 \end{bmatrix}$$

$$\mathbf{g}(\mathbf{q}) = -g \cdot \begin{bmatrix} (m_1 + m_2) \cdot l_1 \cdot \sin(q_1) + m_2 \cdot l_2 \cdot \sin(q_1 + q_2) \\ m_2 \cdot l_2 \cdot \sin(q_1 + q_2) \end{bmatrix}$$

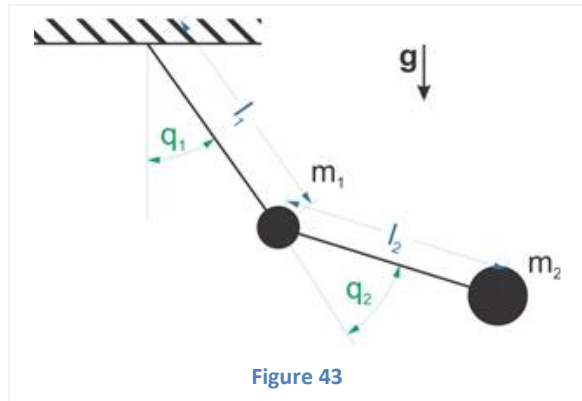


Figure 43

### Problem 41 (Understanding Dynamics)

(a) Which one of the following statements is **TRUE**? (Note: 'I' is a stationary point and coordinate system, 'C' is an arbitrary body-fixed point and coordinate system, and 'G' is the center of gravity.)

(1)  ${}_C \dot{\vec{L}}_G = {}_C \vec{M}_G$

(2)  ${}_I \dot{\vec{L}}_C = {}_I \vec{M}_C$

(3)  ${}_I \left( \dot{\vec{L}}_C \right) = {}_I \left( \vec{M}_C \right)$

(4)  ${}_C \left( \dot{\vec{L}}_I \right) = {}_C \left( \vec{M}_I \right)$

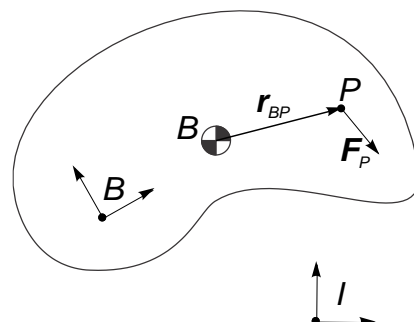
(b) Which one of the following statements does **NOT** correctly project the force  $\mathbf{F}_P$  into the generalized coordinate space? (Note: 'I' is a stationary coordinate system, 'B' is a point at the center of gravity of the body and also denotes a body-fixed coordinate system, and 'P' is the point at which the force acts.)

(1)  $({}_B \mathbf{J}_B^S)^T \cdot {}_B \mathbf{F}_P + ({}_B \mathbf{J}_B^R)^T \cdot {}_B \tilde{\mathbf{r}}_{BP} \cdot {}_B \mathbf{F}_P$

(2)  $({}_B \mathbf{J}_P^S)^T \cdot {}_B \mathbf{F}_P + ({}_B \mathbf{J}_P^R)^T \cdot {}_B \mathbf{r}_{BP} \times {}_B \mathbf{F}_P$

(3)  $({}_B \mathbf{J}_P^S)^T \cdot {}_B \mathbf{F}_P$

(4)  $({}_B \mathbf{J}_P^S)^T \cdot \mathbf{A}_{BI} \cdot {}_I \mathbf{F}_P$



- (c) The flyball governor shown in **Problem 27d** works the following way: When the axis is spinning with an angular rate of  $\dot{\varphi}$ , the balls (with mass  $m$ ) are experiencing a centrifugal force that makes them move outwards, increasing the angle  $\gamma$  at the linkage at the top (with length  $l$ ). The glider, which is coupled to the balls via the linkage, will then move upward and press against the spring until an equilibrium of forces is established. This creates a motion of the external lever arm which can, for example, be used to control a steam-valve. In the configuration shown, the angle of the axis is  $\varphi = 0$  and the angle of the linkage is  $\gamma = 30^\circ$ .

Which one of the following **partial** Mass Matrices describes the contribution of **one** ball to the dynamics of the flyball governor in the configuration shown in Figure 31? The generalized coordinates vector is  $\mathbf{q} = (\varphi, \gamma)^T$ .

$$\begin{aligned}
 (1) \mathbf{M}_{part} &= \begin{bmatrix} \frac{l^2 m}{4} & 0 \\ 0 & \frac{3l^2 m}{4} \end{bmatrix} & (2) \mathbf{M}_{part} &= \begin{bmatrix} \frac{l^2 m}{4} & \frac{\sqrt{3}lm}{2} \\ \frac{\sqrt{3}lm}{2} & l^2 m \end{bmatrix} \\
 (3) \mathbf{M}_{part} &= \begin{bmatrix} \frac{l^2 m}{4} & 0 \\ 0 & l^2 m \end{bmatrix} & (4) \mathbf{M}_{part} &= \begin{bmatrix} \frac{l^2 m}{4} & \frac{\sqrt{3}lm}{2} \\ \frac{\sqrt{3}lm}{2} & \frac{3l^2 m}{4} \end{bmatrix}
 \end{aligned}$$

## Exercise VII Collision and Contact

### Problem 42 (Event Detection in Matlab)

Read the Matlab documentation about event handling when solving ordinary differential equations: <http://de.mathworks.com/help/matlab/math/ode-event-location.html>

Event handling provides `ode45` with the necessary functionality to abort the integration before the time `t_end` is reached. To do event handling in Matlab, we use the function `odeset()` to generate an `options`-struct that contains a handle to an `events()` function. This `events()` function is implemented by the programmer on a problem specific basis and must return three numbers:

- A function residual `value`
- A termination flag `isterminal`
- A direction flag `direction`

If `direction` is `+1`, a positive zero crossing in the residual `value` is detected. If `direction` is `-1`, a negative zero crossing is detected, and if `direction` is `0`, zero-crossings in both directions are detected. If the flag `isterminal` is set to `1`, the integration will abort, otherwise it will continue after registering the event.

### Problem 43 (Equations of Motion for a System with Implicit Constraints)

A bar (with mass  $m$ , inertia  $\theta$ , and width  $2 \cdot d$ ) is resting on a unilateral, **frictionless** constraint that exerts a scalar constraint force  $\lambda$  at the point  $O$  ( $\mathbf{r}_{IO} = \mathbf{0}$ ). The position of the bar is defined by the generalized coordinate vector  $\mathbf{q} = (x, y, \varphi)^T$ , that corresponds to the position and orientation of the body's COG in inertial ( $I$ ) coordinates (Figure 44). You can use *Template\_43\_ImplicitConstraints.m* to help you do the following:

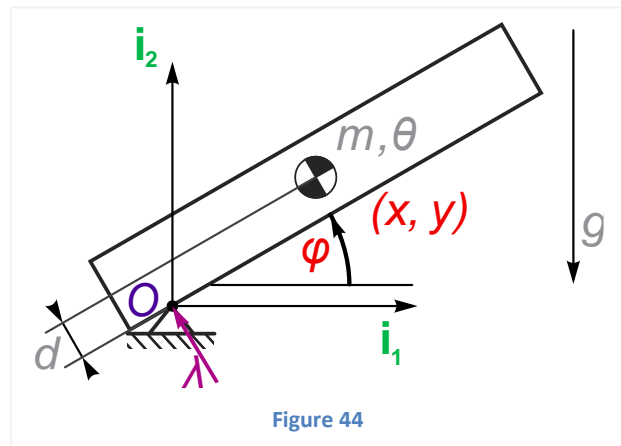


Figure 44

(a) State the mass matrix  $\mathbf{M}$ , and the overall equations of motion for the **unconstrained** system.

(b) Analytically compute the constraint violation  $c$ , the constraint Jacobian  $\mathbf{J}_\lambda$ , and the constraints bias acceleration  $\ddot{\sigma}^\lambda$  for the constrained system. Report the **complete** EOM for the **constrained** system including the constraint force  $\lambda$ .

(c) At the moment  $t_0$ , the bar is at rest ( $\dot{\mathbf{q}}(t_0) = \mathbf{0}$ ) and in the position  $\mathbf{q}(t_0) = (1m, 1m, 45^\circ)^T$ . Compute the acceleration  $\ddot{\mathbf{q}}(t_0)$  and the contact force  $\lambda$  for the parameter values  $m = 0.5\text{kg}$ ,  $\theta = 1\text{kgm}^2$ ,  $d = 0m$ ,  $g = 9.81\text{m/s}^2$ . They should be:  $(-2.4525, -7.3575, -2.4525)$  and  $1.742$ , respectively.

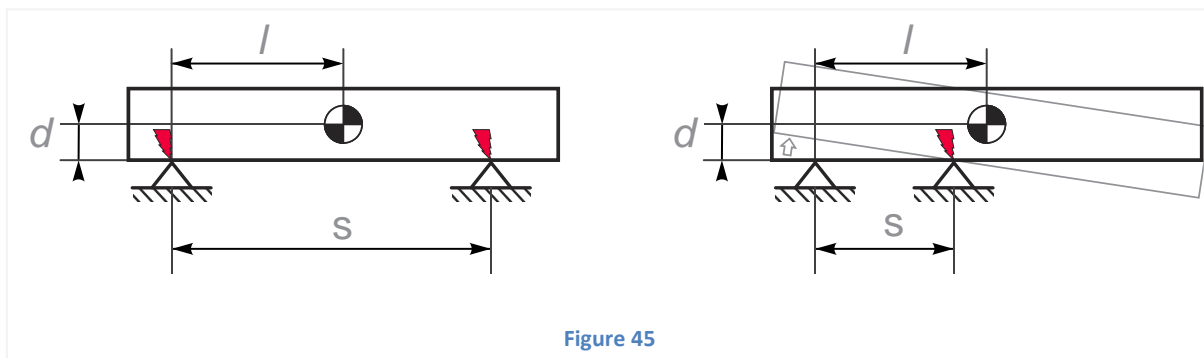


Figure 45

### Problem 44 (Implementing Event Detection)

📖 We want to simulate the system in **Problem 43** and additionally define an event function that triggers when the bar hits a second unilateral constraint which is at the same height as the first (see Figure 45).

Use the template *Template\_44\_EventDetection* to write a Matlab function that integrates the equation of motion of the constrained system until the bar hits the second constraint for  $s = 0.5m$ . The template will generate a plot which should look like in Figure 46.

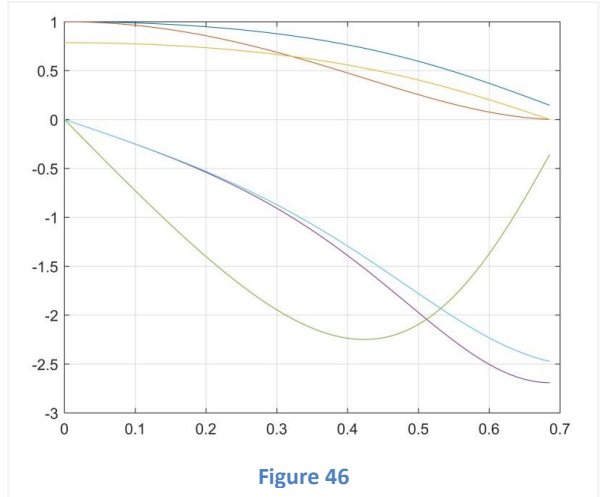


Figure 46

### Problem 45 (Collision of Falling Bar)

✂ At time  $t^-$  the falling bar of **Problem 43** hits a second **frictionless** constraint that is mounted at a distance of  $s$  from the first constraint (see Figure 44 and Figure 45). At the moment of impact, the generalized coordinates are given by  $\mathbf{q}^- = (l^-, d, 0^\circ)^T$  (while we do know  $l^-$  from **Problem 44**, we assume it is an unknown free parameter in this last part).

Compute the values of  $s$  (in relation to  $l^-$ ,  $d$ ,  $m$ , and  $\theta$ ) for which the first contact opens during impact (as shown in the scenario on the right in Figure 45).

To do so, define a suitable constraint space that assumes both contacts remain closed after the impact (as shown in the left picture), and compute the impulse  $\Delta$  delivered by the left contact as a function of  $s$ ,  $l^-$ ,  $m$ ,  $\theta$ , and  $\dot{\mathbf{c}}^-$ . Then figure out, for which  $s$  this impact becomes negative, assuming that the contact on the left was closed before the collision (i.e., that part of  $\dot{\mathbf{c}}^-$  was '0') and assuming the coefficient of restitution is zero.

Hints:

- The Jacobian for the new constraints is fairly simple for  $\mathbf{q}^- = (l^-, d, 0^\circ)^T$ .
- Since the second contact is closing, the associated part of  $\dot{\mathbf{c}}^-$  must be ' $<0$ '.

### Problem 46 (Robotic Arm. 3 – Event Detection)

📖 An inattentive co-worker (I believe the technical term would be 'jerk') left a wire hanging into the workspace of the robot from **Problem 37**. While performing the desired motion, the end-effector of the robot gets entangled in the wire.

This wire defines a unilateral, one-dimensional constraint. It has a length of 'wireLength' and is attached at the 'anchorPoint' which is given as a 3D vector.

- (a) In the provided template file *Template\_46\_RoboticArmCollision.m*, complete the event function `[value, isterminal, direction] = wireStretched(~, y_)` that is used to detect the moment when the wire is fully extended. This should happen at time  $t = 0.93313$ .

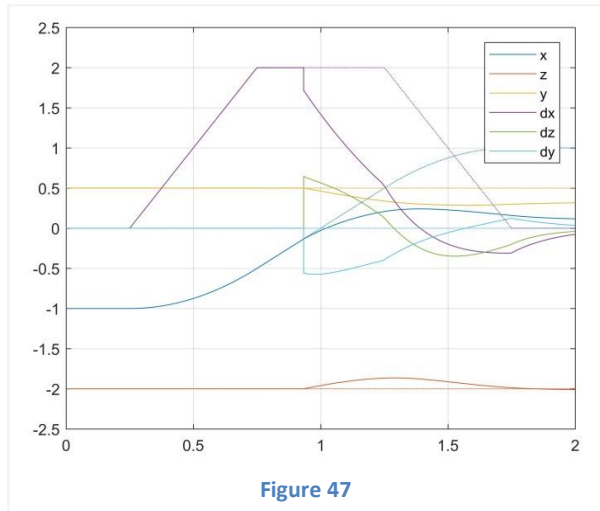


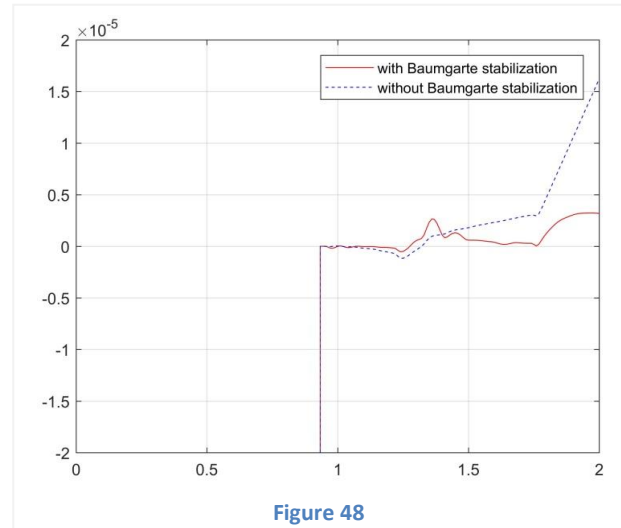
Figure 47

(b) We assume that the wire will not stretch at all. This means that at the moment it gets entangled, we have to compute a fully plastic collision. The function `[J_lambda, sigma_bar_lambda] = GetConstraintTerms()` is provided and computes the necessary components for the collision handling.

Implement the collision computation. Your output should look like in Figure 47. The values for `q_PLUS` and `d_q_PLUS`, should be:

- `q_PLUS`: +0.45214 -0.59389 -1.6775
- `d_q_PLUS`: +0.98849 +0.01024 -0.99379

(c) Can you implement a Baumgarte stabilization by amending the terms for `sigma_lambda` in the function `[J_lambda, sigma_lambda] = GetConstraintTerms()`? By running the problem separately with and without stabilization, you should see the resulting difference in the constraint violation, as shown in Figure 48: the solid red line represents the constraint violation with Baumgarte stabilization and the dashed blue line without it.



✂ Explain (that means: derive) the constraint terms `J_lambda` and `sigma_lambda`.

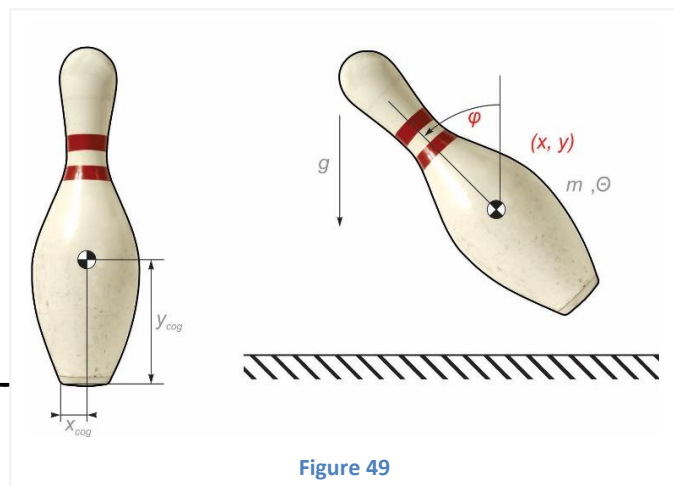
### Problem 47 (Bowling Pin)

✂ The COG of a bowling pin is located at the position  $(x_{cog}, y_{cog})$  measured from the lower left corner (see Figure 49). The position and orientation of the pin are defined by the generalized coordinate vector  $q = (x, y, \varphi)^T$  that corresponds to the position  $(x, y)$  of the COG in inertial coordinates and the rotation  $\varphi$  from vertical. The pin has a mass of  $m$  and an inertia of  $\theta$ .

The pin falls and hits the bowling lane (which is greased and hence frictionless) at time  $t^-$  with  $q^- = (x^-, y^-, \varphi^-)^T$  and  $\dot{q}^- = (\dot{x}^-, \dot{y}^-, \dot{\varphi}^-)^T$ . The angle  $\varphi$  is in a range such that contact happens at the lower left corner.

(a) Draw the constraint violation in the Figure 49. Compute the constraint violation  $c$  and the constraint Jacobian  $J_\lambda$  for the unilateral contact.

$$c = \underline{\hspace{10cm}}$$



$$J_\lambda =$$

(b) Assume a pre-impact velocity of the form  $\dot{q}^- = (\dot{x}^-, \dot{y}^-, 0)^T$  (with  $\dot{y}^-$  being negative). For which angle  $\hat{\varphi}^-$ , does the collision impulse  $\Lambda$  become maximal? Argue **physically** (rather than using math).



$$\hat{\phi}^- = \underline{\hspace{2cm}}$$

(c) Why is the answer to the previous question different when the bowling pin is additionally spinning with a pre-impact rotational velocity  $\dot{\phi}^-$ ? Again, argue **physically** (rather than using math).

(d) Compute the collision impulse  $\Lambda$  and report the **maximal** value  $\Lambda_{\max}$  it can have for a pre-impact velocity of  $\dot{\mathbf{q}}^- = (\dot{x}^-, \dot{y}^-, 0)^T$  (with  $\dot{y}^-$  being negative).

$$\Lambda_{\max} = \underline{\hspace{2cm}}$$