ASEE CoED Journal
Published 30 Dec 2022

Use of Open-source Software in Mechatronics and Robotics Engineering Education – Part II: Controller Implementation

Nima Lotfi[1*], Dave Auslander[2], Luis A. Rodriguez[3], Kenechukwu C. Mbanisi[4] and Carlotta A. Berry[5]

[1]Mechanical and Mechatronics Engineering Department, Southern Illinois University Edwardsville, Edwardsville, IL, United States

[2]Mechanical Engineering, University of California Berkeley, Berkeley, CA, United States

[3]Mechanical Engineering, Milwaukee School of Engineering, Milwaukee, WI, United States

[4]Robotics Engineering, Worcester Polytechnic Institute, Worcester, MA, United States

[5]Electrical and Computer Engineering, Rose-Hulman Institute of Technology, Terre Haute, IN, United States

RESEARCH

Abstract

This paper is the second part of a two-part study on promoting the use of Open-Source Software (OSS) in Mechatronics and Robotics Engineering (MRE) education. Part I demonstrated the capabilities and limitations of several popular OSS, namely, Python, Java, Modelica, and GNU Octave, in model simulation and analysis of dynamic systems, through a DC motor example. The DC motor was chosen as a representative of a large class of dynamic systems described by linear differential equations. The perceptions of MRE community members about the OSS and their applications, gathered through an online survey, were also presented in Part I. In this paper, another fundamental pillar of MRE systems development, i.e., controller implementation, is considered. To this end, the OSS above, along with Gazebo, are used to simulate the closed-loop trajectory tracking performance of a 2-DOF robot manipulator, controlled by a PID controller. Robot manipulators represent a broader category of dynamic systems, which are described by nonlinear differential equations. Furthermore, PID controllers are one of the most versatile closed-loop control methodologies which have been established as the industry standard. Showcasing the implementation of this important category of controllers through OSS can promote their use in a wide range of MRE problems and projects. This paper also provides an overview of the potentials, limitations, and challenges regarding the use of each of the above OSS in solving the aforementioned problems. The OSS are increasingly being adapted as industry standard; furthermore, their numerous benefits pose them as a viable option to complement traditional higher education courses along with facilitating online and remote education. Therefore, this two-part paper and the various problems showcased and solved therein aim to pave the way towards further utilization of the OSS in MRE higher education, reaping the wide range of their benefits, and preparing the students for the future workforce. Full scripts of the codes summarized and discussed in this paper, along with Matlab scripts included to enable comparison, are made freely available on the Github repository of this paper.

Keywords: Controls, Engineering Pedagogy, Mechatronics, Open Source, Robotics, Simulation-Based Learning

OPEN ACCESS

Volume *13*

Issue *1*

***Corresponding author** *nlotfiy@siue.edu*

Submitted *17 Jan 2021*

Accepted *20 Sep 2021*

Citation
Lotfi N., Auslander D., A. Rodriguez L., C. Mbanisi K. and A. Berry C. "Use of Open-source Software in Mechatronics and Robotics Engineering Education – Part II: Controller Implementation," *Computers in Education Journal*, vol. 13, no. 1, 2022. doi:

1 Introduction

The field of Mechatronics and Robotics Engineering (MRE) has experienced an organic and rapid growth in the past few decades, mainly thanks to all the technological advancements in control systems, electronics, computers, and connectivity and increased demand for robotics and automation in industry. This ongoing progress has increasingly resulted in the development of new job roles such as mechatronics or robotics engineers and specialists. To prepare the next generation of engineers to fulfill these responsibilities, various stand-alone courses have been offered in Mechanical Engineering, Electrical Engineering, and Computer Science departments. In recent years, there has been a transition in higher educational institutions to develop minors and majors in Mechatronics and/or Robotics Engineering to meet the industry demands. Authors in [1] share their experiences and the lessons they learnt during 10 years since they started one of the first Robotics Engineering programs in the United States.

An essential part of any MRE educational program should be to provide its students with an interdisciplinary knowledge of mechanical, electrical, computer, software, and systems engineering. Robotics courses have traditionally provided an opportunity to educate the students with such an interdisciplinary knowledge. The entertaining nature of the robots has further established them as attractive learning and motivational platforms for K-12 and freshman students [2–4]. In the past few decades, numerous efforts have been undertaken to develop different robotics courses, some of which are reported in [5–9]. Lessons and experiences gained through these valuable efforts have also been published in the literature to provide a roadmap for the community members who plan to offer robotics courses or to develop new ones [10–14]. Although some of these courses use a commercial robot platform such as Lego Robotics [9, 14], VEX Robotics [15], Turtlebot [16, 17], etc., others employ a custom-built robot platform using open-source hardware such as Arduino [18–20] and Raspberry Pi [15, 21]. The Open-source Software (OSS) in robotics courses have mainly been in the form of a software for microcontroller programming or hardware interface, such as C++ for Arduino, Python for Raspberry Pi, and Robot Operating System (ROS) as overall robot software framework [17, 22]. In this work, as the second part of a two-part paper, the OSS such as Python, Java, Modelica, GNU Octave, and Gazebo are used to implement a PID controller for 2-DOF robot arm simulations. While this robot arm is simple to be implemented in an undergraduate-level course, it is complex enough to expose the students to more advanced topics of simulation and control design for nonlinear dynamic systems.

Control Systems have been the cornerstone of many of the technological advancements since early 20th century. They play a fundamental role in industrial automation, transportation, energy industry and other emerging areas such as robotics, manufacturing, IoT applications, and cyber-physical systems. Therefore, majority of related engineering disciplines include several control courses in their educational curricula. The MRE field, in particular, relies heavily on control systems as controls can be thought of as joints linking various disciplines involved in the system. Consequently, MRE students need to master the design and implementation of control systems to be successful in their careers and to be able to design smart and autonomous systems and processes that will improve human life and welfare.

Commercial products such as Matlab provide extensive and convenient tools for the design and implementation of control systems. Although Matlab and its related toolboxes are commonly available to students in higher education institutions, the students typically lose access to the complete suite of Matlab products once they graduate. Moreover, a lot of industries are migrating towards using the OSS, due to their numerous advantages such as lower ownership costs, higher flexibility and customizability, improved reliability and accessibility, and wider community support. Application of the OSS in developing and implementing control algorithms can further expose the students to the development details of control systems, an aspect that is typically overlooked when using advanced tools such as Matlab and its toolboxes. Therefore, familiarizing the students with the application of the OSS for control implementation can equip them with the skills they would need in the future.

As mentioned earlier, the 2-DOF robot manipulator is considered in this work as a showcase to

demonstrate the application of the OSS in control implementation and closed-loop simulation. The dynamics of the robot manipulators are governed by Euler-Lagrange equations, which result in nonlinear differential equations. Therefore, the example of a robot manipulator is a representative of a larger class of dynamic system. The robot manipulator is assumed to be controlled using a discrete-time PID controller to follow a pre-defined reference trajectory. The PID controllers are extensively used in various applications and contribute to majority of industry controllers. Although most MRE students get exposed to PID controllers and their design, they seldom get to implement and tune a PID controller from the ground up. Implementation of PID controllers using the OSS can familiarize the students with the entire process of the design and implementation. Furthermore, they would also be prepared for practical implementation of PID controllers as the OSS can be used onboard MRE hardware. The ultimate goal of this work is to promote the use of the OSS in MRE education and help with their widespread adoption. The OSS in this work can be introduced/used in a wide range of MRE-related courses, from freshman introductory to senior and graduate-level advanced courses and even, senior design projects, offered in Mechanical Engineering, Electrical Engineering, Mechatronics and Robotics Engineering, and Computer Science programs. Some of such courses include: introduction to computing/programming, dynamic system modeling, introductory and advanced courses on controls, mechatronics, and robotics, etc.

This paper is organized as follows: Section 2 outlines the dynamics of the 2-DOF robot manipulator including the governing Euler-Lagrange equations and the parameters used in simulations. Section 3 details the controller implementation and the desired closed-loop behavior. Furthermore, important code snippets demonstrating the controller implementation using each of the OSS are given in this section. Finally, Section 4 provides an overview of the capabilities, limitations, and the potentials of each of the OSS to be used in MRE education.

2 Robot Manipulator Case Study

The robot, considered in this work, is a 2-DOF planar arm, as can be seen in Figure 1 below.

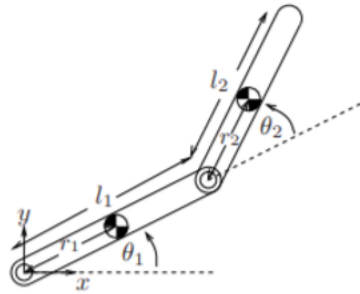

Figure 1. Two-link planar robot arm schematics, as illustrated in [23] Fig. 4.4

The Euler-Lagrange equations describing the dynamics of this robot can be written as

$$M(\theta)\,\ddot{\theta} + C(\theta,\dot{\theta})\,\dot{\theta} \; + g(\theta) = \tau \tag{1}$$

where $\theta = [\theta_1\,,\theta_2]^T$ is the vector of joint angles and the inertia matrix, **M**, Christoffel matrix, **C**, and the gravity vector, **g**, are

$$\begin{aligned} M\left(\theta\right) &= \begin{bmatrix} I_1 + I_2 + m_1{r_1}^2 + m_2\left({L_1}^2 + {r_2}^2\right) + 2m_2L_1r_2\cos\left(\theta_2\right) & I_2 + m_2{r_2}^2 + m_2L_1r_2\cos\left(\theta_2\right) \\ I_2 + m_2{r_2}^2 + m_2L_1r_2\cos\left(\theta_2\right) & I_2 + m_2{r_2}^2 \end{bmatrix} \\ C\left(\theta,\dot{\theta}\right) &= \begin{bmatrix} -m_2L_1r_2sin(\theta_2)\dot{\theta}_2 + b_1 & -m_2L_1r_2sin(\theta_2)(\dot{\theta}_1 + \dot{\theta}_2) \\ m_2L_1r_2sin(\theta_2)\dot{\theta}_1 & b_2 \end{bmatrix} \\ g\left(\theta\right) &= \begin{bmatrix} (m_1r_1 + m_2L_1)gcos(\theta_1) \; + \; m_2r_2gcos(\theta_1 + \theta_2) \\ m_2r_2gcos(\theta_1 + \theta_2) \end{bmatrix} \end{aligned} \tag{2}$$

Table 1. Typical DC motor parameters used in simulations.

Parameter	L_1	L_2	r_1	r_2	m_1	m_2	g
Value	0.25 m	0.25 m	0.125 m	0.125 m	0.5 kg	0.5 kg	9.81 m/s^2
Parameter	I_1		I_2	b_1		b_2	
Value	$m_1L_1{}^2/12$		$m2L_2{}^2/12$	10^{-1} Nm-s/rad		10^{-1} Nm-s/rad	

The robot links are assumed as slender rods and the parameters chosen for the simulations are summarized in Table 1 .

Using Denavit-Hartenberg convention, the forward kinematic equations describing the Cartesian coordinates of the end-effector as functions of individual joint angles can be written as

$$\begin{aligned} x_e &= L_1 cos\left(\theta_1\right) + L_2 cos\left(\theta_1 + \theta_2\right) \\ y_e &= L_1 sin\left(\theta_1\right) + L_2 sin\left(\theta_1 + \theta_2\right) \end{aligned} \tag{3}$$

Finally, using a geometric approach, the inverse kinematic equations for this robot will be

$$\begin{aligned} x_e &= L_1 cos\left(\theta_1\right) + L_2 cos\left(\theta_1 + \theta_2\right) \\ y_e &= L_1 sin\left(\theta_1\right) + L_2 sin\left(\theta_1 + \theta_2\right) \end{aligned} \tag{4}$$

It should be noted that Equation 4 corresponds to the elbow-down solution of the inverse kinematic problem. The elbow-up solution can be obtained using

$$\begin{aligned} \theta_2 &= \text{atan2}\left(-\sqrt{1-D^2}, D\right) \\ \theta_1 &= \text{atan2}\left(y_e, x_e\right) + \text{atan2}\left(-L_2 sin\left(\theta_2\right), L_1 + L_2 cos\left(\theta_2\right)\right) \end{aligned} \tag{5}$$

where the variable D is defined as

$$D = \frac{x_e^2 + y_e^2 - L_1^2 - L_2^2}{2L_1L_2}$$

and can be used to investigate the reachability of the given end-effector coordinates. The atan2 function in Equation 4 and Equation 5 is used to account for the quadrant. Note that the notation used for this function, i.e. atan2(y, x) is to comply with numerical software packages, which is different than the notation used in some Robotics textbooks [24].

3 Closed-loop Simulation of the Robot Arm

In this section, the OSS are used to simulate the closed-loop performance of the 2-DOF robot arm, introduced in Section 2. The robot is assumed to start from an initial state of$\theta(0) = [\theta 1(0),\ \theta 2(0),\ d\theta 1/dt(0),\ d\theta 2\ /dt(0)]T = [\pi/2,\ 0,\ 0,\ 0]$. Forward kinematics in Equation 3 is then used to calculate the cartesian location of the end-effector. Furthermore, it is assumed that the robot end-effector is to track a reference trajectory. Figure 2 shows the implementation flowchart of the closed-loop robot simulations used to track the reference trajectory. This structure, with some minor variations, is followed in all of the introduced software.

The implemented algorithm begins with an initialization stage where memory allocation occurs and the robot properties, solver options, desired path characteristics, and controller properties are specified. Next, the algorithm executes a “for” loop structure that spans the entire simulation duration. The desired robot trajectory is then generated inside the loop such that the end-effector follows a linear path from its initial position until it reaches a previously-defined circle, at which point it dwells and stays fixed for a certain period of time, and then continues to track the circle. Once the desired position of the end-effector on the reference trajectory is determined, the required joint angles are calculated using the robot inverse kinematics in Equation 4 . For simplicity, the actuator dynamics are ignored and joint torques (assumed to be bounded between -10 and 10

N.m) are considered as the control inputs. A discrete PID controller is then used to calculate the joint torques needed to track the desired joint angles and consequently, the desired end-effector position. Once the actuating torques have been obtained, the robot dynamics are simulated with an ODE solver and the robot joint torques, state history and simulation times are stored.

The code structure here closely follows how a discrete controller is implemented in practice. Furthermore, PID controllers are extensively used in industry and academia. Therefore, familiarity with PID controller implementation using various OSS can be extremely beneficial for MRE students and professionals.

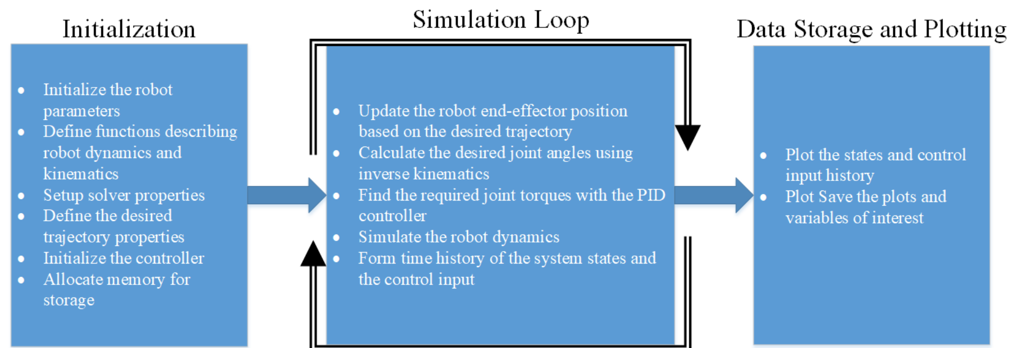

Figure 2. The flowchart of the closed-loop robot simulations.

3.1 Python

Considering the nonlinear and coupled nature of the robot dynamics in Equation 1 , they are solved directly using the solve_ivp command from the scipy.integrate library of Python. One possible way to use this command in simulating the robot dynamics is:

```
1  for i~in range(len(time)-1):
2      t = time[i]
3      tNext = t + dt
4      state0 = np.squeeze(state[:, i])
5      sol = solve_ivp(model, [t, tNext], state0, dense_output= True
           , vectorized= True, args= (exp_dim(tau[:, i]),))
6      state_new = sol.sol(tNext)
7      state = np.column_stack([state, state_new])
8      th_new = exp_dim(np.array([state[0, i+1], state[1, i+1]]))
9      th = np.column_stack([th, th_new])
10     thdot_new = exp_dim(np.array([state[2, i+1], state[3, i+1]]))
11     thdot = np.column_stack([thdot, thdot_new])
```

Code Snippet 1. Python for Equation 1

It should be noted that the simulation of the robot dynamics is performed iteratively within a for loop over the entire duration of the simulation, i.e. time seconds, to facilitate the controller implementation. Therefore, t and tNext are the beginning and end of one iteration of the simulation with a step time of dt seconds. The variable state is used to denote the entire state vector of the robot, i.e.$[\theta 1\ ,\ \theta 2\ ,\ d\theta 1/dt\ ,\ d\theta 2\ /dt\ ]^T$. It is initialized as below before the beginning of the loop:

```
1  th_init = np.array([[np.pi/2],[0]], dtype = 'float')
2  thdot_init = np.array([[0],[0]], dtype = 'float')
3  state0 = np.concatenate((th_init,thdot_init))
4  state = state0
```

Code Snippet 2.

The variables θ (denoted by th) and $d\theta/dt$ (denoted by thdot) are initialized individually using the np.array command and then concatenated with the np.concatenate command to form the initial state vector. It should be noted that the initial state vector should be updated within each iteration of the loop and hence, the command state0 = np.squeeze(state[:,i]. The reason for using the squeeze command to generate the initial state vector for integration at each step is the syntax of the solve_ivp command as it can only accept 1-D arrays.

The callable function model includes the Euler Lagrange equations in Equation 1 represented in a state-space form:

```
def model(t, x, u):
    x1, x2, x3, x4 = x
    th = np.array([x1, x2], dtype = 'float')
    thdot = np.array([x3, x4], dtype = 'float')
    M_inv = np.linalg.inv(M_mat(th))
    thddot = np.dot(M_inv,(u-np.dot(C_mat(th, thdot),thdot)-g_mat
        (th)))
    dxdt = [x3, x4, thddot[0,:], thddot[1,:] ]
    return dxdt
```

Code Snippet 3.

where M_mat, C_mat, and g_mat are the system matrices in Eq. , defined in separate functions. As an example, the Christoffel matrix C_mat is defined as:

```
def C_mat(th, thdot):
    c11 = -m2*L1*r2*np.sin(th[1])*thdot[1]+b1
    c12 = -m2*L1*r2*np.sin(th[1])*(thdot[0]+thdot[1])
    c21 = m2*L1*r2*np.sin(th[1])*thdot[0]
    c22 = b2
    C = np.array([[c11, c12],[c21, c22]], dtype = 'float')
    return C
```

Code Snippet 4.

One of the main challenges for MRE professionals, especially those more familiar with Matlab, when using Python for dynamic system simulations is dealing with indices, array indexing, and array generation within loops. Python lists and array start from an index of 0 as opposed to Matlab which starts from 1. Another challenge is that Python arrays typically lose a dimension when indexing. This will be observed when choosing the ith column of the input vector tau to give as the input to the system at each iteration. The vector tau will have a dimension (size) of $2 \times$ (i+1) at the ith iteration. Therefore, the expression tau[:, i] should have a dimension of 2×1, whereas upon closer investigation, it can be seen that this expression is a 1-D variable with a size of (2,). This discrepancy can be problematic in subsequent vector and matrix operations. Therefore, in this code, the custom function exp_dim, defined as below, is used to expand the dimensions of the array.

```
def exp_dim(var):
    out = np.expand_dims(var, axis = 1)
    return out
```

Code Snippet 5.

Another challenge with Python is appending columns to a matrix within each loop iteration. Despite being very straightforward in Matlab, this functionality in Python requires commands such as np.column_stack. Once the state variables are updated at each iteration, the Cartesian position of the end-effector can be obtained using the forward kinematics (FK_fun)

```
xe_new, ye_new = FK_fun(th[0, i+1], th[1, i+1])
xe_act += [xe_new]
ye_act += [ye_new]
```

Code Snippet 6.

Note that in this code snippet, the expressions xe_act += [xe_new] and ye_act += [ye_new] are used as a way for creating a list to store the actual end-effector position, another less-straightforward feature of Python. To be able to use these syntaxes, the lists should be initialized as xe_act = [xe0] and ye_act = [ye0] where xe0 and ye0 are the initial coordinates of the robot end-effector.

Finally, as mentioned earlier, the control input needed to ensure trajectory tracking is obtained using a PID controller. This controller is implemented within a for loop as below:

```
e = exp_dim(thd[:, i+1]) - exp_dim(th[:, i+1])
E += e*dt
edot = (e - old_e)/dt
tau_new = np.dot(Kp,e) + np.dot(Kd,edot) + np.dot(Ki,E)
tau_new [tau_new > 10] = tau_max
tau_new [tau_new < -10] = tau_min
tau = np.column_stack([tau, tau_new])
old_e = e
```

Code Snippet 7.

where e is the joint angle tracking error, thd is the desired joint angles, E is an approximation of the error integral, edot is an approximation of the error derivative, tau is the control input, and tau_max and tau_min are the maximum (+10 N.m) and minimum (-10 N.m) bounds on the control input, respectively. The controller parameters, tuned to track the reference trajectory, for the 2-DOF robot considered in this work are:

$$K_p = \begin{bmatrix} 20 & 0 \\ 0 & 20 \end{bmatrix} \quad K_d = \begin{bmatrix} 2 & 0 \\ 0 & 0.1 \end{bmatrix} \quad K_i = \begin{bmatrix} 40 & 0 \\ 0 & 40 \end{bmatrix} \tag{6}$$

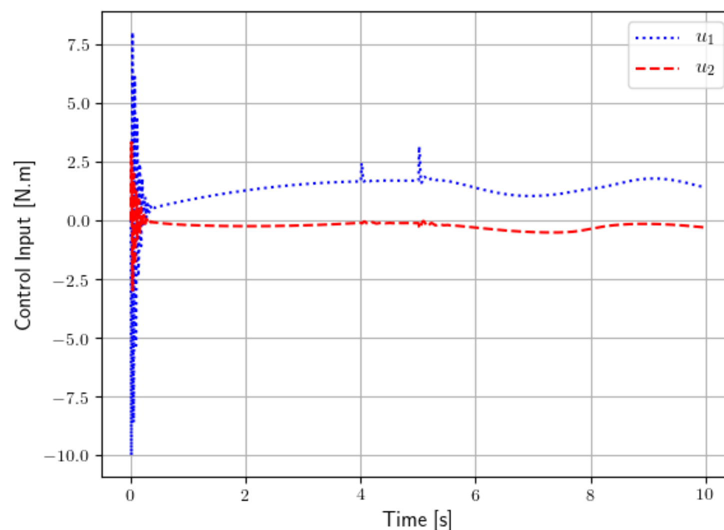

Figure 3. Desired versus actual joint angles for the 2-DOFrobot.

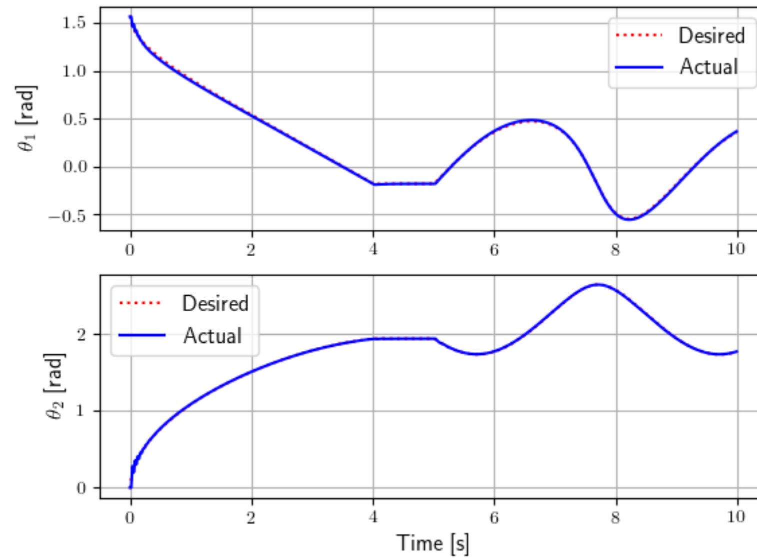

Figure 4. Control input for trajectory tracking of the 2-DOF robot.

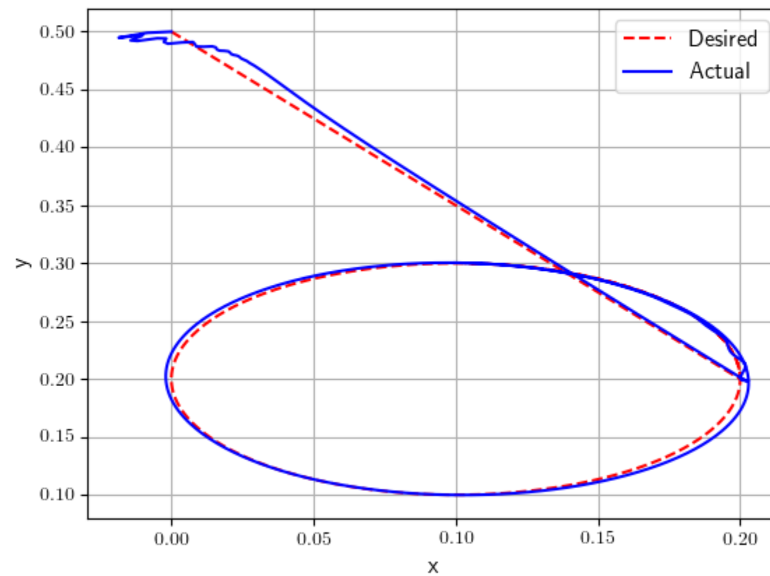

Figure 5. Desired versus actual trajectory for the 2-DOF robot.

The simulation results of the controller implementation can be seen in Figure 3, Figure 4 and Figure 5 . The red dotted lines represent desired signals whereas the blue lines are the actual values of the signals. The initial transient is because of the instantaneous, rather than gradual, change in the velocity at the start and end of the profile. The errors along the paths are caused by a combination of PID gain tuning and how fast the profile is traversed.

3.2 GNU Octave

In Octave, the for loop needed for the simulations is implemented as below:

```
for i~= 1:numel(time)-1
    % Current time and state
    t = time(i); x = state(i,:);
    tNext = t + dt; %Next simulation time or end of interval
    % Define Aliases
    th = x(1:2);
    % Define desired trajectory
    if (t <= tdwell)
      xe_d(i) = (xCirc0 - xe0)/tdwell*t + xe0;
      ye_d(i) = (yCirc0 - ye0)/tdwell*t + ye0;
    elseif (t <= tcircle) % Dwell phase
        xe_d(i) = xCirc0; ye_d(i) = yCirc0;
    else % start generating circle trajectory
        xe_d(i) = xc + r*cos(omega*(t-tcircle)); ye_d(i) = yc + r
            *sin(omega*(t- tcircle));
    end
    % Use Elbow Down Solution, define joint trajectories
    elbow = 1; % Elbow down Sol
    thd(i,:) = invK(RR,xe_d(i), ye_d(i),elbow);
    % Find the control manipulation for joint 1 and joint 2
    tau1 = theta1_Cntrl.PIDStep(th(1), thd(i,1));
    tau2 = theta2_Cntrl.PIDStep(th(2), thd(i,2));
    torque = [tau1;tau2]; % Apply torque to robot
    [~,state_new]= ode45(@(t,x)RR.model(t,x,torque),[t tNext],x);
    % save time, state, and robot joint torques
    torques(i,:)   = torque;
    time(i+1) = tNext;
    state(i+1,:) = state_new(end,1:4);
    %save the state at the end of the simulation
end
```

Code Snippet 8.

The variable state is used to denote the entire state vector of the robot, i.e. $[\theta, d\theta/dt]^T$ where θ= $[\theta_1, \theta_2]^T$ and $d\theta/dt = [d\theta_1/dt, d\theta_2/dt]^T$ for all simulation times. The variables θ, θ_d, and $d\theta/dt$ are denoted by th, thd, and thdot, respectively.

The instance method model from the user-defined TwoLink object named RR implements the Euler Lagrange equations in Equation 1 represented in the state-space form. The model syntax is:

```
function xdot = model(obj,t,x,u)
   % Alias
    th = x(1:2); thdot = x(3:4); torque = u;
   % Inertia and Christoffel Mat
    M = M_mat(obj,th);   C = C_mat(obj,th,thdot);
    % Gravity terms
    G = g_mat(obj,th);
    % Calculate the generalized accelerations
    thddot = inv(M) * (torque - C * thdot - G);
    % Generate the change in the state vector
    xdot = [thdot;thddot];
end
```

Code Snippet 9.

where obj refers to an instance of the Twolink class, M_mat, C_mat, and g_mat are the system

matrices in Eq. , defined in separate functions. As an example, the Christoffel matrix C_mat is defined as:

```
function C = C_mat(obj,th,thdot)
    % Calculate the Christoffel matrix
    k = obj.bet*sin(th(2));
    C11 = -k*thdot(2)  + obj.b1;  C12 = -k*(thdot(1) + thdot(2));
    C21 =  k*thdot(1);            C22 =  obj.b2;
    C = [C11 C12;C21 C22];
end
```

Code Snippet 10.

Finally, as previously mentioned, the control input needed to ensure trajectory tracking is obtained using a PID controller on each of the robot joints as below:

```
    % Find the control manipulation for joint 1 and joint 2
    tau1 = theta1_Cntrl.PIDStep(th(1), thd(i,1));
    tau2 = theta2_Cntrl.PIDStep(th(2), thd(i,2));
```

Code Snippet 11.

where the PIDController objects theta1_Cntrl and theta2_Cntrl evaluate the instance method PIDStep to compute the torque needed at the start of the simulation interval to ensure good tracking. The PIDStep method is implemented as:

```
function u~= PIDStep(obj,procVar, setPoint)
    % Calculate error, error integral, and error derivative
    obj.err   = setPoint - procVar;
    Err       = obj.err * obj.dt;
    errDot = (obj.err - obj.old_err) / obj.dt;
    % Compute P, I~and D contributions
    obj.pTerm = obj.kp * obj.err;
    obj.iTerm = obj.iTerm0 + obj.ki*Err;
    obj.dTerm = obj.kd * errDot;
    % Control Law
    u~= obj.pTerm + obj.iTerm + obj.dTerm;
    % Saturate the controller output value if exceeds limits
    if((u > obj.uMax) || (u < obj.uMin))
        if (u > obj.uMax), u~= obj.uMax;
        else, u~= obj.uMin; end %(uTrial < obj.uMin)
    end
    % update the integrator initial value
    obj.iTerm0 = obj.iTerm;
    obj.old_err = obj.err;  % Update previous error value
end
```

Code Snippet 12.

where obj is an instance of the PIDController class, procVar is the process variable, setPoint is the desired set point, dt is the sample time, err is the tracking error, Err is an approximation of the error integral, errDot is an approximation of the error derivative, u is the control input and uMin and uMax are the lower and upper bounds on the control input. The controller input bounds along with the controller parameters are chosen as before. In the Octave implementation, an object-oriented programming approach was implemented to keep the main program simple and to make the program modular. The simulation results are similar to the results presented in Subsection 3.1.

3.3 Modelica

3.3.1 Equation Mode

The equations for the two-link robot are in matrix form so this problem takes advantage of Modelica's matrix facilities. The state variables are the two joint angles and the two joint angular velocities which are defined as:

```
SI.Angle[2] theta(start={90*Constants.D2R, -90*Constants.D2R});
SI.AngularVelocity[2] omega(start = {0.0, 0.0});
```

Code Snippet 13.

where D2R is the conversion factor from degrees to radians. As before, the start values are the system initial conditions. The M, C and G matrices are defined as:

```
Real[2, 2] M "mass matrix";
Real[2, 2] C "Christoffel matrix";
Real[2] G "Gravity term";
```

Code Snippet 14.

The equation section includes the basic matrix equations and the term-by-term computation of the elements of the matrices (these depend on the angles and angular velocities so cannot be treated as constants)

```
M * der(omega) + C * omega + G = tau;
der(theta) = omega;
// Mass matrix
M[1, 1] = alpha + 2.0 * beta * cos(theta[2]);
M[1, 2] = delta + beta * cos(theta[2]);
M[2, 1] = M[1, 2];
M[2, 2] = delta;
// Christoffel matrix
C[1, 1] = (-beta * sin(theta[2]) * omega[2]) + b1;
C[1, 2] = -beta * sin(theta[2]) * (omega[1] + omega[2]);
C[2, 1] = beta * sin(theta[2]) * omega[1];
C[2, 2] = b2;
// Gravity
G[1] = (m1 * r1 + m2 * L1) * g * cos(theta[1]) + m2 *
r2 * g * cos(theta[1] + theta[2]);
G[2] = m2 * r2 * g * cos(theta[1] + theta[2]);
```

Code Snippet 15.

Note that the first equation makes very clear that these are equations and not computing statements.

Two different methods of implementing feedback control are used in the Modelica: one case used an external C function for the PID control and the other used an internal Modelica class. All the feedback control was implemented as discrete-time control corresponding to typical computer-based control.

The control in this model was implemented using an external C language module. This shows how straightforward it is to interface C code with Modelica models.

To connect to a function in C, a function declaration is made in the first (definitions) section of the Modelica code:

```
function PIDCinit
  input Integer loopID;
  input Real dt, kp, ki, kd, loLim, hiLim;
  output Integer ip "Index for this control loop -- should be
     same as loopID input";
  external "C" ip = PIDCinit(loopID, dt, kp, ki, kd, loLim, hiLim
     );
  annotation(
Include = "#include \"c:\\Users\\x\\Documents\\AAAA-Stuff\\AAAA-
    Working\\Modelica Projects\\TwoLinkRobot\\PID_C.c\"");
end PIDCinit;
```

Code Snippet 16.

and

```
function PIDCstep
  input Integer ip;
  input Real val, setpoint "process value and setpoint";
  output Real mVal "Actuation (manipulated) variable output";
  external "C" mVal = PIDCstep(ip, val, setpoint);
end PIDCstep;
```

Code Snippet 17.

for the initialization function and the function that operates each sample time. The downside of this facility is that the link to the C-file must be done using a fully qualified path, making the program no longer portable.

The control function can be called explicitly in the 'algorithm' section of the Modelica program. This section allows for algorithmic statements (that is, ordinary computing statements, conditionals, etc.). In this case, a sample-time, discrete control is setup by using a 'when' loop with sampling algorithm:

```
algorithm
when sample(0.0, dt) then
     ...
     cmd1 := PIDCstep(loopID1, theta[1], theta1Set);
     cmd2 := PIDCstep(loopID2, theta[2], theta2Set);
end when;
```

Code Snippet 18.

The use of := for these algorithmic statements distinguishes them from equation statements which use the = sign.

As mentioned earlier, the path for the robot end-effector to take is broken into three parts:

1. move from the initial robot position (normally straight up) to the beginning of the "production" path,
2. hold (dwell) briefly at that position, and
3. follow a circular path.

The code to do this is in the algorithm section where the three dots (…) are, above. The inverse kinematics is coded, also as an algorithm, in a separate function, InvKin().

The path following code is:

```
1  // First do the supervisory control (compute path setpoints)
2  tTraj := time - tSetup;
3  if time < (tSetup - tHold) then
4    // Move to start of circle
5    xTraj := (time / ((tSetup - tHold) + 1.0e-6)) * (x0 - xe0) +
       xe0;
6     yTraj := (time / ((tSetup - tHold) + 1.0e-6)) * (y0 -ye0) +
        ye0;
7     (theta1Set, theta2Set) := InvKin(xTraj, yTraj, L1, L2);
8     omegaTraj := 0.0;
9  elseif time < tSetup then
10   // Hold position (dwell)
11   (theta1Set, theta2Set) := InvKin(x0, y0, L1, L2);
12    xTraj := x0;
13    yTraj := y0;
14    omegaTraj := 0.0;
15 else
16   // Draw circle
17   omegaTraj := omegaTraj0;
18   xTraj := radTraj * cos(omegaTraj * tTraj) + xCenter;
19   yTraj := radTraj * sin(omegaTraj * tTraj) + yCenter;     (
       theta1Set, theta2Set) := InvKin(xTraj, yTraj, L1, L2);
20 end if;
```

Code Snippet 19.

This simulation results are identical to the results presented in Subsection 3.1.

3.3.2 Graphical Modeling Mode

The double pendulum/two-link robot is made up of revolute joints and rigid body objects. Although this problem is posed as a two-dimensional problem (the equation-mode solutions use the two-dimensional solution), this model is actually a full three-dimensional model. By fixing the first revolute joint to a mechanical ground, it can only move in two-dimensions but, if instead it were attached to a moving object, such as a turntable, the three-dimensional dynamics would be fully accounted for. The model for the two-link robot is shown in Figure 6 .

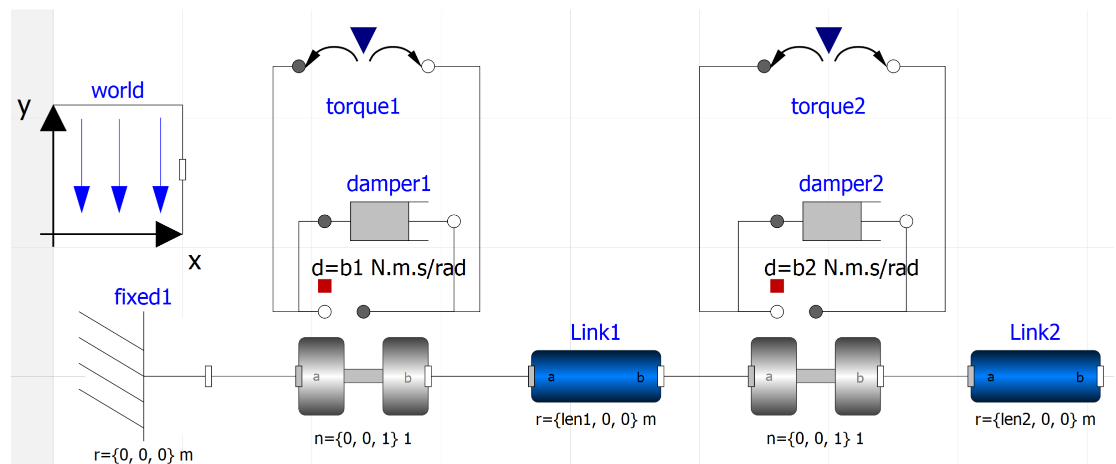

Figure 6. The 2-DOF robot (double pendulum) model in Modelica.

The initial condition for the arm is pointing straight up, i.e. joint 1 at 90° and joint 2 at 0°, as can be seen in Figure 7 .

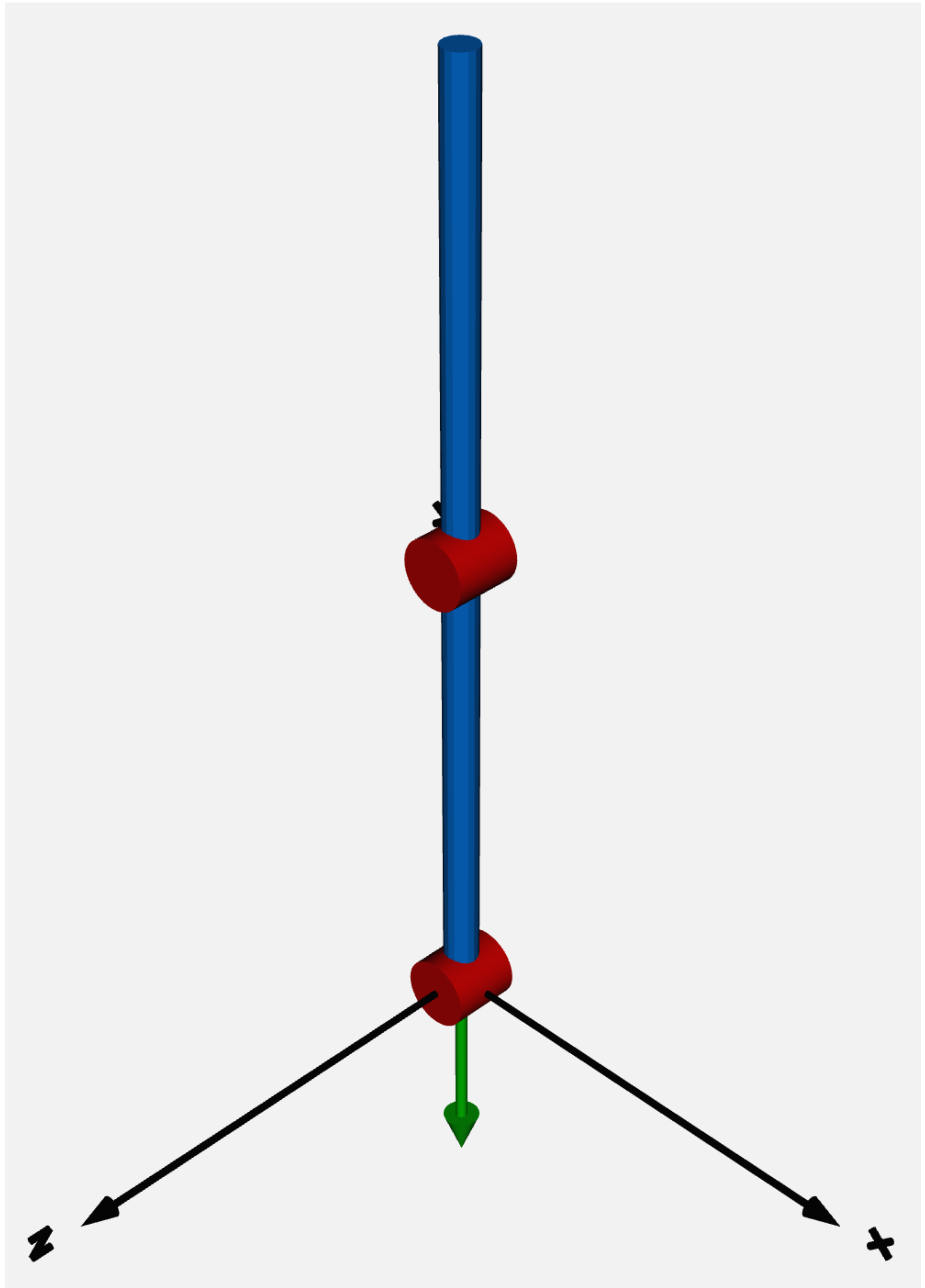

Figure 7. Initial configuration of the 2-DOF robot arm drawn in Modelica.

OpenModelica includes animation capabilities for mechanical system simulations. The animation for this problem can be found at the GitHub repository for the paper [25].

The Modelica model done in graphical mode uses a different means to implement the PID control. In this case, a Modelica class is defined for the PID. The algorithm is implemented in the same algorithm-when-sample structure used above:

```
1  algorithm
2    when sample(0.0, dt) then
3      preErr := err;
4      err := sp - y;
5      integ := integ + ki * err * dt;
6      deriv := kd * (err - preErr) / dt;
7      m := kp * err + integ + deriv;
8      m := max(m, loLim) "Output saturation limits";
9      m := min(m, hiLim);
10   end when;
```

Code Snippet 20.

The wrinkle here is that this executes autonomously under internal control of Modelica. The 'sample' function sets up an internal event that is controlled by the Modelica execution module. The PID objects are defined in the ordinary manner:

```
1  PID pid1(kp = 30.0, ki = 10.0, kd = 0.5, loLim = -10.0, hiLim =
       10.0, dt = 0.005) "Controllers -- PID on angle";
2  PID pid2(kp = 30.0, ki = 10.0, kd = 0.5, loLim = -10.0, hiLim =
       10.0, dt = 0.005);
```

Code Snippet 21.

But the question is how to get data to/from then at the proper times. Modelica does have some synchronization facilities, but a simpler, although probably less efficient, manner was chosen here: put all of the interactions with the controller into the equation section, which operates "continuously"

```
1  pid1.sp = theta1Set;
2  pid1.y = Revolute1.angle;
3  torque1.tau = pid1.m;
4  pid2.sp = theta2Set;
5  pid2.y = Revolute2.angle;
6  torque2.tau = pid2.m;
```

Code Snippet 22.

This assures that the controller has the most up-to-date input data (setpoint and process value) and the system simulation has the most up-to-date controller output (torque command).

The supervisory code for path generation is almost the same as the code used above, except that it too operates in the 'equation' section.

This case produces the same result as above, but also generates an animation, which can be found at [25].

3.4 Java

As noted above, the two-link robot equations are best re-organized for use by conventional ODE solvers. That version of the equations isolates the derivatives by inverting the 'M' matrix (which is 2×2, so easily inverted explicitly). This program is structured in the same way as the DC motor

program, so the only section of interest is the ComputeDerivates() section. Java does not have any built-in support for matrices. Although matrix packages do exist, for this problem the matrix manipulations were written out explicitly (again, the maximum matrix size is 2×2). WARNING: in viewing this code note that in the matrix computations Java uses base-0 indexing while Modelica (and standard matrix notation) uses base-1 indexing. The computation of derivatives thus looks like

```
public double[] computeDerivatives(double t, double[] x)
{
    theta[0] = x[0]; // Copy state variables to local variables
    theta[1] = x[1];
    ComputePositions();
    omega[0] = x[2];
    omega[1] = x[3];
    dxdt[0] = omega[0];  // d theta/dt = omega
    dxdt[1] = omega[1];
    // Be careful of the comparison of equations here -- Java
        uses 0-based indexing
    // while Modelica uses 1-based indexing!!
    // Mass matrix
    M[0][0] = alpha + 2.0 * beta * Math.cos(theta[1]);
    // m(1,1), etc.!
    M[0][1] = delta + beta * Math.cos(theta[1]);
    M[1][0] = M[0][1];
    M[1][1] = delta;
    // Christoffel matrix
    C[0][0] = -beta * Math.sin(theta[1]) * omega[1] + b1;
    C[0][1] = -beta * Math.sin(theta[1]) * (omega[0] + omega[1]);
    C[1][0] = beta * Math.sin(theta[1]) * omega[0];
    C[1][1] = b2;
    // Gravity
    G[0] = (m1 * r1 + m2 * L1) * g * Math.cos(theta[0]) + m2 * r2
         * g * Math.cos(theta[0] + theta[1]);
   G[1] = m2 * r2 * g * Math.cos(theta[0] + theta[1]);
    // Torques
    tau[0] = cmd[0];
    tau[1] = cmd[1];
   double detM = M[0][0] * M[1][1] - M[0][1] * M[1][0];
    dxdt[2] = (M[0][1] * (G[1] - tau[1] + C[1][0] * omega[0] + C
        [1][1] * omega[1]) -
    M[1][1] * (G[0] - tau[0] + C[0][0] * omega[0] + C[0][1] *
        omega[1])) / detM;
    dxdt[3] = (M[1][0] * (G[0] - tau[0] + C[0][0] * omega[0] + C
        [0][1] * omega[1]) -
    M[0][0] * (G[1] - tau[1] + C[1][0] * omega[0] + C[1][1] *
        omega[1])) / detM;
    return dxdt;
}
```

Code Snippet 23.

The Java solution uses almost the same code as the Modelica solutions for PID control and for the path (supervisory) setpoint generation. The PID control is implemented as a full Java class, which makes it easier to work with than the equivalent solutions in Modelica. Otherwise, examination of the Java code shows very similar code sections to the Modelica code and, of course, produces the same result. Gnuplot is used for plotting and the corresponding script is included in the GitHub

repository [25].

3.5 Gazebo/ROS

Gazebo is the default simulator used by Robot Operating System (ROS) developers both in academia and industry for simulation-based prototyping and evaluation. This is because Gazebo was designed with a robust integration to the ROS framework - enabling easy communication interface using standard ROS methods such as topics and services [26]. Within the ROS framework, Gazebo can be used as a node which handles the physics-based interaction between rigid-bodies and the environment, as well as sensors, etc. In this section, the robot modeling, control, and ROS communication specifics to achieve the desired task for the two-DOF robot are discussed.

3.5.1 Robot Modeling

Robot models in Gazebo are defined by a tree structure of interconnected rigid bodies. The rigid body parameters are defined using XML-based formats such as Simulation Description Format (SDF) or Unified Robot Description Format (URDF). The URDF is native to ROS and thus is the more prominent of the two formats when operating in the ROS framework. The URDF provides definitions for the robot links (inertial properties, collision and visual properties), joints (kinematic and dynamic properties), transmission and, with added Gazebo tags, control plugins and geometric materials.

To create a new model, Gazebo provides a model editor which offers simple geometric shapes such as cylinders, spheres and cubes. One may compose the model graphically using the model editor or programmatically using the URDF. Gazebo also allows for custom 3D meshes to be imported. In this work, the two-link robot model (with a stand) was created in URDF using Gazebo-defined cylindrical shapes (see Figure 8) following the parameters defined in **??** .

3.5.2 Robot Control

The most common way of implementing closed-loop control of a robot model in the Gazebo-ROS environment is via the ROS Control package [27]. The ROS Control package is a set of controller plugins which processes joint state data and desired input data to determine expected output for actuation. The package provides various types of controllers: effort-based, position-based, velocity-based, state controller, etc. In this work, two types of effort-based controllers and one state controller are used for the task:

1. effort_controllers/JointPositionController: tracks individually commanded joint positions.
2. effort_controller/JointTrajectoryController: tracks commanded joint trajectories.
3. joint_state_controller/JointStateController: publishes the states of all joints in the model.

The effort-based controllers compute desired forces/torques to joints based on state of the model and desired behavior. These controllers are PID-based and their parameters are defined in a control configuration file.

3.5.3 Framework for simulating the trajectory tracking task

As described above, ROS operates using a node framework where each node is a distinct software program or process, communicating information (messages) with other nodes via topics. In this paper, five active nodes are adopted as graphically illustrated in Figure 9 These nodes include

1. Gazebo simulator (/gazebo): In ROS, the gazebo simulator is spawned as a node which handles all the processes of physics rendering, visualization, etc. This is an existing node in this framework. The created model (defined in an URDF file) is spawned in this simulator and interacts with its environment. The /gazebo node subscribes to controller commands to actuate the robot model and then publishes the state of the entire simulation, especially the robot states (/joint_states) continuously in simulation time.

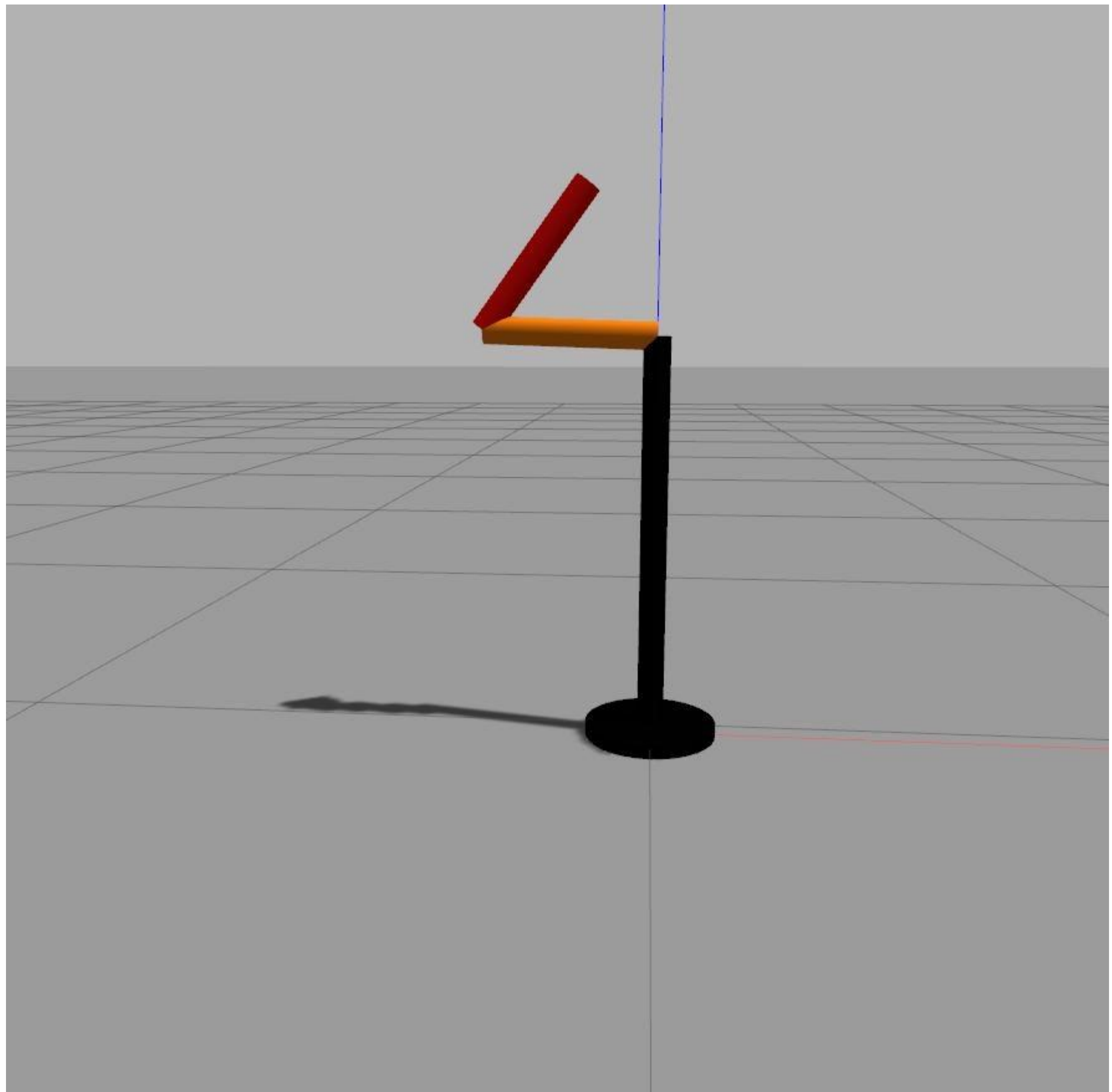

Figure 8. Two-DOF robot model spawned in Gazebo.

2. Circle-drawing program (/draw_circle): This is a custom node written by the authors which follows the pseudo code in Subsection 3.5.4. The /draw_circle node is written in Python using the rospy package. It essentially initializes the circle parameters, computes the joint position or joint trajectory to track (using inverse kinematics) and publishes this as desired input to the respective controller command topics Robot state publisher (/robot_state_publisher): This is an existing ROS node which processes robot joint states to determine robot link/joint frame transformations. Thus, it publishes the robot frame transformation data on a topic called /tf [28].

3. Data recording program (/data_recorder): This is a program written to read robot data (joint states from /joint_states and robot link pose/transform from /tf) and arrange them into a convenient array for post-simulation analysis and storage. This node then publishes the arranged data onto a custom topic called /data_log.

4. Record data (/rosbag_record): This is a common existing tool which enables convenient recording and storage of data available in the ROS communication pipeline in a unique file called a rosbag [29].

3.5.4 *Draw_circle pseudo code*

1. Initialize the ROS node, subscribers and publisher objects
2. Initialize the circle parameters (radius, starting pos, durations: N1, N2
3. Compute the joint configuration (q_init for the circle starting position
4. Publish q_init to the joint_position_controller_command topic
5. Wait N1 seconds
6. Switch controller to a joint_position_controller to a joint_trajectory_controller
7. Compute the joint trajectory (q_traj for completing the circular path
8. Publish q_traj to the joint_trajectory_controller
9. Wait N2 seconds
10. End

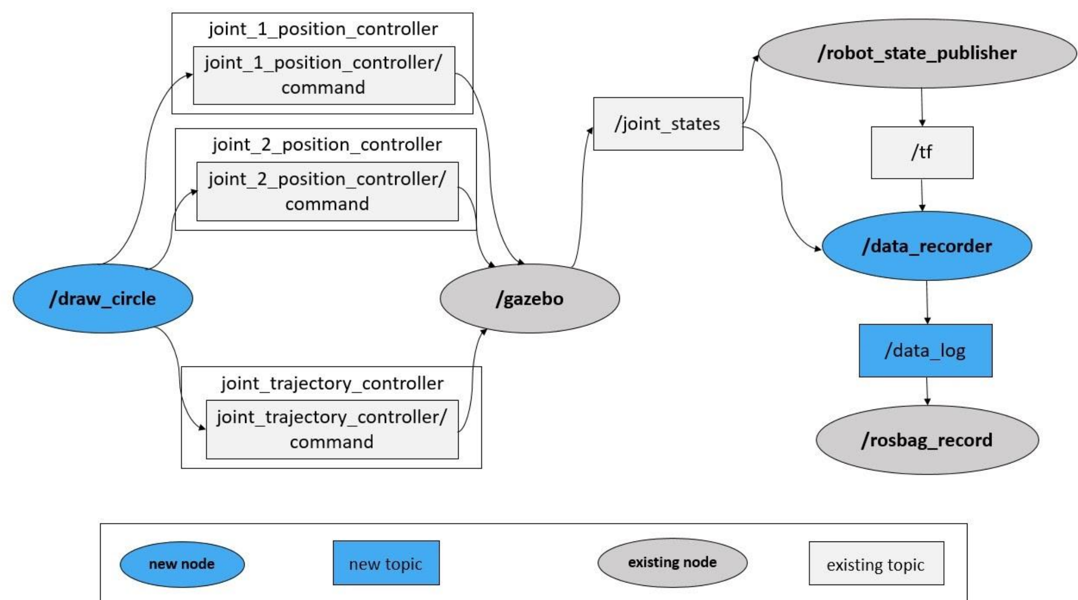

Figure 9. 9. Schematic of the interconnections between ROS nodes and topics.

4 Discussions

This two-part paper demonstrates the use of the OSS such as Python, GNU Octave, Modelica, Java, and Gazebo in the context of MRE education with some simulation showcases. These software offer numerous advantages such as free accessibility, customizability, wide online community support, etc. However, each of these platforms also has its own limitations and challenges. This section reviews some of the potentials and challenges of these software in the context of the problems introduced in this paper. Other challenges facing the applications of the OSS are listed in Part I of this paper, as documented through feedback from the community members. The goal of the paper is that the application showcases and review of the potentials and limitations of each OSS could allow MRE educators to make an informed judgment about the choice of a suitable software for their courses and consequently, facilitate a wider adoption of these software.

Python is a general-purpose programming language which is increasingly being used in various applications and industries and therefore, familiarizing students with Python programming can open up a lot of future opportunities for them. Python also has a large online support community

which can be helpful for troubleshooting and debugging purposes. Despite the community shift towards use of Python 3, there are still some packages and applications, written/compatible with Python 2, which can cause confusion. As for developing and executing Python programs, although a combination of a text editor and command prompt can be used, freely available Integrated Development Environments (IDEs) such as Spyder [30] could provide a more straightforward interface for Python beginners. Installing Python packages could also be challenging at times. As an example, the easiest way to install and integrate Python Control Systems library with Spyder on Windows is through 'pip', despite various methods proposed online and outlined at [31]. The choice of other operating systems could further complicate these issues. As for the application of Python in the context of MRE, Python Control System library and Matlab compatibility module can ease entry into using Python; however, there are still some discrepancies when it comes to generating, indexing, and slicing arrays and matrices and working with them within loops. Online tutorials and articles written by MRE professionals, such as this paper, could help bridge the gap between Python and Matlab. As for data visualization, although Python plots generated with matplotlib might not be as interactive as Matlab plots, the matplotlib library provides numerous options for customizing plots. In summary, Python and the vast collection of its packages would be feasible and beneficial solutions to integrate in MRE education.

The main advantage of using Octave is that it has strong compatibility with MATLAB which allows for greater portability and sharing of programs between the two platforms and nearly eliminates the time required to learn to use Octave with prior MATLAB experience. The open-source nature and availability of the source code also allows tinkerers to experiment, customize, and develop different features. Another advantage of Octave is that it allows for C-style auto-increment and assignment operators like i++ and ++i. It also allows for exponentiation using ^ and **. However, since Octave is not a commercial product it does not yet have all the same built-in functionality and toolbox capabilities as MATLAB, due to the resource and funding limitations. An example of this is the lack of LaTeX support to display equations on plots. Octave, however, does support a subset of TeX functionality which allows for the display of Greek characters, special glyphs, and mathematical symbols. Despite the existence of Octave Control package, it misses functionalities such as controlSystemDesigner (previously known as SISO tool) for control system design and analysis. Furthermore, Octave's user interface and debugging tools are not as mature as MATLAB's. Another limitation of Octave is that it does not have a Simulink environment or a graphical programming environment for simulating and analyzing multi-domain dynamical systems.

The enormous advantage in using Modelica is its focus on physical systems including a large number of libraries for simulating systems containing several energy media. For the MRE world, the core library is the Multibody Mechanics library. This library does three-dimensional, rigid-body dynamics, which, when combined with the Rotational Mechanics and Translational Mechanics libraries, can be used to model a wide variety of mechanical systems. The Electrical, Magnetic, Fluid, and Thermal libraries can interact with the Mechanics libraries to allow simulation of complete systems. In the examples above, the DC motor uses a combination of the Electrical and Rotational Mechanics libraries and the two-link robot uses the Multibody and Rotational libraries. The major disadvantages of Modelica are that it is an entirely different syntax and methodology to learn and its execution times can be slow. For example, there is a noticeable compile time for a small problem in OpenModelica as compared to a compile time that is too short to notice for Java. An institutional advantage to using Modelica is that while educational institutions can very effectively use the open-source OpenModelica version, students entering industry and research labs will often be able to easily transfer their skills to one of the commercial versions.

Java offers the opportunity of a well-structured object-oriented language with efficient execution and very good portability properties. These properties can be used to advantage when dealing with large, complex problems. The obvious disadvantage is that Java is a full-blown programming language so is most useful to people who spend a good part of their lives doing programming. Java does not have native support for advanced numerical mathematics, but packages such as Hipparchus, as used in the above examples, fill that gap nicely.

Compared to other platforms described above, Gazebo provides MRE instructors and students an

avenue to evaluate the integration of a full-fledged robot system in simulation: from prototyping controllers, to simulating virtual actuators and sensors on a realistic virtual model of the physical robot. The level of abstraction enables students to learn about how integrated robot systems are designed in the real world. Gazebo is particularly well suited for this because of its native compatibility with ROS which is the most widely used middleware for robotics in research and industry. However, instructors and students new to robotics may genuinely struggle with the existing high technical overhead required to effectively use the software. For instance, Gazebo (integrally operated with ROS) most commonly operates on Linux OS (although Gazebo supports Windows) which may be less familiar especially to students. Also, although Gazebo provides model editors to create and modify simple robot models using a GUI, users often require knowledge of SDF and URDF to work with more complex robot models. This may also be a challenge for novice students and instructors.

5 Summary, Conclusions, and Future Work

This paper is the second part of the study focused on promoting the application of the OSS in MRE education. In this paper, a 2-DOF robot arm is used as a showcase to demonstrate the application of the OSS in the implementation of a PID controller to achieve trajectory tracking for the robot end-effector. Design and implementation of PID controllers are skills that every MRE graduate should master which, as shown in this paper, can easily be achieved with the OSS. Furthermore, such implementation can also expose the students to the development details of closed-loop control systems. Important code snippets are given and discussed in the paper and the full scripts are made available on the Github repository of the paper along with Matlab scripts, intended to serve as a point of comparison. This two-part paper can provide a comprehensive guide for the utilization of various OSS in simulation, analysis, and control design and implementation of MRE systems. MRE students, instructors, and professionals could choose one or more sections of this paper to learn the application of their software of choice in the design and development of MRE systems. This paper and similar works from MRE professionals can further promote the use of these software and enable the MRE community to reap the numerous benefits of the OSS.

6 References

References

[1] M. A. Gennert and C. B. Putnam, “Robotics as an undergraduate major: 10 years’ experience, 2018 asee annual conference & exposition,” 2018.

[2] R. L. Avanzato, “Collaborative Mobile Robot Design in an Introductory Programming Course for Engineers,” *ASEE*, 1998.

[3] D. Bolick, R. Drushel, and J. Gallagher, “Increasing Accessibility to a First Year Engineering Course in Mobile Autonomous Robotics,” *ASEE Annual Conference*, 2005.

[4] T. Sharpe, R. Maher, J. Peterson, J. Becker, and B. Towle, “Development and Implementation of a Robot Based Freshman Engineering Course,” *ASEE Annual Conference*, 2005.

[5] I. Nourbakhsh, K. Crowley, A. Bhave, E. Hamner, T. Hsiu, A. S. Perez-Bergquist, S. Richards, and K. Wilkinson, “The Robotic Autonomy Mobile Robotics Course: Robot Design, Curriculum Design and Educational Assessment,” *Autonomous Robots*, vol. 18, pp. 103–127, 2005.

[6] A. Soto, P. Espinace, and R. Mitnik, “A Mobile Robotics Course for Undergraduate Students in Computer,” in *IEEE 3rd Latin American Robotics Symposium*, 2006.

[7] P. Ren, J. Terpenny, D. Hong, and R. Goff, “Bridging Theory and Practice in a Senior Level Robotics Course for Mechanical and Electrical Engineers,” *Annual Conference & Exposition*, 2009.

[8] C. N. Thai, "Work in Progress: Development Of a Senior Level Robotics Course for Engineering Students," *ASEE Annual Conference & Exposition*, 2011.

[9] T. Inanc and H. Dinh, "A Low-Cost Autonomous Mobile Robotics Experiment: Control, Vision, Sonar, and Handy Board," *Computer Applications in Engineering Education*, vol. 20, pp. 203–213, 2012.

[10] A. Gilmore, "Design Elements of a Mobile Robotics Course Based on Student Feedback," in *ASEE Annual Conference & Exposition*, 2015.

[11] A. Minaie and R. Sanati-Mehrizy, "An International Study of Robotics Courses in The Computer Science/Engineering Curriculum," *Annual Conference & Exposition*, 2006.

[12] W. W. Walter and W. E. Spath, "Experience Teaching a Multidisciplinary Project-Based Robotics Course Building Autonomous Mobile Robots," in *ASEE Annual Conference & Exposition*, 2011.

[13] P. Wild, K. Firth, and B. Surgenor, "Lessons Learned from a Mobile Robot Based Mechatronics Course," *ASEE Annual Conference*, 2005.

[14] E. Danahy, E. Wang, J. Brockman, A. Carberry, B. Shapiro, and C. B. Rogers, "Lego-Based Robotics in Higher Education: 15 Years of Student Creativity," *International Journal of Advanced Robotic Systems*, vol. 11, pp. 27–27, 2014.

[15] S. He and S. J. Hsieh, *Multi-sensors for Robot Teaming Using Raspberry Pi and VEX Robotics Construction Kit*, Utah, 2018.

[16] R. L. Avanzato and C. G. Wilcox, "Work in Progress: Introductory Mobile Robotics and Computer Vision Laboratories Using ROS and MATLAB," *ASEE Annual Conference & Exposition*, 2018.

[17] S. A. Wilkerson, J. Forsyth, C. Sperbeck, M. Jones, and P. D. Lynn, "A Student Project using Robotic Operating System (ROS) for Undergraduate Research," *ASEE Annual Conference & Exposition*, 2017.

[18] G. W. Recktenwald and D. E. Hall, "Using Arduino as a Platform for Programming, Design and Measurement in a Freshman Engineering Course," in *ASEE Annual Conference & Exposition*, 2011.

[19] W. W. Walter and T. G. Southerton, "Teaching Robotics by Building Autonomous Mobile Robots Using the Arduino," in *ASEE Annual Conference & Exposition*, 2014.

[20] N. Lotfi, J. A. Novosad, and H. Phan-Van, "A Multidisciplinary Course and the Corresponding Laboratory Platform Development for Teaching the Fundamentals of Advanced Autonomous Vehicles," *ASEE Annual Conference & Exposition*, 2019.

[21] S. Abbasi and E. M. Kim, "Integration of C Programming and IoT in a Raspberry Pi-controlled Robot Car in a Freshmen/Sophomore Engineering Core Class," in *2020 ASEE Virtual Annual Conference*, and others, Ed. ASEE, 2020.

[22] C. Luo, J. Wang, W. Zhao, Wang, and L, "Multi-Lab-Driven Learning Method Used for Robotics ROS System Development," *ASEE Annual Conference & Exposition*, 2017.

[23] R. M. Murray, S. S. Sastry, Li, and Z, *A Mathematical Introduction to Robotic Manipulation*. Boca Raton, FL: CRC Press, 1994.

[24] M. W. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*, 2nd ed., and others, Ed. John Wiley & Sons, 03 2020.

[25] " Github Code Repository," 2021. [Online]. Available: https://github.com/nlotfiy/OSS-paper-scripts

[26] Gazebosim, "Gazebo Tutorial," 2014. [Online]. Available: http://gazebosim.org/tutorials?tut=ros_overview

[27] ROS, "ROS Control documentation," 2022. [Online]. Available: http://wiki.ros.org/ros_control

[28] ——, " Ros Tf Documentation," 2017. [Online]. Available: http://wiki.ros.org/tf

[29] ——, "ROS Bag documentation," 2020. [Online]. Available: http://wiki.ros.org/rosbag

[30] S. W. Contributors, "The Scientific Python Development Environment," 2021. [Online]. Available: https://www.spyder-ide.org/

[31] python-control. org, "Python Control Systems Library," 2021. [Online]. Available: https://python-control.readthedocs.io/en/0.9.1/