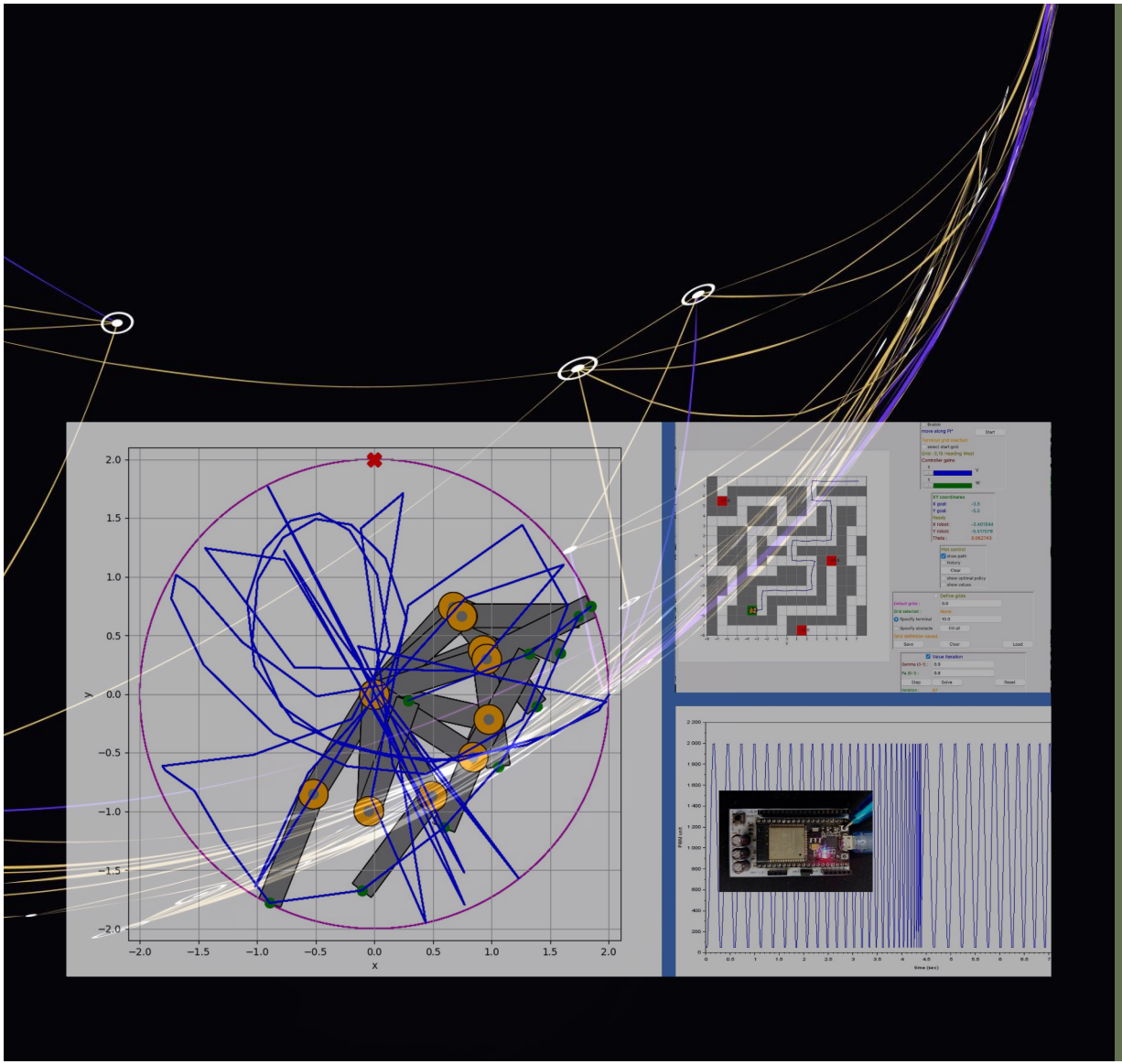


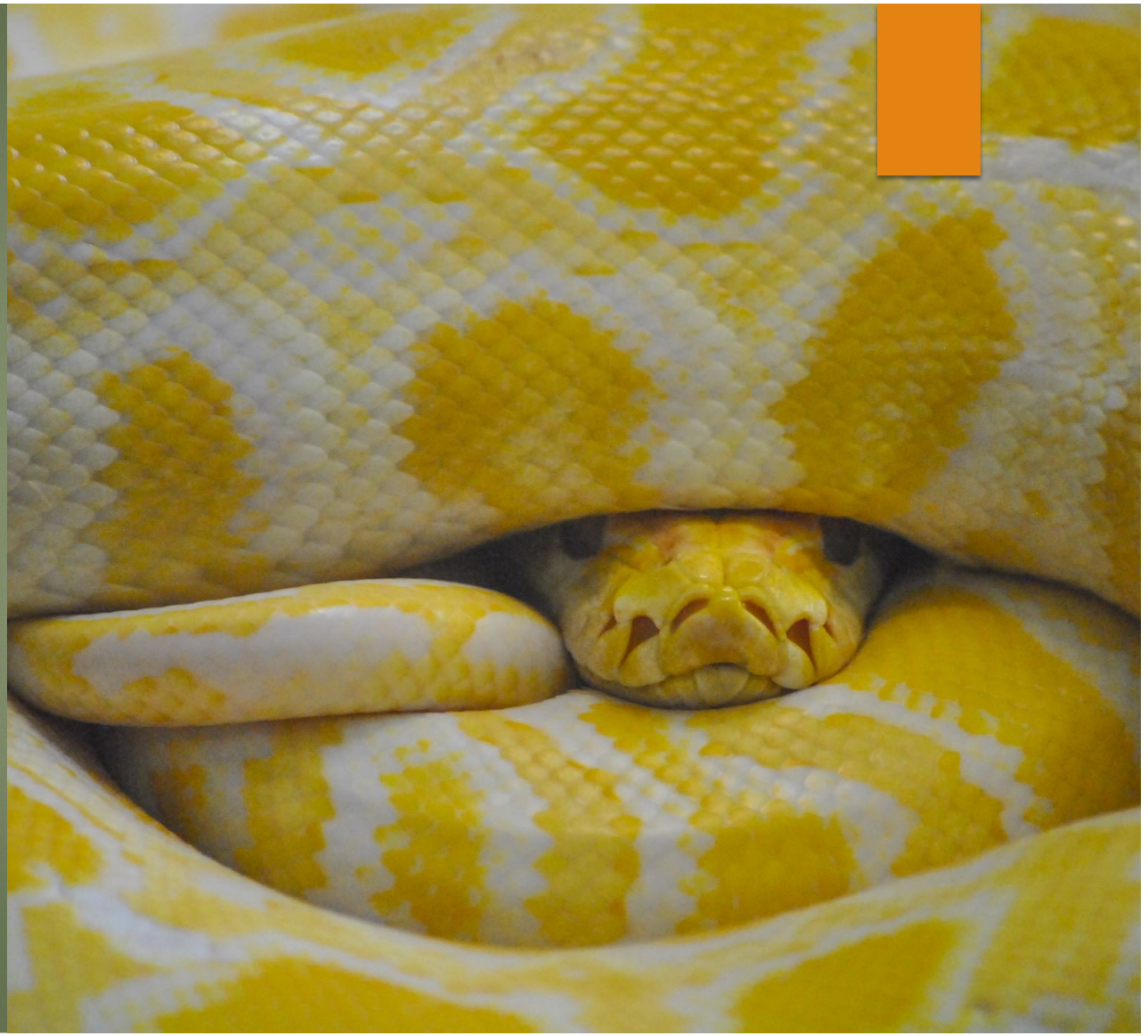
2-link manipulator simulation

Dr.Varodom Toochinda
Dept. of Mechanical Engineering
Kasetsart University

Aside: how to use MQTT on Jupyter notebook

Pro : can design custom GUI

Con : need to write your development program



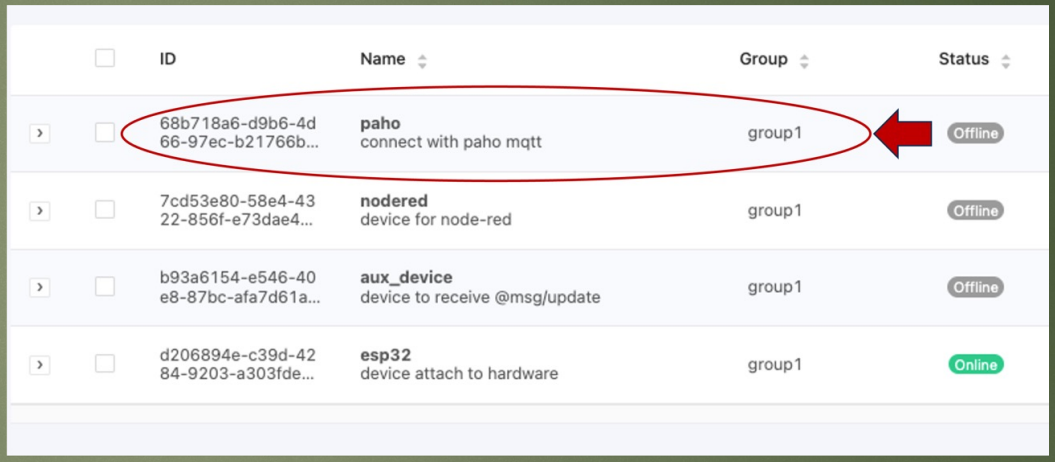
paho_mqtt.ipynb
paho_tk.ipynb

Main steps

Install paho-mqtt package

```
!pip install paho-mqtt
```

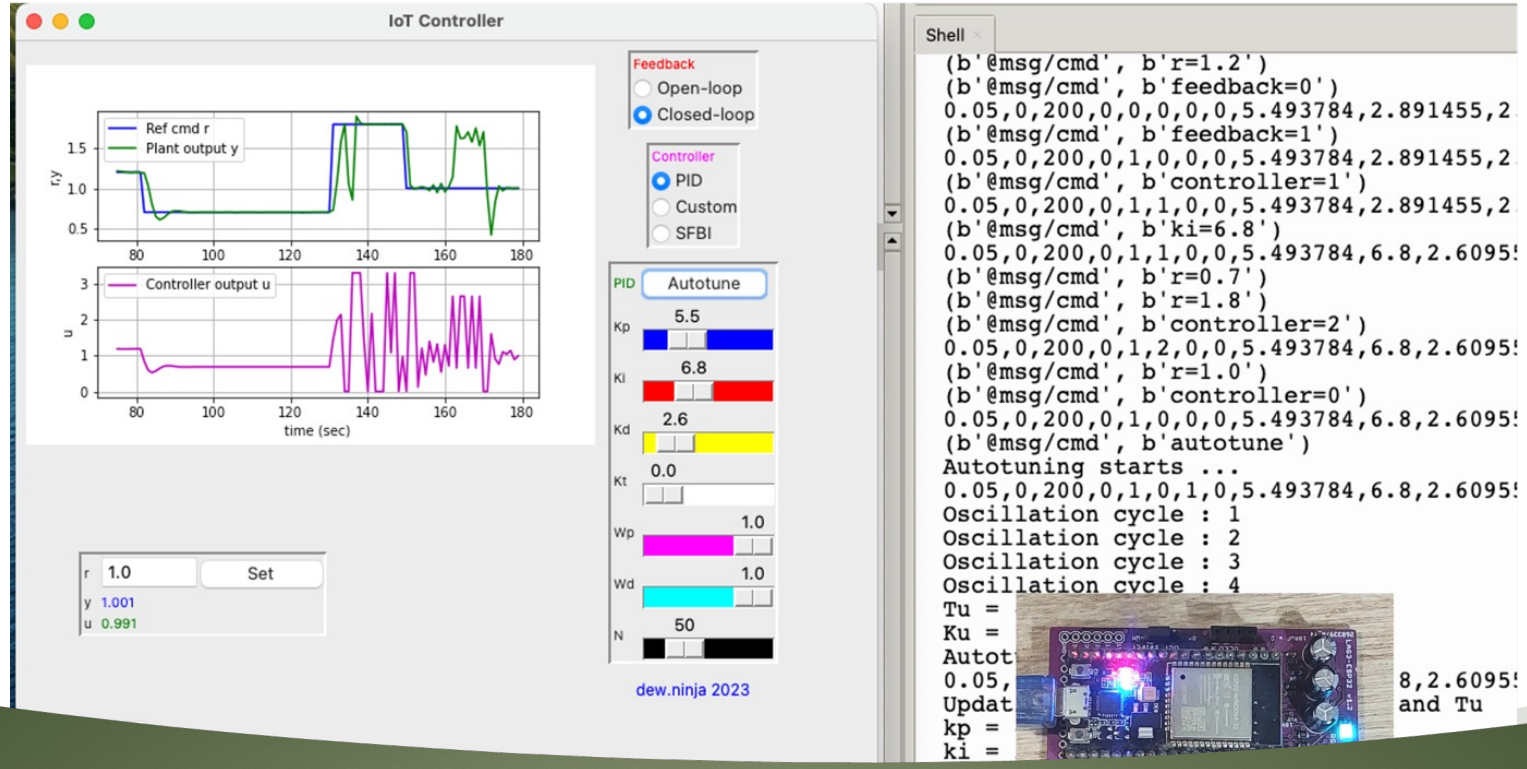
create device on NETPIE 2020
Then group together



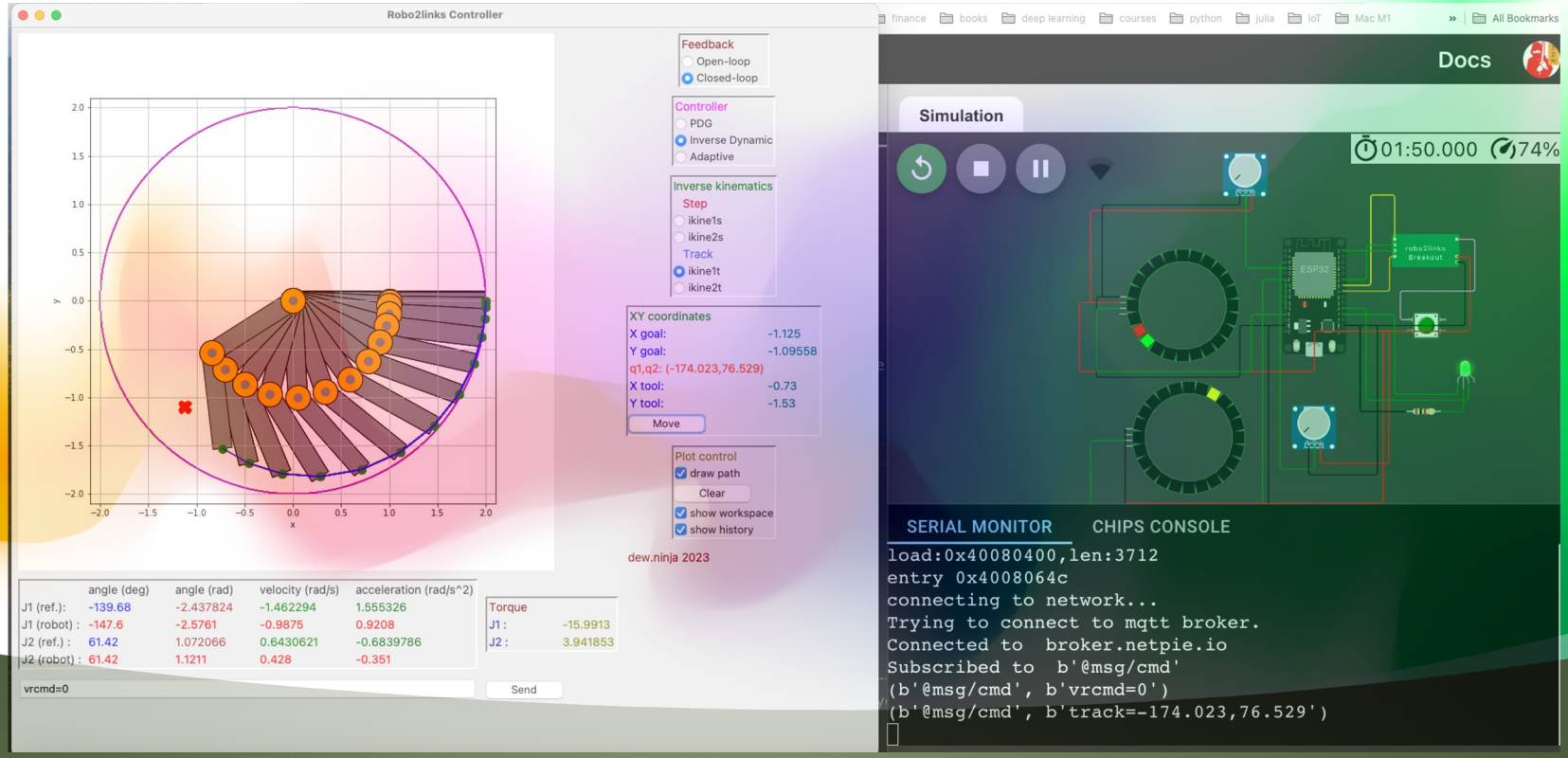
	ID	Name	Group	Status
›	68b718a6-d9b6-4d66-97ec-b21766b...	**paho** connect with paho mqtt	group1	Offline
›	7cd53e80-58e4-4322-856f-e73dae4...	**nodered** device for node-red	group1	Offline
›	b93a6154-e546-40e8-87bc-afa7d61a...	**aux_device** device to receive @msg/update	group1	Offline
›	d206894e-c39d-4284-9203-a303fde...	**esp32** device attach to hardware	group1	Online

Import paho-mqtt

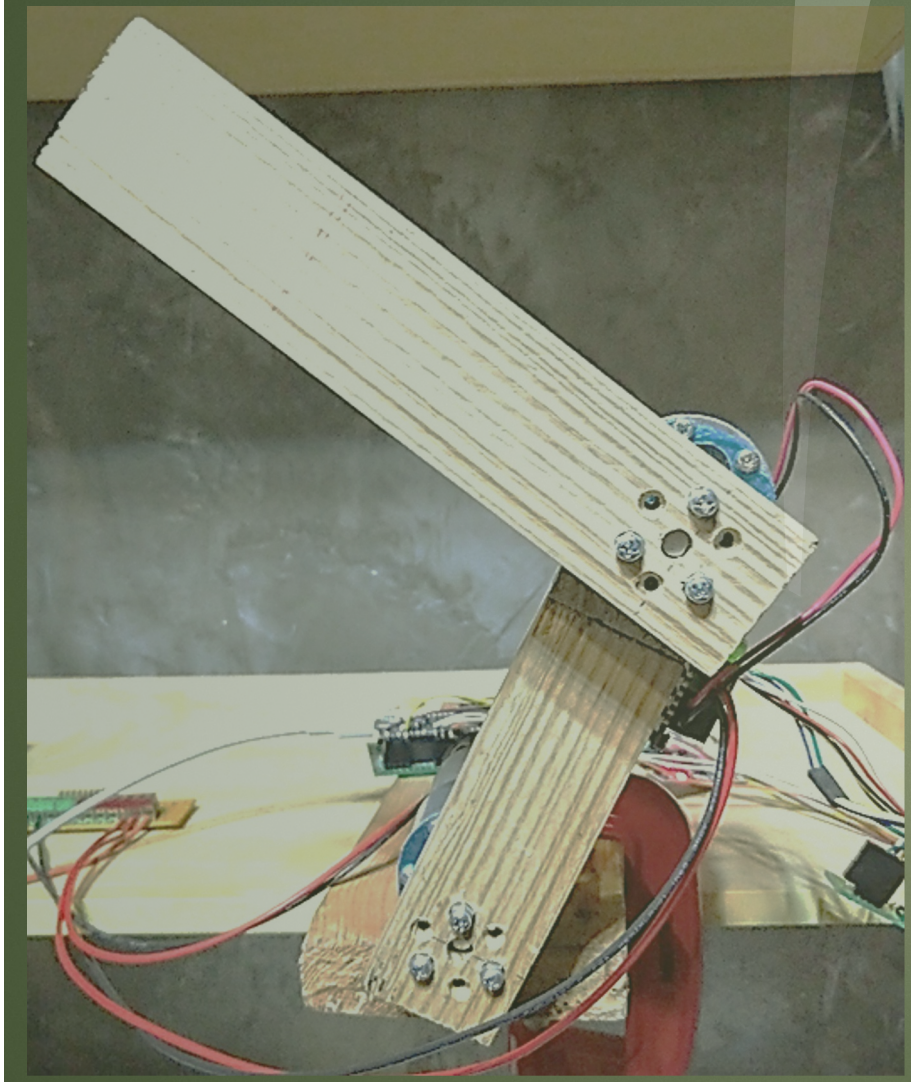
```
import paho.mqtt.client as mqtt
```



Testing MQTT communication



ROBOT CONTROLLER DEVELOPMENT

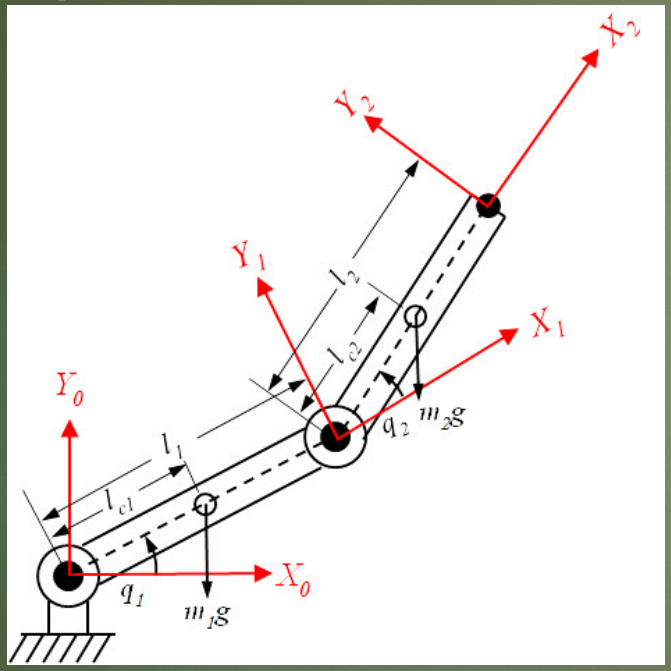


Robot Control

- 2-link manipulator (simulated)
  - independent joint control
  - nonlinear controllers
  - forward & inverse kinematics study

2-link manipulator dynamics

$$d_{11}\ddot{q}_1 + d_{12}\ddot{q}_2 + c_{121}\dot{q}_1\dot{q}_2 + c_{211}\dot{q}_2\dot{q}_1 + c_{221}\dot{q}_2^2 + g_1 = \tau_1$$
$$d_{21}\ddot{q}_1 + d_{22}\ddot{q}_2 + c_{112}\dot{q}_1^2 + g_2 = \tau_2$$



$$d_{11} = m_1 l_{c1}^2 + m_2\left(l_1^2 + l_{c2}^2 + 2l_1 l_{c2}\cos q_2\right) + I_1 + I_2$$
$$d_{12} = d_{21} = m_2\left(l_{c2}^2 + l_1 l_{c2}\cos q_2\right) + I_2$$
$$d_{22} = m_2 l_{c2}^2 + I_2$$

inertia

$$c_{121} = c_{211} = \frac{1}{2}\frac{\partial d_{11}}{\partial q_2} = -m_2 l_1 l_{c2}\sin q_2 = \xi$$
$$c_{221} = \frac{\partial d_{12}}{\partial q_2} - \frac{1}{2}\frac{\partial d_{22}}{\partial q_1} = \xi$$
$$c_{112} = \frac{\partial d_{21}}{\partial q_1} - \frac{1}{2}\frac{\partial d_{11}}{\partial q_2} = -\xi$$

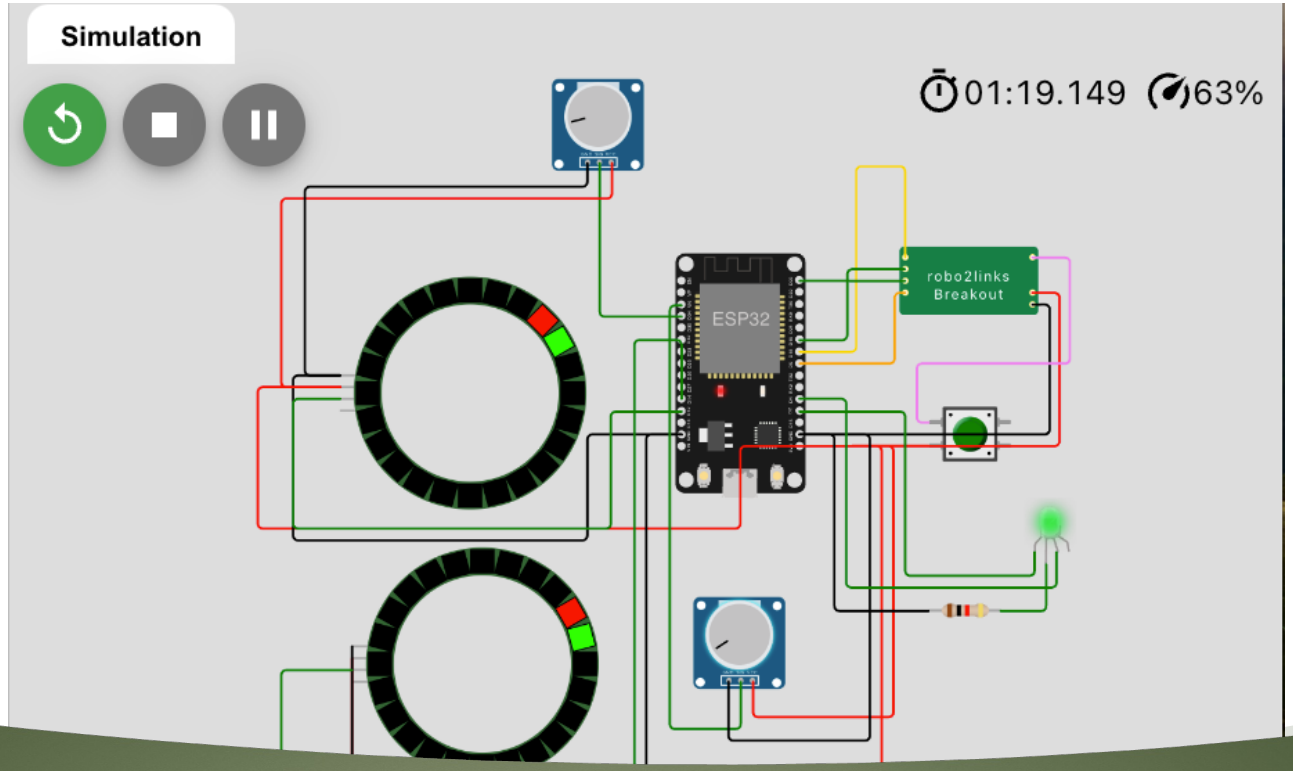
Coriolis

$$C = \begin{bmatrix} \xi\dot{q}_2 & \xi(\dot{q}_1 + \dot{q}_2) \\ -\xi\dot{q}_1 & 0 \end{bmatrix}$$

$$g_1 = \frac{\partial P}{\partial q_1} = (m_1 l_{c1} + m_2 l_1) g\cos q_1 + m_2 l_{c2} g\cos(q_1 + q_2)$$

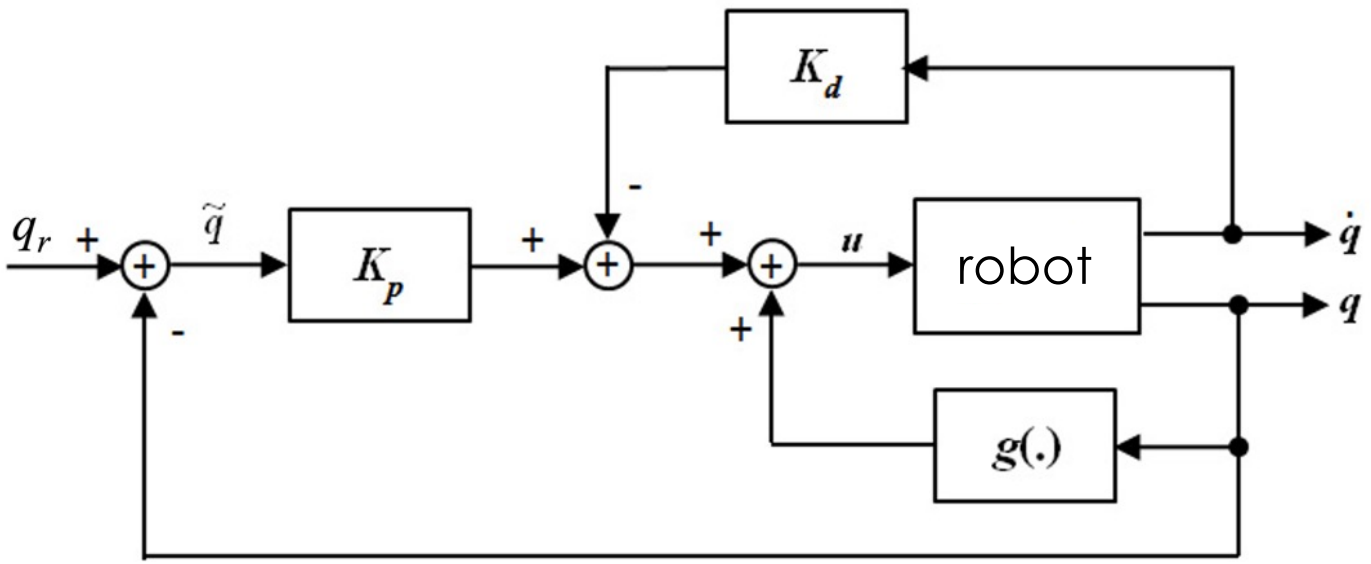
gravity

$$g_2 = \frac{\partial P}{\partial q_2} = m_2 l_{c2} g\cos(q_1 + q_2)$$



Wokwi simulation (no IoT)

https://wokwi.com/projects/389313368946834433



$$u = g(q) + K_p\tilde{q} - K_d\dot{q}$$

PD control with gravity compensation (PDG)

เพิ่มค่าผิดพลาดจากการโมเดลเพื่อสอดคล้องกับความเป็นจริง

```
# add modeling error
m1_e = m1*(1+et*(random.random()-0.5))
l1_e = l1*(1+et*(random.random()-0.5))
lc1_e = 0.5*l1_e # link c.g'
I1_e = I1*(1+et*(random.random()-0.5))
```

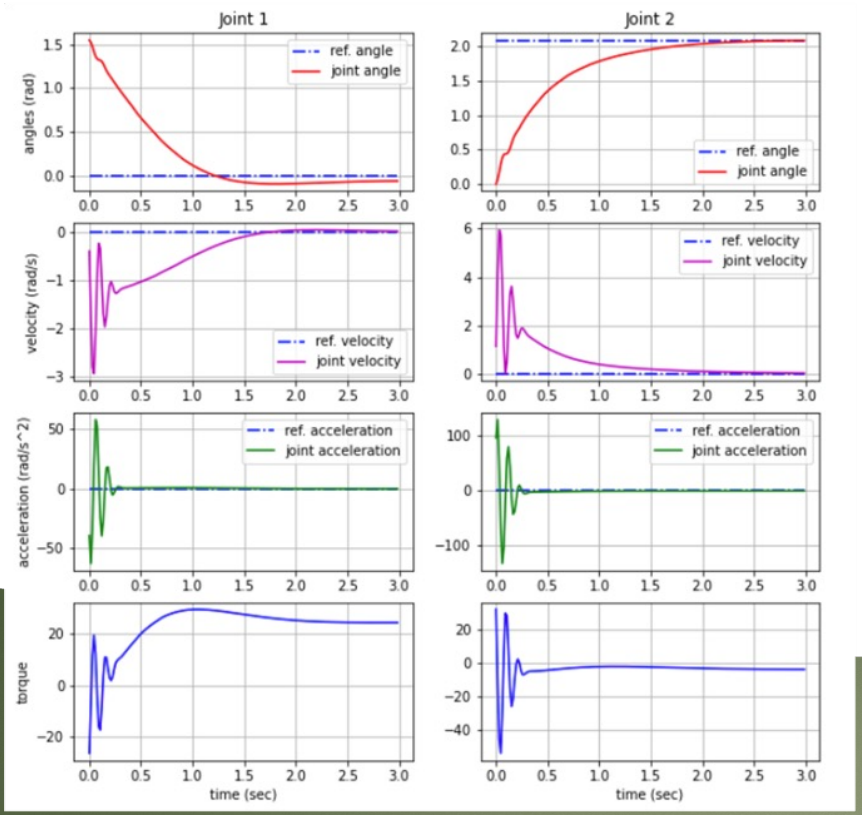
```
m2_e = m2*(1+et*(random.random()-0.5))
l2_e = l2*(1+et*(random.random()-0.5))
lc2_e = 0.5*l2_e # link c.g'
I2_e = I2*(1+et*(random.random()-0.5))
```

```
def PD_grav():
    global tau1, tau2, q1e, q2e
    cos12 = math.cos(q1+q2)
    q1e = q1r - q1
    q2e = q2r - q2
    g1 = ((m1_e*lc1_e + m2_e*l1_e)*g*math.cos(q1) +
                                                m2_e*lc2_e*g*cos12)
    g2 = m2_e*lc2_e*g*cos12
    tau1 = kp1_pdg*q1e - kd1_pdg*qd1 + g1
    tau2 = kp2_pdg*q2e - kd2_pdg*qd2 + g2
```

PD control with gravity compensation (PDG)

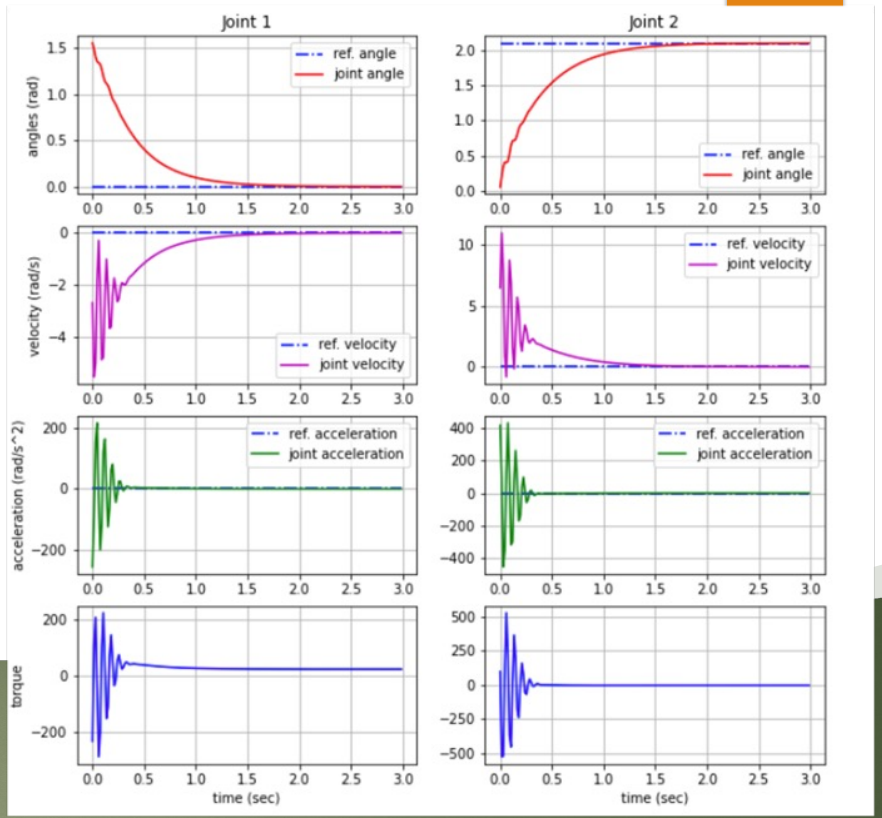
psim = 0

kp_1=20, kp_2=20,kd_1=15,kd_2=15



psim = 1

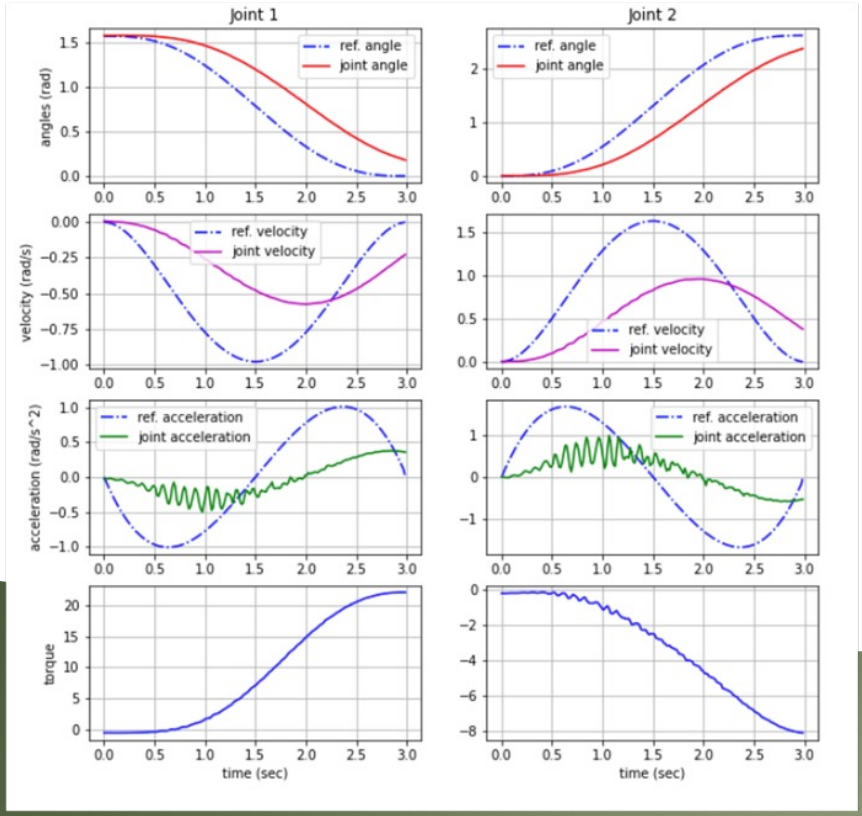
kp_1=250, kp_2=250,kd_1=100,kd_2=100



Step response of PDG

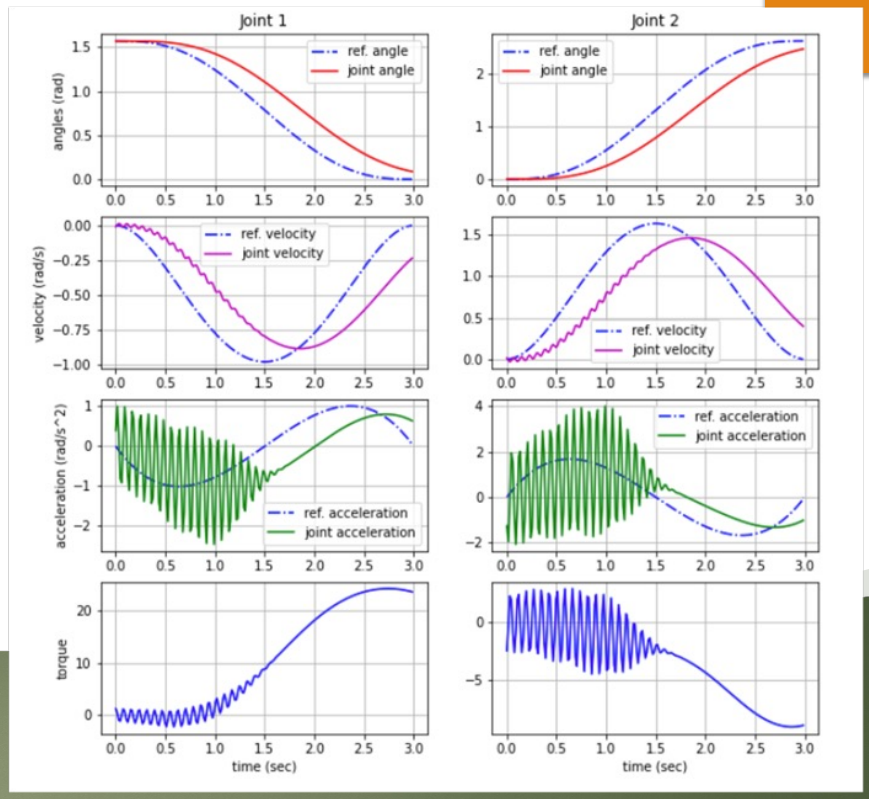
psim = 0

kp_1=20, kp_2=20,kd_1=15,kd_2=15

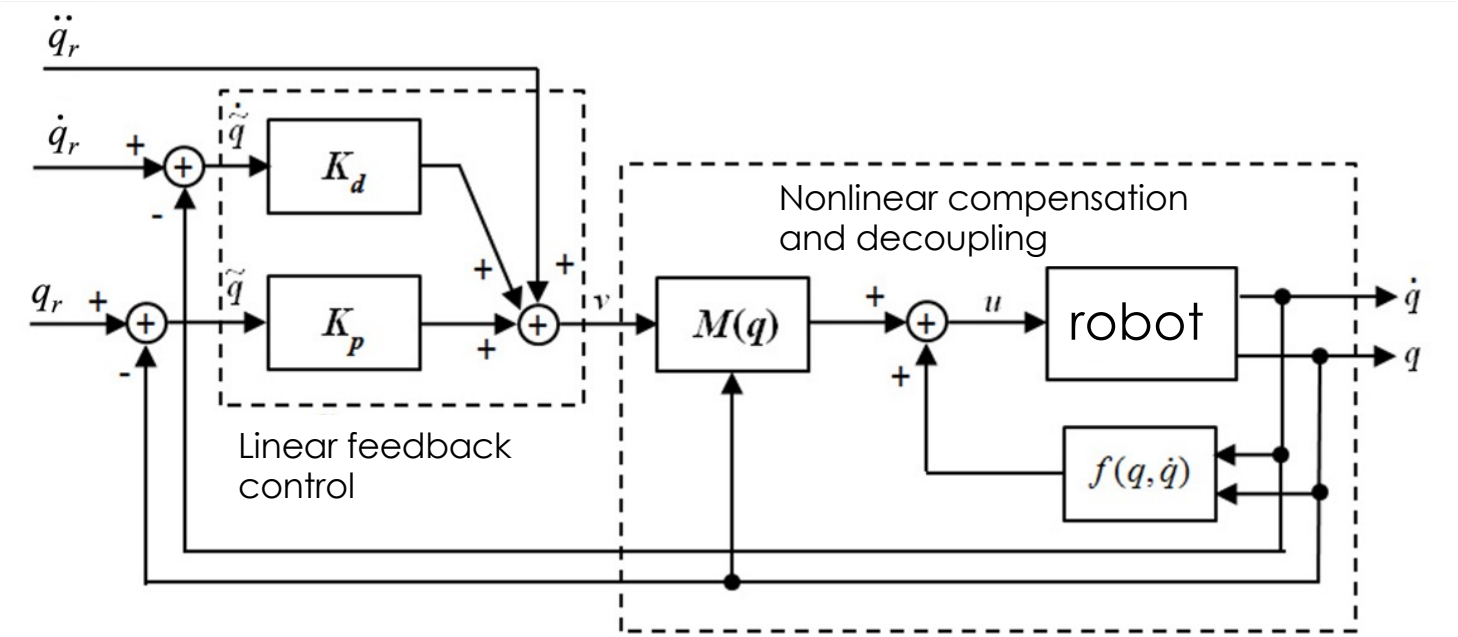


psim = 1

kp_1=250, kp_2=250,kd_1=100,kd_2=100



qpoly trajectory tracking of PDG



inverse dynamics control

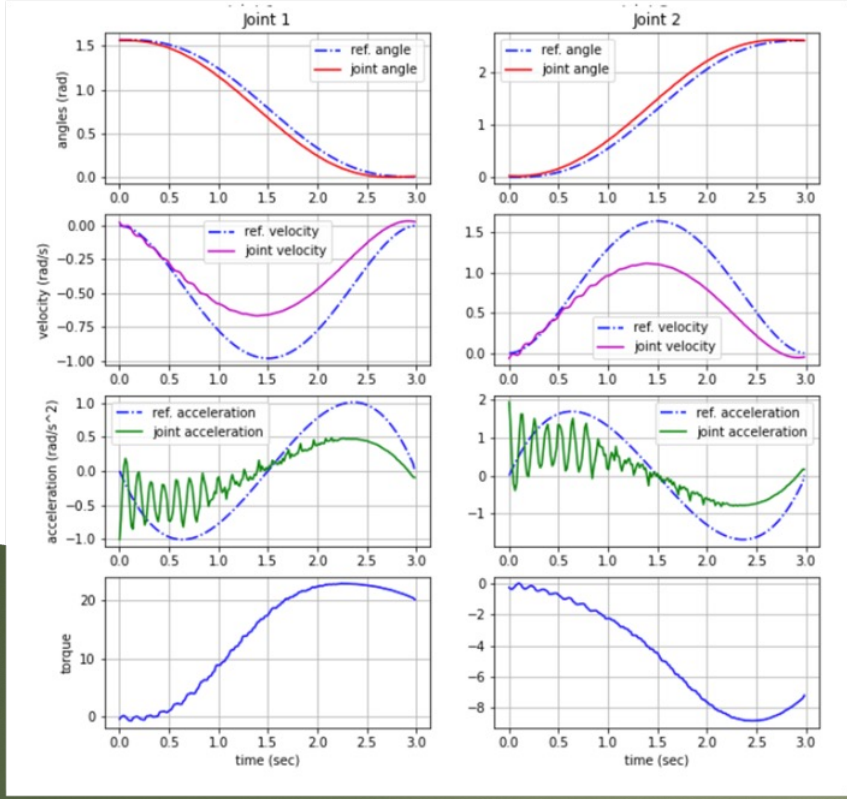
```
def inverse_dynamics():
    global tau1, tau2, q1e, q2e, b
    cos12 = math.cos(q1+q2)
    cos2 = math.cos(q2)

    q1e = q1r-q1  # joint angle error
    q2e = q2r-q2
    qde1 = qd1r-qd1 # joint velocity error
    qde2 = qd2r-qd2

    # inertia matrix element
    d11e = (m1_e*lc1_e**2 +
                    m2_e*(l1_e**2 + lc2_e**2 + 2*l1_e*lc2_e*cos2)
                    + I1_e + I2_e)
    d12e = m2_e*(lc2_e**2+l1_e*lc2_e*cos2)+I2_e
    d21e = d12e
    # coriolis matrix element
    zeta = -m2_e*l1_e*lc2_e*math.sin(q2)
    c11 = zeta*qd2
    c12 = zeta*(qd1+qd2)
    c21 = -zeta*qd1
    # c22 = 0

    # gravity term
    g1 = ((m1_e*lc1_e + m2_e*l1_e)*g*math.cos(q1)+
                                    m2_e*lc2_e*g*cos12)
    g2 = m2_e*lc2_e*g*cos12

    # controller output computation
    v1 = kp1_invd*q1e + kd1_invd*qde1 + qdd1r
    v2 = kp2_invd*q2e + kd2_invd*qde2 + qdd2r

    tau1 = d11e*v1 + d12e*v2 + (c11+b)*qd1 + c12*qd2 + g1
    tau2 = d21e*v1 + d22e*v2 + c21*qd1 + b*qd2 + g2
```

```
d22e = m2_e*lc2_e**2 + I2_e # constant
```

Inverse dynamics function coding with micropython

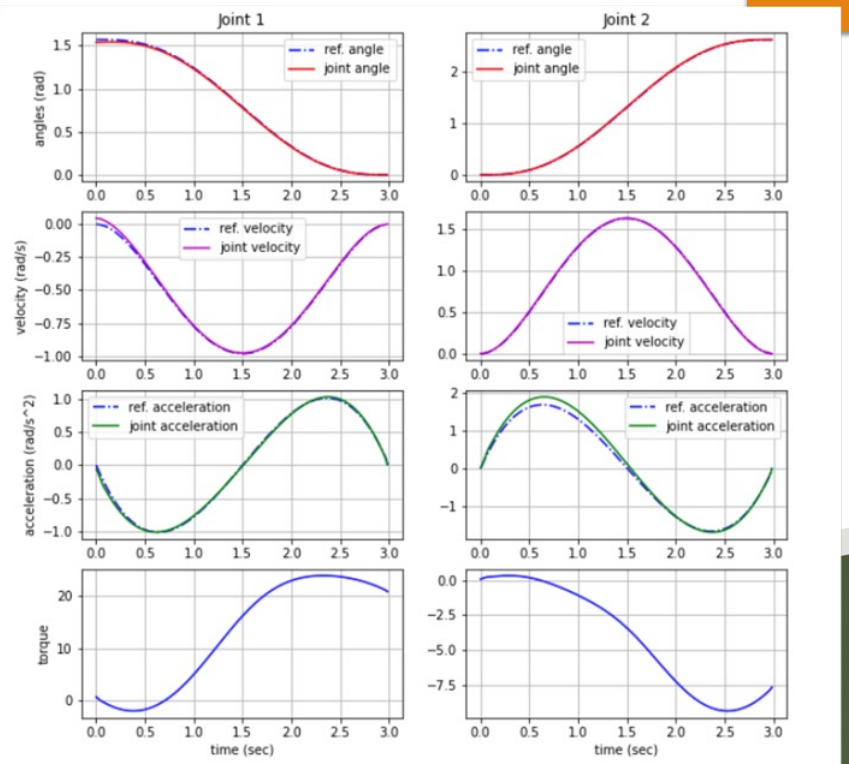
psim = 0

kp_1=_30, kp_2=30,kd_1=10,kd_2=10

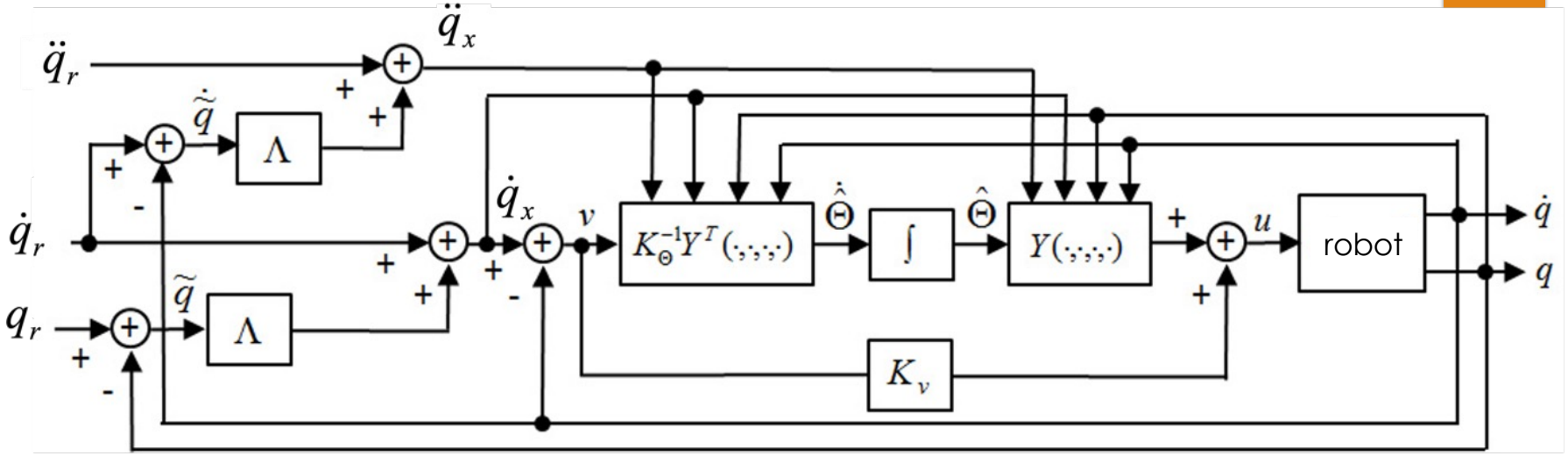


psim = 1

kp_1=70, kp_2=70,kd_1=50,kd_2=50



qpoly trajectory tracking with inverse dynaics control



adaptive controller

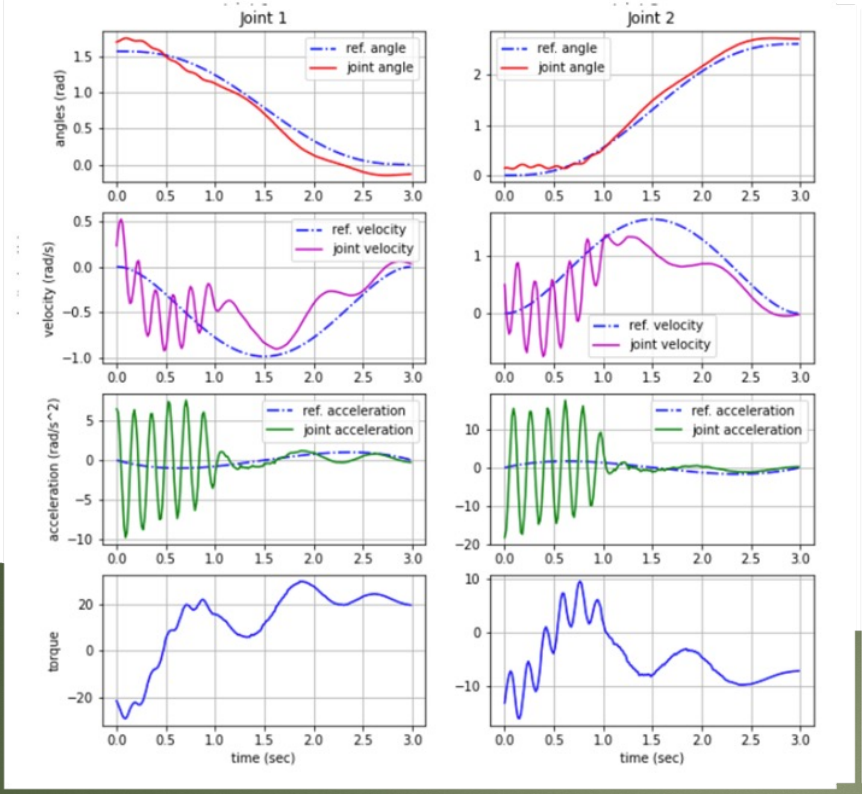
```
def adaptive_control():
    global tau1, tau2, q1e, q2e, qd1e, qd2e, q1, q2, qd1,\
           qd2, qdd1, qdd2

    cos12 = math.cos(q1+q2)
    cos1 = math.cos(q1)
    cos2 = math.cos(q2)
    sin2 = math.sin(q2)

    q1e = q1r-q1  # joint angle error
    q2e = q2r-q2
    qd1e = qd1r-qd1 # joint velocity error
    qd2e = qd2r-qd2

    qd1x = lam da1*q1e + qd1r
    qd2x = lam da2*q2e + qd2r
    qdd1x = lam da1*qd1e + qdd1r
    qdd2x = lam da2*qd2e + qdd2r

    v1 = qd1x -qd1
    v2 = qd2x -qd2

    # create elements of Y matrix
    y11 = qdd1x
    y12 = (cos2*(2*qdd1x + qdd2x) - sin2*(qd1*qd2x +
                                          qd2*qd1x + qd2*qd2x))
    y13 = qdd2x
    y14 = g*cos1
    y15 = g*cos12
    y21 = 0.0
    y22 = cos2*qdd1x + sin2*qd1*qd1x
    y23  = qdd1x + qdd2x
    y24 = 0.0
    y25 = g*cos12
```

```
    # compute theta_dot vector in (8.31)
    thd1_hat = (y11*v1 + y21*v2)/kth
    thd2_hat = (y12*v1 + y22*v2)/kth
    thd3_hat = (y13*v1 + y23*v2)/kth
    thd4_hat = (y14*v1 + y24*v2)/kth
    thd5_hat = (y15*v1 + y25*v2)/kth

    # get theta vector from integrator objects
    th1_hat = Int1.out(thd1_hat)
    th2_hat = Int2.out(thd2_hat)
    th3_hat = Int3.out(thd3_hat)
    th4_hat = Int4.out(thd4_hat)
    th5_hat = Int5.out(thd5_hat)

    y1 = (y11*th1_hat + y12*th2_hat + y13*th3_hat +
                                      y14*th4_hat + y15*th5_hat)
    y2 = (y21*th1_hat + y22*th2_hat + y23*th3_hat +
                                      y24*th4_hat + y25*th5_hat)
    tau1 = y1 + kv1*v1
    tau2 = y2 + kv2*v2
```

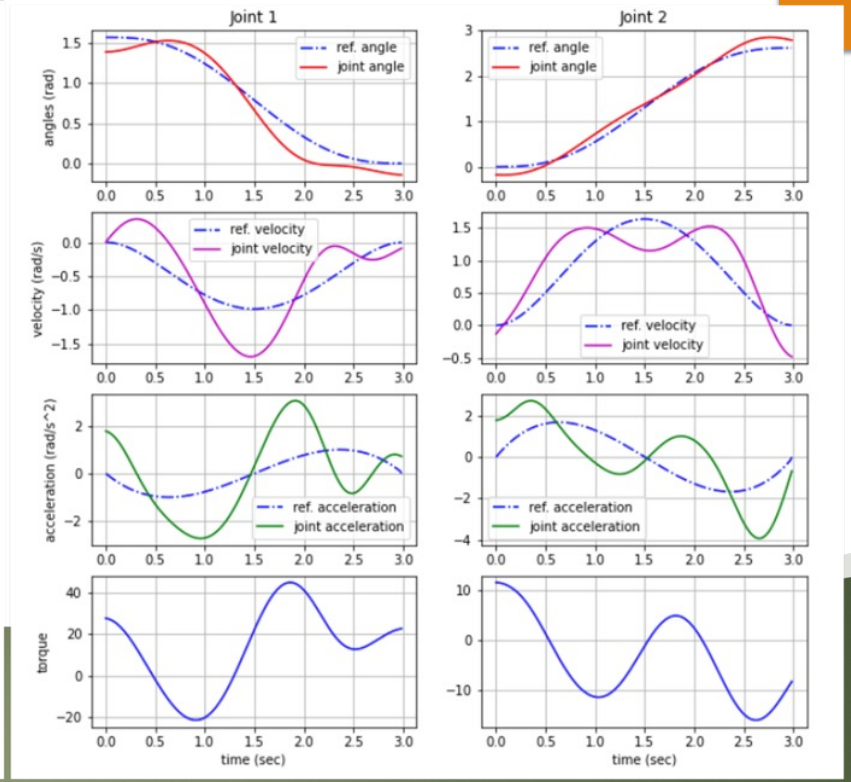
Adaptive control function coding in micropython

lamda1 = 20.0, lamda2 = 12.0, kth = 5000.0, kv1 = 7.5, kv2 = 5.0

psim = 0



psim = 1

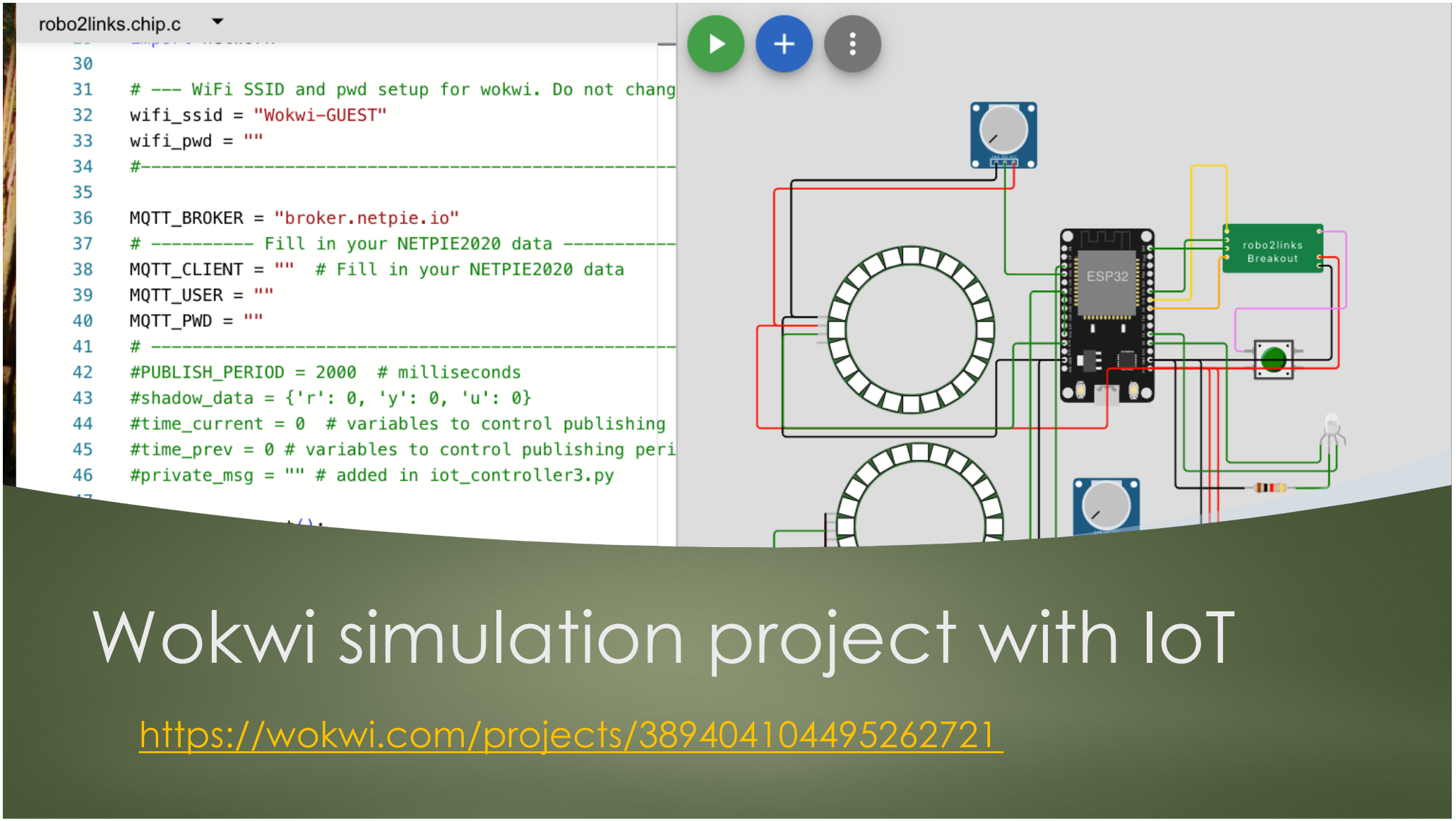


qpoly trajectory tracking with inverse dynaics control

```
90    def update_tk():
91        global updatestr
92        updatestr = ("{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{}"
93                     .format(T, plantsim, datasize, cmdgen_flag,feedback,
94                     controller,q1r,qd1r,qdd1r,q2r,qd2r,qdd2r,q1,qd1,
95                     qdd1,q2,qd2,qdd2,tau1,tau2))
96        # print(updatestr)
97        if online:
98            client.publish('@msg/update', updatestr)
99
```

```
1132    while True:
1133        if online:
1134            client.check_msg()
1135            update_tk()
1136            sleep_ms(1000)  # change this to adjust IoT update rate
1137        else:
1138            user_input()
```

Adding IoT feature to robot control

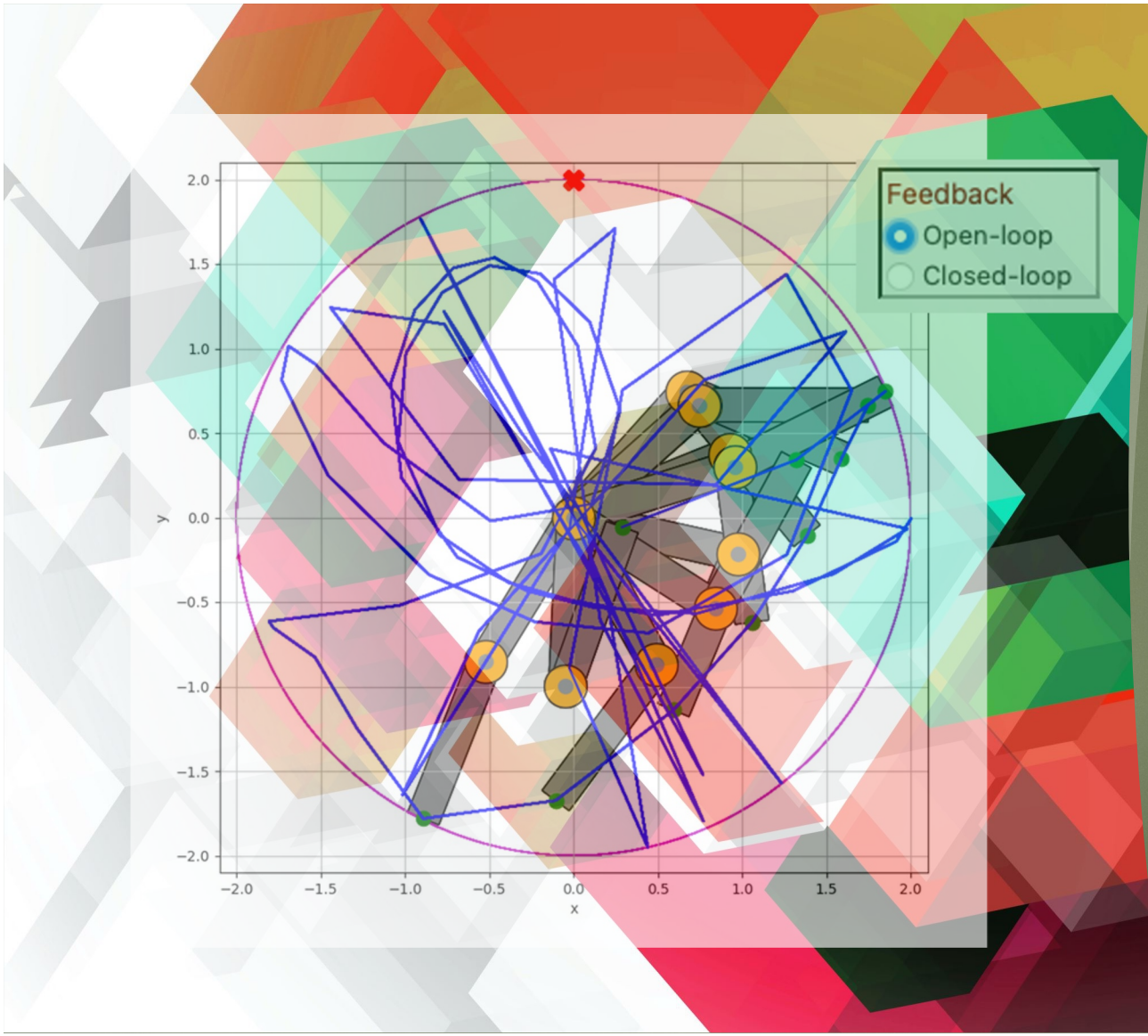


Wokwi simulation project with IoT

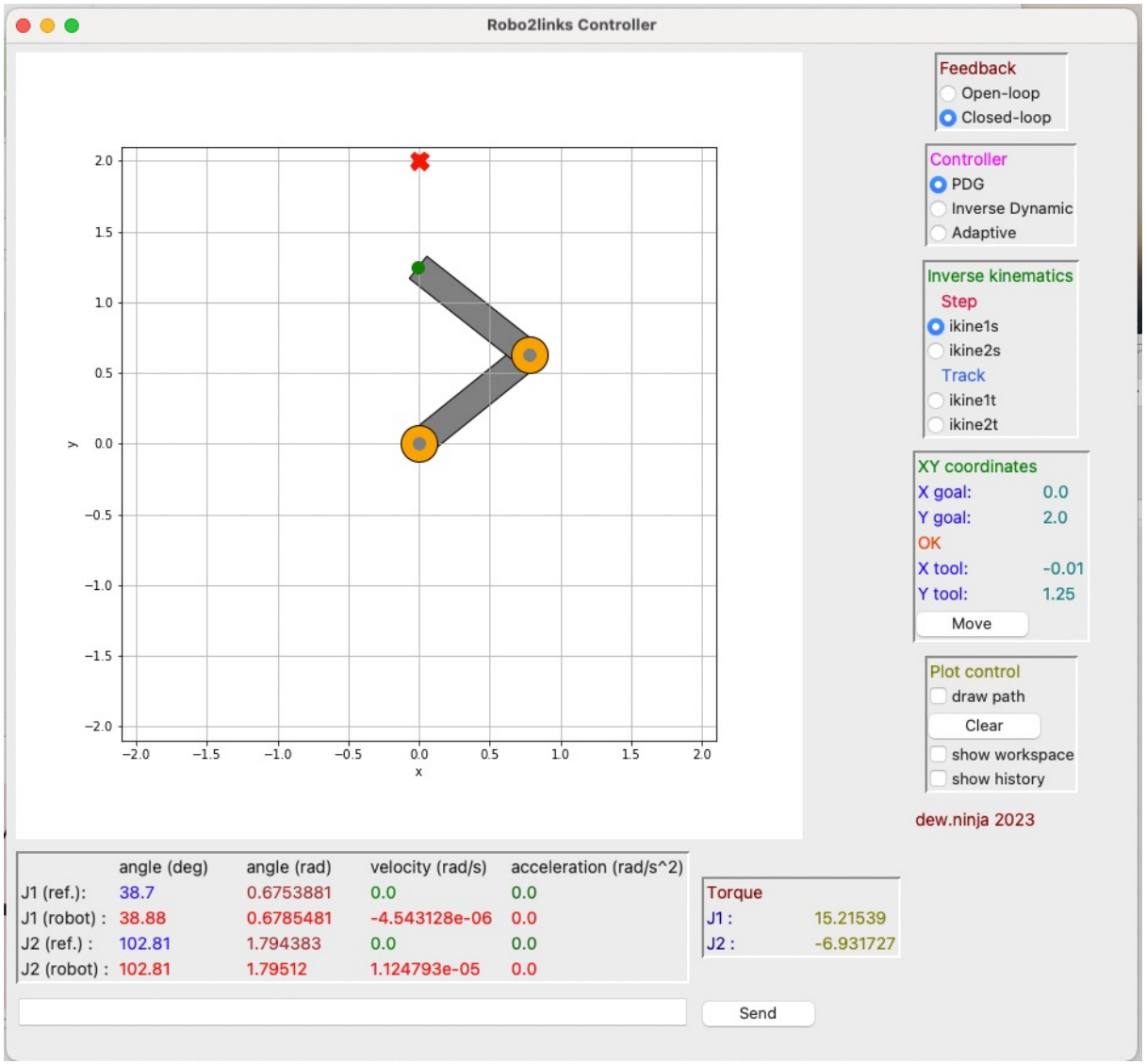
https://wokwi.com/projects/389404104495262721

Alternative : use virtual serial port (VSP) for Wokwi

- Installation procedure is provided as separate video clip.



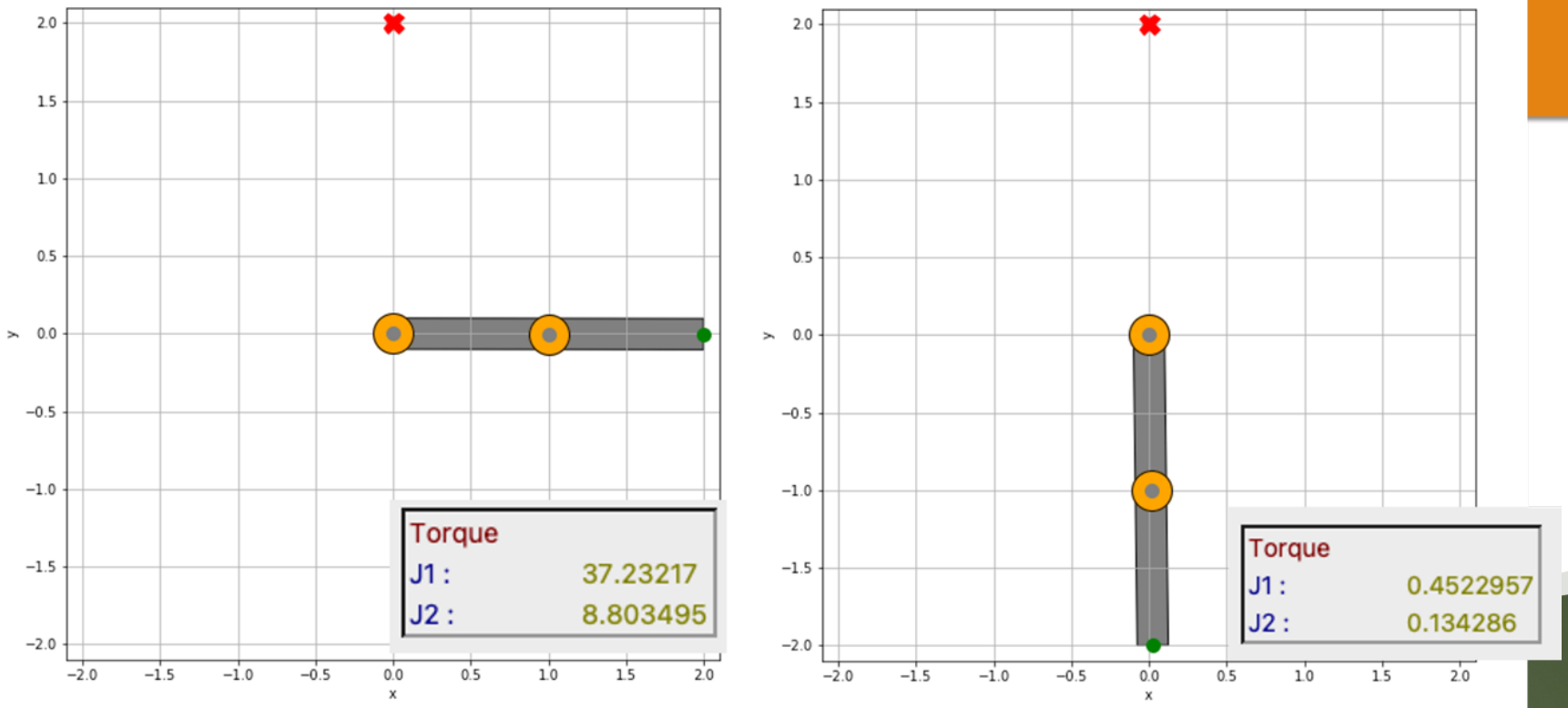
KINEMATICS OF 2-LINK MANIPULATOR



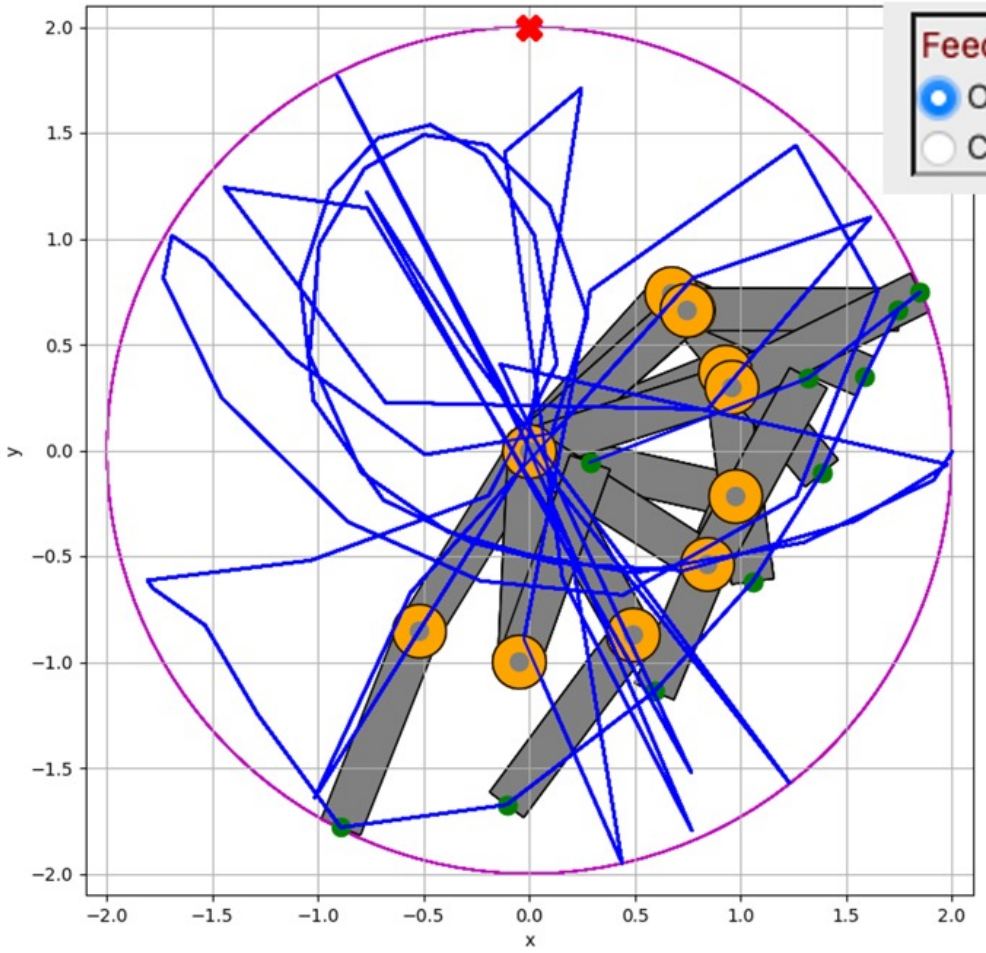
r2 GUI for 2-link manipulator

r2_iot.ipynb

fill in NETPIE information on line 274 - 276



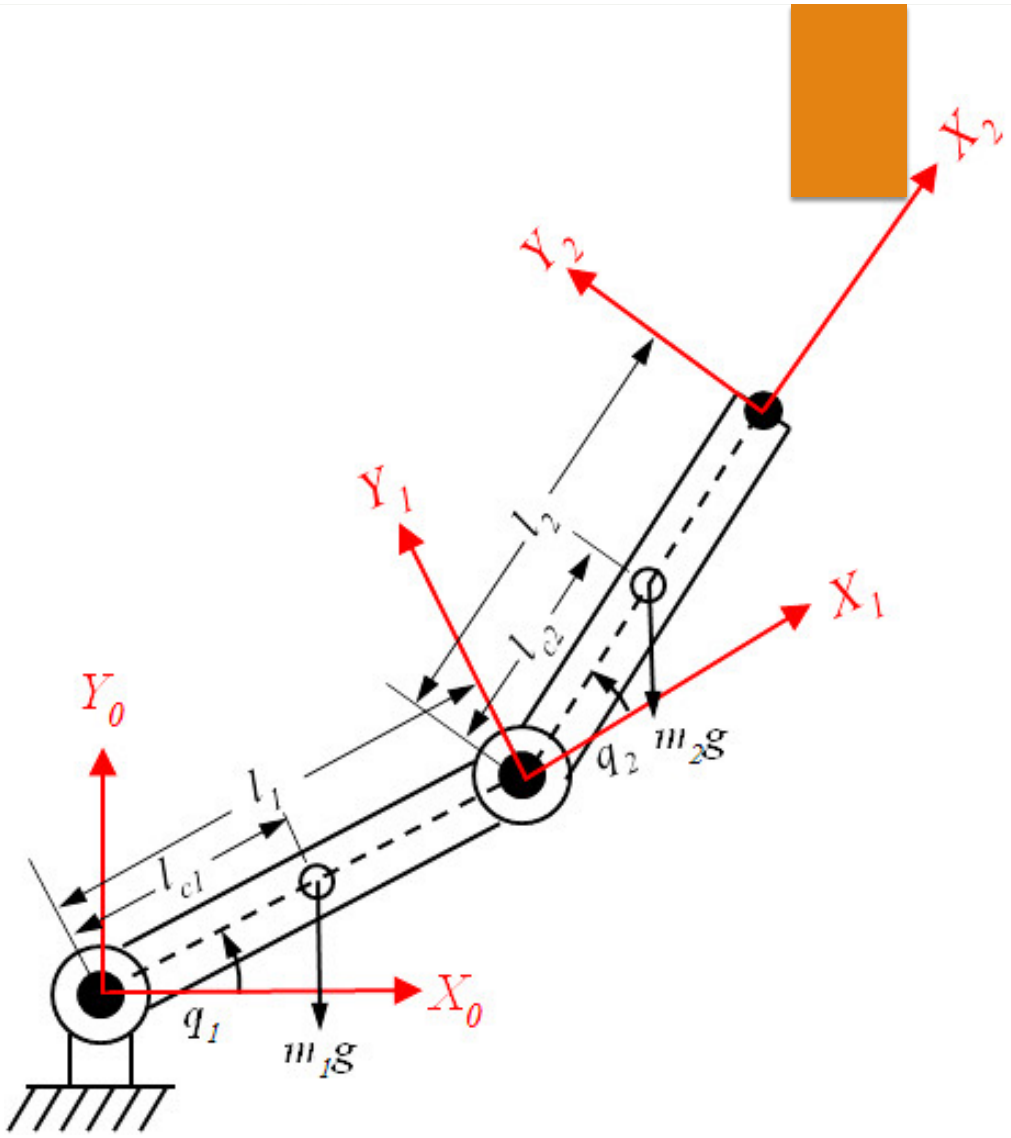
Joint torque depends on robot configuration



Manipulator behavior without controller

2-link manipulator kinematics

- forward kinematics
- inverse kinematics



Forward Kinematics

- Map joint angles q_1, q_2 to x,y and orientation of end-effector
- Code in `animate()` of r2 GUI

$$x = l_1 c_1 + l_2 c_{12}$$

$$y = l_1 s_1 + l_2 s_{12}$$

$$c_i = \cos(q_i), s_i = \sin(q_i), c_{ij} = \cos(q_i + q_j), s_{ij} = \sin(q_i + q_j)$$

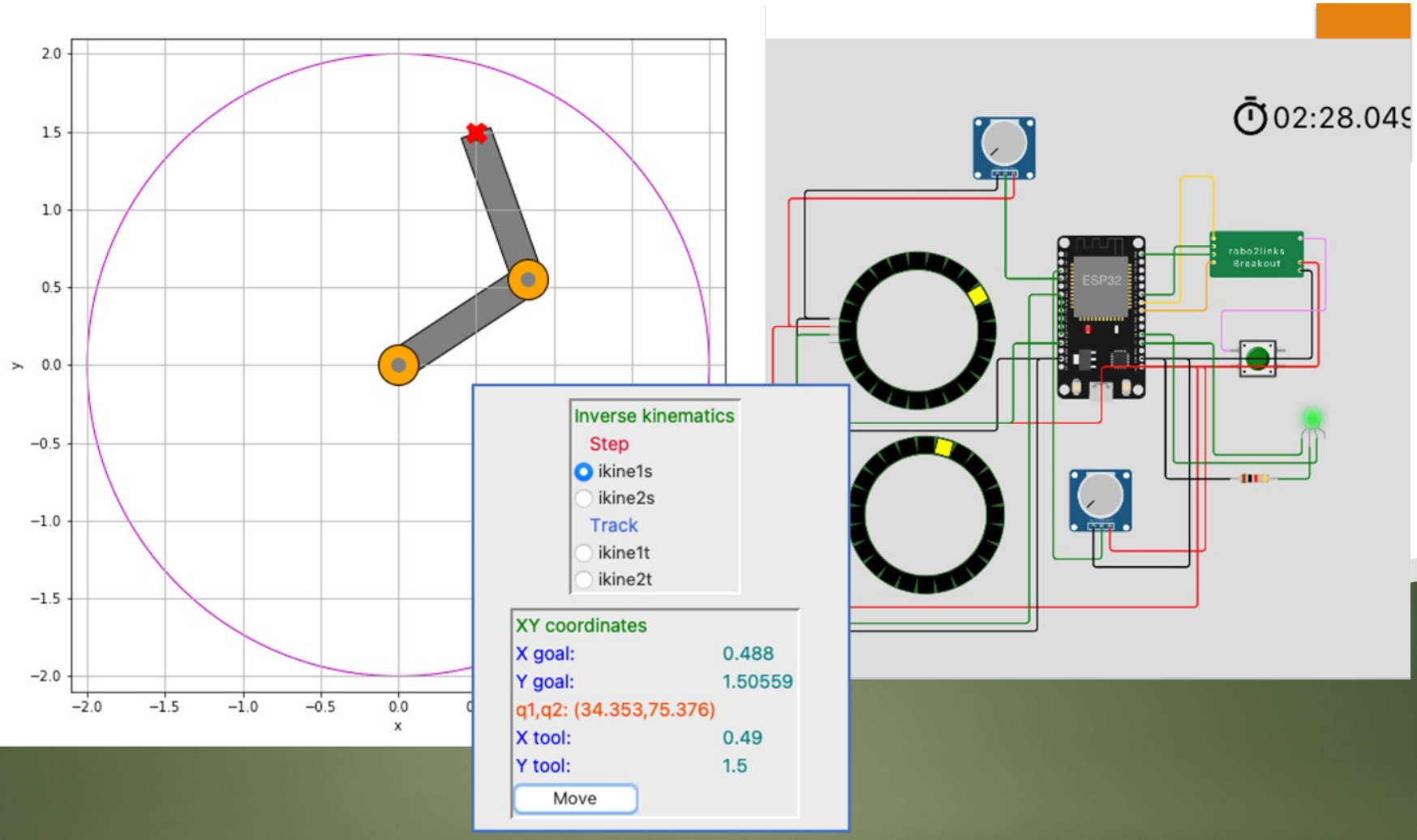
Cases of inverse kinematics problem

- no solution
- unique solution
- multiple solutions

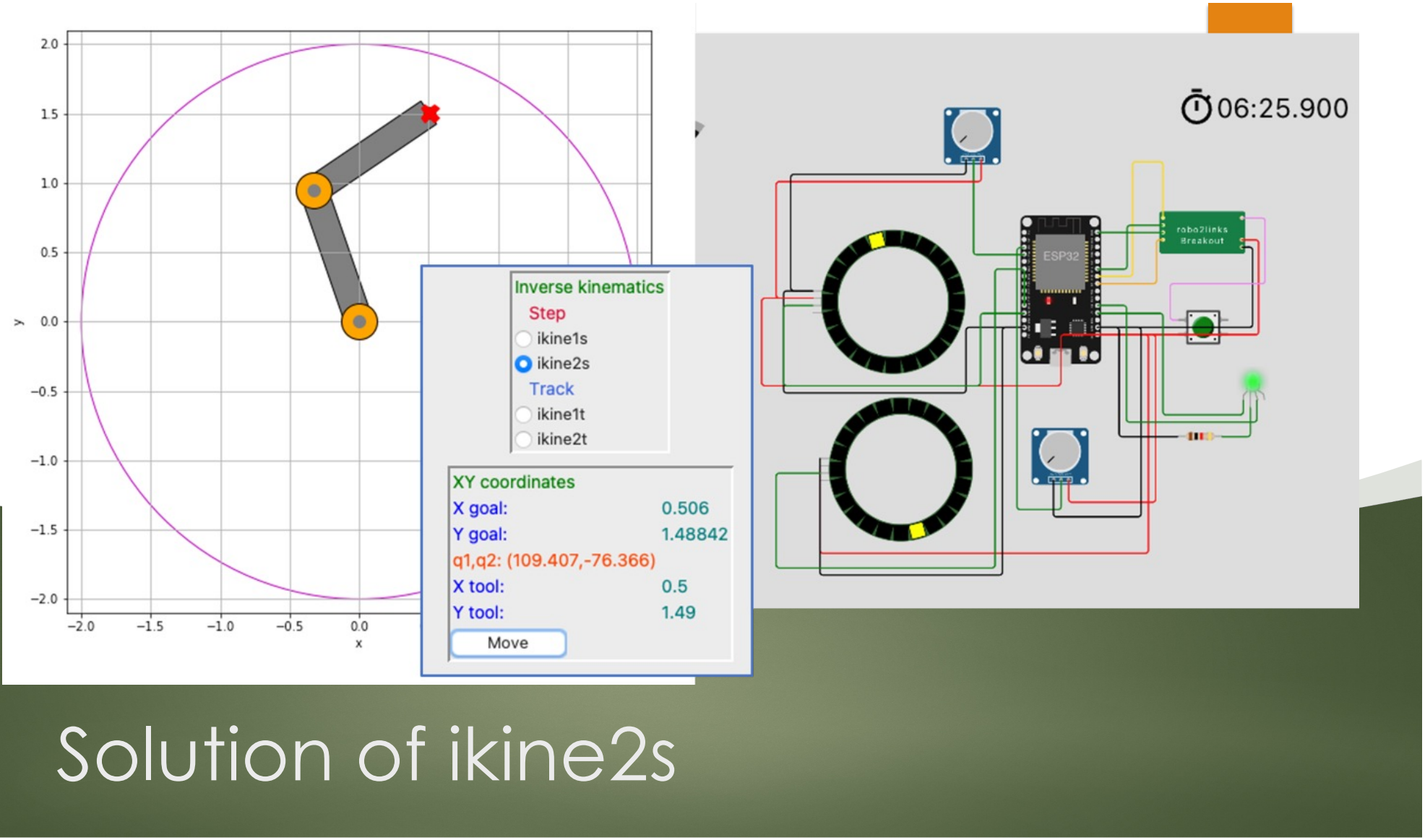
```
    def ikine(self, *args):
        # first check if solution exist
        c2 = (self.xg**2 + self.yg**2 - self.l1**2
              - self.l2**2)/(2*self.l1*self.l2)
        if abs(c2)>1.0: # no solution
            self.issolution = False
            self.ikmsg_txt.set("No solution")
        else:
            self.issolution = True
            self.iktype = self.iktype_txt.get() # ikine types
            # ikine1s or ikine1t
            if self.iktype == '0' or self.iktype == '2':
                s2 = np.sqrt(1 - c2**2)
            else: # ikine2s or ikine2t
                s2 = -np.sqrt(1 - c2**2)
            q2g = np.arctan2(s2,c2)
            k1 = self.l1 + self.l2*c2
            k2 = self.l2*s2
            q1g = np.arctan2(self.yg,self.xg)-np.arctan2(k2,k1)
            self.q1g = self.rad2deg*q1g
            self.q2g = self.rad2deg*q2g
            self.ikmsg_txt.set("q1,q2: ("+str(round(self.q1g,3)) +
                               "," + str(round(self.q2g,3)) + ")")
```

ikine()

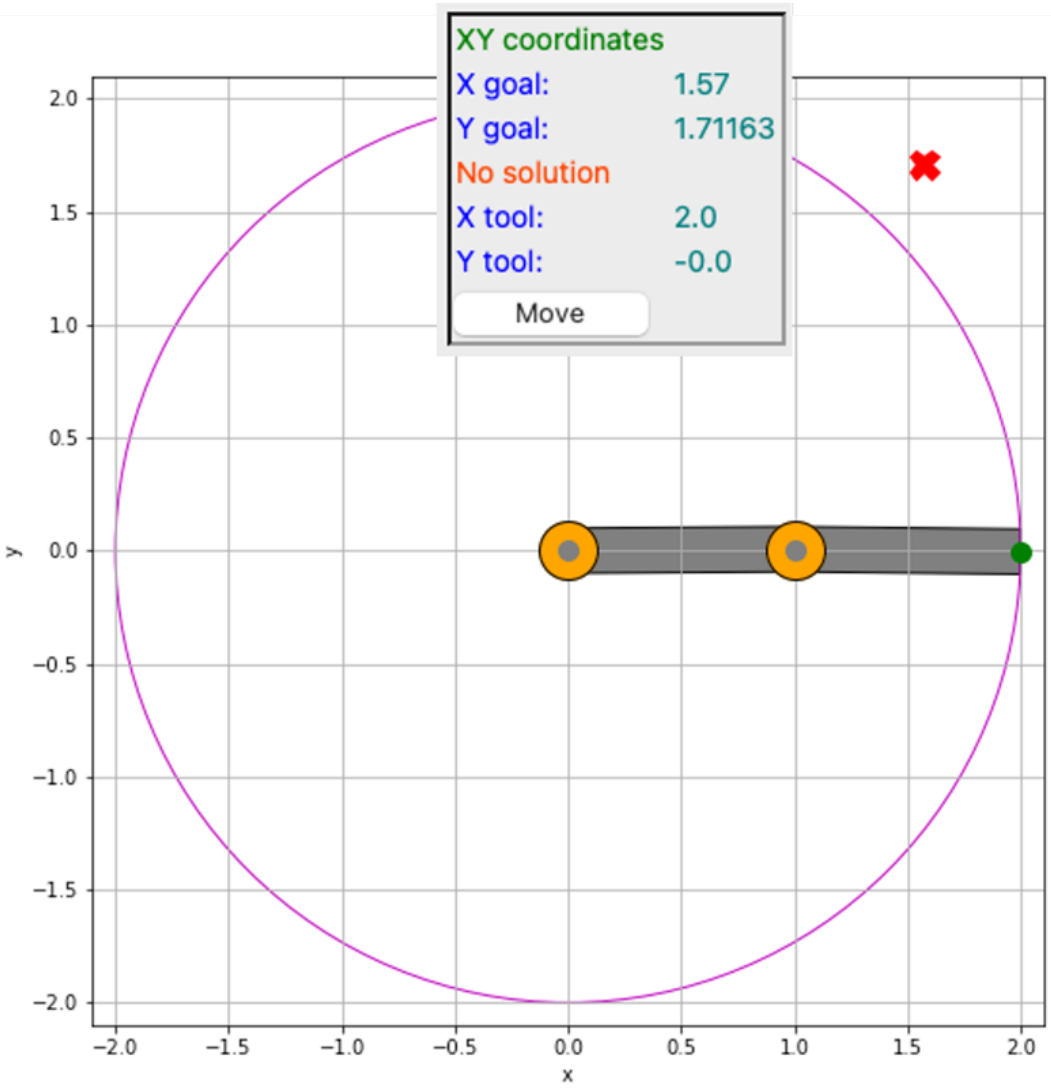
inverse kinematics computation



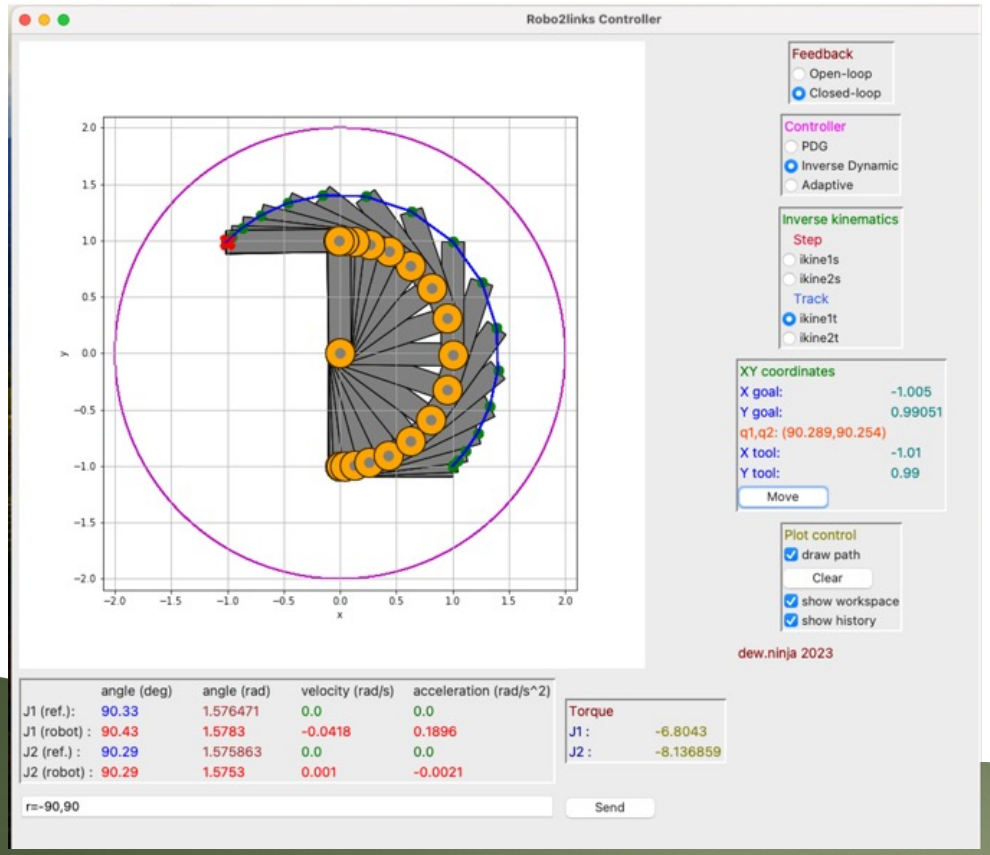
inverse kinematics solution ikine1s

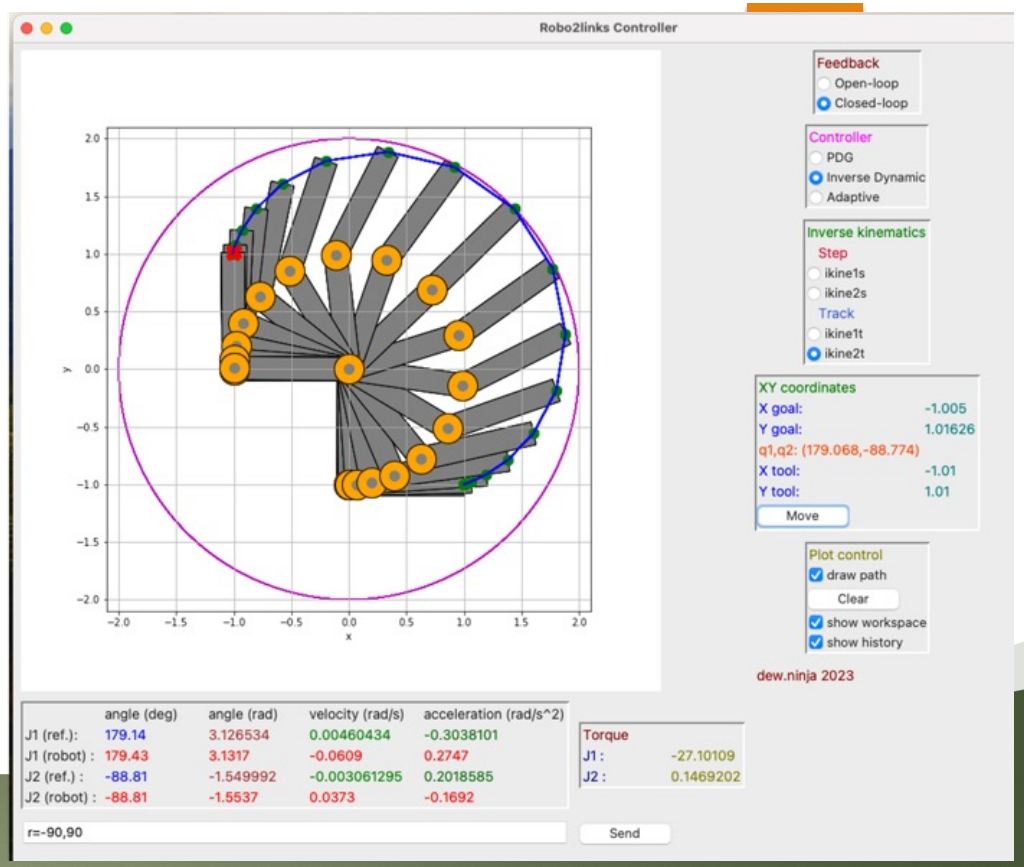


Solution of ikine2s

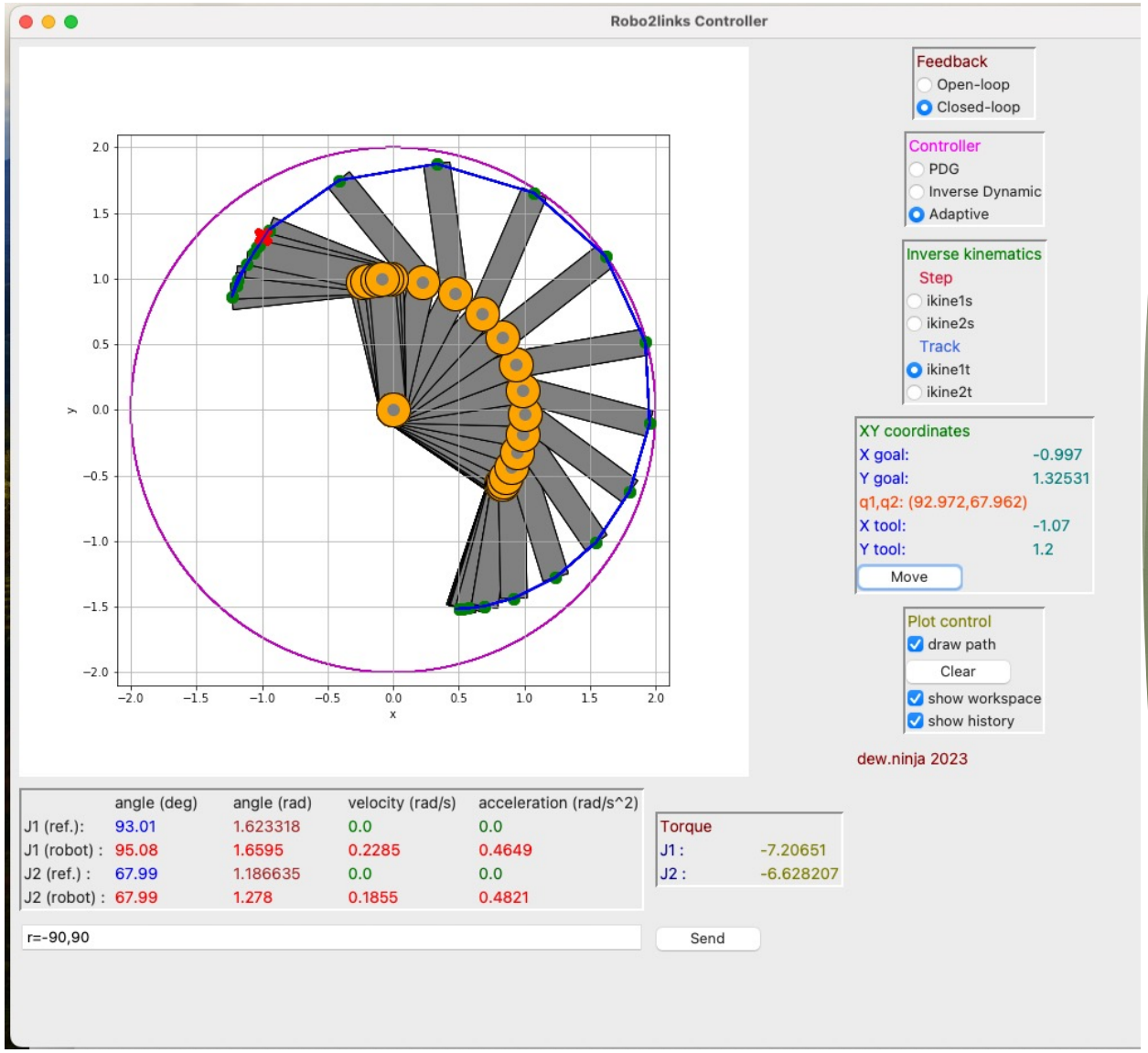


No solution with x,y outside robot work area





Tracking result from ikine1t and ikine2t



Tracking with adaptive control