Robotics
Exercise 4

Lecturer: Jim Mainprice
TAs: Philipp Kratzer, Janik Hager, Yoojin Oh
Machine Learning & Robotics lab, U Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany

May 26, 2021

1 Fun with Euler-Lagrange (6 points)

Consider an inverted pendulum mounted on a wheel in the 2D x-z-plane; similar to a Segway. The exercise is to derive the Euler-Lagrange equation for this system. Strictly follow the scheme we discussed on slide 03:23.

a) Describe the **pose** p_i of every body (depending on q) in (x, z, ϕ) coordinates: its position in the x-z-plane, and its rotation ϕ relative to the world-vertical. (2P)

b) Describe the (linear and angular) velocity v_i of every body. (1P)

c) Formulate the kinetic energy T. (1P)

d) Formulate the potential energy U. (1P)

e) Compute the Euler-Lagrange Equation: (1P)

$$\tau = \frac{d}{dt}\frac{\partial L}{\partial \dot{q}} - \frac{\partial L}{\partial q}$$

Tips:

- Use the generalized coordinates

$$q = (x, \theta) \tag{1}$$

  where x is the position of the wheel and θ the angle of the pendulum relative to the world-vertical.

- The system can be parameterized by
  - m_A, I_A, m_B, I_B: masses and inertias of bodies A (=wheel) and B (=pendulum)
  - r: radius of the wheel
  - l: length of the pendulum (height of its COM)

- In this 3-dim space, the mass matrix of every body is

$$M_i = \begin{pmatrix} m_i & 0 & 0 \\ 0 & m_i & 0 \\ 0 & 0 & I_i \end{pmatrix} \tag{2}$$

  (this allows to compute the kinetic energy of a body i with $\frac{1}{2}v_i^\top M_i v_i$)

2 Local linearization (4 points)

Equations (1) and (2) describe the cart pole dynamics, which is similar to the Segway-type system of Exercise 4, but a little simpler. The state of the cart-pole is given by $x = (p, \dot{p}, \theta, \dot{\theta})$, with $p \in \mathbb{R}$ the position of the cart, $\theta \in \mathbb{R}$

the pendulums angular deviation from the upright position and $\dot{p}, \dot{\theta}$ their respective temporal derivatives. The only control signal $u \in \mathbb{R}$ is the force applied on the cart. The analytic model of the cart pole is

$$\ddot{\theta} = \frac{g\sin(\theta) + \cos(\theta)\left[-c_1 u - c_2 \dot{\theta}^2 \sin(\theta)\right]}{\frac{4}{3}l - c_2 \cos^2(\theta)} \tag{3}$$

$$\ddot{p} = c_1 u + c_2 \left[\dot{\theta}^2 \sin(\theta) - \ddot{\theta}\cos(\theta)\right] \tag{4}$$

with $g = 9.8ms^2$ the gravitational constant, $l = 1m$ the pendulum length and constants $c_1 = (M_p + M_c)^{-1}$ and $c_2 = lM_p(M_p + M_c)^{-1}$ where $M_p = M_c = 1kg$ are the pendulum and cart masses respectively.

Derive the local linearization of these dynamics around $x^* = (0, 0, 0, 0)$. The eventual dynamics should be in the form

$$\dot{x} = Ax + Bu$$

Note that

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ \frac{\partial \ddot{p}}{\partial p} & \frac{\partial \ddot{p}}{\partial \dot{p}} & \frac{\partial \ddot{p}}{\partial \theta} & \frac{\partial \ddot{p}}{\partial \dot{\theta}} \\ 0 & 0 & 0 & 1 \\ \frac{\partial \ddot{\theta}}{\partial p} & \frac{\partial \ddot{\theta}}{\partial \dot{p}} & \frac{\partial \ddot{\theta}}{\partial \theta} & \frac{\partial \ddot{\theta}}{\partial \dot{\theta}} \end{pmatrix}, \quad B = \begin{pmatrix} 0 \\ \frac{\partial \ddot{p}}{\partial u} \\ 0 \\ \frac{\partial \ddot{\theta}}{\partial u} \end{pmatrix}$$

where all partial derivatives are taken at the point $p = \dot{p} = \theta = \dot{\theta} = 0$.

3 Multiple task variables (6 points)

1. To make sure you have an updated version of the repository, run '*git pull*' and '*git submodule update*'
2. For python you can run: *'jupyter-notebook course1-Lectures/02-kinematics/kinematics.ipynb'*
3. For C++ run: *'cd course1-Lectures/02-kinematics'*, *'make'*, *'./x.exe -mode 4'*
4. To understand how to access and use the code, refer to https://github.com/MarcToussaint/robotics-course/blob/master/tutorials/2-features.ipynb

The code contains the solution to last week's exercise where the robot moved his hand in a circle.

With multiple tasks, you can stack each objective and its Jacobian to Phi and PhiJ using *Phi.append(objective)* (C++) or *np.vstack((Phi, objective))* (Python)

a) We've seen that the solution does track the circle nicely, but the trajectory "gets lost" in joint space, leading to very strange postures. We can fix this by adding more tasks, esp. a task that permanently tries to (moderatly) minimize the distance of the configuration q to a natural posture q_{home}. Realize this by adding a respective task. (2P)

b) Make the robot simultaneously point upward with the left hand. Use the following commands to compute the orientation of a robot shape for this. (2P)

For C++:

```
arr y, J;
K.evalFeature(y, J, FS_vectorZ, {"handL"});
```

For Python:

```
F = K.feature(lry.FS.vectorZ, ["handL"])
y, J = F.eval(K)
```

c) Use the task spaces from Exercise 02 question 2.c) to add a task such that the robot looks with his right eye towards the right hand. You can use the following task spaces and the derivative to calculate the Jacobian. (2P)

$$\phi_1(q) = \phi_{h,g}^{\textsf{vec}} - \frac{x^W - \phi_{h,e}^{\textsf{pos}}}{|x^W - \phi_{h,e}^{\textsf{pos}}|} \quad \in \mathbb{R}^3$$

$$\phi_2(q) = \phi_{h,g}^{\textsf{vec}\top}\Big[\frac{x^W - \phi_{h,e}^{\textsf{pos}}}{|x^W - \phi_{h,e}^{\textsf{pos}}|}\Big] - 1 \quad \in \mathbb{R}^1$$

$$
\begin{aligned}
\partial_x \frac{f(x)}{|f(x)|} &= \partial_x \frac{f(x)}{[f(x)^\top f(x)]^{\frac{1}{2}}} \\
&= \frac{1}{[f(x)^\top f(x)]^{\frac{1}{2}}} \partial_x f(x) + f(x) \partial_x \frac{1}{[f(x)^\top f(x)]^{\frac{1}{2}}} \\
&= \frac{1}{[f(x)^\top f(x)]^{\frac{1}{2}}} \partial_x f(x) + f(x) \frac{-1/2}{[f(x)^\top f(x)]^{3/2}} [2 f(x)^\top \partial_x f(x)] \\
&= \frac{1}{|f(x)|} \Big[\mathbf{I} - \frac{f(x) f(x)^\top}{f(x)^\top f(x)}\Big] \partial_x f(x)
\end{aligned}
$$

You can use the following frame to access the right eye.
For C++:

```
K.evalFeature(y, J, FS_vectorZ, {"eyeR"});
```

For Python:

```
F = K.feature(lry.FS.vectorZ, ["eyeR"])
y, J = F.eval(K)
```

4 Controlling an arm to follow a trajectory (8 points)

1. Please update the version of the repository by running the following.

   ```
   git pull
   git submodule update
   make -C rai dependAll
   make -j4
   ```

2. For python you can run: *'jupyter-notebook course1-Lectures/03-dynamics/dynamics.ipynb'*
3. For C++ run: *'cd course1-Lectures/03-dynamics'*, *'make'*, *'./x.exe'*

In the main.cpp we provide a dynamics simulation of a robot arm that simulates the system for $T = 5$ seconds. The task is to write a controller that moves the arm from the initial position to a desired position $q^*_T = 0$ and velocity $\dot q^*_T = 0$ with a sine motion profile (see slide 02:40)

$$q^*_t = q_0 + \frac{1}{2}[1 - \cos(\pi t/T)](q^*_T - q_0).$$

a) Compute the desired velocity $\dot q^*_t$ and acceleration $\ddot q^*_t$ of the motion profile. (2P)
b) Implement the motion profile in the for-loop of the code, such that at each time step t a desired position, velocity and acceleration is available. (1P)
c) Implement direct PD control to follow the trajectory and try to find parameters K_p and K_d (potentially different for each joint) to reach the goal position. (2P)
d) Try to do the same with a PID controller that also includes the integral error (1P)

$$u = K_p(q^* - q) + K_d(\dot q^* - \dot q) + K_i \int_{s=0}^{t} (q^* - q(s))\ ds\ .$$

e) Now use the knowledge of M and F (slide 03:30) in each time step. The matrices M and F of the current robot state can be computed using 'M, F = K.equationOfMotion()' (python) or 'K.equationOfMotion(M, F)' (C++). Use inverse dynamics feedforward control to determine the necessary u (slide 03:34). (2P)

Optional:
Try different starting positions.
Try the same controllers for the arm in `pegArm2.ors`.
Play with `noise` parameter and check stability.