Lab: Forearm movements and LQR model

In this lab, you are going to model human movement data using a computational model from optimal control theory known as the Linear Quadratic Regulator (LQR). You will use data from an experiment conducted in 2011 where participants were instructed to aim towards a very thin line, adopting 5 different strategies, from very fast to very precise. This experiment captures the so-called speed-accuracy tradeoff of human movement, where usually more precise movements will be slower.

This lab has 4 parts: the first part is about basic manipulations of the data, and visualizing that data. The second and third part are "courses", and we will treat them together. The fourth part is about applying what you just learned in the course.

1 Visualizing and pre-processing the data

You will start by using `two_movements.csv`, which is a sample of the dataset with two movements only.

Task 1. *Open the file with any text editor to see how the file is formatted. Then, write a Python script to load movement data into a list.*

Help.

```
# the file is a csv file with ; as separator
with open(path, "r") as _file:  # open file in read ('r') mode
    for line in _file:  # iterate over each line
        values = line.split(';')# to get values from line, use line.split(';')
```

Task 2. *If you look at the time component of the signal, you notice that the sample times are uneven. This will cause problems, so you first need to resample the data to have equally spaced samples. Do that.*

Help. *Use a discretization of 10 ms and NumPy's interp function.*

Task 3. *You can now drop the time signal, which is not informative anymore. Now, compute the average of the two movements that you extracted.*

Help. *The two movements don't have the same length, what should you do?*

You should do for the whole dataset what you have just done for these two movements. To gain some time, you are provided with a pre-processed dataset, see file `extracted_data_extended`.

Task 4. *Load this file, visualize, and briefly describe the dataset. Also display the average trajectory for each condition.*

Help. *Use pickle's load method to load the file. To display graphs, you can use Matplotlib. To display several subplots on the same figure, have a look at matplolib's subplots method. The pickle file when loaded returns a Python dictionnary, where each key is the condition of the experiment and each value is an array with all interpolated and extended trajectories for that condition.*

```
# Pickle
import pickle
with open("extracted data_extended", "rb") as handle:
    dataset = pickle.load(handle, protocol=pickle.HIGHEST_PROTOCOL)
## Matplotlib
import matplotlib.pyplot as plt
fig, axs = plt.subplots(nrows=2, ncols=3)
axs[0,0].plot( ..., ..., ...)
```

Task 5. *For each condition, determine by the eye when the average movement ends (*i.e.*, when it levels off).*

This ends the first phase of the lab, where hopefully you have learned some basic data manipulation and display. We now turn to the modelling part.

2 Single-joint forearm dynamics

To describe forearm movements, we consider the simplified representation in Fig. 1 for flat movements produced *e.g.*, on a table.

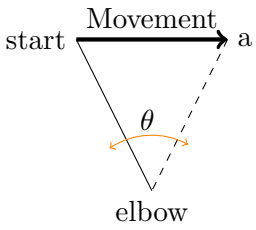

Figure 1: Simple forearm model

We consider that the effect of the neural system controlling the movement is to apply a torque τ around the elbow joint. We also sum up the effects of the various internal and external frictions as a speed-dependent braking torque ($\tau_{\text{break}} = b\dot{\theta}$). Then, using Newton's second law for rotations, we have

$$I\ddot{\theta} = \tau - \tau_{\text{break}}$$
$$I\ddot{\theta} + b\dot{\theta} = \tau$$

with

- Moment of inertia $I = 0.25$ kg.m^2
- effective viscosity $b = 0.2\,\text{kg.m}^2\text{.s}^{-1}$
- torque τ in N.m

Task 6. *Put our system in the matrix form $\dot{x} = Fx + Gu$, with $x = \begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix}$ and $u = \tau$. This form is useful because it is the general form of a linear system, for which many results exist.*

Help. *Write up the equations that link $\dot{\theta}$ and $\ddot{\theta}$ to θ and $\dot{\theta}$, and "stack them" to form the matrix representation.*

You now have a model of the forearm expressed in a canonical linear form. The problem is that it is a continuous time representation, whereas on our computer we will use a discrete time representation.

Task 7. *Use the Euler forward approximation $\dot{x}(t) = \frac{x[(k+1)T] - x[kT]}{T}$ to find the equivalents of F and G in discrete time.*

Help. *Plug the Euler forward approximation into the matrix form, and put in the form $x[(k+1)T] = Ax[kT] + Bu[k]$*

We have now put the system in a form that lets us use the general theory of LQR.

3 Linear Quadratic Regulator

The main assumption we make is that the emergent human behavior is the one that minimizes a cost (to be defined later). In the theory of control systems, we call the type of control that is designed to minimize a cost *optimal control*. We will now go over some results of optimal control.

Consider a generic discrete-time linear system, that we wish to control over N timesteps:

$$x_{k+1} = Ax_k + Bu_k$$
$$x_0 = x_{\text{init}}$$

In controlling it, we want to specify some costs. For example, we would like to make sure the movement ends up close to the target, so we might wish to place a penalty on the final state of the system. If we specify the costs as quadratic costs, the theory becomes much simpler. Therefore, we consider the following costs:

- a cost associated with the end state $x_N^T Q_F x_N$ (*e.g.*, have the speed $\dot{\theta} \sim 0$ at the end of movement, that signals the movement is indeed finished)
- a cost associated with the state $\sum_{k=0}^{N} x_k^T Q x_k$ (*e.g.*, avoid $\dot{\theta}$ getting too high, because there is a physiological speed limit)
- a cost that keeps the control signal "in check" (keep $\sum_{k_0}^{N} u_k^T R u_k$ small) *e.g.*, to avoid high torques.

Here, Q, Q_F, and R are positive definite matrices, and the costs are called quadratic (in one dimension, the cost for end state would be *e.g.*, $q_F x_N^2$, so Q, Q_F, and R are just weights that tell us how we value a cost on this or that component.)

We define our quadratic cost function as

$$J(u_0, u_1, \ldots, u_{N-1}) = \sum_{k=0}^{N-1} (x_k^T Q x_k + u_k^T R u_k) + x_N^T Q_F x_N,$$

and the problem is to determine $(u_0, u_1, \ldots, u_{N-1})$ that minimizes $\mathcal{J}$, subject to the state transition function:

$$\min_{(u_0, u_1, \ldots, u_{N-1})} J \quad \text{subject to } x_{k+1} = Ax_k + Bu_k$$

You may wonder why we picked quadratic costs, and not something else? The reason is that selecting a quadratic cost leads to a very simple feedback control law. To get an intuition why, we write the minimization problem for just one step

$$\min_{u_0} \mathcal{J} = x_1^T Q x_1 + u_0^T R u_0 \quad \text{subject to } x_1 = Ax_0 + Bu_0$$

If we derive with respect to u_0 and then set it to 0, we find the optimal u_0^*

$$\frac{\partial J}{\partial u_0} = 2x_1^T Q \frac{\partial x_1}{\partial u_0} + 2u_0^T R = 0$$
$$u_0^* = -(B^T Q B + R)^{-1} B^T Q A x_0$$

So, the optimal command is built by just "reading" the current state value (x_0), and applying some gain to it (indicated by the product of matrices before). This is very simple strategy, that is easy to implement, and matches well with the fact that feedback information is very important in human motor control.

Dynamic Programming solution We can solve the previous optimization problem via dynamic programming (DP), where we define the value function $V_k(z)$ associated with state z at timestep k:

$$V_k(z) = \min_{u_k,\ldots,u_{N-1}} \sum_{j=k}^{N-1} (x_j^T Q x_j + u_j^T R u_j) + x_N^T Q_F x_N,$$

subject to $x_k = z$, $x_{k+1} = Ax_k + Bu_k$. This is just a version of $\mathcal{J}$ that evaluates the remaining cost from some time step k if the current state is z.

Task 8. *Express $\mathcal{J}$ as a function of V.*

The DP principle (Bellman's Equation) says that the minimum cost-to-go from where you are ($V_k(z)$ is the minimum over the sum of the cost currently incurred (z^TQz) and the cost-to-go from where you end up ($V_{k+1}(Az + Bu_k)$)):

$$\begin{aligned} V_k(z) &= \min_{u_k}(z^TQz + u_k^TRu_k + V_{k+1}(Az + Bu_k)) \\ &= z^TQz + \min_{u_k}(u_k^TRu_k + V_{k+1}(Az + Bu_k)) \end{aligned}$$

and we are going to use this to solve our problem recursively. We first assume that $V_k(z)$ is quadratic *i.e.*, $\exists P_k > 0$ such that $V_k(z) = z^TP_kz$ (you can get the intuition why from the one step minimization for $\mathcal{J}$ that we wrote before). Then, we can write Bellman's equation as

$$V_k(z) = z^TQz + \min_{u_k}(u_k^TRu_k + (Az + Bu_k)^TP_{k+1}(Az + Bu_k))$$

Setting derivative with respect to u_k to zero, we get:

$$2u_k^TR + 2(z^TA^T + u_k^TB^T)P_{k+1}B = 0$$

which gives the control law:

$$u_k^*(z) = -[R + B^TP_{k+1}B]^{-1}B^TP_{k+1}Az$$

You can verify that plugging u_k^* back into Bellman's equation gives a quadratic $V_k(z)$, with

$$P_k = Q + A^TP_{k+1}A - A^TP_{k+1}B(R + B^TP_{k+1}B)^{-1}B^TP_{k+1}A$$

Help. *You can work out the computations by yourself on your spare time. You may find this document on matrix differentation useful.*

The next algorithm then gives the optimal control law.

Finite Horizon LQR

1. Set $P_N = Q_F$
2. For $k = N, \ldots, 1$:

$$P_k = Q + A^TP_{k+1}A - A^TP_{k+1}B(R + B^TP_{k+1}B)^{-1}B^TP_{k+1}A$$

3. For $t = 0, \ldots, N-1$:

$$K_k = -[R + B^TP_{k+1}B]^{-1}B^TP_{k+1}A$$

4. For $t = 0, \ldots, N-1$:

$$u_k^* = K_kx_k$$

4 LQR and Real Data

Task 9. *Write a function that "solves" the LQR (*i.e., *computes P, K and u) for any A, B, C, Q, Q_F, R.*

Help. *Implement the LQR algorithm above. In NumPy, the matrix product between A and B is* `A @ B` *and the transpose of a matrix A^T is* `A.T`*. The inverse of a matrix can be obtained by* ***`numpy.linalg.inv`***

Task 10. *Simulate trajectories that result from an optimal control law determined by the LQR, using $\rho = 10^{-3}$, and*

$$A = \begin{bmatrix} 1 & T \\ 0 & 1 - Tb/I \end{bmatrix}; \quad B = \begin{bmatrix} 0 \\ T/I \end{bmatrix}; \quad N = 100$$
$$y_k = Cx_k; \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix}; \quad Q_F = Q = C^T C; \quad R = \rho.$$

The simulation should return x, y, u and J. What is y? Plot y and u as a function of time.

Task 11. *Try out with multiple values of ρ and compare. Comment on the differences between solutions.*

Task 12. *Now, use the 5 average trajectories from the dataset, and determine ρ for each condition, by finding the value of ρ that produces a trajectory that best matches the average trajectory of that condition.*

Help. *To minimize a function, you can use* ***`scipy.optimize.minimize`****. You can minimize the norm between the two trajectories you are matching, using* ***`numpy.linalg.norm`****.*

Task 13. *Visualize the fitted trajectories on top of the average trajectories. How good is the fit? You can also play with values of Q and Q_F, and see if you can get better fits.*

5 Bonus

Task 14. *Plot the obtained gains. What do you notice? Change values of Q_f to see if your observation holds. What would happen if we picked $N \to \infty$?*

What you observed is called the steady state of the regulator: usually the P_k's converge fast towards a value P_{ss}.

To find P_{ss}, we "simply" solve the equation in the state where $P_k = P_{k+1} = P_{ss}$

$$P_{ss} = Q + A^T P_{ss} A - A^T P_{ss} B (R + B^T P_{ss} B)^{-1} B^T P_{ss} A$$

This is a so-called Discrete Algebraic Riccati Equation (DARE). There are various methods to solve it, but we will use an off-the-shelf solver.

Task 15. *Solve the DARE to find P_{SS}, and verify agreement with the list of P_k's.*

Help. *The* `control` *module has a method to solve the DARE. Install it with* `python3 -m pip install control`*.*

Now, you can simply replace the time variant P_k and K_k by P_{ss} and the associated K_{ss}.

Task 16. *Implement the infinite horizon solution, simulate trajectories, compare with the finite horizon solution and comment.*