Notes on AIMove exercises

Sylvain Calinon, Idiap Research Institute

1 Forward kinematics (FK) for a planar robot manipulator

The forward kinematics of a planar robot manipulator is defined as

$$\begin{aligned}\boldsymbol{f} &= \begin{bmatrix}\boldsymbol{l}^\top \cos(\boldsymbol{L}\boldsymbol{x}) \\ \boldsymbol{l}^\top \sin(\boldsymbol{L}\boldsymbol{x}) \\ \mathbf{1}^\top \boldsymbol{x}\end{bmatrix} \\ &= \begin{bmatrix}l_1\cos(x_1)+l_2\cos(x_1+x_2)+l_3\cos(x_1+x_2+x_3)+\dots \\ l_1\sin(x_1)+l_2\sin(x_1+x_2)+l_3\sin(x_1+x_2+x_3)+\dots \\ x_1+x_2+x_3+\dots\end{bmatrix},\end{aligned}$$

with $\boldsymbol{x}$ the state of the robot (joint angles), $\boldsymbol{f}$ the position of the robot end-effector, $\boldsymbol{l}$ a vector of robot links lengths, $\boldsymbol{L}$ a lower triangular matrix with unit elements, and $\mathbf{1}$ a vector of unit elements.

The position and orientation of all articulations can similarly be computed with the forward kinematics function

$$\begin{aligned}\tilde{\boldsymbol{f}} &= \Big[\boldsymbol{L}\ \mathrm{diag}(\boldsymbol{l})\cos(\boldsymbol{L}\boldsymbol{x}),\quad \boldsymbol{L}\ \mathrm{diag}(\boldsymbol{l})\sin(\boldsymbol{L}\boldsymbol{x}),\quad \boldsymbol{L}\boldsymbol{x}\Big]^\top \\ &= \begin{bmatrix}\tilde{f}_{1,1} & \tilde{f}_{1,2} & \tilde{f}_{1,3} & \dots \\ \tilde{f}_{2,1} & \tilde{f}_{2,2} & \tilde{f}_{2,3} & \dots \\ \tilde{f}_{3,1} & \tilde{f}_{3,2} & \tilde{f}_{3,3} & \dots\end{bmatrix},\end{aligned}$$

with

$$\begin{aligned}\tilde{f}_{1,1} &= l_1\cos(x_1),\\ \tilde{f}_{2,1} &= l_1\sin(x_1),\\ \tilde{f}_{3,1} &= x_1,\end{aligned}$$

$$\begin{aligned}\tilde{f}_{1,2} &= l_1\cos(x_1)+l_2\cos(x_1+x_2),\\ \tilde{f}_{2,2} &= l_1\sin(x_1)+l_2\sin(x_1+x_2),\\ \tilde{f}_{3,2} &= x_1+x_2,\end{aligned}$$

$$\begin{aligned}\tilde{f}_{1,3} &= l_1\cos(x_1)+l_2\cos(x_1+x_2)+l_3\cos(x_1+x_2+x_3),\\ \tilde{f}_{2,3} &= l_1\sin(x_1)+l_2\sin(x_1+x_2)+l_3\sin(x_1+x_2+x_3),\\ \tilde{f}_{3,3} &= x_1+x_2+x_3,\\ &\vdots\end{aligned}$$

In Python, this can be coded for the end-effector position part as

```
D = 3 #State space dimension (joint angles)
x = np.ones(D) * np.pi / D #Robot pose
l = np.array([2, 2, 1]) #Links lengths
L = np.tril(np.ones([D,D])) #Transformation matrix
f = np.array([L @ np.diag(l) @ np.cos(L @ x), L @ np.
    diag(l) @ np.sin(L @ x)]) #Forward kinematics
```

2 Inverse kinematics (IK) for a planar robot manipulator

The Jacobian corresponding to the end-effector forward kinematics function can be computed as (with a simplification for the orientation part by ignoring the manifold aspect)

$$\begin{aligned}\boldsymbol{J} &= \begin{bmatrix}-\sin(\boldsymbol{L}\boldsymbol{x})^\top \mathrm{diag}(\boldsymbol{l})\boldsymbol{L} \\ \cos(\boldsymbol{L}\boldsymbol{x})^\top \mathrm{diag}(\boldsymbol{l})\boldsymbol{L} \\ \mathbf{1}^\top\end{bmatrix} \\ &= \begin{bmatrix}J_{1,1} & J_{1,2} & J_{1,3} & \dots \\ J_{2,1} & J_{2,2} & J_{2,3} & \dots \\ J_{3,1} & J_{3,2} & J_{3,3} & \dots\end{bmatrix},\end{aligned}$$

with

$$\begin{aligned}J_{1,1} &= -l_1\sin(x_1)-l_2\sin(x_1+x_2)-l_3\sin(x_1+x_2+x_3)-\dots,\\ J_{2,1} &= l_1\cos(x_1)+l_2\cos(x_1+x_2)+l_3\cos(x_1+x_2+x_3)+\dots,\\ J_{3,1} &= 1,\end{aligned}$$

$$\begin{aligned}J_{1,2} &= -l_2\sin(x_1+x_2)-l_3\sin(x_1+x_2+x_3)-\dots,\\ J_{2,2} &= l_2\cos(x_1+x_2)+l_3\cos(x_1+x_2+x_3)+\dots,\\ J_{3,2} &= 1,\end{aligned}$$

$$\begin{aligned}J_{1,3} &= -l_3\sin(x_1+x_2+x_3)-\dots,\\ J_{2,3} &= l_3\cos(x_1+x_2+x_3)+\dots,\\ J_{3,3} &= 1,\\ &\vdots\end{aligned}$$

In Python, this can be coded for the end-effector position part as

```
J = np.array([-np.sin(L @ x).T @ np.diag(l) @ L, np.cos(
    L @ x).T @ np.diag(l) @ L]) #Jacobian (for end-
    effector)
```

Numerical estimation of the Jacobian

Section 2 presents an analytical solution for the Jacobian. A numerical solution can alternatively be estimated by computing

$$J_{i,j} = \frac{\partial f_i(\boldsymbol{x})}{\partial x_j} \approx \frac{f_i(\boldsymbol{x}^{(j)}) - f_i(\boldsymbol{x})}{\Delta} \quad \forall i, \forall j,$$

with $\boldsymbol{x}^{(j)}$ a vector of the same size as $\boldsymbol{x}$ with elements

$$x_k^{(j)} = \begin{cases} x_k + \Delta, & \text{if } k = j, \\ x_k, & \text{otherwise,}\end{cases}$$

where Δ is a small value for the approximation of the derivatives.

3 Linear quadratic tracking (LQT)

The LQT problem is formulated as the minimization of the cost

$$\begin{aligned} c &= (\boldsymbol{\mu}_T - \boldsymbol{x}_T)^\top \boldsymbol{Q}_T (\boldsymbol{\mu}_T - \boldsymbol{x}_T) \\ &\quad + \sum_{t=1}^{T-1} \Big((\boldsymbol{\mu}_t - \boldsymbol{x}_t)^\top \boldsymbol{Q}_t (\boldsymbol{\mu}_t - \boldsymbol{x}_t) \;+\; \boldsymbol{u}_t^\top \boldsymbol{R}_t \, \boldsymbol{u}_t \Big) \\ &= (\boldsymbol{\mu} - \boldsymbol{x})^\top \boldsymbol{Q} (\boldsymbol{\mu} - \boldsymbol{x}) \;+\; \boldsymbol{u}^\top \boldsymbol{R} \boldsymbol{u}, \end{aligned} \tag{1}$$

with $\boldsymbol{x} = \left[\boldsymbol{x}_1^\top, \boldsymbol{x}_2^\top, \dots, \boldsymbol{x}_T^\top\right]^\top$ the evolution of the state variable and $\boldsymbol{u} = \left[\boldsymbol{u}_1^\top, \boldsymbol{u}_2^\top, \dots, \boldsymbol{u}_{T-1}^\top\right]^\top$ the evolution of the control variable. $\boldsymbol{\mu} = \left[\boldsymbol{\mu}_1^\top, \boldsymbol{\mu}_2^\top, \dots, \boldsymbol{\mu}_T^\top\right]^\top$ represents the evolution of the tracking target. $\boldsymbol{Q} = \text{blockdiag}(\boldsymbol{Q}_1, \boldsymbol{Q}_2, \dots, \boldsymbol{Q}_T)$ represents the evolution of the required tracking precision, and $\boldsymbol{R} = \text{blockdiag}(\boldsymbol{R}_1, \boldsymbol{R}_2, \dots, \boldsymbol{R}_{T-1})$ represents the evolution of the cost on the control inputs.

The evolution of the system is linear, described by $\boldsymbol{x}_{t+1} = \boldsymbol{A}_t \boldsymbol{x}_t + \boldsymbol{B}_t \boldsymbol{u}_t$, yielding $\boldsymbol{x} = \boldsymbol{S}_{\boldsymbol{x}} \boldsymbol{x}_1 + \boldsymbol{S}_{\boldsymbol{u}} \boldsymbol{u}$ at trajectory level, see Appendix B for details.

The solution of (1) subject to $\boldsymbol{x} = \boldsymbol{S}_{\boldsymbol{x}} \boldsymbol{x}_1 + \boldsymbol{S}_{\boldsymbol{u}} \boldsymbol{u}$ is analytic, given by

$$\hat{\boldsymbol{u}} = \left(\boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{Q} \boldsymbol{S}_{\boldsymbol{u}} + \boldsymbol{R}\right)^{-1} \boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{Q} (\boldsymbol{\mu} - \boldsymbol{S}_{\boldsymbol{x}} \boldsymbol{x}_1).$$

The residuals of this least squares solution provides information about the uncertainty of this estimate, in the form of a full covariance matrix (at control trajectory level)

$$\hat{\boldsymbol{\Sigma}}^{\boldsymbol{u}} = \left(\boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{Q} \boldsymbol{S}_{\boldsymbol{u}} + \boldsymbol{R}\right)^{-1}.$$

4 iLQR optimization

Optimal control problems are defined by a cost function $\sum_{t=1}^{T} c(\boldsymbol{x}_t, \boldsymbol{u}_t)$ to minimize and a dynamical system $\boldsymbol{x}_{t+1} = \boldsymbol{g}(\boldsymbol{x}_t, \boldsymbol{u}_t)$ describing the evolution of a state $\boldsymbol{x}_t$ driven by control commands $\boldsymbol{u}_t$ during a time window of length T.

Iterative LQR (iLQR) solves such constrained nonlinear models by carrying out Taylor expansions on the cost and on the dynamical system so that a solution can be found iteratively by solving a LQR problem at each iteration. A solution in batch form can be computed by minimizing over $\boldsymbol{u} = \left[\boldsymbol{u}_1^\top, \boldsymbol{u}_2^\top, \dots, \boldsymbol{u}_{T-1}^\top\right]^\top$, yielding a series of open loop control commands $\boldsymbol{u}_t$, corresponding to a Gauss-Newton iteration scheme, see Appendix A for an overview of Newton's method for minimization. A solution can alternatively be computed in a recursive form to provide a controller with feedback gains. We consider first the batch iLQR form, as we primarily focus on the planning capability of the approach.

Batch iLQR employs a first order Taylor expansion of the dynamical system $\boldsymbol{x}_{t+1} = \boldsymbol{g}(\boldsymbol{x}_t, \boldsymbol{u}_t)$ around the point $(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)$, namely

$$\begin{aligned} \boldsymbol{x}_{t+1} &\approx \boldsymbol{g}(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_t}(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t) + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}_t}(\boldsymbol{u}_t - \hat{\boldsymbol{u}}_t) \\ \iff \Delta \boldsymbol{x}_{t+1} &\approx \boldsymbol{A}_t \Delta \boldsymbol{x}_t + \boldsymbol{B}_t \Delta \boldsymbol{u}_t, \end{aligned} \tag{2}$$

with error terms $\{\Delta \boldsymbol{x}_t = \boldsymbol{x}_t - \hat{\boldsymbol{x}}_t, \Delta \boldsymbol{u}_t = \boldsymbol{u}_t - \hat{\boldsymbol{u}}_t\}$, and Jacobian matrices $\{\boldsymbol{A}_t = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_t}, \boldsymbol{B}_t = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}_t}\}$.

The cost function $c(\boldsymbol{x}_t, \boldsymbol{u}_t)$ for time step t can similarly be approximated by a second order Taylor expansion around the point $(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)$, namely

$$\begin{aligned} c(\boldsymbol{x}_t, \boldsymbol{u}_t) &\approx c(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) + \Delta \boldsymbol{x}_t^\top \frac{\partial c}{\partial \boldsymbol{x}_t} + \Delta \boldsymbol{u}_t^\top \frac{\partial c}{\partial \boldsymbol{u}_t} + \\ &\frac{1}{2} \Delta \boldsymbol{x}_t^\top \frac{\partial^2 c}{\partial \boldsymbol{x}_t^2} \Delta \boldsymbol{x}_t + \Delta \boldsymbol{x}_t^\top \frac{\partial^2 c}{\partial \boldsymbol{x}_t \boldsymbol{u}_t} \Delta \boldsymbol{u}_t + \frac{1}{2} \Delta \boldsymbol{u}_t^\top \frac{\partial^2 c}{\partial \boldsymbol{u}_t^2} \Delta \boldsymbol{u}_t, \end{aligned} \tag{3}$$

with gradients $\{\frac{\partial c}{\partial \boldsymbol{x}_t}, \frac{\partial c}{\partial \boldsymbol{u}_t}\}$, and Hessian matrices $\{\frac{\partial^2 c}{\partial \boldsymbol{x}_t^2}, \frac{\partial^2 c}{\partial \boldsymbol{x}_t \boldsymbol{u}_t}, \frac{\partial^2 c}{\partial \boldsymbol{u}_t^2}\}$.

At a trajectory level, we denote $\boldsymbol{x} = \left[\boldsymbol{x}_1^\top, \boldsymbol{x}_2^\top, \dots, \boldsymbol{x}_T^\top\right]^\top$ the evolution of the state and $\boldsymbol{u} = \left[\boldsymbol{u}_1^\top, \boldsymbol{u}_2^\top, \dots, \boldsymbol{u}_{T-1}^\top\right]^\top$ the evolution of the control commands. The evolution of the state in (2) becomes $\Delta \boldsymbol{x} = \boldsymbol{S}_{\boldsymbol{u}} \Delta \boldsymbol{u}$, see Appendix B for details.[1]

The minimization problem can then be rewritten in batch form as

$$\begin{aligned} &\min_{\Delta \boldsymbol{u}} \Delta c(\Delta \boldsymbol{x}, \Delta \boldsymbol{u}), \quad \text{s.t.} \quad \Delta \boldsymbol{x} = \boldsymbol{S}_{\boldsymbol{u}} \Delta \boldsymbol{u}, \quad \text{where} \\ &\Delta c(\Delta \boldsymbol{x}, \Delta \boldsymbol{u}) = \Delta \boldsymbol{x}^\top \boldsymbol{g}_{\boldsymbol{x}} + \Delta \boldsymbol{u}^\top \boldsymbol{g}_{\boldsymbol{u}} + \\ &\frac{1}{2} \Delta \boldsymbol{x}^\top \boldsymbol{H}_{\boldsymbol{x}} \Delta \boldsymbol{x} + \Delta \boldsymbol{x}^\top \boldsymbol{H}_{\boldsymbol{x}\boldsymbol{u}} \Delta \boldsymbol{u} + \frac{1}{2} \Delta \boldsymbol{u}^\top \boldsymbol{H}_{\boldsymbol{u}} \Delta \boldsymbol{u}, \end{aligned} \tag{4}$$

with gradients $\boldsymbol{g}_{\boldsymbol{x}} = \frac{\partial c}{\partial \boldsymbol{x}}$, $\boldsymbol{g}_{\boldsymbol{u}} = \frac{\partial c}{\partial \boldsymbol{u}}$, and Hessian matrices $\boldsymbol{H}_{\boldsymbol{x}} = \frac{\partial^2 c}{\partial \boldsymbol{x}^2}$, $\boldsymbol{H}_{\boldsymbol{x}\boldsymbol{u}} = \frac{\partial^2 c}{\partial \boldsymbol{x}\boldsymbol{u}}$ and $\boldsymbol{H}_{\boldsymbol{u}} = \frac{\partial^2 c}{\partial \boldsymbol{u}^2}$.

By inserting the constraint into the cost, we obtain the optimization problem

$$\begin{aligned} \min_{\Delta \boldsymbol{u}} \quad & \Delta \boldsymbol{u}^\top \boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{g}_{\boldsymbol{x}} + \Delta \boldsymbol{u}^\top \boldsymbol{g}_{\boldsymbol{u}} + \frac{1}{2} \Delta \boldsymbol{u}^\top \boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{H}_{\boldsymbol{x}} \boldsymbol{S}_{\boldsymbol{u}} \Delta \boldsymbol{u} + \\ & \Delta \boldsymbol{u}^\top \boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{H}_{\boldsymbol{x}\boldsymbol{u}} \Delta \boldsymbol{u} + \frac{1}{2} \Delta \boldsymbol{u}^\top \boldsymbol{H}_{\boldsymbol{u}} \Delta \boldsymbol{u}, \end{aligned} \tag{5}$$

which can be solved analytically by differentiating with respect to $\Delta \boldsymbol{u}$ and equating to zero, namely,

$$\boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{g}_{\boldsymbol{x}} + \boldsymbol{g}_{\boldsymbol{u}} + \boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{H}_{\boldsymbol{x}} \boldsymbol{S}_{\boldsymbol{u}} \Delta \boldsymbol{u} + 2 \boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{H}_{\boldsymbol{x}\boldsymbol{u}} \Delta \boldsymbol{u} + \boldsymbol{H}_{\boldsymbol{u}} \Delta \boldsymbol{u} = 0, \tag{6}$$

providing the least squares solution

$$\Delta \hat{\boldsymbol{u}} = \left(\boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{H}_{\boldsymbol{x}} \boldsymbol{S}_{\boldsymbol{u}} + 2 \boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{H}_{\boldsymbol{x}\boldsymbol{u}} + \boldsymbol{H}_{\boldsymbol{u}}\right)^{-1} \left(-\boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{g}_{\boldsymbol{x}} - \boldsymbol{g}_{\boldsymbol{u}}\right), \tag{7}$$

which can be used to update the control commands estiamte at each iteration step of the iLQR algorithm.

4.1 iLQR with quadratic cost on $\boldsymbol{f}(\boldsymbol{x}_t)$

We consider a cost defined by

$$c(\boldsymbol{x}_t, \boldsymbol{u}_t) = \boldsymbol{f}(\boldsymbol{x}_t)^\top \boldsymbol{Q}_t \boldsymbol{f}(\boldsymbol{x}_t) + \boldsymbol{u}_t^\top \boldsymbol{R}_t \, \boldsymbol{u}_t, \tag{8}$$

where $\boldsymbol{Q}_t$ and $\boldsymbol{R}_t$ are weight matrices trading off task and control costs. Such cost is quadratic on $\boldsymbol{f}(\boldsymbol{x}_t)$ but non-quadratic on $\boldsymbol{x}_t$.

The cost in (3) then becomes

$$\begin{aligned} \Delta c(\Delta \boldsymbol{x}_t, \Delta \boldsymbol{u}_t) &\approx 2 \Delta \boldsymbol{x}_t^\top \boldsymbol{J}(\boldsymbol{x}_t)^\top \boldsymbol{Q}_t \boldsymbol{f}(\boldsymbol{x}_t) + 2 \Delta \boldsymbol{u}_t^\top \boldsymbol{R}_t \, \boldsymbol{u} + \\ &\Delta \boldsymbol{x}_t^\top \boldsymbol{J}(\boldsymbol{x}_t)^\top \boldsymbol{Q}_t \boldsymbol{J}(\boldsymbol{x}_t) \Delta \boldsymbol{x}_t + \Delta \boldsymbol{u}_t^\top \boldsymbol{R}_t \Delta \boldsymbol{u}_t, \end{aligned} \tag{9}$$

[1] Note that $\boldsymbol{S}_{\boldsymbol{x}} \Delta \boldsymbol{x}_1 = \boldsymbol{0}$ because $\Delta \boldsymbol{x}_1 = \boldsymbol{0}$ (as we want our motion to start from $\boldsymbol{x}_1$).

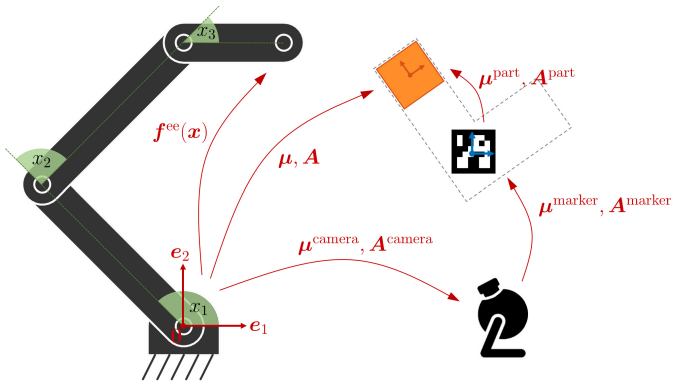

Figure 1: Typical transformations involved in a manipulation task involving a robot, a vision system, a visual marker on the object, and a desired grasping location on the object.

which used

$$\frac{\partial c}{\partial \boldsymbol{x}_t} = 2\boldsymbol{J}(\boldsymbol{x}_t)^\top \boldsymbol{Q}_t \boldsymbol{f}(\boldsymbol{x}_t), \quad \frac{\partial c}{\partial \boldsymbol{u}_t} = 2\boldsymbol{R}_t \boldsymbol{u}, \tag{10}$$

$$\frac{\partial^2 c}{\partial \boldsymbol{x}_t^2} \approx 2\boldsymbol{J}(\boldsymbol{x}_t)^\top \boldsymbol{Q}_t \boldsymbol{J}(\boldsymbol{x}_t), \quad \frac{\partial^2 c}{\partial \boldsymbol{x}_t \boldsymbol{u}_t} = \boldsymbol{0}, \quad \frac{\partial^2 c}{\partial \boldsymbol{u}_t^2} = 2\boldsymbol{R}_t. \tag{11}$$

with $\boldsymbol{J}(\boldsymbol{x}_t) = \frac{\partial \boldsymbol{f}(\boldsymbol{x}_t)}{\partial \boldsymbol{x}_t}$ a Jacobian matrix.

At a trajectory level, the evolution of the tracking and control weights is represented by $\boldsymbol{Q} = \text{blockdiag}(\boldsymbol{Q}_1, \boldsymbol{Q}_2, \dots, \boldsymbol{Q}_T)$ and $\boldsymbol{R} = \text{blockdiag}(\boldsymbol{R}_1, \boldsymbol{R}_2, \dots, \boldsymbol{R}_{T-1})$, respectively.

With a slight abuse of notation, we define $\boldsymbol{f}(\boldsymbol{x})$ as a vector concatenating the vectors $\boldsymbol{f}(\boldsymbol{x}_t)$, and $\boldsymbol{J}(\boldsymbol{x})$ as a block-diagonal concatenation of the Jacobian matrices $\boldsymbol{J}(\boldsymbol{x}_t)$. The minimization problem (5) then becomes

$$\min_{\Delta \boldsymbol{u}} \quad 2\Delta \boldsymbol{x}^\top \boldsymbol{J}(\boldsymbol{x})^\top \boldsymbol{Q} \boldsymbol{f}(\boldsymbol{x}) + 2\Delta \boldsymbol{u}^\top \boldsymbol{R}\, \boldsymbol{u} + \Delta \boldsymbol{x}^\top \boldsymbol{J}(\boldsymbol{x})^\top \boldsymbol{Q} \boldsymbol{J}(\boldsymbol{x}) \Delta \boldsymbol{x} + \Delta \boldsymbol{u}^\top \boldsymbol{R} \Delta \boldsymbol{u}, \quad \text{s.t.} \quad \Delta \boldsymbol{x} = \boldsymbol{S}_{\boldsymbol{u}} \Delta \boldsymbol{u}, \tag{12}$$

whose least squares solution is given by

$$\Delta \hat{\boldsymbol{u}} = \left(\boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{J}(\boldsymbol{x})^\top \boldsymbol{Q} \boldsymbol{J}(\boldsymbol{x}) \boldsymbol{S}_{\boldsymbol{u}} + \boldsymbol{R}\right)^{-1} \left(-\boldsymbol{S}_{\boldsymbol{u}}^\top \boldsymbol{J}(\boldsymbol{x})^\top \boldsymbol{Q} \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{R}\, \boldsymbol{u}\right), \tag{13}$$

which can be used to update the control commands estimates at each iteration step of the iLQR algorithm.

In the next sections, we show examples of functions $\boldsymbol{f}(\boldsymbol{x})$ that can rely on this formulation.

iLQR for robot manipulator

We define a manipulation task involving a set of transformations as in Fig. 1. By relying on these transformation operators, we will next describe all variables in the robot frame of reference (defined by $\boldsymbol{0}$, $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ in the figure).

For a manipulator controlled by joint angle velocity commands $\boldsymbol{u} = \dot{\boldsymbol{x}}$, the evolution of the system is described by $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \boldsymbol{u}_t \Delta t$, with the Taylor expansion (2) simplifying to $\boldsymbol{A}_t = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_t} = \boldsymbol{I}$ and $\boldsymbol{B}_t = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}_t} = \boldsymbol{I}\Delta t$. Similarly, a double integrator can alternatively be considered, with acceleration commands $\boldsymbol{u} = \ddot{\boldsymbol{x}}$ and states composed of both positions and velocities.

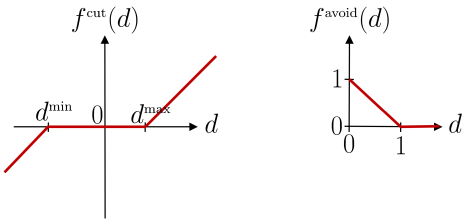

Figure 2: ReLU-like functions used in optimization costs. The derivatives of these functions are simple, providing Jacobians whose entries are either 0 or ± 1.

For a robot manipulator, $\boldsymbol{f}(\boldsymbol{x}_t)$ in (13) typically represents the error between a reference $\boldsymbol{\mu}_t$ and the end-effector position computed by the forward kinematics function $\boldsymbol{f}^{\text{ee}}(\boldsymbol{x}_t)$. We then have

$$\begin{aligned} \boldsymbol{f}(\boldsymbol{x}_t) &= \boldsymbol{f}^{\text{ee}}(\boldsymbol{x}_t) - \boldsymbol{\mu}_t, \\ \boldsymbol{J}(\boldsymbol{x}_t) &= \boldsymbol{J}^{\text{ee}}(\boldsymbol{x}_t). \end{aligned}$$

For the orientation part of the data (if considered), the Euclidean distance vector $\boldsymbol{f}^{\text{ee}}(\boldsymbol{x}_t) - \boldsymbol{\mu}_t$ is replaced by a geodesic distance measure computed with the logarithmic map $\log_{\boldsymbol{\mu}_t}\left(\boldsymbol{f}^{\text{ee}}(\boldsymbol{x}_t)\right)$.

The approach can similarly be extended to target objects/landmarks with positions $\boldsymbol{\mu}_t$ and orientation matrices $\boldsymbol{U}_t$, whose columns are basis vectors forming a coordinate system, see Fig. 3. We can then define an error between the robot end-effector and an object/landmark expressed in the object/landmark coordinate system as

$$\boldsymbol{f}(\boldsymbol{x}_t) = \boldsymbol{U}_t^\top \left(\boldsymbol{f}^{\text{ee}}(\boldsymbol{x}_t) - \boldsymbol{\mu}_t\right), \tag{14}$$

$$\boldsymbol{J}(\boldsymbol{x}_t) = \boldsymbol{U}_t^\top \boldsymbol{J}^{\text{ee}}(\boldsymbol{x}_t). \tag{15}$$

Bounded joint space

The iLQR solution in (13) can be used to keep the state within a boundary (e.g., joint angle limits). We denote $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}^{\text{cut}}(\boldsymbol{x})$ as the vertical concatenation of $\boldsymbol{f}^{\text{cut}}(\boldsymbol{x}_t)$ and $\boldsymbol{J}(\boldsymbol{x}) = \boldsymbol{J}^{\text{cut}}(\boldsymbol{x})$ as a diagonal concatenation of diagonal Jacobian matrices $\boldsymbol{J}^{\text{cut}}(\boldsymbol{x}_t)$. Each element i of $\boldsymbol{f}^{\text{cut}}(\boldsymbol{x}_t)$ and $\boldsymbol{J}^{\text{cut}}(\boldsymbol{x}_t)$ is defined as

$$f_i^{\text{cut}}(x_{t,i}) = \begin{cases} x_{t,i} - x_i^{\min}, & \text{if } x_{t,i} < x_i^{\min} \\ x_{t,i} - x_i^{\max}, & \text{if } x_{t,i} > x_i^{\max} \\ 0, & \text{otherwise} \end{cases},$$

$$J_{i,i}^{\text{cut}}(x_{t,i}) = \begin{cases} 1, & \text{if } x_{t,i} < x_i^{\min} \\ 1, & \text{if } x_{t,i} > x_i^{\max} \\ 0, & \text{otherwise} \end{cases},$$

where $f_i^{\text{cut}}(x_{t,i})$ describes continuous ReLU-like functions for each dimension. $f_i^{\text{cut}}(x_{t,i})$ is 0 inside the bounded domain and takes the signed distance value outside the boundary, see Fig. 2-*left*.

We can see with (10) that for $\boldsymbol{Q} = \frac{1}{2}\boldsymbol{I}$, if $\boldsymbol{x}$ is outside the domain during some time steps t, $\boldsymbol{g}_{\boldsymbol{x}} = \frac{\partial c}{\partial \boldsymbol{x}} = 2\boldsymbol{J}^\top \boldsymbol{Q} \boldsymbol{f}$ generates a vector bringing it back to the boundary of the domain.

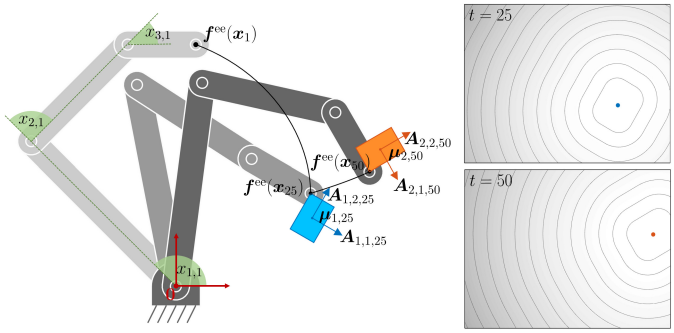

Figure 3: Example of a viapoints task in which a planar robot with 3 joints needs to sequentially reach 2 objects, with object boundaries defining the allowed reaching points on the objects surfaces. *Left:* Reaching task with two viapoints at $t = 25$ and $t = 50$. *Right:* Corresponding values of the cost function for the end-effector space at $t = 25$ and $t = 50$.

Bounded task space

The iLQR solution in (13) can be used to keep the end-effector within a boundary (e.g., end-effector position limits). Based on the above definitions, $\boldsymbol{f}(\boldsymbol{x})$ and $\boldsymbol{J}(\boldsymbol{x})$ are in this case defined as

$$\begin{aligned} \boldsymbol{f}(\boldsymbol{x}_t) &= \boldsymbol{f}^{\text{cut}}\Big(\boldsymbol{e}(\boldsymbol{x}_t)\Big), \\ \boldsymbol{J}(\boldsymbol{x}_t) &= \boldsymbol{J}^{\text{cut}}\Big(\boldsymbol{e}(\boldsymbol{x}_t)\Big)\ \boldsymbol{J}^{\text{ee}}(\boldsymbol{x}), \\ \text{with}\quad \boldsymbol{e}(\boldsymbol{x}_t) &= \boldsymbol{f}^{\text{ee}}(\boldsymbol{x}). \end{aligned}$$

Reaching task with robot manipulator and prismatic object boundaries

The definition of $\boldsymbol{f}(\boldsymbol{x}_t)$ and $\boldsymbol{J}(\boldsymbol{x}_t)$ in (14) and (15) can also be extended to objects/landmarks with boundaries by defining

$$\begin{aligned} \boldsymbol{f}(\boldsymbol{x}_t) &= \boldsymbol{f}^{\text{cut}}\Big(\boldsymbol{e}(\boldsymbol{x}_t)\Big), \\ \boldsymbol{J}(\boldsymbol{x}_t) &= \boldsymbol{J}^{\text{cut}}\Big(\boldsymbol{e}(\boldsymbol{x}_t)\Big)\ \boldsymbol{U}_t^{\top}\ \boldsymbol{J}^{\text{ee}}(\boldsymbol{x}_t), \\ \text{with}\quad \boldsymbol{e}(\boldsymbol{x}_t) &= \boldsymbol{U}_t^{\top}\big(\boldsymbol{f}^{\text{ee}}(\boldsymbol{x}_t) - \boldsymbol{\mu}_t\big), \end{aligned}$$

see also Fig. 3.

Appendices

A Newton's method for minimization

Newton's method attempts to solve $\min_x f(x)$ or $\max_x f(x)$ from an initial guess x_1 by using a sequence of second-order Taylor approximations of f around the iterates, see Fig. 4. The second-order Taylor expansion of f around x_k is

$$f(x_k + t) \approx f(x_k) + f'(x_k)\,t + \frac{1}{2}f''(x_k)\,t^2. \tag{16}$$

The next iterate $x_{k+1} = x_k + t$ is defined so as to minimize this quadratic approximation in t. If the second derivative is positive, the quadratic approximation is a convex function of t, and its minimum can be found by setting the derivative to zero. Since

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(f(x_k) + f'(x_k)\,t + \frac{1}{2}f''(x_k)\,t^2\right) = f'(x_k) + f''(x_k)\,t, \tag{17}$$

the minimum is achieved for

$$t = -\frac{f'(x_k)}{f''(x_k)}. \tag{18}$$

Newton's method thus performs the iteration

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}. \tag{19}$$

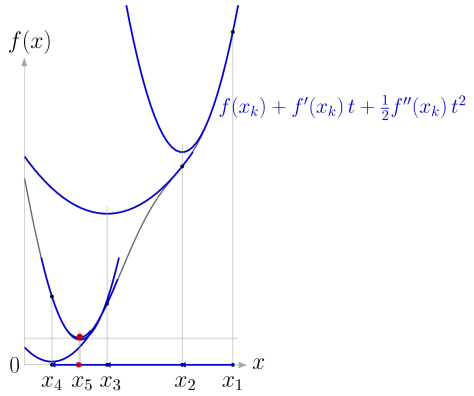

Figure 4: Newton's method for minimization, starting from an initial estimate x_1 and converging to a local minimum (red point) after 5 iterations.

The geometric interpretation of Newton's method is that at each iteration, it amounts to the fitting of a paraboloid to the surface of $f(x)$ at x_k, having the same slopes and curvature as the surface at that point, and then proceeding to the maximum or minimum of that paraboloid. Note that if f happens to be a quadratic function, then the exact extremum is found in one step. Note also that Newton's method is often modified to include a step size (e.g., estimated with line search).

The multidimensional case similarly provides

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{H}(\boldsymbol{x}_k)^{-1}\,\boldsymbol{g}(\boldsymbol{x}_k), \tag{20}$$

with $\boldsymbol{g}$ and $\boldsymbol{H}$ the gradient and Hessian matrix of f (vector and square matrix, respectively).

Gauss-Newton algorithm

The Gauss-Newton algorithm is a special case of Newton's method in which the cost is quadratic (sum of squared function values), with $f(\boldsymbol{x}) = \sum_{i=1}^{R} r_i^2(\boldsymbol{x}) = \boldsymbol{r}^{\top}\boldsymbol{r}$, and by ignoring the second-order derivative terms of the Hessian. The gradient and Hessian can in this case be computed with

$$\boldsymbol{g} = 2\boldsymbol{J}_{\boldsymbol{r}}^{\top}\boldsymbol{r}, \quad \boldsymbol{H} \approx 2\boldsymbol{J}_{\boldsymbol{r}}^{\top}\boldsymbol{J}_{\boldsymbol{r}}, \tag{21}$$

where $\boldsymbol{J}_{\boldsymbol{r}} \in \mathbb{R}^{R\times D}$ isthe Jacobian matrix of $\boldsymbol{r} \in \mathbb{R}^R$. The update rule then becomes

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k - \big(\boldsymbol{J}_{\boldsymbol{r}}^{\top}(\boldsymbol{x}_k)\boldsymbol{J}_{\boldsymbol{r}}(\boldsymbol{x}_k)\big)^{-1}\,\boldsymbol{J}_{\boldsymbol{r}}^{\top}(\boldsymbol{x}_k)\,\boldsymbol{r}(\boldsymbol{x}_k) && (22) \\ &= \boldsymbol{x}_k - \boldsymbol{J}_{\boldsymbol{r}}^{\dagger}(\boldsymbol{x}_k)\,\boldsymbol{r}(\boldsymbol{x}_k). && (23) \end{aligned}$$

B System dynamics at trajectory level

The evolution of a system $\boldsymbol{x}_{t+1} = g(\boldsymbol{x}_t, \boldsymbol{u}_t)$ (expressed in discrete form), can be approximated by the linear system

$$\boldsymbol{x}_{t+1} = \boldsymbol{A}_t(\boldsymbol{x}_t, \boldsymbol{u}_t)\, \boldsymbol{x}_t + \boldsymbol{B}_t(\boldsymbol{x}_t, \boldsymbol{u}_t)\, \boldsymbol{u}_t, \quad \forall t \in \{1, \dots, T\}$$

with states $\boldsymbol{x}_t \in \mathbb{R}^D$ and control commands $\boldsymbol{u}_t \in \mathbb{R}^d$.

With the above linearization, we can express all states $\boldsymbol{x}_t$ as an explicit function of the initial state $\boldsymbol{x}_1$. By writing

$$\begin{aligned}
\boldsymbol{x}_2 &= \boldsymbol{A}_1\boldsymbol{x}_1 + \boldsymbol{B}_1\boldsymbol{u}_1, \\
\boldsymbol{x}_3 &= \boldsymbol{A}_2\boldsymbol{x}_2 + \boldsymbol{B}_2\boldsymbol{u}_2 = \boldsymbol{A}_2(\boldsymbol{A}_1\boldsymbol{x}_1 + \boldsymbol{B}_1\boldsymbol{u}_1) + \boldsymbol{B}_2\boldsymbol{u}_2, \\
&\vdots \\
\boldsymbol{x}_T &= \left(\prod_{t=1}^{T-1} \boldsymbol{A}_{T-t}\right) \boldsymbol{x}_1 \; + \left(\prod_{t=1}^{T-2} \boldsymbol{A}_{T-t}\right) \boldsymbol{B}_1\boldsymbol{u}_1 + \\
&\qquad\qquad \left(\prod_{t=1}^{T-3} \boldsymbol{A}_{T-t}\right) \boldsymbol{B}_2\boldsymbol{u}_2 \; + \; \cdots \; + \; \boldsymbol{B}_{T-1}\boldsymbol{u}_{T-1},
\end{aligned}$$

in a matrix form, we get an expression of the form $\boldsymbol{x} = \boldsymbol{S}_{\boldsymbol{x}}\boldsymbol{x}_1 + \boldsymbol{S}_{\boldsymbol{u}}\boldsymbol{u}$, with

$$\underbrace{\begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{x}_3 \\ \vdots \\ \boldsymbol{x}_T \end{bmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{A}_1 \\ \boldsymbol{A}_2\boldsymbol{A}_1 \\ \vdots \\ \prod_{t=1}^{T-1} \boldsymbol{A}_{T-t} \end{bmatrix}}_{\boldsymbol{S}_{\boldsymbol{x}}} \boldsymbol{x}_1 \quad +$$

$$\underbrace{\begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \boldsymbol{B}_1 & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \boldsymbol{A}_2\boldsymbol{B}_1 & \boldsymbol{B}_2 & \cdots & \boldsymbol{0} \\ \vdots & \vdots & \ddots & \vdots \\ \left(\prod_{t=1}^{T-2} \boldsymbol{A}_{T-t}\right) \boldsymbol{B}_1 & \left(\prod_{t=1}^{T-3} \boldsymbol{A}_{T-t}\right) \boldsymbol{B}_2 & \cdots & \boldsymbol{B}_{T-1} \end{bmatrix}}_{\boldsymbol{S}_{\boldsymbol{u}}} \underbrace{\begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \vdots \\ \boldsymbol{u}_{T-1} \end{bmatrix}}_{\boldsymbol{u}},$$

where $\boldsymbol{S}_{\boldsymbol{x}} \in \mathbb{R}^{dT \times d}$, $\boldsymbol{x}_1 \in \mathbb{R}^d$, $\boldsymbol{S}_{\boldsymbol{u}} \in \mathbb{R}^{dT \times d(T-1)}$ and $\boldsymbol{u} \in \mathbb{R}^{d(T-1)}$.

Transfer matrices for single integrator

A single integrator is simply defined as $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \boldsymbol{u}_t\Delta t$, corresponding to $\boldsymbol{A}_t = \boldsymbol{I}$ and $\boldsymbol{B}_t = \boldsymbol{I}\Delta t$, $\forall t \in \{1, \dots, T\}$, and transfer matrices $\boldsymbol{S}_{\boldsymbol{x}} = \boldsymbol{1}_T \otimes \boldsymbol{I}_D$, and $\boldsymbol{S}_{\boldsymbol{u}} = \begin{bmatrix} \boldsymbol{0}_{D,D(T-1)} \\ \boldsymbol{L}_{T-1,T-1} \otimes \boldsymbol{I}_D\Delta t \end{bmatrix}$, where $\boldsymbol{L}$ is a lower triangular matrix and $\otimes$ is the Kronecker product operator.