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)
```

3 Linear quadratic tracking (LQT)

The LQT problem is formulated as the minimization of the cost

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

with $\boldsymbol{x} = \big[\boldsymbol{x}_1^\top, \boldsymbol{x}_2^\top, \dots, \boldsymbol{x}_T^\top\big]^\top$ the evolution of the state variable and $\boldsymbol{u} = \big[\boldsymbol{u}_1^\top, \boldsymbol{u}_2^\top, \dots, \boldsymbol{u}_{T-1}^\top\big]^\top$ the evolution of the control variable. $\boldsymbol{\mu} = \big[\boldsymbol{\mu}_1^\top, \boldsymbol{\mu}_2^\top, \dots, \boldsymbol{\mu}_T^\top\big]^\top$ represents the evolution of the tracking target.

$\boldsymbol{Q} = \text{blockdiag}(\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots, \boldsymbol{Q}_T)$ represents the evolution of the required tracking precision, and $\boldsymbol{R} = \text{blockdiag}(\boldsymbol{R}_1, \boldsymbol{R}_2, \ldots, \boldsymbol{R}_{T-1})$ 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

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

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)

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

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. 1. The second-order Taylor expansion of f around x_k is

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

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

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(f(x_k) + f'(x_k)\, t + \frac{1}{2} f''(x_k)\, t^2\right) = f'(x_k) + f''(x_k)\, t, \tag{3}$$

the minimum is achieved for

$$t = -\frac{f'(x_k)}{f''(x_k)}. \tag{4}$$

Newton's method thus performs the iteration

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}. \tag{5}$$

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

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{H}(\boldsymbol{x}_k)^{-1}\, \boldsymbol{g}(\boldsymbol{x}_k), \tag{6}$$

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

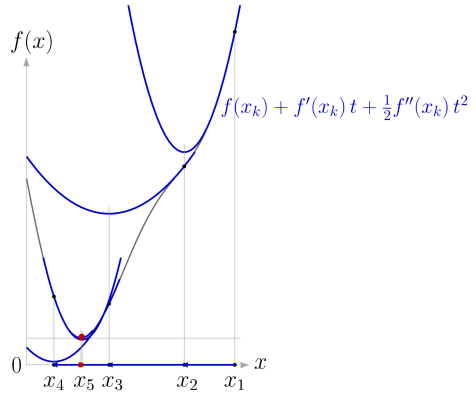

Figure 1: Newton's method for minimization, starting from an initial estimate x_1 and converging to a local minimum (red point) after 5 iterations.

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

$$\boldsymbol{g} = 2\boldsymbol{J}_{\boldsymbol{r}}^\top \boldsymbol{r}, \quad \boldsymbol{H} \approx 2\boldsymbol{J}_{\boldsymbol{r}}^\top \boldsymbol{J}_{\boldsymbol{r}}, \tag{7}$$

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

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \left(\boldsymbol{J}_{\boldsymbol{r}}^\top(\boldsymbol{x}_k) \boldsymbol{J}_{\boldsymbol{r}}(\boldsymbol{x}_k)\right)^{-1} \boldsymbol{J}_{\boldsymbol{r}}^\top(\boldsymbol{x}_k)\, \boldsymbol{r}(\boldsymbol{x}_k) \tag{8}$$

$$= \boldsymbol{x}_k - \boldsymbol{J}_{\boldsymbol{r}}^\dagger(\boldsymbol{x}_k)\, \boldsymbol{r}(\boldsymbol{x}_k). \tag{9}$$

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

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

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

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

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

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

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

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 $\boldsymbol{B}_t = \boldsymbol{I}\Delta t$, $\forall t \in \{1, \ldots, T\}$, and transfer matrices $\boldsymbol{S}_{\boldsymbol{x}} = \mathbf{1}_T \otimes \boldsymbol{I}_D$, and $\boldsymbol{S}_{\boldsymbol{u}} = \begin{bmatrix} \mathbf{0}_{D,D(T-1)} \\ \boldsymbol{L}_{T-1,T-1} \otimes \boldsymbol{I}_D \Delta t \end{bmatrix}$, where $\boldsymbol{L}$ is a lower triangular matrix and $\otimes$ is the Kronecker product operator.