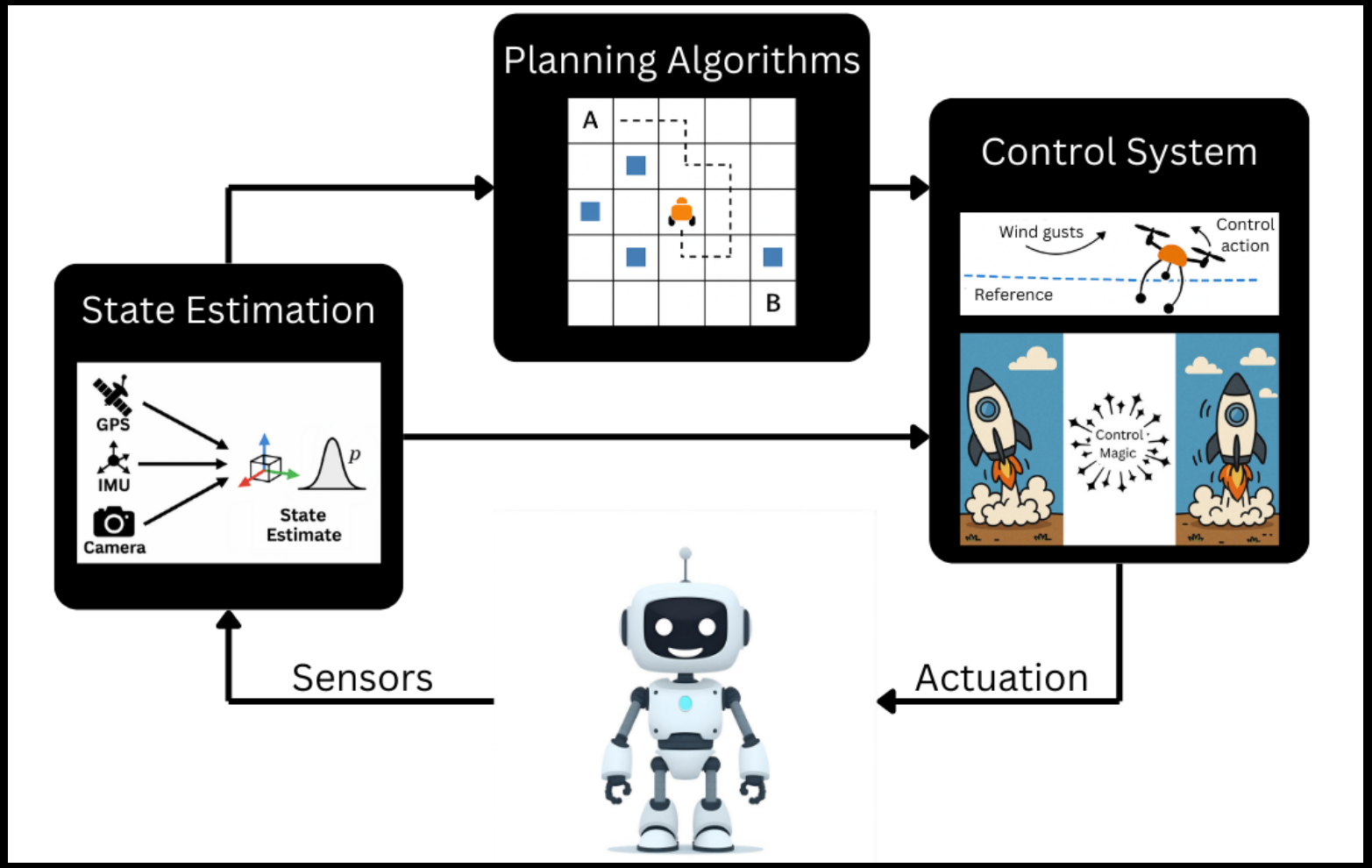


Control Stack Workshop 2

15. November 2025

Joschua Wüthrich

About Me

- PhD Student in Learning-enabled Control at the Intelligent Control Systems Group
- BSc and MSc in Mechanical Engineering
- Project experiences: SpaceHopper, Digital Twins of Surgical Procedures, Formula 1 Powertrain
- Why did I specialize in controls? It‘s a combination of applied math and engineering where abstract math concepts become tangible

Contents

- What is state estimation?
  - Luenberger Observer
  - Kalman Filter
  - Extended Kalman Filter
  - Particle Filter
- What is planning?
  - Dynamic Programming
  - RL

→ Last week assumed perfect states and given reference trajectories

What is State Estimation?

- In a real system you have many different sensors that describe some quantity of your system.

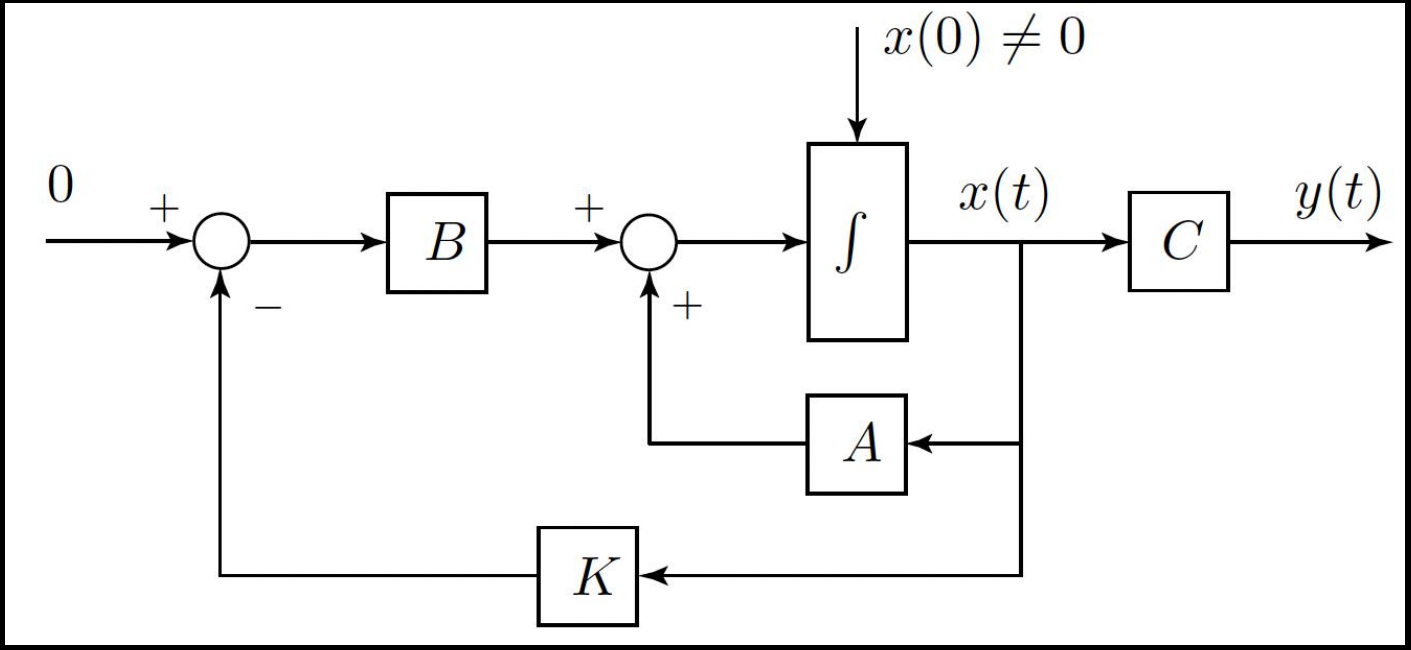
- It either directly captures a specific state or the measurement is due to a combination of different states interacting with each other

→ Temperature sensor vs. dynamic pressure sensor $q = \frac{1}{2}\rho\left(v_x^2 + v_y^2 + v_z^2\right) \rightarrow$ nonlinear combination of velocity components

- Fuse information from different sensors to get better state estimates and estimates states for which you don‘t have a sensor (otherwise could get expensive)
- Via state estimation it‘s also possible to filter noisy measurements

What is State Estimation?

Reminder: a model-based controller needs a model to calculate the actuation signal and for the model we need to know the states



The LQR controller has a state feedback law.

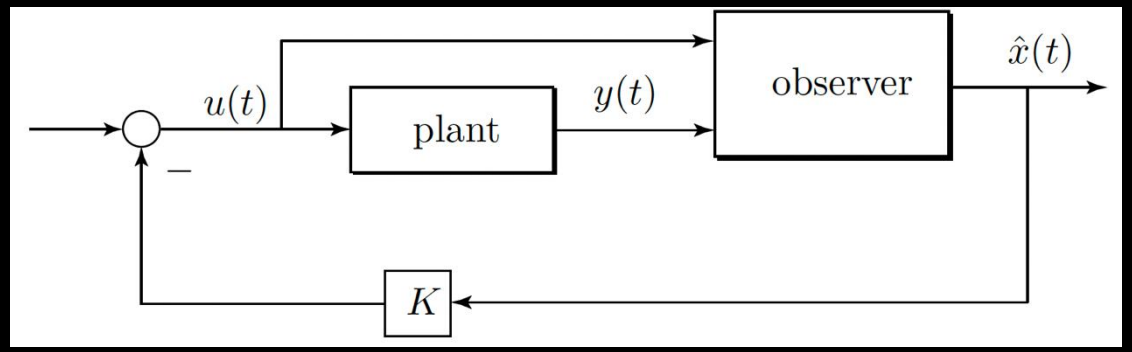
In reality, cannot access these states from the real system.

Estimate them!

State Estimation Methods and When to Use

	Deterministic Model	**Gaussian Stochastic Model**	**General Stochastic Model**
Linear System	Luenberger Observer	Kalman Filter	-
Nonlinear System	-	Extended Kalman Filter	Particle filter

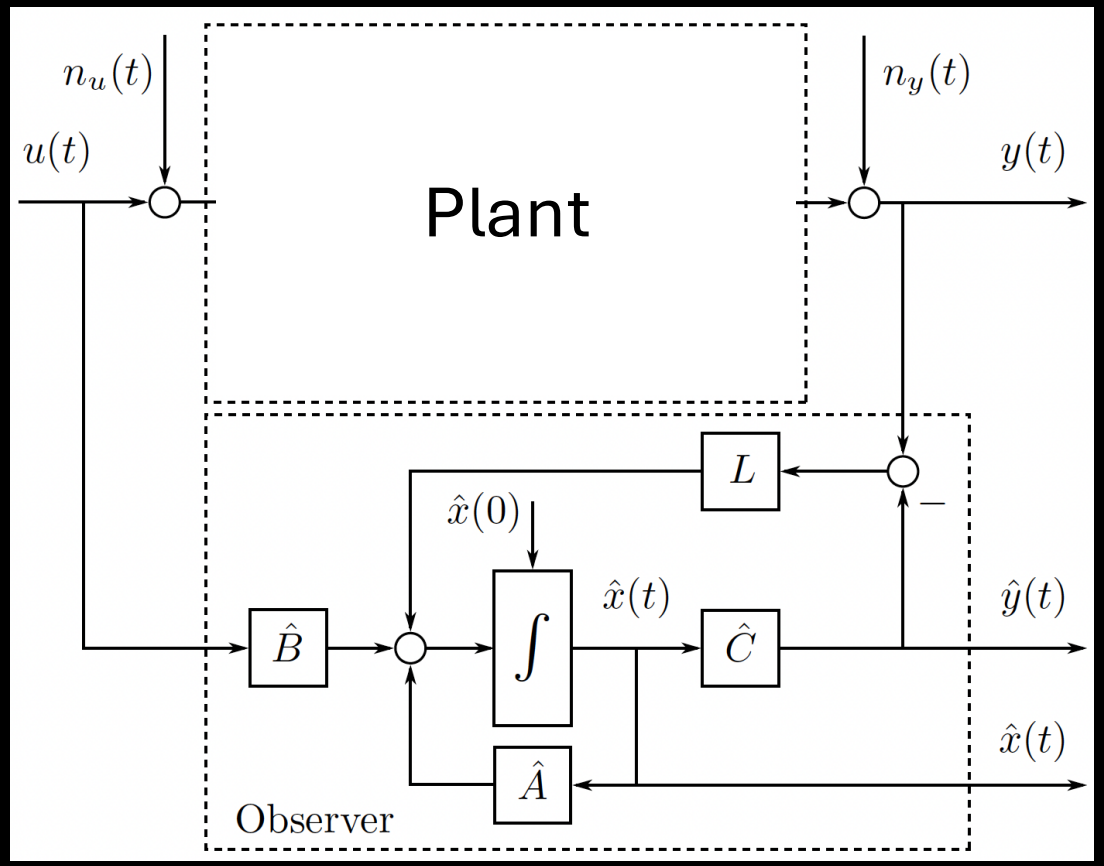
Luenberger Observer



Key idea:

Estimate the system‘s state by comparing the measurement with the output of your model and if there‘s a difference reduce the error by adjusting your state estimate

Luenberger Observer



The observer gain L is obtained as the solution of a state-feedback problem (we now how to solve that from LQR)

Look at error dynamics:

$$e(t) = x(t) - \hat{x}(t) \in \mathbb{R}^n$$

$$\begin{aligned}
\frac{d}{dt}e(t) &= \frac{d}{dt}x(t) - \frac{d}{dt}\hat{x}(t) \\
&= A \cdot x(t) + B \cdot u(t) - [A \cdot \hat{x}(t) + B \cdot u(t) + L \cdot (y(t) - \hat{y}(t))] \\
&= A \cdot (x(t) - \hat{x}(t)) - L \cdot C \cdot (x(t) - \hat{x}(t)) \\
&= [A - L \cdot C] \cdot e(t), \qquad e(0) = x(0) - \hat{x}(0) \neq 0
\end{aligned}$$

Luenberger Observer

- In the LQR setting we had A-BK, here we have now A-LC. This can be transformed to the LQR setting by $\mathrm{A}^T - \mathrm{C}^T \mathrm{L}^T$

- Solve the discrete algebraic Riccati equation but with the following changes $A \to A^T, B \to C^T, Q$ and R are again tuning parameters and obtain the positive definite solution Ψ

- Then it follows that the observer gain is

$$L^T = (C\Psi C^T + R)^{-1} C\Psi A^T$$

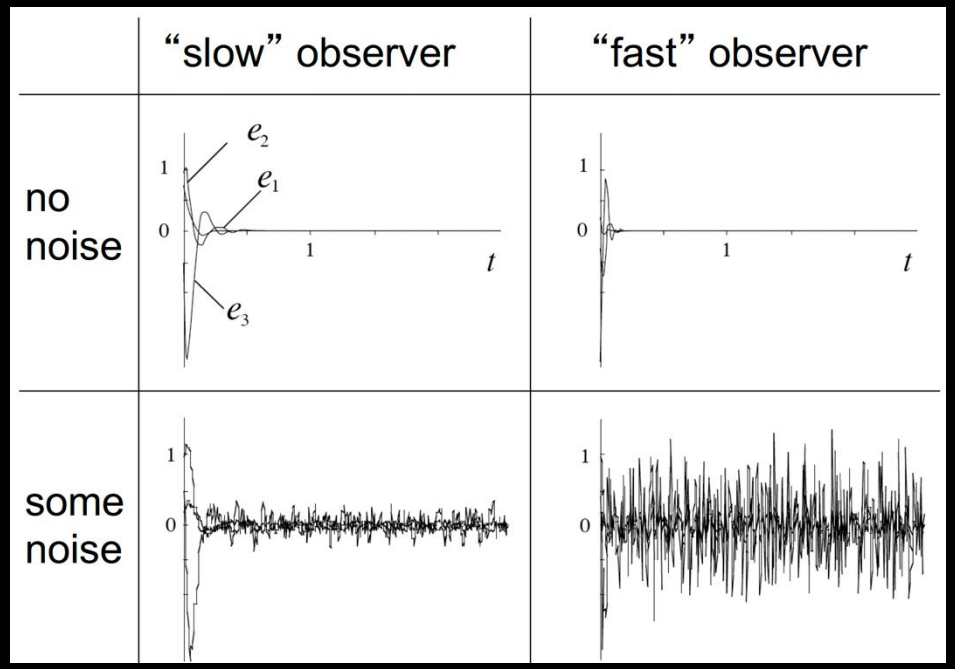
Luenberger Observer – Model

- The model that you can then actually implement in software is your observer model:

$$\frac{\mathrm{d}}{\mathrm{d}t}\hat{x}(t) = \hat{A} \cdot \hat{x}(t) + \hat{B} \cdot u(t) + L \cdot (y - \hat{y}(t))$$
$$\hat{y} = \hat{C} \cdot \hat{x}(t)$$

Luenberger Observer in Practice

- To analyze how fast the errors converge to zero, the eigenvalues of A-LC can be analyzed. Compared to the stability case, faster convergence is not necessarily desired.



As a rule of thumb, the eigenvalues of A-LC should be three times faster than those of A-BK

Either find via tuning of Q and R or do manually pole placement

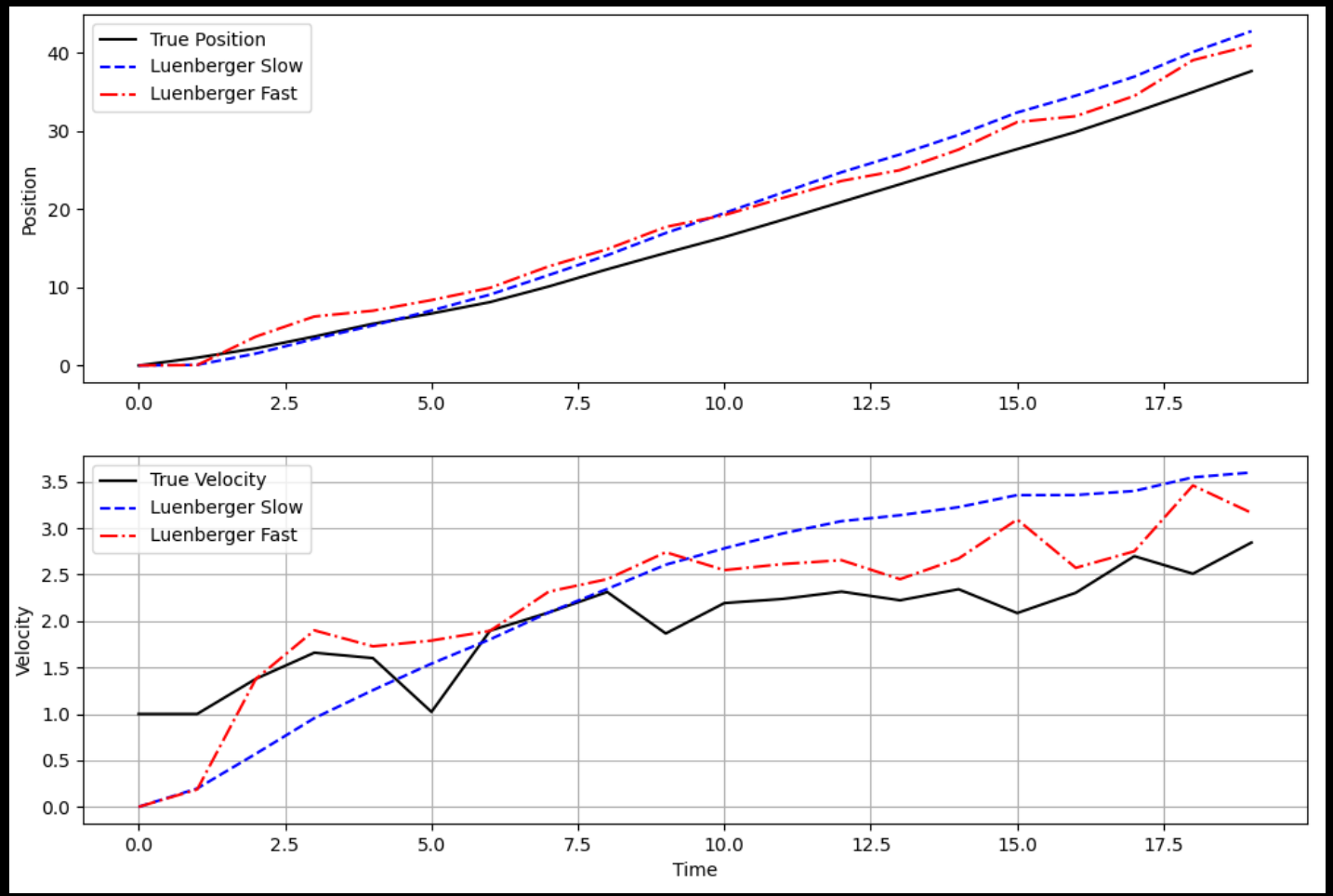
Luenberger Observer – Task 1

- state_estimation_linear_model.py
- Implement a Luenberger observer for the following system (you can assume that A, B, and C are already the discretized matrices):

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

Use: solve_discrete_are, np.linalg.inv

Luenberger Observer – Task 1



Eigenvalues:

„Slow“ $|\lambda| = 0.82279227$

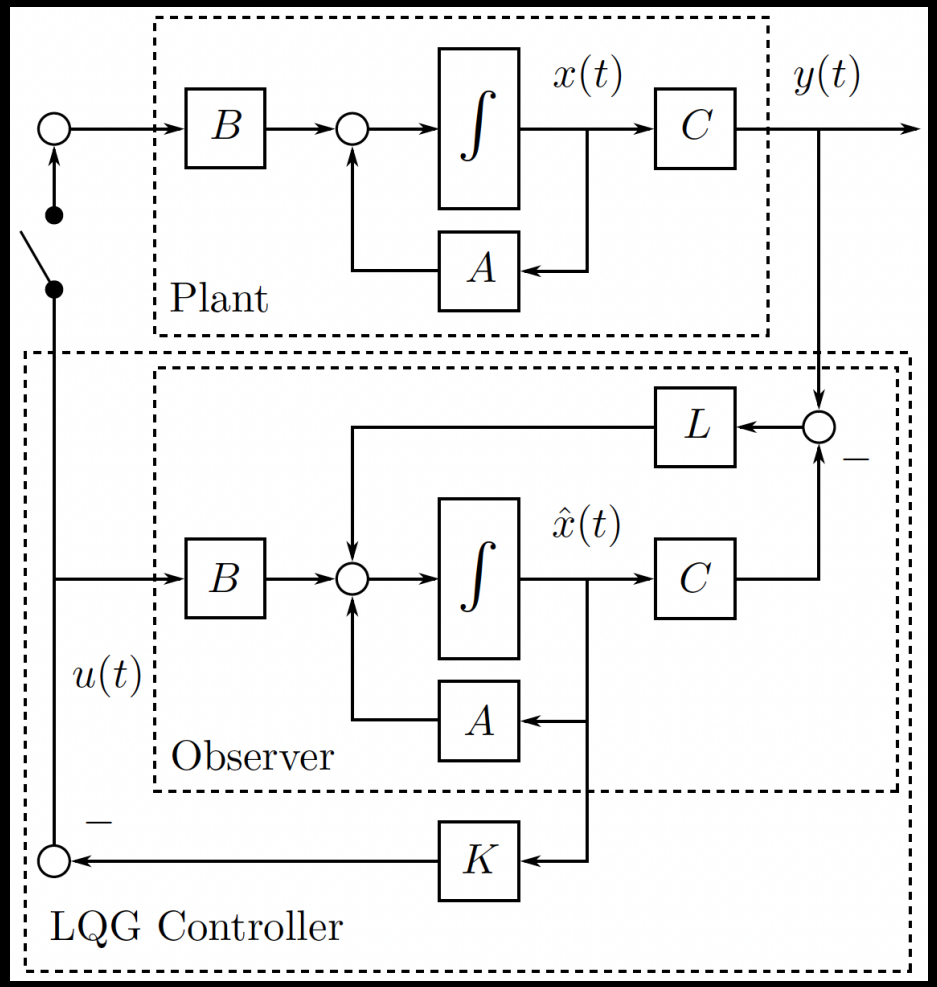
„Fast“ $|\lambda| = 0.46448703$

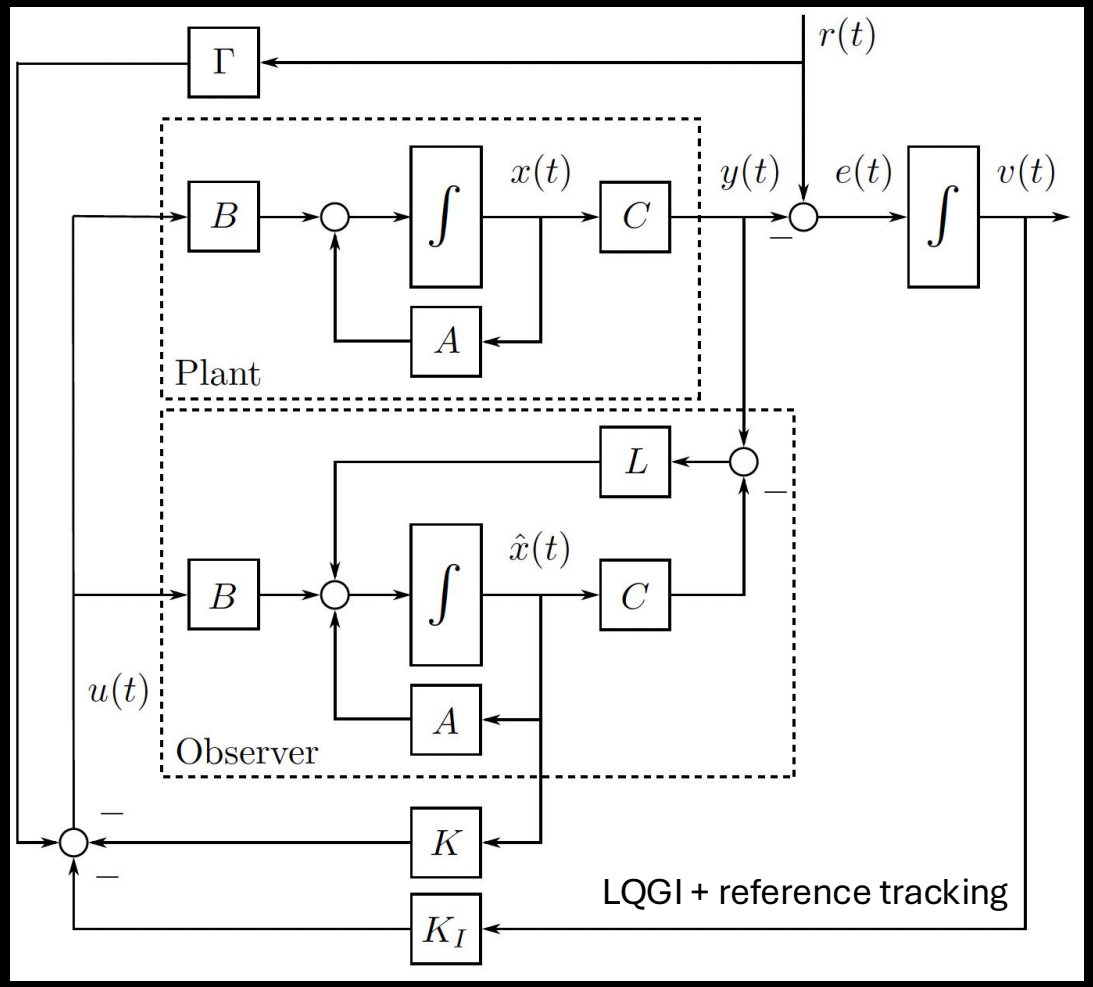
The fast observer is noisier but reacts quicker to changes in the real signal

→ Tradeoff

In this solution, the Q and R gains are still not optimal as we have an offset to the real state + we have stochastic noise

Luenberger Observer – What now?





Luenberger Observer – What now?

- For now still need to assume that the real system behaves like a linear system (assumption is justified if we are close to the linearization point)

- If the noise n_y and n_u are Gaussian white noise signals and if they‘re statistical properties are known, an optimal design of the observer is possible!

→ Kalman Filter

Quick Facts about Kalman

- Rudolf Kalman – US-Hungarian engineer and mathematician
- Was from 1971 to 1997 at ETH
- The famous Kalman Filter was critized heavily by mathematicians, which is why he published it in a mechanical engineering journal
- Ultimately, the filter was implemented in the Apollo program and helped in the first-moon landing
- Nowadays, it‘s used everywhere from signal processing, control systems, to GNC.
- For this work, Obama awarded him with the National Medal of Science in 2008

Kalman Filter

- System structure

$$x(k) = A(k-1)x(k-1) + u(k-1) + v(k-1) \quad (6.1)$$

$$z(k) = H(k)x(k) + w(k) \quad (6.2)$$

$x(k)$: state
$u(k)$: known control input
$v(k)$: process noise
$z(k)$: measurement
$w(k)$: sensor noise

→ In the general, setting the dynamics can be time-varying (A and H) but in our case, usually constant

$x(0) \sim \mathcal{N}(x_0, P_0)$, i.e. $x(0)$ has a Gaussian distribution with mean x_0 and variance P_0,
$v(k) \sim \mathcal{N}(0, Q(k))$, $w(k) \sim \mathcal{N}(0, R(k))$.

→ $x(0), \{v(.)\}, \{w(.)\}$ are mutually independent

Kalman Filter

- I won‘t show the derivation of the KF equations (attend Raff‘s Recursive Estimation course)

- Key concepts: an affine transformation of a gaussian random variable (GRV) is a GRV and a linear combination of two jointly GRVs is a GRV

- Estimation is done in two steps: Bayesian state estimator (priors and a posteriori)
    1. Prior Update: update the prior at k based on all measurements up to k-1
    2. Measurement Update: get a new measurement at k and use this to update your estimate a posteriori at k

Kalman Filter

Initialization: $\hat{x}_{\mathrm{m}}(0) = x_0,\ P_{\mathrm{m}}(0) = P_0$

Step 1 (S1): Prior update/Prediction step

$$\hat{x}_{\mathrm{p}}(k) = A(k-1)\hat{x}_{\mathrm{m}}(k-1) + u(k-1)$$

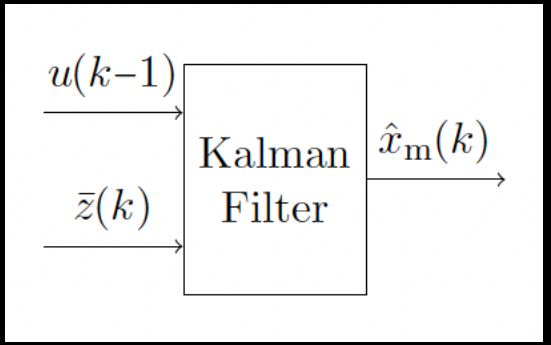
$$P_{\mathrm{p}}(k) = A(k-1)P_{\mathrm{m}}(k-1)A^T(k-1) + Q(k-1)$$

Step 2 (S2): A posteriori update/Measurement update step

Results from above, re-introducing time index k:

$$P_{\mathrm{m}}(k) = \left(P_{\mathrm{p}}^{-1}(k) + H^T(k)R^{-1}(k)H(k)\right)^{-1}$$

$$\hat{x}_{\mathrm{m}}(k) = \hat{x}_{\mathrm{p}}(k) + P_{\mathrm{m}}(k)H^T(k)R^{-1}(k)\left(\bar{z}(k) - H(k)\hat{x}_{\mathrm{p}}(k)\right)$$



We have seen that for the Luenberg Observer!

The PDF of the state x(k) is fully characterized by the mean $\hat{x}_m(k)$ and the variance $P_m(k)$

In the case, where A, H, Q, and R are constant, it can be shown that if P_p converges for $k \to \infty$, it is the solution of the discrete algebraic Riccati equation

Kalman Filter

What if the noise is not zero-mean?

- Variance updates stay the same

- Mean updates change to

$$\hat{x}_p(k) = A(k-1)\hat{x}_m(k-1) + u(k-1) + E[v(k-1)]$$
$$\hat{x}_m(k) = \hat{x}_p(k) + P_m(k)H^T(k)R^{-1}(k)\left(\bar{z}(k) - H(k)\hat{x}_p(k) - E[w(k)]\right)$$

Kalman Filter – Alternative Equations

It’s possible to bring the previous seen measurement update equations into an alternative form.

Depending on your preference you can choose either one.

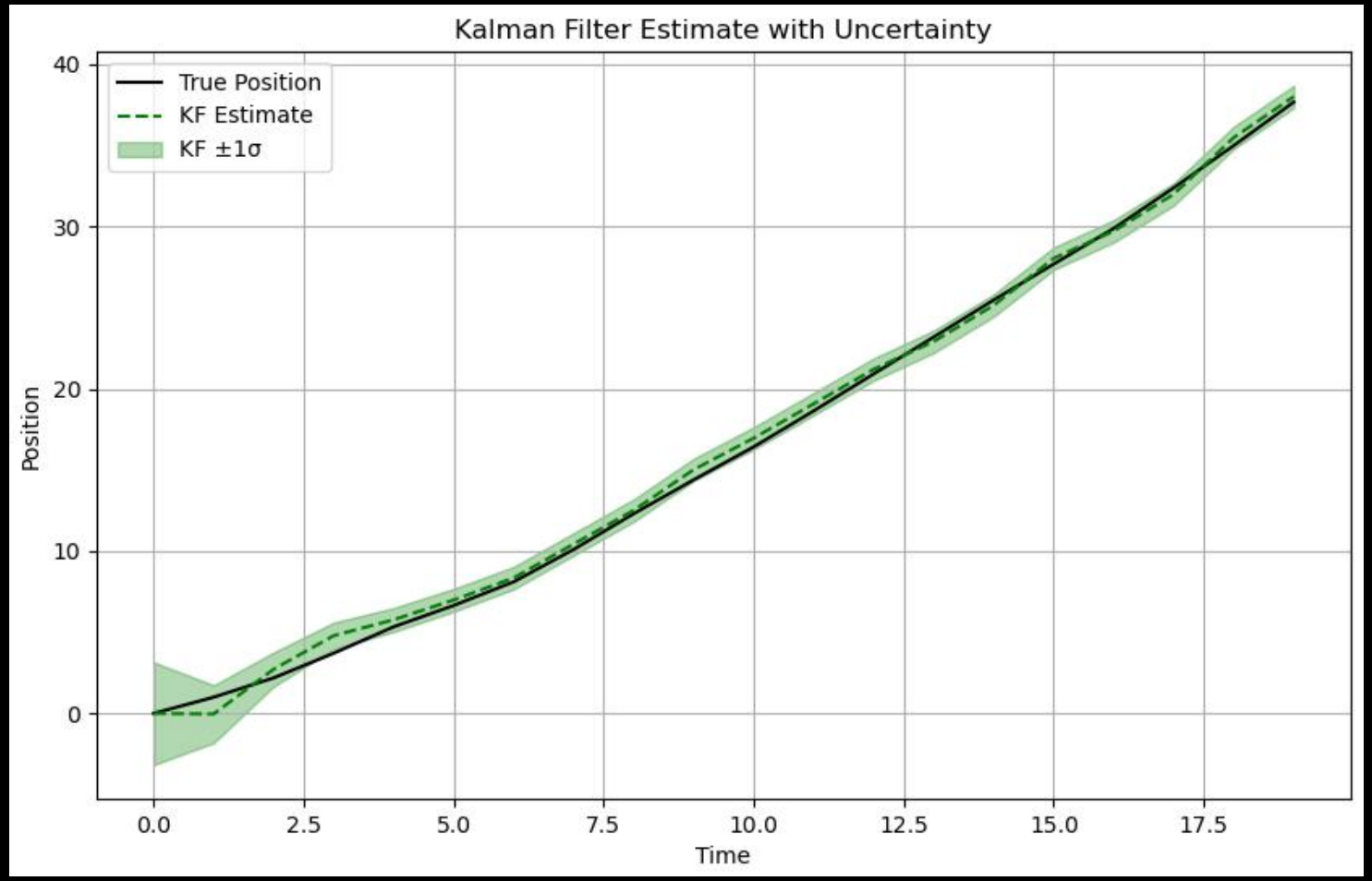
$$
\begin{aligned}
K(k) &= P_{\mathrm{p}}(k) H^T(k) \left(H(k) P_{\mathrm{p}}(k) H^T(k) + R(k) \right)^{-1} \\
\hat{x}_{\mathrm{m}}(k) &= \hat{x}_{\mathrm{p}}(k) + K(k) \left(\bar{z}(k) - H(k) \hat{x}_{\mathrm{p}}(k) \right) \\
P_{\mathrm{m}}(k) &= \left(I - K(k) H(k) \right) P_{\mathrm{p}}(k) \\
&= \left(I - K(k) H(k) \right) P_{\mathrm{p}}(k) \left(I - K(k) H(k) \right)^T + K(k) R(k) K^T(k)
\end{aligned}
$$

Kalman Filter – Task 2

- state_estimation_linear_model.py
- Implement a Kalman Filter for the same system as used for the Luenberg Observer and compare the two observers

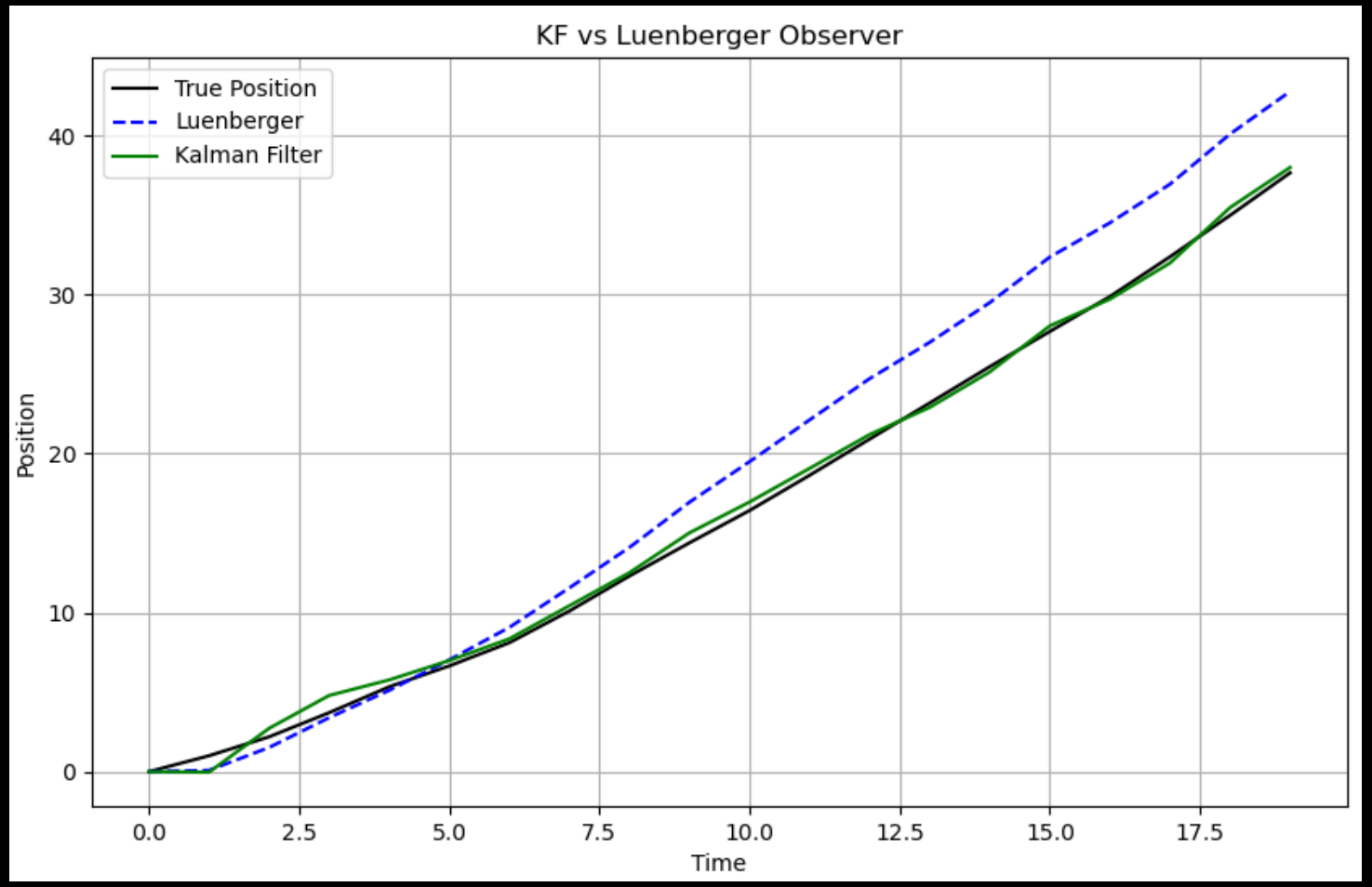
- In my implementation x_kf = x_m, P_kf = P_m, x_pred = x_p

Kalman Filter – Task 2



Since the Kalman Filter is the optimal solution to the Bayesian state estimator problem for linear systems with Gaussian noise, the initial uncertainty is steadily reduced

Kalman Filter – Task 2



Key takeaway: Implementation effort is comparable between the two methods.

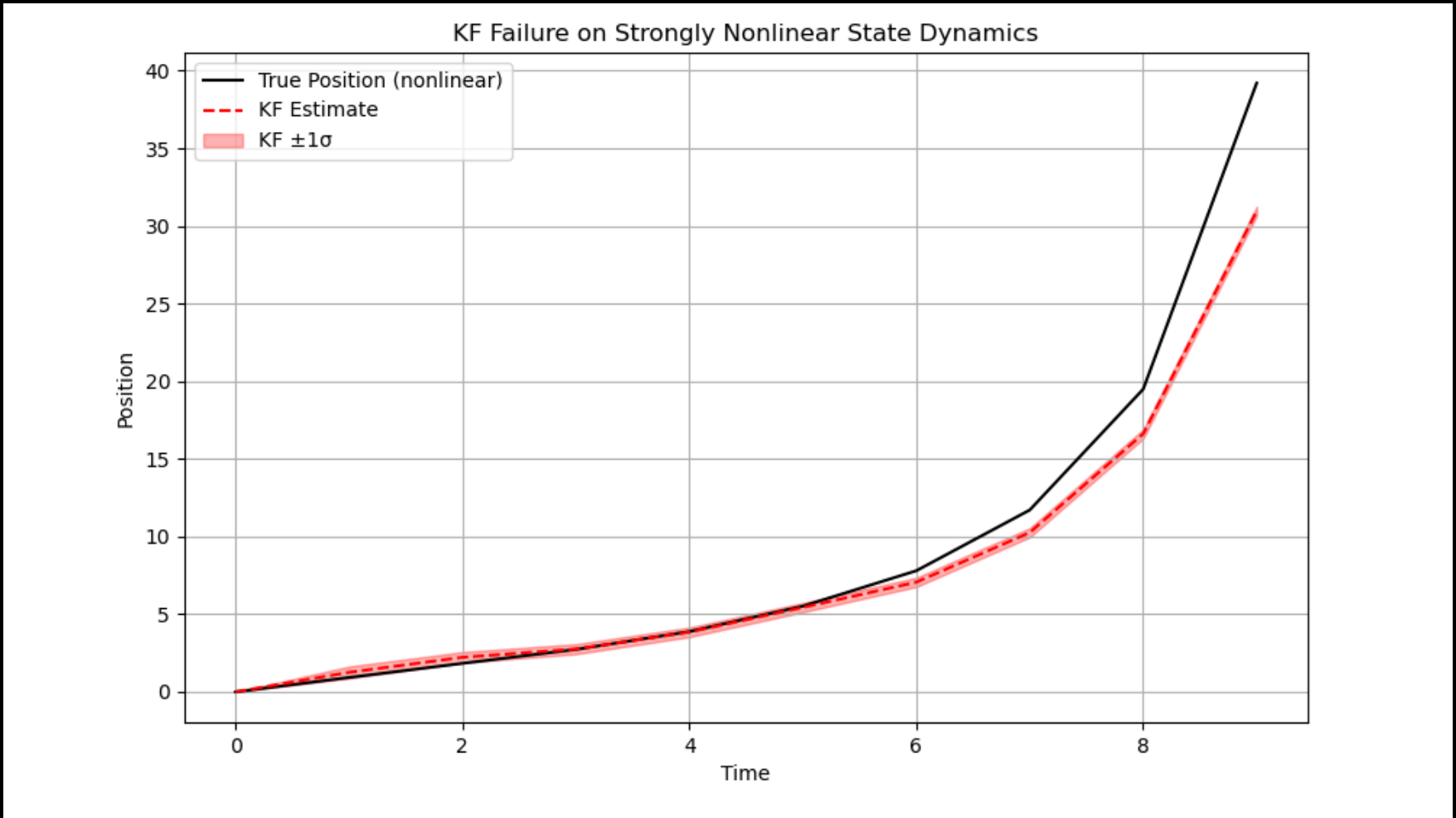
If you take the time and analyze your measurement data (approximate mean and variance) you can get without much effort a better solution and you don‘t need to tune anything!

Kalman Filter – Limitations

- Once we are dealing with a nonlinear system, the key assumption behind the KF, i.e., that our variables always remain GRVs, is not valid anymore.

- Moreover, if the noise is not Gaussian, the KF is not the optimal solution anymore to the estimation problem

Kalman Filter – Limitations (kf_failure_strong_nonlinear_dynamics.py)



Extended Kalman Filter

- The EKF is the extension of the KF to nonlinear systems

- We consider the following nonlinear discrete-time system

$$
\begin{aligned}
x(k) &= q_{k-1}\big(x(k-1), u(k-1), v(k-1)\big) & \mathrm{E}\left[x(0)\right] &= x_0, \mathrm{Var}\left[x(0)\right] = P_0 \\
& & \mathrm{E}\left[v(k-1)\right] &= 0, \mathrm{Var}\left[v(k-1)\right] = Q(k-1) \\
z(k) &= h_k\big(x(k), w(k)\big) & \mathrm{E}\left[w(k)\right] &= 0, \mathrm{Var}\left[w(k)\right] = R(k)
\end{aligned}
$$

where, $x(0), \{v(.)\}, \{w(.)\}$ are mutually independent

Extended Kalman Filter

Key concepts to derive the EKF equations:

- Follow the KF derivation with the process and measurement update notion
- Linearize the state update around your current state estimate $\hat{x}_m$ and $E[v(k-1)] = 0$
- Linearize the measurement equation around prior state estimate $\hat{x}_p$ and $E[w(k)] = 0$

Extended Kalman Filter – Implementation

Initialization: $\hat{x}_{\mathrm{m}}(0) = x_0,\ P_{\mathrm{m}}(0) = P_0.$

Step 1 (S1): Prior update/Prediction step

$$\hat{x}_{\mathrm{p}}(k) = q_{k-1}(\hat{x}_{\mathrm{m}}(k-1), 0)$$
$$P_{\mathrm{p}}(k) = A(k-1)P_{\mathrm{m}}(k-1)A^T(k-1) + L(k-1)Q(k-1)L^T(k-1)$$

where

$$A(k-1) := \frac{\partial q_{k-1}(\hat{x}_{\mathrm{m}}(k-1), 0)}{\partial x} \quad \text{and} \quad L(k-1) := \frac{\partial q_{k-1}(\hat{x}_{\mathrm{m}}(k-1), 0)}{\partial v}.$$

Step 2 (S2): A posteriori update/Measurement update step

$$K(k) = P_{\mathrm{p}}(k)H^T(k)\left(H(k)P_{\mathrm{p}}(k)H^T(k) + M(k)R(k)M^T(k)\right)^{-1}$$
$$\hat{x}_{\mathrm{m}}(k) = \hat{x}_{\mathrm{p}}(k) + K(k)\left(\bar{z}(k) - h_k(\hat{x}_{\mathrm{p}}(k), 0)\right)$$
$$P_{\mathrm{m}}(k) = \left(I - K(k)H(k)\right)P_{\mathrm{p}}(k)$$

where

$$H(k) := \frac{\partial h_k(\hat{x}_{\mathrm{p}}(k), 0)}{\partial x} \quad \text{and} \quad M(k) := \frac{\partial h_k(\hat{x}_{\mathrm{p}}(k), 0)}{\partial w}.$$

Noise variance is scaled by L because if you do the linearization it will be of the form:

$$x \approx f(\hat{x}_m) + A(x - \hat{x}_m) + Bu + Lv$$

For the case, where the noise is not zero-mean the equations need to be adjusted the same way as for the KF.

Extended Kalman Filter – Practical Remarks

- If you want to use the KF as a state observer, we have seen that all the KF matrices are constant (can be computed offline)

- Here, these matrices are linearizations around the current estimates, which is why they are time-varying and therefore cannot be precomputed offline

→Can slightly increase computational time to obtain state estimates.

- If the actual values are close to the ones we linearize about, the linearization is a good approximation of the actual nonlinear dynamics. This assumption might be bad, as for Gaussian PDFs, the noise is unbounded.

Extended Kalman Filter – Practical Remarks

- The EKF is not an optimal state observer anymore (it‘s a possibly crude approximation of the Bayesian state estimator) since for a general nonlinear system it doesn‘t hold

$$E[q(x(k-1), u(k-1), v(k-1),] \neq q(E[x(k-1)], u(k-1), E[v(k-1)])$$

- But in practice it has proven to work quite well → especially, for mildly nonlinear systems with unimodal distributions

- What‘s the alternative? Solving the Bayesian estimator problem but that‘s usually intractable for general nonlinear systems → EKF tradeoff between tractability and accuracy.

Extended Kalman Filter – Task 3

- ekf_kf_comparison.py
- Implement the extended Kalman Filter for the same nonlinear system, where the KF failed.

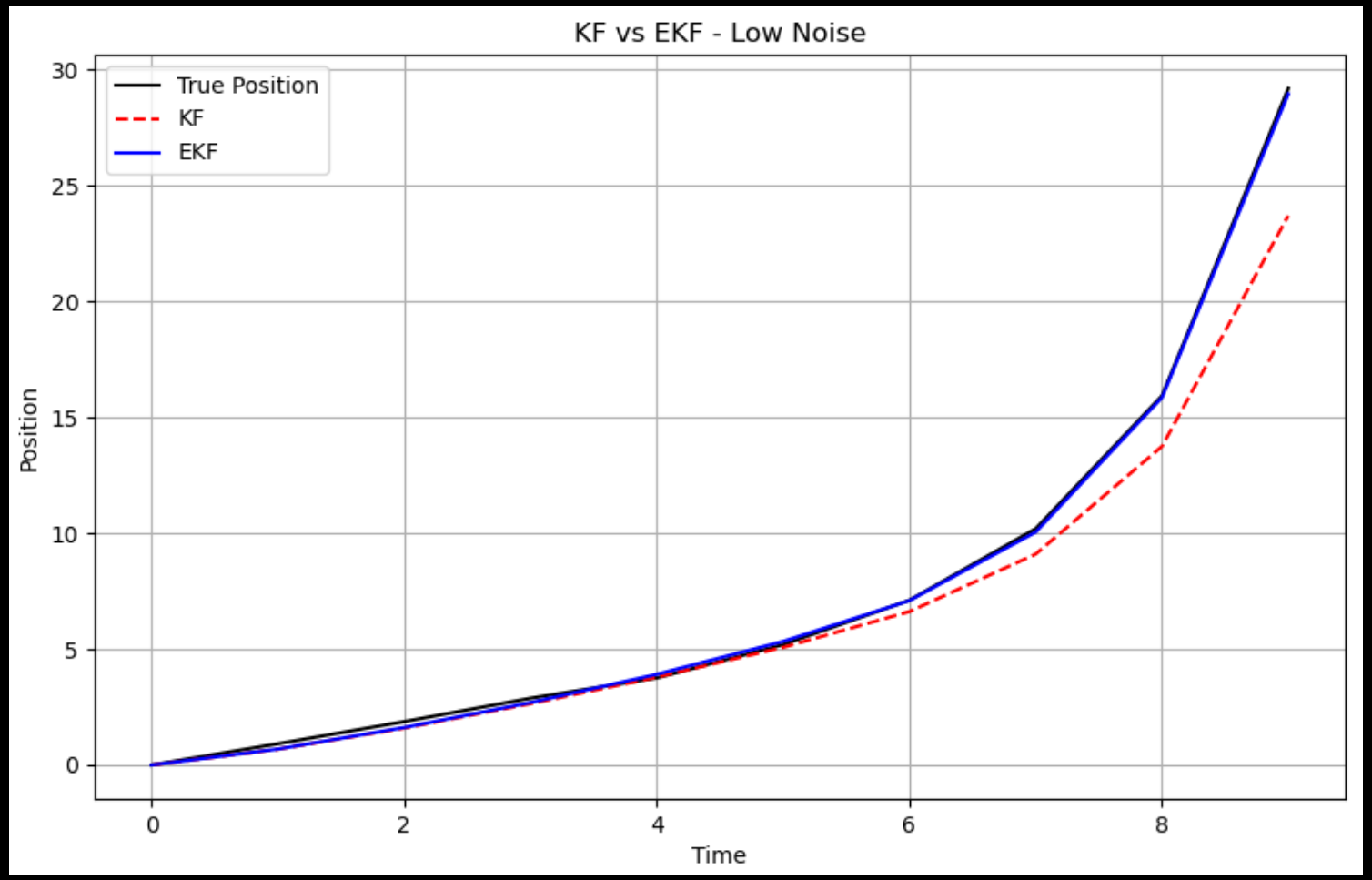
→Hint: noise not correlated among states -> L and M are identity matrices

→In my code: x_pred_ekf = x_p_ekf, x_ekf = x_m_ekf, P_ekf = P_m_ekf

- Compare the KF to the EKF

- Play around with the noise variance (larger variance can lead to larger noise realizations → estimation error can become larger)

Extended Kalman Filter – Task 3

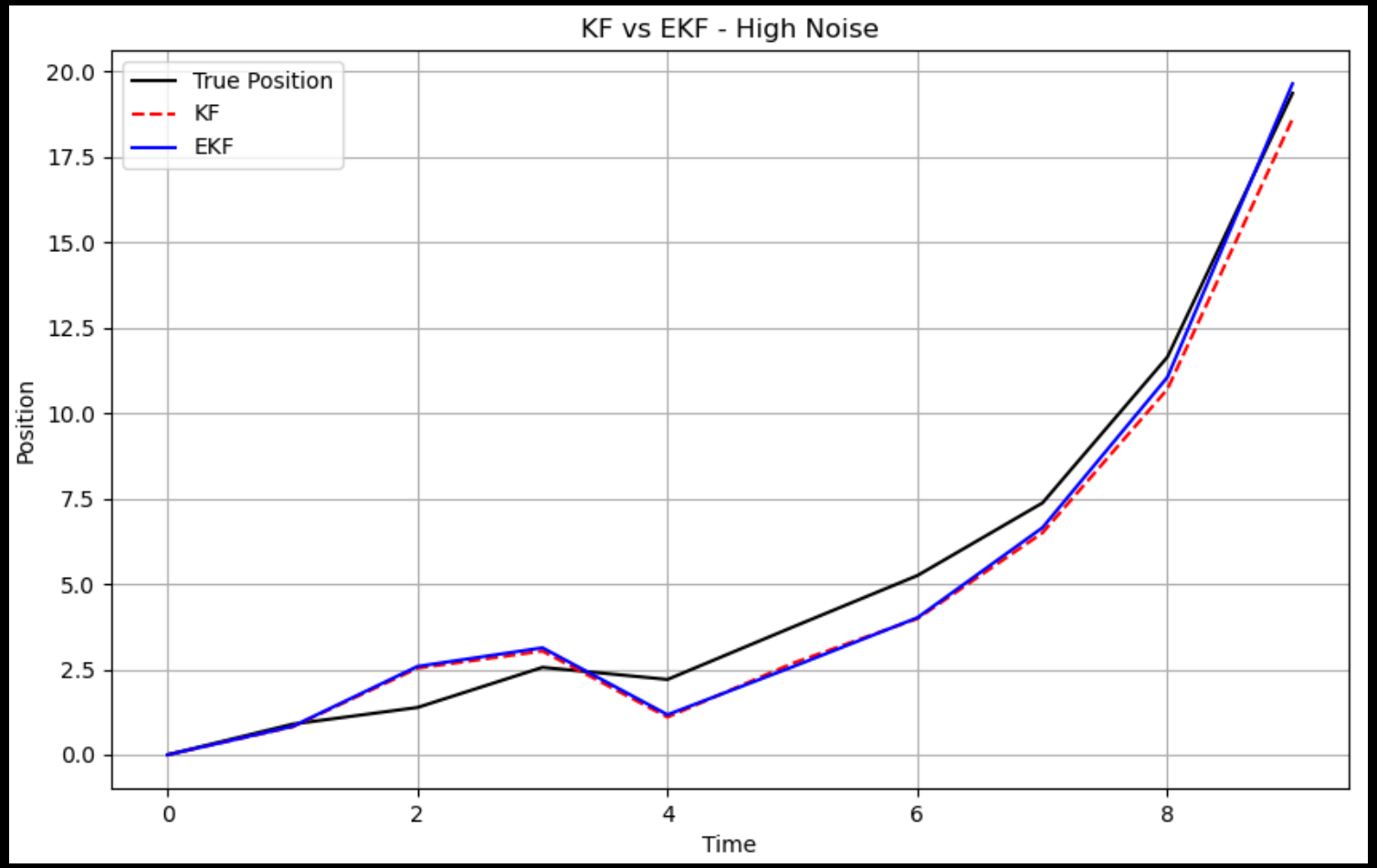


Here we clearly see that the EKF performs better for the nonlinear system compared to the KF.

Moreover, as the noise level is low, it does it's job quite well.

$$e_{avg,KF} = 1.263$$
$$e_{avg,EKF} = 0.946$$

Extended Kalman Filter – Task 3



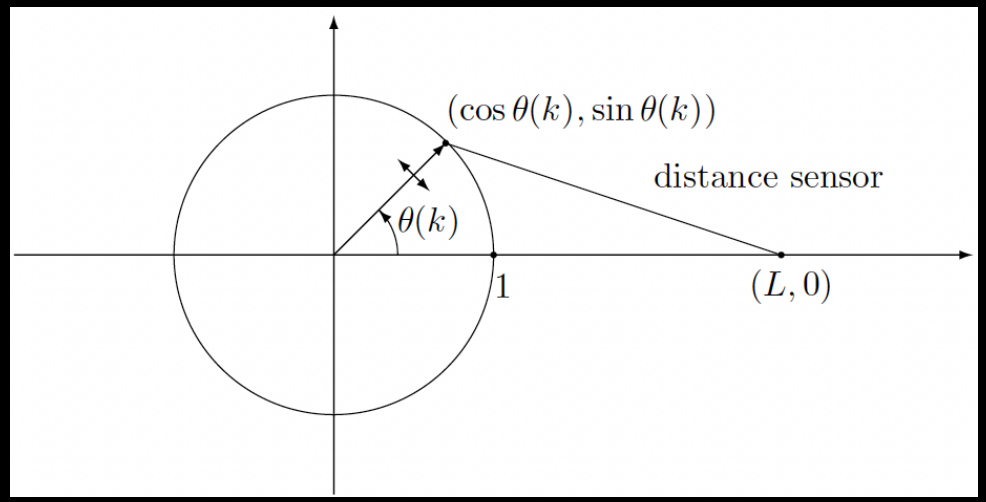
Also in the setting, where the noise level is higher, we still achieve better solutions with the EKF compared to the KF

But, the superiority of performance is decreasing

$$e_{avg,KF} = 1.421$$
$$e_{avg,EKF} = 1.218$$

→ When does the EKF completely fail?

Extended Kalman Filter – Failure (ekf_failure_unit_circle.py)



$$\theta(k) = \mathrm{mod}\left(\theta(k-1) + s(k-1),\, 2\pi\right)$$

s(k-1) ~unif$(-\bar{s} + b, \bar{s} + b)$

b ~unif$(-\bar{s}, \bar{s})$, time-independent

$$x_1(k) = x_1(k-1)$$
$$x_2(k) = \mathrm{mod}\left(x_2(k-1) + x_1(k-1) + v(k-1),\, 2\pi\right)$$

$x_1 = b, x_2 = \theta, x_1(0) \sim [-\bar{s}, \bar{s}], v(k-1) \sim [-\bar{s}, \bar{s}]$

Measurements:

- Distance sensor at (L,0)

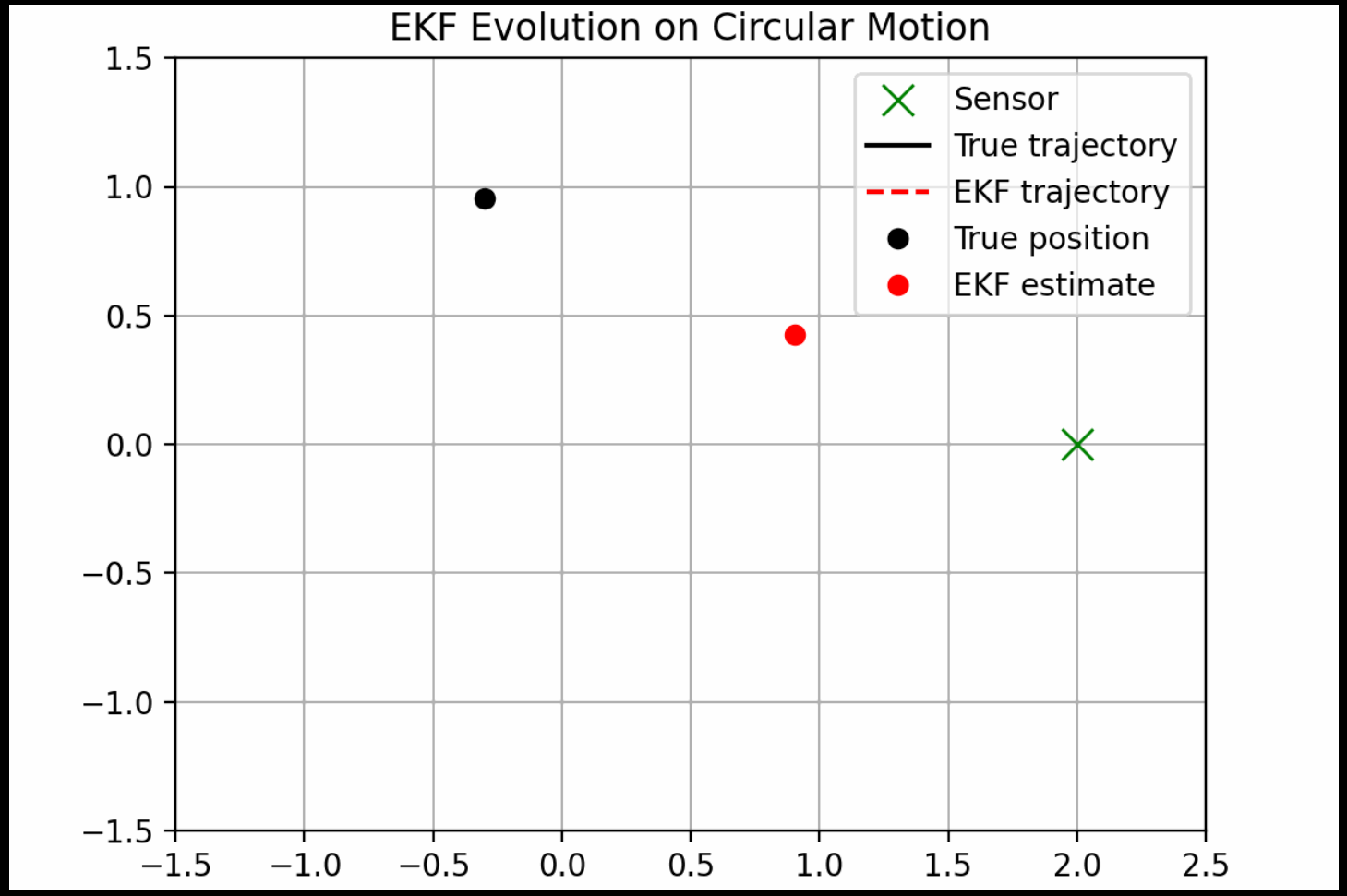
$$z_1(k) = \sqrt{(L - \cos x_2(k))^2 + (\sin x_2(k))^2} + w(k)$$

w(k) ~unif$(-e, e)$

- Half-plane measurement (occasionally)

$$z_2(k) = \begin{cases} 1 & \text{if} \quad x_2(k) \in [0, \pi) \\ -1 & \text{if} \quad x_2(k) \in [\pi, 2\pi) \end{cases}$$

Extended Kalman Filter – Failure (ekf_failure_unit_circle.py)



Since the model is highly nonlinear, especially due to the «mod» operation and since we don't have Gaussian noise anymore, the EKF struggles initially a lot to estimate the state.

In this case it manages to converge at one point, but imagine you have a controller in between that reacts to these noisy estimates

→ This can damage your system

Particle Filter

- The particle filter is an approximation of the Bayesian state estimator for general nonlinear systems with general PDFs.

$$x(k) = q_{k-1}\big(x(k-1), u(k-1), v(k-1)\big)$$

$$z(k) = h_k\big(x(k), w(k)\big)$$

where, $x(0), \{v(.)\}, \{w(.)\}$ are mutually independent and can be discrete or continuous random variables with known PDFs.

Particle Filter

- Basic idea:

  Approximate the state PDF by a large number of samples (particles). In a region, where the PDF takes large values, there is a large number of particles and vice versa.

  Similar to before, we now propagate the particles through the process model (state update), then the particles are weighted according to their measurement likelihood and with resampling a new set of particles is generated.

→ This behavior can be achieved by using Monte Carlo Sampling

Particle Filter

Initialization: Draw N samples $\{\bar{x}_{\mathrm{m}}^n(0)\}$ from $p_{x(0)}$. These are the initial particles.

Step 1 (S1): Prior update/Prediction step

To obtain the prior particles $\{\bar{x}_{\mathrm{p}}^n(k)\}$, the process equation is applied to the particles $\{\bar{x}_{\mathrm{m}}^n(k-1)\}$:

$$\bar{x}_{\mathrm{p}}^n(k) := q_{k-1}(\bar{x}_{\mathrm{m}}^n(k-1), \bar{v}^n(k-1)), \qquad \text{for } n = 1, 2, \ldots, N,$$

which requires N noise samples from $p_{v(k-1)}$.

Step 2 (S2): A posteriori update/Measurement update step

Scale each particle by the measurement likelihood:

$$\beta_n = \alpha \, p_{z(k)|x(k)}\left(\bar{z}(k)|\bar{x}_{\mathrm{p}}^n(k)\right), \qquad \text{for } n = 1, 2, \ldots, N,$$

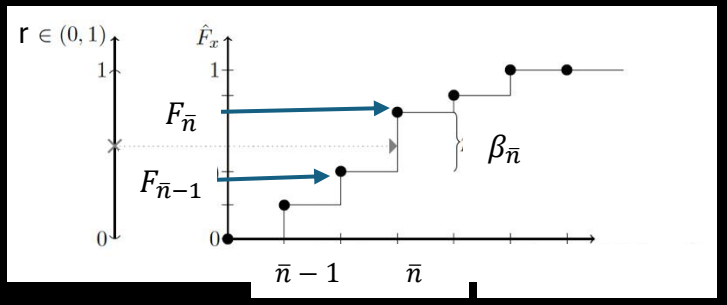
where α is the normalization constant chosen such that $\sum_{n=1}^{N} \beta_n = 1$.

Resample (using the resampling algorithm above) to get the N posterior particles $\{\bar{x}_{\mathrm{m}}^n(k)\}$, all with equal weights.

Algorithm. Repeat N times:

- Select a random number r uniformly on $(0, 1)$.
- Pick particle $\overline{n}$ such that $\sum_{n=1}^{\overline{n}-1} \beta_n < r$ and $\sum_{n=1}^{\overline{n}} \beta_n \geq r$.

$\overline{n}-1 = F_{\bar{n}-1}$ $\quad$ $\overline{n} = F_{\bar{n}}$



Particle Filter – Implementation

Initialization:

- Implement the distribution from the initial state and sample from it (many predefined pdfs available in np.random)

Prior update:

- Evaluate your dynamics for each particle and for each particle you need a new noise realization (as done above for the initial state)

Maximum Likelihood:

- $p_{z(k)|x(k)}$ is usually given by $p_{w(k)}\left(z(k) - h\left(x(k)\right)\right)$ → evaluate maximum likelihood of pdf of w(k) and when you have this formula you can plug in $z(k) - h\left(x(k)\right)$

Resample:

- np.random.choice and pass your weights as an argument

Particle Filter – Practical Remarks

A potential problem with the PF is that all particles might converge to the same one

→These particles are therefore not a good representation of the PDF anymore.

Roughening:

- Perturb the particles after resampling: $\bar{x}_{\mathrm{m}}^{n}(k) \leftarrow \bar{x}_{\mathrm{m}}^{n}(k) + \Delta x^{n}(k)$
- Δx^{n} is drawn from a zero-mean, finite-variance distribution (one way to do it):

  standard deviation: $\sigma_i = K\, E_i\, N^{-\frac{1}{d}}$

K: tuning parameter, typically $K \ll 1$

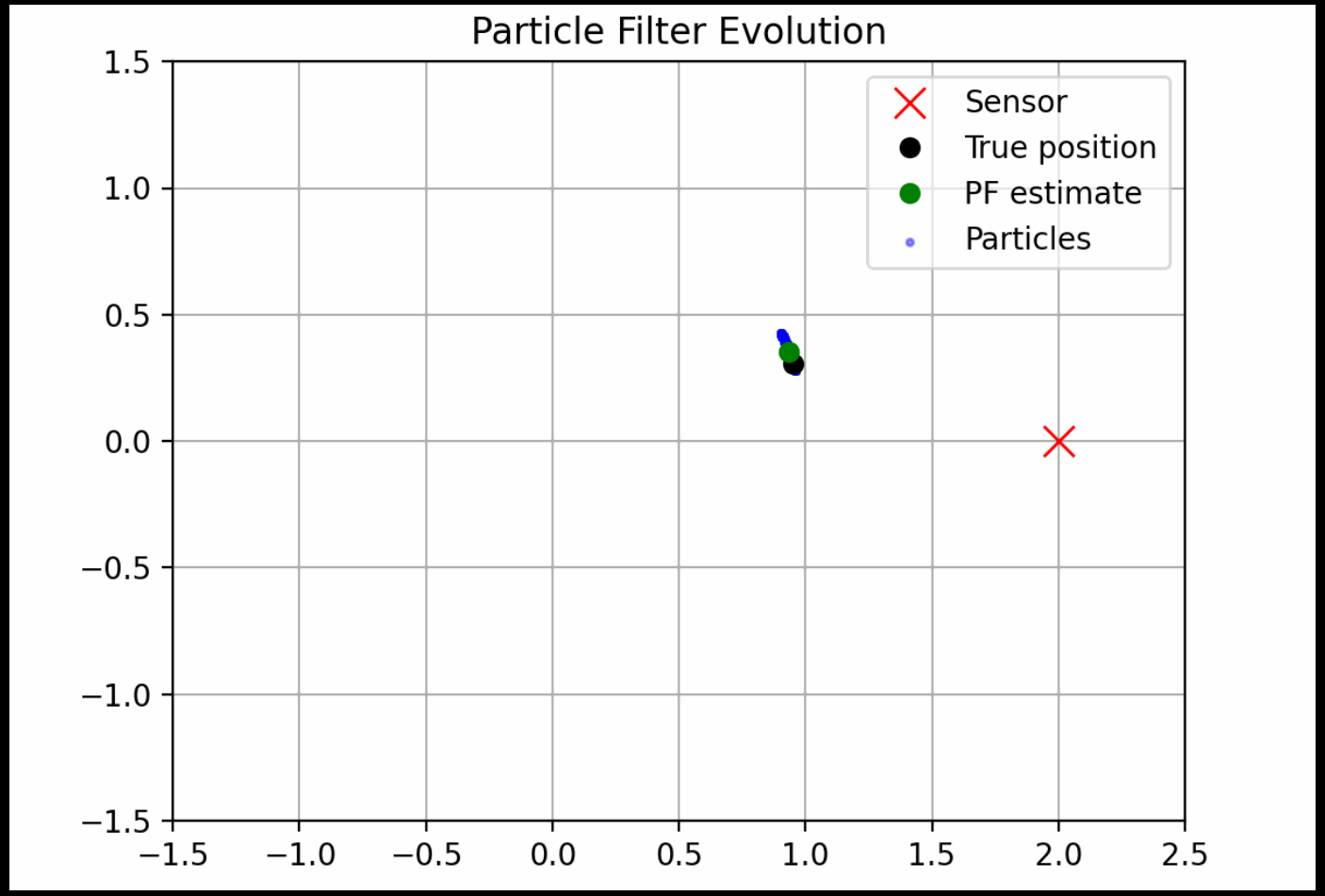
d: dimension of the state space

E_i: $\max_{n_1,n_2} |\bar{x}_{\mathrm{m},i}^{n_1}(k) - \bar{x}_{\mathrm{m},i}^{n_2}(k)|$, the maximum inter-sample variability

Particle Filter – Task 4 (pf_success_unit_circle.py)

- Implement the PF for the mass on the circle that didn‘t work for the EKF before.
- Dynamics, measurement and roughening already implemented
- Use: np.random.choice, np.random.uniform

Particle Filter – Task 4 (pf_success_unit_circle.py)



Even though the implementation is not as straightforward and also needs some engineering to work, once it is done, the performance is very strong!

The particle approach let's us use all the available sensor information and we can track this highly nonlinear system with non-gaussian noise.

Where are we now

- Now we have different methods to estimate the states of our system based on fused data from different sensor sources
- These state estimates can be used inside the control system to calculate the necessary actuation signal to get to a desired setpoint / stabilize the system
- But how do we actually know where we want to go? Either, we predefine a reference trajectory or if we want the system to act autonomously we need a high-level planner.

Planning Algorithms

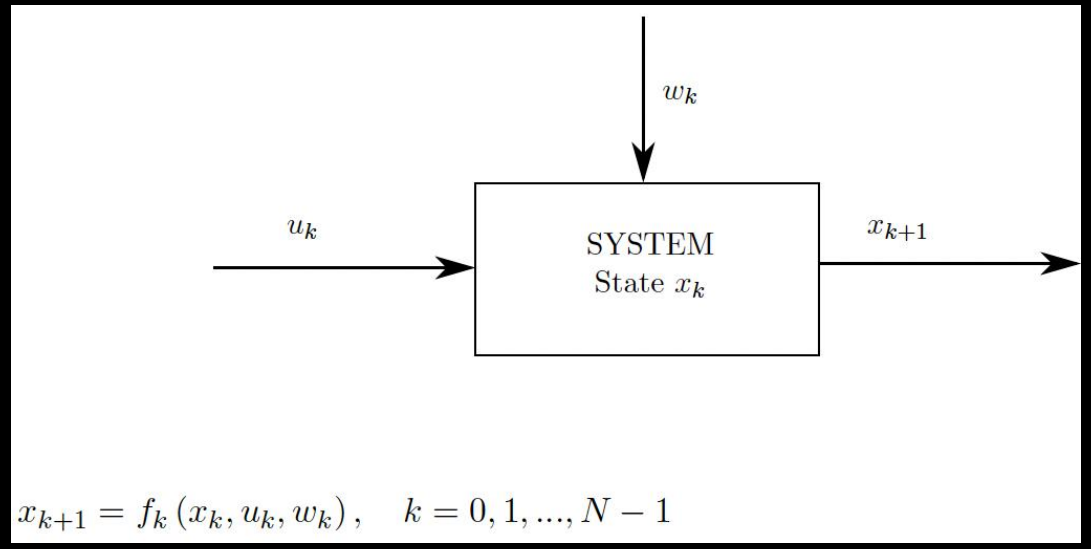
- There is a vast amount of different planning algorithms
  - If you don‘t expect large disturbances, you can simply solve an offline optimization problem to obtain an optimal trajectory (optimal racing line)
  - If you expect larger disturbances and want that the controller reacts to them in a more sophisticated way, you can directly implement the path planning inside an MPC (optimal racing line with competitors on the track)
  - Huge family of shortest path (label correcting algorithm -> breadth-first, depth-first, Dijkstra‘s algorithm, A*-algorithm, RRT)

Planning Algorithms

- What we will look at today:
    - For small state and input spaces, i.e., such that they can be discretized → Dynamic Progamming (gear shift optimization, grid world examples)
    - For continuous state and input spaces and complicated objectives → Reinforcement Learning (determining joint positions to reach desired end-effector positions, then track these joint positions with low-level MPC / PID controllers)

Dynamic Programming

- Problem setup:



$$\underbrace{g_N(x_N)}_{\text{terminal cost}} + \underbrace{\sum_{k=0}^{N-1} \underbrace{g_k(x_k, u_k, w_k)}_{\text{stage cost}}}_{\text{accumulated cost}}$$

k:	Discrete time index	$\boldsymbol{u_k} \in \boldsymbol{U(x_k)}$:	Control input in allowable set of inputs
N:	Time horizon	w_k:	Disturbance vector, $p_{w_k \mid x_k, u_k}$ known
$x_k \in S_k$:	State in allowable set of states	$f_k(.,.)$:	System dynamics

Dynamic Programming

- Key concept: „Principle of Optimality“

  Let $\pi^* = \left(\mu_0^*(.), \mu_1^*(.), \dots, \mu_{N-1}^*(.)\right)$ be an optimal policy. Consider the subproblem that we want to minimize:

$$\underset{(X_{i+1}, W_i | x_i = \mathrm{x})}{\mathrm{E}} \left[g_N(x_N) + \sum_{k=i}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \right]$$

  Then, the truncated policy $\left(\mu_i^*(.), \mu_{i+1}^*(.), \dots, \mu_{N-1}^*(.)\right)$ is optimal for this problem.

- Intuition: If the fastest way to drive from Zurich to Geneva is via Bern, then once you reach Bern, the route from Bern to Geneva must itself be the fastest possible. Otherwise the whole trip wouldn’t be optimal.

Dynamic Programming – Algorithm

Initialization

$$J_N(\mathrm{x}) := g_N(\mathrm{x}), \qquad \forall \mathrm{x} \in \mathcal{S}_N$$

Recursion

$$J_k(\mathrm{x}) := \min_{\mathrm{u} \in \mathcal{U}_k(\mathrm{x})} \mathrm{E}_{(w_k | x_k = \mathrm{x}, u_k = \mathrm{u})} \Big[g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \Big],$$

$$\forall \mathrm{x} \in \mathcal{S}_k, \ k = N-1, \ldots, 0.$$

→If for each k and each $x \in S_k$, $u^* =: \mu_k^*(x)$ minimizes the recursion equation, the policy $\pi^* = \left(\mu_0^*(.), \mu_1^*(.), \ldots, \mu_{N-1}^*(.)\right)$ is optimal.

Dynamic Programming

Implementation remarks:

- For each recursion step, you have to perform the optimization over all possible values of $x \in S_k$, since we don‘t know a priori which states will be visited

→This is why DP works well for grid world examples. If you have a continuous state space, then you need to discretize it → tradeoff: accuracy vs. computation

- The pointwise optimization in $x \in S_k$ gives the optimal policy $\mu^*(.)$

Dynamic Programming

Limitations:

- While the horizon length affects the computation linearly, the issue is with the size of the discretized state and action space.
- If the state space is discretized with n grid points per dimension d, then you have to check n^d states

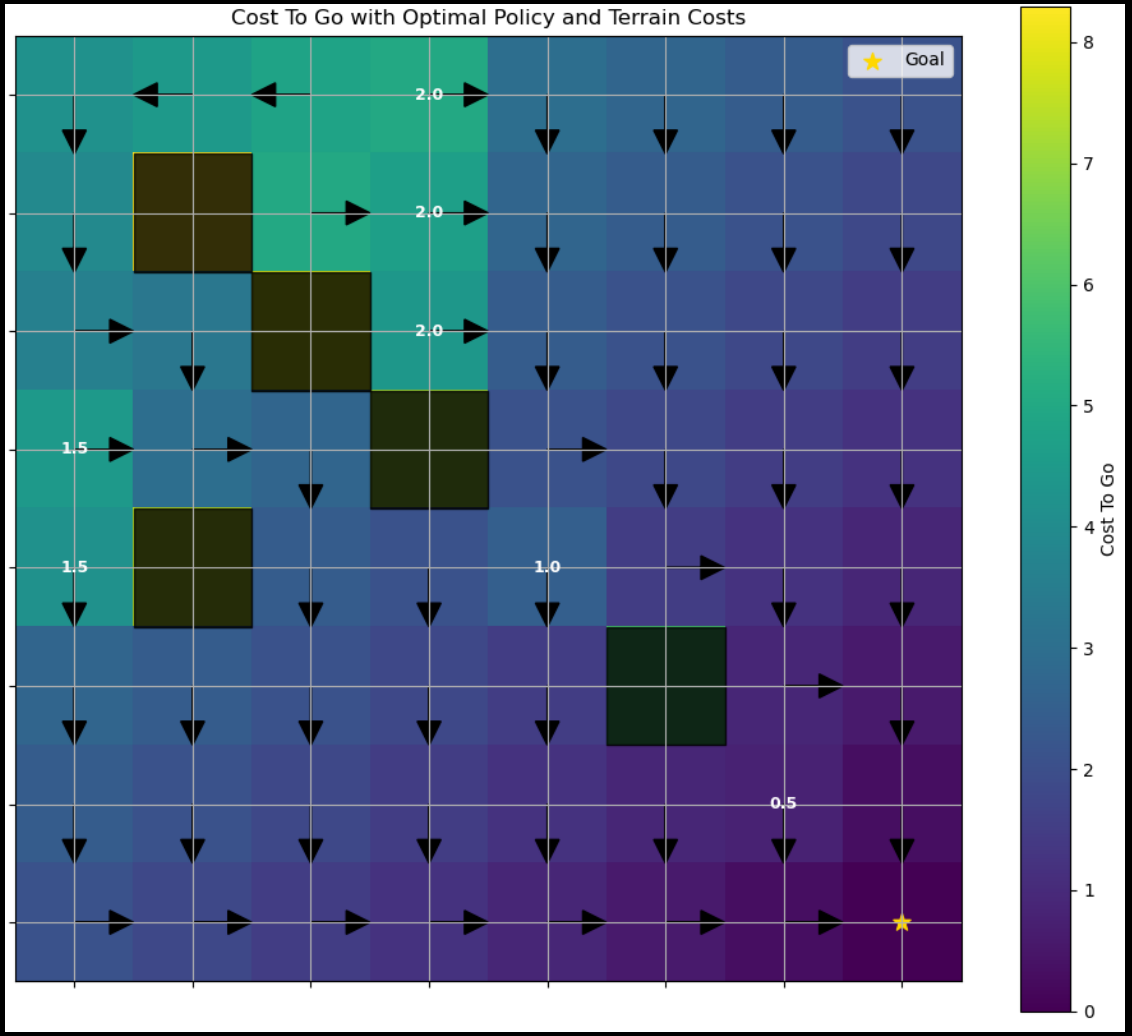
→ Curse of dimensionality

Dynamic Programming – Task 5 (backward_dp_varied_costs.py)

- Implement the backward DP algorithm for an 8x8 grid world, where obstacles are present and some fields incur higher costs. You can move around in this world with up, down, left, and right (everything already available).

- Use: available functions, min, np.argmin

Dynamic Programming – Task 5 (backward_dp_varied_costs.py)



Solution is quite intuitive:

Try to go down as fast as possible and then all the way to the right and avoid the additional cost coordinates.

Interesting starting point (3,7):

Instead of moving around the obstacles, it is cheaper to move through the block with additional cost

Dynamic Programming – Infinite Horizon?

- So far we have looked at finite horizon problems (there is a fixed terminal time).

- What if we have continuous operation or no defined time horizon?

- If instead of backwards you move forward and assume that your cost starts to converge at one point, then you arrive at the Bellman equation (BE):

$$J(\mathrm{x}) = \min_{\mathrm{u}\in\mathcal{U}(\mathrm{x})} \mathop{\mathrm{E}}_{(w|x=\mathrm{x},u=\mathrm{u})} \left[g(x,u,w) + J(f(x,u,w))\right], \quad \forall \mathrm{x} \in \mathcal{S}$$

→must be solved for all x simultaneously, this is analytically only possible for simple systems

Dynamic Programming – BE x Value Iteration

- Considering the following system

$$x_{k+1} = w_k, \qquad x_k \in \mathcal{S},$$
$$\Pr\left(w_k = j | x_k = i, u_k = \mathrm{u}\right) = P_{ij}(\mathrm{u}), \qquad \mathrm{u} \in \mathcal{U}(i),$$

with $q(i, \mathrm{u}) := \underset{(w|x=i,u=\mathrm{u})}{\mathrm{E}} [g(x, u, w)]$

it can be shown that the following value iteration algorithm converges to the optimal cost J^* as $l \to \infty$

$$V_{l+1}(i) = \min_{\mathrm{u} \in \mathcal{U}(i)} \left(q(i, \mathrm{u}) + \sum_{j=1}^{n} P_{ij}(\mathrm{u}) V_l(j) \right), \quad \forall i \in \mathcal{S}^+$$

→You apply this iteration until convergence. In practice it might take forever to reach convergence. Therefore, define a threshold for $\|V_{l+1}(i) - V_l(i)\|, \ \forall i \in \mathcal{S}^+$

Dynamic Programming – BE x Policy Iteration

Initialization: Initialize with a proper policy $\mu^0 \in \Pi$.

Stage 1 (*Policy Evaluation*): Given a policy μ^h, solve for the corresponding cost J_{μ^h} by solving the linear system of equations

$$J_{\mu^h}(i) = q\Big(i, \mu^h(i)\Big) + \sum_{j=1}^{n} P_{ij}(\mu^h(i)) J_{\mu^h}(j)\,, \ \forall i \in \mathcal{S}^+.$$

Stage 2 (*Policy Improvement*): Obtain a new stationary policy μ^{h+1} as follows

$$\mu^{h+1}(i) = \arg\min_{\mathrm{u} \in \mathcal{U}(i)} \left(q(i, \mathrm{u}) + \sum_{j=1}^{n} P_{ij}(\mathrm{u}) J_{\mu^h}(j) \right), \ \forall i \in \mathcal{S}^+.$$

Iterate between Stage 1 and 2 until $J_{\mu^{h+1}}(i) = J_{\mu^h}(i)$ for all $i \in \mathcal{S}^+$.

Under mild conditions (cost-free termination state), it can be shown that policy iteration converges in a finite number of steps.

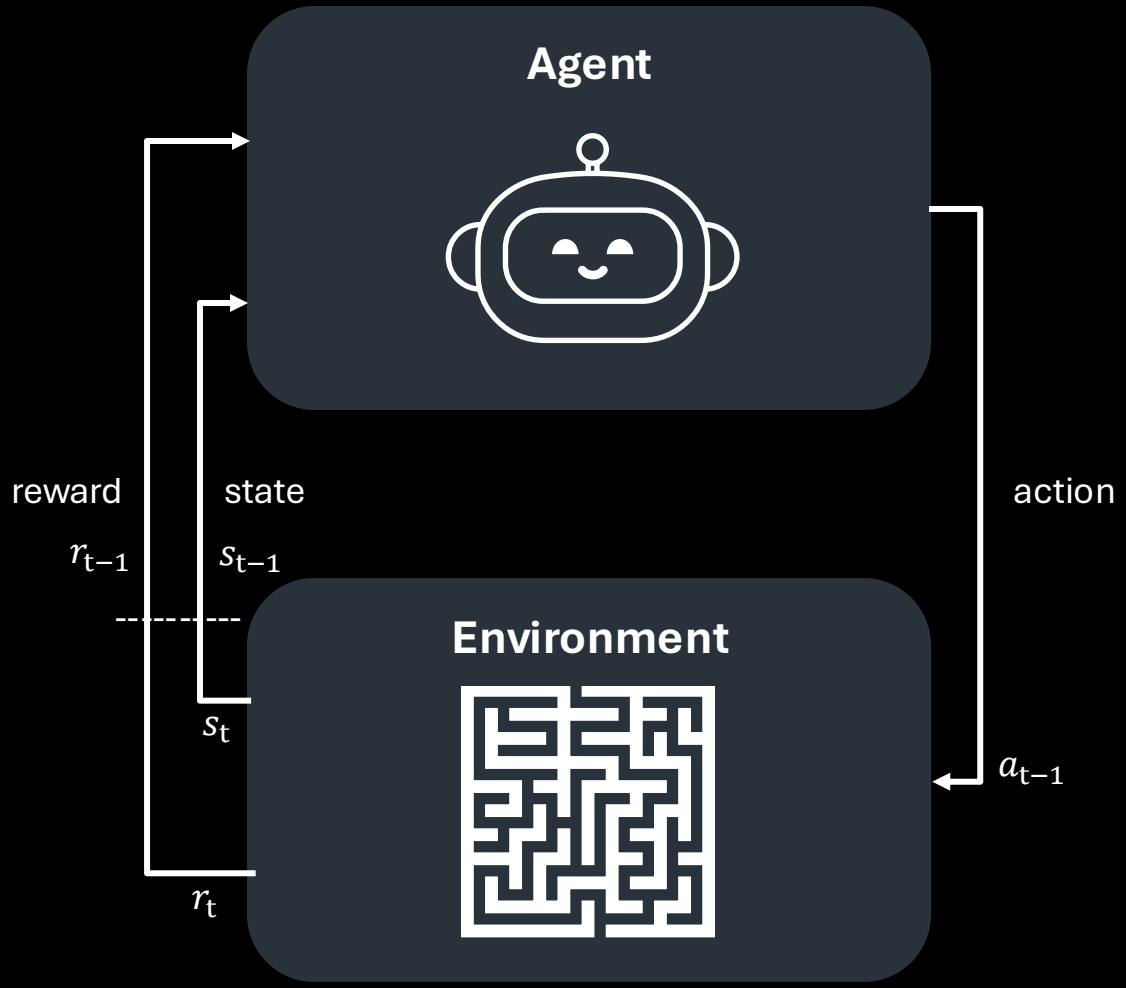
It‘s also possible to apply the algorithm asynchronously.

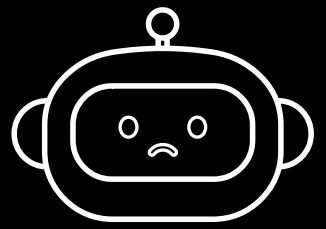
DP → Reinforcement Learning (RL)

- The idea of policy and value iteration algorithms together with the notion of optimal policies which are the optimizers of value functions lies at the heart of RL

- By generating data through interacting with an environment, the RL agent tries to approximate the optimal policy and the optimal value function through smart algorithms.

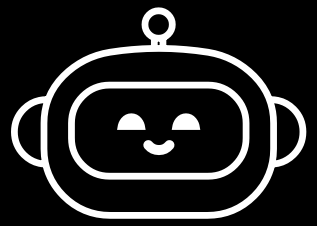
RL Setup





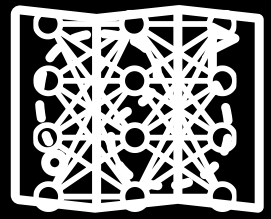
Action 1

Low Reward



Action 2

High Reward



Actions

States

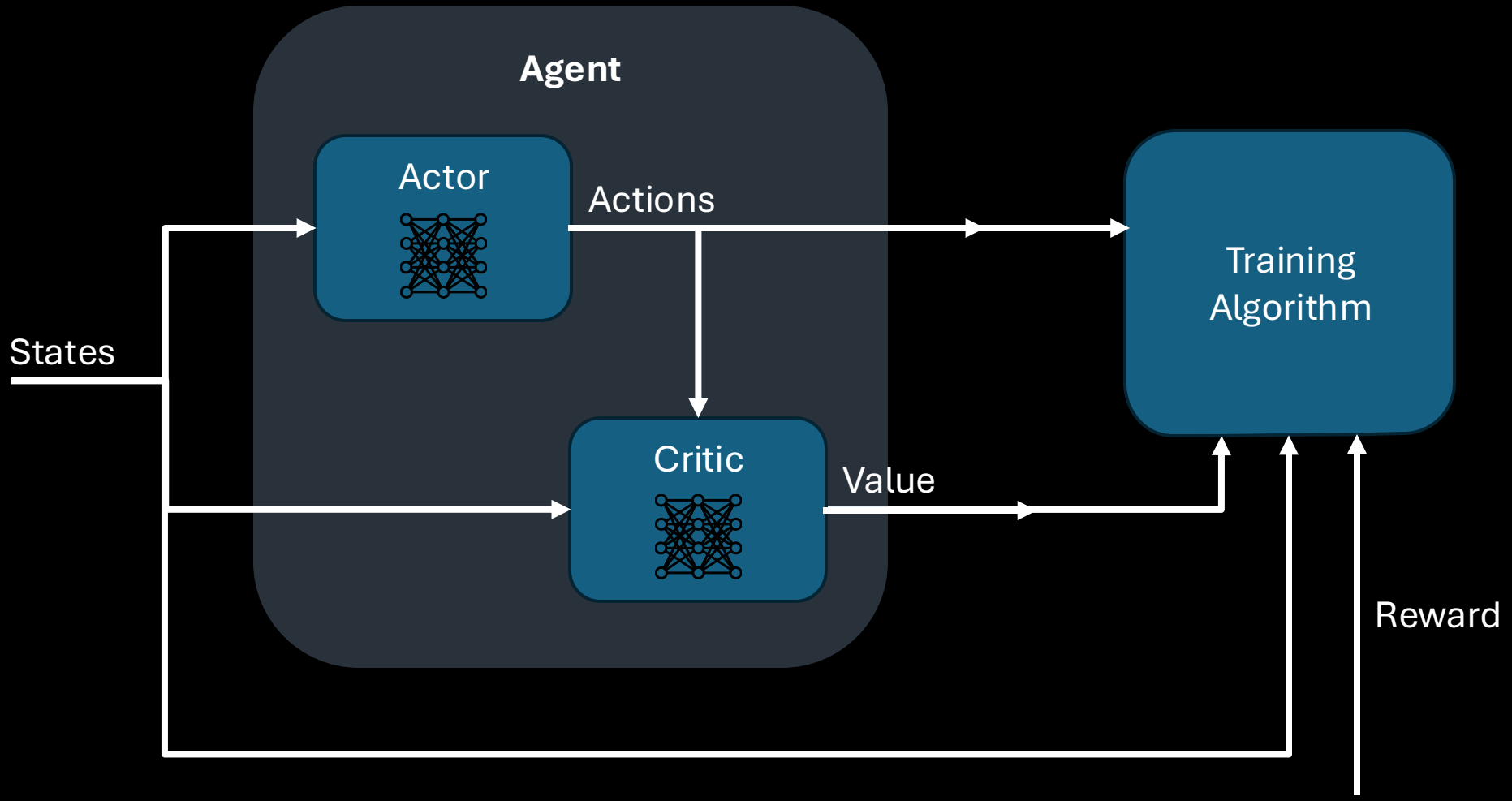
RL Environments

- The RL environment is everything outside of the agent

- There are different forms the environment can take:
  - Simple function (piece of software)
  - Real hardware
  - Simulations environments like Isaac Sim

→ Based on a current state and an action, it needs to output a reward and an updated state

RL Agents


Agent
Actor
Actions
States
Critic
Value
Training Algorithm
Reward

RL Training

Collecting Experiences (S, A, R, S')

Minibatch Sampling

Updating Network Weights

$$\theta_{\mathrm{a,k+1}} = \mathrm{argmax}_{\theta_\mathrm{a}} \mathcal{L}_\mathrm{a}(\theta_\mathrm{a}, s, a, \theta_\mathrm{c})$$

$$\theta_{\mathrm{c,k+1}} = \mathrm{argmin}_{\theta_\mathrm{c}} \mathcal{L}_\mathrm{c}(\theta_\mathrm{c}, s, a)$$

RL Hyperparameters

Learning rate:	How big each update step is. Too high = unstable; too low = slow learning
Discount factor:	How much the agent cares about the future. High = long-term; low = short-term
Batch size:	How many samples per update. Larger = smoother updates; smaller = more exploration
Replay buffer:	Memory of past experiences. Bigger = diverse data; smaller = recent data
Entropy:	How much exploration is encouraged. Higher = more random policies
# of hidden layers:	Network capacity; larger = more expressive but risk of overfitting/instability
Activation function:	Shape of nonlinearity; affects smoothness and stability
Reward scaling:	Normalizes reward magnitude; stabilizes training
Gradient clipping:	Prevents exploding gradients; smooths learning

RL Implementation

- For the environment you need either a piece of software or interface real hardware to software such that if you input a state and an action, it gives you an updated state and a reward.

- Regarding the agent, there exists a large amount of libraries (each has different RL algorithms). Find a tailored algorithm for your problem and read through the documentation to understand what options you can select.

- If you have the RL algorithm fixed, you also know the structure of your agent. If it‘s a neural network (actor-critic), then you design the desired neural network architecture.

RL Implementation

- Key functions are the reset function and the step function.
- At the beginning of each episode, we need to reset the environment and obtain an initial state.
- Next we have a for-loop over the whole episode where in each iteration we evaluate the step function. Then we store all state, action, and reward trajectories as our data.
- Every i-th episode, we then make use of the RL algorithm to update our policy based on the data we collected.

RL as a High-Level Planner

- Often when you have complicated objectives or lots of uncertainties, it makes sense to use RL as a high-level planner that provides a reference that can be tracked by a low-level controller (tracking is a standard control task and can be easily achieved)

- For example if you have a 3D model of your robot, and your goal is to reach a certain position. In simulation, RL does trial and error to find the trajetories of the joint positions to reach the desired target.

- Once the joint positions are known, a low-level controller simply needs to control the torque input to the motors to reach the desired joint positions.

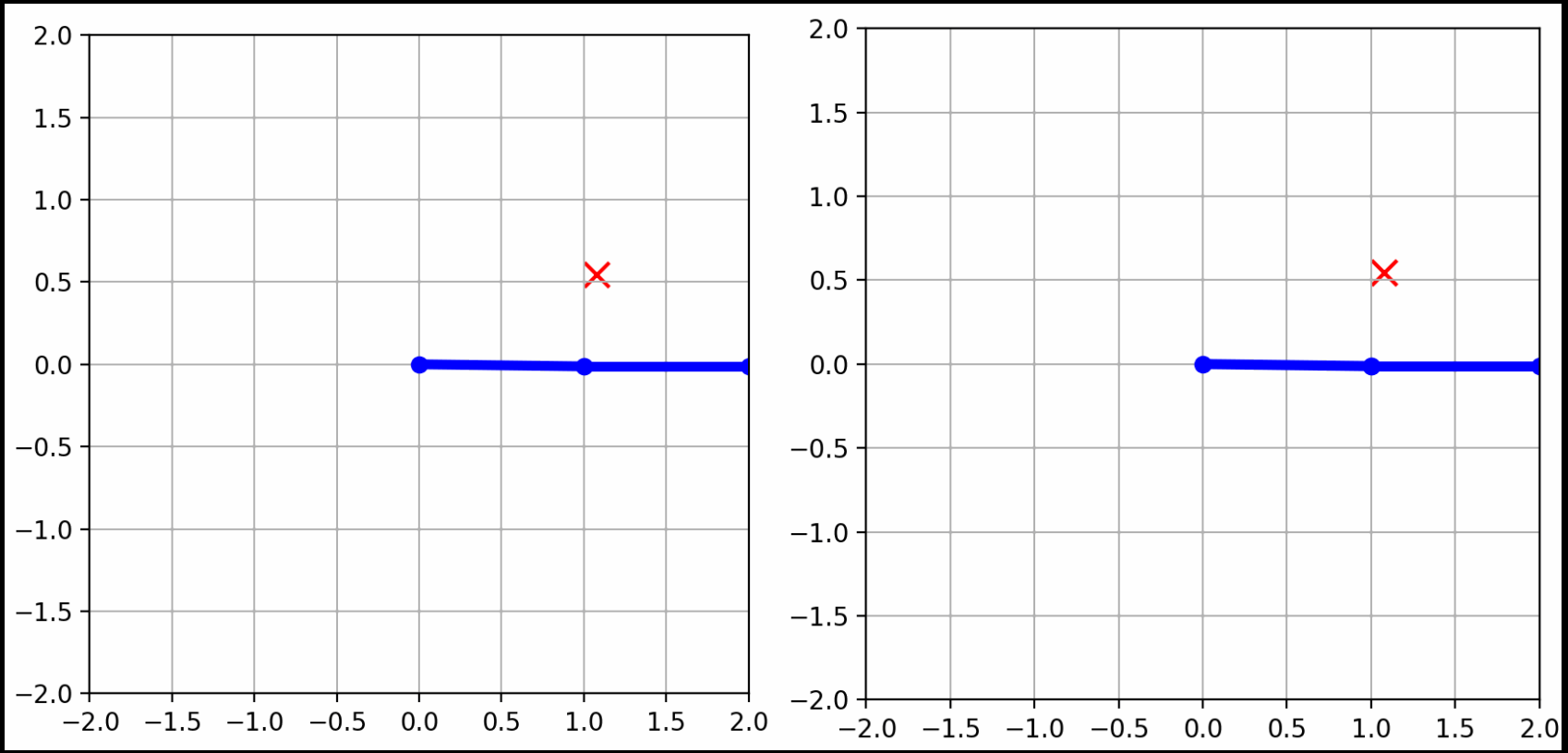
RL – Task 6 (train_end_effector_tracking.py)

- Setup the RL training pipeline, to train an agent to output joint positions for desired end-effector positions (dynamics and reset functions are already available, just implement the step function).

- A low-level PID controller (run_end_effector_tracking_pid.py) will then take care of applying the correct torque to the motors to arrive at these positions (either use your trained model -> takes a few minutes depending on the number of episodes or use my pretrained model)

- Use: np.clip (for actions) and the available functions

RL – Task 6 (train_end_effector_tracking.py)



Left: high P/D ratio

Right: low P/D ratio

→ while RL does a lot, the tuning of the controller is still important!

RL not yet perfectly trained, as there's an offset to the desired end-effector position

→ Hyperparameter tuning

Summary

- State estimation: fuse different sensor sources and extract as much information as possible
- Saw different levels of complexity → start easy, if it works, it works and you can save the computational capacity for other potentially more demanding tasks
- Planning is a crucial module in the control stack if you want that your system becomes autonomous
- Discrete / small state and input spaces use DP, for continuous state and input spaces and complex objectives RL is an interesting alternative

Credits

Lots of material taken from the following courses:

- Control Systems II by Lino Guzzella
- Recursive Estimation & Dynamic Programming and Optimal Control by Raffaelo d‘Andrea