# Basic SLAM Tutorials for Everyone
# Lec. #5. Filtering vs Smoothing (1): Understanding the Kalman Filter from a MAP Perspective

Hyungtae Lim, Ph.D.

<shapelim [at] mit [dot] edu> / <fudxo5143+slam [at] gmail [dot] com>

# Filtering vs Smoothing

## Filtering (e.g., Kalman Filter)

- ▶ Processes data **sequentially**.
- ▶ Maintains only current state $(\hat{x}_k, P_k)$.
- ▶ **Marginalizes** out all past states.
- ⊕ Constant memory and time per step.
- ⊖ Past states cannot be revised.
- ⊖ Linearization errors are permanent.

## Smoothing (e.g., Graph SLAM)

- ▶ Optimizes **all states jointly**.
- ▶ Retains the full trajectory.
- ▶ **Relinearizes** at each solver iteration.
- ⊕ Loop closures improve all poses.
- ⊕ Better accuracy for nonlinear models.
- ⊖ Cost grows with trajectory (mitigated by sparsity).

# Filtering vs Smoothing

## Filtering (e.g., Kalman Filter)

- ▶ Processes data **sequentially**.
- ▶ Maintains only current state $(\hat{x}_k, P_k)$.
- ▶ **Marginalizes** out all past states.
- ⊕ Constant memory and time per step.
- ⊖ Past states cannot be revised.
- ⊖ Linearization errors are permanent.

## Smoothing (e.g., Graph SLAM)

- ▶ Optimizes **all states jointly**.
- ▶ Retains the full trajectory.
- ▶ **Relinearizes** at each solver iteration.
- ⊕ Loop closures improve all poses.
- ⊕ Better accuracy for nonlinear models.
- ⊖ Cost grows with trajectory (mitigated by sparsity).

# 1D State Estimation: Problem Setup

Consider a robot moving along a line. At each time step $k$:

- The robot applies a control input $u_k$ (e.g., move 1 m forward).
- It receives a noisy position measurement $z_k$ from a sensor.

**Motion model** (how the state evolves):

$$x_k = x_{k-1} + u_k + w_k, \qquad w_k \sim \mathcal{N}(0, \sigma_w^2)$$

**Observation model** (what the sensor measures):

$$z_k = x_k + v_k, \qquad v_k \sim \mathcal{N}(0, \sigma_v^2)$$

**Goal:** Estimate the true position $x_k$ at each time step, given all controls $u_{1:k}$ and measurements $z_{1:k}$.

Two fundamentally different approaches:

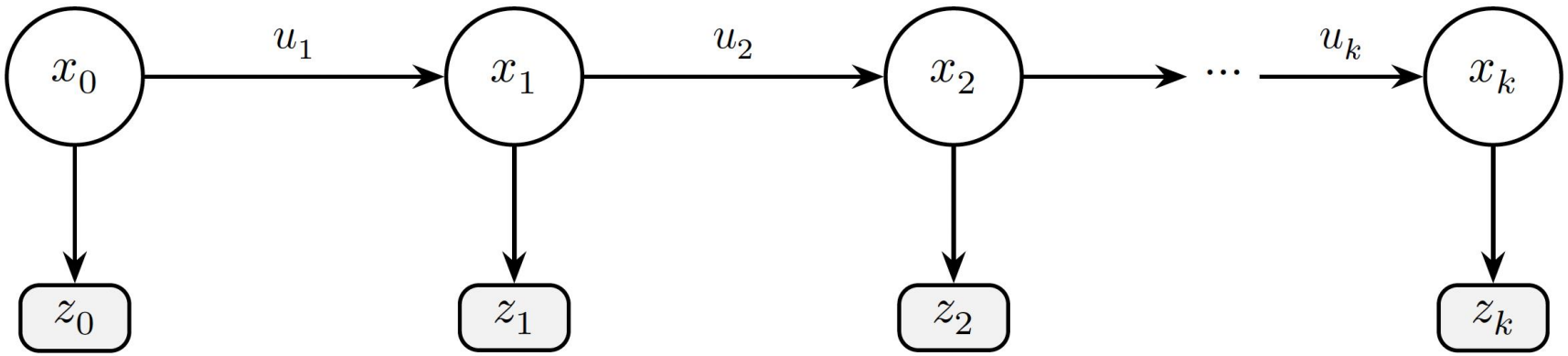
1. **Filtering** (Kalman filter): process measurements one at a time.
2. **Smoothing** (Graph optimization): optimize all states jointly.

# The Markov Assumption

The **Markov assumption** states that the current state depends only on the immediately preceding state and the current input:

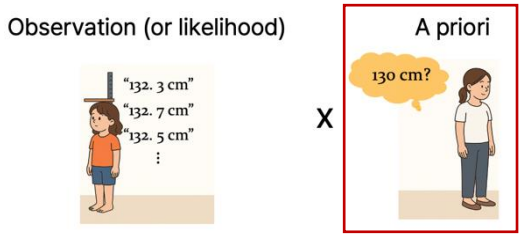
$$p(x_k \mid x_{0:k-1}, z_{1:k-1}, u_{1:k}) = p(x_k \mid x_{k-1}, u_k)$$

This leads to a chain-structured probabilistic model:




- **Circles**: hidden states $x_k$ (to be estimated).
- **Rectangles**: observed measurements $z_k$ (known).
- **Arrows**: causal dependencies (motion model, observation model).

# 1D Kalman Filter: Prediction Step




- *Prediction step* yields the *a priori* estimate
- The predict step *propagates* the state and its uncertainty forward using the motion model.

**Given:** Previous estimate $\hat{x}_{k-1}$ with variance $P_{k-1}$, control input $u_k$.

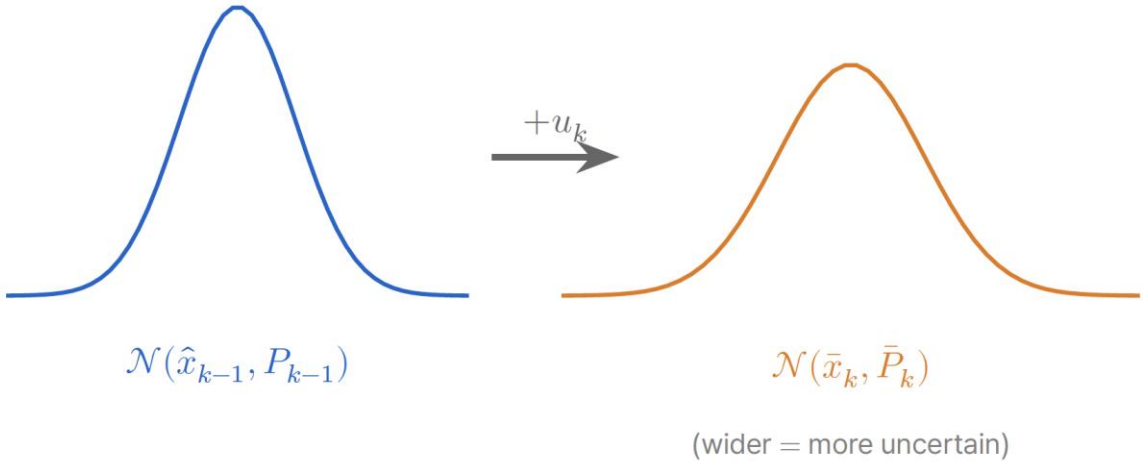
**Predicted state** (prior):

$$\bar{x}_k = \hat{x}_{k-1} + u_k$$

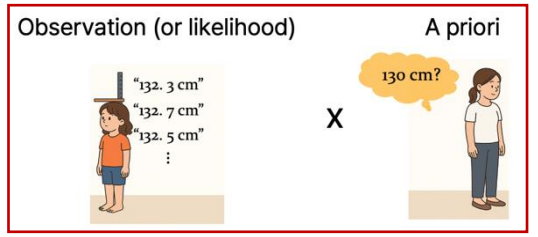
**Predicted variance** (grows):

$$\bar{P}_k = P_{k-1} + \sigma_w^2$$

Prediction **always increases** uncertainty due to motion noise $\sigma_w^2$.



# 1D Kalman Filter: Update Step




- The update step fuses the prediction with the new measurement $z_k$

**Kalman gain** (how much to trust the measurement):

$$K_k = \frac{\bar{P}_k}{\bar{P}_k + \sigma_v^2}$$

**Updated state** (posterior):

$$\hat{x}_k = \bar{x}_k + K_k \left( z_k - \bar{x}_k \right)$$

**Updated variance** (uncertainty shrinks):

$$P_k = \left( 1 - K_k \right) \bar{P}_k$$

**$K_k \to 1$ (trust measurement)**

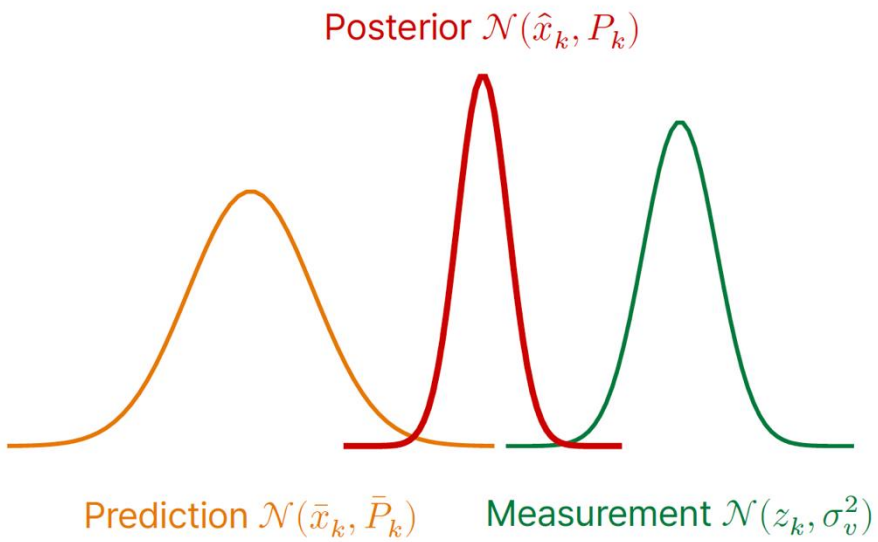
- Small $\sigma_v^2$ (precise sensor).
- Large $\bar{P}_k$ (uncertain prediction).
- $\hat{x}_k \approx z_k$.

**$K_k \to 0$ (trust prediction)**

- Large $\sigma_v^2$ (noisy sensor).
- Small $\bar{P}_k$ (confident prediction).
- $\hat{x}_k \approx \bar{x}_k$.

# ∴ Kalman Filter Update = Weighted Average (MAP)

The update equation is exactly the MAP estimate from Topic 1:




The posterior is a **weighted average**, weighted by precision (inverse variance):

$$\hat{x}_k = \frac{\bar{P}_k^{-1} \cdot \bar{x}_k + \sigma_v^{-2} \cdot z_k}{\bar{P}_k^{-1} + \sigma_v^{-2}} = \frac{\omega_{\text{pred}} \cdot \bar{x}_k + \omega_{\text{meas}} \cdot z_k}{\omega_{\text{pred}} + \omega_{\text{meas}}}$$

Identical to Example B (height estimation with prior). The posterior is always **narrower** than either input.

# Kalman Filter for Multivariable Systems

| 1D | Multivariable |
|---|---|
| **Kalman gain** (how much to trust the measurement): | |
| $K_k = \frac{\bar{P}_k}{\bar{P}_k + \sigma_v^2}$ | $\boldsymbol{K} = \Sigma_1(\Sigma_1 + \Sigma_2)^{-1}$ |
| **Updated state** (posterior): | |
| $\hat{x}_k = \bar{x}_k + K_k\,(z_k - \bar{x}_k)$ | $\hat{\boldsymbol{x}}_{\text{update}} = \boldsymbol{x}_1 + \boldsymbol{K}(\boldsymbol{x}_2 - \boldsymbol{x}_1)$ |
| **Updated variance** (uncertainty shrinks): | |
| $P_k = (1 - K_k)\,\bar{P}_k$ | $\Sigma_{\text{update}} = \Sigma_1 - \boldsymbol{K}\Sigma_1$ |

# Kalman Filter for Multivariable Systems (Cont'd)

- Prediction

$$\bar{\boldsymbol{x}}_k = \boldsymbol{A}\boldsymbol{x}_{k-1} + \boldsymbol{B}\boldsymbol{u}_k$$

$$\bar{\Sigma}_k = \boldsymbol{A}\Sigma_{k-1}\boldsymbol{A}^\intercal + \boldsymbol{Q}$$

Variable
Input
Constant (but modeling is required)
Constant (but tuning is required)

# Kalman Filter for Multivariable Systems (Cont'd)

- Prediction

$$\bar{\boldsymbol{x}}_k = \boldsymbol{A}\boldsymbol{x}_{k-1} + \boldsymbol{B}\boldsymbol{u}_k$$

$$\bar{\Sigma}_k = \boldsymbol{A}\Sigma_{k-1}\boldsymbol{A}^\intercal + \boldsymbol{Q}$$

Variable | Input
Constant (but modeling is required)
Constant (but tuning is required)

- Update: Let $\boldsymbol{z} = \boldsymbol{H}\boldsymbol{x}$ and its corresponding covariance is $\boldsymbol{R}$

$$\boldsymbol{K} = \boldsymbol{H}\bar{\Sigma}_k\boldsymbol{H}^\intercal(\boldsymbol{H}\bar{\Sigma}_k\boldsymbol{H}^\intercal + \boldsymbol{R})^{-1}$$

$$\boldsymbol{H}\hat{\boldsymbol{x}}_k = \boldsymbol{H}\bar{\boldsymbol{x}}_k + \boldsymbol{K}(\boldsymbol{z} - \boldsymbol{H}\bar{\boldsymbol{x}})$$

$$\boldsymbol{H}\hat{\Sigma}_k\boldsymbol{H}^\intercal = \boldsymbol{H}\bar{\Sigma}_k\boldsymbol{H}^\intercal - \boldsymbol{K}\boldsymbol{H}\bar{\Sigma}_k\boldsymbol{H}^\intercal$$

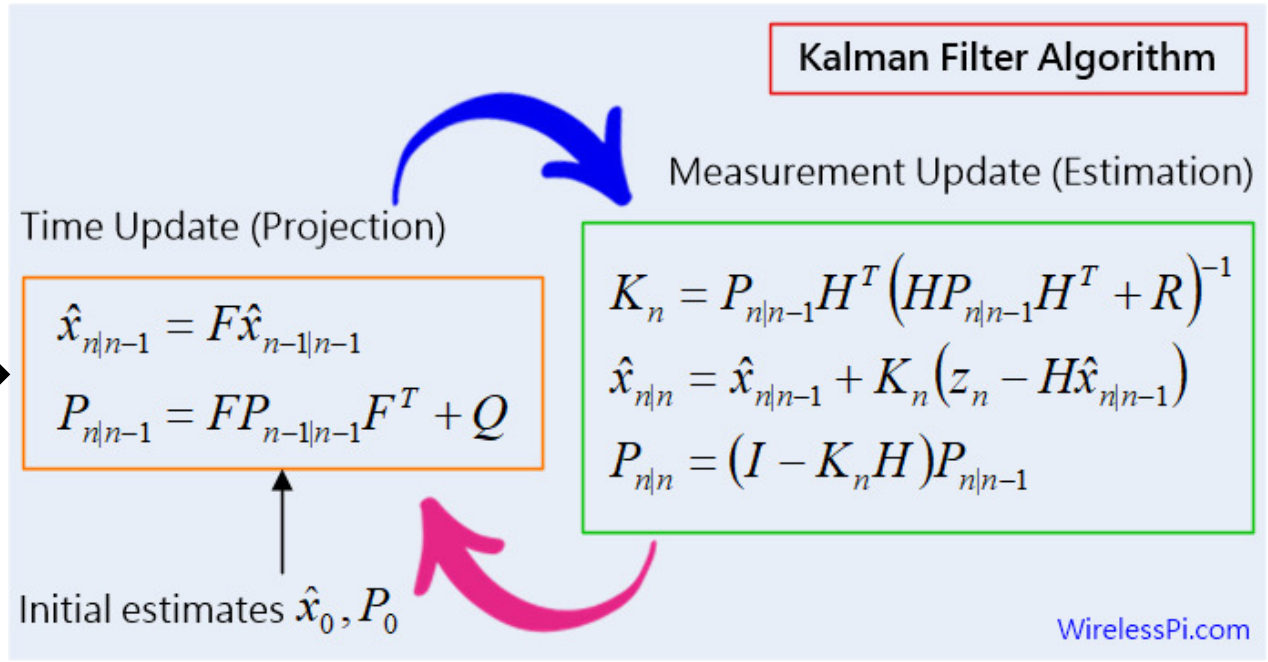
$$\boldsymbol{K} = \Sigma_1(\Sigma_1 + \Sigma_2)^{-1}$$

$$\hat{\boldsymbol{x}}_{\text{update}} = \boldsymbol{x}_1 + \boldsymbol{K}(\boldsymbol{x}_2 - \boldsymbol{x}_1)$$

$$\Sigma_{\text{update}} = \Sigma_1 - \boldsymbol{K}\Sigma_1$$

# Kalman Filter for Multivariable Systems (Cont'd)

$$
\begin{aligned}
\boldsymbol{K}' &= \bar{\Sigma}_k \boldsymbol{H}^\intercal \left(\boldsymbol{H} \bar{\Sigma}_k \boldsymbol{H}^\intercal + \boldsymbol{R}\right)^{-1} \\
\hat{\boldsymbol{x}}_k &= \bar{\boldsymbol{x}}_k + \boldsymbol{K}' \left(\boldsymbol{z} - \boldsymbol{H} \bar{\boldsymbol{x}}\right) \\
\hat{\Sigma}_k &= \bar{\Sigma}_k - \boldsymbol{K}' \boldsymbol{H} \bar{\Sigma}_k
\end{aligned}
$$



# Conclusion

# Conclusion (Cont'd)

- In real-world, we use Extended Kalman Filter (EKF) or Invariant Extended Kalman Filter (IEKF)
- Don't implement from scratch (i.e., DO NOT reinvent the wheel)
  - Leverage well-established libraries

```
class Ekf : public FilterBase
{
public:
  /**
   * @brief Constructor for the Ekf class
   */
  Ekf();

  /**
   * @brief Destructor for the Ekf class
   */
  ~Ekf();

  /**
   * @brief Carries out the correct step in the predict/update cycle.
   *
   * @param[in] measurement - The measurement to fuse with our estimate
   */
  void correct(const Measurement & measurement) override;

  /**
   * @brief Carries out the predict step in the predict/update cycle.
   *
   * Projects the state and error matrices forward using a model of the
   * vehicle's motion.
   *
   * @param[in] reference_time - The time at which the prediction is being made
   * @param[in] delta - The time step over which to predict.
   */
  void predict(
    const rclcpp::Time & reference_time,
    const rclcpp::Duration & delta) override;
};
```

https://github.com/cra-ros-pkg/robot_localization

```
73  class InEKF {
74
75      public:
76          EIGEN_MAKE_ALIGNED_OPERATOR_NEW
77          InEKF();
78          InEKF(NoiseParams params);
79          InEKF(RobotState state);
80          InEKF(RobotState state, NoiseParams params);
81
82          RobotState getState();
83          NoiseParams getNoiseParams();
84          mapIntVector3d getPriorLandmarks();
85          std::map<int,int> getEstimatedLandmarks();
86          std::map<int,bool> getContacts();
87          std::map<int,int> getEstimatedContactPositions();
88          void setState(RobotState state);
89          void setNoiseParams(NoiseParams params);
90          void setPriorLandmarks(const mapIntVector3d& prior_landmarks);
91          void setContacts(std::vector<std::pair<int,bool> > contacts);
92
93          void Propagate(const Eigen::Matrix<double,6,1>& m, double dt);
94          void Correct(const Observation& obs);
95          void CorrectLandmarks(const vectorLandmarks& measured_landmarks);
96          void CorrectKinematics(const vectorKinematics& measured_kinematics);
```

https://github.com/RossHartley/invariant-ekf