Autonomous Mobile Robot Design

Kalman Filter – A Primer

Dr. Kostas Alexis (CSE)

LAB mini courses

Estimation - 01 – Kalman Filter

Subsection 1: A brief introduction

Why using a Kalman Filter?

- Sensor readings are inaccurate, uncertain and noisy.
- Robot models are uncertain.
- In many cases, sensor and process uncertainty can be modeled with simply statistical assumptions and eventually utilize Gaussian models.
- The Kalman Filter is "as is" applicable to Linear Systems with Gaussian Noise but its extensions can deal with much more complicated cases.
- A "must-know" if one is to work on anything related with robotics and navigation systems (and not only).

What is a Kalman Filter?

- **A Kalman Filter is an optimal estimator** – i.e. it infers parameters of interest from indirect, inaccurate and uncertain observations. It is recursive so that new measurements can be processed as they arrive.

- **Optimality in what sense?** If all noise is Gaussian, the Kalman Filter minimizes the mean square error of the estimated parameters.

- **What if the noise is not Gaussian?** Given only the mean and standard deviation of noise, the Kalman Filter is the best linear estimator. Non-linear estimators may be better.

Why is Kalman Filtering so popular?

- Good results in practice due to optimality and structure
- Convenient form for online real-time processing
- Easy to formulate and implement given a basic understanding
- Measurement equations need not to be inverted
- It is highly versatile and used for:
  - Estimation
  - Filtering
  - Prediction
  - Fusion

LAB mini courses

Estimation - 01 – Kalman Filter

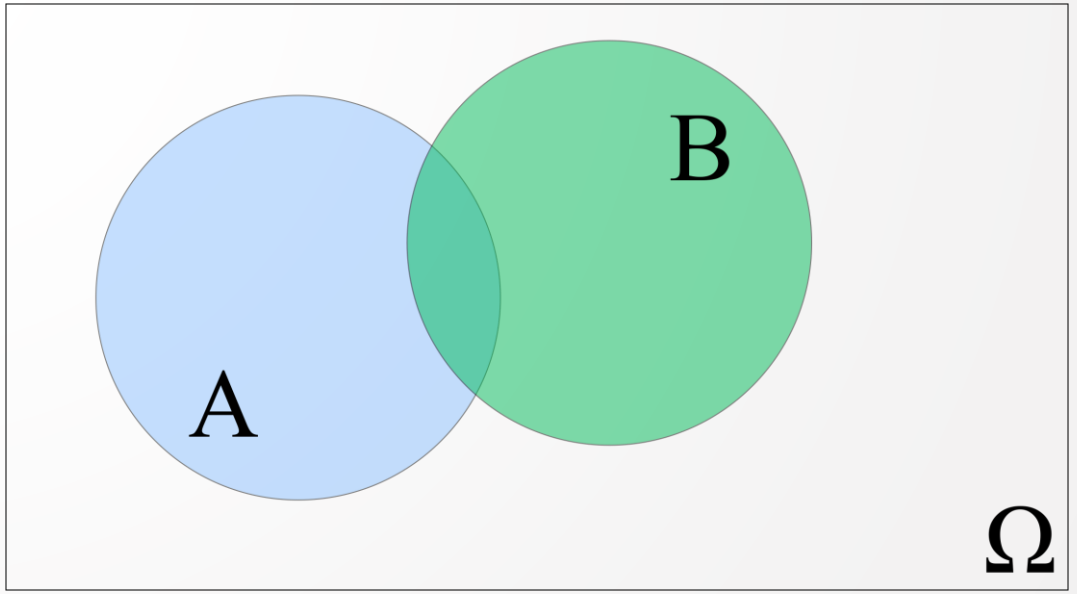
Subsection 2: Recap on Probability and Statistics

Probability theory

- **Random experiment** that can produce a number of outcomes, e.g. a rolling dice.
- Sample space, e.g.: $\{1,2,3,4,5,6\}$
- Event A is subset of outcomes, e.g. $\{1,3,5\}$
- Probability $P(A)$, e.g. $P(A)=0.5$

Axioms of Probability theory

- $0 \leq P(A) \leq 1$
- $P(\Omega) = 1, \;\; P(\emptyset) = 0$
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$



Discrete Random Variables

- X denotes a random variable
- X can take on a countable number of values in $\{x_1, x_2, \ldots, x_n\}$
- $P(X=x_i)$ is the probability that the random variable X takes on value x_i
- $P(.)$ is called the probability mass function

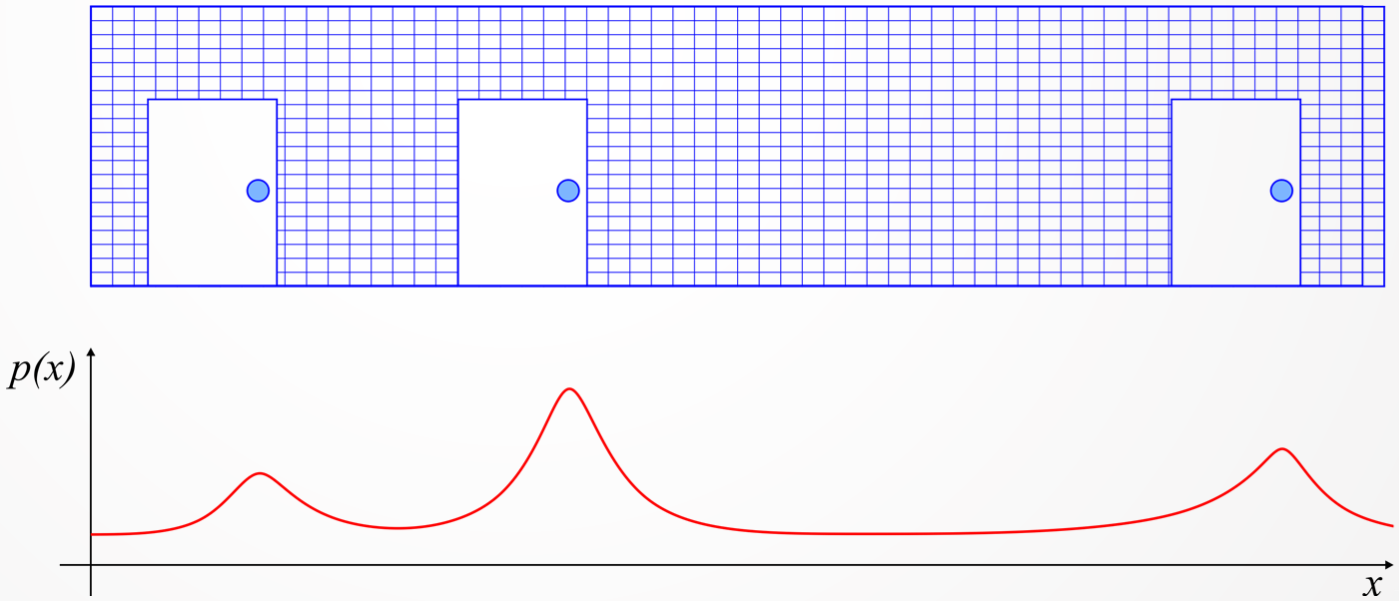
- Example: P(Room)=<0.6,0.3,0.06,0.03>, Room one of the office, corridor, lab, kitchen

Continuous Random Variables

- X takes on continuous values.
- $P(X=x)$ or $P(x)$ is called the **probability density function (PDF)**.

Thrun, Burgard, Fox, "Probabilistic Robotics", MIT Press, 2005

- Example:



Proper Distributions Sum To One

- Discrete Case $$\sum_x P(x) = 1$$
- Continuous Case $$\int p(x)dx = 1$$

Joint and Conditional Probabilities

- $p(X = x, \text{ and } Y = y) = P(x, y)$
- If X and Y are **independent** then:

$$P(x, y) = P(x)P(y)$$

- Is the probability of **x given y**

$$P(x|y)P(y) = P(x, y)$$

- If X and Y are independent then:

$$P(x|y) = P(x)$$

Conditional Independence

- Definition of conditional independence:

$$P(x, y|z) = P(x|z)P(y|z)$$

- Equivalent to:

$$P(x|z) = P(x|y, z)$$
$$P(y|z) = P(y|x, z)$$

- Note: this does not necessarily mean that:

$$P(x, y) = P(x)P(y)$$

Marginalization

- **Discrete case:** $$P(x) = \sum_y P(x, y)$$
- **Continuous case:** $$p(x) = \int p(x, y) dy$$

Marginalization example

P(X,Y)	x1	x1	x1	x1	P(Y) ↓
y1	1/8	1/16	1/32	1/32	1/4
y1	1/16	1/8	1/32	1/32	1/4
y1	1/16	1/16	1/16	1/16	1/4
y1	1/4	0	0	0	1/4
P(X) →	1/2	1/4	1/8	1/8	1

Expected value of a Random Variable

- **Discrete case:** $$E[X] = \sum_i x_i P(x_i)$$
- **Continuous case:** $$E[X] = \int x P(X = x) dx$$
- The expected value is the weighted average of all values a random variable can take on.
- Expectation is a linear operator:

$$E[aX + b] = aE[X] + b$$

Covariance of a Random Variable

- Measures the **square expected deviation from the mean**:

$$Cov[X] = E[X - E[X]]^2 = E[X^2] - E[X]^2$$

Estimation from Data

- Observations: $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathcal{R}^d$
- Sample Mean: $\mu = \frac{1}{n} \sum_i \mathbf{x}_i$
- Sample Covariance:

$$\Sigma = \frac{1}{n-1} \sum_i (\mathbf{x}_i - \mu)(\mathbf{x}_i - \mu)$$

Maximum Likelihood

- Maximum likelihood, also called the maximum likelihood method, is the procedure of finding the value of one or more parameters for a given statistic which makes the known likelihood distribution a maximum.
- For a measurement signal y and the state to be inferred $\hat{x}$, assuming that the additive random noise is Gaussian distributed with a standard deviation σ_k gives:

$$P(y_k, \hat{x}_k) = K_k e^{-(\frac{(y_k - a_k\hat{x}_k)^2}{2\sigma_k^2})}, \; y_k = a_k x_k + n_k$$

$$P(y\ \hat{x}) = \Pi_k K_k e^{-(\frac{(y_k - a_k\hat{x}_k)^2}{2\sigma_k^2})}$$

$$logP(y\ \hat{x}) = -\frac{1}{2}\sum_k \frac{(y_k - a_k\hat{x}_k)^2}{2\sigma_k^2} + \text{constant}$$

LAB mini courses

Estimation - 01 – Kalman Filter

Subsection 3.1: Before going to Kalman – the Bayes Filter

Markov Assumption

- Observations depend only on current state

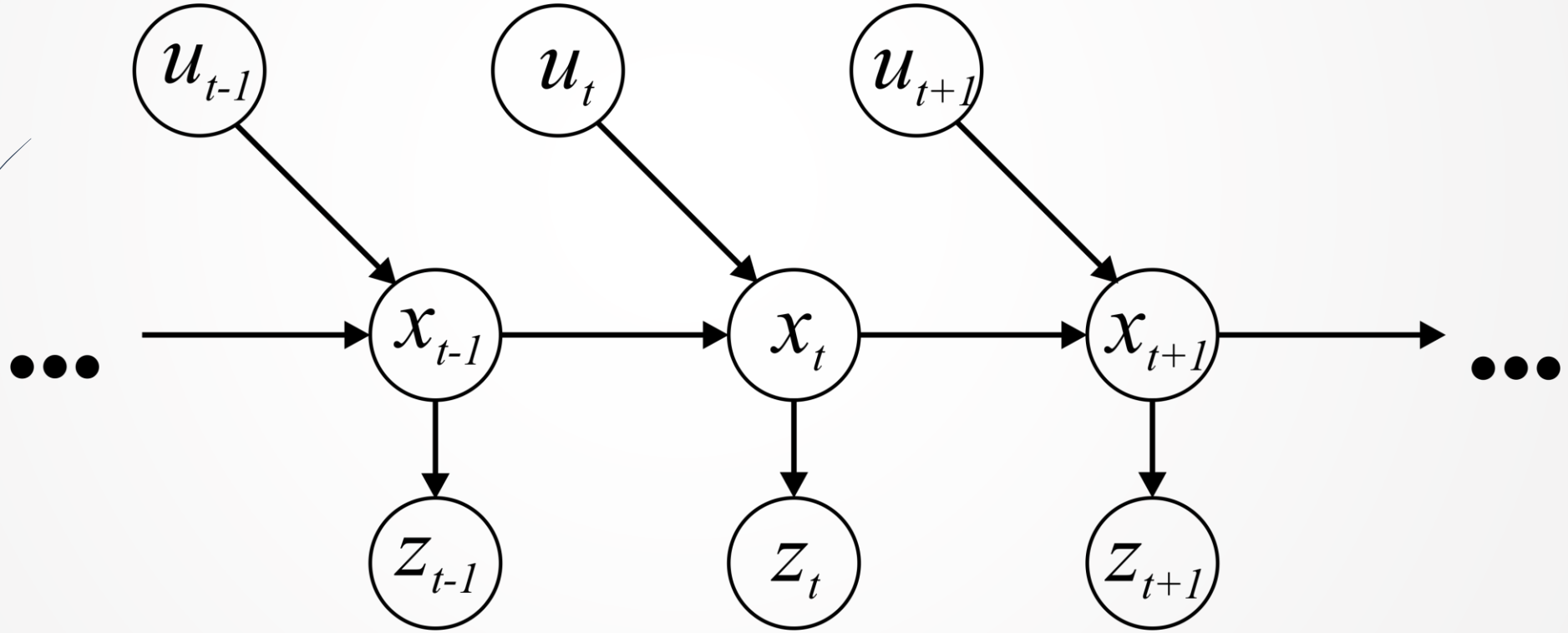
$$P(z_t|x_{0:t}, z_{1:t-1}, u_{1:t}) = P(z_t|x_t)$$

- Current state depends only on previous state and current action

$$P(x_t|x_{0:t}, z_{1:t}, u_{1:t}) = P(x_t|x_{t-1}, u_t)$$

Markov Chain

- A Markov Chain is a stochastic process where, given the present state, the past and the future states are independent.



Underlying Assumptions

- Static world
- Independent noise
- Perfect model, no approximation errors

Bayes Filter

- **Given**
  - Sequence of observations and actions: z_t, u_t
  - Sensor model: $P(z|x)$
  - Action model: $P(x'|x,u)$
  - Prior probability of the system state: $P(x)$
- **Desired**
  - Estimate of the state of the dynamic system: x
  - Posterior of the state is also called belief:

$$Bel(x_t) = P(x_t|u_1, z_1, \ldots, u_t, z_t)$$

Bayes Filter Algorithm

- **For each time step, do:**
  - Apply motion model:

$$\overline{Bel}(x_t) = \sum_{x_t - 1} P(x_t | x_{t-1}, u_t) Bel(x_{t-1})$$

  - Apply sensor model:

$$Bel(x_t) = \eta P(z_t | x_t) \overline{Bel}(x_t)$$

  - η is a normalization factor to ensure that the probability is maximum 1.

Notes

- Bayes filters also work on continuous state spaces (replace sum by integral).
- Bayes filter also works when actions and observations are asynchronous.

LAB mini courses

Estimation - 01 – Kalman Filter

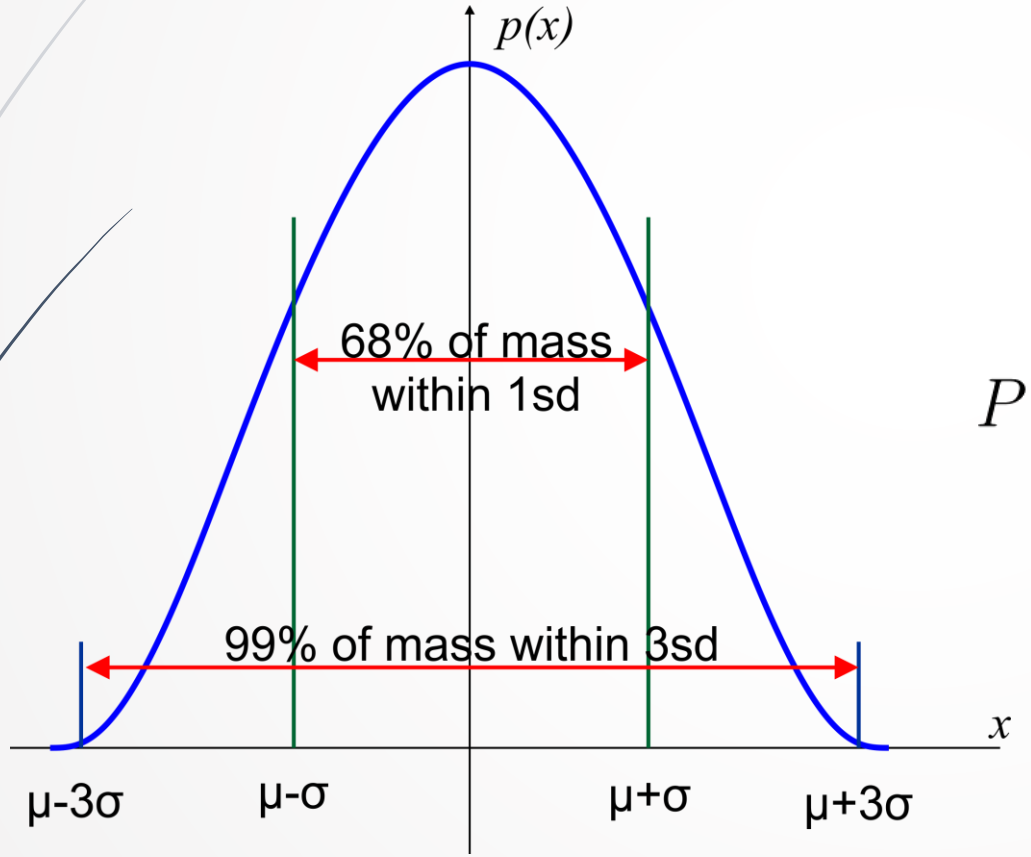
Subsection 3.2: From Bayes to Kalman Filter

Introduction

- The Kalman Filter (KF) has long been regarded as the optimal solution to many tracking and data prediction tasks.

Kalman Filter

- Univariate distribution



$$X \sim \mathcal{N}(\mu, \sigma^2)$$

mean

Variance (squared standard deviation)

$$P(X = x) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2})$$

Kalman Filter

- Multivariate normal distribution: $\mathbf{X} \sim \mathcal{N}(\mu, \Sigma)$
- Mean: $\mu \in \mathcal{R}^n$
- Covariance: $\Sigma \in \mathbf{R}^{n \times m}$
- Probability density function:

$$p(\mathbf{X} = \mathbf{x}) = \mathcal{N}(\mathbf{x}; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp(-\frac{1}{2}(\mathbf{x} - \mu)^T \Sigma^{-1}(\mathbf{x} - \mu))$$

Properties of Normal Distributions

- Linear transformation – remains Gaussian

$$\mathbf{X} \sim \mathcal{N}(\mu, \Sigma), \mathbf{Y} \sim \mathbf{A}\mathbf{X} + \mathbf{B}$$
$$\Rightarrow \mathbf{Y} \sim \mathcal{N}(\mathbf{A}\mu + \mathbf{B}, \mathbf{A}\Sigma\mathbf{A}^T)$$

- Intersection of two Gaussians – remains Gaussian

$$\mathbf{X}_1 \sim \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1), \mathbf{X}_2 \sim \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$$

$$p(\mathbf{X}_1)p(\mathbf{X}_2) = \mathcal{N}\left(\frac{\boldsymbol{\Sigma}_2}{\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2}\boldsymbol{\mu}_1 + \frac{\boldsymbol{\Sigma}_1}{\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2}\boldsymbol{\mu}_2, \frac{1}{\boldsymbol{\Sigma}_1^{-1} + \boldsymbol{\Sigma}_2^{-1}}\right)$$

Properties of Normal Distributions

- Linear transformation – remains Gaussian

$$\mathbf{X} \sim \mathcal{N}(\mu, \Sigma), \mathbf{Y} \sim \mathbf{A}\mathbf{X} + \mathbf{B}$$
$$\Rightarrow \mathbf{Y} \sim \mathcal{N}(\mathbf{A}\mu + \mathbf{B}, \mathbf{A}\Sigma\mathbf{A}^T)$$

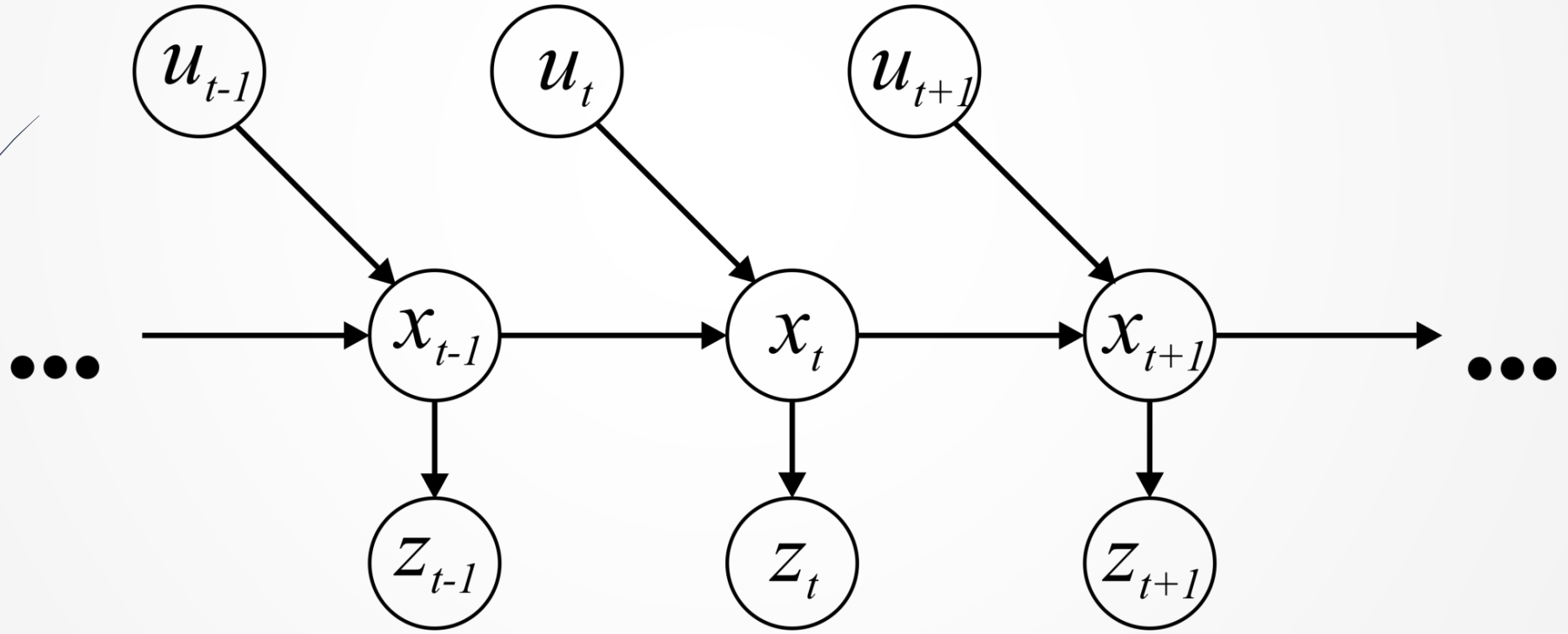
- Intersection of two Gaussians – remains Gaussian

$$\mathbf{X}_1 \sim \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1), \mathbf{X}_2 \sim \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$$

$$p(\mathbf{X}_1)p(\mathbf{X}_2) = \mathcal{N}\left(\frac{\boldsymbol{\Sigma}_2}{\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2}\boldsymbol{\mu}_1 + \frac{\boldsymbol{\Sigma}_1}{\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2}\boldsymbol{\mu}_2, \frac{1}{\boldsymbol{\Sigma}_1^{-1} + \boldsymbol{\Sigma}_2^{-1}}\right)$$

Linear Process Model

- Consider a time-discrete stochastic process (Markov chain)



Linear Process Model

- Consider a time-discrete stochastic process
- Represent the estimated state (belief) with a Gaussian

$$\mathbf{x}_t \sim \mathcal{N}(\mu_t, \Sigma_t)$$

Linear Process Model

- Consider a time-discrete stochastic process
- Represent the estimated state (belief) with a Gaussian

$$\mathbf{x}_t \sim \mathcal{N}(\mu_t, \Sigma_t)$$

- Assume that the system evolves linearly over time, then

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1}$$

Linear Process Model

- Consider a time-discrete stochastic process
- Represent the estimated state (belief) with a Gaussian

$$\mathbf{x}_t \sim \mathcal{N}(\mu_t, \Sigma_t)$$

- Assume that the system evolves linearly over time, then depends linearly on the controls

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_t$$

Linear Process Model

- Consider a time-discrete stochastic process
- Represent the estimated state (belief) with a Gaussian

$$\mathbf{x}_t \sim \mathcal{N}(\mu_t, \Sigma_t)$$

- Assume that the system evolves linearly over time, then depends linearly on the controls, and has zero-mean, normally distributed process noise

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_t + \epsilon_t$$

- With $\epsilon_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$

Linear Observations

- Further, assume we make observations that depend linearly on the state

$$\mathbf{z}_t = \mathbf{C}\mathbf{x}_t$$

Linear Observations

- Further, assume we make observations that depend linearly on the state and that are perturbed zero-mean, normally distributed observation noise

$$\mathbf{z}_t = \mathbf{C}\mathbf{x}_t + \delta_t$$

- With $\delta_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$

Kalman Filter

- Estimates the state x_t of a discrete-time controlled process that is governed by the linear stochastic difference equation

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_t + \epsilon_t$$

- And (linear) measurements of the state

$$\mathbf{z}_t = \mathbf{C}\mathbf{x}_t + \delta_t$$

- With $\epsilon_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$ and $\delta_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$

Kalman Filter

- State $\mathbf{x} \in \mathbb{R}^n$
- Controls $\mathbf{u} \in \mathbb{R}^l$
- Observations $\mathbf{z} \in \mathbb{R}^k$
- Process equation $\underset{n\text{x}n}{\mathbf{A}}\mathbf{x}_{t-1} + \underset{n\text{x}l}{\mathbf{B}}\mathbf{u}_t + \epsilon_t$
- Measurement equation $\mathbf{z}_t = \underset{n\text{x}k}{\mathbf{C}}\mathbf{x}_t + \delta_t$

Kalman Filter

- Initial belief is Gaussian

$$Bel(x_0) = \mathcal{N}(\mathbf{x}_0; \mu_0, \mathbf{\Sigma}_0)$$

- Next state is also Gaussian (linear transformation)

$$\mathbf{x}_t \sim \mathcal{N}(\mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t, \mathbf{Q})$$

- Observations are also Gaussian

$$\mathbf{z}_t \sim \mathcal{N}(\mathbf{C}\mathbf{x}_t, \mathbf{R})$$

Recall: Bayes Filter Algorithm

- For each step, do:
  - Apply motion model

$$\overline{Bel}(\mathbf{x}_t) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) Bel(\mathbf{x}_{t-1}) d\mathbf{x}_{t-1}$$

  - Apply sensor model

$$Bel(\mathbf{x}_t) = \eta p(\mathbf{z}_t|\mathbf{x}_t) \overline{Bel}(\mathbf{x}_t)$$

From Bayes Filter to Kalman Filter

- For each step, do:
  - Apply motion model

$$\overline{Bel}(\mathbf{x}_t) = \int \underbrace{p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)}_{\mathcal{N}(\mathbf{x}_t;\mathbf{A}\mathbf{x}_{t-1}+\mathbf{B}\mathbf{u}_k t,\mathbf{Q})} \underbrace{Bel(\mathbf{x}_{t-1})}_{\mathcal{N}(\mathbf{x}_{t-1};\mu_{t-1},\mathbf{\Sigma}_{t-1})} d\mathbf{x}_{t-1}$$

From Bayes Filter to Kalman Filter

- For each step, do:
  - Apply motion model

$$\overline{Bel}(\mathbf{x}_t) = \int \underbrace{p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)}_{\mathcal{N}(\mathbf{x}_t;\mathbf{A}\mathbf{x}_{t-1}+\mathbf{B}\mathbf{u}_k t,\mathbf{Q})} \underbrace{Bel(\mathbf{x}_{t-1})}_{\mathcal{N}(\mathbf{x}_{t-1};\mu_{t-1},\mathbf{\Sigma}_{t-1})} d\mathbf{x}_{t-1}$$

$$= \mathcal{N}(\mathbf{x}_t; \mathbf{A}\mu_{t-1} + \mathbf{B}\mathbf{u}_t, \mathbf{A}\mathbf{\Sigma}\mathbf{A}^T + \mathbf{Q})$$

$$= \mathcal{N}(\mathbf{x}_t; \bar{\mu}_t, \bar{\Sigma}_t)$$

From Bayes Filter to Kalman Filter

- For each step, do:
  - Apply sensor model

$$\overline{Bel}(\mathbf{x}_t) = \eta \underbrace{p(\mathbf{z}_t|\mathbf{x}_t)}_{\mathcal{N}(\mathbf{z}_t;\mathbf{C}\mathbf{x}_t,\mathbf{R})} \underbrace{\overline{Bel}(\mathbf{x}_t)}_{\mathcal{N}(\mathbf{x}_t;\bar{\mu}_t,\bar{\mathbf{\Sigma}}_t)}$$

$$= \mathcal{N}(\mathbf{x}_t; \bar{\mu}_t + \mathbf{K}_t(\mathbf{z}_t - \mathbf{C}\bar{\mu}), (\mathbf{I} - \mathbf{K}_t)\mathbf{C})\bar{\mathbf{\Sigma}})$$

$$= \mathcal{N}(x_t; \mu_t, \mathbf{\Sigma}_t)$$

  - With $\mathbf{K}_t = \bar{\mathbf{\Sigma}}_t\mathbf{C}^T(\mathbf{C}\bar{\mathbf{\Sigma}}_t\mathbf{C}^T + \mathbf{R})^{-1}$ (**Kalman Gain**)

From Bayes Filter to Kalman Filter

Blends between our previous estimate $\bar{\mu}_t$ and the discrepancy between our sensor observations and our predictions.

The degree to which we believe in our sensor observations is the Kalman Gain. And this depends on a formula based on the errors of sensing etc. In fact it depends on the ratio between our uncertainty Σ and the uncertainty of our sensor observations R.

$$\bar{\mu}_t + \mathbf{K}_t(\mathbf{z}_t - \mathbf{C}\bar{\mu})$$

old mean

Kalman Gain

From Bayes Filter to Kalman Filter

- For each step, do:
  - Apply sensor model

$$\overline{Bel}(\mathbf{x}_t) = \eta \underbrace{p(\mathbf{z}_t|\mathbf{x}_t)}_{\mathcal{N}(\mathbf{z}_t;\mathbf{C}\mathbf{x}_t,\mathbf{R})} \underbrace{\overline{Bel}(\mathbf{x}_t)}_{\mathcal{N}(\mathbf{x}_t;\bar{\mu}_t,\bar{\mathbf{\Sigma}}_t)}$$

$$= \mathcal{N}(\mathbf{x}_t; \bar{\mu}_t + \mathbf{K}_t(\mathbf{z}_t - \mathbf{C}\bar{\mu}), (\mathbf{I} - \mathbf{K}_t)\mathbf{C})\bar{\mathbf{\Sigma}})$$

$$= \mathcal{N}(x_t; \mu_t, \mathbf{\Sigma}_t)$$

  - With $\mathbf{K}_t = \bar{\mathbf{\Sigma}}_t\mathbf{C}^T(\mathbf{C}\bar{\mathbf{\Sigma}}_t\mathbf{C}^T + \mathbf{R})^{-1}$ (**Kalman Gain**)

Kalman Filter Algorithm

- For each step, do:
  - Apply motion model (prediction step)

$$\bar{\boldsymbol{\mu}}_t = \mathbf{A}\boldsymbol{\mu}_{t-1} + \mathbf{B}\mathbf{u}_t$$

$$\bar{\boldsymbol{\Sigma}}_t = \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^\top + \mathbf{Q}$$

  - Apply sensor model (correction step)

$$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + \mathbf{K}_t(\mathbf{z}_t - \mathbf{C}\bar{\boldsymbol{\mu}}_t)$$

$$\boldsymbol{\Sigma}_t = (\mathbf{I} - \mathbf{K}_t\mathbf{C})\bar{\boldsymbol{\Sigma}}_t$$

  - With $\mathbf{K}_t = \bar{\boldsymbol{\Sigma}}_t\mathbf{C}^\top(\mathbf{C}\bar{\boldsymbol{\Sigma}}_t\mathbf{C}^\top + \mathbf{R})^{-1}$

Kalman Filter Algorithm

Prediction & Correction steps can happen in any order.

- For each step, do:
  - Apply motion model (**prediction step**)

$$\bar{\boldsymbol{\mu}}_t = \mathbf{A}\boldsymbol{\mu}_{t-1} + \mathbf{B}\mathbf{u}_t$$

$$\bar{\boldsymbol{\Sigma}}_t = \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^\top + \mathbf{Q}$$

  - Apply sensor model (**correction step**)

$$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + \mathbf{K}_t(\mathbf{z}_t - \mathbf{C}\bar{\boldsymbol{\mu}}_t)$$

$$\boldsymbol{\Sigma}_t = (\mathbf{I} - \mathbf{K}_t\mathbf{C})\bar{\boldsymbol{\Sigma}}_t$$

  - With $\mathbf{K}_t = \bar{\boldsymbol{\Sigma}}_t\mathbf{C}^\top(\mathbf{C}\bar{\boldsymbol{\Sigma}}_t\mathbf{C}^\top + \mathbf{R})^{-1}$

Kalman Filter Algorithm

Prediction & Correction steps can happen in any order.

Prediction

$$\bar{\mu}_t = \mathbf{A}\mu_{t-1} + \mathbf{B}\mathbf{u}_t$$
$$\bar{\Sigma}_t = \mathbf{A}\Sigma\mathbf{A}^\top + \mathbf{Q}$$

Correction

$$\mu_t = \bar{\mu}_t + \mathbf{K}_t(\mathbf{z}_t - \mathbf{C}\bar{\mu}_t)$$
$$\Sigma_t = (\mathbf{I} - \mathbf{K}_t\mathbf{C})\bar{\Sigma}_t$$
$$\mathbf{K}_t = \bar{\Sigma}_t\mathbf{C}^\top(\mathbf{C}\bar{\Sigma}_t\mathbf{C}^\top + \mathbf{R})^{-1}$$

Complexity

- Highly efficient: Polynomial in the measurement dimensionality k and state dimensionality n

$$O(k^{2.376} + n^2)$$

- Optimal for linear Gaussian systems
  - But most robots are nonlinear! This is why in practice we use Extended Kalman Filters and other approaches.

LAB mini courses

Estimation - 01 – Kalman Filter

Subsection 4: Kalman Filter – State Space Derivation

KF State Space Derivation

- Assume that we want to know the value of a variable within a **process of the form**:

$$x_{k+1} = \Phi x_k + w_k$$ Eq.1

  - x_k: the state vector of the process at time k (nx1)
  - Φ: is the state transition matrix of the process from the state k to k+1 – assumed stationary over time – (nxm)
  - w_k: associated white noise process with known covariance (nx1)
- **Observation** on this variable can be modelled in the form:

$$z_k = H x_k + v_k$$ Eq.2

  - z_k: actual measurement of x at time k (mx1)
  - H: noiseless connection between the state vector and the measurement vector – assumed stationary over time – (mxn)
  - v_k: white noise process with known covariance and zero cross correlation with w_k (mx1)

KF State Space Derivation

- For the minimization of the Mean Square Error (MSE) to yield the optimal filter, it must be possible to correctly model the system errors using Gaussian distributions. The **covariances** of the two noise models are assumed stationary over time and given by:

Covariances Formula

$$Q = E\left[w_k w_k^T\right] \quad \text{Eq.3}$$

$$R = E\left[v_k v_k^T\right] \quad \text{Eq.4}$$

- The mean squared error formulation gives:

Error Covariances Formula Eq.5

$$E\left[e_k e_k^T\right] = P_k \rightarrow P_k = E\left[e_k e_k^T\right] = E\left[\left(x_k - \hat{x}_k\right)\left(x_k - \hat{x}_k\right)^T\right]$$

  - P_k: is the error covariance matrix at time k (nxn)

- Let the prior estimate of $\hat{x}_k$ be $\hat{x}'_k$ (gained by knowledge of the system). We can write an **update equation** for the new estimate, combining the old estimate with measurement data:

$$\hat{x}_k = \hat{x}'_k + K_k\left(z_k - H\hat{x}'_k\right) \quad \text{Eq.6}$$

  - K_k: the Kalman Gain
  - $i_k = z_k - H\hat{x}_k$ is the *innovation* or *measurement residual* Eq.7

KF State Space Derivation

- Through substitution of Eq.2 to Eq.6:

$$\hat{x}_k = \hat{x}'_k + K_k\left(Hx_k + v_k - H\hat{x}'_k\right) \qquad \text{Eq.8}$$

- Through substitution of Eq.8 to Eq.5:

$$P_k = E\left[\left[\left(I - K_kH\right)\left(x_k - \hat{x}'_k\right) - K_kv_k\right]\left[\left(I - K_kH\right)\left(x_k - \hat{x}'_k\right) - K_kv_k\right]^T\right] \qquad \text{Eq.9}$$

- The difference $\hat{x}_k - \hat{x}'_k$ is the **error of the prior estimate**. As this is uncorrelated with the measurement noise, the **expectation may be rewritten as**:

$$P_k = \left(I - K_kH\right)E\left[\left(x_k - \hat{x}'_k\right)\left(x_k - \hat{x}'_k\right)^T\right]\left(I - K_kH\right) + K_kE\left[v_kv_k^T\right]K_k^T \qquad \text{Eq.10}$$

- Through substitution of Eq.4 and Eq.5 to Eq.9:

Error Covariances Update

$$P_k = \left(I - K_kH\right)P'_k\left(I - K_kH\right)^T + K_kRK_k^T \qquad \text{Eq.11}$$

  - P'_k is the prior estimate of P_k

KF State Space Derivation

- Eq.11 is the error covariance update equation. The diagonal of the covariance matrix contains the mean squared errors:

$$P_{kk} = \begin{bmatrix} E\left[e_{k-1}e_{k-1}^T\right] & E\left[e_k e_{k-1}^T\right] & E\left[e_{k+1}e_{k-1}^T\right] \\ E\left[e_{k-1}e_k^T\right] & E\left[e_k e_k^T\right] & E\left[e_{k+1}e_k^T\right] \\ E\left[e_{k-1}e_{k+1}^T\right] & E\left[e_k e_{k+1}^T\right] & E\left[e_{k+1}e_{k+1}^T\right] \end{bmatrix} \quad \text{Eq.12}$$

  - The sum of the diagonal elements of a matrix is its *trace*. **In the case of the error covariance matrix, the trace is the sum of the mean squared errors.** Therefore, **the mean squared error may be minimized by minimizing the trace of P_k which in turn will minimize the trace of P_{kk}.**
  - The trace of P_k is first differentiated with respect to K_k and the result set to zero in order to find the conditions of this minimum.
  - Expansion of Eq.11 gives: $P_k = P_k' - K_k H P_k' - P_k' H^T K_k^T + K_k \left(H P_k' H^T + R\right) K_k^T$ Eq.13
    - As the trace of a matrix is equal to the trace of its transpose:

$$T\left[P_k\right] = T\left[P_k'\right] - 2T\left[K_k H P_k'\right] + T\left[K_k \left(H P_k' H^T + R\right) K_k^T\right] \quad \text{Eq.14}$$

    - $T[P_k]$ is the trace of the matrix P_k
  - Differentiating with respect to K_k gives:

$$\frac{dT\left[P_k\right]}{dK_k} = -2(H P_k')^T + 2K_k \left(H P_k' H^T + R\right) \quad \text{Eq.15}$$

KF State Space Derivation

- Setting Eq.14 to zero and re-arranging gives:

$$(HP'_k)^T = K_k \left(HP'_k H^T + R\right)$$ Eq.16

- And solving for K_k:

$$K_k = P'_k H^T \left(HP'_k H^T + R\right)^{-1}$$ **Kalman Gain** Eq.17

- Which is the **Kalman Gain Equation**
- The innovation i_k (Eq.7) has an associated measurement prediction covariance:

$$S_k = HP'_k H^T + R$$ Eq.18

- Finally, through substitution of Eq.17 into Eq.13:

Error covariance matrix update with optimal gain

$$\begin{aligned} P_k &= P'_k - P'_k H^T \left(HP'_k H^T + R\right)^{-1} HP'_k \\ &= P'_k - K_k HP'_k \\ &= (I - K_k H) P'_k \end{aligned}$$ Eq.19

  - Eq. 19 is the update equation for the error covariance matrix with optimal gain. The three equations Eq.6, Eq.17 and Eq.19 develop an estimate for x_k. State project is achieved using:

$$\hat{x}'_{k+1} = \Phi \hat{x}_k$$ Eq.20

KF State Space Derivation

- To complete the recursion it is necessary to find an equation which **projects the error covariance matrix into the next time interval**, k+1. This is achieved by first forming an expression for the prior error:

$$\begin{aligned} e'_{k+1} &= x_{k+1} - \hat{x}'_{k+1} \\ &= (\Phi x_k + w_k) - \Phi \hat{x}_k \\ &= \Phi e_k + w_k \end{aligned}$$

Eq.21

- Extending Eq.5 to k+1:

$$P'_{k+1} = E\left[e'_{k+1} e'^{T}_{k+1}\right] = E\left[(\Phi e_k + w_k)(\Phi e_k + w_k)^T\right]$$

Eq.22

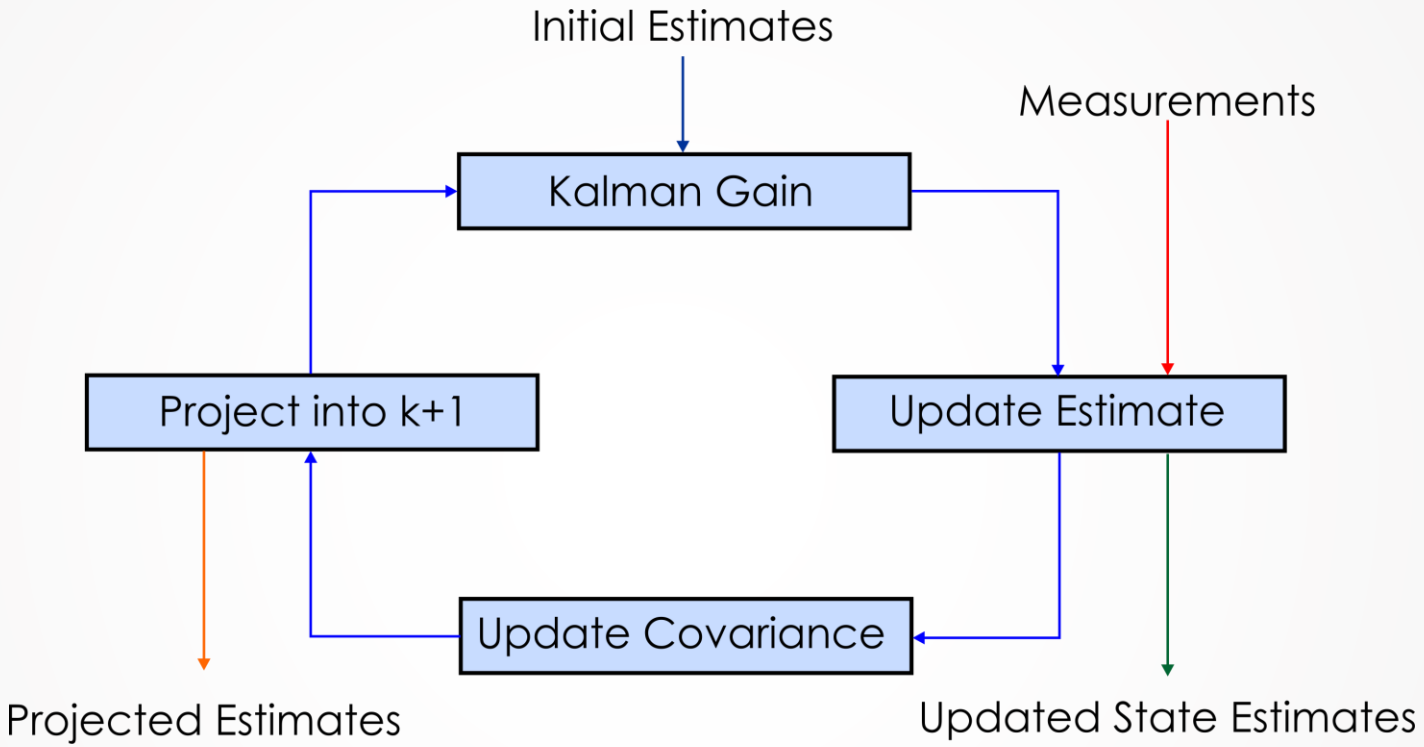
- Note that e_k and w_k have zero cross-correlation because the noise w_k accumulates between k and k+1, whereas the error e_k is the error up until time k. Thus:

$$\begin{aligned} P'_{k+1} &= E\left[e'_{k+1} e'^{T}_{k+1}\right] \\ &= E\left[\Phi e_k (\Phi e_k)^T\right] + E\left[w_k w_k^T\right] \\ &= \Phi P_k \Phi^T + Q \end{aligned}$$

Eq.23

- This completes the recursive filter.

KF State Space Derivation



Description	Equation
Kalman Gain	$K_k = P'_k(HP'_kH^T + R)^-1$
Update/Correct Estimate	$\hat{x}_k = \hat{x}'_k + K_k(z_k - H\hat{x}'_k)$
Update/Correct Covariance	$P_k = (I - K_kH)P'_k$
Project/Predict into k+1	$\hat{x}'_{k+1} = \Phi\hat{x}_k$ $P_{k+1} = \Phi P_k\Phi^T + Q$

Model Covariance Update

- The model parameter covariance has been considered in the its inverted form where it is known as the information matrix. It is possible to formulate an alternative update equation for the covariance matrix using standard error propagation:

$$P_k^{-1} = P_k'^{-1} + HR^{-1}H^T$$ Eq.22

- It can be shown that the covariance updates of Eq.22 and Eq.19 are equivalent. This may be achived by using the identity $P_k \times P_k^{-1} = I$. The original, Eq.19 and alternative Eq.22 forms of the covariance update equations are:

$$P_k = (I - K_k H) P_k' \text{ and } P_k^{-1} = P_k'^{-1} + HR^{-1}H^T$$ Eq.23

- Therefore: $(I - K_k H) P_k' \times P_k'^{-1} + HR^{-1}H^T = I$ Eq.24

- Substitution for K_k gives:

$$\begin{aligned}
&\left[P_k' - P_k'H^T\left(HP_k'H^T + R\right)^{-1}HP_k'\right]\left[P_k'^{-1} + H^TR^{-1}H\right] \\
&= I - P_k'H^T\left[\left(HP_k'H^T + R\right)^{-1}\right. \\
&\left. - R^{-1} + \left(HP_k'H^T + R\right)^{-1}HP_k'H^TR^{-1}\right]H \\
&= I - P_k'H^T\left[\left(HP_k'H^T + R\right)^{-1}\left(I + HP_k'H^TR^{-1}\right) - R^{-1}\right]H \\
&= I - P_k'H^T\left[R^{-1} - R^{-1}\right]H \\
&= I
\end{aligned}$$ Eq.25

LAB mini courses

Estimation - 01 – Kalman Filter

Subsection 4: A Python programming-oriented introduction

Python tools for probability computations

- Mean

```
import numpy as np
x = [1.85, 2.0, 1.7, 1.9, 1.6]
print(np.mean(x))
```

- Median

```
print(np.median(x))
```

- Variance

```
print(np.var(X), "meters squared")
```

- Standard deviation and Variance

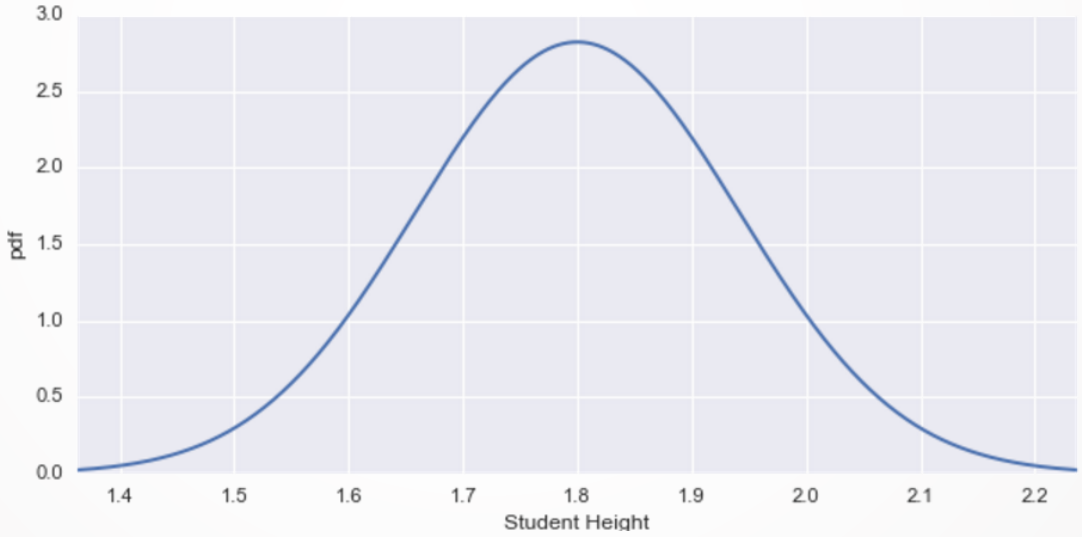
```
print('std {:.4f}'.format(np.std(X)))
print('var {:.4f}'.format(np.std(X)**2))
```

Python tools for probability computations

- Gaussians

```
from filterpy.stats import plot_gaussian_pdf
plt.figure()
ax = plot_gaussian_pdf(mean=1.8, variance=0.1414**2,
                       xlabel='Student Height', ylabel='pdf')
```

Probability Density Function



Python tools for probability computations

- Gaussian Distributions
  - A Gaussian is a Probability Density Function defined by its mean parameter (μ) and its variance (σ2) as follows:

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(x-\mu)^2}{2\sigma^2})$$

```
%load -s gaussian stats.py

def gaussian(x, mean, var):
    """returns normal distribution for x given a
    gaussian with the specified mean and variance.
    """
    return (np.exp((-0.5*(np.asarray(x)-mean)**2)/var) /
            math.sqrt(2*math.pi*var))
```

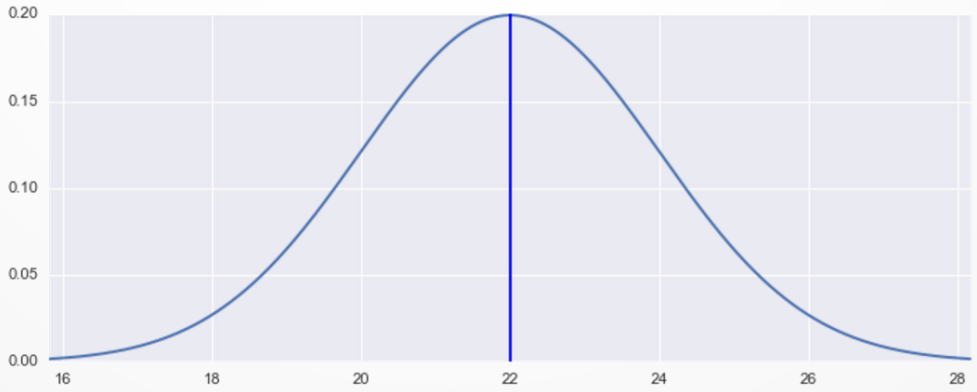
```
from filterpy.stats import gaussian, norm_cdf

with interactive_plot():
    ax = plot_gaussian_pdf(22, 4, mean_line=True, xlabel='$^{\circ}C$')
```

Python tools for probability computations

- Gaussian Distributions

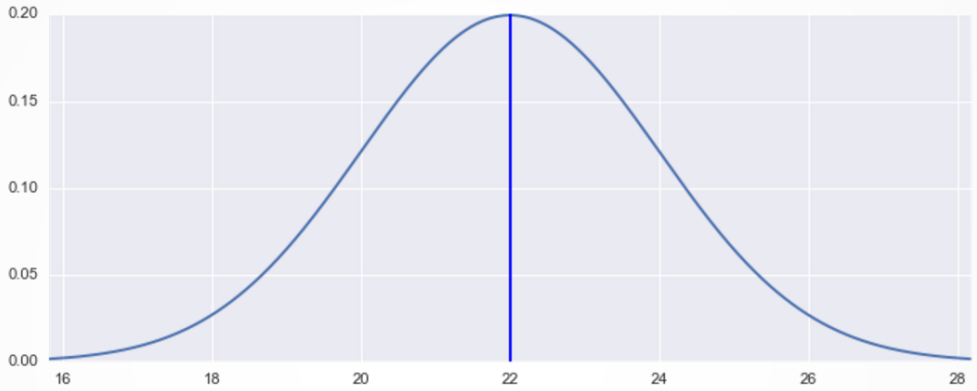
```
%load -s gaussian stats.py

def gaussian(x, mean, var):
    """returns normal distribution for x given a
    gaussian with the specified mean and variance.
    """
    return (np.exp((-0.5*(np.asarray(x)-mean)**2)/var) /
            math.sqrt(2*math.pi*var))
```

```
from filterpy.stats import gaussian, norm_cdf

with interactive_plot():
    ax = plot_gaussian_pdf(22, 4, mean_line=True, xlabel='$^{\circ}C$')
```



Python tools for probability computations

- Gaussian Distributions



- Calculation of the area under the curve (cumulative distribution function)

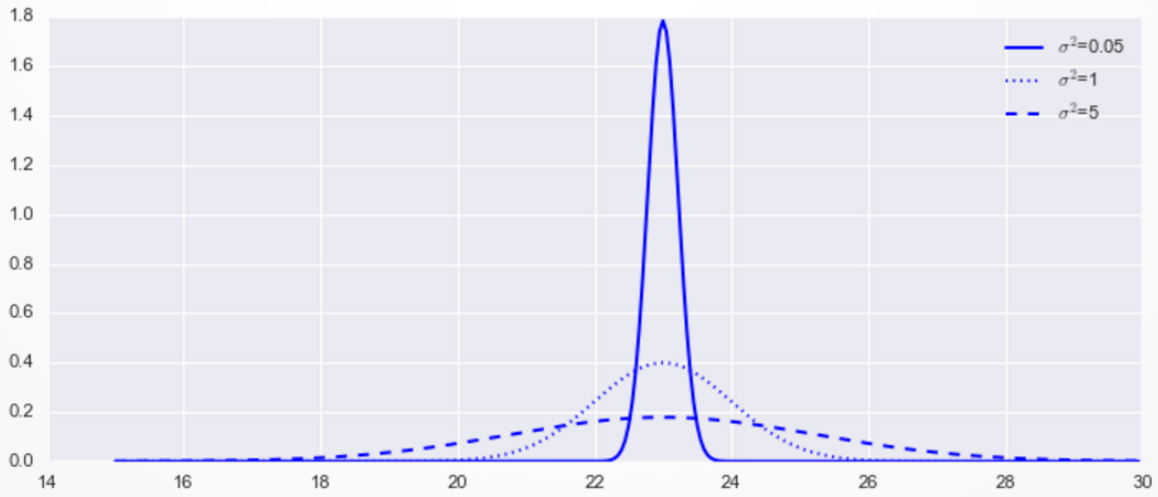
$$\int_{x_0}^{x_1} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) dx$$

```
print('Probability of range 21.5 to 22.5 is {:.2f}%'.format(
      norm_cdf((21.5, 22.5), 22,4)*100))
print('Probability of range 23.5 to 24.5 is {:.2f}%'.format(
      norm_cdf((23.5, 24.5), 22,4)*100))
```

Python tools for probability computations

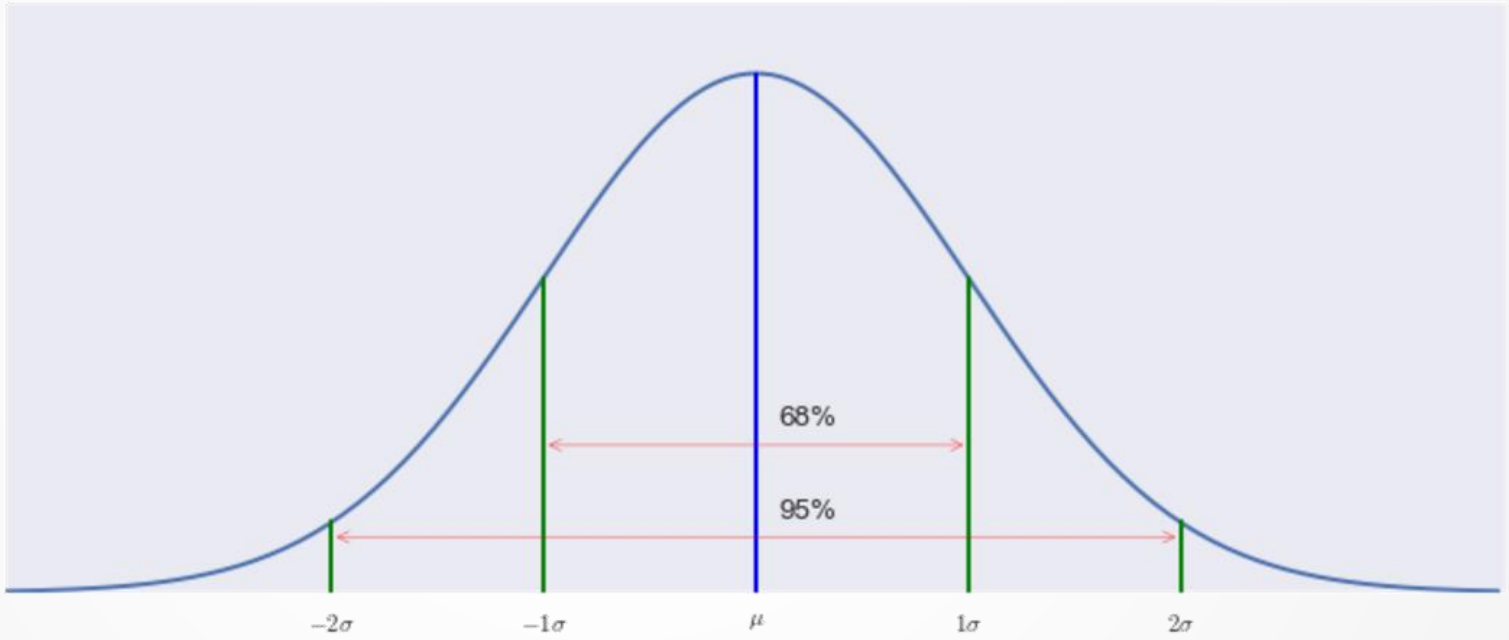
- The Variance and Belief
  - Small standard deviation indicates better confidence.

```
import numpy as np
import matplotlib.pyplot as plt

xs = np.arange(15, 30, 0.05)
with interactive_plot():
    plt.plot(xs, gaussian(xs, 23, 0.05), label='$\sigma^2$=0.05', c='b')
    plt.plot(xs, gaussian(xs, 23, 1), label='$\sigma^2$=1', ls=':', c='b')
    plt.plot(xs, gaussian(xs, 23, 5), label='$\sigma^2$=5', ls='--', c='b')
    plt.legend()
```



Python tools for probability computations

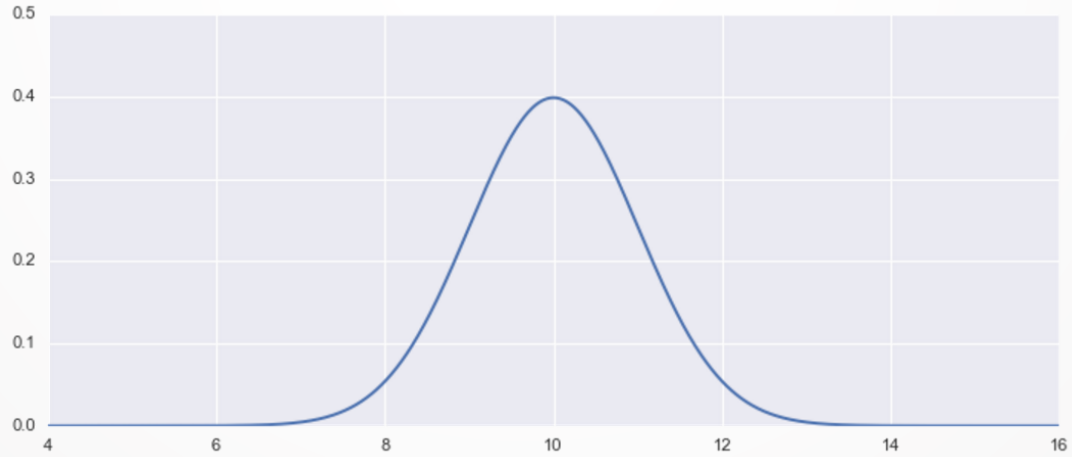
- The 3σ rule
  - Within 3σ around the mean, 99.7% of all the cumulative probability is covered.



Designing a simple Kalman Filter in Python

- Assume we track an object over an axis.
  - Consider that we now believe it is at 10m with a significant uncertainty.

```
from book_format import set_figsize, figsize
from code.book_plots import interactive_plot
import matplotlib.pyplot as plt

import filterpy.stats as stats
with interactive_plot():
    stats.plot_gaussian_pdf(mean=10, variance=1,
                            xlim=(4, 16), ylim=(0, .5))
```

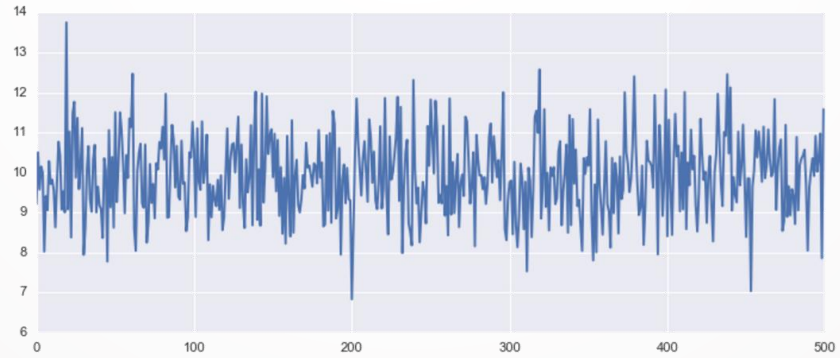


  - Can we get many measurements to robustify our estimate about the position of the robot?

Designing a simple Kalman Filter in Python

- So let's get many measurements…

```
import code.book_plots as bp
import numpy as np
from numpy.random import randn

xs = range(500)
ys = randn(500)*1 + 10.
with interactive_plot():
    #_ = bp.plot_track(xs, ys)
    plt.plot(xs, ys)
print('Mean of readings is {:.3f}'.format(np.mean(ys)))
```



- Getting these measurements as a batch and computing the mean indicates that the position is 10. Can we do that better, online, real-time, recursively?

Designing a simple Kalman Filter in Python

- **Tracking with Gaussian Probabilities**
  - To represent where the object is considered to be per measurement we write the Gaussian:

$$x = \mathcal{N}(\mu, \sigma^2)$$

  - And we can use every new step of measurement update to conduct a prediction and an update step:

$$\bar{x}_{\mathcal{N}} = x_{\mathcal{N}} \bigoplus f_{x_{\mathcal{N}}}(\cdot)$$

Predict

$$x_{\mathcal{N}} = L \bigotimes \bar{x}_{\mathcal{N}}$$

Update

  - where $\bigoplus$ $\bigotimes$ are "unknown" (for now) operators that can perform the predict and update steps.

Designing a simple Kalman Filter in Python

- **Tracking with Gaussian Probabilities: Prediction**
  - We need to represent the change in motion as a Gaussian. Therefore, let f_x be the "change in the motion of the object". So the new position predicted will be:

$$\bar{x} = x + f_x$$

  - Which is suitable because the sum of two Gaussians, is also Gaussian with parameters:

$$\mu = \mu_1 + \mu_2$$
$$\sigma^2 = \sigma_1^2 + \sigma_2^2$$

  - This allows us to write the update step as:

```
def predict(pos, movement):
    return (pos[0] + movement[0], pos[1] + movement[1])
```

Designing a simple Kalman Filter in Python

- **Tracking with Gaussian Probabilities: Update**
  - The update step will have to work on the current prior and the likelihood to derive the next estimate. This can be written as:

$$x = ||\mathcal{L}\bar{x}||$$

  - The prior is represented as a Gaussian. At the same time, the likelihood is the probability of the measurement given the current state – which is also a Gaussian. Multiplication of two Gaussians is a more complex operation taking the form:

$$\begin{aligned}\mathcal{N}(\mu, \sigma^2) &= ||prior \cdot likelihood|| \\ &= \bar{\mu}, \bar{\sigma}^{\in} \cdot \mathcal{N}(\mu_z, \sigma_z^2) \\ &= \mathcal{N}(\frac{\bar{\sigma}^2\mu_z + \sigma_z^2\bar{\mu}}{\bar{\sigma}^2 + \sigma_z^2}, \frac{\bar{\sigma}^2\sigma_z^2}{\bar{\sigma}^2 + \sigma_z^2})\end{aligned}$$

Designing a simple Kalman Filter in Python

- **Tracking with Gaussian Probabilities: Update**

$$\begin{aligned}\mathcal{N}(\mu, \sigma^2) &= ||prior \cdot likelihood|| \\ &= \bar{\mu}, \bar{\sigma}^{\in} \cdot \mathcal{N}(\mu_z, \sigma_z^2) \\ &= \mathcal{N}(\frac{\bar{\sigma}^2\mu_z + \sigma_z^2\bar{\mu}}{\bar{\sigma}^2 + \sigma_z^2}, \frac{\bar{\sigma}^2\sigma_z^2}{\bar{\sigma}^2 + \sigma_z^2})\end{aligned}$$

- Which then is coded as:

```
def gaussian_multiply(g1, g2):
    mu1, var1 = g1
    mu2, var2 = g2
    mean = (var1*mu2 + var2*mu1) / (var1 + var2)
    variance = (var1 * var2) / (var1 + var2)
    return (mean, variance)


def update(prior, likelihood):
    posterior = gaussian_multiply(likelihood, prior)
    return posterior
```

Designing a simple Kalman Filter in Python

Multiplying Gaussians

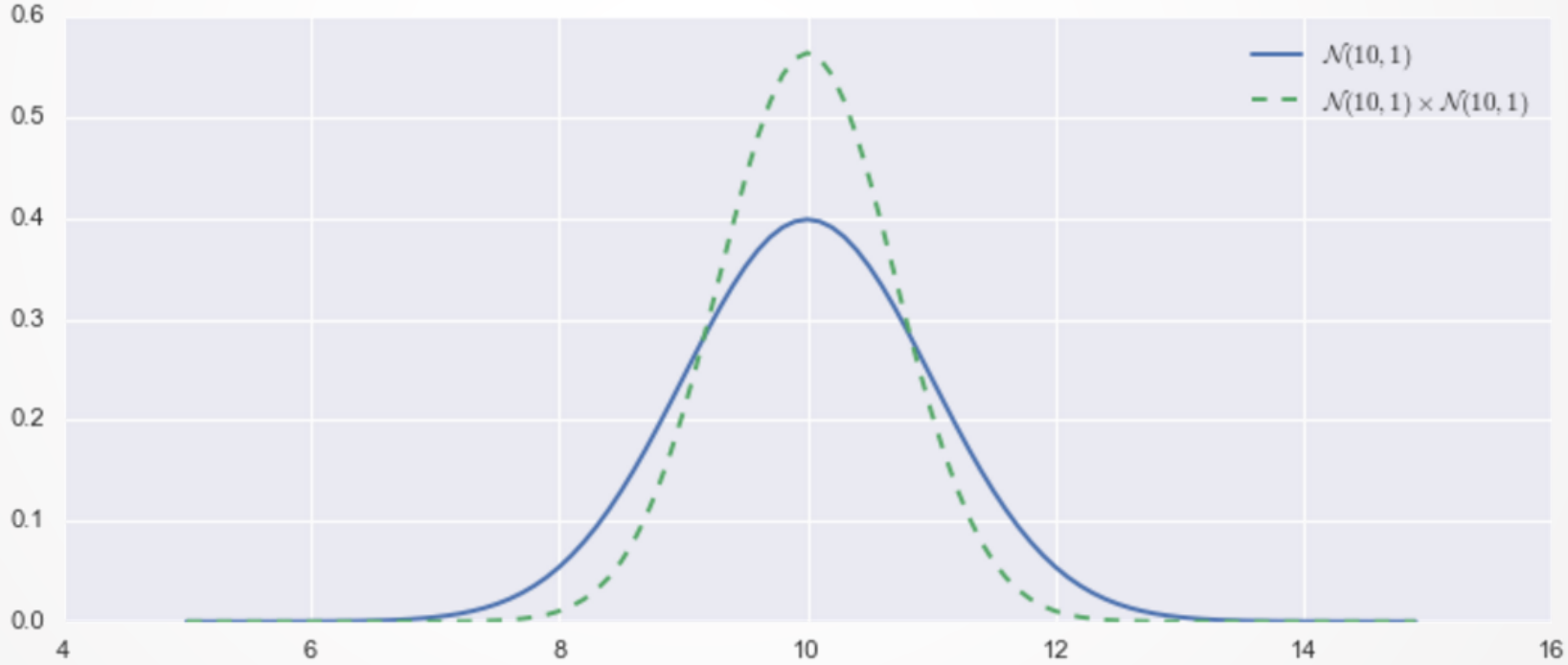
- Gaussian multiplication affects both the mean (now a function of the previous mean values and the variances) as well as the variances (now a function of the previous variances).

```
import filterpy.stats as stats
import matplotlib.pyplot as plt

z = (10., 1.)  # Gaussian N(10, 1)


with interactive_plot():
    product = gaussian_multiply(z, z)

    xs = np.arange(5, 15, 0.1)
    ys = [stats.gaussian(x, z[0], z[1]) for x in xs]
    plt.plot(xs, ys, label='$\mathcal{N}(10,1)$')

    ys = [stats.gaussian(x, product[0], product[1]) for x in xs]
    plt.plot(xs, ys, label='$\mathcal{N}(10,1) \\times \mathcal{N}(10,1)$', ls='--')
    plt.legend()
print(product)
```

Designing a simple Kalman Filter in Python

Multiplying Gaussians

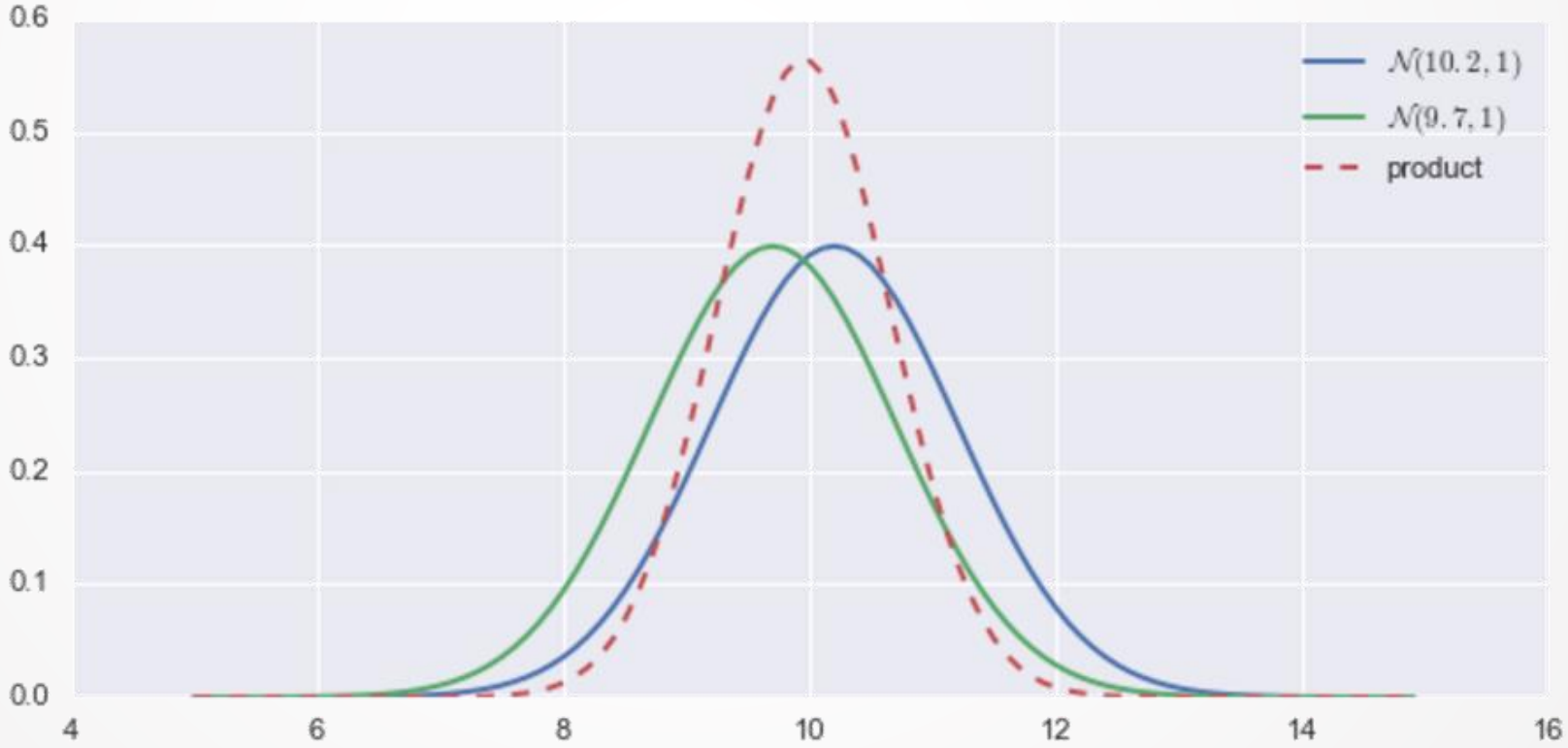
- Gaussian multiplication affects both the mean (now a function of the previous mean values and the variances) as well as the variances (now a function of the previous variances).



Designing a simple Kalman Filter in Python

Multiplying Gaussians

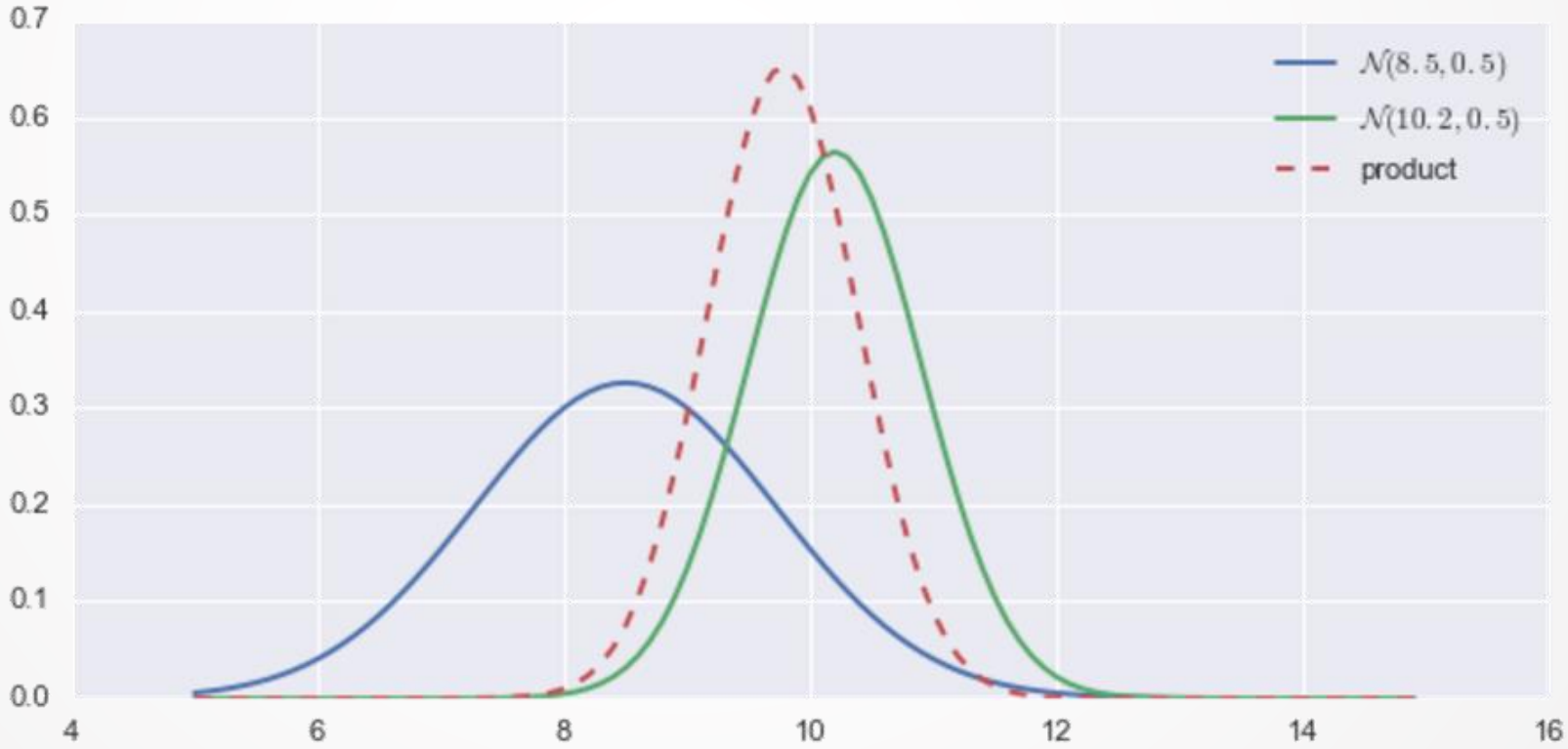
- Gaussian multiplication affects both the mean (now a function of the previous mean values and the variances) as well as the variances (now a function of the previous variances).



Designing a simple Kalman Filter in Python

Multiplying Gaussians

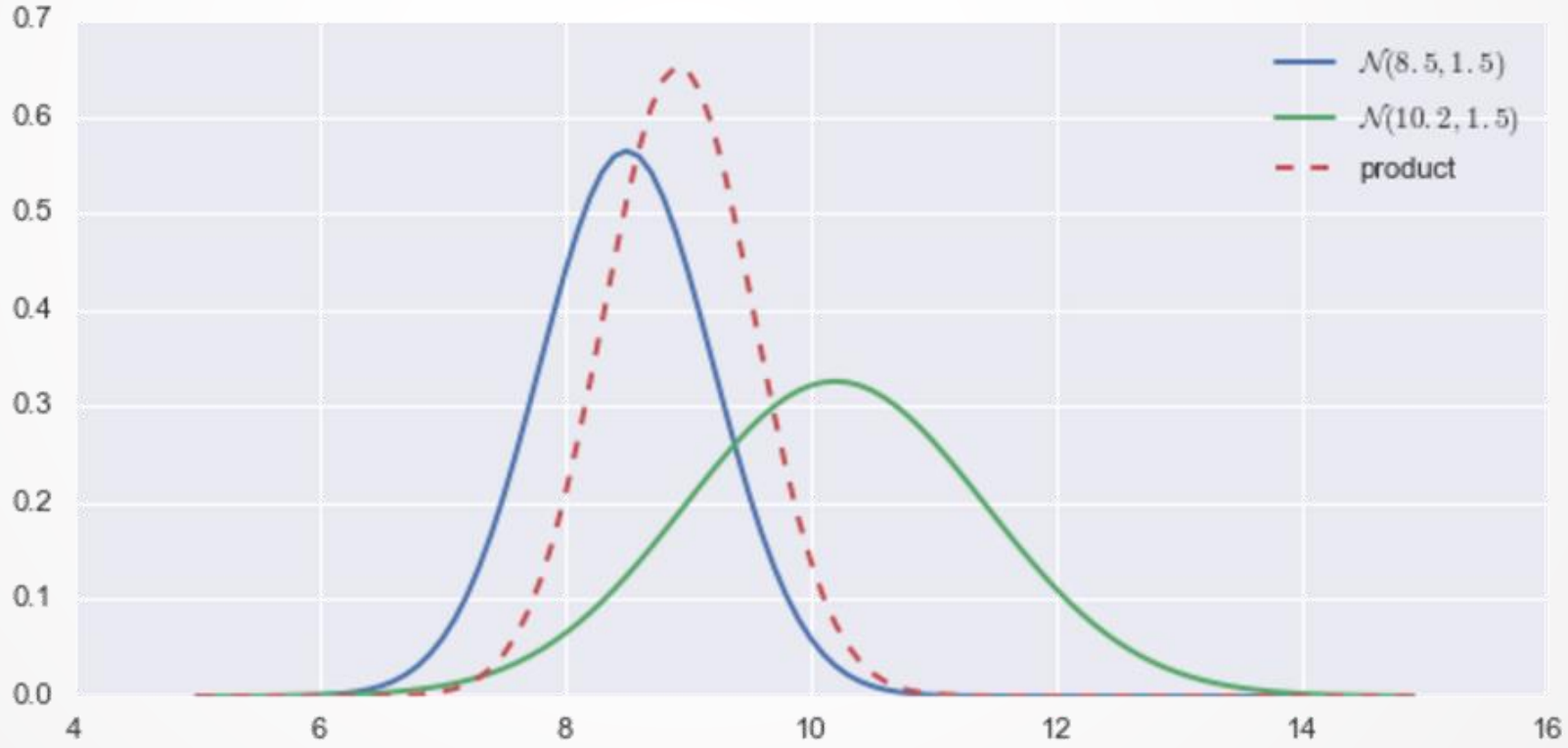
- Gaussian multiplication affects both the mean (now a function of the previous mean values and the variances) as well as the variances (now a function of the previous variances).



Designing a simple Kalman Filter in Python

Multiplying Gaussians

- Gaussian multiplication affects both the mean (now a function of the previous mean values and the variances) as well as the variances (now a function of the previous variances).



Designing a simple Kalman Filter in Python

A First Kalman Filter Implementation

```
import code.kf_internal as kf_internal
from code.kf_internal import DogSimulation

np.random.seed(13)

process_var = 1. # variance in the dog's movement
sensor_var = 2. # variance in the sensor

x = (0., 400.)  # dog's position, N(0, 400)
velocity = 1
dt = 1. # time step in seconds
process_model = (velocity*dt, process_var)

# simulate dog and get measurements
dog = DogSimulation(x[0], process_model[0],
                    sensor_var, process_model[1])
zs = [dog.move_and_sense() for _ in range(10)]

print('PREDICT\t\t\tUPDATE')
print('     x      var\t\t  z\t    x      var')

# run the filter
xs, predictions = [], []
for z in zs:
    prior = predict(x, process_model)
    #print(prior)
    likelihood = (z, sensor_var)
    x = update(prior, likelihood)

    # save results
    predictions.append(prior[0])
    xs.append(x[0])
    kf_internal.print_gh(prior, x, z)
print()
print('final estimate:        {:10.3f}'.format(x[0]))
print('actual final position: {:10.3f}'.format(dog.x))
```

```
import code.kf_internal as kf_internal
from code.kf_internal import DogSimulation

np.random.seed(13)

process_var = 1. # variance in the dog's movement
sensor_var = 2. # variance in the sensor

x = (0., 400.)  # dog's position, N(0, 400)
velocity = 1
dt = 1. # time step in seconds
process_model = (velocity*dt, process_var)

# simulate dog and get measurements
dog = DogSimulation(x[0], process_model[0],
                    sensor_var, process_model[1])
zs = [dog.move_and_sense() for _ in range(10)]

print('PREDICT\t\t\tUPDATE')
print('     x      var\t\t  z\t    x      var')

# run the filter
xs, predictions = [], []
for z in zs:
    prior = predict(x, process_model)
    #print(prior)
    likelihood = (z, sensor_var)
    x = update(prior, likelihood)

    # save results
    predictions.append(prior[0])
    xs.append(x[0])
    kf_internal.print_gh(prior, x, z)
print()
print('final estimate:        {:10.3f}'.format(x[0]))
print('actual final position: {:10.3f}'.format(dog.x))
```

```
import code.kf_internal as kf_internal
from code.kf_internal import DogSimulation

np.random.seed(13)

process_var = 1. # variance in the dog's movement
sensor_var = 2. # variance in the sensor

x = (0., 400.)  # dog's position, N(0, 400)
velocity = 1
dt = 1. # time step in seconds
process_model = (velocity*dt, process_var)

# simulate dog and get measurements
dog = DogSimulation(x[0], process_model[0],
                    sensor_var, process_model[1])
zs = [dog.move_and_sense() for _ in range(10)]

print('PREDICT\t\t\tUPDATE')
print('     x      var\t\t  z\t    x      var')

# run the filter
xs, predictions = [], []
for z in zs:
    prior = predict(x, process_model)
    #print(prior)
    likelihood = (z, sensor_var)
    x = update(prior, likelihood)

    # save results
    predictions.append(prior[0])
    xs.append(x[0])
    kf_internal.print_gh(prior, x, z)
print()
print('final estimate:        {:10.3f}'.format(x[0]))
print('actual final position: {:10.3f}'.format(dog.x))
```

```
import code.kf_internal as kf_internal
from code.kf_internal import DogSimulation

np.random.seed(13)

process_var = 1. # variance in the dog's movement
sensor_var = 2. # variance in the sensor

x = (0., 400.)  # dog's position, N(0, 400)
velocity = 1
dt = 1. # time step in seconds
process_model = (velocity*dt, process_var)

# simulate dog and get measurements
dog = DogSimulation(x[0], process_model[0],
                    sensor_var, process_model[1])
zs = [dog.move_and_sense() for _ in range(10)]

print('PREDICT\t\t\tUPDATE')
print('     x      var\t\t  z\t    x      var')

# run the filter
xs, predictions = [], []
for z in zs:
    prior = predict(x, process_model)
    #print(prior)
    likelihood = (z, sensor_var)
    x = update(prior, likelihood)

    # save results
    predictions.append(prior[0])
    xs.append(x[0])
    kf_internal.print_gh(prior, x, z)
print()
print('final estimate:        {:10.3f}'.format(x[0]))
print('actual final position: {:10.3f}'.format(dog.x))
```

```
import code.kf_internal as kf_internal
from code.kf_internal import DogSimulation

np.random.seed(13)

process_var = 1. # variance in the dog's movement
sensor_var = 2. # variance in the sensor

x = (0., 400.)  # dog's position, N(0, 400)
velocity = 1
dt = 1. # time step in seconds
process_model = (velocity*dt, process_var)

# simulate dog and get measurements
dog = DogSimulation(x[0], process_model[0],
                    sensor_var, process_model[1])
zs = [dog.move_and_sense() for _ in range(10)]

print('PREDICT\t\t\tUPDATE')
print('     x      var\t\t  z\t    x      var')

# run the filter
xs, predictions = [], []
for z in zs:
    prior = predict(x, process_model)
    #print(prior)
    likelihood = (z, sensor_var)
    x = update(prior, likelihood)

    # save results
    predictions.append(prior[0])
    xs.append(x[0])
    kf_internal.print_gh(prior, x, z)
print()
print('final estimate:        {:10.3f}'.format(x[0]))
print('actual final position: {:10.3f}'.format(dog.x))
```

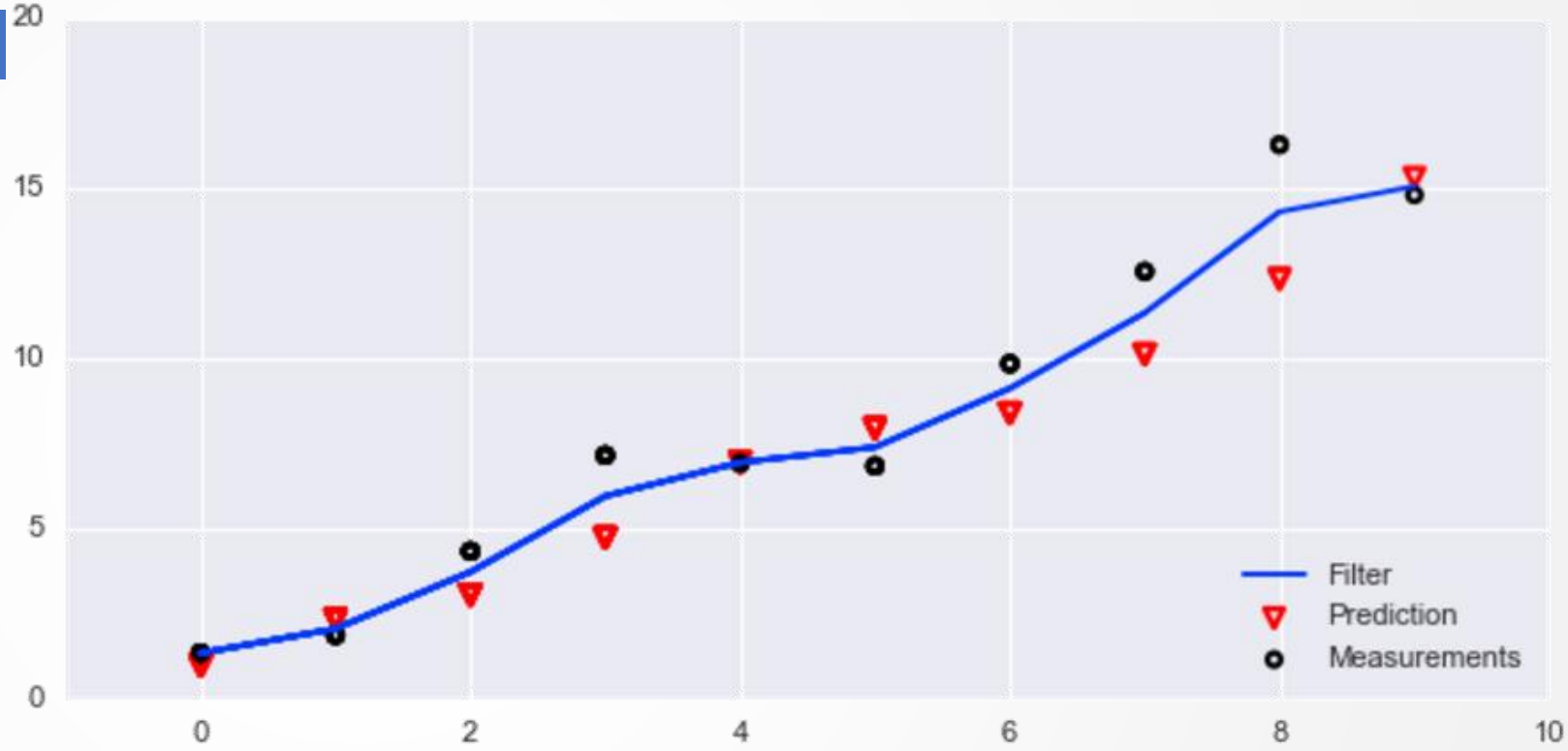
```
import code.kf_internal as kf_internal
from code.kf_internal import DogSimulation

np.random.seed(13)

process_var = 1. # variance in the dog's movement
sensor_var = 2. # variance in the sensor

x = (0., 400.)  # dog's position, N(0, 400)
velocity = 1
dt = 1. # time step in seconds
process_model = (velocity*dt, process_var)

# simulate dog and get measurements
dog = DogSimulation(x[0], process_model[0],
                    sensor_var, process_model[1])
zs = [dog.move_and_sense() for _ in range(10)]

print('PREDICT\t\t\tUPDATE')
print('     x      var\t\t  z\t    x      var')

# run the filter
xs, predictions = [], []
for z in zs:
    prior = predict(x, process_model)
    #print(prior)
    likelihood = (z, sensor_var)
    x = update(prior, likelihood)

    # save results
    predictions.append(prior[0])
    xs.append(x[0])
    kf_internal.print_gh(prior, x, z)
print()
print('final estimate:        {:10.3f}'.format(x[0]))
print('actual final position: {:10.3f}'.format(dog.x))
```

```
import code.kf_internal as kf_internal
from code.kf_internal import DogSimulation

np.random.seed(13)

process_var = 1. # variance in the dog's movement
sensor_var = 2. # variance in the sensor

x = (0., 400.)  # dog's position, N(0, 400)
velocity = 1
dt = 1. # time step in seconds
process_model = (velocity*dt, process_var)

# simulate dog and get measurements
dog = DogSimulation(x[0], process_model[0],
                    sensor_var, process_model[1])
zs = [dog.move_and_sense() for _ in range(10)]

print('PREDICT\t\t\tUPDATE')
print('     x      var\t\t  z\t    x      var')

# run the filter
xs, predictions = [], []
for z in zs:
    prior = predict(x, process_model)
    #print(prior)
    likelihood = (z, sensor_var)
    x = update(prior, likelihood)

    # save results
    predictions.append(prior[0])
    xs.append(x[0])
    kf_internal.print_gh(prior, x, z)
print()
print('final estimate:        {:10.3f}'.format(x[0]))
print('actual final position: {:10.3f}'.format(dog.x))
```

```
import code.kf_internal as kf_internal
from code.kf_internal import DogSimulation

np.random.seed(13)

process_var = 1. # variance in the dog's movement
sensor_var = 2. # variance in the sensor

x = (0., 400.)  # dog's position, N(0, 400)
velocity = 1
dt = 1. # time step in seconds
process_model = (velocity*dt, process_var)

# simulate dog and get measurements
dog = DogSimulation(x[0], process_model[0],
                    sensor_var, process_model[1])
zs = [dog.move_and_sense() for _ in range(10)]

print('PREDICT\t\t\tUPDATE')
print('     x      var\t\t  z\t    x      var')

# run the filter
xs, predictions = [], []
for z in zs:
    prior = predict(x, process_model)
    #print(prior)
    likelihood = (z, sensor_var)
    x = update(prior, likelihood)

    # save results
    predictions.append(prior[0])
    xs.append(x[0])
    kf_internal.print_gh(prior, x, z)
print()
print('final estimate:        {:10.3f}'.format(x[0]))
print('actual final position: {:10.3f}'.format(dog.x))
```

Designing a simple Kalman Filter in Python

Its results…



Designing a simple Kalman Filter in Python

The Actual Kalman Filter Steps

```
for z in zs:
    prior = predict(x, process_model)
    likelihood = (z, sensor_var)
    x = update(prior, likelihood)
```

Designing a simple Kalman Filter in Python

What is happening: The Kalman Gain

- To understand better how the filter works let's go back and remember that the posterior is computed as the likelihood times the prior.
- Therefore the mean of the posterior is: $\mu = \frac{\bar{\sigma}^2 \mu_z + \sigma_z^2 \bar{\mu}}{\bar{\sigma}^2 + \sigma_z^2}$
  - or equivalently:

$$\mu = \left(\frac{\bar{\sigma}^2}{\bar{\sigma}^2 + \sigma_z^2}\right) \mu_z + \left(\frac{\sigma_z^2}{\bar{\sigma}^2 + \sigma_z^2}\right) \bar{\mu}$$

  - As shown we are in fact scaling the measurements and the prior by probabilistic weights:

$$\mu = W_1 \mu_z + W_2 \bar{\mu}$$

  - Let us introduce the Kalman Gain as:

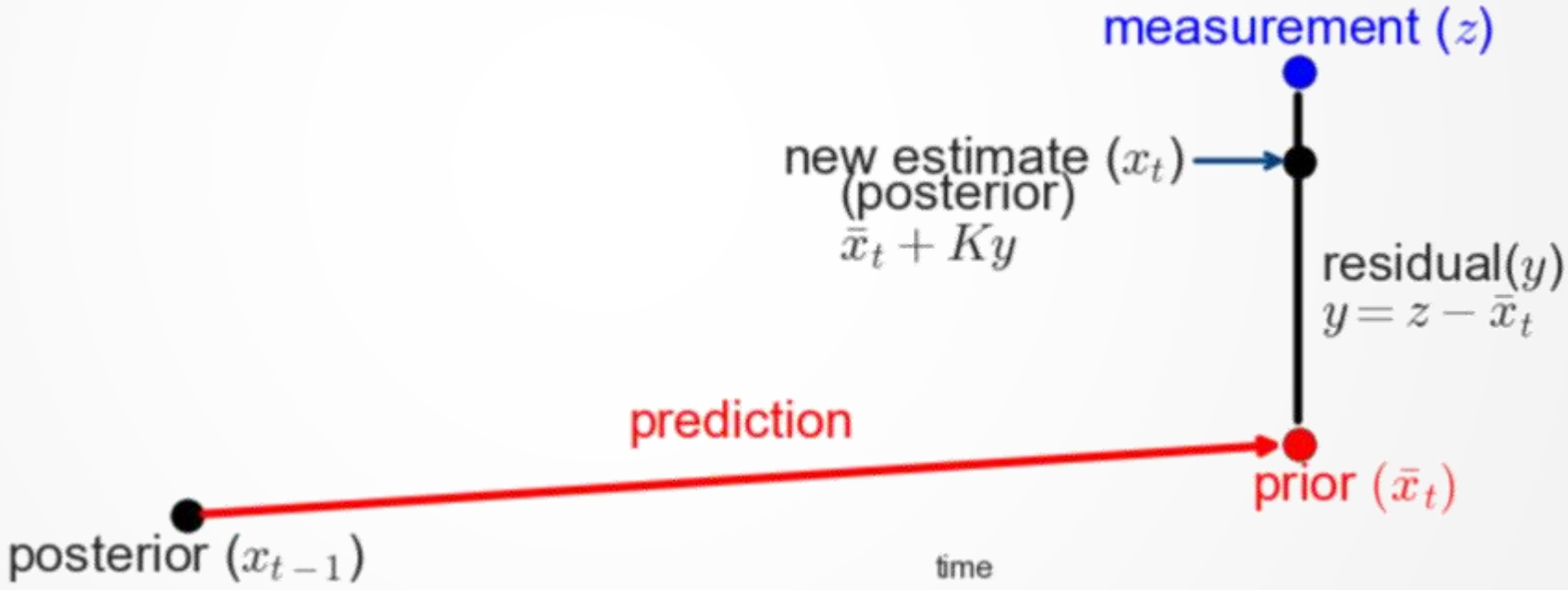
$$K = W_1 \Rightarrow K = \frac{\bar{\sigma}^2}{\bar{\sigma}^2 + \sigma_z^2}$$

  - Then: $\mu = K\mu_z + (1 - K)\bar{\mu} \Rightarrow \mu = \bar{\mu} + K(\mu_z - \bar{\mu})$

Designing a simple Kalman Filter in Python

What is happening: The Kalman Gain

- A visualization of how the Kalman Filter weighting process works:



Designing a simple Kalman Filter in Python

What is happening: The Kalman Gain

- Then the update function of the filter can be written as:

```
def update(prior, measurement):
    x, P = prior        # mean and variance of prior
    z, R = measurement  # mean and variance of measurement

    y = z - x        # residual
    K = P / (P + R)  # Kalman gain

    x = x + K*y           # posterior
    P = (P * R) / (P + R) # posterior variance
    return x, P
```

Multivariate Kalman Filter

- **In fact we implement a discretized continuous-time kinematic filter and they are applicable to Newtonian systems.**
  - For such systems, and under constant velocity assumption (applicable to an iteration):

$$
\begin{aligned}
v &= \frac{dx}{fy} \\
dx &= vdt \\
\int_{x_0}^{x} &= \int vdt \\
x &= vt + x_0
\end{aligned}
$$

  - Note that for many systems/processes, state update integration is not that simple.

Multivariate Kalman Filter

Kalman Filter Algorithm

Initialization

- Initialize the state of the filter
- Initialize our belief in the state

Predict

- Use process model to predict the state at the next time step
- Adjust belief to account for the uncertainty in the prediction

Update/Correct

- Get a measurement and associated belief about its accuracy
- Compute residual between estimated state and measurement
- Compute scaling factor based on whether the measurement or prediction is more accurate
- Set state between the prediction and measurement based on scaling factor
- Update belief in the state based on how certain we are in the measurement.

Multivariate Kalman Filter

Kalman Filter Algorithm

Predict

Univariate	Multivariate
$\bar{x} = x + f_x$	$\bar{\mathbf{x}} = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u}$
$\bar{\sigma}^2 = \sigma^2 + \sigma^2_{f_x}$	$\bar{\mathbf{P}} = \mathbf{F}\mathbf{P}\mathbf{F}^{\mathsf{T}} + \mathbf{Q}$

- $\boldsymbol{X}, \boldsymbol{P}$ are the state mean and covariance. They correspond to x and σ^2
- $\boldsymbol{F}, \boldsymbol{Q}$ are the process mean and covariance. They correspond to f_x and $\sigma_{fx}{}^2$
- $\boldsymbol{B}$ and $\boldsymbol{u}$ are the control allocation matrix and the input vector.

Multivariate Kalman Filter

Kalman Filter Algorithm

Update/Correction

Univariate	Multivariate
$y = z - \bar{x}$	$\mathbf{y} = \mathbf{z} - \mathbf{H}\bar{\mathbf{x}}$
$K = \frac{\bar{\sigma}^2}{\bar{\sigma}^2 + \sigma_z^2}$	$\mathbf{K} = \bar{\mathbf{P}}\mathbf{H}^\mathsf{T}(\mathbf{H}\bar{\mathbf{P}}\mathbf{H}^\mathsf{T} + \mathbf{R})^{-1}$
$x = \bar{x} + Ky$	$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{K}\mathbf{y}$
$\sigma^2 = \frac{\bar{\sigma}^2\sigma_z^2}{\bar{\sigma}^2 + \sigma_z^2}$	$\mathbf{P} = (\mathbf{I} - \mathbf{K}\mathbf{H})\bar{\mathbf{P}}$

- $\boldsymbol{H}$ is the measurement function.
- $\boldsymbol{z}, \boldsymbol{R}$ are the measurement mean and noise covariance.
- $\boldsymbol{y}, \boldsymbol{K}$ are the residual and Kalman gain.
- The details will be different than the univariate filter because these are vectors and matrices, but the concepts are the same:
  - Use a Gaussian to represent our estimate of the state and error
  - Use a Gaussian to represent the measurement and its error
  - Use a Gaussian to represent the process model
  - Use the process model to predict the next state (the prior)
  - Form an estimate part way between the measurement and the prior

Multivariate Kalman Filter

Predict Step

- For the multivariate case we will introduce "hidden" (velocity) variables. Simulation of the motion:

```
import math
import numpy as np
from numpy.random import randn

def compute_dog_data(z_var, process_var, count=1, dt=1.):
    x, vel = 0., 1.
    z_std = math.sqrt(z_var)
    p_std = math.sqrt(process_var)
    xs, zs = [], []
    for _ in range(count):
        v = vel + (randn() * p_std * dt)
        x += v*dt
        xs.append(x)
        zs.append(x + randn() * z_std)
    return np.array([xs, zs]).T
```

- Then load the Kalman prediction, update/correction functions:

```
from filterpy.kalman import predict, update
```

Multivariate Kalman Filter

Predict Step

- For this problem we are tracking both the position and the velocity ("hidden" variable). This requires to use a multivariate Gaussian: $\boldsymbol{x}$ and $\boldsymbol{P}$.
  - Note that variables might be observed (a sensor measures a relevant value) or hidden. Velocity does not have to be hidden always. This was simply an assumption for the example. The state vector contains both these values, although the outputs vector might only contain some of them.

Design State Covariance

- The covariance in a multivariate Kalman Filter has the role of the variance in a univariate KF.
- The **Covariance Matrix** is structured as follows:
  - **Covariance Matrix diagonal terms:** variances of each variable.
  - **Covariance Matrix off-diagonal terms:** covariances among the variables.
    - example:

$$\mathbf{P} = \begin{bmatrix} 500 & 0 \\ 0 & 49 \end{bmatrix}$$

```
P = np.diag([500, 49])
%precision 3
P
```

Multivariate Kalman Filter

Predict Step

Design the Process Model

- The transition model (this is what is computed within the predict function):

$$\bar{\mathbf{x}} = \mathbf{F}\mathbf{x}$$

```
from filterpy.kalman import predict

x = np.array([10.0, 4.5])
P = np.diag([500, 49])

# Q is amount of noise in the process,
# we will learn about it later
Q = 0
x, P = predict(x, P, F, Q)
x
```

- The filter predicts how the state will be as well as the covariance matrix.

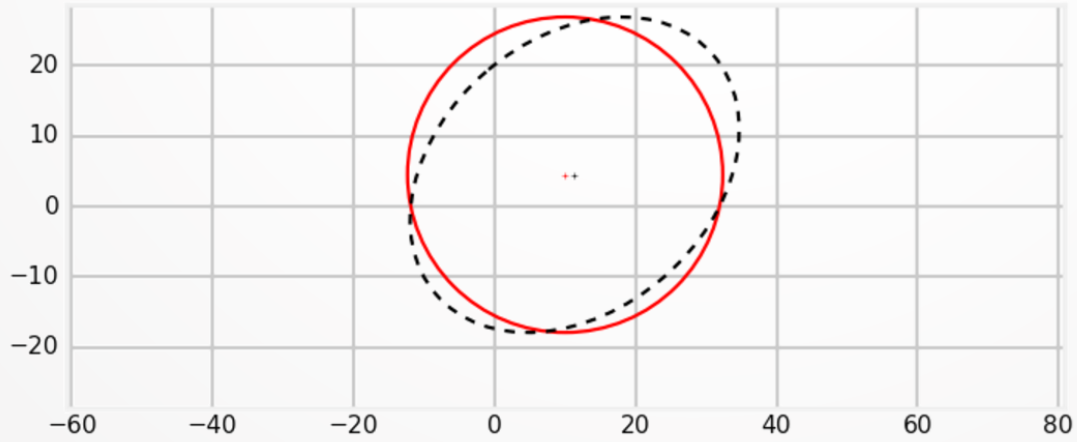
Multivariate Kalman Filter

Predict Step

Design the Process Model

- Let's have a look at the covariance matrix:

```
from code.book_plots import interactive_plot
from filterpy.stats import plot_covariance_ellipse

dt = 0.3
F = np.array([[1, dt], [0, 1]])
x = np.array([10.0, 4.5])
P = np.diag([500, 500])
with interactive_plot():
    plot_covariance_ellipse(x, P, edgecolor='r')
    x, P = predict(x, P, F, Q)
    plot_covariance_ellipse(x, P, edgecolor='k', ls='dashed')
```




```
array([[ 512.25,   24.5 ],
       [  24.5 ,   49.  ]])
```

off-diagonal terms!

Multivariate Kalman Filter

Predict Step

Design the Process Noise

- In general the noise is added (for LTI systems as):

$$\dot{\mathbf{x}} = f(\mathbf{x}) + \mathbf{w} \quad \text{LINEAR:} \quad \dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

- In general refer to the previous sections on how it is modeled. For the moment let's stick on:

$$\mathbf{Q} = E[\mathbf{w}\mathbf{w}^T]$$

```
from filterpy.common import Q_discrete_white_noise
Q = Q_discrete_white_noise(dim=2, dt=1., var=2.35)
Q
```

Multivariate Kalman Filter

Predict Step

Design the Control Function

- The complete –before noise- representation will be:

$$\bar{\mathbf{x}} = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u}$$

- Or in standard continuous time notation:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

- Including noise:

$$\bar{\mathbf{x}} = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{w}$$

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{w}$$

- Overall Kalman Filter Prediction step `predict(x, P, F, Q, B, u)`

- For this example:

```
def predict(x, P, F, Q, B, u):
    u = 0
    B = 0.  # we do not know how to start filtering so we
```

Multivariate Kalman Filter

Predict Step

- **Summary of Design parameters:**
  - $\boldsymbol{X}, \boldsymbol{P}$: the state and covariance
  - $\boldsymbol{F}, \boldsymbol{Q}$: the process model and noise covariance
  - $\boldsymbol{B}, \boldsymbol{u}$: if present, the control matrix and input vector

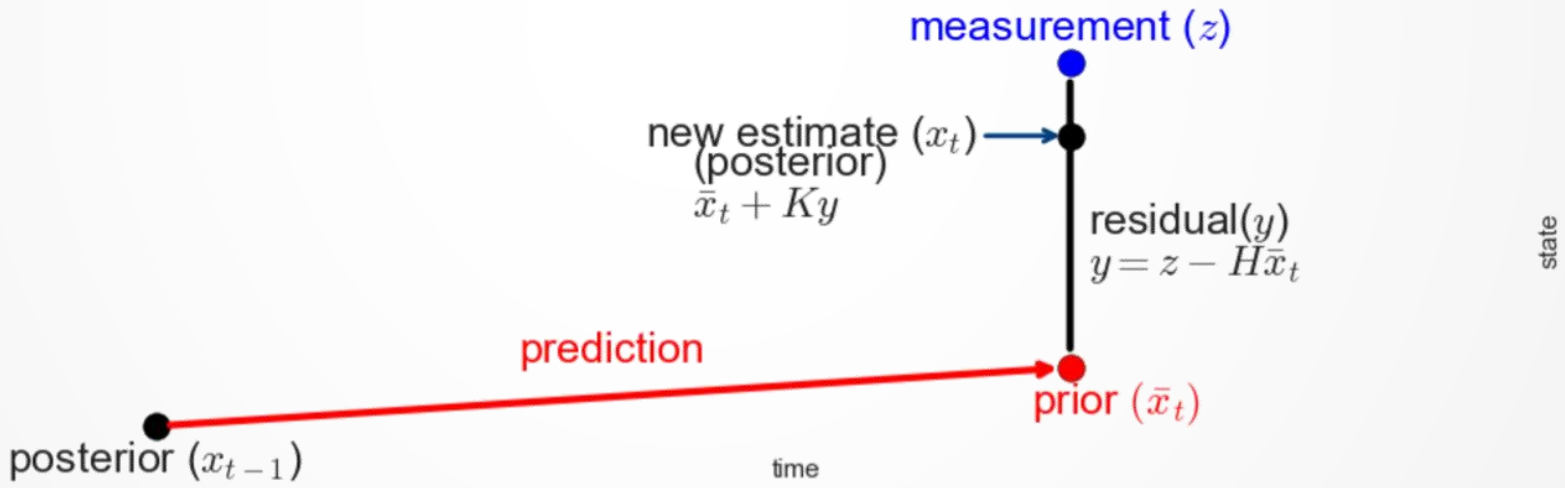
Multivariate Kalman Filter

Update/Correction Step

Design the Measurement Function

- Form of the measurement function:

$$\mathbf{y} = \mathbf{z} - \mathbf{H}\bar{\mathbf{x}}$$



Multivariate Kalman Filter

Update/Correction Step

Design the Measurement

- The measurement has mean $\boldsymbol{z}$ and covariance $\boldsymbol{R}$.
- It corresponds to the covariance matrix on the measurement noise elements, their own variances and covariances.

Multivariate Kalman Filter

Update/Correction Step

- Coding Form of the Kalman Filter update step:

```
from filterpy.kalman import update
z = 1.
x, P = update(x, P, z, R, H)
x
```

Multivariate Kalman Filter

Implementing the Kalman Filter

- **Write the filter**

```
from filterpy.kalman import KalmanFilter
from filterpy.common import Q_discrete_white_noise

def pos_vel_filter(x, P, R, Q=0., dt=1.0):
    """ Returns a KalmanFilter which implements a
    constant velocity model for a state [x dx].T
    """

    kf = KalmanFilter(dim_x=2, dim_z=1)
    kf.x = np.array([x[0], x[1]]) # location and velocity
    kf.F = np.array([[1, dt],
                     [0,  1]])    # state transition matrix
    kf.H = np.array([[1, 0]])     # Measurement function
    kf.R *= R                    # measurement uncertainty
    if np.isscalar(P):
        kf.P *= P                 # covariance matrix
    else:
        kf.P[:] = P
    if np.isscalar(Q):
        kf.Q = Q_discrete_white_noise(dim=2, dt=dt, var=Q)
    else:
        kf.Q = Q
    return kf
```

Multivariate Kalman Filter

Implementing the Kalman Filter

- **Create the filter**

```
dt = .1
x = np.array([0., 0.])
kf = pos_vel_filter(x, P=500, R=5, Q=0.1, dt=dt)
```

Multivariate Kalman Filter

Implementing the Kalman Filter

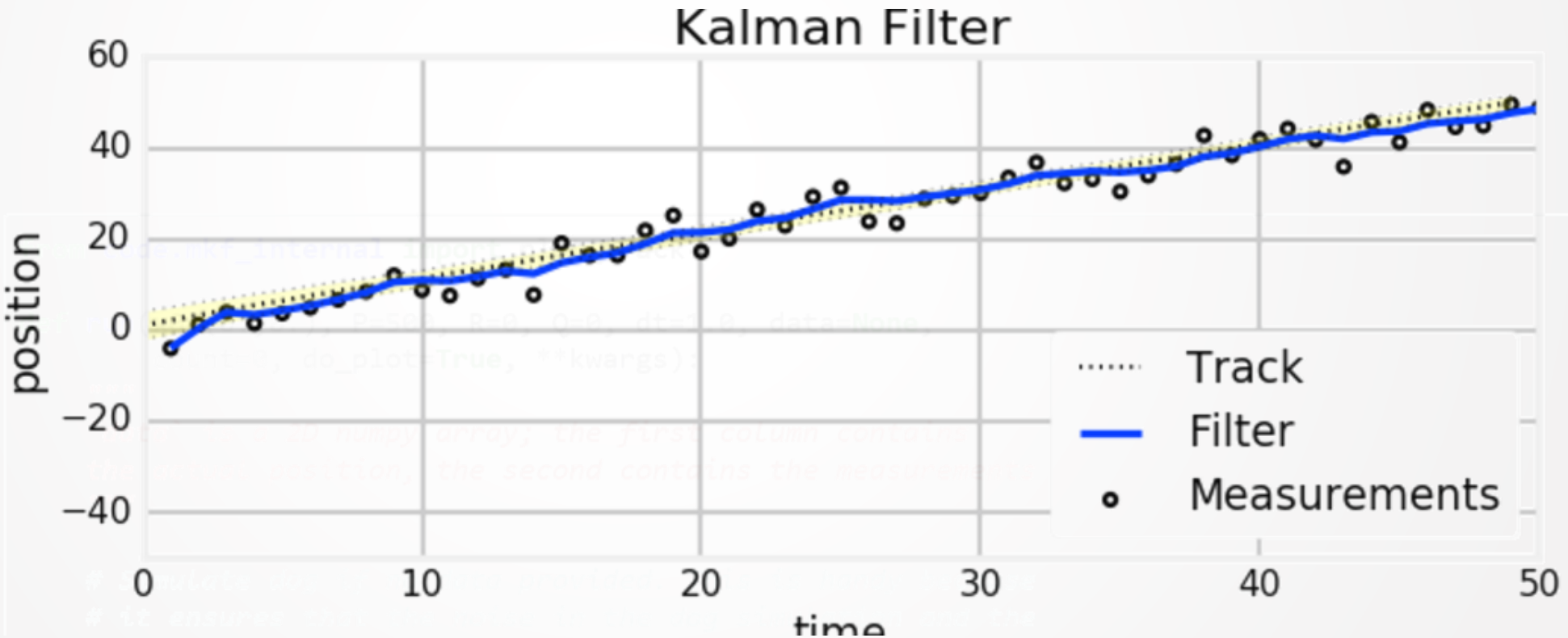
- **Run the filter**

```
from code.mkf_internal import plot_track

def run(x0=(0.,0.), P=500, R=0, Q=0, dt=1.0, data=None,
        count=0, do_plot=True, **kwargs):
    """
    `data` is a 2D numpy array; the first column contains
    the actual position, the second contains the measurements
    """

    # Simulate dog if no data provided. This is handy because
    # it ensures that the noise in the dog simulation and the
    # kalman filter are the same.
    if data is None:
        data = compute_dog_data(R, Q, count)

    # create the Kalman filter
    kf = pos_vel_filter(x0, R=R, P=P, Q=Q, dt=dt)

    # run the kalman filter and store the results
    xs, cov = [], []
    for z in data[:, 1]:
        kf.predict()
        kf.update(z)
        xs.append(kf.x)
        cov.append(kf.P)

    xs, cov = np.array(xs), np.array(cov)
    if do_plot:
        plot_track(xs[:, 0], data[:, 0], data[:, 1], cov,
                   dt=dt, **kwargs)
    return xs, cov
```

Multivariate Kalman Filter

Implementing the Kalman Filter

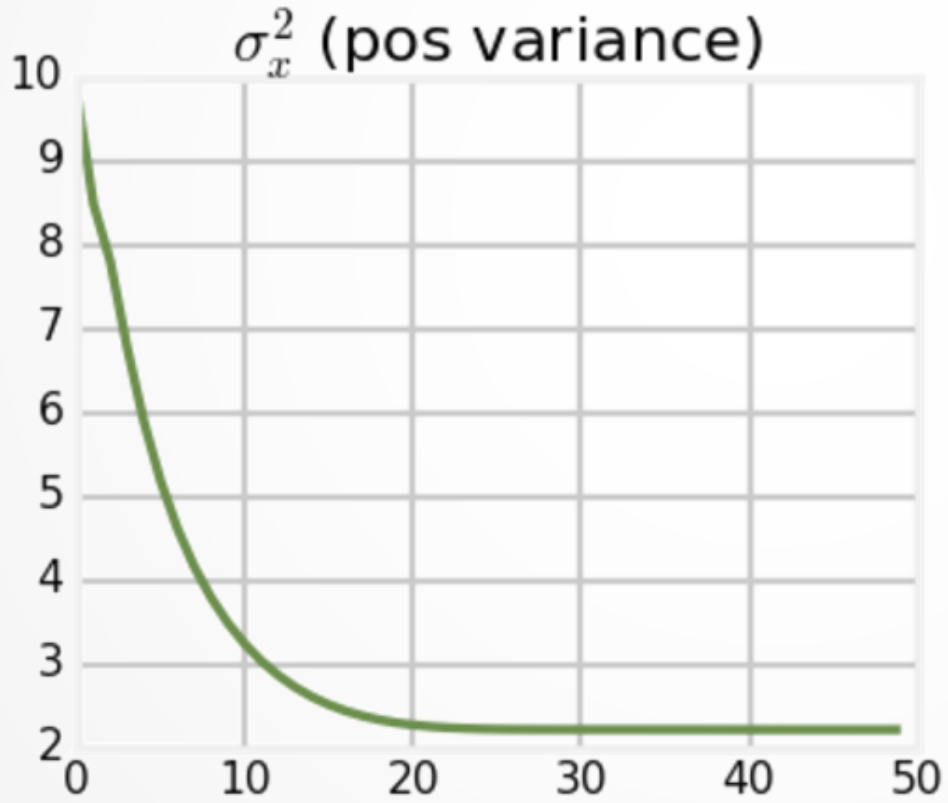
- **Results**

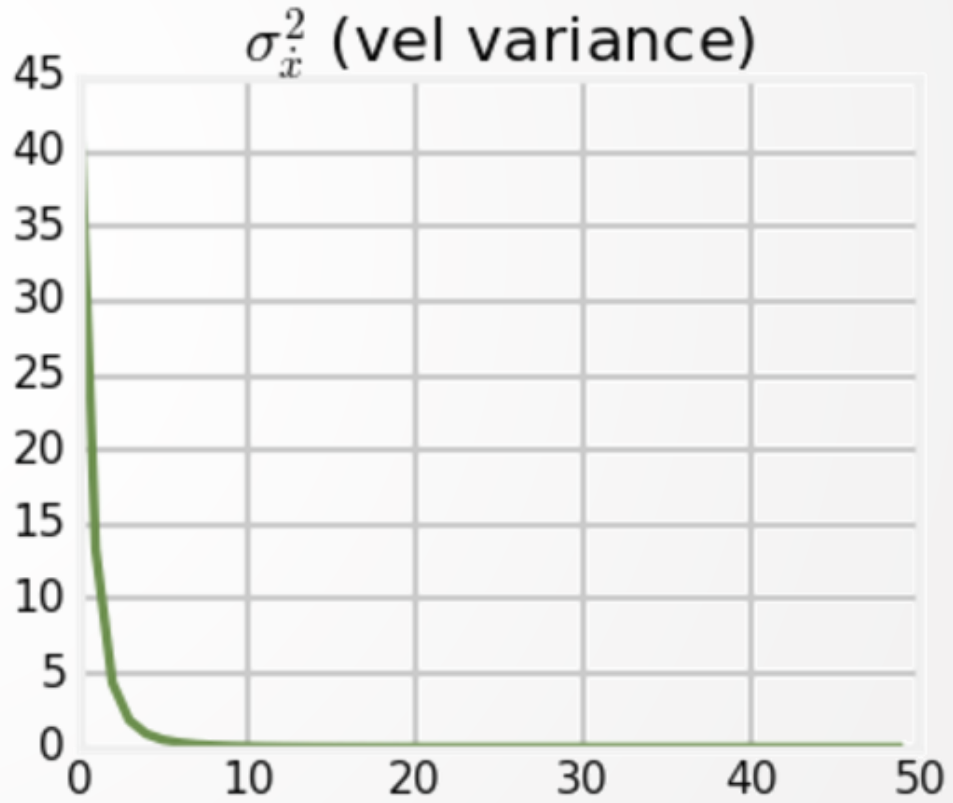


Multivariate Kalman Filter

Implementing the Kalman Filter

- **Results**



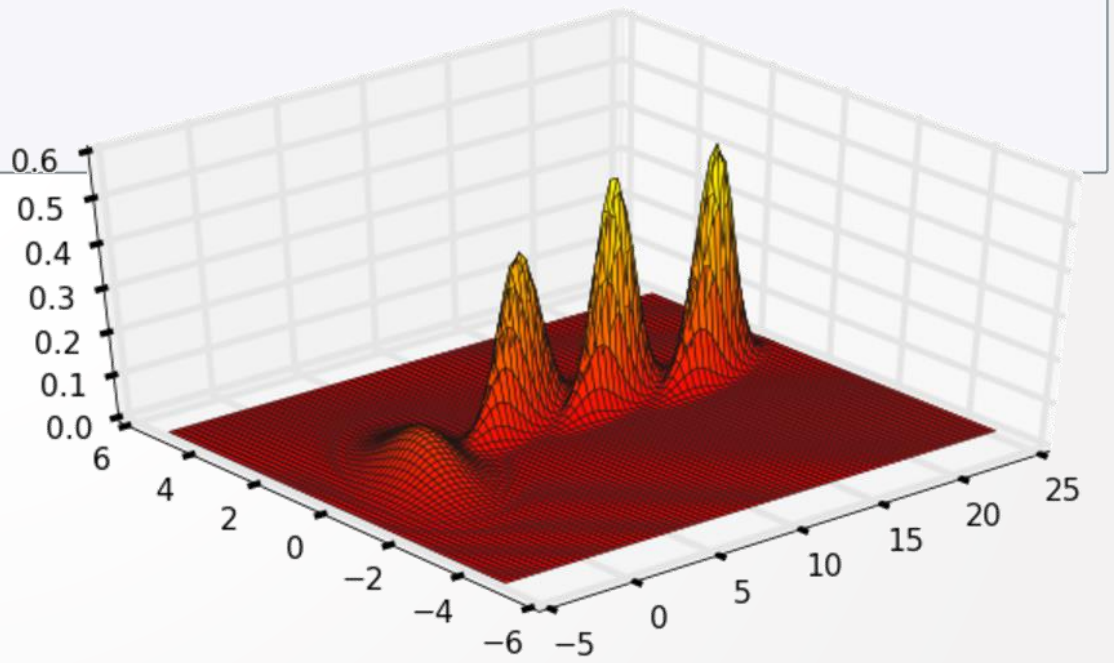


Multivariate Kalman Filter

Implementing the Kalman Filter

- **Results :: Plot the Gaussians**

```
from book_format import set_figsize, figsize
from code.nonlinear_plots import plot_gaussians

P = np.diag([3., 1.])
np.random.seed(3)
Ms, Ps = run(count=25, R=10, Q=0.01, P=P, do_plot=False)
with figsize(x=9, y=5):
    plot_gaussians(Ms[::7], Ps[::7], (-5,25), (-5, 5), 75)
```

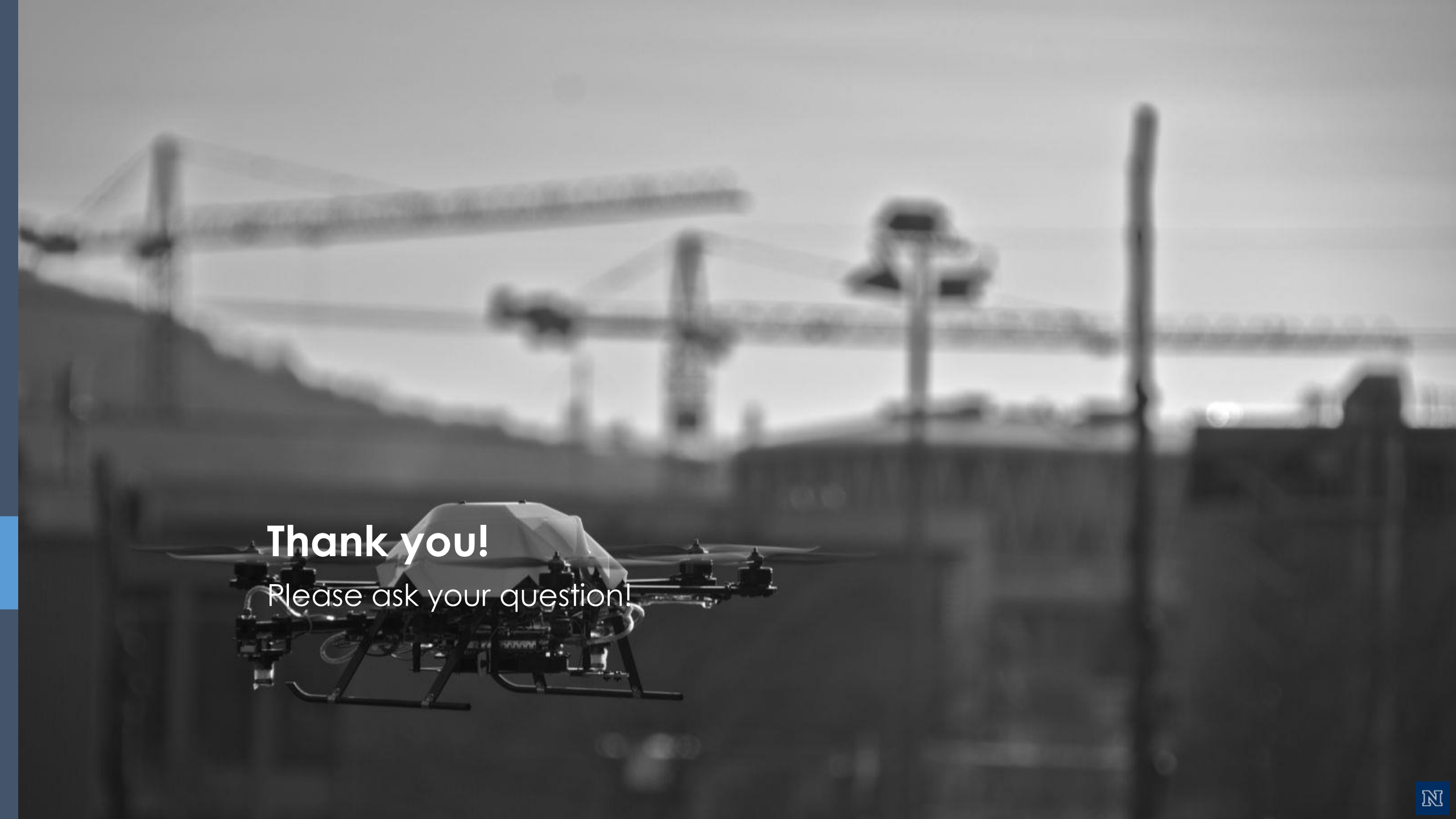


Multivariate Kalman Filter

- **In fact we implement a discretized continuous-time kinematic filter and they are applicable to Newtonian systems.**
  - For such systems, and under constant velocity assumption (applicable to an iteration):

$$
\begin{aligned}
v &= \frac{dx}{fy} \\
dx &= vdt \\
\int_{x_0}^{x} &= \int vdt \\
x &= vt + x_0
\end{aligned}
$$

  - Note that for many systems/processes, state update integration is not that simple.



Thank you!

Please ask your question!