Kalman Filters: from Bayes to Inverse Problems

Mark Asch

2024-06-27

Table of contents

Welcome to *"Kalman Filters: from Bayes to Inverse Problems"*

This book contains a presentation of Kalman filters, from basics to nonlinear and ensemble filters. To understand these well, examples are provided in the form of jupyter notebooks. Then the notion of Bayesian inverse problems (BIP) is introduced. Finally, there is a detailed presentation of the use of the ensemble Kalman filter as a basis for the solution of *inverse problems*. This is denoted EKI, or ensemble Kalman inversion, following the magnificent work of Andrew Stuart and his collaborators.

This book is based upon a number of sources. The original Bayesian formulation for inverse problems (Dashti and Stuart 2015) was the basis for the later ensemble Kalman inversion, presented in a series of papers (Iglesias, Law, and Stuart 2013; Calvello, Reich, and Stuart 2022; Huang et al. 2022).

In (Asch, Bocquet, and Nodet 2016) and (Law, Stuart, and Zygalakis 2015) the reader can find detailed presentations of Kalman filter approaches for data assimilation. In (Asch 2022) there are basic explanations of uncertainty quantification, inverse problems and their use for digital twins.

Author

Mark Asch is Emeritus Professor at the Université de Picardie Jules Verne in the mathematics department.

https://markasch.github.io/DT-tbx-v1/, http://masch.perso.math.cnrs.fr/

Citation

Asch, Mark. Kalman Filters: from Bayes to Inverse Problems. (2024) https://markasch.github.io/kfBIPq/

License

References

Asch, Mark. 2022. *A Toolbox for Digital Twins: From Model-Based to Data-Driven*. Philadelphia, PA: Society for Industrial; Applied Mathematics. https://doi.org/10.1137/1.9781611976977.

Asch, Mark, Marc Bocquet, and Maëlle Nodet. 2016. *Data Assimilation: Methods, Algorithms, and Applications*. Philadelphia, PA: Society for Industrial; Applied Mathematics. https://doi.org/10.1137/1.9781611974546.

Calvello, Edoardo, Sebastian Reich, and Andrew M. Stuart. 2022. "Ensemble Kalman Methods: A Mean Field Perspective." arXiv (to appear in Acta Numerica 2025). http://arxiv.org/abs/2209.11371.

Carrillo, J. A., F. Hoffmann, A. M. Stuart, and U. Vaes. 2024a. "Statistical Accuracy of Approximate Filtering Methods." https://arxiv.org/abs/2402.01593.

———. 2024b. "The Mean Field Ensemble Kalman Filter: Near-Gaussian Setting." https://arxiv.org/abs/2212.13239.

Dashti, Masoumeh, and Andrew M. Stuart. 2015. "The Bayesian Approach to Inverse Problems." In *Handbook of Uncertainty Quantification*, edited by Roger Ghanem, David Higdon, and Houman Owhadi, 1–118. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-11259-6_7-1.

Huang, Daniel Zhengyu, Jiaoyang Huang, Sebastian Reich, and Andrew M Stuart. 2022. "Efficient Derivative-Free Bayesian Inference for Large-Scale Inverse Problems." *Inverse Problems* 38 (12): 125006. https://doi.org/10.1088/1361-6420/ac99fa.

Iglesias, Marco A, Kody J H Law, and Andrew M Stuart. 2013. "Ensemble Kalman Methods for Inverse Problems." *Inverse Problems* 29 (4): 045001. https://doi.org/10.1088/0266-5611/29/4/045001.

Law, Kody, Andrew Stuart, and Konstantinos Zygalakis. 2015. *Data Assimilation: A Mathematical Introduction*. Vol. 62. Texts in Applied Mathematics. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-20325-6.

Sanita Vetra-Carvalho, Lars Nerger, Peter Jan van Leeuwen, and Jean-Marie Beckers. 2018. "State-of-the-Art Stochastic Data Assimilation Methods for High-Dimensional Non-Gaussian Problems." *Tellus A: Dynamic Meteorology and Oceanography* 70 (1): 1–43. https://doi.org/10.1080/16000870.2018.1445364.

Särkkä, S., and L. Svensson. 2023. *Bayesian Filtering and Smoothing*. 2nd ed. Institute of Mathematical Statistics Textbooks. Cambridge University Press. https://doi.org/10.1017/9781108917407.

Wu, Jin-Long, Matthew E. Levine, Tapio Schneider, and Andrew Stuart. 2023. "Learning about Structural Errors in Models of Complex Dynamical Systems." https://arxiv.org/abs/2401.00035.

Part I

Bayes' Theory

1 Bayes' Theorem

1.1 Introduction

Bayes' theorem is at the core of modern uncertainty quantification.

Theorem 1.1 (Bayes Theorem)**.** *If we have two events, a and b, then*

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)},$$

where $P(a|b)$ is the conditional probability of a given b.

In a more general setting, suppose that we have observations, $y_{\text{obs}} \in \mathbb{R}^{N_y}$, of a state variable, x, then the conditional PDF, $\pi_X(x|y_{\text{obs}})$, is given by Bayes' formula

$$\pi_X(x|y_{\text{obs}}) = \frac{\pi_Y(y_{\text{obs}}|x)\pi_X(x)}{\pi_Y(y_{\text{obs}})}.$$

Here, π_X, the *prior* PDF, quantifies our uncertainty about the state/parameters X **before** observing y_{obs}, while $\pi_X(x|y_{\text{obs}})$, the *posterior* PDF, quantifies our uncertainty **after** observing y_{obs}. The conditional PDF $\pi_Y(y_{\text{obs}}|x)$ quantifies the likelihood of observing y given a particular value of x. The denominator, $\pi_Y(y_{\text{obs}})$, is simply a normalizing factor, and can be computed in a post-processing step.

We thus rewrite the formula in Theorem 1.1 as

$$\pi_X(x|y_{\text{obs}}) \propto \pi_Y(y_{\text{obs}}|x)\pi_X(x).$$

1.2 Theory

Still to come...

1.3 Some Examples

Coming soon...

2 Bayesian Filters

All filters can be expressed in a Bayesian form.

> **!** Important
>
> Kalman filters are a special case of the more general Bayesian filter formulation

Coming soon...

Part II

Kalman Filters

3 Kalman Filters

3.1 Introduction

The Kalman filter can be used for

- state estimation—this is the direct filtering (or smoothing) problem
- parameter estimation—this is the inverse problem based on filtering with a pseudo-time

Kalman filters are linear (and Gaussian) or nonlinear. Among the nonlinear approaches, the Ensemble Kalman filter (EnKF) provides a *derivative-free* approach to the solution of inverse problems.

> Definition: filtering problem
>
> Filtering is the sequential updating of the probability distribution of the state of a (possibly stochastic) dynamical system, given partial (or sparse), noisy observations.
> The KF provides an explicit, optimal solution to this problem in the setting of linear dynamical systems, linear observations, and additive Gaussian noise. In this case, we obtain explicit update formulas for the mean and covariance of the resulting posterior Gaussian probability distribution.

3.2 Kalman filter problem - General formulation

We present a very general formulation that will later be convenient for joint state and parameter estimation problems.

We consider a discrete-time dynamical system with noisy state transitions and noisy observations.

- Dynamics:

$$v_{j+1} = \Psi(v_j) + \xi_j, \quad j \in \mathbb{Z}^+$$

- Observations:

$$y_{j+1} = h(v_{j+1}) + \eta_{j+1}, \quad j \in \mathbb{Z}^+$$

- Probability (densities):

$$v_0 \sim \mathcal{N}(m_0, C_0), \quad \xi_j \sim \mathcal{N}(0, \Sigma), \quad \eta_j \sim \mathcal{N}(0, \Gamma)$$

- Probability (independence):

$$v_0 \perp \xi_j \perp \eta_j$$

- Operators:

$$\Psi \colon \mathcal{H}_s \mapsto \mathcal{H}_s, \tag{3.1}$$
$$h \colon \mathcal{H}_s \mapsto \mathcal{H}_o, \tag{3.2}$$

  where $v_j \in \mathcal{H}_s$, $y_j \in \mathcal{H}_o$ and $\mathcal{H}$ is a finite-dimensional Hilbert space.

Filtering problem: estimate (optimally) the state v_j of the dynamical system at time j, given the data $Y_j = \{y_i\}_{i=1}^j$ up to time j. This is achieved by using a two-step *predictor-corrector* method. We will use the more general notation of (Law, Stuart, and Zygalakis 2015) instead of the usual, classical state-space formulation that is used in (Asch, Bocquet, and Nodet 2016) and (Asch 2022).

The objective is to update the filtering distribution $\mathbb{P}(v_j|Y_j)$, from time j to time $j+1$, in the linear, Gaussian case, where - Ψ and h are linear maps - all distributions are Gaussian.

Suppose

$$\Psi(v) = Mv \tag{3.3}$$
$$h(v) = Hv, \tag{3.4}$$

where the matrices $M \in \mathbb{R}^{n \times n}$, $H \in \mathbb{R}^{m \times n}$, with $m \leq n$ and $\text{rank}(H) = m$.

1. Let (m_j, C_j) denote the mean and covariance of $v_j|Y_j$ and note that these entirely characterize the random variable since it is Gaussian.
2. Let $(\hat{m}_{j+1}, \hat{C}_{j+1})$ denote the mean and covariance of $v_{j+1}|Y_j$ and note that these entirely characterize the random variable since it is Gaussian.
3. Derive the map $(m_j, C_j) \mapsto (m_{j+1}, C_{j+1})$ using the previous step.

3.2.1 Prediction/Forecast

$$\mathbb{P}(v_n|y_1,\ldots,y_n) \mapsto \mathbb{P}(v_{n+1}|y_1,\ldots,y_n)$$

- P0: initialize (m_0, C_0) and compute v_0
- P1: predict the state, measurement

$$v_{j+1} = Mv_j + \xi_j \tag{3.5}$$
$$y_{j+1} = Hv_{j+1} + \eta_{j+1} \tag{3.6}$$

- P2: predict the mean and covariance

$$\hat{m}_{j+1} = Mm_j \tag{3.7}$$
$$\hat{C}_{j+1} = MC_jM^{\mathrm{T}} + \Sigma \tag{3.8}$$

3.2.2 Correction/Analysis

$$\mathbb{P}(v_{n+1}|y_1,\ldots,y_n) \mapsto \mathbb{P}(v_{n+1}|y_1,\ldots,y_{n+1})$$

- C1: compute the innovation

$$d_{j+1} = y_{j+1} - H\hat{m}_{j+1}$$

- C2: compute the measurement covariance

$$S_{j+1} = H\hat{C}_{j+1}H^{\mathrm{T}} + \Gamma$$

- C3: compute the (optimal) Kalman gain

$$K_{j+1} = \hat{C}_{j+1}H^{\mathrm{T}}S_{j+1}^{-1}$$

- C4: update/correct the mean and covariance

$$m_{j+1} = \hat{m}_{j+1} + K_{j+1}d_{j+1}, \tag{3.9}$$
$$C_{j+1} = \hat{C}_{j+1} - K_{j+1}S_{j+1}K_{j+1}^{\mathrm{T}}. \tag{3.10}$$

3.2.3 Loop over time

- set $j = j + 1$
- go to step P1

3.3 State-space formulation

In classical (linear) filter theory, a state space formulation is usually used.

$$x_{k+1} = Fx_k + Bu_k + w_k \tag{3.11}$$
$$y_{k+1} = Hx_k + v_k, \tag{3.12}$$

where u is a control input, and $w_k \sim \mathcal{N}(0, Q)$, $v_k \sim \mathcal{N}(0, R)$. Moreover, A is the dynamics and H the observation operator.

The 2-step filter:

Initialization

$$x_0, \quad P_0$$

1. Prediction

$$x^-_{k+1} = Fx_k \tag{3.13}$$
$$P^-_{k+1} = FP_kF^{\mathrm{T}} + Q \tag{3.14}$$

2. Correction

$$K_{k+1} = P^-_{k+1}H^{\mathrm{T}}(HP^-_{k+1}H^{\mathrm{T}} + R)^{-1} \quad (= P^-_{k+1}H^{\mathrm{T}}S^{-1}) \tag{3.15}$$
$$x_{k+1} = x^-_{k+1} + K_{k+1}(y_{k+1} - Hx^-_{k+1}) \tag{3.16}$$
$$P_{k+1} = (I - K_{k+1}H)P^-_{k+1} \quad (= P^-_{k+1} - K_{k+1}SK^{\mathrm{T}}_{k+1}) \tag{3.17}$$

Loop

Set $k = k + 1$ and go to step 1.

> **Tip**
>
> In some cases, the superscripts `f` and `a` are used to denote the forecast/prediction, and analysis/correction variables, respectively. We avoid this, for clarity of notation.

3.4 Passage from Continuous to Discrete for Random Dynamic Systems

Most often, in the use of Kalman filtering, we deal with a continuous dynamic system that is modelled by a system of ODEs, with some random process noise. The passage from the continuous to a discrete-time formulation, which is needed for the KF, requires some attention. Following {cite}Sarkka2023, we will derive the associated discretization of the process matrix and the process noise covariance matrix.

Suppose we have a linear, time invariant (LTI) system of ODEs with an additive random noise term (this is the engineering approach that avoids the use of SDEs and Itô calculus—see {cite}Asch2022)

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = F\mathbf{x}(t) + L\mathbf{w}(t), \quad \mathbf{x}(0) = \mathbf{x}_0,$$

where $\mathbf{w}(t)$ is a white, Gaussian noise process with expectation and covariance

$$\mathrm{E}[\mathbf{w}(t)] = 0, \tag{3.18}$$

$$\mathrm{Cov}[\mathbf{w}(\tau_1)\mathbf{w}(\tau_2)] = \delta(\tau_1 - \tau_2)Q^c \tag{3.19}$$

with Q^c the spectral density matrix of the white noise process, which is the continuous-time analogue of a covariance matrix, and in the scalar case is just the noise variance. Usually

$$Q^c = \mathrm{diag}(q_1^c, \ldots, q_n^c),$$

where $q_1^c, \ldots, q_n^c$ are the spectral densities of $w_1(t), \ldots, w_n(t)$, the components of $\mathbf{w}(t)$.

We now proceed to convert this continous LTI system into its discrete counterpart, needed for the KF algorithm,

$$\mathbf{x}_{k+1} = A_k\mathbf{x}_k + \mathbf{q}_k,$$

where A_k is the discretized process/dynamics transition matrix, and $\mathbf{q}_k \sim \mathcal{N}(0, Q)$ is the Gaussian process noise.

To obtain expressions for A and Q, we need to solve the random ODE system. Doing this, we can show that

$$A_k = \exp(F\Delta t_k) = \sum_{n=0}^{\infty} \frac{(F\Delta t_k)^n}{n!}$$

and

$$Q_k = \int_0^{\Delta t_k} \exp(Fs) L Q^c L^{\mathrm{T}} \exp(Fs)^{\mathrm{T}} \mathrm{d}s.$$

Usually, F is nilpotent to some low order, so the computation of the matrix exponential is relatively easy to do. This will be illustrated in the examples.

4 Example 1 - estimating a constant

In this simple numerical example let us attempt to estimate a scalar random constant, a voltage for example. Let us assume that we can obtain measurements of the constant, but that the measurements are corrupted by a 0.1 volt RMS white measurement noise (e.g. our analog-to-digital converter is not very accurate).

Here, we will use data assimilation notation, where

- `f` denotes forecast (or prediction)
- `a` denotes analysis (or correction)
- `t` denotes the true value.

In this scalar, 1D example, our process is governed by the state equation,

$$x_k = Fx_{k-1} + w_k = x_{k-1} + w_k$$

and the measurement equation,

$$y_k = Hx_k + v_k = x^{\mathrm{t}} + v_k.$$

The state, being constant, does not change from step to step, so $F = I$. Our noisy measurement is of the state directly so $H = 1$.

The time-update (forecast) equations are,

$$x^{\mathrm{f}}_{k+1} = x^{\mathrm{a}}_k\,, \tag{4.1}$$

$$P^{\mathrm{f}}_{k+1} = P^{\mathrm{a}}_k + Q \tag{4.2}$$

and the measurement update (analysis) equations are

$$K_{k+1} = P^{\mathrm{f}}_{k+1}(P^{\mathrm{f}}_{k+1} + R)^{-1}, \tag{4.3}$$

$$x^{\mathrm{a}}_{k+1} = x^{\mathrm{f}}_{k+1} + K_{k+1}(y_{k+1} - x^{\mathrm{f}}_{k+1}), \tag{4.4}$$

$$P^{\mathrm{a}}_{k+1} = (1 - K_{k+1})P^{\mathrm{f}}_{k+1}. \tag{4.5}$$

Initialization

Presuming a very small process variance, we let $Q = 1.e-5$. We could certainly let $Q = 0$ but assuming a small but non-zero value gives us more flexibility in tuning the filter as we will demonstrate below. Let us assume from experience that we know the true value of the random constant has a standard Gaussian probability distribution, so we will seed our filter with the guess that the constant is 0. In other words, before starting we let $x_0 = 0$. Similarly we need to choose an initial value for P_k^a, call it P_0. If we were absolutely certain that our initial state estimate was correct, we would let $P_0 = 0$. However, given the uncertainty in our initial estimate x_0, choosing $P_0 = 0$ would cause the filter to initially and always believe that $x_k^a = 0$. As it turns out, the alternative choice is not critical. We could choose almost any $P_0 \neq 0$ and the filter would eventually converge. We will start our filter with $P_0 = 1$.

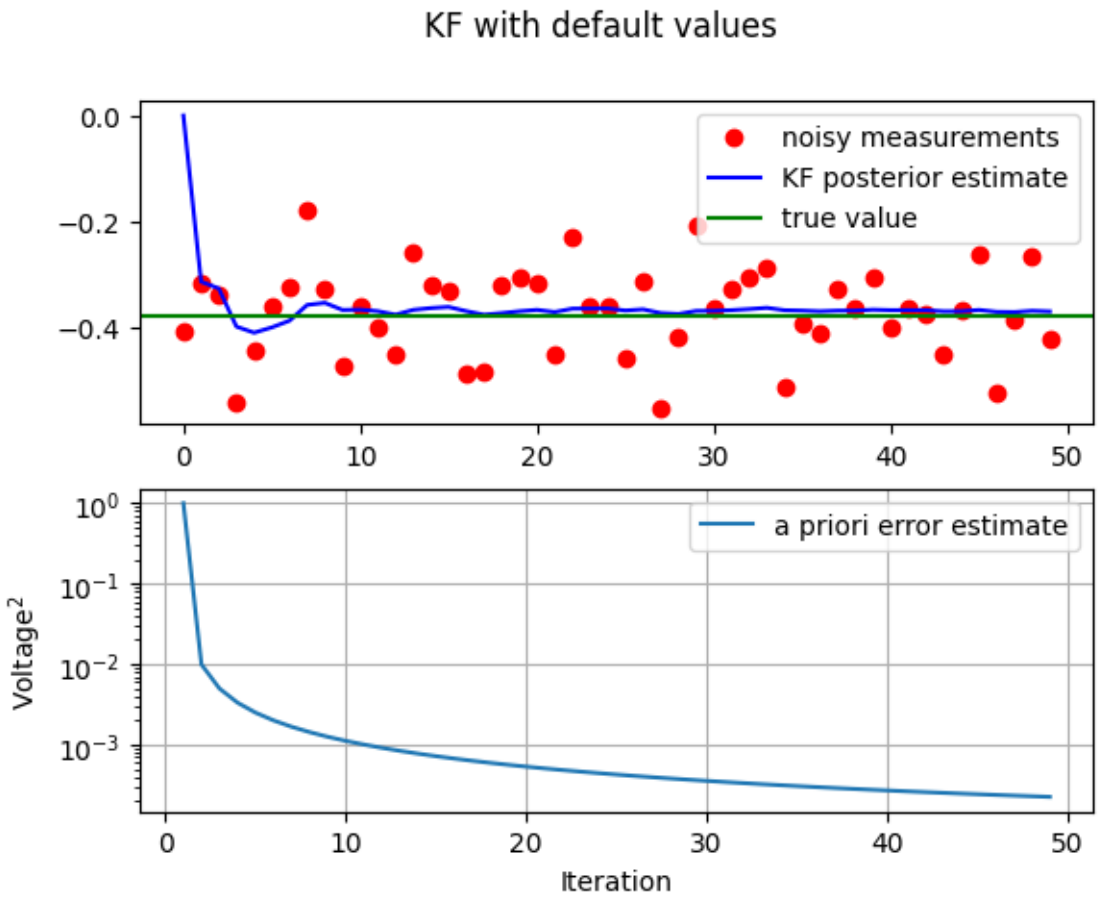
Simulations

To begin with, we randomly chose a scalar constant $y = -0.37727$. We then simulate 100 distinct measurements that have an error normally-distributed around zero with a standard deviation of 0.1 (remember we supposed that the measurements are corrupted by a 0.1 volt RMS white measurement noise).

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(1955)

#plt.rcParams['figure.figsize'] = (10, 6)

def KF_ct(n_iter=50, sig_w=0.001, sig_v=0.1):

    # intial parameters
    n_iter = 50
    sz = (n_iter,) # size of array
    x = -0.37727 # truth value
    z = np.random.normal(x,0.1,size=sz) # observations (normal about x, sigma=0.1)

    Q = sig_w**2 # 1e-6 # process variance

    # allocate space for arrays
    xhat      = np.zeros(sz)      # a posteri estimate of x
    P         = np.zeros(sz)         # a posteri error estimate
    xhatminus = np.zeros(sz) # a priori estimate of x
    Pminus    = np.zeros(sz)    # a priori error estimate
```

```
    K         = np.zeros(sz)         # gain or blending factor

    R = sig_v**2 # 0.1**2 # estimate of measurement variance, change to see effect

    # intial guesses
    xhat[0] = 0.0
    P[0] = 1.0

    for k in range(1,n_iter):
        # time update
        xhatminus[k] = xhat[k-1]
        Pminus[k] = P[k-1] + Q

        # measurement update
        K[k] = Pminus[k]/( Pminus[k] + R )
        xhat[k] = xhatminus[k]+K[k]*(z[k]-xhatminus[k])
        P[k] = (1 - K[k])*Pminus[k]

    fig, (ax1, ax2) = plt.subplots(2,1)
    fig.suptitle('KF with default values')
    ax1.plot(z,'ro',label='noisy measurements')
    ax1.plot(xhat,'b-',label='KF posterior estimate')
    ax1.axhline(x,color='g',label='true value')
    ax1.legend()

    valid_iter = range(1,n_iter) # Pminus not valid at step 0
    ax2.semilogy(valid_iter,Pminus[valid_iter],label='a priori error estimate')
    ax2.grid()
    ax2.legend()
    ax2.set(xlabel='Iteration', ylabel='Voltage$^2$') #, ylim=[0,.01])
```

```
# use all default values
KF_ct()
```



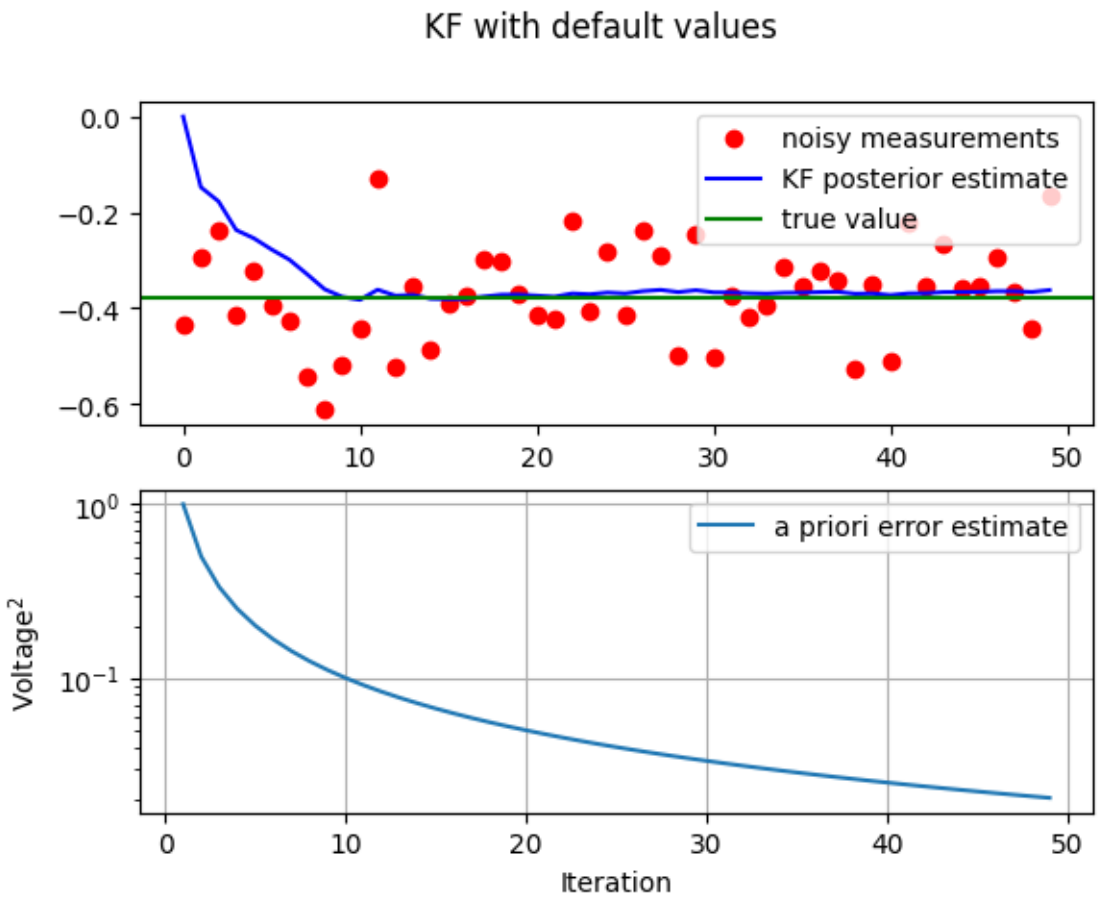
In this first simulation we fixed the measurement variance at $R = (0.1)^2 = 0.01$. Because this is the "true" measurementerror variance, we would expect the "best" performance in terms of balancing responsiveness and estimate variance. This will become more evident in the second and third simulations.

The above figure depicts the results of this first simulation. The true value of the random constant $x = -0.37727$ is given by the solid line, the noisy measurements by the dots and the filter estimate by the blue curve.

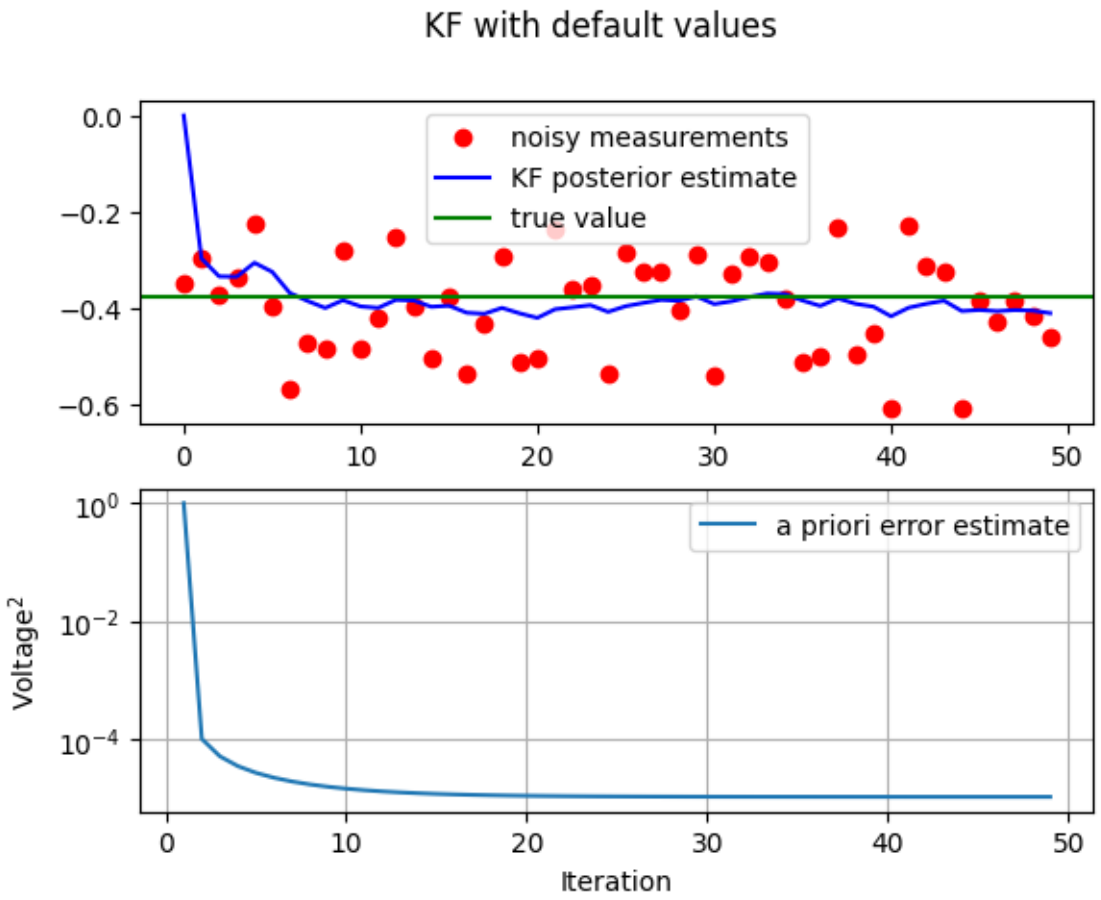
Now, we will see what happens when the measurement error variance R is increased or decreased by a factor of 100 respectively.

- first, the filter was told that the measurement variance was 100 times greater (i.e. $R = 1$) so it was "slower" to believe the measurements.
- then, the filter was told that the measurement variance was 100 times smaller (i.e. $R = 0.0001$) so it was very "quick" to believe the noisy measurements.

```
# increase R = 1
KF_ct(sig_v=1)
```




```
# decrease R = 0.0001
KF_ct(sig_v=0.01)
```



4.1 Conclusion

While the estimation of a constant is relatively straightforward, this example clearly demonstrates the workings of the Kalman filter. In the second Figure (R=1) in particular, the Kalman filtering is evident as the estimate appears considerably smoother than the noisy measurements. We observe the speed of convergence of the variance in the bottom subplot of the respective Figures.

5 Example 2 - scalar, Gaussian random walk

Suppose that we have measurements of the scalar y_k from the Gaussian random walk model

$$x_k = x_{k-1} + w_{k-1}, \qquad w_{k-1} \sim \mathcal{N}(0, Q), \tag{5.1}$$
$$y_k = x_k + v_k, \qquad v_k \sim \mathcal{N}(0, R). \tag{5.2}$$

This very basic system is found in many applications where x_k represents a slowly varying quantity that we measure directly. The process noise, w_k, takes into account fluctuations in the state x_k. The measurement noise, v_k, accounts for measurement instrument errors. The difference with the previous example, is that here x varies randomly over the time-steps.

We want to estimate the state x_k over time, taking into account the measurements y_k. That is, we would like to compute the filtering density,

$$p(x_k \mid y_{1:k}) = \mathcal{N}\left(x_k \mid m_k, P_k\right).$$

We proceed by simply writing down the three stages of the Kalman filter, noting that (as in the previous example) $F_k = 1$ and $H_k = 1$ for this model. We obtain:

- **Initialization:** Define the prior mean x_0 and prior covariance P_0.
- **Prediction:**

$$\hat{x}_k = x_{k-1},$$
$$\hat{P}_k = P_{k-1} + Q.$$

- **Correction:** Define

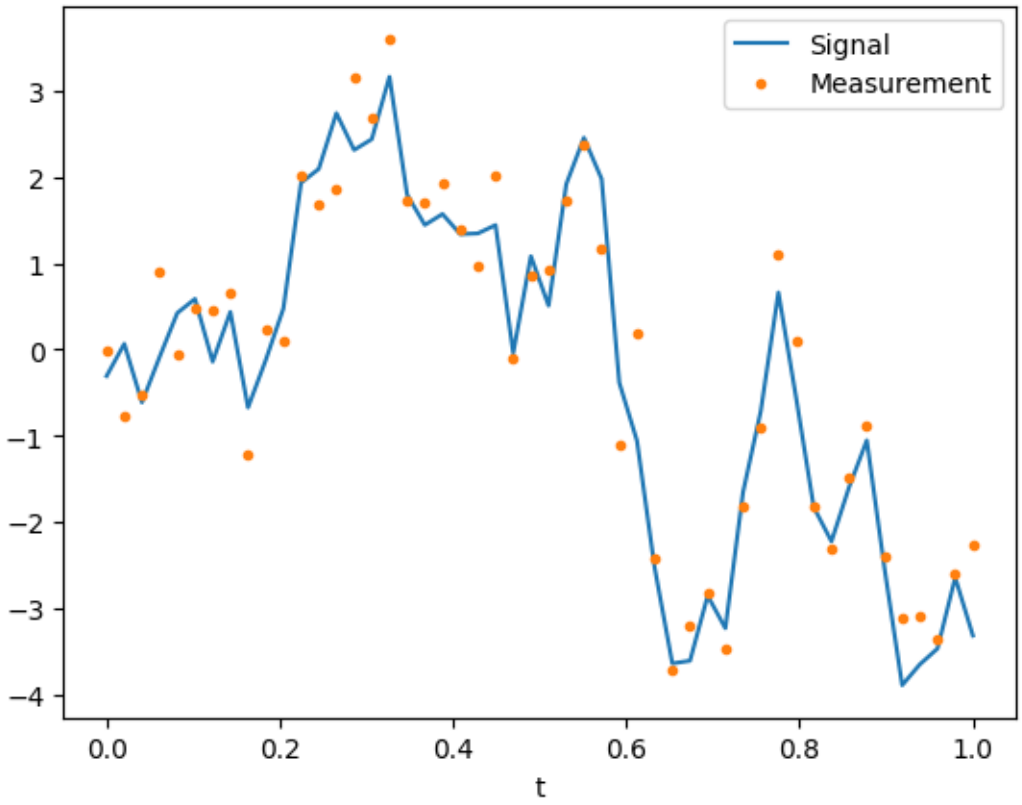
$$d_k = y_k - \hat{x}_k, \quad \text{the innovation},$$
$$S_k = \hat{P}_k + R, \quad \text{the measurement covariance},$$
$$K_k = \hat{P}_k S_k^{-1}, \quad \text{the Kalman gain},$$

  then update,

$$x_k = \hat{x}_k + K_k d_k,$$
$$P_k = \hat{P}_k - \frac{\hat{P}_k^2}{S_k}.$$

> **Note**
>
> Note that this formulation is identical with the previous example, except the nature of the measurements, y_k.

```
# generate the GRW
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(1955)
# time interval and time-step
T = 1
N = 50
dt = T/N
t = np.linspace(0,1,N)
# set parameters
sig_w = 1
sig_v = 0.5
F = 1
Q = sig_w**2
H = 1
R = sig_v**2
# initialize
x0 = 0
P0 = 1
# simulate data
X = np.zeros(N)
Y = np.zeros(N)
x = x0
# loop over time
for j in range(N):
    w = sig_w*np.random.randn()
    x = F*x + w;
    y = H*x + sig_v*np.random.randn()
    X[j] = x
    Y[j] = y
# plot the GRW
plt.plot(t, X, t, Y, '.')
plt.legend(['Signal', 'Measurement'])
plt.xlabel('t')
plt.show()
```



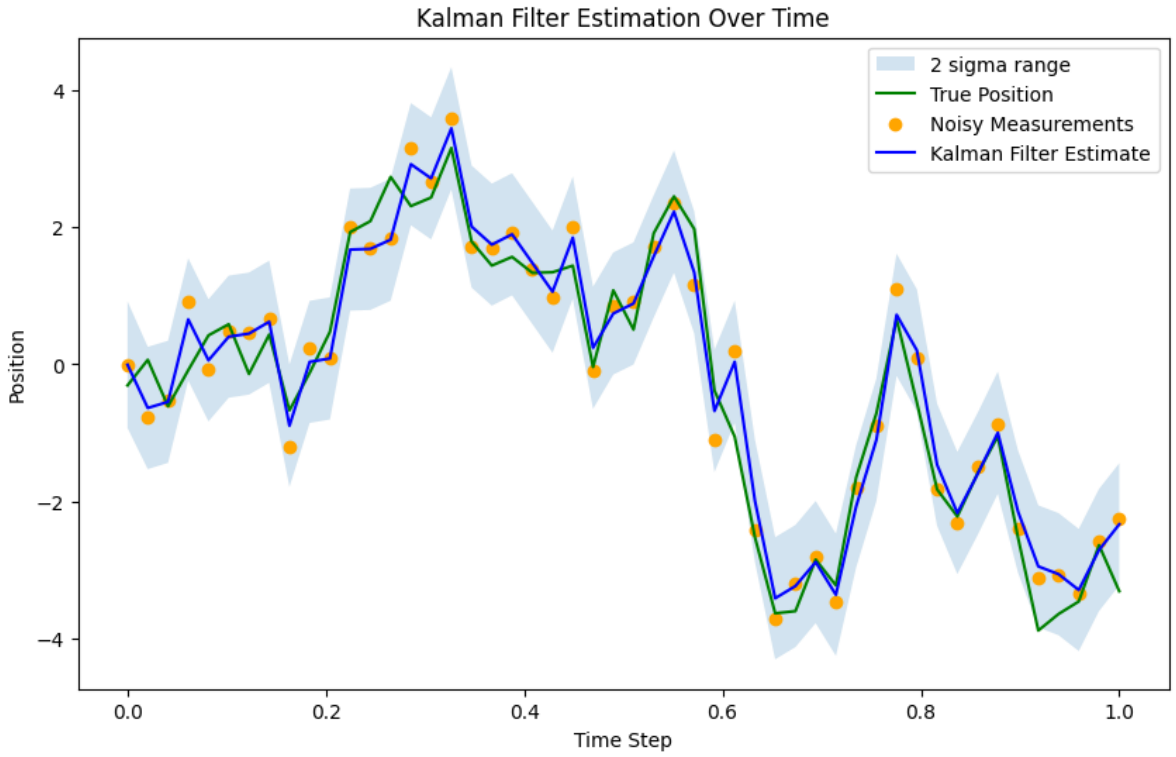
5.1 Implementation of the KF

Here is a straigthforward, matrix-based implementation of the KF that follows exactly the thoeretical formulation above.

For more generality, below we will rewrite the Kalman filter as a class.

```
# scalar KF
x = x0
P = P0
# Allocate space for estimated position
estimated_positions  = np.zeros(N)
estimated_covariance = np.zeros(N)
# Kalman Filter Loop
for k in range(N):
    # Predict
    x = F*x
    P = F*P*F + Q
```

```
    # Correct
    d = Y[k] - H*x
    S = H*P*H + R
    K = P*H / S
    x = x + K*d
    P = P - K*S*K
    # Store the filtered position and covariance
    estimated_positions[k]  = x
    estimated_covariance[k] = P

# Plot the true positions, noisy measurements, and the Kalman filter estimates
# and the 2 sigma upper and lower analytic population bounds
lower_bound = estimated_positions - 1.96*np.sqrt(estimated_covariance)
upper_bound = estimated_positions + 1.96*np.sqrt(estimated_covariance)

fig, ax = plt.subplots(1, figsize=(10,6))
ax.fill_between(t, lower_bound, upper_bound, facecolor='C0', alpha=0.2,
                label='2 sigma range')
ax.legend(loc='upper left')
ax.plot(t, X, label='True Position', color='green')
ax.scatter(t, Y, label='Noisy Measurements', color='orange', marker='o')
ax.plot(t, estimated_positions, label='Kalman Filter Estimate', color='blue')

plt.xlabel('Time Step')
plt.ylabel('Position')
plt.title('Kalman Filter Estimation Over Time')
plt.legend()
plt.show()
```



5.2 Implementation of KF as a class

For more generality, we rewrite the Kalman filter as a class.

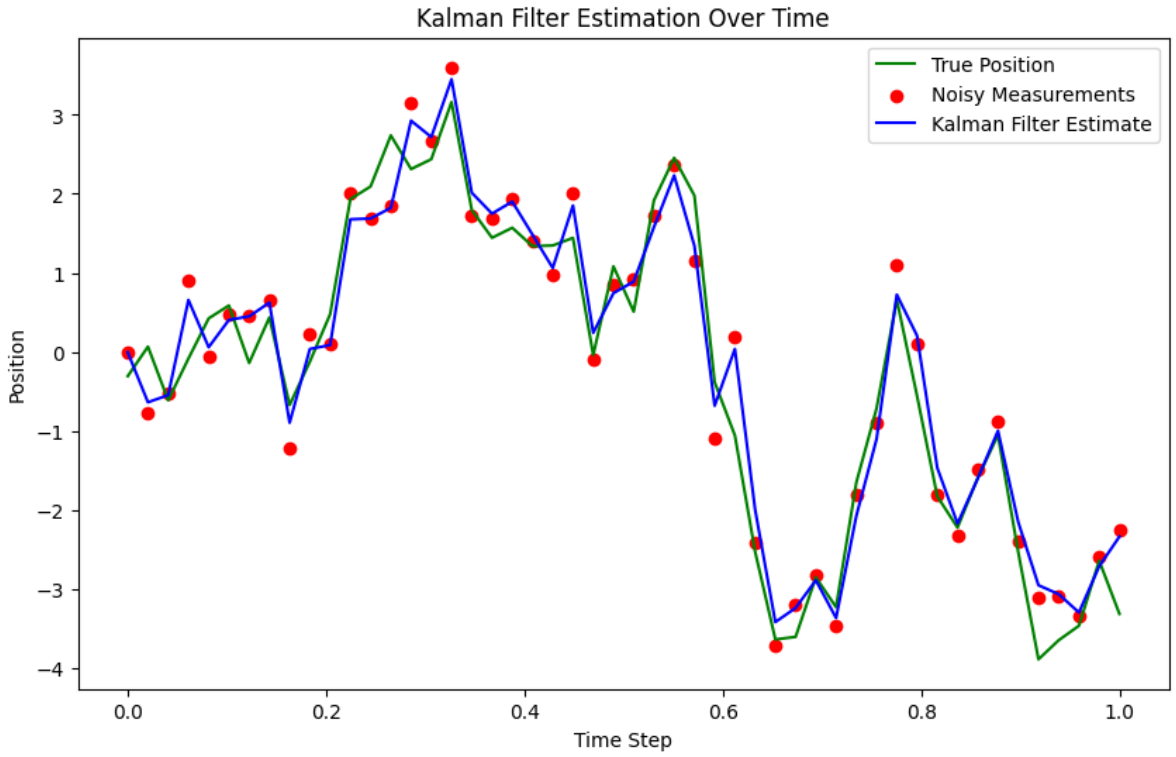
```
class KalmanFilter:
    """
    An implementation of the classic Kalman Filter for a SCALAR linear dynamic systems.
    The Kalman Filter is an optimal recursive data processing algorithm which
    aims to estimate the state of a system from noisy observations.

    Attributes:
        F (np.ndarray): The state transition matrix.
        B (np.ndarray): The control input marix.
        H (np.ndarray): The observation matrix.
        u (np.ndarray): the control input.
        Q (np.ndarray): The process noise covariance matrix.
        R (np.ndarray): The measurement noise covariance matrix.
        x (np.ndarray): The mean state estimate of the previous step (k-1).
        P (np.ndarray): The state covariance of previous step (k-1).
```

```
    """
    def __init__(self, F=None, B=None, H=None, Q=None, R=None, x0=None, P0=None):
        """
        Initializes the Kalman Filter with the necessary matrices and initial state.

        Parameters:
            F (np.ndarray): The state transition matrix.
            B (np.ndarray): The control input marix.
            H (np.ndarray): The observation matrix.
            u (np.ndarray): the control input.
            Q (np.ndarray): The process noise covariance matrix.
            R (np.ndarray): The measurement noise covariance matrix.
            x0 (np.ndarray): The initial state estimate.
            P0 (np.ndarray): The initial state covariance matrix.
        """
        self.F = F  # State transition matrix
        self.B = B  # Control input matrix
        self.u = u  # Control input
        self.H = H  # Observation matrix
        self.Q = Q  # Process noise covariance
        self.R = R  # Measurement noise covariance
        self.x = x0  # Initial state estimate
        self.P = P0  # Initial estimate covariance

    def predict(self):
        """
        Predicts the state and the state covariance for the next time step.
        """
        self.x = self.F @ self.x + self.B @ self.u
        self.P = self.F @ self.P @ self.F.T + self.Q
        #return self.x

    def update(self, z):
        """
        Updates the state estimate with the latest measurement.

        Parameters:
            z (np.ndarray): The measurement at the current step.
        """
        y = z - self.H @ self.x
        S = self.H @ self.P @ self.H.T + self.R
        K = self.P @ self.H.T @ np.linalg.inv(S)
```

```
        self.x = self.x + K @ y
        I = np.eye(self.P.shape[0])
        self.P = (I - K @ self.H) @ self.P
```

```
# generate the GRW
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(1955)
# time interval and time-step
T = 1
N = 50
dt = T/N
t = np.linspace(0,1,N)
# set parameters
sig_w = 1
sig_v = 0.5
F = 1
Q = sig_w**2
H = 1
R = sig_v**2
# initialize
x0 = 0
P0 = 1
# simulate data
X = np.zeros(N)
Y = np.zeros(N)
x = x0
# loop over time
for j in range(N):
    w = Q*np.random.randn()
    x = F*x + w;
    y = H*x + sig_v*np.random.randn()
    X[j] = x
    Y[j] = y# ready to execute the KF...

# Kalman Filter Initialization
F = np.array([[1]])        # State transition matrix
B = np.array([[0]])        # No control input
u = np.array([[0]])        # No control input
H = np.array([[1]])        # Measurement function
Q = np.array([[sig_w**2]]) # Process noise covariance
R = np.array([[sig_v**2]]) # Measurement noise covariance
```

```
x0 = np.array([[0]])       # Initial state estimate
P0 = np.array([[1]])       # Initial estimate covariance

kf = KalmanFilter(F, B, H, Q, R, x0, P0)

# Allocate space for estimated position
estimated_positions = np.zeros(N)

# Kalman Filter Loop
for k in range(N):
    # Predict
    kf.predict()
    # Correct
    measurement = np.array([[Y[k]]])
    kf.update(measurement)
    # Store the filtered position
    estimated_positions[k]  = np.ndarray.item(kf.x[0])

# Plot the true positions, noisy measurements, and the Kalman filter estimates
plt.figure(figsize=(10, 6))
plt.plot(t, X, label='True Position', color='green')
plt.scatter(t, Y, label='Noisy Measurements', color='red', marker='o')
plt.plot(t, estimated_positions, label='Kalman Filter Estimate', color='blue')

plt.xlabel('Time Step')
plt.ylabel('Position')
plt.title('Kalman Filter Estimation Over Time')
plt.legend()
plt.show()
```


Kalman Filter Estimation Over Time
True Position
Noisy Measurements
Kalman Filter Estimate
Position
Time Step
3
2
1
0
−1
−2
−3
−4
0.0
0.2
0.4
0.6
0.8
1.0

6 Example 3 - constant-velocity 1D position tracking

Kalman filters are extensively used in navigation and other GPS-based systems. These can be either linear or nonlinear. In the latter case, one usually resorts to the extended Kalman, filter (EKF) that is designed to deal better with nonlinear state equations—see below.

Process noise design for Q

- diagonal
- constant velocity model
- constant acceleration model

We begin with a very simple case of tracking a 1D position due to constant velocity motion, described by

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v, \quad x(0) = x_0.$$

This equation can be written as a system of two equations

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = F\mathbf{x} + L\mathbf{w}, \quad \mathbf{x}(0) = \mathbf{x}_0,$$

where

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}, \quad F = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{w} = w.$$

In discrete-time, we introduce a time-step, Δt, that is sometimes just set equal to one, and obtain the discrete, state-space formulation

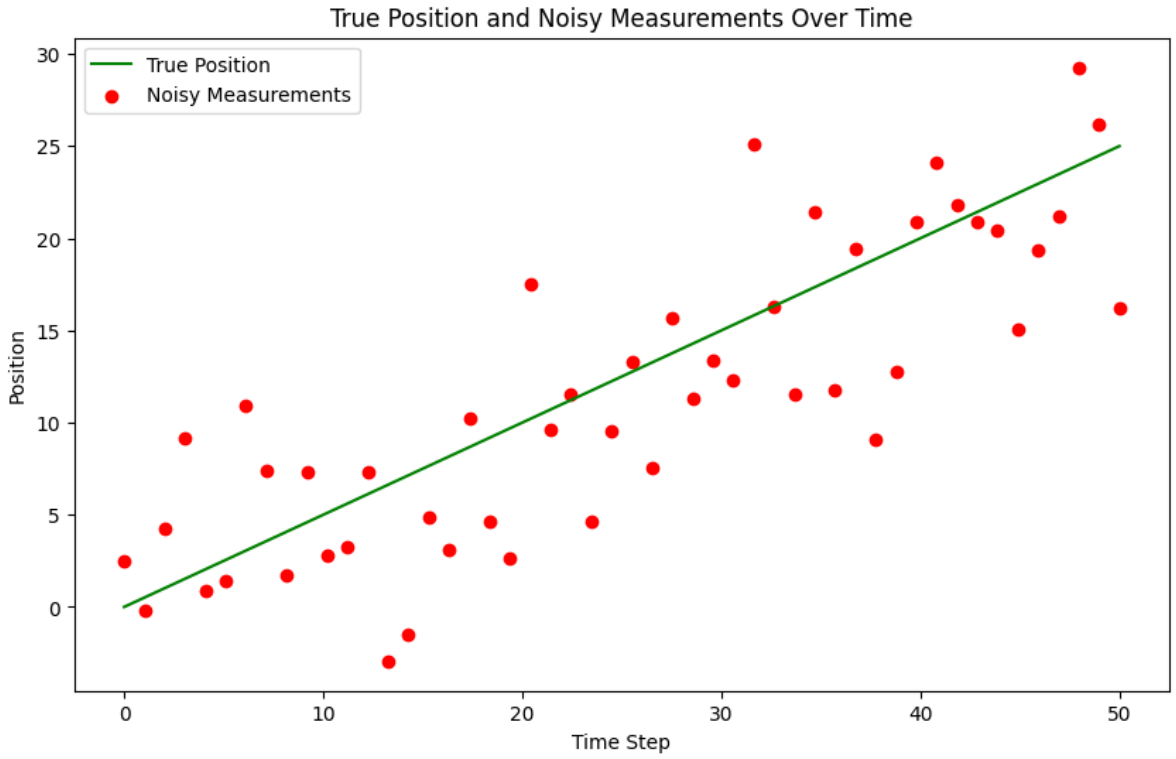
$$\mathbf{x}_{k+1} = F\mathbf{x}_k + Lw_k,$$

where

$$F = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}, \quad L = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

We begin by simulating the motion and generating noisy measurements of the position.

```
import numpy as np
import matplotlib.pyplot as plt
# Set the random seed for reproducibility
np.random.seed(42)

# Simulate the ground truth position of the object
true_velocity = 0.5  # units per time step
num_steps = 50
time_steps = np.linspace(0, num_steps, num_steps)
true_positions = true_velocity * time_steps

# Simulate the measurements with noise
measurement_noise = 5 #1#0  # increase this value to make measurements noisier
noisy_measurements = true_positions + np.random.normal(0, measurement_noise, num_steps)

# Plot the true positions and the noisy measurements
plt.figure(figsize=(10, 6))
plt.plot(time_steps, true_positions, label='True Position', color='green')
plt.scatter(time_steps, noisy_measurements, label='Noisy Measurements', color='red', marker=

plt.xlabel('Time Step')
plt.ylabel('Position')
plt.title('True Position and Noisy Measurements Over Time')
plt.legend()
plt.show()
```



Now, we set up the inputs for the Kalman filter, run it, and plot the results of the estimation.

The process noise covariance matrix, Q, is taken here as

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & \sigma_w^2 \end{bmatrix},$$

where we have supposed that

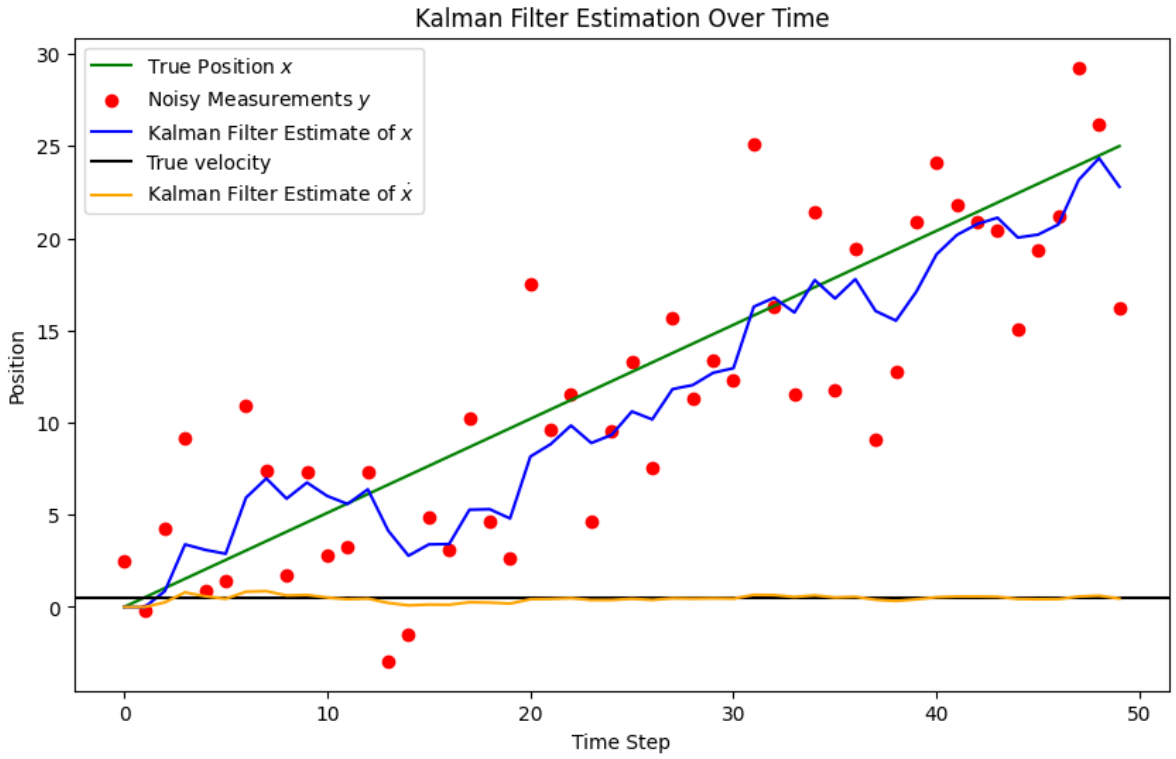
$$w \sim \mathcal{N}(0, \sigma_w^2).$$

This form for Q can be derived from the discretization of a Wiener process, as shown in {cite}Sarkka2023. We will consider a more general case in Example 4, below.

```
sig_v = measurement_noise
sig_w = 0.1 # np.sqrt(3)  # process noise
# Kalman Filter Initialization
F = np.array([[1, 1], [0, 1]])        # State transition matrix
H = np.array([[1, 0]])                # Measurement function
Q = np.array([[1, 0], [0, sig_w**2]]) # Process noise covariance
```

```
R = np.array([[sig_v**2]])            # Measurement noise covariance
x0 = np.array([[0], [0]])             # Initial state estimate
P0 = np.array([[1, 0], [0, 1]])       # Initial estimate covariance

# Initialize the state estimate and covariance
nd = 2
T = 50

x_hat = np.zeros((T, nd, 1))
P     = np.zeros((T, nd, nd))

x_hat[0] = x0
P[0] = P0 #Q
y = noisy_measurements

# Run the Kalman filter
for t in range(1, T):
    # Prediction step
    x_hat[t] = F @ x_hat[t-1]
    P[t] = F @ P[t-1] @ F.T + Q

    # Update step
    K = P[t] @ H.T @ np.linalg.inv(H @ P[t] @ H.T + R)
    x_hat[t] = x_hat[t] + K @ (y[t] - H @ x_hat[t])
    P[t] = (np.eye(nd) - K @ H) @ P[t]
```

```
# Plot the true positions, noisy measurements, and the Kalman filter estimates
time_steps = range(T)
estimated_positions  = x_hat[:,0,:]
estimated_velocities = x_hat[:,1,:]
plt.figure(figsize=(10, 6))
plt.plot(time_steps, true_positions, label='True Position $x$', color='green')
plt.scatter(time_steps, noisy_measurements, label='Noisy Measurements $y$', color='red', mark
plt.plot(time_steps, estimated_positions, label='Kalman Filter Estimate of $x$', color='blue
plt.axhline(0.5,color='k',label='True velocity')
plt.plot(time_steps, estimated_velocities, label='Kalman Filter Estimate of $\dot{x}$', color
plt.xlabel('Time Step')
plt.ylabel('Position')
plt.title('Kalman Filter Estimation Over Time')
plt.legend()
plt.show()
```


Kalman Filter Estimation Over Time
True Position x
Noisy Measurements y
Kalman Filter Estimate of x
True velocity
Kalman Filter Estimate of $\dot{x}$
Position
Time Step
30
25
20
15
10
5
0
0
10
20
30
40
50

7 Example 4 - constant-velocity 2D motion tracking

Here, we will follow a series of examples taken from [4] for tracking vehicle motion in 2D.

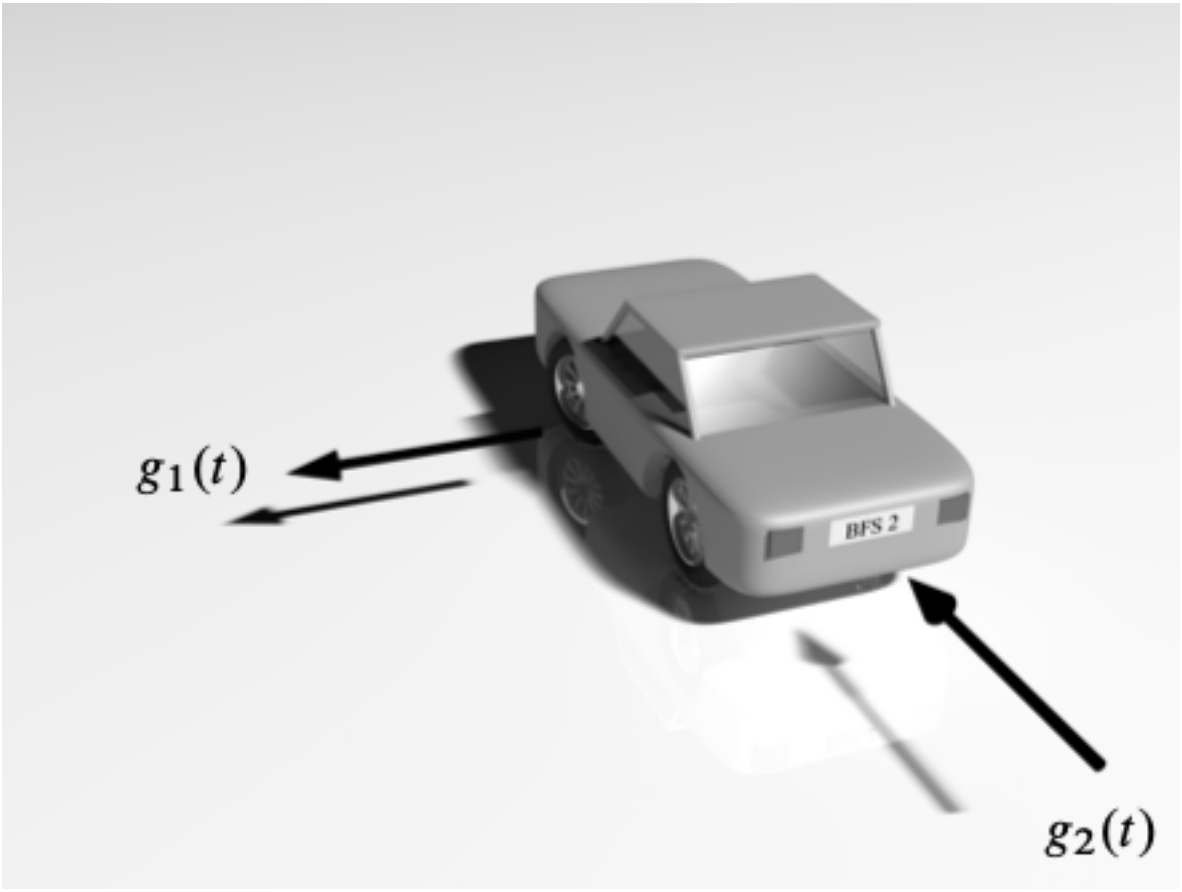


Figure 7.1: 2D motion

Assume that a car is represented by a point mass at its centre (x_1, x_2), and moves in the x_1-x_2 plane subject to noisy forces $g_1(t)$, $g_2(t)$. Then, according to Newton's law of motion, $F = ma$, we can simply write $g_i(t) = ma_i(t)$, where $a_i = \ddot{x}_i$. Now, since $g_i(t)$ are badly known,

or unknown, we can model $g_i(t)/m$ as random processes, and more precisely as independent, white noise (Wiener) processes. We thus obtain the 2nd order system

$$\ddot{x}_1 = w_1(t), \tag{7.1}$$
$$\ddot{x}_2 = w_2(t), \tag{7.2}$$

where $w_i \sim \mathcal{N}(0, \sigma_{w_i}^2)$. Defining the velocities, $x_3 = \dot{x}_1$, and $x_4 = \dot{x}_2$, we can write this as a first order system,

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = F\mathbf{x} + L\mathbf{w}, \quad \mathbf{x}(0) = \mathbf{x}_0,$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}, \quad F = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}.$$

More generally, we can consider a d-dimensional ($d = 1, 2, 3$), continuous-time, constant-velocity model for the motion of an object in space,

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \begin{bmatrix} 0 & I \\ 0 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ I \end{bmatrix} \mathbf{w},$$

which just denotes that the derivaitve of the position is the velocity, and that the derivative of the velocity is the process noise. We can now proceed to discretizing this system, as derived in [4], to obtain

$$\mathbf{x}_{k+1} = F\mathbf{x}_k + \mathbf{q}_k,$$

where

$$F = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

and $\mathbf{q}_k$ is a discrete-time, Gaussian noise process with mean zero and covariance

$$Q = \begin{bmatrix} q_1\Delta t^3/3 & 0 & q_1\Delta t^2/2 & 0 \\ 0 & q_2\Delta t^3/3 & 0 & q_2\Delta t^2/2 \\ q_1\Delta t^2/2 & 0 & q_1\Delta t & 0 \\ 0 & q_2\Delta t^2/2 & 0 & q_2\Delta t \end{bmatrix}.$$

7.1 State-space model

Assuming the above dynamics, and adding a noisy position measurement model, we obtain the linear state-space model

$$\mathbf{x}_{k+1} = F\mathbf{x}_k + \mathbf{q}_k, \quad \mathbf{q}_k \sim \mathcal{N}(0, Q), \tag{7.3}$$
$$\mathbf{y}_{k+1} = H\mathbf{x}_k + \mathbf{r}_k, \quad \mathbf{r}_k \sim \mathcal{N}(0, R), \tag{7.4}$$

with

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \quad R = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}.$$

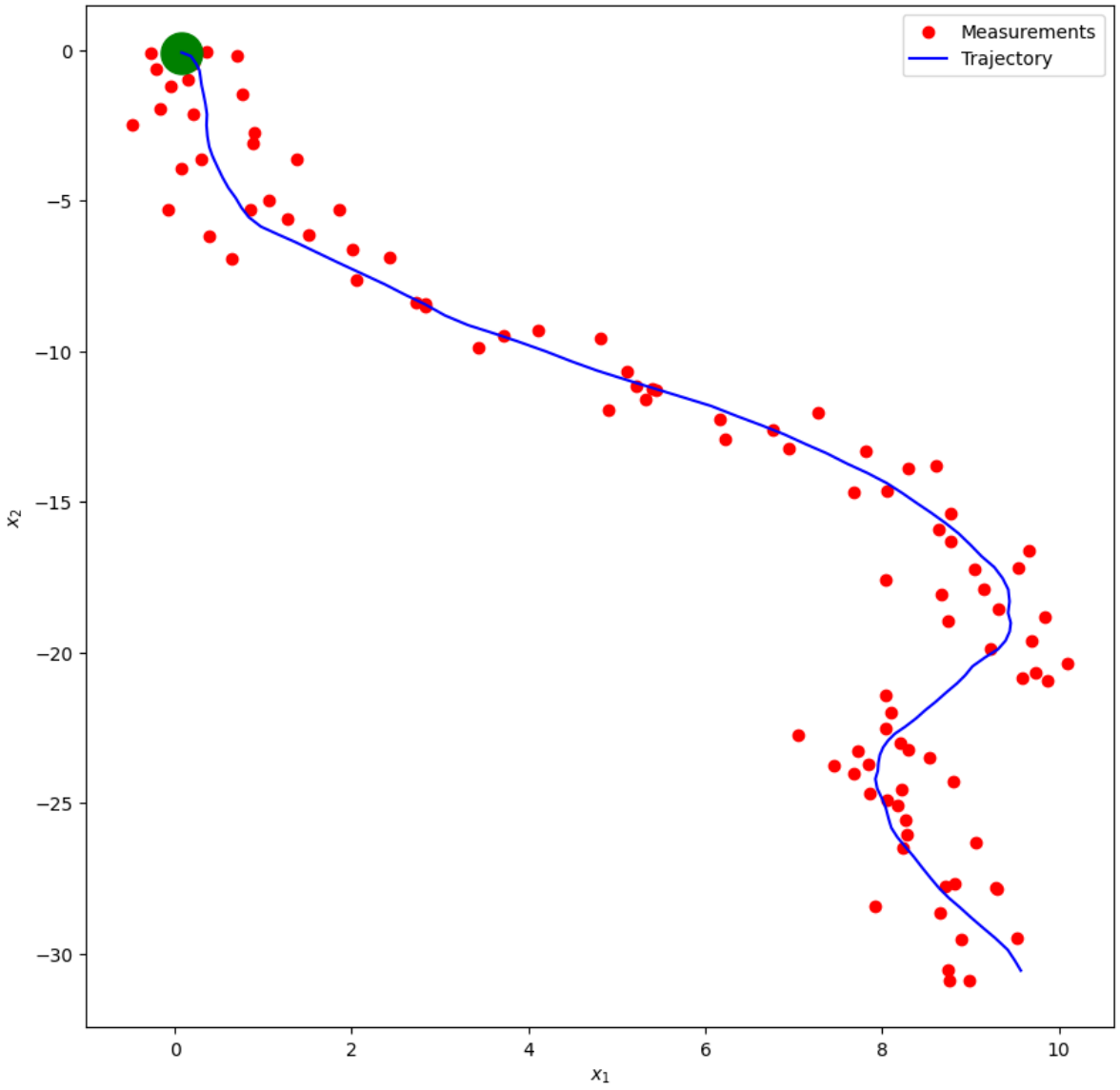
We are now ready to simulate the Kalman filter.

1. definition of all parameters
2. initilization of all matrices
3. simulation and generate noisy measurements - plot
4. Kalman filter
5. Kalman smoother

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import linalg
# we use Sarkka's utilities to streamline a bit...
from common_utilities import generate_ssm, RandomState, rmse, plot_car_trajectory
# initialize
q = 1.    # process noise
dt = 0.1  # time step
s = 0.5   # measurement noise

M = 4  # State dimension
N = 2  # Observation dimension
```

```
A = np.array([[1, 0, dt, 0],
              [0, 1, 0, dt],
              [0, 0, 1, 0],
              [0, 0, 0, 1]])

Q = q * np.array([[dt ** 3 / 3, 0, dt ** 2 / 2, 0],
                  [0, dt **3 / 3, 0, dt ** 2 / 2],
                  [dt ** 2 / 2, 0, dt, 0],
                  [0, dt ** 2 / 2, 0, dt]])

H = np.array([[1, 0, 0, 0],
              [0, 1, 0, 0]])

R = np.array([[s ** 2, 0],
              [0, s ** 2]])

x_0 = np.array([0., 0., 1., -1.])
```

```
# Simulate trajectory and noisy measurements
random_state = RandomState(6)
steps = 100

states, observations = generate_ssm(x_0, A, Q, H, R, steps, random_state)

plot_car_trajectory(observations, states, "Trajectory")
```



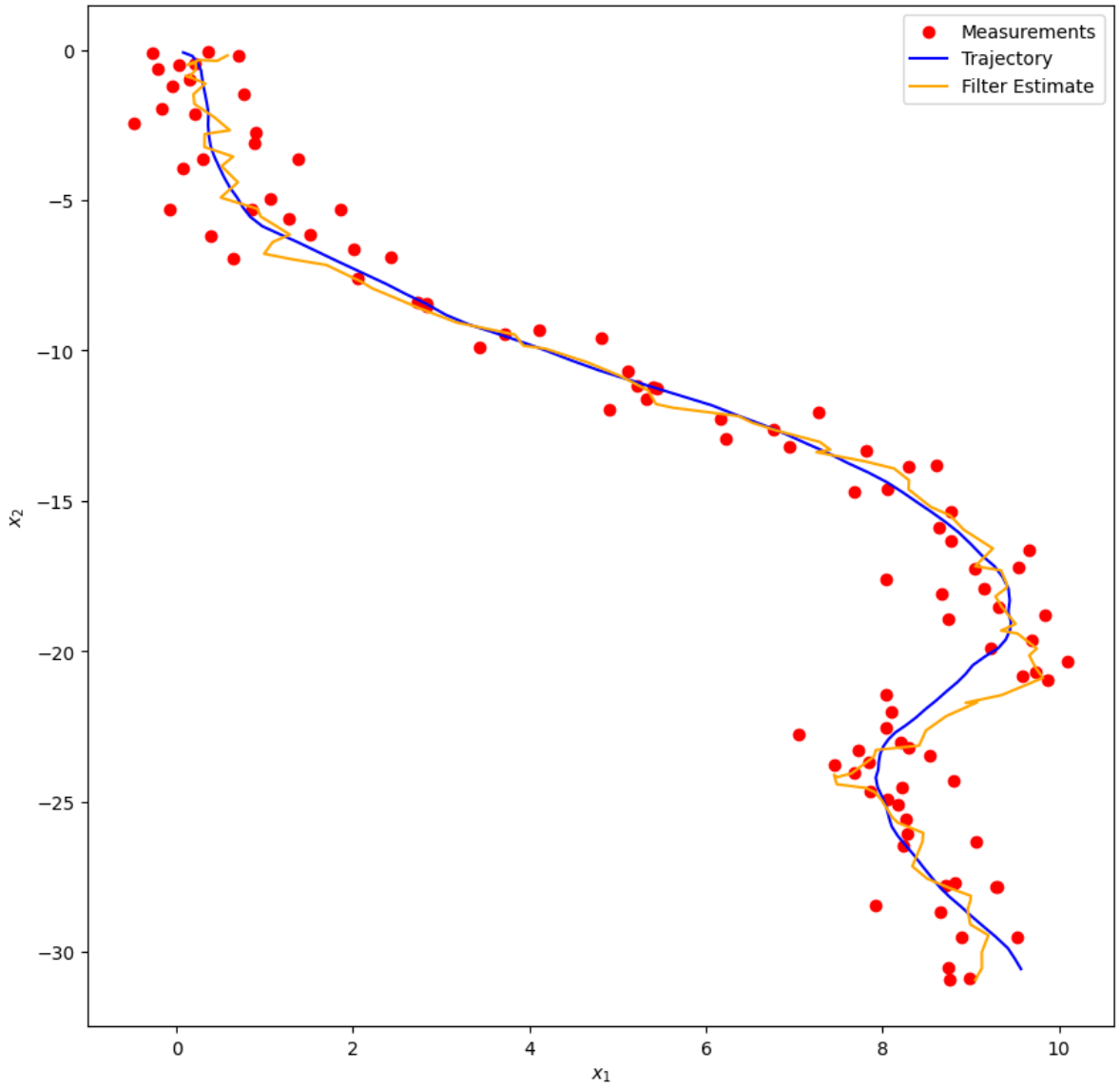
7.2 Kalman filter

```
def kalman_filter(m_0, P_0, A, Q, H, R, observations):
    M = m_0.shape[-1]
    steps, N = observations.shape

    kf_m = np.empty((steps, M))
```

```
    kf_P = np.empty((steps, M, M))

    m = m_0
    P = P_0

    for i in range(steps):
        y = observations[i]
        m = A @ m
        P = A @ P @ A.T + Q

        S = H @ P @ H.T + R
        # More efficient and stable way of computing K = P @ H.T @ linalg.inv(S)
        # This also leverages the fact that S is known to be a positive definite matrix (assu
        K = linalg.solve(S.T, H @ P, assume_a="pos").T

        m = m + K @ (y - H @ m)
        P = P - K @ S @ K.T

        kf_m[i] = m
        kf_P[i] = P
    return kf_m, kf_P
```

```
m_0 = x_0
P_0 = np.array([[1, 0, 0, 0],
                [0, 1, 0, 0],
                [0, 0, 1, 0],
                [0, 0, 0, 1]])

kf_m, kf_P = kalman_filter(m_0, P_0, A, Q, H, R, observations)

plot_car_trajectory(observations, states, "Trajectory", kf_m, "Filter Estimate")

rmse_raw = rmse(states[:, :2], observations)
rmse_kf = rmse(kf_m[:, :2], states[:, :2])
print(f"RAW RMSE: {rmse_raw}")
print(f"KF RMSE: {rmse_kf}")
```

```
RAW RMSE: 0.7131995943918173
KF RMSE: 0.3746597043548562
```



7.3 Kalman smoother

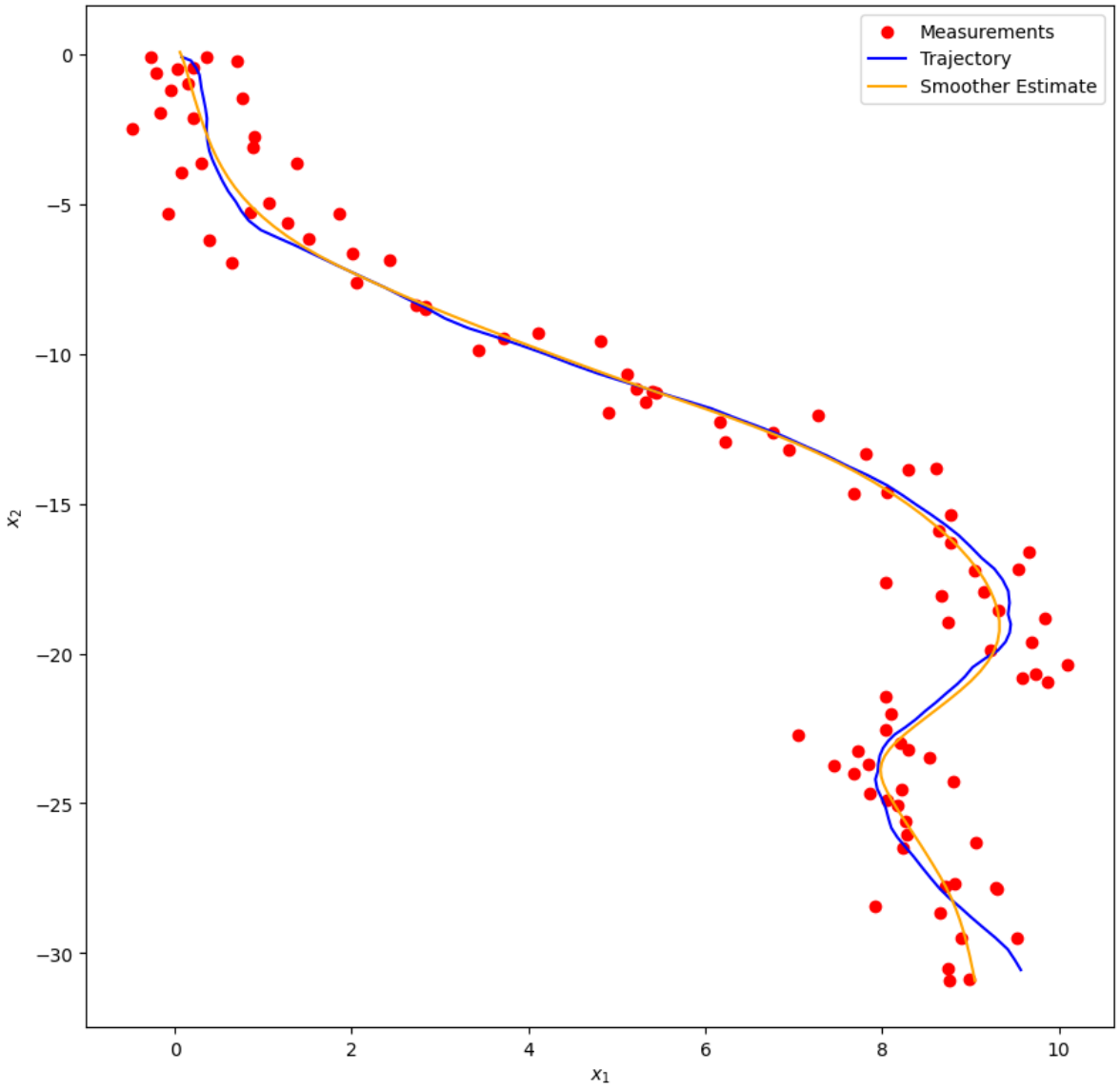
The RTS smoother requires a forward run of the Kalman filter that provides the state and the covariance matrix, for all time steps.

```
def rts_smoother(kf_m, kf_P, A, Q):
    steps, M = kf_m.shape
```

```
    rts_m = np.empty((steps, M))
    rts_P = np.empty((steps, M, M))

    m = kf_m[-1]
    P = kf_P[-1]

    rts_m[-1] = m
    rts_P[-1] = P

    for i in range(steps-2, -1, -1):
        filtered_m = kf_m[i]
        filtered_P = kf_P[i]

        mp = A @ filtered_m
        Pp = A @ filtered_P @ A.T + Q

        # More efficient and stable way of computing Gk = filtered_P @ A.T @ linalg.inv(Pp)
        # This also leverages the fact that Pp is known to be a positive definite matrix (as
        Gk = linalg.solve(Pp, A @ filtered_P, assume_a="pos").T

        m = filtered_m + Gk @ (m - mp)
        P = filtered_P + Gk @ (P - Pp) @ Gk.T

        rts_m[i] = m
        rts_P[i] = P

    return rts_m, rts_P
```

```
rts_m, rts_P = rts_smoother(kf_m, kf_P, A, Q)

plot_car_trajectory(observations, states, "Trajectory", rts_m, "Smoother Estimate")

rmse_rts = rmse(states[:, :2], rts_m[:, :2])

print(f"RAW RMSE: {rmse_raw}")
print(f"KF RMSE: {rmse_kf}")
print(f"RTS RMSE: {rmse_rts}")
```

```
RAW RMSE: 0.7131995943918173
KF RMSE: 0.3746597043548562
RTS RMSE: 0.1857332232186917
```



7.4 Conclusions on Kalman Filters

- workhorse for all linear, Gaussian problems
- cases covered here:
  - track a constant
  - track a random walk
  - movement tracking: scalar consatnt velocity, 2D and 3D tracking

- 2 basic philosophies:
    1. use a KF classs
    2. include KF code each time
    3. use KF module/function
- **Choice:** I prefer (2). Since the KF is coded in only 5 lines, there is no real need for a class and the resulting code remains very readable
- Process noise modelling, to design the matrix Q, is a complex subject. See Saho and references therein.

7.5 References

1. K. Law, A Stuart, K. Zygalakis. *Data Assimilation. A Mathematical Introduction.* Springer. 2015.
2. M. Asch, M. Bocquet, M. Nodet. *Data Assimilation: Methods, Algorithms and Applications.* SIAM. 2016.
3. M. Asch. *A Toobox for Digital Twins. From Model-Based to Data-Driven.* SIAM. 2022
4. S. Sarkka, L. Svensson. *Bayesian Filtering and Smoothing,* 2nd ed., Cambridge University Press. 2023.

Part III

Nonlinear Kalman Filters

8 Nonlinear Kalman Filters

8.1 Introduction

Recall that the Kalman filter can be used for

- state estimation—this is the direct filtering (or smoothing) problem
- parameter estimation—this is the inverse problem based on filtering with a pseudo-time

Kalman filters are linear (and Gaussian) or nonlinear. Here we will formulate the basic nonliner filter, known as the *extended* Kalman filter.

8.2 Recall: Kalman filter problem - general formulation

We present a very general formulation that will later be convenient for joint state and parameter estimation problems. Consider a discrete-time nonlinear dynamical system with noisy state transitions and noisy observations that are also noinlinear.

We consider a discrete-time dynamical system with noisy state transitions and noisy observations.

- Dynamics:

$$v_{j+1} = \Psi(v_j) + \xi_j, \quad j \in \mathbb{Z}^+$$

- Observations:

$$y_{j+1} = h(v_{j+1}) + \eta_{j+1}, \quad j \in \mathbb{Z}^+$$

- Probability (densities):

$$v_0 \sim \mathcal{N}(m_0, C_0), \quad \xi_j \sim \mathcal{N}(0, \Sigma), \quad \eta_j \sim \mathcal{N}(0, \Gamma)$$

- Probability (independence):

$$v_0 \perp \xi_j \perp \eta_j$$

- Operators:

$$\Psi \colon \mathcal{H}_s \mapsto \mathcal{H}_s, \tag{8.1}$$
$$h \colon \mathcal{H}_s \mapsto \mathcal{H}_o, \tag{8.2}$$

where $v_j \in \mathcal{H}_s$, $y_j \in \mathcal{H}_o$ and $\mathcal{H}$ is a finite-dimensional Hilbert space.

Filtering problem:

Estimate (optimally) the state v_j of the dynamical system at time j, given the data $Y_j = \{y_i\}_{i=1}^{j}$ up to time j. This is achieved by using a two-step *predictor-corrector* method. We will use the more general notation of (Law, Stuart, and Zygalakis 2015) instead of the usual, classical state-space formulation that is used in (Asch, Bocquet, and Nodet 2016) and (Asch 2022).

The objective here is to update the filtering distribution $\mathbb{P}(v_j|Y_j)$, from time j to time $j+1$, in the nonlinear, Gaussian case, where

- Ψ and h are nonlinear functions,
- all distributions are Gaussian.

Suppose the Jacobian matrices of Ψ and h exist, and are denoted by

$$\Psi_x(v) = \left[\frac{\partial \Psi}{\partial x}\right]_{x=m}, \tag{8.3}$$
$$h_x(v) = \left[\frac{\partial h}{\partial x}\right]_{x=m}. \tag{8.4}$$

1. Let (m_j, C_j) denote the mean and covariance of $v_j|Y_j$ and note that these entirely characterize the random variable since it is Gaussian.
2. Let $(\hat{m}_{j+1}, \hat{C}_{j+1})$ denote the mean and covariance of $v_{j+1}|Y_j$ and note that these entirely characterize the random variable since it is Gaussian.
3. Derive the map $(m_j, C_j) \mapsto (m_{j+1}, C_{j+1})$ using the previous step.

8.2.1 Prediction/Forecast

$$\mathbb{P}(v_n|y_1,\ldots,y_n) \mapsto \mathbb{P}(v_{n+1}|y_1,\ldots,y_n)$$

- P0: initialize (m_0, C_0) and compute v_0
- P1: predict the state, measurement

$$v_{j+1} = \Psi(v_j) + \xi_j \tag{8.5}$$
$$y_{j+1} = h(v_{j+1}) + \eta_{j+1} \tag{8.6}$$

- P2: predict the mean and covariance

$$\hat{m}_{j+1} = \Psi(m_j) \tag{8.7}$$
$$\hat{C}_{j+1} = \Psi_x C_j \Psi_x^{\mathrm{T}} + \Sigma \tag{8.8}$$

8.2.2 Correction/Analysis

$$\mathbb{P}(v_{n+1}|y_1,\ldots,y_n) \mapsto \mathbb{P}(v_{n+1}|y_1,\ldots,y_{n+1})$$

- C1: compute the innovation

$$d_{j+1} = y_{j+1} - h(\hat{m}_{j+1})$$

- C2: compute the measurement covariance

$$S_{j+1} = h_x \hat{C}_{j+1} h_x^{\mathrm{T}} + \Gamma$$

- C3: compute the (optimal) Kalman gain

$$K_{j+1} = \hat{C}_{j+1} h_x^{\mathrm{T}} S_{j+1}^{-1}$$

- C4: update/correct the mean and covariance

$$m_{j+1} = \hat{m}_{j+1} + K_{j+1} d_{j+1}, \tag{8.9}$$
$$C_{j+1} = \hat{C}_{j+1} - K_{j+1} S_{j+1} K_{j+1}^{\mathrm{T}}. \tag{8.10}$$

8.2.3 Loop over time

- set $j = j + 1$
- go to step P1

8.3 State-space formulation

In classical filter theory, a state space formulation is usually used.

$$x_{k+1} = f(x_k) + w_k \tag{8.11}$$
$$y_{k+1} = h(x_k) + v_k, \tag{8.12}$$

where f and h are nonlinear, differentiable functions with Jacobian matrices D_f and D_h respectively, $w_k \sim \mathcal{N}(0, Q)$, $v_k \sim \mathcal{N}(0, R)$.

The 2-step filter:

8.3.1 Initialization

$$x_0, \quad P_0$$

8.3.2 1. Prediction

$$x^-_{k+1} = f(x_k) \tag{8.13}$$
$$P^-_{k+1} = D_f P_k D_f^{\mathrm{T}} + Q \tag{8.14}$$

8.3.3 2. Correction

$$K_{k+1} = P^-_{k+1} D_h^{\mathrm{T}} (D_h P^-_{k+1} D_h^{\mathrm{T}} + R)^{-1} \quad (= P^-_{k+1} D_h^{\mathrm{T}} S^{-1}) \tag{8.15}$$
$$x_{k+1} = x^-_{k+1} + K_{k+1}(y_{k+1} - h(x^-_{k+1})) \tag{8.16}$$
$$P_{k+1} = (I - K_{k+1} D_h) P^-_{k+1} \quad (= P^-_{k+1} - K_{k+1} S K^{\mathrm{T}}_{k+1}) \tag{8.17}$$

8.3.4 Loop

Set $k = k + 1$ and go to step 1.

8.4 Other nonlinear filters

- unscented Kalman filter
- particle filter

For details, please consult the references.

9 Example 1 - tracking a random sinusoidal signal

Before introducing more general nonlinear dyanmic systems (see the following examples), let us consider a simple case of tracking a random sine signal whose angular velocity and amplitude vary, randomly, over time. We express the nonlinearity through the measurement model, though this could be done with the dynamic model, as in the examples below.

Let the state vector be

$$\mathbf{x}_k = \begin{bmatrix} \theta_k \\ \omega_k \\ a_k \end{bmatrix},$$

where the θ is the angle, ω the angular velocity and a the amplitude of the sine function. The model is then

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = \omega \tag{9.1}$$
$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = w_\omega(t) \tag{9.2}$$
$$\frac{\mathrm{d}a}{\mathrm{d}t} = w_a(t) \tag{9.3}$$

In matrix form, we have

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = F\mathbf{x}(t) + L\mathbf{w}(t), \quad \mathbf{x}(0) = \mathbf{x}_0,$$

where

$$F = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{w}(t) = \begin{bmatrix} w_\omega(t) \\ w_a(t) \end{bmatrix},$$

and the power spectral density of the white noise process $\mathbf{w}(t)$ is

$$Q^c = \begin{bmatrix} q_1 & 1 \\ 0 & q_2 \end{bmatrix}.$$

Using the matrix exponential formulas (see KF-tutorial), we get the discrete system

$$\mathbf{x}_{k+1} = A_k \mathbf{x}_k + \mathbf{q}_k, \tag{9.4}$$
$$y_k = a_k \sin\theta_k + r_k \tag{9.5}$$

where $r_k \sim \mathcal{N}(0, \sigma_r)$ is univariate Gaussian noise,

$$A_k = \begin{bmatrix} 0 & \Delta t & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

and $\mathbf{q}_k \sim \mathcal{N}(0, Q_k)$ with

$$Q_k = \begin{bmatrix} q_1\Delta t^3/3 & q_1\Delta t^2/2 & 0 \\ q_1\Delta t^2/2 & q_1\Delta t & 0 \\ 0 & 0 & q_2\Delta t \end{bmatrix}.$$

Finally, the Jacobian of $h(\mathbf{x}_k) = a_k \sin\theta_k$ is easily seen to be

$$Dh_k = \begin{bmatrix} a_k \cos\theta_k & 0 & \sin\theta_k \end{bmatrix}$$

10 Example 2 - tracking a noisy pendulum

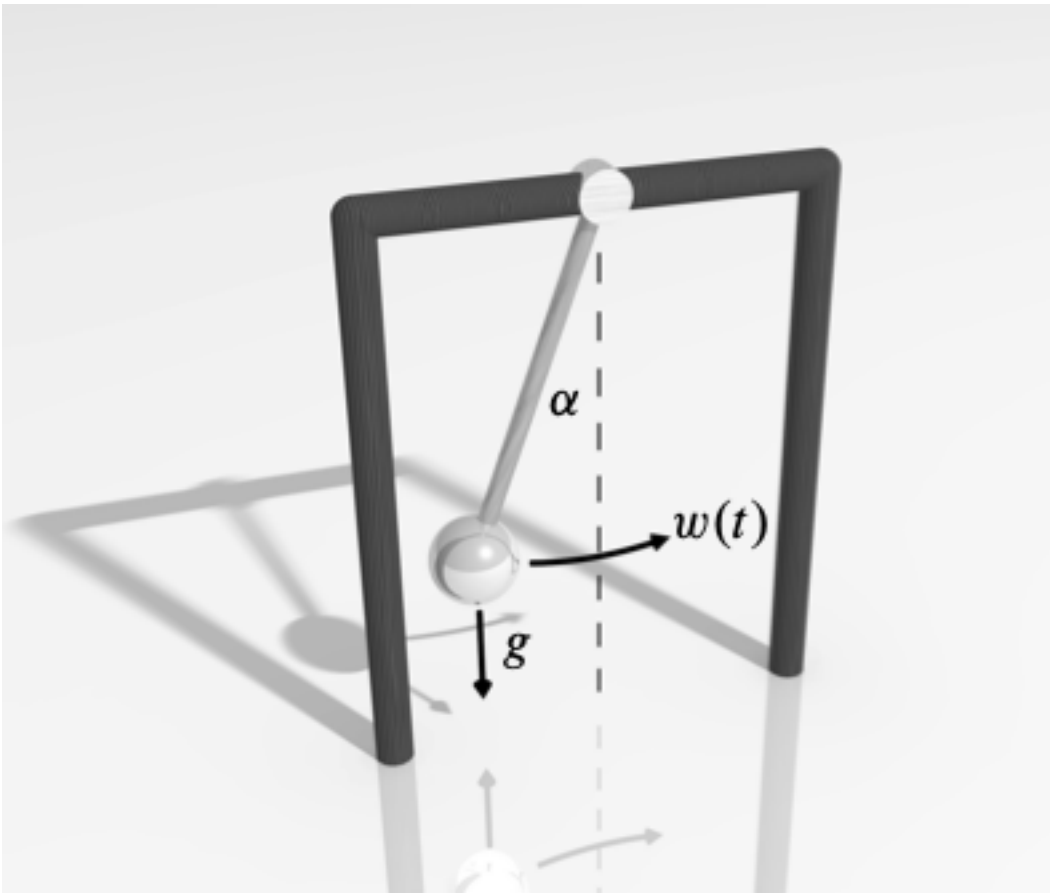


Figure 10.1: Pendulum motion

Consider the nonlinear ODE model for the oscillations of a noisy pendulum with unit mass and length L,

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} + \frac{g}{L}\sin\theta + w(t) = 0$$

where θ is the angular displacement of the pendulum, g is the gravitational constant, L is the pendulum's length, and $w(t)$ is a white noise process. This is rewritten in state space form,

$$\dot{\mathbf{x}} + \mathcal{M}(\mathbf{x}) + \mathbf{w} = 0,$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}, \quad \mathcal{M}(\mathbf{x}) = \begin{bmatrix} x_2 \\ -\dfrac{g}{L}\sin x_1 \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} 0 \\ w(t) \end{bmatrix}.$$

Suppose that we have discrete, noisy measurements of the horizontal component of the position, $\sin(\theta)$. Then the measurement equation is scalar,

$$y_k = \sin\theta_k + v_k,$$

where v_k is a zero-mean Gaussian random variable with variance R. The system is thus nonlinear in state and measurement and the state-space system is of the general form seen above. A simple discretization, based on Euler's method produces

$$\begin{aligned} \mathbf{x}_k &= \mathcal{M}(\mathbf{x}_{k-1}) + \mathbf{w}_{k-1} \\ y_k &= \mathcal{H}_k(\mathbf{x}_k) + v_k, \end{aligned}$$

where

$$\mathbf{x}_k = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_k, \quad \mathcal{M}(\mathbf{x}_{k-1}) = \begin{bmatrix} x_1 + \Delta t x_2 \\ x_2 - \Delta t \dfrac{g}{L}\sin x_1 \end{bmatrix}_{k-1}, \quad \mathcal{H}(\mathbf{x}_k) = [\sin x_1]_k.$$

The noise terms have distributions

$$\mathbf{w}_{k-1} \sim \mathcal{N}(\mathbf{0}, Q), \quad v_k \sim \mathcal{N}(0, R),$$

where the process covariance matrix is

$$Q = \begin{bmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{bmatrix},$$

with components (see KF example for 2D motion tracking),

$$q_{11} = q_c \frac{\Delta t^3}{3}, \quad q_{12} = q_{21} = q_c \frac{\Delta t^2}{2}, \quad q_{22} = q_c \Delta t,$$

and q_c is the continuous process noise spectral density.

For the first-order EKF—higher orders are possible—we will need the Jacobian matrices of $\mathcal{M}(\mathbf{x})$ and $\mathcal{H}(\mathbf{x})$ evaluated at $\mathbf{x} = \hat{\mathbf{m}}_{k-1}$ and $\mathbf{x} = \hat{\mathbf{m}}_k$. These are easily obtained here, in an explicit form,

$$\mathbf{M}_{\mathbf{x}} = \left[\frac{\partial \mathcal{M}}{\partial \mathbf{x}}\right]_{\mathbf{x}=\mathbf{m}} = \begin{bmatrix} 1 & \Delta t \\ -\Delta t \dfrac{g}{L} \cos x_1 & 1 \end{bmatrix}_{k-1},$$

and

$$\mathbf{H}_{\mathbf{x}} = \left[\frac{\partial \mathcal{H}}{\partial \mathbf{x}}\right]_{\mathbf{x}=\mathbf{m}} = \begin{bmatrix} \cos x_1 & 0 \end{bmatrix}_k.$$

Simulations

In the simulations, we take:

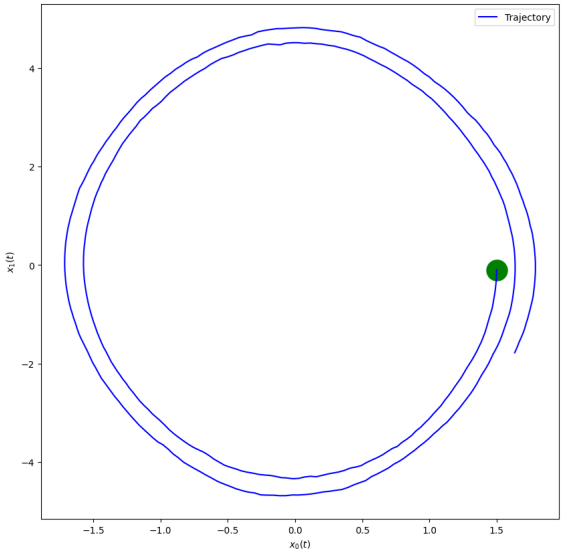
- 500 time steps with $\Delta t = 0.01$.
- Noise levels $q_c = 0.01$ and $R = 0.1$.
- Initial angle $x_1 = 1.8$ and initial angular velocity $x_2 = 0$.
- Initial diagonal state covariance of 0.1.

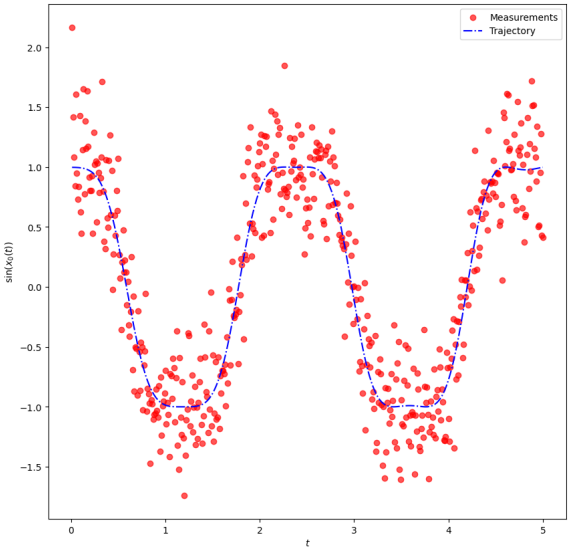
Results are plotted in Figure. We notice that despite the very noisy, nonlinear measurements, the EKF rapidly approaches the true state and then tracks it extremely well.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import linalg, stats
# we use Sarkka's utilities to streamline a bit...
from common_utilities import generate_pendulum, RandomState, rmse, plot_pendulum
# initialize
dt = 0.01  # time step
g  = 9.81  # gravitational acceleration

sig_w = 0.1          # process noise
sig_v = np.sqrt(0.1) # measurement noise

Q  = sig_w**2 * np.array([[dt ** 3 / 3, dt ** 2 / 2],
```

```
                              [dt ** 2 / 2, dt]])
R  = sig_v**2 * np.eye(1)

x_0 = np.array([1.5, 0.])
```

```
# Simulate trajectory and noisy measurements
random_state = RandomState(1)
steps = 500

timeline, states, observations = generate_pendulum(x_0, g, Q, dt, R, steps, random_state)
plot_pendulum(timeline, observations, states, "Trajectory")
```





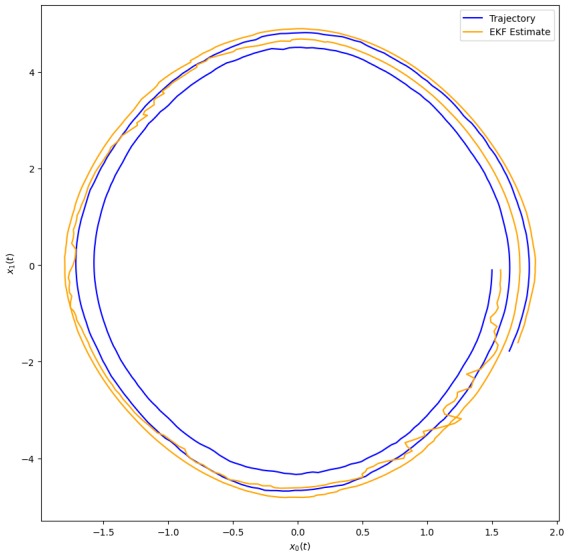
10.1 Extended Kalman Filter (EKF)

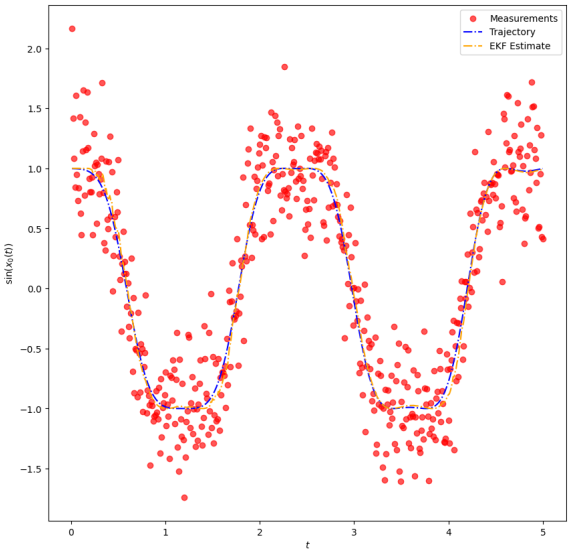
```
def extended_kalman_filter(m_0, P_0, g, Q, dt, R, observations):
    n = m_0.shape[-1]
    steps = observations.shape[0]

    ekf_m = np.empty((steps, n))
    ekf_P = np.empty((steps, n, n))

    m = m_0[:]
```

```
    P = P_0[:]

    for i in range(steps):
        y = observations[i]

        # Jacobian of the dynamic model function
        Df = np.array([[1.,                      dt],
                       [-g * dt * np.cos(m[0]), 1.]])

        # Predicted state distribution
        m = np.array([m[0] + dt * m[1],
                      m[1] - g * dt * np.sin(m[0])])
        P = Df @ P @ Df.T + Q

        # Predicted observation
        h  = np.sin(m[0])
        Dh = np.array([[np.cos(m[0]), 0.]])
        S = Dh @ P @ Dh.T + R

        # Kalman Gain
        K = linalg.solve(S, Dh @ P, assume_a="pos").T
        m = m + K @ np.atleast_1d(y - h)
        P = P - K @ S @ K.T

        ekf_m[i] = m
        ekf_P[i] = P
    return ekf_m, ekf_P
```

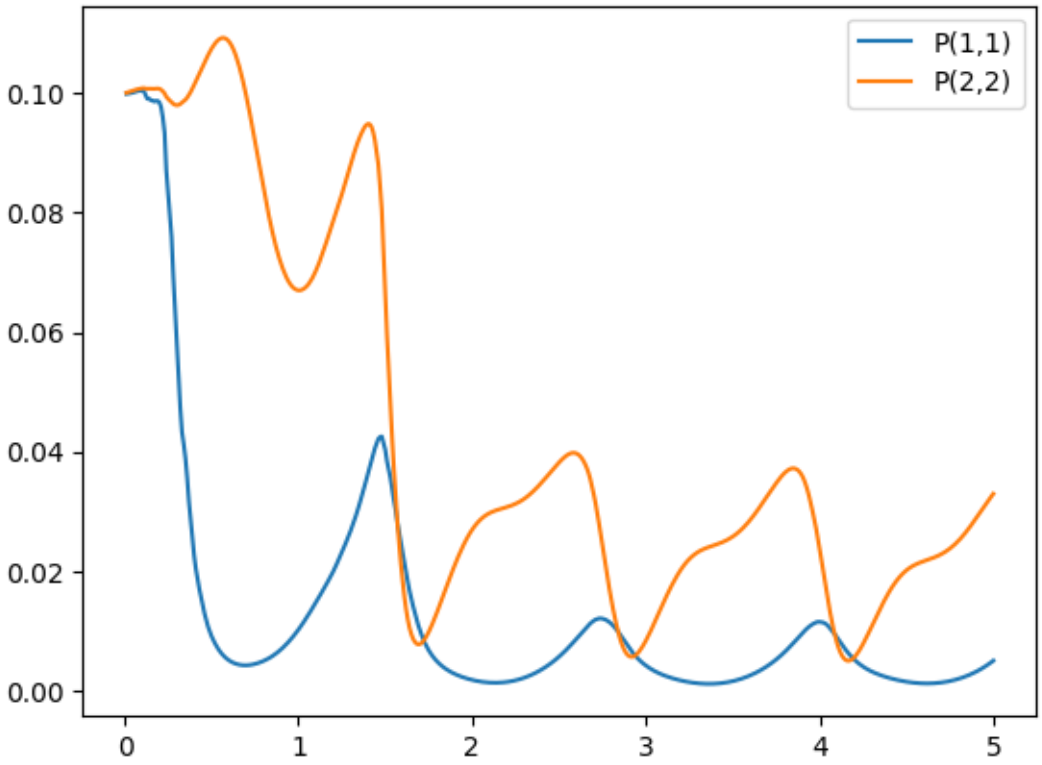
```
# initialize state and covariance
m_0 = np.array([1.6, 0.])  # Slightly off
P_0 = np.array([[0.1, 0.],
                [0., 0.1]])

ekf_m, ekf_P = extended_kalman_filter(m_0, P_0, g, Q, dt, R, observations)
plot_pendulum(timeline, observations, states, "Trajectory", ekf_m, "EKF Estimate")

rmse_ekf = rmse(ekf_m[:, :1], states[:, :1])
print(f"EKF RMSE: {rmse_ekf}")
```

```
EKF RMSE: 0.10306106181239276
```






```
# plot covariances
plt.plot(timeline, ekf_P[:,0,0], timeline,ekf_P[:,1,1] )
plt.legend(['P(1,1)', 'P(2,2)'])
plt.show()
```



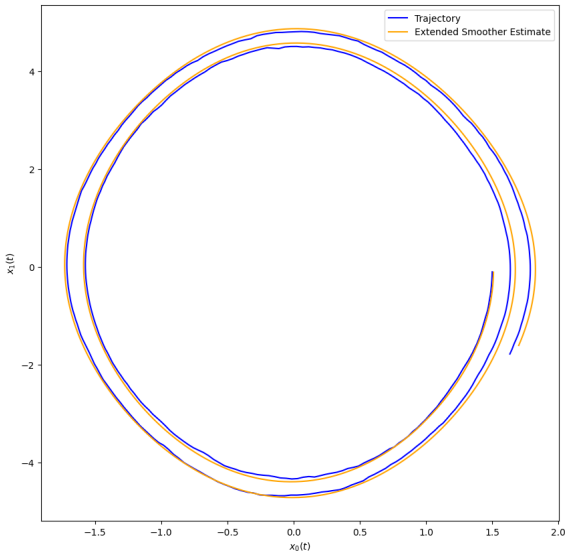
10.2 Extended Smoother

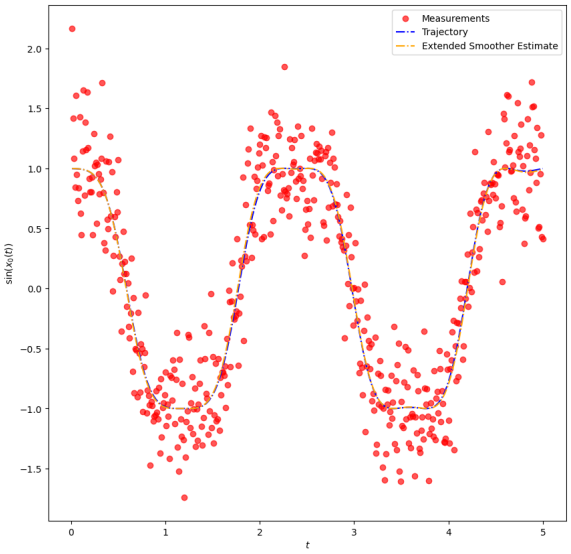
```
def extended_smoother(ekf_m, ekf_P, g, Q, dt):
    steps, M = ekf_m.shape

    rts_m = np.empty((steps, M))
    rts_P = np.empty((steps, M, M))

    m = ekf_m[-1]
    P = ekf_P[-1]

    rts_m[-1] = m
    rts_P[-1] = P

    for i in range(steps-2, -1, -1):
        filtered_m = ekf_m[i]
        filtered_P = ekf_P[i]
```

```
        Df = np.array([[1., dt],
                       [-g * dt * np.cos(filtered_m[0]), 1.]])

        mp = np.array([filtered_m[0] + dt * filtered_m[1],
                       filtered_m[1] - g * dt * np.sin(filtered_m[0])])
        Pp = Df @ filtered_P @ Df.T + Q

        # More efficient and stable way of computing Gk = filtered_P @ A.T @ linalg.inv(Pp)
        # This also leverages the fact that Pp is known to be a positive definite matrix (ass
        Gk = linalg.solve(Pp, Df @ filtered_P, assume_a="pos").T

        m = filtered_m + Gk @ (m - mp)
        P = filtered_P + Gk @ (P - Pp) @ Gk.T

        rts_m[i] = m
        rts_P[i] = P

    return rts_m, rts_P
```

```
rts_m, rts_P = extended_smoother(ekf_m, ekf_P, g, Q, dt)
plot_pendulum(timeline, observations, states, "Trajectory", rts_m, "Extended Smoother Estimat
rmse_erts = rmse(rts_m[:, :1], states[:, :1])
print(f"KF RMSE: {rmse_ekf}")
print(f"ERTS RMSE: {rmse_erts}")
```

```
KF RMSE: 0.10306106181239276
ERTS RMSE: 0.027612762479911554
```





10.3 Conclusions on Extended Kalman Filters

The pros and cons of the EKF are:

- Pros:
  - Relative simplicity, based on well-known linearization methods.
  - Maintains the simple, elegant, and computationally efficient KF update equations.
  - Good performance for such a simple method.
  - Ability to treat nonlinear process and observation models.
  - Ability to treat both additive and more general nonlinear Gaussian noise.
- Cons:
  - Performance can suffer in presence of strong nonlinearity because of the local validity of the linear approximation (valid for small perturbations around the linear term).
  - Cannot deal with non-Gaussian noise, such as discrete-valued random variables.
  - Requires differentiable process and measurement operators and evaluation of Jacobian matrices, which might be problematic in very high dimensions.

In spite of this, the EKF remains a solid filter and remains the basis of most GPS and navigation systems.

10.4 References

1. K. Law, A Stuart, K. Zygalakis. *Data Assimilation. A Mathematical Introduction.* Springer. 2015.
2. M. Asch, M. Bocquet, M. Nodet. *Data Assimilation: Methods, Algorithms and Applications.* SIAM. 2016.
3. M. Asch. *A Toobox for Digital Twins. From Model-Based to Data-Driven.* SIAM. 2022
4. S. Sarkka, L. Svensson. *Bayesian Filtering and Smoothing,* 2nd ed., Cambridge University Press. 2023.

Part IV

Ensemble Filters

11 Ensemble Kalman Filter

The idea behind the EnKF is

1. permit fully nonlinear process and observation models;
2. avoid gradient calculations for linearization, as in the EKF,
3. by replacing mean and covariance with empirical, ensemble averaging.

The EnKF is obtained by - replacing the exact covariance P by the ensemble sample covariance, and - adding noise to the data in order to avoid a shrinking of the ensemble spread and to obtain the correct filtering covariance in the limit

Here are the steps for the so-called **stochastic EnKF**, where we add an artificial, random perturbation to the observations, and we assume we have a linear observation operator, H.

11.1 Stochastic EnKF - linear observation operator

11.1.1 Prediction/Forecast

$$\hat{v}_{k+1}^n = \Psi(v_k^n) + \xi_k^n, \quad n = 1, \ldots, N, \tag{11.1}$$

$$\hat{m}_{k+1} = \frac{1}{N} \sum_{i=1}^{N} \hat{v}_{k+1}^n, \tag{11.2}$$

$$\hat{C}_{k+1} = \frac{1}{N-1} \sum_{i=1}^{N} \left(\hat{v}_{k+1}^n - \hat{m}_{k+1}\right) \left(\hat{v}_{k+1}^n - \hat{m}_{k+1}\right)^{\mathrm{T}}. \tag{11.3}$$

11.1.2 Correction/Analysis

$$S_{k+1} = H\hat{C}_{k+1}H^{\mathrm{T}} + \Gamma, \tag{11.4}$$

$$K_{k+1} = \hat{C}_{k+1}H^{\mathrm{T}}S_{k+1}^{-1}, \tag{11.5}$$

$$y_{k+1}^n = y_{k+1} + \eta_{k+1}^n, \quad n = 1, \ldots, N, \tag{11.6}$$

$$v_{k+1}^n = (I - K_{k+1}H)\hat{v}_{k+1}^n + K_{k+1}y_{k+1}^n, \quad n = 1, \ldots, N. \tag{11.7}$$

Alternatively, defining the innovation $d = y_{k+1}^n - H\hat{v}_{k+1}^n$, we can write the state update more simply as

$$v_{k+1}^n = \hat{v}_{k+1}^n + K_{k+1}d.$$

In words:

1. For a given $N_e \in \mathbb{N}$ generate i.i.d. ensemble of states random variables from the distribution of $X(0)$.
2. For $t \in \mathbb{N}$ recursively repeat the following steps:

- Advance each ensemble member in time, using the nonlinear state equation with independently generated random state noise
- Compute the forecast sample mean and the forecast sample covariance
- Compute the sample Kalman gain
- Add additional perturbation to the observation vector Y using independently generated random variables $\eta(t)$
- Update each forecast ensemble member

Burgers et al. [1998] shows that without the data perturbation, the covariance of the ensemble would go to the zero matrix as t goes to infinity. The data perturbation also guarantees that the relation between the forecast sample covariance and the analysis sample covariance is analogous to the relation between the forecast and analysis covariances in the standard KF.

11.2 Full nonlinear formulation of the ensemble Kalman filter

There are many ways to formulate the EnKF. Following Vetra-Carvalho, et al (Tellus A, 2018), we express the filter in terms of the anomalies of state and observations. This is indispensable for fully nonlinear state and measurement models,

$$x_{k+1}^n = \Psi(x_k^n) + w_k^n, \quad n = 1, \ldots, N_e, \tag{11.8}$$
$$y_{k+1} = \mathcal{H}(x_{k+1}) + v_{k+1}. \tag{11.9}$$

To fix notation:

- state forecast X^{f}, dimension $(N_t \times N_x)$
- *ensemble* state forecast $\mathbf{X}^{\mathrm{f}}$, dimension $(N_t \times N_x \times N_e)$
- observation, Y, dimension $(N_t \times N_y)$

- ensemble state *anomaly*,
$$\mathbf{X}' = \frac{1}{\sqrt{N_e - 1}} \left(\mathbf{X} - \overline{X} \right),$$
dimension $(N_t \times N_x \times N_e)$ with $\overline{X} = (1/N_e) \sum_{e=1}^{N_e} X_e$
- ensemble observation *anomaly*,
$$\mathbf{Y}' = \frac{1}{\sqrt{N_e - 1}} \left(\mathcal{H}(\mathbf{X}) - \overline{\mathcal{H}(\mathbf{X})} \right),$$
dimension $(N_t \times N_y \times N_e)$ with $\overline{\mathcal{H}(\mathbf{X})} = (1/N_e) \sum_{e=1}^{N_e} \mathcal{H}(X_e)$.

Then, the Kalman analysis update is

$$\mathbf{X}^{\mathrm{a}} = \mathbf{X}^{\mathrm{f}} + \mathbf{X}'(\mathbf{Y}')^{\mathrm{T}} S^{-1} D,$$

with

$$S = \mathbf{Y}'(\mathbf{Y}')^{\mathrm{T}} + R \qquad \text{(observation covariance)}, \tag{11.10}$$
$$D = (\mathbf{Y} + \mathbf{y}) - \mathcal{H}(\mathbf{X}) \qquad \text{(innovation)}, \tag{11.11}$$

where $y \sim \mathcal{N}(0, R)$ is the stochastic perturbation, and R is the measurement noise covariance matrix.

Or, defining the Kalman gain matrix as

$$K = \mathbf{X}'(\mathbf{Y}')^{\mathrm{T}} S^{-1},$$

we obtain the classical KF update,

$$\mathbf{X}^{\mathrm{a}} = \mathbf{X}^{\mathrm{f}} + KD.$$

11.3 Summary of EnKF properties

- EnKF represents error statistics by ensembles of (nonlinear) model and (nonlinear) measurement realizations.
- EnKF performs sequential DA that processes measurements recursively in time.
- EnKF is suitable for weather-prediction and any other complex, chaotiic dynamic systems.
- Error propagation is nonlinear (see point 1).
- Filter update is linear and computed in the low rank, ensemble subspace.
- EnKF does not require any gradients, adjoints, linearizations.

12 Example 1: noisy pendulum

Consider the nonlinear ODE model for the oscillations of a noisy pendulum with unit mass and length L,

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} + \frac{g}{L}\sin\theta + w(t) = 0$$

where θ is the angular displacement of the pendulum, g is the gravitational constant, L is the pendulum's length, and $w(t)$ is a white noise process. This is rewritten in state space form,

$$\dot{\mathbf{x}} + \mathcal{M}(\mathbf{x}) + \mathbf{w} = 0,$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}, \quad \mathcal{M}(\mathbf{x}) = \begin{bmatrix} x_2 \\ -\dfrac{g}{L}\sin x_1 \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} 0 \\ w(t) \end{bmatrix}.$$

Suppose that we have discrete, noisy measurements of the horizontal component of the position, $\sin(\theta)$. Then the measurement equation is scalar,

$$y_k = \sin\theta_k + v_k,$$

where v_k is a zero-mean Gaussian random variable with variance R. The system is thus nonlinear in state and measurement and the state-space system is of the general form seen above. A simple discretization, based on Euler's method produces

$$\begin{aligned} \mathbf{x}_k &= \mathcal{M}(\mathbf{x}_{k-1}) + \mathbf{w}_{k-1}, \\ y_k &= \mathcal{H}_k(\mathbf{x}_k) + v_k, \end{aligned}$$

where

$$\mathbf{x}_k = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_k, \quad \mathcal{M}(\mathbf{x}_{k-1}) = \begin{bmatrix} x_1 + \Delta t x_2 \\ x_2 - \Delta t \dfrac{g}{L} \sin x_1 \end{bmatrix}_{k-1}, \quad \mathcal{H}(\mathbf{x}_k) = [\sin x_1]_k.$$

The noise terms have distributions

$$\mathbf{w}_{k-1} \sim \mathcal{N}(\mathbf{0}, Q), \quad v_k \sim \mathcal{N}(0, R),$$

where the process covariance matrix is

$$Q = \begin{bmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{bmatrix},$$

with components (see KF example for 2D motion tracking),

$$q_{11} = q_c \frac{\Delta t^3}{3}, \quad q_{12} = q_{21} = q_c \frac{\Delta t^2}{2}, \quad q_{22} = q_c \Delta t,$$

and q_c is the continuous process noise spectral density.

Simulations

In the simulations, we take: (see also Extended Kalman Filter example)

- 500 time steps with $\Delta t = 0.01$.
- Noise levels $q_c = 0.01$ and $R = 0.1$.
- Initial angle $x_1 = 1.8$ and initial angular velocity $x_2 = 0$.
- Initial diagonal state covariance of 0.1.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import linalg
np.random.seed(6)

# initialize and set up all matrices
q = 0.1 #0.01     # process noise
r = 0.1 #0.02 #0.1      # measurement noise
# need to add the ensemble dimension
Ne = 10   # number of ensemble members
Nx = 2    # state dimension
Ny = 1    # observation dimension
Nt = 500  # time dimension
```

```
dt = 0.01 # time step
g  = 9.81 # gravitational acceleration

x_0 = np.array([1.5, 0.])   # Initial state

m_0 = np.array([1.6, 0.])   # Initial state estimate (slightly off)
#P_0 = np.array([[0.1, 0.],
#                [0., 0.1]]) # Initial estimate covariance
P_0 = np.array([[r, 0.],
                [0., r]]) # Initial estimate covariance


sig_w = q # process noise
sig_v = r # measurement noise

Q  = sig_w**2 * np.array([[dt ** 3 / 3, dt ** 2 / 2],
                          [dt ** 2 / 2, dt]])
R  = sig_v**2 * np.eye(Ny)

# Observation operator (nonlinear)
def Hx(x):
    return np.array([np.sin(x[0])])

# State dynamics (nonlinear)
def Ax(x, dt):
    m = np.array([x[0] + dt*x[1],
                  x[1] - g*dt* np.sin(x[0])])
    return m
```

12.1 Generate noisy observations

To generate the noisy observations, we have 2 options

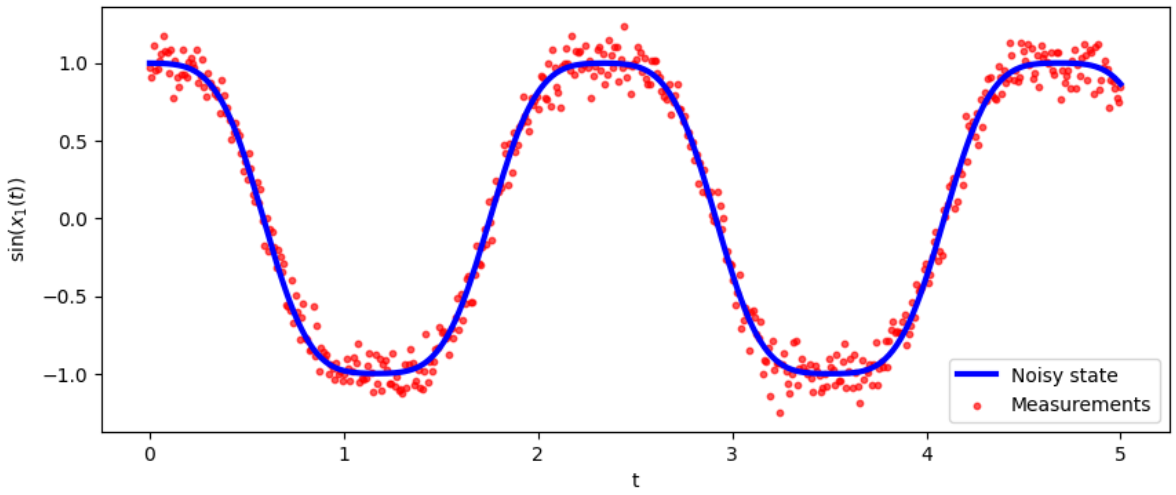
1. Use an accurate RK4 method.
2. Use a simpler, first-order Euler method.

Whatever the approach chosen, it must be used both for the observation generation and for the state evolution within the Kalman filter.

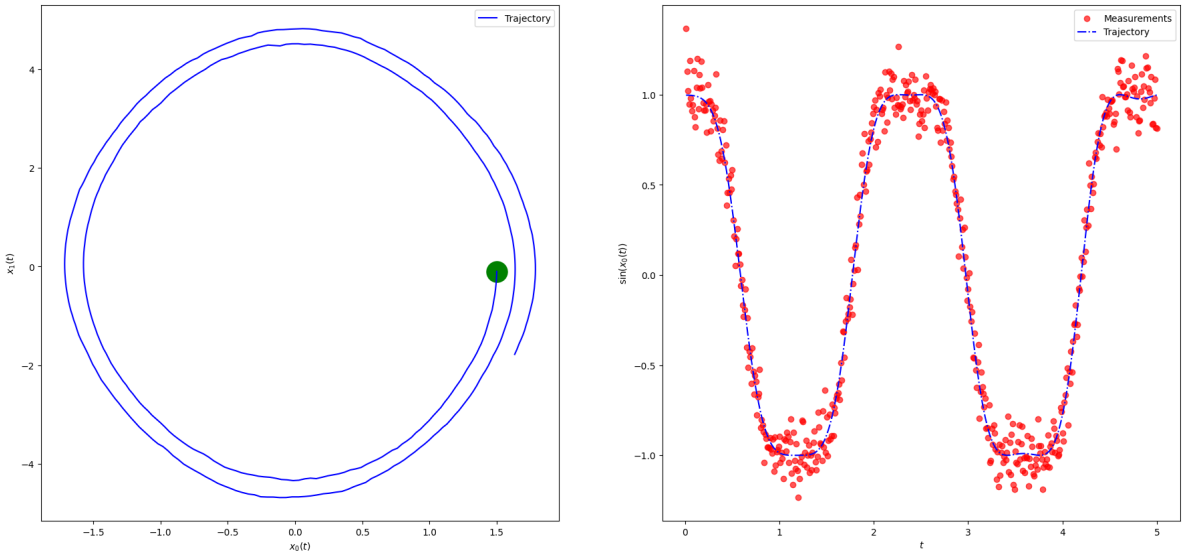
We will use the simpler Euler approximation. For reference, here is the RK4 method.

```
def Pendulum(state, *args): #nonlinear pendulum
    g = args[0]
    L = args[1]
    x1, x2 = state #Unpack the state vector
    f = np.zeros(2) #Derivatives
    f[0] = x2
    f[1] = -g/L * np.sin(x1)
    return f

def RK4(rhs, state, dt, *args):

    k1 = rhs(state,         *args)
    k2 = rhs(state+k1*dt/2, *args)
    k3 = rhs(state+k2*dt/2, *args)
    k4 = rhs(state+k3*dt,   *args)

    new_state = state + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
    return new_state

#np.random.seed(55)
# Solve system and generate noisy observations
T = np.linspace(0, Nt, Nt)
#x0True = np.array([1.8, 0.0]) # True initial conditions
x0True = np.array([1.5, 0.0]) # True initial conditions
#time integration
chol_Q = np.linalg.cholesky(Q) # noise std dev.
sqrt_R = np.sqrt(R)
xxTrue = np.zeros([Nx, Nt])
xxTrue[:,0] = x0True
yy = np.zeros((Nt, Ny))
yy[0] = np.sin(xxTrue[0, 0]) + sig_v * np.random.randn() # must initialize correctly!
for k in range(Nt-1):
    w_k = chol_Q @ np.random.randn(2)
    xxTrue[:,k+1] = RK4(Pendulum, xxTrue[:,k], dt, g, 1.0) + w_k
    yy[k+1,0] = np.sin(xxTrue[0, k+1]) + sig_v * np.random.randn()

# plot results
fig, ax = plt.subplots(nrows=1,ncols=1, figsize=(10,4))
ax.plot(T*dt, np.sin(xxTrue[0, :]), color='b', linewidth = 3, label="Noisy state")
ax.scatter(T*dt, yy[:,0], marker="o", label="Measurements", color="red", alpha=0.66, s=10)
ax.set_xlabel('t')
ax.set_ylabel("$\sin(x_1(t))$", labelpad=5)
```

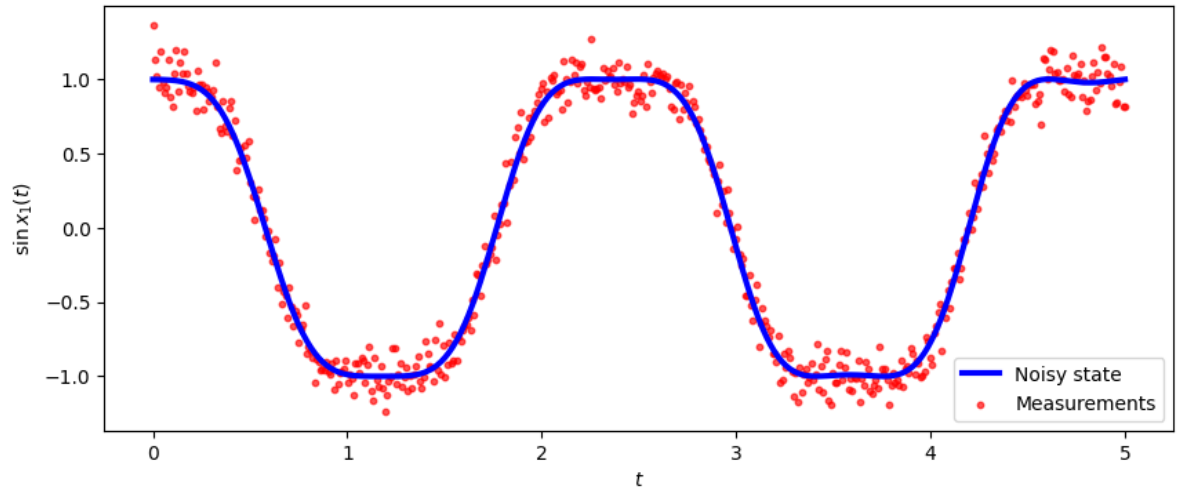
```
ax.legend()
plt.show()
```




```
# Use Euler method
from common_utilities import generate_pendulum, RandomState, rmse, plot_pendulum
random_state = RandomState(1)
steps = Nt
x_0 = x0True
#timeline, states, observations = generate_pendulum(x_0, g, Q, dt, R, steps, random_state)
#plot_pendulum(timeline, observations, states, "Trajectory")
y = np.zeros((Nt, Ny))
timeline, xTrue, y[:, 0] = generate_pendulum(x_0, g, Q, dt, R, steps, random_state)
plot_pendulum(timeline, y, xTrue, "Trajectory")
```

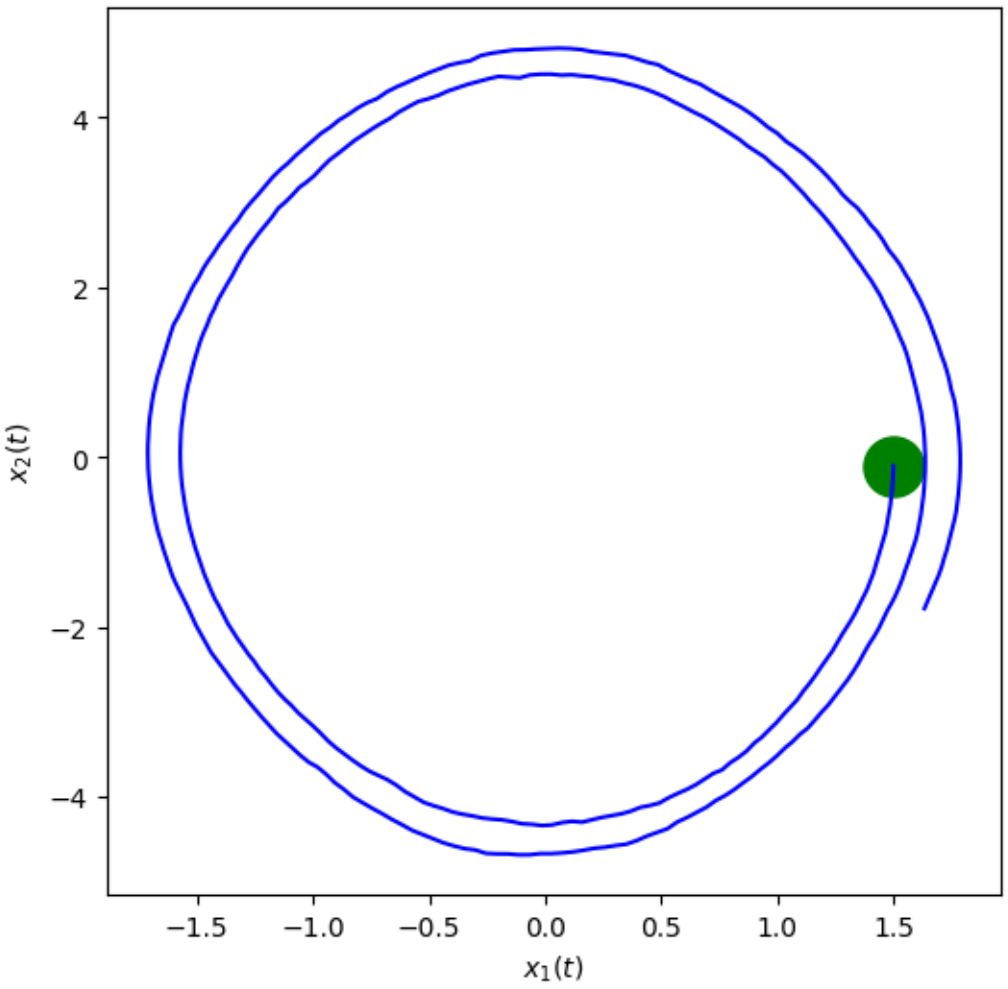



```
fig, ax = plt.subplots(nrows=1,ncols=1, figsize=(10,4))
ax.plot(T*dt, np.sin(xTrue[:, 0]), color='b', linewidth = 3, label="Noisy state")
ax.scatter(T*dt, y[:,0], marker="o", label="Measurements", color="red", alpha=0.66, s=10)
ax.legend()
plt.ylabel('$\sin x_1(t)$')
plt.xlabel('$t$')
plt.show()
```




```
fig, ax = plt.subplots(nrows=1,ncols=1, figsize=(6,6))
#plt.plot(xTrue[0, :], xTrue[1, :], 'b')
```

```
plt.plot(xTrue[:, 0], xTrue[:, 1], 'b')
plt.scatter(xTrue[0, 0], xTrue[0, 1], marker="o", color="green", s=500)
plt.xlabel('$x_1(t)$')
plt.ylabel('$x_2(t)$')
plt.show()
```



12.2 Ensemble Kalman Filter

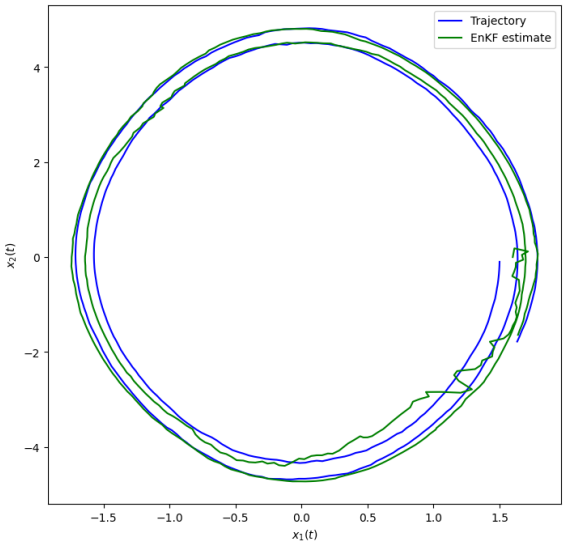
```
def ens_kalman_filter(m_0, P_0, Q, R, Ne, dt, Y):
    Nx = m_0.shape[-1]
```

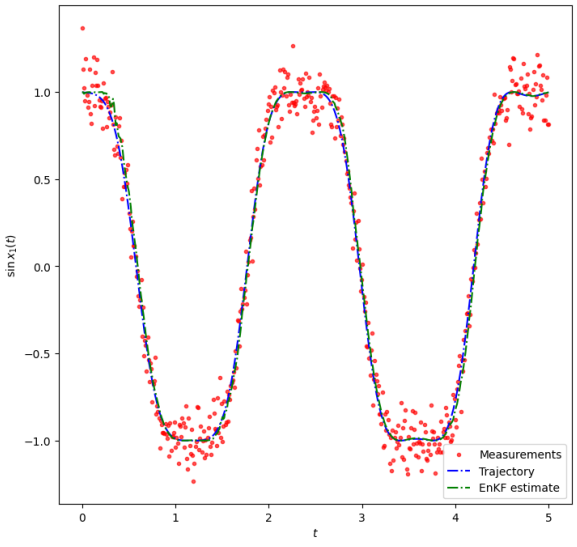
```
    Nt, Ny = Y.shape

    enkf_m = np.empty((Nt, Nx))
    enkf_P = np.empty((Nt, Nx, Nx))
    X      = np.empty((Nx, Ne))
    HXf    = np.empty((Ny, Nx))

    X[:,:] = np.tile(m_0, (Ne,1)).T  + np.linalg.cholesky(P_0)@np.random.randn(Nx, Ne) # init
    P        = P_0 # initial state covariance
    enkf_m[0, :]    = m_0
    enkf_P[0, :, :] = P_0

    for i in range(Nt-1):
        # ==== predict/forecast ====
        Xf = Ax(X[:,:],dt) + np.linalg.cholesky(Q)@np.random.randn(Nx, Ne) # predict state er
        mX = np.mean(Xf, axis=1)        # state ensemble mean
        Xfp = Xf - mX[:, None]          # state forecast anomaly
        #Phat = Xfp @ Xfp.T / (Ne - 1)  # predict covariance (not needed)
        # ==== prepare =====
        HXf = Hx(Xf)                    # nonlinear observation
        mY = np.mean(HXf, axis=1)       # observation ensemble mean
        HXp = HXf - mY[:, None]         # observation anomaly
        S = (HXp @ HXp.T)/(Ne - 1) + R  # observation covariance
        K = linalg.solve(S, HXp @ Xfp.T, assume_a="pos").T  / (Ne - 1)  # Kalman gain
        # === perturb y and compute innovation ====
        ypert = Y[i,:] +  np.linalg.cholesky(R)@np.random.randn(Ny, Ne)
        d = ypert - HXf
        # ==== correct/analyze ====
        X[:,:] = Xf + K @ d         # update state ensemble
        mX = np.mean(X[:,:], axis=1)# state analysis ensemble mean
        Xap = X[:,:] - mX[:, None]  # state analysis anomaly
        P = Xap @ Xap.T / ( Ne - 1)     # update covariance
        # ==== save ====
        enkf_m[i+1] = mX                # save KF state estimate (mean)
        enkf_P[i+1] = P                 # save KF error estimate (covariance)
    return enkf_m, enkf_P
```

```
# initialize state and covariance
x_0 = np.array([1.5, 0.])   # Initial state
m_0 = np.array([1.6, 0.])   # Initial state estimate (slightly off)
P_0 = np.array([[r, 0.],
                [0., r]]) # Initial estimate covariance
```

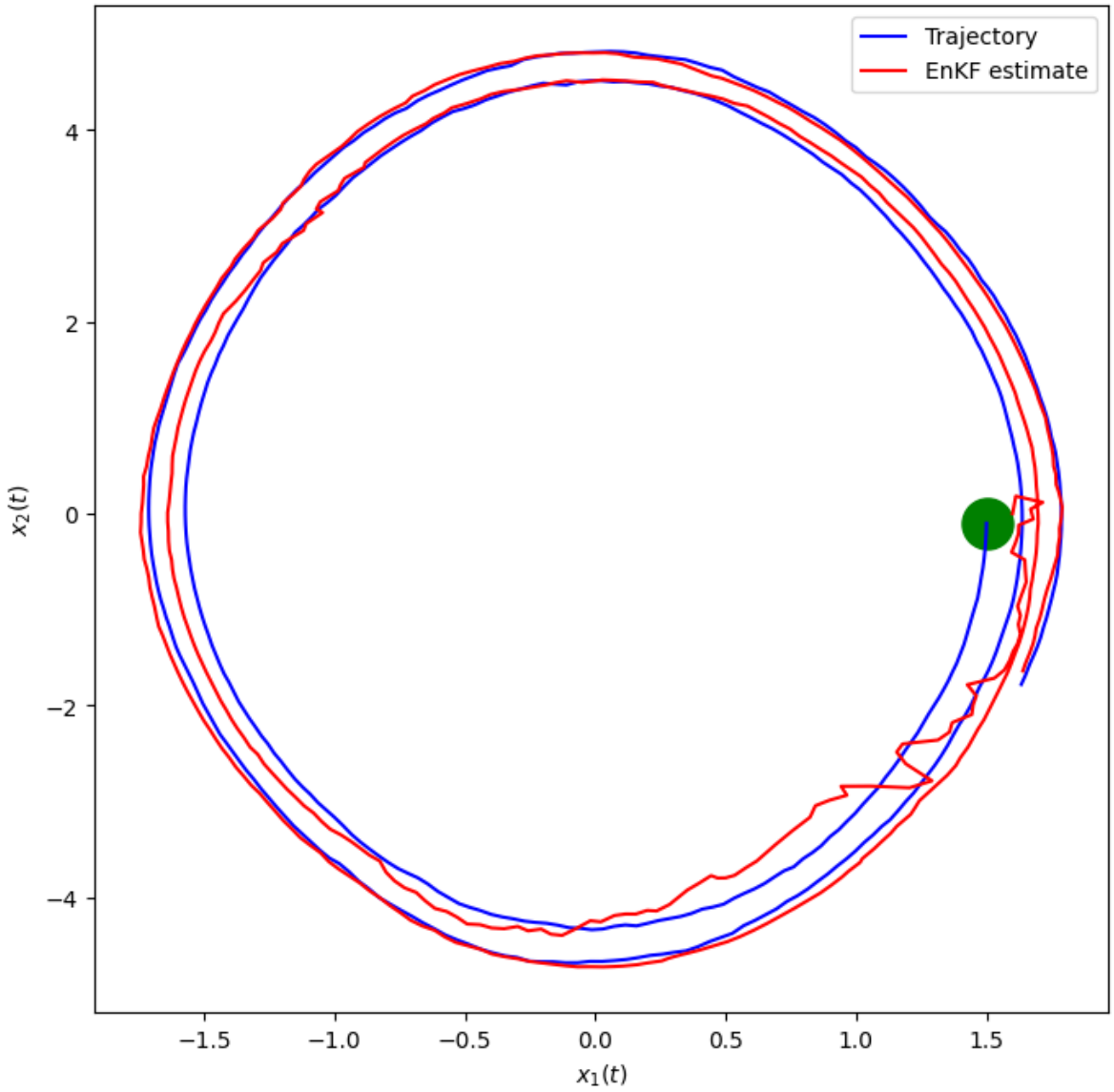
```
enkf_m, enkf_P = ens_kalman_filter(m_0, P_0, Q, R, Ne, dt, y)

#rmse_ekf = rmse(ekf_m[:, :1], states[:, :1])
#print(f"EKF RMSE: {rmse_ekf}")
```

```
fig, axes = plt.subplots(ncols=2, figsize=(18, 8))
x1 = xTrue  # states
x2 = enkf_m # KF estimate
y  = y      # Measurements
timeline = T*dt
axes[1].scatter(timeline, y, marker=".", label="Measurements", color="red", alpha=0.66)
#axes[1].plot(timeline, np.sin(x1[0, :]), linestyle="dashdot", label="Trajectory", color="blu
axes[1].plot(timeline, np.sin(x1[:, 0]), linestyle="dashdot", label="Trajectory", color="blue
axes[1].plot(timeline, np.sin(x2[:, 0]), linestyle="dashdot", label="EnKF estimate", color="g
#axes[0].plot(x1[0, :], x1[1, :], label="Trajectory", color="blue")
axes[0].plot(x1[:, 0], x1[:, 1], label="Trajectory", color="blue")
axes[0].plot(x2[:, 0], x2[:, 1], label="EnKF estimate", color="green")
axes[0].legend()
axes[1].legend()
axes[0].set_xlabel('$x_1(t)$')
axes[1].set_xlabel('$t$')
axes[0].set_ylabel('$ x_2(t)$')
axes[1].set_ylabel('$\sin x_1(t)$')

plt.show()
```





```
fig, ax = plt.subplots(nrows=1,ncols=1, figsize=(8,8))
plt.scatter(x1[0, 0], x1[0, 1], marker="o", color="green", s=500)
plt.plot(x1[:, 0], x1[:, 1], label="Trajectory", color="blue")
plt.plot(x2[:, 0], x2[:, 1], label="EnKF estimate", color="red")
plt.xlabel('$x_1(t)$')
plt.ylabel('$x_2(t)$')
plt.legend()
plt.show()
```



Observations

The ensemble Kalman filter tracks rapidly, smoothly and accurately the noisy trajectory of the pendulum. This can be compared to the performance of the EKF (extended Kalman filter) in Example 2, Section 4, that was less accurate (took longer to catch up) and was more noisy.
Note that the error/anomaly covariance elements, P_{ii}, are noisy, since they are empirically averaged quantities.

13 Example 2: Lorenz63 system

In this example, we will concentrate on DA methods applied to the Lorenz systems of ordinary differential equations. These systems exhibit chaotic behavior and as such are considered as excellent toy models for complex phenomena, in particular for simulation of weather. The Lorenz-63 system is given by

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= -\sigma(x-y), \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= \rho x - y - xz, \\ \frac{\mathrm{d}z}{\mathrm{d}t} &= xy - \beta z, \end{aligned} \tag{13.1}$$

where $x = x(t)$, $y = y(t)$, $z = z(t)$ and σ (ratio of kinematic viscosity divided by thermal diffusivity), ρ (measure of stability) and β (related to the wave number) are parameters.

Chaotic behavior is obtained when the parameters are chosen as

$$\sigma = 10, \quad \rho = 28, \quad \beta = 8/3.$$

Its solution is very sensitive to the parameters and the initial conditions and a small difference in these values can lead to a very different solution. This is the basis of its ill-posedness. This equation is a simplified model for atmospheric convection and is an excellent example of the lack of predictability. It is ill-posed in the sense of Hadamard. In fact the solution switches between two stable orbits, around the points

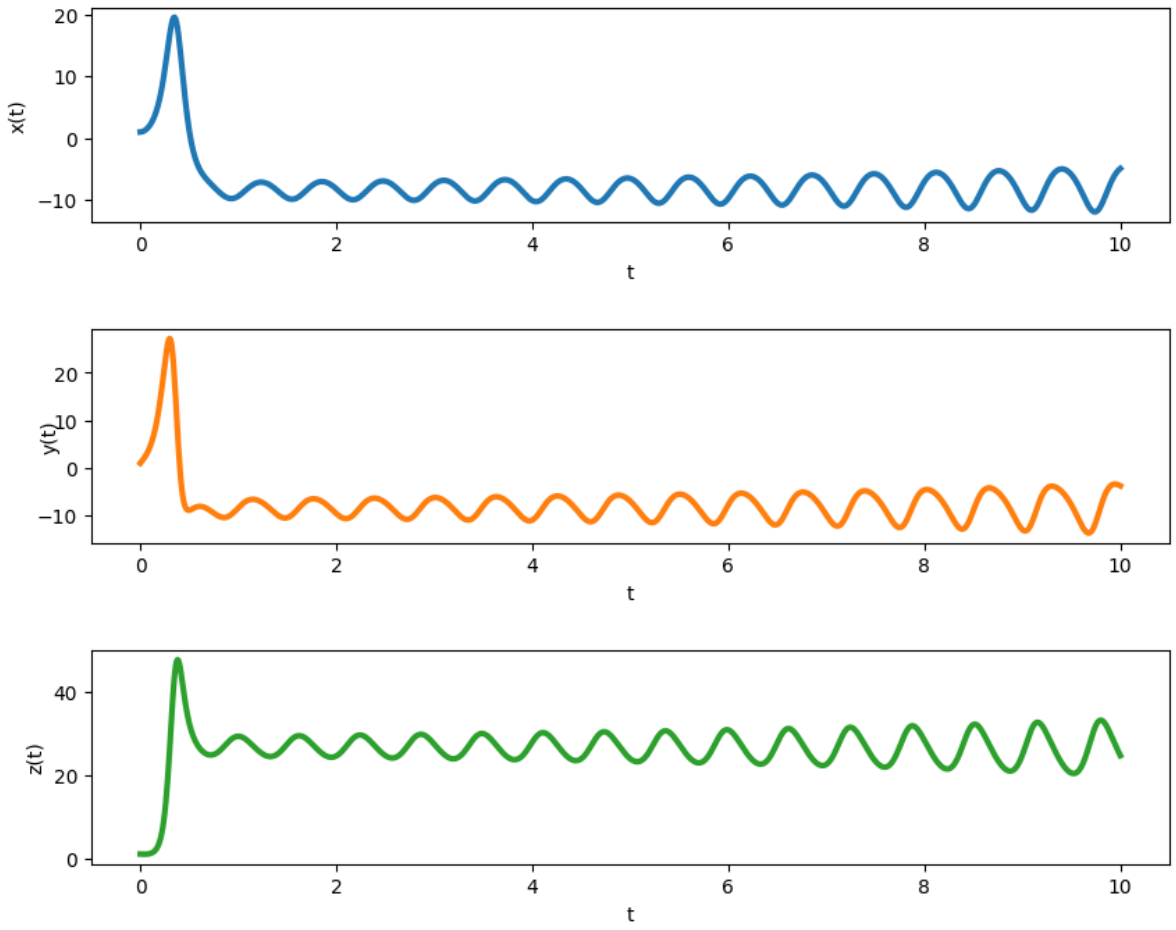
$$\left(\sqrt{\beta(\rho-1)}, \sqrt{\beta(\rho-1)}, \rho-1\right), \quad \left(-\sqrt{\beta(\rho-1)}, -\sqrt{\beta(\rho-1)}, \rho-1\right).$$

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
np.random.seed(1)
```

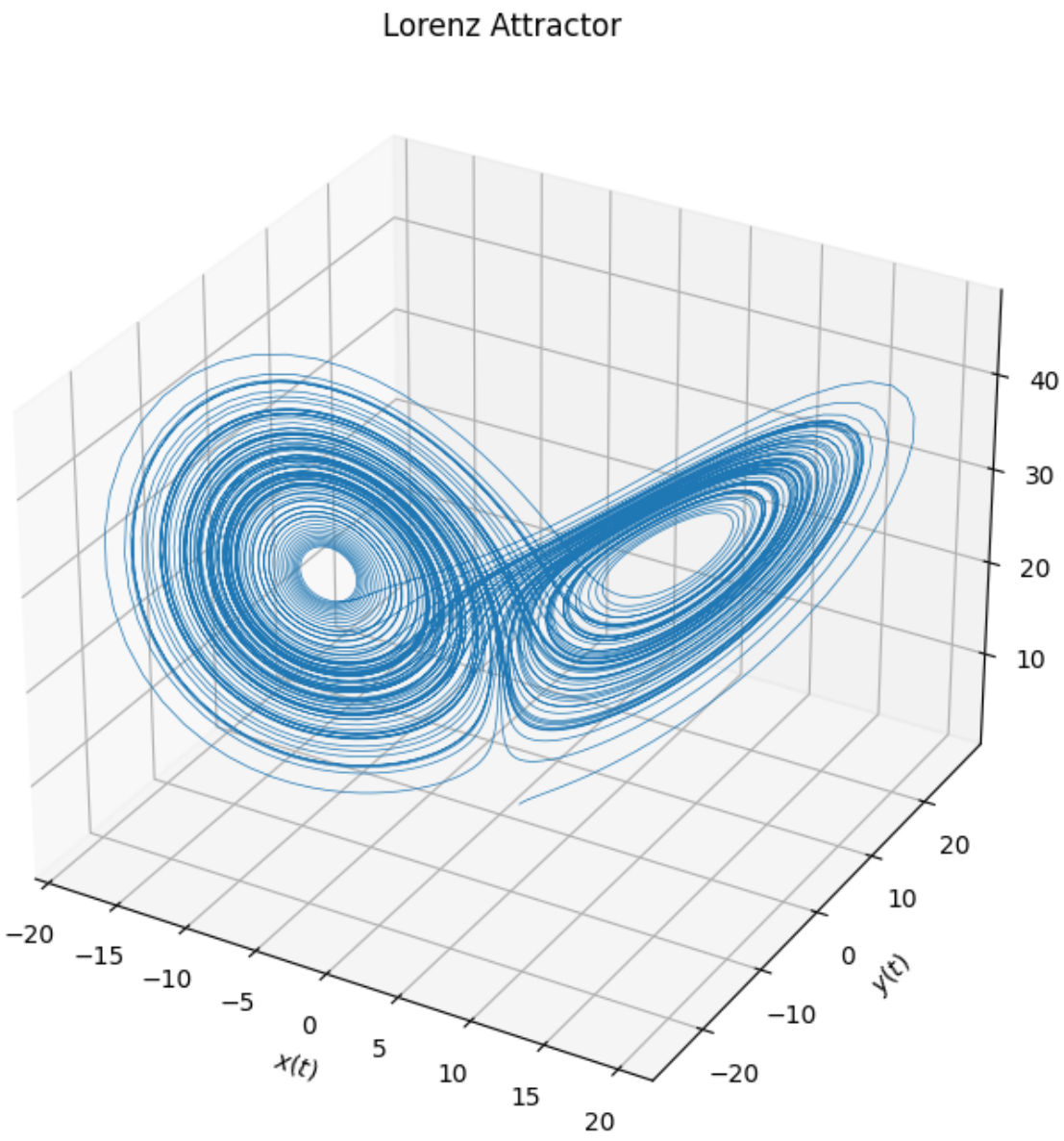
```
def Lorenz63(state, *args):
    """
    Parameters
    ----------
    state : array-like, shape (3,)
       Point of interest in three-dimensional space.
    *args (sigma, rho, beta) : float
       Parameters defining the Lorenz attractor.

    Returns
    -------
    state_dot : array, shape (3,)
       Values of the Lorenz attractor's derivatives at *state*.
    """
    sigma = args[0]
    rho = args[1]
    beta = args[2]
    x, y, z = state #Unpack the state vector
    f = np.zeros(3) #Derivatives
    f[0] = sigma * (y - x)
    f[1] = x * (rho - z) - y
    f[2] = x * y - beta * z
    return f

def RK4(rhs, state, dt, *args):

    k1 = rhs(state,         *args)
    k2 = rhs(state+k1*dt/2, *args)
    k3 = rhs(state+k2*dt/2, *args)
    k4 = rhs(state+k3*dt,   *args)

    new_state = state + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
    return new_state
```

```
# parameters Lorenz63
sigma = 10.0
rho  = 28.0
beta = 8.0/3.0
dt = 0.01
tm = 10
nt = int(tm/dt)
t  = np.linspace(0,tm,nt+1)
```

```
# initialize and solve
u0True = np.array([1,1,1]) # True initial conditions
#time integration
uTrue = np.zeros([nt+1,3])
uTrue[0,:] = u0True
for k in range(nt):
    uTrue[k+1,:] = RK4(Lorenz63,uTrue[k,:], dt, sigma, rho, beta)

# plot results
fig, ax = plt.subplots(nrows=3,ncols=1, figsize=(10,8))
ax = ax.flat

prop_cycle = plt.rcParams['axes.prop_cycle']
colors = prop_cycle.by_key()['color']

for k in range(3):
    ax[k].plot(t,uTrue[:,k], label='True', color = colors[k], linewidth = 3)
    ax[k].set_xlabel('t')

ax[0].set_ylabel('x(t)', labelpad=5)
ax[1].set_ylabel('y(t)', labelpad=-12)
ax[2].set_ylabel('z(t)')
fig.subplots_adjust(hspace=0.5)
```




```
nt = 10000 # need longer time to see the attractor
uTrue = np.zeros([nt+1,3])
uTrue[0,:] = u0True

for k in range(nt):
    uTrue[k+1,:] = RK4(Lorenz63,uTrue[k,:], dt,sigma, rho, beta)

# Plot
ax = plt.figure(figsize=(10,8)).add_subplot(projection='3d')
ax.plot(*uTrue.T, lw=0.5)
ax.set_xlabel("$x(t)$")
ax.set_ylabel("$y(t)$")
ax.set_zlabel("$z(t)$")
ax.set_title("Lorenz Attractor")
plt.show()
```



13.1 Ensemble KF for Data Assimilation

Here we will generalize the ensemble Kalman filter to take into account the possibility of sparse observations. This is usually the case in real-life systems, where observations are only available et fixed instants, and hence the filtering can only be applied at these times. Inbetween observations, the system evolves freely (without correction) according to its underlying state

equation.

Suppose we have N_y measurements/observations at an interval of δt_y. This gives measurements for times $t_0 \leq t \leq t_m$, where $t_m = N_m \delta t_m$. This can be considered as the assimilation window. The system then evolves freely for $t > t_m$ until some final forecast window time t_f. The state, or equation itself is simulated with a smaller δt and for a large number N_t steps, giving $t_f = N_t \delta t$. Usually, for real life systems, we will have

$$\delta t_m \geq \delta t, \quad N_m \leq N_t, \quad t_m \leq t_f.$$

For code testing, we make the simplifying (unrealistic) academic assumption that

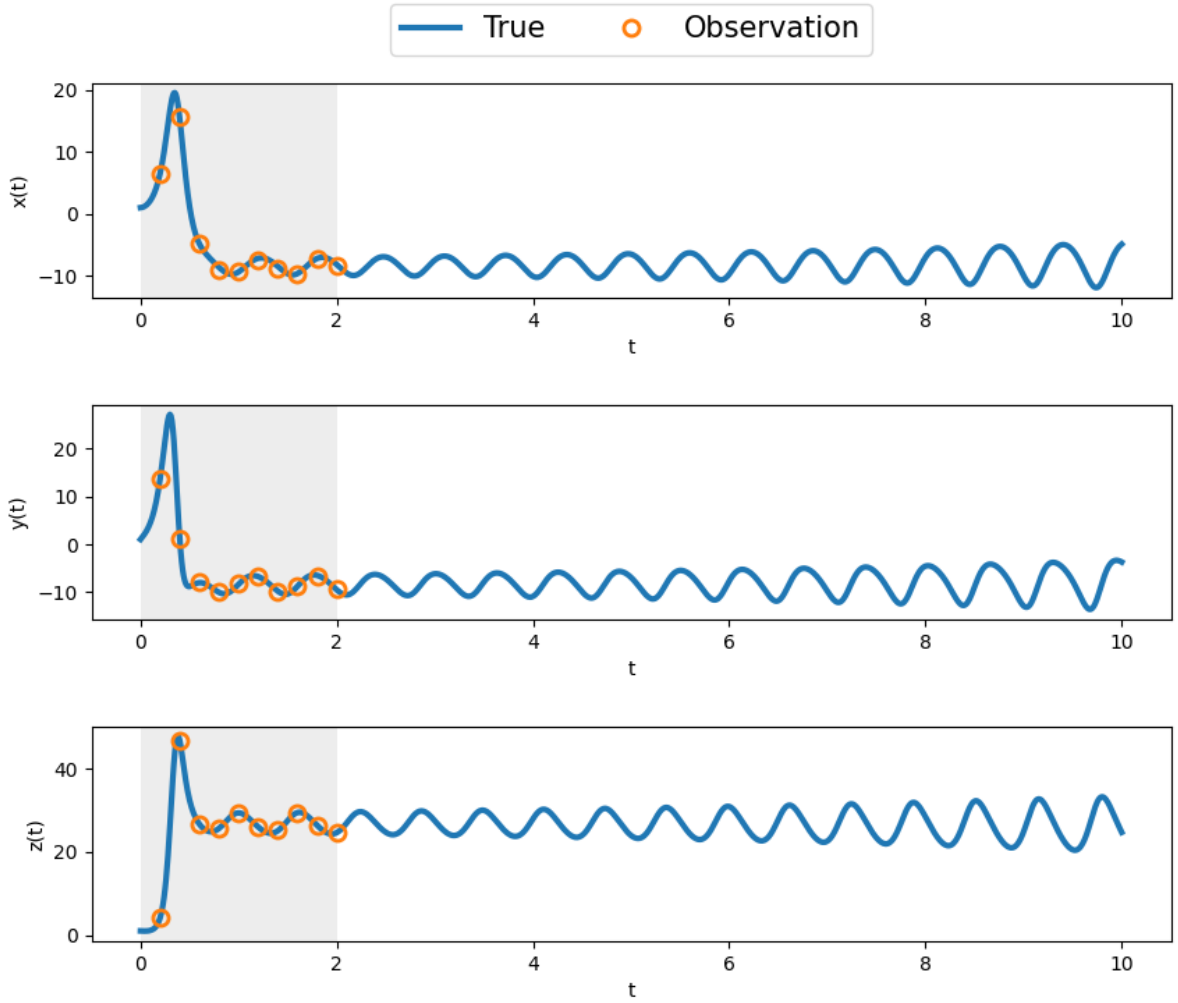
$$\delta t_m = \delta t, \quad N_m = N_t, \quad t_f = t_m.$$

This implies the availabilty of measurements at each (and every) time step. Note that in many of the previous examples, this was indeed the case.

```
def enKF_Lorenz63_setup(dt, T, dt_m, T_m, sig_w, sig_v):
    """
    Prepare input (true state and observations) for the stochastic
    ensemble filter of the Lorenz63 system.

    Parameters:
        dt: time step for state evolution
        T:  time interval for state evolution
        dt_m: time interval between 2 measurements (can equal dt for dense observations)
        T_m: time interval for observations
        sig_w: state noise sd., cov. Q = sig_w**2 x np.eye(3)
        sig_v: measurement noise sd., cov. R = sig_v**2 x np.eye(3)
    """
    # parameters Lorenz63
    sigma = 10.0
    rho   = 28.0
    beta = 8.0/3.0
    dim_x = 3
    dim_y = 3
    # noise covariances
    Q  = sig_w**2 * np.eye(dim_x)
    R  = sig_v**2 * np.eye(dim_y)
    # measurement operator (identity here)
    def H(u):
        w = u
```

```
        return w

    # Solve system and generate noisy observations
    Nt = int(T/dt)     # number of time steps
    Nm = int(T_m/dt_m) # number of observations
    t = np.linspace(0, Nt, Nt+1) * dt # time vector (including 0 and T)
    ind_m = (np.linspace(int(dt_m/dt),int(T_m/dt),Nm)).astype(int) # obs. indices
    t_m = t[ind_m] # measurement time vector
    x0True = np.array([1.0, 1.0, 1.0]) # True initial conditions
    sqrt_Q = np.linalg.cholesky(Q) # noise std dev.
    sqrt_R = np.linalg.cholesky(R)
    # initialize (correctly!)
    xTrue = np.zeros([Nt+1, dim_x])
    xTrue[0, :] = x0True
    y = np.zeros((Nm, dim_y))
    km = 0 # index for measurement times
    y[0,:] = H(xTrue[0,:]) + sig_v * np.random.randn(dim_y)
    for k in range(Nt):
        w_k = sqrt_Q @ np.random.randn(dim_x)
        xTrue[k+1,:] = RK4(Lorenz63,xTrue[k,:], dt,sigma, rho, beta) #+ w_k
        if (km < Nm) and (k+1 == ind_m[km]):
            v_k = sqrt_R @ np.random.randn(dim_y)
            y[km,:]     = H(xTrue[k+1,:]) + v_k
            km = km + 1
    # plot state and measurements
    fig, ax = plt.subplots(nrows=3,ncols=1, figsize=(10,8))
    ax = ax.flat
    #t = T*dt
    for k in range(3):
        ax[k].plot(t,xTrue[:,k], label='True', linewidth = 3)
        ax[k].plot(t[ind_m],y[:,k], 'o', fillstyle='none', \
                   label='Observation', markersize = 8, markeredgewidth = 2)
        ax[k].set_xlabel('t')
        ax[k].axvspan(0, T_m, color='lightgray', alpha=0.4, lw=0)
    ax[0].legend(loc="center", bbox_to_anchor=(0.5,1.25),ncol =4,fontsize=15)
    ax[0].set_ylabel('x(t)')
    ax[1].set_ylabel('y(t)')
    ax[2].set_ylabel('z(t)')
    fig.subplots_adjust(hspace=0.5)

    return Q, R, xTrue, y, ind_m, Nt, Nm
```

```
Q, R, xTrue, y, ind_m, Nt, Nm = enKF_Lorenz63_setup(dt=0.01, T=10, dt_m=0.2, T_m =2, sig_w=0
```




```
def enKF_Lorenz63_DA(x0, P0, Q, R, y, ind_m, Nt, Nm, Ne=10):
    """
    Run DA of the Lorenz63 system using the stochastic
    ensemble filter with sparse observations in the DA
    window, defined by time index set `ind_m`.

    Parameters:

    """
    # parameters Lorenz63
```

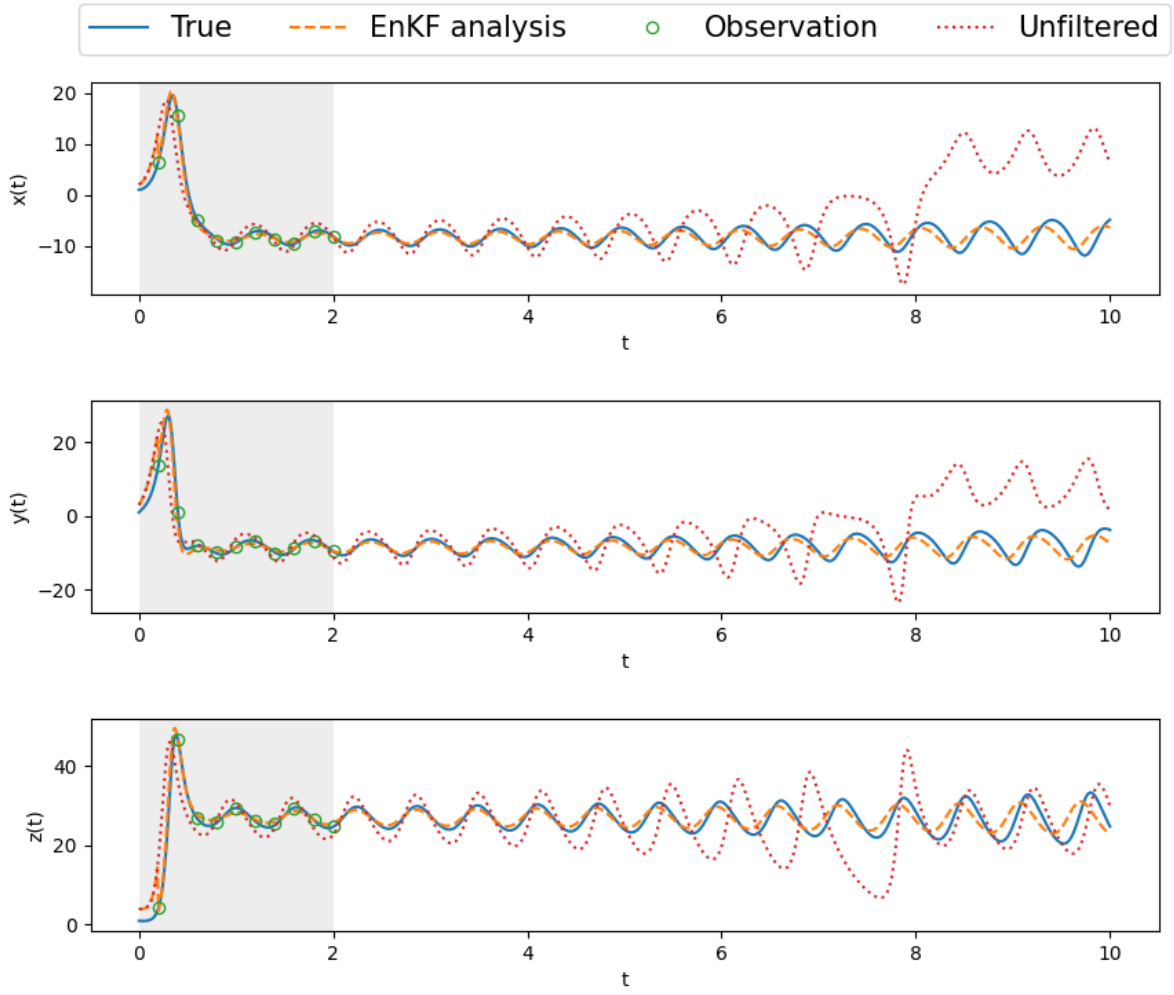
```
sigma = 10.0
rho   = 28.0
beta = 8.0/3.0
def Hx(u):
    w = u
    return w
Nx     = x0.shape[-1]
Ny     = y.shape[-1]
enkf_m = np.empty((Nt+1, Nx))
enkf_P = np.empty((Nt+1, Nx, Nx))
X      = np.empty((Nx, Ne))
Xf     = np.empty((Nx, Ne))
HXf    = np.empty((Ny, Nx))

X[:,:] = np.tile(x0, (Ne,1)).T  + np.linalg.cholesky(P0)@np.random.randn(Nx, Ne) # initia
P       = P0 # initial state covariance
enkf_m[0, :]    = x0
enkf_P[0, :, :] = P0

i_m = 0 # index for measurement times

for i in range(Nt):
    # ==== predict/forecast ====
    for e in range(Ne):
        w_i = np.linalg.cholesky(Q) @ np.random.randn(Nx)#, Ne)
        Xf[:,e] = RK4(Lorenz63, X[:,e], dt, sigma, rho, beta) + w_i # predict state ensem
    mX = np.mean(Xf, axis=1)        # state ensemble mean
    Xfp = Xf - mX[:, None]          # state forecast anomaly
    P = Xfp @ Xfp.T / (Ne - 1)      # predict covariance
    # ==== prepare analysis step =====
    if (i_m < Nm) and (i+1 == ind_m[i_m]):
        HXf = Hx(Xf)                    # nonlinear observation
        mY = np.mean(HXf, axis=1)       # observation ensemble mean
        HXp = HXf - mY[:, None]         # observation anomaly
        S = (HXp @ HXp.T)/(Ne - 1) + R  # observation covariance
        K = linalg.solve(S, HXp @ Xfp.T, assume_a="pos").T  / (Ne - 1)  # Kalman gain
        # === perturb y and compute innovation ====
        ypert = y[i_m, :] +  (np.linalg.cholesky(R)@np.random.randn(Ny, Ne)).T
        d = ypert.T - HXf
        # ==== correct/analyze ====
        X[:,:] = Xf + K @ d          # update state ensemble
        mX = np.mean(X[:,:], axis=1) # state analysis ensemble mean
```

```
            Xap = X[:,:] - mX[:, None]   # state analysis anomaly
            P = Xap @ Xap.T / ( Ne - 1)  # update covariance
            i_m = i_m + 1
        else:
            X[:,:] = Xf  # when there is no obs, then state=forecast
        # ==== save ====
        enkf_m[i+1] = mX                # save KF state estimate (mean)
        enkf_P[i+1] = P                 # save KF error estimate (covariance)
    return enkf_m, enkf_P
```

```
# Initialize and run the analysis
sig_w = 0.0015
sig_v = 0.15
Q  = sig_w**2 * np.eye(3) #* 1.e-6 # for comparison with DT
R  = sig_v**2 * np.eye(3)

x0 = np.array([2., 3., 4.]) # a little off
sig_vv = 0.1
P0 = np.eye(3) * sig_vv**2 # Initial estimate covariance
Ne = 10
Xa, P = enKF_Lorenz63_DA(x0, P0, Q, R, y, ind_m, Nt, Nm, Ne=10)
```

```
# Post-process and plot the results
# generate unfiltered state
Xb = np.empty((Nt+1, 3))
Xb[0,:] = x0
for i in range(Nt):
    Xb[i+1,:] =  RK4(Lorenz63, Xb[i,:], dt, sigma, rho, beta)
# plot state and measurements
t = np.linspace(0, Nt, Nt+1) * dt # time vector
T_m = 2.
fig, ax = plt.subplots(nrows=3,ncols=1, figsize=(10,8))
ax = ax.flat

for k in range(3):
    ax[k].plot(t,xTrue[:,k], label='True')#, linewidth = 3)
    ax[k].plot(t,Xa[:,k], '--', label='EnKF analysis')#, linewidth = 3)
    ax[k].plot(t[ind_m],y[:,k], 'o', fillstyle='none', \
               label='Observation')#, markersize = 8, markeredgewidth = 2)
    ax[k].plot(t,Xb[:,k], ':', label='Unfiltered')#, linewidth = 3)
    ax[k].set_xlabel('t')
    ax[k].axvspan(0, T_m, color='lightgray', alpha=0.4, lw=0)
```

```
ax[0].legend(loc="center", bbox_to_anchor=(0.5,1.25),ncol =4,fontsize=15)
ax[0].set_ylabel('x(t)')
ax[1].set_ylabel('y(t)')
ax[2].set_ylabel('z(t)')
fig.subplots_adjust(hspace=0.5)
```



13.2 Conclusion

The ensemble Kalman filter, even with very sparse observations and a chaotic system, does an excellent job of

1. tracking within the DA window,

2. forecasting way beyond the window, whereas the unfiltered/unassimilated, freely evolving system deviates considerably, as is to be expected from the chaotic Lorenz system.

From $t \approx 6$, the KF starts to deviate very slightly, mostly a phase difference. However, in real situations, there will already be new measurements available by this time. Then the KF analysis will kick in again, and rectify the trajectory.

14 Example 3: SIR Model

In this example, apply DA methods applied to an SIR systems of ordinary differential equations. The SIR system is given by

$$\frac{dS}{dt} = -\beta SI, \quad S(0) = S_0, \tag{14.1}$$
$$\frac{dI}{dt} = \beta SI - \lambda I, \quad I(0) = I_0, \tag{14.2}$$
$$\frac{dR}{dt} = \lambda I, \quad R(0) = R_0, \tag{14.3}$$

where $S = S(t)$ are susceptibles, $I = I(t)$ infected and $R = R(t)$ recovered. The model parameters are β and λ and the famous rate of reproduction is then

$$R_0 = \frac{\beta}{\gamma} S_0.$$

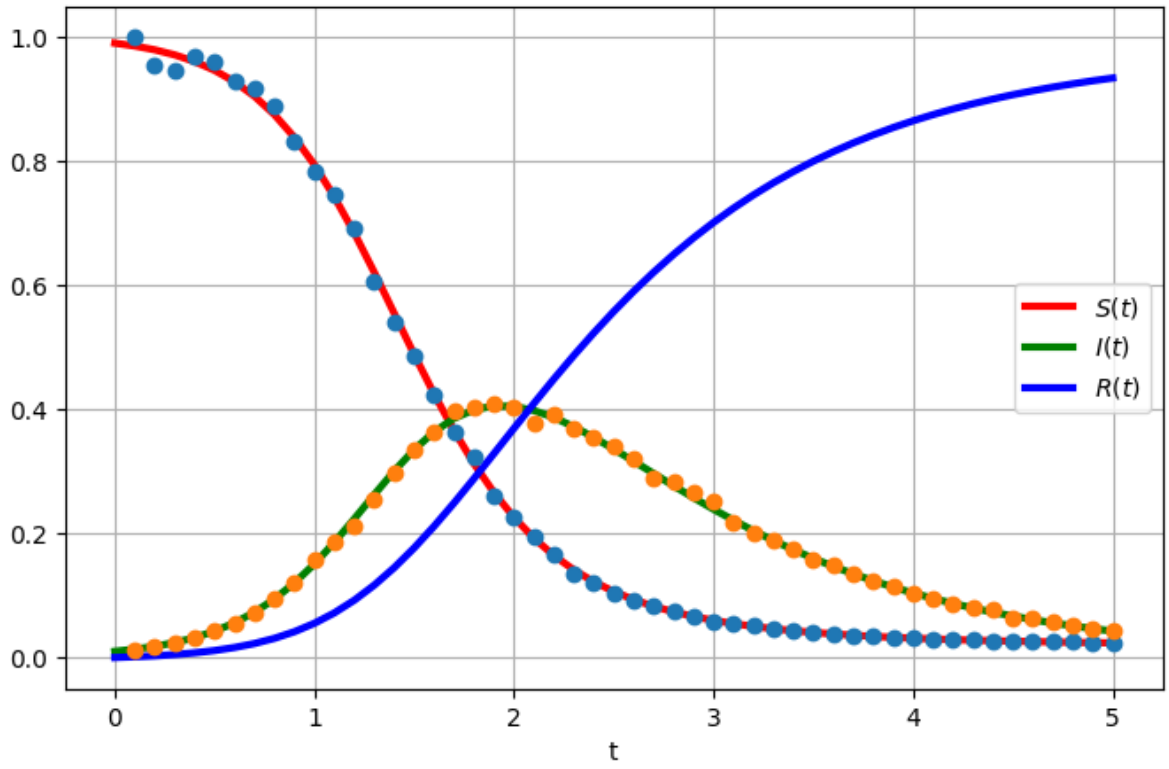
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
np.random.seed(7755)


def SIR(state, *args):
    """
    Parameters
    ----------
    state: array-like, shape (3,)
       Point of interest in three-dimensional space.
    *args: (sigma, lambda) : float
       Parameters defining the SIR dynamics.

    Returns
    -------
    state_dot : array, shape (3,)
```

```
        Values of the derivatives at *state*.
    """
    beta  = args[0]
    lambd = args[1]
    S, I, R = state # Unpack the state vector
    f = np.zeros(3)  # Derivatives
    f[0] = -beta*S*I
    f[1] =  beta*S*I - lambd*I
    f[2] =  lambd*I
    return f

def RK4(rhs, state, dt, *args):

    k1 = rhs(state,         *args)
    k2 = rhs(state+k1*dt/2, *args)
    k3 = rhs(state+k2*dt/2, *args)
    k4 = rhs(state+k3*dt,   *args)

    new_state = state + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
    return new_state
```

```
# parameters SIR
beta  = 4.0
lambd = 1.0
dt = 0.1
tm = 5
nt = int(tm/dt)
t  = np.linspace(0,tm,nt+1)
# initialize and solve
u0True = np.array([0.99, 0.01, 0]) # True initial conditions
#time integration
uTrue = np.zeros([nt+1,3])
uTrue[0,:] = u0True
for k in range(nt):
    uTrue[k+1,:] = RK4(SIR,uTrue[k,:], dt, beta, lambd)
# Observational model.  Lognormal likelihood.
yobs = np.random.lognormal(mean=np.log(uTrue[1::]), sigma=[0.02, 0.02, 0.])

# plot results
fig, ax = plt.subplots(nrows=1,ncols=1, figsize=(8,5))
ax.plot(t,uTrue[:,0], 'r', label='$S(t)$',  linewidth = 3)
ax.plot(t,uTrue[:,1], 'g', label='$I(t)$', linewidth = 3)
```

```
ax.plot(t,uTrue[:,2], 'b', label='$R(t)$', linewidth = 3)
ax.plot(t[1::], yobs[:,0:2], marker="o", linestyle="none")
ax.grid()
ax.legend()
ax.set_xlabel('t')
```

```
Text(0.5, 0, 't')
```



14.1 Ensemble KF for Data Assimilation

Here we will generalize the ensemble Kalman filter to take into account the possibility of sparse observations. This is usually the case in real-life systems, where observations are only available et fixed instants, and hence the filtering can only be applied at these times. Inbetween observations, the system evolves freely (without correction) according to its underlying state equation.

Suppose we have N_y measurements/observations at an interval of δt_y. This gives measurements for times $t_0 \leq t \leq t_m$, where $t_m = N_m \delta t_m$. This can be considered as the assimilation window.

The system then evolves freely for $t > t_m$ until some final forecast window time t_f. The state, or equation itself is simulated with a smaller δt and for a large number N_t steps, giving $t_f = N_t \delta t$. Usually, for real life systems, we will have

$$\delta t_m \geq \delta t, \quad N_m \leq N_t, \quad t_m \leq t_f.$$

For code testing, we make the simplifying (unrealistic) academic assumption that

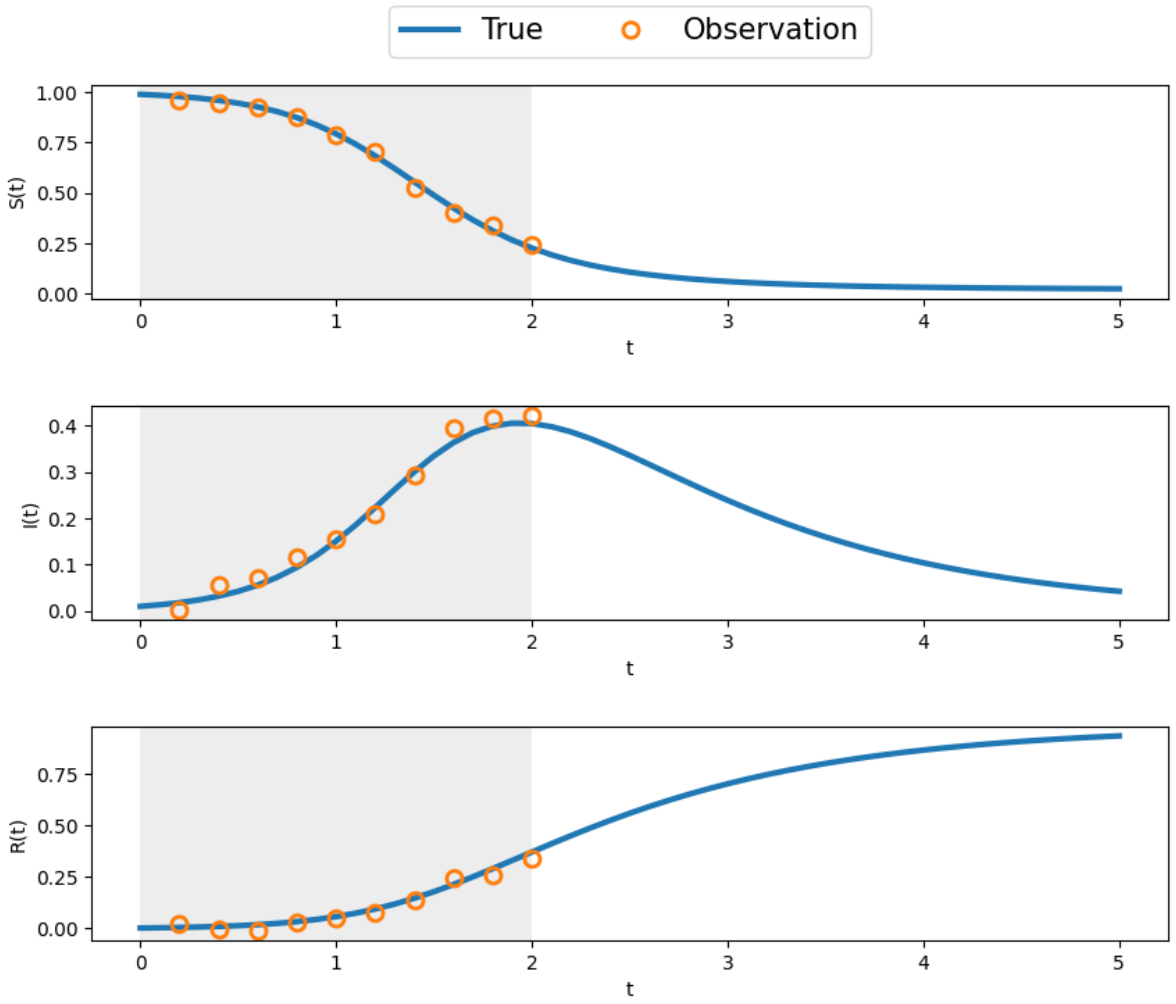
$$\delta t_m = \delta t, \quad N_m = N_t, \quad t_f = t_m.$$

This implies the availabilty of measurements at each (and every) time step. Note that in many of the previous examples, this was indeed the case.

```
def enKF_SIR_setup(dt, T, dt_m, T_m, sig_w, sig_v):
    """
    Prepare input (true state and observations) for the stochastic
    ensemble filter of the Lorenz63 system.

    Parameters:
        dt: time step for state evolution
        T:  time interval for state evolution
        dt_m: time interval between 2 measurements (can equal dt for dense observations)
        T_m: time interval for observations
        sig_w: state noise sd., cov. Q = sig_w**2 x np.eye(3)
        sig_v: measurement noise sd., cov. R = sig_v**2 x np.eye(3)
    """
    # parameters SIR
    beta  = 4.0
    lambd = 1.0
    dim_x = 3
    dim_y = 3
    # noise covariances
    Q  = sig_w**2 * np.eye(dim_x)
    R  = sig_v**2 * np.eye(dim_y)
    # measurement operator (identity here)
    def H(u):
        w = u
        return w

    # Solve system and generate noisy observations
    Nt = int(T/dt)     # number of time steps
    Nm = int(T_m/dt_m) # number of observations
```

```
    t = np.linspace(0, Nt, Nt+1) * dt # time vector
    ind_m = (np.linspace(int(dt_m/dt),int(T_m/dt),Nm)).astype(int) # obs. indices
    t_m = t[ind_m] # measurement time vector
    x0True = np.array([0.99, 0.01, 0]) # True initial conditions
    sqrt_Q = np.linalg.cholesky(Q) # noise std dev.
    sqrt_R = np.linalg.cholesky(R)
    # initialize (correctly!)
    xTrue = np.zeros([Nt+1, dim_x])
    xTrue[0, :] = x0True
    y = np.zeros((Nm, dim_y))
    km = 0 # index for measurement times
    y[0,:] = H(xTrue[0,:]) + sig_v * np.random.randn(dim_y)
    for k in range(Nt):
        w_k = sqrt_Q @ np.random.randn(dim_x)
        xTrue[k+1,:] = RK4(SIR, xTrue[k,:], dt, beta, lambd) #+ w_k
        if (km < Nm) and (k+1 == ind_m[km]):
            v_k = sqrt_R @ np.random.randn(dim_y)
            y[km,:]     = H(xTrue[k+1,:]) + v_k
            km = km + 1
    # plot state and measurements
    fig, ax = plt.subplots(nrows=3,ncols=1, figsize=(10,8))
    ax = ax.flat
    #t = T*dt
    for k in range(3):
        ax[k].plot(t,xTrue[:,k], label='True', linewidth = 3)
        ax[k].plot(t[ind_m],y[:,k], 'o', fillstyle='none', \
                   label='Observation', markersize = 8, markeredgewidth = 2)
        ax[k].set_xlabel('t')
        ax[k].axvspan(0, T_m, color='lightgray', alpha=0.4, lw=0)
    ax[0].legend(loc="center", bbox_to_anchor=(0.5,1.25),ncol =4,fontsize=15)
    ax[0].set_ylabel('S(t)')
    ax[1].set_ylabel('I(t)')
    ax[2].set_ylabel('R(t)')
    fig.subplots_adjust(hspace=0.5)

    return Q, R, xTrue, y, ind_m, Nt, Nm
```

```
Q, R, xTrue, y, ind_m, Nt, Nm = enKF_SIR_setup(dt=0.1, T=5, dt_m=0.2, T_m =2, sig_w=0.001, si
```




```
def enKF_SIR_DA(x0, P0, Q, R, y, ind_m, Nt, Nm, Ne=10):
    """
    Run DA of the SIR system using the stochastic
    ensemble filter with sparse observations in the DA
    window, defined by time index set `ind_m`.

    Parameters:

    """
    # parameters SIR
    beta  = 4.0
    lambd = 1.0
    def Hx(u):
```

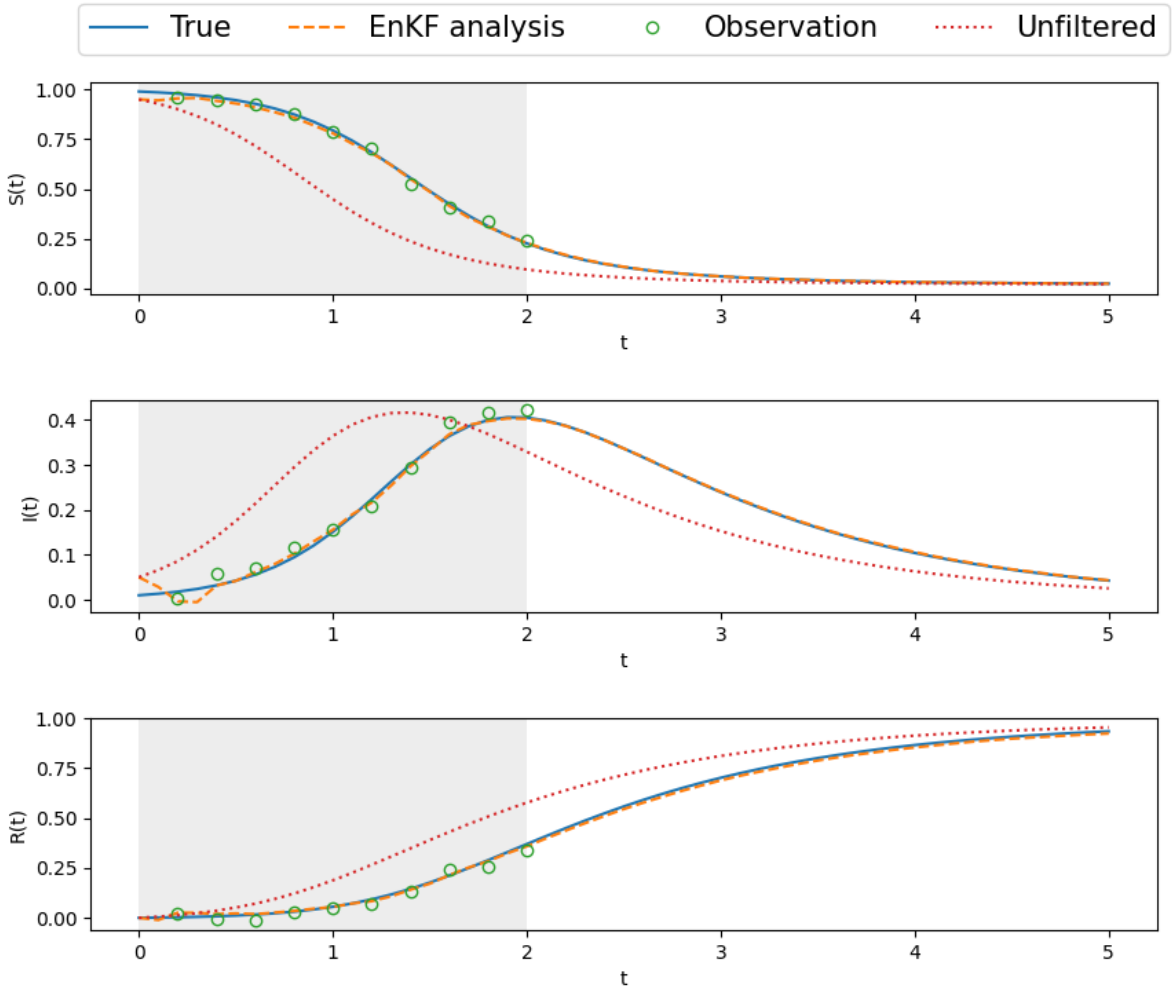
```
    w = u
    return w
Nx     = x0.shape[-1]
Ny     = y.shape[-1]
enkf_m = np.empty((Nt+1, Nx))
enkf_P = np.empty((Nt+1, Nx, Nx))
X      = np.empty((Nx, Ne))
Xf     = np.empty((Nx, Ne))
HXf    = np.empty((Ny, Nx))

X[:,:] = np.tile(x0, (Ne,1)).T  + np.linalg.cholesky(P0)@np.random.randn(Nx, Ne) # initia
P       = P0 # initial state covariance
enkf_m[0, :]    = x0
enkf_P[0, :, :] = P0

i_m = 0 # index for measurement times

for i in range(Nt):
    # ==== predict/forecast ====
    for e in range(Ne):
        w_i = np.linalg.cholesky(Q) @ np.random.randn(Nx)#, Ne)
        Xf[:,e] = RK4(SIR, X[:,e], dt,  beta, lambd) + w_i # predict state ensemble
    mX = np.mean(Xf, axis=1)        # state ensemble mean
    Xfp = Xf - mX[:, None]          # state forecast anomaly
    P = Xfp @ Xfp.T / (Ne - 1)      # predict covariance
    # ==== prepare analysis step =====
    if (i_m < Nm) and (i+1 == ind_m[i_m]):
        HXf = Hx(Xf)              # nonlinear observation
        mY = np.mean(HXf, axis=1)       # observation ensemble mean
        HXp = HXf - mY[:, None]         # observation anomaly
        S = (HXp @ HXp.T)/(Ne - 1) + R  # observation covariance
        K = linalg.solve(S, HXp @ Xfp.T, assume_a="pos").T  / (Ne - 1)  # Kalman gain
        # === perturb y and compute innovation ====
        ypert = y[i_m, :] +  (np.linalg.cholesky(R)@np.random.randn(Ny, Ne)).T
        d = ypert.T - HXf
        # ==== correct/analyze ====
        X[:,:] = Xf + K @ d         # update state ensemble
        mX = np.mean(X[:,:], axis=1)# state analysis ensemble mean
        Xap = X[:,:] - mX[:, None]  # state analysis anomaly
        P = Xap @ Xap.T / ( Ne - 1)     # update covariance
        i_m = i_m + 1
    else:
```

```
            X[:,:] = Xf  # when there is no obs, then state=forecast
        # ==== save ====
        enkf_m[i+1] = mX                # save KF state estimate (mean)
        enkf_P[i+1] = P                 # save KF error estimate (covariance)
    return enkf_m, enkf_P
```

```
# Initialize and run the analysis
sig_w = 0.0015
sig_v = 0.02
Q  = sig_w**2 * np.eye(3) #* 1.e-6 # for comparison with DT
R  = sig_v**2 * np.eye(3)

x0 = np.array([0.95, 0.05, 0]) # a little off [0.99, 0.01, 0]
sig_vv = 0.1
P0 = np.eye(3) * sig_vv**2 # Initial estimate covariance
Ne = 10
Xa, P = enKF_SIR_DA(x0, P0, Q, R, y, ind_m, Nt, Nm, Ne=10)
```

```
# Post-process and plot the results
# generate unfiltered state
Xb = np.empty((Nt+1, 3))
Xb[0,:] = x0
for i in range(Nt):
    Xb[i+1,:] =  RK4(SIR, Xb[i,:], dt, beta, lambd)
# plot state and measurements
t = np.linspace(0, Nt, Nt+1) * dt # time vector
T_m = 2.
fig, ax = plt.subplots(nrows=3,ncols=1, figsize=(10,8))
ax = ax.flat

for k in range(3):
    ax[k].plot(t,xTrue[:,k], label='True')#, linewidth = 3)
    ax[k].plot(t,Xa[:,k], '--', label='EnKF analysis')#, linewidth = 3)
    ax[k].plot(t[ind_m],y[:,k], 'o', fillstyle='none', \
               label='Observation')#, markersize = 8, markeredgewidth = 2)
    ax[k].plot(t,Xb[:,k], ':', label='Unfiltered')#, linewidth = 3)
    ax[k].set_xlabel('t')
    ax[k].axvspan(0, T_m, color='lightgray', alpha=0.4, lw=0)
ax[0].legend(loc="center", bbox_to_anchor=(0.5,1.25),ncol =4,fontsize=15)
ax[0].set_ylabel('S(t)')
ax[1].set_ylabel('I(t)')
ax[2].set_ylabel('R(t)')
```

```
fig.subplots_adjust(hspace=0.5)
```



14.2 Conclusion

The ensemble Kalman filter, even with sparse observations and a nonlinear system, does an excellent job of

1. tracking within the DA window
2. forecasting way beyond the window, whereas the unfiltered/unassimilated, freely evolving system deviates considerably, as is to be expected from the nonlinear SIR system.

15 Deterministic Ensemble Kalman Filters

We have seen the stochastic ensemble Kalman filter in the previous section. It has the advantage of great simplicity, but is known to have unreliable statistical behaviour and can suffer from sampling errors. In particular, it does not converge in the ensemble Kalman inversion (EKI) context - see next section. Filters that do not use (stochastically) perturbed observations are called *deterministic* filters. The family of *ensemble square root filters* are deterministic, in this sense. These filters exhibit exponential convergence in the EKI context.

15.1 The filtering problem.

Nonlinear state equation and observations are given by,

$$x_{k+1}^n = \mathcal{M}(x_k^n) + w_k^n, \quad n = 1, \dots, N_e, \tag{15.1}$$
$$y_{k+1} = \mathcal{H}(x_{k+1}) + v_{k+1}, \tag{15.2}$$

where the noise/error terms

$$w_k \sim \mathcal{N}(0, Q), \quad v_k \sim \mathcal{N}(0, R).$$

> **Filtering problem**
>
> Predict the optimal state from the noisy measurements.

15.2 Recall: Ensemble Kalman Filter (EnKF)

Just to recall the notation, here are the two steps (forecast-analysis, or predict-correct) of the EnKF.

Prediction/Forecast Step

- evolve each ensemble member forward

$$x_{k+1}^n = \mathcal{M}(x_k^n) + w_k^n, \quad n = 1, \ldots, N_e$$

- compute ensemble mean

$$\bar{x} = \frac{1}{N_e} \sum_{n=1}^{N_e} x_n^{\mathrm{f}}$$

- compute covariance

$$P^{\mathrm{f}} = \frac{1}{N_e - 1} X'^{\mathrm{f}} (X'^{\mathrm{f}})^{\mathrm{T}},$$

  where $X'^{\mathrm{f}} = x^{\mathrm{f}} - \bar{x}$ is the ensemble state perturbation/anomaly matrix.

Correction/Analysis Step

- compute the optimal Kalman gain

$$K = P^{\mathrm{f}} H^{\mathrm{T}} (H P^{\mathrm{f}} H^{\mathrm{T}} + R)^{-1}$$

- update the ensemble using perturbed observations

$$x_n^{\mathrm{a}} = x_n^{\mathrm{f}} + K(y_n + \epsilon_y - H x_n^{\mathrm{f}}), \quad n = 1, \ldots, N_e,$$

  where ϵ_n is the stochastic perturbation of the observations y.

15.3 Ensemble Square Root Filters (EnSRF)

> **Idea**
>
> Update the ensemble to preserve a covariance that is consistent with the KF *theoretical* covariance
>
> $$P_n^{\mathrm{a}} = (I - K_n H) P_n^{\mathrm{f}}.$$

Recall the fully nonlinear formulation of the Kalman gain in terms of the state and observation anomalies described in Section 11.2,

$$K = \mathbf{X}'^{\mathrm{f}} (\mathbf{Y}'^{\mathrm{f}})^{\mathrm{T}} S^{-1},$$

where

$$S = \mathbf{Y}'^{\mathrm{f}} (\mathbf{Y}'^{\mathrm{f}})^{\mathrm{T}} + R.$$

Then, to compute the posterior variance, we need to evaluate

$$P_n^{\mathrm{a}} = \mathbf{X}'^{\mathrm{a}}(\mathbf{X}'^{\mathrm{a}})^{\mathrm{T}}.$$

So supposing there is a transform matrix, T, such that

$$\mathbf{X}'^{\mathrm{a}} = \mathbf{X}'^{\mathrm{f}}T,$$

we can substitute in the definition of P_n^{a} to obtain

$$P_n^{\mathrm{a}} = \mathbf{X}'^{\mathrm{f}}T(\mathbf{X}'^{\mathrm{f}}T)^{\mathrm{T}} \tag{15.3}$$
$$= \mathbf{X}'^{\mathrm{f}}(TT^{\mathrm{T}})(\mathbf{X}'^{\mathrm{f}})^{\mathrm{T}}. \tag{15.4}$$

And, on the other hand, for the consistency, we require

$$(I - K_nH)P_n^{\mathrm{f}} = \mathbf{X}'^{\mathrm{f}}\left[I - (\mathbf{Y}'^{\mathrm{f}})^{\mathrm{T}}S^{-1}\mathbf{Y}'^{\mathrm{f}}\right](\mathbf{X}'^{\mathrm{f}})^{\mathrm{T}},$$

where we have used the relations

$$K = P^{\mathrm{f}}H^{\mathrm{T}}(HP^{\mathrm{f}}H^{\mathrm{T}} + R)^{-1}, \tag{15.5}$$
$$P_n^{\mathrm{f}} = \mathbf{X}'^{\mathrm{f}}(\mathbf{X}'^{\mathrm{f}})^{\mathrm{T}} \tag{15.6}$$
$$\mathbf{Y}'^{\mathrm{f}} = H\mathbf{X}'^{\mathrm{f}}. \tag{15.7}$$

Hence, T must satisfy the so-called *square root condition*,

$$TT^{\mathrm{T}} = I - (\mathbf{Y}'^{\mathrm{f}})^{\mathrm{T}}S^{-1}\mathbf{Y}'^{\mathrm{f}} \tag{15.8}$$
$$= I - (\mathbf{Y}'^{\mathrm{f}})^{\mathrm{T}}\left[\mathbf{Y}'^{\mathrm{f}}(\mathbf{Y}'^{\mathrm{f}})^{\mathrm{T}} + R\right]^{-1}\mathbf{Y}'^{\mathrm{f}} \tag{15.9}$$

We can replace the inversion of S by the much simpler and more stable inversion of the diagonal measurement error covariance R using the Sherman-Woodbury-Morrison formula,

$$(A + UCV^{\mathrm{T}})^{-1} = A^{-1} - A^{-1}V(C^{-1} + VA^{-1}U)^{-1}VA^{-1}.$$

Identifying $A = I$, $C = R^{-1}$, $U = \mathbf{Y}^{\mathrm{T}}$, $V = \mathbf{Y}$, we obtain the simpler form of the square root condition

$$TT^{\mathrm{T}} = \left[I + (\mathbf{Y}'^{\mathrm{f}})^{\mathrm{T}}R^{-1}\mathbf{Y}'^{\mathrm{f}}\right]^{-1}. \tag{15.10}$$

This form is the basis of the ETKF, or *transform filter.*

Finally, the square root T can be obtained from the eigenvalue factorization,

$$TT^{\mathrm{T}} = (U\Sigma U^{\mathrm{T}})^{-1} \tag{15.11}$$

and thus

$$T = U\Sigma^{-1/2}U^{\mathrm{T}}.$$

Note that T is not unique, since for any orthogonal matrix $\tilde{U}$, the product $T\tilde{U}$ will also satisfy the square root condition. This leads, in principle, to a large number of alternative forms for T and the resulting square root filters.

It can be shown that, in general, this process can yield a biased and overconfident estimator of the posterior covariance. In order to remedy this, it suffices to ensure that T is symmetric, which is the case in the above derivation.

15.4 ETKF Algorithm

The ensemble transform Kalman filter is one possible implementation of a square root filter. For greater generality, we will use the consistent formulation of (Sanita Vetra-Carvalho and Beckers 2018). The analysis update is given by the general, linear transformation

$$\mathbf{X}^{\mathrm{a}} = \overline{\mathbf{X}}^{\mathrm{f}} + \mathbf{X}'^{\mathrm{f}}(\overline{W} + W'),$$

where the anomaly weight matrix, W', is computed from the square root condition (Equation 15.10), and the weight matrix, $\overline{W}$, is obtained from the formula for the Kalman gain matrix expressed in terms of the square root factorization (Equation 15.11). Each square root filter will just have different forms of these two matrices—all the rest will be the same.

The steps of the ETKF algorithm are as follows.

1. Forecast the state.
2. Compute the means and anomalies of the state forecast and the observations.
3. Setup the square root condition matrix and compute it's eigenvalue decomposition.
4. Compute the two weight matrices.
5. Update the state analysis.

Steps 3 and 4:

$$U\Sigma U^{\mathrm{T}} = I + (\mathbf{Y}'^{\mathrm{f}})^{\mathrm{T}} R^{-1} \mathbf{Y}'^{\mathrm{f}},$$

$$W' = U\Sigma^{-1/2}U^{\mathrm{T}}$$

and

$$\overline{W} = U\Sigma^{-1}U^{\mathrm{T}}(\mathbf{Y}'^{\mathrm{f}})^{\mathrm{T}} R^{-1} D,$$

where the innovation (observations - average)

$$D \doteq Y - \overline{\mathbf{Y}},$$

with

$$\overline{\mathbf{Y}} \doteq \overline{\mathcal{H}(\mathbf{X})}.$$

15.5 ETKF in practice

There are a number of possible modifications that render the filter

- unbiased,
- non-collapsing,
- computationally more stable,
- computationally cheaper.

One pathway is to scale the forecast observation ensemble perturbation matrix Y^{f} so as to normalize the standard deviation—which then equals one—and thus reduce loss of accuracy due to roundoff errors.

A second path is to avoid the eigenvalue decompostion of TT^{T} and replace it by an SVD of T alone. This is particularly advantageous in high dimensions and in the presence of bad conditioning.

16 Example 4: ETKF for Lorenz63 system

In this example, we will use the ensemble transform Kalman filter on the Lorenz system of ordinary differential equations. Recall: the Lorenz-63 system is given by

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= -\sigma(x-y),\\ \frac{\mathrm{d}y}{\mathrm{d}t} &= \rho x - y - xz,\\ \frac{\mathrm{d}z}{\mathrm{d}t} &= xy - \beta z,\end{aligned}$$

where $x = x(t)$, $y = y(t)$, $z = z(t)$ and σ (ratio of kinematic viscosity divided by thermal diffusivity), ρ (measure of stability) and β (related to the wave number) are parameters.

Chaotic behavior is obtained when the parameters are chosen as

$$\sigma = 10, \quad \rho = 28, \quad \beta = 8/3.$$

Its solution is very sensitive to the parameters and the initial conditions and a small difference in these values can lead to a very different solution. This is the basis of its ill-posedness. This equation is a simplified model for atmospheric convection and is an excellent example of the lack of predictability. It is ill-posed in the sense of Hadamard. In fact the solution switches between two stable orbits, around the points

$$\left(\sqrt{\beta(\rho-1)}, \sqrt{\beta(\rho-1)}, \rho-1\right), \quad \left(-\sqrt{\beta(\rho-1)}, -\sqrt{\beta(\rho-1)}, \rho-1\right).$$

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
np.random.seed(1)


def Lorenz63(state, *args):
    """
```

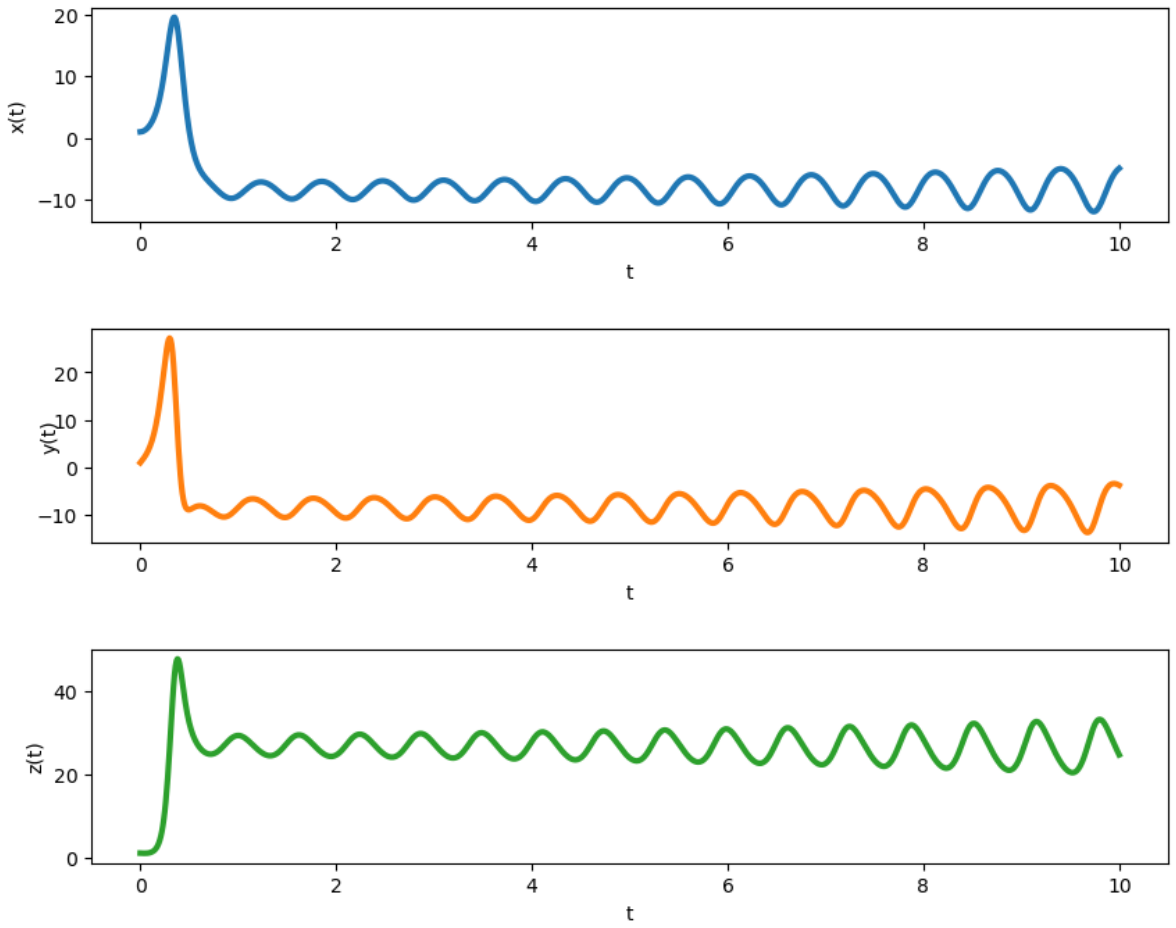
```
    Parameters
    ----------
    state : array-like, shape (3,)
       Point of interest in three-dimensional space.
    *args (sigma, rho, beta) : float
       Parameters defining the Lorenz attractor.

    Returns
    -------
    state_dot : array, shape (3,)
       Values of the Lorenz attractor's derivatives at *state*.
    """
    sigma = args[0]
    rho = args[1]
    beta = args[2]
    x, y, z = state #Unpack the state vector
    f = np.zeros(3) #Derivatives
    f[0] = sigma * (y - x)
    f[1] = x * (rho - z) - y
    f[2] = x * y - beta * z
    return f

def RK4(rhs, state, dt, *args):

    k1 = rhs(state,         *args)
    k2 = rhs(state+k1*dt/2, *args)
    k3 = rhs(state+k2*dt/2, *args)
    k4 = rhs(state+k3*dt,   *args)

    new_state = state + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
    return new_state
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
np.random.seed(1)


def Lorenz63(state, *args):
    """
    Parameters
    ----------
```

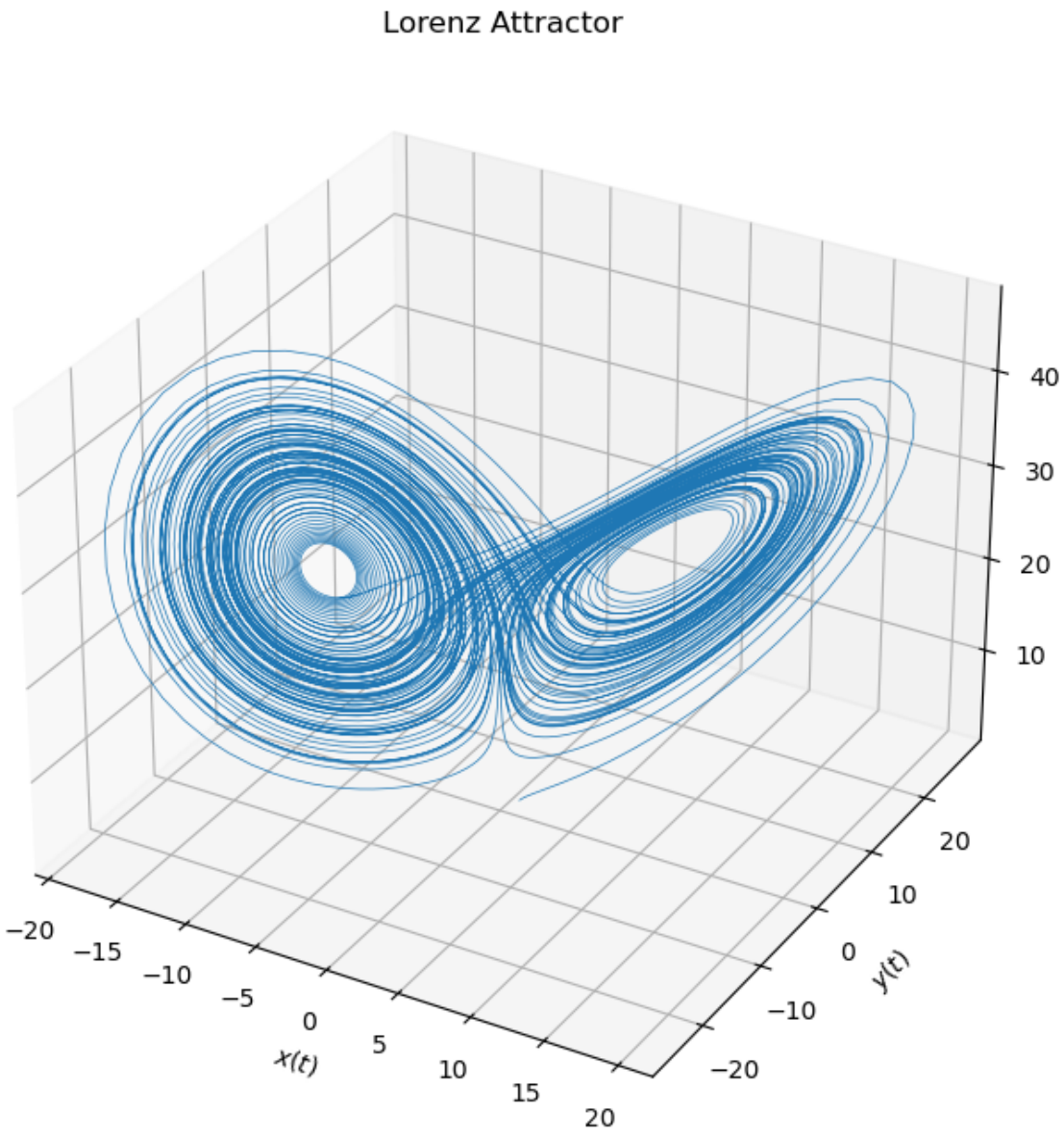
```
    state : array-like, shape (3,)
       Point of interest in three-dimensional space.
    *args (sigma, rho, beta) : float
       Parameters defining the Lorenz attractor.

    Returns
    -------
    state_dot : array, shape (3,)
       Values of the Lorenz attractor's derivatives at *state*.
    """
    sigma = args[0]
    rho = args[1]
    beta = args[2]
    x, y, z = state #Unpack the state vector
    f = np.zeros(3) #Derivatives
    f[0] = sigma * (y - x)
    f[1] = x * (rho - z) - y
    f[2] = x * y - beta * z
    return f

def RK4(rhs, state, dt, *args):

    k1 = rhs(state,         *args)
    k2 = rhs(state+k1*dt/2, *args)
    k3 = rhs(state+k2*dt/2, *args)
    k4 = rhs(state+k3*dt,   *args)

    new_state = state + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
    return new_state
```

```
# parameters Lorenz63
sigma = 10.0
rho  = 28.0
beta = 8.0/3.0
dt = 0.01
tm = 10
nt = int(tm/dt)
t  = np.linspace(0,tm,nt+1)
# initialize and solve
u0True = np.array([1,1,1]) # True initial conditions
#time integration
uTrue = np.zeros([nt+1,3])
```

```
uTrue[0,:] = u0True
for k in range(nt):
    uTrue[k+1,:] = RK4(Lorenz63,uTrue[k,:], dt, sigma, rho, beta)

# plot results
fig, ax = plt.subplots(nrows=3,ncols=1, figsize=(10,8))
ax = ax.flat

prop_cycle = plt.rcParams['axes.prop_cycle']
colors = prop_cycle.by_key()['color']

for k in range(3):
    ax[k].plot(t,uTrue[:,k], label='True', color = colors[k], linewidth = 3)
    ax[k].set_xlabel('t')

ax[0].set_ylabel('x(t)', labelpad=5)
ax[1].set_ylabel('y(t)', labelpad=-12)
ax[2].set_ylabel('z(t)')
fig.subplots_adjust(hspace=0.5)
```




```
nt = 10000 # need longer time to see the attractor
uTrue = np.zeros([nt+1,3])
uTrue[0,:] = u0True

for k in range(nt):
    uTrue[k+1,:] = RK4(Lorenz63,uTrue[k,:], dt,sigma, rho, beta)

# Plot
ax = plt.figure(figsize=(10,8)).add_subplot(projection='3d')
ax.plot(*uTrue.T, lw=0.5)
ax.set_xlabel("$x(t)$")
ax.set_ylabel("$y(t)$")
ax.set_zlabel("$z(t)$")
ax.set_title("Lorenz Attractor")
plt.show()
```



16.1 Ensemble KF for Data Assimilation

Here we will generalize the ensemble Kalman filter to take into account the possibility of sparse observations. This is usually the case in real-life systems, where observations are only available et fixed instants, and hence the filtering can only be applied at these times. Inbetween observations, the system evolves freely (without correction) according to its underlying state

equation.

Suppose we have N_y measurements/observations at an interval of δt_y. This gives measurements for times $t_0 \leq t \leq t_m$, where $t_m = N_m \delta t_m$. This can be considered as the assimilation window. The system then evolves freely for $t > t_m$ until some final forecast window time t_f. The state, or equation itself is simulated with a smaller δt and for a large number N_t steps, giving $t_f = N_t \delta t$. Usually, for real life systems, we will have

$$\delta t_m \geq \delta t, \quad N_m \leq N_t, \quad t_m \leq t_f.$$

For code testing, we make the simplifying (unrealistic) academic assumption that

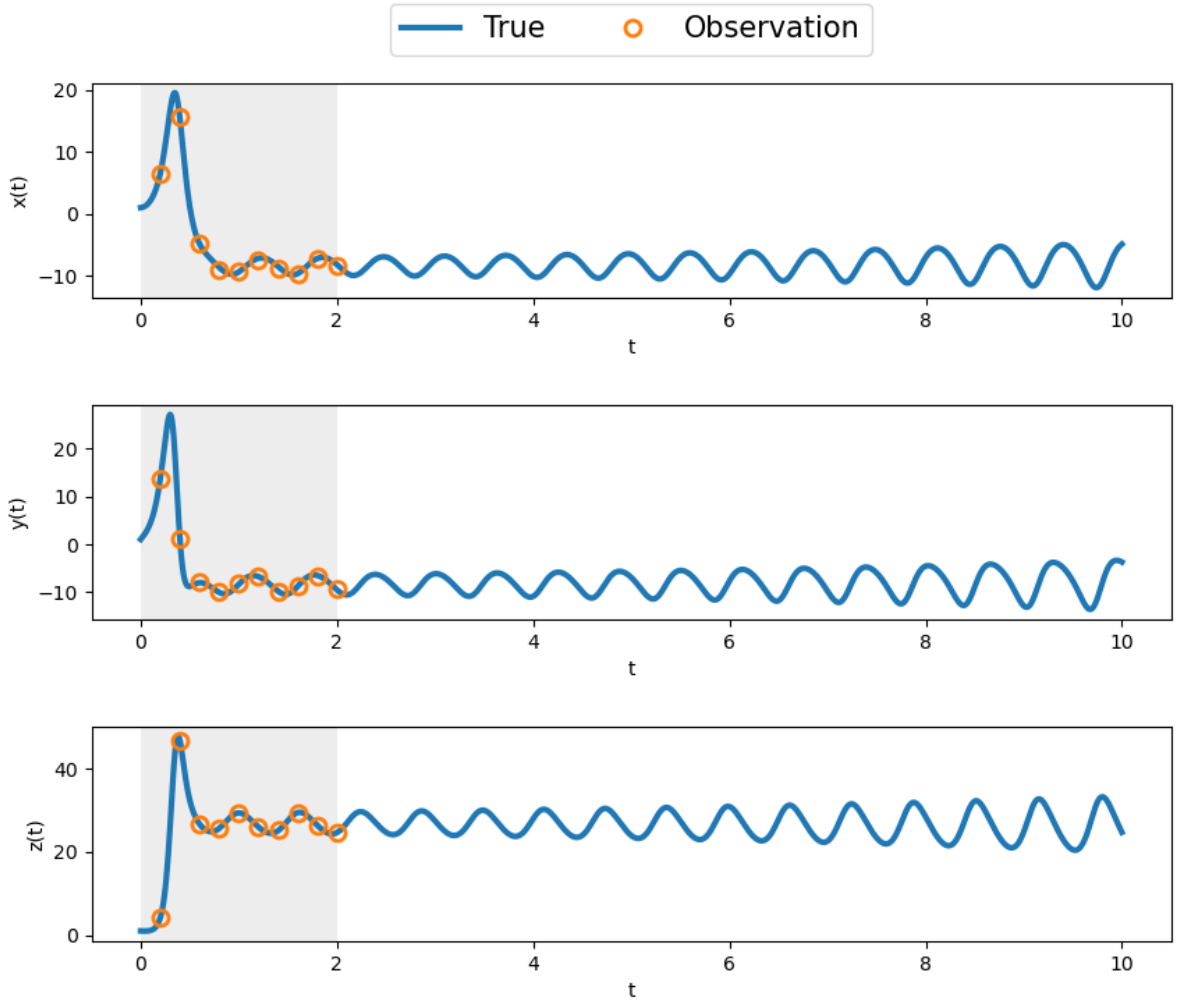
$$\delta t_m = \delta t, \quad N_m = N_t, \quad t_f = t_m.$$

This implies the availabilty of measurements at each (and every) time step. Note that in many of the previous examples, this was indeed the case.

```
def enKF_Lorenz63_setup(dt, T, dt_m, T_m, sig_w, sig_v):
    """
    Prepare input (true state and observations) for the stochastic
    ensemble filter of the Lorenz63 system.

    Parameters:
        dt: time step for state evolution
        T:  time interval for state evolution
        dt_m: time interval between 2 measurements (can equal dt for dense observations)
        T_m: time interval for observations
        sig_w: state noise sd., cov. Q = sig_w**2 x np.eye(3)
        sig_v: measurement noise sd., cov. R = sig_v**2 x np.eye(3)
    """
    # parameters Lorenz63
    sigma = 10.0
    rho   = 28.0
    beta = 8.0/3.0
    dim_x = 3
    dim_y = 3
    # noise covariances
    Q  = sig_w**2 * np.eye(dim_x)
    R  = sig_v**2 * np.eye(dim_y)
    # measurement operator (identity here)
    def H(u):
        w = u
```

```
    return w

# Solve system and generate noisy observations
Nt = int(T/dt)     # number of time steps
Nm = int(T_m/dt_m) # number of observations
t = np.linspace(0, Nt, Nt+1) * dt # time vector (including 0 and T)
ind_m = (np.linspace(int(dt_m/dt),int(T_m/dt),Nm)).astype(int) # obs. indices
t_m = t[ind_m] # measurement time vector
x0True = np.array([1.0, 1.0, 1.0]) # True initial conditions
sqrt_Q = np.linalg.cholesky(Q) # noise std dev.
sqrt_R = np.linalg.cholesky(R)
# initialize (correctly!)
xTrue = np.zeros([Nt+1, dim_x])
xTrue[0, :] = x0True
y = np.zeros((Nm, dim_y))
km = 0 # index for measurement times
y[0,:] = H(xTrue[0,:]) + sig_v * np.random.randn(dim_y)
for k in range(Nt):
    w_k = sqrt_Q @ np.random.randn(dim_x)
    xTrue[k+1,:] = RK4(Lorenz63,xTrue[k,:], dt,sigma, rho, beta) #+ w_k
    if (km < Nm) and (k+1 == ind_m[km]):
        v_k = sqrt_R @ np.random.randn(dim_y)
        y[km,:]     = H(xTrue[k+1,:]) + v_k
        km = km + 1
# plot state and measurements
fig, ax = plt.subplots(nrows=3,ncols=1, figsize=(10,8))
ax = ax.flat
#t = T*dt
for k in range(3):
    ax[k].plot(t,xTrue[:,k], label='True', linewidth = 3)
    ax[k].plot(t[ind_m],y[:,k], 'o', fillstyle='none', \
               label='Observation', markersize = 8, markeredgewidth = 2)
    ax[k].set_xlabel('t')
    ax[k].axvspan(0, T_m, color='lightgray', alpha=0.4, lw=0)
ax[0].legend(loc="center", bbox_to_anchor=(0.5,1.25),ncol =4,fontsize=15)
ax[0].set_ylabel('x(t)')
ax[1].set_ylabel('y(t)')
ax[2].set_ylabel('z(t)')
fig.subplots_adjust(hspace=0.5)

return Q, R, xTrue, y, ind_m, Nt, Nm
```

```
Q, R, xTrue, y, ind_m, Nt, Nm = enKF_Lorenz63_setup(dt=0.01, T=10, dt_m=0.2, T_m =2, sig_w=0
```



16.2 Ensemble Transform Kalman Filter

Here we implement the Livings/Dance unbiased variant of the deterministic ensemble square root filter (Sanita Vetra-Carvalho and Beckers 2018), where

- the forecast anomaly, $\mathbf{Y}'$, is normalized to have unit variance,
- an SVD factorization is used to compute the transformation matrix, T.

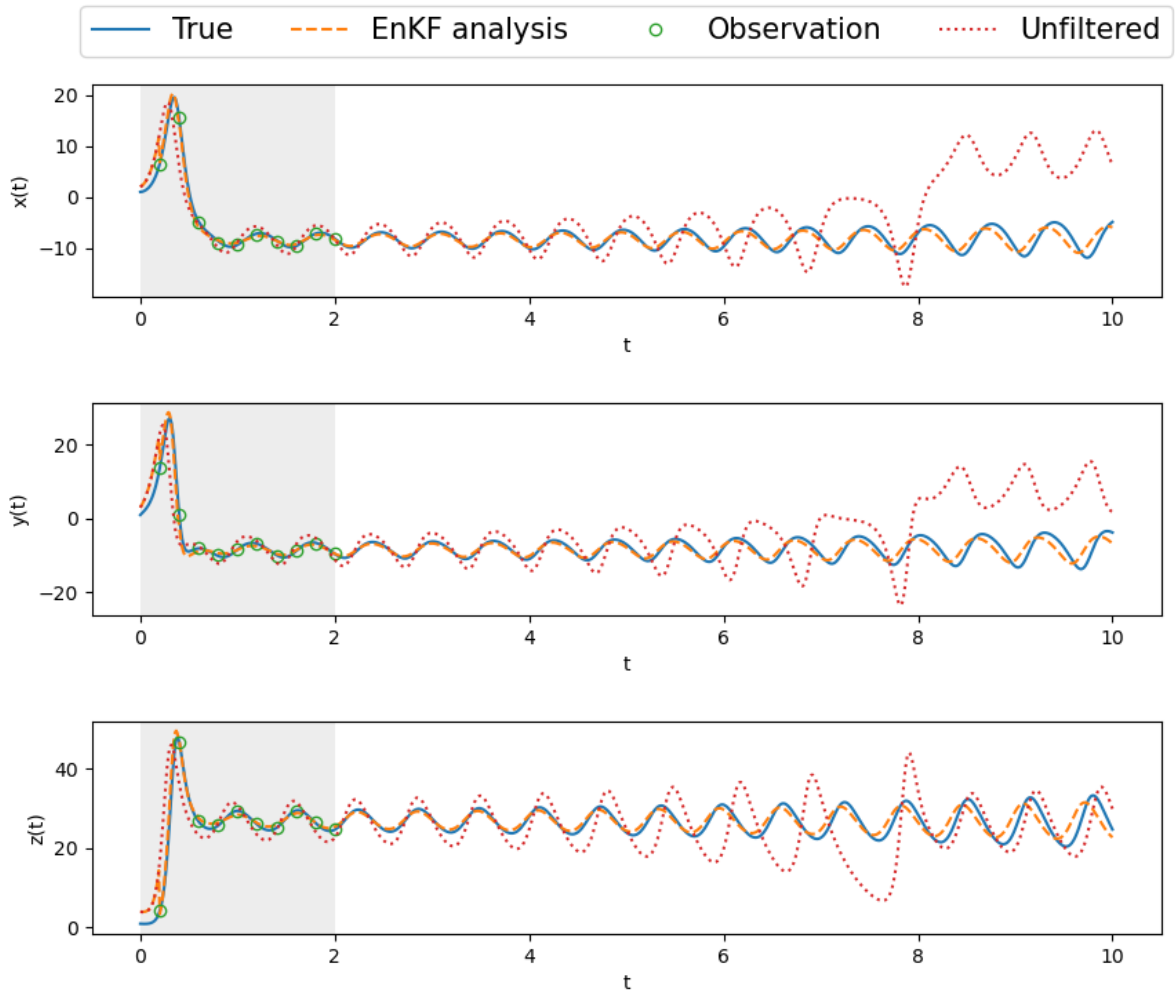
```
def eTKF_Lorenz63_DA(x0, P0, Q, R, Y, ind_m, Nt, Nm, Ne=10):
    """
    Run DA on the Lorenz63 system using the deterministic
    ensemble square root filter (ETKF) with sparse observations
    in the DA window, defined by time index set `ind_m`.

    Parameters:

    """
    # parameters Lorenz63
    sigma = 10.0
    rho   = 28.0
    beta  = 8.0/3.0
    def Hx(u):
        w = u
        return w
    Nx     = x0.shape[-1]
    Ny     = y.shape[-1]
    enkf_m = np.empty((Nt+1, Nx))
    enkf_P = np.empty((Nt+1, Nx, Nx))
    Xa     = np.empty((Nx, Ne))
    Xf     = np.empty((Nx, Ne))
    HXf    = np.empty((Ny, Nx))

    Xa[:,:] = np.tile(x0, (Ne,1)).T  + np.linalg.cholesky(P0)@np.random.randn(Nx, Ne) # initi
    P       = P0 # initial state covariance
    enkf_m[0, :]    = x0
    enkf_P[0, :, :] = P0

    i_m = 0 # index for measurement times

    for i in range(Nt):
        # ==== predict/forecast ====
        for e in range(Ne):
            w_i = np.linalg.cholesky(Q) @ np.random.randn(Nx)#, Ne)
            Xf[:,e] = RK4(Lorenz63, Xa[:,e], dt, sigma, rho, beta) + w_i # predict state ense
        mX  = np.mean(Xf, axis=1)                   # state ensemble mean
        Xfp = (Xf - mX[:, None])#/ np.sqrt(Ne - 1) # state forecast anomaly
        P   = Xfp @ Xfp.T  / (Ne - 1)               # predict covariance
        # ==== prepare analysis step =====
        if (i_m < Nm) and (i+1 == ind_m[i_m]):
            Rd  = np.diag(R)  # Observation error (diagonal)
```

```
            HXf = Hx(Xf)                                    # nonlinear observation
            mY  = np.mean(HXf, axis=1)                      # observation ensemble mean
            HXp = (HXf - mY[:, None]) #/ np.sqrt(Ne - 1) # observation anomaly
            #Scaling of obs. perturbations (Livings, 2005), for numerical stability
            S_hat = np.diag(1/np.sqrt(Rd)) @ HXp / np.sqrt(Ne - 1)
            #svd of S_hat transposed
            U, s, Vh = np.linalg.svd(S_hat.T, full_matrices=False)
            #perturbation weight
            mat = np.diag(1 / np.sqrt(1 + np.square(s)))
            Wp1 = mat @ U .T
            Wp  = U @ Wp1
            #innovation
            d = Y[i_m, :].T - mY
            #mean weight
            D = np.linalg.inv(np.sqrt(R)) @ d
            D2 = Vh @ D
            D3 = np.diag(1/(1+np.square(s))) @ np.diag(s) @ D2
            wm = U @ D3  / np.sqrt(Ne - 1)
            #add pert and mean (!row-major formulation in Python!)
            W = Wp + wm[:,None]
            # ==== correct/analyze ====
            Xa = mX[:,None] + Xfp @ W
            mX = np.mean(Xa[:,:], axis=1)                     # state analysis ensemble mean
            Xap = (Xa[:,:] - mX[:, None]) / np.sqrt(Ne - 1) # state analysis anomaly
            P = Xap @ Xap.T                                   # update covariance
            i_m = i_m + 1
        else:
            Xa[:,:] = Xf  # when there is no obs, then state=forecast
        # ==== save ====
        enkf_m[i+1] = mX                # save KF state estimate (mean)
        enkf_P[i+1] = P                 # save KF error estimate (covariance)
    return enkf_m, enkf_P
```

```
# Initialize and run the analysis
sig_w = 0.0015
sig_v = 0.15
Q  = sig_w**2 * np.eye(3) #* 1.e-6 # for comparison with DT
R  = sig_v**2 * np.eye(3)

x0 = np.array([2., 3., 4.]) # a little off wrt. [1,1,1]
sig_vv = 0.1
P0 = np.eye(3) * sig_vv**2 # Initial estimate covariance
```

```
Ne = 10
Xa, P = eTKF_Lorenz63_DA(x0, P0, Q, R, y, ind_m, Nt, Nm, Ne=10)
```

```
# Post-process and plot the results
# generate unfiltered state
Xb = np.empty((Nt+1, 3))
Xb[0,:] = x0
for i in range(Nt):
    Xb[i+1,:] =  RK4(Lorenz63, Xb[i,:], dt, sigma, rho, beta)
# plot state and measurements
t = np.linspace(0, Nt, Nt+1) * dt # time vector
T_m = 2.
fig, ax = plt.subplots(nrows=3,ncols=1, figsize=(10,8))
ax = ax.flat

for k in range(3):
    ax[k].plot(t,xTrue[:,k], label='True')#, linewidth = 3)
    ax[k].plot(t,Xa[:,k], '--', label='EnKF analysis')#, linewidth = 3)
    ax[k].plot(t[ind_m],y[:,k], 'o', fillstyle='none', \
               label='Observation')#, markersize = 8, markeredgewidth = 2)
    ax[k].plot(t,Xb[:,k], ':', label='Unfiltered')#, linewidth = 3)
    ax[k].set_xlabel('t')
    ax[k].axvspan(0, T_m, color='lightgray', alpha=0.4, lw=0)
ax[0].legend(loc="center", bbox_to_anchor=(0.5,1.25),ncol =4,fontsize=15)
ax[0].set_ylabel('x(t)')
ax[1].set_ylabel('y(t)')
ax[2].set_ylabel('z(t)')
fig.subplots_adjust(hspace=0.5)
```



16.3 Conclusion

The ensemble transform Kalman filter, even with very sparse observations and a chaotic system, does an excellent job of

1. tracking within the DA window,
2. forecasting way beyond the window, whereas the unfiltered/unassimilated, freely evolving system deviates considerably, as is to be expected from the chaotic Lorenz system.

From $t \approx 8$, the KF starts to deviate very slightly, mostly a phase difference. However, in real situations, there will already be new measurements available by this time. Then the KF analysis will kick in again, and rectify the trajectory.

Part V

Inverse Problems

17 Bayesian Inversion

17.1 Introduction

Bayesian approaches, provide a foundation for inference from noisy and limited data, a natural mechanism for regularization in the form of prior information, and in very general cases—e.g., non-linear forward operators, non-Gaussian errors—a quantitative assessment of uncertainty in the results. Indeed, the output of Bayesian inference is not a single value for the quantity of interest, but a probability distribution that summarizes all available information about this quantity, be it a vector of parameters or a function (i.e., a signal or spatial field). Exploration of this posterior distribution—and thus estimating means, higher moments, and marginal densities of the inverse solution—may require repeated evaluations of the forward operator. For complex physical models and high-dimensional model spaces, this can be computationally prohibitive.

Bayesian inference provides an attractive setting for the solution of inverse problems. Measurement errors,forward model uncertainties, and complex prior information can all be combined to yield a rigorous and quantitative assessment of uncertainty in the solution of the inverse problem.

17.2 Theory

A Bayesian Inverse Problem (BIP) is defined as follows:

- Given:
  - observational data and their uncertainties,
  - a (possibly stochastic) forward model that maps model parameters to observations,
  - and a prior probability distribution on model parameters that encodes any prior knowledge or assumptions about the parameters.
- Find:
  - the posterior probability distribution of the parameters conditioned on the observational data.

This probability density function (pdf) is defined as the Bayesian solution of the inverse problem. The posterior distribution assigns to any candidate set of parameter fields our belief (expressed as a probability) that a member of this candidate set is the "true'' parameter field that gave rise to the observed data.

Of course, all of this is summarized in Bayes' theorem, expressed as folllows.

What can be said about the value of an unknown or poorly known variable/parameter, θ, that represents the parameters of the system, if we have some measured data $\mathcal{D}$ and a model $\mathcal{M}$ of the underlying mechanism that generated the data? This is precisely the Bayesian context, where we seek a quantification of the uncertainty in our knowledge of the parameters that according to Bayes' Theorem takes the form

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta)p(\theta)}{p(\mathcal{D})} = \frac{p(\mathcal{D} \mid \theta)p(\theta)}{\int_\theta p(\mathcal{D} \mid \theta)p(\theta)}. \tag{17.1}$$

Here, the physical model that generates the data is represented by the conditional probability (also known as the *likelihood*) $p(\mathcal{D} \mid \theta)$, and the prior knowledge of the system by the term $p(\theta)$. The denominator is considered as a normalizing factor and represents the total probability of $\mathcal{D}$. From these we can then calculate the resulting *posterior probability*, $p(\theta \mid \mathcal{D})$. A generalization can include a model, $\mathcal{M}$, for the parameters. In this case (6.1) can be written as

$$p(\theta \mid \mathcal{D}, \mathcal{M}) = \frac{p(\mathcal{D} \mid \theta, \mathcal{M})p(\theta \mid \mathcal{M})}{p(\mathcal{D} \mid \mathcal{M})}. \tag{17.2}$$

This is depicted in the flowchart.

17.3 Inverse Problem

We consider the inverse problem of recovering an unknown function u from known data y related by

$$y = \mathcal{G}(u) + \eta,$$

where $\mathcal{G}$ denotes the forward model from inputs to observables, $\eta \sim \mathcal{N}(0, \Gamma)$ represents the model error and observation noise, with Γ a positive definite noise covariance matrix. In the Bayesian approach, u is a random variable with *prior* distribution $p_u(u)$. Then, the Bayesian solution of the inverse problem is the *posterior* distribution $p_{u|y}(u)$ of u given y, which by formal application of Bayes' Theorem can be characterized by

$$p_{u|y}(u) \propto \exp\left(-\Phi(u; y)\right) p_u(u),$$

with

$$\Phi(u;y) \doteq \frac{1}{2}\left\|y - \mathcal{G}(u)\right\|_{\Gamma}^{2}.$$

TBC...

17.4 Examples

18 Ensemble Kalman Inversion (EKI)

Estimating the statistics of the state of a dynamical system from partial/sparse and noisy measurements is both mathematically challenging and the subject of numerous real-life applications. These can have great societal impact, such as in the cases of numerical weather forecasting, epidemic evolution modelling, just to name a few.

Ensemble Kalman filtering methodology is known to be effective in high dimensions and in nonlinear settings, as opposed to regular Kalman filters. Moreover, ensemble Kalman filters can be used to solve quite general inverse problems.

The theoretical approach, known as EKI, was originally formulated in (Iglesias, Law, and Stuart 2013) and refined in the review (Calvello, Reich, and Stuart 2022) (to appear in Acta Numerica 2025). A shorter version can be found in the recent preprint (Huang et al. 2022), where extensive numerical tests were performed.

18.1 Properties of EKI

- a derivative- and gradient-free optimization method;
- full uncertainty quantification;
- relatively few evaluations of the (possibly expensive) forward model;
- robust to noise in the evaluation of the model.

18.2 Formulation

Recall the inverse problem: recover an unknown function θ from known data y related by

$$y = \mathcal{G}(\theta) + \eta,$$

where $\mathcal{G}$ denotes the forward model from inputs to observables, $\eta \sim \mathcal{N}(0, \Sigma_\eta)$ represents the model error and observation noise, with Σ_η a positive definite noise covariance matrix.

Conceptually, the idea of EKI is very simple. Just apply the standard EnKF to an augmented system of state plus parameters, for a given number of iterations, to invert for θ. We begin by pairing the parameter-to-data map with a dynamical system for the parameter, and then employ techniques from filtering to estimate the parameter given the data.

The system for EKI is written as,

$$\begin{aligned}\theta_{n+1} &= \theta_n \\ y_{n+1} &= \mathcal{G}(\theta_{n+1}) + \eta_{n+1},\end{aligned}$$

where the operator $\mathcal{G}$ contains both the system dynamics and the observation function.

Consider the stochastic dynamical system

$$\begin{aligned}&\text{evolution:} && \theta_{n+1} = \theta_n + \omega_{n+1}, && \omega_{n+1} \sim \mathcal{N}(0, \Sigma_\omega), && (18.1)\\ &\text{observation:} && x_{n+1} = \mathcal{F}(\theta_{n+1}) + \nu_{n+1}, && \nu_{n+1} \sim \mathcal{N}(0, \Sigma_\nu). && (18.2)\end{aligned}$$

We seek the *best* Gaussian approximation of the posterior distribution of θ for ill-posed inverse problems, where the prior is a Gaussian, $\theta_0 \sim \mathcal{N}(r_0, \Sigma_0)$.

Consider the case:

- $\Sigma_\omega = \frac{\Delta t}{1-\Delta t} C_n$, where C_n is the covariance estimation at the current step.
- $x_{n+1} = \begin{bmatrix} y \\ r_0 \end{bmatrix}$, $\mathcal{F}(\theta) = \begin{bmatrix} \mathcal{G}(\theta) \\ \theta \end{bmatrix}$, and $\Sigma_\nu = \frac{1}{\Delta t} \begin{bmatrix} \Sigma_\eta & 0 \\ 0 & \Sigma_0 \end{bmatrix}$

where r_0 and Σ_0 are prior mean and covariance, and the hyperparameter $0 < \Delta t < 1$ is set to $1/2$.

Linear Analysis

In the linear setting,

$$\mathcal{G}(\theta) = G \cdot \theta \qquad F = \begin{bmatrix} G \\ I \end{bmatrix}$$

The update equations become

$$\begin{aligned}\hat{m}_{n+1} &= m_n \\ \hat{C}_{n+1} &= \frac{1}{1-\Delta t} C_n\end{aligned}$$

and

$$m_{n+1} = m_n + \hat{C}_{n+1}F^T(F\hat{C}_{n+1}F^T + \Sigma_{\nu,n+1})^{-1}(x_{n+1} - Fm_n)$$
$$C_{n+1} = \hat{C}_{n+1} - \hat{C}_{n+1}F^T(F\hat{C}_{n+1}F^T + \Sigma_{\nu,n+1})^{-1}F\hat{C}_{n+1},$$

We have the following theorem about the convergence of the algorithm in the setting of the linear forward model:

Theorem (EKI)

Assume that the prior covariance matrix $\Sigma_0 \succ 0$ and initial covariance matrix $C_0 \succ 0$. The iteration for the conditional mean m_n and covariance matrix C_n characterizing the distribution of $\theta_n|Y_n$ converges exponentially fast to the posterior mean, m_{post}, and covariance, C_{post}.

Reference

Please see Efficient Derivative-free Bayesian Inference for Large-Scale Inverse Problems, the arXiv version of (Huang et al. 2022).

Note

This result has been recently extended to near-Gaussian, nonlinear cases in (Carrillo et al. 2024a) and (Carrillo et al. 2024b).

18.3 Algorithms: EKI, ETKI

We will formulate the inversion for the stochastic ensemble filter (EKI) and then for the ensemble transfer filter (ETKI), the latter having a better convergence behavior.

EKI

- Prediction step:

$$\hat{m}_{n+1} = m_n, \quad \hat{\theta}^j_{n+1} = \hat{m}_{n+1} + \sqrt{\frac{1}{1-\Delta\tau}}\left(\theta^j_n - m_n\right)$$

- Analysis step:

$$\hat{x}^j_{n+1} = \mathcal{F}(\hat{\theta}^j_{n+1}), \qquad \hat{x}_{n+1} = \frac{1}{J}\sum_{j=1}^{J} \hat{x}^j_{n+1},$$

$$\hat{C}^{\theta x}_{n+1} = \frac{1}{J-1}\sum_{j=1}^{J}(\hat{\theta}^j_{n+1} - \hat{m}_{n+1})(\hat{x}^j_{n+1} - \hat{x}_{n+1})^{\mathrm{T}},$$

$$\hat{C}^{xx}_{n+1} = \frac{1}{J-1}\sum_{j=1}^{J}(\hat{x}^j_{n+1} - \hat{x}_{n+1})(\hat{x}^j_{n+1} - \hat{x}_{n+1})^{\mathrm{T}} + \Sigma_{\nu,n+1},$$

$$\theta^j_{n+1} = \hat{\theta}^j_{n+1} + \hat{C}^{\theta x}_{n+1}\left(\hat{C}^{xx}_{n+1}\right)^{-1}(x - \hat{x}^j_{n+1} - \nu^j_{n+1}),$$

$$m_{n+1} = \frac{1}{J}\sum_{j=1}^{J}\theta^j_{n+1},$$

where $\nu^j_{n+1} \sim \mathcal{N}(0, \Sigma_{\nu,n+1})$.

A few remarks.

1. The covariance update equation,
$$C_{n+1} = \hat{C}_{n+1} + \hat{C}^{\theta x}_{n+1}\left(\hat{C}^{xx}_{n+1}\right)\left(\hat{C}^{\theta x}_{n+1}\right)^{\mathrm{T}}$$
does not hold here. This will be remedied by the ETKF.
2. The factor $\sqrt{1/(1-\Delta\tau)}$ produces *covariance inflation* and prevents filter collapse.
3. In the analysis step, noise is added in the θ update instead of in the $\hat{x}$ step to ensure symmetry and positive-definiteness of $\hat{C}^{xx}_{n+1}$.

For the ensemble transform filter (described above in Section 15.3), we need to define some matrix square roots for the covariance matrices $\hat{C}^{xx}_{n+1}$ and $\hat{C}^{\theta x}_{n+1}$, as follows. We denote the matrix square roots $\hat{Z}_{n+1}$, $Z_{n+1} \in \mathbb{R}^{N_\theta \times J}$ of $\hat{C}^{\theta x}_{n+1}$, $C^{\theta x}_{n+1}$ and $\hat{\mathcal{Y}}_{n+1}$ of $\hat{C}^{xx}_{n+1}$, and define,

$$\hat{Z}_{n+1} = \frac{1}{\sqrt{J-1}}\Big(\hat{\theta}^1_{n+1} - \hat{m}_{n+1} \quad \hat{\theta}^2_{n+1} - \hat{m}_{n+1} \quad \dots \quad \hat{\theta}^J_{n+1} - \hat{m}_{n+1}\Big),$$
$$Z_{n+1} = \frac{1}{\sqrt{J-1}}\Big(\theta^1_{n+1} - m_{n+1} \quad \theta^2_{n+1} - m_{n+1} \quad \dots \quad \theta^J_{n+1} - m_{n+1}\Big),$$
$$\hat{\mathcal{Y}}_{n+1} = \frac{1}{\sqrt{J-1}}\Big(\hat{x}^1_{n+1} - \hat{x}_{n+1} \quad \hat{x}^2_{n+1} - \hat{x}_{n+1} \quad \dots \quad \hat{x}^J_{n+1} - \hat{x}_{n+1}\Big).$$

ETKI

- Prediction step :

$$\hat{\theta}_{n+1}^{j} = \theta_n^j + \omega_{n+1}^j,$$
$$\hat{m}_{n+1} = \frac{1}{J}\sum_{j=1}^{J} \hat{\theta}_{n+1}^{j}$$

- Analysis step :

$$m_{n+1} = \hat{m}_{n+1} + \hat{C}_{n+1}^{\theta x}\left(\hat{C}_{n+1}^{xx}\right)^{-1}(x - \hat{x}_{n+1})$$
$$Z_{n+1} = \hat{Z}_{n+1}T$$

where $T = P(\Gamma + I)^{-\frac{1}{2}}P^{\mathrm{T}}$, with

$$\text{SVD:} \quad \hat{\mathcal{Y}}_{n+1}\Sigma_{\nu,n+1}^{-1}\hat{\mathcal{Y}}_{n+1} = P\Gamma P^{\mathrm{T}}$$

Remark
When $\Sigma_\omega = \gamma C_n$, the prediction step can be treated deterministically, as follows,

$$\hat{m}_{n+1} = m_n$$
$$\hat{\theta}_{n+1}^{j} = \hat{m}_{n+1} + \sqrt{1+\gamma}\,(\theta_n^j - m_n)$$

18.4 Conclusions

Kalman-based inversion has been widely used to construct derivative-free optimization and s ampling methods for nonlinear inverse problems. In the paper (Huang et al. 2022), the authors developed new Kalman-based inversion methods, for Bayesian inference and uncertainty quantification, which build on the work in both optimization and sampling. They propose a new method for Bayesian inference based on filtering a novel mean-field dynamical system subject to partial noisy observations, and which depends on the law of its own filtering distribution, together with application of the Kalman methodology. Theoretical guarantees are presented: for linear inverse problems, the mean and covariance obtained by the method converge exponentially

fast to the posterior mean and covariance. For nonlinear inverse problems, numerical studies indicate the method delivers an excellent approximation of the posterior distribution for problems which are not too far from Gaussian.

In terms of performance:

1. The methods are shown to be superior to existing coupling/transport methods, collectively known as iterative Kalman methods.
2. Deterministic, such as ETKF, rather than stochastic implementations of Kalman methodology are found to be favorable.

19 Example 1: One-dimensional EKI

We implement here the original, iterative ensemble Kalman filter as formulated in (Iglesias, Law, and Stuart 2013).

Recall the inverse problem: recover the unknown u from noisy measurements y related by

$$y = \mathcal{G}(u) + \eta,$$

where the noise $\eta \sim \mathcal{N}(0, \Gamma)$.

Start by introducing a pseudo time, $h = 1/N$, and then propagate an ensemble $\{ u_n^{(j)}\}$ of J particles (ensemble members) from "time" nh to $(n+1)h$ according to

$$u_{n+1}^{(j)} = \; u_n^{(j)} + C^{up}(u_n) \left[C^{pp}(u_n) + \frac{1}{h}\Gamma \right]^{-1} \left(y_{n+1}^{(j)} - \mathcal{G}(u_n^{(j)}) \right),$$

where

$$C^{pp}(u) = \frac{1}{J-1} \sum_{j=1}^{J} \left(\mathcal{G}(u^{(j)} - \hat{\mathcal{G}} \right) \otimes \left(\mathcal{G}(u^{(j)} - \hat{\mathcal{G}} \right) \tag{19.1}$$

$$C^{up}(u) = \frac{1}{J-1} \sum_{j=1}^{J} \left(u^{(j)} - \hat{u} \right) \otimes \left(\mathcal{G}(u^{(j)} - \hat{\mathcal{G}} \right) \tag{19.2}$$

$$\hat{u} = \frac{1}{J} \sum_{j=1}^{J} u^{(j)}, \qquad \hat{\mathcal{G}} = \frac{1}{J} \sum_{j=1}^{J} \mathcal{G}(u^{(j)}). \tag{19.3}$$

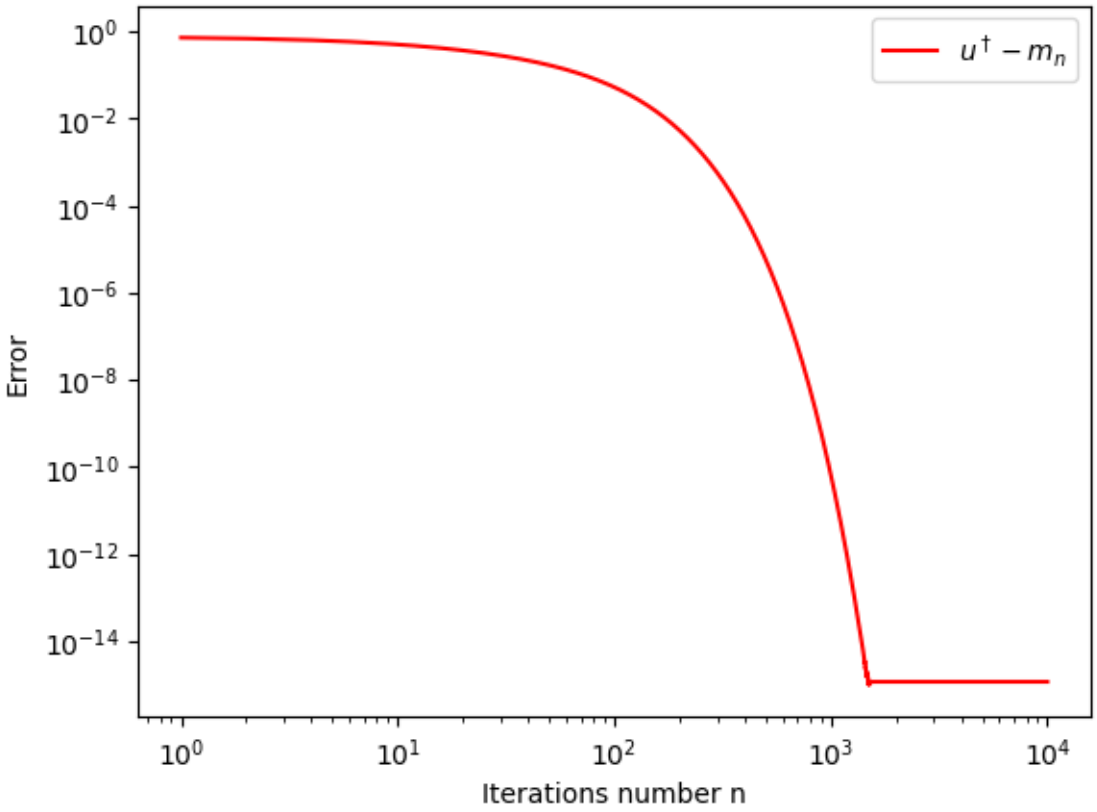
```
import numpy as np
import functools
import matplotlib.pyplot as plt
from functools import partial
```

19.1 Implement the one-dimensional EKI for a linear forward operator $\mathcal{G}$

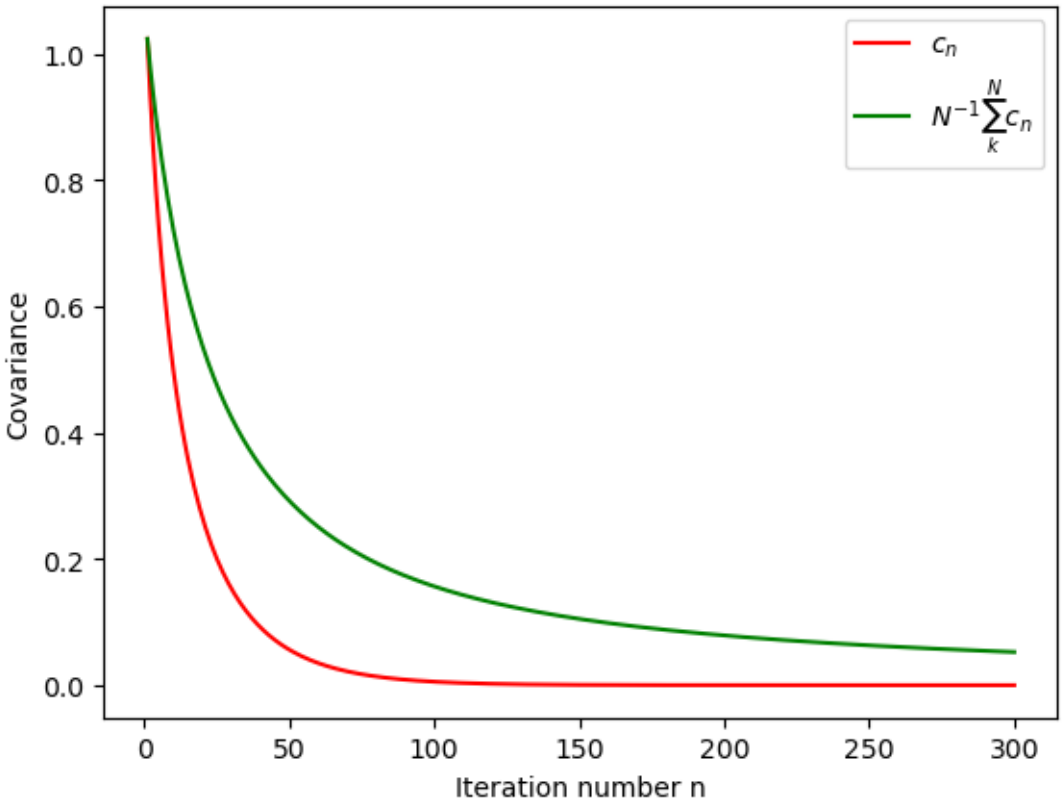
```
def eki_one_dim_lin(m_0, C_0, N, G, gamma, y, delt, h):
    # Inputs:
    # -------
    # m_0, C_0: mean value and covariance of inital ensemble
    # N:        number of iterations
    # G:        one-dimensional forward operator of the model
    # gamma:    covariance of the noise in the data
    # y:        observed data
    # h:        discretization step
    #
    # Outputs:
    # -------
    # U: (JxN) matrix with the computed particles for each iteration
    # m: vector of length N with the mean value of the particles
    # C: vector of length N with the covariance of the particles

    m = np.zeros(N)
    C = np.zeros(N)
    U = np.zeros((J,N))

    #Construct initial ensemble and estimator
    u_0 = np.random.normal(m_0, C_0, J)
    U[:,0] = u_0
    m[0] = np.mean(U[:,0])
    C[0] = (U[:,0] - m[0]) @ (U[:,0] - m[0]).T / (J-1)

    for n in range(1,N):

        # Last iterate under forward operator:
        G_u    = G*U[:, n-1]
        Ghat   = np.mean(G_u)
        U[:,n] = U[:,n-1] + h*(C[n-1] + delt)*G*(1/gamma)*((y - G_u))

        m[n] = np.mean(U[:,n])
        C[n] = (U[:,n] - m[n]) @ (U[:,n] - m[n]).T / (J-1)

    return U,m,C
```

```
#Set Parameters
J = 10
gamma = 1
m_0 = 0
C_0 = 9e-1
m_true = 0
c_true = C_0
G = 1.5
N = 10000
h = 1/100
delt = 1

# Construct data under true parameter
u_true = np.random.normal(m_true,c_true)
y = G*u_true

U,m,c = eki_one_dim_lin(m_0, C_0, N, G, gamma, y, delt, h)
```

```
it=N
iterations=list(range(1,(it+1)))
plt.xlabel('Iterations number n')
plt.ylabel('Error')
plt.loglog(iterations,np.sqrt((u_true*np.ones(N) - m)**2/(u_true**2)),"r",label='$u^\dagger-m
plt.legend(loc="upper right")
plt.show()
```




```
it=300
iterations=list(range(1,it+1))
plt.xlabel('Iteration number n')
plt.ylabel('Covariance')
plt.plot(iterations,c[0:it],"r", label='$c_n$')
plt.plot(iterations,np.divide(np.cumsum(c[0:it]),iterations)[0:it],"g",label='$N^{-1}\sum_k^
plt.legend(loc="upper right")
plt.show()
```



19.2 Implement the one dimensional EKI for an arbitrary forward operator $\mathcal{G}$

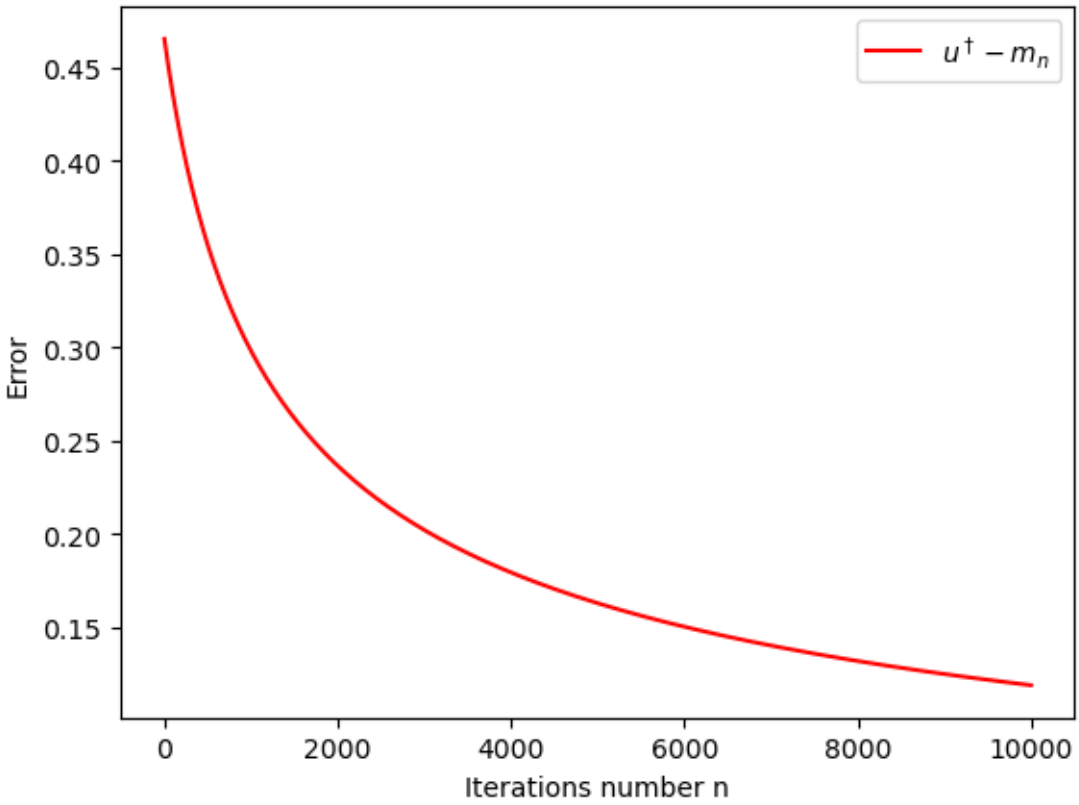
```
def eki_one_dim(m_0, C_0, N, G, gamma, y, h):
    # Inputs:
    # -------
    # m_0, C_0: mean value and covariance of inital ensemble
    # N:        number of iterations
    # G:        one-dimensional forward operator of the model
    # gamma:    covariance of the noise in the data
    # y:        observed data
    # h:        discretization step
    #
    # Outputs:
    # -------
    # U: (JxN) matrix with the computed particles for each iteration
```

```
    # m: vector of length N with the mean value of the particles
    # C: vector of length N with the covariance of the particles

    m = np.zeros(N)
    C = np.zeros(N)
    U = np.zeros((J,N))

    #Construct initial ensemble and estimator
    u_0 = np.random.normal(m_0, C_0, J)
    U[:,0] = u_0
    m[0] = np.mean(U[:,0])
    C[0] = (U[:,0] - m[0]) @ (U[:,0] - m[0]).T / (J-1)

    for n in range(1,N):

        # Last iterate under forward operator:
        G_u = G(U[:,n-1])
        uhat = np.mean(U[:,n-1])
        Ghat = np.mean(G_u)

        cov_up = (U[:,n-1] - uhat) @ (G_u - Ghat).T / (J-1)
        cov_pp = (G_u - Ghat) @ (G_u - Ghat).T / (J-1)

        U[:,n] = U[:,n-1] + cov_up*h/(h*cov_pp + gamma)*(y - G_u)

        m[n] = np.mean(U[:,n])
        C[n] = (U[:,n]-m[n])@(U[:,n]-m[n]).T/(J-1)

    return U, m, C
```

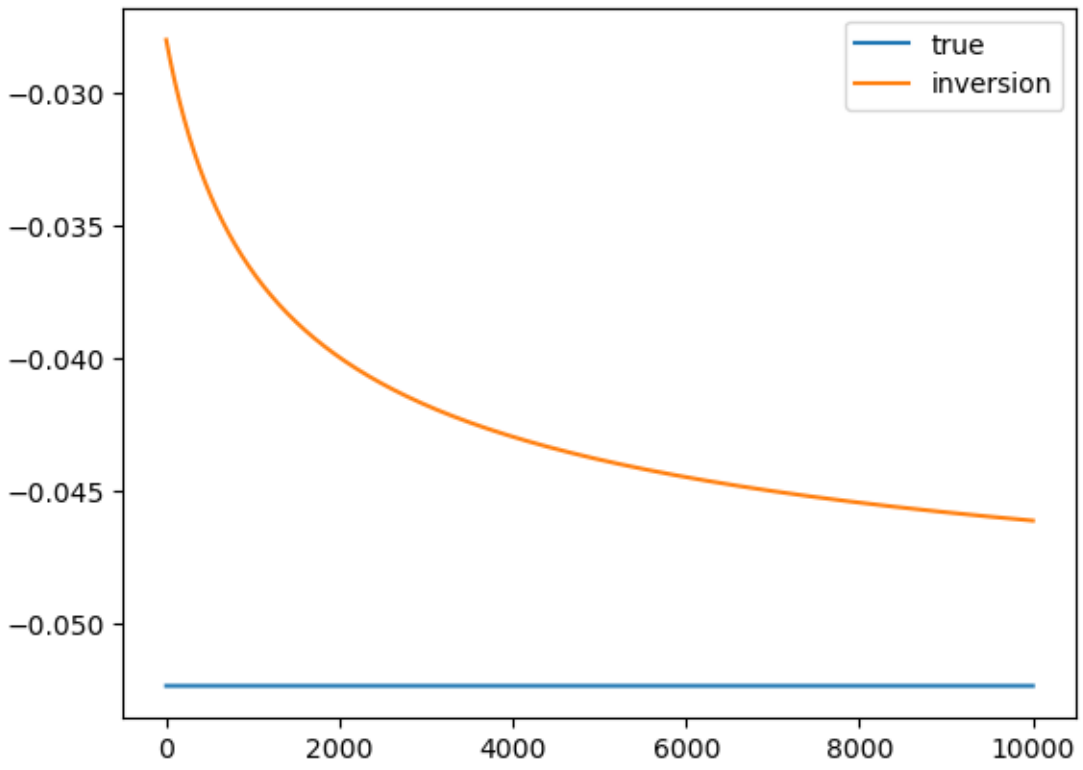
```
#Set Parameters
J = 10
r = 10
k = 10
gamma = 1
m_0 = 0
C_0 = 9e-2
m_true = 0
C_true = C_0
N = 10000
h = 1/100 # 1/N
```

```
def forward_log(z, k, r, h):
    return k/(1 + np.exp(-r*k*h)*(k/z-1))

# Construct data under true parameter
u_true = np.random.normal(m_true, C_true)
y = forward_log(u_true, k, r, h)

# Use partial function
partial_log = functools.partial(forward_log, k=10, r=10, h=1/100)

U, m, C = eki_one_dim(m_0, C_0, N, partial_log, gamma, y, h)
```

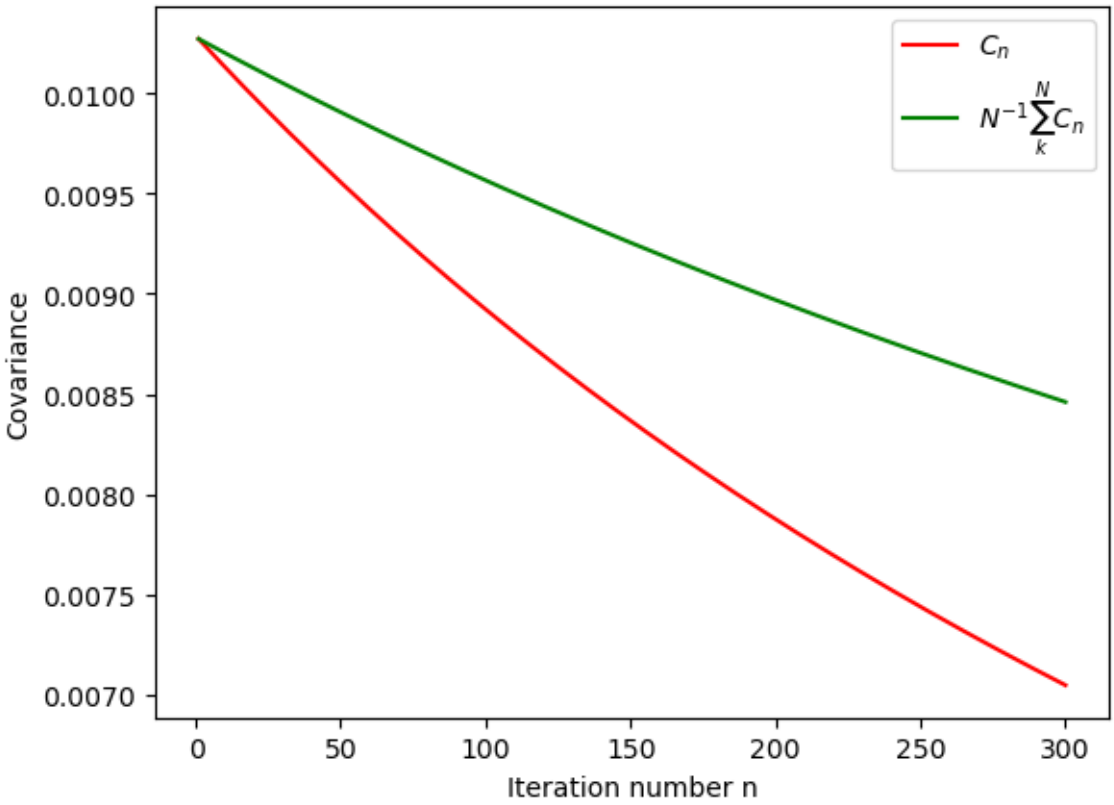
```
it = N
iterations=list(range(1,it+1))
plt.xlabel('Iterations number n')
plt.ylabel('Error')
plt.plot(iterations,np.sqrt((u_true*np.ones(N) - m)**2/(u_true**2)),"r", label='$u^\dagger-m_
plt.legend(loc="upper right")
plt.show()
```




```
plt.plot(iterations, u_true*np.ones(N), label="true")
plt.plot(iterations, m, label="inversion")
plt.legend()
plt.show()
```




```
it = 300
iterations = list(range(1,it+1))
plt.xlabel('Iteration number n')
plt.ylabel('Covariance')
plt.plot(iterations,C[0:it],"r", label='$C_n$')
plt.plot(iterations,np.divide(np.cumsum(C[0:it]),iterations)[0:it],"g",label='$N^{-1}\sum_k^
plt.legend(loc="upper right")
plt.show()
```



19.3 Conclusions

The convergence is very slow, and not very accurate. This will be remedied by the mean-field approach.