Lecture 7. Statistical filtering

Alexander Zhigalov / Dept. of CS, University of Helsinki and Dept. of NBE, Aalto University

Outline / overview

- **Special topic A.** Wavelet analysis
- **Special topic B.** Phase-shuffling
- **Section 1.** Bayesian filtering
- **Section 2.** Kalman filter
- **Section 3.** Kalman filter in python, example 1
- **Section 4.** Kalman filter in python, example 2

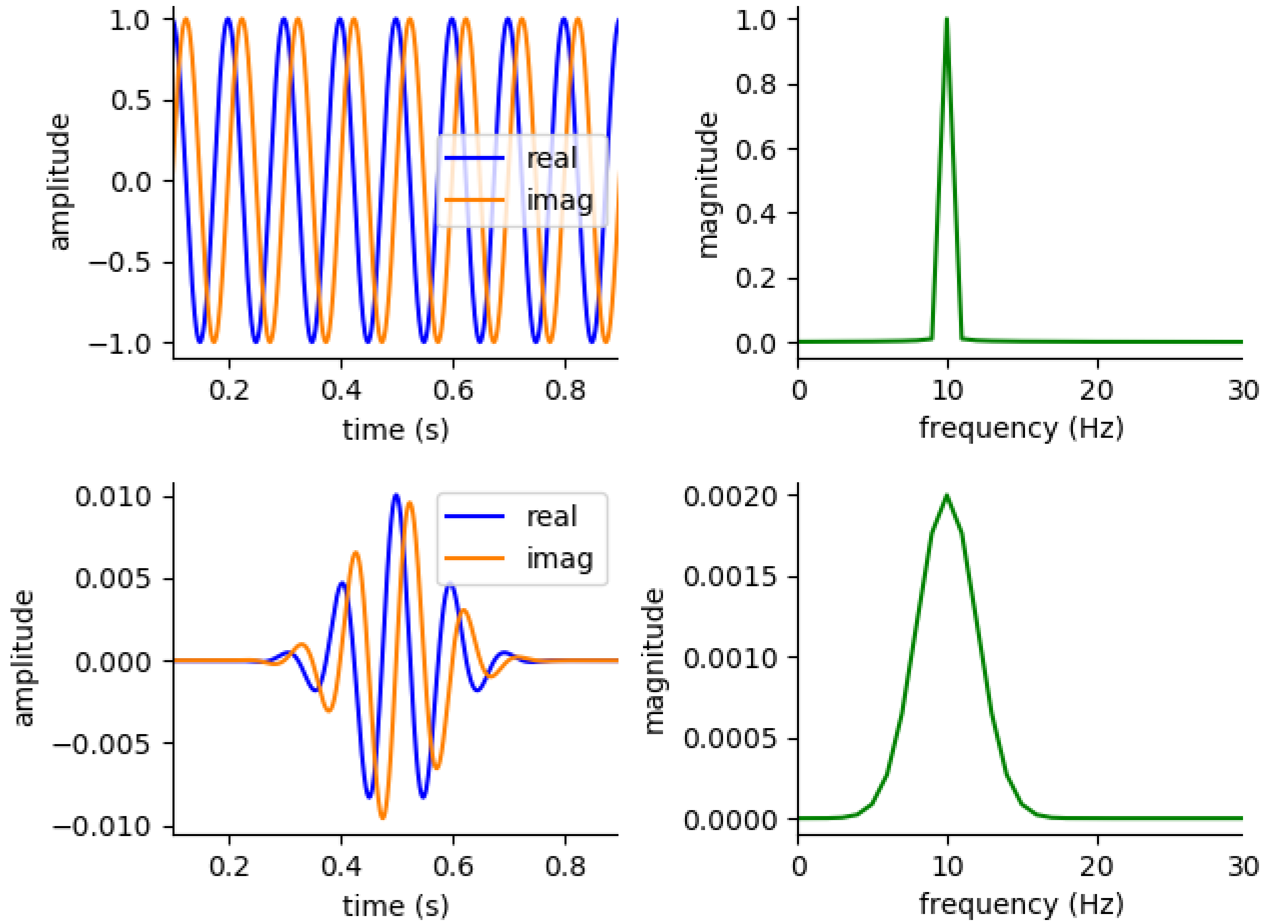
Section A. Wavelet filtering

Wavelets

What is the difference between sine (*complex exponent*) and wavelet?

```
# signal
x = np.cos(2 * np.pi * 10 * t) +
    1j * np.sin(2 * np.pi * 10 * t)
X = np.abs(fft(x)) / N
```

```
# init Morlet wavelet
f0 = 10
m = 5
a, b = init_wavelet(f0, m, fs)

# shape to draw
y = np.concatenate((a, b))
Y = np.abs(fft(y)) / N
```

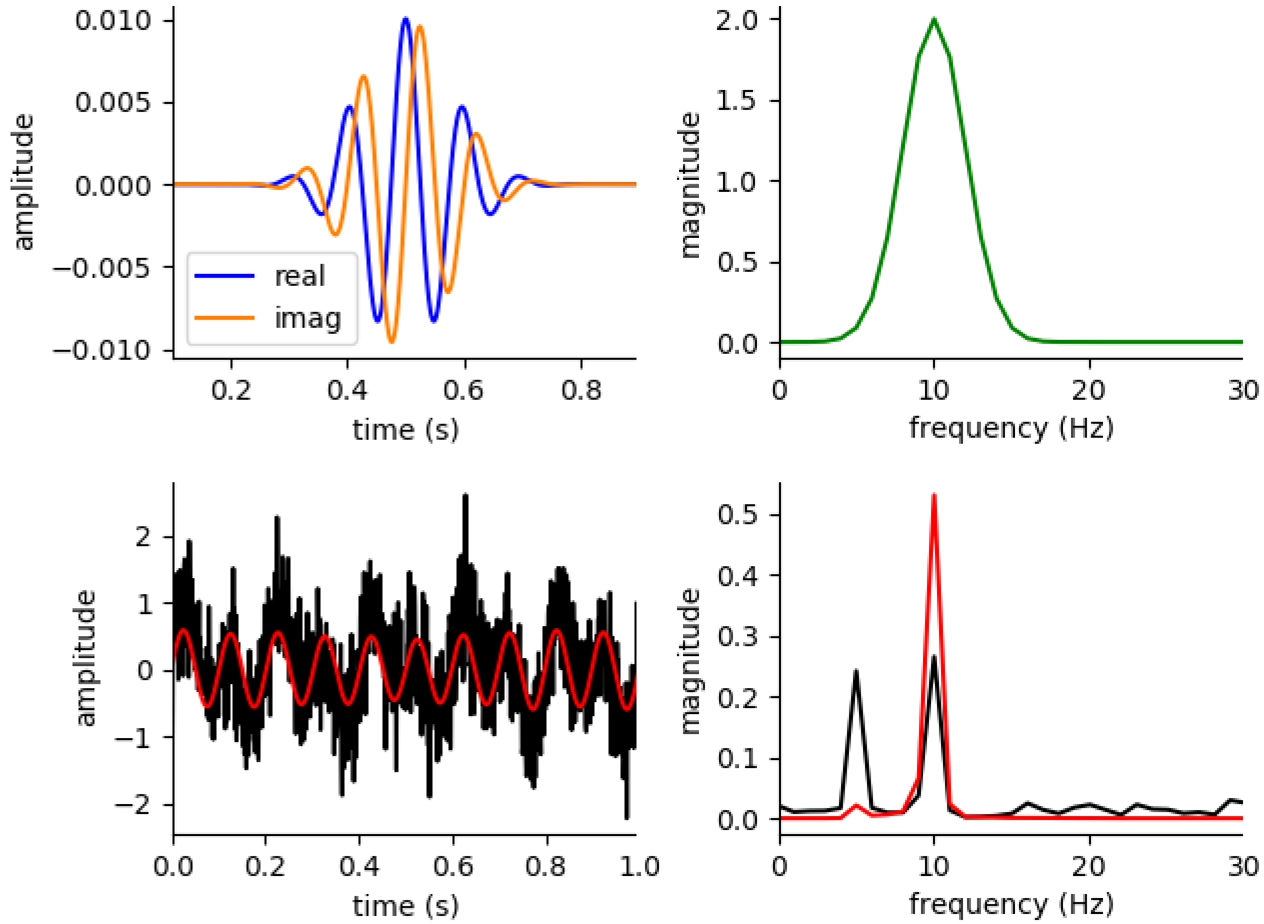


See, “L07_wavelet_vs_sine.py”

Wavelet filtering (1/3)

What are the parameters of **Morlet** wavelet?

```
# signal
x = 0.5 * np.sin(2 * np.pi * 5 * t)  + \
    0.5 * np.sin(2 * np.pi * 10 * t) + \
    0.5 * np.random.randn(N)

# init Morlet wavelet
f0 = 10
m = 5
a, b = init_wavelet(f0, m, fs)

# concatenate halves
w = [b, np.zeros(2 * L, 'complex'), a)]

# filtering
y = ifft(fft(x) * fft(w))

# amplitude spectra
X = np.abs(fft(x)) / N
Y = np.abs(fft(y)) / N
```

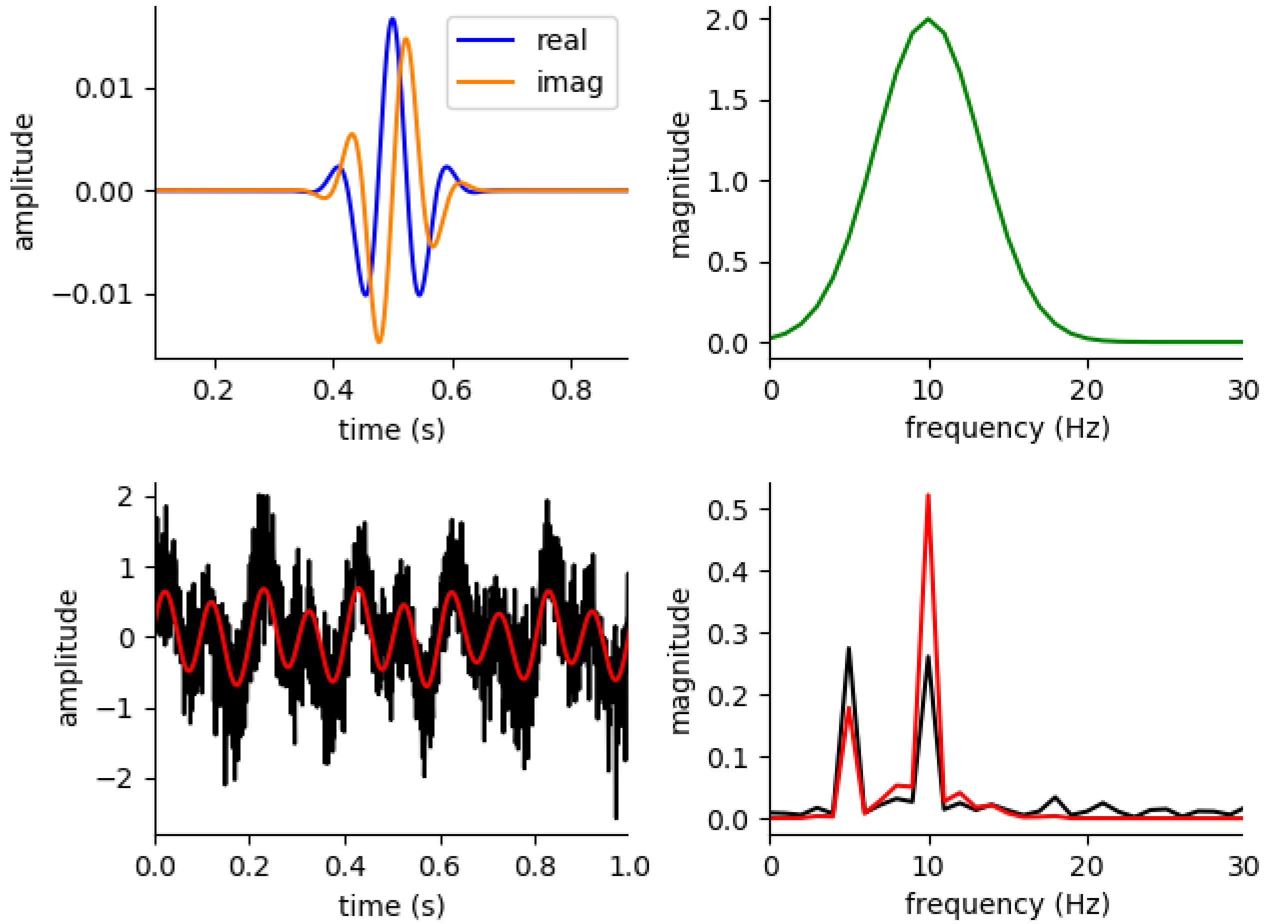


See, “L07_wavelet_filtering.py”

Wavelet filtering (2/3)

How are the time and frequency domains related?

```
# signal
x = 0.5 * np.sin(2 * np.pi * 5 * t)  + \
    0.5 * np.sin(2 * np.pi * 10 * t) + \
    0.5 * np.random.randn(N)

# init Morlet wavelet
f0 = 10
m = 3
a, b = init_wavelet(f0, m, fs)

# concatenate halves
w = [b, np.zeros(2 * L, 'complex'), a)]

# filtering
y = ifft(fft(x) * fft(w))

# amplitude spectra
X = np.abs(fft(x)) / N
Y = np.abs(fft(y)) / N
```

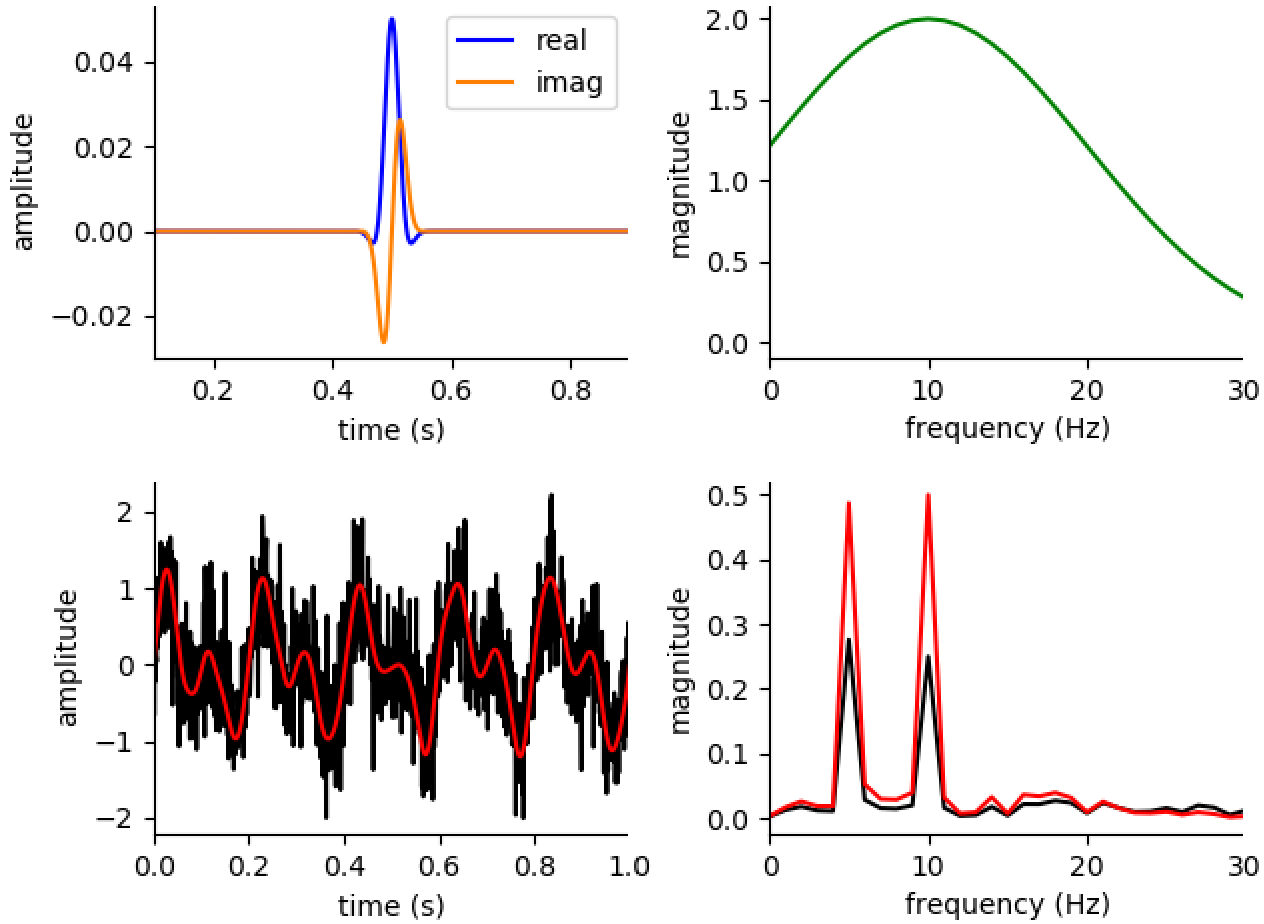


See, “L07_wavelet_filtering.py”

Wavelet filtering (3/3)

Narrow in time – wide in frequency

```
# signal
x = 0.5 * np.sin(2 * np.pi * 5 * t)  + \
    0.5 * np.sin(2 * np.pi * 10 * t) + \
    0.5 * np.random.randn(N)

# init Morlet wavelet
f0 = 10
m = 1
a, b = init_wavelet(f0, m, fs)

# concatenate halves
w = [b, np.zeros(2 * L, 'complex'), a)]

# filtering
y = ifft(fft(x) * fft(w))

# amplitude spectra
X = np.abs(fft(x)) / N
Y = np.abs(fft(y)) / N
```



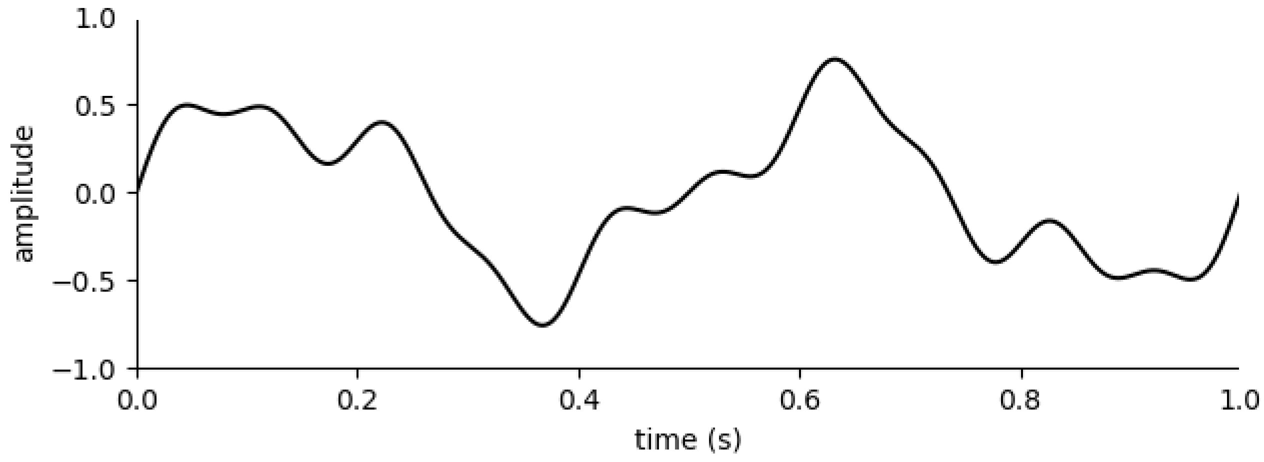
See, “L07_wavelet_filtering.py”

Section B. Phase shuffling

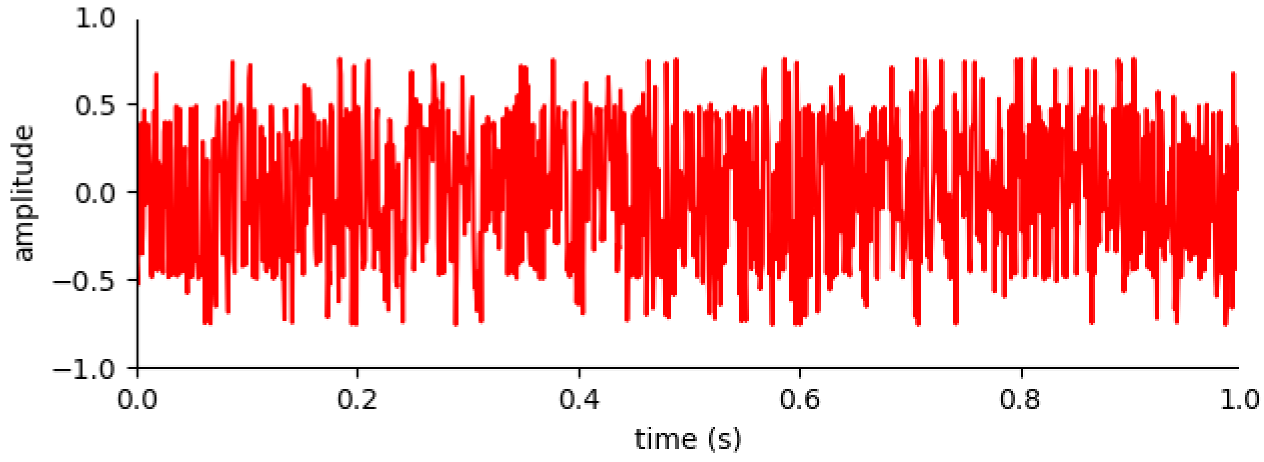
Random permutation (1/2)

How to generate null hypothesis about the signal?

```
# signal
x = 0.5 * np.sin(2 * np.pi * 2 * t)  + \
    0.2 * np.sin(2 * np.pi * 5 * t) + \
    0.1 * np.sin(2 * np.pi * 10 * t)
```




```
permutation
y = x[np.random.permutation(N)]
```

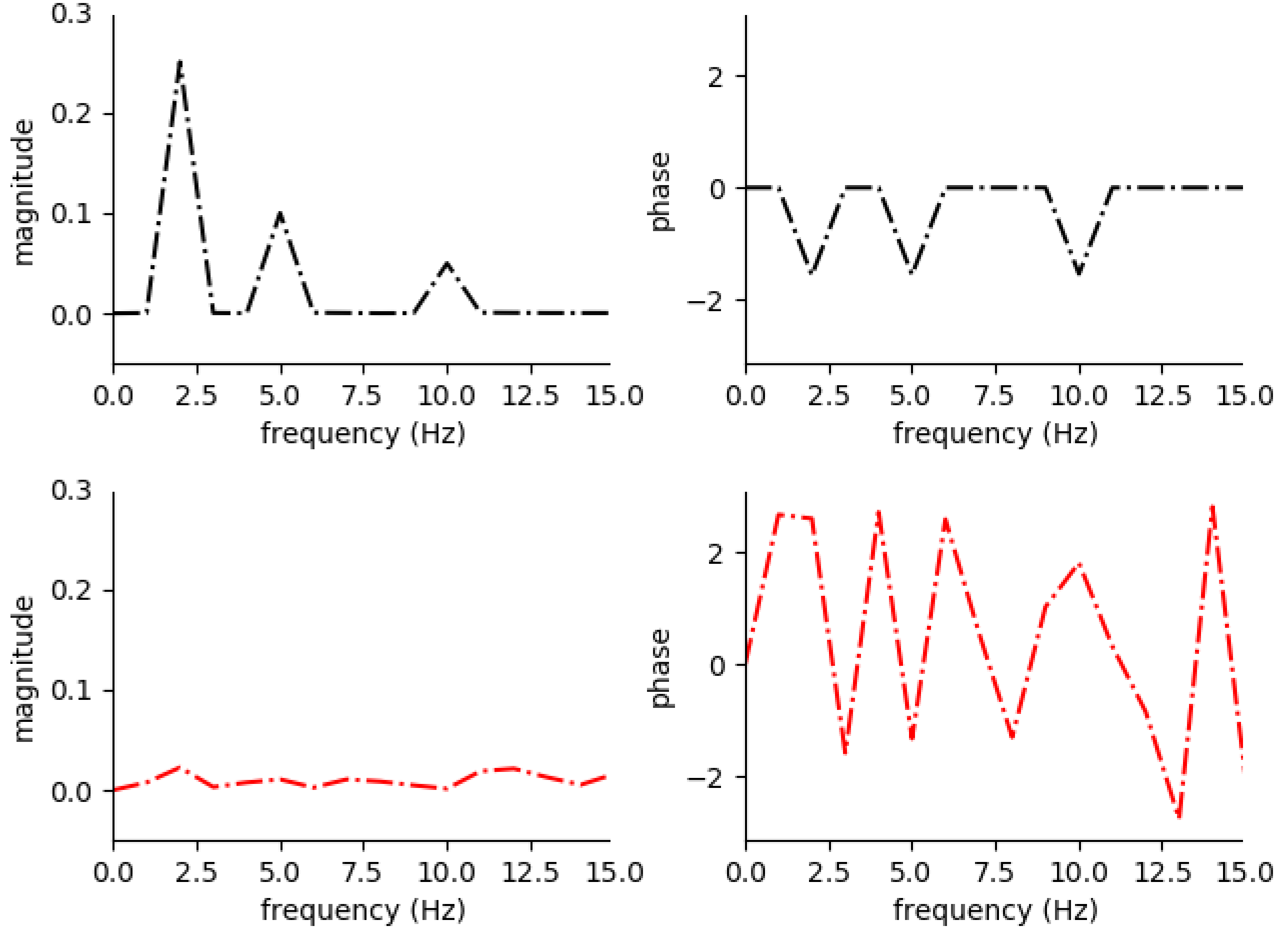


See, “L07_phase_shuffling_rnd.py”

Random permutation (2/2)

Do the changes looks realistic?

```
# fourier transform
X = fft(x, nFFT)
Y = fft(y, nFFT)

# amplitude spectrum
Ax = np.abs(X) / N
Ay = np.abs(Y) / N

# phase spectrum
Px = np.angle(X)
Py = np.angle(Y)
```

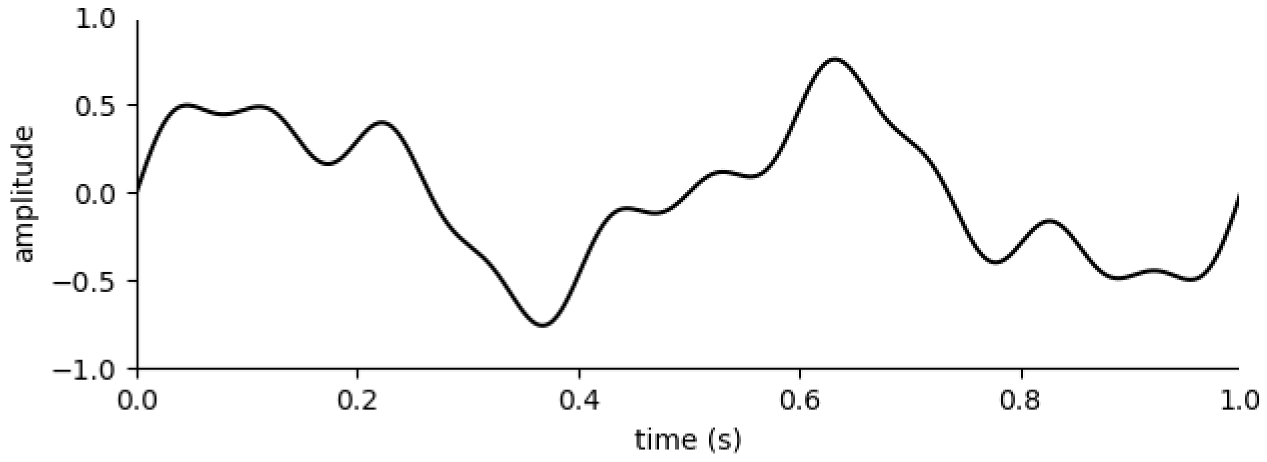


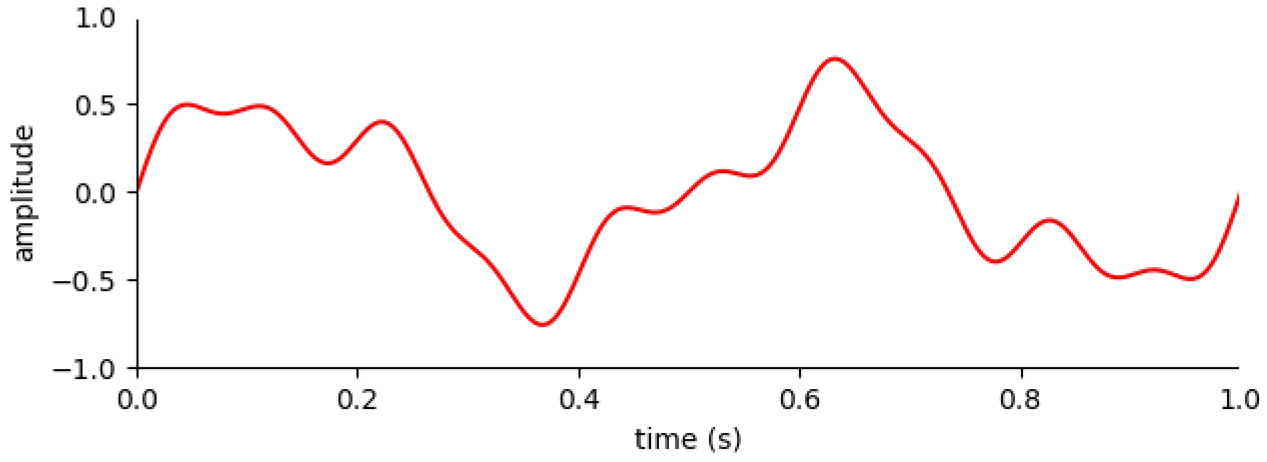
See, "L07_phase_shuffling_rnd.py"

Fourier transform

Property of Fourier transform

```
# signal
x = 0.5 * np.sin(2 * np.pi * 2 * t)  + \
    0.2 * np.sin(2 * np.pi * 5 * t) + \
    0.1 * np.sin(2 * np.pi * 10 * t)


# fourier transform
s = fft(x)
s = np.abs(s) * np.exp(1j * np.angle(s))
y = ifft(s)
```



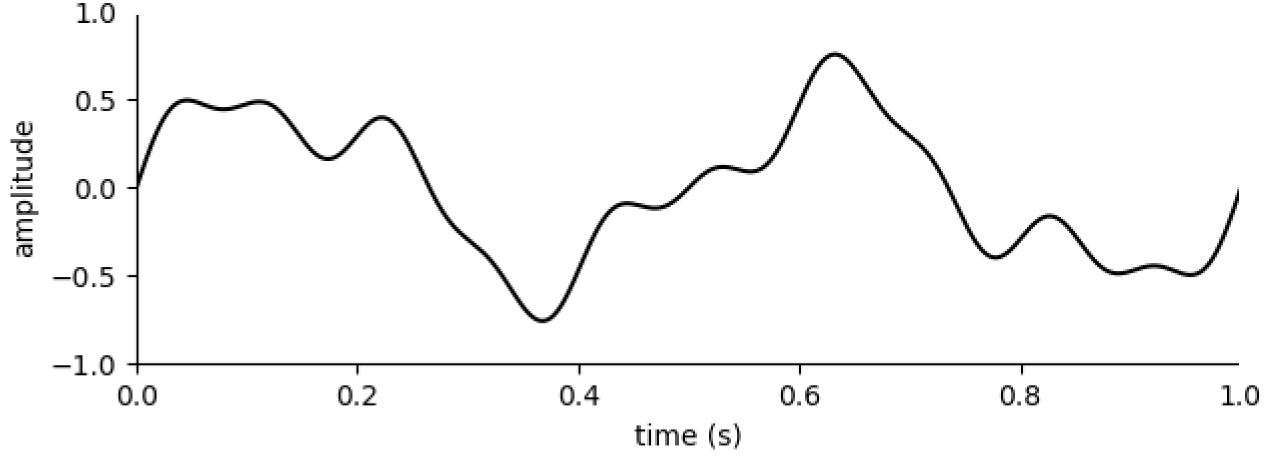


See, “L07_phase_shuffling_2.py”

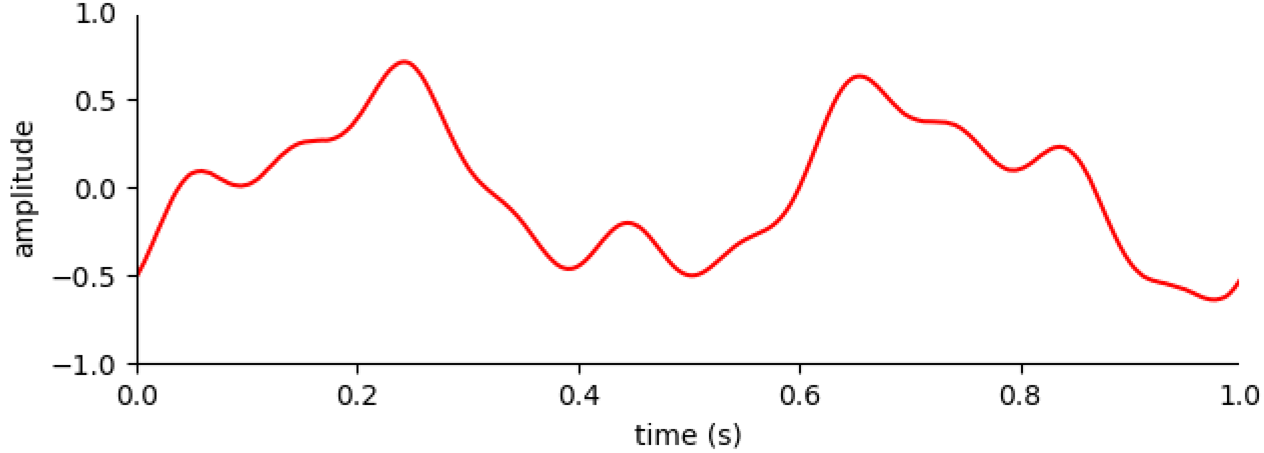
Phase shuffling (1/2)

Could we modify the phase information without changing the amplitude?

```
# signal
x = 0.5 * np.sin(2 * np.pi * 2 * t)  + \
    0.2 * np.sin(2 * np.pi * 5 * t) + \
    0.1 * np.sin(2 * np.pi * 10 * t)
```




```
# phase shuffling
n = len(x)
s = fft(x)

r = np.random.rand(n/2)) * 2 * np.pi - np.pi

s = np.abs(s) * np.exp(1j * [r, -r[::-1]])

y = ifft(s)
```

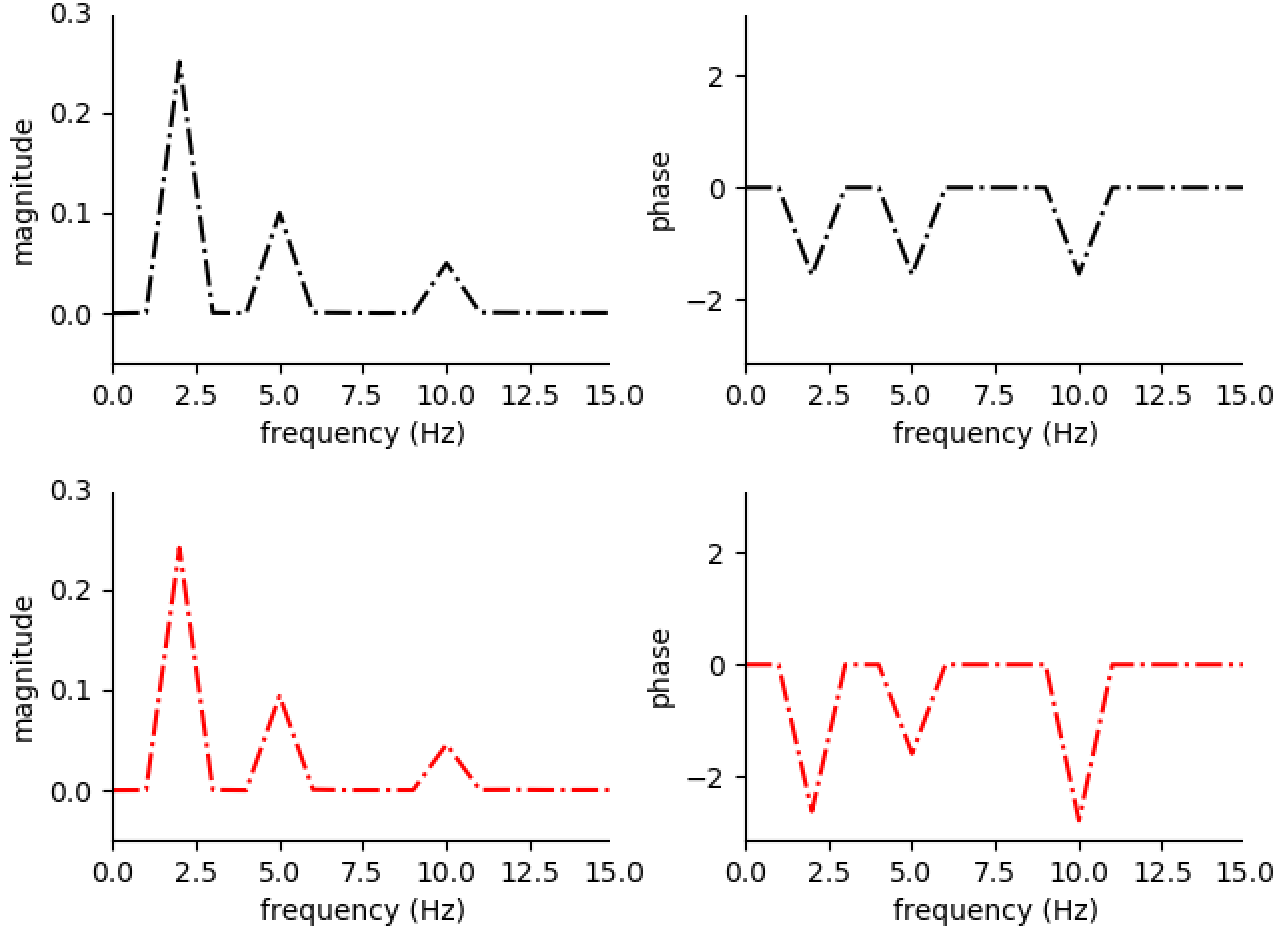


See, “L07_phase_shuffling.py”

Phase shuffling (2/2)

Could we modify the phase information without changing the amplitude?

```
# fourier transform
X = fft(x, nFFT)
Y = fft(y, nFFT)

# amplitude spectrum
Ax = np.abs(X) / N
Ay = np.abs(Y) / N

# phase spectrum
Px = np.angle(X)
Py = np.angle(Y)
```

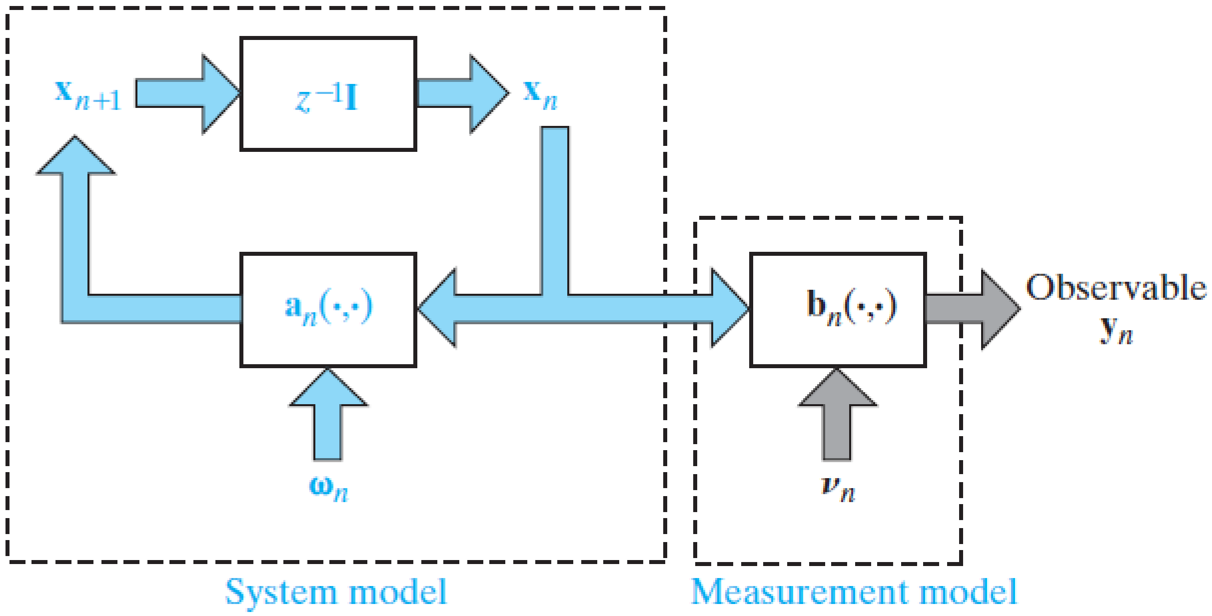


See, “L07_phase_shuffling.py”

Section 1. Bayesian filtering

Bayesian filtering for State estimation of dynamic systems

State-space model



Haykin, 2009, “Neural networks and learning machines”, 3rd edition (Chapter 14)

Dynamic system

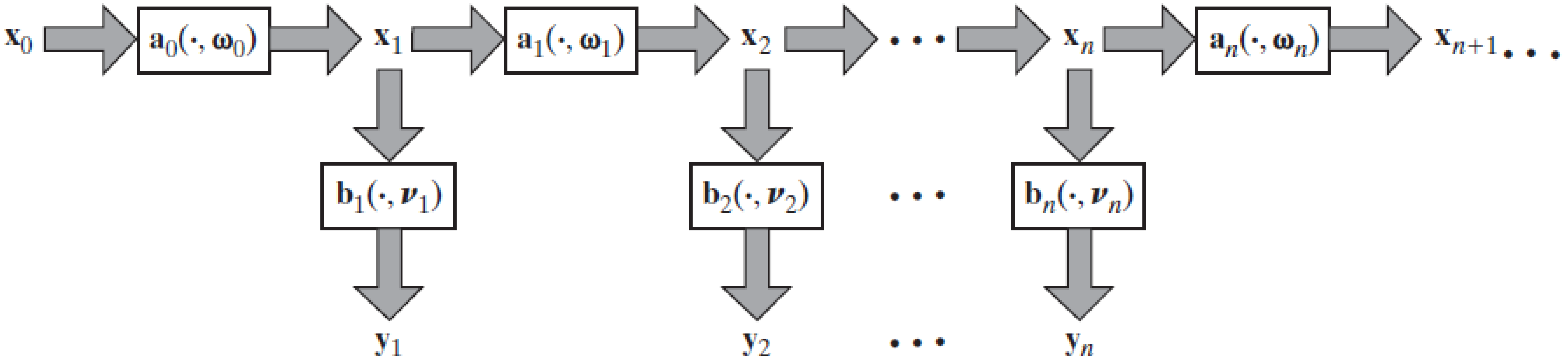
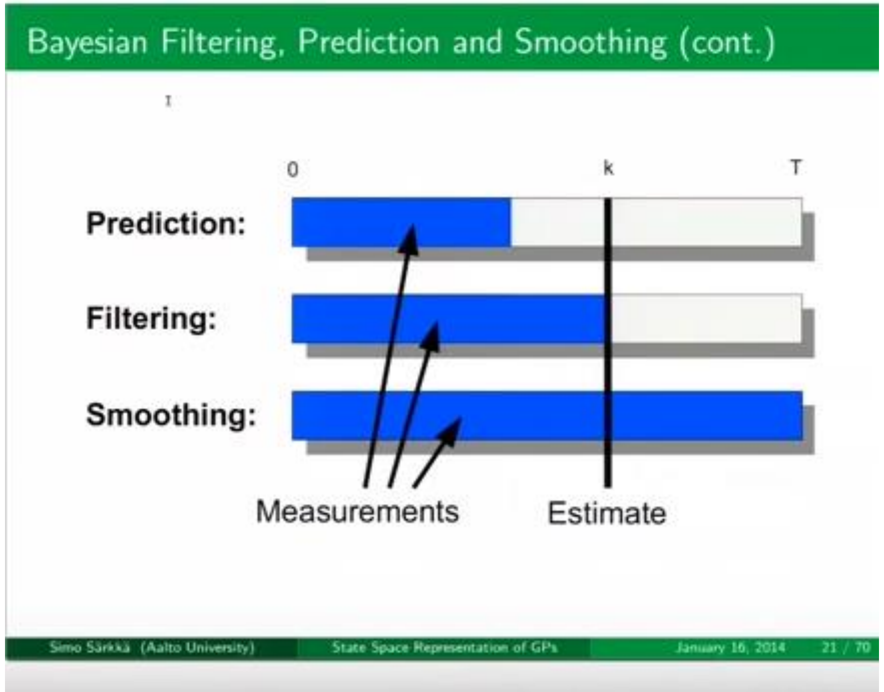


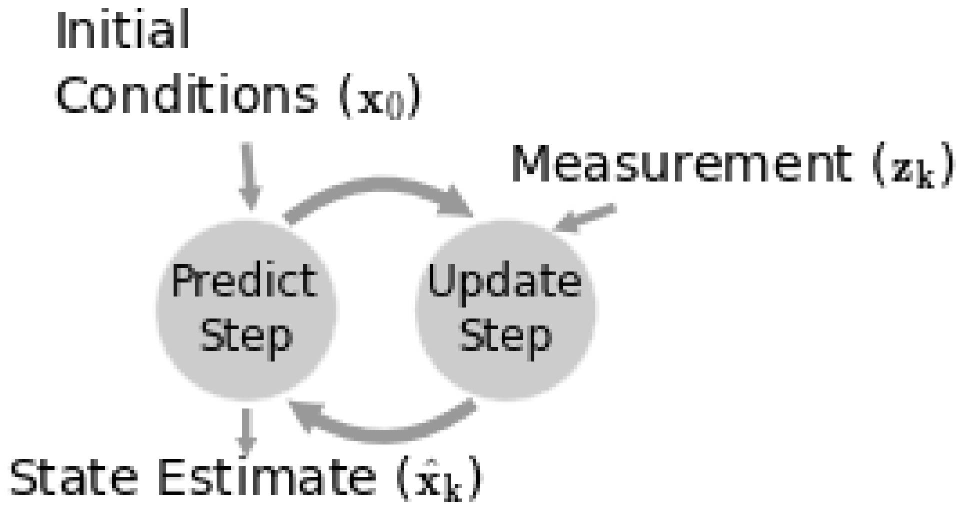
FIGURE 14.2 Evolution of the state across time, viewed as a first-order Markov chain.

State-estimation problem

In any event, the state-estimation problem is called *prediction* if $k > n$, *filtering* if $k = n$, and *smoothing* if $k < n$. Typically, a smoother is statistically more accurate than both the predictor and filter, as it uses more observables. On the other hand, both prediction and filtering can be performed in real time, whereas smoothing cannot.



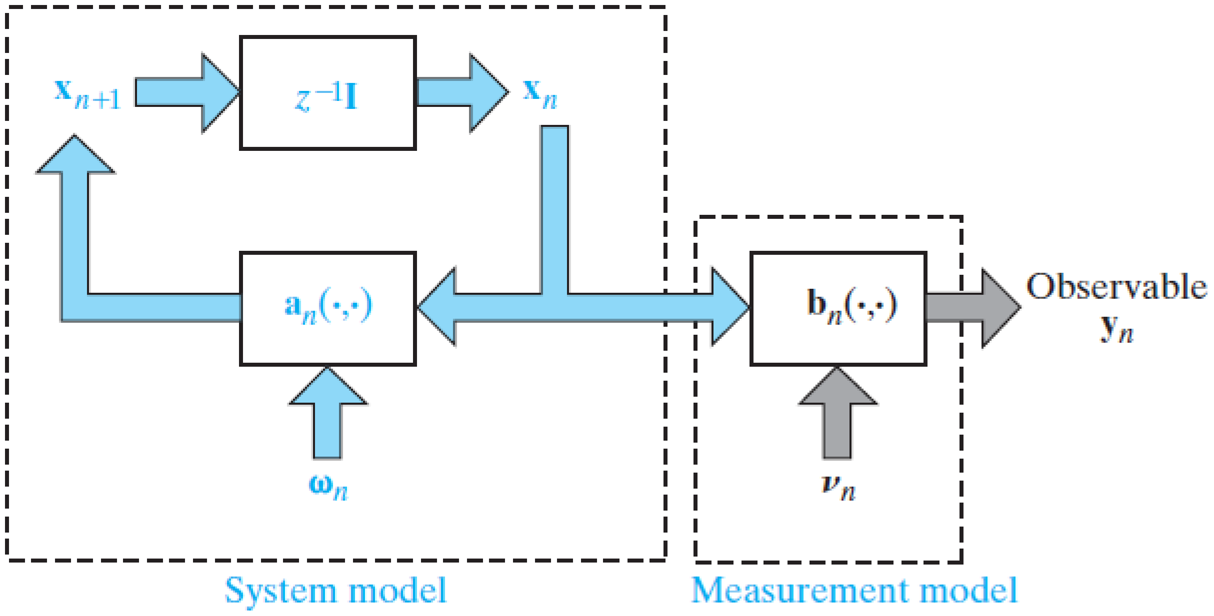
The Discrete Bayesian algorithm



http://nbviewer.jupyter.org/github/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/table_of_contents.ipynb

Section 2. Kalman filter

State-space model (1/3)



Haykin, 2009, “Neural networks and learning machines”, 3rd edition (Chapter 14)

State-space model (2/3)

Linear, Gaussian model

$$\mathbf{x}_{n+1} = \mathbf{A}_{n+1,n}\mathbf{x}_n + \boldsymbol{\omega}_n \tag{14.4}$$

and

$$\mathbf{y}_n = \mathbf{B}_n\mathbf{x}_n + \boldsymbol{\nu}_n \tag{14.5}$$

where $\mathbf{A}_{n+1,n}$ is the *transition matrix* from state $\mathbf{x}_n$ to state $\mathbf{x}_{n+1}$ and $\mathbf{B}_n$ is the *measurement matrix*. The dynamic noise $\boldsymbol{\omega}_n$ and measurement noise $\boldsymbol{\nu}_n$ are both additive and assumed to be *statistically independent zero-mean Gaussian processes*[1] whose covariance matrices are respectively denoted by $\mathbf{Q}_{\omega,n}$ and $\mathbf{Q}_{\nu,n}$.

State-space model (3/3)

The state-space model for the Kalman filter is defined by Eqs. (14.4) and (14.5). This linear Gaussian model is parameterized as follows:

- the transition matrix $\mathbf{A}_{n+1,n}$, which is invertible;
- the measurement matrix $\mathbf{B}_n$, which, in general, is a rectangular matrix;
- the Gaussian dynamic noise $\boldsymbol{\omega}_n$, which is assumed to have zero mean and covariance matrix $\mathbf{Q}_{\omega,n}$;
- the Gaussian measurement noise $\boldsymbol{\nu}_n$, which is assumed to have zero mean and covariance matrix $\mathbf{Q}_{\nu,n}$.

All these parameters are assumed to be known. We are also given the sequence of observables $\{\mathbf{y}_k\}_{k=1}^{n}$. The requirement is to derive an estimate of the state $\mathbf{x}_k$ that is optimized in *the minimum mean-square-error sense.*

Kalman filter summary (1/4)

What Kalman filter does?

The input of the filter is the sequence of observables $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n$, and the output of the filter is the filtered estimate $\hat{\mathbf{x}}_{n|n}$.

Kalman filter summary (2/4)

TABLE 14.1 Summary of the Kalman Variables and Parameters

Variable	Definition	Dimension
$\mathbf{x}_n$	State at time n	M by 1
$\mathbf{y}_n$	Observation at time n	L by 1
$\mathbf{A}_{n+1,n}$	Invertible transition matrix from state at time n to state at time $n+1$	M by M
$\mathbf{B}_n$	Measurement matrix at time n	L by M
$\mathbf{Q}_{\omega,n}$	Covariance matrix of dynamic noise $\boldsymbol{\omega}_n$	M by M
$\mathbf{Q}_{\nu,n}$	Covariance matrix of measurement noise $\boldsymbol{\nu}_n$	L by L
$\hat{\mathbf{x}}_{n\|n-1}$	Predicted estimate of the state at time n, given the observations $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{n-1}$	M by 1
$\hat{\mathbf{x}}_{n\|n}$	Filtered estimate of the state at time n, given the observations $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n$	M by 1
$\mathbf{G}_n$	Kalman gain at time n	M by L
$\boldsymbol{\alpha}_n$	Innovations process at time n	L by 1
$\mathbf{R}_n$	Covariance matrix of the innovations process $\boldsymbol{\alpha}_n$	L by L
$\mathbf{P}_{n\|n-1}$	Prediction-error covariance matrix	M by M
$\mathbf{P}_{n\|n}$	Filtering-error covariance matrix	M by M

Haykin, 2009, “Neural networks and learning machines”, 3rd edition (Chapter 14)

Kalman filter summary (3/4)

TABLE 14.2 Summary of the Kalman Filter Based on Filtered Estimate of the State

Observations = $\{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n\}$

Known parameters

Transition matrix = $\mathbf{A}_{n+1,n}$
Measurement matrix = $\mathbf{B}_n$
Covariance matrix of dynamic noise = $\mathbf{Q}_{\omega,n}$
Covariance matrix of measurement noise = $\mathbf{Q}_{\nu,n}$

Computation: $n = 1, 2, 3, \ldots$

$$\mathbf{G}_n = \mathbf{P}_{n|n-1}\mathbf{B}_n^T[\mathbf{B}_n\mathbf{P}_{n|n-1}\mathbf{B}_n^T + \mathbf{Q}_{\nu,n}]^{-1}$$
$$\boldsymbol{\alpha}_n = \mathbf{y}_n - \mathbf{B}_n\hat{\mathbf{x}}_{n|n-1}$$
$$\hat{\mathbf{x}}_{n|n} = \hat{\mathbf{x}}_{n|n-1} + \mathbf{G}_n\boldsymbol{\alpha}_n$$
$$\hat{\mathbf{x}}_{n+1|n} = \mathbf{A}_{n+1,n}\hat{\mathbf{x}}_{n|n}$$
$$\mathbf{P}_{n|n} = \mathbf{P}_{n|n-1} - \mathbf{G}_n\mathbf{B}_n\mathbf{P}_{n|n-1}$$
$$\mathbf{P}_{n+1|n} = \mathbf{A}_{n+1,n}\mathbf{P}_{n|n}\mathbf{A}_{n+1,n}^T + \mathbf{Q}_{\omega,n}$$

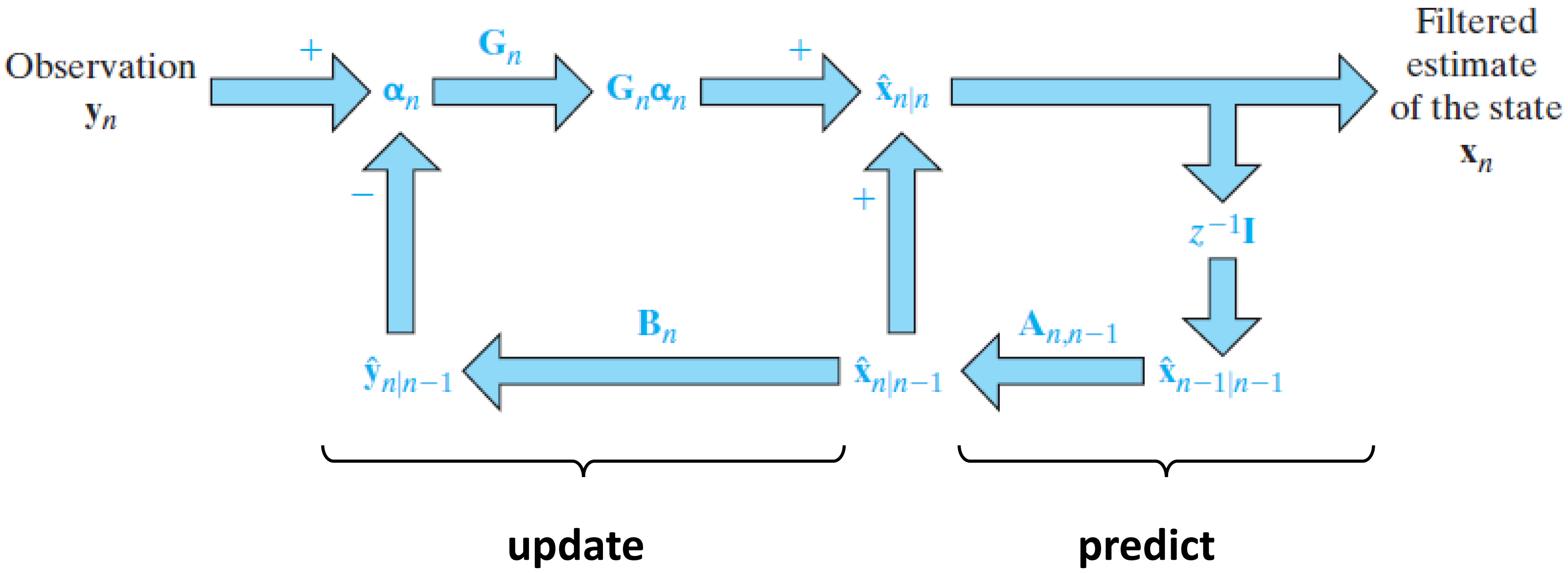
Initial conditions

$$\hat{\mathbf{x}}_{1|0} = \mathbb{E}[\mathbf{x}_1]$$
$$\mathbf{P}_{1,0} = \mathbb{E}[(\mathbf{x}_1 - \mathbb{E}[\mathbf{x}_1])(\mathbf{x}_1 - \mathbb{E}[\mathbf{x}_1])^T] = \boldsymbol{\Pi}_0$$

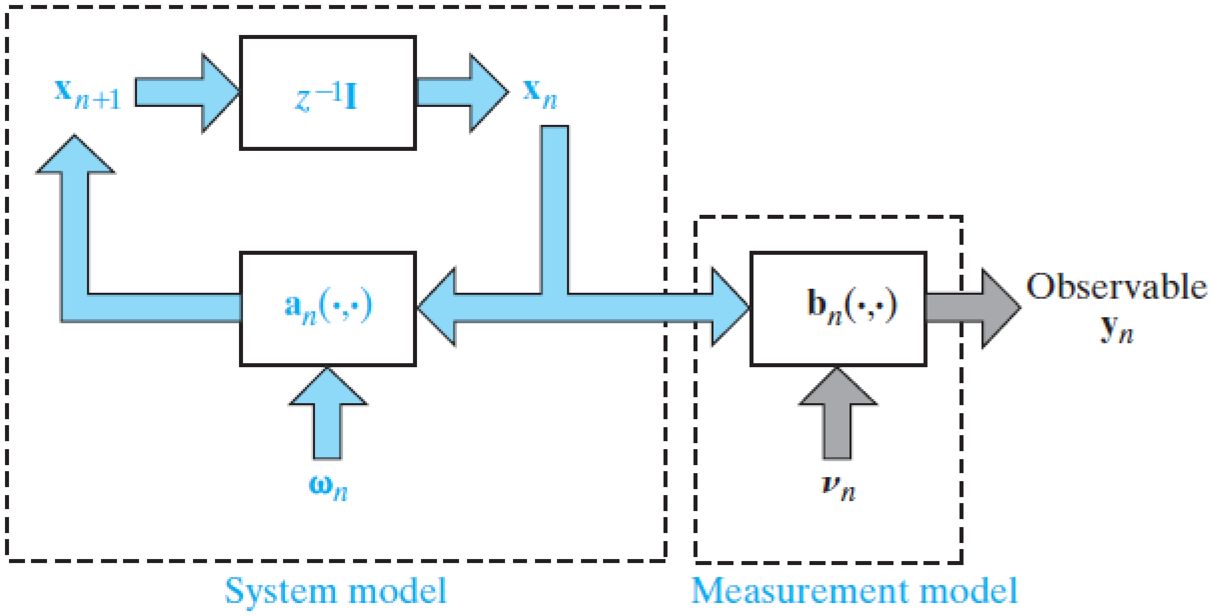
The matrix $\boldsymbol{\Pi}_0$ is a diagonal matrix with diagonal elements all set equal to δ^{-1}, where δ is a small number.

```
% predict
x = A*x;
P = A*P*A' + Q;

% update
y = Y(k, :)'; # observation at time k
a = y - B*x;
G = P*B'*inv(B*P*B' + R);

x = x + G*a;
P = P - G*B*P;
```

Kalman filter summary (4/4)



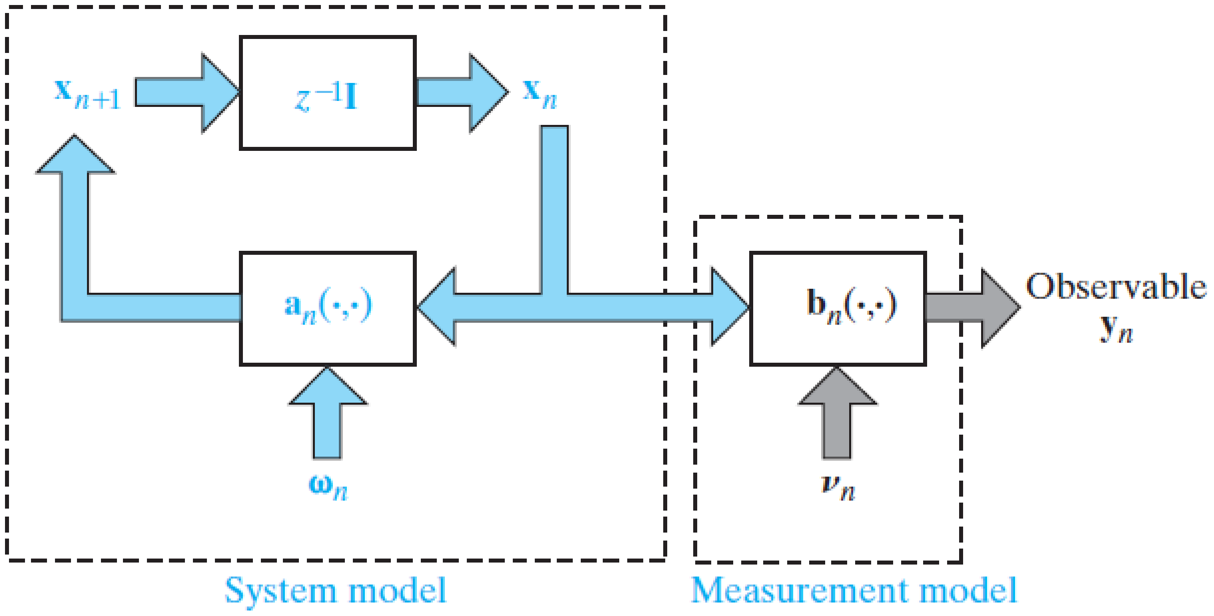
State-space model



Section 3. Kalman filter in python, example 1

Linear dynamic model (1/2)

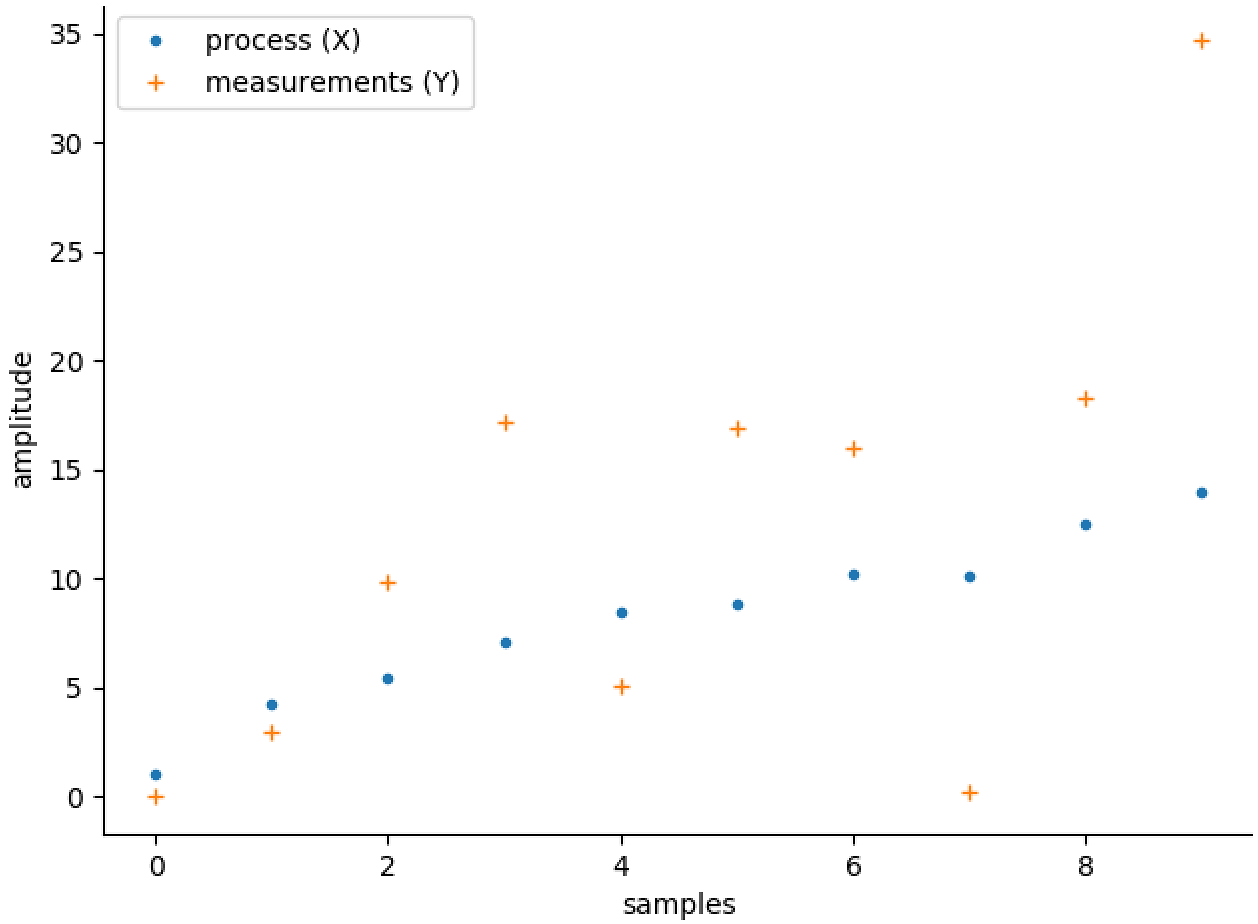
```
# process model
process_var = 1.0  # variance in the process
dx = 1.0
process_model = (dx, process_var)

# measurement model
sensor_var = 100.0 # variance in the sensor
measurement_model = (0, sensor_var)

# generate data
X, Y = generate_data(process_model,
measurement_model)
```



See, "L07_Kalman_in_python_1.py"

Linear dynamic model (2/2)

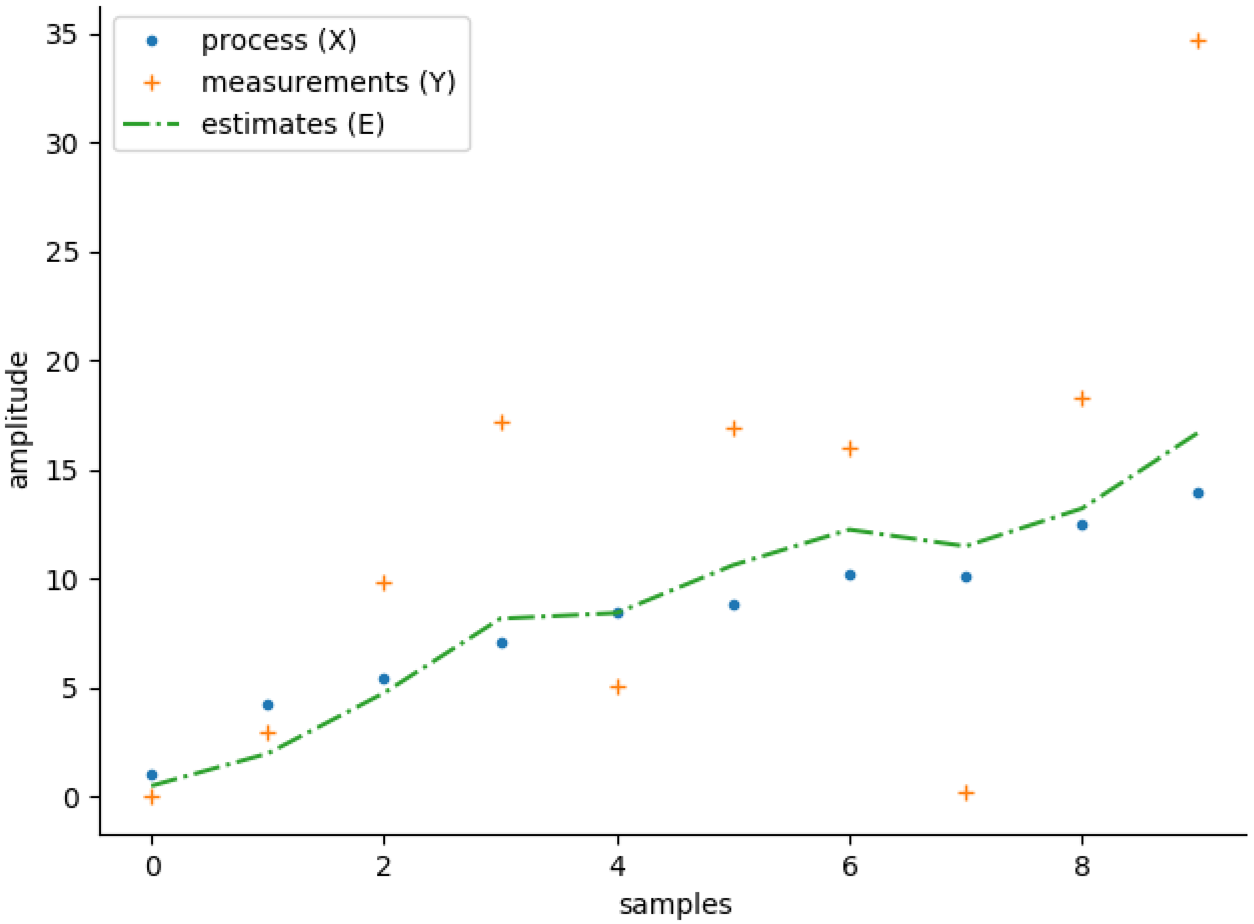
```
def generate_data():

  # process model
  process_noise = np.sqrt(process_var)

  # measurement model
  measurement_noise = np.sqrt(sensor_var)

  # generate data
  for i in range(1, N):

    # process
    X[i] = X[i-1] +
      dx + np.random.randn(1) * process_noise

    # measurement
    Y[i] = X[i] +
      np.random.randn(1) * measurement_noise
```



See, “L07_Kalman_in_python_1.py”

Kalman filter (1/2)

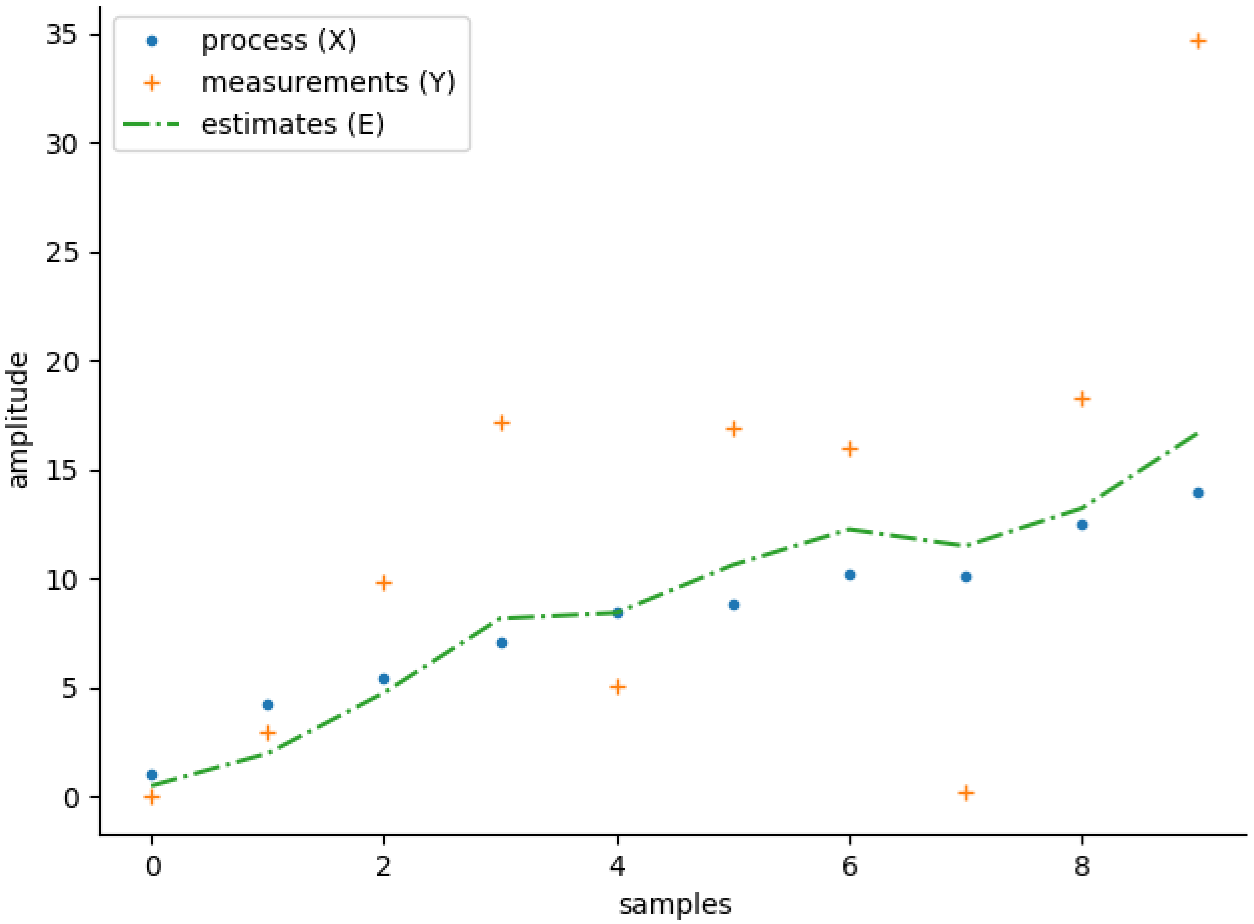
```
# initial condition
x_mu = 0.0
x_var = 100.0
x = (x_mu, x_var) # Gaussian process

# run Kalman filter
for i in range(0, N):

  # predict
  prior_mu, prior_var = predict((x_mu, x_var),
    (dx, process_var))
  prior = (prior_mu, prior_var)

  # update
  y_mu = Y[i] # measurement
  likelihood = (y_mu, sensor_var)
  x_mu, x_var = update(prior, likelihood)

  # process estimates
  E[i] = x_mu
```



See, “L07_Kalman_in_python_1.py”

Kalman filter (2/2)

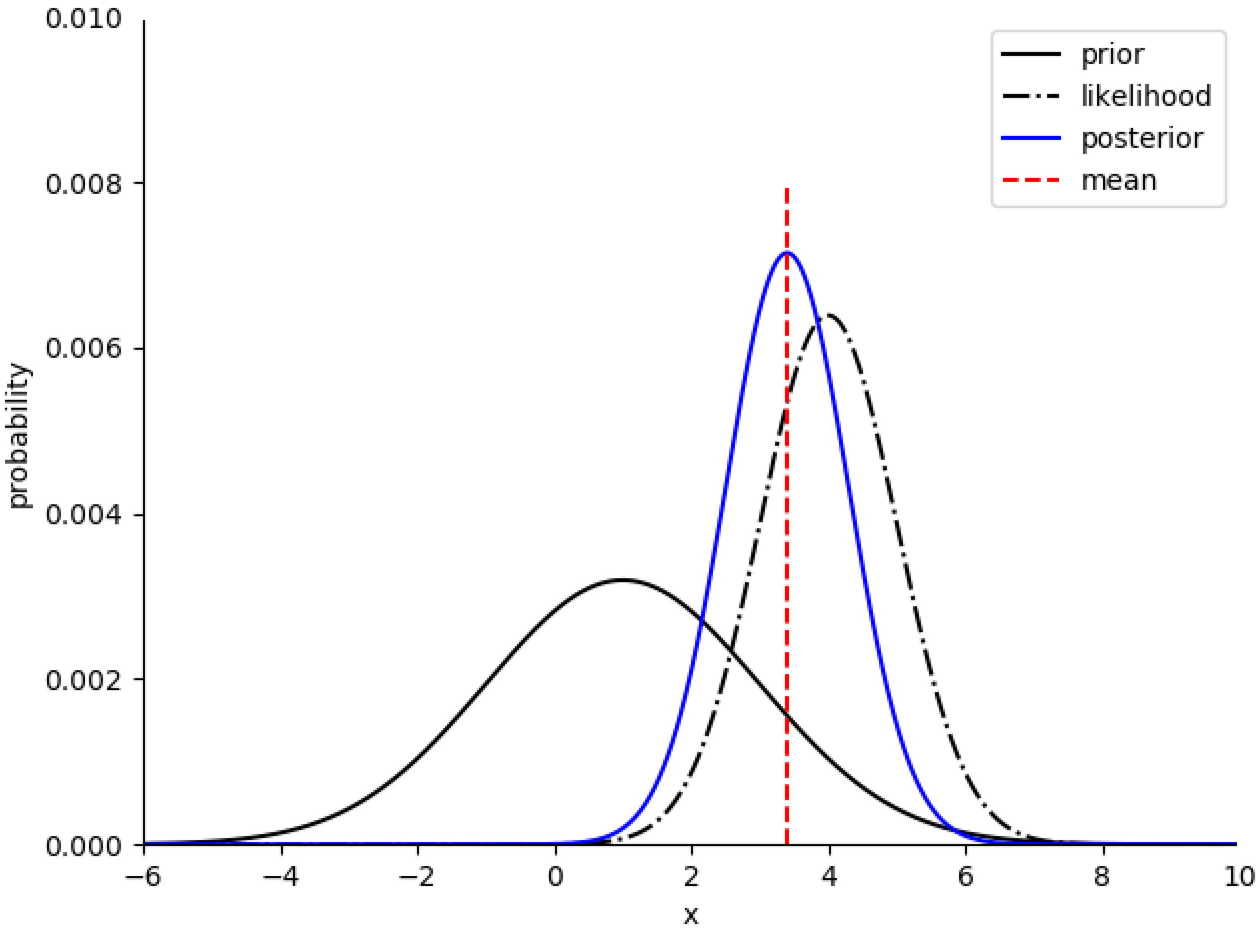
```
def predict(posterior, process):
  x, P = posterior # mean and var of posterior
  dx, Q = process  # mean and var of process

  x = x + dx
  P = P + Q


def update(prior, measurement):

  x, P = prior        # mean and var of prior
  y, R = measurement # mean and var of meas.

  a = y - x           # residual
  G = P / (P + R)     # Kalman gain

  x = x + G*a         # posterior mean
  P = (1 - G) * P     # posterior variance
```



See, “L07_Kalman_in_python_1.py”

Gaussian probabilities

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


def gaussian_multiply(prior, measurement):

  mu1, var1 = prior
  mu2, var2 = measurement
  mean = (var1*mu2 + var2*mu1) / (var1 + var2)
  variance = (var1 * var2) / (var1 + var2)

  return (mean, variance)
```



See, “L07_gaussian_pdf_multiplication.py”

Bayes theorem

Bayes theorem is

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

We implemented the `update()` function with this probability calculation:

$$\texttt{posterior} = \frac{\texttt{likelihood} \times \texttt{prior}}{\texttt{normalization}}$$

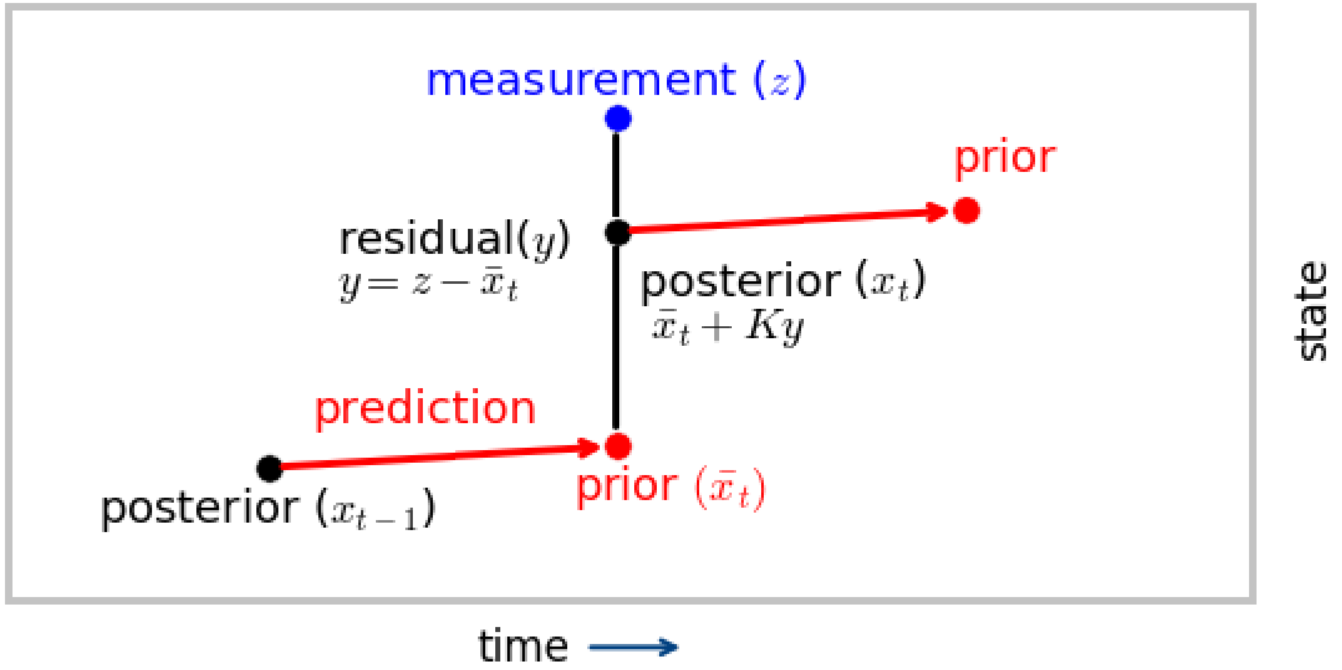
To review, the *prior* is the probability of something happening before we include the probability of the measurement (the *likelihood*) and the *posterior* is the probability we compute after incorporating the information from the measurement.

http://nbviewer.jupyter.org/github/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/table_of_contents.ipynb

Kalman filter flowchart

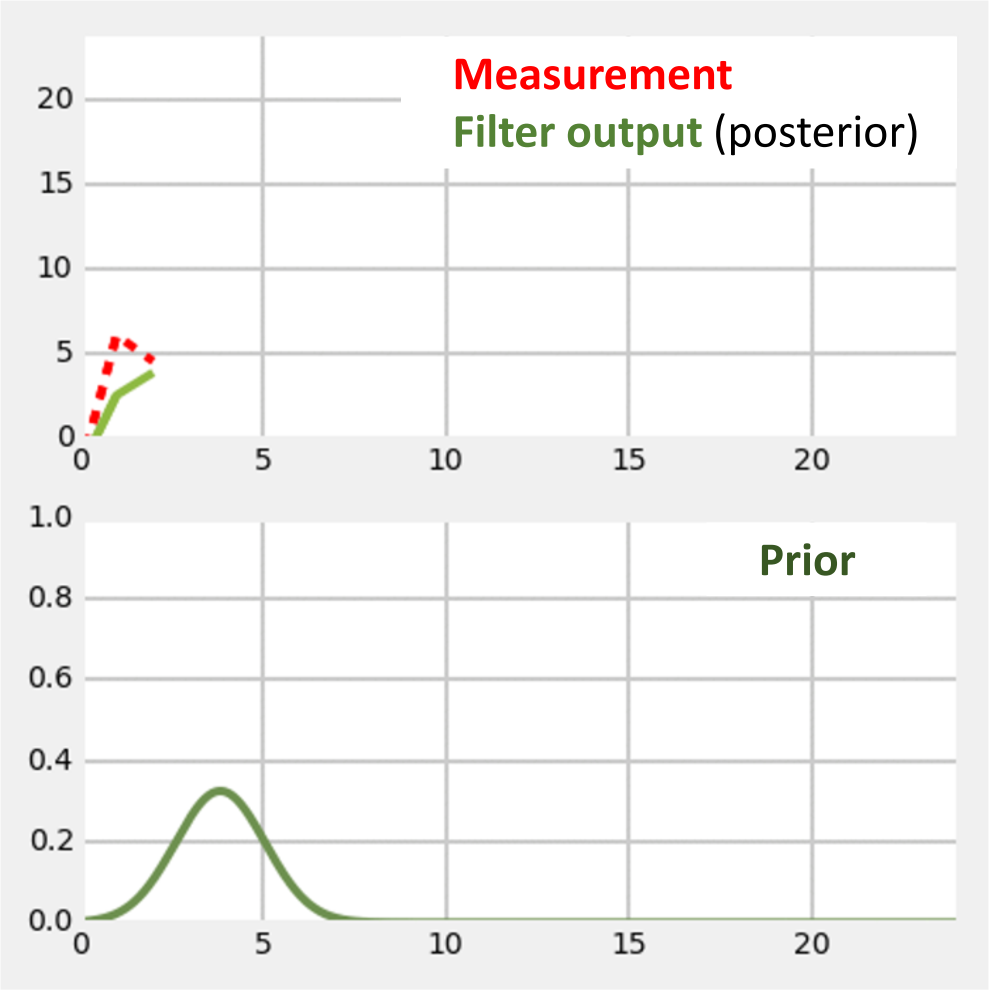
Chapter 4

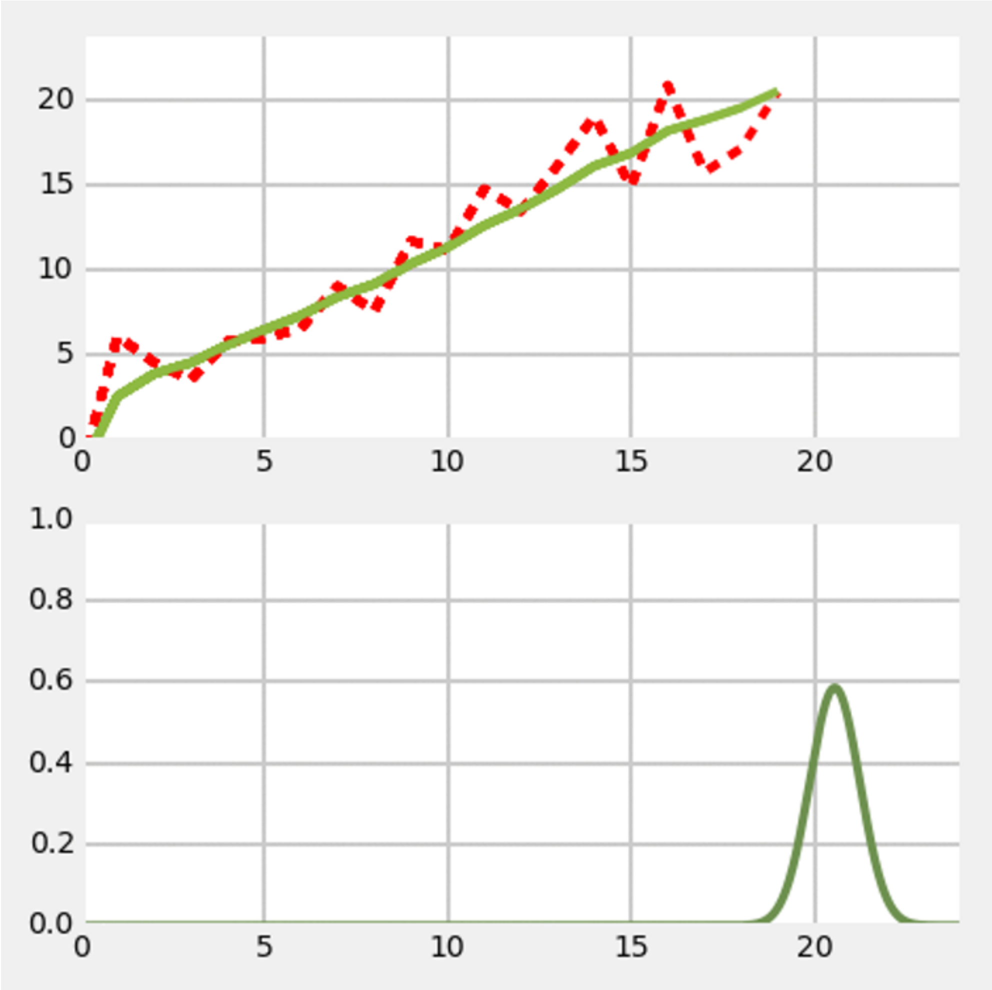
Algorithm flowchart



http://nbviewer.jupyter.org/github/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/table_of_contents.ipynb

Evolution of prior

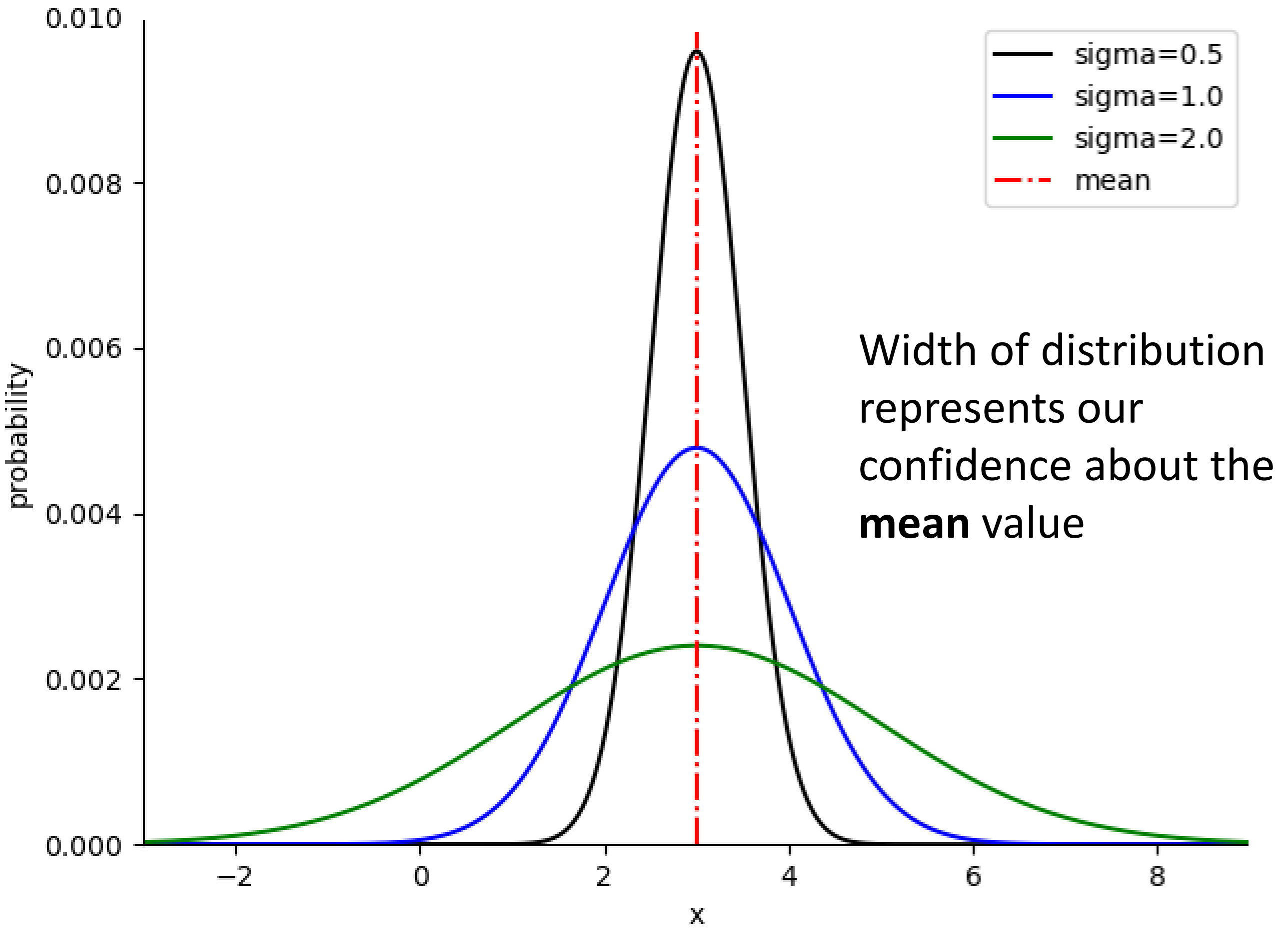




http://nbviewer.jupyter.org/github/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/table_of_contents.ipynb

Gaussian probabilities

```
# parameters
mu = 3.0

# binning
N = 1000
xmin = -6
xmax = 6
b = np.linspace(xmin+mu, xmax+mu, N)

# gaussian pdf
sigma = 0.5
p1 = norm.pdf(b, mu, sigma)
p1 = p1 / np.sum(p1)

sigma = 1.0
p2 = norm.pdf(b, mu, sigma)
p2 = p2 / np.sum(p2)

sigma = 2.0
p3 = norm.pdf(b, mu, sigma)
p3 = p3 / np.sum(p3)
```



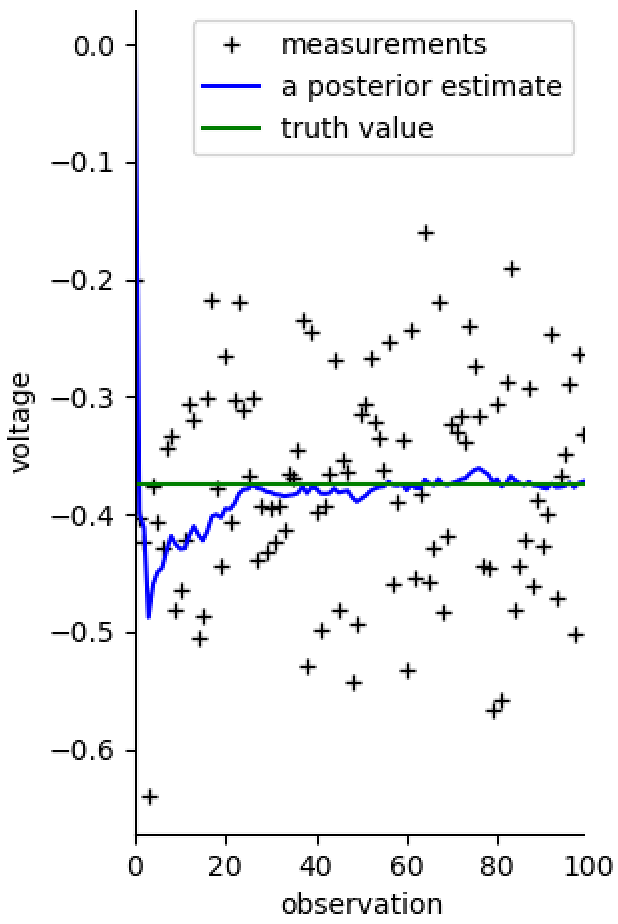
Width of distribution represents our confidence about the **mean** value

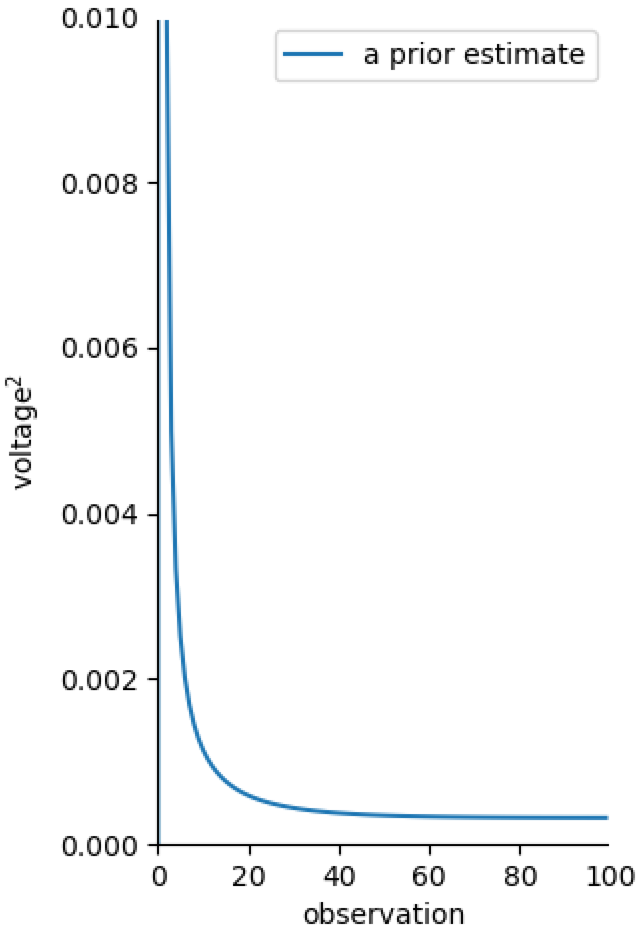
See, “L07_gaussian_pdf.py”

Section 4. Kalman filter in python, example 2

Kalman filter, example 2 (1/4)

```
# parameters
x = -0.375 # truth value (constant process)
y = np.random.normal(x, 0.1, size=N) # noisy
measurements
Q = 1e-5 # process variance
R = 0.01 # measurement variance

# initial guesses
z[0] = 0.0
P[0] = 1.0

# run Kalman filter
for n in range(1, N):
  # prediction
  zm = z[n-1]
  Pm = P[n-1] + Q          # prior
  # measurement update
  G[n] = Pm / (Pm + R)
  a = y[n] - zm
  z[n] = zm + G[n] * a
  P[n] = (1 - G[n]) * Pm   # posterior
```





Literature

- **Python programming language**

- http://www.scipy-lectures.org/, see “materials/L02_ScipyLectures.pdf”

- **Data analysis**

- Haykin, 2009, **“Neural networks and learning machines”**, 3rd edition (Chapter 14), “materials/*.pdf”
- http://nbviewer.jupyter.org/github/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/table_of_contents.ipynb

State-space representation

How does it related to the filtering problem?

```
# sampling parameters
fs = 1000   # sampling rate

# design filter in time domain
f0 = 25
[b, a] = signal.butter(4, f0 / (fs/2), 'low')

# state space representation of the system
[A, B, C, D] = signal.tf2ss(b, a)
```