State Estimation for Distribution Grids

Table of Contents

Table of figures

Introduction

This document is a reference for the state estimation of distribution grids implemented in python using various libraries and benchmark grids. The following libraries are used in the program:

1. Pandapower
   It is an easy to use network calculation program aimed at automation of analysis and optimization in power systems.
   You can install this from http://www.pandapower.org/start/

2. filterpy
   This library provides Kalman filtering and various related optimal and non-optimal filtering software written in Python.
   You can install this from https://github.com/rlabbe/filterpy

3. simbench
   This library contains several benchmark grid data and can be used within pandapower.
   You can install this from https://simbench.readthedocs.io/en/latest/about/installation.html

4. numpy
   installation https://numpy.org/

5. pandas
   installation https://pandas.pydata.org/

6. matplotlib
   installation https://matplotlib.org/users/installing.html

According to the requirement, various files inside these libraries are already edited (details are provided in the respective codes). Therefore, the following files should be placed in the working directory to override the respective files in the packages:

1. Matrix_calc.py
2. Matrix_base.py
3. ppc_conversion.py
4. EKF_iter.py

General outline

There are some fundamental differences between simbench and pandapower. The main difference is in the sign of power. Therefore, it is necessary to convert simbench convention to pandapower convention:

Simbench:

Static generation (sgen): generation is positive power

Pandapower:

Static generation (sgen): generation is negative power

Bus power flow results: generation is negative power and consumption is positive power

Measurements: generation is positive power and consumption is negative power.

General outline of the program:

1. Retrieve a grid and associated profile (timeseries data for loads and generation) from simbench
2. Create the necessary measurements for the grid for each timestep in the profile
3. Run the estimation (eg. Weighted Least Squares or Extended Kalman Filter)
4. Plot the error distribution for WLS and EKF compared to the true state

Retrieving a grid and associated profile

A pandapower network includes various parameter tables like:

- bus
- load
- sgen
- switch
- ext_grid
- line
- trafo
- bus_geodata
- loadcases
- measurements

Some of these tables are pandas dataframe while others are numpy arrays. The number of elements in each table will be according to the grid structure.

Creating measurements

There might be some measurements already available for the network. So, it is advisable to delete these before creating our own measurements. For the new measurements, a powerflow is first run to determine the real state of the grid. After the powerflow, the results will be available in the result tables of the network. For eg.,

‘net.res_bus’ has the bus results like voltage magnitude and angles, P and Q injections, etc.
‘net.res_line’ has the line results like P and Q flows.

Required measurements are then created by adding a random noise (sampled from a normal distribution with given standard deviation) over the true values. Measurements are in the order

- P_(non slack buses)
- P_(slack bus)
- P_(lines)
- Q_(non slack buses)
- Q_(slack bus)
- Q_(lines)
- V_(non slack bus)
- V_(slack bus)

Internal data structures

For the internal calculations within pandapower, there are various other data structures apart from the pandapower network. They are

'ppc' – An internal PyPower datastructure

'eppci' – An internal extended PyPower datastructure

Calculations such as finding the admittance matrix, powerflow, etc are done with 'eppci' internally. Below are the main differences between these datastructures:

Suppose for a sample grid ("1-MV-comm--0-sw" – a medium voltage commercial grid):

The number of buses in the pandapower network (called as 'net' from now on) is 144.

The number of buses in the 'ppc' is 155.

The number of buses in 'eppci' is 150.

Figure 1: Difference in no. of buses

In this grid, there are 139 bus in service, 5 bus not in service and 11 dummy bus for open switches.

net: 139 + 5 (Not seperated)

ppc: 139 + 11 + 5 (Seperated, first 'in service bus' followed by 'dummy bus' and last 'out of service bus' in the table)

eppci: 139 + 11 (Seperated, first 'in service bus' followed by 'dummy bus' in the table)

In the code, these are nBusAct = 139, nBuspp = 144 and nBus = 150.

Most of the calculations are done with 'eppci' and later the required results are extracted. The admittance matrix in this case will be a 150x150 matrix according to the bus number in 'eppci'.

In order to extract the required results (in this case states for 139 in service buses) from net, we need to create a mask. This is done from the data in net['_pd2ppc_lookups']. The repeated numbers indicate the 'out of service bus'. These results are masked out so that they do not appear in the result.

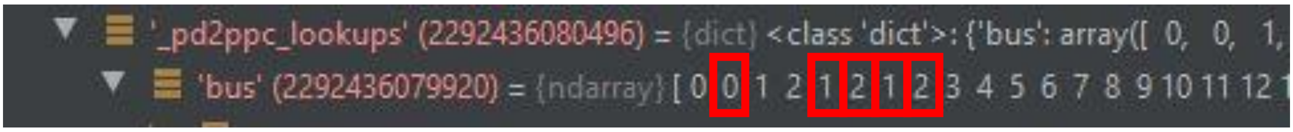


Figure 2: pd2ppc lookup table

Estimation

We want to estimate the values of voltage magnitudes and relative angles of each bus in the grid from the given measurements. If the grid has 'n' buses of which there are 'k' slack buses, at least '2n-k' measurements are needed theoretically for the estimation to converge, but practically '4n' measurements is often considered reasonable (Reference: https://pandapower.readthedocs.io/en/v2.0.1/estimation.html).

The pandapower library has a module called estimation which has several estimators such as:

- **Weighted least squares estimator (WLS):**
  Minimising the squared error of residues (difference between actual measurement and estimated measurement from a measurement function) weighted by measurement error covariance matrix.
- **Least Absolute value estimator (LAV):**
  Minimising the absolute error of residues.
- **Schweppe Huber Generalised Maximum likelihood estimator (SHGM):**
  This estimator combines both WLS and LAV estimators. It uses LAV for outlier points because squared norm contributes large values to the loss function in case of outliers.

We chose to do WLS estimation to compare the results with the Kalman Filter estimation and to visualize the error distribution.

Kalman Filter Formulation

Kalman Filter (KF) is a linear state estimator which combines measurements from the actual system and prediction from the model of the system to find the optimal estimate of the states. Extended

Kalman Filter (EKF) is the extension of KF to the nonlinear case. A detailed review of these are provided at the appendix of the document.

Since the power flow equations are nonlinear, we use EKF to estimate the states. In the Iterated Extended Kalman Filter (IEKF) approach, we are repeating the update/correction step until the difference between the old and new states after update is less than a given threshold.

The state vector ‘x’ is the voltage magnitudes and phase angles of each bus.

$$\mathrm{x} = [\theta_1 .. \theta_N, V_1 .. V_N]^T$$

Measurements ‘z’

$$\mathrm{z} = [P_{bus}, P_{line}, Q_{bus}, Q_{line}, V_{slackbus}]^T$$

Power flow equations:

$$g_P = P_i - V_i \sum_{j=1}^{N} V_j \left[G_{ij} cos\theta_{ij} + B_{ij} sin\theta_{ij}\right] = 0$$

$$g_Q = Q_i - V_i \sum_{j=1}^{N} V_j \left[G_{ij} sin\theta_{ij} - B_{ij} cos\theta_{ij}\right] = 0$$

Prediction step of EKF:

$$\mathrm{x}_\mathrm{k}^- = \mathrm{x}_{\mathrm{k}-1}^+$$

$$P_k^- = P_{k-1}^+ + Q$$

Correction step of EKF:

$$K_k = P_k^- H_k^T \left(H_k P_k^- H_k^T + R\right)^{-1}$$

$$x_k^+ = x_k^- + K_k (z_k - h_k)$$

$$P_k^+ = (I - K_k H_k) P_k^- (I - K_k H_k)^T + K_k R K_k^T$$

Where

Pk is the state error covariance matrix at time step ‘k’

Q is process noise caused due to linearization errors

K is the Kalman gain

R is the measurement error covariance matrix

H is the Jacobian matrix of the measurement function ‘h'

z is measurement vector

h is the measurement estimations

Discussion

In the reference paper "Power system state estimation based on Iterative Extended Kalman Filtering and bad data detection using normalized residual test" (and in some other papers as well), the authors propose to use the prediction step as

$$x_k^- = x_{k-1}^+ + J_k^{-1}[u_k - u_{k-1}] + {J_k}^{-1}e$$

Where 'u' is the power injections (input), $u = [P_{bus}, Q_{bus}] = [P_1 .. P_N, Q_1 .. Q_N]^T$

and 'J' is the jacobian of power flow equations with respect to states.

$$J = \begin{bmatrix} \frac{\partial g_{\mathrm{P}}}{\partial \theta} & \frac{\partial g_{\mathrm{P}}}{\partial \mathrm{V}} \\ \frac{\partial g_{\mathrm{Q}}}{\partial \theta} & \frac{\partial g_{\mathrm{Q}}}{\partial \mathrm{V}} \end{bmatrix}$$

And ${J_k}^{-1}e$ as the process noise.

But we did not get a good estimation when we used this approach. It also tends to have some numerical issues due to the inverting of sparse jacobian matrix.

Reference: https://ieeexplore.ieee.org/abstract/document/7064881

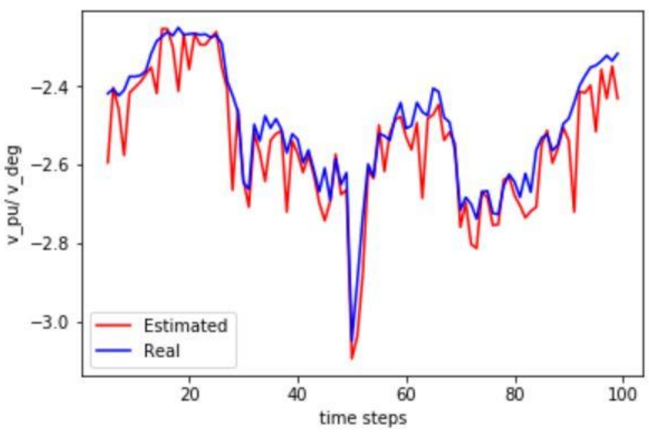


Figure 4: Estimation with Jacobian inverse

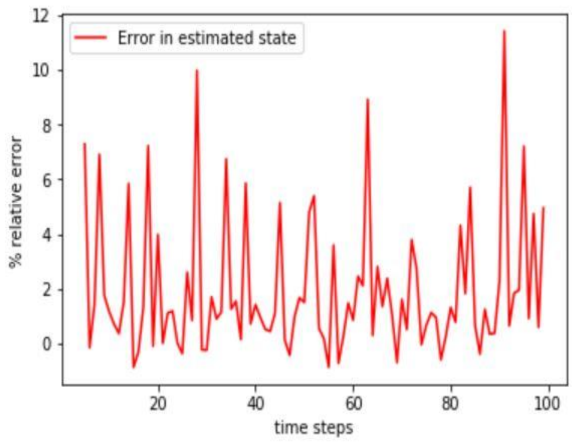


Figure 3: Relative error with Jacobian inverse

Then we decided to use the previous state as the next predicted state (identity matrix as the state transition matrix).

$$\mathrm{x}_{\mathrm{k}}^- = \mathrm{x}_{\mathrm{k-1}}^+$$

The results got improved considerably by following this approach for the update step. We think the problem with the previous method is that we are essentially using a subset of the measurement as the input and we therefore basically end up using part of measurement twice in the model.

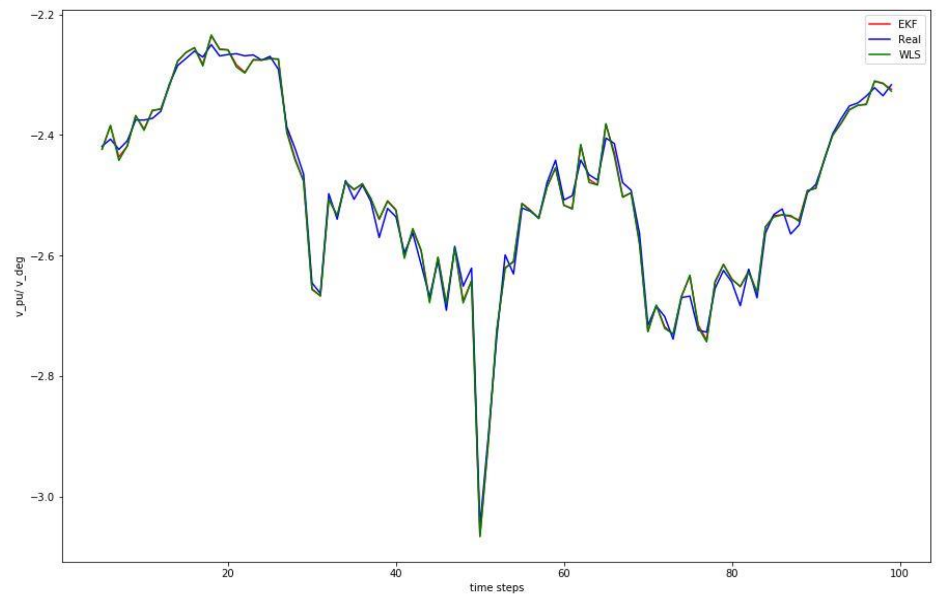


Figure 5: Estimation without Jacobian inverse

To get a general idea of the relative error distribution for IEKF compared to the WLS estimation, we ran the estimation for several times. Each time the measurement will be different even though they are from the same profile because of the random noise added on top of actual values.

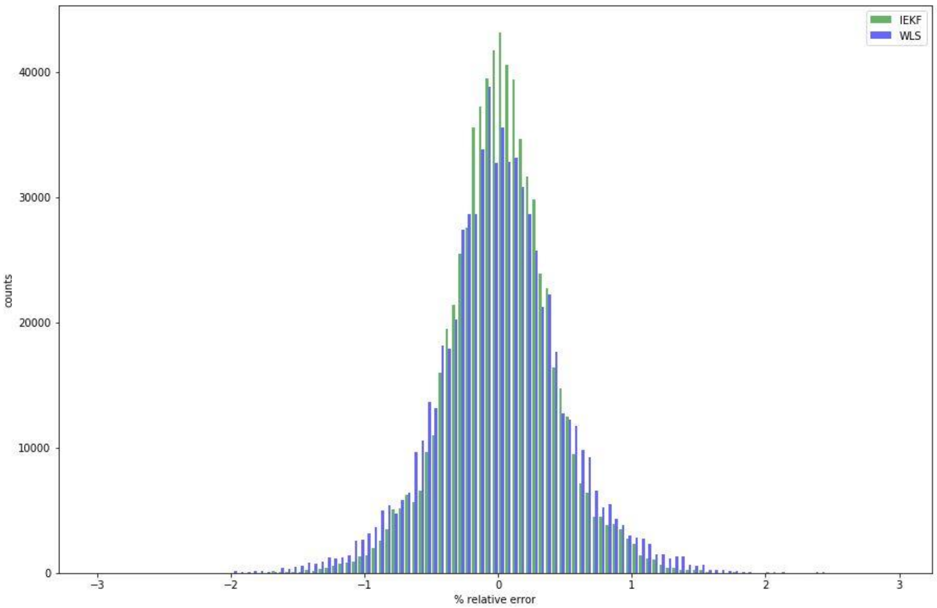


Figure 6: Error distribution for 50 runs

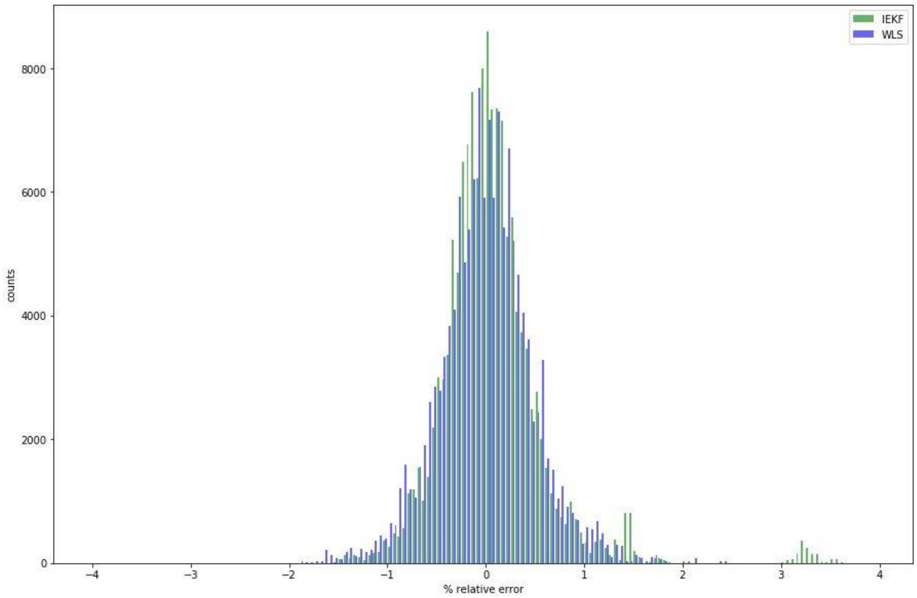


Figure 7: Error distribution for 10 runs

In the error distribution for just 10 runs, there are some peaks at higher error for IEKF (around 1.5% and 3 %) even though in general IEKF performs better than WLS (as seen from distribution for 50 runs). These peaks are due to the higher initial errors for IEKF in every run. This can be seen from the figures below. After some timesteps, the error covariance of the filter decreases and becomes adapted to the model.

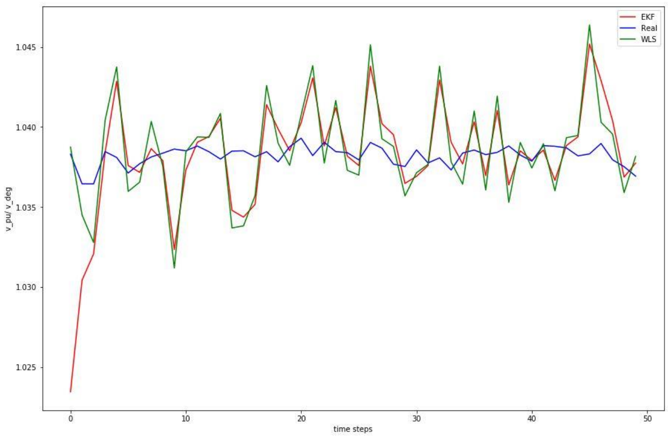


Figure 9: voltage magnitude estimation

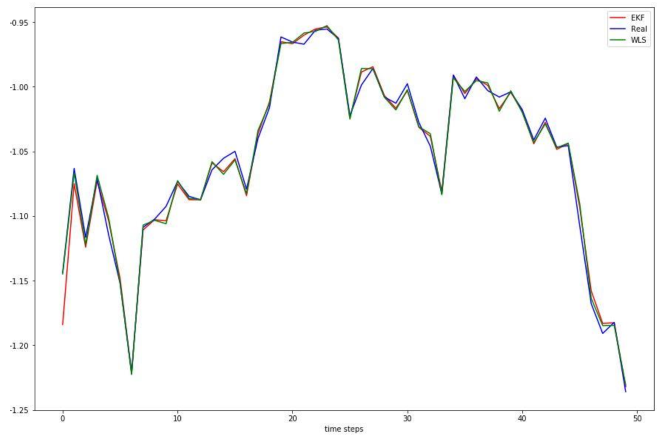


Figure 8: voltage angle estimation

It was observed that with large grids (for e.g., with 150 bus), sometimes the inversion of the matrix $\left(H_k P_k^- H_k^T + R\right)^{-1}$ leads to matrix inversion error if the state change of EKF does not converge to the given threshold within a small number of update steps (on average it converges below 10 update steps).

While debugging the inversion error, we found that the entries in the matrix are increasing after a particular update step and the values later starts to explode. The estimation error covariance matrix, P_k is increasing uncharacteristically which eventually leads to the inversion error. So, it means that we can check for the value of P_k and find out whether there is a chance of matrix inversion problem soon in this time step if we continue to execute the update step. We can then stop the update step for this timestep without the matrix inversion happening. But due to the uncharacteristic increase in the value of P_k , the state vector will be already corrupted and can't be used for that timestep.

Also, if we proceed to the next time step, it will again cause the inversion problem because our model is already corrupted (because we found out about possible inversion error from the corrupted values in P_k). One solution to this problem is to have a copy of the estimation object which is updated every timestep. We copy the current object before doing the prediction and update steps. If we find out that an inversion error is about to happen, we resort to some other means to find the state for that particular timestep (like a WLS estimation) and discard the corrupted object. For the next timestep we use the previous copy of the estimation object so that we don't run into inversion problem in the subsequent steps.

The trace of P_k for a normal run and for an error run is shown below:

```
Timestep: 25
Update step: 1.0, 0.003979568911779835
Update step: 2.0, 0.003977380255462606
Update step: 3.0, 0.003976417195866888
Update step: 4.0, 0.0039772047618974425
Update step: 5.0, 0.003977803561421578
Update step: 6.0, 0.003976979488657795
Update step: 7.0, 0.003978626713782375
Update step: 8.0, 0.003977222453303864
Update step: 9.0, 0.003976819538548271
Update step: 10.0, 0.003975200180241416

Update step: 47.0, 0.003976903910380287
Update step: 48.0, 0.0039767862369320395
Update step: 49.0, 0.003993587101194223
Update step: 50.0, 1.0746473519318798
Update step: 51.0, 64.63336454275625
Update step: 52.0, 14552.514726000541
Update step: 53.0, 42795.412365441385
Update step: 54.0, 6.173591732585055e+42
Update step: 55.0, 1.060484676549918e+93
Update step: 56.0, 2.9462984603917425e+173
```

Figure 11: Trace of Pk for error step

```
Timestep: 26
Update step: 1.0, 0.003984399537847206
Update step: 2.0, 0.0039818263109127425
Timestep: 27
Update step: 1.0, 0.003986929055822201
Update step: 2.0, 0.003985852887162666
Update step: 3.0, 0.00398549608052525
Update step: 4.0, 0.00398528986283079
Update step: 5.0, 0.003985155402591923
Timestep: 28
Update step: 1.0, 0.003991456094700998
Update step: 2.0, 0.003989336025882974
Timestep: 29
Update step: 1.0, 0.003996468596701499
Update step: 2.0, 0.003993402729483002
Update step: 3.0, 0.003993105915020942
Update step: 4.0, 0.003992940745121266
Update step: 5.0, 0.003992840421867042
Update step: 6.0, 0.003992753937843914
Timestep: 30
Update step: 1.0, 0.003998341189200915
Update step: 2.0, 0.003996795333773748
Update step: 3.0, 0.003996631617780348
```

Figure 10: Trace of Pk for normal steps

We can see that after 50 update steps, the matrix starts to explode.

From the above figures, we see that for normal runs, the trace of P_k continuously decrease with each update steps in a timestep. In the error timestep of 25, we can see that the trace of P_k increases in the 4th update step and then onwards it kind of oscillates without a general trend until collapsing.

So, we could also stop after the 4th timestep without considerably corrupting the estimation object.

We are analysing the filter for a possible reason for the increase in the variance of P_k . We looked at the measurement jacobian matrix H but could not find anything unusual happening to it.

In the present implementation, in case of this singularity error, a WLS estimation is done only for that timestep and revert to EKF from the next timestep onwards. Another possible way is to limit the maximum number of update steps in a timestep so that singularity does not arise.

Implementation

The various functions and their descriptions are given below:

We used PyCharm for debugging the code.

- ***find_state_and_measurements():***
  Runs the powerflow to find the real states, bus and line power measurements

- ***make_meas():***
  Create a single measurement in the pandapower network

- ***create_measurements():***
  To create the required number and type of measurements in the network

- ***getBusInfo():***
  Returns different bus numbers and slack bus index

- ***createResultMask():***
  Creates a mask to avoid unwanted buses in the final results

- ***storingArraysInit():***
  Initialise arrays to store the states for visualizing

- ***runWLS():***
  Runs the Weighted least squares estimation

- ***saveArrays():***
  To save the array of states locally for further testing if any

- ***plotError():***
  Plots the error distribution of WLS and EKF estimates

- ***plotStates():***
  Plotting the estimated and real values of a particular state variable to see the error per timesteps

- ***estimate_EKF():***
  Estimate the states with iterated Extended Kalman Filter

- ***runSimulations():***
  Run the estimation of states according the given profile for the given number of times

Appendix

A detailed review of Kalman Filter algorithm and context is given in the following figures:

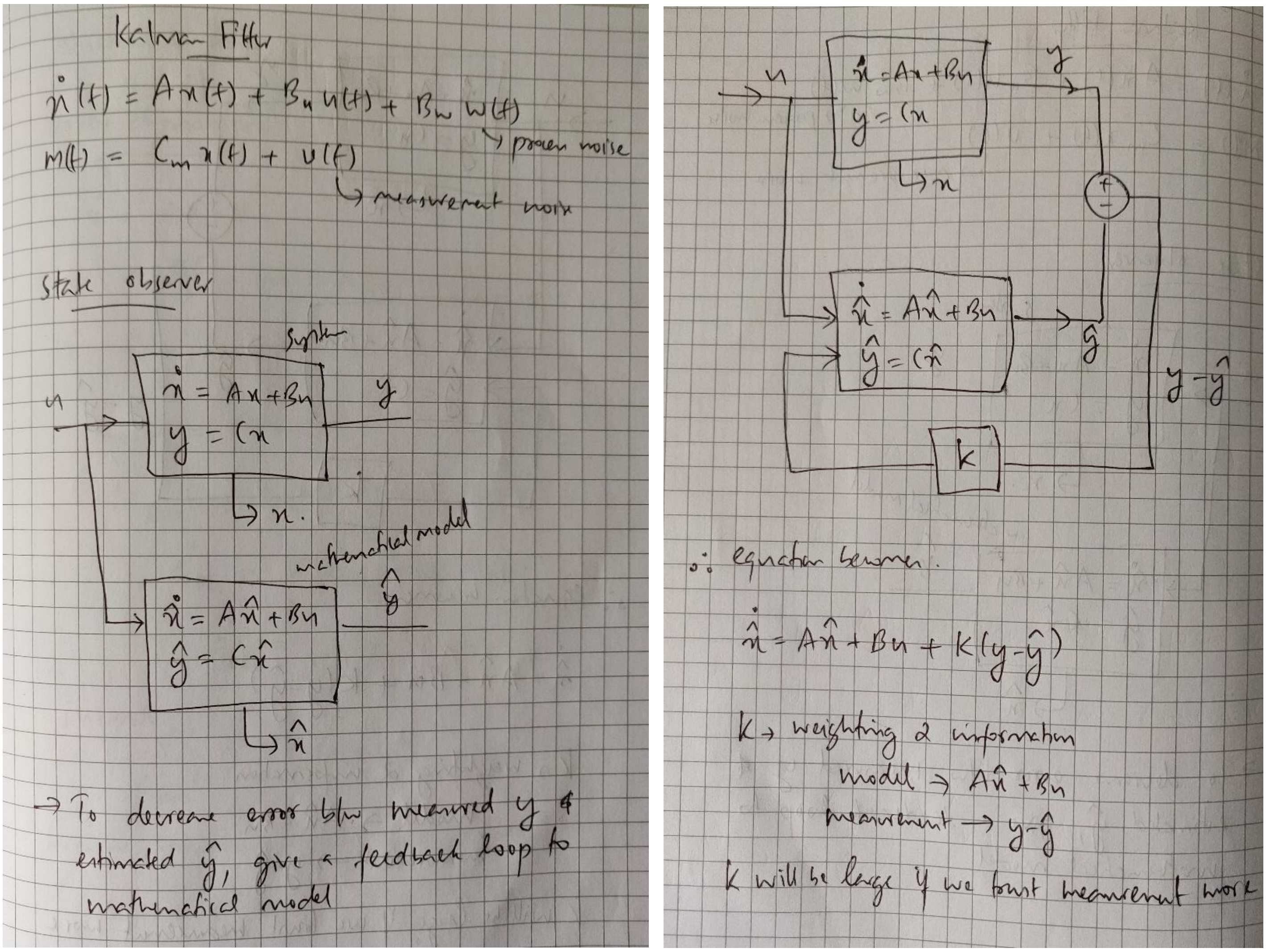


Figure 12: Kalman Filter fig. 1

observation error, $e_{obs} = x - \hat{x}$

error dynamics

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

$$\dot{\hat{x}} = A\hat{x} + Bu + k(y - \hat{y})$$

$$\dot{x} - \dot{\hat{x}} = Ax - A\hat{x} - k(y - \hat{y})$$

$$\dot{e}_{obs} = A(x - \hat{x}) - k(Cx - C\hat{x})$$

$$\dot{e}_{obs} = A(x - \hat{x}) - kC(x - \hat{x})$$

$$\dot{e}_{obs} = A e_{obs} - kC e_{obs}$$

$$\dot{e}_{obs} = (A - kC) e_{obs}$$

Integrating,

$$e_{obs}(t) = e^{(A-kC)t} e_{obs}(0)$$

because, solution of state space equation:

$$x(t) = \underbrace{e^{At} x(0)}_{\text{free motion}} + \underbrace{\int_0^t e^{A(t-\tau)} B u(\tau) \, d\tau}_{\text{forced motion}}$$

∴ If all eigen values of $[A - kC] < 0$ (real part < 0)

then $e_{obs} \to 0$ as $t \to \infty$

$$\Rightarrow \hat{x} \to x$$

* State observer equation

$$\dot{\hat{x}} = A\hat{x} + Bu + k(y - \hat{y})$$

In discrete time

$$\hat{x}_{k+1} = A\hat{x}_k + Bu_k + k(y_k - C\hat{x}_k)$$

Kalman Filter (linear filter)

measurement

$x_k = Ax_{k-1} + Bu_k + w_k$ ← process noise

$y_k = Cx_k + v_k$ ← measurement noise

model

$$\hat{x}_k = A\hat{x}_{k-1} + Bu_k$$

$$\hat{y}_k = C\hat{x}_k$$

Figure 13: Kalman Filter fig. 2

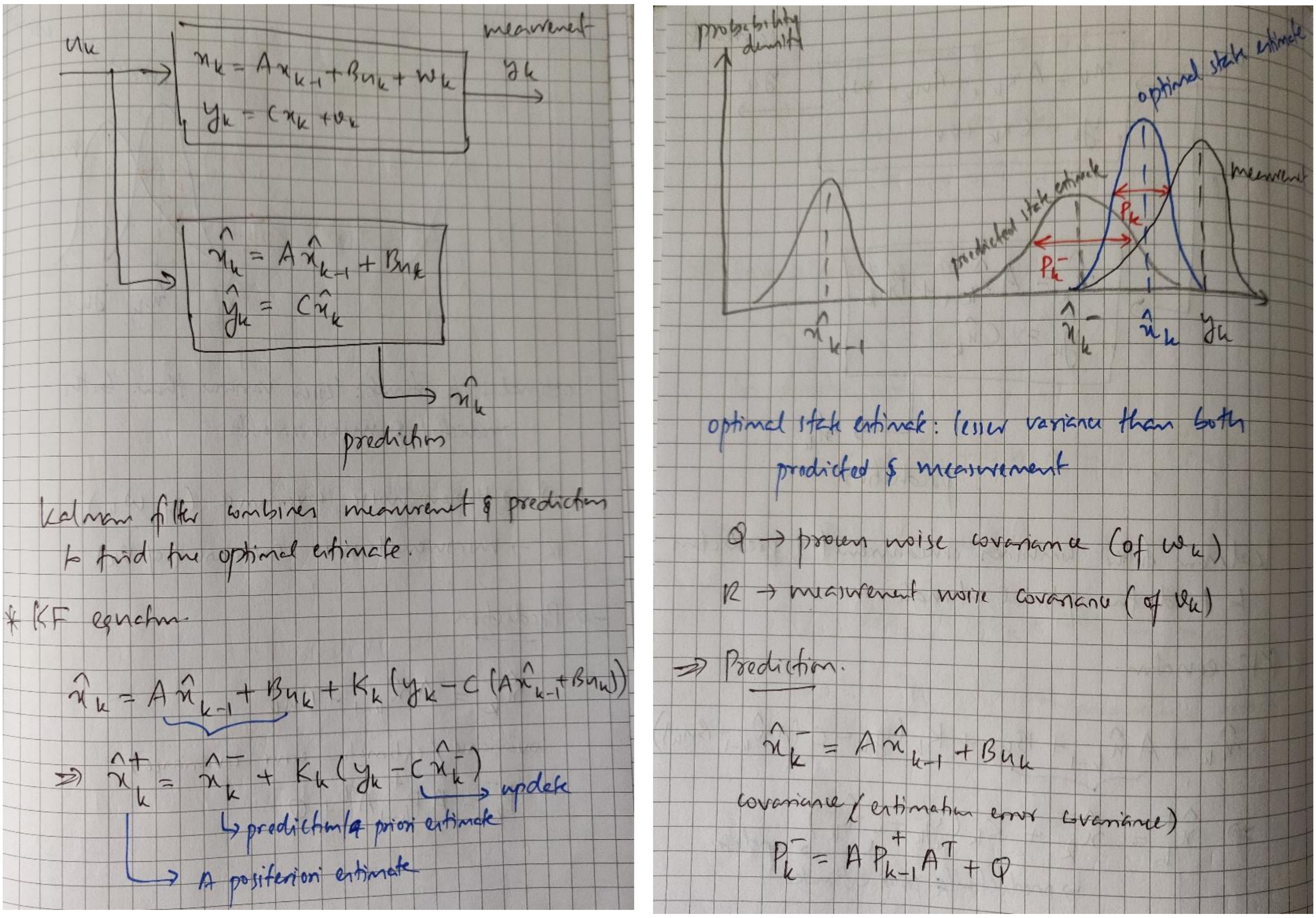


Figure 14: Kalman Filter fig. 3

⇒ Update

Kalman gain, $K_k = P_k^- C^T (C P_k^- C^T + R)^{-1}$

$$\hat{x}_k^+ = \hat{x}_k^- + K_k(y_k - C\hat{x}_k^-)$$

$$P_k^+ = (I - K_k C) P_k^- (I - K_k C) + K_k R K_k^T$$

or $P_k^+ = (I - K_k C) P_k^-$

Note:

If we trust measurement fully → $R = 0$

$$\lim_{R \to 0} K_k = C^{-1}$$

$$\hat{x}_k^+ = \hat{x}_k^- + C^{-1}(y_k - C\hat{x}_k^-)$$

$$\underline{\hat{x}_k^+ = C^{-1} y_k}$$

If we fully trust our model → $P_k^- = 0$

$$\lim_{P_k^- \to 0} K_k = 0$$

$$\hat{x}_k^+ = \hat{x}_k^- + 0$$

$$\underline{\hat{x}_k^+ = \hat{x}_k^-}$$

Kalman Filter is a linear filter that can be applied only to linear system.

But all systems are ultimately non-linear

Linearize a non-linear system

Taylor series expansion of a non-linear function around a nominal point $\bar{x}$

$$f(x) = f(\bar{x}) + f'(\bar{x})\Delta x + f''(\bar{x})\frac{(\Delta x)^2}{2!} + \cdots$$

where $\Delta x = x - \bar{x}$

eg: $\cos(x)$ around $\bar{x} = 0$

$$\cos(x) = \cos(0) - \sin(0)(x - 0) - \cos(0)\frac{(x-0)^2}{2} + \cdots$$

$$\Rightarrow \cos(x) = 1 - \frac{x^2}{2} + \cdots$$

∴ 1st order taylor series expansion = 1

2nd order taylor series expansion = $1 - \frac{x^2}{2}$

Figure 15: Kalman Filter fig. 4

If difference b/w x and $\bar{x}$ is small, taylor series approximation gives a good estimate

Linearizing a function ⇒ expanding the function in 1st order taylor series expansion around some expansion point.

Linearized Kalman Filter

→ start with a non linear system
→ find a linear system whose states represent deviations from normal trajectory of the non-linear system.
→ use Kalman Filter to estimate the deviations from normal trajectory of states
→ Indirectly gives in the estimation of states

General non-linear system model

State eqn: $x_{k+1} = f(x_k, u_k) + w_k$

o/p eqn: $y_k = h(x_k) + v_k$

$f(\cdot)$ → state equation
$h(\cdot)$ → measurement equation.

If any of these has a non-linear term
→ system non-linear.

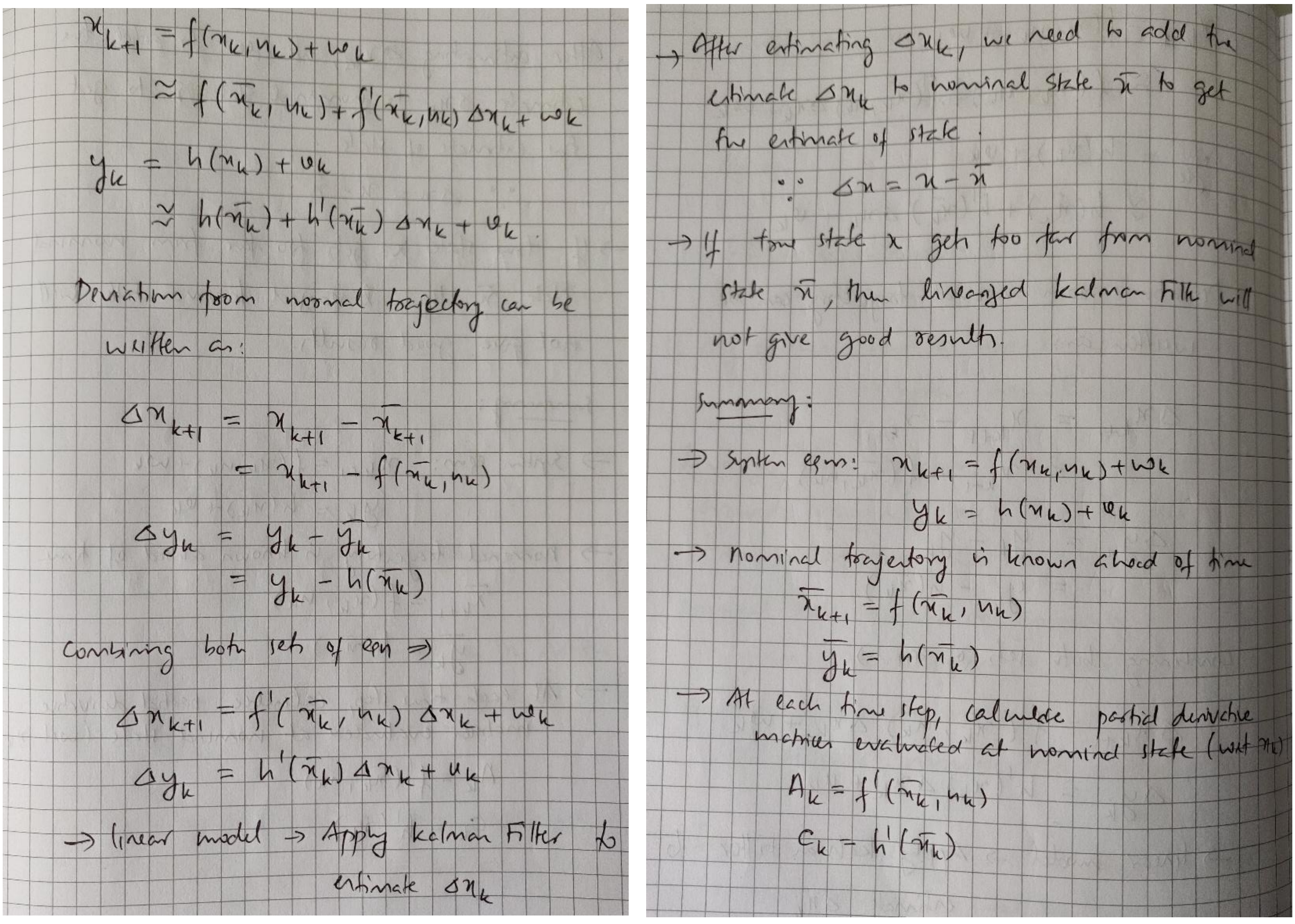
linearize state equation & output equation around nominal state (It is a function of time, so sometimes called trajectory)

nominal state → based on guess of what the system behaviour might look like

eg: if system equation represent dynamics of airplane, nominal state ≐ planned flight trajectory

but actual trajectory will be different, but if the difference is small → Taylor series expansion is reasonably accurate

Figure 16: Kalman Filter fig. 5



$$x_{k+1} = f(x_k, u_k) + w_k$$
$$\approx f(\bar{x}_k, u_k) + f'(\bar{x}_k, u_k)\,\Delta x_k + w_k$$
$$y_k = h(x_k) + v_k$$
$$\approx h(\bar{x}_k) + h'(\bar{x}_k)\,\Delta x_k + v_k .$$

Deviation from normal trajectory can be written as:

$$\Delta x_{k+1} = x_{k+1} - \bar{x}_{k+1}$$
$$= x_{k+1} - f(\bar{x}_k, u_k)$$

$$\Delta y_k = y_k - \bar{y}_k$$
$$= y_k - h(\bar{x}_k)$$

Combining both sets of eqn ⇒

$$\Delta x_{k+1} = f'(\bar{x}_k, u_k)\,\Delta x_k + w_k$$
$$\Delta y_k = h'(\bar{x}_k)\,\Delta x_k + u_k$$

→ linear model → Apply Kalman Filter to estimate Δx_k

→ After estimating Δx_k, we need to add the estimate Δx_k to nominal state $\bar{x}$ to get the estimate of state.

∵ $\Delta x = x - \bar{x}$

→ If true state x gets too far from nominal state $\bar{x}$, then linearized Kalman Filter will not give good results.

Summary:

→ System eqns: $x_{k+1} = f(x_k, u_k) + w_k$
$y_k = h(x_k) + v_k$

→ nominal trajectory is known ahead of time
$$\bar{x}_{k+1} = f(\bar{x}_k, u_k)$$
$$\bar{y}_k = h(\bar{x}_k)$$

→ At each time step, calculate partial derivative matrices evaluated at nominal state (wrt $\bar{x}_k$)
$$A_k = f'(\bar{x}_k, u_k)$$
$$C_k = h'(\bar{x}_k)$$

Figure 17: Kalman Filter fig. 6

→ Define $\Delta y_k = y_k - \bar{y}_k$

$= y_k - h(\bar{x}_k)$

→ Execute KF equations:

$$K_k = P_k C_k^T (C_k P_k C_k^T + R)^{-1}$$

$$\Delta\hat{x}_{k+1} = A_k \Delta\hat{x}_k + K_k(\Delta y_k - C_k \Delta\hat{x}_k)$$

$$P_{k+1} = A_k(I - K_k C_k) P_k A_k^T + Q$$

$$\hat{x}_{k+1} = \bar{x}_{k+1} + \Delta\hat{x}_{k+1}$$

Extended Kalman Filter

What if we don't know the nominal trajectory ahead of time?

EKF → to use our estimate of x as the nominal trajectory in linearized KF.

⇒ set $\bar{x} = \hat{x}$

Substituting this & after some mathematical manipulation, we get EKF algorithm.

→ System eqn: $x_{k+1} = f(x_k, u_k) + w_k$

$y_k = h(x_k) + v_k$

→ At each time step, compute the derivative matrices evaluated at current state estimate

$$A_k = f'(\hat{x}_k, u_k)$$

$$C_k = h'(\hat{x}_k)$$

→ Execute EKF equations:

$$K_k = P_k C_k^T (C_k P_k C_k^T + R)^{-1}$$

$$\hat{x}_{k+1} = f(\hat{x}_k, u_k) + K_k[y_k - h(\hat{x}_k)]$$

$$P_{k+1} = A_k(I - K_k C_k) P_k A_k^T + Q$$

Figure 18: Kalman Filter fig. 7