# Master Thesis. Imputation of Eddy Covariance meteorological time series using Kalman filters

Simone Massaro

January 2023

## Contents

# 1 Introduction

- problem
- state of the art
- our approach
  - uncertainties
  - combination of multiple sources of information [picture?]
  - custom implementation of Kalman filter imputation library
- why is relevant

## 1.1 Eddy Covariance and meteo

Eddy Covariance is a state of the art technique for measuring fluxes exchanges between ecosystems and the atmosphere [Uubinet, M., Vesala, T. Papale, D. (Eds.). Eddy Covariance: A Practical Guide to Measurement and Data Analysis (Springer Netherlands, Dordrecht, 2012)]. The technique allows fluxes measurements at a high temporal resolution and ecosystem level. The use of Eddy Covariance data is key is many ecology research [TODO cite]. The core of standard Eddy Covariance tower is the 3D anemometer and gas analyzers, which allows to estimated the fluxes of interests ($CO_2$, $H_20$). However those are not the only variable measured, as many additional on-site measurements are needed for [TODO cite something like ICOS]

- 
- why there are gaps in the first place [TODO would be good to find nice literature here]

## 1.2 Why need to do meteo gap filling in EC siste

- use of EC data
- ecological modelling that use ED data for validations (eg [5]) they need a complete input
- why it ts important to have the meteo measurements on the site
- gap filling for averages
- meteo are used as drivers for Fluxes gap filling
- use of uncertainties ??

### 1.3 State of the art

- OneFlux
- MDS
    - how does it work?
- ERA-Interim
- OzFlux ([1]) - use Australian meteo network -¿ not applicable
- limitations of current systems (show a plot?)

### 1.4 Distribution of gaps in Fluxnet

- a few very long gaps
- many short and medium gaps
- focus short gaps

### 1.5 Time series imputation

- why Kalman Filter
- comparison of other methods

## 2 Methods

### 2.1 Kalman Filter Theory

#### 2.1.1 Probabilistic Machine Learning

- probability
- Conditional probability
- Bayes theorem
- Gaussian Inference

#### 2.1.2 Notation

[figure:; table with variable names and visualization of gap]

- $t$ Number of time steps
- observations
    - $n$ Number of variables observed

- $y_{:,t}$ or $y_t$ vector of all the $n$ variables at time $t$, $\in \mathbb{R}^n$
- $y_{n,:}$ vector of the $n$th variable at for time steps in $t$, $\in \mathbb{R}^T$
- $y_{n,t}$ $n$th variable at time $t$, $\in \mathbb{R}$
- $Y_M = [x_{:,1}, ...x_{:,t}]$ Matrix with all the $n$ variables at all time steps, $\in \mathbb{R}^{n \times t}$
- $Y$ is a vector obtained by "flattening" $X_M$, by putting next to each other all variable at time $t$, $\in \mathbb{R}^{(n \cdot t)}$
- $y_t^{ng}$ vector of variable that are not missing (ng = not gap)) at time $t$, $\in \mathbb{R}^{n_{ng}}$. Note at different times the shape of this vector can change
- $Y^{ng}$ all observations

- Latent state
  - $k$ Number of variables in latent state
  - $x_{:,t}$ or $x_t$ vector of all the $k$ state variables at time $t$, $\in \mathbb{R}^k$
  - $x_{k,:}$ vector of the $k$th variable at for time steps in $t$, $\in \mathbb{R}^t$
  - $x_{k,t}$ $k$th variable at time $t$, $\in \mathbb{R}$
  - $X_M = [x_{:,1}, ...x_{:,t}]$ Matrix with all the $k$ variables at all time steps, $\in \mathbb{R}^{k \times t}$
  - $X$ is a vector obtained by "flattening" $X_M$, by putting next to each other all variable at time $t$, $\in \mathbb{R}^{(k \cdot t)}$

#### 2.1.3 Basics Kalman Filter

- why Kalman filter
- picture of Kalman filter state

**Description** The latent state ($x$) is modelled using a Markov chain. Which means that the state at time $t$ depends only on the state at time $t-1$ and not the states at previous times

**Assumptions**

$$p(x_t|x_{t-1}) = \mathcal{N}(Ax_{t-1} + b, Q) \tag{1}$$

The observation are derived from the state using a linear map plus random noise

$$p(y_t|x_t) = \mathcal{N}(Hx_t + d, R) \tag{2}$$

### 2.1.4 Time update

The probability distribution of state at time $t$ is computed using the state a time $t-1$
The state at time $t-1$ has a distribution

$$p(x_{t-1}) = \mathcal{N}(m_{t-1}, P_{t-1})$$

Combining this equation with equation 1 and using the properties of a linear map of a Gaussian distribution we obtain:

$$p(x_t) = \mathcal{N}(x_t; m_t^-, P_t^-) \tag{3}$$

where:

- predicted state mean: $m_t^- = Am_{t-1} + Bc_t + d$
- predicted state covariance: $P_t^- = AP_{t-1}A^T + Q$

The mean and the covariance of the state at time 0 are parameters of the models that are learned

### 2.1.5 Measurement update

The probability distribution of state at time $t$ is corrected using the observations at time $t$
This uses equation 2 and the formula for posterior distributions for Gaussian distributions.

$$p(x_t|y_t) = \mathcal{N}(x_t; m_t, P_t) \tag{4}$$

where:

- predicted obs mean: $z_t = Hm_t^- + d$
- predicted obs covariance: $S_t = HP_t^-H^T + R$
- Kalman gain $K_t = P_t^-H^TS_t^{-1}$
- corrected state mean: $m_t = m_t^- + K_t(y_t - z_t)$
- corrected state covariance: $P_t = (I - K_tH)P_t^-$

**Missing observations** If all the observations at time $t$ are missing the correct step is skipped and the filtered state at time $t$ (equation **??**) is the same of the filtered state.
If only some observations are missing a variation of equation **??** can be used.
$y_t^{ng}$ is a vector containing the observations that are not missing at time $t$.
It can be expressed as a linear transformation of $y_t$

$$y_t^{ng} = My_t$$

where $M$ is a mask matrix that is used to select the subset of $y_t$ that is observed. $M \in \mathbb{R}^{n_{ng} \times n}$ and is made of rows which are made of all zeros but for an entry 1 at column corresponding to the of the index non-missing observation.
For example, if $y_t = [y_{0,t}, y_{1,t}, y_{2,t}]^T$ and $y_{0,t}$ is the missing observation then

$$M = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

hence:

$$p(y_t^{ng}) = \mathcal{N}(M\mu_{y_t}, M\Sigma_{y_t}M^T)$$

from which you can derive

$$p(y_t^{ng}|x_t) = p(MHx_t + Mb, MRM^T) \tag{5}$$

Then the posterior $p(x_t|y_t^{ng})$ can be computed similarly of equation **??** as:

$$p(x_t|y_t^{ng}) = \mathcal{N}(x_t; m_t, P_t) \tag{6}$$

where:

- predicted obs mean: $z_t = MHm_t^- + Md$
- predicted obs covariance: $S_t = MHP_t^-(MH)^T + MRM^T$
- Kalman gain $K_t = P_t^-(MH)^T S_t^{-1}$
- corrected state mean: $m_t = m_t^- + K_t(My_t - z_t)$
- corrected state covariance: $P_t = (I - K_t MH)P_t^-$

#### 2.1.6 Smoothing

- Kalman smoothing gain: $G_t = P_t A^T (P_{t+1}^-)^{-1}$
- smoothed mean: $m_t^s = m_t + G_t(m_{t+1}^s - m_{t+1}^-)$
- smoothed covariance: $P_t^s = P_t + G_t(P_{t+1}^s - P_{t+1}^-)G_t^T$

#### 2.1.7 Predictions

The prediction at time t ($y_t$) are computed from the state ($x_t$) using:

$$p(y_t|x_t) = \mathcal{N}(y_t; \mu_{y_t}, O_t) \tag{7}$$
$$\mu_{y_t} = Hx_t + d \tag{8}$$
$$O_t = R + HP_t^s H^T \tag{9}$$

## 2.2 Kalman Filter Implementation

### 2.2.1 Requirements

For this application, the filter implementation needs to have:

- support for gaps
- support for computing gradients of parameters to maximise log-likelihood
- be numerically stable

Existing Kalman filter libraries in Python were evaluated. However, none of them met the requirements:

- statsmodels doesn't support gaps
- pykalman doesn't support parameters gradients and has numerical stability issues
- filterpy doesn't support gaps nor parameters gradients

Therefore a custom library for Kalman Filters was developed using the PyTorch library, which has the advantage of automatic differentiation, possibility to use GPUs and better integration with other Machine Learning methods.

### 2.2.2 Numerical stability

**Background** The direct implementation of Kalman filters is numerically unstable ([6] [3]) therefore the implementation needs to adapt strategies to improve the numerical stability of the filter.
Computers store numbers only with a limited number of decimal digits, hence operations that are mathematically possible cannot be actually implemented on a computer.
[what is numerical stability?]
For Kalman filter is relevant the numerical stability of the inversion of matrix on a digital computer depends on the condition number for inversion, which describes if the matrix is going to be singular on the numerical representation in the computer and thus cannot be inverted. The condition number is the ratio between the biggest singular value (square root of eigenvalues)

$$k(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)} \tag{10}$$

It is one for well-conditioned matrices [todo define] and tends to infinite for ill-conditioned matrices. The matrix cannot be inverted when the reciprocal of the condition number for inversion is close to the machine precision $1/k(A) < \varepsilon$.

**Mitigation strategies**

**Machine precision** The simplest to improve the numerical stability of the Kalman filter is to use higher accuracy representation of numbers, which in practice means to use 64bit floats instead of 32bit floats, which is default in PyTorch.

**Matrix decomposition** Another way to improve the numerical stability is to reduce the condition number of the matrices, in particular for the Kalman filter the key matrix is the state covariance ($P$). If the matrix $P$ is stored as its Cholesky factors $C$, such as that $P = CC^T$ the effective numerical resolution of the filter can be doubled. This can be proved as $\lambda(P) = \lambda^2(C)$. Therefore, if in the filter implementation $P$ is never explicitly computed, the numerical stability of the filter is significantly improved. There are several approached of filter implementations that follow this approach ([7], [todo cite other examples]) and are generally called "square-root filter", which have different tradeoff and often are computationally more expensive that the conventional implementation [need to actually check this].
[todo mention the fact that if we have a chol factor we'll always have a pos definite matrix]

### 2.2.3 Implementation in PyTorch

There are different approaches for square root filtering [cite ...]. According to [6] the best approach is the UD Filter ([2]) since it has the smallest computational cost. However, the filter is based on the $UD$ factorization and a custom matrix factorization [6] and both of those algorithms cannot be efficiently implemented in PyTorch. For the former factorization the `torch.linalg.ldl_factor` performs an $UD$, but it's an experimental function and is not differentiable. The latter would need to be implemented using scalar operations, which aren't efficient with PyTorch eager execution.
For this reason, a square root filter that propagates Cholesky factors of the covariance matrices is implemented. In this way all the required computations can be expressed in QR factorization, which is a numerically stable method and is a routine implemented in PyTorch

### 2.2.4 Time update Square Root Filter

The equations for the time update step of the filter are from [6] eq. 6.60 [cite original paper?]
For notation simplicity, we define $P_t = P_t^{1/2} P_t^{T/2}$ where $P_t^{1/2}$ is lower triangular matrix, and $P_t^{T/2} = (P_t^{1/2})^T$. In the same way, $Q = Q^{1/2} Q^{T/2}$. In the Kalman filter literature $P_t^{1/2}$, is the "square root" of $P_t$, hence the name of this type of filter. The Cholesky decomposition is an algorithm to find a square root of a matrix, however the Cholesky decomposition calculates only one of possibly many square roots of the matrix. In fact, $P_t^{1/2}$ is not unique.

The goal is to derive from the equations of prediction step (eq. 3) an algorithm to obtain $P_t^{1/2}$ given $P_{t-1}^{1/2}$, without explicitly computing $P_t$ or $P_{t-1}$. In this way, the numerical stability of the model is effectively doubled.
If we define

$$W = \begin{bmatrix} AP_{t-1}^{1/2} & Q^{1/2} \end{bmatrix} \tag{11}$$

then using 3 we can prove that

$$WW^T = P_t \tag{12}$$

$$\begin{aligned} WW^T &= \begin{bmatrix} AP_{t-1}^{1/2} & Q^{1/2} \end{bmatrix} \begin{bmatrix} P_{t-1}^{T/2} A^T \\ Q^{T/2} \end{bmatrix} \\ &= AP_{t-1}^{1/2} P_{t-1}^{T/2} A^T + Q^{1/2} Q^{T/2} = AP_{t-1}A^T + Q = P_t \end{aligned} \tag{13}$$

If we can factorize $W = LU$, where $L$ is a lower triangular matrix and $U$ is an orthogonal matrix, such as that $UU^T = I$. Then $WW^T = LU(LU)^T = LUU^T L^T = LL^T$ and by equation 21 $LL^T = P_t$. Hence, $L$ is a square root of $P_t$, which is what we were looking for.
This procedure never explicitly compute $P_t$ and requires only a triangularization of a matrix, which is implemented efficiently and in a numerical stable way by PyTorch `torch.linalg.qr` function.

**PyTorch implementation** PyTorch doesn't support natively a $LU$ decompositions. It implements the QR factorization: $W = QR$, where $Q$ is an orthogonal matrix and $R$ an upper triangular matrix. This can be easily converted into a $LU$ factorization, since by factorizing $W^T$ then $W^{=}QR = (QR^T) = R^T Q^T$ and $R^T$ is a lower triangular matrix.

**Summary** The steps of the Square Root time update are:

1. let $W = \begin{bmatrix} AP_{t-1}^{1/2} & Q^{1/2} \end{bmatrix}$
2. do a QR factorization $W^T = TR$
3. define $P_t^{1/2} = R^T$

### 2.2.5 Measurement update Square Root Filter

A similar procedure can be followed for the measurement update step of the filter. The equations here described are from [3].
The starting point is equation 4 and for simplicity the time subscript are omitted in the following equations
Letting:

$$M = \begin{bmatrix} R^{1/2} & H(P^-)^{1/2} \\ 0 & (P^-)^{1/2} \end{bmatrix} \tag{14}$$

$$V = \begin{bmatrix} S^{1/2} & 0 \\ \bar{K} & P^{1/2} \end{bmatrix} \tag{15}$$

$$\bar{K} = KS^{1/2} \tag{16}$$

Then we can prove that

$$MM^T = UU^T \tag{17}$$

Therefore, if we decompose $M = LU$ then $LL^T = UU^T$ and the bottom left block of size $k \times k$ of $L$ is a square root of $P$.
Here we prove equation 17

$$\begin{aligned} MM^T &= \begin{bmatrix} R^{1/2} & HP^- \\ 0 & (P^-)^{1/2} \end{bmatrix} \begin{bmatrix} R^{T/2} & 0 \\ (P^-)^{T/2}H^T & (P^-)^{T/2} \end{bmatrix} = \\ &= \begin{bmatrix} R^{1/2}R^{T/2} + H(P^-)^{1/2}P^-)^{T/2}H^T & HP^-)^{1/2}P^-)^{T/2} \\ (P^-)^{T/2}P^-)^{1/2}H^T & (P^-)^{1/2}P^-)^{T/2} \end{bmatrix} = \\ &= \begin{bmatrix} S & HP^- \\ (P^-)^T H^T & P^- \end{bmatrix} \end{aligned} \tag{18}$$

$$\begin{aligned} VV^T &= \begin{bmatrix} S^{1/2} & 0 \\ \bar{K} & P^{1/2} \end{bmatrix} \begin{bmatrix} S^{T/2} & \bar{K}^T \\ 0 & P^{T/2} \end{bmatrix} = \begin{bmatrix} S^{1/2}S^{T/2} & S^{1/2}\bar{K}^T \\ \bar{K}S^{T/2} & \bar{K}\bar{K}^T + P^{1/2}P^{T/2} \end{bmatrix} \\ &= \begin{bmatrix} S & S^{1/2}S^{T/2}K^T \\ KS^{1/2}S^{T/2} & KS^{1/2}S^{T/2}K^T + P \end{bmatrix} \\ &= \begin{bmatrix} S & HP^- \\ P^-H^T & KHP + P \end{bmatrix} \end{aligned} \tag{19}$$

We can see that all blocks of $VV^T$ are clearly equal to $MM^T$, but the bottom left one, which is equal due to the filter update for the covariance (equation 4).

**Summary** The steps of the Square Root measurement update are:

1. let $M = \begin{bmatrix} R^{1/2} & H(P^-)^{1/2} \\ 0 & (P^-)^{1/2} \end{bmatrix}$
2. do a QR factorization of $M^T = TV$
3. $P^{1/2}$ is the bottom left $k \times k$ block of $V^T$

### 2.2.6 Predictions Square Root Filter

The prediction equation for the square root filter are similar to the equations for the time update.
We define:

$$W = \begin{bmatrix} HP_{t-1}^{1/2} & R^{1/2} \end{bmatrix} \tag{20}$$

then using equation 3 we can prove that

$$WW^T = O_t \tag{21}$$

$$\begin{aligned} WW^T &= \begin{bmatrix} AP_{t-1}^{1/2} & R^{1/2} \end{bmatrix} \begin{bmatrix} P_{t-1}^{T/2} H^T \\ R^{T/2} \end{bmatrix} \\ &= HP_{t-1}^{1/2} P_{t-1}^{T/2} H^T + R^{1/2} R^{T/2} = HP_{t-1}H^T + R = O_t \end{aligned} \tag{22}$$

**Summary** The steps of the Square Root predictions are:

1. let $W = \begin{bmatrix} HP_{t-1}^{1/2} & R^{1/2} \end{bmatrix}$
2. do a QR factorization of $W^T = TU$
3. define $O_t^{1/2} = U^T$

### 2.2.7 Smoother

[Still need to find equations for square root smoother]

## 2.3 PyTorch module

The key of the Kalman filter library is two PyTorch modules, one that implement the equations of the standard filter and another for the square root filter. One limitation of using PyTorch arises from the fact that the filter is an iterative algorithm, where the states need to be computed sequentially. This makes it impossible to use the vectorization features of PyTorch and affects performance, especially if using GPUs. In order to mitigate this issue, all functions support batches, hence all the operations of the filter are performed in parallel across the different batches.

### 2.3.1 Parameter constraints

An important aspect for implementing a Kalman Filter in PyTorch is having constraints on the covariances parameters. There are 3 parameters in the filters that represent covariances (i.e. $Q$, $R$ and $P0$), which need to be enforced as positive definite matrices.
The positive definite ...

## 2.4 Model Training

### 2.4.1 Data preparation pipeline

### 2.4.2 Hyperparameters

## 2.5 Loss Function

### 2.5.1 Joint distribution of the gap

The goal is to obtain the joint distribution of the variables in the gap $Y^g$, which is $[y_t^g, y_{t+1}^g ... y^g(t+t_g)]$ for a gap that goes from $t$ to $t+t_g$. $Y^g \in \mathbb{R}^{t_g \times n_g}$, where $n_g$ is the number of variables missing in the gap.
For simplicity we are assuming for now that during the gap the variables missing don't change.
The goal is to obtain $p(Y^g|Y^n g)$
From the Kalman smoother it's easy to obtain $p(y_t^g|Y^{ng}) = \mathcal{N}(\mu_t, \Sigma_t)$
However, the problem is that $y_t^g$ and $y_{t+1}^g$ are not independent so it gets more complex. Assuming that $p(y_t^g|y_{t+1}^g) = \mathcal{N}(\mu_{t,t+1}, \Sigma_{t,t+1})$ the joint distribution has the form:

$$p(Y^g|Y^{ng}) = \mathcal{N}\left(\begin{array}{ccccc} \mu_t & \Sigma_t & \Sigma_{t,t+1} & \cdots & \Sigma_{t,t+t_g} \\ \mu_{t+1} & \Sigma_{t+1,t} & \Sigma_{t+1} & \cdots & \Sigma_{t+1,t+t_g} \\ \cdots & \vdots & \vdots & \ddots & \cdots \\ \mu_{t+t_g} & \Sigma_{t+t_g,t} & \Sigma_{t+t_g,t+1} & \cdots & \Sigma_{t+t_g} \end{array}\right)$$

$p(Y_g|Y_{ng}) = \int p(Y_g|X_g)p(X_g|Y)dX_g$

### 2.5.2 Joint distribution state for gaps

**Two states** For simplicity, I am starting with the joint distribution of the filter on a gap where there are no observations and are interested only on the joint distribution of two consecutive states. The aim is to find $p(x_t, x_{t+1} \mid x_t, Y_{1:t})$
The starting point is:

- $x_{t+1} = Ax_t + \varepsilon_{t+1}$
- $p(x_t \mid Y_{1:t}) = \mathcal{N}(x_t; m_t, P_t)$
- $p(\varepsilon_t) = \mathcal{N}(\varepsilon_t; 0, Q)$

Since all distributions are Gaussian, the joint distribution is also Gaussian

$$\begin{aligned} p(x_t, x_{t+1}|x_t) &= \mathcal{N}\left(\begin{bmatrix} x_t \\ x_{t+1} \end{bmatrix}; \begin{bmatrix} m_t \\ Am_t \end{bmatrix}, \Sigma_{x_t,x_{t+1}}\right) \\ \Sigma_{x_t,x_{t+1}} &= \begin{bmatrix} \langle (x_t - \mu_{x_t})(x_t - \mu_{x_t})^T) \rangle & \langle (x_t - \mu_{x_t})(x_{t+1} - \mu_{x_{t+1}})^T \rangle \\ \langle (x_{t+1} - \mu_{x_{t+1}})(x_t - \mu_{x_t})^T \rangle & \langle (x_{t+1} - \mu_{x_{t+1}})(x_{t+1} - \mu_{x_{t+1}})^T \rangle \end{bmatrix} \end{aligned} \tag{23}$$

we can compute the covariance using the expectation operator and its properties.

**Second element on the diagonal**

$$
\begin{aligned}
&\langle (x_{t+1} - \mu_{x_{t+1}})(x_{t+1} - \mu_{x_{t+1}})^T \rangle = \\
&= \langle (Ax_t + \varepsilon_{t+1} - Am_t)(Ax_t + \varepsilon_{t+1} - Am_t)^T \rangle = \\
&= \langle (A(x_t - m_t) + \varepsilon_{t+1})(A(x_t - m_t) + \varepsilon_{t+1})^T \rangle = \\
&= \langle A(x_t - m_t)(x_t - m_t)^T A^T + \varepsilon_{t+1}(x_t - m_t)^T A^T + A(x_t - m_t)\varepsilon_{t+1}^T + \varepsilon_{t+1}\varepsilon_{t+1}^T \rangle = \\
&= \langle A(x_t - m_t)(x_t - m_t)^T A^T \rangle + \langle \varepsilon_{t+1}(x_t - m_t)^T A^T \rangle + \langle A(x_t - m_t)\varepsilon_{t+1}^T \rangle + \langle \varepsilon_{t+1}\varepsilon_{t+1}^T \rangle = \\
&= A\langle (x_t - m_t)(x_t - m_t)^T \rangle A^T + 0 + 0 + \langle \varepsilon_{t+1}\varepsilon_{t+1}^T \rangle = \\
&= AP_tA^T + Q
\end{aligned}
\tag{24}
$$

**off-diagonal element**

$$
\begin{aligned}
&\langle (x_{t+1} - \mu_{x_{t+1}})(x_t - \mu_{x_t}^T \rangle = \langle (Ax_t + \varepsilon_{t+1} - Am_t)(x_t - Am_t)^T \rangle = \\
&= \langle A(x_t - m_t)(x_t - m_t)^T + \varepsilon_{t+1}(x_t - m_t)^T \rangle = \\
&= \langle A(x_t - m_t)(x_t - m_t)^T \rangle + \langle \varepsilon_{t+1}(x_t - m_t)^T A^T \rangle = \\
&= A\langle (x_t - m_t)^T \rangle + 0 = \\
&= AP_t
\end{aligned}
\tag{25}
$$

**Joint distribution state** substituting in equation 23:

$$
p(x_t, x_{t+1} \mid x_t, Y_{1:t}) = \mathcal{N}\left( \begin{bmatrix} x_t \\ x_{t+1} \end{bmatrix}; \begin{bmatrix} m_t \\ Am_t \end{bmatrix}, \begin{bmatrix} P_t & AP_t \\ AP_t & AP_tA^T + Q \end{bmatrix} \right) \tag{26}
$$

**Multiple States** A similar reasoning can be applied to more than two states, but the equations become more complex
To obtain $p(x_t, x_{t+1}, x_{t+2} \mid x_t, Y_{1:t})$ we also need to compute $\langle x_t x_{t+2}^T \rangle$ and $\langle x_{t+2} x_{t+2}^T \rangle$

**Covariance diagonal**

$$
\begin{aligned}
&\langle (x_{t+2} - \mu_{x_{t+2}})(x_{t+2} - \mu_{x_{t+2}})^T \rangle = \\
&= \langle (A(Ax_t + \varepsilon_{t+1}) + \varepsilon_{t+2} - AAm_t)(A(Ax_t + \varepsilon_{t+1}) + \varepsilon_{t+2} - AAm_t)^T \rangle = \\
&= \langle (AAx_t + A\varepsilon_{t+1} + \varepsilon_{t+2} - AAm_t)(AAx_t + A\varepsilon_{t+1} + \varepsilon_{t+2} - AAm_t)^T \rangle = \\
&= \langle AA(x_t - m_t)(x_t - m_t)^T A^T A^T \rangle + \langle A\varepsilon_{t+1}\varepsilon t + 1^T A^T \rangle + \langle \varepsilon_{t+2}\varepsilon_{t+2}^T \rangle = \\
&= AAP_t(AA)^T + AQA^T + Q
\end{aligned}
\tag{27}
$$

which (probably) can be generalized as: [TODO actually need to prove this and check that notation is correct]

$$\langle (x_t - \mu_{x_t})(x_{t+k} - \mu_{x_{t+k}})^T \rangle = A^k P_t (A^k)^T + \sum_{i=0}^{k-1} A^i Q (A^i)^T \tag{28}$$

**Covariance off-diagonal**

$$\langle (x_{t+k} - \mu_{x_{t+k}})(x_{t+k} - \mu_{x_{t+k}})^T \rangle = A^k P_t (A^k)^T \tag{29}$$

**Mean**

$$\langle x_{t+k} \rangle = A^k m_t \tag{30}$$

**Joint distribution state** In this way it is possible to obtain $P(X)$ for any number of states.

$$\begin{aligned} p(X_{t:t+k} \mid x_t, Y_{1:t}) &= \mathcal{N}\left( \begin{bmatrix} x_t \\ \vdots \\ x_{t+k} \end{bmatrix}; \begin{bmatrix} m_t \\ \vdots \\ A^k m_t \end{bmatrix}, \Sigma_{x_t, x_{t+k}} \right) \\ \Sigma_{x_t, x_{t+k}} &= \begin{bmatrix} P_t & \cdots & A^k P_t (A^k)^T \\ \vdots & \ddots & \vdots \\ A^k P_t (A^k)^T & \cdots & A P_t (A^k)^T + \sum_{i=0}^{k-1} A^i Q (A^i)^T \end{bmatrix} \end{aligned} \tag{31}$$

### 2.5.3 Joint distribution state - partial observations

In the case the there are partial observations to the reasoning of the previous paragraph cannot be applied as by combining equations 3 and 6

$$\begin{aligned} m_t^- &= A m_{t-1} + B c_t + d \\ P_t^- &= A P_{t-1} A^T + Q \\ z_t &= M H m_t^- + M d \\ S_t &= M H P_t^- (M H)^T + M R M^T \\ K_t &= P_t^- (M H)^T S_t^{-1} \\ m_t &= m_t^- + K_t (M y_t - z_t) \\ P_t &= (I - K_t M H) P_t^- \\ p(x_t | x_{t-1}, y_t^{ng}) &= \mathcal{N}(x_t; m_t, P_t) \end{aligned} \tag{32}$$

From this equation is not possible to write $x_t$ and linear map of $x_{t-1}$ plus another random variable, since the mean of $x_t$ depends on the covariance of $x_{t-1}$
For the same reason this approach cannot be applied for the smoother.

## 2.6 Metrics

- RMSE
- Normalized RMSE
- ?

## 2.7 Dataset

- fluxnet 2015 dataset
- Hainich
- ERA-Interim
- description variables

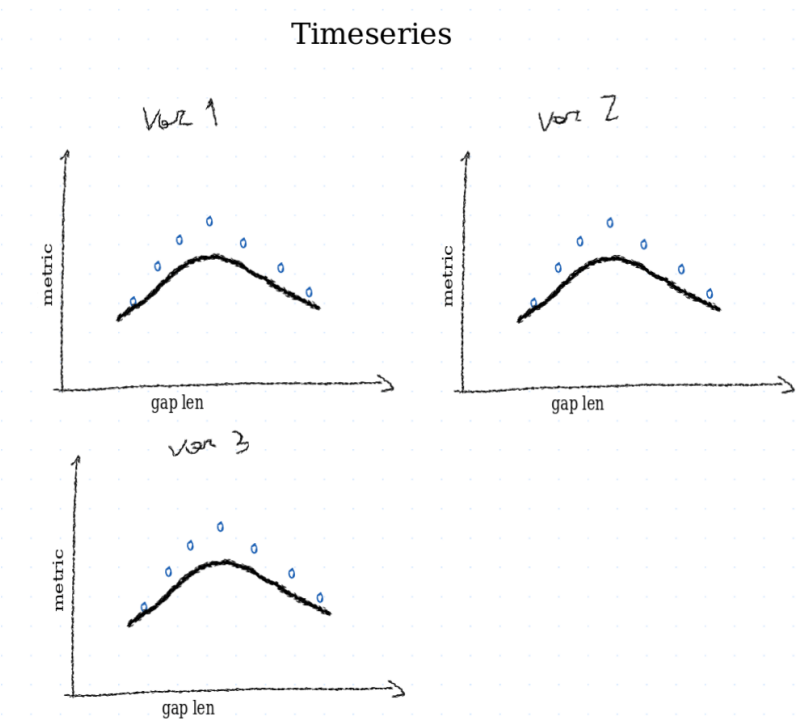
### 2.7.1 Marginal Distribution Sampling implementation

The implementation of the MDS used in the results comparison is from REddyProc ([8]). The algorithm for gap-filling is the same used in the OneFlux pipeline. REddyProc was used because it has an R interface that was easier to use than the C interface from OneFlux. The function `REddyProc::sEddyProc_sMDSGapFill` was used with the defaults setting for gap-filling where used, which are to use tree variables as predictors ()

# 3 Results

## 3.1 Example Timeseries

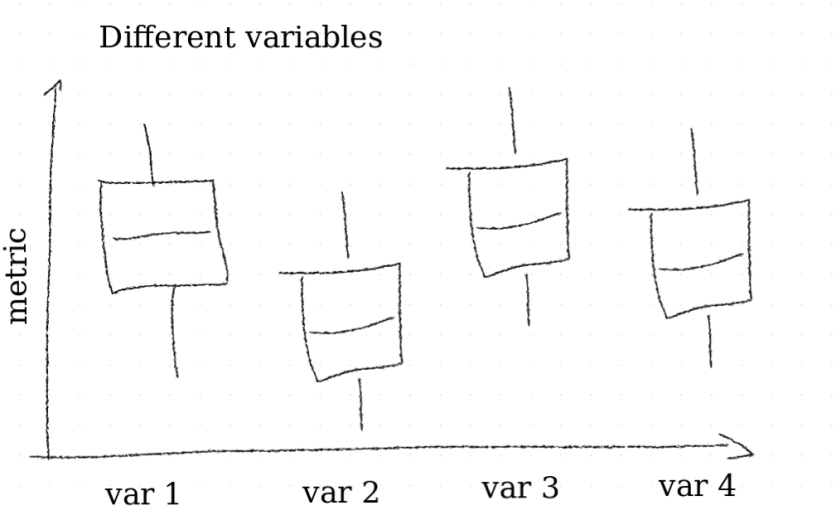
- a few sample variables (max 3/4)
- a representative gap length
- manually choose interesting gaps (2-4)
- assume that all the other variables are there




**timeseries**

## 3.2 Different variables

- do one plot for each variable
- manually choose interesting gaps (2-4)
- a representative gap length
- assume that all the other variables are there




**Box plot metric**

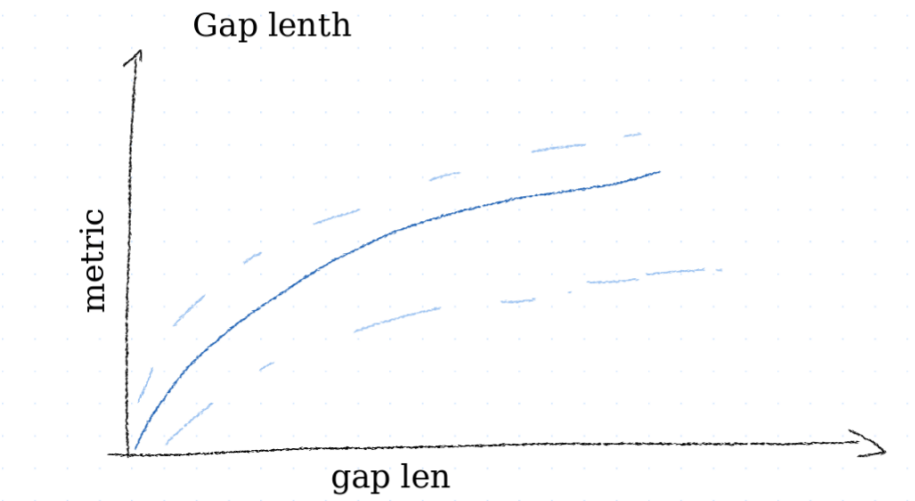
**Table** Metrics for

- each variable
- different gap length?
- other variables missing

**Scatter plot ?**

## 3.3 Gap length

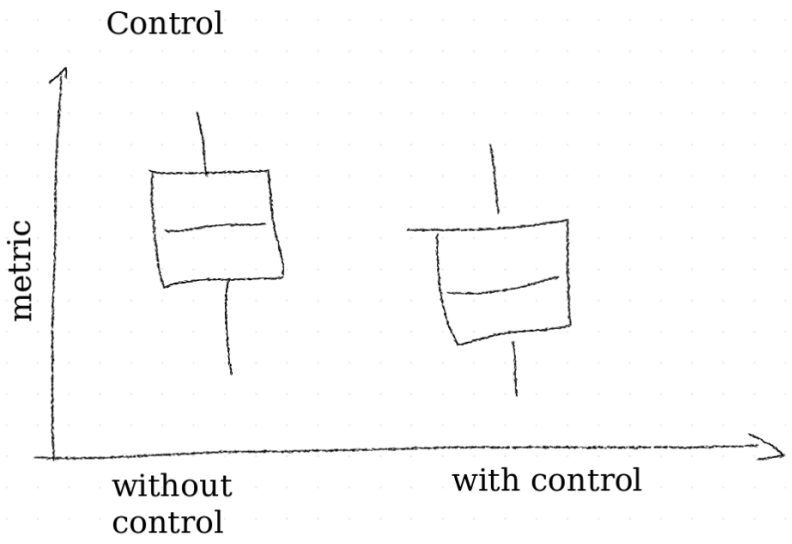
- take many gaps
- average metric between all variables
- assume that all the other variables are present




### 3.4 Control

Goal: show how the use of the control (ERA-5 data) in the gap is impacting the predictions

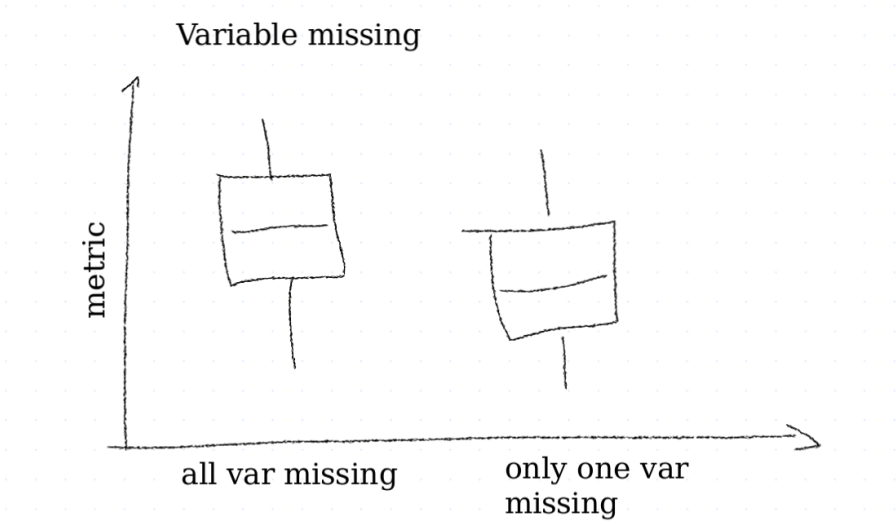
- take many gaps
- average metric between all variables
- assume that all the other variables are present




**boxplot**

## 3.5 Other variables missing

Goal: show how the presence/absence of other variables in the gap is impacting the predictions
- all variables - no variables - mean n variables present




**boxplot**

## 3.6 Comparison state of art

boxplot metric vs gap length for MDS/ERA

## 3.7 Numerical Stability

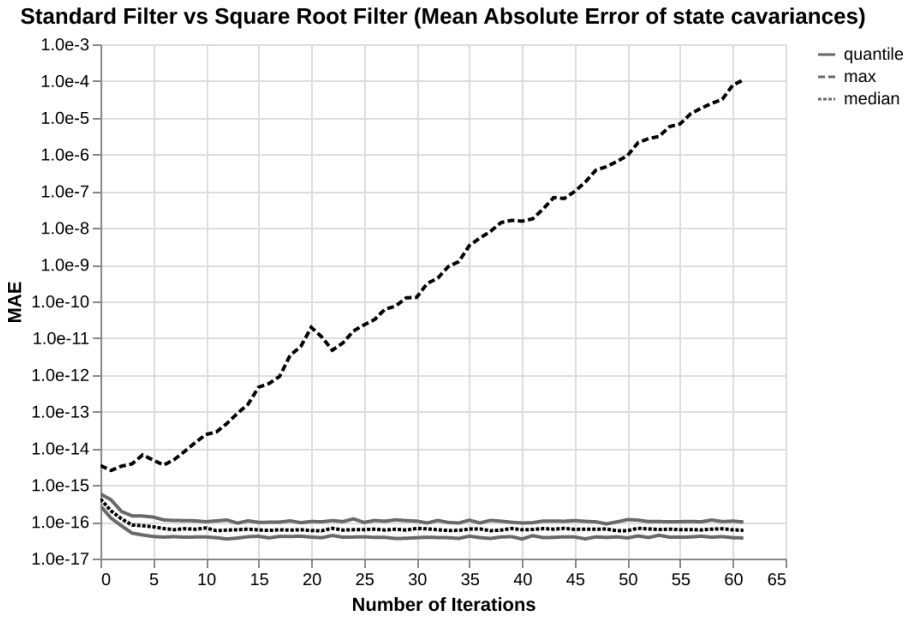



Figure 1: Comparison of standard Kalman Filter implementation and Square Root Filter. For 100 times the filter has been initialized with random parameters (drawn from a uniform distribution range 0-1) and then filtered 62 observations. At each filter iteration calculated the Mean Absolute Error (MAE) between the state covariance from the standard filter and the square root filter. The plot shows the median, 1 and 3 quartile and the maximum of the MAE across the 100 samples. You can see that usually the two filter implementations agree, but for a few parameters combinations the error is growing with the number of iterations of the filter suggesting a numerical stability issue. After 62 iterations the standard filter crashes because the covariance is not positive definite anymore. The initial bigger error can be explained by the slightly different initial state covariance.

# 4 Discussion

## 4.1 Importance and use of better gap filling

### 4.1.1 Variable comparison

difference between variables

## 4.2 Limitations Kalman Filters

- assumptions of the filter
- long gaps
- shift of control variable
- training to different conditions

## 4.3 Model deployment

what is missing to actually use the model in production

- Find optimal context before/after filter
- train model for different sites / fine tuning
- use ERA-5 Land data

## 4.4 Gap filling quality assessment

- realistic gaps properties (length/other variables missing/time of day)
- importance of metrics

## 4.5 Improvements

# 5 Conclusions

# References

# References

[1] Jason Beringer et al. "Technical Note: Dynamic INtegrated Gap-filling and Partitioning for OzFlux (DINGO)". In: *Biogeosciences* 14.6 (Mar. 23, 2017), pp. 1457–1460. ISSN: 1726-4170. DOI: 10.5194/bg-14-1457-2017. URL: https://bg.copernicus.org/articles/14/1457/2017/ (visited on 05/12/2022).

[2] Gerald J. Bierman and Catherine L. Thornton. "Numerical Comparison of Kalman Filter Algorithms: Orbit Determination Case Study". In: *Automatica* 13.1 (Jan. 1, 1977), pp. 23–35. ISSN: 0005-1098. DOI: 10.1016/0005-1098(77)90006-1. URL: https://www.sciencedirect.com/science/article/pii/0005109877900061 (visited on 01/12/2023).

[3] Dan Simon. *Optimal State Estimation Kalman, H and Nonlinear Approaches*. 2006. ISBN: 100-47 1-70858-5.

[4] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. JHU Press, 2013. 781 pp. ISBN: 978-1-4214-0794-4. Google Books: X5YfsuCWpxMC.

[5] K. Kramer et al. "Evaluation of Six Process-Based Forest Growth Models Using Eddy-Covariance Measurements of CO2 and H2O Fluxes at Six Forest Sites in Europe". In: *Global Change Biology* 8.3 (2002), pp. 213–230. ISSN: 1365-2486. DOI: 10.1046/j.1365-2486.2002.00471.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1046/j.1365-2486.2002.00471.x (visited on 01/18/2023).

[6] Mohinder S. Grewal and Angus P. Andrews. *Kalman Filtering: Theory and Practice Using MATLAB, Second Edition*. 2001. ISBN: 0-471-26638-8.

[7] JAMES POTTER and ROBERT STERN. "STATISTICAL FILTERING OF SPACE NAVIGATION MEASUREMENTS". In: *Guidance and Control Conference*. American Institute of Aeronautics and Astronautics, 1963. DOI: 10.2514/6.1963-333. URL: https://arc.aiaa.org/doi/abs/10.2514/6.1963-333 (visited on 01/13/2023).

[8] Thomas Wutzler et al. "Basic and Extensible Post-Processing of Eddy Covariance Flux Data with REddyProc". In: *Biogeosciences* 15.16 (Aug. 23, 2018), pp. 5015–5030. ISSN: 1726-4170. DOI: 10.5194/bg-15-5015-2018. URL: https://bg.copernicus.org/articles/15/5015/2018/ (visited on 05/15/2022).

# Appendix

## 5.1 Proof for eigenvalues of $CC^T$

The eigenvalues of the transpose of a matrix are the same of the eigenvalues of a matrix
The properties of the eigenvalues is that $Ax = \lambda x$ for any vector $x$
Then $CC^T x = C(\lambda x) = \lambda C x = \lambda^2 x$

## 5.2 UD Filter

This is not relevant anymore, keeping this for now
The UD filter ([2]) is named after the $UDU^T$ decomposition[1], which is also

[1] In the literature $U$ is a lower **unit** triangular matrix, but the PyTorch routine just makes lower triangular matrices. In my understanding this makes no difference in the derivation of the equations

known as the $LDL^T$ decomposition ([4]) and it always exists for a positive definite matrix. Where $U$ is a lower triangular matrix and $D$ is a diagonal matrix.
The UD Filter never computes the state covariance matrix $P$, but propagates the $U$ and $D$ components of the matrix to improve the numerical stability. Hence, the filter equations need to be rewritten by using only the $U$ and $D$ components and never the $P$ matrix

**Measurement update** After each observation at time $t$ the state covariance is updated according to equation **??**, which is here repeated (for notation simplicity the time subscripts are removed):

$$P = P^- - P^- H^T (HP^- H^T + R)^{-1} HP^-$$

The goal is to obtain the $U$ and $D$ factors of $P$ given the $U^-$ and $D^-$ factors of $P^-$.
Letting:

- $P = UDU^T$
- $P^- = U^- D^- U^{-T}$, with $U^{-T}$ being the transpose of $U^-$ not the transpose of the inverse of $U$
- $v = U^{-T} H^T$

then

$$UDU^T = \tag{33}$$
$$= U^- D^- U^{-T} - U^- D^- U^{-T} H^T \left(HU^- D^- U^{-T} H^T + R\right)^{-1} HU^- D^- U^{-T} \tag{34}$$
$$= U^- \left[D^- - D^- v (v^T D^- v + R)^{-1} v^T D^-\right] U^{-T} \tag{35}$$

If you do a UD decomposition of $\left[D^- - D^- v (v^T D^- v + R)^{-1} v^T D^-\right] = BDB^T$, where $B$ is a lower triangular matrix and $D$ is a diagonal matrix, then

$$UDU^T = U^- (BDB^T) U^{-T} = (U^- B) D (U^- B)^T)$$

$D$ is the diagonal factor of $P$ as and $U$ is equal to $U^- B$, since $U^- B$ is a lower triangular matrix as is the products of two lower triangular matrices.
Therefore, to perform the measurement update step it is necessary to compute the $UDU^T$ decomposition of $\left[D^- - D^- v (v^T D^- v + R)^{-1} v^T D^-\right]$, which can be implemented using the `torch.linalg.ldl_factor` function.

**Time update** The state covariance at time $t$ is obtained from the state at time $t-1$ according to the equation 3, which is here repeated:

$$P_t = AP_{t-1} A^T + Q$$

The goal is to obtain the $U_t$ and $D_t$ factors of $P_t$ given the $U_{t-1}$ and $D_{t-1}$ factors of $P_{t-1}$.
Letting:

$$Q = GD_QG^T \tag{36}$$
$$W = \begin{bmatrix} AU_{t-1} & G \end{bmatrix} \tag{37}$$
$$D_w = \begin{bmatrix} D_{t-1} & 0 \\ 0 & D_Q \end{bmatrix} \tag{38}$$

where $D_Q$ is diagonal and $G$ is lower triangular (the UD decomposition of Q), then:

$$\begin{aligned} P &= WD_wW^T = \\ &= \begin{bmatrix} AU_{t-1} & G \end{bmatrix} \begin{bmatrix} D_{t-1} & 0 \\ 0 & D_Q \end{bmatrix} \begin{bmatrix} U_{t-1}^T A^T \\ G^T \end{bmatrix} \\ &= AU_{t-1}D_{t-1}U_{t-1}^T A^T + GD_QG^T \end{aligned} \tag{39}$$

Then if we can find decompose $W$ as matrices $U_tV$ where $U_t$ is a lower triangular matrix such as that $VD_wV^T$ is a diagonal matrix then

$$P_t = (U_tV)D_w(U_tV)^T = U_tD_tU_t \tag{40}$$

so we have the $U_t$ and $D_t$ factors of $P_t$
This decomposition can be efficiently implemented in PyTorch using some modifications of the QR decomposition [2]

***PyTorch Implementation*** The QR decomposition in PyTorch decompose a matrix $A = QR$, where $Q$ is an orthogonal matrix and $R$ is an upper triangular matrix. However, for the filter there are two changes that needs to be made:

1. decompose into the product of the lower triangular matrix and a diagonal matrix $A = LQ$
2. have a weighted decomposition regarding a matrix $D$, such as that $QDQ^T = I$ instead of $QQ^T = I$

*1) Lower triangular matrix* To obtain a lower triangular matrix from the $QR$ decomposition, you can compute the QR decomposition of $A^T = QR$ and then $A = R^TQ^T$ where $R^T$ is a lower triangular matrix and $Q^T$ is still an orthogonal matrix.

[2] https://pytorch.org/docs/stable/generated/torch.linalg.qr.html

*2) Weigthed decomposition* The QR decomposition can be used to have a weighted decomposition [3], where $QDQ^T = I$ instead of the $QQ^T = I$. If $D$ is positive definite, it is possible to apply the Cholesky decomposition of $D = CC^T$. Then you can compute the QR factorization of $CA = UR$, where $U$ is an orthogonal matrix and $R$ is an upper triangular matrix. Then can define $Q = C^{-1}U$. This can be proved as $QDQ^T = C^{-1}UCC^TU^T(C^{-1})^T = I$ In the filter context $D$ is a diagonal matrix computing so the Cholesky decomposition can be efficiently done by taking the square root of every element.

*Summary* $W$ can be decomposed as $W = U_t V$ with the following procedure:

$$(C_{D_W} W)^T = QR$$

using `torch.linalg.qr` and then defining $U_t = R^T$ and $V = C_{D_W}^{-1} Q^T$

[3] https://scicomp.stackexchange.com/a/33436