Rapport

Filtre de Kalman

Filtre de Kalman et Kalman étendu

Première implémentation d'un filtre de Kalman
avec une extention à une application non liénaire

Elève: FARHAT Charles

Cours: 4201

Inférence bayésienne dans des modèles markoviens

décembre 2022

Contents

1 Introduction **1**

2 Filtre de Kalman classique **2**
2.1 Présentation théorique du filtre de Kalman 2
2.2 Implémentation 5

3 Filtre de Kalman étendu **13**
3.1 Introduction 13
3.2 Formulation du problème 13
3.3 Implémentation 15
3.4 Résultats 17

4 Conclusion **18**

A Appendices **19**

1. Introduction

Le filtre de Kalman a été inventé par Rudolf Emil Kálmán en 1960. M. Kálmán était un ingénieur et mathématicien hongrois qui a travaillé sur de nombreux problèmes de filtrage et de contrôle de qualité dans les domaines de l'aérospatiale et de la robotique. Sa théorie des filtres de Kalman est considérée comme l'une des contributions les plus importantes de l'ingénierie moderne et est largement utilisée dans de nombreux domaines pour améliorer la précision de l'estimation des états de systèmes dynamiques.

Le filtre de Kalman est un algorithme utilisé pour estimer la valeur d'un système dynamique à partir de mesures soumises à un bruit et incomplètes. Il est souvent utilisé dans les domaines de l'ingénierie, de l'aérospatiale et de la robotique pour améliorer la précision de l'estimation d'un état à un moment donné, en utilisant des mesures précédentes et en prenant en compte le bruit et les erreurs de mesure.

Le filtre de Kalman est basé sur l'hypothèse selon laquelle le système est modélisé par un processus de Markov caché, ce qui signifie que l'état du système à un moment donné ne dépend que de l'état précédent et non des états antérieurs. Le filtre de Kalman utilise des équations de transition de l'état et de mesure pour mettre à jour l'estimation de l'état du système en fonction de nouvelles mesures et de la dynamique du système.

Il existe deux variantes du filtre de Kalman : le filtre de Kalman linéaire et le filtre de Kalman non linéaire. Le filtre de Kalman linéaire est utilisé lorsque le système peut être modélisé par des équations linéaires, tandis que le filtre de Kalman non linéaire est utilisé lorsque le système fait alors appel à des fonctions non linéaires.

Nous commencerons par une étude approfondie du filtre de Kalman dans sa version Linéaire, puis nous étudierons un cas spécifique du filtre de Kalman étendu.

2. Filtre de Kalman classique

2.1 Présentation théorique du filtre de Kalman

On pose le modèle d'évolution suivant :

$$\begin{cases} \mathbf{X}_k &= \mathbf{F}\mathbf{X}_{k-1} + \mathbf{U}_k \\ \mathbf{Y}_k &= \mathbf{H}\mathbf{X}_k + \mathbf{V}_k \end{cases} \tag{1}$$

avec dans ce modèle :

$$\mathbf{X}_0 \sim \mathcal{N}(\mathbf{m}_{0|0},\ \mathbf{P}_{0|0}) \tag{2}$$

et,

$$\begin{cases} \mathbf{U}_k \sim \mathcal{N}(\mathbf{0},\ \mathbf{Q}) \\ \mathbf{V}_k \sim \mathcal{N}(\mathbf{0},\ \mathbf{R}) \end{cases} \tag{3}$$

La donnée probabiliste vient de U et V (ce qui implique que X et Y v.a. aussi) mais F, H sont déterministes !
On se place ici dans le cadre *linéaire* et *gaussien* les solutions sont donc explicites.

D'après 3 et 1 on peut écrire:

$$\begin{aligned} p(\mathbf{X}_k|\mathbf{X}_{k-1} = x_{k-1}) &\sim \mathcal{N}(\mathbf{F}x_{k-1},\ \mathbf{Q}) \\ p(\mathbf{Y}_k|\mathbf{X}_k = x_k) &\sim \mathcal{N}(\mathbf{H}x_k,\ \mathbf{R}) \end{aligned} \tag{4}$$

Nous cherchons la distribution de $\mathbb{E}(\mathbf{X}_k|\mathbf{Y}_{0:k})$ (le meilleur estimateur au sens de l'erreur quadratique moyenne) et, comme le format des équations le fait penser, de façon récursive.
De plus si la distribution de $p(x_k|y_{0:k})$ est normale il suffit de connaitre sa moyenne $\mathbf{m}_{k|k}$ pour avoir $\mathbb{E}(\mathbf{X}_k|\mathbf{Y}_{0:k})$.

Nous allons donc montrer que cette distribution est bien normale et qu'il s'agit en réalité d'un problème de propagation des paramètres dans des lois normales puisque que le problème est liée par une relation de récursivité.

On va donc faire l'hypothèse de récurrence suivante : $p(x_{k-1}|y_{0:k-1}) \sim \mathcal{N}(\mu_{k-1|k-1},\ P_{k-1|k-1})$

Dans le cadre des développements, nous allons avoir besoin de la propriété suivante :

$$p(y_k|x_k, y_{0:k-1}) = p(y_k|x_k) \tag{5}$$

Nous en faisons une démonstration en annexes :

Par ailleurs Bayes nous donne :

$$
\begin{aligned}
p(x_k|y_{0:k}) &= \frac{p(x_k, y_k, y_{0:k-1})}{p(y_{0:k})} \\
&= \frac{p(y_k|x_k, y_{0:k-1})p(x_k, y_{0:k-1})}{\int p(x_k, y_{0:k})dx_k} \\
&= \frac{p(y_k|x_k, y_{0:k-1})p(x_k|y_{0:k-1})p(y_{0:k-1})}{\int p(x_k, y_{0:k})dx_k} \\
&= \frac{p(y_k|x_k, y_{0:k-1})p(x_k|y_{0:k-1})\cancel{p(y_{0:k-1})}}{\int p(y_k|x_k, y_{0:k-1})p(x_k|y_{0:k-1})\cancel{p(y_{0:k-1})}dx_k}
\end{aligned}
\tag{6}
$$

Et d'après 5 on a finalement,

$$
p(x_k|y_{0:k}) = \frac{p(y_k|x_k) \overbrace{p(x_k|y_{0:k-1})}^{terme\ de\ prediction}}{\int p(y_k|x_k)p(x_k|y_{0:k-1})dx_k} \tag{7}
$$

On va procéder par étape et décortiquer ce que cette relation nous permet d'écrire.

On va tout d'abord à exprimer le terme de prédiction comme une loi normale dont on connaitrait les paramètres :

$$
\begin{aligned}
p(x_k|y_{0:k-1}) &= \int p(x_k, x_{k-1}|y_{0:k-1})dx_{k-1} \\
&= \int p(x_k|x_{k-1})p(x_{k-1}|y_{0:k-1})dx_{k-1} \\
&= \int \underbrace{\mathcal{N}(\mathbf{F}x_{k-1},\ \mathbf{Q})}_{p(x_k|x_{k-1})} \underbrace{\mathcal{N}(\mu_{k-1|k-1},\ \mathbf{P}_{k-1|k-1})}_{Hypothese\ de\ recurence} dx_{k-1} \\
&= \mathcal{N}\left(\mathbf{F}\mu_{k-1|k-1},\ \mathbf{Q} + \mathbf{F}\mathbf{P}_{k-1|k-1}\mathbf{F}^T\right)
\end{aligned}
\tag{8}
$$

C'est donc notre étape de prédiction, on va passer de $\mathbf{X}_{k-1}$à $\mathbf{X}_k$ (prédiction) à partir des données $\mathbf{Y}_{0:k-1}$. Cette étape se traduit donc dans la propagation des paramètres d'une loi normale avec la relation de récurrence :

$$
\begin{cases}
\mu_{k|k-1} &= \mathbf{F}\mu_{k-1|k-1} \\
\mathbf{P}_{k|k-1} &= \mathbf{Q} + \mathbf{F}\mathbf{P}_{k-1|k-1}\mathbf{F}^T
\end{cases}
\tag{9}
$$

L'étape de prédiction faite on cherche maintenant à intégrer la nouvelle observation, c'est à dire calculer notre terme : $p(x_k|y_{0:k})$.

On va donc réaliser l'étape de filtrage (ou étape d'update) :

Nous auront pour cela, besoin d'un résultat :

Propriété 1

$$\mathcal{N}(y|\mathbf{H}x,\ \mathbf{R})\mathcal{N}(x|\mathbf{m},\ \mathbf{P})\ =\ q(y)\mathcal{N}(x|\hat{\mathbf{m}},\ \hat{\mathbf{P}})$$

Avec,

$$\begin{aligned}\hat{\mathbf{m}}\ &=\ \mathbf{m}\ +\ \mathbf{K}(y-\mathbf{H}\mathbf{m})\\ \hat{\mathbf{P}} &= (\mathbf{I}-\mathbf{K}\mathbf{H})\mathbf{P}\\ \mathbf{K} &= \mathbf{P}\mathbf{H}^T\left(\mathbf{H}\mathbf{P}\mathbf{H}^T+\mathbf{R}\right)^{-1}\end{aligned}$$

Cette propriété et 4 nous permet d'écrire:

$$\begin{aligned}p(x_k|y_{0:k}) &= \frac{p(y_k|x_k)p(x_k|y_{0:k-1})}{\int p(y_k|x_k)p(x_k|y_{0:k-1})dx_k}\\ &\propto \frac{g(y_k)\mathcal{N}(x_k|\mu_{k|k},\ \mathbf{P}_{k|k})}{\int g(y_k)\mathcal{N}(x_k|\mu_{k|k},\ \mathbf{P}_{k|k})dx_k}\end{aligned} \tag{10}$$

or ici,

$$\frac{g(y_k)\mathcal{N}(x_k|\mu_{k|k},\ \mathbf{P}_{k|k})}{\int g(y_k)\mathcal{N}(x_k|\mu_{k|k},\ \mathbf{P}_{k|k})dx_k} = \frac{\cancel{g(y_k)}\mathcal{N}(x_k|\mu_{k|k},\ \mathbf{P}_{k|k})}{\cancel{g(y_k)}\int \mathcal{N}(x_k|\mu_{k|k},\ \mathbf{P}_{k|k})dx_k} \tag{11}$$

Et, comme on marginalise par rapport à x_k, $\int \mathcal{N}(x_k|\mu_{k|k},\ \mathbf{P}_{k|k})dx_k = 1$ on obtient donc finalement :

$$p(x_k|y_{0:k})\ \propto \mathcal{N}(x_k|\mu_{k|k},\ \mathbf{P}_{k|k})$$

avec :

$$\begin{cases}\mu_{k|k} = \mu_{k|k-1}\ +\ \mathbf{K}(y_k-\mathbf{H}\mu_{k|k-1})\\ \mathbf{P}_{k|k} = (\mathbf{I}-\mathbf{K}\mathbf{H})\mathbf{P}_{k|k-1}\\ \mathbf{K} = \mathbf{P}_{\mathbf{k|k-1}}\mathbf{H}^T\left(\mathbf{H}\mathbf{P}_{\mathbf{k|k-1}}\mathbf{H}^T+\mathbf{R}\right)^{-1}\end{cases} \tag{12}$$

On peut résumer nos différentes étapes par ce schémas:

$$p(x_n|y_{0:n}) \xrightarrow{\text{Prédiction}} p(x_{n+1}|y_{0:n}) \xrightarrow{\text{Update}} p(x_{n+1}|y_{0:n+1})$$

Nous allons donc maintenant nous atteler à l'implémentation d'un filtre de Kalman simple.

2.2 Implémentation

Une expression générale intégrale possible de la position est la suivante: $x(t) = x(t') + \int_{t'}^{t} v(\tau)d\tau$. En discrétisant $\frac{dx(t)}{dt} = v(t)$ on obtient $x_{n+1} \approx x_n + v \times dt$.

Avec une démarche similaire pour les ordonées et en supposant une vitesse constante nous obtenons:

$$\mathbf{X}_{n+1} = \begin{bmatrix} 1 & dt & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & dt \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \mathbf{X}_n \tag{13}$$

Rappel, ici : $X = [x,\ \dot{x}, y, \dot{y}]^T$

De même, en supposant que les mesures contiennent la "vraie" position plus un bruit d'erreur:

$$\mathbf{Y}_{n+1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \mathbf{X}_{n+1} + v_{n+1} \tag{14}$$

En ajoutant dans la première équation un bruit représentant la confiance en ce modèle (vitesse constante) nous obtenons les mêmes expressions que celles énoncées dans la partie précédente. En supposant que le vecteur d'état initial est gaussien et que les bruits de ce modèle sont gaussiens on peut alors appliquer les résutats du Filtre de Kalman !

On remarque ici que H et F sont constantes (ne dépendent pas je n) nous sommes dans le cas simplifié de chaînes homogènes !

Nous allons procéder en 3 étapes pour l'implémentation du filtre de Kalman sur notre exemple de TP.
Dans une première partie nous présenterons notre initialisation avec la création de la trajectoire théorique, des observations et des différents paramètres nécessaires et les conséquences de leur modification.
Dans une seconde partie, nous verrons comment implémenter le cœur du filtre et quelques conditions à respecter.
Finalement nous verrons l'implémentation d'une métrique d'erreur et des résultats finaux !

2.2.1 Initialisation

On va créer nos observations et notre trajectoire à partir du modèle décrit ci-dessus, on va donc écrire le code python suivant :

```python
def creer_trajectoire(F, Q, x_init, T):
    vecteur_x = np.zeros((4, T))
    vecteur_x[:, 0] = x_init
    for i in range(1, len(vecteur_x[0])):
      vecteur_x[:, i] = F@vecteur_x[:, i-1] + \
      np.random.multivariate_normal(np.zeros((4)), Q)
    return vecteur_x

def creer_observations(H, R, vecteur_x, T):
    vecteur_y = np.zeros((2, T))
    for i in range(0, len(vecteur_y[0])):
      vecteur_y[:, i] = H@vecteur_x[:, i] + \
      np.random.multivariate_normal(np.zeros(2), R)
    return vecteur_y
```

Code extraction 1: Initialisation du filter

Avec les matrices $\mathbf{Q}, \mathbf{F}, \mathbf{H}, \mathbf{R}, \mathbf{T}$ données dans l'énoncé et rappelées plus haut.

On obtient alors les exemples de trajectoires et d'observations suivantes:

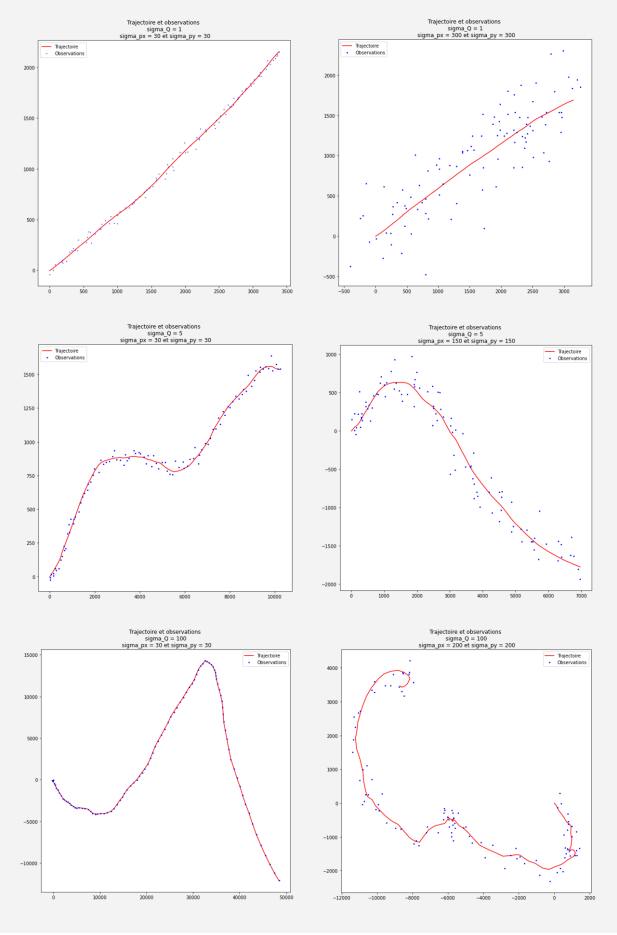


Figure 1: exemple de trajectoire et observations simulées

Lorsque le bruit du modèle est faible (fig (1, 1), (1, 2)) on observe une trajectoire relativement proche d'une droite (ce qui est cohérent vis à vis d'une vitesse constante). En revanche la trajectoire réelle devient rapidement et fortement non linéaire lorsque $sigma_Q$ augment (2e et 3e ligne).
De manière générale l'importance de la dispersion des observations autour de la vraie courbe corrèle bien avec l'augmentation du bruit de mesure et la complexité de la trajectoire semble bien répondre à une augmentation de $sigma_Q$ (forte variation de la vitesse réelle).
On peut donc considérer notre problème initié, il ne reste plus qu'a faire l'implémentation du filtre à proprement parlé.
Avant de passer à son implémentation en code il nous faut considérer les conditions initiales suivantes :

$$\begin{cases} Acceleration \ : \ a_0 = 0 \\ Vitesse : \ v_0^x, \ v_0^y \ = \ [40, \ 20] \\ Position \ : \ [x_0, y_0] \ = \ [3, \ -4] \end{cases}$$

On va par ailleurs initialiser Q et R comme donnée dans l'énoncé (voir code...), de même pour $\mathbf{X}_0$: on va choisir une matrice de covariance $\mathbf{P}_{0|0}$ "faible" si la position de départ est bien connue et inversement si grande incertitude.
On a alors le code python suivant:

```
T_e = 1
T = 100
sigma_Q = 1
sigma_px = 30
sigma_py = 30

F = np.eye(4)
F[0, 1], F[2, 3] = T_e, T_e
Q = np.array([
[T_e**3/3, T_e**2/2, 0, 0],
[T_e**2/2, T_e, 0, 0],
[0, 0, T_e**3/3, T_e**2/2],
[0, 0, T_e**2/2, T_e]], dtype='float64'
)*sigma_Q**2

H = np.zeros((2, 4))
H[0, 0], H[1, 2] = 1, 1
R = np.diag([sigma_px**2, sigma_py**2])

x_init = np.array([3, 40, -4, 20])

x_kalm = x_init  # x^_0|0
P_kalm = np.eye(4)  # P_0|0
```

Code extraction 2: Fonction de calcul du filtre de kalman à l'instant k

On ne mets ici que des extraits du code, le code complet est disponible en annexe.

2.2.2 Implémentation du filtre de Kalman simple

Nous allons donc reprendre les équations décrites en 12 et les implémenter en python, on obtient alors:

```
    def filtre_de_kalman(F, Q, H, R, y_k, x_kalm_prec, P_kalm_prec):

    # prediction :
    m_prediction = F@x_kalm_prec
    P_prediction = Q + F@P_kalm_prec@F.T

    # update
    K = P_prediction@H.T@np.linalg.inv(H@P_prediction@H.T + R)
    # Mise a jour des vecteurs
    x_kalm_k = m_prediction + K@(y_k - H@m_prediction)
    P_kalm_k = (np.eye(len(P_kalm_prec[0])) - K@H)@P_prediction

    # Return a l'etape k des vecteurs
    return [x_kalm_k, P_kalm_k]
```

Code extraction 3: Fonction de calcul du filtre de kalman à l'instant k

Cette fonction nous permet de faire nos étape d'update et de prédiction récursivement sur les données qui arrivent, on peut alors écrire le programme dans sa globalité qui va utiliser les observations générées et garder en mémoire nos prédictions pour un affichage ultérieur :

```
vecteur_x = creer_trajectoire(F, Q, x_init, T)
vecteur_y = creer_observations(H, R, vecteur_x, T)

# Initialisation des differentes variables
# ....

# Estimation des etats
for i in range(1,T):
x_est[:,i],  P_est[:, :,i] = filtre_de_kalman(F, Q, H, R, vecteur_y
 [:,i], x_est[:,i-1], P_est[:, :,i-1])
```

Code extraction 4: Fonction principale filtre de kalman simple

Ici les codes sont allégés pour une meilleure compréhension des étapes clés. Le code complet est disponible en annexe A (projet github)

On se muni également d'une fonction nous permettant de calculer l'erreur quadratique entre notre estimation et nos valeurs réelles sur l'ensemble de nos prédictions (voir source code) : *erreur_quad(x_est, vecteur_x)*

2.2.3 Les résultats

Une fois notre implémentation réalisé, nous allons jouer sur les paramètre du modèle, ici :

- σ_Q nous donne la confiance du modèle : nous allons le faire varier de 1 (grande confiance) à 10 (confiance très faible)
- σ_{px} et σ_{py} représentent le bruit d'observation (nos erreurs d'observation) que nous allons faire varier de 10 (faible bruit) à 300 (très fort bruit)

  On obtient comme courbe de référence (confiance forte et faible bruit) :

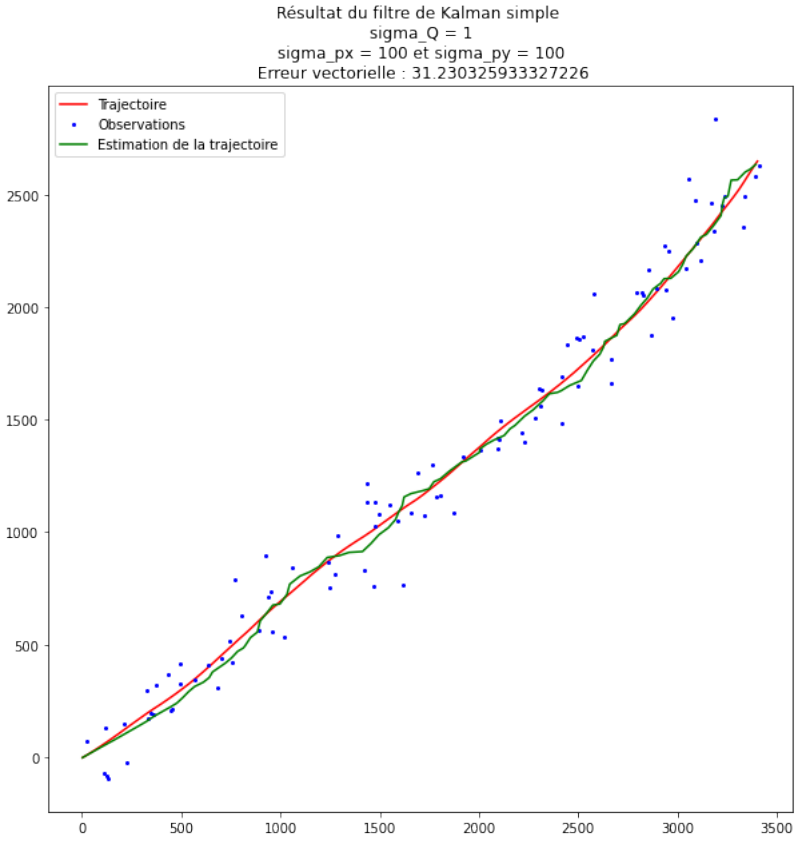


Figure 2: Résultat du filtre de Kalman Simple, courbe en vert : notre estimation

Si on réduit notre confiance sur le modèle (on augmente Q) on obtient alors :

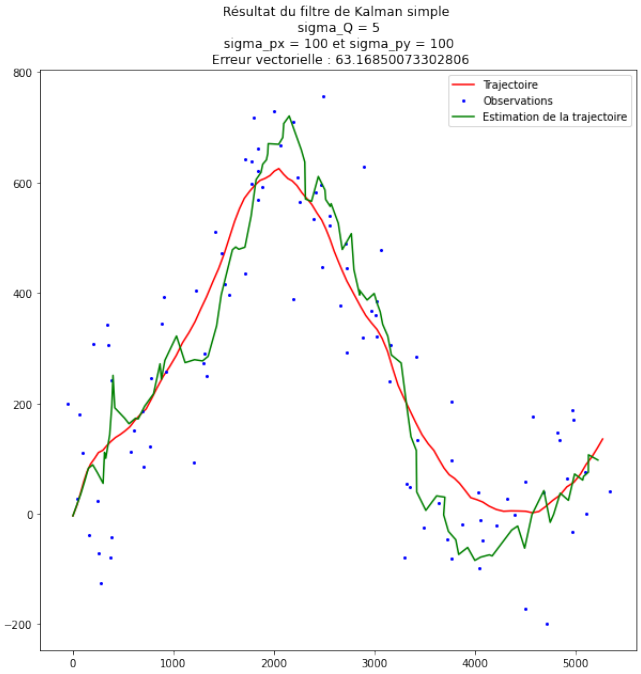


Figure 3: Résultat du filtre de Kalman Simple, courbe en vert : notre estimation

Puis pour une version dégradé ($\sigma_Q = 10$),

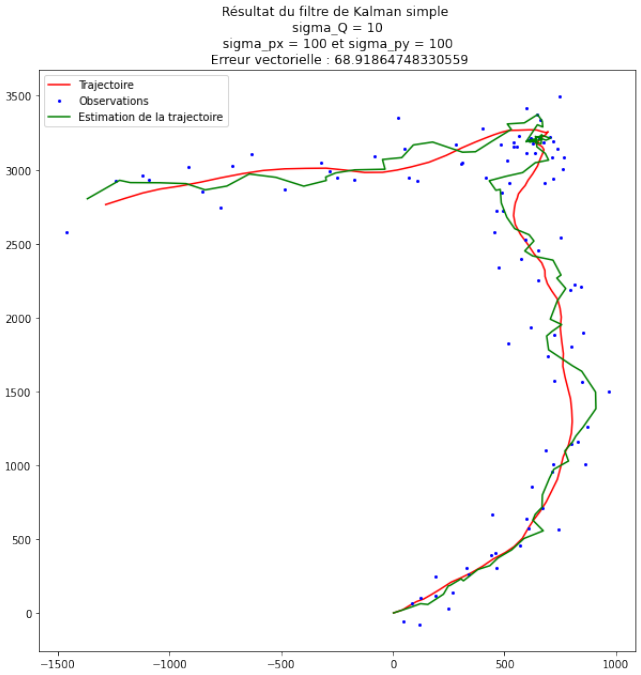


Figure 4: Résultat du filtre de Kalman Simple, courbe en vert : notre estimation

On continue l'expérience avec les autres valeurs comme annoncé plus haut.

On obtient alors de la même façon :

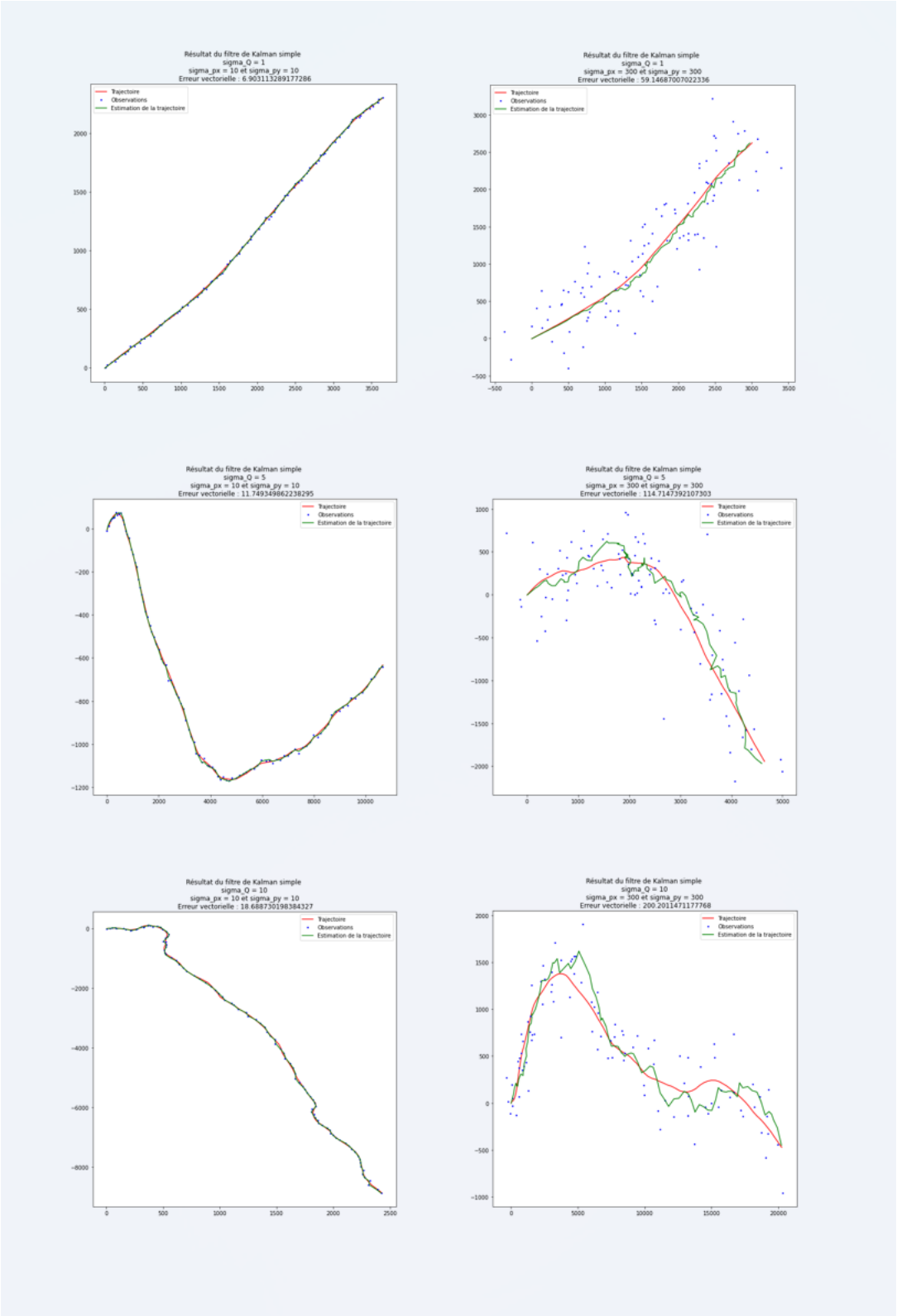


Figure 5: Résultats du filtre de Kalman Simple, avec différentes valeurs de bruit

Nous analysons les résultats sur la prochaine page...

Il semblerait que pour des erreurs de mesures (σ_{px} et σ_{py}) faibles le filtre reste capable de suivre le mouvement en dépit de variations importantes de σ_Q. Cependant nous observons dors et déjà les limites du Filtre de Kalman : si dans les cas de grande confiance dans notre modèle (fig (1,1), 2) l'approximation est bonne malgré des bruits de mesure qui peuvent être conséquents, il ne l'est absolument plus dès que les deux bruits augmentent simultanément. En effet lorsqu'une information est mauvaise le filtre à tendance à s'appuyer sur l'autre pour prédire, mais si les deux sont imprécises les erreurs se cumulent et l'algorithme devient rapidement peu performant (fig (3, 2)).

3. Filtre de Kalman étendu

3.1 Introduction

Certains phénomènes de suivi n'admettent pas une modélisation linéaire de la forme :

$$\begin{cases} \mathbf{X}_k &= \mathbf{F}\mathbf{X}_{k-1} + \mathbf{U}_k \\ \mathbf{Y}_k &= \mathbf{H}\mathbf{X}_k + \mathbf{V}_k \end{cases} \tag{15}$$

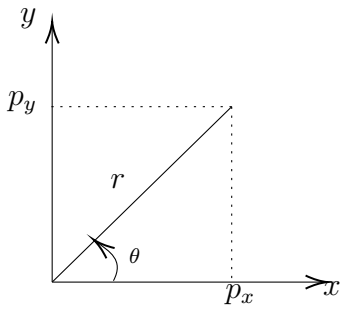
mais une forme plus générale qui peut être exprimée comme :

$$\begin{cases} \mathbf{X}_k &= f(\mathbf{X}_{k-1}) \\ \mathbf{Y}_k &= g(\mathbf{X}_k) \end{cases} \tag{16}$$

avec f et g de $\Re^d$ dans $\Re^d$.
Or si ces fonctions présentent des non-linéarités les résultats explicites du filtre (exprimés partie 1 : 12) ne sont plus valables. Pour palier ce problème une solution est de changer d'algorithme, d'en choisir un plus contraignant en terme de puissance de calcul mais capable de suivre ce type de phénomène. Nous pourrions par exemple nommer celui du filtrage particulaire. Une autre est d'essayer de se ramener à une situation linéaire où nous pourrons à nouveau utiliser les résultats du filtre. C'est l'approche que propose de réaliser cette partie

3.2 Formulation du problème

Dans notre cas nous considérons le passage en coordonnées sphérique pour les observation. On prendra dans la suite ce référentiel :



On va donc représenter nos nouvelles variables, en commencent par notre variable d'état qui ne change pas :

$$\mathbf{X} = \begin{bmatrix} p_x \\ v_x \\ p_y \\ v_y \end{bmatrix} \tag{17}$$

Puis notre variable d'observation qui passe en polaire :

$$\mathbf{Y} = \begin{bmatrix} p_x \\ p_y \end{bmatrix} \xrightarrow{Polaire} \begin{bmatrix} \theta \\ r \end{bmatrix} \tag{18}$$

On peut alors faire le changement de référentiel à l'aide de :

$$\begin{cases} \theta(p_x, p_y) = \arctan\left(\frac{p_y}{p_x}\right) \\ \\ r(p_x, p_y) = \sqrt{p_x^2 + p_y^2} \end{cases} \tag{19}$$

Ce qui nous donne pour la variable d'observation :

$$\begin{aligned} \mathbf{Y}_k &= \begin{bmatrix} \theta(\mathbf{X}_k) \\ r(\mathbf{X}_k) \end{bmatrix} + V_k \\ &= g(\mathbf{X}_k) \end{aligned} \tag{20}$$

avec $v_k \sim \mathcal{N}(0, \mathbf{R})$

Attention, ici g est **non linéaire** (par le arctan et la racine)

On cherche donc à linéariser g, on dispose pour cela de Taylor-Lagrange qui nous dit : Pour toute fonction de $\mathcal{C}^1$ de $\Re^d$ dans $\Re^m$ on peut écrire :

$$f(x) \approx f(x_0) + J(f)(x_0)(x - x_0) \tag{21}$$

avec J la jacobienne de f.

Un rapide calcul nous donne dans notre cas :

$$\mathbf{J}_g = \left.\frac{\partial g}{\partial \mathbf{X}}\right|_{\mathbf{X}_k = \mu_{k|k-1}} = \begin{bmatrix} -\frac{p_y}{p_x^2+p_y^2} & 0 & \frac{p_x}{p_x^2+p_y^2} & 0 \\ \frac{p_x}{\sqrt{p_x^2+p_y^2}} & 0 & \frac{p_y}{\sqrt{p_x^2+p_y^2}} & 0 \end{bmatrix} \tag{22}$$

Ce qui nous permet d'écrire d'après Tailor Lagrange autour de $\mu_{k|k-1}$,

$$\mathbf{Y}_k \approx g(\mu_{k|k-1}) + \mathbf{J}_g(\mu_{k|k-1}) \times (\mathbf{X}_k - \mu_{k|k-1}) + V_k \tag{23}$$

En réarrangeant les termes pour faire apparaitre une écriture sous la forme (1), on peut écrire :

$$\mathbf{Y}_k \approx \mathbf{J}_g(\mu_{k|k-1})\mathbf{X}_k + \underbrace{g(\mu_{k|k-1}) - \mathbf{J}_g(\mu_{k|k-1})\mu_{k|k-1} + V_k}_{r_k,\ Biais\ introduit\ par\ la\ linearisation} \tag{24}$$

Comme on cherche un modèle du type $y_k = H(\mu_{k|k-1})X_k + V_k$ on va implémenter:

$$\underbrace{\widetilde{y_k}}_{\substack{Ce\ que\ l'on\ va \\ implementer\ dans\ le\ code}} = y_k - g(\mu_{k|k-1}) + \mathbf{J}_g(\mu_{k|k-1})\mu_{k|k-1} \tag{25}$$

3.3 Implémentation

Avec notre modélisation du problème ci-dessus, l'implémentation est assez simple, on reprends le code de la partie 1 (filtre Kalman simple) en y apportant quelques modifications pour prendre en compte notre linéarisation.

Pour se faire on va commencer par implémenter le calcul de la jacobienne :

```
1  def evaluate_jacobian(mu):
2    # last prediction format : [p_x, v_x, p_y, v_y]
3    return np.array([
4    [(-mu[2])/(mu[0]**2 + mu[2]**2), 0, mu[0]/(mu[0]**2 + mu[2]**2),
     0],
5    [mu[0]/np.sqrt(mu[0]**2 + mu[2]**2), 0, mu[2] /
6    np.sqrt(mu[0]**2 + mu[2]**2), 0]
7    ])
```

Code extraction 5: Fonction de calcul de la jacobienne de g

Puis on va directement remplacer l'ancienne fonction de calcul d'étape du filtre de Kalman en remplaçant :

$$\begin{cases} \mathbf{H} = \mathbf{J}_g \left(\underbrace{\mu_{k|k-1}}_{\substack{prediction\ que\ l'on\ utilise\ comme\ la\ meilleur\ approximation\ de\ X}} \right) \\ \\ y_{k_{EKF}} = y_k - g(\mu_{k|k-1}) + \mathbf{J}_g(\mu_{k|k-1})\mu_{k|k-1} \end{cases} \tag{26}$$

Ce qui donne en python :

```
def filtre_de_kalman_extended(F, Q, R, y_k, x_kalm_prec, P_kalm_prec
    ):
        # prediction :
        mu_prediction = F@x_kalm_prec
        # On va utiliser m_prediction comme meilleure approximation de X
        # a l'instant k !
        P_prediction = Q + F@P_kalm_prec@F.T

        # Variables upadte pour l'update de l'EKF:
        y_k_new = y_k - g(mu_prediction) + \
        evaluate_jacobian(mu_prediction)@mu_prediction
        # La nouvelle matrice H issue de la linearisation:
        H = evaluate_jacobian(mu_prediction)

        # update
        K = P_prediction@H.T@np.linalg.inv(H@P_prediction@H.T + R)
        # Mise a jour des vecteurs
        x_kalm_k = mu_prediction + K@(y_k_new - H@mu_prediction)
        P_kalm_k = (np.eye(len(P_kalm_prec[0])) - K@H)@P_prediction

        # Return a l'etape k des vecteurs
        return [x_kalm_k, P_kalm_k]
```

Code extraction 6: Filtre de Kalman Etendu

Avant de passer aux résultats, il est important de noter que nos observations sont donc dans le format de coordonnées polaire, pour faire l'affichage sous forme de graph il faut donc les retransformer dans le système cartésien, ce que l'on fait dans le programme principal qui lui ne change pas par rapport à la première version :

```
# Meme exercice mais avec l'EKF:
vecteur_x = creer_trajectoire(F, Q, x_init, T)
vecteur_y = creer_observation_EKF(R_EKF, vecteur_x, T)

# Estimation des etats
#....


# On fait la transformation inverse pour l'affichage des donnees
    observees en cartesien:
vecteur_y_converted = np.zeros((2,100))
for i in range(T):
vecteur_y_converted[0, i] = vecteur_y[1, i]*np.cos(vecteur_y[0, i])
vecteur_y_converted[1, i] = vecteur_y[1, i]*np.sin(vecteur_y[0, i])

# Plot the data:
draw_all2(vecteur_x, vecteur_y_converted, x_est, sigma_Q, sigma_px,
    sigma_py, erreur_quad(x_est, vecteur_x))
```

Code extraction 7: Filtre de Kalman Etendu

On peut donc maintenant passer aux résultats.

3.4 Résultats

On va procéder comme pour la partie 1. On simule avec les même valeurs de σ_Q et de σ_{px}, σ_{py}.
On obtient alors :

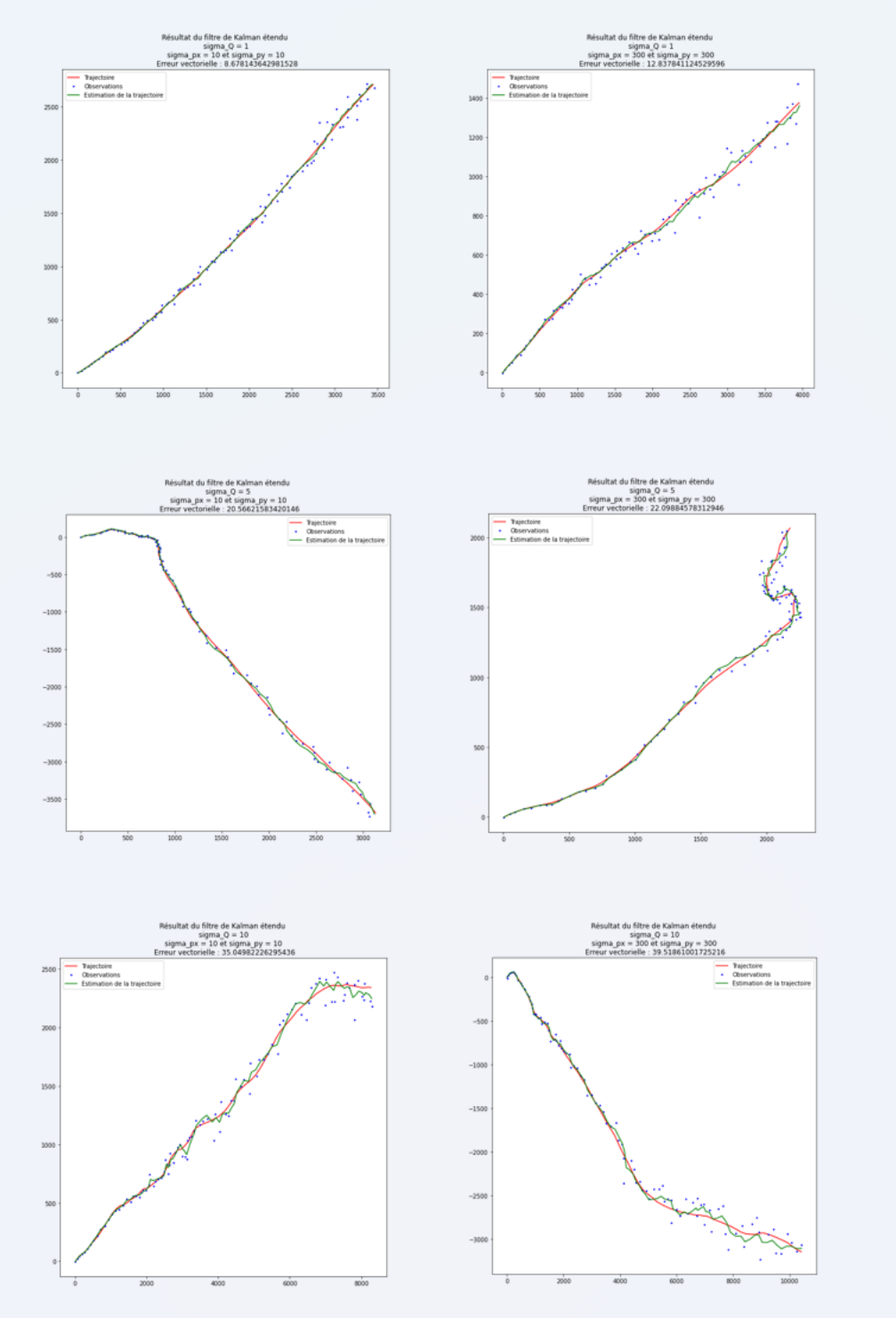


Figure 6: Résultats du filtre de Kalman étendu, avec différentes valeurs de bruit

On peut faire plusieurs remarques :
L'évolution du comportement de notre filtre avec nos différents niveaux de bruits semble correspondre à ce que l'on a obtenu dans la première partie. A la différence qu'il semble perdre en qualité au fur et à mesure que l'on augmente le nombre d'observation, on s'en rend bien compte en fin de nos trajectoires la prédiction commence à être bruité. Cela peut avoir comme origine la propagation du biais liée à notre linéarisation. Il dépend de notre prédiction liée elle même à ce même biais, on a donc un phénomène amplificateur sur l'erreur de prédiction, qui va se propager (forme récursive du filtre de kalman) et s'amplifier. L'utilisation d'une telle approche n'est donc viable qu'avec des bruit de mesures très faibles (ce qui requiers une grande attention dans le choix des outils de mesure) et une grande attention sur notre linéarisation.
Ce qui nous mène au second problème, on a linéarisé une fonction non linéaire, mais cette linéarisation peut poser problème si la fonction ne peut pas être linéarisé localement autour de la dernière estimation. Ici on considère un arctan comme localement linéaire (ok pour ce cas d'application). Mais il existe de nombreux cas ou l'EKF ne sera pas utilisable à cause de ce biais de linéarisation trop grand et la non linéarité de la fonction localement.

4. Conclusion

Nous avons vu à travers ce TP la théorie globale du filtre de Kalman ainsi qu'un exemple d'implémentation de ce même filtre. Il nous ai apparu l'importance de la sélection du modèle dans le cadre linéaire, la complémentarité du modèle de filtre étendu permettant une brève extension à certaines fonctions non linéaires ainsi que ses limites dans le cadre de fonctions peut localement linéarisables.
Cette méthode robuste à fait ses preuves dans de nombreux cas d'application, il aurait été inintéressant ici de présenter un cas plus pratique de son utilisation. Nous laisserons le soin au lecteur de cet exercice.
Dans le prochain TP nous verrons une méthode plus générale évitant cette étape de non linéarisation : le filtrage particulaire.

A. Appendices

Le code source complet est disponible ici : https://github.com/CharlesFarhat/coursTSP/tree/2A/MAT4201