2E ANNÉE

PROJET2A

Detection et prédiction de la trajectoire d'une balle

Auteur :
DHAINAUT MARIN
VARRE ARTHUR

12 Mars 2024

Table des matières

1 Introduction

Lors de ce projet de deuxième année nous avions pour objectif de développer un module de détection et de tracking d'objet dans le cadre de la RoboCup.

les exigences étaient les suivante :

— Avoir une prédiction rapide et précise (En mesure d'arrêter un tir de l'ordre de 50km/h)
— Utilisé les caméras ainsi que le module JeVois Pro
— Être en mesure de détecter et traquer différent type d'objet

Pour répondre a ces exigence nous avons organiser notre réflexions selon deux axes :

— Détection et traitement des objet grâce au caméra
— Cracking et prédiction a l'aide d'un filtre de kalman

de plus nous devions respecter le cahier des charge suivant que nous avons organiser via un diagramme pieuvre :

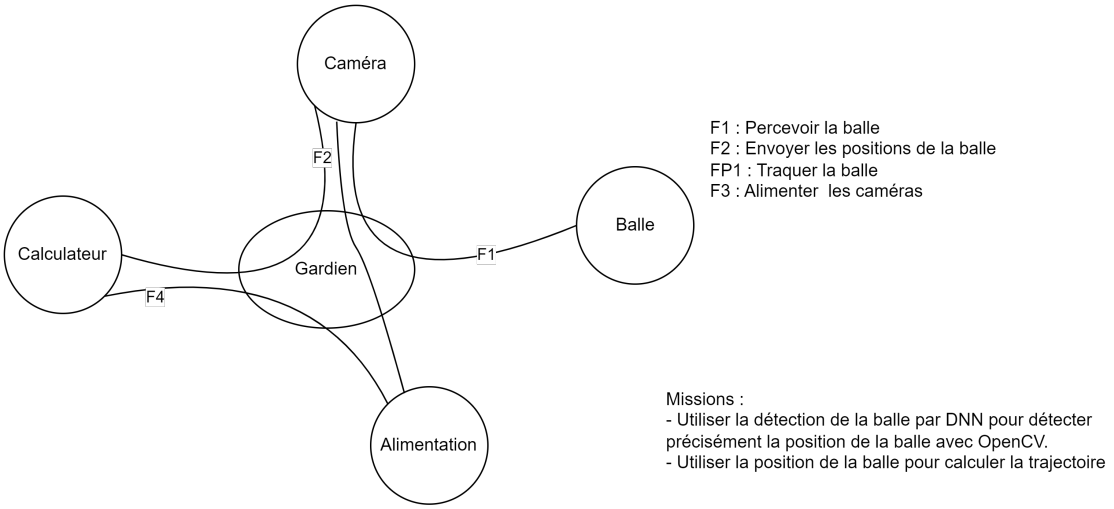

FIGURE 1 – diagramme pieuvre des exigences

ainsi qu'un tableau plus détaillé :

Détection de balle	
Objectif	Utiliser la détection de la balle par DNN pour détecter précisément la position de la balle avec OpenCV en utilisant le processeur de la jeVois pro.
Contraintes	La détection OpenCv ne doit pas ralentir la fréquence d'acquisition de la caméra (60fps)
Crucialité	Haute

Envoi des détections	
Objectif	Envoyer le centre de la balle et le rayon ainsi que les coordonné dans le référentiel du robot en USB au PC
Contraintes	Le grand angle de la camera, son orientation ne doivent pas affecter la position de la balle
Crucialité	moyen

Prédiction de la trajectoire	
Objectif	Prédire la trajectoire de la balle de façon précise (+/- la largeur du bras/2) a l'aide des première détections
Contraintes	les detection bruité
Crucialité	Evaluation de la faisabilité

FIGURE 2 – Cahier des charges

2 Détection de ballon de foot

La détection précise de ballons est essentielle pour espérer produire une trajectoire fiable pour l'extrapoler. Pour cela, nous allons utiliser une approche combinée entre une détection d'objet par réseau neuronal et un algorithme de détection de balle jaune.

2.1 Détection de balle par réseau neuronal

Pour détecter les objets dans leur environnement les robots de la RoboCup utilisent YOLO (You Look Only Once), une famille d'algorithme temps réel de détection d'objets. Plus spécifiquement, nous avons pris en main la version "YOLOv7-tiny" qui tourne à environ 100fps sur les caméras JeVois Pro grâce au processeur neuronal Hailo-8 de 26 TOPS. Le choix de la version "tiny" est crucial pour assurer une détection rapide tel que décrit dans la partie 2.4. Les données de sortie de la caméra à notre disposition sont représentées dans le tableau 1. Dans l'ordre les données signifient : type fiabilité centreXBoundingBox centreYBoundingBox largeurBoundingBox hauteurBoundingBox

but 94 -194 -449 416 172
ballon 85 -138 -252 131 124
poteau 80 -178 -452 31 169
robot_rct 77 941 -283 53 235
poteau 74 172 -444 41 169
humain 61 -1000 -563 334 1065
YellowBall 777 -163 -260 110 107

TABLE 1 – Sortie des détections de la caméra

2.2 Limitations des Bounding Box YOLO

Malgré l'efficacité de YOLO, ses bounding box, qui définissent l'emplacement et la taille de l'objet détecté, peuvent parfois manquer de précision. Même une petite différence de pixels dans la bounding box à distance peut se traduire par une erreur significative dans la position réelle du ballon. Pour atténuer ce problème, une idée est d'utiliser la bounding box de YOLO dans un script OpenCV complémentaire qui affine le processus. En effet, la taille d'entrée du réseau neuronal est 640x640. Nous perdons donc plus de la moitié des informations de l'image ce qui est problématique lorsque le ballon est éloigné et donc ne mesure que quelques pixels. Nous allons donc utiliser la détection YOLO pour extraire une région de l'image 1920x1080 entière. Faire une recherche restreinte dans la zone définie par la bounding box de YOLO est ainsi peu couteux et utilise l'image dans sa pleine résolution. Nous avons donc cherché plusieurs moyens de détecter un ballon dans une sous image en utilisant la librairie OpenCV en c++.

2.3 Détection de ballons par OpenCV

Pour détecter les ballons nous avons premièrement utilisé un algorithme qui réalise successivement ces étapes :

1. Conversion en espace de couleur HSV (Teinte, Saturation, Valeur) : Cette conversion nous permet de mieux isoler les couleurs et de travailler plus efficacement avec les tons et de s'affranchir de la luminosité dans l'image.

2. Détermination expérimentale des seuils pour le jaune : À l'aide d'expérimentations et d'analyses visuelles, nous avons déterminé les valeurs de seuil haut et bas appropriées pour détecter les ballons jaunes dans l'image.

3. Binarisation de l'image : Nous appliquons ensuite la méthode de binarisation pour séparer les pixels de l'image en deux catégories : ceux qui sont dans la plage de couleurs jaunes et ceux qui ne le sont pas.

4. Érosion et dilatation : Pour lisser les contours et éliminer le bruit, nous appliquons une érosion suivie d'une dilatation de même intensité à l'image binarisée. La figure 3 montre le résultat de la détection de coutours pour différents distances et conditions.

5. Recherche du plus petit rectangle englobant les contours : En utilisant les fonctions de détection de contours d'OpenCV, nous identifions le plus petit rectangle englobant les contours des objets jaunes, correspondant à la balle jaune dans notre cas.

6. Renvoi des coordonnées du rectangle : Les coordonnées de ce rectangle sont ensuite renvoyées en tant que détection de "YellowBall", conformément au format des détections produites par la caméra avec YOLO.

Nous avions initialement exploré l'utilisation de l'algorithme HoughCircles fourni par OpenCV pour détecter les cercles dans les images. Cet algorithme fonctionnait bien dans des cas optimaux, mais il présentait une limitation importante : en raison de la déformation optique, il ne parvenait à "fiter" un cercle de manière précise que lorsque le ballon était directement en face de la caméra. Cette limitation est due au fait que l'algorithme HoughCircles ne peut détecter que des cercles parfaits et ne fonctionne pas avec des ellipses.

En pratique, bien que l'algorithme HoughCircles puisse parfois détecter des ballons elliptiques, le centre de la détection par Hough oscillait constamment entre l'aphélie et la périhélie de la forme elliptique du ballon déformé. Cette instabilité rendait la détection peu fiable et imprécise, ce qui nous a poussés à écarter cette option assez rapidement.

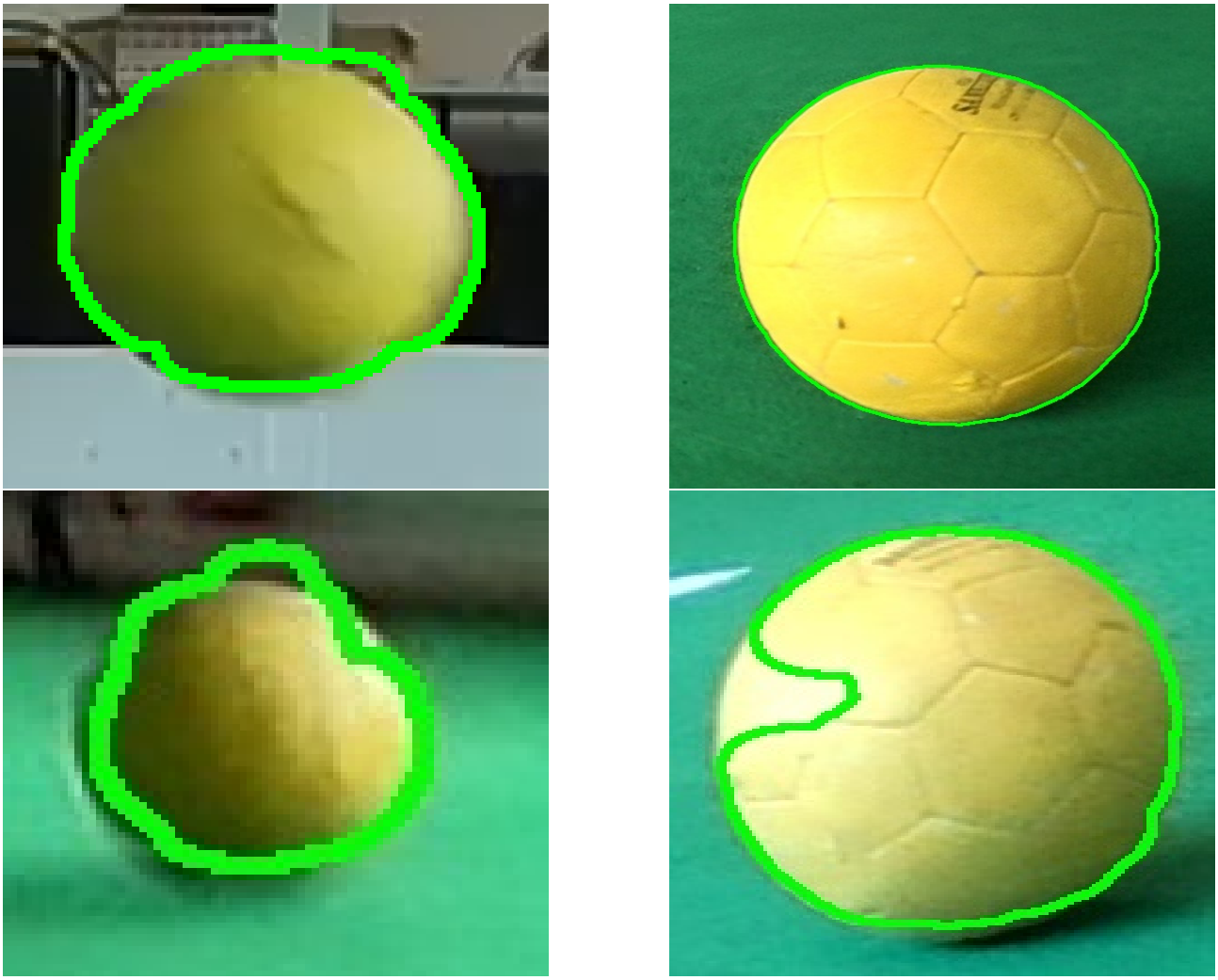

FIGURE 3 – Exemple de seuillage de ballons

2.4 Résultats de la détection mixte

La détection de ballon mixte nous a donné pleinement satisfaction. En effet, comme le montre la figure 4 les positions sont bien plus stables avec la méthode mixte qu'avec YOLO uniquement. De plus, le tableau 2 indique l'écart-type normalisée des différentes coordonnées des deux méthodes pour un ballon immobile. On remarque que la largeur et la hauteur des bounding boxs sont bien plus stables avec la méthode mixte qu'avec YOLO, ce qui est très important dans le cas du ballon pour déterminer sa distance relative au robot.

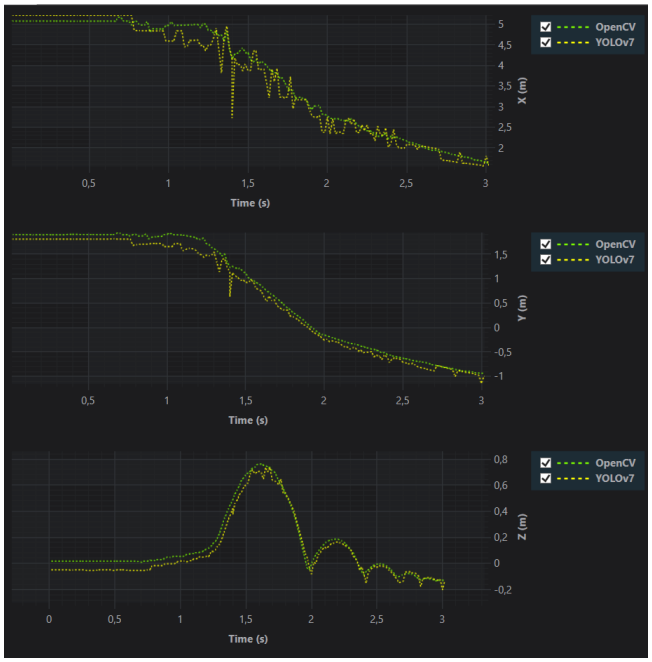

FIGURE 4 – Comparaison des méthodes de détection de balle

Il est important de noter que pour obtenir le plus de points de mesures possible il faut utiliser la totalité des performances de la caméra. Pour cela, nous avons poussé la fréquence d'acquisition des images à 60 fps. Il est donc nécessaire pour obtenir une fréquence de restitution de 60 fps que le réseau neuronal tourne au moins à 60 fps, en théorie. En pratique il faut qu'il tourne bien plus vite car la caméra induit des temps de pré-processing (environ 3ms) et post-processing (environ 1ms) pour adapter les tailles d'images à celle attendue par le réseau neuronal. Il reste donc seulement quelques millisecondes pour faire la détection par seuillage OpenCV. Nous avons eu quelques problèmes de vitesse au départ car l'érosion et la dilatation était trop gourmand. Finalement en réduisant leur amplitude nous sommes parvenus à atteindre entre 55 et 58 fps.

σ normalisé (%)	X	Y	Largeur	Hauteur
Yolo	-117,683	-1,604	4,926	11,021
Mixte	-43,575	-1,633	4,251	4,437

TABLE 2 – Tableau comparatif de la dispersion d'une balle immobile à 5m

3 Description des modèles

Dans l'objectif de pouvoir traquer ou prédire une grande partie des objets qui entoure le robot nous avons décider de définir plusieurs type d'objet qui sont pour l'instant fixer a 3 :

— Flying
— OnGround
— Static

Chacun de ces type possède un modèle d'état différent, En réalité dans le modèle d'état on se rend compte que seul la matrice B et le vecteur u change en fonction du type, nous les définissons ci dessous :

— flying : $Bu = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -mg \end{pmatrix}$

— OnGround : $Bu = \begin{pmatrix} 0 \\ coeffriction \\ 0 \\ 0 \\ coeffriction \\ 0 \end{pmatrix}$

— Statik : $Bu = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$

— m est la masse de l'objet
— coeffriction traduit le frottement lorsque la balle roule sur le sol

3.1 Exemple de la determination du modèle d'état flying

Pour le modèle d'état flying on considère une balle dans l'espace soumis a la gravité, on negligera les frottement, en effet la durée du tir étant très courte et les frottements sur une sphère étant moindre on peut s'en affranchir.

$$m\vec{a} = \vec{F_{\text{pesanteur}}} \tag{3.1}$$

où :
— m est la masse de la balle,
— $\vec{a}$ est le vecteur accélération de la balle,
— $\vec{F_{\text{pesanteur}}}$ est la force de pesanteur agissant sur la balle (généralement, $\vec{F_{\text{pesanteur}}} = -m\vec{g}$, où $\vec{g}$ est le vecteur accélération gravitationnelle),

L'équation du PFD projetée sur les axes x, y, et z pour la balle est donnée par :

$$ma_x = F^x_{\text{pesanteur}} \tag{3.2}$$
$$ma_y = F^y_{\text{pesanteur}} \tag{3.3}$$
$$ma_z = F^z_{\text{pesanteur}} \tag{3.4}$$

soit :

$$ma_x = 0 \tag{3.5}$$
$$ma_y = 0 \tag{3.6}$$
$$ma_z = -mg \tag{3.7}$$

où :
— m est la masse de la balle,
— a_x, a_y, a_z sont les accélérations de la balle dans les directions x, y, et z respectivement,
— $F^x_{\text{pesanteur}}$, $F^y_{\text{pesanteur}}$, $F^z_{\text{pesanteur}}$ sont les composantes de la force de pesanteur agissant sur la balle le long des axes x, y, et z respectivement (généralement, $F^x_{\text{pesanteur}} = 0$, $F^y_{\text{pesanteur}} = 0$, $F^z_{\text{pesanteur}} = -mg$),

Le vecteur d'état X est donné par :

$$X = \begin{pmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ z \\ \dot{z} \end{pmatrix}$$

où :
— x est la position de la balle dans la direction x,
— $\dot{x}$ est la vitesse de la balle dans la direction x,
— y est la position de la balle dans la direction y,
— $\dot{y}$ est la vitesse de la balle dans la direction y,
— z est la position de la balle dans la direction z,
— $\dot{z}$ est la vitesse de la balle dans la direction z.

Le modèle d'état est donné par les matrices suivantes :

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad C = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$B = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \qquad D = 0$$

$$\text{Tq } \dot{X} = AX + Bu \text{ et } Y = CX + D$$

où :
— A est la matrice d'état,
— B est la matrice d'entrée,
— C est la matrice d'observation,
— D est la matrice de transmission directe,
— u = -mg

4 Filtre de Kalman

Les filtres de Kalman sont une famille de filtres qui permettent de calculer de nombreuses choses différentes. Dans notre cas, nous utilisons un filtre de Kalman dans le but de prévoir la trajectoire d'un objet.

Ce filtre fonctionne de manière récursive et utilise plusieurs mesures bruitées pour définir l'état final. Il passe par plusieurs étape :

— Modèle dynamique
— Prédiction
— Estimation

4.1 Modèle dynamique

La première étape est la définition du modèle dynamique, cette étape a été précédemment définie dans la partie 2. En plus du modèle d'état, il est nécessaire pour le fonctionnement du filtre de Kalman de définir des matrices de covariance :

— Matrice Q : La matrice Q dans le filtre de Kalman est une matrice de covariance qui représente l'incertitude du processus, c'est-à-dire l'incertitude associée à la prédiction de l'évolution de l'état du système au fil du temps. Cette matrice est utilisée dans le modèle dynamique du système pour prendre en compte le bruit du processus, c'est-à-dire les fluctuations aléatoires qui affectent l'évolution réelle du système.

— Matrice R : La matrice R dans le filtre de Kalman est une matrice de covariance qui représente l'incertitude des mesures, c'est-à-dire l'incertitude associée aux observations ou aux mesures du système. Cette matrice est utilisée pour prendre en compte le bruit de mesure, elle est définie empiriquement.

4.2 Prédiction

Le filtre de Kalman commence par prédire l'état futur du système en utilisant le modèle dynamique et l'estimation précédente de l'état. Cette prédiction estime où le système devrait être à l'instant k+1 en fonction de ses états passés et des commandes appliquées. On obtient cette prédiction grâce au calcul suivant :

$$\mathbf{x}_{\text{pred}_k} = \mathbf{A}\mathbf{x}_{\text{est}_{k-1}} + \mathbf{B}u \qquad \mathbf{p}_{\text{pred}_k} = \mathbf{A}\mathbf{x}_{\text{est}_{k-1}}\mathbf{A}^T\mathbf{Q}$$

Avec :

— $xpred_k$: le vecteur x prédit a l'instant k
— $xest_k$: le vecteur x estimé a l'instant k
— $ppred_k$: la matrice p de la covariance prédite a l'instant k
— A,B,Q : les matrices du modèle d'état

4.3 Estimation

Ensuite le filtre se sert des observations a l'état précèdent pour corrigé la prédiction précédente ainsi on peut ensuite prédire de nouveau le prochaine état a l'aide de la dernière prédiction corrigé. Cette correction est appelé l'estimation dans le filtre. On l'effectue grâce au calcul suivant :

$$\mathbf{tt}_k = \mathbf{C}\mathbf{p}_{\mathbf{pred_k}}\mathbf{C}^T + \mathbf{R}\ \mathbf{K}_k = \mathbf{p}_{\mathbf{pred}_k}\mathbf{C}^{\mathrm{T}}\mathbf{tt}_k^{-1}\ \hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K_k}(\mathbf{observation} - \mathbf{C_k}\mathbf{X}_{\mathbf{pred}})$$

4.4 Résumé

En résumé il est crucial de comprendre pour utiliser le filtre son sens de fonctionnement, le schéma ci dessous synthétise ce sens :

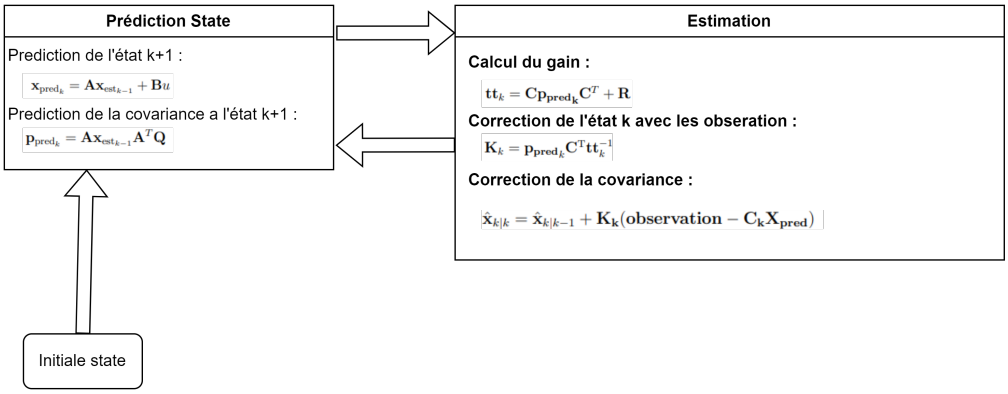

FIGURE 5 – Fonctionnement du filtre de kalman

5 Détection et tracking des objets

La stratégie pour la détection et le tracking des objets et la suivante :

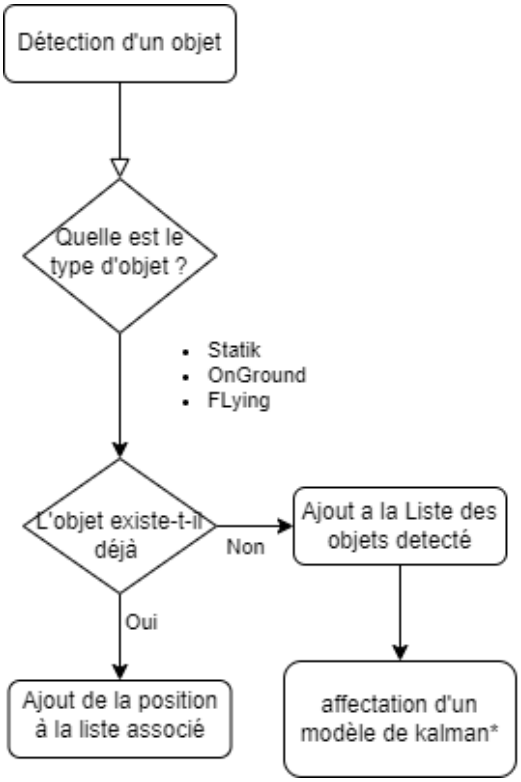

FIGURE 6 – stratégies de détection des objets

Pour déterminer si un objet a déjà été détecté, nous comparons sa position actuelle à celle enregistrée précédemment dans le filtre de Kalman. Si cette différence est supérieure à un seuil défini à 1 mètre, nous considérons que l'objet est nouveau, et nous l'ajoutons à notre liste d'objets détectés.

Chaque objet de la liste possède un filtre de Kalman associé qui le suit. Si aucune détection n'apparaît pendant une durée de 3 secondes, nous considérons que l'objet a disparu, et nous le supprimons de notre liste d'objets détectés.

6 Implementation en C#

Nous avons ensuite implémenté la théorie ci-dessous en C# et avons pu réaliser des tests via des replays issus de la caméra. À l'issue de ces tests, nous avons pu observer que la détection de la balle avec la méthode du seuillage était très efficace et donnait des résultats prometteurs. De plus, voici les résultats obtenus avec le filtre de Kalman :

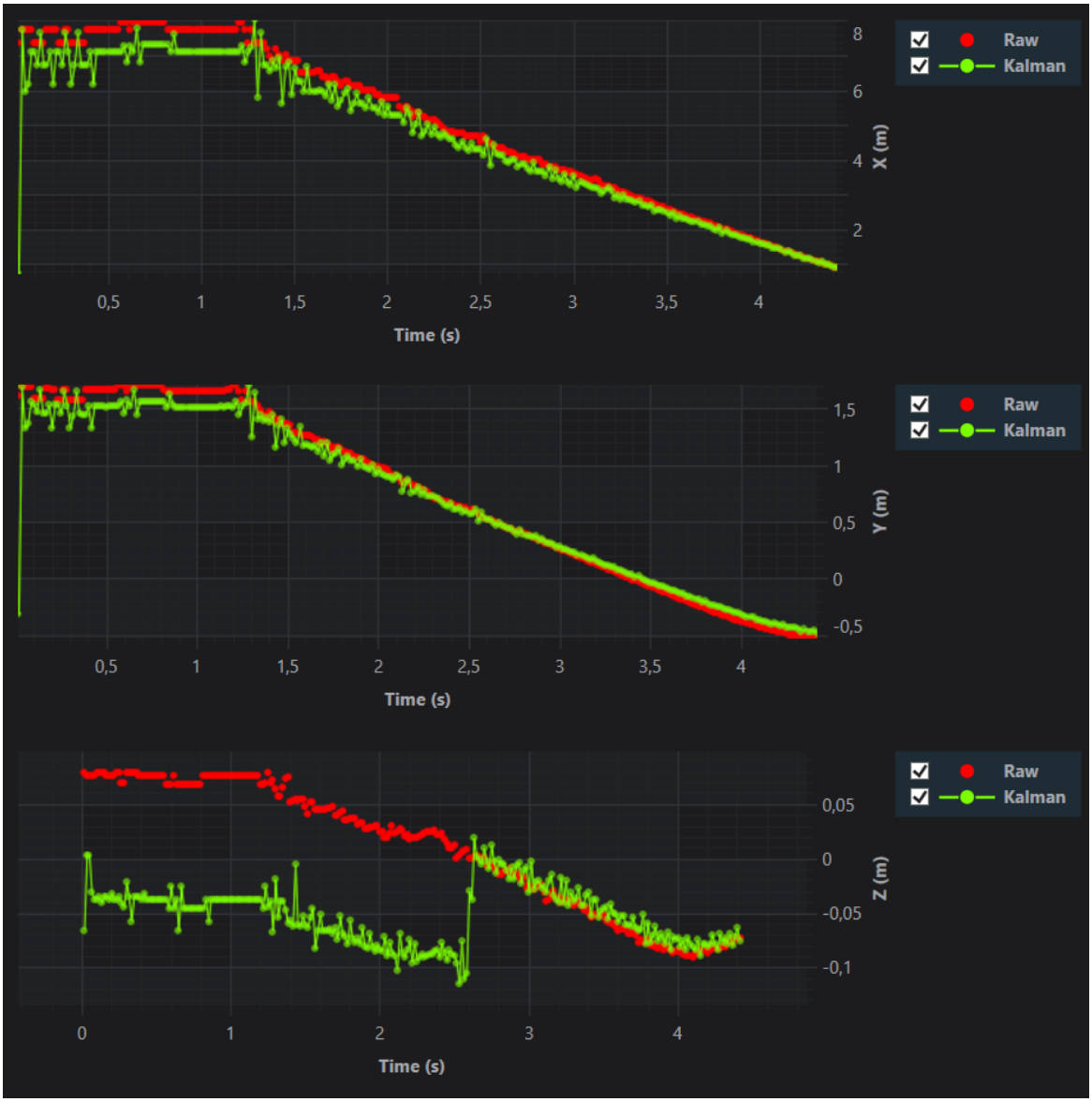

FIGURE 7 – Résultats du filtre de Kalman pour un ballon au sol

7 Conclusion

En conclusion, nous observons que notre filtre de Kalman présente une convergence trop lente pour permettre une prédiction précise à temps pour arrêter un tir ou suivre une passe rapide. Par conséquent, nous pensons qu'une régression linéaire serait mieux adaptée pour répondre au problème qui nous a été posé.

8 Annexes

```
public class CameraKalman
{
    public CameraKalman(ObjectType objectType, string type, double mass,
    double seuilDistance, double timeStamp, double xPos, double yPos, double
    zPos, double frictioncoeff = 0)
    {
      this.type = type;
      this.seuilDistance = seuilDistance;
      this.timeStamp = timeStamp;
      x = xPos;
      y = yPos;
      z = zPos;
      this.mass = mass;

      double[,] A =
      {
        {0,1,0,0,0,0 },
        {0,0,0,0,0,0 },
        {0,0,0,1,0,0 },
        {0,0,0,0,0,0 },
        {0,0,0,0,0,1 },
        {0,0,0,0,0,0 },
      };
      MatrixA = Matrix<double>.Build.DenseOfArray(A);

      double[,] C =
      {
        {1,0,0,0,0,0 },
        {0,1,0,0,0,0 },
        {0,0,1,0,0,0 },
        {0,0,0,1,0,0 },
        {0,0,0,0,1,0 },
        {0,0,0,0,0,1 },
      };

      MatrixC = Matrix<double>.Build.DenseOfArray(C);
      MatrixQ = Matrix<double>.Build.DenseDiagonal(6, 6, 4);
      MatrixR = Matrix<double>.Build.DenseDiagonal(6, 6, NoiseMeasurement);

      double[,] B;
      switch (objectType)
      {
        case ObjectType.flying:
          B = new double[,]
          {
            {0 },
            {0 },
            {0 },
            {0 },
            {0 },
            {-gravity*this.mass },
          };
          MatrixB = Matrix<double>.Build.DenseOfArray(B);
          break;
```

```
53
54          case ObjectType.Onground:
55            B = new double[,]
56            {
57              {0 },
58              {frictioncoeff },
59              {0 },
60              {frictioncoeff },
61              {0 },
62              {0 },
63            };
64            MatrixB = Matrix<double>.Build.DenseOfArray(B);
65            break;
66
67          case ObjectType.statik:
68            B = new double[,]
69            {
70              {0 },
71              {0 },
72              {0 },
73              {0 },
74              {0 },
75              {0 },
76            };
77            MatrixB = Matrix<double>.Build.DenseOfArray(B);
78            break;
79          default:
80            break;
81
82        }
83
84      }
85      public void UpdateParamsWithDetection(KalmanObject obj)
86      {
87        oldX = x;
88        oldY = y;
89        oldZ = z;
90        x = obj.x;
91        y = obj.y;
92        z = obj.z;
93        vX = (oldX - x)/timestep;
94        vY = (oldY - y)/timestep;
95        vZ = (oldZ - z)/timestep;
96        timestep = (double)1/60;
97        timeStamp += timestep;
98      }
99
100     public double DistanceToNewDetection(KalmanObject obj)
101     {
102       return Math.Abs(Math.Sqrt(
103         Math.Pow(obj.x - x, 2) +
104         Math.Pow(obj.y - y, 2) +
105         Math.Pow(obj.z - z, 2)));
106     }
```

```
107
108   public List<KalmanObject> kalmanObjectList = new();
109   Point3D kalmanPos;
110   List<(double, Vector<double>)> EstimatedTraj = new();
111   public void ProcessKalmanObject(KalmanObject newObj, int nbiter)
112   {
113     // on incremente le pas de temps par s curit  si jamais on ne voit pas
114     // de nouvel object dans la nouvelle liste des d tections.
115     // Si l'objet est de nouveau per u on remplacera timestep par 1/60
116     for (int i = 0; i<kalmanObjectList.Count; i++)
117     {
118       kalmanObjectList[i].timestep += (double)1/60;
119     }
120
121     // Si on detecte un element on le compare avec les objects du meme type
      :
122     // si la distance entre l'ancienne et la nouvelle d tection (kalman si
      disponible, sinon brute)
123     // est sup rieur    seuilDistance on c r    un nouvel object sinon on
      it re le filtre de kalman
124     // avec la nouvelle observation
125
126     // On cherche tous les objets du meme type puis on cherche si il
      appartient    un des groupes
127     List<KalmanObject> sameObjectTypeList = kalmanObjectList.Where(x => x.
      type == newObj.type).ToList();
128     List<KalmanObject> distanceToObjectList = sameObjectTypeList.Where(x =>
      x.DistanceToNewDetection(newObj) < x.seuilDistance).OrderBy(x => x.
      DistanceToNewDetection(newObj)).ToList();
129
130     if (distanceToObjectList.Count > 0)
131     {
132       // On l'associe au plus proche
133       KalmanObject closestObj = distanceToObjectList.First();
134       int oldObjindex = kalmanObjectList.FindIndex(x => x == closestObj);
135       closestObj.UpdateParamsWithDetection(newObj);
136       closestObj.canFilterBeInitialized = true;  // on ne peut initialiser
      le filtre que si on a eu 2 positions pour calculer la vitesse
137       kalmanObjectList[oldObjindex] = closestObj;
138     }
139     // Si on a pas de groupe assez proche on en c r    un nouveau
140     else
141     {
142       if (newObj != null)
143       {
144         newObj.UpdateParamsWithDetection(newObj);
145         kalmanObjectList.Add(newObj);
146       }
147     }
148
149     // Si on a des groupes trop vieux on les supprimes
150     if (kalmanObjectList.Count > 0)
151     {
152       List<int> indexToRemove = new();
```

```
153       for (int i = 0; i < kalmanObjectList.Count; i++)
154       {
155         if (newObj.timeStamp - kalmanObjectList[i].timeStamp > 1) // 1
    seconde
156         {
157           indexToRemove.Add(i);
158         }
159       }
160       foreach (int index in indexToRemove)
161       {
162         kalmanObjectList.RemoveAt(index);
163       }
164     }
165
166     // On failt kalman
167     for (int i = 0; i<kalmanObjectList.Count; i++)
168     {
169       if (kalmanObjectList[i].canFilterBeInitialized && !kalmanObjectList[i
    ].isFilterInitialized)
170       {
171         InitFilter(kalmanObjectList[i]);
172         kalmanObjectList[i].isFilterInitialized = true;
173       }
174       if (kalmanObjectList[i].isFilterInitialized)
175       {
176         IterateFilter(kalmanObjectList[i]);
177       }
178     }
179   }
180
181   static void InitFilter(KalmanObject obj)
182   {
183
184     //Initi des variables internes du filtre
185     Vector<double> xInit = Vector<double>.Build.DenseOfArray(new double[] {
    obj.x, obj.vX, obj.y, obj.vY, obj.z, obj.vZ });
186
187     Matrix<double> pInit = Matrix<double>.Build.DenseDiagonal(obj.MatrixQ.
    RowCount, obj.MatrixQ.RowCount, 0.1);
188
189     obj.xPred = Vector<double>.Build.Dense(obj.MatrixQ.RowCount, 0);
190     obj.pPred = Matrix<double>.Build.Dense(obj.MatrixQ.RowCount, obj.MatrixQ
    .RowCount, 0);
191     obj.xEst = Vector<double>.Build.Dense(obj.MatrixQ.RowCount, 0);
192     obj.pEst = Matrix<double>.Build.Dense(obj.MatrixQ.RowCount, obj.MatrixQ.
    RowCount, 0);
193     obj.K = Matrix<double>.Build.Dense(obj.MatrixQ.RowCount, obj.MatrixR.
    RowCount, 0);
194     obj.Inx = Matrix<double>.Build.DenseDiagonal(obj.MatrixQ.RowCount, obj.
    MatrixQ.RowCount, 1);
195
196     obj.xPred = xInit;
197     obj.xEst = xInit;
198     obj.pPred = pInit;
```

```
199       obj.pEst = pInit;
200     }
201
202     void IterateFilter(KalmanObject obj)
203     {
204       if (obj.z <= 0) {
205         obj.MatrixB[5,0] = 0;
206       }
207
208       // Pr diction
209       obj.xPred = obj.MatrixA.Multiply(obj.xEst)+obj.MatrixB.Column(0);
210       obj.pPred = obj.MatrixA.Multiply(obj.pEst.Multiply(obj.MatrixA.Transpose
      ())) + obj.MatrixQ;
211       kalmanPos = new Point3D(
212         obj.xEst[0] + (obj.xPred[1])*1/60,
213         obj.xEst[2] + (obj.xPred[3])*1/60,
214         obj.xEst[4] + (obj.xPred[5])*1/60);
215
216       // Estimation
217       Vector<double> observation = Vector<double>.Build.DenseOfArray(new
      double[]
218       {
219         obj.x,
220         obj.vX,
221         obj.y,
222         obj.vY,
223         obj.z,
224         obj.vZ,
225       });
226
227       //Formule magique !
228       var tt = obj.MatrixC.Multiply(obj.pPred).Multiply(obj.MatrixC.Transpose
      ()) + obj.MatrixR;
229       obj.K = obj.pPred.Multiply(obj.MatrixC.Transpose()).Multiply(tt.Inverse
      ());
230       obj.xEst = obj.xPred + obj.K.Multiply(observation - obj.MatrixC.Multiply
      (obj.xPred));
231       //obj.xEst = observation;
232
233       if (double.IsNaN(obj.xEst[0]))
234         obj.xEst[0] = 0;
235
236       obj.pEst = (obj.Inx - obj.K.Multiply(obj.MatrixC)).Multiply(obj.pPred);
237     }
238 }
```

Listing 1 – Example C++ code