

# **Appontage automatique d'un drone sur une plate-forme mobile**

**Tom Ravaud<sup>1</sup>**

Encadré par

**José-Luis Vilchis-Medina<sup>2</sup>**

**Pascal Morin<sup>3</sup>**

<sup>1</sup>**École des Ponts ParisTech, Champs-sur-Marne, France**  
`tom.ravaud@eleves.enpc.fr`

<sup>2</sup>**Institut de Recherche de l'École navale (IRENav), Lanvéoc, France**  
`jl.vilchis_medina@ecole-navale.fr`

<sup>3</sup>**Institut des Systèmes Intelligents et de Robotique (ISIR), Paris, France**  
`pascal.morin@sorbonne-universite.fr`

# Table des matières

<b>Introduction</b>	<b>3</b>
<b>1 Environnement de simulation</b>	<b>7</b>
1.1 La simulation en robotique . . . . .	7
1.1.1 Les simulateurs . . . . .	7
1.1.2 Premiers pas avec ROS et Gazebo . . . . .	8
1.2 Modélisation et commande de la plate-forme mobile . . . . .	9
1.2.1 Description du système . . . . .	9
1.2.2 Motorisation et contrôle . . . . .	11
1.3 Ajout de capteurs : le cas de la caméra . . . . .	14
1.4 Simulation du drone . . . . .	15
<b>2 Le <i>template tracking</i> pour l'estimation de <i>pose</i></b>	<b>19</b>
2.1 Le problème de l'estimation de <i>pose</i> . . . . .	19
2.1.1 Contexte théorique . . . . .	19
2.1.2 Le problème P-n-P . . . . .	23
2.1.3 Obtention de correspondances 2D-3D . . . . .	27
2.2 Le <i>template tracking</i> . . . . .	29
2.2.1 <i>Template tracking</i> basé <i>features</i> . . . . .	30
2.2.2 <i>Template tracking</i> basé <i>template</i> . . . . .	37
2.3 Analyse des résultats . . . . .	44
2.4 Estimation d'état . . . . .	45
2.4.1 Equations d'état . . . . .	45
2.4.2 Filtrage de Kalman . . . . .	47
2.4.3 Mise en application et analyse des résultats . . . . .	49
<b>3 Asservissement visuel</b>	<b>50</b>
3.1 L'asservissement visuel, plusieurs approches . . . . .	50
3.2 <i>PBVS</i> . . . . .	53
3.3 <i>IBVS</i> . . . . .	55
3.4 Prise en compte du sous-actionnement . . . . .	58
<b>4 Du décollage à l'appontage</b>	<b>60</b>
4.1 Modélisation du comportement du drone par une machine d'état . . . . .	60
4.2 Décision de l'appontage . . . . .	62
<b>Perspectives et conclusion</b>	<b>64</b>

## *TABLE DES MATIÈRES*

---

<b>Annexes</b>	<b>65</b>
A    La calibration de la caméra . . . . .	65
B    Les transformations géométriques des images . . . . .	66
C    Focus sur l'homographie . . . . .	68
D    RANSAC . . . . .	69
<b>Bibliographie</b>	<b>73</b>

# Introduction

## Mise en contexte

La robotique mobile vise à concevoir des systèmes autonomes capables d'agir dans leur environnement à partir de la perception qu'ils en ont [12]. Elle connaît un essor sans précédent depuis le début des années 2000 pour ses applications aussi variées que les domaines auxquels elles se rattachent. Le secteur militaire ne fait pas exception, où l'utilisation de robots permet à l'homme de s'affranchir de tâches difficiles et dangereuses [21].

Ce projet de stage, initié par l'Institut de Recherche de l'École navale (IRENav), ambitionne d'étudier et de simuler l'appontage automatique d'un drone sur une plate-forme mobile. Une telle étude a une application concrète et immédiate pour l'appontage d'hélicoptères sur un navire dans un contexte militaire. L'appontage est l'une des missions les plus délicates et les plus à risques auxquelles les pilotes d'hélicoptères sont confrontés. En cause, la zone réduite sur laquelle l'hélicoptère peut se poser, les mouvements du navire ou la faible visibilité à bord de l'appareil, cela couplé à des conditions météorologiques souvent défavorables [48]. C'est pourquoi des dispositifs ont fait l'objet d'études et embarquent désormais les navires français, à l'instar de *SAFECOPTER* d'ArianeGroup.

Ce vaste sujet qui m'a été proposé fera à terme l'objet d'une thèse au sein du même laboratoire de recherche. Ainsi, il s'agit plus ici d'une première approche visant à couvrir l'essentiel des problématiques rencontrées dans l'appontage automatique, qu'une étude approfondie centrée sur l'un des enjeux du sujet. Avant de poursuivre, il est important de préciser les hypothèses que nous avons faites sur les conditions de l'appontage :

- Le drone utilisé est à voilure tournante, de type *VTOL*, de l'anglais *Vertical Take-off and Landing*. Autrement dit, les solutions développées doivent pouvoir être appliquées sans distinction aux cas de l'hélicoptère avec un seul rotor principal, du quadrirotor ou de l'hexarotor [20] ;
- Le drone est équipé d'une suite minimale de capteurs qui lui permettent d'estimer sa position ou sa vitesse en temps réel (GPS et centrale inertuelle), ainsi que d'une caméra ;
- La plate-forme est coopérative, on suppose connaître son visuel et ses dimensions à l'avance.

## Bref état de l'art

Nombre d'études se sont intéressées au problème de l'atterrissement automatique d'un drone sur une plate-forme mobile en s'aidant de la vision [14][13]. Globalement, toutes font ressortir deux étapes : la détection de la plate-forme et la synthèse de la commande du drone.

Il existe plusieurs manières de détecter la plate-forme ; la plus simple consiste à intégrer

un GPS à la plate-forme, afin qu'elle communique sa position au drone. Cependant, les mesures par GPS sont rarement assez précises et la fréquence d'acquisition est trop basse pour rendre compte de la cinématique de la plate-forme en temps réel. L'utilisation d'un GPS RTK (*Real Time Kinematic*) n'est pas non plus envisageable dans la plupart des cas car trop onéreuse [8]. Aussi, il n'est pas toujours possible de modifier la plate-forme pour l'équiper de capteurs et d'envoyer leurs mesures au drone en temps réel. A l'inverse, l'emploi d'une caméra, embarquée par le drone, connaît un grand succès et est presque systématique, en complément ou non d'un GPS sur la plate-forme, en tant qu'elle constitue aujourd'hui un capteur léger, peu encombrant, à faible coût et qui fournit une grande quantité d'informations en temps réel sur l'environnement. D'autres types de capteurs ont été envisagés, à l'instar des LIDAR (*Light Detection And Ranging*), des capteurs à ultrasons ou infrarouges, mais ils ne sont pas toujours adaptés à des applications en extérieur<sup>1</sup>, ou aux contraintes de coût et d'encombrement [13]. Si certains de ces capteurs constituent de bonnes sources d'information complémentaires, nous privilégions la caméra comme solution principale pour la détection de la plate-forme.

Différentes approches ont été expérimentées pour la détection de la plate-forme dans le plan image, suivie ou non d'une reconstruction de sa position et de son orientation dans un repère de l'espace. Dans la grande majorité des cas, l'utilisation de marqueurs visuels facilite la reconnaissance de la plate-forme ; il peut s'agir aussi bien de formes que de couleurs caractéristiques. On retrouve par exemple le classique "H" majuscule [43, 23], les cercles concentriques de différentes tailles pour la détection à plusieurs niveaux d'éloignement [29, 25] et les populaires marqueurs ArUco [8] ou AprilTags [11, 13]. La solution du marqueur est pratique lorsque l'on veut se concentrer sur les problématiques de contrôle du drone, mais n'est pas suffisante quand on traite l'appontage appliqué à une situation réelle dans sa globalité. C'est pourquoi nous supposerons connaître à l'avance le visuel de la plate-forme, sans pour autant contraindre sa nature.

La commande du drone par vision fait appel aux techniques de l'asservissement visuel. Il existe deux approches principales : la première travaille dans l'espace 3D tandis que la seconde travaille directement dans le plan image. Elles cherchent toutes les deux à faire converger le drone vers une position cible, définie soit explicitement dans l'espace, soit implicitement à travers la position d'indices visuels dans le plan image, et sont connues sous le nom d'asservissement visuel basé position (*PBVS* pour *Position-Based Visual Servoing*) et basé image (*IBVS* pour *Image-Based Visual Servoing*) [4]. Ces méthodes ont chacune leurs avantages et leurs inconvénients, si bien que des tentatives de combiner les deux existent [5, 37, 28]. Aussi, il est important de signaler que ces stratégies ont initialement été développées pour positionner l'effecteur de bras manipulateurs, dont les six degrés de liberté sont commandés indépendamment, par rapport à une cible fixe. Ici, le problème est autre : nous cherchons à stabiliser un drone sous-actionné (car seuls quatre des six degrés de liberté peuvent être commandés) par rapport à une cible mobile. Si la prise en compte du mouvement de la plate-forme consiste à ajouter à la commande un terme qui dépend explicitement ou non de sa vitesse par rapport au drone [8, 5], le sous-actionnement est un défi plus large auquel quelques solutions ont été apportées [48, 30, 8]. Nous aurons l'occasion de discuter plus en détails de ces travaux par la suite.

A partir de la loi de commande synthétisée pour positionner le drone par rapport à une cible, certains font le choix de le faire atterrir à mesure qu'il se rapproche de la plate-forme pour des questions de rapidité de la mission [8], tandis que d'autres tentent

---

1. En particulier pour l'appontage sur un navire, le problème de la réflexion des ondes sur l'eau réduit la gamme de capteurs disponibles.

de stabiliser le drone au-dessus de la plate-forme et d'atterrir dans un second temps, conformément aux pratiques des pilotes d'hélicoptères de la Marine nationale [48]. Cette subdivision de l'appontage nous intéresse spécialement ; nous étudierons la possibilité d'intercaler une étape de décision du moment opportun pour apponter après avoir stabilisé le drone au-dessus de la plate-forme. Cette idée n'a pas ou peu été traitée à ce jour. Aussi, en amont de l'appontage, une phase d'approche est nécessaire de manière à ce que le drone soit suffisamment proche de la plate-forme pour qu'elle rentre dans son champ de vision. Ce n'est qu'à partir de ce moment là que l'asservissement visuel peut démarrer pour positionner le drone avec plus de précision par rapport à sa cible. Cette étape peut être réalisée simplement en envoyant au drone les coordonnées GPS de la plate-forme lorsqu'une telle communication est possible [11], ou par exploration de l'environnement [49]. Nous n'étudierons pas ce point par la suite et supposerons avoir la plate-forme dans le champ de vision au début de l'expérience. L'intégration logicielle de ces différentes tâches élémentaires qui composent l'appontage se sert d'une modélisation haut niveau du comportement du drone par une machine d'état [10, 8].

## Objectifs du stage

Le travail se fera en grande partie en simulation. Des expérimentations pratiques seront menées en parallèle des simulations dans le cadre de la thèse principalement. A cet effet, le laboratoire possède une plate-forme six axes, de type *Stewart*, et un drone hexarotor (Figure 1).



(a) Plateforme *Stewart*

(b) Drone DJI F550

FIGURE 1 – Matériel de l'IRENav

Pour structurer l'étude, nous avons dégagé plusieurs axes majeurs :

- La mise en place d'un simulateur dans lequel tester les solutions imaginées ;
- L'extraction de connaissances de l'environnement à l'aide des capteurs embarqués par le drone, et en particulier de la caméra ;
- La synthèse d'une loi de contrôle pour l'appontage du drone.

Nous nous appuierons sur l'écosystème **ROS** et le simulateur physique **Gazebo** pour développer notre environnement de simulation (Chapitre 1). Ensuite, nous traiterons le problème de l'estimation de la position et de l'orientation de la plate-forme par vision (Chapitre 2) et nous présenterons plusieurs stratégies de contrôle du drone, à nouveau par vision, connues sous le nom d'"asservissement visuel" (Chapitre 3). Enfin, nous tenterons de relier logiquement les différentes phases de l'appontage pour le simuler dans son entièreté (Chapitre 4).

## Ressources

Le code source développé pendant le stage est accessible en ligne sur ce [dépôt Github](#). Plusieurs vidéos de démonstration ont été réalisées, les liens vers celles-ci seront donnés au fil du texte.

# Chapitre 1

## Développement de l'environnement de simulation

Le sujet d'étude étant cadré, il est possible de s'attarder sur les méthodes déjà éprouvées pour la détection de la plate-forme et l'apportage du drone. Seulement, pour leur mise en pratique et leur validation, il nous faut du matériel (un drone et ses capteurs, un contrôleur de vol, une plate-forme, ...), ou une simulation numérique de ce dernier. C'est cette dernière option que nous avons adoptée dans un premier temps. Après avoir discuté des avantages de la simulation en robotique, nous présenterons les principales étapes de la conception de notre environnement de simulation.

### 1.1 La simulation en robotique

#### 1.1.1 Les simulateurs

En robotique, un simulateur désigne un outil qui permet de tester les performances d'un robot en travaillant sur un modèle numérique de ce dernier, et non directement sur sa version physique. Plus qu'une option, la simulation est en réalité un passage presque obligatoire dans la recherche en robotique mobile, et plus particulièrement en robotique mobile aérienne où l'on est amené à travailler avec des drones sujets à d'importants accidents.

Donnons une liste non exhaustive des avantages qu'a la simulation par rapport aux essais sur le robot réel :

- Un grand nombre d'essais peut être effectué en peu de temps (*try and error*). La modification du code source est également très rapide. En cela, la simulation robotique représente un gain de temps ;
- Comme suggéré précédemment, elle permet de travailler sans abîmer la version physique du robot. C'est un gain d'argent ;
- Elle est un outil de prototypage rapide et d'aide à la conception de nouveaux systèmes ;
- Il est possible de modifier les réglages du simulateur pour accroître la robustesse des algorithmes (conditions météorologiques, luminosité, température, ...).

Il existe de nombreux simulateurs, certains étant libres comme [Gazebo](#), [Webots](#) ou [OpenRAVE](#), et d'autres propriétaires comme [CoppeliaSim](#), [Isaac Sim](#) ou [RoboDK](#). La fonction première de ces logiciels est de reproduire le comportement qu'auraient des systèmes dans la réalité, soumis à la physique. Pour ce faire, ils sont équipés d'un moteur

physique, programme qui permet d'appliquer les équations de la mécanique newtonienne aux corps simulés. On en trouve à nouveau plusieurs, comme [Bullet](#), [ODE](#) (*Open Dynamics Engine*) ou [PhysX](#). Le choix d'un simulateur dépend des besoins de l'application. Dans le cas des bras manipulateurs, le monde de la simulation est largement dominé par les solutions industrielles conçues par les constructeurs (ABB, KUKA, Stäubli, ...) pour leurs propres produits. Malheureusement, ces dernières ne concèdent que peu de liberté à l'utilisateur quant à la personnalisation des systèmes simulés (ajout de nos propres capteurs, modification des pièces des robots, ...). Si ce n'est pas forcément gênant lorsque l'on étudie des bras manipulateurs, il ne serait pas envisageable de travailler de cette manière en robotique mobile. Cela justifie l'intérêt grandissant porté aux solutions *open source* [9]. Nous travaillerons dans toute la suite avec le logiciel de simulation Gazebo pour sa popularité, la grande liberté qu'il nous offre et sa compatibilité avec ROS.

[ROS](#), ou *Robot Operating System*, est un ensemble d'applications et d'outils qui permet de créer l'architecture logicielle d'un robot. Nous l'utiliserons conjointement à Gazebo pour simuler nos robots. Notez que de cette façon, il est aisément de passer de la simulation à l'expérimentation réelle ; les programmes ROS restent inchangés, seul Gazebo est remplacé par le monde réel.

### 1.1.2 Premiers pas avec ROS et Gazebo

Lorsque j'ai débuté mon stage, l'environnement de simulation n'existant pas encore, il était à construire de toutes pièces en fonction des besoins du stage et de la thèse qui en découlerait. En l'occurrence, nous voulions avoir d'un côté une plate-forme capable de reproduire des dynamiques de bateaux sur la mer, et d'un autre un drone équipé de plusieurs capteurs (GPS, centrale inertuelle, caméra). A ce stade, deux options peuvent être considérées : travailler à partir de modèles *open source* disponibles en ligne, ou les concevoir nous-même. Le compromis que j'ai trouvé a été de modéliser et de simuler une plate-forme mobile très simplifiée, et d'adapter un modèle de drone existant et très utilisé dans la recherche à nos usages. En pratique, le développement de l'environnement de simulation a fait intervenir trois étapes : la conception de la plate-forme (Section 1.2), l'intégration de capteurs (Section 1.3) et la simulation du drone (Section 1.4). Ces sections n'ont pas la prétention de donner une liste exhaustive des fonctionnalités de ROS et de Gazebo, mais plutôt de faire comprendre leur intérêt et leur cadre d'application à celui qui n'a jamais utilisé ces logiciels. Notez qu'il existe deux versions différentes de ROS, ROS1 et ROS2. Aussi, on trouve plusieurs distributions pour chacune de ces versions. S'il existe de grandes différences d'implémentation selon les versions et les distributions, les concepts essentiels de ROS restent eux inchangés. A titre informatif, nous avons travaillé à partir de la distribution *noetic* de ROS1, mais ce texte a été écrit pour être le plus général possible et valable quelles que soient les versions et distributions choisies. La grande majorité des informations données dans la suite ont été acquises par la lecture des livres *Programming robots with ROS* de Quigley Morgan [40] et *Mastering ROS for Robotics Programming : Design, Build and Simulate Complex Robots Using the Robot Operating System* de Joseph Lentin et Jonathan Cacace [24], et des tutoriels officiels des sites de Gazebo [15] et ROS [50].

## 1.2 Modélisation et commande de la plate-forme mobile

### 1.2.1 Description du système

Lorsque l'on veut simuler un système mécanique, la première étape est de le modéliser ou de le décrire. La description d'un système prend la forme d'un ensemble de solides reliés entre-eux par des liaisons. Dans notre cas, nous cherchons à décrire une plate-forme pouvant se mouvoir selon les six degrés de liberté de la manière la plus simple possible. Pour ce faire, nous allons faire en sorte de commander directement les paramètres de la translation  $\mathbf{t}$  et de la rotation  $\mathbf{R}$  de la plate-forme par rapport au monde (Figure 1.1). La

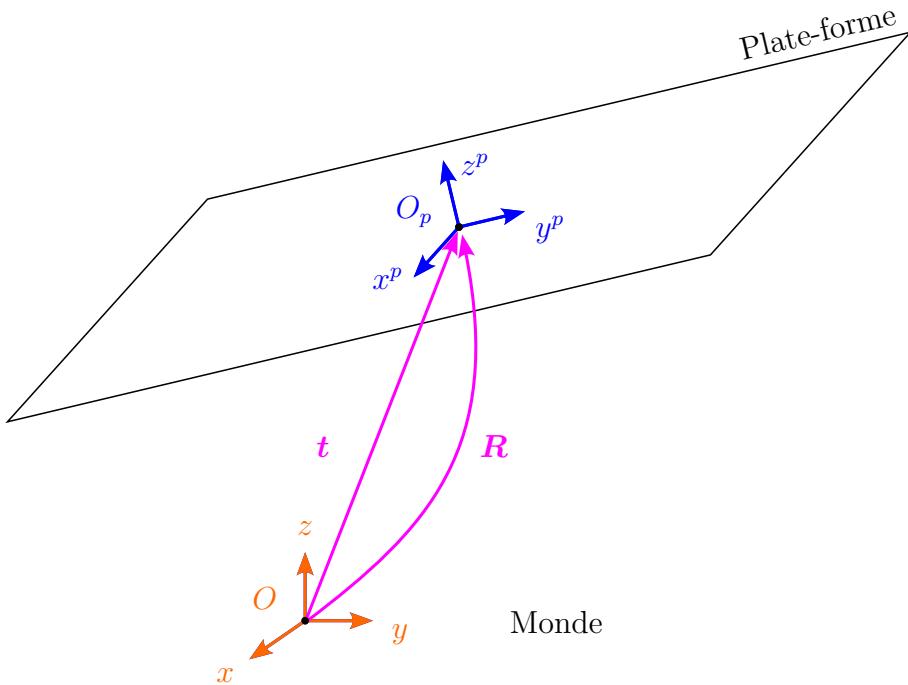


FIGURE 1.1 – Mouvements la plate-forme dans l'espace

description que nous avons adoptée est représentée sur la Figure 1.2. Celle-ci comprend trois liaisons glissières pour la translation de la plate-forme dans l'espace et trois liaisons pivots successives pour son orientation. L'ordre et le choix des pivots ont leur importance ; ils sont déduits du paramétrage de la rotation choisie, en l'occurrence ici par les angles d'Euler selon la convention ZYX. Autrement dit, la rotation de la plate-forme peut être décrite selon trois rotations élémentaires :

1. Une rotation autour de l'axe  $Z$  de la base initiale ;
2. Une rotation autour de l'axe  $Y$  de la base obtenue après la première rotation ;
3. Une rotation autour de l'axe  $X$  de la base obtenue après la seconde rotation.

Dans la marine et l'aviation, on parle couramment d'angle de lacet (*yaw* en anglais) pour la première rotation, d'angle de tangage (*pitch* en anglais) pour la seconde et d'angle de roulis (*roll* en anglais) pour la troisième.

Lorsque l'on travaille avec Gazebo et ROS, cette description doit être inscrite dans des fichiers XML. Pour Gazebo, le format standard de modélisation est le **sdf**, tandis

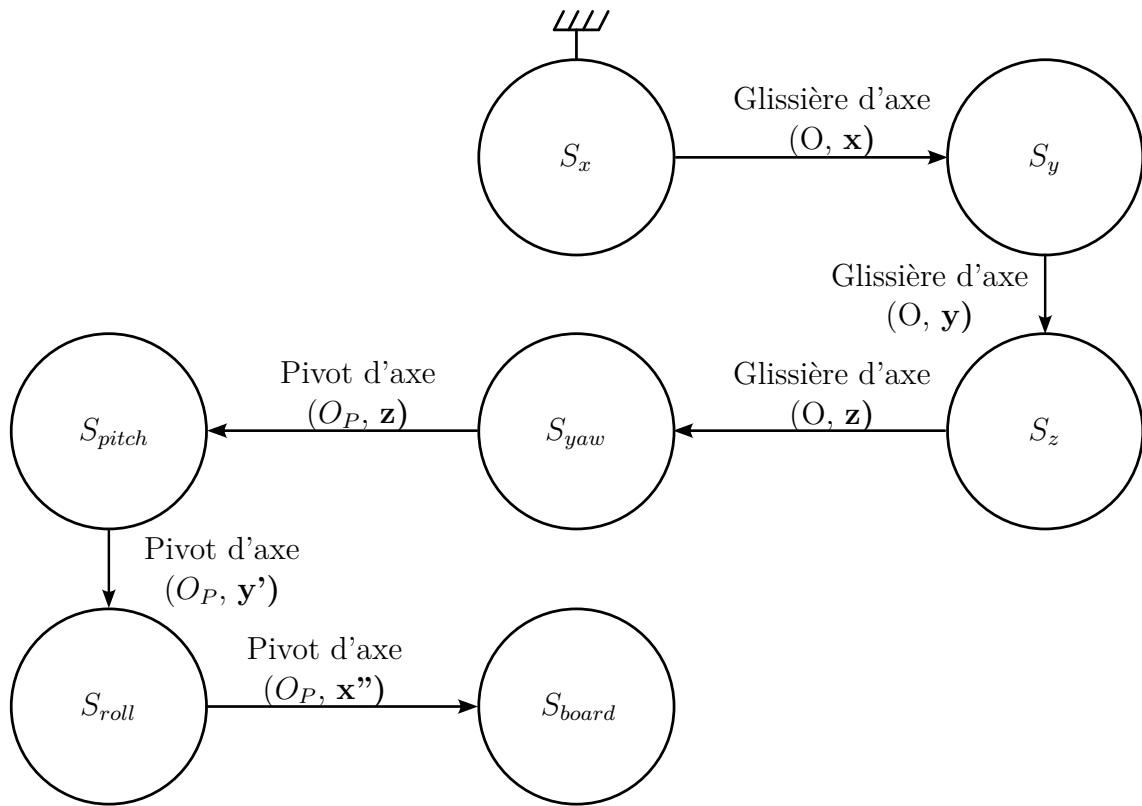


FIGURE 1.2 – Graphe des liaisons de la plate-forme

que pour ROS il s'agit de l'`urdf`. En pratique, ces deux formats sont très proches ; il est ais  de passer d'un format   l'autre, et donc pas n cessaire d' crire deux descriptions diff rentes pour notre plate-forme. Cependant, il est important de noter qu'ils n'offrent pas tous les deux les m mes possibilit s. Le format `urdf` est plus limit  que le format `sdf` ; il n'est par exemple pas possible de mod liser un syst me dont le graphe des liaisons contient des cycles en `urdf`. Pour contrebalancer ces  carts, il est possible d'int grer dans la description `urdf` d'un robot des extraits de description `sdf`, ce sont ce qu'on appelle des *plugins* Gazebo. Enfin, il est commun de remplacer le fichier de description `urdf` par un fichier `xacro`. Le format `xacro` reprend le lexique et la syntaxe du format `urdf` mais permet en plus de cr er des macros et de d finir des propri t s pour simplifier l' criture et la lecture de la description du robot. Les conversions entre ces diff rents formats sont r sum es dans la Figure 1.3.

Au-del  de la structure de la plate-forme que nous avons donn e (Figure 1.2), il s'agit d'indiquer dans ces fichiers de description la g om trie et les propri t s m caniques des solides et des articulations. Si l'on cherche   atteindre un certain niveau de d tails, les pi ces peuvent  tre mod lis es et leurs propri t s m caniques calcul es dans un logiciel de CAO (Conception Assist e par Ordinateur). Toujours dans l'optique d'aller au plus simple, la mod lisation de notre plate-forme n'ayant pas vocation  tre r aliste, nous nous sommes content s d'assembler des volumes sommaires (parall lep p des rectangles, sph res, ...) directement d finis dans le fichier de description. Aussi, nous avons rendu les solides interm diaires invisibles, afin de n'avoir visuellement qu'une plaque flottant dans les airs (Figure 1.4).

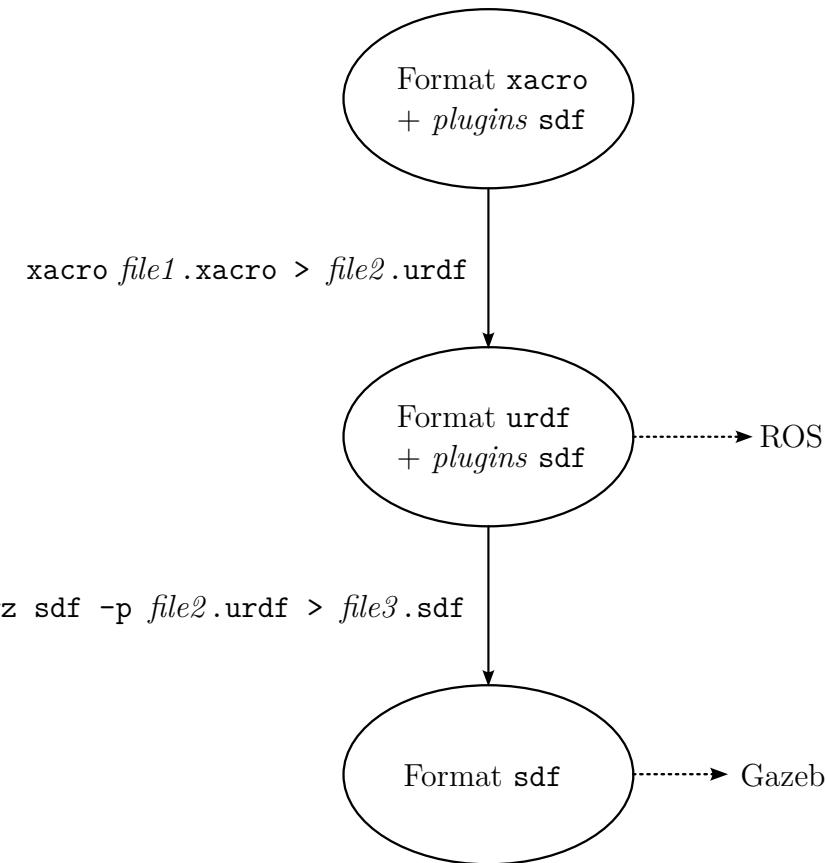


FIGURE 1.3 – Conversion des formats de description des robots

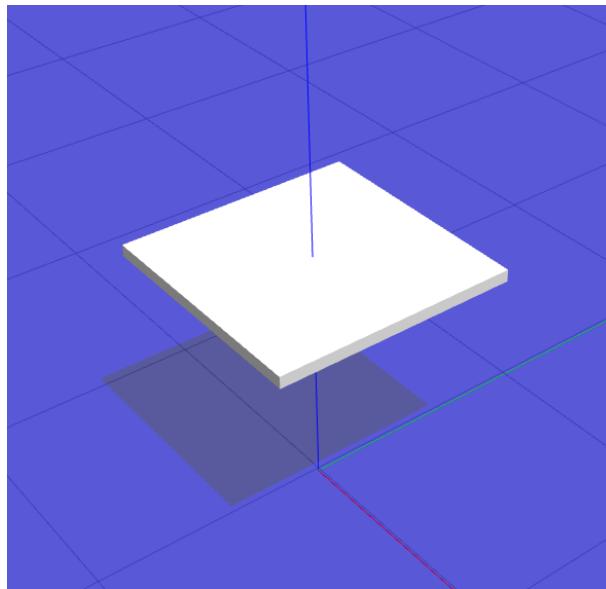


FIGURE 1.4 – Visualisation dans Gazebo de la plate-forme

### 1.2.2 Motorisation et contrôle de la plate-forme

A ce stade de la modélisation, rien ne retient la plate-forme de s'écrouler sur le sol sous l'action de la gravité. Ainsi, la deuxième étape est d'équiper la plate-forme d'actionneurs et de les commander individuellement pour la positionner et l'orienter dans l'espace. Les actionneurs sont indiqués par des balises dans le fichier de description ; ce qui est plus

complexe est la mise en place de la liaison avec ROS pour pouvoir les commander. Pour cela, nous avons utilisé un *package* ROS, nommé `ros_control`. Avant de continuer, il est nécessaire de comprendre grossièrement comment ROS fonctionne et comment est assurée la communication avec Gazebo.

L'idée première de ROS est de créer une architecture logicielle très modulaire pour commander un robot. Au lieu de n'avoir qu'un seul programme qui intègre l'ensemble des fonctionnalités du robot, plusieurs tournent en parallèle et s'échangent des données. Les différents programmes s'appellent des noeuds et les informations qui transitent entre eux sont des messages. Il existe trois différents types d'interfaces à travers lesquelles les messages peuvent être échangés :

**Les topics** sont des canaux sur lesquels des messages circulent en continu. A tout instant, un noeud peut publier de nouveaux messages sur un *topic*, ou s'abonner à un *topic* pour lire les messages qui y circulent. C'est l'interface classiquement utilisée pour échanger des mesures de capteurs ;

**Les services** sont des fonctions qui réalisent des tâches courtes. A tout instant, un noeud peut faire appel à un service et recevra une réponse en retour ;

**Les actions** sont des fonctions qui réalisent des tâches plus longues, comme le déplacement d'un robot jusqu'à une position cible. A tout instant, un noeud peut demander qu'une action soit réalisée et suivra l'évolution de son état durant son exécution.

Les messages échangés sur ces interfaces suivent des conventions précises : leurs types sont normalisés par ROS et leur utilisation doit avoir du sens (il ne faut par exemple pas stocker les mesures d'un LIDAR dans un message de type `Image`, mais dans un message de type `LaserScan`, bien qu'ils soient tous deux implémentés par des structures de données similaires). Derrière ces règles, il y a le concept de modularité logicielle des robots. En d'autres termes, le comportement global d'un robot peut être décomposé en un ensemble de sous-comportements qui, pour certains, ne sont pas spécifiques au robot étudié et dont l'implémentation peut être reprise de robots existants ou remplacée par un code qui réalise la même tâche de façon différente. A un niveau supérieur, les *packages* regroupent des noeuds, des définitions de nouveaux messages, de services et d'actions, des descriptions de robots, ...

Le *package* `ros_control` que nous venons d'évoquer intègre des correcteurs sous forme de noeuds qui, à partir d'une position de consigne et de la position courante d'une liaison, génèrent une commande à appliquer à l'actionneur concerné pour réduire l'erreur<sup>1</sup>. La position courante d'une liaison est extraite de la simulation en temps réel par un *plugin* Gazebo nommé `joint_state_publisher`, qui la publie sur le *topic* `joint_states` et les noeuds s'abonnent à un topic `command` sur lequel la consigne est précisée. Par défaut, les correcteurs sont des PID (Proportionnel - Intégral - Dérivé), que nous devons régler pour obtenir le comportement escompté. Pour nous aider dans cette tâche, ROS met des outils à notre disposition<sup>2</sup>.

---

1. Il est également possible d'asservir les articulations en vitesse ou en effort.

2. Le plus puissant d'entre eux est la reconfiguration dynamique de l'application `rqt`, qui permet de voir les effets qu'a une modification des paramètres du correcteur sur le système sans avoir à relancer la simulation.

## Création de nos premiers nœuds ROS

Nous pouvons désormais envoyer manuellement des consignes en position aux différents actionneurs de la plate-forme depuis la ligne de commande. Cependant, notre objectif était de pouvoir reproduire des mouvements complexes, comme ceux du pont d'un navire dans la mer. Pour y arriver, il faut être en mesure de publier une séquence de positions de la plate-forme. La solution est d'écrire un nœud qui publie à la bonne fréquence les consignes en position sur le *topic command* associé à chaque actionneur. Pour nos essais, nous nous sommes contentés d'envoyer des consignes en position sinusoïdales. La représentation "boîte noire" du nœud est donnée Figure 1.5. Nous utiliserons cette représentation à la mention de chaque nouveau nœud, afin d'orienter le lecteur qui s'intéresse au code source du projet.

### ROS Node : sinusoid\_pos

#### Published topics :

```
mobile_platform/roll_joint_position_controller/command
mobile_platform/pitch_joint_position_controller/command
mobile_platform/yaw_joint_position_controller/command
mobile_platform/tx_joint_position_controller/command
mobile_platform/ty_joint_position_controller/command
mobile_platform/tz_joint_position_controller/command
```

FIGURE 1.5 – Représentation "boîte noire" du nœud de commande de la plate-forme

Aussi, nous avons implémenté une fonction de remise à zéro des positions des actionneurs de la plate-forme sous la forme d'un service, au sein d'un nouveau nœud (Figure 1.6). Ce nœud ne comprend que la partie *serveur* du service, c'est à dire qu'il le définit mais ne l'appelle pas. Une fois défini, il suffit de l'appeler avec une commande ou d'ajouter une partie *client* à tout noeud qui souhaite profiter du service. Cet utilitaire nous évite d'avoir à stopper la simulation pour rétablir l'état initial de la plate-forme.

### ROS Node : joints\_reset

#### Published topics :

```
mobile_platform/roll_joint_position_controller/command
mobile_platform/pitch_joint_position_controller/command
mobile_platform/yaw_joint_position_controller/command
mobile_platform/tx_joint_position_controller/command
mobile_platform/ty_joint_position_controller/command
mobile_platform/tz_joint_position_controller/command
```

#### Service server :

```
reset_joints
```

FIGURE 1.6 – Représentation "boîte noire" du nœud de réinitialisation de la plate-forme

### 1.3 Ajout de capteurs : le cas de la caméra

Les capteurs jouent un rôle important dans en robotique mobile, ce sont eux qui permettent au robot de connaître son environnement (informations extéroceptives) mais aussi d'acquérir des données internes (informations proprioceptives) [12]. Il est essentiel de les simuler correctement. Le fonctionnement d'un capteur en simulation est codé dans un *plugin* Gazebo. Les principaux capteurs utilisés en robotique mobile (caméra, LIDAR, centrale inertielle, ...) disposent déjà de *plugins* génériques qui codent leur comportement mais il est aussi possible d'ajouter nos propres capteurs à partir de modèles que Gazebo met à notre disposition.

Le premier capteur que nous avons cherché à intégrer à notre simulation est une caméra, fixe, positionnée de façon à observer la plate-forme de dessus (Figure 1.7) ; cette configuration nous permet déjà de traiter la partie du sujet relative à l'extraction de connaissances de l'environnement par vision (Chapitre 2). Pour simuler un modèle spé-

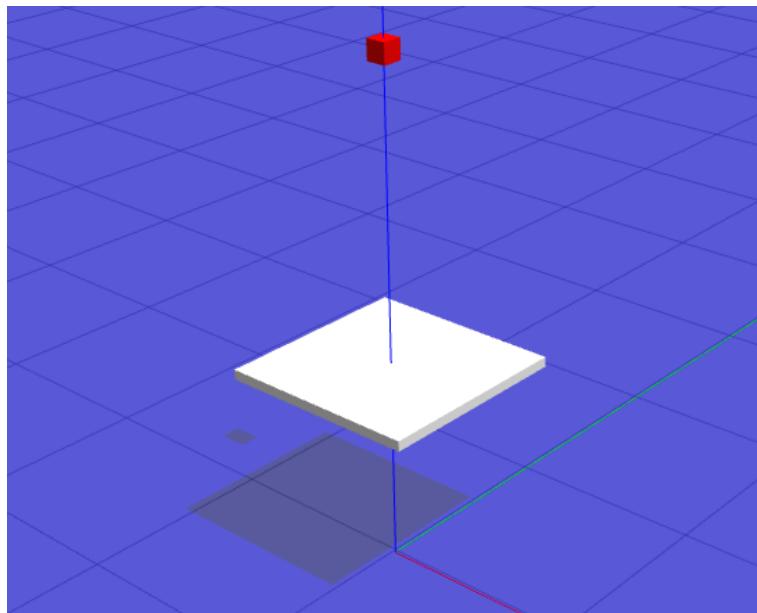


FIGURE 1.7 – Visualisation dans Gazebo de la plate-forme et la caméra (modélisée par un cube rouge)

cifique de caméra, il est nécessaire de renseigner des paramètres (distance focale, résolution de l'image, coefficients de distorsion, ...). Comme nous ne disposons pas encore de caméra, ces paramètres ont été choisis arbitrairement, tout en respectant les ordres de grandeur des paramètres des caméras de type *webcam*. A l'ajout de la caméra dans la simulation, plusieurs nouveaux *topics* sont créés, mais seuls certains d'entre eux nous seront utiles :

**camera\_info** contient les paramètres de calibration interne de la caméra et ses coefficients de distorsion ;

**image\_raw** contient les images générées par la caméra ;

**image\_raw\_depth** contient les images de profondeur de la scène. L'obtention de ce *topic* nécessite l'utilisation d'une caméra RGB-D<sup>3</sup>.

3. En plus de l'image en couleur de la scène, la caméra RGB-D fournit la distance à la caméra de chaque point visible sur l'image. Si cette mesure n'est pas indispensable pour traiter le problème de l'appontage,

Au-delà de la caméra, d'autres capteurs peuvent être intégrés à la simulation à des fins purement pratiques, à l'instar d'une centrale inertielle sur la plate-forme, qui estime en temps réel son orientation par rapport au repère du monde. En réalité, il existe des méthodes plus adaptées pour extraire de la simulation les transformations exactes entre différents repères.

Enfin, je souhaite mentionner l'outil `rosbag` que ROS fournit. Celui-ci permet de créer des enregistrements de plusieurs *topics* pendant la simulation, stockés dans des fichiers `bag`. Ces enregistrements peuvent ensuite être rejoués sans avoir à lancer la simulation. Travailler de cette manière permet de libérer des ressources de calcul<sup>4</sup>, de tester plusieurs algorithmes sur les mêmes données, d'utiliser les mesures de capteurs à des fins d'apprentissage, ... Cette solution a été largement utilisée dans ce projet durant la phase de développement des algorithmes.

## 1.4 Simulation du drone

La simulation du drone nous a intéressée dans un second temps, une fois les algorithmes d'estimation de pose (Chapitre 2) effectifs. L'objectif était d'ajouter un drone équipé de capteurs (IMU, caméra, GPS, ...) à l'environnement Gazebo qui puisse être commandé en vitesses depuis des noeuds ROS. Le modèle n'ayant pas grande importance, nous nous sommes orientés vers le drone *Iris* (Figure 1.8). Ce dernier, produit par l'entre-



FIGURE 1.8 – Modèle de drone *Iris*

prise américaine 3D Robotics, connaît un grand succès dans le monde de la recherche pour son faible coût, la suite complète de capteurs qu'il intègre et la facilité de remplacement de ses pièces.

Dans la réalité comme dans la simulation, la mission d'apportage fait intervenir trois niveaux de contrôle du drone :

- Au niveau le plus haut, la consigne d'apponter est envoyée au drone, et le drone doit apponter ;
- A un niveau intermédiaire, des consignes en vitesses (vitesses linéaires et vitesse de lacet) sont envoyées au drone, et le drone doit atteindre ces vitesses ;

---

elle s'avère être une source d'information complémentaire intéressante pour accroître la robustesse et la stabilité de certaines méthodes.

4. Dans cette optique, il est aussi possible d'utiliser uniquement la partie *server* de Gazebo, sans la partie *client* (l'interface graphique utilisateur du logiciel).

— Au niveau le plus bas, des consignes en vitesses de rotation sont envoyées aux moteurs du drone, et les moteurs doivent atteindre ces vitesses.

Ainsi, même si notre étude ne s'intéresse qu'au niveau le plus haut, pour faire apponter le drone il faut qu'il soit asservi en vitesse, et pour être asservi en vitesse, il faut que les moteurs soient asservis en vitesse de rotation. Il est donc primordial de simuler correctement les deux autres niveaux de contrôle pour mener à bien nos essais. Heureusement pour nous, il existe des solutions logicielles et matérielles pour réaliser ce travail à notre place : les contrôleurs de vol. Tous les composants sont reliés au contrôleur de vol, c'est en quelque sorte la carte mère du drone. Il recueille les mesures des capteurs et les consignes envoyées par l'utilisateur, puis calcule et envoie les commandes aux actionneurs. Pour fonctionner, il doit contenir un programme, à l'instar d'[Ardupilot](#) ou de [PX4](#). Nous avons opté pour Ardupilot pour la grande communauté qui soutient son développement.

Nous allons devoir simuler numériquement le fonctionnement d'Ardupilot, mais aussi de toute la partie matérielle du drone (contrôleur de vol, capteurs, actionneurs, *Electronic Speed Controller*, ...) : on parle de simulation SITL, ou *Software-in-the-Loop*. En pratique, la simulation SITL consiste en un script Python en téléchargement libre, compatibles avec de nombreux systèmes, dont le drone *Iris*.

Dès lors, il est possible de simuler le drone en utilisant uniquement ce script, indépendamment de ROS et Gazebo, et de lui envoyer des commandes à l'aide du protocole de communication [MAVLink](#). Ce protocole, couramment utilisé pour communiquer avec les drones, passe par l'envoie de messages définis au format XML. Cependant, ces messages étant longs à écrire à la main, on préfère utiliser une station de contrôle au sol, ou *Ground Control Station*, pour faciliter l'envoie de commandes aux drones. Les stations de contrôle au sol proposent généralement de nombreuses fonctionnalités, comme la planification de missions, la visualisation du drone sur une carte ou la définition d'une limite virtuelle que le drone ne peut pas franchir. A cet effet, nous avons fait le choix d'utiliser [MAVProxy](#) pour sa simplicité d'utilisation en ligne de commande, sa compatibilité avec tout système autonome qui utilise le protocole MAVLink et l'interface minimaliste qu'il présente<sup>5</sup>. A ce stade, nous pouvons piloter le drone à partir de commandes simples prescrites par MAVProxy, et le voir évoluer sur une carte en deux dimensions (Figure 1.9).

Maintenant que le drone est simulé et que l'on peut aisément lui envoyer des commandes, la seconde étape est d'inclure le drone en question dans notre environnement de simulation Gazebo. Pour cela, comme pour chaque système que l'on cherche à simuler, il faut écrire un fichier de description du drone. Cette fois, contrairement à ce que nous avons fait pour la plate-forme, nous importons les pièces du drone modélisées dans un logiciel de CAO et disponibles en *open source* dans le monde Gazebo et les relierons entre-elles de manière à reconstituer le drone<sup>6</sup>. Ensuite, il faut établir la connexion entre le drone inerte que nous venons d'intégrer à Gazebo et la simulation SITL. Une fois la connexion faite, nous pouvons comme avant envoyer des messages MAVLink au drone ou utiliser MAVProxy pour le mettre en mouvement, mais cette fois ses déplacements sont aussi reproduits dans Gazebo (Figure 1.10).

Finalement, il ne nous manque plus qu'à pouvoir contrôler le drone avec ROS. La solution consiste à installer le *package* `mavros` qui comprend des noeuds qui ont la capacité de lire et d'écrire des messages MAVLink. Ces derniers publient sur des *topics* la position,

5. Pour des fonctionnalités plus évoluées, il peut être intéressant de se diriger vers des stations de contrôle telles que [QGroundControl](#).

6. En réalité, ce travail avait déjà été fait pour nous, mais le principe reste le même pour tout système complexe que l'on veut simuler dans Gazebo.

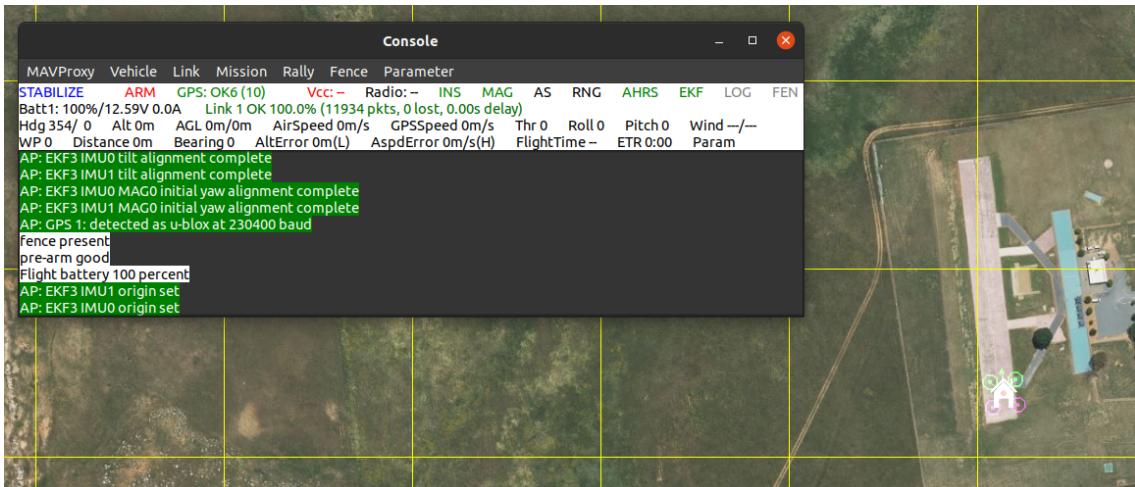


FIGURE 1.9 – Aperçu de la console de la station de contrôle au sol MAVProxy et de la carte sur laquelle évolue le drone

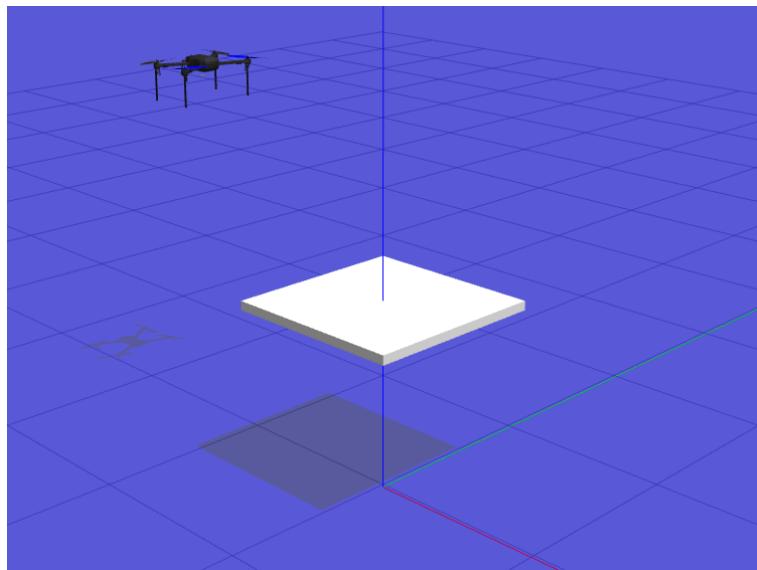


FIGURE 1.10 – Intégration du drone *Iris* à l'environnement de simulation

l'orientation ou la vitesse du drone, ainsi que les données des capteurs, et définissent des services ou actions qui permettent d'envoyer des consignes au drone. A partir de ces fonctions, des nœuds utilitaires ont été mis en place pour faire décoller et atterrir le drone.

Une vidéo de l'environnement de simulation est accessible [ici](#).

Ce premier chapitre nous a permis de nous familiariser avec la simulation robotique et de présenter deux outils centraux pour le développement de solutions robotiques, ROS et Gazebo. Néanmoins, nous n'avons couvert jusqu'à maintenant qu'une petite partie des fonctionnalités de ROS et de Gazebo utilisées au cours de ce stage. Afin de ne pas surcharger ce chapitre, il me semble être préférable d'introduire les nouveaux concepts liés à la simulation que nous allons découvrir au fil de la discussion. Pour les démarquer du reste, j'utiliserai des encadrés bleus comme celui qui suit.

Cet encadré apporte des précisions concernant l'implémentation logicielle ou les outils de développement utilisés. La lecture de son contenu n'est pas nécessaire à la compréhension des solutions développées mais sert de point de repère à celui qui s'intéresse au code source du projet.

# Chapitre 2

## Le *template tracking* pour l'estimation de *pose* par vision de la plate-forme

### 2.1 Le problème de l'estimation de *pose*

Maintenant que nous disposons d'un environnement de simulation opérationnel, il est temps de s'intéresser à la stratégie d'apportage du drone. Mais avant de donner des commandes au drone, il nous faut déjà extraire de l'environnement un certain nombre d'informations. En effet, si les capteurs embarqués par le drone nous permettent de calculer en temps réel et avec précision sa position ou sa vitesse, il nous faut également trouver un moyen de d'obtenir de telles données sur la plate-forme mobile. Il serait envisageable d'équiper la plate-forme de capteurs et de communiquer leurs mesures au drone en temps réel, mais la mise en place d'un place d'un système de la sorte est complexe, onéreuse et suppose surtout que l'on puisse modifier la plate-forme. Ici, nous nous plaçons dans un cadre plus général, où seule une image de la face supérieure de la plate-forme est connue à l'avance<sup>1</sup>. Pour composer avec ce type de situations, la caméra est couramment employée, car elle délivre à moindre coût de riches informations sur la scène. C'est pourquoi nous l'utiliserons comme source première de connaissances sur la plate-forme. Les images brutes générées par la caméra n'étant pas directement exploitables, il nous faudra d'abord expliciter le passage du monde réel au plan image, et calculer à partir de ces dernières des informations utiles à l'apportage automatique.

#### 2.1.1 Contexte théorique

##### Formation géométrique des images

La première étape est de faire le lien entre le monde réel en trois dimensions et l'image en deux dimensions que nous en avons grâce à la caméra. Cela renvoie à la question de la formation des images sur le capteur photographique, qui dépend directement du modèle de caméra considéré. Le modèle le plus simple et le plus utilisé est le sténopé, ou *pinhole camera*, qui suppose que le centre optique de la caméra est situé à l'ouverture, *ie* là où les rayons se croisent. Mathématiquement, le modèle sténopé consiste en une projection

---

1. Cette hypothèse est cohérente avec l'application sur laquelle se fonde ce projet : l'apportage automatique d'un hélicoptère sur un bateau militaire.

perspective du monde observé sur un plan. C'est une approximation réaliste pour les caméras standards. La Figure 2.1 schématisé le modèle de caméra sténopé, associé à la

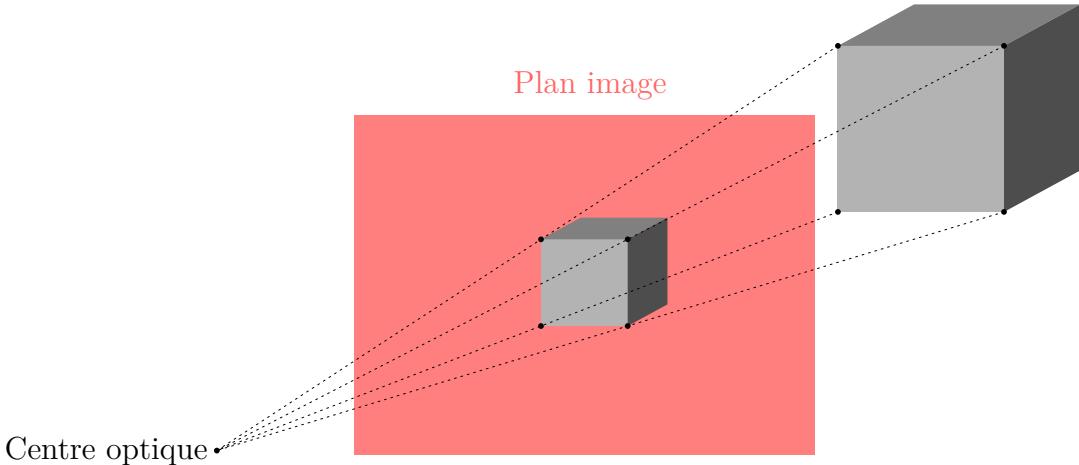


FIGURE 2.1 – Illustration du modèle de caméra sténopé

projection perspective :

Tous les rayons qui passent par un point de l'espace 3D et par sa projection 2D dans le plan image se coupent en un unique point, le centre optique de la caméra.

C'est le modèle que nous utiliserons dans toute la suite de l'étude. Profitons-en pour introduire des notations (Figure 2.2) :

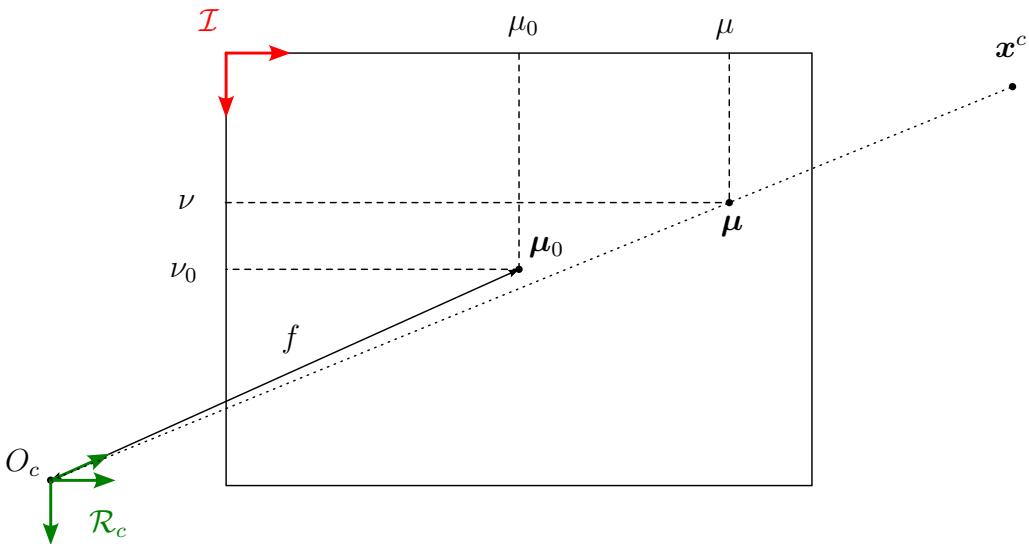


FIGURE 2.2 – Modèle de caméra sténopé

- $\mathbf{x}^c = (x^c, y^c, z^c)^T$  est un point de l'espace exprimé dans le repère  $\mathcal{R}_c$  de la caméra ;
- $\boldsymbol{\mu} = (\mu, \nu)^T$  désigne la projection de  $\mathbf{x}^c$  dans l'image  $\mathcal{I}$  ;
- $\boldsymbol{\mu}_0 = (\mu_0, \nu_0)^T$  est le centre de l'image ;
- $f$  est la distance focale de la caméra <sup>2</sup>.

2. La focale est la distance séparant la surface du capteur de la caméra du foyer image. Notez qu'en

Il est ais  de calculer la projection perspective  $\mathbf{p}^c = (p_x^c, p_y^c, p_z^c)^T$  de  $\mathbf{x}^c$  sur l'image  $\mathcal{I}$    l'aide du th or me de Thal s :

$$\begin{cases} p_x^c = f \frac{x^c}{z^c} \\ p_y^c = f \frac{y^c}{z^c} \\ p_z^c = f \end{cases}$$

Enfin, on retrouve  $\boldsymbol{\mu}$  en exprimant  $p_x^c$  et  $p_y^c$  en pixels et en nous servant de  $\boldsymbol{\mu}_0$  pour tenir du compte du fait que l'origine de l'image se situe non pas en son centre mais en son coin sup rieur gauche. Pour cela, nous devons d'abord d finir :

- $k_\mu$  la densit  de pixels de l'image selon  $\mu$  (en pixels/mm), telle que  $\frac{1}{k_\mu}$  est la longueur (en mm) d'un pixel selon  $\mu$ ;
- $k_\nu$  la densit  de pixels de l'image selon  $\nu$  (en pixels/mm), telle que  $\frac{1}{k_\nu}$  est la longueur (en mm) d'un pixel selon  $\nu$ .

D'o ,

$$\begin{cases} \mu = \mu_0 + k_\mu p_x^c \\ \nu = \nu_0 + k_\nu p_y^c \end{cases}$$

On pose

$$\begin{cases} \alpha_\mu = f k_\mu \\ \alpha_\nu = f k_\nu \end{cases}$$

les distances focales exprim es en pixels. En supposant les pixels carr s, on note :

$$\alpha_\mu = \alpha_\nu = \alpha$$

Ainsi,

$$\begin{cases} \mu = \mu_0 + \alpha \frac{x^c}{z^c} \\ \nu = \nu_0 + \alpha \frac{y^c}{z^c} \end{cases} \quad (2.1)$$

Il est courant d'utiliser les coordonn es homog nes pour ´crire la projection sur le plan image sous forme matricielle :

$$\tilde{\boldsymbol{\mu}} = \mathbf{K} \mathbf{x}^c$$

avec  $\tilde{\boldsymbol{\mu}}$  l'expression de  $\boldsymbol{\mu}$  en coordonn es homog nes, c'est   dire telle que

$$\tilde{\boldsymbol{\mu}} \propto \begin{bmatrix} \mu \\ \nu \\ 1 \end{bmatrix}$$

et

$$\mathbf{K} = \begin{bmatrix} \alpha & 0 & \mu_0 \\ 0 & \alpha & \nu_0 \\ 0 & 0 & 1 \end{bmatrix}$$

Cette matrice  $\mathbf{K}$  est appell e "matrice de calibration interne", ou simplement "matrice de calibration".

---

réalit  le plan image est situ  de l'autre c t  du centre de la cam ra et que l'image appara t renvers e. Ici, nous repr sentons de fa on ´quivalente l'image droite du m me c t  que l'objet afin d'illustre la d marche.

## La transformation rigide en trois dimensions

Cette section étend la transformation rigide en deux dimensions mentionnée dans l'Annexe B à l'espace 3D. Ce concept nous servira principalement pour réaliser des changements de repères entre les différents repères 3D que nous manipulons - celui de la caméra, celui de la plate-forme, et plus tard, celui du drone. Montrons comment un tel changement de repère est réalisé.

Considérons un point de la plate-forme dont les coordonnées sont  $\mathbf{x}^p = (x^p, y^p, z^p)^T$  dans le repère de la caméra et  $\mathbf{x}^c = (x^c, y^c, z^c)^T$  dans le repère de la plate-forme. Pour passer du repère de la plate-forme au repère de la caméra, on réalise une transformation rigide (aussi appelée Euclidienne), composée d'une rotation et d'une translation :

$$\mathbf{x}^c = \mathbf{R}\mathbf{x}^p + \mathbf{t} \quad (2.2)$$

où  $\mathbf{R} \in \mathcal{M}_3(\mathbb{R})$  est une matrice de rotation et  $\mathbf{t} \in \mathcal{M}_{3 \times 1}(\mathbb{R})$  est un vecteur translation (Figure 2.3).

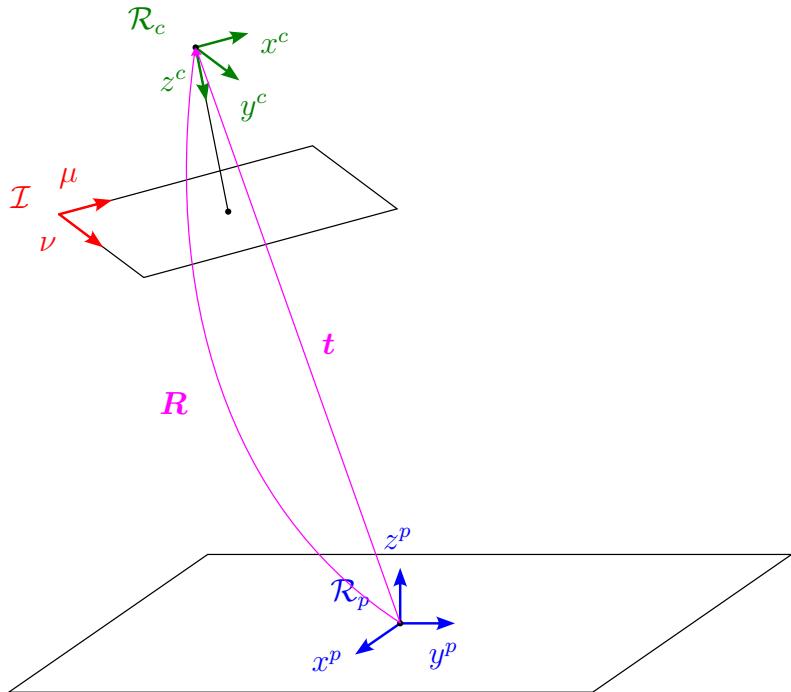


FIGURE 2.3 – Transformation rigide pour passer du repère  $\mathcal{R}_p$  de la plate-forme au repère  $\mathcal{R}_c$  de la caméra

Il est commun d'utiliser les coordonnées homogènes de manière à réécrire la transformation Euclidienne comme une transformation linéaire dans l'espace homogène :

$$\tilde{\mathbf{x}}^c = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tilde{\mathbf{x}}^p \quad (2.3)$$

où  $\tilde{\mathbf{x}}^c$  et  $\tilde{\mathbf{x}}_p$  sont les expressions respectives de  $\mathbf{x}^c$  et  $\mathbf{x}^p$  en coordonnées homogènes. En vision par ordinateur, la quatrième composante des vecteurs homogènes est généralement

prise égale à 1, ce qui permet d'encore alléger l'expression précédente :

$$\boldsymbol{x}^c = [\boldsymbol{R} \quad \boldsymbol{t}] \begin{bmatrix} x^p \\ y^p \\ z^p \\ 1 \end{bmatrix}$$

Cependant, nous continuerons d'utiliser l'expression 2.3, plus largement utilisée pour décrire une transformation rigide.

A ce stade, il est important de noter que la matrice de transformation homogène

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

est paramétrée par six valeurs, trois pour la translation et trois pour la rotation. Si le paramétrage de la translation est trivial, il existe plusieurs manières de paramétriser une rotation dans l'espace 3D. On peut la représenter par une rotation unique autour d'un axe à déterminer ou la décomposer en trois rotations élémentaires successives. C'est cette deuxième option que nous choisissons car elle est plus simple à visualiser. Ensuite, se pose la question du choix des axes des rotations élémentaires ; on peut prendre des axes fixes comme des axes mobiles. Pour ne pas avoir à mélanger plusieurs paramétrages angulaires, nous adoptons le paramétrage par les angles de Cardan selon la convention ZYX, beaucoup utilisé dans l'aviation et la marine sous le nom d'angles de roulis-tangage-lacet<sup>3</sup> :

1. On tourne autour de l'axe  $Z$  de la base initiale ;
2. On tourne autour de l'axe  $Y$  de la base obtenue après la première rotation ;
3. On tourne autour de l'axe  $X$  de la base obtenue après la seconde rotation.

Enfin, ajoutons que nous pouvons facilement inverser la relation 2.3 pour passer du repère de la caméra au repère de la plate-forme :

$$\tilde{\boldsymbol{x}}^p = \boldsymbol{T}^{-1} \tilde{\boldsymbol{x}}^c \quad \text{avec} \quad \boldsymbol{T}^{-1} = \begin{bmatrix} \boldsymbol{R}^T & -\boldsymbol{R}^T \boldsymbol{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

### 2.1.2 Le problème P-n-P

La *pose* d'un objet fait référence à sa position et son orientation dans un repère donné<sup>4</sup>. Elle est décrite par trois composantes dans le plan (deux translations et une rotation) et six dans l'espace (trois translations et trois rotations). Le problème de l'estimation de *pose* consiste alors à déterminer ces composantes à partir d'un certain nombre d'informations sur la scène issues de différents capteurs (couleurs, textures, données géométriques, mesures de distances, ...). L'estimation de *pose* trouve un grand nombre d'applications dans l'industrie ; elle permet aux robots manipulateurs de saisir des objets sans connaître leur *pose* au préalable<sup>5</sup>, aux voitures autonomes d'éviter les obstacles et sert de socle à la réalité augmenté pour l'intégration précise d'objets virtuels à la réalité.

3.  $X$  pour le roulis,  $Y$  pour le tangage et  $Z$  pour le lacet.

4. En vision par ordinateur, la *pose* est exprimée par rapport au repère de la caméra. Autrement dit, il s'agit de la transformation rigide qui permet de passer du repère de la caméra au repère de l'objet d'intérêt.

5. On parle du problème de *bin picking*.

Dans le cadre de notre étude, la connaissance de la position et de l'orientation de la plate-forme est d'intérêt pour la synthèse de stratégies de contrôle du drone. La *pose* est l'information la plus riche que nous pouvons avoir sur la plate-forme à un instant donné ; il est possible d'en déduire sa vitesse et son accélération par dérivation et de faire des prédictions sur son état futur après avoir réalisé plusieurs calculs de *pose*. Savoir où est positionné et comment est orienté le pont du bateau sont les informations que cherche à obtenir le pilote qui souhaite apponter.

Il existe plusieurs solutions au problème d'estimation de *pose*. Le choix d'une solution doit prendre en compte la spécificité de la situation à laquelle nous faisons face ; des questions peuvent déjà nous orienter :

- La caméra est-elle calibrée ?
- Quelle est la connaissance de l'objet que l'on a à l'avance (aucune, motif, nuage de points, modèle 3D, ... ) ?
- L'objet a-t-il une nature particulière (objet plan, couleur unique, revêtement réfléchissant, objet déformable, ... ) ?
- La solution doit-elle être robuste aux occlusions, aux changements d'éclairage, ... ?

Ici, nous optons pour une stratégie classique d'estimation de *pose* à partir de paires de points 3D dans le repère de l'objet et de leurs projections 2D dans le plan image avec une caméra calibrée. L'estimation de *pose* dans ces conditions est connue sous le nom de problème P-*n*-P, pour *Perspective-n-Point*. Ainsi, l'estimation de la position et de l'orientation de la plate-forme dans le repère de la caméra fera intervenir plusieurs étapes :

1. Calibrer la caméra ;
2. Déterminer des correspondances 2D-3D ;
3. Résoudre un problème P-*n*-P.

Nous commencerons par présenter une manière de résoudre un problème P-3-P, *i.e.* estimer la *pose* à partir de trois correspondances. Puis, nous verrons comment obtenir de telles correspondances en temps-réel. Le processus de calibration de la caméra ne sera pas détaillé dans cette partie (voir Annexe A pour plus de précisions) car nous ne cherchons pas encore à simuler fidèlement une caméra en notre possession<sup>6</sup>.

## Le problème P-*n*-P

Le problème P-*n*-P désigne le problème d'estimation de la rotation et de la translation entre le repère de l'objet et le repère de la caméra à partir de *n* points 3D dans le repère de l'objet et de leurs projections 2D dans le plan image, lorsque la matrice de calibration interne  $\mathbf{K}$  est connue, *i.e.* que la caméra est calibrée. En réalité, la connaissance de trois correspondances est suffisante pour résoudre le problème, il s'agit d'un problème P-3-P. Néanmoins, le problème P-3-P admet plusieurs solutions (entre deux et quatre) ; une quatrième correspondance est nécessaire pour lever l'ambiguïté. Montrons comment nous pouvons estimer la *pose* d'un objet à partir de quatre correspondances<sup>7</sup> en résolvant plusieurs problèmes P-3-P minimaux.

Soient les quatre correspondances  $(\boldsymbol{\mu}_i, \mathbf{x}_i^p)_{i \in \{1,2,3,4\}}$ . La résolution du problème d'estimation de *pose* se fait en deux étapes :

---

6. C'est à l'utilisateur de renseigner les paramètres de la caméra qu'il veut simuler dans le logiciel de simulation.

7. Dans notre cas, il peut s'agir de quatre points caractéristiques de la plate-forme, comme ses quatre coins. La Section 2.1.3 décrit plusieurs méthodes pour calculer quatre correspondances.

1. Calculer  $\{\mathbf{x}_i^c\}_{i \in \{1,2,3,4\}}$ , les coordonnées des points de la plate-forme dans le repère de la caméra ;
2. Calculer  $\mathbf{R}$  et  $\mathbf{t}$ , déplacement rigide qui permet le passage du repère de la plate-forme au repère de la caméra.

**1. Calcul des coordonnées des points dans le repère de la caméra.** Prenons trois des quatre correspondances,  $(\boldsymbol{\mu}_i, \mathbf{x}_i^p)_{i \in \{1,2,3\}}$ , parmi lesquelles nous en tirons une paire repérée par les indices  $(i, j)_{i \neq j} \in \{1, 2, 3\}^2$ . Il est possible d'établir une relation de contrainte sur les distances de la caméra aux points :

$$d_{ij}^2 = x_i^2 + x_j^2 - 2x_i x_j \cos(\theta_{ij}) \quad (2.4)$$

où

- $d_{ij}$  est la distance entre les deux points de la plate-forme  $\mathbf{x}_i^p$  et  $\mathbf{x}_j^p$  ;
- $\theta_{ij}$  est l'angle entre les deux rayons passant respectivement par  $\mathbf{x}_i^p$  et  $\boldsymbol{\mu}_i$ , et  $\mathbf{x}_j^p$  et  $\boldsymbol{\mu}_j$  ;
- $x_i$  et  $x_j$  sont les distances séparant le centre de la caméra et les points  $\mathbf{x}_i^p$  et  $\mathbf{x}_j^p$ .

La connaissance de la correspondance permet de calculer  $d_{ij}$  :

$$d_{ij} = \|\mathbf{x}_i^p - \mathbf{x}_j^p\|_2$$

et  $\cos(\theta_{ij})$  :

$$\begin{aligned} \cos(\theta_{ij}) &= \frac{\overrightarrow{C\boldsymbol{\mu}_j}^T \overrightarrow{C\boldsymbol{\mu}_i}}{\|\overrightarrow{C\boldsymbol{\mu}_j}\|_2 \|\overrightarrow{C\boldsymbol{\mu}_i}\|_2} \\ &= \frac{\overrightarrow{C\boldsymbol{\mu}_j}^T \overrightarrow{C\boldsymbol{\mu}_i}}{\sqrt{\overrightarrow{C\boldsymbol{\mu}_j}^T \overrightarrow{C\boldsymbol{\mu}_j}} \sqrt{\overrightarrow{C\boldsymbol{\mu}_i}^T \overrightarrow{C\boldsymbol{\mu}_i}}} \\ &= \frac{\boldsymbol{\mu}_j^T (\mathbf{K} \mathbf{K}^T)^{-1} \boldsymbol{\mu}_i}{(\boldsymbol{\mu}_j^T (\mathbf{K} \mathbf{K}^T)^{-1} \boldsymbol{\mu}_j)^{1/2} (\boldsymbol{\mu}_i^T (\mathbf{K} \mathbf{K}^T)^{-1} \boldsymbol{\mu}_i)^{1/2}} \end{aligned}$$

avec  $C$  le centre optique de la caméra et  $\mathbf{K}$  la matrice de calibration interne.

Ainsi, si l'on écrit l'Equation 2.4 pour chacune des trois paires, on obtient un système de trois équations aux trois inconnues  $x_1$ ,  $x_2$  et  $x_3$ . Par substitution, il est possible d'écrire un polynôme de degré 8 en  $x_1$  ne contenant que des termes pairs. On peut poser  $x = x_1^2$  et le résoudre ; on obtient au plus quatre solutions pour  $x$ . Comme  $x_1$  est positif, on a  $x_1 = \sqrt{x}$ . Enfin,  $x_2$  et  $x_3$  sont déterminés de manière unique pour chaque valeur possible de  $x_1$ .

Nous pourrions continuer la résolution et avoir jusqu'à quatre solutions pour la *pose* ; il s'agit du problème P-3-P classique. Cependant un moyen simple de contourner le problème est d'ajouter dès maintenant la quatrième correspondance  $(\boldsymbol{\mu}_4, \mathbf{x}_4^p)$ , de reproduire la démarche précédente pour chacun des quatre sous-ensembles de trois correspondances et de conserver la solution commune pour  $x_1$ ,  $x_2$ ,  $x_3$  et  $x_4$ .

Les coordonnées des points dans le repère de la caméra peuvent maintenant être calculés :

$$\forall i \in \{1, 2, 3, 4\}, \mathbf{x}_i^c = x_i \mathbf{K}^{-1} \boldsymbol{\mu}_i$$

**2. Calcul de la transformation rigide entre le repère de la plate-forme et le repère de la caméra.** A ce stade, nous disposons des coordonnées des quatre points 3D à la fois dans le repère de la plate-forme, et dans le repère de la caméra. Nous cherchons à estimer la transformation rigide, *ie* la matrice de rotation  $\mathbf{R}$  et le vecteur translation  $\mathbf{t}$ , telle que (Equation 2.2) :

$$\forall i \in \{1, 2, 3, 4\}, \quad \mathbf{x}_i^c = \mathbf{R}\mathbf{x}_i^p + \mathbf{t} + \boldsymbol{\varepsilon}_i$$

où  $\boldsymbol{\varepsilon}_i$  est un bruit aléatoire.

Nous allons chercher la solution au sens des moindres carrés :

$$(\hat{\mathbf{R}}, \hat{\mathbf{t}}) = \arg \min_{\mathbf{R} \in SO(3), \mathbf{t}} \left( \sum_i \|\mathbf{R}\mathbf{x}_i^p + \mathbf{t} - \mathbf{x}_i^c\|_2^2 \right)$$

où  $\hat{\mathbf{R}}$  et  $\hat{\mathbf{t}}$  désignent les valeurs approchées de  $\mathbf{R}$  et  $\mathbf{t}$ . On note :

$$e : (\mathbf{R}, \mathbf{t}) \mapsto \sum_i \|\mathbf{R}\mathbf{x}_i^p + \mathbf{t} - \mathbf{x}_i^c\|_2^2$$

l'erreur que l'on cherche à minimiser. Sans détailler les calculs, il vient

$$\mathbf{t} = \bar{\mathbf{x}}^c - \mathbf{R}\bar{\mathbf{x}}^p \tag{2.5}$$

de  $\frac{\partial e}{\partial \mathbf{t}} = 0$ , où  $\bar{\mathbf{x}}^c$  et  $\bar{\mathbf{x}}^p$  sont les moyennes des  $\{\mathbf{x}_i^c\}$  et  $\{\mathbf{x}_i^p\}$ . On substitue  $\mathbf{t}$  dans l'erreur par son expression en fonction de  $\mathbf{R}$  (Equation 2.5), d'où :

$$\hat{\mathbf{R}} = \arg \min_{\mathbf{R} \in SO(3)} \left( \sum_i \|\mathbf{R}\mathbf{y}_i^p - \mathbf{y}_i^c\|_2^2 \right) \tag{2.6}$$

avec

$$\mathbf{y}_i^p = \mathbf{x}_i^p - \bar{\mathbf{x}}^p \quad \text{et} \quad \mathbf{y}_i^c = \mathbf{x}_i^c - \bar{\mathbf{x}}^c$$

Après développement,

$$\|\mathbf{R}\mathbf{y}_i^p - \mathbf{y}_i^c\|_2^2 = (\mathbf{y}_i^p)^T \mathbf{y}_i^p - 2(\mathbf{y}_i^c)^T \mathbf{R}\mathbf{y}_i^p + (\mathbf{y}_i^c)^T \mathbf{y}_i^c$$

où seul le terme  $-2(\mathbf{y}_i^c)^T \mathbf{R}\mathbf{y}_i^p$  dépend de  $\mathbf{R}$ . On peut réécrire le problème d'optimisation 2.6 :

$$\hat{\mathbf{R}} = \arg \max_{\mathbf{R} \in SO(3)} \left( \sum_i (\mathbf{y}_i^c)^T \mathbf{R}\mathbf{y}_i^p \right) \tag{2.7}$$

On peut montrer que

$$\sum_i (\mathbf{y}_i^c)^T \mathbf{R}\mathbf{y}_i^p = \text{tr} \left( \mathbf{R}(\mathbf{Y}^c)^T \mathbf{Y}^p \right)$$

où  $\mathbf{Y}^c$  et  $\mathbf{Y}^p$  sont les matrices dont les colonnes sont les vecteurs  $\{\mathbf{y}_i^c\}$  et  $\{\mathbf{y}_i^p\}$ . En posant  $\mathbf{L} = (\mathbf{Y}^c)^T \mathbf{Y}^p$ , 2.7 devient :

$$\hat{\mathbf{R}} = \arg \max_{\mathbf{R} \in SO(3)} (\text{tr}(\mathbf{R}\mathbf{L}))$$

On prend la décomposition en valeurs singulières de  $\mathbf{L}$  :

$$\mathbf{L} = \mathbf{U}\Sigma\mathbf{V}^T$$

et  $\hat{\mathbf{R}}$  est donnée par :

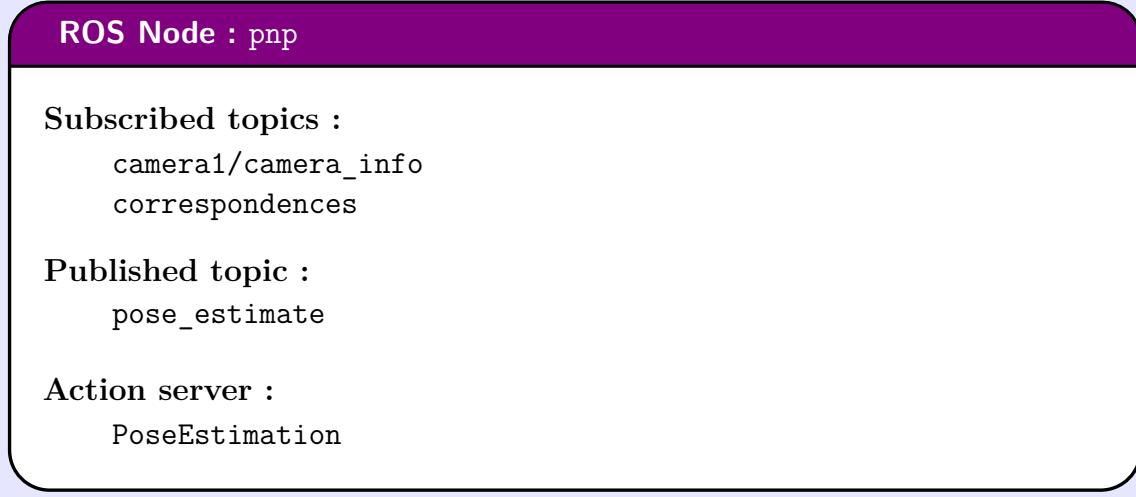
$$\hat{\mathbf{R}} = \mathbf{U}\mathbf{V}^T$$

Il ne reste plus qu'à calculer  $\hat{\mathbf{t}}$  à l'aide de la relation 2.5.

Il existe bien évidemment d'autres algorithmes pour résoudre le problème P-3-P, certains étant plus précis et plus rapides que celui présenté, ainsi que des extensions de ces stratégies lorsque le nombre de correspondances est supérieur à quatre. Par ailleurs, dans ce dernier cas il est intéressant d'utiliser RANSAC (Annexe D) pour écarter les correspondances aberrantes et améliorer la robustesse de l'estimation de *pose*.

En pratique, la plupart des algorithmes efficaces de résolution des problèmes P-n-P et P-3-P sont implémentés dans la méthode `solvePnP` de la librairie OpenCV. Nous indiquerons ceux que nous utiliserons par la suite.

La structure du noeud ROS pour l'estimation de *pose* est donnée ci-dessous.



### 2.1.3 Obtention de correspondances 2D-3D

La question qu'il convient désormais de se poser est :

Comment obtenir les correspondances 2D-3D à partir desquelles estimer la *pose* ?

C'est un problème beaucoup plus ouvert que le précédent, et dont la solution est à construire en fonction de l'application. Dans notre cas, nous nous intéresserons principalement à trois méthodes distinctes :

- Une méthode basée sur la détection et l'identification de marqueurs passifs ArUco (Figure 2.4) ;
- Une méthode basée sur la détection et le *tracking* de points d'intérêt ;
- Une méthode basée sur la détection et le *tracking* de templates.

Pour chacune de ses méthodes, nous supposons connaître à l'avance le visuel de la face supérieure de la plate-forme.

#### Utilisation de marqueurs ArUco

Il existe une large gamme de marqueurs visuels, parmi lesquels on retrouve des motifs simples à identifier (Figure 2.5), les LEDs ou les sphères réfléchissantes. L'ensemble des marqueurs carrés, dont fait partie ArUco, sont très intéressants pour de l'estimation de *pose* car il suffit, lorsque la caméra est calibrée, de connaître la position de leurs quatre

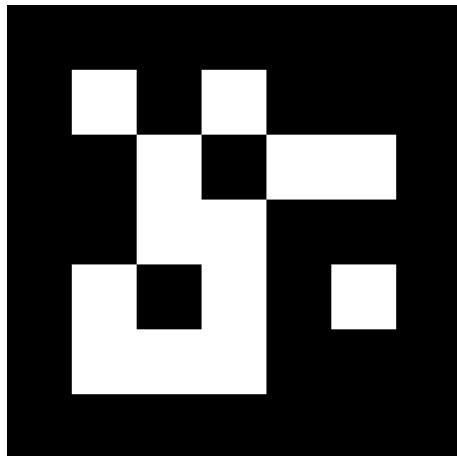


FIGURE 2.4 – Exemple d'un marqueur ArUco (le n°0 d'un dictionnaire  $5 \times 5$ )

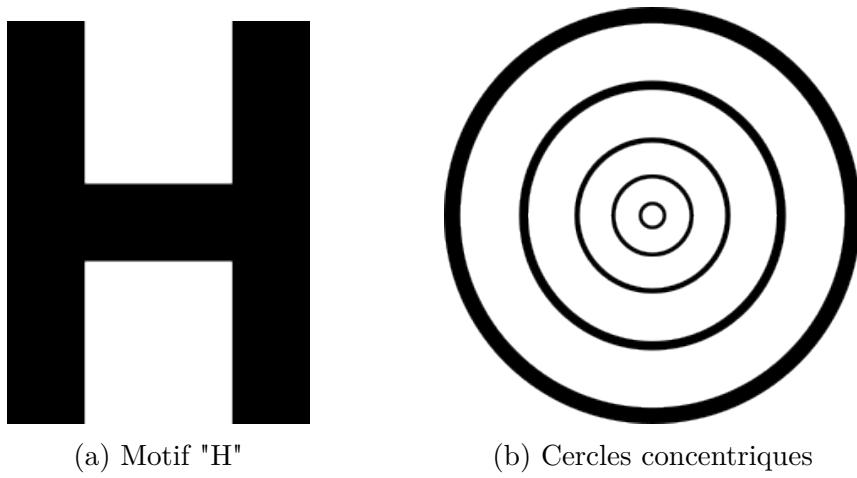


FIGURE 2.5 – Utilisation de marqueurs

coins dans l'image pour y parvenir (Section 2.1.2). Aussi, ils contiennent généralement un code binaire qui permet leur identification.

L'utilisation de marqueurs ArUco est la méthode que l'on retrouve le plus dans la littérature pour traiter les stratégies d'appontage automatique sur une cible en mouvement. En effet, son principe est relativement simple, elle ne requiert que peu de calculs, les algorithmes sont déjà implémentés dans les librairies populaires et sa mise en place demande un moindre effort. De plus, elle a su prouver son efficacité pour l'estimation de *pose* en temps réel et embarqué. Pour ces raisons, elle est l'outil idéal pour tester rapidement des stratégies de contrôle de drone sans avoir à maîtriser les techniques de vision par ordinateur. Néanmoins, dans une situation réaliste d'appontage d'un drone, ou d'un hélicoptère, sur une plate-forme mobile, ou un bateau, dans un contexte militaire, l'utilisation de marqueurs est limitée et il faut bien souvent s'adapter à des systèmes déjà existants.

La capacité du drone d'apponter sur une plate-forme mobile quelconque est un des points clés du stage. C'est pourquoi cette première solution ne constitue qu'une référence par rapport à laquelle nous pourrons comparer nos deux autres approches, en termes de temps de calcul et de précision afin de réaliser une estimation de *pose* en temps réel.

Présentons tout de même les idées qui sous-tendent l'introduction des marqueurs ArUco. Le premier argument concerne la création même des marqueurs. Comme nous

l'avons dit, chaque marqueur contient un identifiant unique déterminé par une grille de cases carrées de couleur noire ou blanche. La taille de cette grille peut varier, on parle de marqueurs  $4 \times 4$ ,  $5 \times 5$ , etc.... Plus la grille est grande, plus les possibilités de marqueurs sont importantes<sup>8</sup>. Pour éviter de confondre des marqueurs très proches entre eux, on sélectionne un sous-ensemble réduit de l'ensemble des marqueurs d'une taille donnée, en veillant à ce qu'ils soient le plus distincts possible les uns des autres. Un tel sous-ensemble est généré par un algorithme qui chercher à maximiser une distance entre marqueurs, puis stocké dans un dictionnaire.

L'ensemble des codes permettant la génération des marqueurs ArUco, leur détection ou leur identification est implémenté dans le module `aruco` d'OpenCV. Ce dernier intègre également des dictionnaires de marqueurs prédéfinis ; on les retrouve [en ligne](#).

Le processus de détection des marqueurs se compose d'une étape de détection des régions candidates dans l'image, puis d'une étape d'identification qui vise à éliminer celles qui ne contiennent pas de marqueur ou qui contiennent un marqueur absent du dictionnaire. Sans entrer dans les détails des algorithmes utilisés, après que l'image a été convertie en nuances de gris :

1. Les contours sont détectés dans l'image ;
2. Ne sont gardés que les contours susceptibles de correspondre aux bords des marqueurs, *i.e.* les contours qui forment des quadrilatères ;
3. Chaque région candidate est divisée en une grille et la présence de bordures noires est testée pour écarter celles qui ne contiennent pas de marqueur ;
4. Le code de chaque marqueur est extrait et comparé à ceux des marqueurs du dictionnaire ;
5. Les positions des coins de chaque marqueur sont prises égales aux intersections de deux de ses arêtes consécutives.

La détection des marqueurs ArUco en simulation a été implémentée dans un nœud ROS en C++. Le C++ introduit des étapes supplémentaires par rapport au développement en Python. Par exemple, il est nécessaire de recompiler le projet à chaque modification du code, il faut également être plus explicite sur les types manipulés et les conversions de types.

L'estimation de *pose* à l'aide d'un marqueur ArUco est sensible aux occlusions : un marqueur ne peut être détecté et identifié uniquement lorsqu'il est entièrement présent dans l'image. Pour pallier ce problème, une solution consiste à utiliser une planche de marqueurs au lieu d'un seul. Ainsi, on peut perdre un marqueur mais toujours être en mesure de trouver des correspondances, et plus il y a de marqueurs, plus il y a de correspondances, et moins l'estimation de *pose* est sensible au bruit.

## 2.2 Le *template tracking*

Le *template matching* consiste à identifier dans une image la région qui correspond à une image de référence, le *template*. Il s'agit d'une alternative à la recherche d'un motif dans l'image avec moins d'hypothèses sur la région d'intérêt. Il existe deux approches principales au problème de *template matching* [27] :

8. Il existe  $2^{n \times n}$  marqueurs différents de taille  $n \times n$ , soit déjà 65 536 possibilités pour la taille  $4 \times 4$ .

**L'approche basée sur l'utilisation de *features*** dont l'idée est de détecter des indices visuels dans l'image et le *template* (les-dites *features*<sup>9</sup>), de les décrire, et de les faire correspondre. Une fois ces correspondances établies, il est possible de calculer la transformation (Annexe B) qui modélise le mieux le passage du *template* à l'image<sup>10</sup>. Cette approche s'appuie sur une description locale de l'image.

**L'approche globale, ou *template-based*** qui cherche directement le *template* dans l'image, souvent en minimisant la différence entre l'intensité des pixels d'une zone de l'image et l'intensité des pixels du *template*. A nouveau, nous cherchons la transformation entre le *template* et l'image.

Ces approches ont chacune leurs avantages et leurs inconvénients vis à vis des défis du *template matching* (occlusions, changement d'éclairage, sensibilité aux différentes transformations, ...). La description locale d'une image par des *features* est moins sensible aux occlusions (seuls quelques points suffisent pour caractériser une image, il n'est généralement pas gênant d'en perdre quelques uns à cause d'une occlusion) et peut facilement être rendue indépendante de l'échelle, de la rotation ou des changements de luminosité de l'image<sup>11</sup>. A l'inverse, considérer le *template* comme un tout permet représenter des motifs plus complexes.

Dans le cadre de notre étude, il est essentiel de pouvoir réaliser les calculs en temps-réel sur une séquence d'images avec une puissance de calcul limitée (les solutions sont susceptibles d'être exécutées par l'ordinateur embarqué du drone). C'est le critère qui constraint le plus notre choix de solutions.

Avec cette idée en tête, il est d'ores et déjà possible de proposer une alternative à la démarche classique du *template matching* que nous avons présentée. Au lieu d'effectuer la détection des *features* ou du *template* dans l'image entière (ce procédé est coûteux), il est commun de les définir uniquement dans la première image et de les suivre dans les suivantes en se servant de leurs positions précédentes. Dans ce cas, nous parlerons plutôt de *template tracking* [45]. Dans cette section, nous proposerons à la fois une approche globale de *template tracking* (*ie* sans l'utilisation de *features*), et une approche basée *features*.

### 2.2.1 *Template tracking* basé *features*

Du fait du nombre élevé de pixels dont elles sont composées, les images sont des structures de données volumineuses. Pour autant, toute l'information qu'elles contiennent n'est pas toujours nécessaires dans le cadre de certaines applications de vision par ordinateur. De ce constat est née l'idée de résumer les images par des *features*, c'est à dire des caractéristiques visuelles, qui leur sont propres et qui suffisent à leur identification.

Il existe de nombreuses *features* différentes que l'on peut trouver dans une image :

- des points caractéristiques ;
- des lignes droites ;
- des superpixels ;

9. Il s'agit en général de points, mais on retrouve également des lignes ou des ellipses, plus robustes dans certains cas.

10. Lorsque le *template* est carré, il est possible de caractériser sa position dans l'image par la position de ses coins. Sans cette hypothèse, on utilise une transformation géométrique à cet effet ; ce problème est aussi connu sous le nom de recalage d'images, ou *image registration*.

11. Les points *SIFT* (*Scale- Invariant Feature Transform*) offrent la description locale la plus complète parmi les solutions existantes [33].

— des ellipses ;

— ...

Nous nous intéresserons uniquement aux points caractéristiques, ou points d'intérêt, dans la suite de l'étude pour leur simplicité et leur généralité [31].

### Tracking de points

La méthode de *tracking* que nous avons utilisée a été introduite par Lucas et Kanade [34]. Elle cherche à suivre un point d'images en images en ne se basant pas seulement sur son intensité mais également sur celle de ses voisins. Cela permet de réduire les erreurs dues au bruit dans l'image. Ce principe peut être étendu pour tracker des fenêtres plus grandes, mais l'on perd le côté local de la description [1].

Considérons une séquence d'images dont on extrait l'image  $\mathcal{I}_k$  obtenue à l'instant  $k$ . Soit  $\boldsymbol{\mu} = (\mu, \nu)^T$  les coordonnées d'un pixel dans cette image, on repère par  $i_k(\boldsymbol{\mu})$  son intensité convertie en niveau de gris. L'idée du *tracking* est de calculer le déplacement  $\delta\boldsymbol{\mu}$  de ce pixel dans l'image suivante  $\mathcal{I}_{k+1}$ . Pour cela, on suppose que son intensité reste constante d'une image à la suivante :

$$i_{k+1}(\boldsymbol{\mu} + \delta\boldsymbol{\mu}) = i_k(\boldsymbol{\mu}) \quad (2.8)$$

On fait également l'hypothèse que le déplacement  $\delta\boldsymbol{\mu}$  est petit<sup>12</sup>. Ainsi, un développement en séries de Taylor au premier ordre nous permet de réaliser l'approximation suivante :

$$i_{k+1}(\boldsymbol{\mu} + \delta\boldsymbol{\mu}) \approx i_{k+1}(\boldsymbol{\mu}) + \nabla i_{k+1}(\boldsymbol{\mu})^T \delta\boldsymbol{\mu} \quad (2.9)$$

où  $\nabla i_{k+1}(\boldsymbol{\mu}) = \left( \frac{\partial i_{k+1}}{\partial \mu}, \frac{\partial i_{k+1}}{\partial \nu} \right)^T$  est issu du calcul des gradients de l'image  $\mathcal{I}_{k+1}$ <sup>13</sup>.

En soustrayant l'Equation 2.8 à l'Equation 2.9, on obtient :

$$i_{k+1}(\boldsymbol{\mu}) - i_k(\boldsymbol{\mu}) + \nabla i_{k+1}(\boldsymbol{\mu})^T \delta\boldsymbol{\mu} \approx 0 \quad (2.10)$$

Cette équation contient deux inconnues,  $\delta\mu$  et  $\delta\nu$ ; il existe une infinité de solutions<sup>14</sup>. Une idée simple pour avoir plus d'équations pour un pixel est de supposer que l'on peut modéliser la transformation locale de l'image par une translation, *ie* que les pixels dans un voisinage du pixel considéré se déplacent de la même quantité [34]. On applique cette fois l'Equation 2.10 sur une fenêtre  $\mathcal{W}$  centrée sur le pixel dont on cherche à estimer le déplacement et on obtient le système suivant :

$$\forall \boldsymbol{\mu}^{\mathcal{W}} \in \mathcal{W}, i_{k+1}(\boldsymbol{\mu}^{\mathcal{W}}) - i_k(\boldsymbol{\mu}^{\mathcal{W}}) + \nabla i_{k+1}(\boldsymbol{\mu}^{\mathcal{W}})^T \delta\boldsymbol{\mu} + \varepsilon_k^{\boldsymbol{\mu}^{\mathcal{W}}} = 0 \quad (2.11)$$

avec  $\varepsilon_k^{\boldsymbol{\mu}}$  un bruit aléatoire. On réécrit l'Equation 2.11 sous forme matricielle :

$$\mathbf{G}\delta\boldsymbol{\mu} + \boldsymbol{\varepsilon} = \mathbf{d}$$

où

$$\mathbf{G} = \left( \nabla i_{k+1}(\boldsymbol{\mu}^{\mathcal{W}}) \right)_{\boldsymbol{\mu}^{\mathcal{W}} \in \mathcal{W}}^T \in \mathcal{M}_{|\mathcal{W}| \times 2}(\mathbb{R})$$

12. En pratique, le déplacement d'un point d'une image à la suivante doit être inférieur au pixel.

13. Les gradients de l'images peuvent être calculés par différences finies, ou par convolution avec un noyau de Sobel ou de Scharr [39]. Pour rendre le calcul plus robuste au bruit, il est commun d'appliquer une convolution par un noyau Gaussien à l'image avant dérivation [31].

14. Cette équation est très proche de l'équation de contrainte du flux optique. L'indétermination du flux optique porte le nom de problème d'ouverture; la composante du flux normale au gradient est inconnue [45].

et

$$\mathbf{d} = \left( i_k(\boldsymbol{\mu}^{\mathcal{W}}) - i_{k+1}(\boldsymbol{\mu}^{\mathcal{W}}) \right)_{\boldsymbol{\mu}^{\mathcal{W}} \in \mathcal{W}}^T \in \mathcal{M}_{|\mathcal{W}| \times 1}(\mathbb{R})$$

Dans ce contexte linéaire, on peut facilement calculer la solution au sens des moindres carrés :

$$\hat{\delta\boldsymbol{\mu}} = \arg \min_{\delta\boldsymbol{\mu} \in \mathbb{R}^2} \|\mathbf{G}\delta\boldsymbol{\mu} - \mathbf{d}\|_2^2 = \arg \min_{\delta\boldsymbol{\mu} \in \mathbb{R}^2} ((\mathbf{G}\delta\boldsymbol{\mu} - \mathbf{d})^T (\mathbf{G}\delta\boldsymbol{\mu} - \mathbf{d}))$$

Elle vérifie la condition d'optimalité :

$$\mathbf{G}^T \mathbf{G} \hat{\delta\boldsymbol{\mu}} = \mathbf{G}^T \mathbf{d} \quad (2.12)$$

Lorsque  $\mathbf{G}^T \mathbf{G} \in \mathcal{M}_{2 \times 2}(\mathbb{R})$  est inversible et bien conditionnée, la solution est donnée par :

$$\hat{\delta\boldsymbol{\mu}} = \mathbf{G}^+ \mathbf{d}$$

où

$$\mathbf{G}^+ = (\mathbf{G}^T \mathbf{G})^{-1} \mathbf{G}^T$$

désigne la matrice pseudo-inverse à gauche de  $\mathbf{G}$ . Ainsi, il est aisément et rapidement de calculer le déplacement d'un pixel d'une image à la suivante à l'aide de cette technique.

Cette solution est pertinente lorsque les fenêtres se déplacent d'une image à l'autre selon une translation pure. En réalité, leur déplacement est rarement aussi simple. Il est possible de tenir compte de transformations plus complexes (Annexe B), où tous les points de la fenêtre admettent des déplacements différents, mais l'ajout de paramètres accroît le coût du calcul [47]. Une autre possibilité consiste à pondérer l'influence des pixels selon leur position dans la fenêtre, *i.e.* les pixels proches du centre de la fenêtre ont plus d'importance que ceux qui en sont éloignés. La solution au sens des moindres carrés pondérés est notée :

$$\hat{\delta\boldsymbol{\mu}} = \arg \min_{\delta\boldsymbol{\mu} \in \mathbb{R}^2} ((\mathbf{G}\delta\boldsymbol{\mu} - \mathbf{d})^T \mathbf{W} (\mathbf{G}\delta\boldsymbol{\mu} - \mathbf{d}))$$

avec  $\mathbf{W} \in \mathcal{M}_{|\mathcal{W}|^2}(\mathbb{R})$  la matrice diagonale de pondération<sup>15</sup>. Elle vérifie la condition d'optimalité

$$\mathbf{G}^T \mathbf{W} \mathbf{G} \hat{\delta\boldsymbol{\mu}} = \mathbf{G}^T \mathbf{W} \mathbf{d} \quad (2.13)$$

et vaut

$$\hat{\delta\boldsymbol{\mu}} = (\mathbf{G}^T \mathbf{W} \mathbf{G})^{-1} \mathbf{G}^T \mathbf{W} \mathbf{d}$$

lorsque  $\mathbf{G}^T \mathbf{W} \mathbf{G} \in \mathcal{M}_{2 \times 2}(\mathbb{R})$  est inversible.

### Limites de cette approche et améliorations

**Conditionnement du problème.** Lorsque nous traitons un problème de moindres carrés linéaires, nous nous ramenons à la résolution d'un système d'équations linéaires de la forme  $\mathbf{A}\mathbf{x} = \mathbf{b}$  dans  $\mathbb{R}^n$ , où  $\mathbf{A}$  est une matrice carrée, supposée inversible (Equations 2.12 et 2.13). Il existe différentes méthodes numériques stables pour résoudre ce système (directes, itératives, ...), mais l'obtention d'une solution raisonnable n'est assurée que

15. Chaque coefficient diagonal définit le poids d'un pixel dans la minimisation. Lorsqu'ils sont tous pris égaux à 1, nous retrouvons la solution des moindres carrés linéaires exposée précédemment. Les poids peuvent être attribués selon une distribution Gaussienne pour donner plus d'importance aux pixels proches du centre de la fenêtre [47].

dans le cas où les erreurs sur la définition du problème sont faibles et la matrice  $\mathbf{A}$  bien conditionnée. Dans notre application,  $\mathbf{A} \equiv \mathbf{G}^T \mathbf{G}$  (sans la pondération) est une matrice  $2 \times 2$  que l'on peut inverser de manière directe ; c'est surtout la question de son conditionnement qui nous intéresse.

On parle de conditionnement d'une application, ou de façon équivalente lorsque l'application est linéaire, de conditionnement d'une matrice. Le conditionnement de la matrice  $\mathbf{A}$  est représenté par le nombre sans dimension

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\|$$

défini pour une norme matricielle  $\|\cdot\|$  arbitraire, vérifiant

$$\|\mathbf{A}\| \|\mathbf{A}^{-1}\| \geq \|\mathbf{A}\mathbf{A}^{-1}\| = 1$$

Une matrice est dite bien conditionnée lorsque son conditionnement est proche de 1, est moins bien conditionnée à mesure qu'il s'en éloigne. L'interprétation du conditionnement est liée à la propagation de l'erreur par l'application : une petite erreur sur  $\mathbf{b}$  se répercute sur  $\mathbf{x}$  de manière d'autant plus amplifiée que  $\kappa(\mathbf{A})$  est grand. Mathématiquement, en munissant  $\mathbb{R}^n$  d'une norme  $\|\cdot\|$  et  $\mathcal{M}_n(\mathbb{R})$  de la norme induite, si  $\mathbf{x}$  est solution du système  $\mathbf{Ax} = \mathbf{b}$  et  $\mathbf{x} + \delta\mathbf{x}$  du système  $\mathbf{A}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}$ , alors

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(\mathbf{A}) \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

Cette inégalité exprime la majoration de l'erreur relative pour une erreur sur le second membre. Lorsque le conditionnement est grand, il devient difficile d'obtenir une estimation de  $\mathbf{x}$  fiable [32].

Ici,  $\mathbf{G}^T \mathbf{G}$  est symétrique et à coefficients réels ; en utilisant la norme  $\|\cdot\|_2$  induite par la norme euclidienne pour les vecteurs, on peut réécrire  $\kappa(\mathbf{A})$  de la façon suivante :

$$\kappa(\mathbf{A}) = \left| \frac{\lambda_{max}}{\lambda_{min}} \right|$$

avec  $\lambda_{max} = \max(\lambda_1, \lambda_2)$  et  $\lambda_{min} = \min(\lambda_1, \lambda_2)$ , où  $\lambda_1$  et  $\lambda_2$  désignent les valeurs propres de  $\mathbf{A}$ . Ainsi, la matrice  $\mathbf{A}$  est mal conditionnée dès lors que  $\lambda_1$  et  $\lambda_2$  sont très petites ou que  $\lambda_{max} \gg \lambda_{min}$ . Ces conditions sont vérifiées dans des régions particulières de l'image :

- $\lambda_1$  et  $\lambda_2$  sont petites lorsque l'intensité des pixels varie peu dans  $\mathcal{W}$  ;
- $\lambda_1$  et  $\lambda_2$  sont très éloignées lorsque le pixel d'intérêt est sur un contour.

Il en résulte que tous les points ne sont pas bons à être traqués : il est préférable de travailler avec des points localisés dans des régions texturées, *i.e.* tels que  $\lambda_1$  et  $\lambda_2$  sont proches et grandes.

**Cas des grands déplacements.** Nous avons jusque là supposé que le déplacement  $\delta\boldsymbol{\mu}$  du pixel traqué était très petit (Section 2.2.1). Cette hypothèse est peu valide en pratique car les objets se déplace vite ou parce qu'on arrive pas à capturer des images à assez haute fréquence.

La solution est d'utiliser une pyramide de résolutions. L'idée générale est de diminuer la résolution de l'image courante et de l'image suivante jusqu'à ce que le déplacement du pixel d'intérêt soit inférieur au pixel et que l'approximation 2.9 redevienne valide [2].

L'algorithme fonctionne de la façon suivante :

- La résolution des images est réduite au minimum ;
- Le déplacement des pixels est calculé entre les deux images en utilisant la méthode présentée ;
- Les images de résolution supérieure sont considérées, le déplacement calculé précédemment est appliqué à l'image courante ;
- Le déplacement est calculé entre les deux images puis ajouté au déplacement précédent ;
- Les images de résolution supérieure sont considérés ;
- On réitère le processus jusqu'aux images de résolution initiale.

La fonction `calcOpticalFlowPyrLK` de la librairie OpenCV réalise le *tracking* d'un ensemble de point d'une image à la suivante à l'aide d'une pyramide de résolutions. Elle prend également en charge les éventuelles pertes de *tracking* (les points qui disparaissent sur l'image suivante sont rejetés) et ne tient compte que des bons points, au sens du bon conditionnement de la matrice  $\mathbf{G}^T \mathbf{G}$ .

### Les points d'intérêt, leur détection, leur description et le *matching*

**Les points de Harris et leur limite.** Le premier point sur lequel nous allons nous concentrer est la détection des points dans l'image. Ensuite, nous pourrons les détecter sur plusieurs images d'une même scène et les mettre en correspondance, c'est ce qu'on appelle le *matching*. Avant toute chose, il s'agit de définir ce qu'on entend par "point d'intérêt". Grossièrement, c'est un point qui se distingue des autres dans une image. En réalité, il n'existe pas de définition unique du point d'intérêt : on peut construire autant de points d'intérêts différents qu'il y a de propriétés qui font qu'un point diffère des autres. Un point caractéristique peut par exemple être un point qui a une intensité particulière. Cependant, il serait difficile de l'identifier dans une autre image dont les conditions d'éclairage ou l'échelle auraient changé. Idéalement, nous voulons détecter les mêmes points dans des images différentes même s'il y a des différences d'échelle, de perspective ou de luminosité entre elles. C'est pourquoi un point caractéristique est construit à l'aide de ses voisins dans l'image.

Les points les plus simples sont ceux de Harris. C'est aussi ceux avec lesquels nous avons commencé. Ils sont construits sur une hypothèse simple :

Un point caractéristique est au centre d'une fenêtre  $\mathcal{W}$  qui, lorsqu'elle est translatée dans n'importe quelle direction, voit son intensité varier de façon importante.

Visuellement, un point de Harris est un coin dans l'image. Nous allons montrer comment ils sont construits.

Soit un pixel de coordonnées  $\boldsymbol{\mu} = (\mu, \nu)^T$  dans une image  $\mathcal{I}$ . Nous cherchons à quantifier la variation d'intensité  $e$  dans la fenêtre  $\mathcal{W}$  centrée sur ce pixel, induite par une translation  $\delta\boldsymbol{\mu}$  :

$$e(\boldsymbol{\mu}, \delta\boldsymbol{\mu}) = \sum_{\boldsymbol{\mu}^w \in \mathcal{W}} (i(\boldsymbol{\mu}^w + \delta\boldsymbol{\mu}) - i(\boldsymbol{\mu}^w))^2$$

En supposant le déplacement  $\delta\boldsymbol{\mu}$  petit, nous utilisons la même simplification que pour le *tracking* (Equation 2.9) :

$$i(\boldsymbol{\mu}^w + \delta\boldsymbol{\mu}) \approx i(\boldsymbol{\mu}^w) + \nabla i(\boldsymbol{\mu}^w)^T \delta\boldsymbol{\mu}$$

Il vient :

$$\begin{aligned}
 e(\boldsymbol{\mu}, \delta\boldsymbol{\mu}) &\approx \sum_{\boldsymbol{\mu}^W \in \mathcal{W}} \left( \nabla_i (\boldsymbol{\mu}^W)^T \delta\boldsymbol{\mu} \right)^2 \\
 &= \sum_{\boldsymbol{\mu}^W \in \mathcal{W}} \left( [\nabla_i (\boldsymbol{\mu}^W)^T \delta\boldsymbol{\mu}]^T [\nabla_i (\boldsymbol{\mu}^W)^T \delta\boldsymbol{\mu}] \right) \\
 &= \delta\boldsymbol{\mu}^T \left[ \sum_{\boldsymbol{\mu}^W \in \mathcal{W}} (\nabla_i (\boldsymbol{\mu}^W) \nabla_i (\boldsymbol{\mu}^W)^T) \right] \delta\boldsymbol{\mu} \\
 &= \delta\boldsymbol{\mu}^T [\mathbf{G}^T \mathbf{G}] \delta\boldsymbol{\mu}
 \end{aligned} \tag{2.14}$$

où  $\mathbf{G}^T \mathbf{G} \in \mathcal{M}_{2 \times 2} \mathbb{R}$  est la matrice déjà rencontrée pour le *tracking* (Section 2.2.1). Comme nous l'avons brièvement fait remarquer, les valeurs propres de cette matrice indiquent le changement de l'intensité selon les deux directions principales orthogonales du gradient dans la fenêtre  $\mathcal{W}$ . En notant à nouveau  $\lambda_1$  et  $\lambda_2$  les deux valeurs propres de  $\mathbf{G}^T \mathbf{G}$ , le pixel considéré est un coin lorsque  $\lambda_1$  et  $\lambda_2$  sont proches et grandes. C'est également la condition que nous avions énoncée concernant les surfaces texturées, ce qui signifie qu'un point caractéristique ainsi détecté est aussi un bon point à traquer. Plus précisément, les points d'Harris sont tels que :

$$\frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2} \geq s$$

avec  $s$  un seuil fixé. Notez que peu après que le *tracking* a été introduit, Shi et Tomasi ont proposé un nouveau critère de sélection des points à traquer [44] :

$$\min(\lambda_1, \lambda_2) \geq s$$

Ces deux types de points peuvent être calculés à l'aide des méthodes pré-codées `cornerHarris` et `goodFeaturesToTrack` d'OpenCV.

Nous avons commencé nos essais avec des points de Harris pour leur intérêt au delà du *tracking*.

**La description et le *matching* des points caractéristiques.** Une fois les points caractéristiques détectés sur une image, nous pouvons les traquer sur les images suivantes. Néanmoins, nous ne savons pas à quoi correspondent ces points sur la plate-forme. Or, nous avons besoin de correspondances entre les points sur l'image et leur homologue dans le repère de la plate-forme pour pouvoir estimer la *pose* de la caméra (Section 2.1.2). C'est là que le *template* intervient : nous connaissons à l'avance le visuel de la plate-forme, le *template* et la position dans le repère de la plate-forme de tous les points qui y figurent, il suffit de détecter des points sur le *template* et de les relier à ceux que nous avons calculé sur l'image courante du drone. Cela fait intervenir deux étapes :

la **description** des points détectés ;

le ***matching*** des points entre les deux images.

Une nouvelle fois, il n'existe de façon unique ni pour décrire les points, ni pour les mettre en relation. Soit  $\boldsymbol{\mu}_1$  les coordonnées d'un pixel dans une première image  $\mathcal{I}_1$ , et  $\boldsymbol{\mu}_2$  les coordonnées d'un pixel dans une seconde image  $\mathcal{I}_2$ . La méthode la plus simple pour savoir si  $\boldsymbol{\mu}_1$  et  $\boldsymbol{\mu}_2$  renvoient au même point consiste à calculer la corrélation entre les intensités

des pixels dans le voisinage  $\mathcal{W}_1$  de  $\mu_1$  dans  $\mathcal{I}_1$  et celles des pixels dans le voisinage  $\mathcal{W}_2$  de  $\mu_2$  dans  $\mathcal{I}_2$ . Autrement dit, chaque point est décrit par la fenêtre dont il est le centre et est mis en relation avec un autre point si l'intensité des pixels de leur fenêtre est proche. Mathématiquement, si  $\mathcal{W}_1$  et  $\mathcal{W}_2$  sont des fenêtres carrées de taille  $2(n+1) \times 2(n+1)$  pixels,  $\mu_1$  et  $\mu_2$  renvoient au même point si et seulement si

$$\sum_{(k, l) \in \{-n, \dots, n\}^2} \left[ i_1 \left( \mu_1 + \binom{k}{l} \right) - i_2 \left( \mu_2 + \binom{k}{l} \right) \right]^2 \leq s$$

avec  $s$  un seuil à définir.

Cette première description simpliste des points par des fenêtres carrées n'est cependant pas suffisante lorsqu'il y a des changements d'échelle, de luminosité ou de perspective entre les images. Pour compenser ces écarts, il faudrait également redimensionner les fenêtres, leur appliquer une transformation projective et normaliser les intensités. Au delà de la description, la détection des points par la méthode d'Harris est invariante à la rotation (un coin reste un coin après une rotation) mais pas au changement d'échelle (des coins sur une image peuvent devenir des points d'un contour lorsqu'elle est agrandie). Or, dans notre étude, nous travaillons avec des images de la plate-forme prises à partir d'un point quelconque de l'espace. Par conséquent, nous avons choisi de travailler avec des points et des descripteurs plus évolués, invariants à la rotation et au changement d'échelle, et robuste à la transformation projective : les points et les descripteurs *SIFT*, pour *Scale-Invariant Feature Transform* [33]. Nous ne détaillerons pas son fonctionnement. Notez que si les points et les descripteurs *SIFT* sont les plus performants à l'heure actuelle, ce sont aussi les plus coûteux. C'est pourquoi des variantes plus légères ont été développées (points *SURF* et *FAST*, descripteurs *BRIEF*, ...).

En pratique, nous avons seulement besoin de détecter des points sur la première image, à la suite de quoi ils seront traqués à l'aide de la méthode de Lucas et Kanade. Éventuellement, nous pourrions être amenés à refaire une détection si l'on perdait le *tracking*. Dans les deux cas, l'exécution ponctuelle de l'algorithme n'est pas limitante pour notre application en temps réel.

En réalité, le développement d'alternatives à *SIFT* a aussi été motivé par le fait que l'utilisation de l'algorithme était protégée par un brevet jusqu'en 2020. Il intègre désormais les versions récentes de la librairie OpenCV au sein de la classe `sift`.

### La procédure de *template tracking*

Une fois la position des points sur l'image suivante obtenue, nous pouvons calculer l'homographie entre le *template* et l'image courante (Annexe C). Nous en déduisons la position des coins du *template* dans l'image courante, et nous nous servons de ces quatre correspondances pour estimer la *pose* de la caméra dans le repère de la plate-forme (Section 2.1.2).

La vidéo d'illustration du *template tracking* basé sur l'utilisation de points caractéristiques est accessible [ici](#).

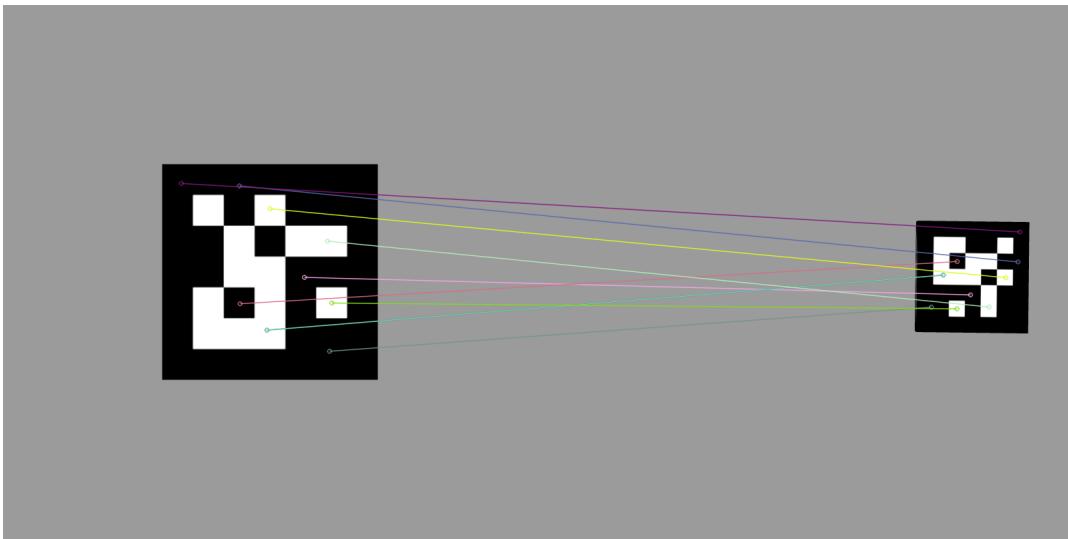


FIGURE 2.6 – *Matching* de points *SIFT* après rotation et changement d'échelle

La structure du noeud ROS pour le calcul des correspondances 2D-3D par *template tracking* basé *features* est donnée ci-dessous.

#### ROS Node : features\_based\_template\_tracking

**Subscribed topics :**  
`camera1/image_raw`

**Published topics :**  
`correspondences`

**Action server :**  
`TemplateTracking`

### 2.2.2 *Template tracking* basé *template*

Contexte du *template tracking* global et introduction aux notations utilisées

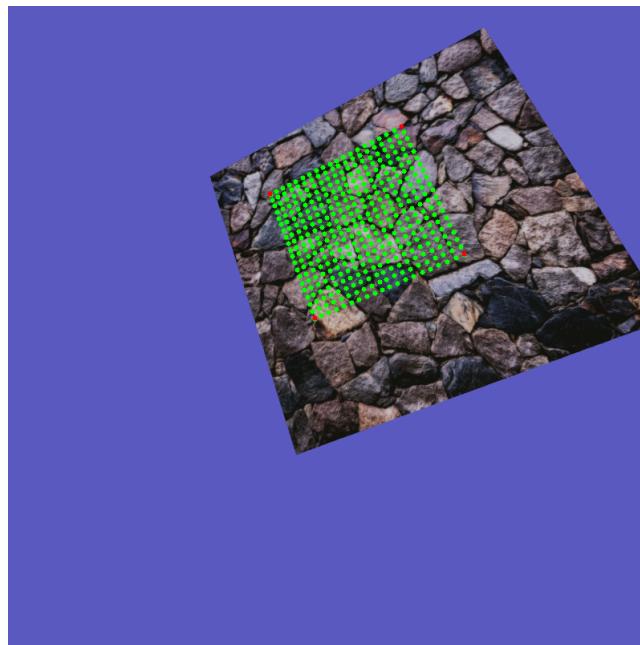
La solution que nous présentons a été introduite en 2001 par Jurie et Dhome [26]. Elle s'appuie sur l'apprentissage hors-ligne d'une fonction de régression linéaire qui lie la variation d'intensité dans une région de l'image à une transformation géométrique. Le fait que cet apprentissage soit réalisé en amont permet de tracker le *template* en temps réel.

Définissons le *template* comme étant une image rectangulaire<sup>16</sup> texturée (Figure 2.7). Soit  $\mathcal{I}_k$  l'image traitée à l'instant  $k$ . Dans cette image, la position du *template* est paramétrée par la position de ses quatre coins  $\boldsymbol{\mu}_k$  (Figure 2.8) :

$$\boldsymbol{\mu}_k = (\boldsymbol{\mu}_k^1, \boldsymbol{\mu}_k^2, \boldsymbol{\mu}_k^3, \boldsymbol{\mu}_k^4)^T$$

qui est un vecteur de dimension  $8 \times 1$ , où  $\boldsymbol{\mu}_k^i = (\mu_k^i, \nu_k^i)^T$  est la position du coin  $i$ .

16. Notez que la forme du *template* peut être quelconque. Nous le choisissons rectangulaire pour sim-

FIGURE 2.7 – Exemple de *template*FIGURE 2.8 – Exemple d'image  $\mathcal{I}_k$ . Le *template* est discréteisé en une grille uniforme (points verts) et ses coins  $\mu_k$  sont repérés (points rouge).

La connaissance de ces quatre points est suffisante pour calculer la position du *template* dans l'image, *i.e.* la transformation géométrique (Annexe B) qui permet de passer du *template* de référence (Figure 2.7) au *template* dans l'image  $\mathcal{I}_k$ . Dans notre cas, la transformation choisie est une homographie, représentée par la matrice  $\mathbf{F}_k$ . Avec ces notations, nous pouvons réécrire l'objectif du *template tracking* :

Il s'agit de suivre le *template* dans une séquence d'images en estimant les paramètres  $\mu_k$  de la fonction de déformation (ou de transformation)  $\mathbf{F}_k$  à chaque nouvelle image  $k$  [19].

Au lieu de travailler avec l'ensemble des pixels qui composent le *template*, nous l'échantillonnons en un sous-ensemble de  $n_p$  points répartis en une grille (Figure 2.8). A cette grille, nous associons le vecteur  $\mathbf{i}_k$  de dimension  $n_p \times 1$  des intensités<sup>17</sup> obtenues en chacun

---

plifier nos explications.

17. Les images sont converties en nuances de gris avant d'être traitées. Les intensités correspondent donc à des nuances de gris.

des  $n_p$  points :

$$\mathbf{i}_k = (i_k^1, i_k^2, \dots, i_k^{n_p})^T$$

Nous faisons dans la suite l'hypothèse que l'intensité d'un point se déplaçant d'une image à l'autre dans une séquence reste constante ; nous en tirons une conséquence immédiate :

$$\forall i, j \in \{1, 2, \dots, n_p\}, \mathbf{i}_i = \mathbf{i}_j = \mathbf{i}^i \quad (2.15)$$

où  $\mathbf{i}^*$  est le vecteur des intensités extrait du *template* de référence (Figure 2.9).

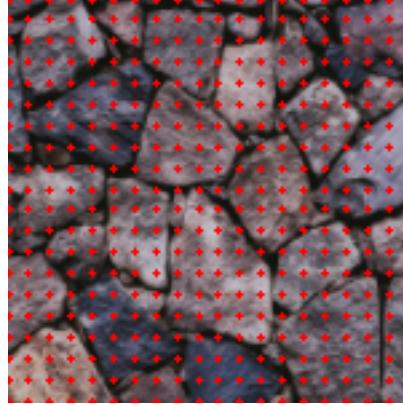


FIGURE 2.9 – Échantillonnage du *template*

La méthode de *template tracking* comporte deux étapes : l'apprentissage hors-ligne d'une fonction de régression linéaire et le *tracking* du *template* en temps réel.

## Apprentissage

Dans la phase d'apprentissage, nous partons d'une image de référence  $\mathcal{I}^*$  (Figure 2.10) et calculons le vecteur  $\mathbf{i}^*$  des intensités aux points de la grille dans le *template* (Figure 2.9). Nous estimons facilement la transformation  $\mathbf{F}^*$  entre le *template* de référence et le *template* dans cette image de référence. Nous notons  $\boldsymbol{\mu}^*$  la position des coins du *template* dans l'image de référence.

Ensuite, nous générerons  $n_t$  petites transformations de la grille par rapport à sa position de référence et notons  $t$  l'une d'entre-elles (Figure 2.11). En pratique nous tirons aléatoirement  $\delta\boldsymbol{\mu}_t$ , les déplacements de chacun des quatre coins du *template*, de l'ordre de quelques pixels, que nous additionnons à  $\boldsymbol{\mu}^*$  pour obtenir  $\boldsymbol{\mu}_t$ , et calculons l'homographie  $\mathbf{F}'_t$  qui réalise le passage de la position du *template* dans l'image de référence à sa nouvelle position après déplacement, de manière à pouvoir déplacer les autres points de la grille. Avec ces nouvelles notations :

$$\mathbf{F}_t = \mathbf{F}'_t \mathbf{F}^* \quad (2.16)$$

Nous pouvons extraire, toujours sur l'image de référence, le vecteur des intensités  $\mathbf{i}_t$  obtenu après cette transformation. Ainsi, il est possible d'associer à chaque transformation une variation d'intensité des points de la grille, ou dans l'autre sens, à chaque variation d'intensité une transformation. C'est la base du fonctionnement du *tracking*. Pour tenter de généraliser cette relation, nous introduisons la fonction de régression  $f$  telle que :

$$\delta\boldsymbol{\mu}_t = f(\delta\mathbf{i}_t) + \boldsymbol{\varepsilon}_t \quad (2.17)$$

où

$$\delta\boldsymbol{\mu}_t = \boldsymbol{\mu}_t - \boldsymbol{\mu}^* \quad \text{et} \quad \delta\mathbf{i}_t = \mathbf{i}^* - \mathbf{i}_t$$



FIGURE 2.10 – Exemple d'image de référence dans laquelle nous retrouvons le *template* (cadre rouge), et que nous pouvons utiliser pour réaliser l'apprentissage

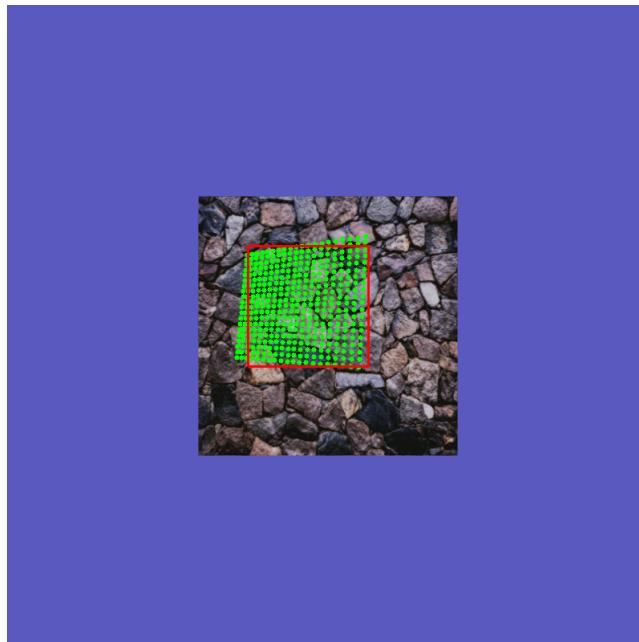


FIGURE 2.11 – Exemple d'une petite déformation du *template* (la nouvelle position du *template*, *ie* de la nouvelle grille, est dessinée en vert) dans l'image de référence

représentent respectivement le petit déplacement des coins du *template* (*ie* la petite modification des paramètres de l'homographie  $\mathbf{F}'_t$ ) et la variation d'intensité associée et où  $\epsilon_t$  est un bruit aléatoire.

Différentes fonctions de régression ont été utilisées à des fins de *tracking* [3]. Nous nous appuierons sur la plus simple d'entre-elles, la modélisation par des hyperplans de Jurie et Dhome [26], définie par :

$$f(\delta \mathbf{i}_t) = \mathbf{A} \delta \mathbf{i}_t$$

où  $\mathbf{A} \in \mathcal{M}_{8 \times n_p}(\mathbb{R})$ . L'objectif de cette étape d'apprentissage est d'approcher la matrice  $\mathbf{A}$ .

Pour ce faire, nous rassemblons les  $n_t$  déplacements et différences d'intensité calculés dans des matrices  $\mathbf{Y}$  et  $\mathbf{H}$  :

$$\mathbf{Y} = (\delta\boldsymbol{\mu}_1, \delta\boldsymbol{\mu}_2, \dots, \delta\boldsymbol{\mu}_{n_t}) \in \mathcal{M}_{8 \times n_t}(\mathbb{R}) \quad \text{et} \quad \mathbf{H} = (\delta\mathbf{i}_1, \delta\mathbf{i}_2, \dots, \delta\mathbf{i}_{n_t}) \in \mathcal{M}_{n_p \times n_t}(\mathbb{R})$$

et réécrivons la relation 2.17 :

$$\mathbf{Y} = \mathbf{AH} + \boldsymbol{\varepsilon} \quad \text{avec } \boldsymbol{\varepsilon} = (\boldsymbol{\varepsilon}_1, \boldsymbol{\varepsilon}_2, \dots, \boldsymbol{\varepsilon}_{n_t}) \quad (2.18)$$

De l'équation 2.18, nous déduisons l'expression de l'estimateur des moindres carrés ordinaires de  $\mathbf{A}$  :

$$\hat{\mathbf{A}} = \mathbf{Y}\mathbf{H}^+$$

où

$$\mathbf{H}^+ = \mathbf{H}^T(\mathbf{HH}^T)^{-1}$$

désigne la matrice pseudo-inverse à droite de  $\mathbf{H}$ . Notez que l'inversion de la matrice  $\mathbf{HH}^T$  est coûteux (inversion d'une matrice de dimension  $n_p \times n_p$  où  $n_p$  est typiquement de l'ordre de plusieurs centaines de points). Ce calcul n'est pas gênant dans le cas où il est effectué hors ligne mais dès lors que l'on veut mettre à jour le prédicteur  $\hat{\mathbf{A}}$  ou modifier le *template* en temps réel, il est nécessaire de changer d'approche. A cet effet, des schémas différents pour l'apprentissage de  $\hat{\mathbf{A}}$  ont été mis en place [18], mais nous nous contenterons dans la suite d'utiliser celui que nous venons de présenter. L'apprentissage est synthétisé dans le l'algorithme 1.

---

#### Algorithm 1: Template Learning

---

**Data:** Reference image  $\mathcal{I}^*$ , corners reference position  $\boldsymbol{\mu}^*$

**Result:** Linear predictor  $\hat{\mathbf{A}}$

$\mathbf{Y} \leftarrow \mathbf{0}_{\mathcal{M}_{8 \times n_t}(\mathbb{R})};$

$\mathbf{H} \leftarrow \mathbf{0}_{\mathcal{M}_{n_p \times n_t}(\mathbb{R})};$

Sample the template in  $\mathcal{I}^*$  with a grid of  $n_p$  points;

Extract the intensity vector  $\mathbf{i}^*$  from  $\mathcal{I}^*$ ;

**for**  $t \leftarrow 1$  **to**  $n_t$  **do**

Add random displacement vector  $\delta\boldsymbol{\mu}_t$  to  $\boldsymbol{\mu}^*$  to obtain  $\boldsymbol{\mu}_t$ ;

Compute the homography matrix  $\mathbf{F}'_t$  between  $\boldsymbol{\mu}^*$  and  $\boldsymbol{\mu}_t$ ;

Apply  $\mathbf{F}'_t$  to each point of the grid;

Get  $\mathbf{i}_t$  in  $\mathcal{I}^*$  at the new grid points;

$\delta\mathbf{i}_t \leftarrow \mathbf{i}^* - \mathbf{i}_t$ ;

Fill the  $t^{th}$  column of  $\mathbf{Y}$  with  $\delta\boldsymbol{\mu}_t$ ;

Fill the  $t^{th}$  column of  $\mathbf{H}$  with  $\delta\mathbf{i}_t$ ;

**end**

$\hat{\mathbf{A}} \leftarrow \mathbf{Y}\mathbf{H}^T(\mathbf{HH}^T)^{-1};$

---

#### Tracking

Une fois la matrice  $\hat{\mathbf{A}}$  calculée et stockée dans la mémoire, nous sommes en mesure de réaliser le *tracking*.

A l'initialisation, nous supposons avoir la transformation  $\mathbf{F}'_1$  qui permet le passage du *template* dans l'image de référence utilisée pour l'apprentissage au *template* dans la première image  $\mathcal{I}_1$  de la séquence.

Nous notons à nouveau  $k$  un instant de la séquence. Nous connaissons la position des coins du *template*  $\boldsymbol{\mu}_k$  dans l'image  $\mathcal{I}_k$ , à partir de laquelle nous pouvons calculer la transformation  $\mathbf{F}'_k$ . Le but est de déterminer leur position  $\boldsymbol{\mu}_{k+1}$  dans l'image suivante. Grace à l'apprentissage, nous pouvons estimer un déplacement  $\delta\boldsymbol{\mu}_k$  :

$$\delta\boldsymbol{\mu}_k = \hat{\mathbf{A}}\delta\mathbf{i}_k$$

avec

$$\delta\mathbf{i}_k = \hat{\mathbf{i}}_{k+1} - \mathbf{i}_k = \hat{\mathbf{i}}_{k+1} - \mathbf{i}^* \quad (\text{condition 2.15})$$

où  $\hat{\mathbf{i}}_{k+1}$  est le vecteur des intensités calculé dans l'image  $\mathcal{I}_{k+1}$  pour la position précédente de la grille. Le point clé du *tracking* est la signification du déplacement  $\delta\boldsymbol{\mu}_k$  que nous venons de calculer. Précédemment, nous avons appris une relation (équation 2.18) qui lie une différence d'intensité dans l'image à un déplacement du *template* par rapport à sa position dans l'image de référence. Le déplacement  $\delta\boldsymbol{\mu}_k$  ainsi calculé n'égal au déplacement des points du *template* que dans le cas où l'image  $\mathcal{I}_k$  coïncide avec l'image de référence. Il est néanmoins possible de contourner ce problème relativement aisément [26], en rapportant le déplacement par rapport au *template* dans l'image de référence en un déplacement par rapport au *template* dans l'image précédente à l'aide de la transformation  $\mathbf{F}'_k$  connue. Nous obtenons ainsi la mise à jour suivante de la position des coins du *template* :

$$\tilde{\boldsymbol{\mu}}_{k+1} = \tilde{\boldsymbol{\mu}}_k + \mathbf{F}'_k \tilde{\boldsymbol{\delta}\boldsymbol{\mu}}_k$$

où  $\tilde{\mathbf{x}}$  représente les coordonnées homogènes du vecteur  $\mathbf{x}$ . Finalement, à partir de  $\boldsymbol{\mu}_{k+1}$ , nous pouvons obtenir la transformation  $\mathbf{F}'_{k+1}$ , et en déduire  $\mathbf{F}_{k+1}$  en utilisant la relation 2.16. Nous synthétisons un pas de *tracking* dans l'algorithme 2 et l'illustrons dans la figure 2.12.

---

**Algorithm 2:** Template Tracking

---

**Data:** Pre-computed  $\hat{\mathbf{A}}$ , previous homography  $\mathbf{F}'_k$ , homography  $\mathbf{F}^*$ , reference intensities  $\mathbf{i}^*$ , reference position  $\boldsymbol{\mu}^*$ , image  $\mathcal{I}_{k+1}$ , previous sampling grid, reference grid

**Result:** Homography  $\mathbf{F}_{k+1}$

Get  $\hat{\mathbf{i}}_{k+1}$  in  $\mathcal{I}_{k+1}$  at the previous grid position;

$\delta\mathbf{i}_k \leftarrow \hat{\mathbf{i}}_{k+1} - \mathbf{i}^*$ ;

$\delta\boldsymbol{\mu}_k \leftarrow \hat{\mathbf{A}}\delta\mathbf{i}_k$ ;

Apply  $\mathbf{F}'_k$  to  $\boldsymbol{\mu}^* + \delta\boldsymbol{\mu}_k$  to obtain  $\boldsymbol{\mu}_{k+1}$ ;

Compute the homography matrix  $\mathbf{F}'_{k+1}$  between  $\boldsymbol{\mu}^*$  and  $\boldsymbol{\mu}_{k+1}$ ;

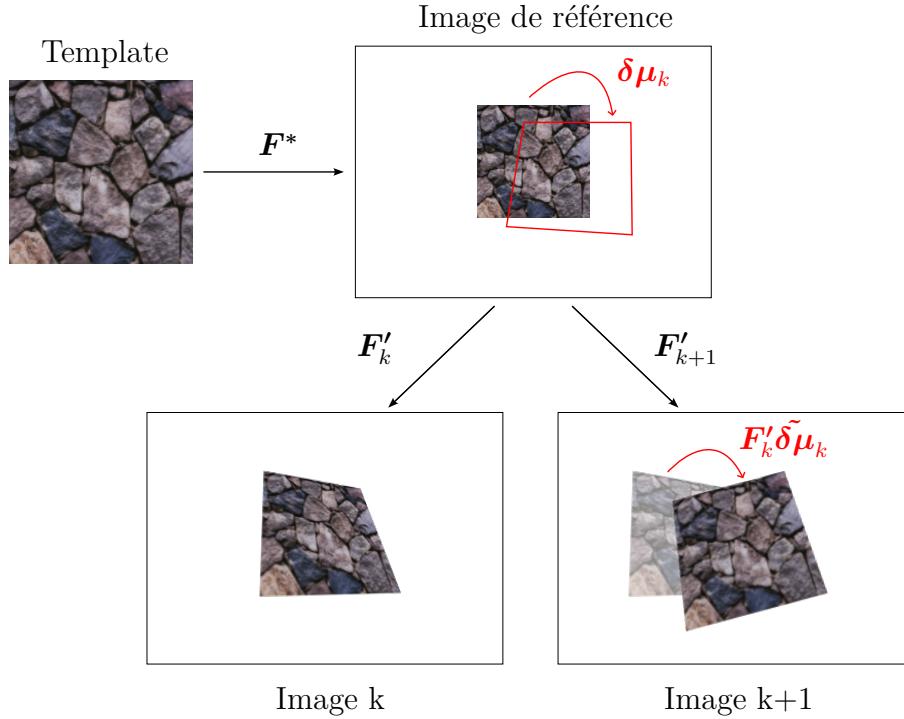
Compute the new grid by applying  $\mathbf{F}'_{k+1}$  to the reference grid;

$\mathbf{F}_{k+1} \leftarrow \mathbf{F}'_{k+1} \mathbf{F}^*$ ;

---

## Améliorations de la méthode

Comme nous l'avons évoqué, il est possible d'accélérer l'étape d'apprentissage [18], mais il existe également des solutions pour rendre la méthode plus robuste aux grands déplacements [18, 3], aux changements d'éclairage [27, 18] ou aux occlusions partielles [27].

FIGURE 2.12 – Schéma d'un pas de *tracking*

**Robustesse aux grands déplacements.** Une première amélioration simple du *tracking* est d'appliquer l'étape décrite dans l'algorithme 2 plusieurs fois consécutives pour chaque image. Néanmoins, l'apprentissage que nous avons mis en place s'appuie sur des petits déplacements du *template* dans l'image. Ainsi, dès lors que le déplacement du *template* d'une image à la suivante est grand, *ie* que la vitesse de déplacement de l'objet est élevée, les résultats de la prédiction sont moins précis. Une solution complémentaire consiste à entraîner plusieurs matrices  $(\hat{A}_l)_{l \in \{1, 2, \dots, n_l\}}$ , où  $n_l$  désigne un nombre de niveaux, en réduisant à chaque nouveau niveau la plage des déplacements dans laquelle nous effectuons des tirages aléatoires. Concernant le *tracking*, cela conduit à exécuter l'algorithme 2  $n_l$  fois à chaque image, pour chaque matrice  $\hat{A}_l$ .

**Robustesse aux changements d'éclairage.** Il est possible de rendre la méthode indépendante des changements de luminosité en normalisant localement les intensités. Par exemple [27], nous pouvons transformer l'intensité  $i(\mu)$  au pixel positionné en  $\mu = (\mu, \nu)$  par :

$$i(\mu) \mapsto \frac{i(\mu) - i_{min}^\mu}{i_{max}^\mu - i_{min}^\mu}$$

où  $i_{max}^\mu$  et  $i_{min}^\mu$  sont respectivement les intensités maximale et minimale calculées dans une petite fenêtre centrée sur  $\mu$ . Cette fonction est à appliquer à la fois pour l'apprentissage et la *tracking*.

**Robustesse aux occlusions.** Les méthodes de *template tracking* (ou *matching*) qui n'utilisent pas de *features* (comme c'est le cas ici) sont très sensibles aux occlusions. De plus, dans notre cas précis, si l'on peut convenablement supposer qu'aucun objet tierce ne viendra s'interposer entre le drone et la plate-forme pendant le vol, il est moins évident de considérer que la plate-forme restera dans le champ de vision de la caméra

durant toute la durée de la manœuvre d'appontage<sup>18</sup>. Une solution pour s'affranchir de ce problème a été proposée par Jurie et Dhome en 2002 [27]. Elle consiste à se servir non pas d'un seul mais de plusieurs *templates* obtenus en subdivisant le *template* initial. Les étapes d'apprentissage et de *tracking* sont réalisées pour chacun des sous-*templates*, et les résultats pondérés et fusionnés pour approcher la transformation réelle du *template* initial.

La structure du noeud ROS pour le calcul des correspondances 2D-3D par *template tracking* basé *template* est donnée ci-dessous.

### ROS Node : template\_based\_template\_tracking

**Subscribed topics :**  
camera1/image\_raw

**Published topics :**  
correspondences

**Action server :**  
TemplateTracking

La vidéo du *template tracking* basé sur l'utilisation d'un *template* est accessible [ici](#).

## 2.3 Analyse des résultats

Les algorithmes que nous implémentons doivent être validés selon différents critères :

- La précision des résultats ; nous introduirons une fonction d'erreur pour la quantifier et comparer différentes solutions entre elles.
- Leur robustesse vis à vis des données réelles ; des essais supplémentaires doivent idéalement être faits sur de vraies images, hors simulation, pour rendre compte de leur comportement face aux changements de luminosité ou aux occlusions partielles.
- Leur complexité en espace et en temps ; les calculs doivent pouvoir être réalisés par le petit ordinateur embarqué du drone.

**Précision et stabilité.** La première question que nous nous posons est : comment quantifier l'erreur sur la *pose* estimée ? Pour y répondre, nous allons construire une fonction d'erreur qui compare la *pose* estimée de la plate-forme à sa *pose* réelle obtenue en simulation.

La simulation est très avantageuse pour tester des algorithmes ; *Gazebo* joint un repère à chaque solide de l'environnement et calcule la transformation rigide entre deux repères à notre place. Ainsi, nous pouvons calculer la transformation entre le repère de la plate-forme et le repère caméra. Connaissant la matrice de calibration interne  $\mathbf{K}$  de la caméra, il aurait aussi été possible de retrouver la position des coins du *template* dans le plan

<sup>18</sup> Du fait du sous-actionnement du drone, le *template* peut sortir de l'image à cause d'un mouvement en roulis ou en tangage que l'on n'aurait pas prévu.

image à partir de leur position (connue et fixe) exprimée dans le repère de la plate-forme, mais nous avons fait le choix de ne comparer que les *poses* entre-elles.

Dans Gazebo, nous avons tiré profit du package `tf` pour ajouter de nouveaux repères à la simulation, calculer des transformations entre plusieurs repères et les extraire en direct de la simulation.

Nous avons calculé la matrice de transformation homogène qui permet de passer de la *pose* estimée à la *pose* réelle, le but étant que cette transformation soit la plus proche possible de l'identité. Une fois cette transformation calculée, nous pouvons en extraire une matrice de rotation et un vecteur de translation. Pour la translation, nous avons pris comme score la norme euclidienne du vecteur de translation. Quant à la rotation, nous pris la norme euclidienne des angles d'Euler calculés à partir de la matrice de rotation à l'aide de la formule de Rodrigues.

### Courbe

Nous observons erreurs stable dans le temps, de l'ordre du millimètre pour la translation et du centième de radian pour la rotation lorsque les déplacements de la plate-forme ne sont pas trop importants. Ces erreurs peuvent aller jusqu'au centimètre pour la translation et au dixième de radian lorsque les déplacements de la plate-forme sont plus importants.

**Sensibilité aux changements d'éclairage et aux occlusions.** La normalisation des intensités permet de réduire grandement la sensibilité aux changements d'éclairage du *template tracking*. Pour mener une analyse complète, il est nécessaire de tester les algorithmes sur des images réelles.

**Calcul en temps réel et embarqué.** Il convient de distinguer le temps d'apprentissage du temps de *tracking*. L'apprentissage pouvant être réalisé hors-ligne (nous connaissons le motif de la plate-forme à l'avance et supposons qu'il ne change pas au cours du temps), sa durée a peu d'importance pour notre étude. C'est sur le temps d'exécution d'une étape de *tracking* que nous allons concentrer notre attention. L'algorithme doit être assez rapide pour tourner en temps réel sur la carte embarquée du drone<sup>19</sup>. Pour simplifier l'évaluation de la rapidité du *tracking*, nous ne ferons que des calculs en simulation, mais nous comparerons la durée d'une itération de *tracking* de nos algorithmes avec la durée d'une détection de marqueur ArUco, les marqueurs ArUco étant régulièrement utilisés pour des applications de *tracking* embarqué.

Après plusieurs essais, nous observons qu'il est possible d'atteindre une durée d'itération dix fois moins importante avec l'approche locale, et cinq fois moins importante avec l'approche globale par rapport à la détection de marqueurs ArUco (Tableau 2.1).

## 2.4 Estimation de l'état de la plate-forme

### 2.4.1 Construction des équations d'état de la plate-forme

La *pose* que nous venons d'estimer est à prendre comme la mesure d'un capteur qui, comme toute mesure de capteur, est bruitée. Aussi, si nous avons vu des méthodes pour

19. La complexité en espace mémoire a également de l'importance même si nous ne traitons pas ce point avec rigueur ici.

Coût d'une itération (ms)	
ArUco	20
Locale	2
Globale	4

TABLE 2.1 – Coût des méthodes d'estimation de *pose*

rendre l'estimation de *pose* robuste aux occlusions partielles, celles-ci ne fonctionnent plus lorsque la plate-forme est totalement hors du champ de vision de la caméra. Ces raisons m'ont amené à considérer en plus un modèle simpliste de la dynamique de la plate-forme par rapport à un repère fixe du monde. Ainsi, lorsque des estimations de *pose* sont disponibles, le modèle ne fait que les lisser, et lorsqu'elles ne sont pas disponibles, il nous fournit quand même une solution.

Soient  $(x, y, z)$  les coordonnées de l'origine du repère  $\mathcal{R}_p$  de la plate-forme dans un repère fixe du monde  $\mathcal{R}_w$ , et  $(\theta, \psi, \phi)$  les angles d'Euler selon la convention ZYX qui paramètrent l'orientation de  $\mathcal{R}_p$  dans  $\mathcal{R}_w$ . En supposant que le navire (ou la plate-forme) oscille peu, nous imposons les contraintes suivantes [42, 8] :

$$\forall \chi \in \{x, y, z, \theta, \psi, \phi\}, \quad \chi^{(5)} = \frac{d^5 \chi}{dt^5} = 0 \quad (2.19)$$

Posons

$$\mathbf{x} = (x, y, z, \theta, \psi, \phi, \dots, x^{(5)}, y^{(5)}, z^{(5)}, \theta^{(5)}, \psi^{(5)}, \phi^{(5)})^T$$

l'état de la plate-forme. A partir de maintenant, nous ne parlerons plus de *pose* relative de la plate-forme par rapport à la caméra, mais d'état de la plate-forme par rapport au monde. Nous allons construire des équations d'état pour la plate-forme qui tiennent compte des contraintes 2.19, et tenter d'approcher l'état à partir des estimations de *pose*. C'est ce qu'on appelle l'observation d'état ou le filtrage.

La formule de Taylor-Young à l'ordre 5 nous permet d'écrire :

$$\chi(t + dt) = \sum_{l=0}^5 \frac{\chi^{(l)}(t)}{l!} dt^l + o(dt^5) \quad (2.20)$$

où  $\chi$  désigne à nouveau  $x, y, z, \theta, \psi$  ou  $\phi$ . En notant  $t = k.dt$  avec  $k \in \mathbb{N}$ , et  $\chi(t) = \chi(k.dt) = \chi_k$ , nous pouvons réécrire l'Equation 2.20 sous forme discrète :

$$\chi_{k+1} = \sum_{l=0}^5 \frac{\chi_k^{(l)}}{l!} dt^l + o(dt^5)$$

De la même façon, on a

$$\chi'_{k+1} = \sum_{l=0}^4 \frac{\chi_k^{(l+1)}}{l!} dt^l + o(dt^4)$$

jusqu'à

$$\chi_{k+1}^{(4)} = \chi_k^{(4)} + \chi_k^{(5)} dt + o(dt)$$

Enfin, on impose que les dérivées cinquièmes par rapport au temps sont constantes :

$$\chi_{k+1}^{(5)} = \chi_k^{(5)}$$

Ces relations nous permettent d'écrire l'équation d'évolution discrète de l'état du système :

$$\boldsymbol{x}_{k+1} = \boldsymbol{A}\boldsymbol{x}_k + \boldsymbol{\alpha}_k$$

avec

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{I}_6 & dt\boldsymbol{I}_6 & (dt^2/2)\boldsymbol{I}_6 & (dt^3/6)\boldsymbol{I}_6 & (dt^4/24)\boldsymbol{I}_6 & (dt^5/120)\boldsymbol{I}_6 \\ \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \boldsymbol{I}_6 & dt\boldsymbol{I}_6 & (dt^2/2)\boldsymbol{I}_6 & (dt^3/6)\boldsymbol{I}_6 & (dt^4/24)\boldsymbol{I}_6 \\ \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \boldsymbol{I}_6 & dt\boldsymbol{I}_6 & (dt^2/2)\boldsymbol{I}_6 & (dt^3/6)\boldsymbol{I}_6 \\ \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \boldsymbol{I}_6 & dt\boldsymbol{I}_6 & (dt^2/2)\boldsymbol{I}_6 \\ \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \boldsymbol{I}_6 & dt\boldsymbol{I}_6 \\ \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \boldsymbol{I}_6 \end{bmatrix}$$

et  $\boldsymbol{\alpha}_k \in \mathcal{M}_{36 \times 1}(\mathbb{R})$  un bruit aléatoire.

D'autre part, nous incluons les contraintes 2.19 au sein de l'équation de mesure discrète du système :

$$\boldsymbol{y}_k = \boldsymbol{C}\boldsymbol{x}_k + \boldsymbol{\beta}_k$$

où

$$\boldsymbol{y}_k = (x_k, y_k, z_k, \theta_k, \psi_k, \phi_k, 0, 0, 0, 0, 0, 0)^T,$$

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{I}_6 & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} \\ \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \mathbf{0}_{\mathcal{M}_6(\mathbb{R})} & \boldsymbol{I}_6 \end{bmatrix}$$

et  $\boldsymbol{\beta}_k \in \mathcal{M}_{12 \times 1}(\mathbb{R})$  désigne un bruit aléatoire.

## 2.4.2 Filtrage de Kalman

S'il existe des techniques pour estimer l'état  $\boldsymbol{x}$  lorsque  $\boldsymbol{x}$  et  $\boldsymbol{y}$  sont déterministes, ici nous considérons que les variables  $\boldsymbol{x}$  et  $\boldsymbol{y}$  sont aléatoires, et nous tenons compte d'un bruit de modèle  $\boldsymbol{\alpha}$ <sup>20</sup> et d'un bruit de mesure  $\boldsymbol{\beta}$ <sup>21</sup>. Pour traiter le problème de l'estimation d'état dans un tel contexte stochastique nous allons utiliser un filtre de Kalman.

Rappelons rapidement la forme classique du filtre de Kalman [22], *ie* lorsque les équations d'état discrètes sont de la forme :

$$\begin{cases} \boldsymbol{x}_{k+1} = \boldsymbol{A}_k \boldsymbol{x}_k + \boldsymbol{u}_k + \boldsymbol{\alpha}_k \\ \boldsymbol{y}_k = \boldsymbol{C}_k \boldsymbol{x}_k + \boldsymbol{\beta}_k \end{cases}$$

où  $\boldsymbol{\alpha}_k$  et  $\boldsymbol{\beta}_k$  sont des bruits aléatoires gaussiens indépendants et blancs, de matrices de covariance  $\boldsymbol{\Gamma}_{\boldsymbol{\alpha}_k}$  et  $\boldsymbol{\Gamma}_{\boldsymbol{\beta}_k}$  respectivement. Le filtre de Kalman travaille sur l'espérance et la variance des variables, plutôt que sur leur densité de probabilité. En particulier, il cherche à annuler l'espérance de la différence entre l'état du système et la prédiction qu'il en fait, et à minimiser la variance de la prédiction. Chaque nouvelle estimation de l'état se décompose en deux étapes successives :

**La correction**, qui consiste à mettre à jour l'espérance de l'état  $\hat{\boldsymbol{x}}_k$  et sa matrice de covariance  $\boldsymbol{\Gamma}_k$  lorsqu'une nouvelle mesure arrive, à l'aide d'un estimateur linéaire orthogonal non-biaisé. Ses équations sont données par :

$$\begin{cases} \hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k \tilde{\boldsymbol{y}}_k & (\text{estimation corrigée}) \\ \boldsymbol{\Gamma}_{k|k} = \boldsymbol{\Gamma}_{k|k-1} - \boldsymbol{K}_k \boldsymbol{C}_k \boldsymbol{\Gamma}_{k|k-1} & (\text{matrice de covariance corrigée}) \end{cases}$$

20. Dans notre cas, le bruit de modèle représente les termes d'ordre supérieur de la formule de Taylor.

21. Le bruit de mesure comprend toutes les erreurs accumulées dans la procédure d'estimation de pose (erreurs de calibration de la caméra, connaissance imparfaite du modèle 3D de la plate-forme, écarts sur la position des points dans l'image, ...).

avec

$$\begin{cases} \tilde{\mathbf{y}}_k = \mathbf{y}_k - \mathbf{C}_k \hat{\mathbf{x}}_{k|k-1} & (\text{innovation}) \\ \mathbf{S}_k = \mathbf{C}_k \boldsymbol{\Gamma}_{k|k-1} \mathbf{C}_k^T + \boldsymbol{\Gamma}_{\beta_k} & (\text{matrice de covariance de l'innovation}) \\ \mathbf{K}_k = \boldsymbol{\Gamma}_{k|k-1} \mathbf{C}_k^T \mathbf{S}_k^{-1} & (\text{matrice de gain de Kalman}) \end{cases}$$

Les notations  $\hat{\mathbf{x}}_{k|k-1}$  et  $\hat{\mathbf{x}}_{k|k}$  désignent l'espérance de l'état avant et après avoir pris en compte la mesure  $\mathbf{y}_k$ . Il en va de même pour  $\boldsymbol{\Gamma}_{k|k-1}$  et  $\boldsymbol{\Gamma}_{k|k}$ .

**La prédition**, qui donne une estimation future de l'espérance de l'état et de sa matrice de covariance, à partir de l'équation d'évolution, avant même d'avoir reçu la mesure correspondante. Elle s'exprime de la façon suivante :

$$\begin{cases} \hat{\mathbf{x}}_{k+1|k} = \mathbf{A}_k \hat{\mathbf{x}}_{k|k} + \mathbf{u}_k & (\text{estimation prédictive}) \\ \boldsymbol{\Gamma}_{k+1|k} = \mathbf{A}_k \boldsymbol{\Gamma}_{k|k} \mathbf{A}_k^T + \boldsymbol{\Gamma}_{\alpha_k} & (\text{matrice de covariance prédictive}) \end{cases}$$

La notation  $\hat{\mathbf{x}}_{k+1|k}$  désigne la prédition de l'espérance de l'état, sans avoir la connaissance de la mesure  $\mathbf{y}_{k+1}$ . Il en va de même pour  $\boldsymbol{\Gamma}_{k+1|k}$ .

Appliquer ces équations à notre système en temps réel n'est pas envisageable du fait des larges matrices que l'on manipule. Cependant, notre système a la particularité d'être stationnaire, *i.e.*  $\mathbf{A}$  et  $\mathbf{C}$  ne dépendent pas du temps lorsque le pas de discréttisation temporelle  $dt$  est constant, et l'on considère que  $\forall k \in \mathbb{N}$ ,  $\boldsymbol{\Gamma}_{\alpha_k} = \boldsymbol{\Gamma}_{\alpha}$  et  $\boldsymbol{\Gamma}_{\beta_k} = \boldsymbol{\Gamma}_{\beta}$ . Les équations du filtre de Kalman peuvent être réécrites dans le cas présent.

$$\begin{aligned} \boldsymbol{\Gamma}_{k+1|k} &= \mathbf{A} \boldsymbol{\Gamma}_{k|k} \mathbf{A}^T + \boldsymbol{\Gamma}_{\alpha} \\ &= \mathbf{A}(\mathbf{I}_{36} - \mathbf{K}_k \mathbf{C}) \boldsymbol{\Gamma}_{k|k-1} \mathbf{A}^T + \boldsymbol{\Gamma}_{\alpha} \\ &= \mathbf{A}(\mathbf{I}_{36} - (\boldsymbol{\Gamma}_{k|k-1} \mathbf{C}^T \mathbf{S}^{-1}) \mathbf{C}) \boldsymbol{\Gamma}_{k|k-1} \mathbf{A}^T + \boldsymbol{\Gamma}_{\alpha} \\ &= \mathbf{A}(\mathbf{I}_{36} - (\boldsymbol{\Gamma}_{k|k-1} \mathbf{C}^T (\mathbf{C} \boldsymbol{\Gamma}_{k|k-1} \mathbf{C}^T + \boldsymbol{\Gamma}_{\beta})^{-1}) \mathbf{C}) \boldsymbol{\Gamma}_{k|k-1} \mathbf{A}^T + \boldsymbol{\Gamma}_{\alpha} \end{aligned}$$

D'après le théorème du point fixe,

$$\lim_{k \rightarrow +\infty} \boldsymbol{\Gamma}_{k+1|k} = \lim_{k \rightarrow +\infty} \boldsymbol{\Gamma}_{k|k-1} = \boldsymbol{\Gamma}$$

est solution de l'équation de Riccati :

$$\boldsymbol{\Gamma} = \mathbf{A}(\boldsymbol{\Gamma} - \boldsymbol{\Gamma} \mathbf{C}^T (\mathbf{C} \boldsymbol{\Gamma} \mathbf{C}^T + \boldsymbol{\Gamma}_{\beta})^{-1} \mathbf{C} \boldsymbol{\Gamma}) \mathbf{A}^T + \boldsymbol{\Gamma}_{\alpha}$$

qui, si elle ne peut pas être résolue analytiquement, peut être résolue numériquement en itérant un grand nombre de fois :

$$\boldsymbol{\Gamma}_{k+1} = \mathbf{A}(\boldsymbol{\Gamma}_k - \boldsymbol{\Gamma}_k \mathbf{C}^T (\mathbf{C} \boldsymbol{\Gamma}_k \mathbf{C}^T + \boldsymbol{\Gamma}_{\beta})^{-1} \mathbf{C} \boldsymbol{\Gamma}_k) \mathbf{A}^T + \boldsymbol{\Gamma}_{\alpha}$$

en partant de  $\boldsymbol{\Gamma}_0$  grand. L'avantage est que  $\boldsymbol{\Gamma}$  peut être pré-calculé. Dès lors, l'expression du filtre du Kalman asymptotique est beaucoup plus simple et ne fait intervenir qu'une quantité moindre de calculs :

$$\hat{\mathbf{x}}_{k+1|k} = \mathbf{A} \hat{\mathbf{x}}_{k|k-1} + \mathbf{A} \mathbf{K} (\mathbf{y}_k - \mathbf{C} \hat{\mathbf{x}}_{k|k-1})$$

avec

$$\mathbf{K} = \boldsymbol{\Gamma} \mathbf{C}^T (\mathbf{C} \boldsymbol{\Gamma} \mathbf{C}^T + \boldsymbol{\Gamma}_{\beta})^{-1}$$

la matrice de gain de Kalman, également calculée à l'avance.

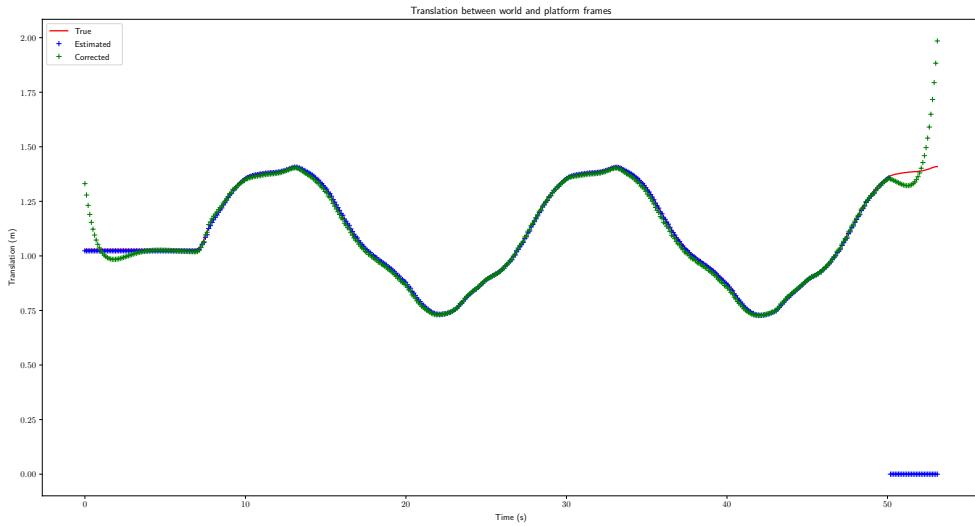


FIGURE 2.13 – Estimation de la translation pour passer du repère du monde au repère de la plate-forme

### 2.4.3 Mise en application et analyse des résultats

Nous avons considéré un modèle dynamique de la plate-forme par rapport au monde. Or, jusqu'à présent, nous estimons la *pose* de la plate-forme par rapport à la caméra. C'est pourquoi nos estimations de *poses* brutes ne peuvent pas être utilisées dans le filtre de Kalman. Il faut d'abord utiliser la connaissance que l'on a de la position et de l'orientation du drone dans l'espace (sa position est obtenue par fusion des données inertielles et du GPS) pour en déduire une estimation de la pose de la plate-forme par rapport au monde.

Quant à l'implémentation pratique du filtre de Kalman, nous avons choisi arbitrairement l'estimation initiale de l'état car nous n'avons pas de connaissance *a priori*, et nous avons pris les variances sur l'estimation suffisamment grandes pour retranscrire cette incertitude. Enfin, nous avons supposé que les bruits sur le modèle et sur les mesures étaient des signaux blancs gaussiens et nous avons précisé une variance plus faible de l'incertitude sur l'estimation de *pose*.

Un tracé de l'évolution de la distance entre l'origine du repère du monde et l'origine du repère de la plate-forme est donné sur la figure 2.13. On compare la distance réelle obtenue en simulation avec la distance calculée avec l'estimation de *pose* uniquement et celle corrigée par le modèle de dynamique de la plate-forme. Comme l'on a basé nos calculs sur le filtre asymptotique et que l'initialisation de l'état de la plate-forme est arbitraire, on observe la présence d'une erreur en début d'expérience qui s'annule rapidement. Ensuite, la *pose* corrigée suit fidèlement la *pose* réelle de la plate-forme, jusqu'au moment où l'on interrompt l'estimation de *pose* (au bout de 50 secondes). Dès lors, seul le modèle dynamique de la plate-forme est utilisé pour estimer la *pose* de la plate-forme. Si le filtre de Kalman renvoie toujours des estimations, celles-ci sont imprécises en raison de la simplicité et de la localité du modèle dynamique utilisé.

# Chapitre 3

## Asservissement visuel

La mission d'appontage d'un hélicoptère sur un navire se décompose en plusieurs étapes : le pilote commence par se rapprocher du navire jusqu'à l'avoir en vue, puis stabilise l'hélicoptère à proximité du navire, pour finalement atterrir lorsque les conditions sont favorables. A l'heure actuelle, cette approche est privilégiée par la majorité des marines [48]. Nous avons fait le choix dans cette étude de nous inspirer de cette séquence de manœuvres humaine pour l'appontage du drone avec l'idée de développer plus tard la partie de décision du moment opportun pour apponter (4). Il est important de noter que peu d'études se sont intéressées à cet aspect, la majorité favorisant la rapidité d'exécution de la mission, plutôt que d'assurer la sécurité de son déroulement.

Ce chapitre se concentre sur la phase de stabilisation du drone au-dessus de la plate-forme mobile. Le drone est supposé être suffisamment proche de la plate-forme, de manière à l'avoir dans son champ de vision en début de manœuvre. Les méthodes sur lesquelles nous allons nous concentrer s'appuient sur l'estimation de la position de la plate-forme dans l'image (Section 2.2) ou dans le monde réel (Section 2.1.2) par vision pour le contrôle du drone. Elles sont connues sous le nom d'**asservissement visuel**.

Après avoir passé en revue les différentes approches qui existent en asservissement visuel (Section 3.1), nous insisterons sur les deux approches classiques (Sections 3.2 et 3.3) et étudierons les améliorations qui peuvent être apportées pour traiter le problème du sous actionnement (Section 3.4).

### 3.1 L'asservissement visuel, plusieurs approches

L'asservissement visuel est le nom donné à l'ensemble des stratégies de contrôle des mouvements d'un robot basée sur la vision. Dans notre cas, nous nous intéressons exclusivement à la situation où une caméra monoculaire est fixée au robot. Ainsi, les mouvements du robot induisent des mouvements de la caméra, et c'est en cherchant à décrire les mouvements de la caméra que nous allons contrôler le robot<sup>1</sup>. S'il existe de nombreuses approches en asservissement visuel pour le contrôle d'un robot à partir des images issues de la caméra qu'il embarque, toutes peuvent être écrites selon un schéma générique [4].

Leur objectif commun est de minimiser une erreur  $e$  définie par :

$$e(t) = s(t) - s^* \quad (3.1)$$

---

1. Il existe d'autres configurations, en fonction du nombre de caméras utilisées et leur position ; ces dernières ne sont parfois pas fixées au robot, mais l'observent de l'extérieur [4].

où  $\mathbf{s} \in \mathbb{R}^k$  désigne un vecteur de caractéristiques visuelles, ou *features*, calculées à partir de l'image courante, et  $\mathbf{s}^* \in \mathbb{R}^k$  des valeurs cibles. Ici, l'on suppose que la position cible à atteindre est fixe, *i.e.*  $\mathbf{s}^*$  est constant, et que les variations de  $\mathbf{s}$  dépendent à la fois des déplacements de la caméra et des déplacements de la plate-forme. Les techniques d'asservissement visuel présentées dans ce chapitre ont à l'origine été conçues pour des applications au contrôle de bras manipulateurs, en particulier pour le positionnement et l'orientation de l'effecteur par rapport à une cible immobile. Dans notre cas, nous souhaitons stabiliser un drone par rapport à une cible mobile. C'est pourquoi nous avons introduit une dépendance entre les variations de  $\mathbf{s}$  et les mouvements de la plate-forme. Les schémas d'asservissement visuel se distinguent par la façon dont le vecteur  $\mathbf{s}$  est choisi. Il peut s'agir de simples indices extraits de l'image, comme de quantités qui résultent d'un calcul intermédiaire. Nous spécifierons les deux cas les plus majoritairement rencontrés dans les sections suivantes. Ensuite, dans l'optique de synthétiser un correcteur en vitesse, il s'agit d'établir la relation entre  $\dot{\mathbf{s}}$ ,  $\mathbf{v}_c \in \mathbb{R}^3$  et  $\boldsymbol{\omega}_c \in \mathbb{R}^3$ , les vitesses linéaire et angulaire de l'origine du repère de la caméra, et  $\mathbf{v}_p$  et  $\boldsymbol{\omega}_p$ , les vitesses linéaire et angulaire de l'origine du repère de la plate-forme :

$$\dot{\mathbf{s}} = \mathbf{L} \left( \begin{bmatrix} \mathbf{v}_c \\ \boldsymbol{\omega}_c \end{bmatrix} - \begin{bmatrix} \mathbf{v}_p \\ \boldsymbol{\omega}_p \end{bmatrix} \right)$$

où  $\mathbf{L} \in \mathcal{M}_{k \times 6}(\mathbb{R})$  est nommée matrice d'interaction et dépend du choix de  $\mathbf{s}$ . Par dérivation de l'erreur 3.1, il vient

$$\dot{\mathbf{e}} = \dot{\mathbf{s}} = \mathbf{L} \left( \begin{bmatrix} \mathbf{v}_c \\ \boldsymbol{\omega}_c \end{bmatrix} - \begin{bmatrix} \mathbf{v}_p \\ \boldsymbol{\omega}_p \end{bmatrix} \right) \quad (3.2)$$

De manière à avoir une décroissance exponentielle de l'erreur, nous posons :

$$\dot{\mathbf{e}} = -\lambda \mathbf{e} \quad (3.3)$$

avec  $\lambda \in \mathbb{R}$  un hyper-paramètre à déterminer<sup>2</sup>. Ainsi, en inversant la relation 3.2 et en injectant 3.3, il est possible d'exprimer la vitesse instantanée de la caméra qui entraîne la décroissance exponentielle de l'erreur :

$$\begin{bmatrix} \mathbf{v}_c \\ \boldsymbol{\omega}_c \end{bmatrix} = -\lambda \mathbf{L}^+ \mathbf{e} + \begin{bmatrix} \mathbf{v}_p \\ \boldsymbol{\omega}_p \end{bmatrix} \quad (3.4)$$

où  $\mathbf{L}^+$  désigne la matrice pseudo-inverse à gauche de  $\mathbf{L}$ . Cette loi permet d'asservir en position une caméra dont on peut commander chacun des six degrés de liberté, raison pour laquelle elle s'est révélée être particulièrement efficace pour le contrôle de bras manipulateurs. Lorsque l'on travaille avec un drone, seuls quatre des six degrés de liberté peuvent être commandés, et cette relation mérite une plus fine attention (Section 3.4).

Nous allons maintenant nous concentrer sur le choix de  $\mathbf{s}$ , l'expression de la matrice  $\mathbf{L}$  associée et les performances que l'on peut espérer d'un tel contrôle (stabilité et précision). Il existe globalement deux grandes approches :

**L'asservissement visuel basé position** ou *Position-Based Visual Servoing (PBVS)*, dont l'objectif est d'annuler une erreur dans l'espace tridimensionnel ;

2. Ce paramètre  $\lambda$  conditionne la vitesse de convergence vers la position cible. Lorsqu'il est trop faible, la convergence est lente et il devient impossible de suivre des cibles en mouvements. Lorsqu'il est trop grand, le système a tendance à osciller autour de la position cible, ce qui peut être néfaste dans certains cas.

**L'asservissement visuel basé image** ou *Image-Based Visual Servoing (IBVS)*, dont l'objectif est d'annuler une erreur directement dans le plan image.

De manière plus illustrée, le but de l'asservissement visuel basé position peut être de produire des consignes en vitesse qui réduisent l'écart entre la position et l'orientation courantes du drone dans un repère donné et une position et une orientation cibles exprimées dans le même repère (Figure 3.1). C'est la solution la plus naturelle et qui justifie nos travaux d'estimation de la *pose* relative du drone par rapport à la plate-forme (Section 2.1.2). D'un autre côté, l'asservissement visuel basé image est plus étonnant mais non moins pertinent. Il se passe de l'étape d'estimation de *pose* et se contente de travailler dans le plan image. Il peut consister à identifier des points particuliers dans l'image qu'il cherche à rapprocher de positions cibles pré-définies dans le plan image (Figure 3.2).

Comme nous allons le voir, ces deux techniques ont chacune leurs avantages et leurs inconvénients (Tableaux 3.1 et 3.2), et de multiples approches dérivées ont vu le jour pour tenter d'étendre leurs performances et leur cadre d'applicabilité. Pour l'asservissement visuel basé image, il est possible d'utiliser des primitives géométriques plus riches que les points, à l'instar des lignes ou des ellipses [6]. Aussi, pour palier l'instabilité dans l'estimation de *pose* qui résulte du bruit dans l'image, il existe d'autres méthodes d'estimation de paramètres 3D. C'est le cas de la décomposition de l'homographie [5, 38]. Enfin, les *features* qui conditionnent le comportement de la caméra dans l'espace tridimensionnel, mais aussi dans le plan image, sont idéalement choisies de façon à ce que la matrice d'interaction soit la plus proche possible de la matrice identité. On se ramène ainsi au contrôle d'un système linéaire, où chaque degré de liberté est associé à une unique *feature*, ce qui induit un parfait découplage et assure la stabilité globale du contrôle en position [6]. Cette situation rêvée est en pratique difficile à atteindre. Des études tentent tout de même de s'en rapprocher ; c'est le cas de l'asservissement visuel dit 2 1/2 D, approche hybride entre les asservissements visuels basés image et position, l'un servant pour le contrôle de la vitesse angulaire et l'autre pour celui de la vitesse linéaire [5, 37]. Pour aller plus loin, on peut également imaginer trouver directement dans l'image six *features* différentes, chacune reliée à un degré de liberté [5].

Citons une dernière approche d'asservissement visuel, basée sur l'utilisation du flux optique, qui se révèle particulièrement prometteuse lorsque la mesure de la vitesse linéaire du drone n'est pas disponible. La connaissance de la vitesse linéaire du drone nécessaire pour asservir le robot en vitesse est classiquement calculée par intégration de l'accélération linéaire obtenue à l'aide d'accéléromètres puis recalée à plus faible fréquence avec les mesures du GPS. Cependant, le GPS ne peut pas être utilisé en intérieur, là où les drones trouvent des applications pour l'inspection, c'est pourquoi des solutions alternatives sont à construire. Le flux optique peut être défini de la façon suivante :

Le flux (ou flot) optique est le champ de déplacement perçu par un observateur en mouvement relatif avec les objets de son environnement [20].

Il est calculé à partir des images uniquement et contient à la fois une information sur la vitesse du drone et sur sa distance aux obstacles<sup>3</sup>. Aussi, couplé à une centrale inertuelle, un capteur de flux optique peut être utilisé pour estimer la vitesse du drone [29, 41]. Cette approche peu utilisée s'inspire du vivant : des insectes, qui ne perçoivent qu'un champ de déplacement, démontrent de remarquables capacités pour atterrir sur des cibles en mouvement, à l'instar des abeilles sur les fleurs. Des résultats pratiques ont prouvé

<sup>3</sup>. Lorsque l'on connaît la vitesse linéaire du drone, on peut estimer la distance aux obstacles à l'aide du flux optique

la pertinence de cette approche d'asservissement visuel originale qui n'utilise que des informations implicites sur la vitesse linéaire du drone [17].

Avant de détailler les deux approches basiques de l'asservissement visuel, il est important de noter que les calculs que nous avons décris représentent initialement l'étape de modélisation, ensuite suivie par une étape de contrôle. Typiquement, dans le cas des bras manipulateurs, systèmes pour lesquels l'asservissement visuel a d'abord été développé, une fois les vitesses à réaliser par l'effecteur calculée, le modèle cinématique inverse est à utiliser pour obtenir les vitesses à envoyer aux actionneurs. Dans notre cas, cette étape de contrôle est directement puisque l'on commande directement les vitesses du drone dans l'espace tridimensionnel.

## 3.2 Asservissement visuel basé position, ou *PBVS*

L'asservissement visuel basé position, où asservissement visuel 3D, est la méthode la plus naturelle lorsque l'on cherche à contrôler le drone, en tant que nous agissons directement sur la position du drone dans l'espace tridimensionnel. Elle justifie également le fait d'estimer la *pose* relative de la plate-forme (Section 2.1.2). En effet, la connaissance de la *pose* de la plate-forme est nécessaire à la stabilisation du drone par rapport à cette dernière. Détailons un des paramétrages possibles pour l'asservissement visuel basé position du drone.

On note  $\mathcal{R}_c$  et  $\mathcal{R}_c^*$  le repère courant et le repère cible de la caméra. La position et l'orientation du repère courant de la caméra par rapport au repère  $\mathcal{R}_p$  de la plate-forme est connue par estimation de *pose*. La position et l'orientation du repère cible est définie à l'avance par rapport au repère de la plate-forme. Soient  $\mathbf{t}_e$  et  $\mathbf{R}_e$  la position et l'orientation relative du repère cible par rapport au repère courant de la caméra (Figure 3.1). Ainsi, notre objectif est de faire tendre  $\mathbf{t}_e$  vers le vecteur nul et  $\mathbf{R}_e$  vers la matrice identité, afin que  $\mathcal{R}_c$  et  $\mathcal{R}_c^*$  coïncident.

Il est possible de choisir le vecteur  $\mathbf{s}$  de caractéristiques visuelles de la façon suivante :

$$\mathbf{s} = \begin{bmatrix} \mathbf{t} \\ \theta\mathbf{u} \end{bmatrix}$$

où  $\mathbf{t}$  désigne le vecteur translation du repère de la plate-forme au repère courant de la caméra et  $\theta\mathbf{u}$  est le paramétrage angle/axe de la rotation entre le repère courant et le repère cible de la caméra, aussi représentée par la matrice de rotation  $\mathbf{R}_e$ . Le paramétrage angle/axe pour la rotation est privilégié pour limiter les singularités. Il vient :

$$\mathbf{s}^* = \begin{bmatrix} \mathbf{t}^* \\ \mathbf{0} \end{bmatrix}$$

où  $\mathbf{t}^*$  est le vecteur translation du repère de la plate-forme au repère cible de la caméra. On obtient (Définition 3.1) :

$$\mathbf{e} = \mathbf{s} - \mathbf{s}^* = \begin{bmatrix} \mathbf{t} - \mathbf{t}^* \\ \theta\mathbf{u} \end{bmatrix} = \begin{bmatrix} \mathbf{t}_e \\ \theta\mathbf{u} \end{bmatrix}$$

On montre que la matrice d'interaction  $\mathbf{L}$  associée à ce choix est donnée par [4] :

$$\mathbf{L} = \begin{bmatrix} -\mathbf{I}_3 & [\mathbf{t}]_\times \\ \mathbf{0} & \mathbf{L}_{\theta\mathbf{u}} \end{bmatrix}$$

avec

$$\mathbf{L}_{\theta u} = \mathbf{I}_3 - \frac{\theta}{2} [\mathbf{u}]_{\times} + \left( 1 - \frac{\text{sinc}(\theta)}{\text{sinc}^2\left(\frac{\theta}{2}\right)} \right) [\mathbf{u}]_{\times}^2$$

où  $[\mathbf{x}]_{\times}$  est la matrice anti-symétrique associée au vecteur  $\mathbf{x} \in \mathbb{R}^3$ . La démonstration de l'expression précédente étant longue, elle est laissée au lecteur par soucis de concision. Ce dernier pourra néanmoins trouver des pistes dans certains papiers [36, 7]. Ensuite, il suffit d'appliquer la loi de contrôle générale 3.4 à chaque pas de temps pour la commande du drone. Notez que la vitesse linéaire et la vitesse angulaire de la plate-forme peuvent être calculées aisément à partir de l'estimation de la *pose* relative de la plate-forme par rapport au drone et de l'état du drone estimé par fusion des mesures inertielles et GPS.

L'asservissement visuel basé position a des avantages indéniables pour le contrôle d'un drone en position. En particulier, le fait de travailler directement dans l'espace tridimensionnel offre des garanties théoriques quant à la stabilité du système. En effet, le noyau de l'inverse de la matrice d'interaction construite dans cette section ne contient que le vecteur nul, ce qui assure que lorsque la vitesse relative de la caméra par rapport à la plate-forme aura convergé vers zéro, l'erreur  $e$  sera nulle. Le problème est que l'erreur que nous avons définie dépend de l'estimation de *pose*. Si la caméra est mal calibrée, ou que le modèle de la plate-forme est mal connu, la *pose* de la plate-forme calculée peut être éloignée de la *pose* réelle. Ainsi, une erreur  $e$  nulle ne signifie pas que le drone a convergé vers la position cible, et nous ne sommes pas non plus en mesure de quantifier l'écart entre celle-ci et la position finale du drone. Enfin, l'on suppose ici que la plate-forme est constamment dans le champ de la caméra. Or, il est difficile d'assurer que les commandes ne feront pas sortir la plate-forme du plan image [4]. Ceci est d'autant plus délicat que le drone est sous-actionné. Les avantages et les inconvénients de cette méthode sont résumés dans le tableau 3.1.

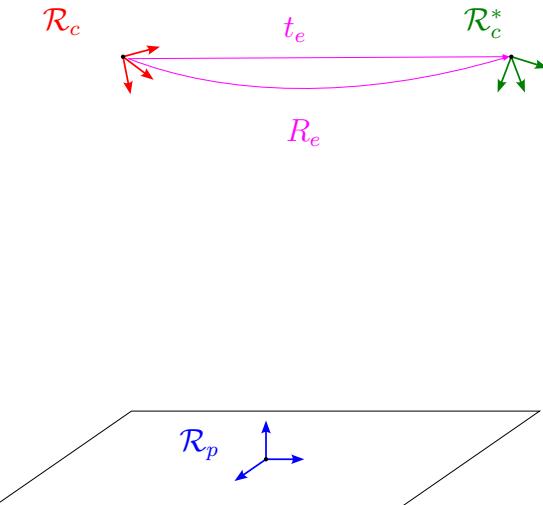


FIGURE 3.1 – Illustration du *PBVS* où l'on cherche à annuler l'écart entre la position et l'orientation courantes de la caméra (repère rouge), et une position et une orientation cibles (repère vert)

Avantages	Inconvénients
Contrôle directement la trajectoire de la caméra dans l'espace cartésien ;	Pas de contrôle sur ce qu'il se passe dans le plan image ;
Convergence globale ;	Modèle de la plate-forme nécessaire ; Erreurs si caméra mal calibrée ou modèle imprécis ;

TABLE 3.1 – Avantages et inconvénients du *PBVS*

### 3.3 Asservissement visuel basé image, ou *IBVS*

Pour les inconvénients cités précédemment, l'asservissement visuel basé position n'est pas la méthode de choix pour le contrôle de drone par asservissement visuel. On lui préfère l'asservissement visuel basé image, méthode qui paraît moins naturelle, mais qui pourtant se révèle d'une grande simplicité de mise en oeuvre et démontre des résultats très satisfaisants.

L'asservissement visuel basé image résulte d'un travail directement dans le plan image. Le calcul intermédiaire de la position et de l'orientation de la plate-forme dans l'espace 3D n'est plus nécessaire et l'on peut faire en sorte de conserver la plate-forme dans le champ de vision de la caméra plus facilement. L'asservissement visuel basé image peut consister à rapprocher des points détectés et identifiés dans l'image de positions prédéfinies, également définies dans le plan image (Figure 3.2). Soit  $\mu$  et  $\nu$  les coordonnées d'un point dans le plan image. Nous avons montré (Equation 2.1) que l'on pouvait relier ces coordonnées dans le plan image, aux coordonnées de ces mêmes points exprimées dans le repère 3D de la caméra :

$$\begin{cases} \mu = \mu_0 + \alpha \frac{x^c}{z^c} \\ \nu = \nu_0 + \alpha \frac{y^c}{z^c} \end{cases} \quad (3.5)$$

où  $\mu_0$  et  $\nu_0$  désignent les coordonnées du centre de l'image,  $\alpha$  est la distance focale de la caméra et  $x^c$ ,  $y^c$  et  $z^c$  sont les coordonnées du point dans le repère de la caméra. Posons :

$$\begin{cases} \tilde{\mu} = \mu - \mu_0 \\ \tilde{\nu} = \nu - \nu_0 \end{cases}$$

En dérivant la relation 3.5 par rapport au temps, nous obtenons :

$$\begin{bmatrix} \dot{\tilde{\mu}} \\ \dot{\tilde{\nu}} \end{bmatrix} = \alpha \begin{bmatrix} (\dot{x}^c z^c - x^c \dot{z}^c)/z^{c2} \\ (\dot{y}^c z^c - y^c \dot{z}^c)/z^{c2} \end{bmatrix} = \frac{\alpha}{z^c} \begin{bmatrix} 1 & 0 & -x^c/z^c \\ 0 & 1 & -y^c/z^c \end{bmatrix} \begin{bmatrix} \dot{x}^c \\ \dot{y}^c \\ \dot{z}^c \end{bmatrix} \quad (3.6)$$

Ensuite, nous pouvons relier la vitesse du point exprimée dans le repère de la caméra, à la vitesse relative de la caméra par rapport à la plate-forme  $\mathbf{v} = \mathbf{v}_c - \mathbf{v}_p$  et  $\boldsymbol{\omega} = \boldsymbol{\omega}_c - \boldsymbol{\omega}_p$  :

$$\begin{bmatrix} \dot{x}^c \\ \dot{y}^c \\ \dot{z}^c \end{bmatrix} = -\mathbf{v} - \boldsymbol{\omega} \wedge \begin{bmatrix} x^c \\ y^c \\ z^c \end{bmatrix} = \begin{bmatrix} -v_x - \omega_y z^c + \omega_z y^c \\ -v_y - \omega_z x^c + \omega_x z^c \\ -v_z - \omega_x y^c + \omega_y x^c \end{bmatrix}$$

où  $\mathbf{v} = (v_x, v_y, v_z)^T$  et  $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)^T$ . En injectant ces expressions dans 3.6, il vient :

$$\begin{bmatrix} \dot{\tilde{\mu}} \\ \dot{\tilde{\nu}} \end{bmatrix} = \begin{bmatrix} -\alpha/z^c & 0 & \tilde{\mu}/z^c & \tilde{\mu}\tilde{\nu}/\alpha & -\tilde{\mu}^2/\alpha - \alpha & \tilde{\nu} \\ 0 & -\alpha/z^c & \tilde{\nu}/z^c & -\tilde{\nu}^2/\alpha - \alpha & -\tilde{\mu}\tilde{\nu}/\alpha & -\tilde{\mu} \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \boldsymbol{\omega} \end{bmatrix}$$

Un point ne suffit pas pour déterminer de manière unique  $\mathbf{v}_c$  et  $\boldsymbol{\omega}_c$ . Il en faut au moins trois, mais pour accroître la robustesse de leur estimation, nous en considérerons plus. Si  $N$  est le nombre de points choisi, nous allons chercher à résoudre le système :

$$\begin{bmatrix} \dot{\tilde{\mu}}_1 \\ \dot{\tilde{\nu}}_1 \\ \dot{\tilde{\mu}}_2 \\ \dot{\tilde{\nu}}_2 \\ \vdots \\ \dot{\tilde{\mu}}_N \\ \dot{\tilde{\nu}}_N \end{bmatrix} = \begin{bmatrix} -\alpha/z_1^c & 0 & \tilde{\mu}_1/z_1^c & \tilde{\mu}_1\tilde{\nu}_1/\alpha & -\tilde{\mu}_1^2/\alpha - \alpha & \tilde{\nu}_1 \\ 0 & -\alpha/z_1^c & \tilde{\nu}_1/z_1^c & -\tilde{\nu}_1^2/\alpha - \alpha & -\tilde{\mu}_1\tilde{\nu}_1/\alpha & -\tilde{\mu}_1 \\ -\alpha/z_2^c & 0 & \tilde{\mu}_2/z_2^c & \tilde{\mu}_2\tilde{\nu}_2/\alpha & -\tilde{\mu}_2^2/\alpha - \alpha & \tilde{\nu}_2 \\ 0 & -\alpha/z_2^c & \tilde{\nu}_2/z_2^c & -\tilde{\nu}_2^2/\alpha - \alpha & -\tilde{\mu}_2\tilde{\nu}_2/\alpha & -\tilde{\mu}_2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ -\alpha/z_N^c & 0 & \tilde{\mu}_N/z_N^c & \tilde{\mu}_N\tilde{\nu}_N/\alpha & -\tilde{\mu}_N^2/\alpha - \alpha & \tilde{\nu}_N \\ 0 & -\alpha/z_N^c & \tilde{\nu}_N/z_N^c & -\tilde{\nu}_N^2/\alpha - \alpha & -\tilde{\mu}_N\tilde{\nu}_N/\alpha & -\tilde{\mu}_N \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \boldsymbol{\omega} \end{bmatrix}$$

En posant

$$\mathbf{s} = \begin{bmatrix} \tilde{\mu}_1 \\ \tilde{\nu}_1 \\ \tilde{\mu}_2 \\ \tilde{\nu}_2 \\ \vdots \\ \tilde{\mu}_N \\ \tilde{\nu}_N \end{bmatrix}$$

et

$$\mathbf{L} = \begin{bmatrix} -\alpha/z_1^c & 0 & \tilde{\mu}_1/z_1^c & \tilde{\mu}_1\tilde{\nu}_1/\alpha & -\tilde{\mu}_1^2/\alpha - \alpha & \tilde{\nu}_1 \\ 0 & -\alpha/z_1^c & \tilde{\nu}_1/z_1^c & -\tilde{\nu}_1^2/\alpha - \alpha & -\tilde{\mu}_1\tilde{\nu}_1/\alpha & -\tilde{\mu}_1 \\ -\alpha/z_2^c & 0 & \tilde{\mu}_2/z_2^c & \tilde{\mu}_2\tilde{\nu}_2/\alpha & -\tilde{\mu}_2^2/\alpha - \alpha & \tilde{\nu}_2 \\ 0 & -\alpha/z_2^c & \tilde{\nu}_2/z_2^c & -\tilde{\nu}_2^2/\alpha - \alpha & -\tilde{\mu}_2\tilde{\nu}_2/\alpha & -\tilde{\mu}_2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ -\alpha/z_N^c & 0 & \tilde{\mu}_N/z_N^c & \tilde{\mu}_N\tilde{\nu}_N/\alpha & -\tilde{\mu}_N^2/\alpha - \alpha & \tilde{\nu}_N \\ 0 & -\alpha/z_N^c & \tilde{\nu}_N/z_N^c & -\tilde{\nu}_N^2/\alpha - \alpha & -\tilde{\mu}_N\tilde{\nu}_N/\alpha & -\tilde{\mu}_N \end{bmatrix}$$

nous retrouvons le schéma générique de l'asservissement visuel exprimé par la relation [3.2](#). De plus, on note :

$$\mathbf{s}^* = \begin{bmatrix} \tilde{\mu}_1^* \\ \tilde{\nu}_1^* \\ \tilde{\mu}_2^* \\ \tilde{\nu}_2^* \\ \vdots \\ \tilde{\mu}_N^* \\ \tilde{\nu}_N^* \end{bmatrix}$$

le vecteur des positions cibles à atteindre dans le plan image. Notez que si la robustesse de l'estimation de la vitesse de la caméra augmente avec le nombre de points, il en va de même du coût pour inverser la matrice d'interaction. En pratique, nous veillerons à ne pas considérer un trop grand nombre de points. Aussi, nous avons dit que cette méthode permettait de nous passer d'estimation de *pose* de la plate-forme ; or la matrice d'interaction dépend de la cote des points dans le repère de la caméra et, lorsque la plate-forme est en mouvement, l'estimation de la vitesse de la plate-forme est également requise.

En réalité, il est possible d'approcher la matrice d'interaction, en remplaçant la cote des points dans le repère de la caméra par la cote des points cibles, qui elle est connue :

$$\hat{\mathbf{L}} = \begin{bmatrix} -\alpha/z^{c*} & 0 & \tilde{\mu}_1/z^{c*} & \tilde{\mu}_1\tilde{\nu}_1/\alpha & -\tilde{\mu}_1^2/\alpha - \alpha & \tilde{\nu}_1 \\ 0 & -\alpha/z^{c*} & \tilde{\nu}_1/z^{c*} & -\tilde{\nu}_1^2/\alpha - \alpha & -\tilde{\mu}_1\tilde{\nu}_1/\alpha & -\tilde{\mu}_1 \\ -\alpha/z^{c*} & 0 & \tilde{\mu}_2/z^{c*} & \tilde{\mu}_2\tilde{\nu}_2/\alpha & -\tilde{\mu}_2^2/\alpha - \alpha & \tilde{\nu}_2 \\ 0 & -\alpha/z^{c*} & \tilde{\nu}_2/z^{c*} & -\tilde{\nu}_2^2/\alpha - \alpha & -\tilde{\mu}_2\tilde{\nu}_2/\alpha & -\tilde{\mu}_2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ -\alpha/z^{c*} & 0 & \tilde{\mu}_N/z^{c*} & \tilde{\mu}_N\tilde{\nu}_N/\alpha & -\tilde{\mu}_N^2/\alpha - \alpha & \tilde{\nu}_N \\ 0 & -\alpha/z^{c*} & \tilde{\nu}_N/z^{c*} & -\tilde{\nu}_N^2/\alpha - \alpha & -\tilde{\mu}_N\tilde{\nu}_N/\alpha & -\tilde{\mu}_N \end{bmatrix}$$

Il a été montré que le système converge avec cette approximation, seule la trajectoire des points dans le plan image est modifiée [4]. Pour autant, la trajectoire des points dans le plan image n'est pas à négliger, car elle conditionne la trajectoire que le drone aura dans l'espace tridimensionnel et peut faire perdre au drone la cible de vue. Pour cela, il est préférable d'avoir une estimation de la cote des points dans le repère de la caméra et d'utiliser  $\frac{1}{2}(\mathbf{L} + \hat{\mathbf{L}})$  comme matrice d'interaction [4].

Enfin, il est possible de calculer la vitesse de la plate-forme sans estimation de sa *pose*. La vitesse de la plate-forme peut s'écrire :

$$\begin{bmatrix} \mathbf{v}_p \\ \boldsymbol{\omega}_p \end{bmatrix} = \begin{bmatrix} \mathbf{v}_c \\ \boldsymbol{\omega}_c \end{bmatrix} - \mathbf{L}^+ \mathbf{e}$$

où  $\mathbf{e}(t)$  peut être approché par différences finies.

Au-delà du calcul de la cote des points dans le repère de la caméra pour améliorer la trajectoire réalisée par le drone, il est important de remarquer que l'asservissement visuel basé image ne garantit pas la convergence du système vers la position désirée. La convergence est assurée, mais il n'est pas certain que le noyau de l'inverse de la matrice d'interaction ne contienne que le vecteur nul. Ainsi, le système peut converger vers un minimum local. Les avantages et les inconvénients de l'asservissement visuel basé image sont récapitulés dans le tableau 3.2.

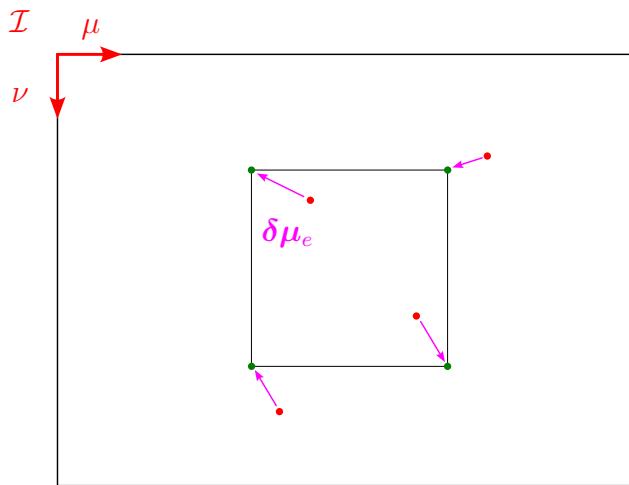


FIGURE 3.2 – Illustration de l'*IBVS* où l'on cherche à annuler l'écart entre la position courante de points de l'image (points rouge) et une position cible (points verts)

Avant de poursuivre, notez que dans le cas où l'on souhaiterait estimer la *pose* de la plate-forme en parallèle de l'*IBVS*, il existe une solution qui engendre de moindres coûts

Avantages	Inconvénients
Pas besoin de modèle de la plate-forme ; Robuste aux erreurs de calibration ;	Estimation de la profondeur des points nécessaires ; Convergence assurée seulement dans un voisinage de la position cible ;

TABLE 3.2 – Avantages et inconvénients de l'*IBVS*

supplémentaires. En effet, de la même manière que nous avions relié la vitesse des points dans l'image à la vitesse relative de la caméra par rapport à la plate-forme dans l'espace tridimensionnel, nous pouvons relier le petit déplacement des points entre deux images au petit déplacement relatif de la caméra par rapport à la plate-forme :

$$\begin{bmatrix} \delta\tilde{\mu} \\ \delta\tilde{\nu} \end{bmatrix} = \begin{bmatrix} -\alpha/z^c & 0 & \tilde{\mu}/z^c & \tilde{\mu}\tilde{\nu}/\alpha & -\tilde{\mu}^2/\alpha - \alpha & \tilde{\nu} \\ 0 & -\alpha/z^c & \tilde{\nu}/z^c & -\tilde{\nu}^2/\alpha - \alpha & -\tilde{\mu}\tilde{\nu}/\alpha & -\tilde{\mu} \end{bmatrix} \begin{bmatrix} \boldsymbol{\delta t} \\ \boldsymbol{\delta\theta} \end{bmatrix}$$

où  $\delta\tilde{\mu}$  et  $\delta\tilde{\nu}$  correspondent au petit déplacement d'un point et  $\boldsymbol{\delta t}$  et  $\boldsymbol{\delta\theta}$  respectivement à la petite translation et à la petite rotation de la caméra dans l'espace 3D [51]. Ainsi, pour éviter d'avoir à effectuer un calcul supplémentaire à chaque pas de temps pour estimer la *pose* de la plate-forme, nous pouvons le réaliser une fois au début, puis additionner les déplacements successifs de la caméra en utilisant la pseudo-inverse de la matrice d'interaction déjà calculée pour l'asservissement visuel basé image. Lorsque l'erreur accumulée à chaque petit déplacement devient trop importante, il est possible de recalculer la *pose* en la calculant à nouveau à l'aide de la méthode classique.

La vidéo d'illustration de la stratégie d'asservissement visuel basé image est disponible [ici](#).

### 3.4 Prise en compte du sous-actionnement

Les techniques d'asservissement visuel ont à l'origine été conçues pour les bras manipulateurs, dont les six degrés de liberté peuvent être commandés. Malheureusement, le drone est un système sous-actionné, car seuls quatre des six degrés de libertés peuvent être commandés indépendamment. En effet, il existe un couplage entre le déplacement longitudinal du drone et le tangage et entre le déplacement latéral et le roulis. Généralement, la commande d'un drone est composée des trois vitesses linéaires et la vitesse de roulis. En raison de ce sous-actionnement, il devient délicat de fournir au drone les vitesses estimées par asservissement visuel comme consigne. Par exemple, avec l'asservissement basé position, le risque est de perdre de vue la plate-forme si le drone s'incline trop lorsqu'il se déplace ; avec l'asservissement basé image, les commandes en vitesses sont faussées, voire en contradiction avec celles qu'on aimerait donner au drone.

Illustrons les limites de l'application naïve de l'asservissement visuel basé image (Section 3.3) au cas du drone.

Alors comment modifier l'asservissement visuel basé image en conséquence ? Le plus simple consiste à utiliser une caméra équilibrée, de façon à ce que ni le roulis, ni le tangage du drone n'engendrent de déplacement de la caméra [11]. Sans cette solution matérielle, d'autres réponses ont été apportées. Il est par exemple possible de retirer de la matrice

d'interaction les deux colonnes associées aux vitesses de roulis et de tangage avant de l'inverser, de manière à éliminer la dépendance du déplacement des points dans l'image en les angles de roulis et de tangage [48]. D'autres approches plus fines ont été imaginées et validées [30, 8].

# Chapitre 4

## Du décollage à l'appontage

Après avoir stabilisé le drone au dessus de la plate-forme, il ne reste plus qu'à apponter. C'est la tâche la plus risquée car une trop forte inclinaison de la plate-forme lors de la descente du drone peut conduire rapidement à l'accident. Notez que nous avons fait le choix de décomposer l'appontage en deux étapes, conformément aux pratiques des pilotes d'hélicoptères : d'abord le drone (ou l'hélicoptère) se stabilise au dessus de la plate-forme et la suit dans son mouvement, puis l'atterrissage est amorcé lorsque des conditions données sont réunies. Des études font le choix de rapprocher le drone de la plate-forme, tout en le faisant descendre, en une seule étape pour gagner du temps sur la manœuvre. Pour des questions de sécurité, et la problématique de décision du moment de l'appontage, nous avons fait le choix de cette décomposition.

Dans cette section, nous présentons comment, d'un point de vue logiciel, nous avons défini les tâches élémentaires que le drone doit accomplir pour apponter, et comment nous les avons reliées de manière logique (Section 4.1). Enfin, nous apporterons de premier éléments de réflexions à la questions de la décision du moment opportun de l'appontage, une fois le drone stabilisé à proximité de la plate-forme (Section 4.2).

### 4.1 Modélisation du comportement du drone par une machine d'état

Jusqu'ici, nous avons vu que l'appontage pouvait être décomposé en deux étapes à haut niveau : la stabilisation du drone et le suivi de la plate-forme, *ie* le *tracking* de la plate-forme, et l'appontage. Lorsque nous considérons que la mission débute avec le drone à terre, nous pouvons ajouter à ces deux phases, le décollage du drone, et l'approche. L'approche consiste à rapprocher grossièrement le drone de la plate-forme, de façon à ce que cette dernière soit entièrement présente dans son champ de vision et que l'asservissement visuel puisse débuter. Nous n'avons pas traité cette phase ici, mais elle peut être réalisée à moindre coût lorsque le drone et la plate-forme disposent d'un GPS et que les deux systèmes ont la capacité de communiquer entre eux : la plate-forme envoie ses coordonnées au drone qui s'en rapproche [11]. Cette localisation de la plate-forme à basse fréquence ne permet pas au drone de suivre dynamiquement les mouvements de la plate-forme, mais elle est suffisante pour l'étape d'approche. Sans ces disponibilités matérielles, une autre solution consiste à explorer l'environnement jusqu'à détecter et identifier la plate-forme dans l'image [49].

Pour résumer, la mission du drone comprend quatre tâches élémentaires :

1. Décollage ;
2. Approche ;
3. Stabilisation et suivi de la plate-forme ;
4. Appontage.

Il faut maintenant relier ces tâches selon une certaine logique. Comme nous l'avons mentionné, ce qui détermine la fin de la phase d'approche, et par conséquent le début de la phase de suivi de la plate-forme est la présence de la plate-forme dans le champ de vision de la caméra. Aussi, si la plate-forme disparaît du plan image au cours de la manœuvre, il est nécessaire de répéter la phase d'approche. Enfin, lorsque le drone est stabilisé à proximité de la plate-forme et que la configuration drone/plate-forme est propice à l'appontage, la consigne d'apponter est donnée. L'ensemble des états successifs du drone et des événements de transition entre eux, peuvent être représentés par une machine d'état (Figure 4.1) [8].

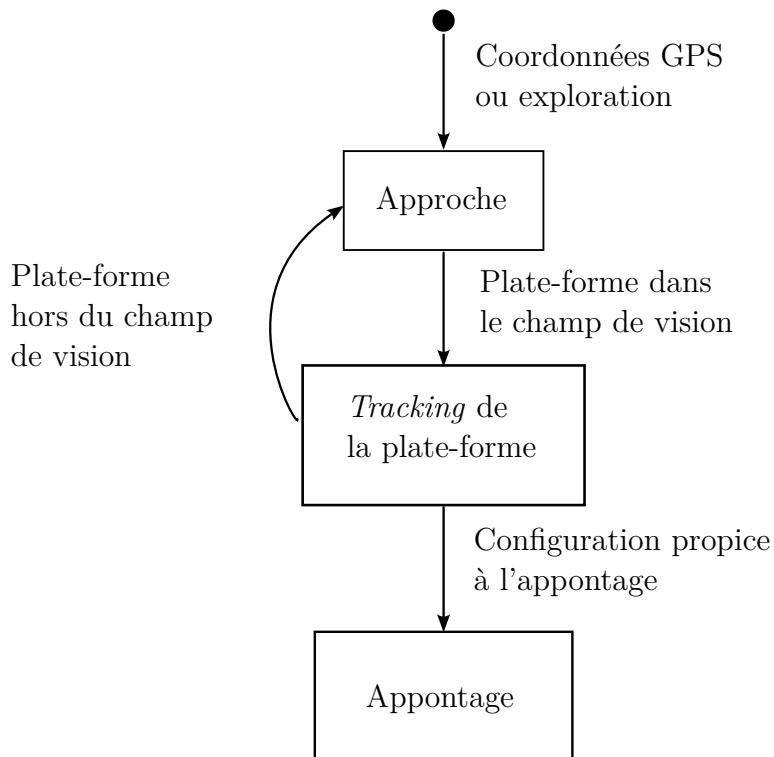


FIGURE 4.1 – Comportement haut niveau du drone

Au-delà de son aide à la représentation et à la visualisation du fonctionnement d'un système, la machine d'état est également d'intérêt pour l'implémentation logicielle haut niveau du comportement d'un système. Dans notre cas, chaque bloc a été codé sous la forme d'un service ou d'une action ROS qui permet son activation ou sa désactivation à la demande. Ces blocs ont ensuite été mis en correspondance sous forme de machine d'état à l'aide du package ROS `smach`.

La vidéo d'une séquence complète d'appontage est accessible [ici](#).

## 4.2 Décision de l'appontage

Le dernier bloc qu'il nous reste à traiter concerne la décision d'apponter. Il cherche tout d'abord à répondre à la question suivante :

Comme caractériser une configuration {drone + plate-forme} favorable à l'appontage ?

Dans un premier temps, nous avons mis en place une approche réactive simple. Celle-ci consiste à détecter le moment où la plate-forme est à l'horizontale à partir des estimations de *pose* et le drone suffisamment proche de sa position cible au-dessus de la plate-forme. Une fois ces conditions réunies, la consigne de descente est envoyée au drone.

Cette approche est réaliste dès lors que les vitesses linéaires et angulaires de la plate-forme ne sont pas trop élevées, relativement à la vitesse de descente du drone. Cette hypothèse est *a priori* vérifiée dans le contexte d'appontage sur un bateau. Pour s'en assurer, il est possible de calculer des statistiques sur les mesures d'orientation de la plate-forme sur un interval de temps, dans le domaine temporel ou fréquentiel après avoir calculé la transformée de Fourier du signal, avant de donner la consigne d'apponter. Néanmoins, une analyse plus fine devrait pouvoir prédire l'état futur du bateau de manière à ce que la configuration favorable à l'appontage intervienne au moment précis où le drone entre en contact avec ce dernier. Une première possibilité est de travailler avec un modèle dynamique de la plate-forme. Il est cependant très difficile de construire un tel modèle (Section 2.4) ; cela nécessite l'intervention d'experts et les calculs à effectuer seront probablement coûteux. Une seconde possibilité préconise l'utilisation des données collectées en ligne pour interpoler l'orientation de la plate-forme. A nouveau, il est possible de travailler dans l'espace 3D [46], ou bien directement dans le plan image, où les prédictions concernent la position future des points caractéristiques dans le plan image. Nous avons fait le choix de travailler dans l'espace 3D, et nous avons cherché à prédire l'orientation de la plate-forme à partir de modèles de mouvements localement linéaires et quadratiques. Les résultats sont données pour une prédiction de l'angle de roulis de la plate-forme une seconde en avance pour un mouvement sinusoïdal de la plate-forme (Figure 4.2).

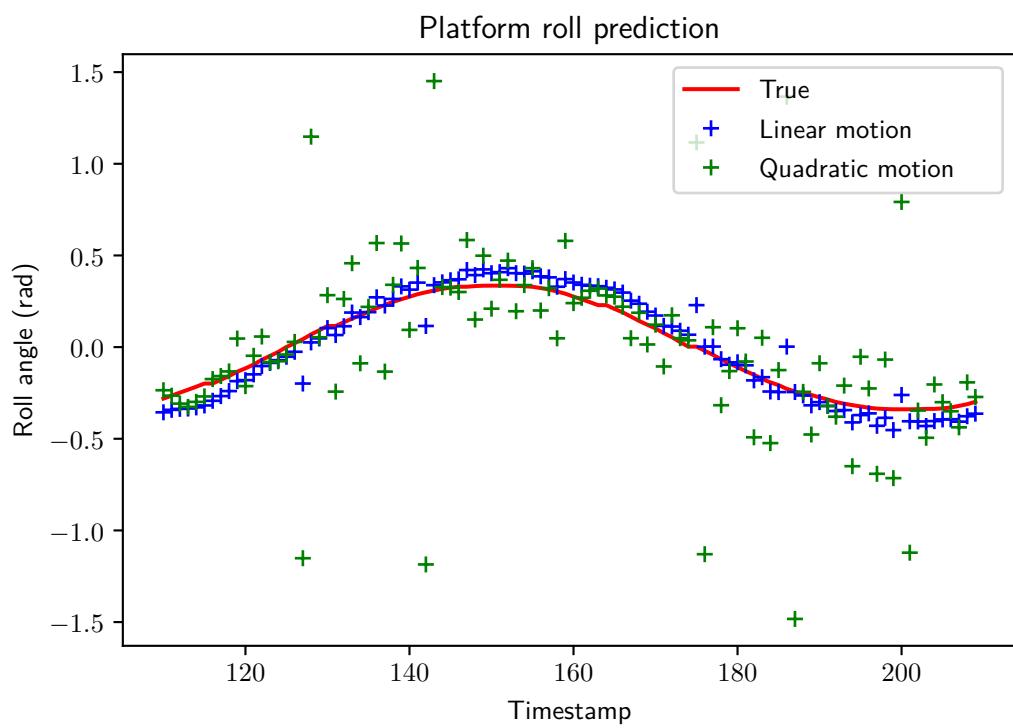


FIGURE 4.2 – Prédiction du roulis de la plate-forme à partir de modèles de mouvements locaux

# Perspectives et conclusion

Ce stage a été l'occasion d'initier un nouveau projet de recherche du laboratoire, celui de l'appontage automatique d'un drone sur une plate-forme mobile. En particulier, les enjeux et les problématiques du sujet ont été mis en avant et des solutions élémentaires ont été apportées pour l'ensemble des phases qui composent la mission d'appontage, du décollage à l'atterrissage. Ces solutions font intervenir des méthodes plus ou moins classiques de vision par ordinateur et de navigation réactive pour la robotique mobile : détection et *tracking* d'un objet plan dans l'image, estimation de *pose*, estimation d'état, asservissement visuel, ou encore prise de décision. De plus, un simulateur a été développé pour faciliter l'implémentation logicielle de nouvelles techniques avant les essais sur le drone réel.

Ainsi, ce stage ne constitue qu'un travail introductif à un sujet large et complexe qui appelle à être poursuivi dans le cadre de stages, et de la thèse sur le sujet qui vient de débuter. Là où mon stage visait à explorer l'état de l'art des approches pour l'appontage automatique, cette thèse reprendra la division en blocs élémentaires de la mission et mettra l'accent sur certains d'entre-eux pour y apporter des solutions innovantes. Une attention spéciale sera notamment portée au traitement du problème du sous-actionnement du drone pour l'asservissement visuel, et à la partie décisionnelle de l'appontage en tirant profit de la logique non-monotone.

Au-delà de ces aspects, d'autres approches sont dignes d'intérêt, à l'instar du développement de schémas d'asservissement visuel plus complexes, qui mêlent les solutions basées image et position, ou de la modélisation plus fine du comportement dynamique d'un bateau. Ce dernier point permettrait en particulier d'étendre le sujet à de nouvelles disciplines, et de croiser les expertises du laboratoire pour apporter des solutions réellement novatrices au problème de l'appontage automatique d'un drone sur un bateau.

Finalement, les applications d'un tel sujet sont diverses. Les solutions techniques présentées ont été choisies avec le contexte d'appontage automatique d'un drone sur un bateau en tête, mais ce cadre peut être étendu à l'appontage sur tout type de support mobile, et à l'aide au pilotage d'hélicoptères.

# **Annexes**

## **A La calibration de la caméra**

## B Les transformations géométriques des images

Une transformation géométrique est une transformation qui modifie la géométrie d'une image<sup>1</sup> sans altérer la valeur de ses pixels [16]. Nous ne parlerons ici que des transformations paramétriques [45] qui appliquent une déformation globale à l'image dépendante de peu de paramètres<sup>2</sup>.

Mathématiquement, la transformation d'une image peut être représentée par une fonction  $f$  qui, appliquée aux coordonnées  $\mu = (\mu, \nu)^T$  d'un pixel de l'image originale, donne les coordonnées  $\mu'$  de ce même pixel dans la nouvelle image :

$$\mu' = f(\mu)$$

Il est cependant courant d'associer aux transformations que nous présentons dans cette partie une matrice  $F \in \mathcal{M}_{3 \times 3}(\mathbb{R})$  telle que :

$$\tilde{\mu}' = F\tilde{\mu}$$

où  $\tilde{\mu}$  et  $\tilde{\mu}'$  sont les expressions en coordonnées homogènes de  $\mu$  et  $\mu'$ <sup>3</sup>.

Décrivons brièvement les transformations les plus importantes, de la plus simple à la plus complexe :

**La translation** déplace l'image originale selon les deux axes de son repère. On introduit un vecteur de translation  $t \in \mathcal{M}_{2 \times 1}(\mathbb{R})$  pour la caractériser.

**La transformation Euclidienne** ajoute à la précédente une rotation  $R \in \mathcal{M}_{2 \times 2}(\mathbb{R})$  de l'image autour d'un axe normal à son plan.

**La similarité** contient en plus un changement d'échelle  $s \in \mathbb{R}$ .

**La transformation affine** autorise en plus l'étirement et la symétrie de l'image initiale selon les deux axes de son repères. Nous utilisons une matrice  $A \in \mathcal{M}_{2 \times 3}(\mathbb{R})$  pour la décrire. Cette transformation nous est bien plus utile que les précédentes car elle permet déjà de modéliser avec précision des régions planes parallèles au plan image d'une caméra et qui se déplacent sous une translation, une rotation autour de l'axe optique ou une petite rotation autour d'un axe parallèle au plan image [35].

**L'homographie** est la plus complète de ces transformations. Nous la représentons par une matrice  $H \in \mathcal{M}_{3 \times 3}(\mathbb{R})$ <sup>4</sup>. Concrètement, l'homographie (ou transformation projective) transforme un quadrilatère en n'importe quel autre quadrilatère. Elle est employée dans beaucoup d'applications en vision par ordinateur et est d'un grand intérêt pour notre étude. En effet, nous pouvons également voir l'homographie comme le changement de point de vue d'une surface plane. Par exemple, pour

1. Ce sont des transformations dans le plan 2D. Toutefois, elles peuvent être étendues au cas des transformations de l'espace 3D ; en particulier, la transformation rigide permet de décrire la *pose* d'un repère par rapport à un autre.

2. Il existe également une autre classe de transformations géométriques, les transformations basées sur le maillage [45], qui permettent de réaliser des déformations locales et contiennent généralement plus de degrés de libertés.

3. Notez que les vecteurs homogènes sont définis à un facteur d'échelle près. C'est un point auquel il faut faire particulièrement attention lorsque l'on travaille avec des homographies.

4. La matrice  $H$  de l'homographie est définie à un facteur d'échelle près. Nous pouvons la normaliser sans perte de généralité (par exemple, la choisir de sorte que sa norme de Frobenius vole 1 ou que sa composante sur la troisième ligne et troisième colonne soit égale à 1). D'où la présence de 8 degrés de liberté et non de 9 (Tableau 1).

notre sujet, les déplacements de la plate-forme plane dans l'image peuvent être modélisés par une homographie.

La figure 3 représente par un symbole chacune de ces transformations tandis que le tableau 1 en résume les propriétés essentielles.

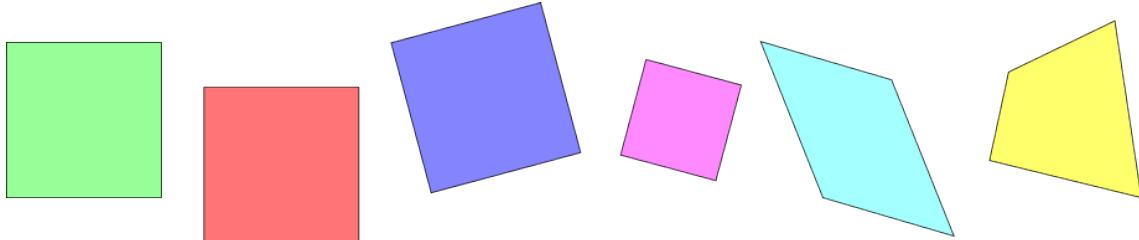


FIGURE 3 – Illustration des différentes transformations géométriques. De gauche à droite : forme initiale, translation, transformation Euclidienne, similarité, transformation affine, homographie.

Transformation	Matrice $F$	Degrés de liberté	Invariants
Translation	$\begin{bmatrix} \mathbf{I} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$	2	Orientation
Euclidienne (rigide)	$\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$	3	Longueurs
Similarité	$\begin{bmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$	4	Angles
Affine	$\begin{bmatrix} \mathbf{A} \\ \mathbf{0}^T & 1 \end{bmatrix}$	6	Parallélisme
Projective (homographie)	$[\mathbf{H}]$	8	Lignes droites

TABLE 1 – Les transformations géométriques

## C Focus sur l'homographie

## D RANSAC

# Bibliographie

- [1] Simon Baker and Iain Matthews. Lucas-Kanade 20 Years On : A Unifying Framework. *International Journal of Computer Vision*, 56(3) :221–255, February 2004.
- [2] Jean-Yves Bouguet et al. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel corporation*, 5(1-10) :4, 2001.
- [3] T. Chateau and J. T. Lapresté. Realtime Kernel based Tracking. *ELCVIA : electronic letters on computer vision and image analysis*, pages 27–43, 2009.
- [4] Francois Chaumette and Seth Hutchinson. Visual servo control. I. Basic approaches. *IEEE Robotics & Automation Magazine*, 13(4) :82–90, December 2006.
- [5] Francois Chaumette and Seth Hutchinson. Visual servo control. II. Advanced approaches [Tutorial]. *IEEE Robotics & Automation Magazine*, 14(1) :109–118, March 2007.
- [6] François Chaumette. Lecture notes in Visual Servoing. GDR Robotique, 2010.
- [7] Vilas K. Chitrakaran, Darren M. Dawson, Warren E. Dixon, and Jian Chen. Identification of a moving object’s velocity with a fixed camera. *Automatica*, 41(3) :553–562, March 2005.
- [8] Gangik Cho, Joonwon Choi, Geunsik Bae, and Hyondong Oh. Autonomous ship deck landing of a quadrotor UAV using feed-forward image-based visual servoing. *Aerospace Science and Technology*, 130 :107869, November 2022.
- [9] Hao Deng, Jing Xiong, and Zeyang Xia. Mobile manipulation task simulation using ROS with MoveIt. In *2017 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pages 612–616, July 2017.
- [10] Davide Falanga, Alessio Zanchettin, Alessandro Simovic, Jeffrey Delmerico, and Davide Scaramuzza. Vision-based autonomous quadrotor landing on a moving platform. In *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, pages 200–207, October 2017. ISSN : 2475-8426.
- [11] Yi Feng, Cong Zhang, Stanley Baek, Samir Rawashdeh, and Alireza Mohammadi. Autonomous Landing of a UAV on a Moving Platform Using Model Predictive Control. *Drones*, 2(4) :34, December 2018.
- [12] David Filliat. Lecture notes in Mobile Robotics. École Nationale Supérieure de Techniques Avancées Paris, 2021.
- [13] Alvika Gautam, Mandeep Singh, Pedda Baliyarasimhuni Sujit, and Srikanth Saripalli. Autonomous Quadcopter Landing on a Moving Target. *Sensors*, 22(3) :1116, February 2022.
- [14] Alvika Gautam, P.B. Sujit, and Srikanth Saripalli. A survey of autonomous landing techniques for UAVs. In *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1210–1218, May 2014.

- [15] Gazebo. Tutorials. Available at <https://classic.gazebosim.org/tutorials>.
- [16] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge University Press, Cambridge, UK, 2nd ed edition, 2004. OCLC : 171123855.
- [17] Bruno Herissé, Tarek Hamel, Robert Mahony, and François-Xavier Russotto. Landing a VTOL Unmanned Aerial Vehicle on a Moving Platform Using Optical Flow. *IEEE Transactions on Robotics*, 28(1) :77–89, February 2012.
- [18] Stefan Holzer, Slobodan Ilic, David Tan, Marc Pollefeys, and Nassir Navab. Efficient Learning of Linear Predictors for Template Tracking. *International Journal of Computer Vision*, 111(1) :12–28, January 2015.
- [19] Stefan Holzer, Marc Pollefeys, Slobodan Ilic, David Joseph Tan, and Nassir Navab. Online Learning of Linear Predictors for Real-Time Tracking. In *Computer Vision – ECCV 2012*, volume 7572, pages 470–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [20] Bruno Hérissé. *Asservissement et Navigation Autonome d'un drone en environnement incertain par flot optique*. phdthesis, Université Nice Sophia Antipolis, November 2010.
- [21] Luc Jaulin. *Mobile Robotics*. ISTE Press Ltd and Elsevier Ltd, 2015.
- [22] Luc Jaulin. Kalmooc : Mooc on the Kalman Filter. École Nationale Supérieure de Techniques Avancées Bretagne, 2022.
- [23] Ren Jin, Hafiz Muhammad Owais, Defu Lin, Tao Song, and Yifang Yuan. Ellipse proposal and convolutional neural network discriminant for autonomous landing marker detection : JIN <span style="font-variant: small-caps;">et al.Journal of Field Robotics, 36(1) :6–16, January 2019.
- [24] Lentin Joseph and Jonathan Cacace. *Mastering ROS for robotics programming : design, build, and simulate complex robots using the Robot Operating System*. Packt, Birmingham Mumbai, second edition edition, 2018.
- [25] Youeyun Jung, Hyochoong Bang, and Dongjin Lee. Robust marker tracking algorithm for precise UAV vision-based autonomous landing. In *2015 15th International Conference on Control, Automation and Systems (ICCAS)*, pages 443–446, October 2015. ISSN : 2093-7121.
- [26] F. Jurie and M. Dhome. A simple and efficient template matching algorithm. In *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, volume 2, pages 544–549, Vancouver, BC, Canada, 2001. IEEE Comput. Soc.
- [27] F. Jurie and M. Dhome. Real Time Robust Template Matching. In *Proceedings of the British Machine Vision Conference 2002*, pages 10.1–10.10, Cardiff, 2002. British Machine Vision Association.
- [28] Olivier Kermorgant and F. Chaumette. Combining IBVS and PBVS to ensure the visibility constraint. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS'11*, pages 2849–2854, San Francisco, USA, United States, 2011.
- [29] Sven Lange, Niko Sunderhauf, and Peter Protzel. A vision based onboard approach for landing and position control of an autonomous multirotor UAV in GPS-denied environments. In *2009 International Conference on Advanced Robotics*, pages 1–6, June 2009.

- [30] Daewon Lee, Tyler Ryan, and H. Jin. Kim. Autonomous landing of a VTOL UAV on a moving platform using image-based visual servoing. In *2012 IEEE International Conference on Robotics and Automation*, pages 971–976, May 2012. ISSN : 1050-4729.
- [31] Vincent Lepetit and Pascal Monasse. Lecture notes in Computer Vision. Ecole des Ponts ParisTech, 2022.
- [32] Antoine Levitt. Lecture notes in Practical Scientific Computing. Ecole des Ponts ParisTech, 2021.
- [33] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2) :91–110, November 2004.
- [34] Bruce D Lucas, Takeo Kanade, et al. *An iterative image registration technique with an application to stereo vision*, volume 81. Vancouver, 1981.
- [35] Yi Ma. *An invitation to 3-D vision : from images to geometric models*. Number 26 in Interdisciplinary Applied Mathematics Imaging, vision, and graphics. Springer, New York, NY, nachdr. edition, 2010.
- [36] Ezio Malis. *Contributions a la modélisation et a la commande en asservissement visuel*. PhD thesis, Université de Rennes, 1998.
- [37] Ezio Malis, François Chaumette, and Sylvie Bouvet. *2D 1/2 Visual Servoing*. report, INRIA, 1998.
- [38] Ezio Malis and Manuel Vargas. *Deeper understanding of the homography decomposition for vision-based control*. report, INRIA, 2007.
- [39] OpenCV. Image Gradients. Available at [https://docs.opencv.org/4.x/d5/d0f/tutorial\\_py\\_gradients.html](https://docs.opencv.org/4.x/d5/d0f/tutorial_py_gradients.html).
- [40] Morgan Quigley, Brian Gerkey, and William D. Smart. *Programming robots with ROS*. O'Reilly & Associates Incorporated, Sebastopol, first edition edition, 2015. OCLC : ocn895728559.
- [41] Matthew B. Rhudy, Yu Gu, Haiyang Chao, and Jason N. Gross. Unmanned Aerial Vehicle Navigation Using Wide-Field Optical Flow and Inertial Sensors. *Journal of Robotics*, 2015 :e251379, October 2015.
- [42] Jose Luis Sanchez-Lopez, Srikanth Saripalli, Pascual Campoy, Jesus Pestana, and Changhong Fu. Toward visual autonomous ship board landing of a VTOL UAV. In *2013 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 779–788, May 2013.
- [43] S. Saripalli, J.F. Montgomery, and G.S. Sukhatme. Visually guided landing of an unmanned aerial vehicle. *IEEE Transactions on Robotics and Automation*, 19(3) :371–380, June 2003.
- [44] Jianbo Shi and Tomasi. Good features to track. In *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 593–600, June 1994. ISSN : 1063-6919.
- [45] Richard Szeliski. *Computer Vision : Algorithms and Applications*. Texts in Computer Science. Springer International Publishing, Cham, 2022.
- [46] Yufei Tao, Christos Faloutsos, Dimitris Papadias, and Bin Liu. Prediction and indexing of moving objects with unknown motion patterns. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 611–622, Paris France, June 2004. ACM.

- [47] Carlo Tomasi and Takeo Kanade. Detection and tracking of point features. *International Journal of Computer Vision*, 9 :137–154, 1991.
- [48] Quang Huy Truong. *Méthodes d’asservissement visuel pour l’appontage d’hélicoptères*. phdthesis, Institut Supérieur de l’Aéronautique et de l’Espace (ISAE), May 2018.
- [49] T. K. Venugopalan, Tawfiq Taher, and George Barbastathis. Autonomous landing of an Unmanned Aerial Vehicle on an autonomous marine vehicle. In *2012 Oceans*, pages 1–9, October 2012. ISSN : 0197-7385.
- [50] ROS Wiki. Tutorials. Available at <http://wiki.ros.org/ROS/Tutorials>.
- [51] Youding Zhu and K. Fujimura. 3D head pose estimation with optical flow and depth constraints. In *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.*, pages 211–216, Banff, Alberta, Canada, 2003. IEEE.