Autonomous Spidercam Motion Control

Student: Hugo Germain

Supervisor: Gabriel J. Brostow

MSc Computer Graphics, Vision and Imaging

University College London

September 2017

Autonomous Spidercam Motion Control

Student: Hugo Germain

Supervisor: Gabriel J. Brostow

MSc Computer Graphics, Vision and Imaging

University College London

September 2017

Abstract

Spidercams have proved to enable elaborate aerial videography and are now widely used in sporting events coverage. Typically deployed over stadiums they can go in any direction and deliver omni-directional viewpoints. In comparison to a movie crane, the Spidercam can be easily mounted at a higher altitude, and cover a much wider range. When filming, Spidercam operators can use it to track distant targets and create cinematic-looking shots. However, performing sophisticated shots with a Spidercam is hard because both the dolly position and the camera orientation have to be hand-steered. In practice, it requires the expertise of multiple operators, and thus comes at an extra cost, coordination and planning. In an effort to minimize expenses and to facilitate trajectory planning, we propose a novel software and hardware to perform autonomous motion control on the Spidercam. We developed a prototype that lets the Spidercam travel along custom paths, while keeping the target in frame, with little to no supervision. More specifically, we let the user select a trajectory support, and the software delivers real-time motor commands to regulate the target position in frame.

This thesis delivers an end-to-end approach to self-driven Spidercams. By leveraging existing deep-learning techniques, we manage to track robustly and in real-time a wide range of targets, from objects to humans or animals. We account for common perturbations like out-of-plane rotations, motion blur or partial occlusion. We replace the typically used dolly with a 360° camera and develop a dedicated user interface. Our solution uses a custom hardware design, and presents a new software to control it. We present a set of motion presets, and evaluate them over different use cases, both in indoor and outdoor environments. Live tracking is achieved over custom spline-based trajectories, and we demonstrate the system's capacities through our self-built prototype.

Lastly, we explore the system's limitation and suggest several paths for future work. We show how promising this project could be if deployed on a larger scale. With more time and budget, one could hope to derive this prototype with even greater precision and high-end cameras. Beyond the sporting industry, emphasis is made on resurrecting the Spidercam to other content, like wildlife documentaries or complex cinema action scenes.

Acknowledgements

I wish to express my sincere gratitude to Pr. Gabriel J. Brostow, for his implication, guidance and enthusiasm, not only throughout this project but also in his teaching of Machine Vision.

I would like to thank my parents, Jean-Philippe and Nathalie Germain, for their encouragement and for allowing me to pursue my studies abroad.

Lastly, I wish to thank the Prism Research Group at UCL, who helped perfect this thesis through their advice and discussion.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of this thesis is to develop a self-driven Spidercam, that can create on-demand cinematic-looking shots with very little supervision. Thanks to its high speed, altitude and wide coverage, Spidercams have proved to enable a wide range of cinematic shots. When filming sporting events, they can hover fields to follow players and travel along complex trajectories. They can reproduce camera movements done by standard movie cranes, while flying on longer distances, at higher speeds and in potentially more complex and narrow environments (over buildings or streets for instance).

In theory, the dolly head (gyro-stabilized camera-carrier) can move in any direction and deliver omni-directional viewpoints, although in practice performing sophisticated shots can be difficult. In fact, the crew controlling the camera pose and position is formed by several experienced operators. This comes at an extra cost, and requires thorough coordination. Therefore, we want to develop an autonomous Spidercam, that would drive itself just as well as an operating crew, and require very little to no supervision. This would not only minimize the typical Spidercam expenses, but also facilitate its control and coordination. Building such a system will be the main goal of this thesis.

Several challenges are raised by this project. First, the Spidercam should be able to follow any moving target within its current viewpoint. That is, the tracking algorithm should be robust and run in real-time. In 2016, the Action Camera project by W. Li et al. [1] implemented a motion control system for a steadicam. However, the tracking algorithm implemented did not account for common perturbations like changes in appearance, lighting or out-of-plane rotations. Our tracking algorithm should account for such perturbations, and still be running in real-time on a powerful laptop. Secondly, the Spidercam should be able to keep the tracked target in frame, regardless of its trajectory. It should move fast enough to follow a variety of moving objects, while traveling along a predefined support. The software should generate precise and responsive motor commands, and be self-regulated. Our system should be affordable, easy to carry, mount and unmount, while providing all the requirements mentioned above. Lastly, the user interface should be intuitive, and provide all the motion controls an industrial Spidercam would, from motion regulation to viewpoint and camera control.

We propose in this thesis an end-to-end approach to self-driven Spidercams, that match all the specifications mentioned above. We implement a novel piece of software and hardware that achieves autonomous motion control in real-time. Our system performs live tracking of any type of moving target, and regulates the motors accordingly to keep it in frame. The user interface we developed lets the Spidercam move along any specified support, and could theoretically reproduce camera motion close to the one of an operating crew.

This novel approach to autonomously motion controlled Spidercams could redefine its sphere of action. Indeed, despite the fact that it is usually rooted in semi-permanent infrastructures, we believe the Spidercam could be employed for a much wider range of filming applications than just sporting events. One could imagine deploying a self-driven Spidercam for wildlife documentaries, following the course of animals over several hundred meters, and hours of autonomous filming. The Spidercam could be used to film timelapses with complex trajectories, provided the motors can move slowly enough. In the longer term, it could even be employed to film action scenes in the cinema industry, tracking fast moving targets more accurately than a typical camera operator.

(a) (b)

Figure 1.1: (*a*) SC250-Field Spidercam model hovering a stadium, manufactured by Spidercam GmbH (*b*) Our system overview: After acquiring the 360° live stream from the camera, we pre-process it and run a tracking algorithm. The new target locations is used to generate velocity commands, which are sent to the motorized units wirelessly to operate on the pulleys.

To satisfy the specifications mentioned above, we make the following hypothesis: using a deep-learning based tracking algorithm, along with a 360° camera and our own hardware and software will allow us to build a self-driven Spidercam at a low cost. We deliver a proof of concept of a self-driven Spidercam that can create complex, tracker-driven shots autonomously. In validating this hypothesis, we reach three objectives:

- We build a working hardware prototype that can be easily deployed in a wide range of environments, by a single person and under an hour.
- We develop a tracking and guidance system for a 360° camera, that enables live feedback and self-navigation for the Spidercam.
- We demonstrate a set of predefined trajectories to allow for accessible and high-level on-demand filming.

1.1 The Spidercam

The Spidercam (or *Spydercam*) is a cable-driven, stabilized system that allows a television or cinema camera to move in space. Typically used in the sports filming industry, it has recorded many football, tennis, basketball and soccer games since its creation in 2003. It was derived from the *Skycam*, a similar system that was originally designed by the Steadicam inventor Garret Brown in 1984. Its main retailer is now Spidercam GmbH[1], located in Germany.

The Spidercam consists of four motorized winches, usually located at the four corners of a stadium, and a dolly which both carries and stabilizes the camera (see figure 1.1). The dolly is hung to four cables, each linked to one of the four reels. The coordinated movement of the motors allow an operator to perform complex camera trajectories in space, while also having full control over the camera orientation and settings. The dolly can move in any direction, and provides camera pan, tilt and roll. On industrial Spidercams, two of the four kevlar cables transmit the 4K video stream and dolly commands through fiber optics communications.

With very few recorded incidents, the Spidercam has also filmed many concerts and worldwide-transmitted events like the Superbowl. The Spidercam's well-establishing makes it even more appealing for us to investigate ways of automating it. However, Spidercams being industrially distributed and owned by private manufacturers, there is very little available documentation regarding their design and software. Part of this project's challenge is to build our own prototype from scratch. More specifically, we have to design both a novel software and hardware, deriving all the command laws, inter-modular transmissions and motor regulation.

[1] http://www.spidercam.tv/

1.2 Tracking

Tracking (sometimes referred to as *video tracking*) is in Computer Vision the process of localizing a moving target within consecutive frames of a video. The target can be of any type, and is typically defined by a *bounding-box*, encircling the object and adding a notion of scale. The usual tracking process is done by defining an initial bounding-box (in a supervised or unsupervised way), and predicting for each subsequent frame the updated bounding-box location, sometimes exploiting prior knowledge on the target position in the previous frame.

Tracking can be found in many applications. For surveillance programs, one may want to track a pedestrian using CCTV cameras. In autonomous driving, we may want to locate multiple obstacles at once, and keep track of each of them individually to predict a correct vehicle trajectory. In gesture recognition, we may want to track specific limbs, along with their orientation. In this thesis, we will only consider the subtask of single-target tracking. Indeed, we want to operate the Spidercam autonomously, keeping a predefined object in frame. We need a fast and efficient tracking algorithm to analyse camera frames and generate appropriate responses.

Performing video tracking is hard for several reasons. First, finding a efficient way of locating the target in the new frame can be difficult. If we only consider a Region of Interest (RoI) centered around the previous target, then we may loose any fast-moving object (*i.e.* going out of the scope of the search region). On the other hand if we process the whole image, then the analysis may be time-consuming. Indeed, it is common for video-tracking applications to want to achieve real-time performance. Efficiency is therefore a key component of such algorithms. Secondly, tracking a moving object robustly can be particularly difficult. Unlike static images, video sequences present a number of artifacts that we have to account for. Namely, camera shake, changes in lighting and appearance, motion blur, occlusion and out-of-plane rotations. Such perturbations require more sophisticated methods that simple template matching. Thus, the video tracking problem can be simplistically summed up to efficiently following a target in a robust and accurate manner.

Illustrious tracking algorithms like the Kernelized Correlation Filter[46] by Henriques et al. or the Adaptive Correlation Filter[45] by Bolme et al. achieve real-time performance, while being robust to some perturbations. However, tracking algorithms have recently taken a new turn with the profusion of deep-learning. Indeed, several methods exploiting the strength of pre-trained convolutional neural networks have achieved outstanding results, outperforming previously state-of-the trackers. Algorithm like the SSAT [13], MDNet [23] or Faster R-CNN[29] (see section 2.4) deliver great performance both in accuracy and robustness, over challenging datasets like the VOT2014[7] or VOT2016[5]. Nonetheless, such accuracy and robustness come at a cost: speed. Indeed, the SSAT or MDNet currently run nowhere close to real-time. A remaining challenge is therefore to find deep-learning approaches that perform at great speed. Currently, few methods tackle this problem.

The tracker we implement in this thesis is the SiamFC[24]. We demonstrate its efficiency over both the VOT2015[6] validation dataset and our own acquired data.

1.3 360° Cameras

Industrial Spidercams are mounted with dolly heads. These camera-carriers both ensure stability and full camera pose control. In an effort to minimize cost as well as the overall head weight, we opt for an alternative: a 360° camera.

360° cameras (or *omnidirectional cameras*) provide (by definition) 360° field of views. Modern models typically consist of two 180° fisheye lenses, mounted in opposite directions. For this thesis, the benefits of using a 360° camera instead of a dolly are twofold: it allows omnidirectional viewing, with no need for motorized units. Controls like panning, tilting or rolling can be reproduced virtually and with more precision on a 360° stream. Because no motors are needed to perform these operations, the camera head is a lot lighter, and at a much lower cost.

We demonstrate in this thesis a 360° live stream processing method for tracking, and put it in application through a prototype user interface.

1.4 Organization

This thesis is at the junction of different fields. It requires knowledge in robotics, computer vision and machine learning. This variety creates an interesting challenge while constituting the richness of this project. This thesis consists of six chapters, this introduction being the first one. The next chapter present the project's related work. Then, the third chapter provides an in-depth understanding of the Spidercam. In this chapter, the reader can learn about the hardware and software architecture. We present design choices, give details on the command laws, algorithms and information transmission across the system. In the fourth chapter, we propose a self-driving approach for the Spidercam. We demonstrate the SiamFC tracker, and introduce a set of predefined camera operations and user interfaces for autonomous motion control. The fifth chapter proposes several experiences to evaluate our Spidercam prototype, as well as the results from these experiments. Lastly, chapter six will conclude this thesis, and provide critical analysis and openings for future work.

We hope the reader will find in this thesis a proof of concept that could potentially redefine the way Spidercams are used today.

Chapter 2

Related Work

This chapter presents an overview of the existing literature in domains relevant to this project. We will first explain the premise of this thesis, giving some background work done on a prescription action camera by Wenbin Li, Peter Hedman and Gabriel Brostow (thesis supervisor). Then, we will present literature occurrences of machine-assisted capture (Section 2.2), as well as aerial filming (Section 2.3). The last section of this chapter will give a thorough analysis of deep-learning based tracking methods (Section 2.4), surveying state-of-the-art algorithms and commonly used techniques.

2.1 Background

This thesis originally stems from the Action Camera rig developed by Wenbin Li, Peter Hedman and Gabriel Brostow. An extensive system description can be found in the paper draft [1]. We can see in the Action Camera project many similarities to this thesis. The system they developed allows a camera operator to be assisted when filming with a steadicam-type setup. The camera-carrier rig does not only acts as a stabilizer, it also controls the camera pan and tilt, and guides the operator movements through a vibrotactile navigation belt. This is particularly useful in the context of TV or cinema filming, as it allows to perform solo-operations, reducing the cost of hiring extra-operators, and facilitating the overall shooting coordination.

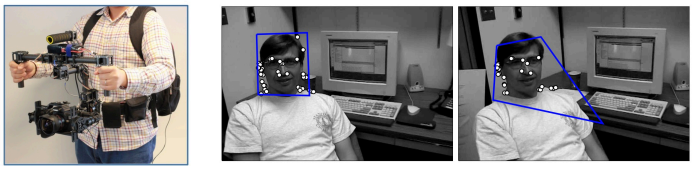
Their hardware enables complex tasks like scene recapturing or active tracking. In the former, a camera operator films an initial outdoors scene. If the operator was to come back days later to re-film the same scene, the system would guide him along a prescribed path and orient the camera orientation to match the original footage. In the latter - the one we are most interested in - the action camera can orient itself to keep a target centered in frame. As the camera operator moves around, the systems tries to relocate the object or person to follow, and the motors regulate the camera orientation accordingly.

Historically, performing active tracking has been studied in numerous projects. We name active tracking the technique of controlling a camera orientation (typically 2 degrees of freedom or more) based on a target location. For instance, one may want to follow a target in real-time and keep it centered in the frame. In practice, orienting the camera and tracking are usually decoupled. The first occurrence of such a method was proposed by Daniilidis et al. [9]., who designed an optical-flow based tracker, linked to a rotating camera. Ever since, trackers have evolved (Smeulders et al. [10], Wu et al. [11]) and have become more robust. Newer systems, like the one described by Kalal et al. [12] include learning algorithms to account for new models learned online while tracking.

Active tracking in the Action Camera currently suffers from several limitations. First, it does not account for common artifacts inherent to video tracking [3]. Perturbations like changes in lighting and appearance, partial or full occlusion, motion blur or out-of-plane rotations are still the subject of many research papers, and are only beginning to be solved by both offline and online machine learning techniques. The tracking algorithm used in the Action Camera is the Consensus-based Matching and Tracking (CMT) algorithm by Nebehay et al. [2]. This algorithm is keypoint-based and works under planarity assumption (that is, the target to track should be planar). When this assumption does not

hold, the tracking algorithm (involving homographies estimation) will fail. However, it is scale and color independent, and can handle deformable objects. Secondly, to compensate for failure cases, targets are pre-captured under different angles. This means that for every new sequence to be filmed, the camera operator has to pre-record the scene at least once. This can come at an extra cost in the context of TV or cinema filming, and is sometimes plainly not feasible.

While this task of active tracking is very similar to what we would like to achieve on the Spidercam, this makes us grasp the need for a robust tracking algorithm.



(a) (b)

Figure 2.1: (*a*) Action Camera rig prototype (figure taken from [1] Brostow et al.) The motorized camera rig was for the most part 3D-printed, and consists of two IMU sensors and three brushless motors for the camera orientation. It can hold a DSLR Camera (Canon 550D) and additionally contains a camera pose controller, a vibrotactile navigation belt (for the ReCapture mode), and a laptop (*b*) Homography estimation failing in the CMT [2] due to target non-planarity (figure taken from [2], Nebehay et al.)

One main challenge of this piece of hardware is dealing simultaneously with camera pose prescriptions and camera stabilization. One has to make sure the camera is stabilized as much as possible, while also orienting the camera towards the selected target. The camera poses are acquired using two IMU receivers. Then, interleaved sequences are computed, resulting from two separate threads dealing with the prescription command and the stabilization. This allows to update the signals at a rather fast pace (70Hz) and low latency (8ms). The effective control sequence is then simply defined as the average between the prescription and stabilization signals. This fast update rate gives the system a lot of responsiveness when performing active tracking and orienting the camera. Such a responsiveness should also be a requirement in the Spidercam.

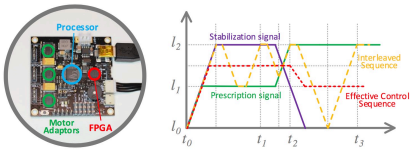


Figure 2.2: Pose Controller generating interleaved sequences (figure from [1], Brostow et al.), the generated signals are averaged from prescription and stabilization signals.

This Action Camera project is a useful background for the Spidercam project, and we can retain several important notions. First, one could see this thesis as an improvement of the active tracking mode, applied to a novel piece of hardware that is the Spidercam. Having seen the limitations of the CMT tracker, we can grasp the necessity for more robustness when implementing our tracking software, especially regarding out-of-plane rotations. Then, we can see why maintaining a fast and responsive system when multiple processes are running is a challenging requirement. This will become a key component when designing and optimizing the Spidercam. Lastly, we hold on to the notion of *prescription*, which is essentially predefined motion controls that the camera should follow based on the tracker response. Just like in the Action Camera project, the flying camera should be able to follow prescribed paths, partially or fully substituting for the need of operator supervision.

2.2 Machine Assisted Capture

Beyond the scope of the Action Camera project, the topic of machine-assisted capture has been covered in many applications. One example of using machines to help capturing images is the stop-motion assisted capturing proposed by X. Han et al. [18]. They demonstrate a video interface capable of allowing animators to record hand-held stop-motion sequences in a continuous fashion. By resorting to keyframe-based capturing and video segmentation and completion, they greatly speed up the animating process, and generate high-quality videos even on an amateur level. A second example of machine-assisted capture is the first-person hyper-lapse video generating algorithm by Kopf et al. [19]. Hyper-lapse videos are similar to timelapse videos, with added camera motion. The proposed algorithm recreates hyper-lapses from first-person videos (often quite shaky) by estimating the 3D camera trajectory, computing a new smoothed path, and generating a new video from stabilized and blended images. A last example of machine-assisted capture is the Video Texture algorithm proposed by Schöld et al. [4]. Through the use of distance matrices, they demonstrate a method to generate seamless looping video sprites from arbitrary inputs. They also introduce the concept *video-based animation*, in which video texture generation is guided by a high-level user interface.

Besides these examples, using machines to facilitate video capture and generation has been used extensively. This thesis is no exception and adds up to this list, using active video tracking to assist the Spidercam motion controls.

2.3 Aerial Systems

Spidercams are often used to perform shots that come close to aerial footage, typically achievable with drones. Srikanth et al. [20] demonstrate how drones can be used for machine-assisted capture, by providing computational rim illumination. Their system was designed to perform path-planning for drones equipped with rim lights, allowing for a specific control of illumination, both in indoors and outdoors setups. The Horus system by Joubert et al. [21] demonstrates an interactive interface to design and execute drone shots autonomously, using Google Maps keyframes and precise timing controls. Horus offers both novice and expert cinematographers an easy access to assisted capturing, resulting in very compelling aerial shots.

Although the Spidercam prototype presented here is designed to film on a much lower scale, this inspires us to create a keyframe-based interactive user interface, to perform complex shots.

2.4 Deep-learning based Video Tracking

Convolutional Neural Networks (CNNs) have recently known a resounding success in performing image-related tasks, and especially those of image classification and object recognition. They have been used to perform single or multi-target video tracking. In these architectures, the role of the neural network is typically to predict bounding-box locations (optionally, its size) in new, subsequent video images. Using deep-learning techniques to perform video-tracking is hard because there is an inherent inconsistence between using a network to predict both an object class and its location [23]. Besides, the strength of CNNs lies in the large size of training datasets. For Visual Tracking, annotated video sequences are far less numerous than single image datasets like ImageNet[1] or COCO[2].

Performing online training is quite computationally expensive. This greatly slows the tracking algorithm at runtime and makes it reliant on offline training. Most existing CNN-based trackers thus heavily rely on pre-trained networks, to compensate for the lack of video training data. If doing so works well for classifying and even locating targets in images, the main challenge remains to stay robust to effects inherent to video sequences. That is, changes in appearance and lighting, object occlusion, out-of-plane rotation, motion blur, and background clutter.

[1] http://www.image-net.org/
[2] http://mscoco.org/

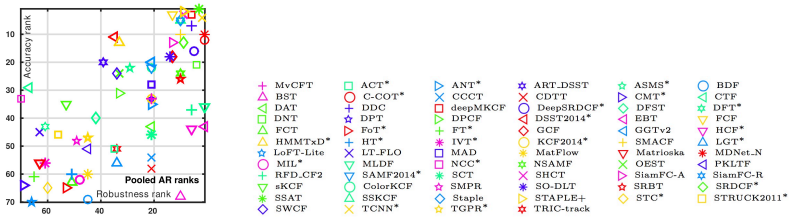


Figure 2.3: The AR-rank plot from the VOT2016 results [5] Highest performing trackers are located in the top right-hand corner of the diagram, while the lowest ones are in the bottom left-hand corner.

To account for those effects, one must come up with a more advanced structure or technique than a standard object classifier. However, complex CNN architectures then quickly become computationally intractable for real-time online evaluation.

In this section we will try to provide the reader with common notions when performing deep-learning based video tracking, among several literature examples. First, we will introduce the Visual Object Tracking Challenge, which annually benchmarks the highest performing trackers and is usually a good indicator of state-of-the-art methods. Then, we will review three different algorithms, present their strength but also their limitations. Lastly, we will discuss the Speed-Accuracy Trade-off of deep-learning based video tracking.

2.4.1 The Visual Object Tracking Challenge

Every year holds the Visual Object Tracking (VOT) challenge, which ranks submitted Video Trackers, and provides training and validation datasets. Its importance has grown over the past years, and it has now become a reference in evaluating trackers, providing benchmarks, and discussing the main aspects of each method. Their results are published and publicly available ([5],[6],[7]).

Their main metric of ranking is the Accuracy-Robustness plot (or AR-plot). The *accuracy* of a tracker is defined as the overlap (intersection over union), between its predicted bounding-box and a ground truth. As explained by Kristan et al. in their recent VOT2016 tracking measurement submission [22], it can be mathematically defined as :

$$\phi_T = \frac{A_t^G \cap A_t^T}{A_t^G \cup A_t^T}, \text{ with } \begin{cases} A_t^G : \text{ the Area of the Ground Truth at time t} \\ A_t^T : \text{ the Area of the Tracker at time t} \end{cases}$$

The *robustness*, on the other hand, is defined as the frequency of failure of the tracker on a given sequence (i.e. each time $\phi_t = 0$). These metrics can then be plotted and deliver a visual layout of highest performing video trackers according to these two criteria.

Ideally, we would like to use a tracker that is robust, accurate and fast. Top results from VOT2016 [5] amongst CNN-based methods include the MLDF Tracker (no.1 in Robustness [13]), and the SSAT Tracker (based on the Multi-Dimensional Neural Net [23]).
These two trackers achieve outstanding results but are particularly slow ($< 1fps$). Nonetheless, they introduce key notions and will help grasp the difficulty of building a tracker that is both fast and robust.

2.4.2 Deep-Learning Based Trackers Review

To understand the whys and wherefores of deep-learning based video tracking, one should first consider surveying state-of-the-art architectures. Here, for historical reasons, we will first present the Faster-RCNN, which acts as a fast object detector. Then, we will briefly study some of the most recent CNN-based tracking algorithms, that proved to deliver very high performances at the VOT2016 Challenge [5] in terms of robustness and accuracy. Top results from VOT2016 amongst CNN-based methods include the MLDF Tracker (no.1 in Robustness [13]), and the SSAT (based on the Multi-Dimensional Neural Net [23]).

Faster R-CNN

The Faster R-CNN algorithm by Ren et al. [29] is a fast object detector, which is an extension of the Fast R-CNN [28], itself being based on the R-CNN [16] object detector.

R-CNN, which stands for Region-based Convolutional Neural Network (CNN) and proposed by Girshick et al. [16], is a Neural Net which performs object detection by feeding region proposals to a pre-trained CNN. The CNN extracts a fixed-length feature vector, which is then classified by a class-specific linear SVM, to rank the region with the highest score (greedy non-maxima suppression). The estimated target is then refined using bouding-box regression [32]. The approach used here to perform localization is the "Recognition using regions" paradigm, as described in [30]. We treat the localization problem as a regression problem, that is outputing 4 values for each input image (x and y bounding-box coordinates, width and height). For every given image, we extract domain-independent region proposals, using a method called *Selective Search*. This method, described in [31], relies on superpixel merging along with a trained SVM, to output class-independent regions. In opposition with the more standard "sliding-window" approach, it allows to only retain both relevant and diverse regions and makes the overall classification more efficient. Using an image warping function, we feed fixed-sized regions into a neural net (5 convolutional and 2 Fully convolutional layers). We extract a low-dimensional feature vector, which is then fed into the SVM region classifier. To account for domain-specific regions and include new domains, we continue learning while testing using a Stochastic Gradient Descent on gathered minibatch samples (positive and negative). The main issue with the R-CNN structure is that it is a "multi-stage pipeline" [28]. The typically used softmax function is replaced here by an SVM, which means a lot more storage internally, and heavier computation. Moreover, it was not designed to be fast, and performs on average at 47s per image. This is mostly due to the fact that the computation is not shared when making a forward pass for each region proposal.

The motivation behind the Fast R-CNN [28] paper by Girshick et al. was therefore to build a Fast version of the R-CNN algorithm. The idea is to rely instead on a Spatial Pyramid Pooling Network (or SPPnet [33]) as the main CNN structure, and feed the extracted features to a "RoI pooling layer". This layer allows to feed fixed-size inputs to the fully connected layers, which can then extract the region score and its regressed bounding-box. A key component that makes this network faster is that unlike previously, the computation is shared across the whole network. Indeed, the SPPnet revolves around using a single shared feature layer computed from the whole image. The feature vectors are then extracted from this shared layer. Thanks to this structure, the training can be performed in a "single-stage" [28], by using a multi-task loss function.

The most recent evolution of Fast R-CNN is called Faster R-CNN [29] and was proposed by Ren et al. Again, the motivation behind this algorithm was to make it closer to real-time performance. It achieves higher performance for an average per-frame running time of 10ms. The main difference with the previous iteration is that it replaces the region proposal algorithm (selective search [31]) with a dedicated Region Proposal Network (RPN). That way, we can merge more seamlessly both networks and obtain faster results.

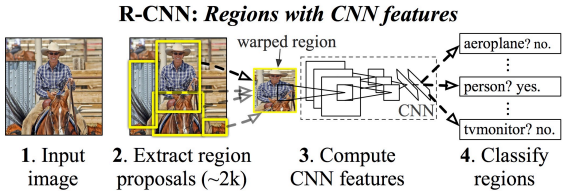
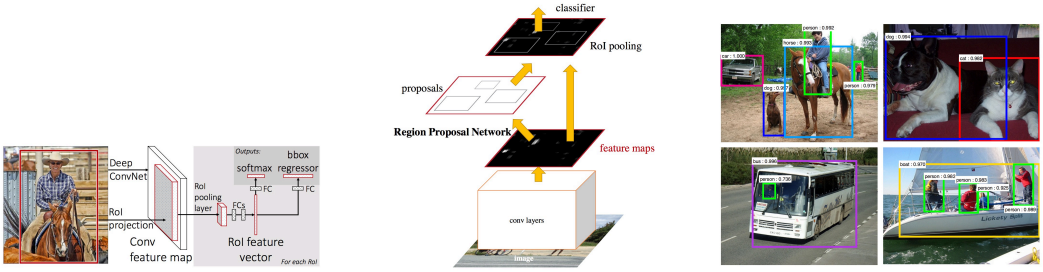


Figure 2.4: R-CNN tracking framework, from [16]. The R-CNN acts as a fast object detector rather than properly a video tracker. Extracted and warped regions of interested are fed into a pre-trained CNN classifier, combined with an SVM.



(a) (b) (c)

Figure 2.5: (*a*) Fast R-CNN architecture from [28]. By resorting to a RoI Pooling Network and a Spatially Pooling Network to share the computation, the computation is made a lot faster (*b*) Faster R-CNN Architecture from [29]. This time a Region Proposal Network (RPN) is used to extract and merge additional tracking information from each image (*c*) Example detections using RPN proposals on PASCAL VOC 2007, from [29]

The Faster R-CNN has achieved for several months state-of-the-art performance both in terms of speed, robustness and accuracy. Nonetheless, there are a couple of issues inherent to addressing the visual tracking problem using such a network. First, it does not account for temporal consistency. Each new frame is treated independently, which means that targets can jump from one location to the next with a lot of jittering, which is not ideal when tracking a target for a motorized camera rig. Moreover, it does not take into account effects inherent to video sequences, such as motion blur, occlusion, or changing in lighting conditions. It was designed and trained to work on photographs, and newer models (like the MLDF [13] or MDnet [23]) achieve better results by differentiating domain-specific and domain-independent information. Faster R-CNN does not learn either from new targets online, which means any new object that does not belong to a pre-trained class will not be detected.

As a result, the Faster R-CNN still performs very well as an object detector, introduces compelling notions for video tracking with CNNs, but is not the most up-to-date algorithm to be used in visual tracking anymore.

MLDF

The MLDF Tracker, or *Multi-Level Deep Feature Tracker* [13], was submitted to the VOT2016 challenge and was ranked first in robustness among the CNN-based trackers. It relies on a pre-trained VGG Network [34], whose output features are then fed into a Multi-Level Network (MLN). This network can then predict the bounding-box location, while handling effects like scale-variations.

The VGG Network is at the basis of this tracker. It comes from a paper [34] published by Simonyan et al. in order to study the impact of the depth of a neural network (i.e. the number of layers) on its performance. The point they make is that using a deeper network with smaller receptive fields will lead to higher-performance. This is due to the fact that adding more layers means we can insert more ReLU filters in between, which make the network more discriminative when classifying. Moreover, using smaller receptive fields and a deeper structure results overall in less parameters to compute. This makes the global training and testing faster. Training on VGG is done by using a standard multinomial logistic regression, and mini-batch gradient descent along with momentum (weights from previous descent steps). Thanks to a multi-scale training phase (input images are rescaled), it can account for objects of various sizes when classifying.

The main concept behind the MLDF architecture, is to separate high-level from low-level semantic information [13]. By feeding an image into the VGG Network, we an extract two feature maps, one which is coming from a top-level layer, and another from a lower level. The top level one is robust to appearance change, and encodes more *global* features. It performs well for distinguishing different classes of objects, but will fail to discriminate two objects from the same class. The lower-level one, on the other

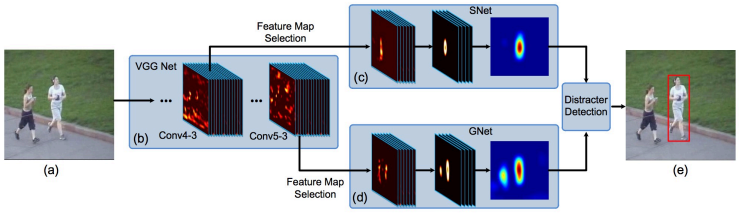


Figure 2.6: MLDF Architecture by Wang et al. [13] High and low-level feature maps are extracted from the VGG Network and fed into two neural networks : *SNet* and *GNet*. Each network processes the feature maps, and either one of them is used to generate the bounding-box prediction

hand, encodes more *detailed* features, and can discriminate two objects from the same class much better. However, it will fail when effects like deformation or occlusion occur.

The idea is then to switch between using the upper or lower layer to predict the target position. Each layer is inputed to a specific network (SNet and GNet). To predict a target location, we feed a Region of Interest (RoI) based on the previous target location and extracted saliency maps [35]. Unlike the Faster R-CNN, taking into account the previous target location allows for more consistency between each frame, and avoids phenomenons like jittering. Besides, the concept of having a multi-level structure can make up for object occlusions and changes in appearance, while also distinguishing co-occuring objects.

Overall the MLDF performs very well in terms of Accuracy (index of 0.490 [5]), and Robustness (index of 0.233 [5]). This can be explained by a much more complex structure, which aims at differencing global and local information. However, it performs very poorly in terms of speed. This is the main issue with this tracker. In the described experiment [13] the tracker was implemented on a 3.4GHz CPU PC with a Titan GPU, and can run at 3fps. This is very far from real-time, and therefore not ideal for us if we want to have a responsive system. The main reason for this low speed is that the pipeline presented here is complex (3 separate networks), and each part of it has to be optimized separately. We have two networks on top of a single one, bounding boxes to generate, and so on. Therefore, while this tracker may be performing very accurately and robustly, it may not be the ideal solution for our implementation, in terms of speed. It is nonetheless interesting to see how a multi-level structure can cope with issues inherent to video sequences and visual tracking.

We now proceed to study the SSAT Tracker, which also also relies on multi-dimensionality and was ranked first in Accuracy at VOT2016 [5].

SSAT

The SSAT (for *Scale-and-State Aware Tracker*) by Qi et al. [5], was ranked number one in accuracy at VOT2016 among the CNN-based trackers. As the MLDF tracker, it uses Multi-Dimensionality to account for a higher semantic level of information. The main difference is that multi-dimensionality is used here rather to distinguish domain-specific and domain-independent information.

Indeed, the main motivation behind this tracker is to learn from video sequences that present lots of different characteristics, not only in terms of object labels, but also in their appearance, lightings, occlusion, and motion. As in the previous models, we start by training a CNN to perform classification, and then use this information to perform tracking. Over multiple video sequences, we can gather domain specific information (*e.g.* object classes), and domain-independent information (*e.g.* overall lighting conditions). This distinction can be expressed in the CNN structure used by the SSAT tracker.

The SSAT is in fact a simple extension of the MDnet [23] by Nam et al., which was ranked first in accuracy and robustness at VOT2015 [6], and can be described as follows. A first set of shared layers, adapted at each training iteration, embody the domain-independent information. It can be seen as a smaller version of the VGG [34] network. Then, a set of K separate branches, which are all trained independently and iteratively, encode domain-specific information. They act as binary classifiers, using a simple softmax cross-entropy loss function.

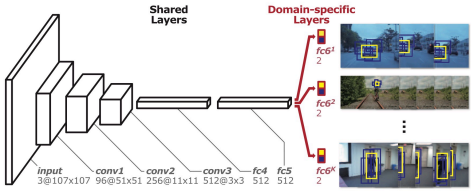


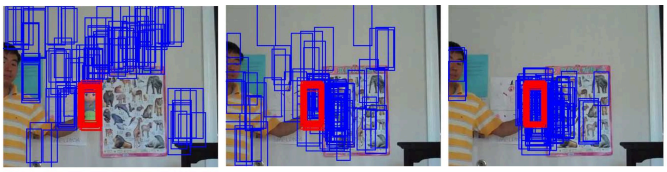
Figure 2.7: Multi-Domain Network (MDnet) Architecture at training, by Nam et al. [23] By using a combination of domain-independent and domain-specific layers, the tracker achieves high robustness at test time

Training is done using a standard Stochastic Gradient Descent, performed independently on each branch. As explained by Nam et al., once the training is done we can perform tracking by constructing as new binary classifier, and feeding in regions of interest (RoI) sampled around the previous target location. From there we simply retrieve the highest scoring target using :

$$\boldsymbol{x}^* = argmax_{x^i}(f^+(x^i))$$

The performance of the MDnet can be improved by making a couple of adjustments. First, we can perform long and short-term updates. The long-term updates are performed at a predefined rate, and renew the network weights, resulting in more robustness. The short-term updates are performed anytime the tracker accuracy goes below a certain threshold (e.g. $f^+(x^*) < 0.5$), and allow for more adaptiveness. More specifically, they can allow the network to adapt to changes in appearance, as for instance out-of-plane rotations. This aspect is particularly useful when comparing it for instance to the pre-existing CMT tracker, which doesn't allow for such modifications. Both updates are done using respectively long-term and short-term training samples. On top of that, we apply hard mini-batch mining [36]. This technique consists in sampling negative samples around the selected positive samples (target), to update the network. This makes the MDnet more discriminative in its classification, and avoids effects like target drifting over time. Lastly, we use a bounding-box regression algorithm , to estimate a more accurate target location from the highest-scoring target proposal. It consists in training a simple linear regression model to predict a new target location (x,y) and bounding-box size (w,h). It was initially used by Felzenszwalb et al. [32] and more widely reused ever since. In the MDnet implementation we only train the bounding-box model at the initial frame to make the computation run faster.

The SSAT tracker adds one more evolution to the MDnet structure, by training a state model, to infer on target occlusion. Images whose target are classified as occluded will be discarded when training. This allows for an even more accurate network, making the SSAT first in accuracy at VOT2016 [5]. The SSAT visual tracker performs very accurately [5], and quite robustly too (0.291). Moreover, this tracker is among the rare ones that make an effort to account for characteristics unique to video sequences, by training two separate sets of layers. The tracking is also performed using previously estimated targets, allowing for a certain consistency across the frames. However, as we could be expecting with such a complicated structure, this framework performs very slowly. It was not designed for speed, and actually



(a) 1st minibatch (b) 5th minibatch (c) 30th minibatch

Figure 2.8: Hard-Minibatch mining, from [36] By collecting positive and negative samples online we update the network's weights and gradually improve its robustness for sequence-specific tracking.

delivers slower results than the MDLF Tracker. This again is quite problematic for us, and we can begin to sense the trade-off there is between having a complex and high-performing network, versus a simpler but faster architecture.

Ideally, we would need to have a structure that performs as fast as possible while preserving accuracy, in order for our prescriptions to follow the target close enough and exploit the hardware high update frequency [1]. Nonetheless, we also need to have a robust algorithm and allow for instance for out-of-plane rotations.

2.4.3 Speed-Accuracy Trade-off

If these existing models have proved to deliver a high-accuracy and robustness, they perform rather heavy operations which prevents them from operating in real-time. Indeed, their complex Neural Network structure (deep networks, fully-connected layers, multiple nets being used) is very computationally expensive at test time, due to very high-dimensional operations. Besides, performing forward and backward (SGD) passes on batches of RoIs greatly slows down the tracking , and any kind of additional operation like bounding-box regression becomes quickly intractable in real-time. Moreover, they treat the tracking problem as a classification problem, searching for occurrences of class-specific objects or trying to discern the foreground from the background, thus discretizing the span of possible targets. Lastly, there is a trade-off between the precision of the tracker and its computational time linked directly to the amount of RoIs sampled. To overcome these limitations, alternative approaches must be introduced.

A simpler structure for instance, would allow for single-shot evaluations and make the training and implementation easier. The question remaining is whether we can afford to loose accuracy in the profit of a faster algorithm for this thesis. As detailed in Section 3.1.1, the tracker will have to run on a laptop. If trackers like MLDF or SSAT already run very slowly on a decent desktop PC with high-end GPUs, we can only expect it to be worse on smaller devices. Besides, the motors having a refresh rate of about 20Hz, using a slow algorithm would create a bottleneck, and could very easily mean that the target we follow could go out of the frame. If having our target slightly off-centered by a few pixels is acceptable, loosing our target while shooting is not. Therefore speed is definitely a decisive factor, and picking a slow but very accurate tracker does not seem to be a wise choice for the Spidercam. In this thesis we will be using the SiamFC tracker, which is a deep-learning based tracker that was designed for speed and has proved to deliver high performances. It will be introduced in greater details in Section 4.1.

Chapter 3

The Spidercam

One could see this Spidercam project as the union of three distinct parts: designing and assembling the hardware, ensuring a proper communication between every module, and finally deriving one or several tracker-based or tracker-independent command-laws. This chapter will cover the first two parts, detailing the Spidercam building process. With very little to no references of a similar system built before (at this scale), we have to come up with all the hardware and software-related decisions, and derive most equations by hand.

In this chapter, we will present the Spidercam overall architecture, covering both the hardware and software design. We will first present an overview of its functioning, as well as its design choices and sizing. Secondly, we will present the software architecture, including the ROS framework, User Interface (UI), network layout and motor command law. We will explain difficulties encountered when dealing with multiple streams, as well as the solutions we found. By the end of this chapter, the reader will have a good understanding of the Spidercam functioning. The following chapter will exploit this prototype to implement a tracker-based autonomous-flying software.

3.1 Hardware Architecture

We begin by presenting the hardware side of this project. This includes the design, dimensioning, and overall functioning of the system. We will explain the reasoning behind each decision, and hopefully provide the reader with a full understanding of the Spidercam architecture.

3.1.1 Specifications

Industrial Spidercams are usually designed to hover a stadium, with massive winches weighting around $700lbs$ each and fully stabilized camera-carriers. For the purpose of this research project we have to come up with a smaller-sized, portable and affordable structure. We first define a range of specifications for our system to respect, which will help us create a first design for the Spidercam:

- **Dimensions:** The system should be extensible in width, height depth, with virtually no upper limit to the range to which it could operate. Theoretically, we should be able to mount the Spidercam on any support, across parks, buildings, outdoor or indoor environment (provided it meets all necessary safety regulations). It should be portable, easy to carry, mount and unmount.
- **Weight and Speed:** The system should be able to carry a lightweight camera (around $500g$), at a controllable speed. It should be fast enough to hover targets such as slow animals, and slow enough to perform videos such as timelapses.
- **Coverage:** The camera should be able to travel the whole volume defined by the four motorized units and their orthogonal projection on the floor. The camera should also be able to point in any given direction.
- **Stability:** The system should present as little vibration as possible, so that there should be no visible camera shake on the footage. While doing so, the system should maintain the highest speed achievable.
- **Trajectory:** The camera should be able to travel along any predefined trajectory within the flying area. It should be controllable in velocity. The user should be able to input (x, y, z) velocity vectors for the camera to follow, as well as (ϕ, θ) viewpoint angles.
- **Versatility:** The system should be easily adaptable to any configuration, both physically (*i.e.* width, height and depth) and in its control (trajectory inputs). It should provide a user-friendly interface.
- **Autonomy:** The system should be able to operate on its own, guided by a predefined set of command or trajectory, and actioned by a responsive tracking algorithm.
- **Supervision:** At all time the operator should have full control to start and stop the system. In case of emergency, he must be able to take control over the tracking algorithm, or quickly stop the motors.
- **Safety:** We will assume that the flying area is completely unobstructed, and that both the camera and the cables can move freely within that space with no risk of damaging the hardware. Any operator located outside of the flying area should not be put at risk in any way.

3.1.2 Design

Given the available space at our disposition to run experiments and the budget allowed for this project, we choose a tripod-based system. Using tripods comes with multiple advantages. One can easily move them around, adjust their height, and transport them where necessary. They fit in a room, and offer standard mounts to attach a motorized unit at its top. Lastly, they are stable enough to ensure a robust structure. We opt for a set of 4 large tripods, with an height reaching up to 3 meters.

At the top of each tripod, we attach a motorized unit. This unit must be able to receive orders from a laptop, and action on a winch accordingly, to pull or release the cable as needed. The camera mount is then attached at the junction of the four cables, as shown on the figure below. If the motors operate on the cables in a synchronized and coordinated fashion, one should be able to move the camera in any direction within the flying area and at a height below the tripod heads.

(a) Front View (b) Top View

Figure 3.1: Initial 360° Spidercam concept sketches: Four tripods hold a motorized unit in charge of pulling or releasing a cable linked to the camera mount. The camera position is thus defined by the action of the motors which, coordinated, allow for any trajectory within the volume defined by the tripods.

Camera Choice

Industrial Spidercams are equipped with a Dolly head, on which is typically mounted a TV Camera. This Dolly provides the operator with full control on the pan and tilting on the camera, while delivering smooth motion and acting as a stabilizer.

For this project, it is part of our requirements that we should be able to look in any given direction (as a real Spidercam would). However, mounting a stabilizer head (as the one used in the Action Camera project [1]) comes with several limitations. First, it greatly increases the weight of the flying unit. This results in the need for stronger motors, overall structure, and larger costs. Then, these stabilizers are compatible with DSLRs or mirrorless cameras. This means that at any given time, we only see a small subset of the environment - the one provided by the camera viewpoint. Thus, we risk loosing any fast moving target, and not be able to find them again. This is especially true in a narrow environment, where the camera-target distance is small, making the target motion appear faster, and leaving little room for motion prediction. Lastly, real-time wireless streaming from a DSLR (while recording), is not currently offered by any manufacturer. If we were to use such a camera, we would have to opt for a wired communication. Doing so could generate cable tangling, head jiggling, and would not be ergonomic for the operator.

An alternative would be to use a 360° camera. Indeed, these cameras solve most of the issues mentioned above: First, they are lightweight (usually under $300g$). They offer (by definition) omnidirectional field of view, allowing us to track a fast moving target anywhere in the environment, and possibly relocate a target that has been lost. Lastly (for some model) provide wireless streaming. With such cameras, performing tilt, pitch or pan is equivalent to cropping a certain region of the 360° (equirectangular) image, meaning we do not need to use any kind of dolly or motorized head. However it does come with two limitations. First, that the camera is no longer externally stabilized, and secondly that the cropped region quality can hardly be better than one of a DSLR or mirrorless camera. Current high-end 360° cameras made of two lenses offer a quality of at most 5.7K. When cropping a subwindow in a given direction, the maximum resolution one can obtain without observing too much distortion is usually around 1280×720 pixels. In order to achieve higher quality crops, one would have to purchase 360° rigs made

of several (> 2) cameras[1], which is beyond the weight limitations and budget of this project.

With the time and budget constraints of this research project, we decide to opt for a 360° camera, conscious that it means loosing some image quality and possibly some stabilization. Among the existing 360° camera models, we made the following short comparative survey:

Model	Record Quality	Stream Quality	Weight	Wifi	Record while streaming
Garmin VIRB 360	4K at 30fps	720p at 30fps	160g	Yes	Yes
Ricoh Theta S	1080p at 30fps	720p at 15fps	125g	Yes	Yes
Insta360 Air	2.5K at 30fps	-	26.5g	No	-
MIJia360	3.5K at 30fps	-	109g	No	-

Table 3.1: Commercial 360° Camera model comparison, based on relevant criteria for the Spidercam. We find that only two cameras provide wireless connectivity, the Garmin VIRB 360 being superior in image quality over the Ricoh Theta S.

For the reason mentioned earlier, we only retain models providing wireless streaming (using an RTSP Protocol). While the Ricoh Theta S is lightweight, well documented and affordable, the overall quality is quite disappointing. If we wish to crop a specific region of a 1080p equirectangular video, we can only hope to obtain a 360p first-person-view type output. The Garmin VIRB 360 on the other hand, has less documentation and is slightly heavier, but provides a much higher video quality, both in recording and streaming. Besides, it is quite ergonomic and resistant (waterproof up to 10m). Thus, it is the model we will retain for our project.

Motorized Unit

Each motorized unit's main function is to operate on a winch to pull or drag one of the four cables. However, it should do so in a very precise and synchronized fashion, to ensure the desired movement. This means that each unit must be able to receive orders from a master command (in our case, a laptop). Several alternatives exist to build such a unit.

First, one has to choose an appropriate servo-motor. In order to gain some time we opt for an off-the-shelf servo-motor, that has built-in velocity command inputs. This type of motor can be controlled with an Arduino, meaning we don't have to explicitly control the voltage or currents. Besides, it comes with built-in PID regulator, offering an already stable yet dynamic response.

If using an Arduino as a controller board makes the control of the motor easy, it also comes with limitations. First, it has to act as a proper power supply for the motor, to fully exploit its capacities. Then, it must provide fast and reliable wireless communication with the master laptop. Indeed, the tripods being potentially very far apart, we want to avoid at all cost wired communications. Unfortunately, Arduinos were not designed to communicate wirelessly. Thus, we have to use a separate board to perform this task.

Quite naturally, we opt for a Raspberry Pi. Indeed, it provides Wifi and Bluetooth connections, is affordable, and can communicate with Arduinos through either the serial ports, or USB port. The reason why a Raspberry Pi alone could not control the motors is twofold. First, it would not provide sufficient voltage (as explained further below, common motors require around $12V$ for it to reach it full capacities). Secondly, controlling a servo-motor is done using an electrical signal of variable width (PWM, for Pulse Width Modulation). Currently, the Raspberry Pis do not offer stable PWM signals for high-baudrates, leading to communication errors and jittering. On the other hand, Arduinos have very stable PWM signals, and are therefore much more appropriate.

[1] *e.g.* the Insta360™ Pro, which films in 8K

As a result, we come up with the following unit, made of a combination of an Arduino controller board and a Raspberry Pi:

- A Rasperry Pi (model 3) receives orders from the laptop through a ROS Multiple Machine interface (detailed further below). These orders consist of velocity commands and are transmitted with an SSH protocol. This velocity command is processed by the Raspberry Pi, and transmitted to the controller board, with a USB to UART cable. This cable converts USB signals emitted by the Raspberry Pi to TTL serial transmissions, and is manufactured by FTDI (leader in USB to UART connections).
- Then, the Arbotix-M, an Arduino controller board specifically designed to control the AX-12W and AX-12A Dynamixel motors receives these orders, again through a ROS interface. These velocity commands are again processed to match the motors nomenclature, and transmitted through a 3-pin serial TTL Half-Duplex protocol.
- Finally, the Dynamixel motor receives the velocity command, and remains at this speed until further commands are sent.
- The motor's winches operating directly on the camera position, the loop is closed by receiving the 360° camera's live stream. This stream offers information on the actual motion of the camera, and allows the laptop to send new commands accordingly.

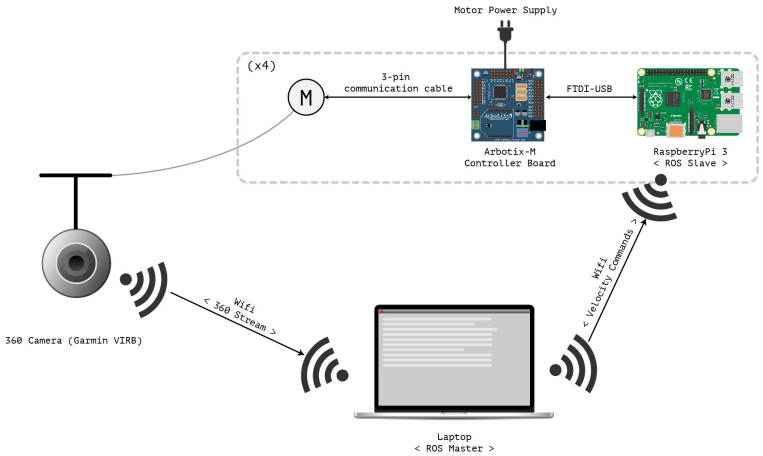


Figure 3.2: Schematic of the hardware architecture of the Spidercam. Each motorized unit communicates with the laptop through a ROS Multiple Machine interface (detailed below). A Raspberry Pi acts as a wireless receiver, and sends the commands to an Arduino controller board. This board both acts as a transmitter, and a power supply for the motor. It ensures a proper serial communication, while regulating the voltage and current to exploit the motor's full capacity. The four motors operating jointly can move the camera in any direction, and the actual motion can be observed through the camera video stream. One can then exploit the video information (namely, by performing live target tracking) to generate commands for the motors.

Sizing

An important aspect when designing such a piece of hardware is dimensioning. Indeed, before purchasing any pieces, it is useful to have an idea of which type of equipment is suitable. The estimated weight of the camera being around 160g, there is no doubt that any four-set of tripod will be able to carry that weight. However, we must make sure that the motors will be capable enough - *i.e.* have enough *torque* - to drag and release such a weight.

Let r be the radius of the pulley mounted on the motor, M_{cam} the weight of the camera, and M_{cable} the weight of one cable. The norm of the torque $\boldsymbol{\tau}$ that each motor has to carry can be simply computed as follow:

$$\boldsymbol{\tau} = ||\vec{F}|| \times r = \frac{M_{cam} + 4 \times M_{cable}}{4} \times r$$

If we then write the maximum mechanical power of the motor as P_{motor}, and ω the motor speed in $rad.s^{-1}$, we can derive:

$$||\vec{F}|| \times \vec{v}_{max} = \boldsymbol{\tau} \times \vec{\omega}_{max} = P_{motor}$$

If we now take an upper weight limit of 500g for the camera, neglect the weight of the cables, and consider an upper radius limit of 5cm for the pulley, we obtain:

$$\boldsymbol{\tau_{max}} = 6.13 \times 10^{-2} N.m$$

Looking at the specifications of the AX-12W Servo-Motor by Dynamixel, we find that it has a stall torque of $0.2N.m$, so more than 3 times what we need for each motor. Besides, it delivers a no-load speed of $N = 470rpm$, or $\Omega = \frac{2\pi N}{60} = 49.2rad.s^{-1}$. For a pulley radius of 5cm, that delivers a maximum speed of $0.75m.s^{-1}$. Given the available space to run experiments, and the jiggling that would be caused by too great a speed, it is more than sufficient for this project.

The reason why we chose this particular motor as a reference is not unusual. The Dynamixel motor series were specifically designed for such research projects. They have now made their way amongst the most commonly-used small-sized and affordable servo-motors, offering a wide range of compatibility with most common interfaces. More interestingly, they are compatible with Arduino Controller Boards, and ROS Interfaces. For all these reasons we will use the AX-12W Servo-Motor to operate the SpiderCam.

Dynamixel AX-12W Specifications

The AX-12W motor has the following specifications:

Motor	Operating Voltage	Stall Torque	No-load Speed	Operating Angle
AX-12W	12V	0.2 N.m	470 RPM (wheel mode)	300° (joint mode)

Table 3.2: Robotis Dynamixel AX-12W Specifications [50]

The specification indicates some crucial informations. First, we can see as expected that in order to exploit the motor's full capacities we need to provide a 12V power voltage. This substantiates the choice we made earlier of using a dedicated controller board to power the motors. From the specification, we also learn that the motor has two operating modes: The *wheel mode* and the *joint mode*

- In the *joint* mode, the motor is controlled in position, with a precision of 0.29°. We have access to the position feedback, which is useful if we want to track the actual position of the camera. However this mode comes with a major downside: its range is limited to 300°, that is less than a full turn. Given the maximum winch radius one could hope to mount on the motor, this would only allow us to move by a couple of centimeters.
- The *wheel* mode, on the other hand, has an unlimited range. In this mode, we control the motor in velocity rather than position. This is especially useful if we want to generate smooth trajectories and have control over the start/stop dampening. However, because the motor does continuous turns we loose all feedback on position. This comes at a price if we need to track the actual camera motion accurately.

Unfortunately, most servo-motors of this size and budget come with this "position *vs.* velocity" command paradigm. Most motors actually only offer a *joint* mode-type control. From now on, we will use the AX-12W in *wheel* mode and try to make up with the lack of position feedback as much as we can.

3D-Printed Pieces

While most parts of this rig are mass-produced, we need some unique parts that cannot be found on the market. More specifically, we need a set of four pulleys (or winches) to be attached on the Dynamixel motor plate. We can obtain the motor specifications and blueprints[2] as references, and design our own pulleys accordingly using any CAD Software. Similarly, the Garmin VIRB 360 can be mounted on a GoPro mount. Using the GoPro specifications, we can redesign our own custom mount to be compatible with the camera and include four cable ties. Using a 3D printer, we obtain those pieces at a very low price. Both 3D models can be downloaded from the project GitHub[3].

Figure 3.3: 3D-printed pieces *.stl* models (pulley and camera mount)

3.2 Software Architecture

We now focus on the software side of this project. More specifically, we will present the ROS Architecture we built to link all the modules involved together. We will present the main command law, as well as the user interface we built and the different operating modes the user has access to. We will then list the difficulties encountered when working with multiple streams and networks, as well as workarounds we found to solve them.

3.2.1 Motors Command Law

A core function to implement is the conversion of a Cartesian camera position (x, y, z) to the four motor velocity commands (v_1, v_2, v_3, v_4). Each motor operating on a cable linked to the camera mount, we aim at finding how much cable length should be pulled or released in order to obtain the desired trajectory.

Formulation

To solve this problem, we will first consider the case of a single motor. A convenient frame to work in is the spherical frame. Indeed, spherical coordinates can be written as (ρ, θ, ϕ). They can easily be converted to cartesian coordinates (x, y, z) and back as such:

$$\begin{cases} x = \rho sin(\phi)cos(\theta) \\ y = \rho sin(\phi)sin(\theta) \\ z = \rho cos(\phi) \end{cases} \Leftrightarrow \begin{cases} \rho = \sqrt{x^2 + y^2 + z^2} \\ \phi = cos^{-1}(\frac{z}{\rho}),\ \rho \neq 0 \\ \theta = tan^{-1}(\frac{y}{x}),\ x \neq 0 \end{cases}$$

If we set a motor to be at the origin of the world, and write (x, y, z) to be the camera coordinates, then the ρ value is in fact the cable length between the pulley and the camera mount. The problem can now be reformulated as estimating $\delta\rho$ (the cable displacement) for every motor, in order to move the camera by $(\delta x, \delta y, \delta z)$.

[2] http://www.trossenrobotics.com/productdocs/AX-12A.pdf
[3] https://github.com/germain-hug/SpiderCamera/tree/master/models/3D_Printing

Solving

We make the following proposition: If we know the (x, y, z) coordinates of the camera at $(t-1)$, then we can solve $\delta\rho$ for any $(\delta x, \delta y, \delta z)$ displacement at time t. Indeed, knowing (x, y, z) at $(t-1)$, we can compute ρ_{t-1} for each motor using the backward transformation mentioned above. Then, for a new input (x^*, y^*, z^*) at t, we can proceed similarly to compute ρ_t. Note that we do not retain the θ or ϕ values. The displacement $\delta\rho$ can then simply be expressed as $\delta\rho = \rho_t - \rho_{t-1}$. If we treat (x, y, z) as an absolute *position*, then $\delta\rho$ is proportional to the actual value by which the motors position should be displaced (winch radius). If we treat (x, y, z) as a *velocity*, then we can control the motors in *wheel* mode (endless turn, velocity command), and $\delta\rho$ is proportional to the motor's velocity.

Hyperstatism

In theory for a single motor, we would also have to compute and update update the θ and ϕ values, as moving the cable by $\delta\rho$ would still allow for an infinite number of (θ, ϕ) values. In practice we have no control over these values. Even more so, because we are not using one but four motors, the ϕ and θ coordinates are in fact constrained by the three other motors. Isolating one motor i, we can consider the three other motors $j \in [1..4], j \neq i$ to be fixed, and to create an overdetermined system. Thus, regardless of the ϕ_t and θ_t values obtained for each motor, updating every ρ valye correctly will move the camera in the desired position. Note that this system is statically indeterminate, or *hyperstatic*. Indeed, the camera has three degrees of freedom. Each motor suppresses one degree of freedom, thus in order to fully define the camera position, only three sets of cables would be required. If we were to use only three sets of cables the problem would become *isostatic*. However, in order to reach more coverage and in respect the traditional Spidercam structure, we decide to use a four-cable tied architecture.

Algorithm

Here is the pseudo-code used for the motor command law algorithm:

Algorithm 1 Motor Command Law Algorithm

```
procedure COMMAND LAW
    h ← System Initial Configuration (hyperparams)
    for i in 1:4 do
        // Compute initial motor coordinates using the camera as the world origin
        m[i] ← i^th motor x, y, z coordinates w.r.t h
    while process running do
        if new displacement (x, y, z) then
            for i in 1:4 do
                // Compute spherical displacement
                prev ← cart_to_spher(m[i][0], m[i][1], m[i][2])
                new ← cart_to_spher(m[i][0] - x, m[i][1] - y, m[i][2] - z)
                delta ← (new − prev) // (δρ, δθ, δφ)
                // Update motor location
                m[i] ← m[i] + deltah
                // Publish motor velocity command (δρ)
                publish(delta[i][0])
```

3.2.2 ROS Architecture

Our system contains a variety of hardware (Arduino, Raspberry Pi, Laptop, 360 Camera) which all have to communicate. If one could design specific commands and protocols for each inter-communication, a more unified and generic interface has already been built for such scenarios: ROS.

ROS

ROS[4] (which stands for *Robot Operating System*) is a set of tools developed in 2007 by Willow Garage[5]. It was designed as a middleware to provide common functionalities within heterogeneous machine clusters. Among these functionalities we find generic message passing, procedure calls, diagnostics and many robotic-related libraries. This framework is now commonly used in many robotics projects, due to its versatility, responsiveness and low-latency. It is compatible with Linux, Arduino, and Raspberry Pi (among others), which is ideal for us.

The key elements used in ROS and for the reader to understand are the following:

- Master: Withing a running ROS environment, there is one ROS Master. Its purpose is to make sure every node can locate each other, as well as provide a parameter server. When dealing with multiple computers, one should make sure that each machine can communicate with the ROS Master (see subsection below for more details).
- Nodes: Each ROS Node can be seen as an individual process. It can either be written in C++ or Python, and can communicate with other nodes through topics. More specifically, it can subscribe and/or publish to one or several topic.
- Topics: ROS Topics act as streams of information, to which nodes can subscribe or publish. Topics can only carry one predefined package type, referred to as message. A message type could be for instance a velocity command or a camera pose, although one can define a custom message type. The information is transported through a TCP/IP protocol, and is reachable by any node within the same ROS environment that subscribed to that topic.

The strength of ROS not only lies in the fact that it provides a reactive and generic framework, but also because it is modular. It makes it very easy for a developer to plug in or out new nodes, create branches, receive feedback or cut communications.

ROS Multiple Machines

As explained above, there should only be one Master within a ROS environment. This means that, in the case where several computers are running, they should all be able to declare themselves to the Master. Luckily for us, ROS comes with a built-in functionality to deploy a system across multiple machines, referred to as *ROS Multiple Machines*. Setting up this service is simple, we only need to declare the following environment variables for each machine:

$$\text{ROS Master} \begin{cases} \text{ROS\_IP} := \text{<Master IP Address>} \\ \text{ROS\_MASTER\_URI} := \text{http://<Master IP Address> : 11311} \end{cases}$$

$$\text{ROS Slave} \begin{cases} \text{ROS\_IP} := \text{<Slave IP Address>} \\ \text{ROS\_MASTER\_URI} := \text{http://<Master IP Address> : 11311} \end{cases}$$

We define ROS Slaves as any ROS machine that is not the ROS Master. One could see here the premise of a first issue: all the machines should be running on the *same* network. This issue will be discussed in 3.2.3 when dealing with the camera stream.

[4] http://www.ros.org/
[5] http://www.willowgarage.com/

Architecture

Having presented the ROS environment, we now detail the ROS architecture we built for the Spidercam. The nodes can be found on both[6][7] GitHub repositories made for the project: a first repository for the laptop, and another one for the Raspberry Pis and Arduino boards. This splitting is done accordingly to the ROS Master and Slave(s) distribution. In a way, the laptop represents the control part, while the motorized units act as the operative part. Below is a detailed list of the nodes developed for the Spidercam, along with their roles and specifications. We advise the reader to use the figure 3.4, which gives an illustrated overview of the ROS architecture and summarizes the node layout presented below. The ROS nodes are the following:

- run_tracking_from_cam: The run_tracking_from_cam performs the target tracking, and runs on the laptop. It is part of the siam_tracker ROS package we built. It takes as input the raw stream of frames from the 360 camera, located in the shared memory. The tracker used is the SiamFC, a deep-learning based tracker designed to run in real-time. An in-depth description of the tracker can be found in its dedicated chapter (see section 4.1). Details about 360 streaming can be found in section 4.2. The node outputs the x and y bounding-box location as well as its size (ratio is fixed) to the /bbox topic, under the Point type. The node is initialized with an initial bounding-box hand-drawn by the user. It starts publishing as soon as a new frame is processed.

- compute_cmd_vel: The compute_cmd_vel node converts bounding-box locations to velocity commands. It subscribes to the /bbox topic, and publishes cmd_vel_motors messages to the /cmd_vel topic. The cmd_vel_motors message type is a custom type, made of four Float32 values, corresponding to the four velocity commands for each motor. This node is part of the bbox_to_cmd_vel package, runs on the laptop, and has two purposes:

  - Establishing which should be the new (x, y, z) coordinates of the camera based on the tracking input. A different computation is made depending on the space we are moving in, and whether we use a predefined spline-based trajectory.
  - Computing the four motors position or velocity commands using the command law specified earlier, and one of the predefined operating mode. An exhaustive list of control presets can be found in 4.3.

- keyboard_teleop: This node reads input from the keyboard and publishes corresponding motor velocity command. It is part of the same ROS package as the compute_cmd_vel node, and also publishes cmd_vel_motors messages to /cmd_vel. We design the keyboard commands to allow for intuitive planar motion, as well as vertical trajectories. Just like the compute_cmd_vel node, the keyboard_teleop node converts (x, y, z) displacements to four velocity commands using the algorithm presented earlier.

- safety_check: The purpose of this node is threefold. First, it should extract the appropriate velocity command from the four commands sent by either compute_cmd_vel or keyboard_teleop. Then, it should perform some security checks, before approving the velocity value and finally sending it to the controller board. Among the security checks, the node verifies that the desired velocity value is within the acceptable range of the motors input (-1023 to 1023). This extra layer of safety avoids having incoherent values that would lead to erratic commands, which could be harmful for the hardware. Besides, this node makes it easy to insert other constraints such as a velocity limit, or possibly more complicated operations such as a flying range limit within the flying area. While we do not need to implement such limitations for this project, the reader may grasp the utility of such a node in other case scenarios. One safety_check node runs on each of the four Raspberry Pis, with a unique ID. This node is part of the cmd_vel_controller package. Each of the four node (running simultaneously) subscribes to /cmd_vel, and publishes Float32 messages to /cmd_vel_approved_i, where i is the unit ID (1 to 4).

[6] https://github.com/germain-hug/SpiderCamera
[7] https://github.com/germain-hug/SpiderCam_Raspi

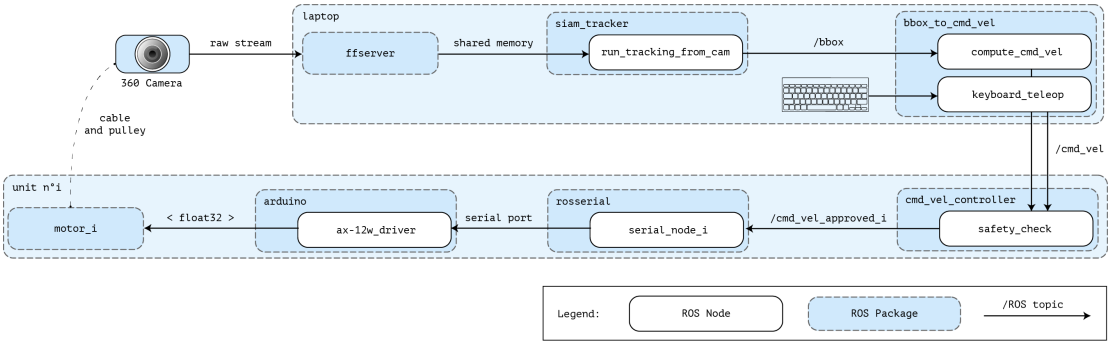


Figure 3.4: ROS Architecture Overview: The ROS nodes can be split in two. The nodes running on the laptop (ROS Master) acquire and process the live 360 stream, to generate velocity commands. They embody the control part of the system. The nodes running on the motorized unit (Raspberry Pi and Arduino) acquire and process the velocity commands, to operate on the motor's velocity. They constitute the operative part of the system. We add an extra keyboard input to allow a user to perform custom, tracker-independent camera movements.

- serial_node_i: To transmit the velocity values from the Raspberry Pi to the Arduino we need to use the serial port. By default, the ROS packages are not transmitted through serial ports or network sockets. However, this ROS transmission can be ensured by the rosserial package, which acts as a wrapper to communicate through serial ports or network sockets. This package is open source[8] and can be used freely. We run it on each Raspberry Pi with a unique node ID to avoid conflicts, and pay attention to setting a compatible baud rate (in our case, 1Mbps).

- ax−12w_driver: The ax−12w_driver node runs on the Arduino. It receives messages through the FTDI-USB link (detailed earlier), by subscribing to the /cmd_vel_approved_i topic, transmitted through the /serial_node. Its purpose is to merely transmit the commands to the motors through the 3-pin communication cable, using the appropriate registers. Indeed, in order to set appropriate commands we have to set specific registers to the desired hexadecimal value. We refer to the Dynamixel series documentation [50] to locate the corresponding registers. Note that in order to set the motor to "wheel mode" (velocity commands), we need to set the clockwise and counter-clockwise angle limit (registers 8 and 9) to zero beforehand. By doing so, we loose access to the register 30 and 31, which indicate the motor position. To set a register value, we rely on the Arbotix Library[9]. We build the velocity command by concatenating the hexadecimal bytes of the velocity absolute value (0 to 1023) with the 0 or 1 hexadecimal bytes for clockwise and counter-clockwise rotations respectively.

Area	Address (Hexadecimal)	Name	Description	Access	Initial Value (Hexadecimal)
	32 (0X20)	Moving Speed(L)	Lowest byte of Moving Speed (Moving Velocity)	RW	-
	33 (0X21)	Moving Speed(H)	Highest byte of Moving Speed (Moving Velocity)	RW	-

Figure 3.5: Excerpt from the Dynamixel AX-12W documentation [50], where we can read the specifications for each register and assign, for instance, velocity commands to the motor

[8] http://wiki.ros.org/rosserial
[9] https://github.com/vanadiumlabs/arbotix.git

3.2.3 Streams and Multiple Networks

Among the difficulties encountered when implementing the Spidercam software, came the streaming and networking issues. Indeed, acquiring the live 360° stream turned out to be harder than expected, and ensuring a proper ROS compatibility with two Wifi networks had to be approached with care.

RTSP

Common real-time camera wireless streams rely on the Real Time Streaming Protocol (RTSP). This protocol is a combination of the Real Time Transport Protocol (RTP) and the Real Time Control Protocol (RTCP). While the RTP is based on UDP, the RTSP relies on TCP and is somewhat similar to HTTP. This protocol is now widely used not only in camera streaming but also in VOD[10], Television Services and other Media Streaming platforms. The Garmin VIRB 360 streams equirectangular images on rtsp://192.168.0.1/livePreviewStream.

Quite surprisingly and despite its popularity, the OpenCV library does not support RTSP. In fact, there is no "easy" way of plugging an RTSP stream into our real-time tracker. Thus, we have to find an alternative to acquire and process the live stream.

Shared Memory

One commonly used RSTP streaming client and server is ffserver. It can process both audio and video, and is configurable through a configuration file. Launching an ffserver to output acquired images to a shared memory space, we can use it along with ffmpeg. Indeed, as mentioned in their documentation[11], "ffserver works by forwarding streams encoded by ffmpeg, or pre-recorded streams which are read from disk". Thus, when acquiring the stream we first launch our pre-configured ffserver, and then begin streaming with ffmpeg. When configuring the server and ffmpeg commands, one should pay particular attention to the audio and video configuration, in order to avoid any conflict. For the Garmin VIRB 360 we used the following specifications:

Command	Audio Sample Rate	Preset	Maximum Rate	Buffer Size	Format	Video Bitrate
ffmpeg	11025	ultrafast	800k	1200k	flv	64

Table 3.3: ffpmeg specifications for the Garmin VIRB 360 live stream

Wifi Networks

As mentioned earlier, all ROS nodes should run on the same network - in our case, the one to which the laptop is connected. However, the laptop also needs to connect to the Garmin VIRB 360 network, to acquire the stream. A first approach would then be to use the camera's network as the primary ROS network. All four Raspberry Pis and the laptop would then connect to its network and use it to communicate through the ROS interface. However, doing so would make us dependent on the camera battery. It would drain more energy, probably create a bottleneck, and prevent us from testing the Spidercam with the keyboard tele-operations when the camera is turned off. Thus, it is preferable to have a dedicated network for the ROS interface.

We choose to add to the laptop an additional Wireless Network Adapter. That way, we can both stream from the camera using one adapter, and act as a ROS Master using the other. We opt for a low-cost Qualcomm Atheros QCA6174 Wireless Network Adapter. We could use any Wifi network, but for convenience we use our own smartphone as a hotspot, which provides the Raspberry Pis and the laptop with a unique IP Address. That way, we do not have to update the ROS environment variables upon each reboot.

[10] VOD = Video On Demand
[11] https://www.ffmpeg.org/ffserver.html

3.2.4 Interpolation

Operating in *wheel* mode, the velocity changes can be quite abrupt. In order to prevent jiggling and camera shake, we implement linear dampening over a custom duration inside the compute_cmd_vel node. We find that adding this linear dampening greatly reduces the camera jiggling, especially at start and stop. Further experiments are conducted in chapter 5.

3.2.5 Multi-Threading

While ROS nodes are all individual processes running in parallel (or even on different machines), we make sure to speed up computation within each node using multi-threading. More specifically, we resort to multi-thread in two nodes: the tracking node run_tracking_from_cam, and the command law node compute_cmd_vel.

The tracking node is in charge of acquiring the 360° stream, reprojecting it and running the SiamFC tracker (see section 4.1). For the model we chose, we find that the tracker runs at around 15-20fps. The reprojection, on the other hand, performs at around 100fps. Besides, the stream runs at a 30fps frame rate, with a lag of around 0.5s. Therefore, in order to minimize lagging we divide the program into two threads. A first thread reads the very last frame written in the shared memory by the server (see 3.2.3). It reprojects it using the equirectangular to stereographic mapping (see 4.2), places the processed frame in a queue of size 1, and checks for new frames. If a new frame is detected, the frame in queue is overwritten. As a result, if the tracker is too slow we will loose intermediate frames. However in that manner we make sure that no accumulative lagging is generated, and we are always as close as possible to the present time. A second thread is dedicated to the tracker. If an unprocessed frame has been placed in queue, it runs the SiamFC algorithm on that frame and publishes the new bounding-box locations (see 3.2.2).

The command law node also has to ensure multiple functions. One of them is performing the interpolation explained in 3.2.4. When given a velocity command, the servo-motors apply this speed until further command is delivered. Thus, in order to apply a custom velocity interpolation, we have to send a continuous stream containing every intermediate velocity value. Quite naturally, we create a specific thread within compute_cmd_vel to perform this task and avoid blocking the main command thread. The process then runs as such: when a new bounding-box location is published, the velocity command is computed and handed to the interpolation thread. This thread performs live interpolation based on the previous commands and custom dampening duration.

3.2.6 Implementation

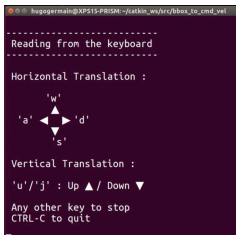
Code

All the ROS Nodes are implemented in Python. We use the Kinetic ROS Version[12], along with the Arbotix[13] library and rosserial[14] package to respectively drive the motor and transfer data through the Raspberry Pi's serial port. The Bézier spline drawing is adapted from J. L. C. Rodríguez[15]. We implement every other node and ROS package described earlier ourselves.

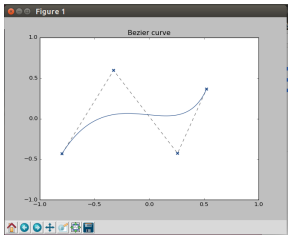
[12] http://wiki.ros.org/kinetic
[13] http://wiki.ros.org/arbotix
[14] http://wiki.ros.org/rosserial
[15] https://gist.github.com/Juanlu001/7284462

User Interface

For testing and operating purposes, we implement two main user interfaces. The first one is designed to operate the Spidercam from the laptop keyboard, while the second one provides a spline-drawing interface to create custom trajectories.



(a) keyboard_teleop UI



(b) compute_cmd_vel UI with the spline mode

Figure 3.6: User Interfaces to control the Spidercam: (*a*) : Keyboard Tele-operation UI, running from the keyboard_teleop node. This node allows for tracker-independent operations, as well as custom trajectories. Pressing keys *accumulate* velocity commands, making it easy for a user to create more complex trajectories. (*b*) : When using a spline-based trajectory, the user can hand-draw custom splines in the $(\vec{x}, \vec{y})$, $(\vec{y}, \vec{z})$ or $(\vec{x}, \vec{z})$ plane. The spline is then subdivided into a chain of velocity vectors for the system to send to the motors.

Hyper-parameters

Before running the Spidercam we have to set some hyper-parameters. First, the flying area dimensions (width, depth and height), as well as the initial camera position. By default, we place it at half of the width and depth, at a predefined height. Then, we set the operating mode we want the camera to follow. The modes currently available are horizontal motion with a fixed top-down view, vertical motion with a fixed side view, and custom spline motion with custom fixed view. Lastly, we set the dampening duration and speed multiplicative factor.

Chapter 4

Autonomous Flying

In this chapter, we propose a novel self-driving approach for the previously built Spidercam. We first demonstrate the SiamFC tracker, which can be used as a control input for the Spidercam. Indeed when filming we often want to keep a target in frame. With a 360° camera one only has to rotate the viewpoint, although doing so does not fully exploit the Spidercam capabilities. Instead, we want to generate motion commands from this visual input, that will make the Spidercam follow trajectories *while* keeping the target centered. We will start by presenting the SiamFC. The SiamFC video tracker is a deep learning based tracker, which delivers fast and robust performances even on reprojected a 360° stream. In this chapter, we will also describe the equirectangular to stereographic reprojection, used as a pre-processing step before handing the raw equirectangular stream to the tracker.

Lastly, we will cover a non-exhaustive list of possible "prescriptions", that is predefined camera movements and trajectories. In this last section we will provide the user on-demand filming presets to make an even more intuitive and autonomous use of the Spidercam.

4.1 SiamFC Tracker

In this section we aim at performing an accurate and fast tracking of an arbitrary target. This target location can then act as a controller, or at least guidance in the Spidercam control. The CMT-Tracker [2] implemented previously in the Action Camera suffered from limitations such as the lack of support for *out-of-plane* rotations, partial occlusions and overall robustness. In order to perform real-time and accurate tracking - possibly accounting for some class-specific information - we choose to turn to Deep-Learning based approaches. With the recent promising results of *CNN*s in numerous image-related tasks, we implement a specific tracker: the SiamFC.

The SiamFC Tracker was introduced in the *"Fully-Convolutional Siamese Networks for Object Tracking*" paper [24] by Bertinetto et al. in 2016. It is one of the few CNN-based trackers that was specifically designed to work in (or beyond) real-time. After being submitted at VOT2016 it was ranked 5th overall, and strongly outperforms its concurrent in terms of speed while maintaining a good accuracy and robustness. To achieve such results, alternative approaches to the tracking problem must be considered, namely the use of a Fully-Convolutional Network, offline pre-training and feed-forward evaluation.

We will first present common methods, architectures and notions of deep-learning based tracking, among which figure Fully Convolutional and Siamese Networks. Using the deep-learning trackers review of the Related Work chapter (section 2.4), we aim at finding alternatives to common tracker flaws.

Then, we will demonstrate the SiamFC Tracker. The SiamFC relies on a Siamese Fully-Convolutional Network. It gets rid of all the heavy design choices mentioned in the Related Work chapter ([13], [23], [29]), and still performs surprisingly accurately. It applies no online training, no RoI sampling, and no Bounding-Box regression. Yet, despite its simplicity, it achieves state-of-the-art results. We will detail its advantages but also its limitations.

4.1.1 Convolutional Neural Networks

Convolutional Neural Networks, or CNNs, have proved to deliver high performances in tasks like Image Classification [15] or Object Detection [16]. By training on large datasets, CNNs allow to obtain high-level features, unlike the typical hand-crafted features that were used in the past. Tracking and Image Labeling are obviously two very different tasks, although there are now various methods to perform tracking from CNN-based object detection.

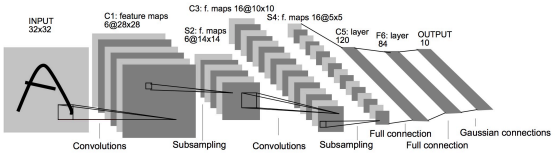


Figure 4.1: Architecture of LeNet-5 [8] Illustration of a Convolutional Network Architecture, used for character recognition

A CNN is essentially a stack of multi-dimensional layers, each resulting from a convolution ran across the previous layer. This convolution is defined by a set of weights, which form a *kernel*. The convolution step size is called a *stride*, and we can deal with borders either using zero-padding (or *same padding*), or by not going past the edges when we convolve (*valid padding*). The global layer structure of a CNN usually forms a pyramid. The bottom layers are wide but shallow, while the top ones are narrow but deeper. Among the standard layers, we can mention the *pooling* layers, which allow to reduce the layer size without loosing information (as we would if we were to naively increase the strides). *ReLUs* (or Rectified Linear Units), allow us to introduce non-linearities in between layers, which makes them more discriminative. On top of the CNN are usually inserted a few fully-connected layers, as well as a softmax function to then perform classification. The strength of CNNs lies in the way features are extracted. The weights are *learned*, using training data, *back-propagation* and optimization (e.g. Stochastic Gradient Descent). This provides us with high-level features, that are much more adaptable than the typically hand-crafted features.

The features obtained from the Deep Neural Nets can for instance be exploited to extract high-level semantic information of targets. It is what Wang et al. made use of in their MLDF [13] tracker (studied in section 2.4). Moreover, they are robust to appearance changes, which in our case is very useful. It is for instance the approach used by Girshick et al. in their object detector R-CNN [16]. Although it performs quite slowly, their algorithm allows for a robust and precise target location. One approach by Krizhevsky et al. uses a pre-trained Stacked Denoising Autoencoder to perform tracking, as described in [15]. As explained in the MLDF paper [13], pre-training is not always required. Other alternatives, like the one described by Li et al. in their paper on Robust Visual Tracking using a single CNN [17], propose a tracking from foreground segmentation, with exclusively online training.

Over the many different CNN structures that were implemented to perform tracking, our task will therefore be to implement one that is robust, accurate, adaptive and fast. We will be using metrics to measure these factors such as those used in the VOT Challenge (see subsection 2.4.1).

4.1.2 Tracking Workflow

Modern CNN-based trackers ([13], [23], [29]) have a lot of similarities in their architecture and workflows. Typically, such trackers work as follows. An initial template is given, usually by a user input, and defines the *target* for the tracker to follow over the frames. For every new frame a batch of Regions Of Interests (RoIs) is sampled around the previous template location, and fed inside a convolutional Neural Network. This Network outputs for each RoI the likelihood of the object to belong to a certain class (identical to the template class). Based on their outputted scores, we can predict which new target location is the most likely, and iterate. One can then proceed to do additional operations. For instance, one may want to train a model to perform Bounding-Box regression [32] which can refine the predicted Bounding-Box

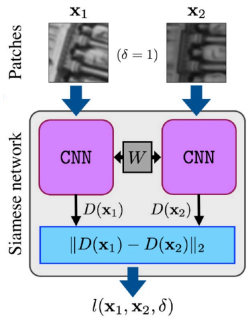


Figure 4.2: Schematic of a Siamese Network [25]. The Neural Network shares its weights for both inputs and is trained to encode similarity between two patches. At training time, we input two RoIs that are labeled as being similar or dissimilar. We compute the L_2 norm of their descriptor, and use a contrastive loss function to update the Network. At test time, we can proceed to evaluate the similarity of a new patch with a template using a simple feed-forward evaluation.

location. One can also use the rejected samples as negative samples to do some supplementary online training, to refine the model and allow for a better adaptiveness.

However, each tracker implements a variation of this generic workflow. The specificity of each tracker usually lies in the network architecture, pre-processing and additional operations made to increase the tracker robustness.

4.1.3 Siamese Networks

With the wish to create a real-time tracker, Bertinetto et al. came up with a novel approach to adress these issues. Instead of treating the tracking problem as a classification-and-localization problem, they treat it as a similarity-learning problem. Regardless of the template label, we will only aim at finding similar patches in new frames. More specifically, we try to find the most *similar* patch to the template. This means that we can track virtually *any* target, regardless of its class, as long as we have a metric to measure its similarity with other images. The network proposed comes with a couple of additional specifications, to allow for even better results. The similarity function mentioned earlier is encoded by a CNN. In this case, is a Fully-Convolutional Siamese Network.

A first occurrence of a Siamese Network can be seen in the *"Signature Verification using a Siamese Time Delay Neural Network"* paper [40] by Bromley et al. (1994). In their paper, they train a Network to assess whether two signatures are matching. A Siamese Network can be described as a pair or CNNs that share their weights, *i.e.* a duplicated network that will be trained to encode similarity between two inputs. By computing the L_2 norm of the two outputed descriptors we obtain a distance function that can be used for both evaluation and training. This L_2 norm can be simply defined as:

$$d_D(\boldsymbol{x}_1, \boldsymbol{x}_2) = ||D(\boldsymbol{x}_1) - D(\boldsymbol{x}_2)||_2$$

Ideally, we would want our distance to be zero when $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ describe both patches, and infinite when they do not, i.e.:

$$d_D(\boldsymbol{x}_1, \boldsymbol{x}_2) = \begin{cases} 0 & \text{for identical objects} \\ \infty & \text{otherwise} \end{cases}$$

During training, a common loss function being used in Siamese Networks is the contrastive loss function. As explained in [25] generic loss function for this structure could be as follows

$$l(\boldsymbol{x}_1, \boldsymbol{x}_2, \delta) = \delta . l_P(d_D(\boldsymbol{x}_1, \boldsymbol{x}_2)) + (1 - \delta) . l_N(d_D(\boldsymbol{x}_1, \boldsymbol{x}_2))$$

where l_P and l_N are the partial loss functions defined as :

$$\begin{cases} l_P(d_D(\boldsymbol{x}_1, \boldsymbol{x}_2)) = d_D(\boldsymbol{x}_1, \boldsymbol{x}_2) \\ l_N(d_D(\boldsymbol{x}_1, \boldsymbol{x}_2)) = max(0, m - d_D(\boldsymbol{x}_1, \boldsymbol{x}_2)) \end{cases}$$

The margin m is defined arbitrarily according to our data. What remains now is to learn the parameters of our model with both positive and negative samples. Using annotated datasets of Multi View Stereo images, we can obtain some positive samples. The amount of negative samples, on the other hand is virtually infinite. This imbalance motivates the creation of methods to sample from these pairs of images. More specifically, we use "aggressive" mining of hard positives and negatives : at each epoch of training we use in priority samples with a high loss function response to update our weights. This method applied to both negative and positive samples is called "Fracking" in [25] and is said to "*improve the discriminative capability of the learned descriptors*".

4.1.4 Fully Convolutional Networks

Among the existing categories of Convolutional Neural Networks, we consider the fully convolutional networks. We define by fully convolutional networks, networks that are made of exclusively convolutional layers, *i.e.* convolutions, average pooling, or element-wise activation functions . These types of networks have been used for various tasks like Image Segmentation [38], Classification [39] and more recently, Tracking ([24], [41]).

Fully convolutional (FC) networks rely on the rule that compositions of convolutional layers will output a non-linear filter, where other nets with Fully-Connected layers will output a non-linear function. In essence, where a regular networks will output multi-class probabilities, or low dimensional values for each input, a fully convolutional layer will rather output a heatmap. This gives a major advantages when dealing with localization problems. In semantic segmentation, we can output a dense pixel-wise heatmap related to each class, allowing us to obtain much more detailed contouring. In tracking, this allows us to obtain a dense map of the similarity of a template with a Search Region ([24], [41]).

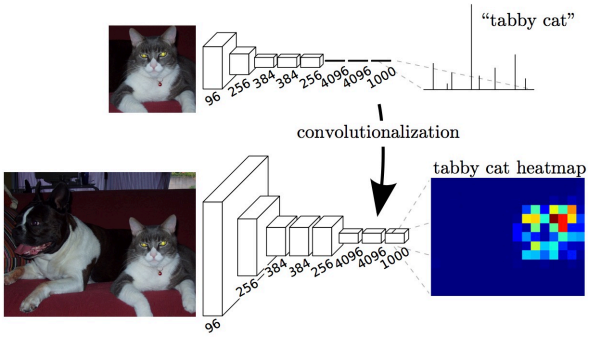


Figure 4.3: Semantic Segmentation using Fully Convolutional Networks [38] We replace the standard Fully-Connected layers (that will output one-hot labeling) with Convolutional layers (convolutionalization). Doing so will output instead a heatmap for each label, allowing (after upsampling) a pixel-wise interpretation (figure from [38])

Moreover, FC Networks can take arbitrary-sized input. The output will then have a directly related spatial dimension. In tracking, this allows for a lot of flexibility in the input sizes, while preserving interpretability. Because we operate with filters on local regions that are translation-invariant, we can easily then upsample our output heatmap to match the input size, using the stride information. This upsampling function can even be a Non-Linear function, and be treated as a backward convolution (or deconvolution), as demonstrated in [38].

4.1.5 SiamFC Algorithm

At each new-frame, we feed into a fully convolutional network a search window, centered on our previous target location, and retrieve its descriptor. As explained earlier, thanks to its fully-convolutional properties, we can compute the cross-correlation (with valid padding) of the outputted search window descriptor ($22 \times 22 \times 128$) with the template descriptor ($6 \times 6 \times 128$). We then obtain a *dense map* of the similarity function (17×17). In essence, each pixel in the cross-correlation heatmap will correspond to the similarity of the template with a sub-window translated accordingly. Unlike the SINT [41] tracker, which is also based on a siamese network and relies on RoI sampling, here we perform a single-shot evaluation of the whole search window.

In practice, because the (17×17) cross-correlation window is rather coarse, we upsample it with a bicubic interpolation. We can then finds its maximum and use the stride values of the network to retrieve the corresponding target location.

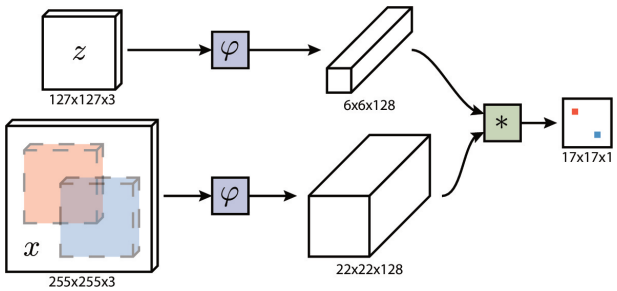


Figure 4.4: Original Siam-FC Architecture [24] To evaluate the similarity of the exemplar z with every translated sub-window of the search region x, we compute the cross-correlation of both descriptors in a single evaluation. Using bicubic interpolation, we upsample our output map to retrieve the new target location.

To allow for scale variation, we feed a batch of three search windows, one being downscaled and one upscaled. If either one of the window shows a higher response, we linearly interpolate the new template size accordingly. To penalize large displacements, we weight our cross-correlation arrays by a cosine window before evaluating the maximum. We also apply a penalty for scale changes.

Lastly, we update our template over time using a rolling average of our newly estimated targets, with a predefined refresh rate.

Algorithm 2 Siam-FC Online Tracking Algorithm

```
procedure ONLINE TRACKING
    x ← Bounding Box User Input
    while !VideoStream.empty() do
        // Extract search region and generate upscaled and downscaled versions
        s ← extract_search_region(x, new_frame)
        z ← generate_upscaled_downscaled(s)
        // Compute cross-correlation window and add penalties
        (f_x, f_z) ← siamese_network(x, z)
        c ← cross_corr(f_x, f_z)
        c ← c + penalties
        // Retrieve argmax and update bounding box
        idx ← argmax(c)
        x ← new_bounding_box(idx, new_frame)
```

4.1.6 Alternative Real-time Trackers

Unlike previously mentioned trackers that treat the tracking problem as a classification problem, the SiamFC considers rather a regression network. We aim to compute a similarity score, encoded in the cross-correlation computation. The use of a very simple feed-forward network can be found in other instances of real-time trackers like YOLO9000 [26] or the GOTURN [42]. However, unlike the YOLO9000 tracking algorithm which also performs in real-time, the SiamFC tracker is label-independent. Because it acts as a similarity function, it can track virtually any object in a robust manner. Besides, unlike the GOTURN tracker which predicts the location of a bounding box in a new image, the SiamFC tracker is intrinsically translation-invariant, thanks to its fully-convolutional architecture. This aspect is especially useful when training, as it requires much fewer data. Without this translation-invariant property, we would have to provide for each training sample every possible translation offsets, which becomes quickly intractable for real-time performance.

4.1.7 Network Architecture

The SiamFC pre-trained network comes with different architectures from which we can choose. The simplest one consists of a single convolutional layer, batch-normalization and ReLU, while the most complex one has five convolutional layers, along with several pooling, batch-normalization and ReLU layers. As expected, there is trade-off in those models between speed and accuracy. We find experimentally that the largest network runs at around 20fps on our laptop, and that simpler models will run at 60-90fps but are much less robust to changes in appearance or less distinctive targets.

Here is the architecture of the network used :

Layer	Support	Channel Map
conv1	11×11	96×3
pool1	3×3	-
conv2	5×5	256×48
pool2	3×3	-
conv3	3×3	384×256
conv4	3×3	384×192
conv5	3×3	256×192

Table 4.1: Network Architecture for the SiamFC, including the layer types, receptive fields sizes (Support) and channel maps ($output \times input$ depth)

To increase the training efficiency a parallel approach is used, as described by Krizhevsky et al. [43]. In their paper *ImageNet Classification with Deep Convolutional Neural Networks*, they describe among other CNN efficiency improvements a GPU-optimized architecture implementation. In essence, they describe a branching method that splits most of the network weights across both GPUs. This explains the inconsistency between the channel maps in the table above. Such a method is employed in the pre-trained nets released by Bertinetto et al. in [24]. When importing the layers, we make sure to replicate such an architecture even when making preliminary tests on CPU-only laptops.

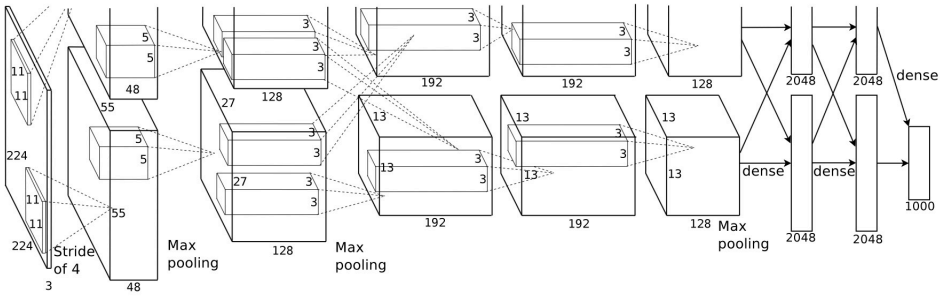


Figure 4.5: Example of a GPU-parallelization network architecture from Krizhevsky et al. [43]. Here, one GPU is dedicated to the upper layers, while the second one only deals with bottom layers. To allow activations to travel across both sides, some layers from opposite levels are connected.

4.1.8 Training

The Siamese Network is pre-trained on both negative and positive pairs, extracted from the ImageNet Video dataset, gathering more than one million annotated frames from 4500 different videos. While negative pairs are easy to sample (cf. fracking in section 4.1.3 and [25]), positive pairs are defined as two samples around the same target being at most separated by T frames [24]. We note that thanks to its symmetrical and fully-convolutional properties the similarity function $f(x_1, x_2)$ is commutative, and does not depend on the target location (respectively).

4.1.9 Implementation

The SiamFC official source code [1] has been made available recently by Bertinetto et al., with a Tensorflow Port [2]. We reuse this code and adapt it to our needs, specifically plugging an OpenCV input to acquire and pre-process the 360° stream, as detailed in the next section.

The code (in Python) essentially translates a MatConvNet *.mat* pre-trained net into a Tensorflow-compatible architecture, properly copying the weights and layers to obtain the same functioning. On a laptop equipped with an Intel-i7 CPU and GeForce GTX 1050 GPU, the tracker runs at around 20fps. On simpler models (one or two convolutional layers) the tracker runs at up to 90fps, but looses the target much more often. For our purposes, we prefer a close-to-real-time but robust performance, and retain the 5-convolutional layer model.

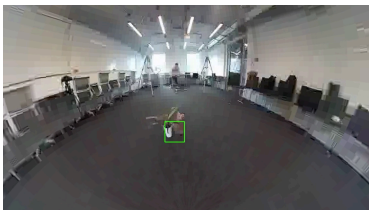


Figure 4.6: Tracking User Interface: At start the user controls the camera reprojection (latitude and longitude), and draws the initial bounding-box (in green). The tracking algorithm then runs for all subsequent frames and drops intermediate frames to avoid accumulating any lag. When tracking we can choose to set the camera viewpoint as fixed, or update its orientation to keep the target centered.

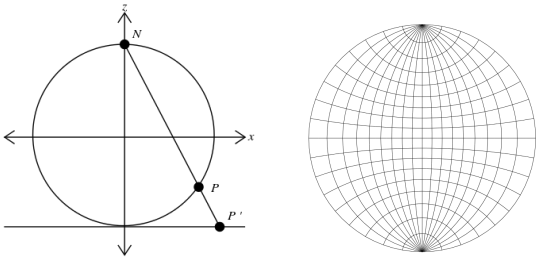
[1] https://github.com/bertinetto/siamese-fc
[2] https://github.com/torrvision/siamfc-tf

4.2 Equirectangular to Stereographic Reprojection

In order to run our tracking algorithm we need to rectify the equirectangular stream provided by the 360 Camera to a less-distorted, "first-person" viewpoint. Indeed, the raw equirectangular stream has very strong distortions on the top and bottom edges for vertical wrapping, and other distortions vertically for horizontal wrapping. We pick a stereographic reprojection, vastly used in VR applications and easily computable from an equirectangular image.

Principle

The Stereographic projection can be described as a function projecting sphere points onto a plane, from the sphere's North pole. Historically it was used to build Celestial Charts, dating back to the Ancient Greek Era (referred at the time as *Planisphaerium* [49]). We can build a mapping function taking (x, y) image coordinates and outputting (ϕ, λ) latitude and longitude coordinates. Typically we provide an extra parameter of distance (or radius) R, defining how "wide" our point of view is.



(a) Stereographic projection schematic

(b) Wulff net, or Stereonet

Figure 4.7: Stereographic Projection [48] On the left, the schematic illustrates how each point P' is computed from the sphere point P (here, our equirectangular image). On the right, the Wulff net, traditionally used to graph by hand latitude and longitude coordinates onto a plane (figure from [48])

Formulas

The projection formulas were derived by E. W. Weisstein [49] and can be found online [3]. For a reprojection centered around (ϕ_1, λ_0), the **forward projection** is given by :

$$\begin{cases} x(\phi, \lambda) = k \times cos(\phi) \times sin(\lambda - \lambda_0) \\ y(\phi, \lambda) = k(cos(\phi_1) \times sin(\phi) - sin(\phi_1) \times cos(\phi) \times cos(\lambda - \lambda_0)) \end{cases}$$

$$\text{with } k = \frac{2 \times R}{1 + sin(\phi_1) \times sin(\phi) + cos(\phi_1) \times cos(\phi) \times cos(\lambda - \lambda_0)}$$

The **backward projection** is, on the other hand, given by:

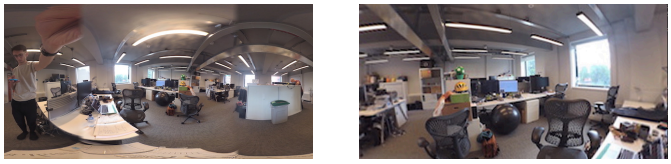
$$\begin{cases} \phi(x, y) = sin^{-1}\left(cos(c) \times sin(\phi_1) + \frac{y \times sin(c) \times \cos(\phi_1)}{\rho}\right) \\ \lambda(x, y) = \lambda_0 + tan^{-1}\left(\frac{x \times sin(c)}{\rho \times cos(\phi_1) \times cos(c) - y \times sin(\phi_1) \times sin(c)}\right) \end{cases}$$

$$\text{with } \begin{cases} \rho(x, y) = \sqrt{x^2 + y^2} \\ c(x, y) = 2 \times tan^{-1}\left(\frac{\rho}{2 \times R}\right) \end{cases}$$

[3] http://mathworld.wolfram.com/StereographicProjection.html

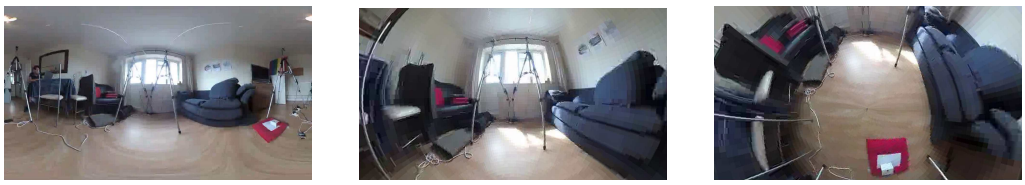
Artifacts

Because we are using OpenCV *remap* function when implementing the reprojection, and lack a shader-like framework (*e.g.* OpenGL), we pre-compute the 360 projection maps for every latitude. We then apply an online longitude rotation by *rolling* the equirectangular image horizontally. We obtain an on-the-fly reprojection close to real-time, not penalizing our performance by much.



(a) Raw Image (b) Reprojected Image

Figure 4.8: Stereographic reprojection algorithm applied to pictures taken with the 360 camera (5.7K resolution). The reprojection is high-resolution, presents little deformation and is exploitable for tracking.



(a) Raw Stream (b) Reprojected Stream (c) Top-down Reprojection

Figure 4.9: Stereographic reprojection algorithm applied to the stream delivered by the 360 camera. The reprojection obtained presents visible artifacts due to a poor resolution, as well as stitching issues.

A first observation we can make from the reprojected stream is that a low resolution in the equirectangular results in even poorer results in the reprojected image. So much that we start to observe ”chunks” of pixels, which may penalize the tracking. These artifacts, stemming from the low bandwidth of the camera server, are quite limiting and prevent us from tracking small or too distant targets accurately.

In addition to the visible artifacts due to a poor resolution, we can add two other perturbations. The first one comes from the difference of exposure on both 180° cameras. In an indoor environment, where lighting can be unequally distributed, we observe stitching issues. Looking at the raw image above, we can easily distinguish both frames, not only in the cropping of the hand holding the camera, but also in the global illumination of the room. This can become a problem when looking at an overlapping region of the camera, revealing the seem and delivering unaesthetic results.

The last perturbation to account for is the stitching occurring at the two poles of the sphere. On the upper and lower parts of the equirectangular image, large surfaces have to be contained in small, distorted regions. When reprojecting back, we loose information and observe overlapping due to the inaccuracies in the equirectangular stream. This overlapping is not only visible and unaesthetic, it also prevents us from performing tracking around this area. As a result, we must avoid vertical, “topdown” shots, and opt for a slightly tilted angle when filming.

4.3 Prescriptions

We define a non-exhaustive set of operating modes to run the Spidercam. These modes consist different combinations of trajectories, camera viewpoints and target motions. They can be used as presets for easy cinematic controls of the Spidercam. These modes will be tested and studied in the next chapter.

Description	Trajectory	Support	Trajectory d.o.f[4]	Viewpoint	Viewpoint d.o.f
Vertical motion, fixed viewpoint	Linear	$\vec{z}$	1	Frontal	0
Vertical motion, orientable viewpoint	Linear	$\vec{z}$	1	Equatorial	1 (horizontal)
Horizontal motion, fixed viewpoint	Linear	$\vec{x}/\vec{y}$	1	Top-down	1 (vertical)
Planar motion, fixed viewpoint	Planar	$(\vec{x}, \vec{z})$	2	Frontal	0
Horizontal motion, fixed viewpoint	Planar	$(\vec{x}, \vec{y})$	2	Topdown	0
Spline-based motion, fixed viewpoint	Spline	$(\vec{x}, \vec{z})$ / $(\vec{x}, \vec{y})$	1	Frontal / Top-down	0
None / Keyboard teleoperation	-	-	0	Any	2

Table 4.2: Non-exhaustive list of operating modes for the Spidercam. We differentiate three types of trajectories: linear, planar and spline-based. The first suggested mode is a vertical linear motion with a fixed viewpoint. Here, the target's y position triggers could be used to create shots were the target only induces motion along one axis. Any motion on the other axis is neglected, allowing for instance to keep the target centered around a fixed latitude, with no constraint on the longitude. One could then design a similar mode where, in addition, the 360 camera can (virtually) pan while it is tracking. Target motion along the vertical axis induce *camera motion*, while target along the horizontal axis induce *viewpoint motion*. That way, the target can be kept centered in the frame, while only traveling vertically. This could be used in cases where the target's path encircles the camera (*e.g.* runners in a stadium). Horizontal and planar motions can be typically used to hover or follow ground-moving targets. Here, we set a fixed viewpoint and allow motion along the $\vec{x}$ axis, $\vec{y}$ axis, or the whole $(\vec{x}, \vec{y})$ plane. As the target moves, the system will try to keep it in the same position on the frame as it was originally, along one axis (linear motion) or two axis (planar). Then, we propose a spline-based trajectory. In this mode, the user can draw any trajectory, projected on an horizontal or frontal plane. This allows the camera to follow complex path, again triggered by the target motion (in x, y or both directions). Lastly, we suggest an additional mode where the camera is fixed, and we only perform tracking by reorienting the viewpoint, to keep the target in frame.

Chapter 5

Experiments and Results

In this chapter, we will evaluate our autonomous Spidercam through a series of experiments. We will first test the Spidercam hardware and software, measuring its trajectory, accuracy and speed for arbitrary keyboard inputs. We will investigate the impact of speed and velocity damping on the system precision and stability, both for vertical and horizontal motion. Then, we will test our SiamFC tracker on reprojected 360° streams. Again, we will be measuring its speed and accuracy, and compare it to results from standard validation data. Lastly, we will test our autonomous Spidercam as a whole, combining the hardware, software and tracking algorithm, on a variety of trajectories. We will demonstrate how having a visual feedback improves the Spidercam performance, and how it follows custom spline trajectories with target-responsive motion. We will be testing the self-driving Spidercam both indoors and outdoors.

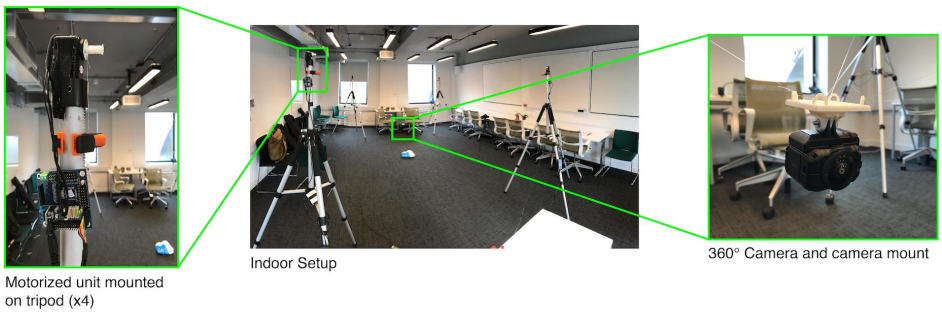


Figure 5.1: Spidercam Indoor setup: In this setup, we deploy the four tripods and motorized units at each corner of the room, and track a wide range of targets.

Figure 5.2: Spidercam Outdoor setup: In this setup, we deploy the four tripods and motorized units at each corner of the garden, and track a remote controlled toy car.

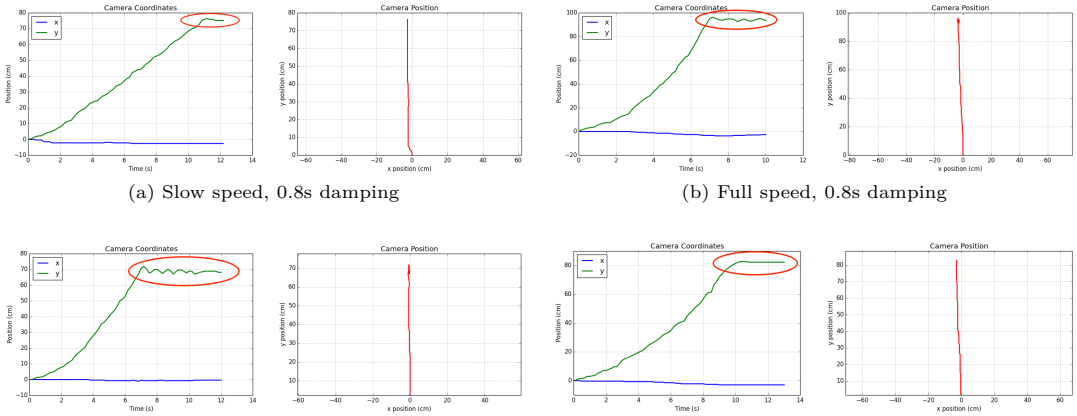
5.1 Spidercam Speed and Accuracy

The first series of experiments we design aims at measuring the Spidercam overall accuracy and speed for arbitrary trajectories. Here, we try to assess how well our structure performs for linear, planar and spline-based motions. To perform these measurements we setup two DSLR cameras, to obtain a right and a front view of the system. After calibration, we are able to compute the camera position with the precision of 1cm on both axis, and reconstruct the actual trajectory. While assessing vertical and horizontal motion is easy (the ground truth is known in space), assessing spline-based trajectories is much harder, as it is subject to scale and we lack equipment like top-down cameras, with projected grid patterns. Therefore, for spline-based trajectories we will content ourselves with qualitative results, in terms of stability rather than precision. First, we will measure the precision and the effect of speed and velocity damping on vertical and horizontal linear motion. We conduct these experiments in an indoor environment.

5.1.1 Vertical Motion

We first consider a simple vertical motion, rising up over the course of around 80cm. For this trajectory, we plot its path over time, for several speed and velocity damping values. The results can be observed on figure 5.3, showing the (x, y) coordinates as well as the camera position (front view).

First, we can notice the importance of adding velocity damping in the trajectory precision. Indeed, using a damping time of $0.8s$ seems to yield good precision for a slow to medium speed ($0.05m.s^{-1}$ to $0.15m.s^{-1}$). For a full speed (around $0.3m.s^{-1}$), we find that at least a 1.5s damping is necessary to preserve stability. However, it does make the overall system slower at startup and stop. One slight artifact that can be observed is that at constant velocity, we observe small oscillations. These oscillations are even more visible on the outputted 360° video. We make the hypothesis that they come from the cables elasticity, and are being sustained by the motors which run at slightly different speeds for different torques. Overall, the actual trajectory is rather close to the ground truth, meaning our system is quite accurate for vertical trajectories. This was to be expected, as the four motors operate with the same velocity value when the camera is on the perpendicular bisector of the four tripods. Hence, unless there was to be a desynchronization of the units, we can expect the vertical trajectories to be fairly precise. We measure an average error of $3cm$ to the vertical ground truth trajectory. We also note that the system is quite responsive. We measure a lag under $< 10ms$ for all keyboard tele-operations, which is quite promising for responsive autonomous controls.



(a) Slow speed, 0.8s damping

(b) Full speed, 0.8s damping

(c) Full speed, 0.1s damping

(d) Full speed, 1.5s damping

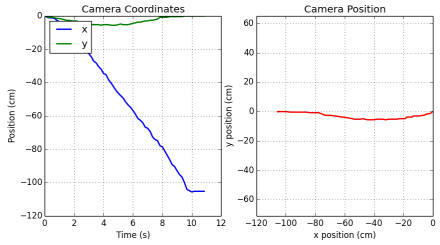
Figure 5.3: Vertical Trajectories (front view). We show results for various speeds and damping periods, over a vertical, bottom-to-top path. The impact of velocity damping is circled in red.

5.1.2 Horizontal Motion

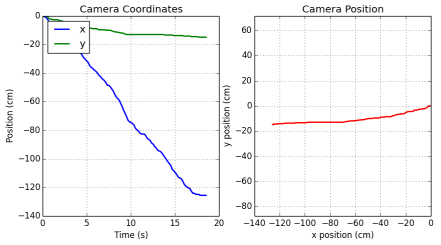
We proceed similarly, this time for horizontal trajectories. We will perform similar measurements for right to left trajectories, over the course of around $1.2m$. Results can be found on figure 5.4. We find that the horizontal trajectories are overall poorer than the vertical ones. Indeed, for a slow speed and when the camera is rather close to the ground, we find that the Spidercam stays rather close to its prescribed trajectory. However, when increasing the speed, the system has difficulty maintaining a constant height, resulting in a camera vertical plunge, sometimes by up to 20cm. Again this can be explained by the motors capacities. Referring to the AX-12W documentation we can see the impact of speed on the motor's torque. As the speed increases, the torque decreases, meaning that at higher speeds the motors pulling up cable will have difficulty maintaining their nominal speed. Because we are now dealing with horizontal motion, in which the motors apply asymmetric motion, we find that half of the motors will pull up wire slower than the other half will release it. Because we have no way of measuring the actual velocity of the motors this is hard to counter-balance, and results in these observable plunges.

This is even more visible when the camera is at a higher altitude. In such configuration, the tension on the cables is quite strong, and the motors releasing the cables will end up rotating faster than they should, while the motors rewinding the cables will rotate slower. For a medium speed we observe a camera dive of almost $30cm$. Again, we can also notice some slight pseudo-oscillations at a constant velocity for the same reasons as mentioned earlier.

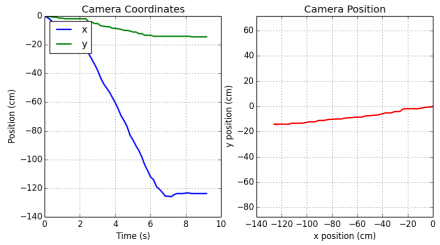
Lastly, looking at figure 5.4 we do notice what seems to be a higher stability when stopping at full speed, even though the damping period was only of $0.8s$. The figure here is in fact misleading. Because we used a $25fps$ DSLR with only $1080p$ quality, we miss some higher frequency oscillations. While such oscillations were not present on vertical motions, we observe them with our naked eye when stopping abruptly on a horizontal motion. These oscillations have an amplitude less than $1cm$, which is less than those observed vertically, but with a frequency of around 10Hz. These vibrations can be explained by asymmetric cable tensions when the motors stop (two of them are pulling, two releasing). While they are not visible on the DSLR footage, they can be observed on the recorded footage. Therefore, unlike what is indicated by figure 5.4, one should be aware of such perturbations and keep on using a correct velocity damping period.



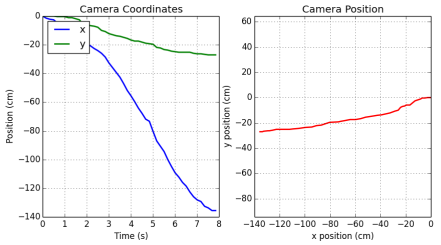
(a) Slow speed, 40cm height



(b) Medium speed, 40cm height



(c) Full speed, 40cm height



(d) Medium speed, 180cm height

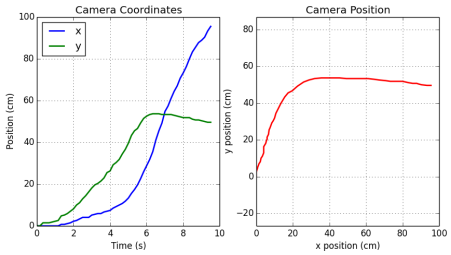
Figure 5.4: Horizontal Trajectories (front view). We show results for various speeds, over a vertical, right-to-left path. (d) features a trajectory at a higher level.

5.1.3 Results

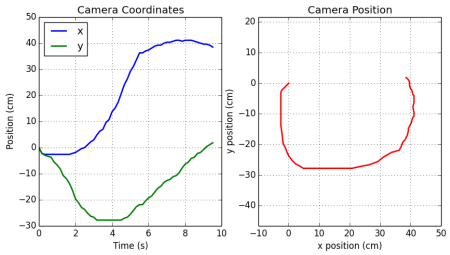
Decoupling the vertical and horizontal motion show overall promising results for the Spidercam. We find that these motions present their own artifacts, but impose the same constraints. That is, (1) we should be using velocity interpolation and damping to prevent any abrupt changes that would lead to perturbations, and (2) we should be aware of the motors limitations. They seem to be inducing trajectory errors, as well as sustaining pseudo-oscillations on the wires.

We conduct further experiments on spline-based trajectories, although we cannot assess their precision due to a lack of a reprojected ground truth. In this mode, we discretize a hand-drawn spline into a chain of velocity vectors, and iterate through the chain. As expected, spline-based trajectories present combinations of previously observed vertical and horizontal motion perturbations: self-sustained oscillations and camera plunges on horizontal motion. Results can be seen on figure 5.5 below. If most spline-based trajectories present vertical plunges similar to those observable on horizontal motions, one could still say that they are good enough on larger scales and for rough filming operations.

Additionally, we should take into account that the system operates in *open loop*, meaning we do not have any feedback on the actual trajectory to correct it. Besides, when tele-operating it we can manually counter-balance for most perturbations, by for instance sending vertical motion commands to prevent the camera from "diving". Considering this, we can again state that our Spidercam operates accurately enough in open loop. We will study the system in closed loop in section 5.3.



(a) 'Rise' type shot



(b) 'Circle' type shot

Figure 5.5: Spline-based trajectories (Side and Front view respectively). In this mode the user can draw any Bezier spline within either the (x, y), (x, z) or (y, z) plane. As expected, horizontal motion suffers from camera 'plunging', and the overall motion presents visible pseudo-oscillations self-sustained by the motors and cables elasticity.

5.2 SiamFC Speed and Accuracy

We now aim at evaluating the SiamFC tracker on its own, before merging both ends in the next section. First, we evaluate it on validation data sequences from TempleColor[1], VOT2014[7] and VOT2016[5]. Then, we move on to evaluate four custom video sequences captured with the Garmin 360 VIRB. We design three indoor sequences and one outdoor video, each trimmed to a 10s length.

5.2.1 Validation Data

We use validation data from 129 videos, coming from TempleColor, VOT2014 and VOT2016, cumulating to 98037 frames. Pre-assembled and labeled sequences from these datasets can be found on the CFNet[44] repository[2] made available by Valmadre et al. We are interested in computing two factors: the Intersection over Union (IoU), as well as the tracking speed. The IoU gives us any idea on the tracker accuracy, while the speed is crucial for responsiveness. Here are the results obtained on the five video sequences from figure 5.6:

Sequence	'Bolt'	'Crossing'	'Fish'	'Girl'	'Racing Car'
IoU	63.71	75.82	58.52	8.37	78.43
Speed (fps)	25.85	22.95	25.56	26.64	25.17

Table 5.1: Tracking results on sequences from VOT2016[5], as shown in figure 5.6

The average IoU for the 129 videos is **50.07**, and average speed of **23.66**fps. This means that on average, our tracker's estimated bounding box has a 50% overlap with the ground truth, and runs very close to real-time. As shown on figure 5.6, the 'Girl' sequence corresponds to a failure case, which explains a low IoU. Several "hard" sequences show failure in the validation set, often due to partial or full occlusion. These failures explain the rather low average IoU but do not compromise the tracker's robustness in other "easier" sequences (*e.g.* "Crossing"). These results are quite promising, and justify picking this tracker.

5.2.2 Captured Data

To assess our tracker on the 360° reprojected stream, we design three indoor sequences and one outdoor video, each trimmed to a 10s length. As shown on figure 5.7, we track a notebook, a plastic bag, the operator head and lastly a remote-controlled car for the outdoor sequence. We try to feature common perturbations such as out-of-plane rotations, backlighting, shakiness, and reprojection distortion. Unwillingly, we also feature low streaming bandwidth, which results in visible chunks of pixels.

To evaluate our data, we write a custom Python script to compute the Intersection over Union (IoU) over a subset of the sequences frames (400 frames in total), using hand-drawn ground-truth for each frame. We measure the following values:

Sequence	Notebook	Plastic Bag	Operator Head	RC Car
IoU	73.42	70.65	68.30	69.24
Speed (fps)	20.92	19.87	21.23	20.15

Table 5.2: Tracking results on our own captured data, for a subset of 100 frames per sequence.

[1] http://www.dabi.temple.edu/ hbling/data/TColor-128/Temple-color-128.zip
[2] https://drive.google.com/file/d/0B7Awq_aAemXQSnhBVW5LNmNvUU0/view

The results are rather good compared to the validation dataset results and considering the list of artifacts mentioned above. Moreover we should state that no sequence has demonstrated failure (IoU= 0), and that the tracker has proved to be surprisingly robust despite the 360° reprojection distortion as well as the low bandwidth leading to very poor image quality. In terms of speed, we reach **20.54***fps* on average for the most complex CNN structure. The reason why it is slightly slower than on validation data is because of the higher image resolution as well as pre-processing streaming acquisition and reprojection step. Nonetheless, it is still close to real-time and fast enough for our testing purposes and target speeds.

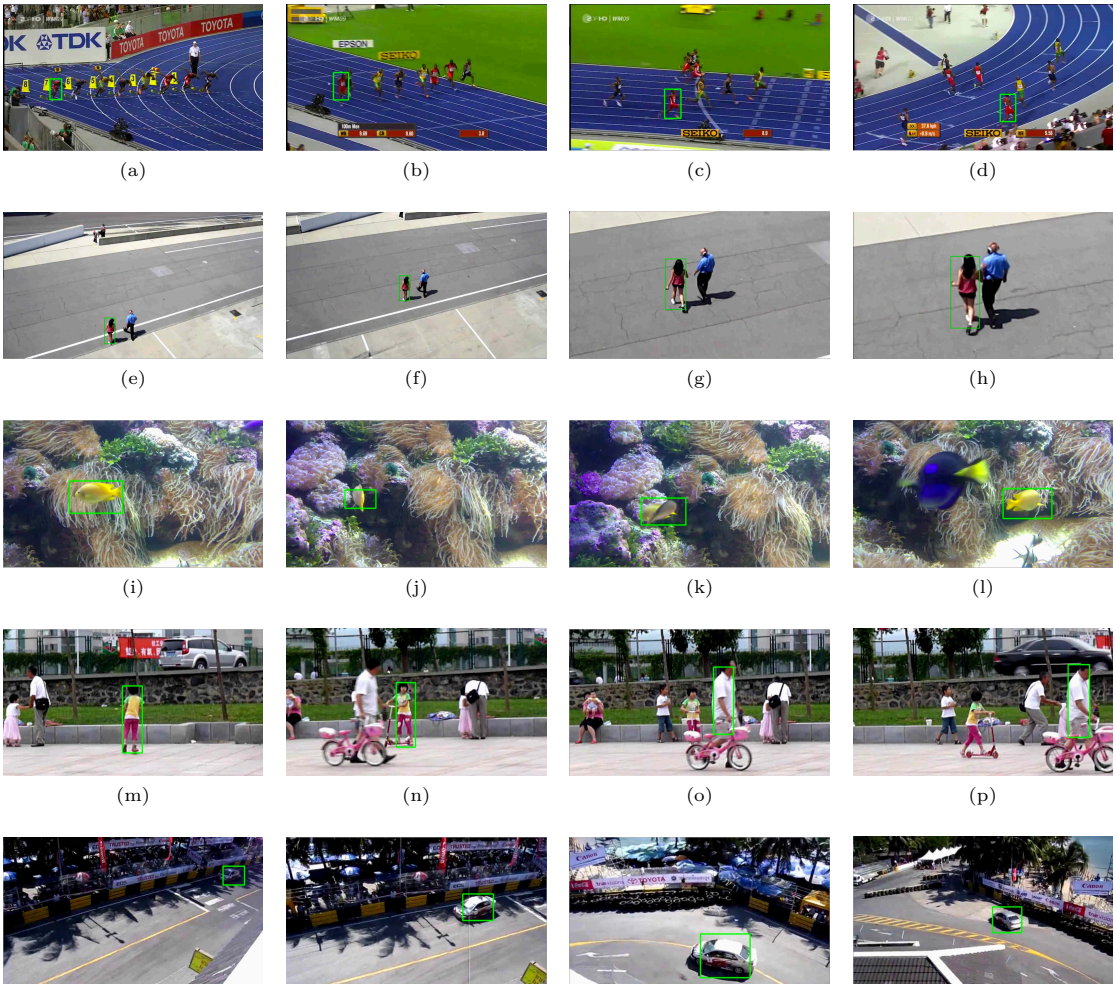


Figure 5.6: SiamFC tracker results we obtained on the VOT2016 Validation Set [5]. The template is defined exclusively by the very first frame (a, e, i, m, q), and updating with a rolling average. The algorithm tracks a variety of targets subject to changes in appearance (b, c, d, j, l), scale (f, g, h), or both combined (r, s, t). The tracker is also robust to occlusion as shown in (k). It also delivers robust results with complex backgrounds and camera shake. The fourth sequence shows a case of failure, where the tracker looses the target (o) and does not manage to find it again.

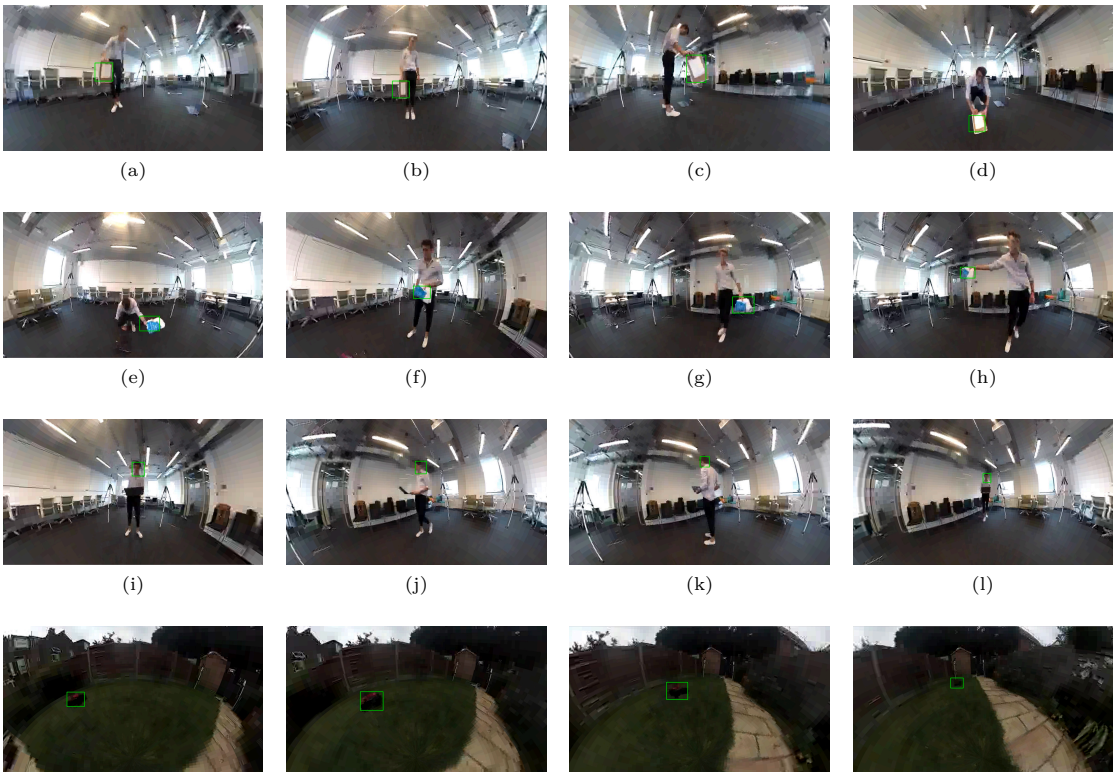


Figure 5.7: SiamFC results on the reprojected 360° stream. We design three indoors video sequences, tracking a notebook (a, b, c, d), a plastic bag (e, f, g, h) and the operator head (i, j, k, l). We add an extra outdoor tracking sequence of a remote-controlled car (m, n, o, p). We find that the tracker performs very well even when with poor streaming bandwidth, out-of-plane rotation (k), backlighting (m, n, o, p) and distortion from the stereographic reprojection (see section 4.2)

The SiamFC Tracker seems to be an appropriate choice for the Spidercam. Its flexibility, robustness to common perturbations and speed are sufficient for our live-tracking purposes. Besides, it is compatible with a reprojected 360° live stream. If we do record some occurrences of failures for some validation videos, and it has trouble dealing with full occlusions, we can consider it is good enough for the scope of this research project.

5.3 Autonomous Flying

Lastly, we investigate the Spidercam working in *closed loop* - that is exploiting the SiamFC to provide tracking information. Here, we generate motion commands using the predicted bounding-box coordinates, and a predefined trajectory support (linear, planar, spline-based, see section 4.3 for more details). Since we have already identified the Spidercam's limitations when it operates in open loop, we will now demonstrate how closing the loop through the camera tracking can improve its performance, but also reveal new issues.

We set up the Spidercam to track the position of an arbitrary target. Having already tested the tracker and for test purposes we pick an "easy" target (namely a notebook), to avoid failure cases which would deliver erroneous motion controls. We distinguish two types of motion commands generation. First, we set the camera to move freely along a linear or planar support, and compute velocity commands to keep the target centered in frame. Then, we set the camera to follow a specific spline, and use the target as a trigger to generate velocity commands. In both of these modes, the system can be considered to be working in a closed loop, as the tracking feedback helps readjusting the commands in real-time.

5.3.1 Target-centered motion

We start out by experimenting with a mode where we allow the camera to move anywhere within a predefined support, to keep the target centered in frame. Results can be found on figure 5.9.

Vertical motion

In the first experiment we run, the Spidercam is constrained to move along a vertical axis. We allow latitude rotation in the 360° camera viewpoint, meaning we are motion-independent on the horizontal axis. When the target moves up or down, we send commands to move the camera vertically accordingly.

The first measurement we can do is to estimate the delay between the target motion and actual camera movement. We find that over our experiments the delay is on average $1.8s$. The main source of this lag comes from the camera streaming and acquisition, on which we have little control. In comparison, the $20ms$ of tracking and $< 10ms$ of motor operation are negligible. However, taking into account the velocity damping, we find that for reasonably slow targets the created motion is smooth and responsive enough. As shown on fig. 5.9 (a), the Spidercam matches the ground truth motion accurately.

However, we do encounter a new perturbation: *overshooting*. While we were running our experiments, we found that the Garmin 360 VIRB was occasionally dropping some frames and freezing (sometimes up to $1s$) on past frames. This comes as a major drawback, as (1) we do not have any control on the camera streaming server, and (2) this prevents the Spidercam to adjust in real-time its velocity commands. As a result, we observe overshooting (see fig. 5.9 (b)): The Spidercam detects that the target is moving down, and sends appropriate motion commands. Then, because the frame freezes we keep on sending commands to lower the camera, where in fact it has already gone past the target. This overshoot creates

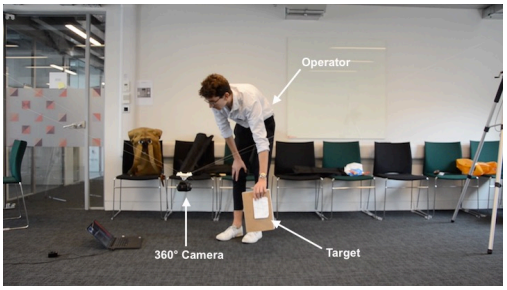
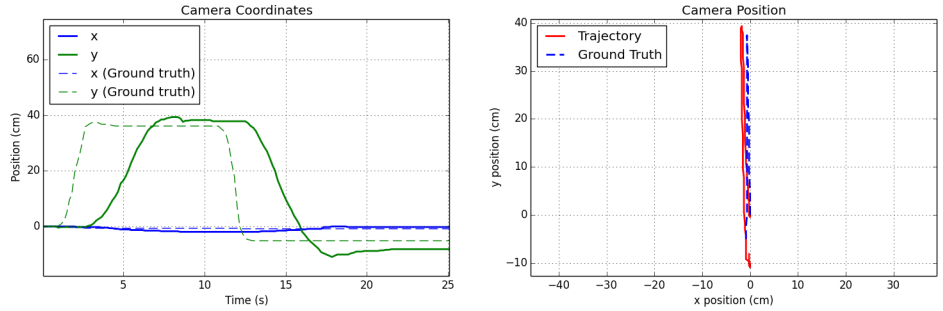
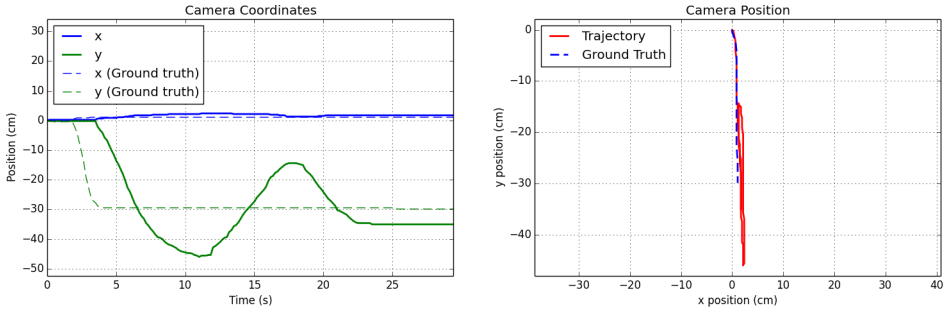


Figure 5.8: Tracking experiment setup: We setup the camera to track the target in real-time and generate motion controls to either keep the target centered or follow a spline trajectory

oscillations, of up to $30cm$ and lasting more than $15s$. This has a considerable impact on the Spidercam motion and stresses the importance of a fast and reliable streaming client and server.



(a) Vertical Tracking



(b) Case of severe overshoot on vertical tracking, due to camera broadcast freezing

Figure 5.9: Target-centered vertical motion flying. In this experiment, the camera is constrained to travel along the z axis. Using a custom Python script and external DSLR recordings we plot the following trajectories, both for the camera and the target. We measure an average $1.8s$ delay, and present on (b) a severe case of overshooting. Note that the featured static error does not reflect the actual static error, as both the target and the camera were observed from the same viewpoint and lie in different planes. Overall and when the streaming server does not fail (as shown in (a)), the camera follows the target fairly accurately and quite responsively as well.

Planar motion

Then, we investigate planar motion. Figure 5.10 shows the Spidercam moving in the (x, z) plane, and trying to keep up with a target. Just like with vertical motion, we observe a $1.8s$ delay, and record occasional cases of slight overshoot due to the camera server freezing. However, we can make one interesting observation. In comparison to the system working in open loop, the horizontal motion is much more accurate. Indeed, even though the camera is usually a bit below the target, it manages to catch up with the target eventually and readjust its altitude. This can be explained simply by the presence of visual feedback. When we were previously sending horizontal motion commands, the Spidercam had no way of knowing it was going down. Here, when the camera starts plunging, the visual feedback is fast, and an additional velocity command is created to compensate for this failure.

Therefore, not only do we manage to create a closed-loop system, we also demonstrate how closing the loop helps enhancing the Spidercam's performance. Let us now consider another test case, where we do not try to keep the target centered in frame, rather provide a specific path for the camera to follow.

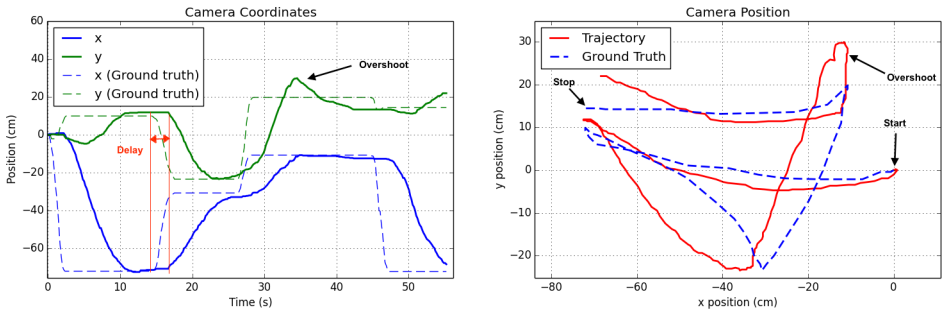
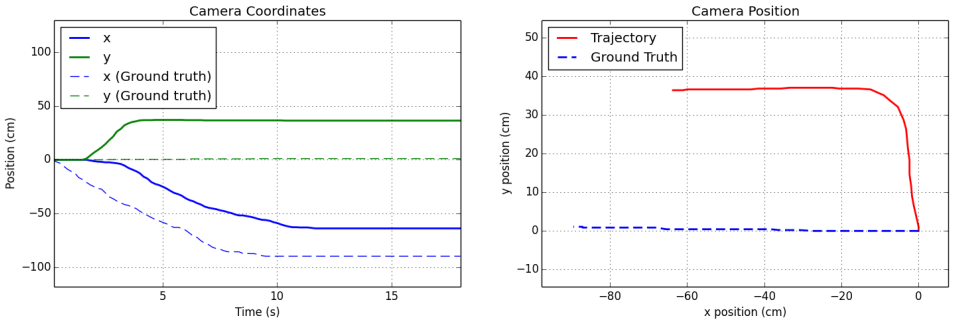


Figure 5.10: Target-centered planar motion flying (side view) In this experiment, the camera is constrained to travel along the (x, z) plane. Again we record an average $1.8s$ delay, and occasional slight overshoots. Here, we demonstrate the benefit of having visual feedback for a more accurate horizontal motion, in comparison to open-loop motion control.

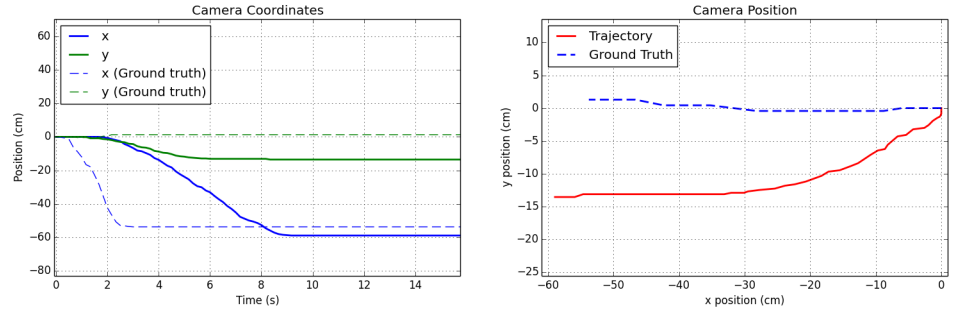
5.3.2 Spline-based motion

Using a predefined spline trajectory, one can design more complex shots. Here, we do not necessarily aim at keeping the target centered in frame, but rather have the camera travel along pre-defined paths. Using the spline-drawing interface implemented earlier, we can again create a chain of velocity commands. Now, instead of manually driving along that chain, we make the target position trigger motion commands. For instance, when doing a 'rising and hover' shot (see fig. 5.11 (a)), we can state that the target's vertical position is mapped to the chain of velocity commands. Therefore, with a top-down view, as the targets moves further away the camera will first rise up, and then start following the target from behind.

Again, in comparison to open loop spline-motion, we manage to maintain a much more accurate trajectory thanks to the visual feedback and tracking information. As when studying the open loop system, it is hard to superimpose a velocity ground truth, which is up to scale and can be hardly reprojected in space. Nonetheless, we observe satisfying results in terms of global trajectory and responsiveness to target motion.



(a) 'Rise and hover' shot: As the target (ground truth) moves far away (travels vertically on frame), the camera rises and hovers the target (nb: motion from right to left on 'Camera Position' figure)



(b) 'Plunge and follow' shot: As the target travels, the camera dives and gets closer. Note that here the camera is actually placed above the target and has a top-down view, and that the second figure does shows relative position to the ground truth.

Figure 5.11: Spline-based motion (side view). In this experiment, the camera is constrained to travel along a predefined spline. We use the target information (ground truth) to trigger velocity commands. On the two examples shown here, we again demonstrate how the tracking data is used to improve the camera trajectory.

Chapter 6

Conclusion

This thesis demonstrates a self-driven Spidercam. As explained in the introduction creating complex shots with a Spidercam is hard, and requires the expertise of multiple operators. Instead, we presented our own piece of hardware and software that delivers a proof of concept on achieving similar shots in an autonomous fashion. Exploiting the power of deep-learning based tracking and 360° cameras, we came up with a low cost, portable and autonomous structure. We provide the user with predefined motion setups, and showed easily they can re-create classic cinematic shots. The results we obtained are very promising, and hopefully open a way for even more complex, refined and professional-looking shots. One could see many future applications for such a system. For instance it could be used to re-create timelapses, over more sophisticated paths than simple lines (typically obtained today with sliders). It could also help to automate light-field capture (see fig. 6.1), delivering spatial images arrays. However, we managed to identify in the previous section some important issues, and will address a few more in this chapter.

To conclude, this chapter first delivers a critical analysis of our system (see section 6.1), including an exhaustive list of its limitations, and then proposes paths for future work (see section 6.2).

6.1 Critical Analysis

Having studied our own prototype in the previous chapter, we already identified a couple of issues. We can split those issues in three categories: hardware-related, video tracking-related and autonomous driving related. In this section, we try to come up with an exhaustive list of limitations for our self-driven Spidercam.

6.1.1 Hardware

Examining the Spidercam in open loop (see section 5.1), we managed to identify two main issues. First, we found that while vertical motion was performing well, horizontal motion suffered greatly from the motors unreliable torque, leading to the camera "diving" over the course of dozens of centimeters. While an error in precision of a couple of centimeters is tolerable - given the scale of the Spidercam, such a plunge is very visible and prevents us from having an accurate motion control. Secondly, we found that the Spidercam presents visible oscillations, both vertically and horizontally. These vibrations occurred either when traveling at constant speed, or when starting and stopping abruptly. If adding velocity damping is a simple workaround to prevent oscillations due to brutal velocity changes, suppressing the former oscillations is much harder. Indeed, they are self-sustained by both the cable elasticity and motor poor torque. Once the camera starts slightly vibrating, the motor actual speeds are affected, and strengthen these vibrations. Thus, we should consider addressing those hardware limitations in a future version of the self-driven Spidercam.

6.1.2 Video tracking

The video tracking algorithm has proved to deliver very robust results, even on very poor quality reprojected streams. The tracking is fast, resistant to changes in lighting, appearance, out-of-plane rotations, and backlighting. Nonetheless, we can point out three flaws to our video-tracking module. First, we found that we are very dependent on the camera streaming server. As encountered in our experiments, occasional failures in streaming and frame freezing lead to major overshoot in the camera trajectory. Having no control on the streaming server, we have very little options to solve this issue, but perhaps opt for a more expensive or our own-crafted 360° camera. Secondly, we can notice that the reprojected and cropped 4K video is still not close to a professional-quality video. We have no control on common photography parameters like lens aperture, zoom or focus, and the cropped output is at a much lower resolution than now standard Full-HD. Again, the only improvement here would be to purchase a much higher resolution camera like the Insta360 Pro[1], which offers 8K filming. While this is an interesting path for an industrial autonomous Spidercam, it does not only comes at a very high cost, it also poses constraints of live-streaming and real-time processing. Lastly, even though we obtained satisfying video tracking performance, we could identify several ways of improving the SiamFC tracker. Indeed, it does not account for any online training, nor does it consider temporal consistency. Exploring those paths could make the tracker even more robust, and for instance help it to be more robust to partial or full occlusion.

6.1.3 Autonomous driving

The autonomous driving experiments showed how working in a closed loop with the tracker information helps improving the Spidercam overall performance. However, we found that we still have a $1.8s$ delay, which is responsive enough for slow-camera motion, but not for action-type filming. As explained earlier, this delay can hardly be reduced as it essentially stems from the stream acquisition and processing. Then, on spline-based motion we observe that the trajectory is often still not quite matching the desired input, because of hardware limitations.

Figure 6.1: Autonomous Spidercam applied to light field capture or 3D reconstruction: We setup a script to fly the camera along a predefined grid, and capture an object from a discrete array of viewpoints. Doing so, one could capture light fields and create effects such as refocusing, or 3D animations [47]. One could also imagine performing detailed 3D reconstruction with a minimal number of takes, using next best view estimation. The ability to let the Spidercam run by itself shows how promising it could be used to automate and refine such tedious tasks.

[1] https://www.insta360.com/product/insta360-pro/

6.2 Future Work

Having stated these issues, let us now explore paths for future work. In this section, we propose both short or long-term improvements that would greatly improve the Spidercam performance.

6.2.1 Motor sizing and stability

Studying the hardware, it has become clear that the Spidercam suffers from the motor's torque limitations. Despite our efforts to over-dimension the motors it seems like a high torque has a strong impact on the actual motor's velocity, leading to erroneous coordination in the cables movements. Thus picking a better, more reliable motor would greatly improve the trajectory precision, and avoid the camera going down on horizontal motion. However, chances are this will not help get rid of self-sustained oscillations. Even with a more robust motor, the cable elasticity and slight oscillations in motors velocity could be sufficient to generate and maintain vibrations. Therefore, we must find another way to compensate for these vibrations. One option would be to add a stabilizing head. This could be either a passive (using weights similar to those found on hand-held stabilizer) or active gimbal. A motorized stabilizing head could surely get rid of most oscillations, however it comes at a cost of extra weight. This would mean even bigger motors, and therefore an overall much higher cost.

Secondly, as suggested above, another improvement would be to opt for a more reliable 360° camera, or at least one on which we have control over the streaming server. This would prevent major overshooting and help make the Spidercam failuresafe. Again this comes at a higher cost, and a budget compromise may have to be found.

6.2.2 Target orientation and anticipation

Software-wise, we propose two paths for future work. The first one would be to make an even more clever tracking algorithm, that would not only try to localize a target, but also estimate its orientation. This could be especially useful if we were to design shots where the camera has to keep the same angle with an actor's face or any other type of target. Such an algorithm would add an extra layer of target information and would broaden the range of applications of the Spidercam. A second path would be to aim at reducing the delay in the autonomous motion control. Because it can hardly be reduced hardware-wise, we must come up with a software solution. One solution possible would be to perform motion anticipation. By training a system (*e.g.* a Gaussian Process) one could make predictions on the target motion and send anticipated commands to the Spidercam. Thus, we would artificially reduce the delay between target motion and camera motion. However, we must consider that we should account for different types of motion for different targets. Being able to let the Spidercam drive itself, the training phase could be easily automated over the course of many hours or days, with little requirement for supervision.

6.2.3 Deep-Learning based tracking

One reproach that could be made to the SiamFC Tracker is that it does not account for any online fine-tuning. The only on-the-fly adaptation being performed in the SiamFC algorithm is a template update using a rolling average. However, the networks parameters are not updated online, and thus do not account for some sequence-specific aspects, like changes in lighting, specific target motion or partial occlusion. This choice of not performing any online learning is deliberate. Updating such a network would involve methods like Stochastic Gradient Descents, which are too heavy to allow for a real-time performance. If we were to include an online training method, we would have to redesign the Network, using for instance a Correlation Filter. In the past, Correlation-Filter based tracking has proved to deliver very fast and robust results. They exploit the idea that correlating a template with a filter can be done in a very fast manner with a Fast Fourier Transform. Even more so, one can estimate efficient filters by formulating a minimization problem in the Fourier domain. One notable example of this idea is the Adaptive Correlation Filter made by Bolme et al. [45]. They achieve an outstanding 669fps tracking solely by computing a Minimum Output Sum of Squared Error (MOSSE) filter. This filter can

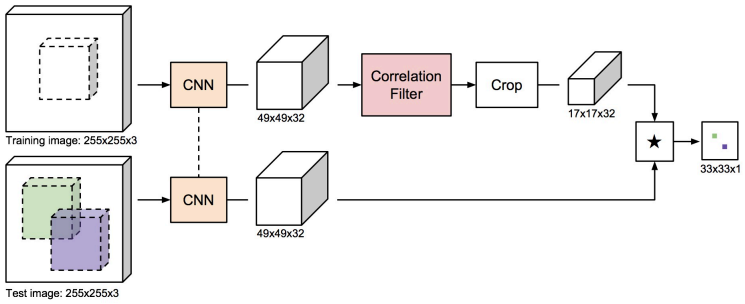


Figure 6.2: CFNet Architecture proposed by J. Valmadre et al [44]: An extra Correlation Filter is added on the upper branch to allow fast online retraining and achieve better results

be computed very efficiently in the Fourier Domain. However, as presented in their paper it does not work well with out of plane rotations, and suffers from drifting.

J. Valmadre et al. reused the useful properties of the Correlation Filters to build the CFNet, an extended version of the SiamFC Tracker [44], and obtain a even more robust real-time tracker. A Correlation Filter (CF) is added on top of one branch of the Siamese Network. The cross-correlation is then computed between the Siamese Network Descriptor of the Search Window, and the Correlation Filter Descriptor of the Template (see fig. below). One motivation behind the use of such an extra layer is that one can treat the Correlation Filter as a differentiable layer, which allows to back-propagate the errors in the Siamese Network and update their weights in an end-to-end fashion. The main advantage of the Correlation Filter lies in the fact that it provides a very efficient online training solution. Indeed, one can retrain the network online without any SGD, and rather by solving a ridge-regression problem in the Fourier domain [46]. By using a Correlation Filter, we both retain speed and accuracy. We get both an efficient on-the-fly learning, along with the possibility to use a pre-trained network. If the SiamFC Tracker already works very robustly for most sequences, this CFNet has proved to obtain even better results for smaller architectures. Due to the unavailability of such a pre-trained network in open-source, and with the time-constraints imposed by this research project, we chose to use the original, 5-layer baseline of the SiamFC Tracker. However, this is a promising path to explore for future improvements.

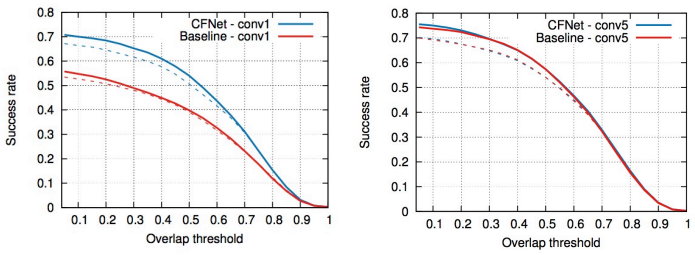


Figure 6.3: Success rates comparison on the VOT2016 validation set, for the original SiamFC Baselines [24] and CFNet versions [44]. Dotted lines describe a rolling average update rate of 0, while the solid lines represent a rate of 0.01 (our case). We find that for a 5-convolutional layer architecture (the one we use) improvements are not significant. However, for a much smaller network (a single convolutional layer, max-pooling, batch-normalization and ReLU), the improvements are outstanding, close to matching the results of the most robust SiamFC architecture. These promising results would allow us to obtain the results of a complex model

Bibliography

[1] Wenbin Li, Peter Hedman, Gabriel J. Brostow, "Prescription Camera Motion Control", draft

[2] Nebehay, G., and Pflugfelder, R. 2015. "Consensus-based Matching and Tracking of Keypoints for Object Tracking". In *IEEE Conference on Computer Vision and Pattern Recognition* (CVPR), 2784–279.

[3] E. Maggio and A. Cavallaro, "Video Tracking: Theory and Practice", 2011

[4] A. Schödl, R. Szeliski, D. H. Salesin, I. Essa, "Video Textures", Siggraph 2000

[5] Matej Kristan, Aleš Leonardis, Jiri Matas, Michael Felsberg, Roman Pflugfelder, Luka Čehovin, Tomas Vojir, Gustav Häger, Alan Lukežič, and Gustavo Fernandez, "The Visual Object Tracking VOT2016 Challenge Results", *ECCV* 2016 Workshops, Springer, 2016.

[6] Matej Kristan, Jiri Matas, Aleš Leonardis, Michael Felsberg, Luka Čehovin, Gustavo Fernandez, Tomas Vojir, Gustav Häger, Georg Nebehay et al., "The Visual Object Tracking VOT2015 challenge results" Visual Object Tracking Workshop 2015 at *ICCV2015*, 2015

[7] Matej Kristan, Roman Pflugfelder, Aleš Leonardis, Jiri Matas, Luka Čehovin, Georg Nebehay, Tomas Vojir, Gustavo Fernandez, Alan Lukežič et al., "The Visual Object Tracking VOT2014 challenge results" Visual Object Tracking Workshop 2014 at *ECCV2014*, 2014

[8] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, Volume: 86, Issue: 11, Nov 1998

[9] Daniilidis, K., Krauss, C., Hansen, M., and Sommer, G. 1998. "Real-time tracking of moving objects with an active camera", *Real-Time Imaging* 4, 1, 3–20.

[10] Smeulders, A. W., Chu, D. M., Cucchiara, R., Calder-Ara, S., Dehghan, A., and Shah, M. 2014. "Visual tracking: An experimental survey" *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 7, 1442–1468.

[11] U, Y., Lim, J., and Yang, M.-H. 2013. "Online object tracking: A benchmark". In *IEEE Conference on Computer vision and pattern recognition* (CVPR), IEEE, 2411–2418.

[12] Kalal, Z., Mikolajczyk, K., and Matas, J. 2012. "Tracking-learning-detection" In *IEEE Transactions on Pattern Analysis and Machine Intelligence* (PAMI) 34, 7, 1409–1422.

[13] Lijun Wang, Wanli Ouyang, Xiaogang Wang, and Huchuan Lu1, "Visual Tracking with Fully Convolutional Networks", In *IEEE International Conference on Computer Vision* 2015

[14] N. Wang and D. Yeung. "Learning a deep compact image representation for visual tracking" *NIPS*, 2013. 2, 8

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks" *NIPS*, 2012. 1, 2, 3

[16] R. Girshick, J. Donahue, T. Darrell, and J. Malik "Rich feature hierarchies for accurate object detection and semantic segmentation" *CVPR*, 2014.

[17] H. Li, Y. Li, and F. M. Porikli, "Robust online visual tracking with a single convolutional neural network" *ACCV*, 2014.

[18] Han, X., FU, H., Zheng, H., Liu, L., and Wang, J. 2013 "A video-based system for hand-driven stop-motion animation" *IEEE Computer Graphics and Applications* 33, 6 (Nov), 70–81.

[19] Kopf, J., Cohen, M. F., and Szeliski, R. 2014. "First-person hyper-lapse videos" *ACM Transactions on Graphics* (TOG) 33, 4, 78.

[20] Manohar Srikanth, Kavita Bala, Frédo Durand, "Computational rim illumination of dynamic subjects using aerial robots", *Computers & Graphics* 52 (2015) 142–154

[21] N. Joubert, M. Roberts, A. Truong, F. Berthouzoz, and P. Hanrahan "An interactive tool for designing quadrotor camera shots", ACM Transactions on Graphics, SIGGRAPH Asia 2015.

[22] M. Kristan, J. Matas, A. Leonardis, T. Vojir, R. Pflugfelder, G. Fernandez, G. Nebehay, F. Porikli and L. Cehovin, "A Novel Performance Evaluation Methodology for Single-Target Tracker", *IEEE Transactions on pattern analysis and machine intelligence*, 2016

[23] Hyeonseob Nam, Bohyung Han, "Learning Multi-Domain Convolutional Neural Networks for Visual Tracking", arXiv:1510.07945 2015

[24] L. Bertinetto, J. Valmadre, J. F. Henriques, A. Vedaldi, P. H. S. Torr, "Fully-Convolutional Siamese Networks for Object Tracking", Department of Engineering Science, University of Oxford, arXiv:1606.09549v2, 2016

[25] E. Simo-Serra, E. Trulls, L. Ferraz, I. Kokkinos, F. Moreno-Noguer, "Fracking Deep Convolutional Image Descriptors", ICLR 2015, arXiv:1412.6537v2

[26] J. Redmon, A. Farhadi,"YOLO 9000: Better, Faster, Stronger", University of Washington, Allen Institute for AI, arXiv:1612.08242v1 2016

[27] J. Redmon, A. Farhadi, S. Divvala, R. Girshick, "You Only Look Once: Unified, Real-Time Object Detection", University of Washington, Allen Institute for AI, Facebook AI Research, http://pjreddie.com/yolo/ 2016

[28] R. Girshick,"Fast R-CNN", Microsoft Research, arXiv:1504.08083v2, 2015

[29] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", arXiv:1506.01497v3, 2016

[30] C. Gu, J. J. Lim, P. Arbelaz, and J. Malik "Recognition using regions", *CVPR*, 2009.

[31] K. E. A. van de Sande, J. R. R. Uijlings, T. Gevers, A. W. M. Smeulders, "Segmentation As Selective Search for Object Recognition", *IEEE International Conference on Computer Vision*, 2011

[32] P. F. Felzenszwalb, R. B. Girshick, D. McAllester and D. Ramanan, "Object Detection with Discriminatively Trained Part Based Models" *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 32, No. 9, September 2010.

[33] K.He,X.Zhang, S.Ren, and J.Sun, "Spatial Pyramid Pooling in Deep Convolutional Networks for visual recognition" *ECCV,2014* 1,2,3,4,5,6,7

[34] K. Simonyan, A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", arXiv:1409.1556, 2015

[35] K. Simonyan, A. Vedaldi, A. Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", arXiv:1312.6034v2, 2014

[36] K.-K.Sungand, T.Poggio, "Example-based learning for view-based human face detection" *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(1):39–51, 1998

[37] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich "Going deeper with convolutions" CoRR, abs/1409.4842, 2014

[38] J. Long, E. Shelhamer, T. Darrell, "Fully Convolutional Networks for Semantic Segmentation", UC Berkeley, CVPR2015

[39] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, Y. LeCun "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks", arXiv:1312.6229

[40] J. Bromley, I. Guyon, Y. LeCun, E. Sickinger, R. Shah "Signature Verification using a Siamese Time Delay Neural Network",AT&T Bell Laboratories, 1994

[41] R. Tao, E. Gavves, A.W.M. Smeulders, "Siamese Instance Search for Tracking", arXiv:1605.05863,

[42] D. Held, S. Thrun, S. Savarese, "Learning to Track at 100 FPS with Deep Regression Networks", Stanford University, arXiv:1604.01802v2, 16 Aug 2016

[43] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", University of Toronto, NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing Systems, Pages 1097-1105

[44] J. Valmadre, L. Bertinetto, J.F. Henriques, A. Vedaldi, P. H. S. Torr, "End-to-end representation learning for Correlation Filter based tracking", University of Oxford, CVPR 2017, arXiv:1704.06036v1

[45] D. S. Bolme, J. R. Beveridge, B. A. Draper, Y. M. Lui, "Visual Object Tracking using Adaptive Correlation Filters", CVPR 2010

[46] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, "High-Speed Tracking with Kernelized Correlation Filters", arXiv:1404.7584v3, 5 Nov 2014

[47] R. Ng, "Digital Light Field Photography", PhD Thesis, July 2006

[48] Weisstein, Eric W. "Stereographic Projection." From MathWorld–A Wolfram Web Resource, `http://mathworld.wolfram.com/StereographicProjection.html`

[49] "Planisphaerium" –A Wikipedia Web Resource, `https://en.wikipedia.org/wiki/Planisphaerium`

[50] Dynamixel Robotis Web Resource `http://support.robotis.com/en/techsupport_eng.htm`