Energy-Sensitive Vision-Based Autonomous Tracking and Landing of a UAV

Georgios Zamanakos, Adam Seewald, Henrik Skov Midtiby, and Ulrik Pagh Schultz
SDU UAS Center, Mærsk Mc-Kinney Møller Institute, University of Southern Denmark
Contact email: ups@mmmi.sdu.dk

Abstract

In this paper, we present a robust, vision-based algorithm for autonomous tracking and landing on a moving platform in varying environmental conditions. We use a novel landing marker robust to occlusions to track the moving platform and the YOLOv3-tiny CNN to detect ground-based hazards in an agricultural use case. We perform all computations onboard using an NVIDIA Jetson Nano and analyse the impact on the flight time by profiling the energy consumption of the landing marker detection algorithm and YOLOv3-tiny CNN. Experiments are conducted in Gazebo simulation using an energy modeling tool to measure the energy cost as a function of QoS. Our experiments test the energy efficiency and robustness of our system in various dynamic wind disturbances. We show that the landing marker detection algorithm can be run at the highest QoS with only a marginal energy overhead whereas adapting the QoS level of YOLOv3-tiny CNN results in a considerable power saving for the system as a whole. The power saving is significant for a system executing on a fixed-wing UAV but only marginal if executing on a standard multirotor UAV.

I. INTRODUCTION

Unmanned Aerial Vehicles (UAVs) are increasingly used for applications such as monitoring, surveillance, transportation of small payloads, and agricultural applications [**?**], [**?**]. One of the major constraints of such applications is their limited level of autonomy due to battery limitations. Extending the flying time of a UAV is normally done by having it land in order to replace or charge the battery before continuing the mission. Performing landings autonomously can however be challenging depending on the environment and whether the landing platform is stationary or mobile. Moreover, relying solely on the availability of a Global Positioning System (GPS) signal for autonomous precision landing is not considered safe, since GPS signals can be temporarily lost or even tampered with. As an alternative, in this paper we investigate the use of a novel vision-based autonomous landing system and evaluate its robustness towards environmental conditions such as visual disturbances and wind.

Extension of the flight time can be also achieved by using *energy-sensitive algorithms* that can reduce energy consumption by reducing the Quality of Service (QoS). With this approach, energy-costly computations such as computer vision are adapted by selecting the desired quality of service to match the available energy [**?**]. By combining energy-sensitive algorithms with autonomous landing capabilities, we aim to increase the total availability of the UAV to perform operations, by extending the flight time and using autonomous recharging when needed.

The main contribution of this paper concerns the experimental study of a robust vision-based algorithm for autonomous tracking and landing in varying environmental conditions. The algorithms are executed on an NVIDIA Jetson Nano companion computer controlling a simulated drone. The vision-based tracking and landing algorithms provide novel capabilities in terms of tolerance to visual disturbance and varying environmental conditions such as wind. Our experiments are based on an agricultural use-case where a multirotor UAV performs visual identification of ground-based hazards while tracking and landing on a moving platform.

II. State of the Art

Vision-based autonomous landing on a marker has been extensively studied by many researchers. Key distinctions include whether the marker is on a moving platform, the type of the marker, the algorithms used to detect it, as well as the mounted sensors on-board of the UAV.

For stationary platforms, one of the first experiments with vision-based autonomous landing was conducted by Saripalli et al. [**?**]. Here, a helicopter with a color camera facing vertically towards the ground would land on an ”H”-shape pattern (similar to ones found on a helipad) using a hierarchical behavior-based control architecture. In physical tests a marker of $122{\times}122\,\mathrm{cm}$ size was detected for a maximum altitude of $10\,\mathrm{m}$. A landing marker inspired by a QR code but consisting of three artificial markers is demonstrated by Yuan et al. [**?**], and was shown to provide a $6-$Degree Of Freedom (DOF) pose over an altitude range of 0-$20\,\mathrm{m}$. Our work is however focused on the ability to land on moving platforms.

Saripalli et al. [**?**] also demonstrated the use of a Kalman Filter to track a moving platform. However, all the computations were performed offline. Similarly, an ArUco marker was used as a landing marker by Lee et al. [**?**] to detect a moving platform. The control of the UAV is performed based on the error provided by the vision algorithm but all the computations were performed off-board. Arrar, et al. [**?**] focus on extending the detection range by using an AprilTag [**?**] as a landing marker. Again all the computer vision algorithms were also executed off-board. Conversely, a crucial aspect of our application is to perform all the computations on-board, and to evaluate them according to their energy efficiency as a function of QoS.

The design of the marker and choice of sensors can facilitate doing the computations onboard. Chen et al. [**?**] utilized a marker consisting of a circle and rectangles of different colors along with a LiDAR scanning range finder for height estimation. The marker was detected by performing color segmentation on the incoming image frame. By fusing the height measurement from the LiDAR into the vision measurement, a relative pose of the UAV from the moving platform was obtained. A color segmentation approach was also implemented by Lee et al. [?]. A red rectangle was used as a landing marker while the detection was done by a vertically facing camera with a fish-eye lens. The setup accounted for a successful landing from an altitude of $70\,\mathrm{m}$. In the above two cases an on-board companion computer is used to perform all the computations on the UAV. However, we in this work do not consider a color segmentation approach as a safe option, since for a realistic (outdoor) case it would be difficult, if not impossible, to ensure that the landing marker will be the only object of a specific color in the scene.

The use of a hybrid camera system consisting of a fish-eye IR camera and a stereo camera was demonstrated by Yang et al. [?]. An ArUco marker was used to mark the moving platform and a convolutional neural network (CNN) YOLO v3 was trained specifically for marker detection. A similar approach concerning the detection of a landing marker was demonstrated by Nguyen et al. [?] in which a specific CNN was trained to detect a landing marker pattern: successful detection of a $1\,\mathrm{m}\times 1\,\mathrm{m}$ marker size was demonstrated from a distance of $50\,\mathrm{m}$. An AprilTag marker was used as a landing marker by Kyritsis et al. [?]. The identification of the AprilTag marker was performed through Graphics Processing Unit (GPU). In the above three cases, the researchers have utilized the companion's computer GPU to detect the landing marker. In the agricultural use-case addressed in this paper, the GPU is however needed for a CNN to detect ground hazards, and since the GPU cannot normally run different algorithms simultaneously, the CPU should be used for detecting the landing marker. By doing so, a different QoS can be chosen for each algorithm.

To account for the energy modeling of computer vision algorithms, we considered the work previously carried by Nardi et al. [?]. The authors present SLAMBench, a framework that investigates Simultaneous Localisation and Mapping (SLAM) algorithms configuration alternatives for energy efficiency. In our work, we use `powprofiler`, a generic energy modeling tool [**?**]. This tool enables measuring the energy impact of different configurations of the ROS-based system implementing the agricultural use-case. The `powprofiler` tool is part of the TeamPlay toolchain, which aims to make tradeoffs between energy and other non-functional properties accessible to the developer. In this paper, we present extensions to `powprofiler` that facilitates the initial exploration of the energy usage of complex ROS-based systems.

Other approaches to energy modeling, such as the mission-based energy models studied by Sadrpour et al. [**?**], [**?**], focus mostly on ground-based autonomous vehicles instead of the UAVs. Morales et al. [**?**] extensively investigated the relation between motion and energy in a robot, but do not account for the energy required for computation. Energy modeling of mobile robots as carried by Mei et al. [**?**], [**?**], [**?**] has provided the ground for the concept of modeling computation for energy-sensitive algorithm design. Indeed, the approach employed in this paper has evolved from an energy-efficient motion planning technique in [**?**], a design strategy that allows accounting for motion and computations separately in [**?**], to an energy-efficient deployment algorithm in [**?**].

The battery in our system is considered in the context of a drone being able to perform its mission while accounting for the eventuality of a battery shortage; to this end, we investigated the approach presented by Berenz et al. [**?**], where a battery management mission-based dynamic recharge approach is presented. A set of recharge stations are used, along with self-docking capable robots. Our approach similarly allows landing on a moving platform for recharging, which is in the context of this paper is considered in the proximity of the drone. The actual landing is handled by the proposed algorithm, and we also account for the energy required for executing this algorithm during landing.

Taking into account varying environmental conditions and unpredictable movements of the platform to land on is relevant for the use of landing in outdoor, mobile scenarios. Regarding wind conditions, an AprilTag marker was used by Feng et al. [?] with a constant wind speed of $5\,\mathrm{m/s}$ as an external disturbance in a simulation environment. Nevertheless, a fluctuation in the wind's magnitude and direction is likely to happen in realistic cases. Concerning estimation of the moving platform's position and velocity, similar to our approach a Kalman Filter or Extended Kalman Filter (EKF) has been used for the estimation [?], [?], [?], whereas Yang et al. [?] constructed a velocity observer algorithm by calculating the actual moving distance of the moving platform over a period of time.

III. Energy-Sensitive Mission Deployment

A. Overall approach

The energy-sensitive design is a mission-oriented concept that adjusts the computations to the mission being performed while taking into account energy requirements, including energy consumed by actuation, computation, and the presence of a limited power source. Specifically, in our agricultural use-case, the concept is employed to profile and eventually adapt the computationally heavy algorithms performing autonomous tracking, landing, and hazard detection. This adaptation enables energy-sensitivity, in the sense that QoS parameters can be modified to enable the mission to be completed at the highest possible QoS level that does not exceed the available energy budget. Tradeoffs between QoS parameters can be performed by an end-user, i.e., trading the robustness towards wind during landing for precision of hazard detection.

The energy-sensitive design using `powprofiler` relies on empirical experiments to measure the actual power consumption on the robot hardware [?]. In this paper, we focus on the initial profiling using of energy usage of the companion computer, which from the point of view of energy consumption can be studied independently from the specific drone it is mounted in.

First, the developer specifies the maximum and minimum QoS level for each algorithm running on the system. During mission execution the levels are statically defined: automatic adaptation during different phases of the mission is currently being investigated and is considered future work. Then, the developer executes the system to empirically determine the power consumption. This can be done in two different ways:

1) Automatically using `powprofiler` to control the experiment execution [**?**]. For a ROS-based system, we assume that the algorithms are wrapped as ROS nodes, and we require the developer to specify the QoS parameters using a ROS configuration. We use a configuration file in a key-value pair format which is then interpreted by `powprofiler`, enabling `powprofiler` to iterate through all possible combinations and empirically sample the energy consumption of each combination of QoS parameters. Once all combinations have been iterated through, `powprofiler` automatically combines the energy consumption data into a complete model.
2) Semi-automatically using `powprofiler` to sample energy and combine the results of all experiments, but allowing the developer to control all aspects of the experiment execution. This approach is new and is described in more detail later in this section. Basically, a ROS node interfaces to `powprofiler` and is used by the developer to start/stop sampling in a given configuration. Once all experiments have been completed, `powprofiler` is invoked by the developer to combine the energy consumption data into a complete model.

Regardless of the approach, `powprofiler` builds a single model mapping QoS to total system energy consumption. Coarse-grained sampling is employed to reduce the number of experiments, and missing values are automatically inferred from the others by the means of a multivariate linear interpolation.

In the context of this paper, sampling experiments are iterated in a simulated environment with different configurations. For example, the autonomous tracking allows changing the tracking algorithm QoS in terms of frequency, the landing algorithm in terms of frequency, and hazard detection QoS in terms of frequency.

B. Semi-automatic energy profiling

A ROS node has been developed for the purposes of the semi-automatic approach used in this paper. This node allows automatic generation of the basic energy models that map time to the instantaneous power consumption. To activate this functionality, the developer simply publishes on a ROS topic to start the model generation, with `powprofiler` accounting for the invocation of an asynchronous thread which collects data from the energy sensors. Similarly, the developer publishes on another ROS topic to stop the model generation, while `powprofiler` finalizes collecting data from sensors, builds the basic energy model, and stores it for later processing.

Once all the basic energy models for the desired QoS ranges have been collected, QoS ranges are specified in a configuration file: the developer defines what QoS configuration corresponds to which basic model (instantaneous power consumption as a function of time). Running `powprofiler` using this configuration file as a parameter generates the complete model that maps QoS to energy consumption.

IV. Vision-Based Autonomous Tracking and Landing

The vision-based autonomous tracking and landing can be split into four main sub-problems: detection of the moving platform, navigation, guidance, and control of the UAV. From an energy-sensitive design approach, our focus is on the computer vision algorithms used to detect the moving platform and the parameterization by a QoS influencing energy consumption and performance, as described in Section **??**. Furthermore, the navigation block is designed to increase the robustness and overall performance of the system, as described in Section **??**. Last, a model

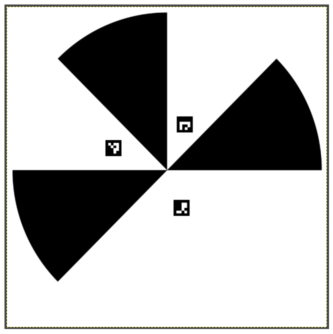

Fig. 1. The landing marker

of dynamically changing wind disturbances is analysed and described in Section **??**, allowing the system to be tested in a more realistic simulation.

A. Detection of the moving platform

To mark the moving platform a special pattern is constructed, consisting of an n-fold marker [**?**] along with three ArUco markers [**?**] with different ids. This pattern will be referred to as the *landing marker* and can be seen in Figure **??**. The n-fold marker is primarily used to detect the moving platform from a high altitude, while the ArUco markers are used as extra landmarks in case the marker is partially visible in the image frame.

To evaluate the computer vision algorithm for detecting the landing marker, real images of 640×480 pixel size were captured with an Intel RealSense D435 camera. In the Gazebo simulation a color camera with the same distortion coefficients as the Intel camera is used to output a 640×480 pixel image at 10 frames per second (fps).

To extract the pixel coordinates of the tip of the n-fold marker, a kernel size of 13×13 pixels consisting of a real and imaginary part is created. For every pixel in the image, a convolution is performed with this kernel and the magnitude of the convolution is stored. The pixel with the highest magnitude is considered as a candidate tip of the n-fold marker. For that candidate pixel only, an estimation of the orientation/phase of the marker

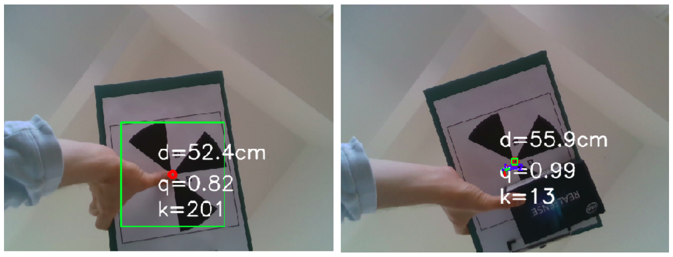

Fig. 2. Detection of the landing marker under different occlusions. On the left an occlusion on the tip of the n-fold marker and on the right an occlusion on a sector of the n-fold marker.

is made and an overall normalized quality score between 0.0 and 1.0 is calculated. If the score is above a desired threshold value then the pixel is accepted as the tip of the n-fold marker. If an n-fold marker is detected, the result will be the pixel coordinates of the tip of the n-fold marker along with its orientation/phase.

Since a convolution is a computationally expensive process, an increase in the kernel size would also increase the computation time and therefore the energy consumption. However, a higher computation time and energy consumption is preferred over a non-detection of the n-fold marker. To balance between energy consumption and effective marker detection, an adaptive kernel selection function is created to ensure the selection of a proper kernel size based on a threshold quality score value. In Figure **??** an Intel RealSense D435 Depth camera is used to capture the image and measure also the distance d from the n-fold marker. Based on a desired quality q, the proper kernel size k is selected. It is seen that an occlusion on the tip of the n-fold marker results in a significant increase in the selected kernel size.

To detect the ArUco markers the standard OpenCV library is used and if any ArUco markers are detected, their central pixel coordinates and pose are stored.

The next step is to convert these pixel coordinates into a real-world relative position *[X, Y, Z]* according to a local coordinate frame. The origin of the local coordinate frame *[0, 0, 0]* is defined as the center of the landing marker and alignment according to the North, East, Down (*NED*) frame. The available sensor measurements and sensor fusion algorithms from the flight controller are used in this process. The PX4 flight controller outputs through mavlink messages the altitude and the attitude of the UAV (roll, pitch, yaw). For the *Z* component, the altitude of the UAV from the flight controller's EKF is used. To obtain the *X, Y* components, an algorithm is constructed to convert pixel coordinates into real world *X, Y* coordinates in meters:

1) The pose of the camera in UAV's *BODY* frame is calculated by utilizing the roll and pitch IMU data.
2) The normalized coordinates of the four image corners, according to the camera's horizontal, vertical field of view and the camera's pose from step 1, are calculated.
3) The coordinates of the four image corners (in meters), with respect to the UAV's *BODY* frame, are determined by using the normalised coordinates from step 2 along with the UAV's altitude. The result is a projection plane of the image corners on the ground.
4) The perspective homography matrix is calculated between the two planes, the image plane and the world plane from step 3.
5) The homography matrix from step 4 is used to convert the pixel coordinates of the tip of the n-fold marker from the image plane, into coordinates (in meters) in the UAV's *BODY* frame.
6) The coordinates from step 5 are in respect to the UAV's *BODY* frame. To convert them into the UAV's *NED* frame, the yaw IMU data from the flight controller is used.
7) The coordinates from step 6, are converted from UAV's *NED* frame into the landing site's local coordinate frame.
8) For ArUco markers only, an offset vector in the x, y axis is added depending on the distance of each ArUco marker from the tip of the n-fold marker. It is assumed that this vector is prior known.

The mean measurements from the detected n-fold and/or ArUco markers are used to determine the position and orientation of the landing marker. The result is an *[X, Y, Z]* relative position of the UAV from the moving platform along with the yaw orientation of the landing marker.

B. Navigation

The purpose of the navigation block is to provide an accurate prediction for the state of the UAV at any given time. This is important because it will allow us to process images at different fps according to a desired QoS. Furthermore, the overall robustness of the system is increased in case the moving platform is not detected in every image frame. A velocity estimator for the moving platform will also be implemented as a part of the navigation block.

The variables of interest that describe the state of the UAV for this project are:

- The relative position of the UAV from the moving platform, x_k, .
- The attitude of the UAV *[roll, pitch, yaw]*, obtained from the flight controller's IMUs.
- the velocity of the UAV, $\dot{x}_k$, in *NED* frame, obtained from the flight controller's EKF.
- the acceleration of the UAV, $\ddot{x}_k$, in *NED* frame, obtained by differentiating the velocities.

I think we might have used a bad name for this value. It currently overlaps with the estimated position of the UAV relative to the landing target.

The altitude, attitude, velocity and acceleration variables of the UAV's state are obtained from the flight controller's onboard sensors and already implemented sensor fusion algorithms (EKF). To fuse those state variables with the obtained position measurements from Section **??**, a Kalman Filter is implemented. The measurements from the flight controller are used in the prediction step.

These values are also described on the next page.

Prediction Step:

$$\hat{x}_k = F \cdot \hat{x}_{k-1} + G_k \cdot u_k, \tag{1a}$$

$$P_k = F \cdot P_{k-1} \cdot F^\top + Q_k, \tag{1b}$$

where $x_k \in \mathbb{R}^2$ is the position of the UAV, $u_k = \begin{bmatrix} \dot{x}_k & \ddot{x}_k & \dot{x}^l_k \end{bmatrix}^\top$ is its control with x_k, x^l_k being the positions of the UAV and the moving paltform respectively. Given $I := I_2$, a 2×2 identity matrix, the matrix $F = I$, the noise covariance matrix $Q_k = \Delta t_k \cdot \sigma^2_{\text{IMU}} \cdot I$, and $\sigma^2_{\text{IMU}} \in \mathbb{R}$ is the variance of the velocities retrieved from the flight controller's data. Further, Δt_k is the time interval the flight controller output the data (usually around 33 ms). The input matrix is given by:

Is this the right word? [Ad3:] actually no/yes– diatrabes between controls people stating that the model of the system is perfect since is a model [cite Hector]

Henrik, little lengthy with implicitly saying "is a 2 by 2 identity matrix"; what about this one?

$$G_k = \begin{bmatrix} \Delta t_k & 0 & \frac{\Delta t_k^2}{2} & 0 & -\Delta t_k & 0 \\ 0 & \Delta t_k & 0 & \frac{\Delta t_k^2}{2} & 0 & -\Delta t_k \end{bmatrix},$$

[Ad3:] Actually G_k might still go easily inline. Something like $G_k = \Delta t_k \begin{bmatrix} I & \Delta t_k I/2 & -I \end{bmatrix}$. Up to you

The system is initialized with $P_0 = I$ and $\hat{x}_0$ coming from the initial measurement.

[Ad3:] the itemize doesn't really add any information the reader doesn't know so far (takes space also)... Makes sense in a thesis, report, not in a paper.

- x_k is the position of the UAV in the landing site's local coordinate system (aligned with NED frame),
- Δt_k is the time interval the PX4's EKF outputs the v_x, v_y data, usually around 33ms,
- $\dot{x}_k$ is the linear velocity of the UAV in NED frame, estimated from the PX4's EKF,
- $\ddot{x}_k$ is the linear acceleration of the UAV in NED frame, estimated on the companion computer from differentiating the velocities.
- $\dot{x}^l_k$ is the linear velocity of the moving platform in NED frame,

estimated from the velocity estimator for the moving platform,

- σ^2_{IMU} is the variance of the velocities based estimated by the PX4's EKF.

[Ad3:] from here to windspeed, I just apply my earlier math suggestion. Henrik, Georgios please check/change/revise.

In the correction step, the observed measurements from the downward looking camera are used to correct and update the estimated position of the UAV $\hat{x}_k$. However due to the computation time needed to detect the landing marker, the incoming measurement is delayed by a time $\Delta t^{\text{obs}}_k \in \mathbb{R}_{\geq 0}$. Physically, let us define the displacement $\tilde{x}^{\text{obs}}_k \in \mathbb{R}$ as

$$\tilde{x}^{\text{obs}}_k := \bar{\dot{x}}_k \, \Delta t^{\text{obs}}_k, \tag{2}$$

where $\bar{\dot{x}}_t \in \mathbb{R}$ is the mean velocity measured empirically. Such displacement is later employed in the observed measurements. The correction step of the Kalman Filter is

$$K_k = P_k \cdot H^T (H \cdot P_k \cdot H^T + R_k)^{-1}, \tag{3a}$$

$$\hat{x}^f_k = \hat{x}_k + K_k (x^{\text{obs}}_k + \tilde{x}^{\text{obs}}_k - \hat{x}_k), \tag{3b}$$

$$P^f_k = (I - K_k \cdot H) P_k, \tag{3c}$$

where $H = I$, and x^{obs}_t is the position measured empirically at time t. Let us further define $R_t := \sigma^2_{\text{obs}} I$ where $\sigma^2_{\text{obs}} \in \mathbb{R}$ is the variance of the velocities measured empirically.

The values of the estimated state $\hat{x}^f_t$, and error covariance matrix P^f_t, are used as input to the next iteration of the prediction step $\hat{x}_{t+1}, P_{t+1}$.

A velocity estimator is also constructed to determine the magnitude and direction of the moving platform's velocity vector. The magnitude is calculated by differentiating two sequential *X, Y* positions of the tractor, obtained by detecting the n-fold marker from the vertically facing camera as explained in section **??** and taking into consideration the UAV's *NED* velocities according to the PX4's EKF. A low-pass filter is used to provide a smooth estimation of the velocity's magnitude by filtering out high frequency noise. High frequency noise can be caused by oscillations of the UAV along with a fast update rate in the landing marker detection algorithm.

[Ad3:] confusing a bit; if we differentiate to estimate the velocity (dx/dt if x is the position) how we use low pass filter to estimate the same velocity we differentiated?

In the agricultural use-case the moving platform is likely to change its direction up to 180 degrees. Furthermore, it is assumed that the moving platform is a nonholonomic system, like a tractor. To compensate for sudden turns, the moving platform's yaw orientation will be taken into account. Based on the velocity's magnitude and moving platform's yaw orientation, the moving platform's velocity ($\dot{x}_k^l$) in *NED* frame can be obtained.

C. Wind Disturbances

[Ad3:] the force is F. Why are we then calling it $force$ later on?

The exact and accurate estimation of the applied wind forces on a body is a complex matter studied by the field of fluid dynamics. We use a simplified approach, as follows. We assume that the wind will be applied on an area of $0.09\,\mathrm{m}^2$. Such an area is emulating a UAV with extra payload attached on its frame. Two different wind speeds of $8\,\mathrm{m/s}$ and $12\,\mathrm{m/s}$ will be used to calculate the applied wind forces on that area. The wind forces are considered to be applied on the center of gravity of the UAV with direction parallel to the ground.

The magnitude of the applied force, corresponding to a certain wind speed is calculated by the following equations [**?**], [**?**]:

[Ad3:] the force is F. Why are we then calling it $force$?

$$\begin{aligned} F &= p_d \cdot A \\ p_d &= \tfrac{\varrho \cdot v^2}{2} \end{aligned} \tag{4}$$

[Ad3:] can be one line, and actually, I would trust the reader to search for this one. Is really basic physics.

where F is the force, p_d is the dynamic pressure, A is the area of the applied pressure, ϱ is the density of the air (around $1.2\,\mathrm{kg/m^3}$), v is the wind velocity in $\mathrm{m/s}$.

By solving the above equations the applied force on the UAV is found to be $3.45\,\mathrm{N}$ for an $8\,\mathrm{m/s}$ wind speed, and $7.76\,\mathrm{N}$ for a $12\,\mathrm{m/s}$ wind speed. The wind forces will be applied on the UAV in Gazebo simulation, to test the performance of the whole system. This will be done by constructing a program that will apply the wind forces on the virtual UAV.

To simulate a wind pattern the wind direction and magnitude must be defined. The wind direction is assumed to remain the same for the whole duration of the experiments. That direction vector is defined as *[0.8, 0.2]* according to Gazebo's *(x, y)* axes. The magnitude of the wind is calculated as follows:

- An initial wind force of either $3.45\,\mathrm{N}$ or $7.76\,\mathrm{N}$ is chosen.
- An update cycle of $5.5\,\mathrm{s}$ is chosen between two different applied forces.
- A random float number between 0.9 and 1.2 is chosen at every update cycle. This number is defined as *Random_noise*.
- The applied wind force is calculated as:

  [Ad3:] this is just $F_k = w \cdot F \cdot r$? where w, r is the direction vector you defined earlier *[0.8, 0.2]* and r a random number.

  $$force_x = -x_{norm} * force * Random\_noise$$
  $$force_y = -y_{norm} * force * Random\_noise$$

  where $x_{norm} = 0.8$, $y_{norm} = 0.2$

  For an altitude of $6.0\,\mathrm{m}$ and above, *force* $=$ $force_{init}$ where $force_{init}$ is either $3.45\,\mathrm{N}$ or $7.76\,\mathrm{N}$
- For a UAV altitude of $6.0\,\mathrm{m}$ and below, the wind force decreases as:

$\textit{force} = \textit{force}_{init} * (\textit{altitude}/6.0)$

- For a UAV altitude of $3\,\mathrm{m}$ and below, *force* $= 0.5N$.

This model is created to simulate different scaling in the wind's magnitude and will be used for all simulated tests.

V. Evaluation

We evaluate our approach in terms of the quality of the overall functionality and the energy efficiency of the algorithms. All algorithms are executed on an embedded companion computer interfaced to a simulation running on a standard computer.

A. Use-case: agricultural safety

We evaluate our approach based on a simulated use-case where a multirotor UAV identifies hazardous objects around a moving platform, and lands on the moving platform to recharge. No communication link is considered between the moving platform and the UAV and no GNSS positioning is assumed to be available. The system can thus be considered as a fallback for fault-tolerance.

Object detection and classification is performed by feeding the input image from the downward facing camera into a *YOLOv3-tiny* CNN [**?**] implemented in ROS [**?**]. Four different classes are selected: cars, humans, tractors, and cows. Based on the CNN's predictions and onboard sensors, the UAV maps the detected objects onto a 2D map.

A simulated field is created in Gazebo with objects placed in random positions and orientations as seen in Figure **??**.

Pre-trained weights for *YOLOv3-tiny*, were initially used but the performance on detecting objects from a downward facing camera was not satisfactory. Since a dataset for detecting the above four classes from a top view was not available, we created an artificial dataset based on the Gazebo models. The dataset consists of 1200 images, 300 for each class. We trained the *YOLOv3-tiny* by its default training parameters for 5000 epochs, for an input image size of 416×416 pixels.

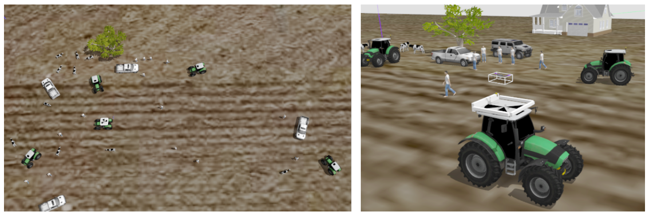

Fig. 3. On the left, a top view of the Gazebo scene. On the right, a view of the UAV attempting a landing on the moving platform

B. *Experimental setup*

All experiments are performed in Gazebo simulation under Ubuntu 18.04 and ROS Melodic on a i7-8550U 1.8GHz (4.0GHz Boost), 8GB DDR4 laptop. The *PX4* Software In The Loop (SITL) Firmware v1.10.2 is used as the flight controller and the *IRIS* quadcopter is used as the UAV platform. A vertically facing RGB camera is placed on the UAV providing a 640×480 pixel image at 10 fps. An *Nvidia Jetson Nano* with Ubuntu 18.04 and ROS Melodic is used as the UAV's companion computer. All the computer vision, guidance, and control algorithms execute on the *Nvidia Jetson Nano*, similar to how they would be deployed if the Nano was a companion computer in a physical drone. Energy profiling is performed directly on the *Nvidia Jetson Nano* using `powprofiler` as outlined in Section **??**.

Two groups of experiments are conducted. The first group evaluates the energy consumption and QoS of the *tracking* mode and the second group evaluates the energy consumption and QoS of the *landing* mode. For both groups, the experiments start with the tractor moving at a constant speed of $0.3\,\mathrm{m/s}$, according to a square path similar to that of a plowing tractor, and the UAV taking off and hovering at an altitude of $25\,\mathrm{m}$. After reaching the desired altitude, the UAV starts searching for the landing marker in the image frame. Once the landing marker is detected, the UAV commences its actions.

In *tracking* mode, the UAV will follow the moving platform at a fixed altitude and use its vertically facing camera to map the environment, while in *landing* mode the UAV will follow the moving platform and gradually lower its altitude until it lands on it. Both *tracking* and *landing* modes are tested under three different cases of no wind disturbances, wind disturbances of $8\,\mathrm{m/s}$ and wind disturbances of $12\,\mathrm{m/s}$ according to the wind model described in Section **??**.

C. Results

The first group of experiments was conducted to test the *tracking* mode and evaluate its energy efficiency and QoS. For the energy evaluation, eight tests were executed for different fps rates for the *YOLOv3-tiny* ROS node (4fps, 1fps, 0.5fps, 0.1fps) and the *landing marker* detection ROS node (10fps, 0.5fps) as seen in Figure **??**. For a 4fps update rate for *YOLOv3-tiny*, a power consumption of $6.30\,\mathrm{W}$ is observed while for a 1fps and 0.1fps update rates, the power consumption drops to $4.8\,\mathrm{W}$ and $3.9\,\mathrm{W}$ accordingly. By reducing the update rate for *landing marker* detection, from 10fps to 0.5fps, a further power saving of $0.15\,\mathrm{W}$- $0.19\,\mathrm{W}$ is achieved.

For the QoS evaluation, twelve tests were executed for different fps rates for the *YOLOv3-tiny* ROS node (4fps, 0.1fps) and for the *landing marker* detection ROS node (10fps, 0.5fps) under three different cases of wind disturbances. The QoS is determined as the number of correctly detected objects. An object is considered to be correctly detected if it is detected within a distance of $2\,\mathrm{m}$ from the its actual position and classified with the correct class. The detection results can be seen in Figure **??**. The best results were obtained for a high fps update rate in *YOLOv3-tiny* and *landing marker* detection, under no wind disturbances, since 28 out of the 32 objects were detected. Nevertheless, for the same high fps values but with a wind speed of $12\,\mathrm{m/s}$, only 18 out of 32 objects were correctly detected.

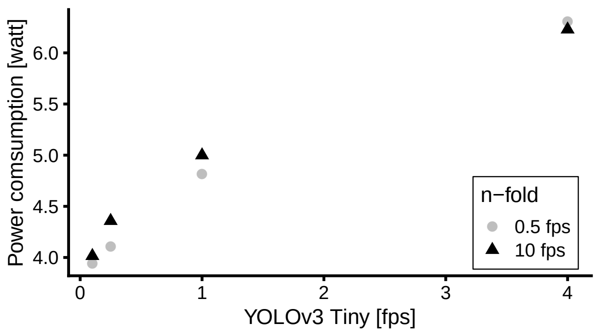

Fig. 4. Power consumption during tracking mode. YOLOv3 Tiny fps: 0=0.1fps, 1=0.5fps, 2=1fps, 3=4fps.

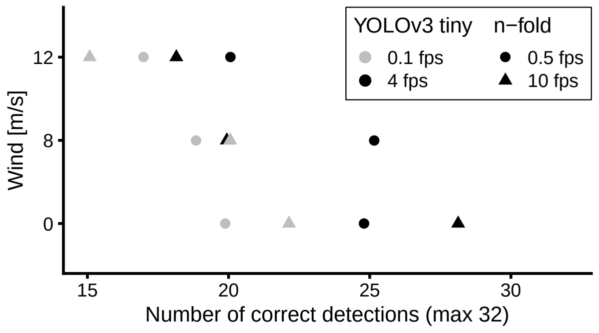

Fig. 5. Number of correctly detected objects under different conditions. Grey color denotes a 0.1fps and black color denotes a 4fps update rate in *YOLOv3-tiny* ROS node. Circles denote a 0.5fps and triangles denote a 10fps update rate in the *landing marker* detection ROS node.

The second group of experiments was conducted to test the *landing* mode and evaluate its energy efficiency and QoS. For the energy evaluation, eight tests were executed for different fps rates for the *landing marker* detection ROS node (10fps, 2fps, 1fps, 0.5fps) using two different cases. In the first case, the kernel size remains fixed at 22×22 pixels and in the second case an adaptive selection kernel size algorithm is used. The landing time is also taken under consideration as seen in Figure **??**. The largest difference in the power consumption is $0.14\,\mathrm{W}$ and is observed between

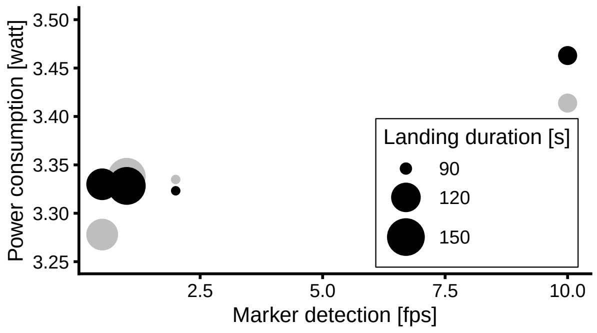

Fig. 6. Power consumption during landing mode. The black circles denote a fixed kernel size while the grey circles denote an adaptive kernel size.

the 0.5fps and the 10fps update rate for the *landing marker* detection. However the landing time is reduced by $30\,\mathrm{s}$ when using the 10fps update rate. The adaptive kernel size in most cases outperforms the fixed kernel size by $0.11\,\mathrm{W}$. Furthermore the adaptive kernel size can compensate for marker occlusions which will increase the overall robustness of the system.

For the QoS evaluation, twelve tests were executed for different fps rates for the *landing marker* detection ROS node (10fps, 2fps, 1fps, 0.5fps) under three different cases of wind disturbances. The QoS is determined as the mean squared error (MSE) between the predicted position of the moving platform, from the navigation block, and the moving platform's actual position. Four different altitude bins were used as seen in Figure **??**. A large MSE of around $3\,\mathrm{m}^2$ is observed for an altitude greater than $20\,\mathrm{m}$ for a 0.5fps rate, while an MSE close to zero is observed for an altitude of less than $5\,\mathrm{m}$ for an update rate of 10fps. Furthermore, wind disturbances do not seem to have an influence on the MSE. We believe the larger MSE for the *y* coordinates compared to the *x* coordinates is caused by a sudden change of the moving platform's direction on the *y* axis.

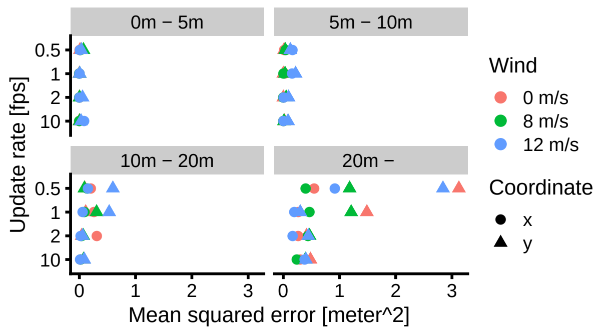

Fig. 7. Position error during landing and how it depends on the marker detection rate at different altitudes and wind disturbances. Circles denote errors in the x direction and triangles errors in the y direction.

D. Discussion

The experiments show that both tracking mode and landing mode are supported by the system, in a simulated environment with a moving platform and random wind conditions. Moreover, the performance of both modes is sensitive to the QoS, with a high success rate of both modes at high QoS levels, and significantly lower performance at lower QoS levels.

The potential energy savings from having an energy-sensitive algorithm that can adapt its QoS by changing the fps values for the *YOLOv3-tiny* and *landing marker* ROS nodes should be seen in relation to the total energy consumption of the UAV. As a concrete example, consider a DJI Phantom 4 multirotor and a Sky-Watch Cumulus fixed-wing (the fixed-wing would need to circle while tracking and would need VTOL capabilities to land, but we nevertheless include it for comparison). We estimate[1] that the Phantom uses roughly $140\,\mathrm{W}$ while cruising whereas the Cumulus uses roughly $40\,\mathrm{W}$ while cruising. The maximal saving gained from changing the *YOLOv3-tiny* rate is $6.30\,\mathrm{W} - 3.9\,\mathrm{W} = 2.4\,\mathrm{W}$ whereas the maximal

[1]From information on the respective product pages regarding battery capacity and maximal flight time.

saving gained from changing the *landing marker* rate is $0.2\,\mathrm{W}$. For the Cumulus, there is thus a 6.5% potential energy saving, whereas the potential energy saving only is 1.9% for the Phantom. For the Cumulus this saving is considered large enough to significantly impact the flying time of the drone, with a total energy saving of $23.4\,\mathrm{kJ}$. For the Cumulus, the potential saving from adapting the *landing marker* QoS is however only 0.5%.

For the tracking mode, changing the *landing marker* rate provided a minor saving of $0.14\,\mathrm{W}$, but at the cost of increased landing time. Therefore, although the higher-QoS computer vision algorithm is marginally more expensive by $0.14\,\mathrm{W}$, the UAV will overall save energy because of a reduced flight time.

VI. Conclusion and Future Work

In this paper we presented a robust, energy-sensitive, vision-based algorithm for autonomous tracking and landing in varying environmental conditions, by experimentally executing all the necessary algorithms on the *Nvidia Jetson Nano* companion computer. Our experiments show that the proposed computer vision algorithms for detecting the moving platform can be run at the highest QoS level with only a marginal energy overhead, whereas adapting the QoS level of *YOLOv3-tiny* CNN results in a considerable power saving for the system as a whole. This power saving is significant if the system was executing on a fixed-wing UAV, but only marginal if executing on a multirotor UAV.

In terms of future work, we are interested in automatically adapting the QoS level to the available battery, and in testing this approach on a physical drone.

Acknowledgment

This work is supported and partly funded by the European Union's Horizon2020 research and innovation program under grant agreement No. 779882 (TeamPlay).