SAURUSS

Smart
Autonomous
UAV
Recognizer for
Universal
Surveillance
System



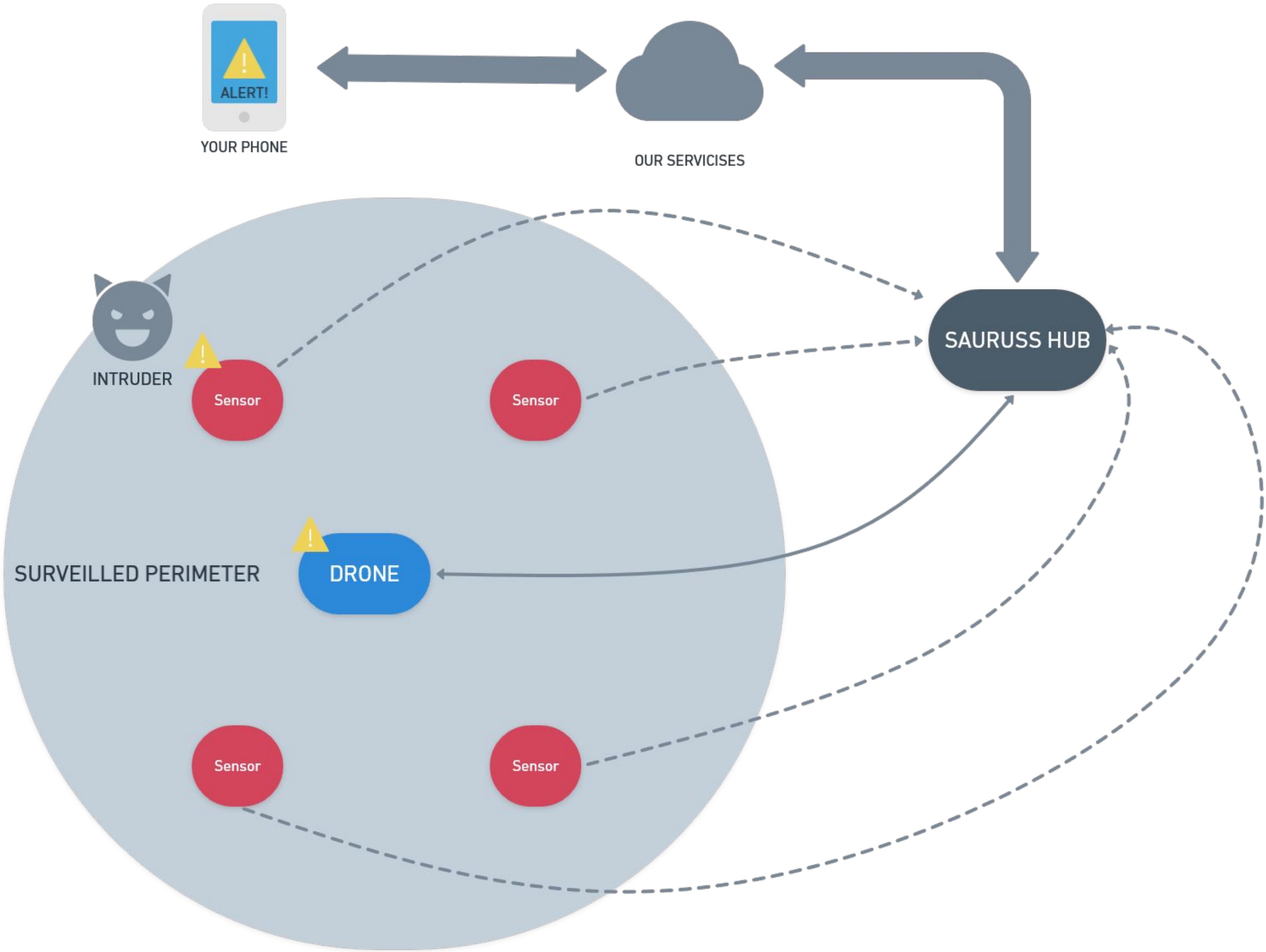
Our System

SAURUSS consist of:

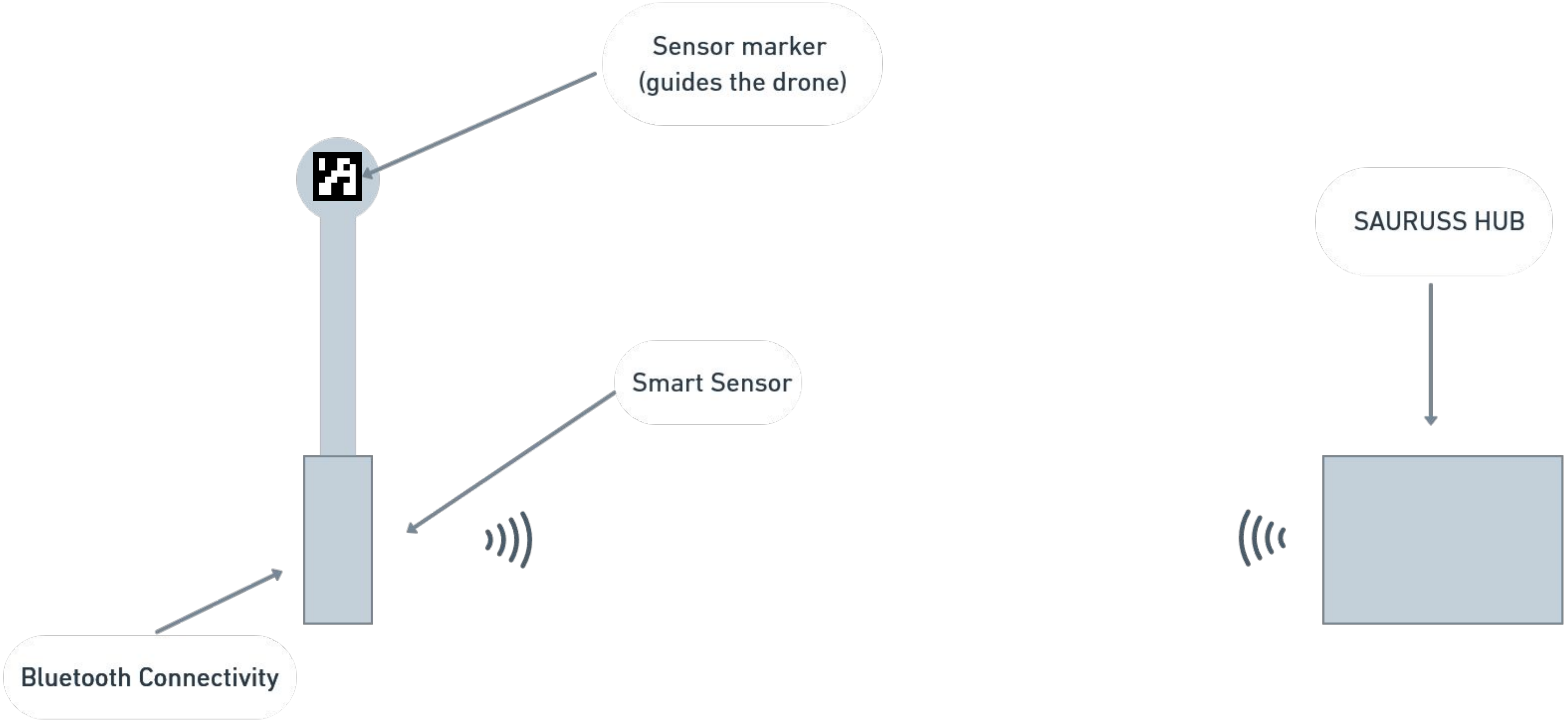
- our bridge that does all the heavy lifting for image processing and person detection
- our drone, equipped with a camera, connected wirelessly to the bridge
- our sensors, with bluetooth connection to our bridge

How it Works?

- If our sensors sense an intruder inside the surveilled perimeter, our drone will fly towards it.
- Our smart hub will tell if there is someone and notify you, through our app, with a pic and video proof! 📸



OUR SENSORS AND SAURUSS HUB

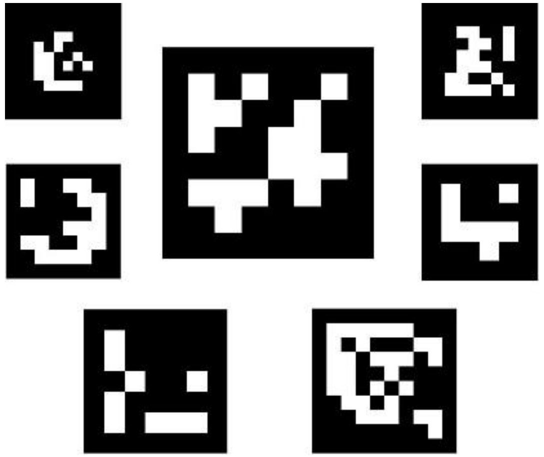


HW & SW used for our 3D System 🔎

OpenCV

RaspberryPi

ArUco library



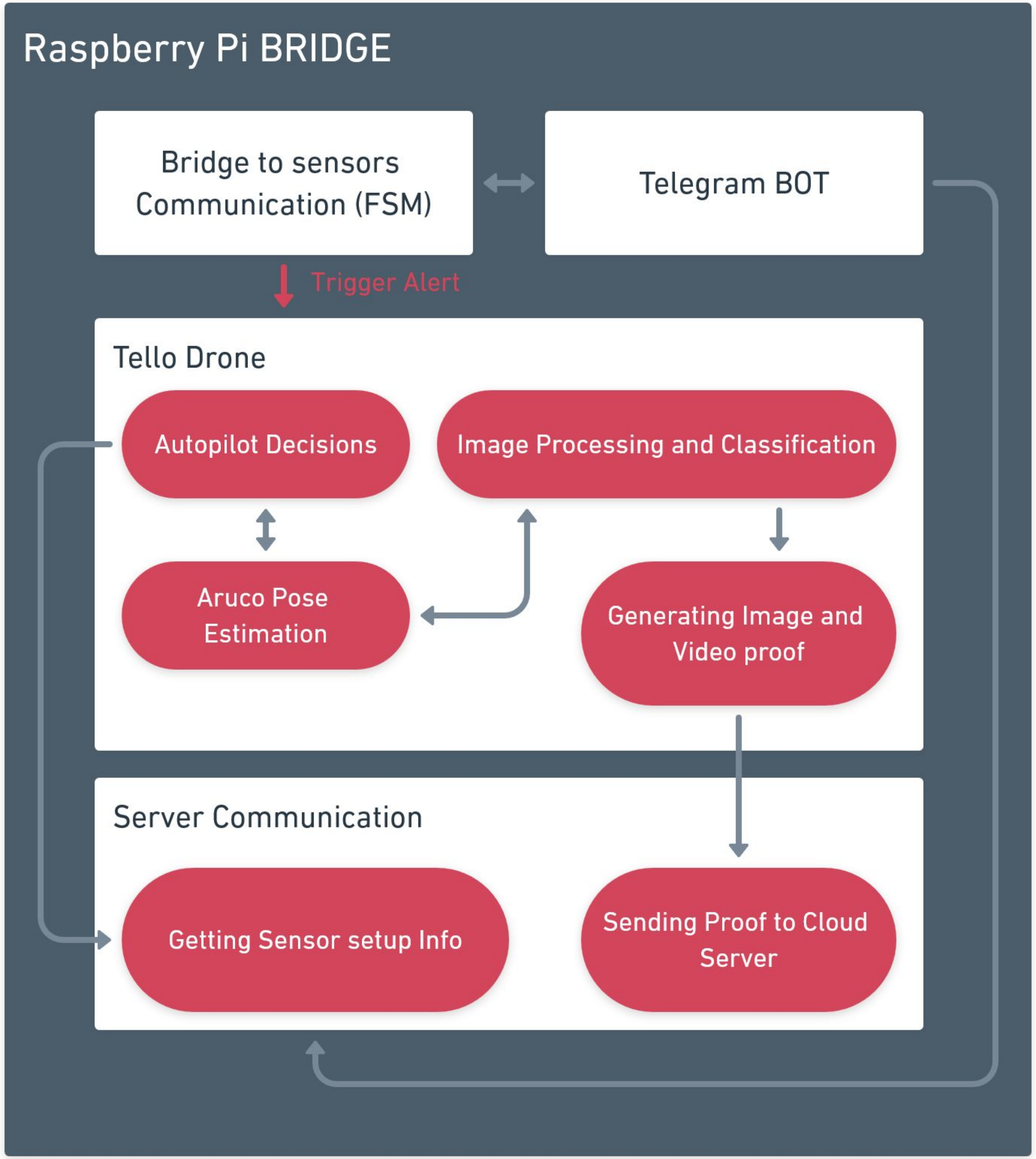
YOLO

Raspberry Pi bridge

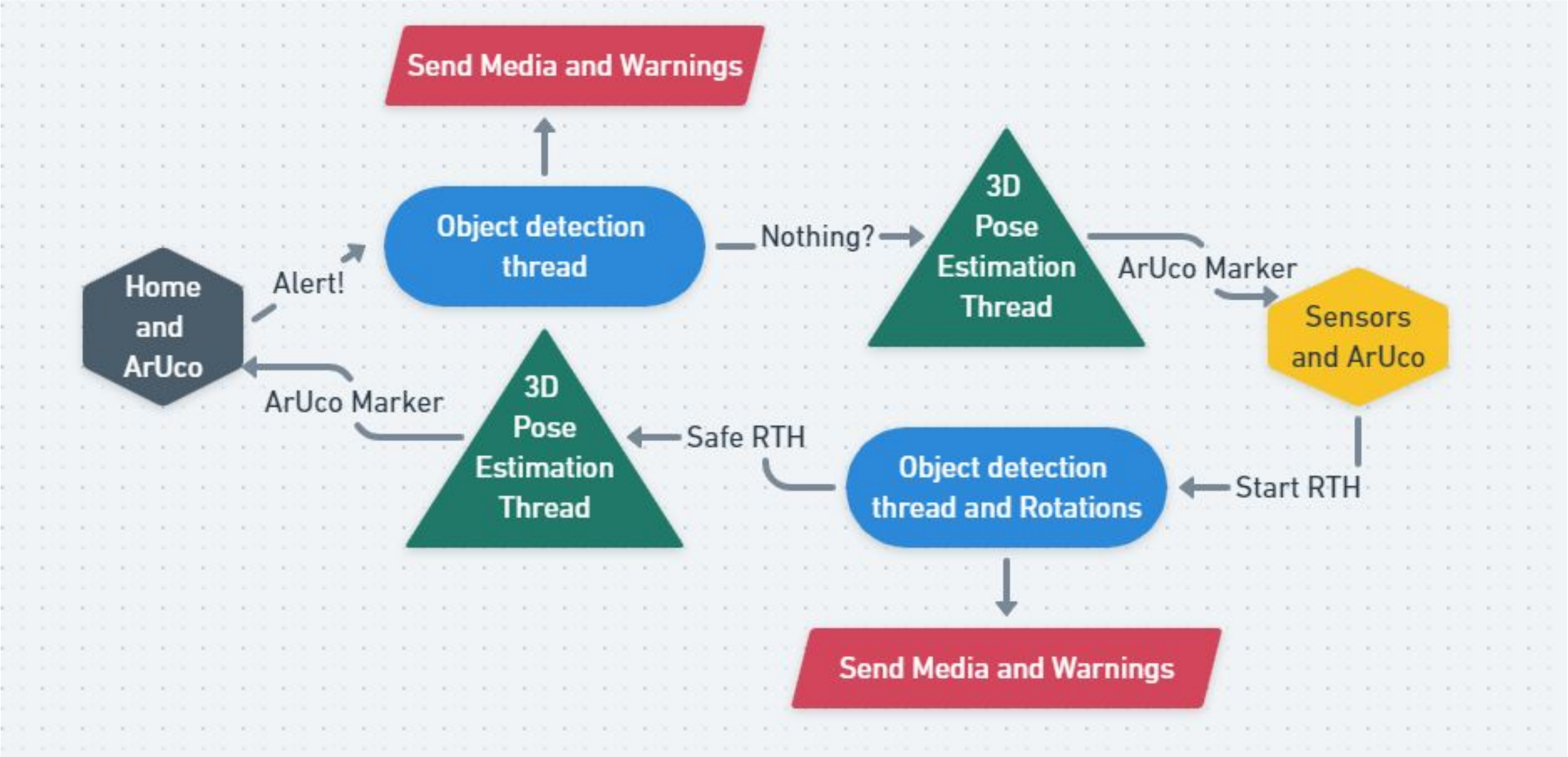
- Bridge is the ***center of the drone's decisions*** and also of the frame elaboration done by the camera

- Visual and Object recognition activities are based on *AI algorithms* (OpenCV and YOLO)

- When the drone sees a human, an alarm is set off by the bridge



Graphic description



Drone Autonomous Driving & use of ArUco markers ⬜

How to help the drone go where it needs to?

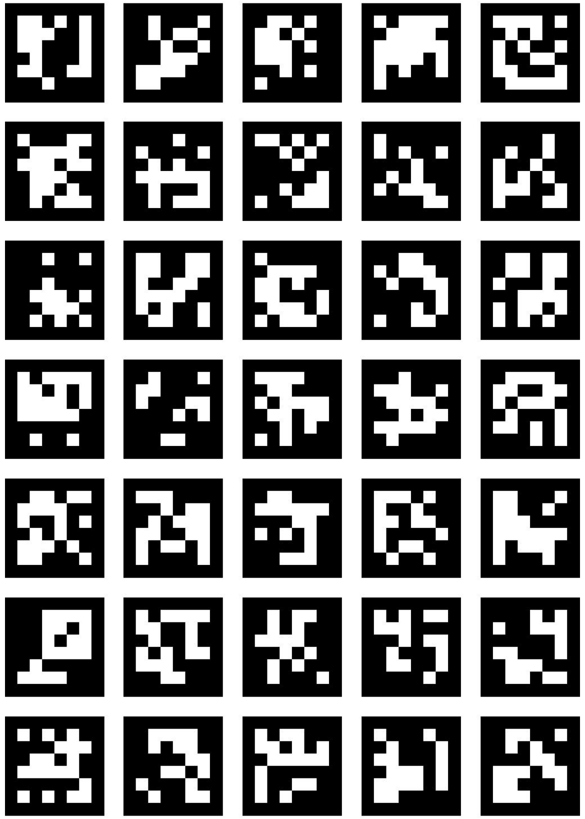
PROBLEMS

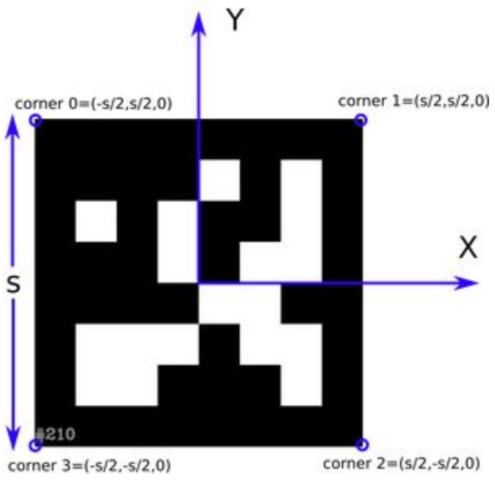
1. No GPS in our drone
2. Low accuracy in move commands
3. Low accuracy for RTH (Return to Home)

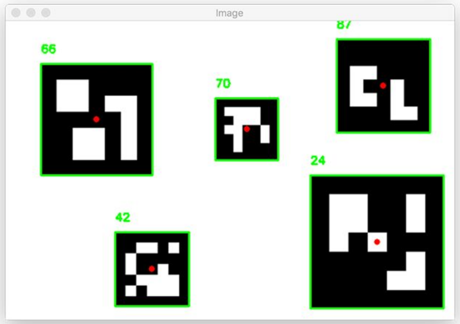
SOLUTION

- Use of **ArUCO markers** for improving drones self-driving algorithm!

What are ArUCO Markers?







ArUco is a popular library that **uses binary square fiducial markers for pose estimation** in computer vision applications

- A single marker provides *4 corners to achieve camera pose*
- *Robust Internal binary coding*, for possible implementations of error detection techniques.
- Once a dictionary is chosen, *each ArUco marker has its own ID*

Our Algorithm - Pseudo Code 1

```
corners, ids, rejected = aruco.detectMarkers(image, dictionary, parameters,
cameraMatrix, distCoeff)
```

- `cameraMatrix` and `distCoeff` are used because our drone camera is not ideal (more on this later) and has intrinsic parameters useful for image processing
- The functions return an array of array with the identified ArUco markers **corners** in the scene

```
ret = aruco.estimatePoseSingleMarkers(corners, marker_size,
parameters,cameraMatrix, distCoeff)
```

- The functions return an array of array (**ret**) where, for each ArUco marker found, there is a rotation vector **rvec** and translation vector **tvec** inside (more on this later)
- **rvec** and **tvec** have 3 components each (one for each axis in the three dimensional space)

Our Algorithm - Pseudo Code 2

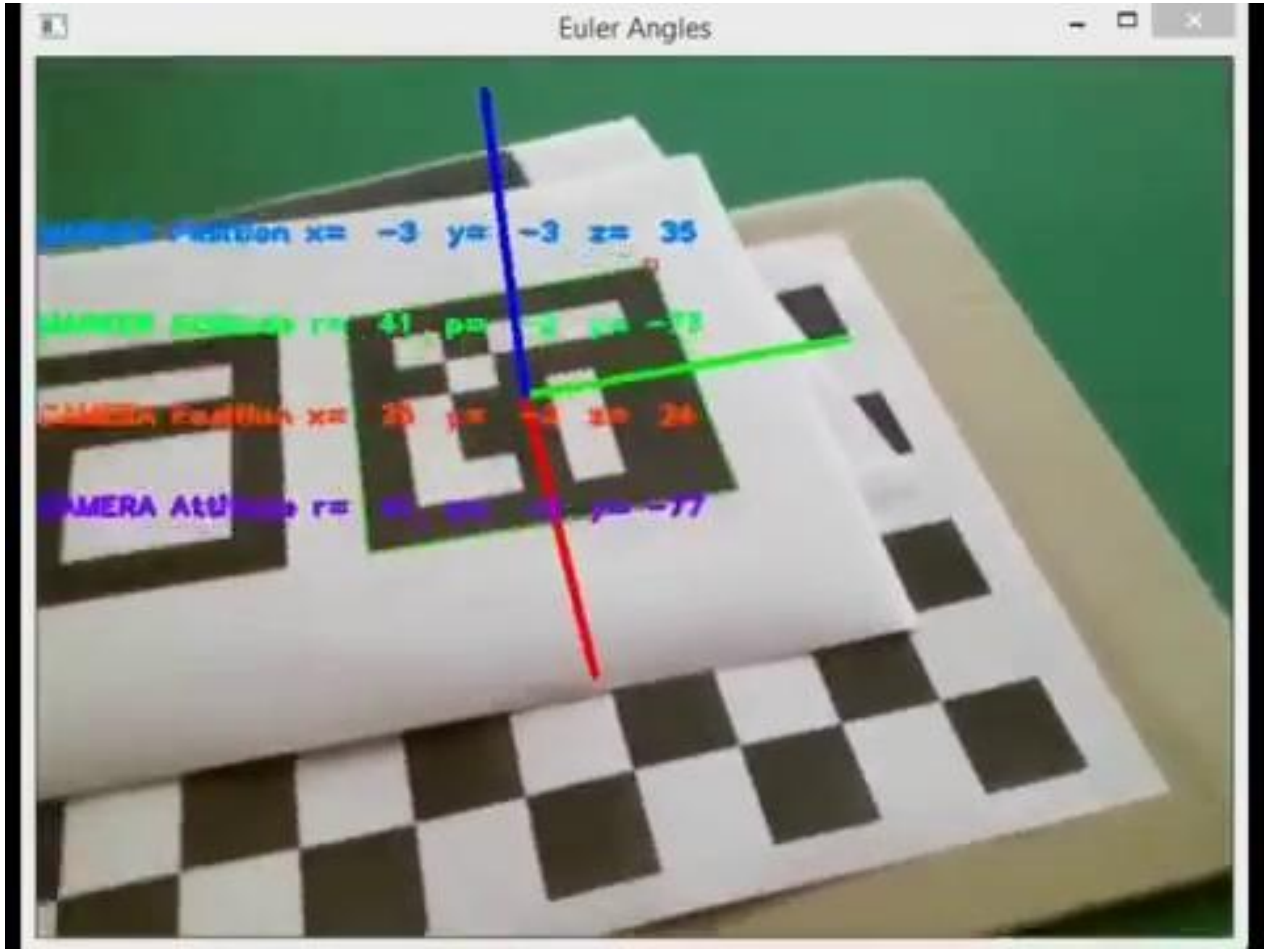
```
R_ct = np.matrix(cv2.Rodrigues(rvec))
roll_marker, pitch_marker, yaw_marker = rotationMatrixtoEulerAngles(R_ct)
```

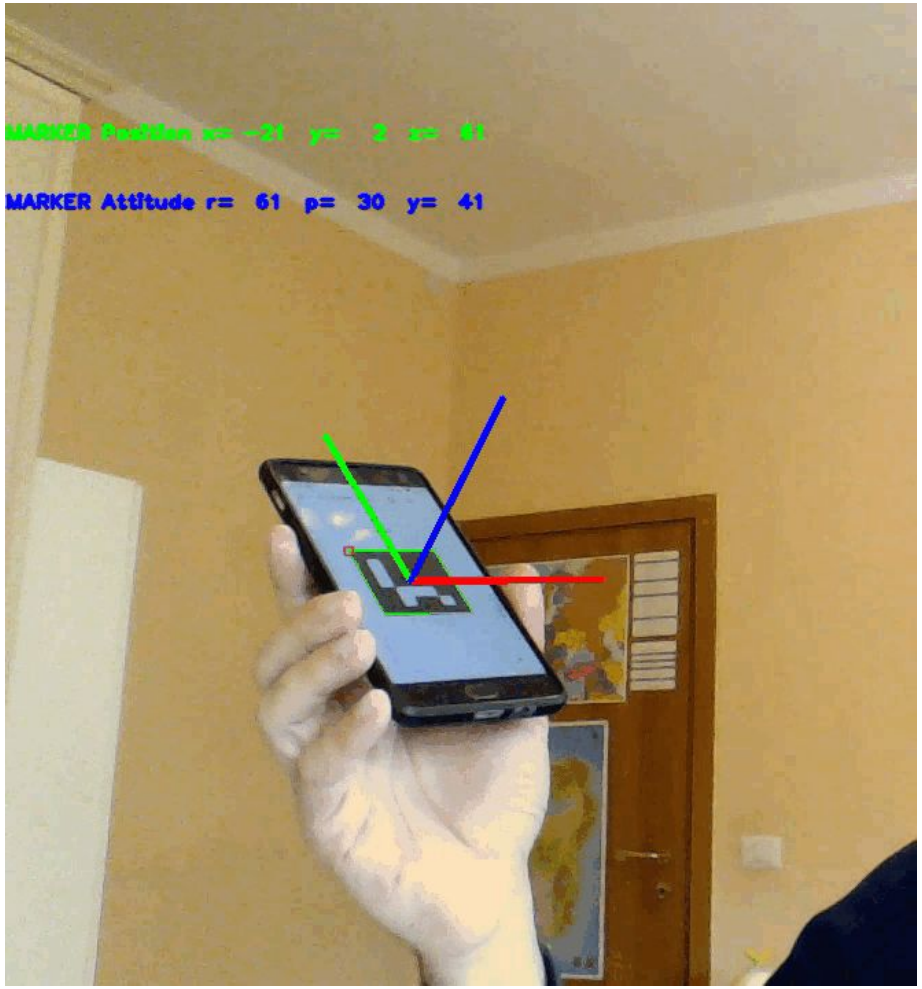
- With the first function we obtained the Rotation matrix (`R_ct`)of the ArUco tag with respect to the camera
- In the second function we obtained the **roll**, **pitch** and **yaw** of the markers with respect to the camera →<u>We now can compute useful informations for our self-driving algorithm!</u>

For our self-driving drone we only need:

- **Distance along z-axis (tvec[0])**
- **Marker yaw and pitch respect to camera**
- **Center position of the marker C = (X,Y) (we use the corners pixel position)**

ArUco marker pose estimation





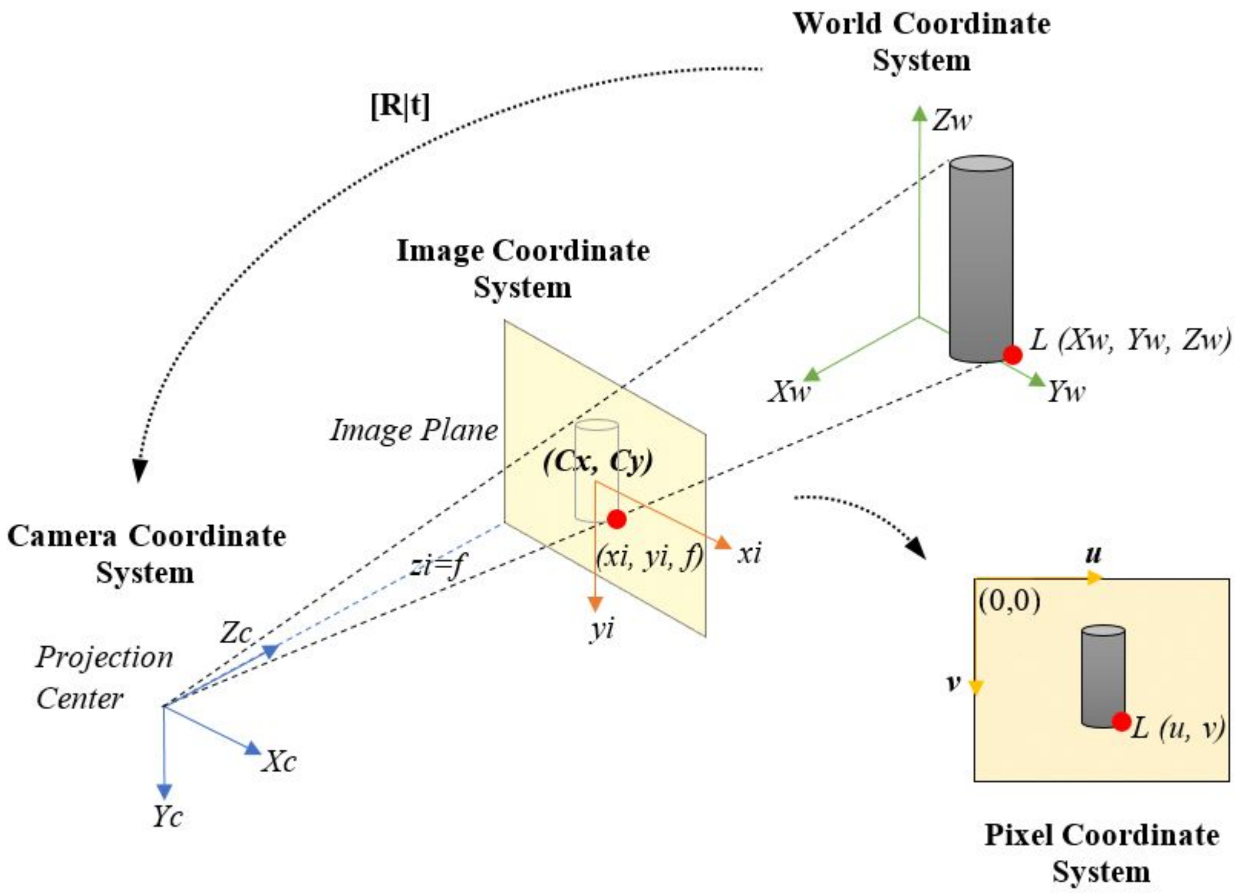
A look under the hood 🧐

understanding
pose estimation...

...and it's challenges

Pinhole camera model

The **pinhole camera model** describes the mathematical relationship between the coordinates of a point in three-dimensional space and its projection onto the image plane of an ideal pinhole camera, where the aperture of the camera is described as a point and no lenses are used to focus the light.



Distortion problem

The ideal pinhole camera model does not include, for example, geometric distortions or blurring of objects caused by lenses and apertures of finite size.

Solution: Camera Calibration

Calibration is necessary in order to derive the **intrinsic parameters of the camera**, such as:

- **Distortion of the image**

$$Distortion\ coefficients = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

- **Focal length**
- **Optical center**

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Harris Corner Detector

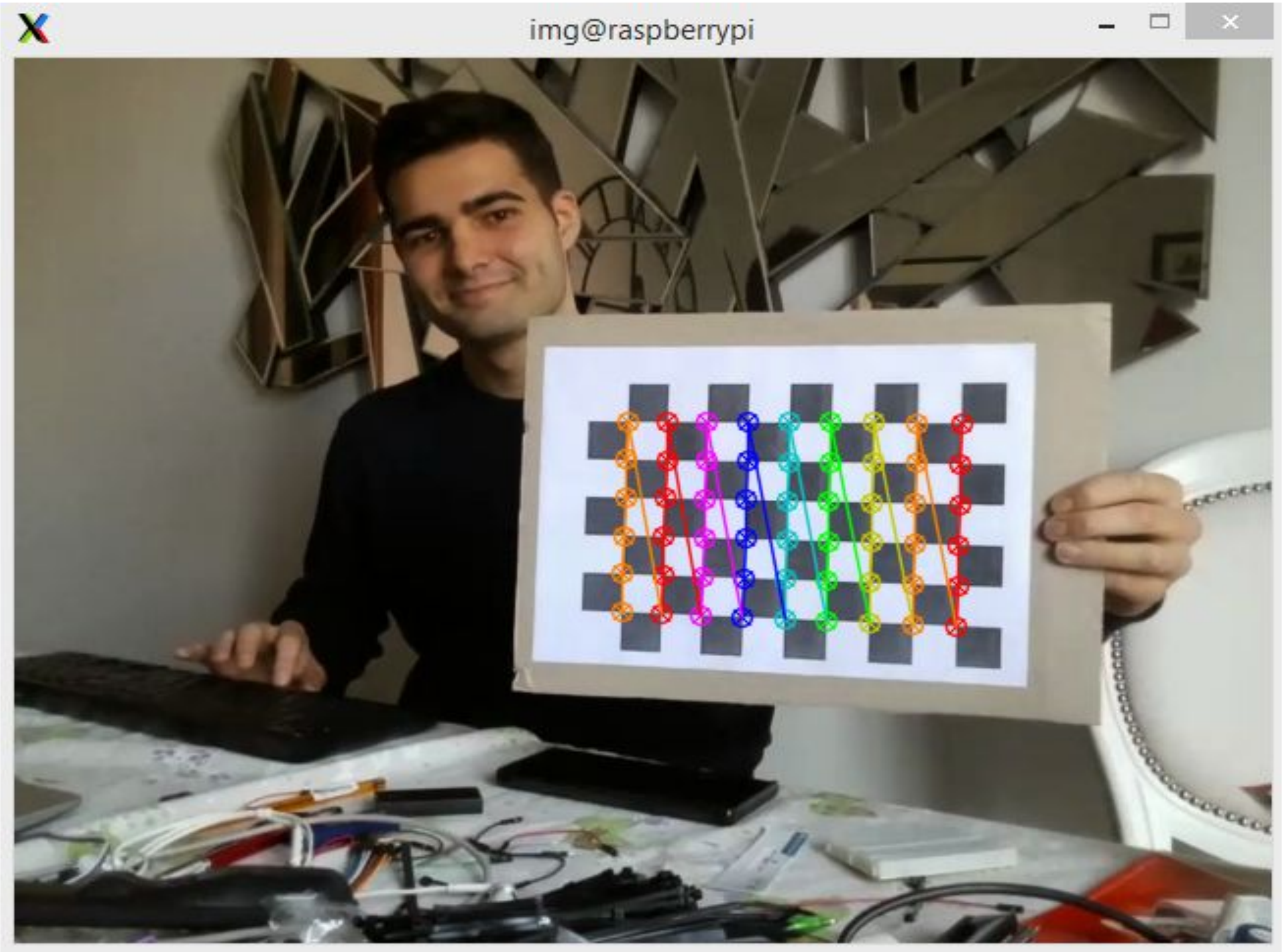
Thanks to Harris corner detection algorithm it’s possible to get the intersection of an angle in an image “I” in a window “W”

$$f(\Delta x, \Delta y) = \sum_{(x_k, y_k) \in W} \left(I(x_k, y_k) - I(x_k + \Delta x, y_k + \Delta y) \right)^2$$

Where (dx, dy) is the shift of the point (x,y)

Chessboard calibration

The calibration of the drone camera was done using a 9x6 **chessboard** (number of intersections between chess).

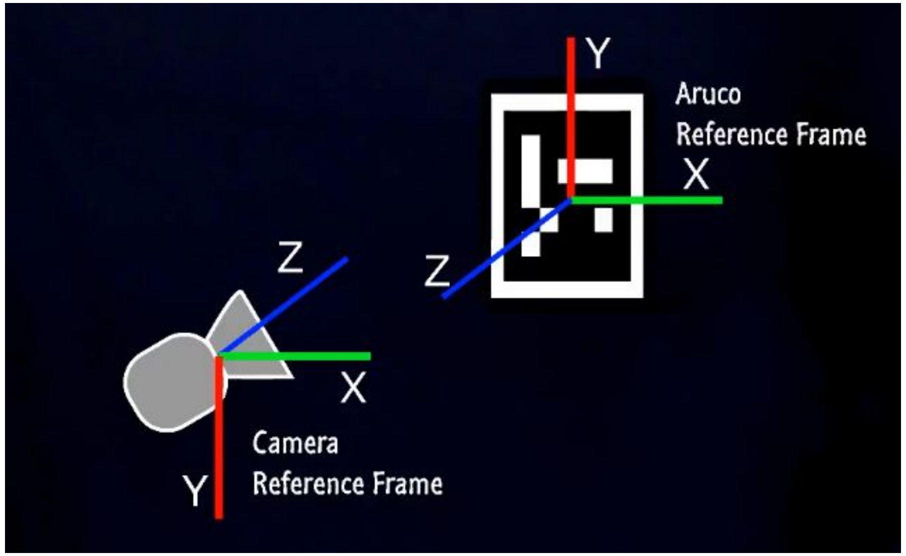


Camera and Marker Pose estimation problem

PROBLEM

- **We have two reference systems**: one for the drone camera, one for the ArUco marker

- **We need to get the pose of the drone camera with respect to the marker**



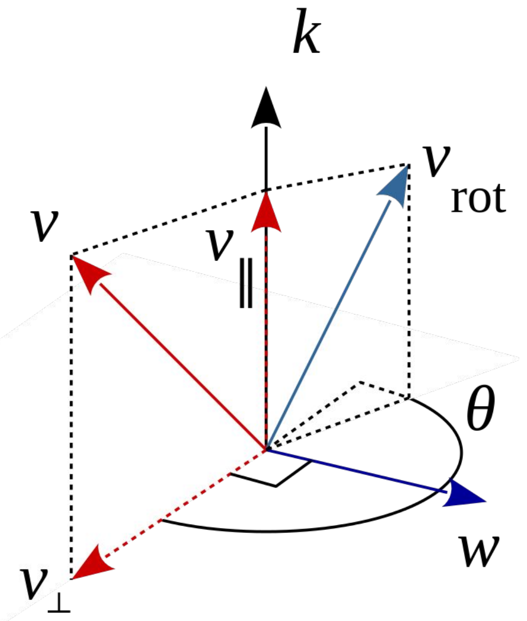
SOLUTION

- We do that with the **estimatePoseSingleMarkers** OpenCV function and we obtain:
  - the **rotation vector (rvec)** - rotation of marker respect to the camera (Rodrigues notation)
  - the **translation vector (tvec)** - position of marker respect to the camera

Calculating the Camera-to-Marker Rotation Matrix

But we actually need the **Rotation Matrix** **R_ct** from marker to camera for computing the **yaw**, **pitch** and **roll** of the marker to camera and vice versa

Thankfully, OpenCV helps us with the function `cv2.Rodrigues(rvec)` that returns the **R_ct** matrix (more info are available in the OpenCV online documentation here)



rvec[0] = rx

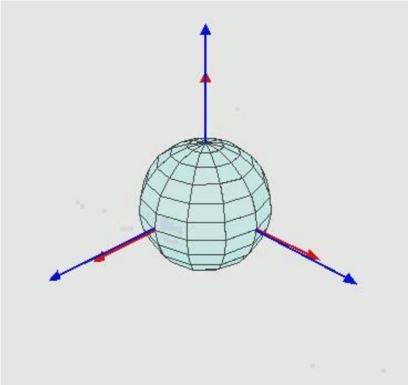
rvec[1] = ry

rvec[2] = rz

$$R = \begin{bmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{bmatrix}$$

Calculating Roll - Pitch - Yaw from Rotation Matrix

$$R = R_z(\alpha)\,R_y(\beta)\,R_x(\gamma) = \overset{\text{roll}}{\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}} \overset{\text{yaw}}{\begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}} \overset{\text{pitch}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}}$$



Rotation Matrix to Euler Angles formula

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

$$\mathbf{R} = \mathbf{R_z R_y R_x}$$

The 3 Euler angles are

$$\theta_x = atan2(r_{32}, r_{33})$$

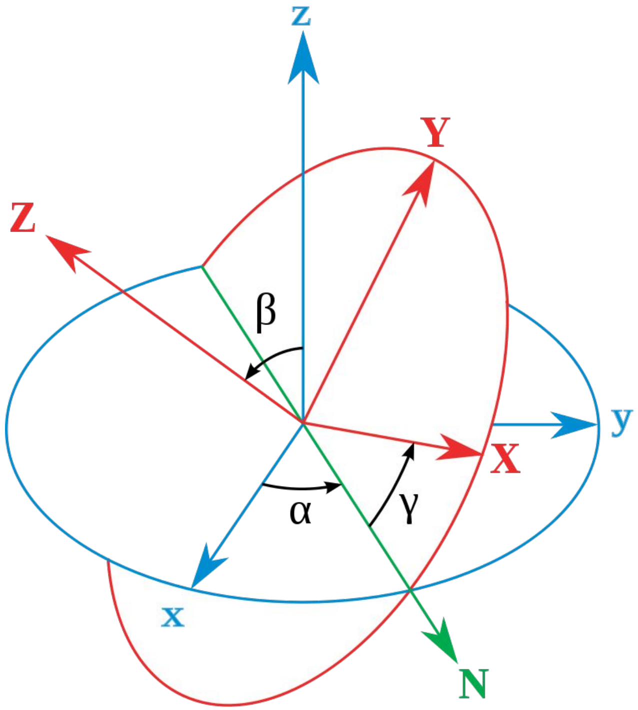
$$\theta_y = atan2\left(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}\right)$$

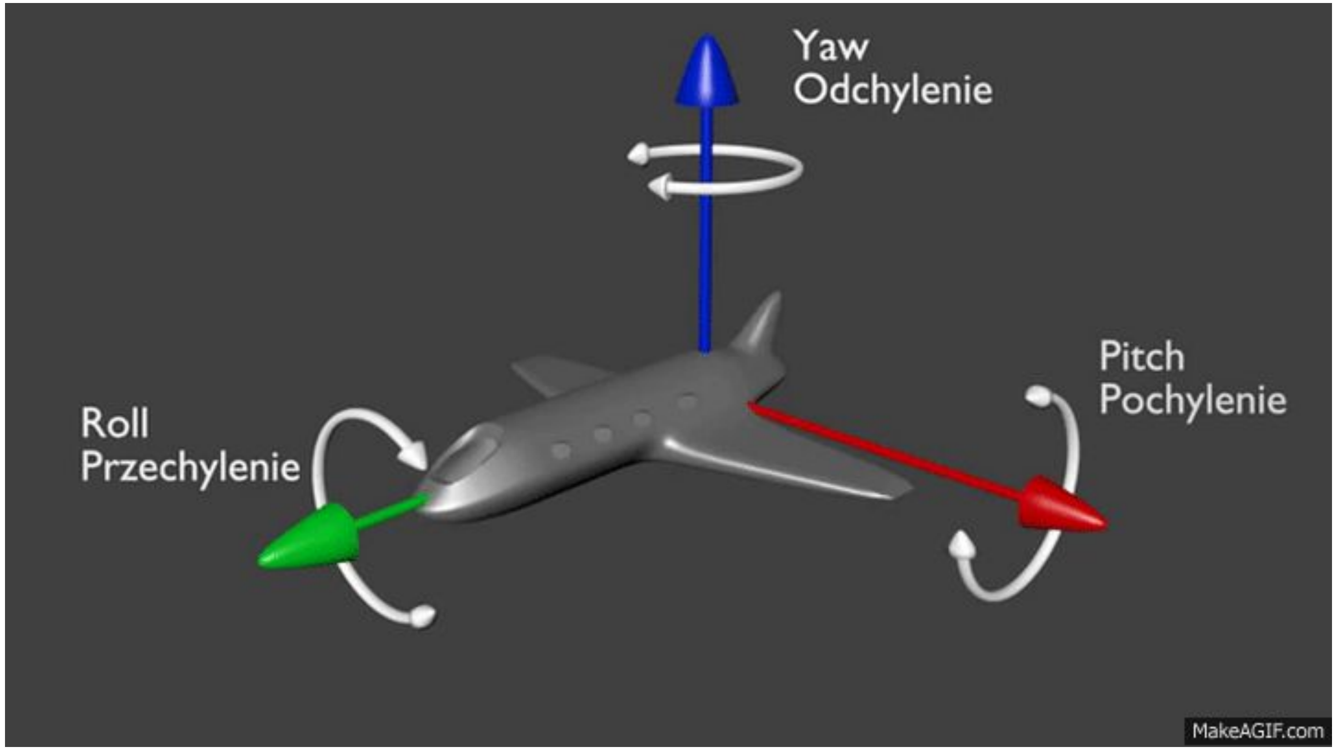
$$\theta_z = atan2(r_{21}, r_{11})$$

$\theta_X \rightarrow$ Roll

$\theta_Y \rightarrow$ Pitch

$\theta_Z \rightarrow$ Yaw





Roll Pitch Yaw

pitcher **d**o**o**r

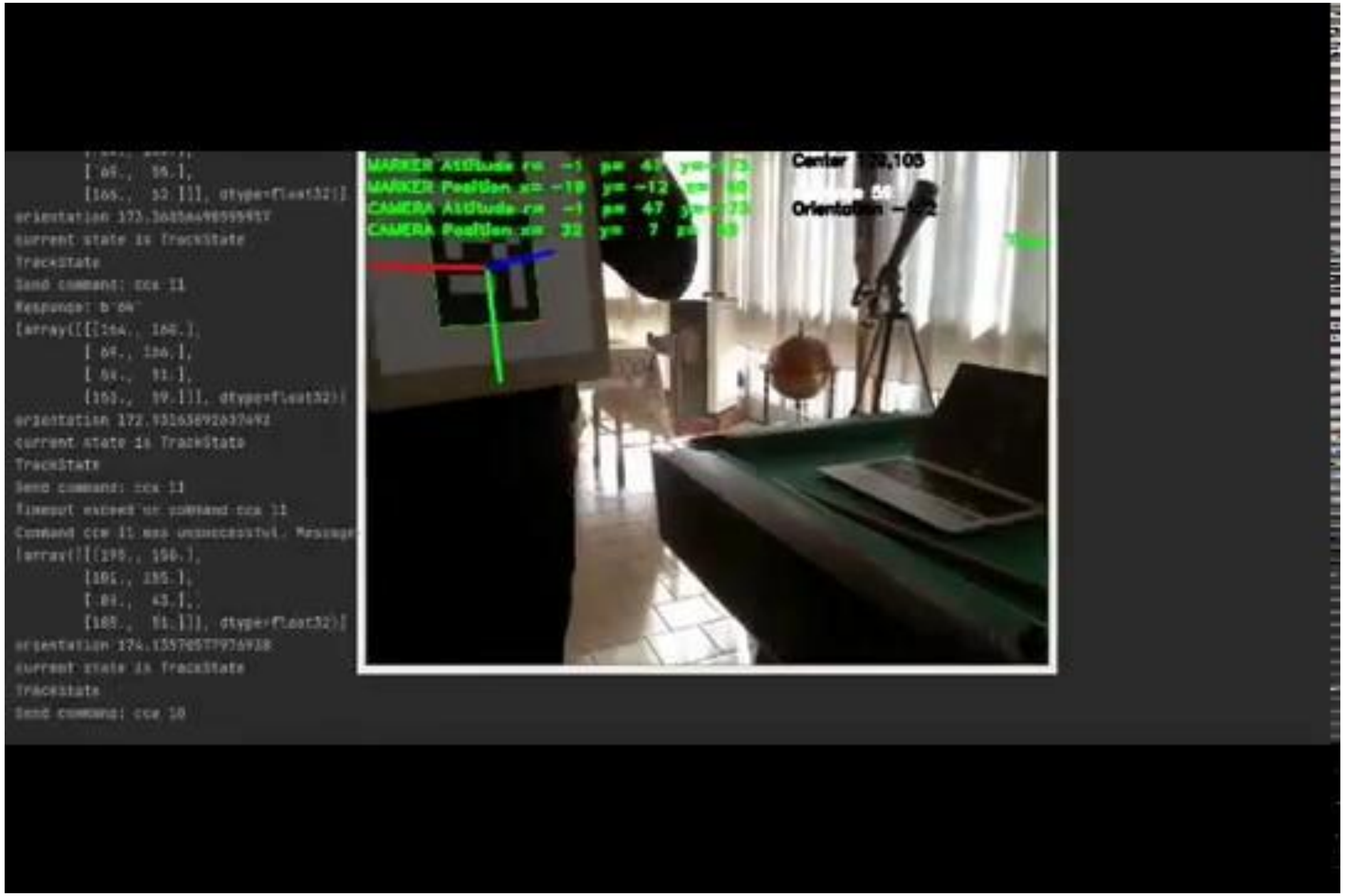
Telling the drone what to do: Tello SDK 2.0

Once we obtained the roll, pitch, yaw and the distance between the drone camera and the marker, we need to interact with the drone with the following elementary functions:

- **tello.move(dir, dist)**
- **tello.go(x,y,z,speed)**
- **tello.rotate_counter/clockwise(theta)**
- **tello.takeoff()**
- **tello.land()**

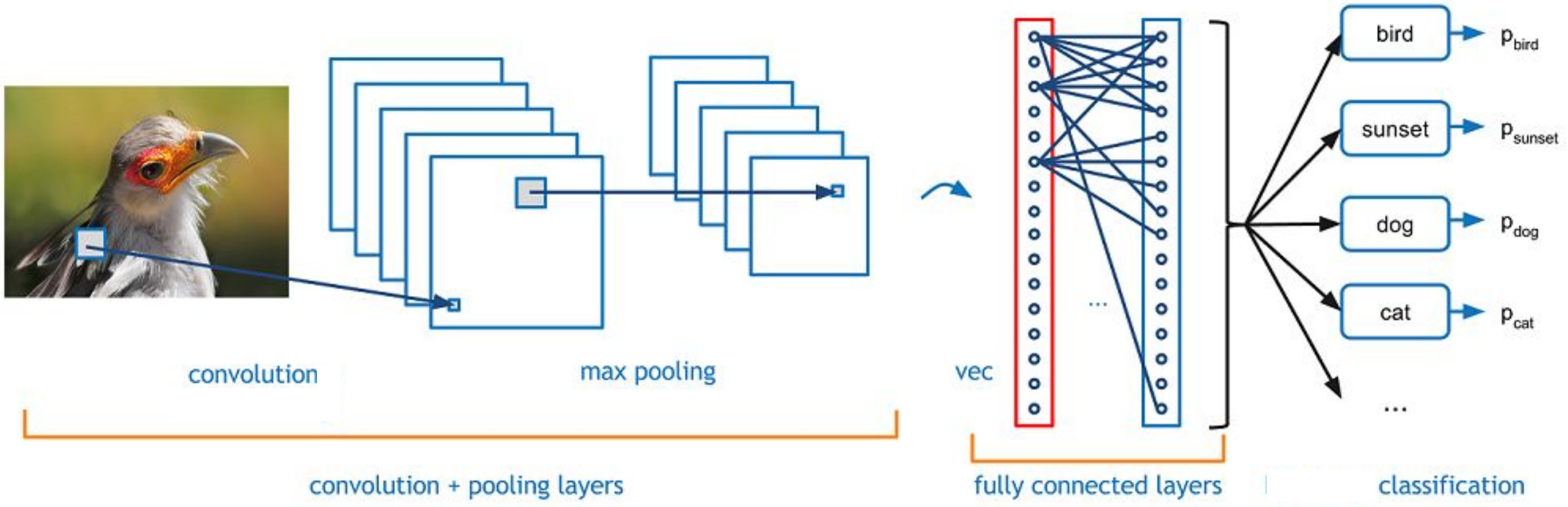
with tello.get_time_of_flight() you can estimate the distance traveled. but it's not accurate for environmental agents.

Tello drone pose estimation and tracking



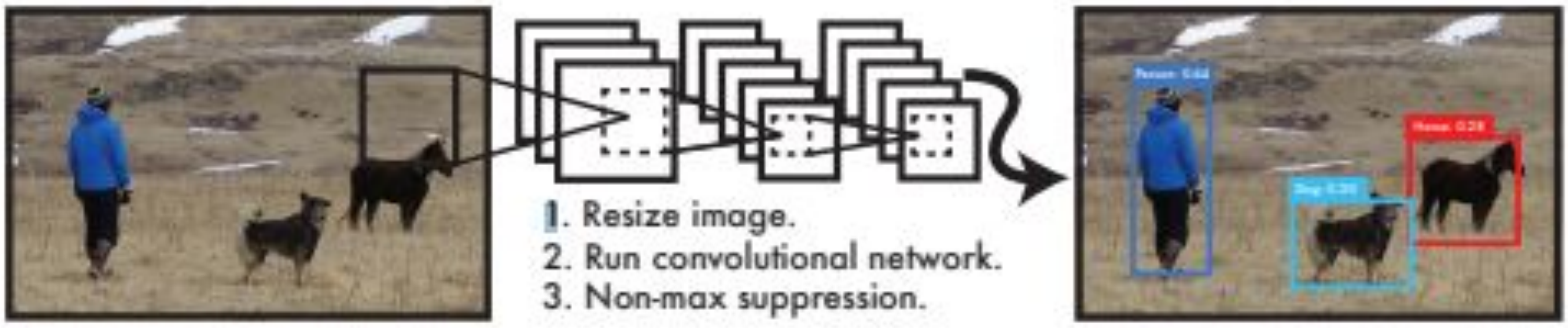
Drone Object Recognition & YOLO 👀

What a CNN is



convolution
max pooling
vec
...
bird
sunset
dog
cat
...
p_{bird}
p_{sunset}
p_{dog}
p_{cat}
convolution + pooling layers
fully connected layers
classification

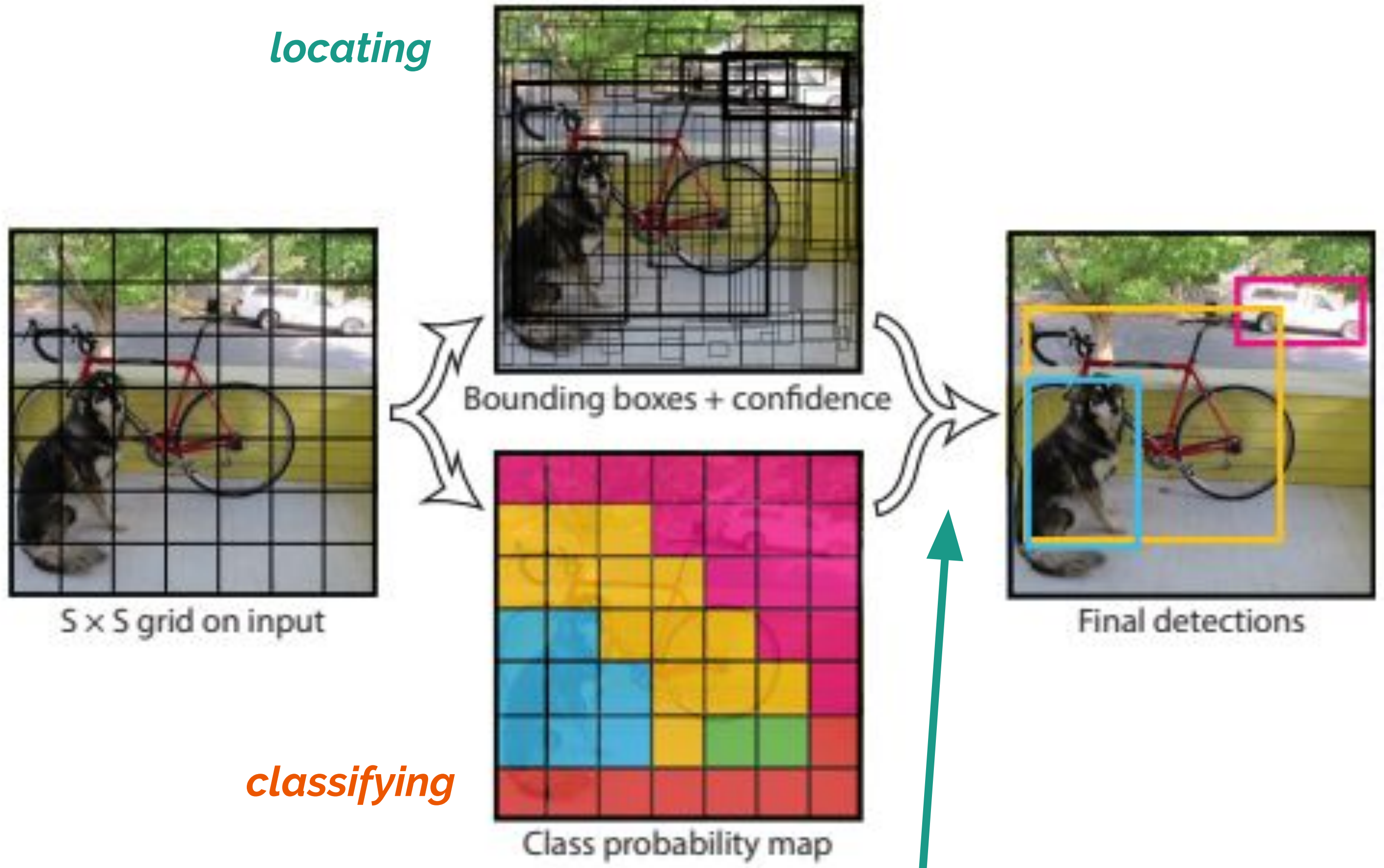
YOLO: Real-Time Object_Detection

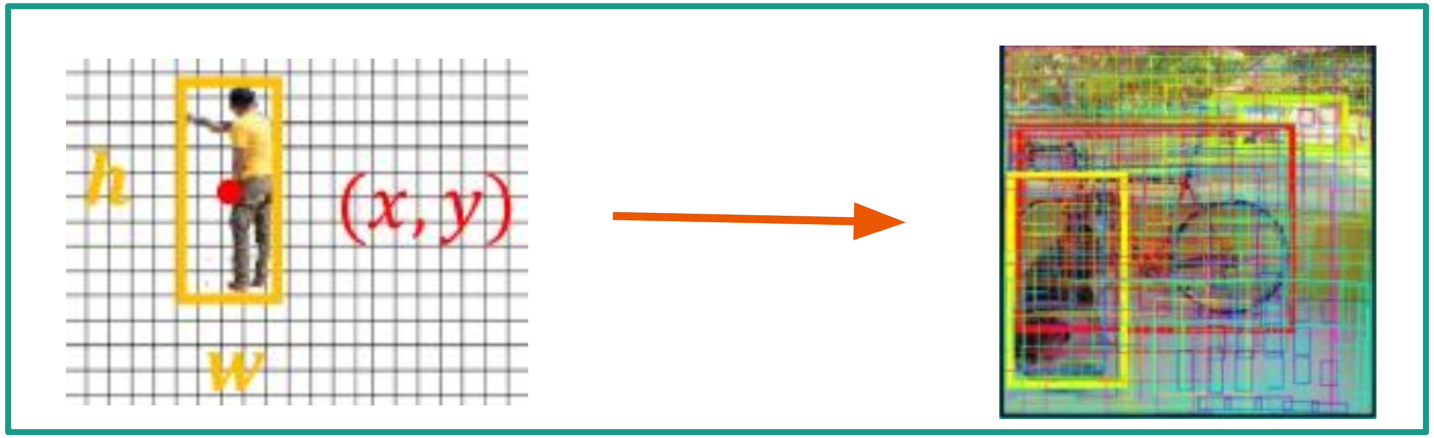


- base network runs at **45 frames** per second (up to 155)
- pretrained weights on **COCO** dataset
- Weights: "yolov3-**tiny.weight**"

  It was decided to use the TINY version of the weights and the configuration in order to cope with the bridge calculation limits

- contains class “**person**” and it’s easy to use





- Network uses features from the entire image to predict each bounding box
- Image splitted into an SxS grid
- **Pr(Classi|Object) ∗ Pr(Object)** class-specific confidence scores for each box
- Non-Max Suppression

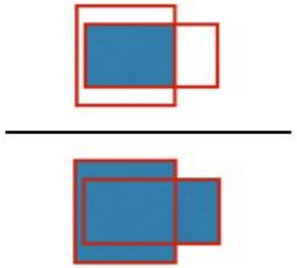
Non max-suppression

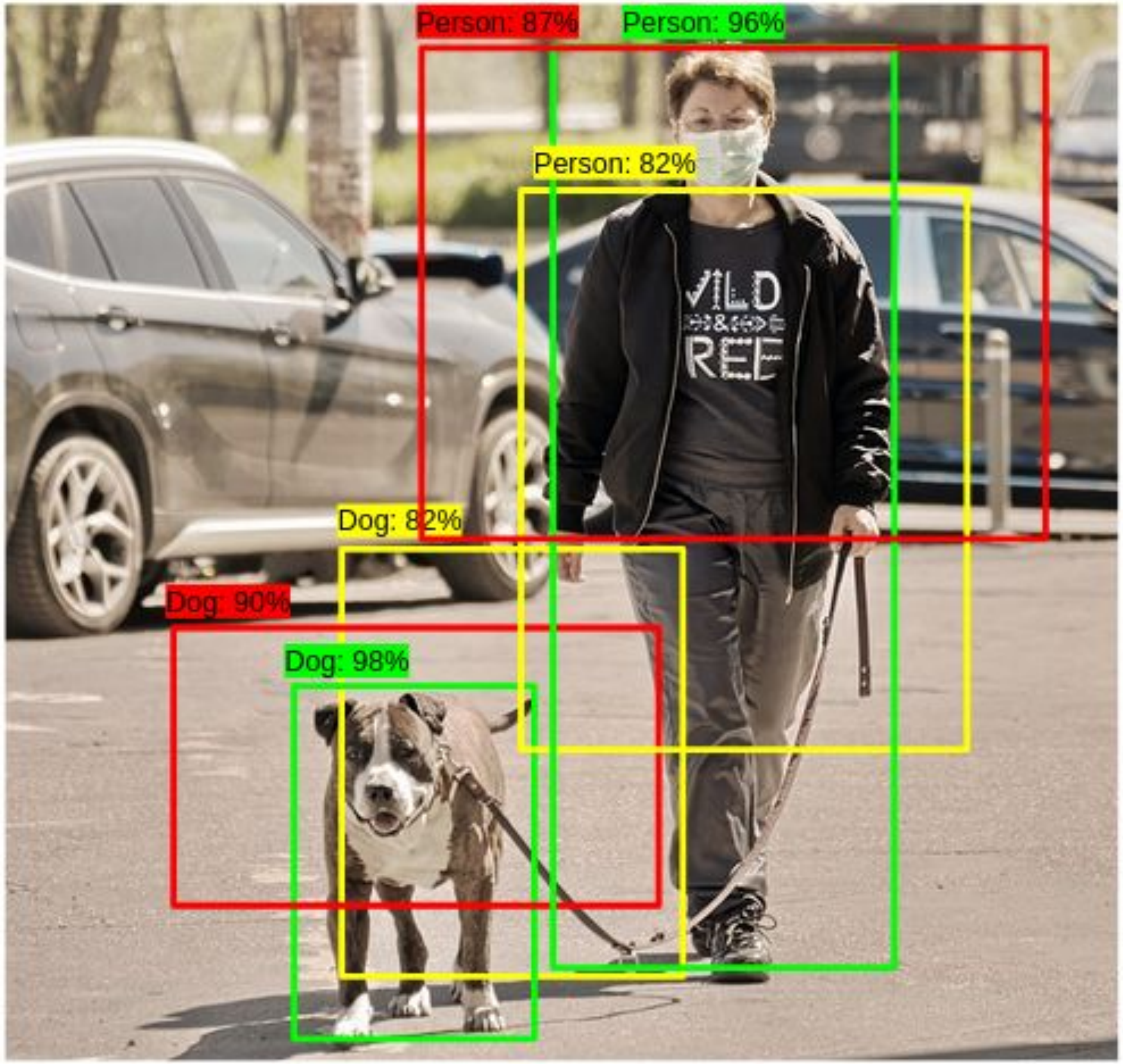
1. select the bounding box with the highest objectiveness score

2. remove all the other boxes with high *overlap*

Intersection over Union

$$IoU = \frac{B_1 \cap B_2}{B_1 \cup B_2} =$$

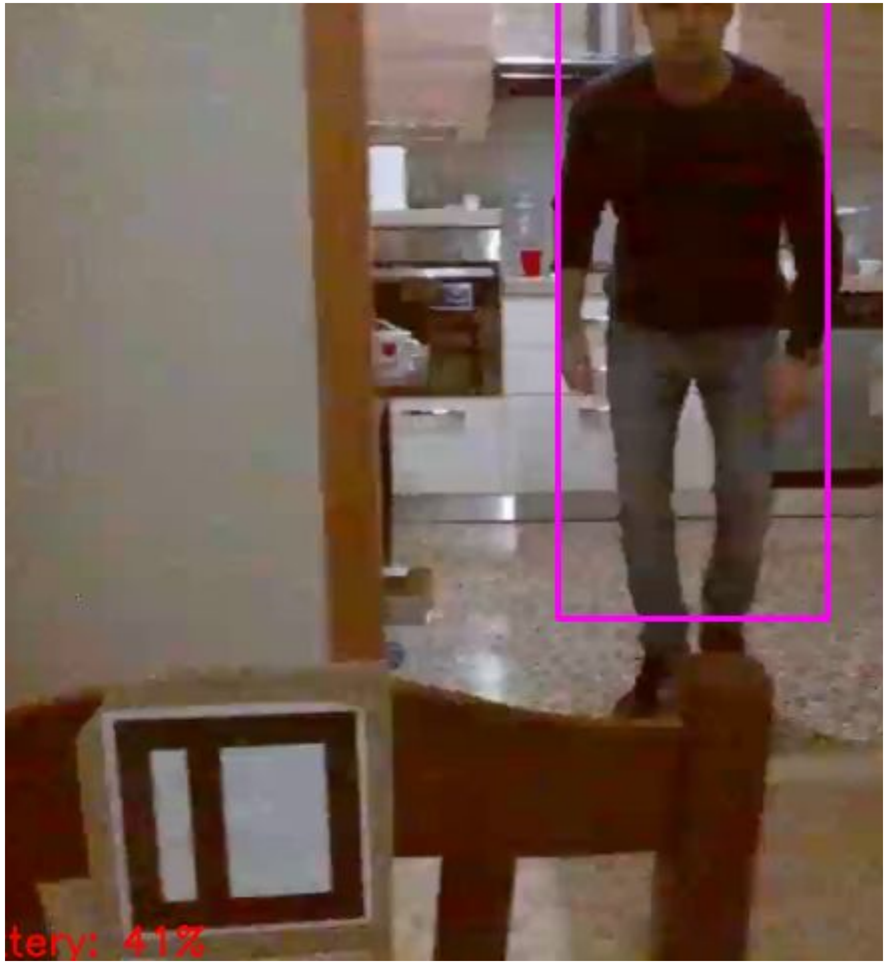




Our Algorithm - YOLO Pseudo Code

```
findObject(boxes,labels,threshold)

load image
create blob from image
net.setInput(blob)
outputs = net.forward()

boxes , labels, scores = array(), array(),array()

iterate over number of boxes

    classID = argmax(scores)
    confidence = scores[classID]

take only boxes with
confidence > NMS_threshold

#in our specific case
iterate over all possible classes & boxes
if class associated with box  == “person”

!!send alarm!!
```

YOLO examples



DEMO VIDEO