System Design Project
2015-2016

Velosso Technical Guide

Paul Sinclair : s1337523
Dovile Vitonyte : s1237357
Karen McCulloch : s1340952
Krassy Gochev : s1346707
Tomislav Jovanovic : s1341139
James Friel : s1332298

Mentor
Nantas Nardelli

Contents

1 System Architecture

The system can be roughly split into 4 separate sections:

- Hardware: the LEGO robot structure, sensors and motors, and the Arduino controlling and monitoring all these
- Vision: the software which takes the raw vision feed and retrieves information about each object
- Strategy: the software which decides which commands to send to the robot based on information from the vision system
- Communications: the software which sends and monitors commands to the robot from the PC and monitors commands sent to the PC from the robot

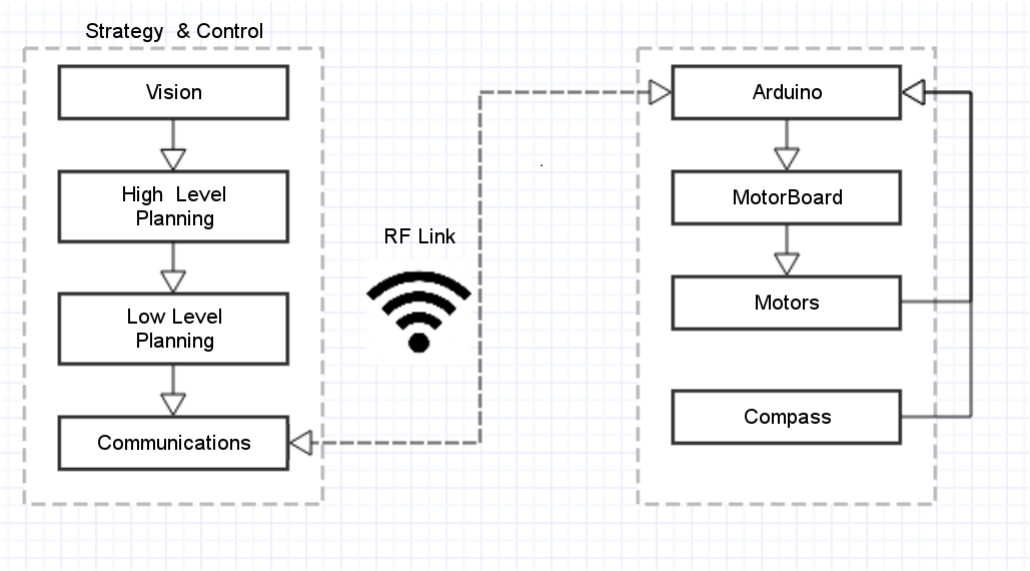

Figure 1: Full System Diagram

These interact with each other through the following interfaces:

- Arduino-Communications: The communication and Arduino systems interface directly, based on the communication protocol implemented by both. The PC end can be found at `comms/CommsThread.py` while the Arduino end can be found at `arduino/arduino.ino`.
- Communications-Strategy: An API is exposed by the communication system to provide a high-level API for the planner to perform various actions. This can be found at `comms/RobotController.py`. It uses the communication protocol to implement high-level actions via multiprocess communication to the communication system, which runs as a separate thread in parallel with the strategy process.
- Vision-Strategy: The vision and strategy communicate through `planning/worldAPI.py`. This instantiates a `WorldApi` object which can be read by the strategy. This then gets updated with the state of the ball and all robots in the game by the vision system, through the socket that is opened in `vision/sender.py`, which sends the data in a continuous loop. The reason that this method was chosen was that we could have one vision server that both teams could use, as opposed to having a separate running vision for each group. The nature of sockets, as well as their ease of use, made us choose them for this.

This architecture is based on that suggested in one of the introductory talks, but tailored more to fit our team and skill set, so we could divide the tasks up appropriately.

2 Hardware

2.1 Components

The Velosso robot consists of various parts and components. The main rationale for using the items listed below is to support full compass-driven and self-correcting holonomic motion for a lightweight and fast three-wheeled robot.

- Arduino Xino RF, as the main processing system of the robot;
- A Power & IO Board, directly interfaced with the Arduino, to break out digital, analog and I2C connections;
- A Motor Board for easy connections to the robots, addressed via I2C;
- A rotary encoder board, to easily connect and gather feedback from the motors' rotary encoders. Also addressed via I2C;
- A 6-axis MEMS magnetometer/accelerometer to create a tilt-compensated compass;
- 3x holonomic wheels, as well as various bits of Lego, comprising an equilateral triangular frame with a holonomic wheel on each side;
- 3x NXT interactive servo motors, attached to the wheels;
- 2x PF Medium motors, attached to the kicker and the grabber;
- 10x AA rechargeable batteries, encased in an 10-way battery holder;
- A green top-plate with 5 coloured dots, in accordance with the rules for this year;
- Connectors and wiring for all of the above.

2.2 Lego

When initially designing the robot, several different styles were considered including a reverse reliant robin and four wheeled holonomic design before settling on our current triangular, three wheeled design. This was because it allows us to fully utilise the holonomic wheels, giving the robot the ability to move in any direction and rotate itself while moving, while keeping it strong, compact and cheap, so more could be spent on high quality sensors.

2.2.1 Kicker and Grabber

We decided to use a fairly standard kicker and grabber setup, similar to those used in many robots in previous years, with a two clawed grabber able to pull the ball in to the paddle-style kicker. We chose this as it was a robust system which has produced good results in previous years, and with new rules about maximum kick speed and the ability to pass to an ally, the extra power which we may have been able to get from a solenoid kicker would not have been worth it.

2.2.2 Motors

We chose to use NXT motors for the wheels as they are the most powerful motor by a significant margin, 3.1W to the PF's 1.53W, and have built in rotary encoders, which we were wanting to use from the start. We then went with PF Medium motors for the kicker and grabber due to their balance of compact size and decent power.

2.3 Arduino System

The Arduino sub-system consists of a finite-state-machine-based implementation for concurrent Serial events, Action events and Sensor events, interfaced with an internal circular command buffer. The main ways in which this is achieved are by using the Arduino's serial interrupt-driven interface to execute serial events, and by finite-state-machines for both the command-based state of the Arduino, and the state of execution of each command.

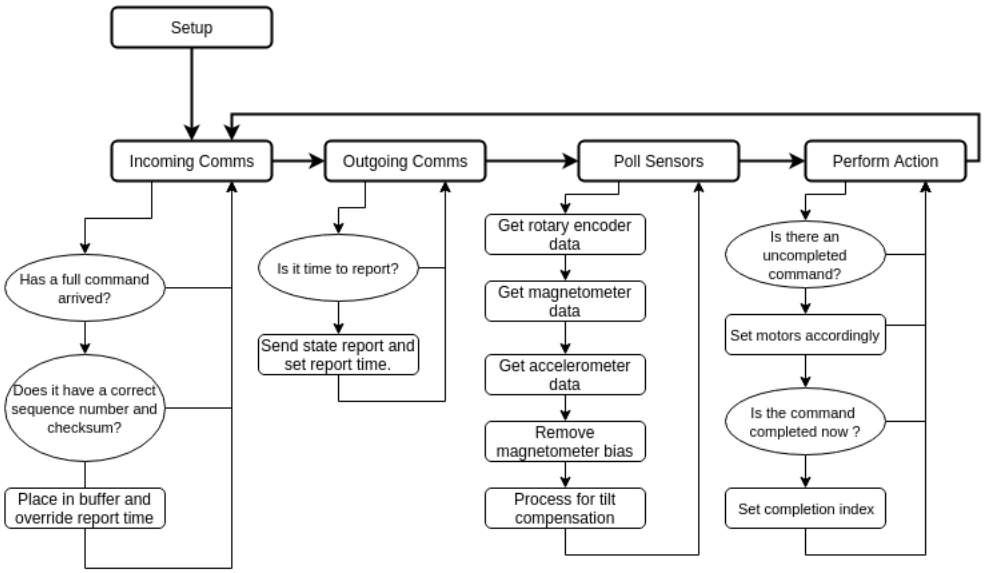

Figure 2: FSMs of the Arduino system

During setup, the Arduino is initialized into an idle state, and at each iteration of the main Arduino program loop, the following things happen:

- Sensor data is requested, or received. which alternate per loop iteration;
- Serial events are performed. Subject to a time-out, serial data is gathered and buffered into commands, if it exists;
- If there are any commands to be executed, the master finite-state machine of the Arduino changes its state to the identifier of the command, which triggers executing a part of each command. In addition to the Arduino's master finite-state machine reflecting its state based on which command it is performing, there are finite state machines controlled by it, to reflect the state of execution of each action.
- If there are no commands left to be executed, the Arduino stays in its idle state.

This implementation provides concurrency in performing and buffering actions, as well as gathering sensor data, allowing for the robot to have a larger representation of the planning system's strategy, as well as reduce communication latency. In addition, for specific cases,

the system also provides a direct way for the planning system to restore it to an idle state by flushing its buffer.

The completion of actions is based on sensory data, gathered at each main loop iteration from the rotary encoders and other sensors or by user-defined time-steps. Subsequently, it is important that no large delays are used in the Arduino, as this might disrupt the concurrent implementation.

The rationale for such a system was that implementing it would provide a very responsive interface to the strategy system. Up to 64 commands (more than enough for a single action) could be queued for the robot to perform, whilst all movement commands would be able to be corrected during execution both from the server side, by sending a command that overwrites the previous arguments, as well as from the client side, based on the tilt-compensated compass' values.

2.4 Sensory data

The data from the robot's sensors consists of rotary encoder data, accelerometer data and magnetometer data. An IR sensor was also considered for the purpose of knowing whether the robot has the ball or not. It was decided against it due to spatial constraints and improved accuracy of the vision system.

Communication to the sensors was done via I2C addressing, with the rotary encoders immediately responding with the necessary data, and the magnetometer and accelerometer setting a flag in one of their registers, indicating whether new data is ready. Each time new data is collected from the magnetometer and accelerometer, tilt-compensation for the magnet heading is performed by first eliminating electromagnetic interference, then using the accelerometer value to get the X-projection of the required magnet heading.

2.5 Compass-based holonomics

The compass-based holonomics are the most sophisticated aspect of the robot's system. Integrating a compass and using a concurrent Arduino implementation provides the opportunity for the robot to correct itself at each time step (15 milliseconds), whilst optionally receiving a new heading to correct itself towards from the comms system. Additionally, the compass provides the opportunity to use absolute magnetic coordinates, rather than ones relative to the robot which was considered a nice way to clear out any imperfections in the vision feed.

Compass-based holonomics is performed on the basis of two vector arguments - directional and rotational. Since maximum speed was always required, absolute magnetic angles were used instead, for both optimization and clarity. The directional angle indicated the angle the robot needs to be moving at, while the rotational angle indicated the angle the robot needs to be facing. Converting the two angles into motor speeds was based on matrix algebra from a Robocup SSL team's paper [1]. As referenced, three forces are calculated based on the motor positions, relative to the robot, as well as the arguments - accelerations in the X and Y coordinates, as well as a rotational one.

The first argument served to calculate the X and Y accelerations, and the second one served to provide the required rotational one, based on the difference between the robot's current heading and the required one. A useful optimization was removing the X and Y components

for very short distances or very large rotations required to be performed, effectively rotating the robot in-place in those use cases and both improving the accuracy and overall time required for a command to be performed.

3 Communication System

The communication system serves to interface the robot and the vision/strategy/planning system. On the server side is a Ciseco SRF Stick, while the Arduino uses it's own RF radio on the client side. Both operate at a baud rate of 115200bd. The communication protocol used is a simplified variant of the Transmission Control Protocol. All Commands share the following specification:

- All commands are of 4-byte length
- The first byte of each command is the command's ID number, AND-ed with a sequence number;
- The second and third byte are the data arguments of each command;
- The final byte is a checksum, calculated based on the number of set bits in the previous three commands, as well as their position to provide location independence;
- There are special bytes reserved for communication, all with a Hamming Distance greater than or equal to 2.

On the Arduino side, commands are received, validated via their sequence number and checksums, and a full state representation of command success and sensor data is encoded into 8 bytes and sent back to the server-side at a tuning temporal parameter. This serves to provide the server side with a complete representation of the robot's state, as well as cope with the high data loss and corruption rates that the hardware had been challenging us with.

The following command ID-s are used:

- CMD_ROT_MOVE_CCW: Perform anticlockwise rotation and movement;
- CMD_ROT_MOVE_CW: Perform clockwise rotation and movement;
- CMD_HOLO_MOVE: Perform holonomic motion or overwrite arguments, if currently moving holonomically;
- CMD_GRAB: Initiate grabbing action for the robot's grabber;
- CMD_UNGRAB: Initiate releasing action for the robot's grabber;
- CMD_KICK: Initiate kicking action for the robot's kicker;
- CMD_FLUSH: Flush robot's internal command buffer to restore idle state;
- CMD_STOP: Perform full stop.

The movement commands contain arguments indicating directions to move at, whilst the grab and ungrab command contain arguments indicating whether the grab should be atomic. Atomic in this case is defined as independent of the other commands, effectively allowing the robot to grab/ungrab whilst moving. All arguments for rotation and motion are their integer values in degrees and centimeters.

A higher-level API is provided by the server-side communication system via a robot controller (`comms/RobotController.py`), so as to allow for a wider range of motions than each single command's arguments can provide, as well as to provide a report about the robot's state and run the communication system as a parallel process.

4 Vision

The Vision System (see Figure 3) is a separate thread that runs independently to other subsystems. The goal of this system is to distinguish objects that are on the pitch and to constantly update the `WorldApi` object in the planner with this current world state, namely center coordinates and orientation vector of each robot, as well as coordinates of the ball. The system consists of three main modules:

- `camera.py` connects to the camera feed, gets a new image frame every 100ms and then returns an undistorted image of the pitch.
- `tracker.py` returns information about the robots and the ball in the given frame (see § 4.3)
- `sender.py` is a core module that is responsible for retrieving processed frames from `camera.py`, passing them into `tracker.py` to get the required object information and then sending this information to `world.py`.

There are also additional helper modules that support algebraic calculations and vector transformation that are used for determining robot orientations. Another module `update_colors.py` supports color re-calibration on the fly during the game.

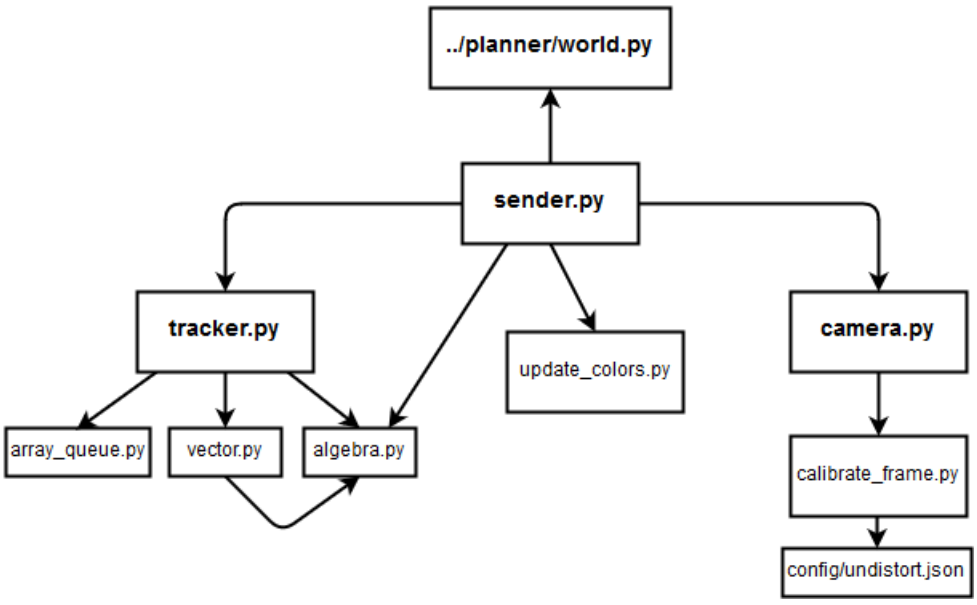

Figure 3: Diagram of Vision System

4.1 Vision calibration

The Vision calibration sub-system uses a graphical user interface in order to obtain the correct color thresholds. Therefore, once `color_calibration.py` is running:

- A window with the live video feed shows up and the user has to follow the instructions printed in the terminal by clicking on the corresponding colors.
- Once the user clicks on the screen, the BGR value of the pixel is saved and then converted to HSV value using OpenCV's functionality.
- Later, the HSV values for each color are used to initialize the sliding track-bars so that it becomes easier and quicker for the user to calibrate the color thresholds manually.
- After the user has finished clicking on the colors, the corresponding color mask appears together with the initialized track-bars and live video feed, allowing the user to adjust the HSV values until there is no noise in the mask.
- Once the user finishes manually adjusting the thresholds, the system saves the obtained values and writes them into `vision/config/colors.json`, which is used by `tracker.py`.

4.2 Image Processing

To get valuable information from video stream, image processing for each video frame was applied. Since we decided to rely only on the colours, the following operations [2] were performed:

- Changing color-space from RGB to HSV which abstracts colour (hue) by separating it from saturation and pseudo-illumination. This conversion eases object detection.
- Applying Gaussian blur to smooth the image and remove noise.
- Applying morphological transformation to obtain masks for the specified colour, which reduces even more noise.

For noise reduction we also considered applying median filtering, however this filter was slowing image processing significantly, causing a vision delay of over 2 seconds.

The decision to track objects by only relying on the colours was based on our goal to have the object tracking perform as fast as possible. Therefore, image processing is kept simple and efficient by only using colour thresholds and improving tracking accuracy by building a more complex top layer that deals with missing colours or their incorrect values.

4.3 Object tracking and positioning

Object tracking takes place in `vision/tracker.py` In order to start object tracking and orientation, the `RobotTracker` object for our robot has to be initialized with the parameters of our team color and the number of pink dots on its top plate. This initialization gives us the necessary information about other robots so we can distinguish between them.

The relevant information (robot orientation and its coordinates) of a requested robot is retrieved by calling the `getRobotOrientation(self, frame, side, position)` function, which takes a frame that is used for information retrieval, as well as the team colour (as `side` argument) and robot colour (as `position` argument). When this function is called, information is returned by following these steps:

- First it finds where the robot might be by searching for its team color (applying a mask with the color threshold values obtained during the calibration process).

- If it is searching for the pink robot, the system looks for the three closest pink contours around the robot center, otherwise it looks for the three closest green contours.
- Once it knows the coordinates of the pink and green circles on the top plate, it calculates the direction vector of the robot.
- For a pink robot, the orientation vector is found by taking the center coordinates of the top plate together with the center of the bright green dot, initializing the vector using these coordinates, then rotating this 215 degrees anticlockwise.
- For a green robot, the same method as above is applied but using the pink dot instead of the green when initializing the direction vector

For object tracking we also considered using background subtraction to give specific regions to search for the objects. However, due to the huge noise in the analog video feed, the accuracy of this method was far too low to be of use.

4.4 Sending information using multi-threading

- After calibration and tracking have been done, `sender.py` takes the following arguments as input: pitch number (0 for 3.D04, 1 for 3.D03), team colour (`yellow` or `bright_blue`), our robot colour (`pink` or `green`) and ball colour (`red` or `blue`), which are used to initialize the appropriate trackers.
- After that, a socket is opened so we can send the information to the client, the `worldAPI`.
- The `getBallCoordinates()` and `getAllRobots()` functions are called in the main loop from the initialized trackers, which update the world continuously.
- The information is sent using the `socket.send_pyobj(obj)` function, where in our case `obj` is a dictionary containing the position and rotation of all the robots and the position of the ball.

5 Strategy and Planning

The computer makes decisions about what command to send to the robot in `planning/robotAI.py`. This system operates in discrete 'ticks' of 0.1 seconds. Each tick, it updates its beliefs about the state of the game using the most recent information from the vision system, then calls the strategy module to calculate the most optimal action to be taken this tick based on these beliefs. The system includes several important constraints to ensure the robot conforms to the rules of the SDP game.

5.1 Strategy

The strategy module, found in `planning/strategy.py`, describes the high level tactics which the robot uses throughout the game. Each tick, the system decides upon a goal to aim for, then sends commands to the robot to try and achieve this goal. Its possible goals are to collect the ball, shoot, pass, confuse the enemy to make them move out of the way, prepare to receive a pass, block the enemies' pass, or guard our goal line.

This module is designed to be easily swapped, allowing for quick tactic changes each half to reflect the opponents' playsyle. The currently implemented system is shown by the FSM in Figure 4. This is a strong default strategy designed to perform well against most robots, though it can be weak against robots which are able to shoot in a direction they're not facing.

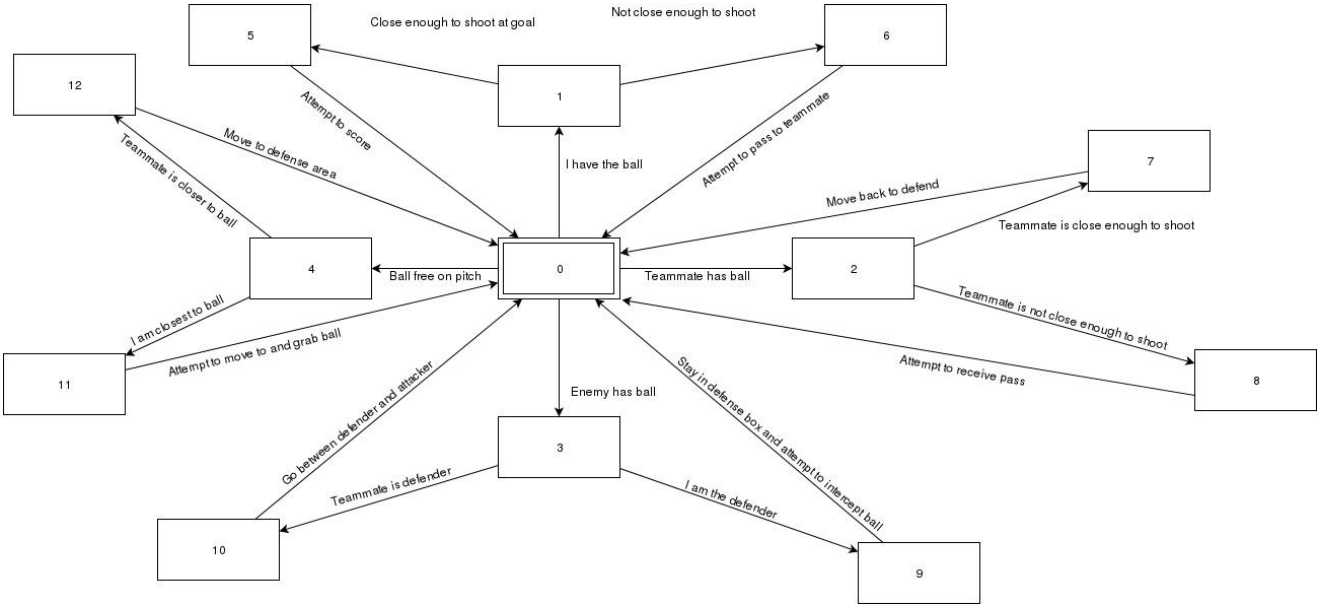

Figure 4: FSM of the given strategy

5.2 Planning

Once the strategy system has decided upon a goal, the planning module, found in `planning/goals_new.py`, decides what commands to send to the robot in order to help achieve this goal. For example, if the goal is to block an enemy's pass, we find the midpoint between the two enemy robots and drive there as fast as possible, then turn to face the enemy with the ball, ready to intercept. We have chosen to keep this low level planning separate from the high level strategy as the low level stuff is independent of any tactics; whatever your clever plans, you'll be doing the same thing when you do decide to go and fetch that ball. This means that only the tactics get changed when you switch in a new strategy module, not the actual execution of achieving each goal.

5.2.1 Obstacle Avoidance

In order to not run into other robots, the system implements a line of sight function, that using trigonometry determines if the current action would result in the robot fouling by driving into another robot. If it detects this would happen, rather than taking a straight path to its destination, it takes a ghost walk there (see § 5.2.3).

This same function is also used to determine if the robot can shoot safely for the goal, pass to an ally and determine if intercepting the ball is feasible.

5.2.2 Goal Boxes

Under the rules of SPD football you cannot enter the opponents goal box or your own if a teammate is currently residing there. In order to conform with this, once the system has generated an action for the current tick it checks it will not violate the aforementioned rule. If the current action would result in fouling, the robot will move, at most, half the distance to the desired point. While this does not guarantee safety, as the current action is updated every 100ms, no action will ever be executed long enough to reach this target position.

5.2.3 Ghost Walking

Ghost walking is the name given to the action of safely randomly moving when no other action can be executed. This works by generating a random movement within 30cm of the robot, containing a random rotation value also. This tactic is useful to keep the robot constantly moving and also to confuse opponents with occasional erratic movements.

6 References

[1] - http://people.idsia.ch/ foerster/2006/1/omnidrive_kiart_preprint.pdf
[2] - http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_tutorials.html