Robot Sensors MARRtino

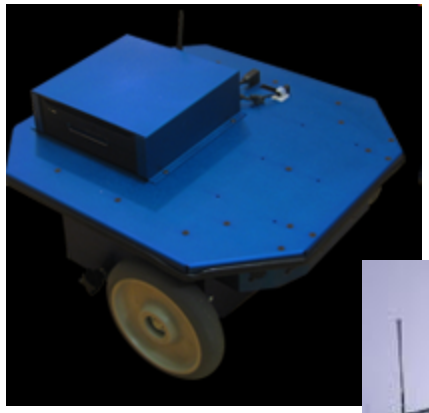
Giorgio Grisetti, Tiziano Guadagnino

Outline

- Robot Devices
  - Overview of Typical sensors and Actuators
  - Operating Devices in ROS
- Mobile Bases
- MARRTino
  - Hardware
  - Firmware

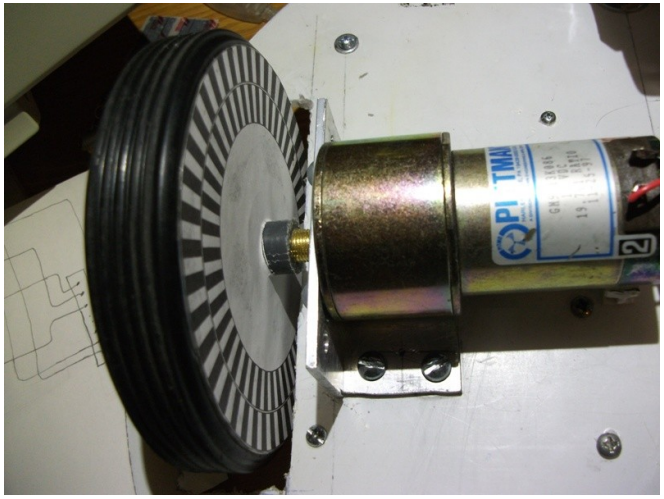
Mobile Base

- A mobile platform is a device capable of moving in the environment and carrying a certain load (sensors and actuators)
- At low level the inputs are the desired velocities of the joints, and the output is the state of the joints
- At high level it can be controlled with linear/angular velocity, and provides the relative position of the mobiel base w.r.t. an initial instant, obtained by integrating the joint's states (odometry).



Proprioceptive Sensors for Ego-Motion

- Wheel encoders mounted on the wheels
- IMU:
  - Accelerometers
  - Gyros



- The estimate of ego-motion is obtained by ***integrating*** the sensor measurements of these devices. This results in an accumulated drift due to the noise affecting the measurement
- In absence of an external reference there is ***no way*** to recover from these errors

Exteroceptive Sensors

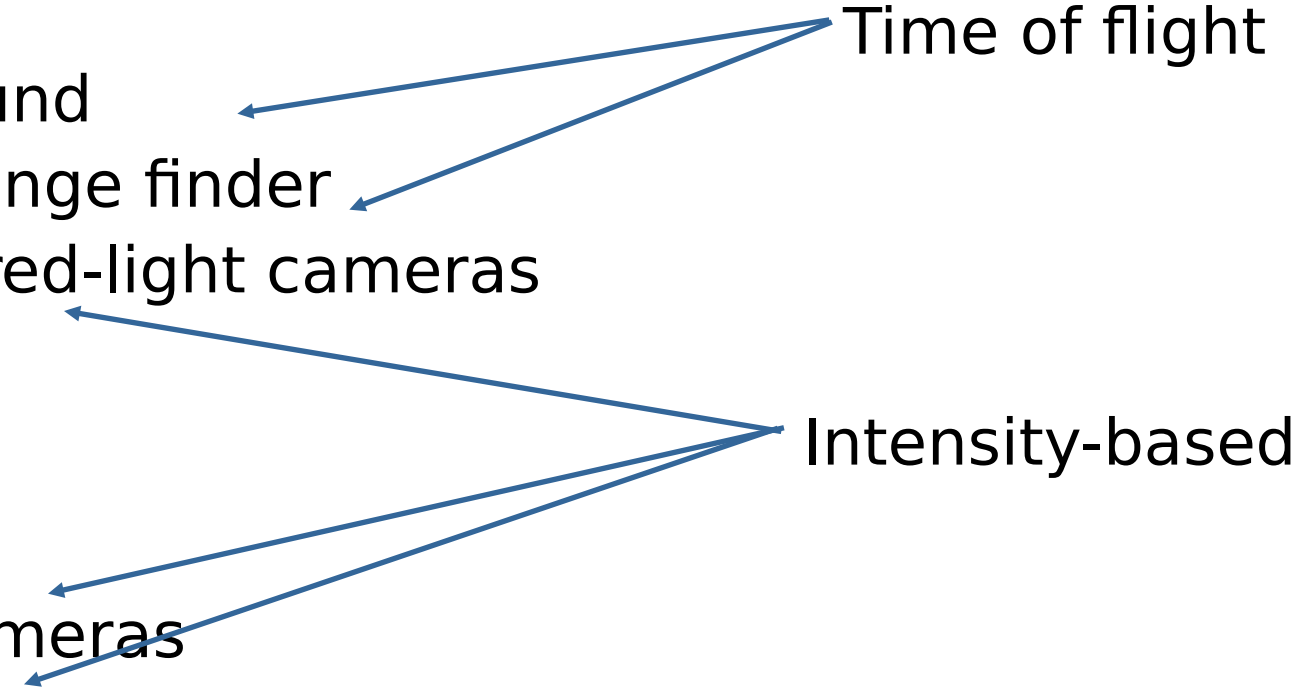
Perception of the environment

Active:

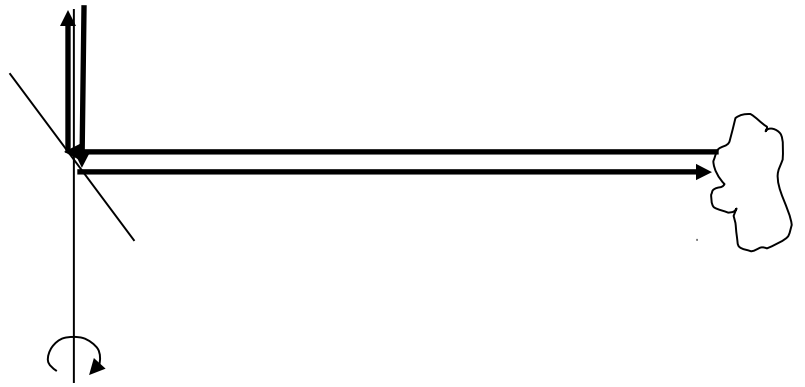
- Ultrasound
- Laser range finder
- Structured-light cameras
- Infrared

Passive:

- RGB Cameras
- Tactiles



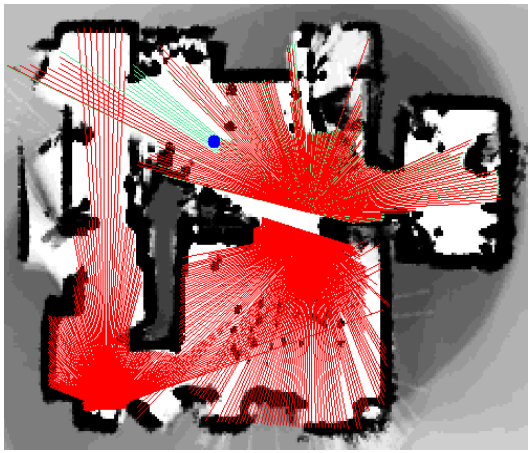
Laser Range Scanner

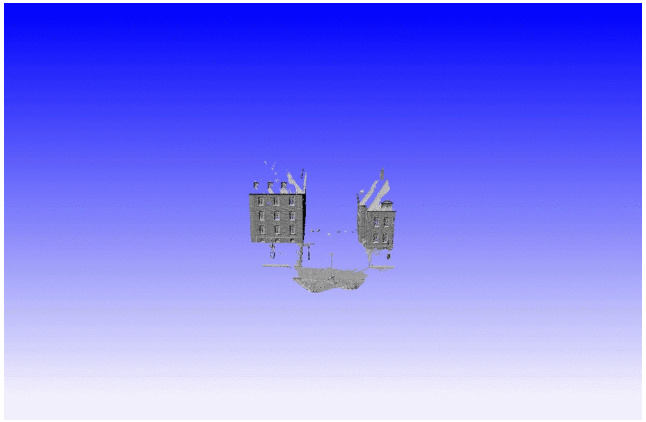


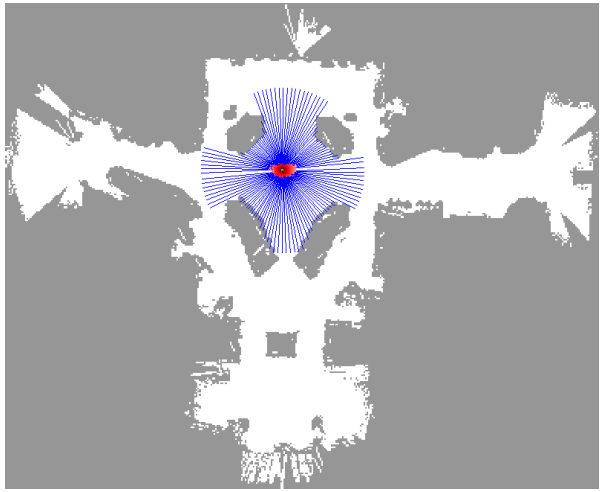
Properties

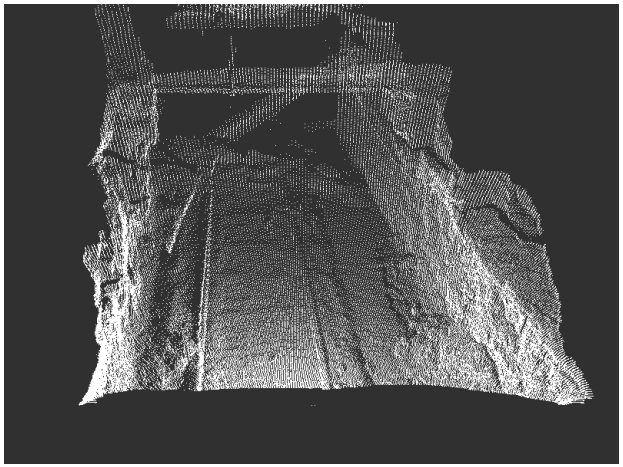
- High precision
- Wide field of view
- Approved security for collision detection

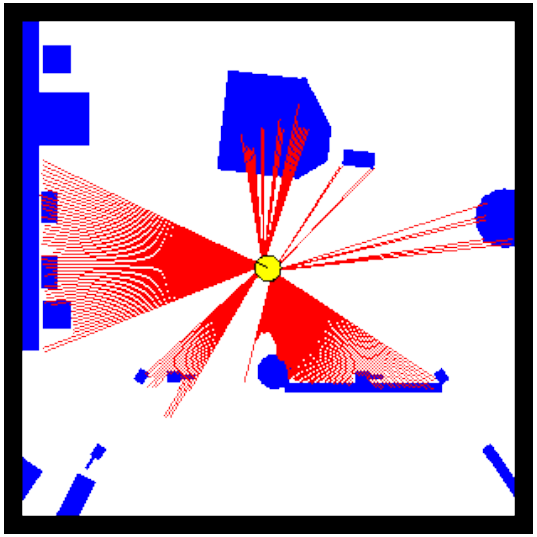
Typical Scans



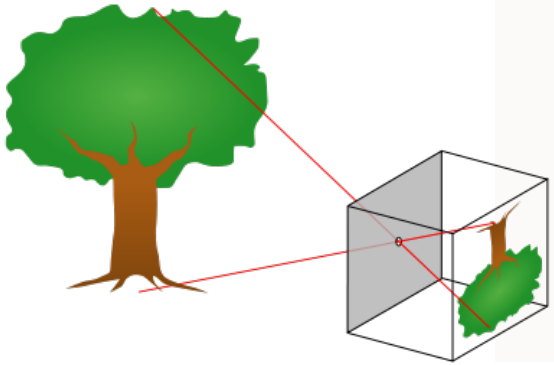


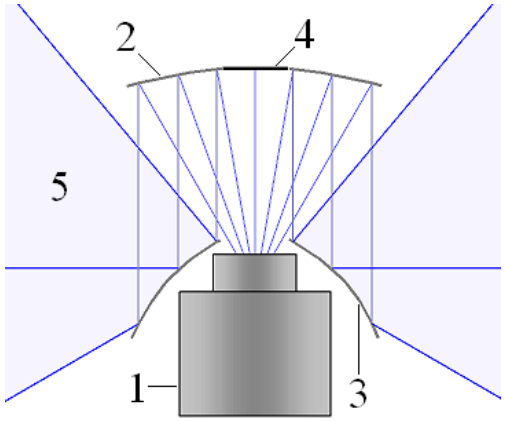


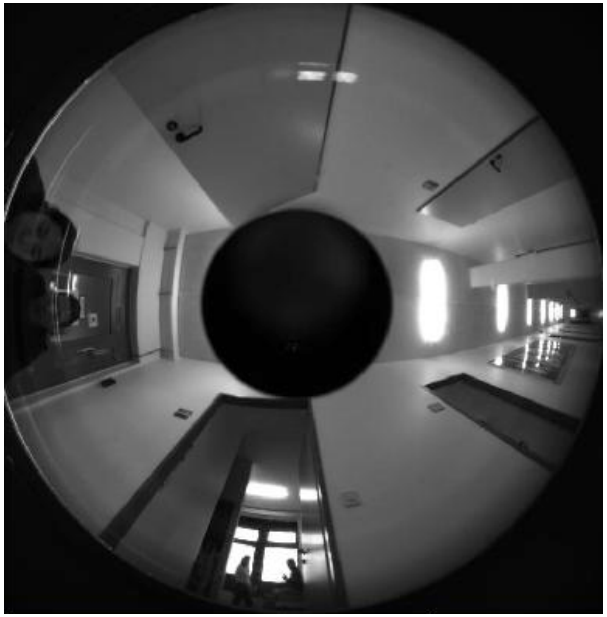




RGB Monocular Camera





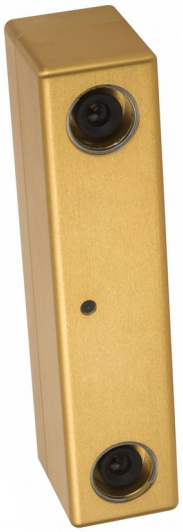


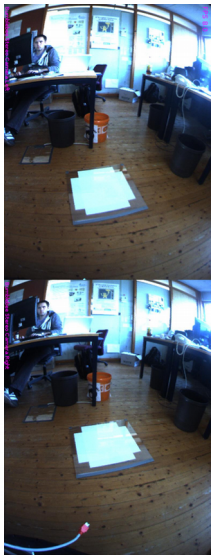
RGB Monocular Camera

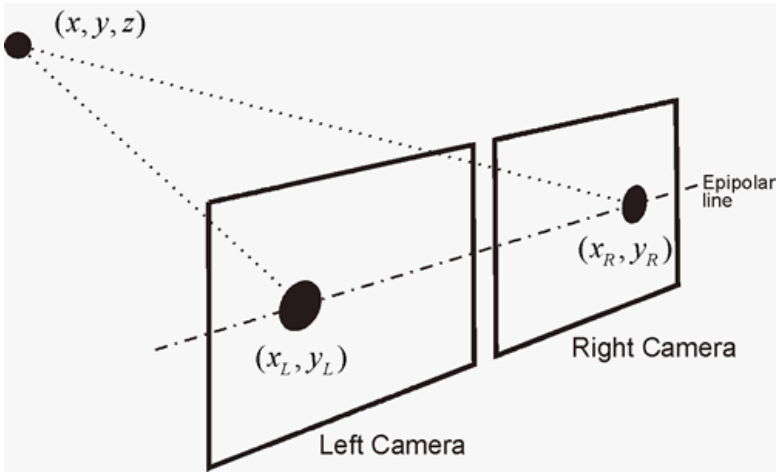
- Cameras measure the intensity of the light projected onto a (typically planar) ccd through a system of lenses and/or mirrors
- Provide a lot of information
- Project 3D onto 2D, which results in the unobservability of the depth
- The scene can eb reconstructed by multiple images (see SfM)

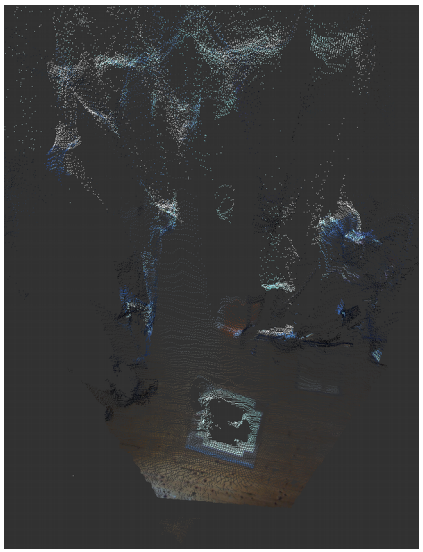
Stereo Camera

reconstruction
from top





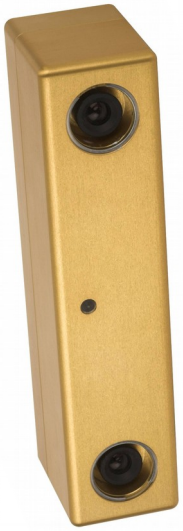


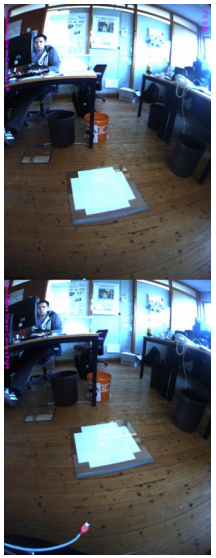


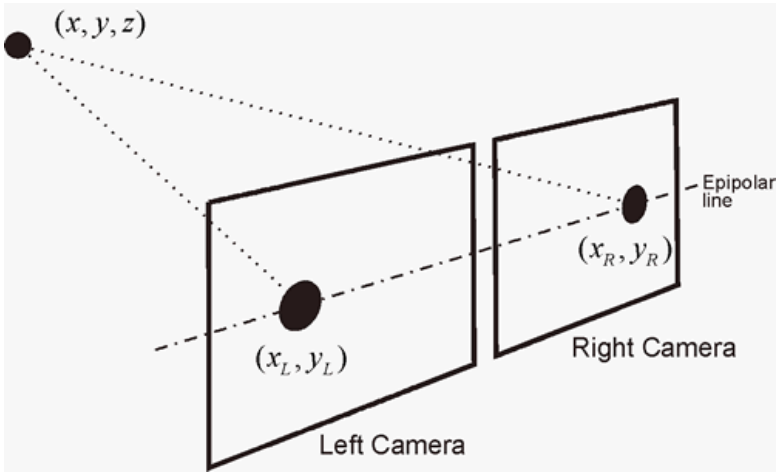
- Stereo cameras are combination of 2 monocular cameras that allow triangulation, given a known geometry.
- If the corresponding points in the images are known, we can reconstruct the 3D scene.
- Error in the depth depends on the distance!
- Sensible to lack of texture

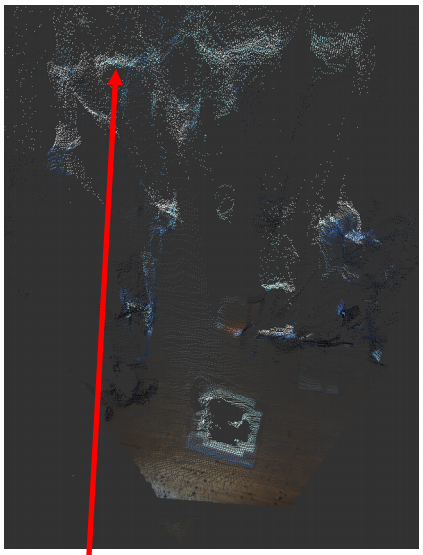
Stereo Camera

reconstruction
from top





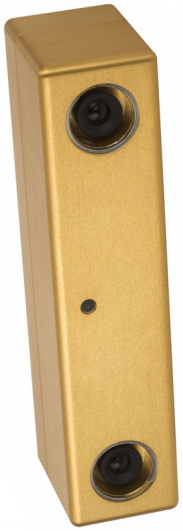


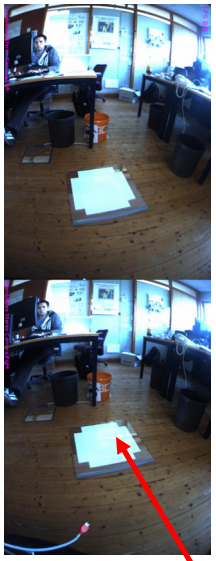


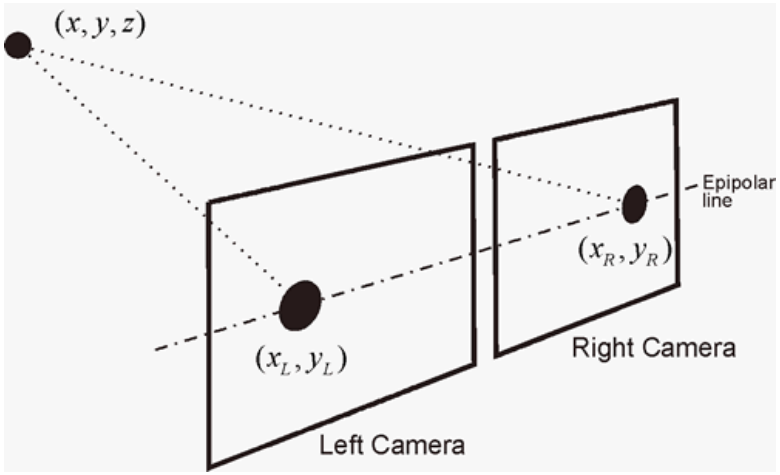
- Stereo cameras are combination of 2 monocular cameras that allow triangulation, given a known geometry.
- If the corresponding points in the images are known, we can reconstruct the 3D scene.
- Error in the depth depends on the distance!
- Sensible to lack of texture

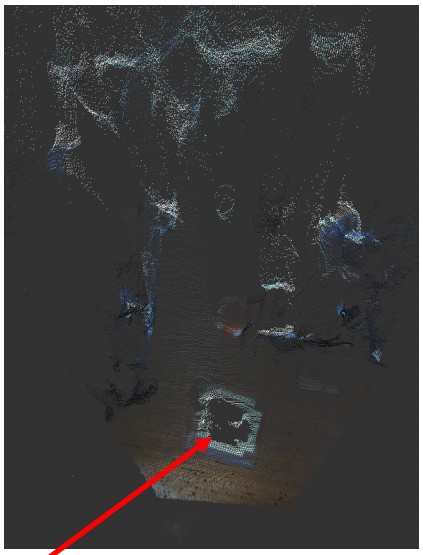
Stereo Camera

reconstruction
from top









- Stereo cameras are combination of 2 monocular cameras that allow triangulation, given a known geometry.
- If the corresponding points in the images are known, we can reconstruct the 3D scene.
- Error in the depth depends on the distance!
- Sensible to lack of texture

How to access a Device in ROS?

- Each device is a node
- The input topics are the commands that the device can output
- The output topics are the feedback given by the device.

- In sensor_msgs/ many messages for the common sensors are defined.
- Use rosmsg show <message_name> to see the format of a message.

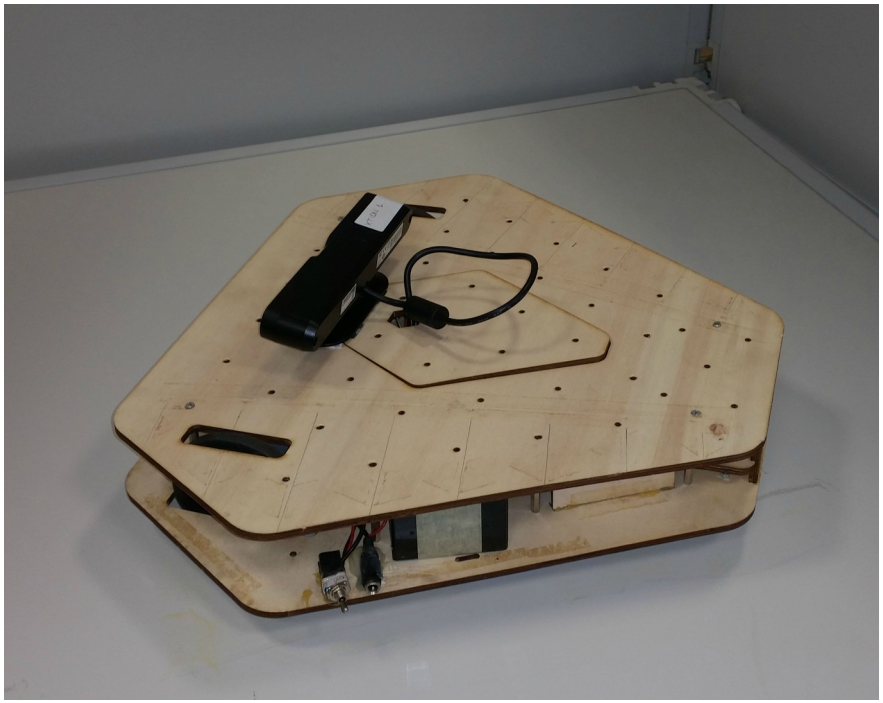
- To start a device it is sufficient to start the corresponding node and to give it the necessary configuration parameters. These include
  - Specific devices parameters (e.g. which serial port/usb device , the resolution of an image, and so on..)
  - The **name** of the **reference frame** in the sensor

MARRtino

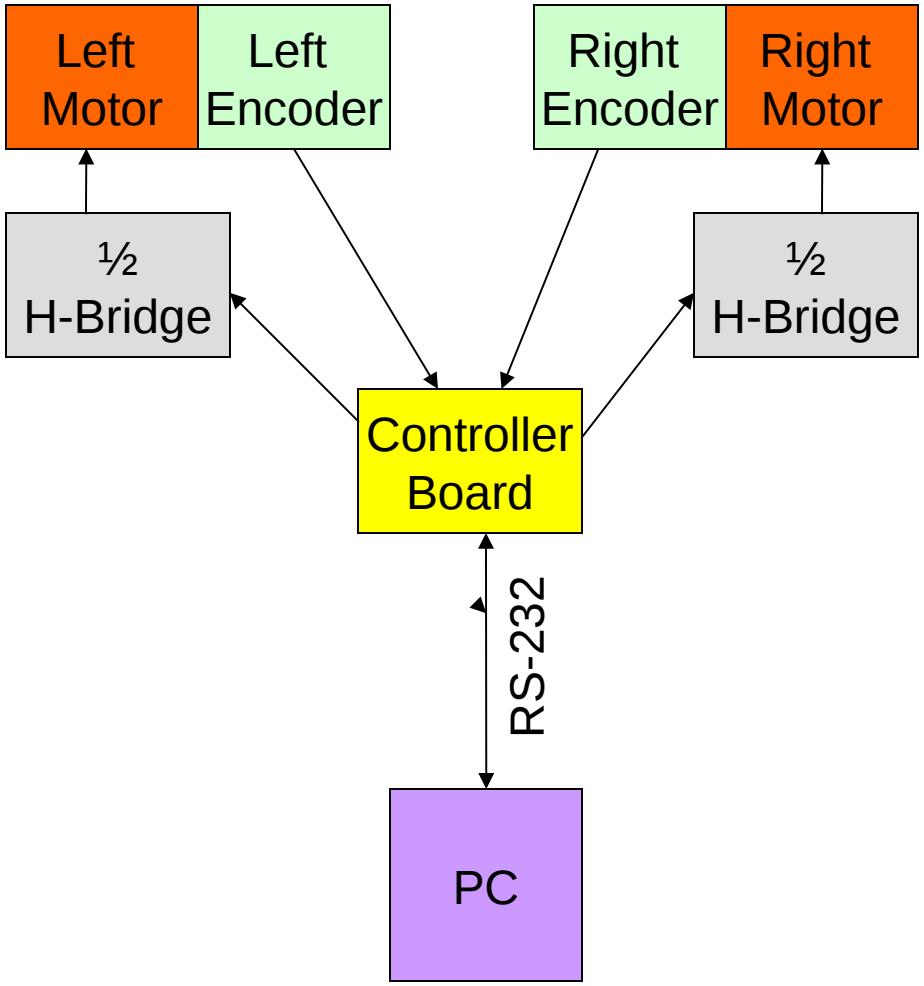
- Is a simple but complete mobile base designed to be used in the MARR course.
- The cost of the parts is around 300 euro
- It is entirely open source
- It is integrated in ros through a simple node that publishes/subscribes standard topics

MARRtino

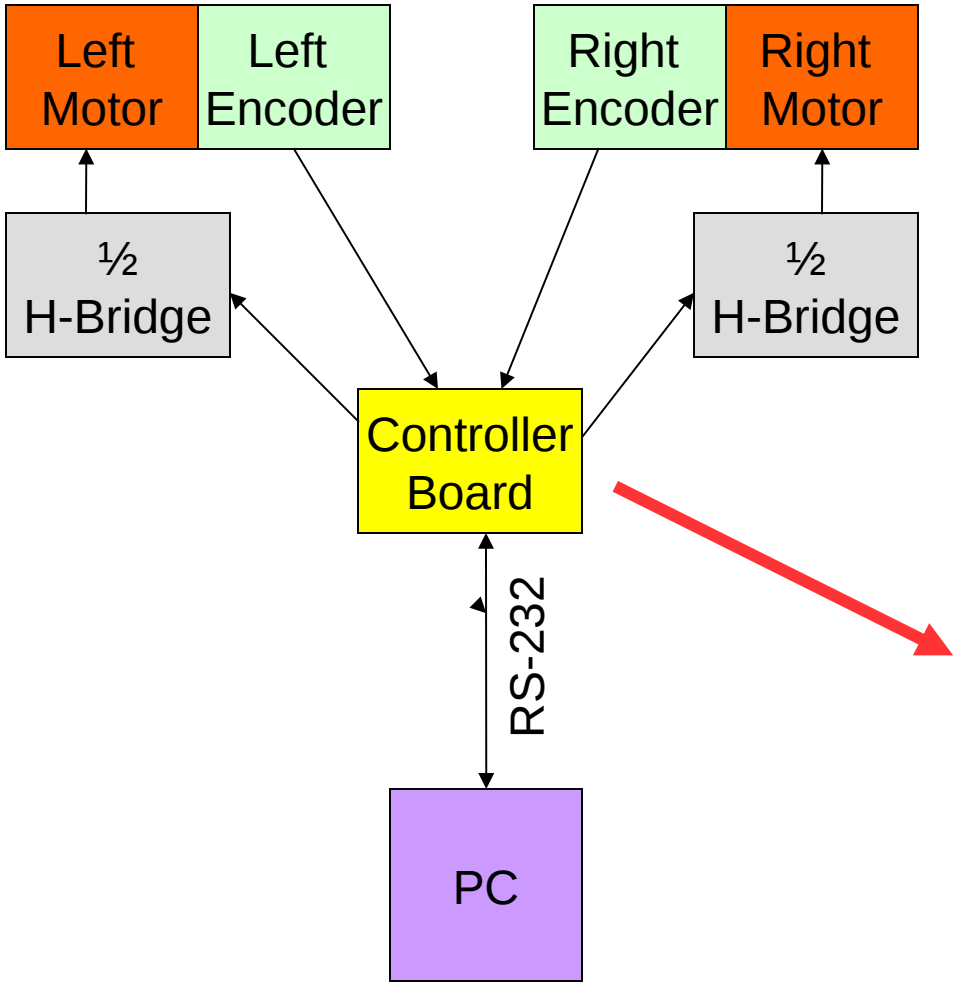
- Is a simple but complete mobile base designed to be used in the MARR course.
- The cost of the parts is around 300 euro
- It is entirely open source
- It is integrated in ros through a simple node that publishes/subscribes standard topics

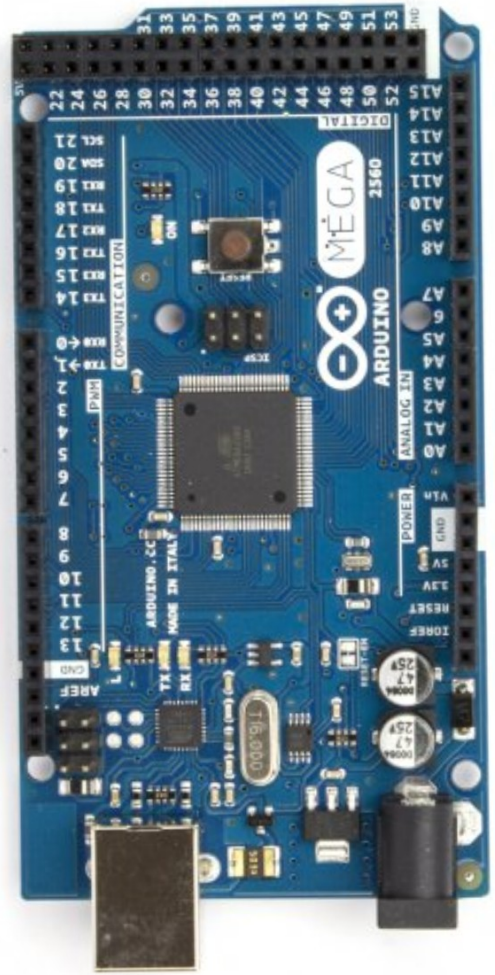


MARRtino: Electronics



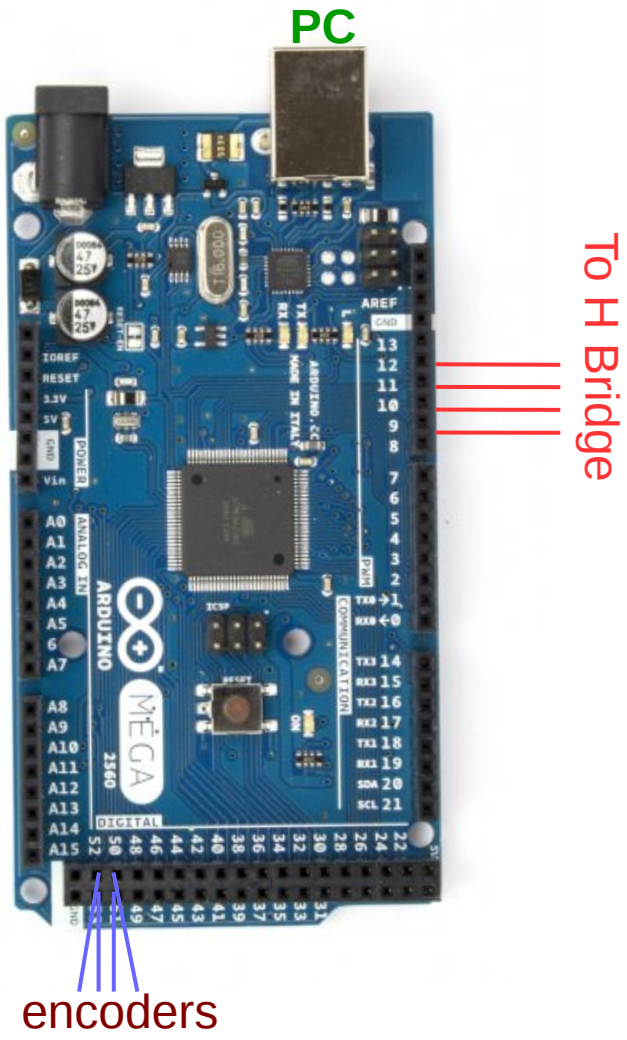
MARRtino: Electronics





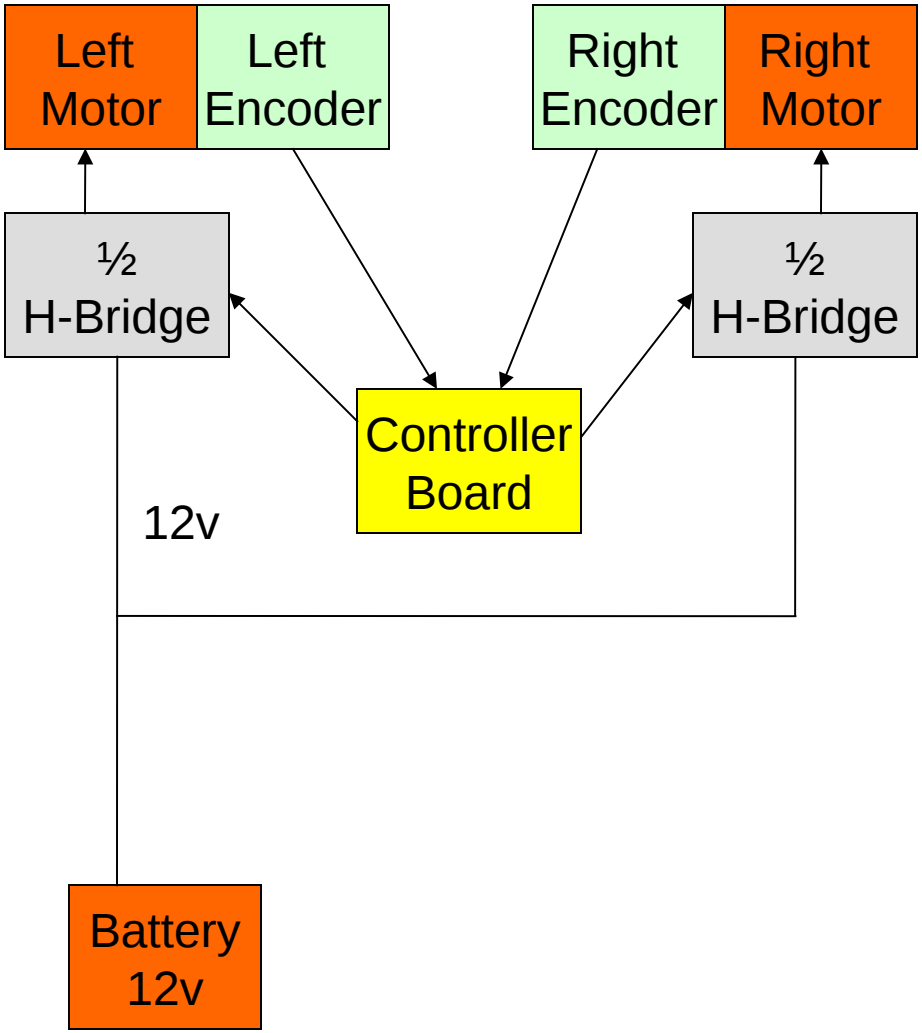
MARRtino: Electronics

- The PC communicates with arduino through USB
- Each encoder provides two signals
- Each PWM requires at least 2 wires
- The wiring of the PWM depends on the H-Bridge used



MARRtino: Power

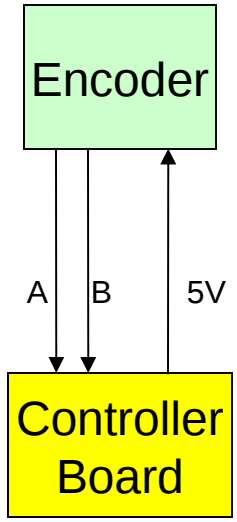
- H bridges: 12 V from both batteries, 5V from logic generated from internal regulator
- The system can either charge the batteries or be powered ON.



Controller is powered through USB
Controller and H bridges share the GND

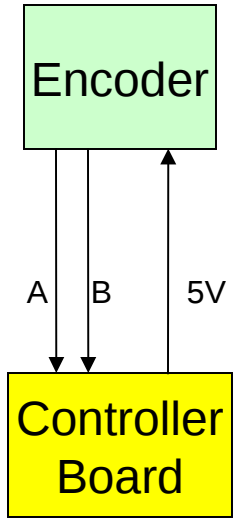
MARRtino: Encoders

- Each encoder has two signals (A, B) and requires a 5V voltage supplied by the controller board
- The encoders are managed by the Quadrature Encoder Module (QEI) of the controller, that takes care of counting ticks and direction



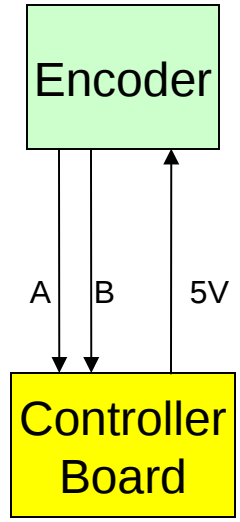
MARRtino: Encoders

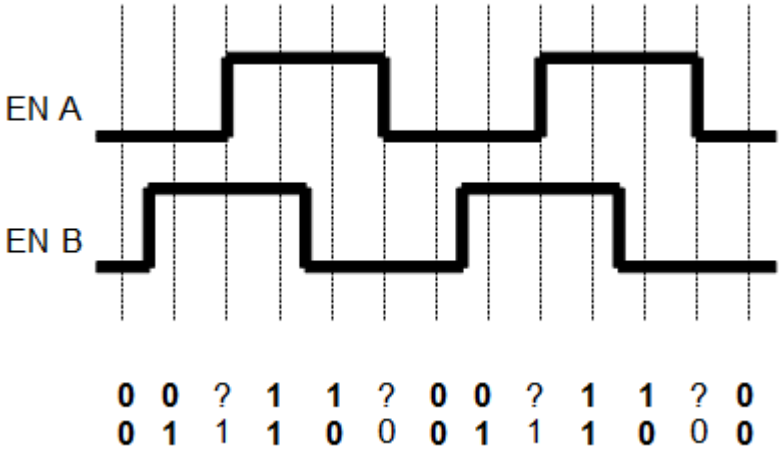
- Each encoder has two signals (A, B) and requires a 5V voltage supplied by the controller board
- The encoders are managed by the Quadrature Encoder Module (QEI) of the controller, that takes care of counting ticks and direction



MARRtino: Encoders

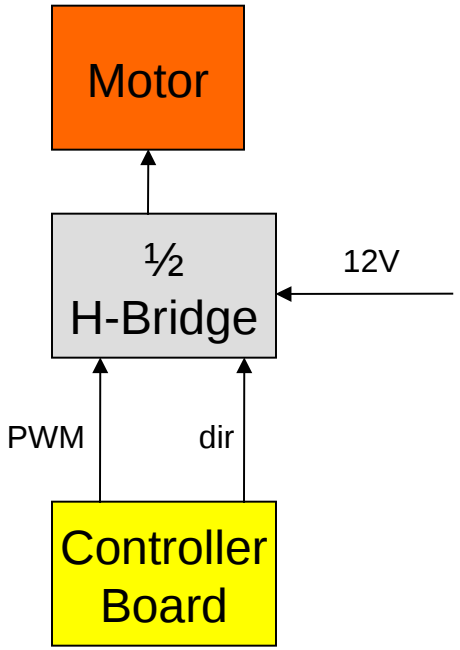
- Each encoder has two signals (A, B) and requires a 5V voltage supplied by the controller board
- The encoders are managed by interrupt on edges, that takes care of counting ticks and direction





MARRtino: H Bridge

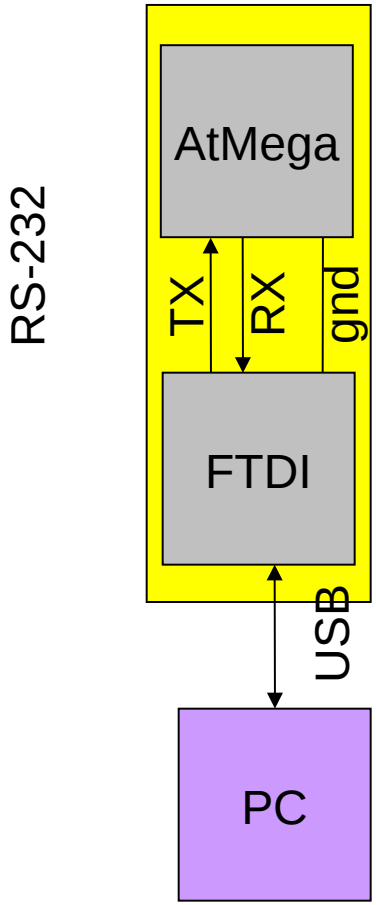
- The motor is connected to the H Bridge, that provides the necessary voltage and current to drive it.
- The H bridge requires 12V power directly from the battery
- The controller board controls the H bridge by*
  - A square wave whose duty cycle is proportional to the voltage applied to the motor, that controls the speed (PWM)
  - A direction pin, that reverts the voltage when asserted, causing the motor to rotate in the opposite direction



* Other modes are possible, please refer to the wiki for details

MARRtino: PC connection

- The robot communicates with the PC through an RS232 interface at TTL levels (0-5V)
- The TTL-RS232 is converted in USB through an FTDI chip
- The device is visible on Linux as /dev/ttyXXX



MARRTino: Firmware

- The controller runs an event driven C program that
  - Executes PID controllers on both wheels
  - Integrates the odometry
  - Communicates with the PC
  - Implements a watchdog
- The PC periodically sends to the controller the desired translational/rotational velocities
- The controller periodically sends to the PC state packets that include the integrated odometry since the last message and the battery state
- If the controller does not receive any message from the PC for a while it sets the speed to 0 (safety measure)

MARRTino: Wiki

- The wiki describes all the steps to:
  - obtain the firmware code
  - flash the firmware on the board
  - run the web client and connect the parts while configuring the board
  - Starting a ROS node

- **www.dis.uniroma1.it/~spqr/MARRtino**
- **https://gitlab.com/srrg-software/srrg2_orazio**

More about Firmware

- Orazio: is the firmware for a mobile robot developed mainly by Giorgio, running on the control board
- Features
  - multi architecture (ATMega, dsPIC)
  - supports a variable number of joints
  - support sonars
  - layered architecture
    - current/joints/platform

Orazio: Subsystems

Modules running on the control board

- System: communication, watchdog, packet selection
- Joints: control of the motors at joint level
- Drive: control of the platform (base velocities/odometry)
- Sonar: sonars

Each subsystem has

- has a status (updated by the program on the control board and sent periodically to the host)
- parameters (stored in EEPROM)
- can receive commands

Packets

```
  // here are the limits
typedef uint8_t PacketType;
typedef uint8_t PacketSize;
typedef uint16_t PacketSeq;

typedef struct {
  PacketType type;
  PacketSize size;
  PacketSeq  seq;
} PacketHeader;

//! sent from the pc to the robot causes
//! the robot to send a ParamPacket to the
//!PC (with the same seq)
typedef struct {
  PacketHeader header;
  //0: send current params
  //1: load params from eeprom
  //2: write current params to eeprom,
  uint8_t action;
  // identifies the parameter class
  // 0: system
  // 1: joints
  // 2: drive
  uint8_t param_type;
  // the index of a subdevice
  uint8_t index;
} ParamControlPacket;
#define PARAM_CONTROL_PACKET_ID 1
```

```
  ....

  //! sent from the pc to the robot causes
  typedef struct {
    PacketHeader header;
    float translational_velocity;
    float rotational_velocity;
  } DifferentialDriveControlPacket;
#define DIFFERENTIAL_DRIVE_CONTROL_PACKET_ID 8

  typedef struct {
    PacketHeader header;
    float ikr;
    float ikl;
    float baseline;
    float max_translational_velocity;
    float max_translational_acceleration;
    float max_translational_brake;
    float max_rotational_velocity;
    float max_rotational_acceleration;
    uint8_t right_joint_index;
    uint8_t left_joint_index;
  } DifferentialDriveParamPacket;
#define DIFFERENTIAL_DRIVE_PARAM_PACKET_ID 9

  ...`
```

Orazio: Packets

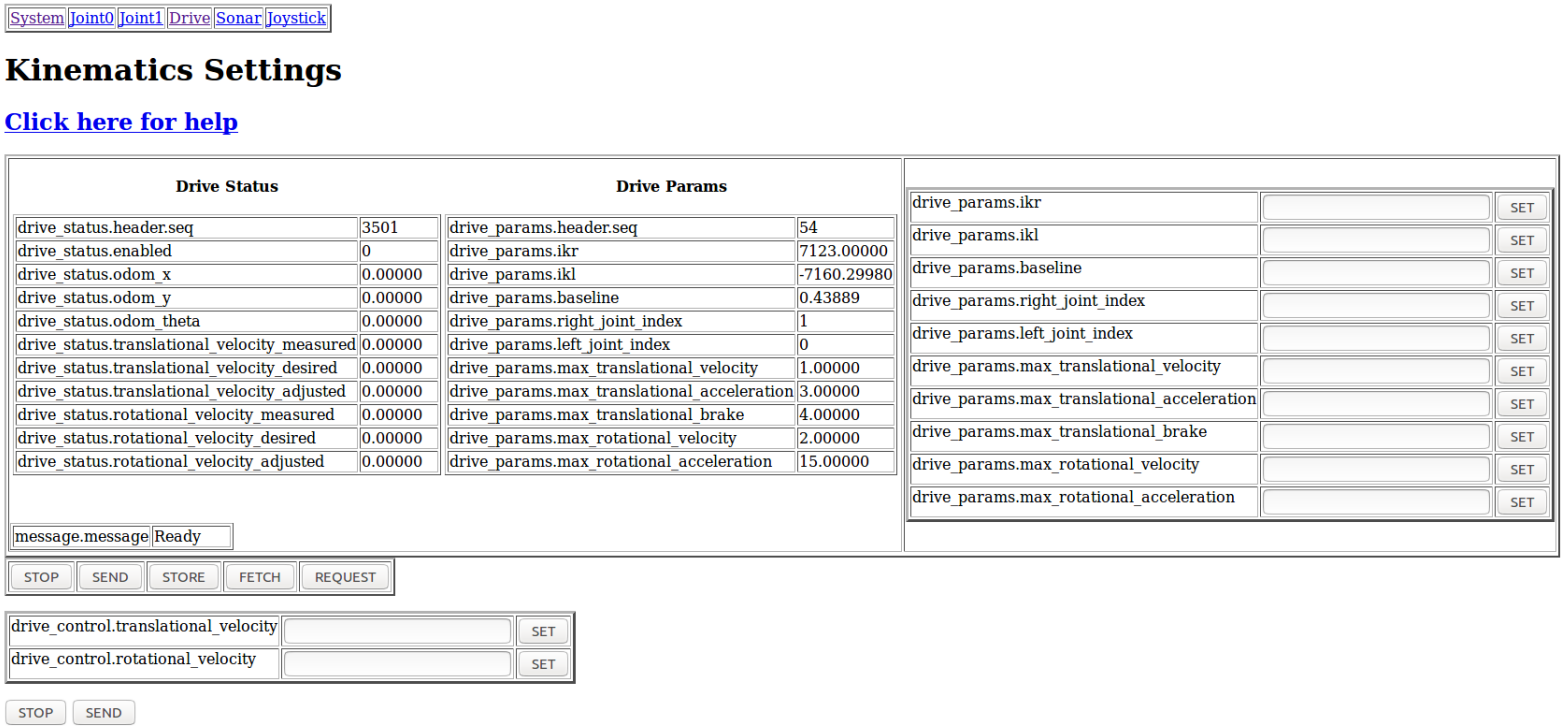
- The communication between host and controller happens by sending serial packets (see orazio_packets.h)
- Each packet has a unique id, and carries one type of information
- Packets from PC:
  - issue commands (e.g. setSpeed)
  - control parameters
    - set (write)
    - save (to EEPROM)
    - restore (from EEPROM)
    - get (read)
- Packets from robot
- Status of subsystems
  - Parameters
  - Responses

Host Interaction

- Orazio sends periodically at a configurable rate status packets of all subsystems of interest
- The host reads these packets, and the "client" program stores the last packet of each type in a variable (for later inspection)
- All packets of a same interval constitute an "epoch"
- If needed, the client sends some command to the host
- After each epoch, the client sends a "keepalive" packet to notify the control board it is still running
- OrazioClient is the "C" class that implements these functionalities

WEB interface

- Client application that communicates with orazio
  - used to set the parameters, and issue controls
  - program starts a local web server
  - from host_build
    - orazio -serial-device /dev/ttyACM0 -resource-path ../html
    - Hint: flash the firmware and follow the instructions to connect the devices



Client Interface (C library)

```
// opaque object
struct OrazioClient;

// opens connection
struct OrazioClient* OrazioClient_init(const char* device, uint32_t baudrate);

// destroyes orazio and close connection
void OrazioClient_destroy(struct OrazioClient* cl);

// sends a packet, the header should be compiled to match the packet type and size
PacketStatus OrazioClient_sendPacket(struct OrazioClient* cl, PacketHeader* p, int timeout)

// retrieves a packet, the header should be compiled to match the packet type and size
// the rest is filled
PacketStatus OrazioClient_get(struct OrazioClient* cl, PacketHeader* dest);

// how many motors is this firmware supporting?
uint8_t OrazioClient_numJoints(struct OrazioClient* cl);

// 1 cycle of handshake, advances the epoch
PacketStatus OrazioClient_sync(struct OrazioClient* cl, int cycles);

// queries all parameters (to be done after syncing)
PacketStatus OrazioClient_readConfiguration(struct OrazioClient* cl, int timeout);
```

Minimal C program to connect to the robot

```
static struct OrazioClient* client=0;
char* default_serial_device="/dev/ttyACM0";

int main(int argc, char** argv) {
  // 1. create an orazio object and initialize it
  client=OrazioClient_init(serial_device, 115200);
  if (! client) {
    printf("cannot open client on device [%s]\nABORTING", serial_device);
    return -1;
  }

  // 2. synchronize the serial communication
  printf("Syncing ");
  for (int i=0; i<50; ++i){
    OrazioClient_sync(client,1);
    printf(".");
    fflush(stdout);
  }
  printf(" Done\n");

  // 3. read the configuration
  if (OrazioClient_readConfiguration(client, 100)!=Success){
    return -1;
  }
```

Minimal C program to connect to the robot

```
int retries=10;
// to read a packet from the client, the program has
// to preconfigure a truct with the parameter type
SystemParamPacket system_params={
  .header.type=SYSTEM_PARAM_PACKET_ID,
  .header.size=sizeof(SystemParamPacket)
};
OrazioClient_get(client, (PacketHeader*)&system_params);

// we enable the sending of all packets
system_params.periodic_packet_mask=
  PSystemStatusFlag
  |PJointStatusFlag
  |PSonarStatusFlag
  |PDriveStatusFlag;
// we send the new parameters (n retries)
OrazioClient_sendPacket(client, (PacketHeader*)&system_params, retries);
// we read the parameters again
OrazioClient_get(client, (PacketHeader*)&system_params);
```

Minimal C program to connect to the robot

```
  int max_epochs=100;
  for(int epoch=0; epoch<max_epochs; ++epoch){
    OrazioClient_sync(client,1);
    char output_buffer[1024];
    SystemStatusPacket system_status={
      .header.type=SYSTEM_STATUS_PACKET_ID,
      .header.size=sizeof(SystemStatusPacket)
    };
    OrazioClient_get(client, (PacketHeader*)&system_status);
    Orazio_printPacket(output_buffer,(PacketHeader*)&system_status);
    printf("SYSTEM: %s\n",output_buffer);
    DifferentialDriveStatusPacket drive_status = {
      .header.type=DIFFERENTIAL_DRIVE_STATUS_PACKET_ID,
      .header.size=sizeof(DifferentialDriveStatusPacket)
    };
    OrazioClient_get(client, (PacketHeader*)&drive_status);
    Orazio_printPacket(output_buffer,(PacketHeader*)&drive_status);
    printf("DRIVE: %s\n",output_buffer);
  }
  printf("Done\n");
  OrazioClient_destroy(client);
}
```

Task: write a C++ class to connect to Orazio

- The object should encapsulate the most common functionalities of the c driver
  - sync
  - read odometry
  - read sonar
  - read speed
- It should be easy to use
- The destruction of the object should cause the connection to close in a clean way

- Fill the cpp file of oraziocpp.cpp to handle the case
- Compile with make issued from the directory /host_build
- test by executing oraziocpp_test from host_build

For the next time: ROS

- Install ROS from here (this version)

  http://wiki.ros.org/kinetic/Installation/Ubuntu
- Do all until "environment setup" (1.6)
- Clone repo : https://gitlab.com/tizianoGuadagnino/lab_ai_gi
- Run the script

  install_labaigi_software.sh (from repo)