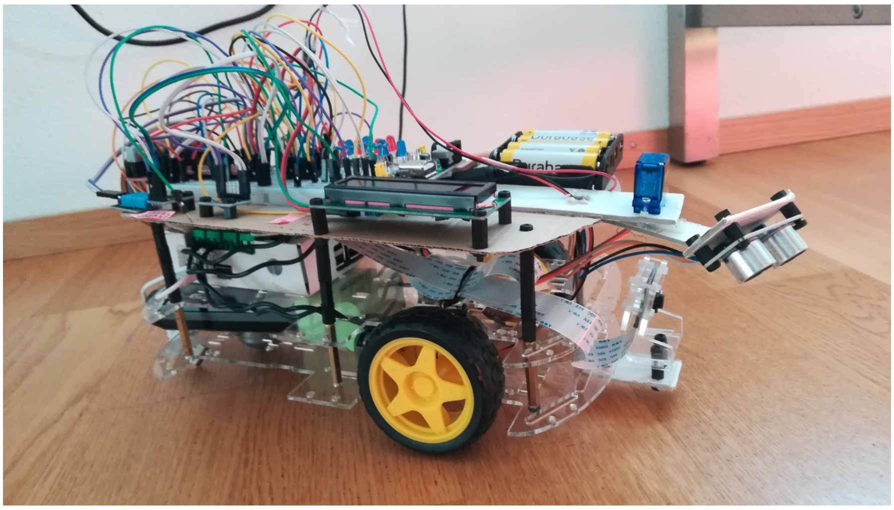
U.L.I.S.S.E. HPC
A self-driving HPC Robot based on Raspberry PI and Message Passing Interface

by Bulun Skunk Works (bulunskunkworks@gmail.com)
Zurich (CH), 2020



YouTube: https://www.youtube.com/channel/UCDNomXKPl-qkB3vEt4UxPUg
GitHub: https://github.com/BulunSkunkWorks/Ulisse-HPC

Abstract

Nowadays is the construction of a self-driving robotic car quite a straightforward endeavor. In fact, the availability of affordable and reliable hardware, as well as the plethora of tutorials on the Internet, make robotics an accessible and fascinating field for hobbyists, advanced makers or just beginners, and off course for all curious minds.
From my early experiments with Raspberry PI I appreciated the ingenious hardware made available in such a small space that jointly with the power of Linux as Operating System made this tiny development platform the magic box to bring my childhood dream, i.e. building my own robot, a reality.
In this experiment I used a Raspberry PI model 3B with 1 GB of RAM and 1 GHz quad-core Arm CPU, and as I progressed with the construction of the robot by adding sensors I faced all possible challenges that helped me to understand the complexity of robotics. One of the most complex challenges, core topic of this paper, was the decreasing robot's performance due to the real time environment analysis by means of the camera for object tracking and all other installed sensors.
To overcome this issue and paving the ground for further developments, I leveraged the knowledge I acquired during my University studies when, as fellow researcher of the Italian National Institute of Nuclear Physics, I studied Quantum Reactive Scattering of atmospheric chemical reactions. At these times I overcame the high time-consuming complex calculations by means of the High Performance Computing (HPC) principles implemented using the Message Passing Interface to run the code on Supercomputers, such as the Cray T3E I extensively used (academic references).

What follows will describe the principles implemented for the robot's navigation, the used components, as well as the HPC pseudo-code developed using the Message Passing Interface.

The Robot and its components

Having set a high target, i.e. the materialization of a childhood dream, from the outset I defined Critical To Quality requirements that my robot had to fulfill:

- Interact with the environment via Computer Vision.
- Rely on an Artificial Intelligence engine for real-time autonomous movement decisions.
- Anticipate the computational scalability required by future developments.

Being enthusiastic of both Sci-Fi and of Space Travels I dreamed really big, therefore I imagined my robot to be used in space missions, e.g. for the exploration of Mars in competition with Elon :), and I squeezed my brain to find a suitable and exciting name. And finally I came out with **U.L.I.S.S.E HPC, i.e. Universal Lander for InterStellar Space Exploration High Performance Computing**. Cool isn't it?
The first challenge was to equip the Robot with vision and the ability to analyze the environment. Among all possible solutions available on the Internet I found particular suited for my purpose OpenCV, a popular open source library for Computer Vision and Machine Learning for which clear Object Tracking tutorials from Pyimagesearch explain how to identify an object and track it. I implemented this functionality in Python to let the robot identify and track a yellow Tennis ball, and based on its position on the screen then decide the moving direction. In simple terms, the visual field captured by the Raspberry PI camera is divided in to 9 sectors:

1	2	3
4	5	6
7	8	9

Since the Robot' goal is to approach the Tennis ball and "see" it at the center of the visual field (i.e. within the quadrant number 5), the logic to steer the Robot is straightforward:

- If the Tennis ball is spotted within any quadrant of the first row on the top, it means it is far from the Robot, therefore the Gear Motors are activated forward to reach the ball and bring the ball in any quadrant of the second row.
- If the Tennis ball is spotted within any quadrant of the third row on the bottom, it means it is close to the Robot, therefore the Gear Motors are activated backward to distance the ball and bring the ball in any quadrant of the second row.
- If the Tennis ball is spotted within any quadrant of the column on the left, the Gear Motors are activated to steer the Robot on the left and bring the ball to the center column.
- If the Tennis ball is spotted within any quadrant of the column on the right, the Gear Motors are activated to steer the Robot on the right and bring the ball to the center column.

Once the ball is at the center of the visual field the Robot's goal is reached and just stops moving. During this journey the Robot may encounter obstacles that are avoided using a standard Distance sensor which, in real time, determines whether an obstacle is detected within a distance of 15 cm. To widen the detection range of the distance sensor and to limit the possibility of Catch-22 situations where the Robot cannot decide which direction to go, I mounted the Distance sensor on a Servo motor which continuously moves left to right. The logic is, again, very simple:

- If the servo is turned left and the obstacle is detected within 15 cm range, then the Robot turns right.
- If the servo is turned right and the obstacle is detected within 15 cm range, then the Robot turns left.
- If the servo is in front and the obstacle is detected within 15 cm range, then the Robot by default turns right. Why right instead of left? No reasons whatsoever, without additional information to decide which direction to go, left and right are equivalent.

Why the threshold to change direction is set to 15 cm? This value has been set after experimenting with balls of different sizes and by assessing the elapsed time the Robot needs to analyze the position of the ball and then decide the turning direction. Let's not underestimate the complexity of Object detection, either via OpenCV or any other AI packages for Computer Vision and Machine Learning. Object detection is a very computational-intensive and time-consuming task, therefore though in principle any objects, or people, or animals could be detected and analyzed in real time, the Raspberry PI will show up its computational limits. Nevertheless, a yellow Tennis ball resulted sufficiently simple for a relatively quick real time detection in a normal living environment, thus allowing for a smooth Robot steering.
A robotic lander to be used in a space exploration program should be equipped with other components to analyze the environment and send back to Earth useful information (don't forget, I dream really big :)), here below is the full list of components I used:

- Raspberry PI Model 3B
- Raspberry PI Camera 5MP
- Motor Shield from SB Components
- GPS Module NEO 6M
- Distance sensor HC-SR04
- Micro Servo SG90
- DHT-11 sensor for humidity and temperature detection.
- ADXL345 3-axis Accelerometer.
- QMC5883L 3-axis Compass.
- LCD 1602.
- Shift Register 74HC959 to enlighten 8 LEDs (somebody remember Knight Rider? :))
- Chassis equipped with 4 Gear Motors.
- Battery for Raspberry PI of 5V.
- Battery for Motors of 7.2 V.
- Breadboard Power.
- 10 Various resistors 220 Ω.
- About 6 meters of wires.

The installed sensors provide information that are displayed on the LCD as well as reported on the Raspberry PI Camera real-time Frame for a Sci-Fi-like Robot navigation as shown in picture 1.
The used Motor Shield from SB Components is a powerful piece of hardware, in fact other than 4 DC Motor ports it is equipped with LED Arrows indicating the motors direction, 2 IR ports as well as with one Distance Sensor port. Not bad for such a small board, however its full utilization requires a significant amount of GPIOs pins. As such in order to equip the robot with all the listed components I disabled the LED Arrows, both the 2 IR ports and the Distance Sensor ports, with the end result of freeing up 4 pins. The resulting GPIOs schema looks like in picture 2. Other GPIOs are freed up by using only 2 Motors instead 4 as both the Chassis and the Motor Shield would allow with two additional advantages, i.e. less power is needed, and change of direction will be much smoother encountering only a very little resistance on the rear.

The High Performance Computing framework

The above list of components equips the Raspberry PI with interesting multi-sensors capabilities and many others may be added, either using the GPIO pins or the I^2C port. However soon a relevant problem will surface, i.e. the overall Robot' performance will degrade as more sensor or devices are connected. In fact, in general robots and sensors are controlled with Software Code which performs an infinite loop to check every single sensor and then action the motors as needed. Tricks can be applied to reduce the overhead added by the sensors, e.g. there is no need to detect Temperature and Humidity in real time, similarly the GPS position does not vary significantly within a limited space of operations. Nevertheless these are just workarounds to overcome the bigger problem of real time environment analysis via Object Detection and the listed sensors, and the performance of the Robot will not improve significantly hindering any ambition of scaling up by adding new components or functionalities.

Capitalizing on my academic experience and relying on the quad-core Arm architecture of the Raspberry PI, I overcame the performance challenge by engineering a task-farm High Performance Computing Framework implemented with the Message Passing Interface (MPI) library. In simple terms, the developed parallel computing framework decouples the independent time-consuming processes (e.g. sensors detection, gear motors activation) which are then coordinated by a master process (aka, the "Lord of the Processes" :)), as exemplified in the schematics in picture 3. As the schema illustrates, the Robot steered by the HPC Software Code implemented with the `mpi4py` implementation of the MPI library runs 6 copies of the same software codes, each one identified by the unique Rank ID assigned by the MPI interface at startup. The beauty of this parallel architecture is that it could be run on a supercomputer or on a pool of networked servers with the aim of distributing the workload across all the available computing units. In this present example the Raspberry PI mounted in the Robot executes all processes at the cost of (i) linearly increasing the RAM consumption with the number of executed processes (under the assumption off course of not using dynamic memory allocation), and (ii) increasing the CPU load. The decision of how many processes to run in parallel and which tasks each one will perform (e.g. which sensors to control) is a matter of choice mostly driven by the available CPU/RAM and by the degree of parallelization needed.

An additional drawback introduced by the parallelization consists of the additional latency necessary for the processes to communicate to each other for data transfer (i.e. message passing) and to coordinate the overall software execution and robot steering. In our case the communication latency is quite low due to two factors: (i) the data transferred are in very limited amounts (few Bytes), and (ii) by the limited number of two-ways communication channels in the implemented HPC Framework. To further clarify, as showed in the schema above the 6 Processes perform the following tasks:

- **Rank ID 0** – This is the Master Process controlling all the others (how much I love to call it the "Lord of the Processes" :)):
  - It sends the Start/Stop signal to all the other Processes.
  - From Process with Rank ID 3 receives Temperature and Humidity.
  - From Process with Rank ID 5 receives the distance from an obstacle.
  - Dispatches Temperature, Humidity and Distance from obstacle to process with Rank ID 4 to display on the LCD.
- **Rank ID 1** – Worker Process in charge to:
  - determine the ball position on screen via Computer Vision and send to the process with Rank ID 2 the direction of movement.
  - Display on Frame the navigation information as in Picture 2.
- **Rank ID 2** – Worker Process in charge for the Motor steering.

- **Rank ID 3** – Worker Process in charge for the non-mission critical sensors:
  - Controls the GPS, the DHT for Humidity and Temperature measurements, the Accelerometer and Compass.
- **Rank ID 4** – Worker Process in charge to:
  - control the LCD to display: (i) a welcome message at the beginning of the mission, (ii) the measured temperature, humidity and distance from obstacle.
  - Activates the LEDs via the Shift Register. In the present setup this process serves uniquely to overload the Raspberry Pi and test the hardware and software setup in preparation of future developments.
- **Rank ID 5** – Worker Process in charge for steering the Distance Sensor and Servo Motor.

The MPI library provides with a powerful set of high level methods to cope with all possible Parallel Computing problems. For our case it's just enough to mention that the nature of the computational problem we want to solve, i.e. steer a robot in real time, just needs a limited set of methods to govern the point to point communication among the master process and the worker processes: one method is needed to send a message, one method is needed to receive the message. MPI offers different sets of these capabilities, however the decision of which one to chose must consider that the communication overhead may result in a slow-moving traffic jam where even the basic robot functionalities are severely hindered. In the current implementation of the HPC Framework I leveraging the blocking communication methods `MPI_send` and `MPI_recv` (instead of the non-blocking `MPI_Isend` and `MPI_Irecv`). In fact, the nature and granularity of the implemented parallel computational model does not lead to long idle time waiting for a message to be received, therefore I preferred to avoid the non-blocking model which, on the other hand, would require a more complex parallel model to deal with the asynchronous communication among processes.

Final considerations and further developments

The MPI library is easy to use to cope with software performance issues, under the assumption of decoupling independent computational tasks to be executed in parallel and able to communicate with each other to pass the required information. As such for a good load balancing and performance optimization it is essential to identify the most time consuming tasks to be parallelized, in the example of the U.L.I.S.S.E. HPC Robot would be the real time object detection and tracking. However the detection of a tennis ball is still a manageable task for the Raspberry PI I've used, and the implemented parallelization resulted in successfully smoothing the overall Robot navigation and give a fluid user experience.
Additionally, the proposed HPC Framework presents a remarkable advantage: it's scalable. This means that any future extension of the Robot capabilities, e.g. by adding new sensors and components, will leverage the same parallelization model by executing an additional Worker Process in charge of activating the new sensors. In doing this we should never forget the mentioned drawback, i.e. the limited RAM resource that may prove not sufficient if too many Worker Processes were to be executed.

The MPI pseudo code

```
#
# Import MPI library
#

from mpi4py import MPI
import sys

#Name of Individual MOTORS
m1 = PiMotor.Motor("MOTOR1",1)
m2 = PiMotor.Motor("MOTOR2",1)

motorAll = PiMotor.LinkedMotors(m1,m2)

def main():

#
# Defaults Values
#
  mpi_rank = 0
  mpi_size = 0

  mpi_comm = MPI.COMM_WORLD
  mpi_size = MPI.COMM_WORLD.Get_size()
  mpi_rank = MPI.COMM_WORLD.Get_rank()
  mpi_name = MPI.Get_processor_name()

  mpi_dest_rank = 0
  mpi_source_proc = 0
  mpi_data = [0,                        # 0/-1: continue/terminate
              0,                        # leftright
              0,                        # distance
              "Bearing (deg): N/A",            # Bearing
              0,                        # Temperature
              0,                        # Humidity
              "Latitude: N/A",                 # Latitude
              "Longitude: N/A",                # Longitude
              "Altitude: N/A",                 # Altitude
              "X-Axis G: N/A",                 # x-Axis Acceleration (in G)
              "Y-Axis G: N/A",                 # y-Axis Acceleration (in G)
              "Z-Axis G: N/A",                 # z-Axis Acceleration (in G)
              0,                                 # arrow_flash
              "FRONT",                         # direction
             0,                         # throttle
              "!@ N/A",                        # Distance txt
             0]                         # rotational throttle

  if( mpi_rank == 0):
#
# MPI MASTER PROCESS
#

#
# START THE WORKERS PROCESS UP
#
#    mpi_data[0] = 0

    for dest_rank in range (1, mpi_size ):
      mpi_comm.send( mpi_data, dest = dest_rank, tag=1 )

    mpi_status = MPI.Status()

#
# START NAVIGATION
```

```
#
    mission_start = time.time()

    while True:
# receive "OK" signal from workers

        mpi_data = mpi_comm.recv( source=MPI.ANY_SOURCE, tag = 1, status=mpi_status )
        mpi_source_proc = mpi_status.Get_source()

        if( mpi_source_proc == 1 ):
# Received Video Stream response

          if( mpi_data[0] == -1 ):
# Received Termination signal
            if( dbg_cfg == "full" ):
              outputfile.write( "\n" + mission_txt + " -  From Worked ID: " +
str(mpi_source_proc ) + " received Video Stream: " + str( leftright ) + "\n" )

            break
        elif( mpi_source_proc == 2 ):
# Received OK from Motors
          if( dbg_cfg == "full" ):
            outputfile.write( "\n" + mission_txt + " -  From Worked ID: " +
str(mpi_source_proc ) + " received OK from MOTORS " + "\n" )

        elif( mpi_source_proc == 4 ):
# Received OK from Shift Register

          if( dbg_cfg == "basic" ):
            outputfile.write( "\n" + mission_txt + " -  From Worked ID: " +
str(mpi_source_proc ) + " received OK from Shift register\n" )

        elif( mpi_source_proc == 5 ):
# Received Distance from Obstacle

          leftright= mpi_data[1]
          cm = mpi_data[2]

          if( dbg_cfg == "full" ):
            outputfile.write( "\n" + mission_txt + " -  From Worked ID: 5 received
Distance = " + str(cm) + " - leftright=" + str(leftright)+"\n")

          #
          #  CHECK IF OBSTACLES ARE IN RANGE
          #

          if( cm < 0 ):
            distance_txt = "!@ N/A!"

          elif( cm > 300 ):
            distance_txt = "!@ >300 cm"

          else:
            distance_txt = "!@: " + str( cm ) + " cm"

          mpi_data[ 15 ] = distance_txt

          if( cm > 0 and cm < 15 ):
            if ( leftright == 0 ):
              direction = "RIGHT"

            else:
              direction = "LEFT"

          else:
             direction = "FRONT"
```

```
          direction_txt = "Direction: " + direction

          mpi_data[ 12 ] = arrow_flash
          mpi_data[ 13 ] = direction
          mpi_data[ 14 ] = throttle
          mpi_data[ 16 ] = rotate_throttle

          # Display messages on LCD
          mpi_comm.send( mpi_data, dest = 4, tag = 1 )
          mpi_data = mpi_comm.recv( source= 4, tag = 1, status=mpi_status )

          # Move the Robot to avoid obstacle

          if( motor_cfg == '1' and mpi_size > 2 and direction != "FRONT"):
            mpi_comm.send( mpi_data, dest = 2, tag = 1 )

        elif( mpi_source_proc == 3 ):
# Received DHT, GPS, Accelerometer, Compass

        old_temp = mpi_data[4]
        old_hum = mpi_data[5]
        mpi_data[0] = 0

        if(mpi_source_proc != 2):
          mpi_data[4] = old_temp
          mpi_data[5] = old_hum

        mpi_comm.send( mpi_data, dest=mpi_source_proc, tag=1 )

# MASTER sends termination signal to all Working processes
#

    mpi_data[0] = -1

    for mpi_dest_rank in range (3, mpi_size ):
      outputfile.write( "MPI MASTER PROCESS, sending termination signal to Worker: " +
str( mpi_dest_rank ) +"\n")

      mpi_comm.send( mpi_data, dest=mpi_dest_rank, tag=1 )

    print("\nMASTER PROCESS - Game over, man. Game over!" )

  elif ( mpi_rank == 3 ):
#
# STEER THE DHT, GPS, Accelerometer, Compass
#

    while True:
      mpi_data = mpi_comm.recv( source=0, tag=1 )

      if( mpi_data[0] == -1 ):
# received exit signal, leave While loop
        outputfile.write( "\n" + mission_txt + " - DHT, GPS Worker ID: " +
str( mpi_rank ) + " - Received exit signal.\n" )
        print("\nWorker ID: ", mpi_rank, " - Game over, man. Game over!" )

        break

# DETERMINE BEARING
      if( compass_cfg == '1' ):
        if( startCompass_time == 0 ):
          startCompass_time = time.time()

        if( time.time() - startCompass_time > float( compass_time ) ):
          startCompass_time = 0

          reading = bearing( sensor, dbg_cfg )
```

```
        if( reading >= 0 and reading <=22.5 ):
          dir = "N"
          elif( reading > 22.5 and reading <= 67.5 ):
          dir = "NE"
        elif( reading > 67.5 and reading <= 112.5 ):
          dir = "E"
          elif( reading > 112.5 and reading <= 157.5 ):
          dir = "SE"
        elif( reading > 157.5 and reading <= 202.5 ):
          dir = "S"
        elif( reading > 202.5 and reading <= 247.5 ):
          dir = "SO"
        elif( reading > 247.5 and reading <= 292.5 ):
          dir = "O"
        elif( reading > 292.5 and reading <= 337.5 ):
          dir ="NO"
        elif( reading > 337.5 and reading <= 360 ):
          dir = "N"

          mpi_data[ 3 ] = "Bearing (deg): " + str(round( reading, 1 ) ) + " (" + dir +
")"
          old_mpidata = mpi_data[ 3 ]
        else:
          mpi_data[ 3 ] = old_mpidata

# DETERMINE ACCELERATION
      if( accl_cfg == '1' ):
        if( startAccl_time == 0 ):
          startAccl_time = time.time()

        if( time.time() - startAccl_time > float( accl_time ) ):
          startAccl_time = 0

          if( dbg_cfg == "full" ):
            mission_time_txt = str( datetime.timedelta( seconds = time.time() ) )
            mission_txt = "Mission Time: " + mission_time_txt
            outputfile.write("\n" + mission_txt + " - Worker ID 3 - Reading Acceleration
information" )

          tmp_data = xlr8( dbg_cfg )

          mpi_data[ 9 ] = "X-Axis accl. (G): " + str( round( float(tmp_data[ 0 ]), 4  ) )
          mpi_data[ 10 ] = "Y-Axis accl. (G): " + str( round( float(tmp_data[ 1 ]), 4 ) )
          mpi_data[ 11 ] = "Z-Axis accl. (G): " + str( round( float(tmp_data[ 2 ]), 4 ) )
          old_acclx = mpi_data[ 9 ]
          old_accly = mpi_data[ 10 ]
          old_acclz = mpi_data[ 11 ]
        else:
          if( dbg_cfg == "full" ):
            print("old_acclx = ", old_acclx )
            print("old_accly = ", old_accly )
            print("old_acclz = ", old_acclz )

          mpi_data[ 9 ] = old_acclx
          mpi_data[ 10 ] = old_accly
          mpi_data[ 11 ] = old_acclz

# DETERMINE GPS POSITION

      if( gps_cfg == '1' ):
          if( startGPS_time == 0 ):
           startGPS_time = time.time()

          if( time.time() - startGPS_time > float( gps_time ) ):
            startGPS_time = 0
```

```
          dataout = pynmea2.NMEAStreamReader()
          newdata=ser.readline().decode('UTF-8')

          if newdata[0:6] == "$GPGGA":
              newmsg=pynmea2.parse(newdata)
              lat=newmsg.latitude
              lat_min=newmsg.latitude_minutes
              lat_sec=newmsg.latitude_seconds
              lat_dir=newmsg.lat_dir

              lng=newmsg.longitude
              lng_min=newmsg.longitude_minutes
              lng_sec=newmsg.longitude_seconds
              lng_dir=newmsg.lon_dir

              num1 ='{:.5f}'.format(lat)
              num2 ='{:.5f}'.format(lat_min)
              num3 ='{:.5f}'.format(lat_sec)

              gps = "Latitude= " + str(num1) + " deg " + \
                        str( num2 ) + string.printable[68] + " " + \
                        str( num3) + string.printable[63] + " " + lat_dir

              old_lat = gps
              mpi_data[6] = gps

              num1 ='{:.5f}'.format(lng)
              num2 ='{:.5f}'.format(lng_min)
              num3 ='{:.5f}'.format(lng_sec)

              gps = "Longitude= " + str(num1) + " deg " + \
                        str( num2 ) + string.printable[68] + " " + \
                        str( num3) + string.printable[63] + " " + lng_dir

              mpi_data[7] = gps
              old_lng = gps

              alt = newmsg.altitude
              alt_units = newmsg.altitude_units
              gps = "Altitude = " + str( alt ) + " " + alt_units
              mpi_data[8] = gps
              old_alt = gps

          else:
              mission_time_txt = str( datetime.timedelta( seconds = time.time() ) )
              mission_txt = "Mission Time: " + mission_time_txt
              outputfile.write( "\nNo GPS Readings\n" )
              mpi_data[6] = old_lat
              mpi_data[7] = old_lng
              mpi_data[8] = old_alt
        else:
          mpi_data[ 6 ] = old_lat
          mpi_data[ 7 ] = old_lng
          mpi_data[ 8 ] = old_alt


# MEASURE TEMPERATURE AND HUMIDITY - BUT ONLY every dht_time seconds

      if( dht_cfg == '1' ):
        if( startDHT_measure == 0 ):
          startDHT_measure= time.time()

        if ( time.time() - startDHT_measure > float( dht_time ) ):
          startDHT_measure = 0

          humidity, temperature = Adafruit_DHT.read_retry(sensorDHT, DHT11_PIN)
```

```
          if humidity is not None and temperature is not None:
            mpi_data[4] = temperature
            mpi_data[5] = humidity

          else:
            print
            print('WARNING!!!')
            print('-->Failed to get reading from the DHT sensor. Try again!')

            mpi_data[4] = 0
            mpi_data[5] = 0

# sending to Worked ID 1 navigation info for display on Frame
      mpi_comm.send( mpi_data, dest = 1, tag = 1 )

# sending back to Master "OK" signal
      mpi_comm.send( mpi_data, dest = 0, tag = 1 )

  elif( mpi_rank == 4 ):
# Display on LCD and steer the Shift Register

# Display the Welcome Message on Startup

    welcome( display )

#
# Steer the Shift Register
#

    while True:
      mpi_data = mpi_comm.recv( source=0, tag=1 )

      if( mpi_data[0] == -1 ):
# received exit signal, leave While loop
        if( lcd_cfg == '1' ):
          display.lcd_clear()
          lcd( display, "Game over, man.", 1 )
          lcd( display, "GAME OVER!", 2  )

        break

      Temperature_txt = "T: " + str( mpi_data[ 4 ] )
      Humidity_txt = "H: " + str( mpi_data [5 ] )

      if( lcd_cfg == '1' ):
        display.lcd_clear()
        lcd( display, Temperature_txt +" "+ Humidity_txt, 1 )
        lcd( display, mpi_data[ 15 ], 2 )

# Steer the Shift Register and switch the LEDs on
      if( shift_cfg == '1' ):
        shift_reg( dbg_cfg, SER_PIN, SCK_PIN, RCK_PIN )

# sending back to Master "OK" signal

      mpi_data[0] = 0
      mpi_comm.send( mpi_data, dest = 0, tag = 1 )

  elif( mpi_rank == 2 ):
#
# ACTIVATE MOTORS
#

    while True:
      mpi_status = MPI.Status()
      mpi_data = mpi_comm.recv( source=MPI.ANY_SOURCE, tag=1, status = mpi_status )
      mpi_source_proc = mpi_status.Get_source()
```

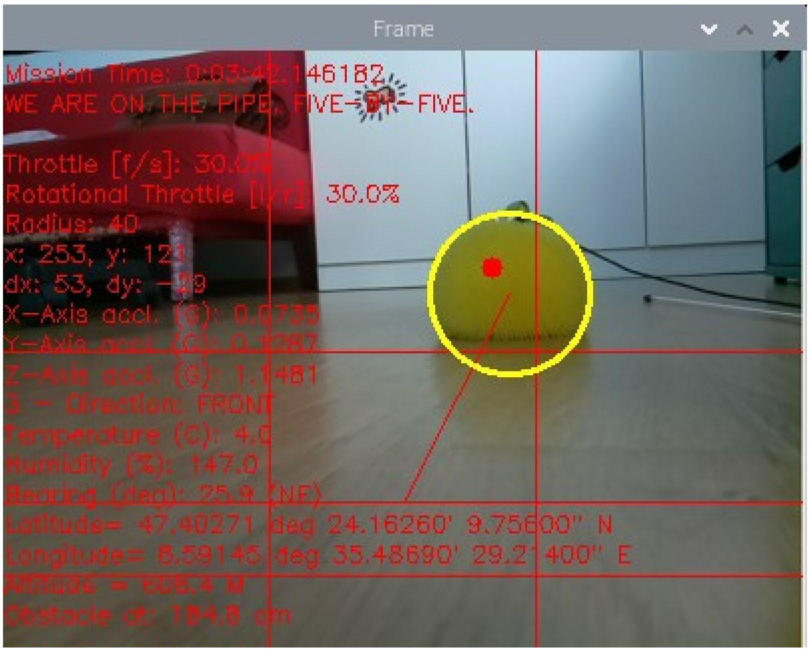
```
      if( mpi_data[0] == -1 ):
# received exit signal, leave While loop
        outputfile.write( "Worker ID: " + str( mpi_rank ) + " - Received exit signal.\
n" )
        print("\nWorker ID: ", mpi_rank, " - Game over, man. Game over!" )

        break

# Invoke the Move function passing:
# - forward throttle,
# - rotational throttle
# - direction
# - debug parameters

      move( mpi_data[ 12 ], mpi_data[ 13 ], mpi_data[ 14 ], mpi_data[ 16 ], fwd_time,
rotate_time, dbg_cfg )

# sending back to Master "OK" signal

      mpi_data[0] = 0
      mpi_comm.send( mpi_data, dest = 1, tag = 1 )

  elif( mpi_rank == 5 ):
#
# STEER THE DISTANCE SENSOR and SERVO MOTOR
#

#
# Initialize the Servo Motor
#
    mpi_recv_status = 0
    angle = 0
    leftright = 0 # 0 = left, 1 = right

    while True:
      mpi_data = mpi_comm.recv( source=0, tag=1 )

      if( mpi_data[0] == -1 ):
# received exit signal, leave While loop

        if( servo_cfg == '1' ):
          SetAngle( pwm_servo, SERVO_PIN, 50 )
          pwm_servo.stop()

        break

      if( servo_cfg == '1' ):
#
# ROTATE DISTANCE SENSOR
#

        if ( angle == 0 ):
          leftright = 0
        elif (angle == 100 ):
          leftright = 1

        SetAngle( pwm_servo, SERVO_PIN, angle )

        if ( leftright == 0 ):
          angle = angle + 50
        elif ( leftright == 1 ):
          angle = angle - 50

        mpi_data[1] = leftright
```

```
# Check if obstacle are in range

      cm = -1

      while( cm < 0 ):
        cm = distance( TRIG_PIN, ECHO_PIN )

        if( cm < 0 ):
          distance_txt = "Obstacle at: NOT DETECTED!"

        elif( cm > 300 ):
          distance_txt = "Obstacle at: >300 cm"

        else:
          distance_txt = "Obstacle at: " + str( cm ) + " cm"

      mission_time_txt = str( datetime.timedelta( seconds = time.time() ) )
      mission_txt = "Mission Time: " + mission_time_txt

      outputfile.write("Worker ID: " + str( mpi_rank ) + " " + mission_txt + " " +
distance_txt + "\n")

# sending back to Master "OK" signal

      mpi_data[2] = cm
      mpi_comm.send( mpi_data, dest = 1, tag = 1 )
      mpi_comm.send( mpi_data, dest = 0, tag = 1 )

  elif ( mpi_rank == 1 ):

#
# VIDEO STREAM
#
# Define the lower and upper boundaries of the "green" ball
# in the HSV color space
#

    greenLower = (29, 86, 6)
    greenUpper = (64, 255, 255)

#
# Grab the reference to the webcam and start the Video stream,
# warm the camera up and open the output video stream
#
    vs = VideoStream(src=0).start()

    while True:

      mpi_status = MPI.Status()
      mpi_data = mpi_comm.recv( source=MPI.ANY_SOURCE, tag=1, status = mpi_status )
      mpi_source_proc = mpi_status.Get_source()

      if( mpi_data[0] == -1 ):
# received exit signal, leave While loop

        break

#
# OPEN FRAME OUTPUT
#

      frame = frame_capture( args, vs )
      frame = vs.read()

#
# Resize the frame, blur it, and convert it to the HSV color space
#
```

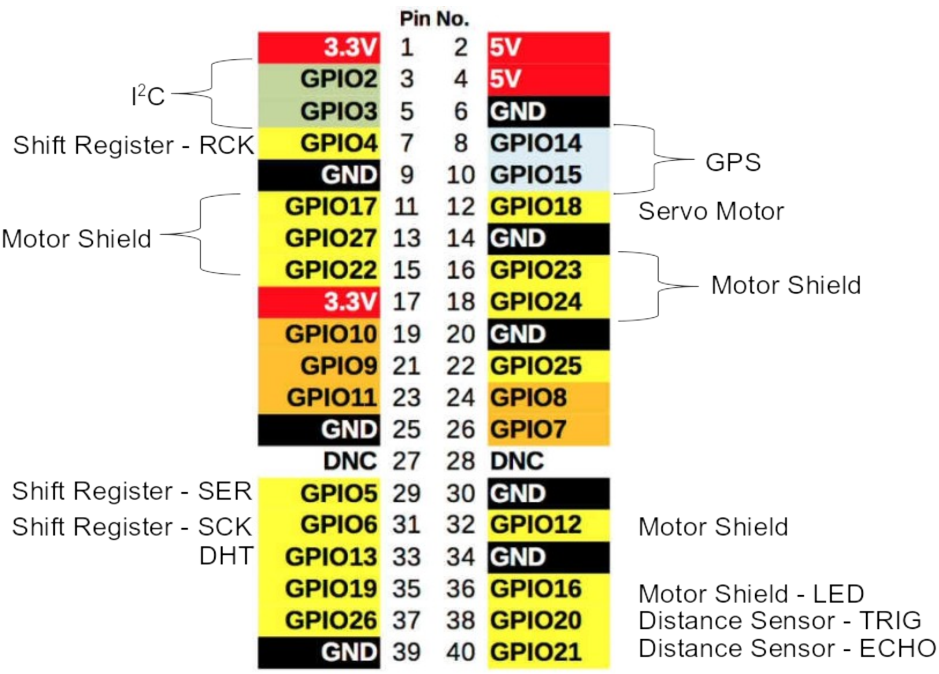
```
      frame = imutils.resize(frame, width=frame_size)
      blurred = cv2.GaussianBlur(frame, (11, 11), 0)
      hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)

#
# Construct a mask for the color "green", then perform a series
# of dilations and erosions to remove any small blobs left in the mask
#

      mask = cv2.inRange(hsv, greenLower, greenUpper)
      mask = cv2.erode(mask, None, iterations=NrIterations )
      mask = cv2.dilate(mask, None, iterations=NrIterations )

#
# find contours in the mask and initialize the
# current (x,y) center of the ball
#

      cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
                cv2.CHAIN_APPROX_SIMPLE)
      cnts = imutils.grab_contours(cnts)

#
# Only proceed if at least one contour was found
#

        if len(cnts) > 0:

#
# Find the largest contour in the mask,
# then use it to compute the minimumn enclosing circle and centroid
#

          c = max(cnts, key=cv2.contourArea)
          ((x, y), radius) = cv2.minEnclosingCircle(c)
          M = cv2.moments(c)
          center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

#
# Only proceed if the radius meets a minimum size
# and draw the vector to the center of the circle
#

          cv2.line( frame, ( int(x), int(y) ), (frame.shape[1]/2, frame.shape[0]*3/4 ),
(0, 0, 255), 1 )

          if( radius > 5 ):

#
#  CHECK IN WHICH X-QUADRANT THE BALL IS
#

            if( ( int( x ) - int( frame.shape[ 1 ] / 3 ) < 0 ) ):
              direction = "LEFT"

            elif( ( int( x ) - int( frame.shape[ 1 ]*2/3 ) > 0 ) ):
              direction = "RIGHT"

            else:
#
#  CHECK IN WHICH Y-QUADRANT THE BALL IS

              if( ( int( y ) - int( frame.shape[ 0 ]*6/8 ) < 0 ) ):
                direction = "FRONT"

              elif( ( int( y ) - int( frame.shape[ 0 ]*7/8 ) > 0 ) ):
```

```
                direction = "BACK"

              else:
                direction = "HALT"

#
# MOVE THE BALL TO THE CENTER QUADRANT
#
          if( motor_cfg == '1' and mpi_size > 2 ):
            mpi_data[ 12 ] = arrow_flash
            mpi_data[ 13 ] = direction
            mpi_data[ 14 ] = throttle
            mpi_data[ 16 ] = rotate_throttle

            mpi_comm.send( mpi_data, dest = 2, tag = 1 )
            mpi_status = MPI.Status()
            mpi_data = mpi_comm.recv( source= 2, tag = 1, status=mpi_status )

#
# end if len (cnts > 0 )
#

        else:
          x = 0
          y = 0
          dX = 0
          dY = 0

          direction = "FRONT"

           if( motor_cfg == '1' and mpi_size > 2 ):
              mpi_data[ 12 ] = arrow_flash
              mpi_data[ 13 ] = direction
              mpi_data[ 14 ] = throttle
              mpi_data[ 16 ] = rotate_throttle

              mpi_status = MPI.Status()
              mpi_comm.send( mpi_data, dest = 2, tag = 1 )
              mpi_data = mpi_comm.recv( source= 2, tag = 1, status=mpi_status )

# Write on Frame received information
#
      write( frame, 0, 15, mission_txt )
      write( frame, 0, 30, target_lock_txt )
      write( frame, 0, 135, xaccl_txt)
      write( frame, 0, 150, yaccl_txt)
      write( frame, 0, 165, zaccl_txt)
      write( frame, 0, 180, direction_txt )
      write( frame, 0, 195, Temperature_txt )
      write( frame, 0, 210, Humidity_txt )
      write( frame, 0, 225, Bearing_txt )

# sending back to Master "OK" signal

      mpi_data[0] = 0
      mpi_comm.send( mpi_data, dest = 0, tag = 1 )
```

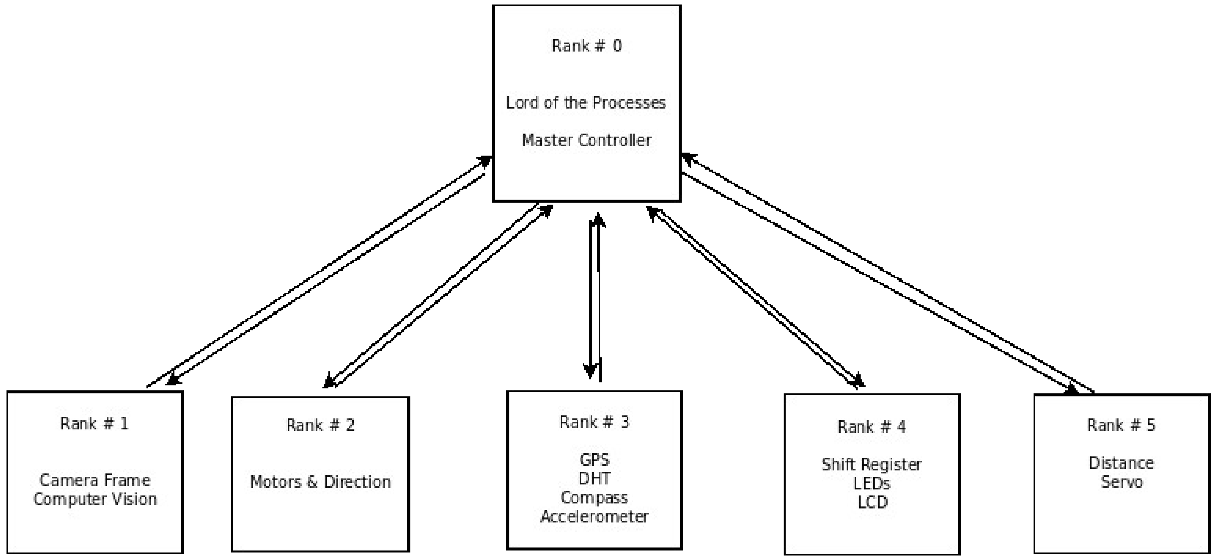
Picture 1



Picture 2



Picture 3



Other Pictures